

University of Tennessee, Knoxville
Trace: Tennessee Research and Creative Exchange

Doctoral Dissertations | Graduate School

5-2015

Neuroscience-Inspired Dynamic Architectures

Catherine Dorothy Schuman
University of Tennessee - Knoxville, cschuman@vols.utk.edu

This Dissertation is brought to you for free and open access by the Graduate School at Trace: Tennessee Research and Creative Exchange. It has been accepted for inclusion in Doctoral Dissertations by an authorized administrator of Trace: Tennessee Research and Creative Exchange. For more information, please contact trace@utk.edu.

To the Graduate Council:

I am submitting herewith a dissertation written by Catherine Dorothy Schuman entitled "Neuroscience-Inspired Dynamic Architectures." I have examined the final electronic copy of this dissertation for form and content and recommend that it be accepted in partial fulfillment of the requirements for the degree of Doctor of Philosophy, with a major in Computer Science.

J. Douglas Birdwell, Major Professor

We have read this dissertation and recommend its acceptance:

Mark E. Dean, Tsewei Wang, Itamar Arel, Bruce MacLennan

Accepted for the Council:
Carolyn R. Hodges

Vice Provost and Dean of the Graduate School

(Original signatures are on file with official student records.)

Neuroscience-Inspired Dynamic Architectures

A Dissertation Presented for the

Doctor of Philosophy

Degree

The University of Tennessee, Knoxville

Catherine Dorothy Schuman

May 2015

© by Catherine Dorothy Schuman, 2015
All Rights Reserved.

For my nieces and nephews: Hannah, Austin, Addison, Aiden, and Anna-Beth.
May this inspire you to do great things.

In memory of my grandparents:
Fred Wright (1905 – 1973) and Dorothy Babcock Wright (1909 – 1971)
Richard Schuman, Jr. (1920 – 2007) and Catherine Dunn Schuman (1917 – 2014)

Acknowledgements

I would like to thank the National Science Foundation Graduate Research Fellowship Program and the Department of Electrical Engineering and Computer Science Bodenheimer Fellowship (especially Dr. Robert Bodenheimer and Mr. Mike Crabtree) for providing me funding to conduct research of my choosing. I would also like to thank the Department of Electrical Engineering and Computer Science for allowing me to serve as a teaching assistant throughout my time as a graduate student.

I would like to thank my advisor, Dr. Doug Birdwell, for his constant support and feedback, and for being willing to go down this unknown path with me. I would like to thank Dr. Mark Dean for believing in my work and for developing the accompanying hardware implementation. I would also like to thank the other members of my committee, Dr. Tsewei Wang, Dr. Itamar Arel, and Dr. Bruce MacLennan for not only their insight and support, but also for the amazing courses they offered that helped to inspire aspects of this work. I would like to thank Dr. Jim Plank for teaching me how to do computer science research and for allowing me to be his teaching assistant for five years. I would like to thank Dr. Lynne Parker for her mentorship and Dr. Mike Langston for his support.

I would like to thank the senior design team who built the first hardware implementation: Chris Daffron, Josh Willis, Jacob Tobin, Belinda Loi, David Bashai, Joonki Rhee, and Tommy Nguyen. I would also like to thank Chris, Josh, and Jason Chan for their ongoing work with the hardware implementation. I would like

to thank Meg Drouhard for her amazing visualizations of NIDA networks, some of which are included in this work. I would also like to thank our undergraduate research assistants, Luke Bechtel and Ryan Wagner, for their dedication in working with NIDA and DANNA. I would like to thank UT for their Newton Cluster, on which I ran many of the simulations described in this work, and Gerald Raggianthi for his prompt replies to emails and his dedicated efforts to keep Newton up and running as often as possible. I would like to thank Randy Bond, Bryan Burke, Markus Iturriaga Woelfel, and all of the EECS IT staff for keeping all of our computing resources up and running. I would like to thank Dana Bryson, Julia Elkins, Kim Cox, Melanie Kelley, Linda Robinson, Sonja Hill, and all of the EECS staff who do amazing work and don't get nearly the recognition that they deserve.

I would like to thank Lily Huxford and Maggie Bouldin for being some of the best friends anyone could ask for, and for understanding my crazy schedule. I would like to thank Denise Koessler Gosnell, Zahra Mahoor, Lonnie Yu, and Nick Lineback for being the best graduate study group ever, and Ashleigh Ellison, Jordan Deyton, Chad Braden, Chris Craig, Wade Jenkins, Blake Hitchcock, and Rob Fletcher for being the best undergraduate study group ever. I would like to thank Denise, Zahra, Meg, Nicole Pennington, Sadika Amreen, and Casey Miller (the Systers) for their passionate support of women in electrical engineering and computer science.

I would like to thank my parents, John and Sherry Schuman, for their unwavering support and love. I would like to thank my sister, Tracey Lawson, for listening to every stressed phone call, and my brother, Alan Oliver, for getting me interested in computer science in the first place. I would like to thank my aunts and uncles, my many cousins, and my other family members, for their support, even when they didn't understand what I was talking about it. I would like to thank my nieces and nephews, Hannah Lawson, Austin Oliver, Addison Oliver, Aiden Oliver, and Anna-Beth Oliver, for being a welcome distraction from research and for being just generally awesome people. Without their support, this would not have been possible.

This material is based upon work supported by the National Science Foundation Graduate Research Fellowship Program under Grant No. DGE-0929298. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author and do not necessarily reflect the views of the National Science Foundation.

Abstract

Biological brains are some of the most powerful computational devices on Earth. Computer scientists have long drawn inspiration from neuroscience to produce computational tools. This work introduces neuroscience-inspired dynamic architectures (NIDA), spiking neural networks embedded in a geometric space that exhibit dynamic behavior. A neuromorphic hardware implementation based on NIDA networks, Dynamic Adaptive Neural Network Array (DANNA), is discussed. Neuromorphic implementations are one alternative/complement to traditional von Neumann computation. A method for designing/training NIDA networks, based on evolutionary optimization, is introduced. We demonstrate the utility of NIDA networks on classification tasks, a control task, and an anomaly detection task. There are known neural structures (such as cortical columns) that are repeated many times in the brain, and there are other structures that are useful for a variety of different tasks. We speculated that "useful structures" will also emerge in NIDA networks. Three methods for identifying useful substructures within a NIDA network are presented: common structure, activity-based, and causality paths. We explored reusing activity-based useful substructures over the course of evolutionary optimization, but the results for those tests were inconclusive. One component of biological brains that is often ignored in biologically-inspired computation is the influence of affective, or emotion-related, systems. We define artificial affective systems and explore the effect of affective systems in NIDA networks in terms of

behavior of the network and on the evolutionary optimization design method. We conclude with an outline of future research opportunities identified during this effort.

Table of Contents

1 Introduction **1**

2 Related Work **11**

2.1 Neuroevolution 12

2.2 Embedding in a Geometric Space 26

2.3 Dynamic Networks 30

2.3.1 Continuous Time Neural Networks 30

2.3.2 Spiking Neural Networks 32

2.4 Recognition and Reuse of Useful Structures 38

2.5 Affective Components 44

2.6 Neuromorphic Hardware 54

2.7 Summary 63

3 Architecture and Design Method **66**

3.1 Neuroscience-Inspired Dynamic Architecture 66

3.2 Design Method 72

3.2.1 Representation 73

3.2.2 Performance 74

3.2.3 Creation of Offspring 76

3.2.4 Advantages and Disadvantages of Evolutionary Optimization . 78

3.2.5 Sequential Implementation 82

3.2.6 Multi-Process Implementations 83

3.3 Results . . . 102
3.3.1 Anomaly Detection: Network Traffic . . . 104
3.3.2 Control: Pole Balancing . . . 113
3.3.3 Classification: Handwritten Digit . . . 126
3.4 Fixed Structure Weight Training . . . 138
3.5 Effect of Mutation and Crossover Rates on Training . . . 141
3.6 Comparison to Standard Benchmarks . . . 144
3.6.1 Breast Cancer . . . 148
3.6.2 Diabetes . . . 152
3.6.3 Iris . . . 155
3.6.4 Summary . . . 159
3.7 Summary . . . 159

4 Recognition and Reuse of Useful Structures 161
4.1 Recognizing Useful Substructures . . . 162
4.1.1 Common Structures . . . 162
4.1.2 Activity-Based . . . 175
4.1.3 Causality Paths . . . 181
4.2 Reuse of Useful Structures . . . 189
4.2.1 Results . . . 193
4.3 Summary . . . 198

5 Affective Systems and Multiple Interacting Networks 201
5.1 Affective System Description . . . 202
5.1.1 Simple Affective System . . . 204
5.1.2 Results for Simple Affective System . . . 206
5.1.3 Complex Affective System . . . 221
5.2 Adaptation to Design Method . . . 223
5.2.1 Simple Affective System . . . 224
5.2.2 Complex Affective System . . . 224

5.3 Results . . . 229
5.3.1 Effect of Simple Affective System . . . 229
5.3.2 Effect of Complex Affective System . . . 236
5.4 Multiple Interacting Networks . . . 247
5.5 Summary . . . 249

6 Hardware Implementation **250**
6.1 Dynamic Adaptive Neural Network Array . . . 251
6.2 DANNA Design Method . . . 254
6.3 Summary . . . 259

7 Future Directions **262**
7.1 Architecture and Method . . . 262
7.2 Useful Substructures . . . 272
7.3 Affective Systems and Multiple Interacting Networks . . . 274
7.4 Hardware Implementations . . . 278
7.5 Real-Time Learning . . . 283
7.6 Summary . . . 287

8 Conclusion **289**

Bibliography **293**

Appendix **329**

A A Brief Introduction to the Brain **330**
A.1 Neurons and Synapses . . . 330
A.2 Parts of the Brain . . . 334
A.3 Development of the Brain . . . 336
A.4 Memory, Learning, and Plasticity in Biological Brains . . . 338

B Artificial Neural Networks **341**

B.1 Neural Network Architectures 341

B.2 Neuron Models 343

B.3 Training a Neural Network 346

B.4 A Brief History of Artificial Neural Networks 349

C Evolutionary Algorithms **352**

C.1 A Brief History of Evolutionary Algorithms and Related Methods . . 358

D Reinforcement Learning **360**

E Theoretical Results **366**

Vita **371**

List of Tables

2.1 This table summarizes the works in Section 2.1. In particular, it notes whether the work allows for structural changes as part of the evolution, whether crossover is used during the evolutionary optimization, and whether a local search method is used to discover weights. Cells with an X indicate that that method has been used in the corresponding work. 27

3.1 Approximate probabilities for each mutation type for the anomaly detection problem. 105

3.2 Estimated probability of detection (P_d), probability of false alarm (P_{fa}), and probability of missed detection (P_m) of increases and decreases for N_+ and N_-, respectively, for three different test types (large, medium, and small changes in mean arrival rate). 106

3.3 Parameter values used in single pole balancing application. 115

3.4 Test cases used in f_2 for the single pole balancing application. 117

3.5 Approximate probabilities for each mutation type for the pole balancing problem. 119

3.6 Number of time steps two networks are able to continue balancing the pole. The cut-off point for these networks was 100000 time steps. . . 122

3.7 Input Encoding Ranges. Each of the four state space variables were split into three ranges. There is a corresponding input neuron for each of these ranges. When the state space is updated (or set initially), the range that each value belongs to is calculated, resulting in four ranges. Then, a pulse is applied to each of the four neurons. 122
3.8 Approximate probabilities for each mutation type for the pole balancing problem. 123
3.9 Number of time steps four networks are able to continue balancing the pole. The cut-off point for these networks was 100000 time steps. . . 125
3.10 Approximate probabilities for each mutation type for the handwritten digit problem. 129
3.11 Accuracy breakdown for networks scanning by column trained with firing and not firing methods 133
3.12 Classification error rate on the testing set for the breast cancer classification problem. The number of training and testing instances used are listed; when unknown they are left blank. The number of testing instances listed with the (*) indicate that the instances with unknown data were not included as part of the testing or training set. The results are listed in increasing order of testing error rate. NIDA results are bolded. 151
3.13 Classification error rate on the testing set for the diabetes classification problem. The number of training and testing instances used are listed; when unknown they are left blank. The results are listed in increasing order of testing error rate. NIDA results are bolded. 154
3.14 Classification error rate on the testing set for the iris classification problem. The number of training and testing instances used are listed; when unknown they are left blank. The results are listed in increasing order of testing error rate. NIDA results are bolded. 157

4.1 Accuracy breakdown for a network trained to recognize images of the digit 7. 184

5.1 Network and affective system parameters. 231
5.2 Evolutionary algorithm parameters. 231
5.3 Network parameters. 241
5.4 Mutation rates. 241
5.5 Optimization parameters. 241

A.1 Neurotransmitters and their effects [188, 238]. 333

List of Figures

3.1 An example NIDA network with input, output, and hidden neurons labeled. Synapse labels and directionality are given in Figure 3.2. . . 69

3.2 An example NIDA network with inhibitory and excitatory synapses labeled and directionality of the synapses noted. 70

3.3 An example NIDA network event simulation. In this figure, there is a change-in-charge event traveling along a synapse towards a neuron. When that event reaches the neuron, it causes the neuron's charge level to increase over the neuron's threshold, which causes that neuron to fire. We show the change-in-charge event reaching the neuron and two newly created change-in-charge events in two separate pictures, but the two middle pictures occur at the same time in the simulation. . . 71

3.4 Two neuron oscillator, connected with excitatory synapses. By setting the weights and thresholds correctly, one may design this network so that the firing of one neuron causes an interleaved firing of both neurons at regular intervals. 73

3.5 Example plane crossover parent and child networks. In this example, the part of parent 1 to the left of the plane goes to child 1, and the part to the right of the plane goes to child 2. Similarly, the part to the left of the plane of parent 2 goes to child 2, and the part to the right of the plane goes to child 1. 79

3.6 Flow chart depicting the crossover operation. 80

3.7 Flow chart depicting the evolutionary optimization used. 84
3.8 Overview of the client and server processes and their associated threads. One client process is shown in expanded view so that the threads within it can be seen. Communication between threads is shown with arrows. 85
3.9 Flow chart for the initialization/client accept thread of the server process. 87
3.10 Flow chart for the client-communication thread of the server process. The fitness subroutine is shown in Figure 3.13, and the parent subroutine is shown in Figure 3.16. 88
3.11 Flow chart for the initialization thread of the client process. 90
3.12 Flow chart for the simulation thread of the client process. 93
3.13 Flow chart for the fitness subroutine of the client thread of the server process. 95
3.14 Flow chart for the GA thread of the server process. 96
3.15 Flow chart for the communication thread of the client process. 97
3.16 Flow chart for the parent subroutine of the client thread of the server process. 98
3.17 Example of a run of GAv2 in which there are 12 client processes. The server process has a threshold of mature networks set to 10. It is noted when each client process sends a new fitness value to the server process, when that process is selected for deletion, when it is selected to be a parent, and when it receives a new child network. The current number of mature networks in the server process is shown with a blue line, and the threshold of mature networks required to initiate the reproduction process is shown as well. 100
3.18 Flow chart for the client thread of the server process, which enters one of three modes, the operations of which are encapsulated in colored boxes: sending (orange), receiving (yellow), or completed (blue). . . . 103

3.19 Simulation results with each detection network. The mean arrival rate of the test run is shown as the blue line. The output firing events for N_+ and N_- are shown in red (+) and green (–). The events are discrete, and only their times are meaningful (see zoomed–in region showing both inputs and outputs); their vertical placement is for convenience. Example false alarm events (FA) are indicated. 107

3.20 ROC curves for Neyman–Pearson optimal detectors (left) and a NIDA network that detects increases in mean arrival rate. Note that the optimal detectors assume equal *a priori* probability of the hypotheses and arriving events are from one of two processes having the indicated mean arrival rates, while the NIDA network detector was trained to detect *any* increase in arrival rate within a specified range and made no assumptions about *a priori* statistics. 110

3.21 Probability of detection (P_d), probability of false alarm (P_{fa}), and the probability of a missed detection (P_m) results for nine different networks that performed reasonably well on detecting positive changes for the three test cases: large, medium, and small positive changes in mean arrival rate. The networks are grouped based on the fitness function they were trained under. 112

3.22 A diagram of the environment with the state variables labeled (not drawn to scale). 114

3.23 Fitness value (roughly the number of time steps the network can keep the pole balanced) vs. epoch for the best network in the population. Results are shown for five networks for each fitness function. 119

3.24 Results for two networks trained using f_1 when tested on 110 different start conditions. 120

3.25 Results for two networks trained using f_2 when tested on 110 different start conditions. 121

3.26 Fitness value (roughly the number of time steps the network can keep the pole balanced) vs. epoch for the best network in the population. Results are shown for five networks for each fitness function. 124
3.27 Results for the best performing networks for f_1 and f_2 when tested on 110 different start conditions. 124
3.28 Example digits from the MNIST handwritten database [142]. Rather than give an entire image as input to a NIDA network, input is given in one of three ways: (1) a row at a time, (2) a column at a time, or (3) both a row and a column at a time. 127
3.29 A voting scheme amongst networks of an ensemble is used to determine the digit for a particular image. Each network in the ensemble (represented by a square on the grid) receives an image on input and simulates activity within the network. Based on that activity, the network may or may not cast a vote (casting a vote is represented by shading in the grid). The digit with the most votes is the guessed digit for the ensemble. 130
3.30 Classification results for the first fitness function trained with the firing method. Elements along the diagonal indicate correctly classified digits. All other values are incorrectly classified. Note that the coloring is on a log scale, which was used so that incorrect classifications can be more easily observed. 131
3.31 Classification results for the first fitness function trained with the not firing method. Elements along the diagonal indicate correctly classified digits. All other values are incorrectly classified. Note that the coloring is on a log scale, which was used so that incorrect classifications can be more easily observed. 132

3.32 Classification results for the second fitness function. Elements along the diagonal indicate correctly classified digits. All other values are incorrectly classified. Note that the coloring is on a log scale, which was used so that incorrect classifications can be more easily observed. 134
3.33 Classification results for combined ensemble of networks trained with the first fitness function with the firing method, the first fitness function with the not firing method, and the second fitness function (entropy-based). Elements along the diagonal indicate correctly classified digits. All other values are incorrectly classified. Note that the coloring is on a log scale, which was used so that incorrect classifications can be more easily observed. 135
3.34 Network size information for the networks produced. The first six rows of the plots correspond to networks produced using the first fitness function (alternating firing and not firing method), while the last row of networks corresponds to networks produced using the second fitness function (based on entropy). The box and whisker plots show the lower to upper quartile values, with a red line at the median. The whiskers indicate the range of the data, with outliers also shown. Numbers of neurons (first column) and synapses (second column) are given for each digit type and each scanning type. 136
3.35 Flow chart showing the fixed-structure weight training method used. . 139
3.36 Heat maps showing the average, maximum, and minimum fitness values for varying values of N and varying s multiplier values, where $S = s*N$. The values where $N <= s$ are displayed as white squares in the heat map. Higher values indicate better performing networks. 140

3.37 Results from the pole balancing problem (discrete encoding and fitness function f_2) for varying crossover and mutation rates. Each block in the map represents the average fitness value over 10 runs of the best network after training for two hours for that crossover and mutation rate combination. 142
3.38 Results from the digit recognizers for the handwritten digit problem (first fitness function, scanning by column) for varying crossover and mutation rates. Each block in the map represents the average fitness value over 10 runs of the best network after training for two hours for that crossover and mutation rate combination. 143
3.39 Bar chart depicting results for the breast cancer classification task. Results are grouped by the number of training/testing instances, where the number of training instances is listed first and the number of testing instances is listed second. Each method is labeled. 150
3.40 Bar chart depicting results for the diabetes classification task. Results are grouped by the number of training/testing instance, where the number of training instances is listed first and the number of testing instances is listed second. It is also noted if the number of training/testing instances is unknown. Each method is labeled. 153
3.41 Bar chart depicting results for the iris classification task. Results are grouped by the number of training/testing instances, where the number of training instances is listed first and the number of testing instances is listed second. It is also noted if the number of training/testing instances is unknown. Each method is labeled. 158

4.1 Common structures (cycles or paths) appearing in at least eight of nine networks that performed well in detecting increases in mean packet arrival rate. Red synapses are inhibitory and blue synapses are excitatory. A histogram showing the distances of the path/cycle is also given for each common path/cycle. More common paths are in Figure 4.2. 164
4.2 Common structures (cycles or paths) appearing in at least eight of nine networks that performed well in detecting increases in mean packet arrival rate. Red synapses are inhibitory and blue synapses are excitatory. A histogram showing the distances of the path/cycle is also given for each common path/cycle. More common paths are in Figure 4.1. 165
4.3 Common structures (cycles or paths) appearing in seven of nine networks that performed well in detecting increases in mean packet arrival rate. Red synapses are inhibitory and blue synapses are excitatory. A histogram showing the distances of the path/cycle is also given for each common path/cycle. 166
4.4 Common paths from input to output for networks recognizing the digit 0 in images. The total number of networks in which that path appears is also given, along with a histogram of the delays for that path. . . . 169
4.5 Common paths from input to output for networks recognizing the digit 1 in images. The total number of networks in which that path appears is also given, along with a histogram of the delays for that path. . . . 169
4.6 Common paths from input to output for networks recognizing the digit 2 in images. The total number of networks in which that path appears is also given, along with a histogram of the delays for that path. . . . 170
4.7 Common paths from input to output for networks recognizing the digit 3 in images. The total number of networks in which that path appears is also given, along with a histogram of the delays for that path. . . . 170

4.8 Common paths from input to output for networks recognizing the digit 4 in images. The total number of networks in which that path appears is also given, along with a histogram of the delays for that path. . . . 171
4.9 Common paths from input to output for networks recognizing the digit 5 in images. The total number of networks in which that path appears is also given, along with a histogram of the delays for that path. . . . 171
4.10 Common path from input to output for networks recognizing the digit 6 in images. The total number of networks in which that path appears is also given, along with a histogram of the delays for that path. . . . 172
4.11 Common paths from input to output for networks recognizing the digit 7 in images. The total number of networks in which that path appears is also given, along with a histogram of the delays for that path. . . . 173
4.12 Common paths from input to output for networks recognizing the digit 8 in images. The total number of networks in which that path appears is also given, along with a histogram of the delays for that path. . . . 173
4.13 Common paths from input to output for networks recognizing the digit 9 in images. The total number of networks in which that path appears is also given, along with a histogram of the delays for that path. . . . 174
4.14 Network activity and simulation plot for time steps 100 to 200, when the simulation has just started. 176
4.15 Network activity and simulation plot for time steps 300 to 400, when an increase in mean arrival rate has occurred. 176
4.16 Network activity and simulation plot for time steps 500 to 600, when an increase in mean arrival rate has occurred more than 100 time steps previously. 177
4.17 Network activity and simulation plot for time steps 1000 to 1100, when an increase in mean arrival rate has occurred more than 100 time steps previously. 177

4.18 Network activity and simulation plot for time steps 1300 to 1400, when a decrease in mean arrival rate has just occurred. 178
4.19 Network activity and simulation plot for time steps 1600 to 1700, when a decrease in mean arrival rate has occurred more than 100 time steps previously. 178
4.20 A substructure (shown in yellow in the highlighted portion of the network) that is noticeably more active than other neurons in the network when processing the shown handwritten image of a zero. (Figure created by Margaret Drouhard and included with her permission.) . . 179
4.21 The same substructure as in Figure 4.20 that is, again, noticeably more active than other neurons in the network when processing the shown handwritten image of a zero. (Figure created by Margaret Drouhard and included with her permission.) 180
4.22 Activity of all neurons in the network depicted in Figures 4.20 and 4.21. The y-axis gives the neurons labeled by position in a three-dimensional cube. The three most active neurons are those contained in substructure s, highlighted in 4.20 and 4.21. 182
4.23 Example images of written 7's. The variations in the way 7's are drawn may contribute to the variation in causality paths that occur. 183
4.24 Event path from a network trained to recognize the digit 7 processing an input image of a 7. The event traced by the path is the first firing of the output neuron within the final time window, which signals that the network correctly identifies the image as a 7. (Figure created by Margaret Drouhard and included with her permission.) 185

4.25 Event path from a network trained to recognize the digit 7 processing an input image of the digit 2. The event traced by the path is the last output neuron fire. The fire occurs prior to the final time window, indicating that the network recognizes that the image does not show the digit 7. (Figure created by Margaret Drouhard and included with her permission.) 185
4.26 Subnetworks of networks trained to recognize various digits (0-3) in handwritten digit images. In these subnetworks, the neurons and synapses are sized based on the number of times they appear in a causality path for a firing event in the last 50 time steps of the simulation, which signifies a detection of a particular image. (Figure created in part by Margaret Drouhard and included with her permission.) 187
4.27 Subnetworks of networks trained to recognize various digits (4-7) in handwritten digit images. In these subnetworks, the neurons and synapses are sized based on the number of times they appear in a causality path for a firing event in the last 50 time steps of the simulation, which signifies a detection of a particular image. (Figure created in part by Margaret Drouhard and included with her permission.) 188
4.28 Subnetworks of networks trained to recognize various digits (8,9) in handwritten digit images. In these subnetworks, the neurons and synapses are sized based on the number of times they appear in a causality path for a firing event in the last 50 time steps of the simulation, which signifies a detection of a particular image. (Figure created in part by Margaret Drouhard and included with her permission.) 189
4.29 An example substructure, hand-designed to recognize a vertical line is given on the left. On the right is a network that is built from many of these substructures so that it will detect a vertical line in any location in a large grid. (Figure created in part by Margaret Drouhard and included with her permission.) 190

4.30 An example network is shown, along with the useful substructure extracted from the network based on N for that network is also given. (Figure created in part by Margaret Drouhard and included with her permission.) 192
4.31 A box plot showing the effect of the inclusion of the library on the double pole balancing task. The y-axis gives the difference in performance between the same seeded population when using the library and when not using the library. If the result is positive, it means the library helped, and if the result is negative it means the library was detrimental to the performance of the evolutionary algorithm. The median is shown as the red line for each digit, the box gives the quartiles, and outliers are also shown. The average is shown as a dot. 194
4.32 A bar graph showing the effect of the inclusion of the library on the double pole balancing task. The three bars shows the number of tests (out of 2000) in which the inclusion of the library helped performance, hurt performance, and had no effect on performance, respectively. . . 195
4.33 A box plot showing the effect of the inclusion of the library on the handwritten digit task, with a box for each digit type (0-9). The y-axis gives the difference in performance between the same seeded population when using the library and when not using the library. If the result is positive, it means the library helped and if the result is negative it mean the library was detrimental to the performance of the evolutionary algorithm. The median is shown as the red line for each digit, the box gives the quartiles, and outliers are also shown. The average is shown as a dot. 196

4.34 A bar graph showing the effect of the inclusion of the library on the handwritten digit task. The three bars shows the number of tests (out of 1000) in which the inclusion of the library helped performance, hurt performance, and had no effect on performance, respectively. Results are given for each digit type. 197
4.35 Graph showing activity in a single network, in which the nodes are events and the edges indicate causality. 199

5.1 Simplistic view of affective systems interacting with each other, the neocortex, and other parts of the brain using neurotransmitters. . . . 203
5.2 Simulation results for a single desired firing rate. Results for both $\lambda = 0$ and $\lambda = 0.9$ are shown. 208
5.3 Box and whisker plots for the final threshold value. Plots shown for 100 networks at simulation completion for various desired firing rates. Results shown are for $\lambda = 0.9$; results for $\lambda = 0$ are nearly identical. . 209
5.4 Simulation results for an increase in desired firing rates. Results for both small increases of 150 and large increases of 600 in the firing rate are shown. 211
5.5 Simulation results for a decrease in desired firing rates. Results for both small decreases of 150 and large decreases of 600 in the firing rate are shown. 211
5.6 Simulation results for many changes in the firing rate. Results shown are for a network with $\lambda = 0$. 212
5.7 The effect of α and λ on time to settle and percent overshoot. Results shown are for one network structure and one desired firing rate. . . . 214
5.8 Simulation results for two extreme values of α. Results shown are for one network structure and one desired firing rate with $\lambda = 0.9$. The two values of α show are $\alpha = 0.00025$ and $\alpha = 0.005$. 215

5.9 Histograms for the final synapse weights in the network at simulation completion. Results shown are for varying desired firing rates and one network structure with $\lambda = 0.9$. 216
5.10 Box and whisker plots for the final synapse weights of 100 networks at simulation completion. Results shown are for desired firing rate of 900 and $\lambda = 0.9$. 217
5.11 Simulation results for different LTP/LTD parameters. Results shown are for $\lambda = 0$. 218
5.12 Color map showing the changes in synapse weights over a single simulation. Results shown are for $\lambda = 0$. Note that the color bar is on a log scale. The firing rates over the course of the simulation are also shown. The LTP/LTD refractory period used is 10. 219
5.13 Color map showing the changes in synapse weights over a single simulation. Results shown are for $\lambda = 0$. Note that the color bar is on a log scale. The firing rates over the course of the simulation are also shown. The LTP/LTD refractory period used is 50. 220
5.14 Example NIDA network with an embedded subnetwork. (Figure created in part by Margaret Drouhard and included with her permission.) 222
5.15 Example of a computational network with placeholders for two types of affective systems (shown as dark red and light blue cubes in the network). (Figure created in part by Margaret Drouhard and included with her permission.) 225
5.16 A combined network, comprised of a computational network and affective system networks. This is the same computational network shown in Figure 5.15, but an affective network from each population has replaced the placeholders shown in that figure. (Figure created in part by Margaret Drouhard and included with her permission.) 227

5.17 Example of a combined network building method. Vectors are formed of IDs that represent networks in each population. The affective vectors are shuffled. These vectors are then used to decide how to construct the combined networks. 228
5.18 Selection method used to produce child populations from the current populations in the evolutionary optimization method updated to deal with complex affective systems. 230
5.19 Six starting conditions for the fitness function. This graph shows the six starting conditions that are tested as part of the fitness function for the simple pole balancing problem. $\dot{x}$ and $\dot{\theta}$ are assumed to be zero. 232
5.20 Box and whisker plots summarizing the complexity of the networks. The number of neurons, the number of synapses and the desired firing rate are shown for the sixteen networks trained with affective systems. 233
5.21 Box and whisker plots showing the results for networks tested with and without affective systems. The results shown in the box plots are the overall averages of the sixteen networks over 1260 test cases (10 random points chosen in each of 126 boxes). 234
5.22 Box and whisker plots showing the results for networks trained with and without affective systems. The results shown in the box plots are the overall average of the networks over 1260 test cases. Since only twelve of the networks trained without affective networks converged, the results are shown only for those twelve networks, as well as the twelve networks with affective systems with the corresponding random seeds. The square in the plot is an outlier. 235
5.23 Plot showing the generalization results on a network with an affective system. Each point on the contour plot is the average of 10 test cases. 237

5.24 Plot showing the generalization results on a network tested without an affective system. The network in this case is the same network as in Figure 5.23, but the affective system is not operating during the tests. Each point on the contour plot is the average of 10 test cases. 238
5.25 Plot showing the generalization results on a network trained without an affective system. The network was trained without an affective system, but the original population on the evolutionary algorithm was the same as the population for Figure 5.23 (they used the same random seed). Each point on the contour plot is the average of 10 test cases. . 239
5.26 Box plot showing the pole balancing performance of networks trained with and without complex affective networks. The averages plotted are over 1260 test cases (as described in Section 5.3.1). Average fitness value and average error of the pole from vertical per time step are shown. The box extends from the lower to upper quartile, with whiskers showing the range of data. Outliers are also plotted, and the average is plotted as a dot. 243
5.27 Heat maps showing the performance of the best performing network (with respect to time balanced from each starting condition) trained with complex affective systems. Each square in each heat map is the average of 10 test cases. 244
5.28 Heat maps showing the performance of the best performing network (with respect to time balanced from each starting condition) trained without complex affective systems. Each square in each heat map is the average of 10 test cases. 245
5.29 Heat maps showing the performance of the best performing network (with respect to error from vertical of the pole) trained with complex affective systems. Each square in each heat map is the average of 10 test cases. 245

5.30 Heat maps showing the performance of the best performing network (with respect to error from vertical of the pole) trained without complex affective systems. Each square in each heat map is the average of 10 test cases. 246
5.31 A computational network in the multiple interacting networks case is comprised only of placeholders, not neurons or synapses. In this example, there are four affective types, which correspond to four possible roles in the team. (Figure created in part by Margaret Drouhard and included with her permission.) 248

6.1 Grid structure of the hardware implementation based on the NIDA architecture. (Figure created by Dr. Doug Birdwell and Dr. Mark Dean and included with their permission.) 252
6.2 Segments of a grid of two networks in which line crossover produces a child with isolated neurons (neurons that are unable to connect to another element because they are surrounded by neurons instead of synapses). 255
6.3 An example of a network in which the paths defined make a large portion of the potential grid elements unusable. 256
6.4 An example of a 4x4 element block, where elements labeled N are neurons and elements labeled with $\uparrow$, $\rightarrow$, $\downarrow$, or $\leftarrow$ are synapses where the direction of the arrow indicates the direction of the synapse. . . . 257
6.5 Example cycles in a tiled example of the 4x4 element blocks. 257

6.6 A network that was partially trained to recognize changes in input firing rate. This network is made up of 4x4 blocks similar to those given in Figure 6.4. The parameters of the network (thresholds of neurons and weights and delays of synapses) were trained using evolutionary optimization. The neurons are represented by ovals and the synapses are represented by arrows. The red oval for the output indicates which synapse corresponds to the output synapse. The green oval is the input neuron. Neurons are labeled by their row and column in the grid. Excitatory synapses are blue, and inhibitory synapses are red. . 258

6.7 An example run of the network given in Figure 6.6. The green dots correspond to input pulses, the red dots correspond to output pulses and the blue line shows the packet firing rate. 259

7.1 NIDAs can be used to make decisions at each node of a decision tree. For example, the decision can be based on whether the output neuron of the network fires in a particular time window. Based on the decision, a new NIDA network can be simulated on the input to further classify it. 267

7.2 A GUI can be useful in which the user may select from various problems/tasks, which will create and distribute jobs to any available resources (e.g., nodes in a cluster), gather results, and display results in a meaningful manner. The current visual analytics tool would be included as a way to analyze networks and to display results meaningfully. 269

7.3 Graphic overviewing potential future directions for neuroscience-inspired dynamic architectures (NIDA). 270

7.4 Graphic overviewing potential future directions for the NIDA design method. 270

7.5 Graphic overviewing potential future directions for developing tasks and interaction tools for NIDA networks and the associated design method. 271
7.6 Graphic overviewing potential future directions for exploring useful substructures in NIDA networks. 275
7.7 Graphic overview of potential future directions for affective networks and multiple-interacting NIDA networks. 279
7.8 New element connectivity so that each element is connected to 16 other elements in its neighborhood. Four (offset) copies of the overlay (shown in bold) can be utilized to provide complete coverage of the network's elements. Alternatively, only a subset of the elements may implement larger-distance connectivity. The structure can be repeated with greater spacing between connected elements (and more copies for complete coverage) to provide even longer distance direct connections. (Figure created by Dr. Doug Birdwell and included with his permission.) 281
7.9 Graphic overviewing potential future directions related to the hardware implementation of NIDA networks, DANNA. 284
7.10 General real time learning scheme. 286

A.1 A typical biological neuron*. 335
A.2 The human brain†. 336

B.1 An example of a fully-connected feed-forward neural network. 342
B.2 An example of a recurrent neural network. The recurrent connections are bolded in the network. The delay elements are indicated with boxes labeled with D. 343
B.3 A typical neuron model. 344
B.4 Threshold activation function. 345
B.5 Piecewise-linear activation function. 345
B.6 Sigmoid activation function. 346

D.1 An agent decides its action based on the current state of the system. An action of the agent affects both the reward for the agent and the state of the environment. 362

1

Introduction

"Whatever the problem, be part of the solution. Don't just sit around raising questions and pointing out obstacles."

– Tina Fey, *Bossypants*

Biological brains are some of the most powerful computational devices on Earth. They are able to perform complex, nonlinear tasks using large numbers of relatively simple building blocks. They are robust, able to extrapolate information from specific situations/examples and apply that information to other, related situations/examples, and are adaptable to changes in environments. Although we have considerably more devices available today to aid in our understanding of the brain and its many functions, we still do not fully understand how the brain operates.

"The study of the mind and brain is the last frontier in science. Although the field has made enormous progress over the past several decades, understanding of the basic principles of thought and brain function are still far more unknown than known."

– Michael Tarr*

While neuroscientists and computational neuroscientists puzzle over how the brain works, computer scientists have long recognized the great value of the brain as a computational device.

"I will discuss the points of similarity between these two kinds of 'automata' [modern computing machines and the human nervous system]....There are elements of dissimilarity...not only in rather obvious respects of size and speed but also in much deeper-lying areas: These involve principles of functioning and control, of over-all organization, etc."

– John von Neumann, *The Computer and the Brain* [260]

This recognition was the catalyst of the development of the artificial intelligence and machine learning communities. Many different types of artificial neural network architectures and learning methods have been proposed (such as spiking neural network models, deep machine learning, and hierarchical temporal memory). These architectures and methods are capable of powerful computations and exhibit some of the characteristics of the brain that are so desirable, such as generalizations from specific instances and robustness.

Since the inception of the science of computation, there have been ebbs and flows in the development of artificial neural networks. They have long been recognized as potentially valuable computational tools, but interest waned as desired results were not forthcoming (see Appendix B.4). A resurgence of interest in artificial neural network architectures occurred in recent years. This resurgence has been attributed to

*Co-director of Carnegie Mellon's Center for the Neural Basis of Cognition, quoted in 2010. Quoted from http://www.cmu.edu/homepage/health/2010/summer/the-last-frontier.shtml.

several factors. Notably, the focused interest of neuroscientists on understanding the brain helped spur the renewal of artificial neural network research. As researchers discover more detailed information about how the brain works, more complex and biologically-driven neural network architectures can be developed. The introduction of deep machine learning and deep belief networks and their powerful classification abilities has also contributed to the resurgence [28]. A third generation of artificial neural networks, spiking neural networks, has emerged in the last few decades. Spiking neural networks have inherently dynamic properties based on those same properties in the brain [99, 149]. This focus on a new type of neural network has also contributed to the renewal of interest in neural network architectures.

Recent realizations regarding hardware have contributed to the resurgence of interest in neural network architectures.

> *"Things seem to be slowing down. You might characterize it as maybe a sign that Moore's Law is having some issues."*
>
> – Jack Dongarra[†]

Researchers have observed a plateauing of the performance of von Neumann architectures over the last decade. Moore's Law, the observation that the number of transistors per integrated circuit was doubling approximately every two years or 18 months, is either coming to an end or has already ended. The steady increase in computing power and capability that we have seen since the beginning of computer science is at an end, at least for von Neumann architectures. Hardware developers have sought to combat this decline by increasing the number of cores per machine. That is, instead of increasing the computing power of a single core, we are now producing many core machines. Unfortunately, it can be extremely difficult to write software that fully exploits the parallel nature of our new machines.

[†]Director of the Innovative Computing Laboratory, quoted in 2014. Quoted from http://www.wired.com/2014/06/supercomputer_race/

> *"There is no guarantee that we can make future parallel computing ubiquitous and as easy to use as yesterday's sequential computer, but unless we aggressively pursue efforts [such as development of parallel architectures], it will be game over for future growth in computing performance."*
>
> – Report by the National Academy of Sciences[‡]

At the same time that our computing capabilities are leveling off and our skills as parallel programmers are failing us, we are accumulating data at an astounding rate.

> *"As of 2012, about 2.5 exabytes of data are created each day, and that number is doubling every 40 months or so. More data cross the internet every second than were stored in the entire internet just 20 years ago."*
>
> – Andrew McAfee and Erik Brynjolfsson, Harvard Business Review[§]

Ideally, we would like to process and/or analyze these data, perhaps in real-time. Much of this analysis requires large-scale clusters or super computers, which are power-hungry and do not adequately scale to quickly processing large amounts of data because of communication delays and bandwidth limits between cores. Beyond the issue of "big data," consumers are expecting smarter and smarter devices, from self-driving cars to smart homes, all of which may benefit from low-power, fault-tolerant, adaptable devices that are not currently feasible with traditional von Neumann architecture machines.

A variety of new architectures are being proposed to replace the traditional von Neumann architecture, such as quantum computing and molecular computing. Neuroscience and artificial neural networks have inspired one of the major proposed non-von Neumann approaches: neuromorphic computing.

[‡] "The Future of Computing Performance: Game Over or Next Level?", *National Academy Press*, 2011

[§] Quoted in 2012, from http://hbr.org/2012/10/big-data-the-management-revolution/ar

"The guiding philosophy [of neuromorphic computing] is not to reiterate or simulate the brain in complete detail, but to search for organizing principles that can be applied in practical devices."

– Don Monroe in [173]

These architectures tend to be based on spiking neural networks and can have analog implementations. Some of the efforts are more biology-driven than computation-driven, while others are implementing more traditional artificial neural network architectures (see Section 2.6). These architectures are inherently non-von Neumann, as the computational units and memory are integrated (in the neurons and synapses). There is some question, however, as to how these new hardware devices will be programmed. A new architecture's utility cannot be verified until it can be shown to be useful on a wide variety of applications, *and* it can be shown that new applications can be developed at reasonable cost (in terms of both time and computational power) for that device (that is, that they can be programmed relatively easily).

The importance of dynamics in the brain and thus in neural network architectures has been noted from the beginning of the development of artificial neural networks.

"In a single system, and with a constant set of connections between neurons in the system, the direction in which an entering excitation will be conducted may be completely dependent on the timing of other excitations. Connections are necessary but may not be decisive in themselves; in a complex system, especially, time factors must always influence the direction of conduction. The older ideas of neural transmission gave synaptic connections too much rigidity as a determinant of behavior."

– D.O. Hebb, *The Organization of Behavior: A Neurophysiological Theory* [113]

This work introduces a novel neural network architecture, neuroscience-inspired dynamic architectures (NIDA). Two major features of NIDA networks are dynamic

behavior and embedding in a three-dimensional space. Both of these characteristics are biologically-inspired. NIDA networks are event-driven, where events include spikes or firing events in the network. They are efficiently simulated using discrete-event simulations. NIDA networks have an inherent notion of delay, which is related to distance in the geometric space in which the networks are embedded, so spatial relationships play a major role in the operation of the network. By including a notion of space, we may also leverage findings from neuroscience about the underlying architecture and connectivity of the brain. As noted above, understanding the brain is one of the "last frontiers" in science; there are significant characteristics of the brain and the behavior of the brain that have not yet been discovered. As discoveries about structural properties of the brain are made (such as topographic maps of sensory input [125, 54], rich-club neurons [107], and connectivity patterns [242]), we can exploit these structural properties in our NIDA networks because of the spatial relationships the architecture allows.

The development of a new neural network architecture requires the development of a training/learning method for that architecture (that is, a way to program the architecture for different applications). We propose an evolutionary optimization method for training NIDA networks. This optimization is inspired by works in neuroevolution, a set of methods in which an evolutionary strategy is used to train artificial neural networks. Our evolutionary optimization design method discovers not only parameter values of the networks; it discovers the structure of the network and thus the dynamics of the network as well. We chose an evolutionary optimization method because it allowed a degree of flexibility in designing these networks that was not available through other design methods. Moreover, the use of evolutionary optimization to design neural networks evokes a biologically-inspired model, as biological brains were also evolved as opposed to pre-designed.

"The evolutionary algorithm– of variation and selection, repeated– searches for solutions in a world where the problems keep changing, trying all sorts of variants and doing more of what works."

– Tim Harford, *Adapt: Why Success Always Starts with Failure* [106]

One of the major pitfalls of evolutionary optimization is that it can often result in slow training time. We propose and explore several ways in which the speed of this optimization may be able to be improved. One possible way of improving evolution is to discover useful substructures with well-performing networks and exploit them by reusing those substructures in other networks.

"Lack of modularity is one of the reasons that computational evolution can currently generate designs for simple tasks, but has difficulty in scaling up to higher complexity."

– Nadav Kashtan and Uri Alon [131]

The replication or reuse of useful substructures has a biological motivation as well. We recognize in the brain that there are many patterns of repeated structure, such as cortical columns, as well as structures that serve one purpose but may also be useful in other circumstances (see Section 2.4 for more detail). By building smaller networks and using them as building blocks for larger, more complex networks in the evolutionary optimization or by extracting useful subnetworks and replicating or reusing them in other networks, we may be able to improve the evolutionary optimization design method.

"The lower levels in the neural edifice of reason are the same ones that regulate the processing of emotions and feelings, along with the body functions necessary for an organism's survival. In turn, these lower levels maintain direct and mutual relationships with virtually every bodily organ, thus placing the body directly within the chain of operations that generate the highest reaches of reasoning, decision making, and, by extension, social behavior

and creativity. Emotion, feeling and biological regulation all play a role in human reason."

– Anthony Damasio, *Descartes' Error: Emotion, Reason, and The Human Brain* [63]

We also consider the inclusion of affective systems within the NIDA network architecture. Affective systems are the biological systems that are related to emotions in the brain and are primarily linked with the limbic system. We explore different potential affective system types, as well as their effect on learning and optimization. We propose the use of affective systems as a means for modulating behavior of a network based on sensory information and environmental cues that may not be directly related to the task at hand.

As noted above, the global community is searching for alternatives to the von Neumann architecture as Moore's Law ends and the amount of data gathered skyrockets. We have entered the foray of neuromorphic computing by developing a hardware implementation of NIDA networks. This hardware implementation, Dynamic Adaptive Neural Network Array (DANNA), developed in collaboration with Dr. Mark Dean, shares many characteristics with NIDA networks. Moreover, DANNAs may be produced with a similar design method based on evolutionary optimization so that they may be programmed to complete tasks. This allows us to overcome at least one hurdle of neuromorphic systems, the problem of how to program the device.

We have provided a brief perspective on the current state of research in neuroscience, computer science, and neuromorphic architectures. The decline of Moore's Law and the limits of von Neumann architectures are motivations for the development of a new architecture and design method that may be extended to a hardware model. Another aim is to develop a flexible architecture and design method that will work for a wide variety of tasks. This spurred our development of a spiking neural network model trained with evolutionary optimization. By using

an evolutionary strategy that trains over structures, dynamics, and parameters, we take the burden of choosing a structure and network type off the user and place that burden on the learning method. Since the optimization method determines the characteristics of the network, it can avoid biases and limitations that hand-tooled networks may have.

This area is ripe for exploration and research, which is the primary goal of this work. In particular, we sought to explore a new architecture and design method inspired by a significant amount of related work in neuroscience and computer science and demonstrated its utility on an initial set of applications. We explored ways to better understand the behavior of the architecture and method (through the recognition of useful substructures), proposed a way to improve the design method (through the reuse of useful substructures), and proposed the inclusion of affective-like systems. We have just begun to explore the opportunities for hardware implementations based on this architecture. This work is the tip of an iceberg of potential future work, and as such does not answer conclusively whether this architecture and design method are definitively better than other proposed methods. However, we strongly believe that the results presented herein justify further exploration.

This work is organized as follows. Chapter 2 provides an overview of relevant work. In particular, this chapter covers related work in neuroevolution, networks in geometric space, dynamic neural networks, artificial neural networks exploiting regularity or modularity, and affective-related machine learning literature. We also cover related works in neuromorphic hardware.

Chapter 3 introduces the neuroscience-inspired dynamic architecture and its associated design method based on evolutionary optimization. This chapter demonstrates the utility of NIDA networks on several different applications. Chapter 4 discusses the recognition and reuse of useful substructures in NIDA networks. Chapter 5 presents multiple artificial affective system methodologies, as well as results on the effect of those affective systems on NIDA networks. It also extends the idea

of affective networks to the development of teams of multiple interacting NIDA networks. Chapter 6 provides a brief introduction to the hardware implementation based on NIDA networks, Dynamic Adaptive Neural Network Array (DANNA), and the associated software developed alongside DANNA is discussed.

This work generated many future research directions, which are detailed in Chapter 7. Finally, Chapter 8 provides a brief overview of the work and offers a perspective on future research. The appendices provide background on relevant topics: neuroscience (Appendix A), artificial neural networks (Appendix B), evolutionary algorithms (Appendix C), and reinforcement learning (Appendix D). These provide a convenient source for readers to refresh their knowledge of these topics. Appendix E also provides some theoretical results about NIDA and its associated design method that may be useful for future work.

It is clear that discovery and exploration of new approaches for the development of computing devices are essential.[¶] It is also clear that users and consumers are expecting greater intelligence in the operation of their computing devices. The biological brain has long been recognized as an extremely efficient computing device with many desirable properties that complement the computational strengths of von Neumann systems. What is unclear, however, is how computing devices that exhibit similar behavior as the biological brain will be developed and emerge in the market place. We doubt the need to simulate the details of the biological brain to elicit complex behaviors. What properties are important? How do we efficiently teach these devices to complete tasks? A winning solution has not emerged. It will take significant investments to fully answer these questions. This work takes a moderate step in this direction and proposes possible answers to some of the questions.

> *"It is common sense to take a method and try it. If it fails, admit it frankly and try another. But above all, try something."*
>
> – Franklin Roosevelt, *Looking Forward*

[¶]See previously cited National Academy of Science report for more details.

2

Related Work

"And all dared to brave unknown terrors, to do mighty deeds, to boldly split infinitives that no man had split before – and thus was the Empire forged."

– Douglas Adams, *The Hitchhiker's Guide to the Galaxy*

The neuroscience-inspired dynamic architecture (NIDA) and design method introduced in this work are based on five concepts:

1. Neural network structures evolve over the course of training.
2. Neural networks can be embedded into a geometric space.
3. Neural networks are composed of dynamic elements and operate on a continuous time scale.
4. Useful substructures in neural network structures can be recognized and reused.
5. Multiple neural network structures having different goals can interact via connections that exchange information between networks (e.g. affective systems in biological networks).

In the following sections, the biological inspiration and justification of each of these ideas is presented. Then, a description of related work for that idea is given. We are also developing a hardware implementation based on NIDA, which fits in the realm of neuromorphic computing. As such, other efforts in neuromorphic computing are also discussed in this chapter.

2.1 Neuroevolution

We may categorize evolution of the structure of the brain in four forms (see Appendix C for more detail):

- Long-term evolution: Affects the gross structure of the brain and may occur over hundreds to millions of years of evolution of species.
- Moderate-term evolution: Also called epigenesis, affects the structure of the brain at a finer level, and occurs over generations.
- Short-term evolution: The evolution that occurs over a person's lifetime, affecting the number and connectivity of neurons in the brain.
- Very short-term evolution: Evolution or learning that occurs on a day-to-day basis in the brain.

Neuroevolution algorithms use evolutionary algorithms to train neural networks. The first neuroevolution algorithms that were developed only evolved the strength of the connections between the neurons; they did not affect the structure by adding or deleting connections or neurons. They only dealt with one form of evolution described above: very short-term evolution.

The training of the connection weights in neural networks is typically formulated as an optimization problem. In particular, some error is minimized. The error used may be the mean squared error between the actual output and the expected output in supervised learning or the temporal difference error as used in reinforcement

learning [248]. The weights of the networks are then trained using algorithms such as back propagation [216] or conjugate gradient [180]. These algorithms rely on gradient-based optimization algorithms using steepest or gradient related descent directions [31]. There are many drawbacks to using these gradient-based optimization algorithms. In particular, gradient-based algorithms rely on the differentiability of error functions, and they are likely to converge to local optima.

Evolutionary algorithms had been applied in the field of optimization to similarly complex problems, as they are less likely to become trapped in non-optimal solutions. It was a natural extension to apply evolutionary algorithms to weight training in neural networks, as this problem can be formulated as an optimization problem through which an error is minimized. Yao presented a thorough overview of algorithms that use evolutionary algorithms to train the weights of neural networks [274]. Yao notes several advantages of evolutionary algorithms over gradient-based algorithms:

1. Evolutionary algorithms do not depend on gradient information, which may be unavailable or difficult to calculate.

2. Evolutionary algorithms can be applied to any neural network architecture, whereas gradient-based algorithms have to be adapted for different architectures.

3. Evolutionary algorithms are much less sensitive to initial conditions.

4. Evolutionary algorithms always search for global optima, rather than local optima.

It is important to note that evolutionary algorithms rely on a fitness function, rather than an error. This fitness function can often be easily translated to reinforcement learning problems, where the fitness function is the reward received over the course of the episode or the value of the reinforcement learning value function at that state.

In [174], Montana and Davis were motivated to use genetic algorithms because of the gradient-based algorithms' propensity for finding local, rather than global, optima. Their experiments showed that genetic algorithms converged in fewer iterations than standard back-propagation on a typical classification (supervised learning) task.

In [81], Fogel *et al.* note that back-propagation algorithms are limited to multiple-layer perceptron networks, and although other techniques can be constructed for different neural network architectures, it is difficult to construct these algorithms. This restriction to certain architectures, as well as the known limitation of finding local minima, inspired the authors to use an evolutionary algorithm to train the weights and biases of a neural network, as the algorithm can adapt to any network architecture without added complexity to the algorithm itself. This allows for an easy extension to recurrent neural network structures, for example, for which learning with gradient-based algorithms has been shown to be difficult [27]. Their algorithm is restricted to mutation of only the weights.

In [25], Beer and Gallagher note that recurrent neural networks are a good mechanism for adaptive neural controllers. They use a genetic algorithm to evolve the weights of a continuous time neural network. This work will be discussed in more detail in Section 2.3.1.

In [266], Wieland notes that traditional algorithms are difficult to use when creating neural networks to control complex systems, which led to his use of a genetic algorithm to train the weights in a recurrent neural network. He successfully applied his method to solve several variations of the inverted pendulum control task. In [218], Saravanan and Fogel note a similar difficulty with back-propagation and other gradient-based algorithms in that these algorithms often require an accurate model of a control system's dynamics. They also used evolutionary algorithms to overcome this difficulty and similarly, they applied their method to the inverted pendulum control task. In [117], Igel applied an evolution strategy for training the weights in the neural networks and successfully trained networks to solve several variations of the inverted pendulum control task.

In [67], Dominic *et al.* also applied a genetic algorithm to train the weights of a neural network. They noted that genetic algorithms are especially useful in reinforcement learning tasks, as opposed to supervised learning tasks, because gradient information is not directly available for these tasks. The authors demonstrated that for the inverted pendulum task, the genetic algorithm produced a network in a comparable number of steps as a temporal difference method. Furthermore, the genetic algorithm was more robust than the temporal difference method over a number of learning conditions. This robustness is one reason evolutionary algorithms as a training algorithm are more attractive than other methods.

In [102] and [103], Gomez *et al.* introduce an algorithm called Cooperative Synapse NeuroEvolution (**CoSyNE**), which uses both crossover and mutation to evolve the weights in a fixed architecture neural network. The authors compare the performance of CoSyNE to a wide variety of neuroevolution techniques and also to a wide variety of reinforcement learning techniques for solving the inverted pendulum problem. They found that the other neuroevolution techniques in general, and CoSyNE in particular, were more efficient (several magnitudes smaller in CPU time) and reliable (able to tackle more complicated tasks) than typical reinforcement learning techniques, such as Q-learning and SARSA(λ) on this task. The results of these papers demonstrate that neuroevolution can be an important tool for reinforcement learning tasks, especially those with large state-spaces and non-Markov environments.

All of the previous algorithms dealt with only one form of evolution: very short-term evolution. The structure of the network was fixed. The structure of the network includes the general architecture (i.e., feed-forward, recurrent, etc.), the number and layout of neurons (i.e., how many neurons should be included in a particular layer), and the number and nature of the connections (i.e., how the neurons should be connected). For these types of algorithms the structure of the neural network is mostly determined via experimentation. That is, a certain structure is tested, and if that structure does not work, more neurons or connections are added manually, increasing the complexity, until the network is able to handle the problem. This

requires significant hand-tuning by the experimenter/researcher. Knowledge about the problem can be applied and intuition developed to decide what type of structure is required by certain problems. For each problem, a new structure needs to be determined and the selection of this structure relies entirely upon the knowledge of the structure designer. In [117], Igel notes that the structure of the network made a large difference in the performance of his genetic algorithm. Networks with and without bias parameters and networks with different numbers of hidden neurons performed very differently in his experiments. Because the structure has such a large effect on the efficacy of the network, an algorithm that learns what structure is needed to solve a particular problem is much more attractive than an algorithm that relies on prior knowledge or hand-tuning to design a structure.

Hyperparameter optimization is an optimization over the parameters of a learning system, such as the learning rate or number of neurons in a neural network [30]. A second optimization algorithm (e.g., back-propagation) is nested within the hyperparameter optimization to perform the actual learning process to define the weight values of the network. Most hyperparameter optimization is done simply by trying all sets of combinations of values for hyperparameters and selecting the set of values that produces the results closest to those desired. Hyperparameter optimization based on random searches has had success in other fields, including support vector machines [83] and deep belief networks [30]. Many neuroevolution methods could be considered hyperparameter optimization methods, where the hyperparameter is the number of neurons per layers or the number of layers in the network. However, rather than trying all combinations of sets of possible values for those parameters, neuroevolution uses a randomized search (evolutionary optimization) to find desirable values. In most neuroevolution methods (though not all) this optimization is done simultaneously with the weight training, rather than in a nested loop (e.g., where the outer loop finds the number of hidden neurons and the inner loop optimizes the weights of the structure defined).

Constructive and destructive algorithms are algorithms that attempt to learn a network structure, rather than relying on the trial and error approach. Constructive algorithms start with very small networks and increase their size by adding neurons and connections as needed for a particular problem [74, 189]. Destructive algorithms such as pruning begin with overly complex networks. Connections and neurons are then deleted to yield a minimal structure [61]. These constructive and destructive algorithms would seem to solve the problem of finding a neural network architecture to use. However, there is a fundamental issue with these algorithms. Constructive and destructive algorithms follow strict sets of rules; for example, a constructive algorithm may only be able to add a single neuron at a time to a hidden layer. These algorithms therefore only explore a strict subset of possible architectures. Some authors [1, 11] assert that they tend to become stuck in structural local optima. Essentially, constructive and destructive algorithms are similar to gradient-based algorithms such as back-propagation in that they may not fully explore the search space while searching for optima.

Just as evolutionary algorithms have been applied in the case of weight training for neural networks, evolutionary algorithms have also been developed to learn the structure of a neural network for a particular problem. Yao [274] and Floreano *et al.* [77] give thorough descriptions of algorithms that evolve both the weights and the structures of networks.

There are several drawbacks to using evolutionary algorithms. Although the final overall solution may be more optimal than the solution reached by a gradient-based algorithm, evolutionary algorithms typically take longer to find a solution. Applying evolutionary algorithms to neural networks in particular comes with a variety of issues. Branke [44] points out several considerations that need to be made when combining neural networks and evolutionary algorithms. He notes that the important factors are:

1. Representation: How to represent the networks in the population. Evolutionary algorithms usually work with strings of real or binary numbers.

2. Performance: How to measure performance. There has to be a performance metric to gauge how "fit" a member of the population is.

3. Creation of offspring: How to create offspring in a population. This is usually done through mutation, crossover (recombination), or both.

These are major issues that must be addressed by every neuroevolutionary algorithm. Representations of a network need to maintain a link to the functionality of the network; otherwise, operations such as crossover will have no meaning. Performance is a key metric and is a problem-specific issue. For example, supervised learning problems have an associated error, which would need to be converted into a fitness, while reinforcement learning problems have associated rewards, which would also need to be converted to fitnesses. The mechanisms of offspring creation are usually closely related to the representation of the networks in populations.

Branke also points out one of the major concerns with using evolutionary algorithms for neural networks: the competing conventions problem, where two functionally equivalent networks can have structural differences or different representations. Crossover operations applied to two networks that are functionally equivalent but structurally different can frequently yield offspring that do not make any sense. Branke notes that this is a major problem to avoid when designing an evolutionary algorithm.

Since time required for training is often an issue, several neuroevolution methods combine evolutionary algorithms and gradient-based searches in order to improve learning speed. The evolutionary algorithms are often used for global searches, while the gradient-based searches are used for fine-turning. We note in Section 7 that it may be prudent to combine multiple types of learning algorithms in order to improve learning speed for our networks.

In [265], Whitley *et al.* propose a genetic algorithm that, on a pre-defined set of neurons, discovers a combination of connections that is sufficient to solve the problem using back-propagation. They use recombination, and as a result, connections are added and deleted with each generation. They use the algorithm to learn smaller neural networks for solving a problem and to discover topologies that increase the learning speed (in terms of number of back-propagation training cycles) of the networks.

Koza and Rice give an algorithm for evolving the weights and architecture of a network, in which the number of layers, the number of neurons per layer, and the connectivity between neurons can be evolved. They use their algorithm to generate a network that serves as a one-bit adder [139].

In [64], Dasgupta and McGregor design an algorithm that evolves the structure and weights of a network simultaneously, but in their structural evolution, they only consider the connectivity of a fixed set of neurons. They apply their algorithm to two simple classification (supervised learning) tasks.

In [264], White and Ligomenides develop a hybrid algorithm called **GANNet** (Genetic Algorithm Neural Network) which develops feed-forward neural networks, and then uses other techniques, such as back-propagation, to refine the weights of the network. The idea was to use the genetic algorithm to get close to global optima and then use a gradient-based technique to refine the search.

Maniezzo introduces an algorithm called **ANNA ELEONORA** (Artificial Neural Networks Adaptation: Evolutionary LEarning Of Neural Optimal Running Abilities) in [157]. This algorithm has both serial and parallel implementations. ANNA ELEONORA uses a technique called granularity encoding that allows the algorithm to identify a good length of a coding string for the networks. Maniezzo uses his algorithm to learn Boolean functions, to learn neural controllers, and to interpret EEG data. Parallel implementations are often suited to evolutionary algorithms because the populations may be large. We implement parallel versions of our algorithm as well.

In [11], Angeline *et al.* develop a neuroevolution algorithm called **GNARL** (GeNeralized Acquisition of Recurrent Links). The resulting networks of GNARL have "gnarled" interconnections, rather than symmetric connections developed by other algorithms. Both the weights and the structures are evolved simultaneously in GNARL. They apply their method successfully to several different types of problems, including language induction and search and collection. This is one of the seminal works in the field, and one of the major sources of inspiration for our work.

In [250], Tang *et al.* develop an algorithm for optimizing neural network topology. They divide their encoding into two types of genes: control genes, which are the genes for layers and neurons in the structure, and connection genes, which represent the weights between neurons. This structure maintains a balance between learning new structure and exploiting the structure already learned, analogous to the idea of the exploration vs. exploitation problem discussed in Appendix D. They use their algorithm to develop neural networks that implemented different Boolean functions.

Liu and Yao introduce a population-based learning algorithm (**PBLA**) that learns weights and architectures [146]. However, each individual in the population is evaluated after partial training, which is done using a modified version of back-propagation. If a network is not performing well enough using just back-propagation (i.e., the error between the expected and produced value has not lowered significantly), simulated annealing is tried. Finally, if it is still not performing well, the architecture is mutated. They use this approach to attempt to reduce the computational cost of the evolutionary algorithm. They successfully apply their algorithm to several parity tasks. This approach is similar to the proposed hierarchical evolutionary strategy discussed above, in that different types of evolution (very short-term, short-term, and moderate-term) are tried. In particular, the combination of a genetic algorithm at a higher level and another algorithm, such as simulated annealing, at a lower level is one that we plan to explore in future research.

Moriarty and Miikulainen developed **SANE** (Symbiotic, Adaptive Neuro-Evolution). SANE evolves a population of neurons with a genetic algorithm, rather than a

population of networks. The neurons are combined using neural network blueprints, which track the neurons that perform well together, to form networks. They evolve neurons rather than networks because no single neuron can solve the problem, so diversity is maintained. They compared SANE to reinforcement learning methods such as Q-learning, as well as to existing neuroevolution methods of that time, and found it to be more efficient at solving the inverted pendulum problem [177] and a mobile robotics task [176]. SANE is a symbiotic evolution technique, in that the individuals of the population cooperate rather than compete. Symbiotic evolution allows diversity to remain, forming "species" in the population that work together. This is analogous to biological systems, in which a variety of species interact and depend on one another to succeed. This algorithm trades off network complexity for neuron complexity. In our work, we do the opposite; our proposed neuron structure is very simple, and we combine these simple structures to form more complex networks. However, this does relate to our work in useful substructures, except we evolve subnetworks that work together rather than single neurons.

Gomez and Miikklulainen introduce a neuroevolution algorithm based on SANE called **ESP** (Enforced Sub-Populations) in [101]. They also introduce a more complex version of the pole-balancing or inverted pendulum problem in which the cart and pole move in two dimensions. ESP, like SANE, evolves a population of neurons rather than a population of networks. The neurons are then combined to form networks. ESP also introduces the idea of using sub-populations (or species) to maintain diversity.

Yao and Liu introduce an evolutionary system called **EPNet** for evolving the architecture and weights of feed-forward artificial neural networks in [273]. In this paper, they attempt to maintain a behavioral link between parent and child by using node splitting rather than adding a fully connected node to a layer. EPNet also encourages simplicity in the network by always testing to see if a deletion will improve the network before testing an addition. They applied EPNet successfully to parity problems, medical diagnosis problems and time series prediction problems. They found that their networks generalized better than other networks developed or trained

using other methods. This is one of the main reasons a neuroevolution approach was selected in this work.

In [276], Yao and Liu introduce five mutation operations that, as in [273], are chosen in succession to maintain simpler networks if possible. The five mutation operators they introduce (given in the order they are tried) are: hybrid training (train using a modified back propagation algorithm), neuron deletion, connection deletion, connection addition, and neuron addition. We use some similar mutation operations (e.g., synapse addition/deletion, neuron addition/deletion), but only a single mutation is tried at each step, and the mutations are selected based on user-defined probabilities.

Pujol and Poli use a two-dimensional grid as the chromosome for their neuroevolution algorithm. The grid allows both mutation and crossover in the network structures. They successfully applied their neuroevolution algorithm to binary classification problems [200]. We use a similar grid-like structure that allows for mutation and crossover in NIDA, though our grid-structure is on the network itself (rather than on an indirect representation such as a chromosome). A two-dimensional grid structure is also used in our hardware implementation, DANNA, described in Chapter 6.

In [1], Abbass claims that the complexity of a network's architecture greatly affects the generalization ability of a network in that simpler networks frequently generalize better than more complex networks (a claim that is often made in the field). He formulates the problem of evolving a network that will solve the task as well as generalize well (i.e., will be a relatively simple network) as a multi-objective optimization problem. He then applies an evolutionary algorithm to the multi-objective optimization problem, which trains both the architecture and the weights simultaneously. He applies his approach to classification of medical data, a supervised learning task.

Stanley and Miikkulainen introduced **NEAT** (NeuroEvolution of Augmenting Topologies) in [245]. NEAT begins with simple structures and adds structure as

needed for a particular problem. They use both crossover and mutation in their algorithm. An innovation that NEAT uses is speciation, which allows structures to develop before they are deemed unfit by the genetic algorithm. NEAT avoids the competing conventions problem by maintaining a global innovation number for a gene, which gives the structures historical markings to know which genes can be crossed over. They also use the historical markings to maintain species. Because NEAT starts from a minimal architecture and only adds structure as needed, the dimensionality of the search space is minimized, making the search simpler. In [246], NEAT was augmented so that it also learned learning rules for each connection in the network.

Leung *et al.* introduce a genetic algorithm that learns both the weights and the structure of the network [143]. In their algorithm, the connectivity of the neurons is evolved, but the number of neurons is fixed (or altered by hand). They introduce switches to the links to allow for changes in connectivity. They apply their algorithm to a forecasting problem, and they also use their algorithm to realize an associative memory (a problem we are also planning to tackle; see Section 7.1).

In [5], Alba and Chicano propose hybrid methods, in which a genetic algorithm is used to avoid local minima, and then a gradient-based method is used to refine the search, avoiding the pitfall of the potentially slow convergence of an evolutionary method. The only pitfall of gradient-based algorithms that this avoids is the propensity to find local, rather than global, optima, so it is not applicable in more complex problems. However, they apply their method successfully to classification tasks in medicine.

In [75], Fieldsend and Singh again formulate the task of finding structure and weights for a neural network as a multi-objective optimization problem. Their algorithm adds and deletes neurons, adds and deletes connections, and adjusts the weights of connections in the networks. They apply their technique to a forecasting task, generating populations of evolutionary neural networks to forecast international stock indexes.

Garcia-Pedrajas *et al.* present an evolutionary algorithm that uses combinatorial optimization to avoid the competing conventions problem (also known as the permutation problem) [92]. In particular, they define a maximum number of hidden nodes, each of which has a set of parameter values (weight values coming into that node and weight values going out of that node). These nodes are then either included or excluded during their crossover operation, which they implement using either a genetic algorithm or simulated annealing to do the combinatorial optimization. They compare this technique with classical crossover operations on 25 real-world classification problems, including the three benchmark problems we apply NIDA to in Section 3.6.

In [186], Palmes *et al.* developed **MGNN** (Mutation-based Genetic Neural Network). MGNN evolves structure and weights at the same time using only mutation, not crossover. The encoding scheme used in MGNN and the use of the genetic algorithm allows for a less rigid and thus faster fitness computation, making MGNN more efficient than other evolutionary algorithms that have fitness functions requiring more computation. The authors applied MGNN to several classification problems successfully.

In Kassahun's dissertation, he introduces **EANT** (Evolutionary Acquisition of Neural Topologies) [132]. EANT uses a genetic encoding that allows a network to be evaluated without decoding. EANT does structural modification and weight modification at different time scales, exploiting not only ideas of long-term evolution and very short-term evolution introduced at the beginning of this section, but also how the two operations occur in time in relation to each other. Siebel and Sommer improve upon EANT with EANT2 in [236]. Rather than using evolution strategies for weight optimization, EANT2 uses CMA-ES (Covariance Matrix Adaptation Evolution Strategy), which avoids a random adaptation of the weights. EANT2 is compared to **NEAT** on a visual servoing task; on this task, it outperforms NEAT. In [237], Siebel *et al.* extend their work on EANT2 by introducing a pruning of unnecessary connections from the networks. They note that pruning the network

improves the numerical stability of the parameter optimization process. We have implemented structural pruning in our network, but it may also be useful to prune based on activity in the network (see Section 7.1).

In [244], Stanley *et al.* note a weakness of evolutionary algorithms: they do not scale well to very large networks. In this paper, the authors introduce HyperNEAT, which uses compositional pattern producing networks (CPPNs) to produce connectivity patterns, rather than networks. The advantage is that CPPNs can shift up to higher dimensions without further evolution. HyperNEAT is applied to a visual discrimination task and a food gathering task. Both of these tasks can scale up by increasing the resolution sampled. For the visual discrimination task, a network could be trained at a low resolution and then scaled up to a higher resolution task, where it would perform almost as well as on the low resolution task with no extra training.

Hausknecht *et.al.* compare four different neuroevolution algorithms to temporal difference learning methods, planning algorithms, and human expertise on Atari game playing [109]. Specifically, CMA-ES for weight-training, NEAT, and HyperNEAT are examined, as well as Conventional-Neuroevolution (CNE) in which the topology of the network is fixed and only the weights are updated. Specifically, they examined the use of feed-forward neural networks in playing a set of Atari games. They were able to show that neuroevolution methods generally outperformed temporal difference methods and, in some cases, outperformed planning algorithms as well. Neuroevolution methods also outperformed the best human results on four of the 55 games tested. Moreover, they were able to show that neuroevolution methods were robust to changes in the setup of the task, the resolution of the input provided, and the evolution time. This work demonstrates yet again that the neuroevolution methods are more suited to tackle problems in which the user has very little knowledge about the problem *a priori*.

This section discusses many of the important works in the field of neuroevolution. A summary of the key features of these works is included in Table 2.1. These works,

among other things, demonstrate that neuroevolution methods work well on a wide variety of tasks. We note that biological systems are "trained" in an evolutionary way, which is one of the main reasons we chose an evolutionary algorithm as the training algorithm for our system. There are several other reasons, including:

- No prior knowledge about the structure of the network is needed for each problem. The structure of the network is known to have a major impact on the efficacy of the network.
- Neuroevolution methods are less likely to become stuck in local optima than gradient-based algorithms. They also require less information than gradient-based algorithms (e.g., they do not require error information).
- Neuroevolution methods have been shown to create networks that generalize better than networks trained with other algorithms. Biological networks also have this characteristic.

The works in this section provide significant evidence for each of these points. Several other neuroevolution methods will be discussed in the following sections, as they incorporate not only neuroevolution, but at least another one of the five ideas on which this work is based into their algorithms.

Our work is fundamentally different from the works presented in this section in that these works operate on traditional artificial neural networks. Our design method based on evolutionary optimization was developed to work with NIDA networks, which fall into the category of spiking neural networks.

2.2 Embedding in a Geometric Space

The brain exists in the traditional three-dimensional space. Neurons are separated by some distance, and the distances between neurons have an effect on the way connections between these neurons operate. Neurons in the brain can have input

Table 2.1: This table summarizes the works in Section 2.1. In particular, it notes whether the work allows for structural changes as part of the evolution, whether crossover is used during the evolutionary optimization, and whether a local search method is used to discover weights. Cells with an X indicate that that method has been used in the corresponding work.

Author(s) or Method	Structural Changes	Crossover	Nested Local Search Methods
Montana and Davis [174]		X	
Fogel *et al.* [81]			
Beer and Gallagher [25]		X	
Wieland [266]		X	
Igel [117]		X	
Dominic *et al.* [67]		X	
CoSyNE [102, 103]		X	
Whitley *et al.* [265]	X	X	
Koza and Rice [139]	X	X	
Dasgupta and McGregor [64]	X	X	
GANNet [264]	X	X	X
ANNA ELEONORA [157]	X	X	X
GNARL [11]	X	X	
Tang *et al.* [250]	X	X	
PBLA [146]	X		X
SANE [177, 176]	X	X	
ESP [101]	X	X	
EPNet [273, 276]	X		X
Pujol and Poli [200]	X	X	
Abbass [1]	X	X	X
NEAT [245, 246]	X	X	
Leung *et al.* [143]	X	X	
Alba and Chicano [5]	X	X	X
Fieldsend and Singh [75]	X	X	
Garcia-Pedrajas *et al.* [92]	X	X	X
MGNN [186]	X		
EANT [132]	X	X	
EANT2 [236, 237]	X	X	X
HyperNEAT [244]	X	X	

connections, output connections, or both to thousands of other neurons in the brain. Most of a neuron's connections are with neurons that are spatially close [242].

There is evidence that the geometric organization of neurons and connections in certain parts of the brain is meant to reflect the geometric organization of the physical world. For example, it was found that in the visual system, neurons that process physical information received from adjacent photoreceptors are adjacent in the brain [54]. These types of mappings also exist for the auditory system and the body surface [125]. The visual cortex, for example, has a surface that is organized as a map of the visual field and is composed of layers that do various signal processing steps [127]. This organization of the visual system has inspired artificial visual processing systems, including Arathorn's map-seeking circuits [12].

In certain biological neural networks, it is clear that the geometric relationship between inputs and the neurons that process those inputs is important. As noted in [244] and [95], in most artificial neural network literature, the geometry of the network is not considered. This leaves the network to learn not only the problem at hand, but also the geometric relationship between the inputs. This complicates the learning process because it adds an extra dimension to the problem. In addition, there is no concept of locality in the network. Connections are not seen as long or short, and there can be no concept of topographic regularity because there is no concept of distance at all.

Some convolutional neural networks, especially those that are doing image processing or image recognition, impose a topographic structure on the layers of the network so that each neuron in the layer corresponds to a two-dimensional point in the image [28].

Stanley *et al.* recognized the lack of geometric relationships as an issue in neural network literature in [244]. In **HyperNEAT**, discussed in section 2.1, the algorithm constructs a network in a cube, allowing geometric relationships between inputs and outputs to be recognized in the formation of the network. In fact, the connectivity of the network is determined as a function of the geometry. The authors demonstrate

the utility of neural networks in geometric space in a visual discrimination task. In the visual discrimination task, HyperNEAT was compared against a technique that does not embed its neural network into a geometric space. HyperNEAT outperformed this method on this task because of its inherent ability to learn the geometric concept underlying the task.

The work with HyperNEAT is continued by Gauci and Stanley in [95]. In this paper, the authors compare HyperNEAT with methods that are completely unaware of geometry. The task they use for comparison is playing checkers. Checkers, like many board games, has rules that are inherently geometric. They were able to show that HyperNEAT and a version of NEAT that was aware of geometry outperformed similar methods that were unaware of geometry on this task. They were also able to find solutions more quickly than the other methods, likely because they did not have the extra task of learning the geometric relationship between inputs.

Hawkins and George introduce hierarchical temporal memory (HTM) in [110]. Hierarchical temporal memory is a neural network structure that is organized into regions that are made of columns (inspired by cortical columns in the neocortex). They make use of a sparse distributed representation in which only a small subset of neurons are active at any given time in the network. Geometric location of cells in regions is important as cells are connected to their neighbors within the region. Synapses in HTM have binary weights. They use Hebbian-like learning mechanisms to train the weights; this learning mechanism allows for on-line learning. They have used HTM for anomaly detection and prediction tasks.

These works demonstrate that imposing geometric relationships on neural networks can have a significant impact on the ability of the network to learn certain tasks. We will also argue that the geometric space component allows for easier representation of the networks for the evolutionary algorithm.

2.3 Dynamic Networks

The biological brain is a dynamic system. It operates on a continuous time scale. In the biological brain, there is propagation delay between one neuron firing and another neuron receiving that pulse. These delays implement a form of short-term memory, allowing a comparison of the events in the recent past to current inputs. Events in the brain, such as the firing of an neuron, occur at specific times, and any event that is created by an event (such as other neurons receiving charge as a result of a firing event) occurs at a later time. This delay, though sometimes very small, nonetheless affects the operation of the brain. In fact, Hebb notes that time factors have a major influence on the behavior of the brain [113]. This has been further explored in many works, including the review article by Markram [160], which explores the history of spike-timing plasticity. This section covers two types of dynamic neural network architectures: continuous time recurrent neural networks and spiking neural networks.

2.3.1 Continuous Time Neural Networks

Recurrent neural networks include the notion of time through delay elements. They allow networks to store past information received and to use that information for later cycles. These recurrent connections are necessary in order for a neural network to solve any problem where past behavior has an impact on the way the network should behave in the present. As discussed in Appendix B, there are two types of recurrent neural networks: discrete-time and continuous-time recurrent neural networks. Continuous-time recurrent neural networks (CTRNNs) more accurately model the dynamical behavior of biological neural networks, and thus, are the chosen type for this work. However, CTRNNs typically use differential equations to determine behavior, rather than discrete-event systems, which is the chosen method for this work.

Funahashi and Nakamura proved that continuous time recurrent neural networks can approximate any finite time trajectory of a given n-dimensional dynamical system [84]; thus, any continuous curve can be approximated by the output of a CTRNN. This is a major theoretical result in the field, justifying the use of CTRNNs for these types of problems.

Beer discussed in detail the dynamics of one and two neuron CTRNNs [22]. He showed that even these small networks are capable of complex behaviors. In this work, he also suggests composing these small networks into larger networks, using evolutionary algorithms, in order to produce even more complex behavior, an idea that is echoed in our section on modularity (Section 2.4). Beer also noted the importance of treating a learning agent such as a neural network as a dynamical system in [23]. He describes an evolutionary algorithm approach to developing CTRNNs in this work. He successfully applied this algorithm to models of chemotaxis, walking, and sequential decision-making and learning.

Lu noted that CTRNNs (using differential equations) with propagation delays do not always have stable solutions. He noted three conditions on the weight matrices that are required in order for stability to occur [147]. There has also been significant work in the field of stability of dynamic systems, some of which will also apply to spiking neural networks [68, 148].

Gallagher and Vigraham, like Beer, used a genetic algorithm to train a CTRNN. They successfully applied their algorithm to a single-legged walker controller. Moreover, they outlined a hardware implementation of their algorithm, using a hardware genetic algorithm engine on the same chip as reconfigurable VLSI neurons [87].

Beer continued his work on CTRNNs with [24]. In this work, he explored the connections between the agent and the environment's dynamical systems. He applied an evolved CTRNN to a labeling and discrimination task, in which the agent had to identify an object as a circle or a diamond and either catch or avoid the object, depending on the label. By studying the performance of this task on several levels

(the entire agent-environment system, the agent-environment system that defines the interactions, and the agent system alone), he was able to show how the CTRNN had learned to complete this task.

Continuous-time systems with propagation delay are also being explored as an important component in developing brain-like systems. The goal of the Blue Brain project [159] is to simulate mammalian brains. They began by simulating a network of 10,000 neurons meant to represent one cortical column in a rat. They have used their simulations to explore a variety of properties about the brain, including the response properties of neurons [137], synaptic organization [195], and single-cell dynamics [111].

In this work, we use a discrete-event system to represent a CTRNN with propagation delay. We include a notion of refractory period and boundedness of charge in our network in order to guarantee stability of the CTRNN (though not necessarily the closed-loop system of which it may be a part), regardless of network structures. This allows us to avoid the restrictions placed on network structure given in [147]. Our neurons, however, are significantly simpler than most used in CTRNNs. Thus, our networks may require more neurons than CTRNNs to solve similar problems.

2.3.2 Spiking Neural Networks

Spiking neural networks also include the notion of time in neural networks. In particular, in spiking neural networks, neurons fire when a threshold or membrane potential is reached (based on an "integrate-and-fire" model), rather than on a clocked or cycled system (as in traditional neural networks) [149]. NIDA networks fit loosely within this definition of neural networks as well, as neurons accumulate charge and fire when a threshold is exceeded. However, the neuron model used in the NIDA network definition is much simpler than the "integrate-and-fire" model in typical spiking neural networks (see Section 3.1 for how NIDA neurons work).

Ghosh-Dastidar and Adeli note that two research areas have emerged around spiking neural networks: (1) studying spiking neural networks as artificial neural networks and computational devices, which we call "computation-driven" research and (2) using spiking neural networks to better understand neuroscience topics (within the field of computational neuroscience), which we call "biology-driven" research [98]. Our work primarily deals with the former topic, as we are not attempting to explicitly model any biological processes; however, insights gained from our results may seed research topics within computational neuroscience.

Spiking neural networks have been trained using a variety of methods, including unsupervised techniques [150, 151, 182, 271] and gradient-based techniques [41, 42, 97, 162, 194, 272]. Kasabov notes the usefulness of spiking neural networks in processing spatiotemporal information in [129] and gives an overview of various spiking neural network methods, including single neuronal models, evolving spiking neural networks, and computational neuro-genetic models. Since our work primarily deals with evolutionary optimization techniques, in this chapter we focus on works that make use of these techniques as the design method.

Floreano *et al.* describe and implement an evolutionary algorithm to design spiking neural networks for vision-directed robot control tasks. In these works, the neuron model is a spike response model (a generalized integrate and fire model). In [79], a preliminary genetic algorithm is described, and this model is expanded on and implemented on board robots in [78].

Hagras *et al.* note that spiking neural networks are more computationally powerful, noise robust, and have simpler interfaces than traditional neural networks [105]. They make use of an adaptive genetic algorithm, which adapts the crossover and mutation rates over the course of learning, a method that we are also considering. They implement a refractory period on neurons by drastically lowering the charge on the neuron and slowly returning the charge to a neutral state, and they make use of delay to encode input information from the environment. They demonstrate that

the adaptive genetic algorithm produces better results for their network model for a mobile robot control problem.

Pavlidis *et al.* note the importance of timing in firing in biological brains to motivate the use of spiking neural networks over traditional artificial neural networks [193]. They propose an evolutionary algorithm design method that makes use of differential evolution with migration. In this method, subpopulations are evolved and the best members of subpopulations are "migrated" to other subpopulations. They show that this method produces comparable results to traditional artificial neural networks on exclusive-or, the Pima Indian diabetes classification problem and the iris classification problem.

Belatreche *et al.* use an evolutionary strategy to design spiking neural networks with dynamic synapses [26]. The dynamic synapses in their spiking neural networks do not have associated weights; instead, the magnitude of the effect on the post-synaptic neuron is determined by the number of incoming spikes. They successfully apply their evolutionary strategy and spiking neural network model to exclusive-or, the iris classification problem, and the Pima Indian diabetes classification problem.

Jin *et al.* use Pareto (or multi-objective) optimization to design spiking neural networks with feed-forward architectures [123]. They use error information for classification problems as well as the number of connections and total length of synaptic connections as optimization criteria for the evolutionary strategy. They make use of crowded tournament selection to maintain diversity in the population of networks. They demonstrate that their spiking neural network model and design method produce comparable or better results than traditional neural networks in some cases on the breast cancer and Pima Indian diabetes classification problems.

Schliebs *et al.* introduce a quantum-inspired spiking neural network, in which evolvable spiking neural networks (introduced in [270, 271]) are optimized with a quantum-inspired evolutionary algorithm, which is an estimation of a distribution algorithm [223]. The neural model used gives more weight to early spikes than later spikes. They apply their network and optimization algorithm to the two-spiral

problem and the uniform hypercube dataset problem. They note that their method performs well when compared with classical methods.

Russell *et al.* use a genetic algorithm to develop spiking neural networks to form central pattern generators (CPGs) [217]. They make use of linear integrate-and-fire neurons and use a maximum likelihood technique to help determine parameters of the neurons. They also validate their maximum likelihood technique by attempting to replicate real neural data. They implement their genetic algorithm such that the fitness function can be calculated in either simulation or *in silico*. The genetic algorithm uses both behavioral memory (which uses two fitness functions) and staged evolution (breaking the problem into stages that are solved sequentially) in order to improve convergence. They successfully evolve CPGs that can generate firing patterns to mimic walking frequencies.

Battllori *et al.* use a genetic algorithm to adjust weights and delays of a spiking neural network trained to control a robot [21]. They used imitation learning, or learning the robot's behavior based on the behavior of a heuristic rule-based controller. The authors manually selected the structure of the neural network and went through several tests in which they altered the structure to attempt to find a network with the best performance. They note that they did not obtain their desired result, but they attribute this to the small number of neurons and connections used. It is clear from their description that they would have benefited from a training method that discovered structure as well as parameters. They also noted that frequency encoding of input (stronger stimuli produce more spikes than weaker stimuli) was superior to delay encoding (stronger stimuli generate pulses later than weaker stimuli).

Capuozzo and Livingston introduce a compact evolutionary algorithm for spiking neural networks that is meant to be used in a micro-controller [50]. They use a very simple neuron model similar to the model we use, but their neurons do not accept input when they are in their refractory periods. The evolutionary algorithm they propose optimizes over weights, thresholds, biases, and refractory periods, but they do not account for structure in their evolution. They do note, however, that it

would be useful to include structural evolution as part of their optimization method. Capuozzo and Livingston use their method to evolve spiking neural networks that learn how to count sequentially and to generate specific frequencies of pulses.

O'Halloran *et al.* made use of spiking neural networks trained with a genetic algorithm for breast cancer classification based on tumor information [184]. They used frequency encoding to encode information about a tumor as input to a single hidden layer neural network. They trained over only the thresholds and weights of the networks. They tested both a direct classifier that takes information about the tumor and classifies it as either benign or malignant, and a two-stage classifier in which the first classifier decides if a tumor is large or small before passing on the information to the second classifier, which decides if it is benign or malignant. They showed that the spiking neural network classifiers trained with genetic algorithms outperformed linear discriminant analysis classifiers in terms of both accuracy and robustness.

Rossant *et al.* use CMA-ES to discover parameters for a single neuron model in order to develop a model that will produce a particular spike train [215]. They note that this problem requires a global optimization method that does not require gradient information, which is why they have chosen genetic algorithms. They make use of graphics processing units (GPUs) in order to parallelize their simulation of the neurons and speed up parameter selection.

Carlson *et al.* use a spiking neural network trained with spike-timing-dependent plasticity (STDP) and evolutionary algorithms to encode graded inputs and to develop a self-organizing receptor field [53]. In particular, they examine synaptic scaling with STDP in order to avoid runaway synaptic dynamics. They use the evolutionary algorithm to tune 14 parameters, including synaptic weights, homeostatic parameters (used in synaptic scaling), STDP parameters, and the maximum firing rate of the network. They showed that synaptic scaling stabilized the firing rate and synaptic weights in a spiking neural network.

Baladhandapani and Nachimuthu use spiking neural networks trained with genetic algorithms for brain tumor classification in [18]. In particular, they apply their neural

networks to 3D MRIs of brain tissue and brain tumors. They analyze output in terms of firing rate, where different firing rates correspond to different classifications. Baladhandapani and Nachimuthu compare their spiking neural networks with feed-forward neural networks trained with back propagation and demonstrate that the spiking neural networks outperform the traditional neural networks on this task.

Kasabov *et al.* develop a personalized modeling system based on spiking neural networks to predict stroke occurrence in [130]. They make use of a spiking neural network in three-dimensional space, where neurons that are closer together are more likely to be connected than those that are far apart. Their system parameters are trained with a genetic algorithm, and the learning is ongoing as new data is presented. They allow for small structural changes, such as additions of output neurons or combining of output neurons. They note that a similar method can be applied to other prediction problems, including machine failure prediction, financial market crash prediction, and earthquake prediction.

Asher *et al.* use a neural network inspired by the primate brain to perform a visually guided reaching task in [14]. They train the parameters of their neural network (the weights of the connections and the parameters governing the dynamics of the network) with an evolutionary algorithm that uses crossover, mutations, and roulette wheel selection. Their architecture makes use of complex lateral and feedback connections in order to process temporal sensory information. They determined that many different parameter settings could produce similar behavior in networks with the same general architecture.

Carlson *et al.* present a method for using GPUs to train spiking neural networks with evolutionary algorithms [51, 52]. In particular, they make use of the GPUs to simulate a population of networks in parallel and use the evolutionary algorithm to tune the parameters of the spiking neural networks. They use their method to evolve spiking neural networks that act as self-organized receptive fields, similar to those found in biological brains. They use an indirect encoding scheme to represent the parameters that are being tuned, which include the Hebbian learning, homeostasis

(or weight stabilization), maximum input firing rate, and synaptic weight parameters. The networks utilize STDP as part of training.

This section presents work on dynamic network structures. Our NIDA networks are also dynamic architectures and are most closely related to spiking neural network architectures. One major difference between our work and most of the work presented in this section is the ability of our network architecture to change as needed over the course of training. Though this adds another level of complexity to our search space, it also allows for more potential solutions and avoids the need for prior assumptions about how much structure is required to solve a particular problem. Instead, the evolutionary optimization determines the structure needed.

2.4 Recognition and Reuse of Useful Structures

In the context of the present work, useful structures can be referred to as modules.

Modularity is prevalent in natural organisms. As pointed out by Raff in [203] and Riedl in [212], modularity is key to biological order. Raff characterizes modules as having several properties:

- They have autonomous, discrete organization.
- They contain hierarchical units and may be part of larger hierarchical entities.
- They have physical locations within the developing system.
- They exhibit varying degrees of connectivity to other modules.
- They can undergo transformations over time.

Raff points out that biological modules undergo duplication and divergence. Through the course of duplication, mutation can occur and results in a divergence of a single module type. The result is two module types which have similar but ultimately

different functions [203]. Modularity as a mechanism for evolving complex biological organisms is also discussed in [108, 131, 190, 261, 262].

In [49], Caellli *et al.* note that the "modularity paradigm" is used to explain the processes by which the brain accomplishes a variety of tasks, including vision, speech, motor control, and memory. Vision, for example, has been broken down into subtasks, such as feature encoding, motion encoding, and feature processing, and for each of these subtasks, there is a dedicated network architecture. There are also modules at an even lower level in the visual system that process information in parallel across the visual field. Another type of proposed module in the brain is the internal model of outside systems that can be used for prediction. For example, there is evidence that the cerebellum contains internal models of motor apparatus [133, 269].

In [243], Stanley notes some of the patterns in organisms:

- Repetition: Multiple instances of the same substructure.
- Repetition with variation: Multiple instances of similar substructures.
- Symmetry: Repetition with respect to some axis; for example, bilateral symmetry of the body. The overall structure of the brain is divided symmetrically into two hemispheres.
- Imperfect symmetry: Overall symmetry is maintained, but details are different. Again, the brain is a good example. The brain is divided into hemispheres, but the hemispheres have different structures and functions.
- Elaborated regularity: Regularities from earlier generations are built upon and repurposed.
- Preservation of regularity: Overall structure is mostly preserved across generations, with small details varying between parents and children in most cases. A large change in structure is relatively rare compared to the frequency of changes within structures. For example, four-limbed animals rarely produce offspring with a different number of limbs.

It is obvious the reuse of a structure is an extremely prevalent characteristic of biological systems, including the brain. However, there is another important note about the reuse of useful structures. Certain substructures evolved for one particular task in the brain have been shown to be useful for other tasks. For example, in [199], Ptito *et al.* show that, in hamsters, the auditory processing system, when rewired to receive visual input, can successfully process the visual input. That is, the substructure that processes auditory input is also a useful structure for processing visual input. It has also been shown that local neural network structures are reused for different functions. In [10], Anderson and Penner-Wilger show that a local neural circuit is involved in finger awareness, number representation, and other diverse functions. A substructure that is useful for one task may be very similar to a structure that is useful for a totally different task, making it worthwhile to re-use useful substructures in a variety of different contexts. The difficulty lies, however, in determining which substructures in a network are useful.

Modular neural networks are inspired by the idea that the neurons in the brain are sparsely connected, and their connections are typically clustered. Connectivity occurs in a hierarchical pattern. In [49], Caelli *et al.* give an overview of the work done in modular neural networks. These works include applications of modular neural networks to pattern recognition theory and application tasks, image processing, modeling, control, language and speech processing, target detection and recognition, and time series analysis.

In [56], Cho and Shimohara describe an evolutionary algorithm that evolves modular neural networks. Their modules are designed to model neocortical mini-columns. The internal structure of their modules are fixed, and the weights of all of the connections within a module are fixed. Intermodule connections are learned using a Hebb rule. The parameters of the Hebb learning rule, as well as the structure of the connections between modules are learned using a genetic algorithm. Cho continued his work with modular neural networks in [55] and [57]. In these works, the authors apply the modular neural networks to a classification task. The n best modular neural

networks for the classification task are chosen, and the final classification is based on a combination of the outputs of the neural networks.

In [90], Garcia-Pedrajas *et al.* use cooperative coevolution to evolve co-adapated subcomponents. In cooperative coevolution, several species are evolved together, and cooperation among the individuals of the population is encouraged by rewarding the individuals based on how well they cooperate to solve a particular problem. The authors introduce a cooperative coevolutionary model called COVNET. In COVNET, a modular neural network is evolved. Modules must learn how to combine with other modules in order to be useful for the problem at hand. COVNET also develops a measure of fitness for the modules. In their framework, there is no need for any prior knowledge of how to decompose the problem. They call their modules "nodules," and the nodules are made up of nodes and connections. Networks are made by combining nodules. A restriction they place on their population of networks is that every network must have the same number of nodules. The nodule network is evolved using algorithms similar to **GNARL** [11] and **EPNet** [273]. They also evolve their networks using genetic algorithms that combine the nodules. Their population of networks is designed to evolve more slowly than the population of nodules. Garcia-Pedrajas *et al.* continue their work in [91], in which they use their nodule network algorithm to evolve ensembles of networks, rather than just a single network.

In [211], Reisinger *et al.* introduce **Modular NEAT** (based on **NEAT**). They note that evolution could proceed more efficiently if symmetries inherent in the domain could be recognized, so that the algorithm could spend less time solving the same sub-problem more than once. Their algorithm reuses neural substructures to encourage modularization and regularity. This algorithm performs better than **NEAT** on the same problem, and it generates modular solution networks automatically, without outside insight about symmetries in the task. In [247], Stanley, points out that **Modular NEAT** has the pitfall that a module can appear anywhere in the network, causing a major change to the function of the network that can radically alter or destroy functionality of that network.

Hornby notes that modularity, regularity and hierarchical characteristics must be exhibited in order for design systems to scale well [116]. Hornby defines modularity in much the same way as above: an encapsulated group of elements that can be manipulated as a unit. Regularity is a repetition or similarity in design (such as repetition, repetition with variation, symmetry and symmetry with variation discussed in [243]). Hierarchy is the number of layers of encapsulated modules in the structure of a design. Hornby defines a metric on each of these characteristics: modularity, regularity and hierarchy. He uses these metrics to demonstrate that networks evolved with an algorithm that supports all three generally perform better on a design task than algorithms that do not support any or all of the three characteristics.

Similar to Hornby, Lipson notes in [145] that the scalability of evolutionary processes depends heavily on their ability to make use of functional modularity, structural regularity, and hierarchy. He also notes that the notions of modularity and regularity are distinct. A system can be modular without exhibiting regularity; for example, a system can be made up entirely of distinct modules, each with their own purpose. Similarly, a system can distribute regularity without modularity; in fact, regularity is also demonstrated in smooth systems. Finally, Lipson notes that increases in modularity can result in decreased performance for a particular system, but that modularity can help in the long run. In the context of the present work, this means that, for a particular problem, the modular neural network may not be the most efficient representation or network for the problem. There may be a more compact solution. However, the modular network is likely more easily evolved than the more compact solution, resulting in a faster learning time.

Jung and Reggia [124] note that in some cases, there is prior knowledge about a specific problem and the required architectures that can significantly reduce the neural network search space. They employ a hierarchical high-level descriptive language for initial architectures, identifying the pieces of the architecture that should be evolved, thus reducing the space of architectures to search. This initial description allows one

to describe a group of neural networks of interest at a high level, in terms of possible modules and intermodule connections. This notion of exploitation of prior knowledge is one that we are considering exploring in our future work.

Mabu *et al.* use a graph-based evolutionary algorithm called Genetic Network Programming (**GNP**) [152]. The main characteristic of GNP is that it reuses nodes so that the representation of the network can be compact. They also proposed an extended algorithm that trains the network to solve reinforcement learning tasks. They call this algorithm GNP with Reinforcement Learning (GNP-RL), and they applied GNP-RL to a canonical Tileworld reinforcement learning task.

In [126], Kamioka *et al.* propose an evolutionary algorithm in which networks are described in terms of the subnetwork modules. The structure of the networks evolves by adding new subnetworks or connections, by merging two subnetworks as a higher-level subnetwork, or by changing the existing connections. Again, the goal of using modularity in their neuroevolution algorithm is to increase the scalability of the algorithm so that larger and more complex networks can be evolved.

In [275], Yao and Islam discuss a variant of modular neural networks, which is neural network ensembles. Neural network ensembles are analogous to modular neural networks where members of the ensemble correspond to modules. They note that these ensembles combine a set of neural networks which learn to decompose a complex problem into simpler sub-problems and solve the simpler subproblems more efficiently. One of the main issues with evolving ensembles is maintaining diversity in the population so that the members do not all perform the same task.

Mouret and Doncieux note that organisms evolved by co-opting structures adapted for one function to solve new problems, which is a process they call exaptation. In [179], they propose an evolutionary algorithm to exploit exaptation and modularity to evolve complex systems. They successfully apply their algorithm to evolve a network to compute a Boolean function with modular structure.

Durr *et al.* develop a modular encoding scheme for networks (or a modular genetic representation of the network) so that modules of the "genome" of the network are

decoded into modules of the neural network [71]. They use a genetic algorithm to find the optimal network scheme, and because their genetic algorithm allows for creation and deletion of modules of the network, it optimizes over the topology of the network. They note that one advantage of the modular representation is that the genetic algorithm may automatically decompose the architecture required to solve a problem into functionally separate modules. They successfully apply their network representation and algorithm to a control problem and show that their method is more robust and more successful than non-modular methods.

Several concepts discussed in this section relate to features incorporated into this work (see Chapter 4):

- Networks or modules useful for one problem may also be useful for another problem.
- Modularity, regularity, and hierarchical structure can be exploited to speed up convergence.
- Incorporation of modules and hierarchical structure allows for larger networks to be evolved more quickly and produce more robust networks.

2.5 Affective Components

In [63], Damasio asserts, based on the history of research and study of individuals with brain damage to portions of the brain typically associated with emotion, that emotion-related systems are an important part of a human's overall ability to learn, reason, and make decisions. Because these systems are extremely instrumental to the learning process in humans, we speculate that they will also be very useful in the learning process in artificial neural networks.

In comparing ANNs to their biological counterparts, we note that ANNs typically perform the type of work that is accomplished in the cerebral cortex. The cerebral cortex is responsible for the planning and execution of tasks in the human brain; it

has distinct regions that carry out a variety of functions, including visual, auditory and memory-related functions [127]. ANNs, unlike the cerebral cortex, are trained to perform a highly specific, usually well-defined task. However, the role of an ANN can be thought of as analogous to a very small network in the cerebral cortex. In the biological brain, the cerebral cortex receives information from the environment through sensory organs, and it interacts with the environment similarly (for example, by stimulating muscle movements). The operation of the cerebral cortex is also influenced by activity in the lower parts of the brain, such as the basal ganglia and the limbic system.

An ANN, like the cerebral cortex, receives and transmits information to its environment, often through other mechanisms that function like sensory systems. The interaction of the cerebral cortex with the basal ganglia and the limbic system, however, is not typically included in a neural network architecture. Among its many functions, parts of the limbic system are responsible for the affective system. Different parts of the limbic system are associated with different emotions. These affective systems communicate with the cerebral cortex through neurotransmitter pathways. In [188], Panksepp defines several major "emotional" operating systems of the brain. These systems include the SEEKING system, the FEAR system, the PANIC system, and the RAGE system.

The purpose of the SEEKING system is to stimulate curiosity. Evolutionarily, the SEEKING system probably drove animals to become very interested in exploring their worlds in order to find food, water, warmth, and other essentials for survival. Panksepp speculates that this system is efficient at promoting learning, especially in finding effective ways to obtain information.

Panksepp attributes the SEEKING system to the self-stimulation system that goes from the midbrain up to the cortex, which is also called the reward or reinforcement system of the brain. A major neurotransmitter associated with this system is dopamine. Panksepp notes that the dopamine circuits arising from the midbrain nuclei are likely part of the SEEKING system. These circuits tend to energize and

mediate the planning and foresight that occur at higher levels in the brain, as well as to promote eagerness and directed purpose in animals.

Another emotional operating system Panksepp describes as part of the brain is the FEAR system. Panksepp speculates that the FEAR system developed during evolution to encourage the animal to avoid situations that may lead to pain or death. The FEAR system acts partially as a counterpoint to the SEEKING system in that SEEKING encourages exploration in the hopes of finding something good whereas FEAR discourages exploration because of the higher chance of encountering an unknown predator or obstacle that would lead to harm. Panksepp notes that there are portions of the brain that are known to cause fear in animals and humans.

It is important to note that there is significant neural interaction between the FEAR and SEEKING systems (as well as the other emotional operating systems), meaning that these systems have great influence over one another. We speculate that both of these systems have an effect on learning and that it would be beneficial to include aspects of both curiosity and fear into the learning process of general tasks.

The PANIC and RAGE systems, on the other hand, have potential applications in specific learning tasks. In the brain, the PANIC system is responsible for feelings of pain associated with separation distress; for example, the PANIC system is responsible for the feelings of loss accompanying the death of a parent or child and general feelings of loneliness. The PANIC system fosters social behavior by creating feelings of attachment and dependence. We speculate that a PANIC-like mechanism in an artificial system would help to encourage group behavior, for example, in a predator-prey environment in which two or more predators or prey are attempting to cooperate.

The RAGE system, like the FEAR system, works in opposition of the SEEKING system. The RAGE system is responsible for feelings of anger, especially anger in response to a restriction of one's freedom to explore. The purpose of this system in nature is to energize behavior when an animal feels restrained or frustrated, but also to encourage animals to defend themselves against opponents. We speculate that the inclusion of a RAGE system would be useful for some learning applications. RAGE

and PANIC also have significant neural interaction and work in opposition to each other.

Variations of the concept of the SEEKING system have been presented in the machine learning literature, although they are typically framed as reinforcement learning systems in which there is an internal (or intrinsic) reward for the agent exhibiting curiosity in its environment. These types of systems put a new spin on the problem of exploration vs. exploitation.

In [224], Schmidhuber discusses an approach to reinforcement learning that includes a notion of curiosity. In this method, curiosity is implemented by encouraging the improvement of the model's reliability by exploring situations and experiences that are unfamiliar to the model (situations in which the model performs poorly). Schmidhuber implements this idea in the context of Q-learning, described in [248]. In this method, Schmidhuber predicts the error of the model and chooses to pursue situations in which the predicted error between expectation and reality is highest, thus encouraging the agent to explore unknown areas.

Singh, Barto, and Chentanez discuss intrinsic motivation in [20, 239]. They define intrinsic motivation as being moved to do something because it is inherently enjoyable, and they note that curiosity is driven partially by intrinsic motivation. The authors present a reinforcement learning framework using intrinsic motivation learning that allows for the autonomous development of skills. Intrinsic rewards are generated for each event, and the rewards are proportional to the error in the prediction of the event according to the model, which gives the agent incentive to seek out unfamiliar situations. In [20], the skills that are developed using this learning framework are basic skills that can be combined to form more complex behaviors, which is a form of functional modularization. Singh *et al.* discuss intrinsic motivation from an evolutionary perspective. They speculate that the evolutionary process encouraged exploration, play, and discovery because they led to successful reproduction and survival [240].

Ahn and Picard present a new framework for affective-cognitive learning for affective agents to improve human-computer interaction in [4]. The affective agent used in their work is a robotic computer that is meant to interact with a user. The goal of the robotic computer is to make the user attentive or to make the user show pleasure. This robotic computer, in addition to the external rewards from the user, also receives internal rewards to encourage the agent to continue working towards its dual goals. In this context, the internal reward is less concerned about encouraging curiosity and more concerned about improving the general emotional state of the agent, which in their simulation can either be a "feeling good" emotional state or a "feeling bad" emotional state.

Schmidhuber follows up his work in [224] with [225, 226]. In [225], he discusses a reinforcement learning scheme in which curiosity is rewarded. This scheme is applied to developmental robotics, which is a field in which robots attempt to discover, in an unsupervised and experimental way, how their environment works. Again, an intrinsic reward is defined. Schmidhuber calls this reward a curiosity reward, which is given in response to improvements in the agent's predictor. In this case, the agent's predictor is the mechanism by which the agent predicts a future event, given the current environment.

In [226], Schmidhuber gives an overview of many reinforcement learning works that use intrinsic motivation (i.e., an internal reward for curiosity) to aid the learning process. These works, he claims, all contain four vital ingredients:

1. An adaptive world model, which can be thought of as the predictor. This world model is essentially everything the agent currently knows about how the world works.

2. A learning algorithm that improves the world model.

3. Intrinsic rewards that measure improvements made to the world model due to the learning algorithm.

4. A reward optimizer or reinforcement learner, which chooses actions that will optimize future reward.

Simsek and Barto discuss an intrinsic reward mechanism for efficient exploration in [60]. They considered intrinsic rewards only in the context of improving exploration. In typical reinforcement learning methods, there is a tradeoff between exploration and exploitation. That is, an agent explores its environment to find the highest reward, but it also exploits what it currently knows about the environment to obtain a higher reward. Simsek and Barto, rather than considering exploitation, consider only the case in which the agent does not care about obtaining a high reward currently, but only about exploring the environment so that exploitation can be done later. In essence, they explore the instance in which an agent is given a training period. During this training period, it does not matter whether the agent is gaining reward. Then, after the training period, a second period begins, in which the agent must perform well (i.e., receive higher rewards). They apply their two-stage algorithm to a maze task, in which their algorithm outperformed other reinforcement learning methods.

In [128], Kaplan and Oudeyer frame their intrinsically motivated system based on research in psychology. They frame learning into two concepts: assimilation and accommodation. Assimilation is the process by which organisms (or agents) incorporate aspects of their environment into their pre-existing cognitive structures. Accommodation is the process by which organisms adapt their cognitive structure to fit the environment. They define a control architecture that implements a form of artificial curiosity and show how this artificial curiosity affects the trajectories of a robot. They were able to show that the agent's learning process (when learning using the control architecture proposed) can be broken into the two phases of assimilation and accommodation.

Schembri *et al.* also discuss intrinsically motivated reinforcement learning. The internal reward given in their model is to encourage the acquisition of general-purpose building-block behaviors that can be combined to solve specific tasks, again similar

to the modularity discussed in Section 2.4. The model in their work overcomes two problems present in previous work on reinforcement learning with intrinsic motivations: the previous work was only applicable to problems with abstract states and actions, and the method was limited to hardwired salient events to form and train skills, limiting agents' autonomy. The authors overcome these problems by using evolutionary techniques to allow the system to autonomously discover salient events. They use neural networks to cope with continuous states and noisy environments [219].

Takeuchi, Shouno, and Tsujino develop a reinforcement learning method that uses modular neural networks with intrinsic rewards in [249]. The modular system is able to decompose the observed state space and allows for stability in the calculation of the intrinsic rewards. They continue their work in [252], in which they propose two basal ganglia models for autonomous behavior: one is a basal ganglia system model, which allows for more rapid learning of the internal state model, and the other is the basal ganglia spiking neuron model, which has the capacity for probabilistic selection of action.

As for the FEAR, PANIC, and RAGE systems, the works on these topics usually concern human-computer interaction, human-robot interaction, and autonomous robots or deal with neuromodulatory systems (discussed below). It is worth noting that intrinsic punishment (implemented similarly to intrinsic rewards, but modeling FEAR, RAGE or PANIC) may be worth exploring in reinforcement learning systems.

In [210], Reilly and Bates built agents that react to a user based on one of several emotions. Those emotions included joy, distress, hope, fear, pride, shame, admiration, reproach, love, hate, gratification, gratitude, remorse, and anger.

Michaud *et al.* discuss artificial emotions in groups of autonomous robots in [166]. The artificial emotions that they associate with their robots are joy, sadness, anger and fear. These emotions are associated with the energy level of the robots. The different emotions help govern the behavior of the group of robots. For instance, the

fear level of a robot can be used to determine which robot in a group waiting to recharge should be allowed to recharge first.

Banik *et al.* also apply affective ideas to multi-robot teamwork, especially task allocation, cooperation, and interaction among the members of a multi-robot team. In their work, emotions play a role in adaptation (changes in behavior), action selection, managing social regulation, sensory integration, and motivation and learning [19].

In [120], Izumi *et al.* also use joy, anger, fear, and sadness to aid in behavior generation in robots. Behaviors are then generated based both on the environment and on the current emotional state of the robot.

Malfaz *et al.* describe a biologically inspired architecture for a robot that is meant to be autonomous and social [154]. Their goal is to design robots that can live with humans, not just with other robots, and they speculate that adding emotional dynamics is an important step in designing these sorts of robots. They argue that robots require emotions for several reasons. One of these reasons is that emotions provide opportunities for learning. They decided to implement three emotions (happiness, sadness, and fear), but they acknowledged that it is currently unknown how many emotions are required in order for complex emotional behavior to emerge.

A third more recent collection of work related to affective systems concerns neuromodulation controllers, proposed by Krichmar in [140]. Krichmar proposes using neuromodulatory systems (systems that use neurotransmitters to modulate activity in biological neural networks) to act as behavioral controllers in artificial systems. He notes that these systems are important for not only emotion, but also goal-directed behavior, attention, and decision-making. Krichmar notes that these systems are subcortical, are reciprocally connected with both the frontal cortex and parts of the limbic system, have associated chemical transmitters, and have similar effects. He claims that the major difference between the different systems, such as fear or rage, are the stimuli that provoke their response. He proposes a network model that uses neuromodulatory systems to make decisions about whether to exploit actions that are

known to be successful or to explore new actions. He notes that an autonomous robot exploring an unknown environment could benefit from neuromodulatory systems.

Cox and Krichmar implement neuromodulatory systems for a neural network that controls a robot in [59]. In particular, they implemented dopamine for "wanting" behavior, serotonin for risk avoidance, and acetycholine for attention. They trained the robots in an environment where there were "find" stimuli and "flee" stimuli, and they were able to elicit the behavior from the neuromodulation systems that they desired.

In [62], Daly *et al.* describe a neuromodulatory system for a developmental network, a structure based on finite automata in which temporal information is stored. They add reward and punishment functions that are based on serotonin and dopamine to their developmental network system. They connect their reward/punishment systems directly to the motor neurons in their network. Their networks are divided so that three different subnetworks receive sensory information: one corresponding to a processing/computing area, one corresponding to an area that will generate dopamine, and one corresponding to an area that will generate serotonin. They initialize their network with "feelings" of loneliness or danger. These feelings govern how the agent responds to and interacts with the environment. In their experiments, an agent explores an environment where there is an attractor and a repulser. The agent receives dopamine (reward) when it goes near an attractor and receives serotonin (punishment) when approaching the repulser. It uses these mechanisms to train the network. A similar method is used by Paslaski *et al.* in [191] for a visual recognition task.

Krichmar describes an agent with a neurmodulatory system for autonomous robots in [141]. In his simulations, there are four possible states for the agent and four possible actions (corresponding to four inputs and four outputs). He includes a dopaminergic system for rewards, a serotonergic system for punishments (because serotonin is implicated in the control of stress and risk-behavior), and cholinergic and noradrenergic systems for attention and uncertainty. Neurons with sigmoid

activation functions are used to represent the various systems, with one neuron for the dopaminergic system, one for the serotonergic system and four for the cholinergic/noradrenergic systems. Short-term plasticity was used to update weights in the network based on an update rule. They found that the agent was more likely to explore when the dopaminergic system was active, more likely to select harm aversive behaviors when the serotonergic system was active, and more likely to display selective attention and filtering of unimportant information when the cholinergic/noradrenergic systems were active, all of which are consistent with the behaviors they hoped to elicit. However, the networks were selectively connected so that the neurons corresponding to these systems were only connected to actions that corresponded to their desired response. For example, the dopaminergic system was not connected to the "find home" action, which would be a harm aversive action.

Soltoggio and Stanley introduce a Hebbian-type plasticity (in fact, anti-Hebbian learning) to update synapse weights based on neuromodulatory systems in [241]. They utilize both noise and saturation in their learning process and show that operant reward learning emerges from their simple learning rule. They allow weights to saturate, producing highly excitatory or inhibitory responses, a property that is deemed undesirable by other types of Hebbian learning. They exploit this saturation and neural noise to build their bottom-up operant learning system. In their networks, there are two synapse connections between two neurons, one inhibitory and one excitatory. The learning rule updates the difference in these weights, either making them further apart or closer together. They observe the effect of their learning system on Braitenberg vehicles. They note that, in this experiment, learning takes place in stages and adapts to changes in the environment. They successfully apply their system to the n-armed bandit problem and again show that the system can adapt to changes in the environment.

Unlike previous sections, in which direct inspiration was taken from the works listed, our proposed work deviates from the ideas presented here. In particular, we note that these emotional subsystems (like SEEKING and FEAR) were evolved over

time; that is, these systems emerged as useful and were propagated across generations. It is only recently that we have given these systems specific names and attributed to them the emotions that we feel. However, it is evident from biology that these systems are useful for learning. It is also evident that a SEEKING type system in reinforcement learning problems improves performance on certain task.

Rather than attempt to replicate the behavior of these systems, we propose instead that one or more affective subsystems evolve along with a network structure. In this way, the affective system required for the problem can be developed, rather than attempt to decide *a priori* which type of system would be most useful for a task and try to model that system explicitly based on known biological properties, as is done in many of the works on neuromodulatory systems.

2.6 Neuromorphic Hardware

We introduce a hardware implementation of NIDA networks in this work. This hardware implementation fits in the realm of neuromorphic hardware, or hardware that is based on or inspired by neuroscience. There are several major research efforts in the field of neuromorphic hardware. This section presents a few of those efforts. In general, there are two major goals/approaches to neuromorphic hardware. The first approach is to emulate biological brains as closely as possible. This approach often has the goal of learning more about neuroscience and the function of the brain (biology-driven). The second approach is to take inspiration from biology but to attempt to develop a new computing architecture to further the field of computing (computation-driven).

IBM Cognitive Computing Project

One of the major efforts torward neuromorphic computing is IBM's cognitive computing project, which has been sponsored by DARPA's Systems of Neuromorphic

Adaptive Plastic Scalable Electronics (SyNAPSE) program. This project encompasses both biology-driven and computation-driven projects.

One of the first steps in a neuromorphic computing program is the development of a simulation. Ananthanarayanan *et al.* demonstrate the ability to simulate 1.6 billion neurons and 8.87 trillion synapses with thalamocortical connectivity and 900 million neurons and 9 trillion synapses with probabilistic connectivity [7]. Their simulations, which take place on a supercomputer with nearly 150,000 CPU cores and 144 TBs of total memory, include spiking neurons, synapses with spike-time dependent plasticity, and delays in the axons. Their neurons are simulated in a clock-driven simulation, but the simulation of the synapses is event-driven. They show that the propagation patterns of spikes in the network agree with observations made in animals, though they note that is impossible to directly validate the performance of the network with respect to biological systems. This aspect of the IBM project seems to be primarily biology-driven.

Merolla *et al.* present a digital neurosynaptic core with crossbar memory in [163] as part of IBM's cognitive computing project. A crossbar memory is used to minimize communication costs, and an asynchronous, event-driven simulation is used to keep power costs low. They are able to demonstrate that these neurosynaptic cores (composed of leaky integrate-and-fire neurons) can be trained offline to implement a restricted Boltzmann machine.

Seo *et al.* develop a digital circuit to implement a network of spiking neurons with a learning algorithm to allow on-chip learning as part of IBM's cognitive computing project [233]. They design a neuromorphic chip using a processor-in-memory approach for more efficient communication between neurons (processors) and synapses (memory). They lower power consumption in their chip by using near-threshold circuits and an event-driven simulation design. They implement full connectivity in their neurons. Their neurons are integrate-and-fire models, and they include inherent learning rules. They demonstrate a spike-time dependent learning

process for learning patterns and a Hopfield-like network in this architecture. This approach seems to be primarily computation-driven.

Preissl *et al.* develop yet another simulator of neural architectures (Compass) for IBM's cognitive computing project in [198]. Their simulation is capable of simulating up to 65 billion neurons and 16 trillion synapses (split into neurosynaptic cores). They use a macaque wiring diagram to design connectivity between the neurons and cores. Their cores have both computational units (neurons) and memory (synapses) and these elements are brought into "extreme proximity," in contrast with the traditional von Neumann architecture. These simulations are used to show that the architecture is capable of multiple applications, including image and audio classification, attention mechanisms, and spatio-temporal feature extraction.

Jackson *et al.* develop nanoscale electronic synapses for neuromorphic chips in [121]. They demonstrate programming schemes that can reproduce biological spike-time dependent plasticity (STDP) dynamics in 40nm and 10nm elements. They include simulations of leaky integrate-and-fire neurons that have straightforward implementations in hardware. They note that though their devices seem to emulate biological spike-time dependent plasticity in biological brains, it is not currently clear what role STDP plays in cognition. They note that these synapses may be useful for either biology-driven or computation-driven goals.

Rajendran *et al.* continue their work in nanoscale neuromorphic devices in [204]. In this work, synapses are implemented using memristive random access memory devices, while neurons are implemented in traditional CMOS circuits. They compare this implementation to a strict CMOS implementation and note that the analog circuits can be made much smaller than the digital circuits, but they consume roughly twice as much power than the digital circuits.

The most recent advance in IBM's cognitive computing project is the TrueNorth digital chip [164]. This architecture uses a core as a basic building block. Each core has 256 input lines (axons) and 256 outputs (neurons) connected by 256-by-256 directed, programmable synaptic connections. Neurons in each core can connect to

axons in any other core (in a grid of cores). They claim that their architecture is scalable because cores on a chip and chips themselves can be tiled in two dimensions, spike events address a pool of neurons on a target core rather than single neuron to neuron communication, and defects on the core or chip level do not significantly affect performance (so it is robust to errors). They also claim that their architecture is flexible, because their neurons allow for configurability and each neuron-synapse is also programmable. They note that active power in their system is proportional to firing activity in the network. They have built the TrueNorth system using these chips. TrueNorth has 1 million spiking neurons and 256 million synapses. This system includes offline learning algorithm software that maps existing algorithms to the architecture, including convolutional networks, restricted Boltzmann machines, and support vector machines; however, they do not discuss the efficiency of these offline learning algorithms. Based on the inclusion of these algorithms, it appears that this approach is primarily computation-driven.

Dynamically Reconfigurable Silicon Array with Conductance-Based Synapses

Vogelstein *et al.* introduce a silicon array of spiking neurons that includes synapses within the neuron itself [258]. The synapse within the neuron has a weight that depends on the event coming into the neuron (a conductance-based model). They also introduce a system called Integrate-and-Fire Array Transceiver (IFAT), which include neurons similar to those introduced in [258], but makes use of virtual synapses, stored in an off-chip look-up table, rather than stored on the chip along with the neurons [255]. They note that the IFAT architecture could easily include spike-time dependent plasticity as a learning mechanism, because the synapses are virtual [259].

Vogelstein *et al.* [258, 259] note that their goal in designing the IFAT was for ease of use and reconfigurability, as compared with the other systems, which tend to be more concerned with power costs or speed of simulation. Their approach is more computation-driven rather than biology-driven, though they do implement a model of dynamics and waves of neural activity in a rat hippocampus using the IFAT system

[259]. IFAT has also been used to detect salience in a visual image [255], spatially modulate a visual image [257], detect spatially oriented changes in real time [155], and compute a nonlinear pooling function [256].

FACETS and BrainScaleS

The FACETS (Fast Analog Computing with Emergent Transient States)* project and its continuation BrainScaleS (Brain-inspired multiscale computation in neuromorphic hybrid systems)† are large scale initiatives to develop neuromorphic architectures and simulations. They are part of the Human Brain Project in Europe. Their goal is provide software and hardware that can be used by non-experts, specifically by computational neuroscientists for modeling, so these projects fall primarily into the biology-driven category. A predecessor of the FACETS model is described by Schemmel *et al.* in [222]. In this work, a VLSI implementation of a spiking neural network model with spike-time dependent plasticity is described.

Ehrlich *et al.* describe the initial development of the FACETS hardware system in [72]. In particular, they present a configurable wafer-scale neural ASIC system that allows for large neural networks (more than a million neurons and a billion synapses). They describe a communication prototype based on layers that is capable of both local and long-range connectivity. Different layers communicate differently. For example, layer 0 represents communication between synapse groups and is a continuous time, analog connection, whereas layer 2 is long range connections between neural cores and is discrete and packet-based. They also describe a mapping software, which maps biological networks to the hardware-constrained networks on the FACETS system.

Schemmel *et al.* continue to develop wafer-scale FACETS hardware in [221]. In particular, they note that the FACETS system uses wafer-scale integration. Wafers are not cut into dies, but the chips are interconnected directly on the wafer. They discuss the initial version of the neural core, called the High Input Count Analog

*Website: http://facets.kip.uni-heidelberg.de/index.html.
†Website: https://brainscales.kip.uni-heidelberg.de/index.html.

Neural Network (HICANN), which contains mixed signal neuron and synapse circuits, as well as interface capabilities. Their neurons allow up to 16,000 input synapses, and their synapses include plasticity mechanisms. The neuron implementation allows for use of the adaptive exponential integrate-and-fire model and a simple conductance-based integral and fire model. Their implementation also allows for programmable topology of the network.

Millner *et al.* describe a VLSI implementation of the adaptive exponential integrate-and-fire model neurons for the FACETS hardware system in [168]. They demonstrate in simulation and in hardware that, by tuning parameters, this model can produce spike-frequency adaptation, bursting, regular spiking, irregular spiking and transient spiking. Their neuron has twelve parameters, seven of which are specified for each neuron individually, three of which are globally specified, and two of which are fixed. They scale the biological parameters in order to translate them for use in the hardware system. They allow for parameters to be read out from each neuron and allow for reading of parameters from two arbitrary neurons simultaneously. However, their neuron is not optimized to be low in power, though they note that it is significantly lower in power than supercomputer simulations. The full integration of these neurons within the FACETS chip is given in [220].

Brüderle *et al.* describe the entire workflow of the FACETS system, from software down to hardware in [47]. In this work, an overview of the FACETS hardware system, built on the HICANN chips, is described, as well as the interface to a host computer. The interface to a host computer is made up of the communication system between HICANN chips, an FPGA controller, and a Gigabit Ethernet layer. Their software interface is built using the modeling language PyNN (a Python package), a simulator-independent language for describing spiking neural network models. They develop software to map PyNN networks to the FACETS hardware. They also describe a set of benchmarks and their results on the system: layer 2/3 attractor memory model, synfire chains, and self-sustained asynchronous irregular states.

Pfeil *et al.* describe using a chip on the design in [222] within an integrated development environment that they use to emulate different types of neural networks [196]. Their architecture allows for almost arbitrary network topologies and a large number of parameters for synapses and neurons in order to allow for maximum configurability. In this work, they demonstrate in simulation and in hardware the ability of this model to produce six neural network models: a synfire chain with feedforward inhibition, a balanced random network, a soft winner-take-all network, a cortical layer 2/3 attractor model, an insect antennal lobe model, and a liquid state machine. They claim that their success on these network models demonstrates the universal applicability of their system.

Neurogrid

The Neurogrid project implements neurons as analog computational neurons in silicon. The goal of the project is biology-driven; in particular, the goal is to allow computational neuroscientists to perform real-time simulations of millions of neurons and their connections. In [58], Choudhary *et al.* demonstrate that their Neurogrid implementation is able to perform arbitrary mathematical computation. In particular, they communicate a scalar stimulus to the Neurogrid network, quadratically transform the value and compute the time integral in the network. They precompute the required weights and program those weights into the network.

Gao *et al.* develop a dynamical system mapping of quantitative neuronal models onto neuromorphic hardware [88]. They reproduce cortical modeling using neuromorphic approaches by making links between parameters in the neuron model and parameters in the neuromorphic implementation. This mapping was demonstrated on the Neurogrid architecture.

Benjamin *et al.* describe Neurogrid in detail and present comparisons to other architectures in [29]. Neurogrid implements all neuron elements except for the soma with shared electronic circuits and implements all circuits except for the axonal arbor in an analog manner. They chose to interconnect the network with a tree structure

in order to maximize throughput. Neurogrid includes a software component for visualization and hardware for real time simulations. The overall cost of Neurogrid in terms of area consumed, energy used, and time required is lower than those found in the BrainScale project and one of IBM's earlier chips. (The comparison was performed prior to TrueNorth's release.)

SpiNNaker (Spiking Neural Network Architecture)

SpiNNaker (Spiking Neural Network Architecture) is another component of the Human Brain Project in Europe. SpiNNaker's goal is not to produce biologically-accurate models but to produce models that are useful in computation by taking inspiration from the brain; thus, it is computation-drive. Furber and Temple discuss various issues with neuromorphic systems in [86] and provide an overview of the early SpiNNaker model. They optimize their architecture to work with neuron models including leaky integrate and fire and Izhikevich models. They claim that modeling a large system of spiking neurons is an embarrassingly parallel problem. They make use of embedded processors in large numbers, where one processor on a chip acts a monitor and the other processors model groups of up to a thousand neurons. Communication is done via a packet switching communications network. Furber and Temple emphasize the three-dimensionality of biological brains and notes that SpiNNaker can accommodate three-dimensional (and higher dimension) networks.

In [135], Khan *et al.* describe how SpiNNaker chips can be connected together to form large systems of neurons. They note that the simulation of activity in SpiNNaker systems is event-driven. The system of SpiNNaker chips is connected to a host computer, which configures the system. Neural networks are mapped to the hardware by first assigning neurons to processors and generating routing information by generating minimal routing tables (using software called SpiNNit). They implement a simple spiking neuron model and a layered neural network model trained with parallel distributed processing using their SpiNNaker chips.

Navaridas *et al.* describe the interconnection network of SpiNNaker and investigate its stability in [183]. The interconnection network of SpiNNaker is the system that implements communication between chips and is used to simulate synaptic connections. SpiNNaker chips are arranged in a grid, and each chip has six connections to other chips (north, south, east, west, southwest, and northeast). They investigated the stability of the network under worst-case scenarios and high degradation due to faults and found that SpiNNaker had a highly stable network. They have also calculated theoretical throughput and distance-related properties for a system of SpiNNaker chips.

Jin *et al.* provide an overview of SpiNNaker in [122]. In particular, they discuss the implementation of Izhikevich neurons for SpiNNaker. They note that the SpiNNaker architecture is guided by three principles: a virtualized topology, bounded asynchrony (i.e. not clock based), and energy frugality. They tested their SpiNNaker chip on a real application, "doughnut hunter," in which the neural network on SpiNNaker controls an agent that moves to chase a doughnut that appears in a random environment.

Rast *et al.* describe SpiNNaker as a neuromimetic chip, in that it retains some properties of a biological brain (massive parallelism and event-driven dynamics), but also has enough general purpose programmability to work with a wide range of network models [207]. They implement both a spiking neural network model with Izhikevich neurons and a classical multi-layer perceptron with delta-rule back-propagation synapses. Rast *et al.* also examine leaky integrate-and-fire neurons within the SpiNNaker architecture in [205]. In this work, they allow for multiple synaptic models with and without spike time dependent plasticity (STDP). They examine single neuron dynamics, spike propagation, and oscillatory network activity. Rast *et al.* also explore networks that contain both Izhikevich neurons and leaky integrate-and-fire neurons on SpiNNaker chips in [206], demonstrating flexibility in neural model selection, unlike many other neuromorphic implementations.

In [235], Sharp *et al.* examine the power usage of four interconnected SpiNNaker chips with a total of 10,000 spiking neurons and four million synapses. They note

their plans to interconnect 50,000 SpiNNaker chips in the future. They found that, in a real time simulation of this network, the hardware consume 100 nJ per neuron per millisecond, which they claim is the smallest quantity for any digital computer, and 43 nJ per synaptic event.

In [85], Furber *et al.* describe the current state of the SpiNNaker project. They note that, in one way, SpiNNaker is just another supercomputer or cluster. However, they also note that it has many differences from traditional super-computers. It is made up of medium performance components, in particular, 200 MHz ARM9 cores, rather than more expensive high performance components. They note that it disregards three major components of traditional supercomputing: memory coherence, synchronization, and determinism. They also note that it can be very difficult to design software to run on such a system because of its lack of these traditional components. In other words, they note that a mechanism to design software for SpiNNaker would be useful.

Summary

There are many approaches to neuromorphic computing, including both digital and analog implementations as well as hybrid approaches. These efforts are often either biology-driven or computation-driven in the way they are conceived and the way they intend to be used. Our hardware falls into the computation-driven realm, as we are less concerned with emulating what occurs in biology than we are with producing useful computation devices.

2.7 Summary

This chapter overviews the computer science and engineering literature on neuroevolution, neural network structures embedded in geometric space, neural networks operating on a continuous-time scale and spiking neural networks, and neural network structures that recognize and reuse useful substructures. Some of the

major works of the machine learning community that incorporate ideas of emotion or neuromodulatory systems into their learning systems are also reviewed. The commonality of these five components is that they are inspired by concepts in neuroscience; that is, they are each an effort to push the artificial neural network towards the biological neural network. This chapter also overviews efforts to produce neuromorphic hardware. The various works in this section have inspired many of the components of our research effort. In particular, our approaches are:

1. We design neural networks (parameters and structure) using an evolutionary optimization method. We use both crossover and mutation in our method and currently use a direct representation of the network. Moreover, our design method works on graph-like network structures, rather than the traditional, layered approach, which is fundamentally different than the approaches presented in Section 2.1.

2. Our neural networks exist in three-dimensional space. Other neural network approaches also have spatial relationships in their networks. One of the key points of our spatial layout is that the locations of neurons and synapses in the three-dimensional space has an effect on the dynamics of the network.

3. Our neural network falls into the category of spiking neural networks. Our neurons are relatively simple compared to most other spiking neural networks, but we expect that we can trade off neuron complexity for network complexity.

4. We intend to make use of useful substructures in our networks. In particular, we want to identify and reuse useful subnetworks in order to improve optimization. Most other works do not discuss how to identify useful subnetworks, though some note the importance of reuse of substructures.

5. We intend to include affective systems in some of our networks. Other machine learning techniques include some notion of affective systems, and the most closely related to our work is Krichmar's neuromodulation, as discussed in

Section 2.5. However, unlike their work, we intend to evolve our affective systems alongside our neural networks, rather than predefining the behavior those affective systems.

6. Our proposed hardware implementation differs from most other neuromorphic implementations in the programmability of the structure of the networks. Other neuromorphic implementations choose a fixed structure (usually a fully-connected network), whereas we allow the user to specify the number of neurons and synapses, as well as the connectivity of the system.

As our approach is presented in more detail in the remaining chapters, many of the topics in this chapter will be revisited and referenced as inspiration for the work.

3

Architecture and Design Method

"Let us step into the night and pursue that flighty temptress, adventure."

– J.K. Rowling, *Harry Potter and the Half-Blooded Prince*

This chapter describes the titular neuroscience-inspired dynamic architecture (NIDA) and the associated design method based on evolutionary optimization [40, 228, 229, 230]. A sequential implementation and a parallel implementation of the design method are described. Results on several tasks using NIDA networks and the basic design method are also presented.

3.1 Neuroscience-Inspired Dynamic Architecture

Our neuroscience-inspired dynamic architecture (NIDA) is made up of neurons and synapses and is defined as a grid in three-dimensional space. Maximum x, y, and z (called M_x, M_y, M_z) values and minimum x, y, and z values (called m_x, m_y, and m_z) are defined for a network, as well as the granularity of the grid, which specifies how close two neurons in the grid may be. Neurons are defined at coordinates in the grid.

The neurons within the network can be at coordinates (x, y, z), where $m_x \leq x \leq M_x$, $m_y \leq y \leq M_y$, and $m_z \leq z \leq M_z$. The granularity parameter specifies how close two neurons in the grid can be. For example, a granularity of one implies that the minimum distance between two neurons in the grid is one unit, and it also implies that all of the neurons in the grid will be located at integer coordinates. We may also specify larger or smaller granularities. Thus, there is a bound on the number of possible neurons in any given bounded cube, dependent on the maximum and minimum defined values as well as the granularity parameter. We chose a three-dimensional representation because biological networks are three-dimensional and because it allows for easy representation in the evolutionary optimization (discussed in Section 3.2). However, depending on the application, a two-dimensional network may be required. This can be achieved by setting one of the maximum and minimum values at zero (for example, $M_z = m_z = 0$). In this case, networks that are formed will be located in a two-dimensional grid, rather than a three-dimensional bounded box. We may also extend NIDA networks to exist in larger dimensional spaces.

Simulations in NIDA networks take place at the network level and are discrete event simulations. Networks have associated event queues, in which different event types are specified to occur at some time in the simulation. A unit of simulation time is the amount of time it takes for charge to travel one unit in space. For example, if two neurons are connected and are one unit apart (i.e., a neuron at (0,0,0) and a neuron at (0,0,1)) then one unit of simulation time is the amount of time required for charge to travel from one of the neurons to the other. It may be advantageous to adjust this charge propagation velocity during training, in particular if multiple networks interact (see Chapter 5).

Neurons are located at a point in three-dimensional space. Two neurons cannot occupy the same point in space in the same network. Neurons can be input neurons, output neurons, both types or neither type, depending on the requirements of the network. Neurons that are neither input nor output neurons are known as hidden neurons. Each neuron has an associated threshold and refractory period in firing.

Neurons are connected to other neurons via synapses. These synapses are directed, so each neuron has a set of synapses *to* other neurons and a set of synapses *from* other neurons. The primary actions of a neuron are changes in charge and firing. Charge is received by a neuron from its synapses. The charge on a neuron is accumulated until that neuron's threshold is reached. When the threshold is reached, if the neuron is *not* in its refractory period, the neuron fires, and the neuron's charge is reduced to zero (or neutral, as the charge and/or threshold may also be negative). If the neuron is within its refractory period, then the neuron maintains its charge but does not fire. Thus, a neuron can accumulate charge during its refractory period, but it cannot fire during this period. As soon as a neuron fires, it enters its refractory period. This is similar to what occurs in biological neurons, although it is not impossible for a biological neuron to fire during its refractory period [65]. If the neuron's charge exceeds its threshold at the end of its refractory period, it will not fire unless stimulated by another event that changes the charge on the neuron in such a way that the threshold is exceeded. Currently in our model, the charge values and threshold values of the neurons are bounded between -1 and 1, though these values may be altered as needed. By bounding the charge on our neurons and by imposing a refractory period on a neuron, we guarantee that the behavior of the network is stable. In particular, neither charge nor firing activity may become unbounded. Based on the refractory period and the number of neurons in the network, a maximum firing rate for the network may be determined. For example, if the minimum refractory period of neurons in the network is one, then the maximum firing rate of the network (number of neurons firing within any given time unit) is the number of neurons in the network.

Synapses in our implementation are defined by the neurons they connect. Specifically, each synapse goes *from* one neuron (the pre-synaptic neuron) *to* another neuron (the post-synaptic neuron). Any two neurons in the network may be connected by one or more synapses (including a neuron to itself, in which the delay is 0). For simplicity, the synapse also maintains the distance between these two neurons and the weight (or strength) of the synaptic connection. The distance between the two

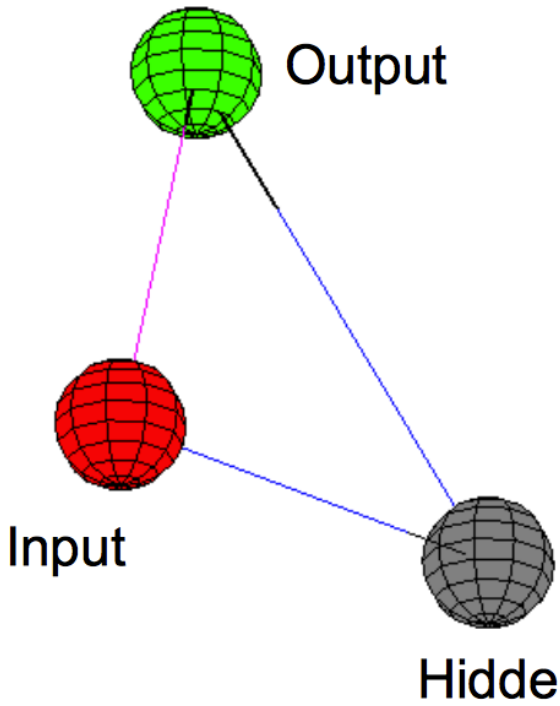


Figure 3.1: An example NIDA network with input, output, and hidden neurons labeled. Synapse labels and directionality are given in Figure 3.2.

neurons affects how long it takes for charge to travel along the synapse, and the weight of the synaptic connection determines how much charge arrives at the post-synaptic neuron after the first pre-synaptic neuron fires. The weight of the synapse is currently bounded between -1 and 1; again, these values may be adjusted according to the needs of a particular simulation. As is the case in biological neurons, synapses may be inhibitory (negative), in which case the firing of pre-synaptic neuron results in a decrease of charge at the post-synaptic neuron, or they may be excitatory (positive), in which case the firing of the pre-synaptic neuron results in the increase of charge at the post-synaptic neuron.

NIDA simulations do not currently implement the concept of myelination; if two synapses are each of length d, then it takes the same amount of time for charge to travel from one end of each synapse to the other, so distance is synonymous with delay. This could be adjusted, however, by altering the propagation velocity on a synapse by synapse basis, or for the entire network.

The major actions associated with synapses are processes similar to long-term potentiation (LTP) and long-term depression (LTD), forms of Hebbian learning described in Appendices A.4 and C. If charge traveling along a synapse from neuron A to neuron B causes neuron B to fire, then the weight of that synapse increases. In our implementation, LTD occurs at that synapse if charge is received by neuron B during its refractory period. LTP increases the weight of the synaptic connection by a fixed value (specified for the network), and LTD decreases the weight of the synaptic connection by the same fixed value. Synapses have a refractory period associated with LTP and LTD, which prevents changes to the weights from occurring too rapidly.

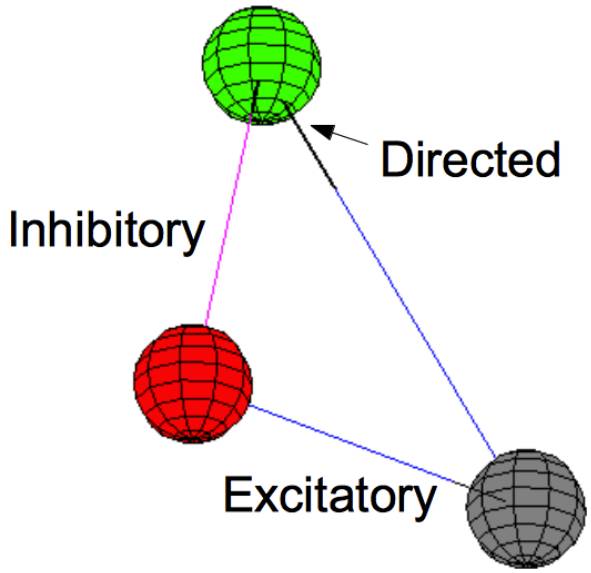


Figure 3.2: An example NIDA network with inhibitory and excitatory synapses labeled and directionality of the synapses noted.

The event types used in every simulation are change-in-charge in a neuron, firing of a neuron, and an input pulse. The change-in-charge of a neuron and the firing of a neuron are internal events. They are caused by events within the network. Input to a neuron events (or pulse events) are events in which the network interacts with its environment. Additional events are used as part of the simple simulated affective system, which will be discussed in Section 5.1.1. A change-in-charge event in the network is associated with a single synapse and is triggered by a single fire event.

The post-synaptic neuron is the neuron on which charge is changed. A change-in-charge event may trigger at most one fire event. A fire event in the network is associated with a neuron and may be triggered or affected by multiple change-in-charge events or input (pulse) events. Fire events also trigger change-in-charge events upon all outgoing synapses from the neuron that fired. Figure 3.3 shows a brief event simulation example.

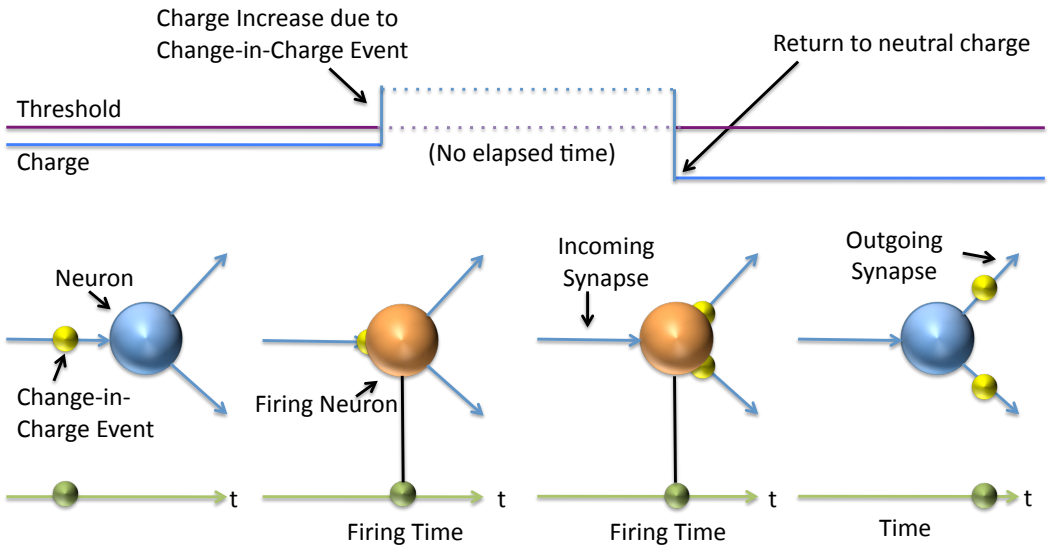


Figure 3.3: An example NIDA network event simulation. In this figure, there is a change-in-charge event traveling along a synapse towards a neuron. When that event reaches the neuron, it causes the neuron's charge level to increase over the neuron's threshold, which causes that neuron to fire. We show the change-in-charge event reaching the neuron and two newly created change-in-charge events in two separate pictures, but the two middle pictures occur at the same time in the simulation.

NIDA falls into the category of continuous-time recurrent neural networks (CTRNNs). As discussed in the literature review, CTRNNs use differential equations to determine behavior. NIDA uses discrete events instead. This may make the operation of the network slower, but it allows for propagation delay to play a complex role in the network's behavior. In CTRNNs with propagation delay, stability of the network can be an issue. To avoid this problem in our network structure, we impose

a refractory period on the neurons (similar to those that exist in biological neurons) in order to limit firing, and we impose a charge limit on the neurons in order to limit the amount of charge a neuron can acquire. Based on preliminary testing, we selected a refractory period of one unit of simulation time. This refractory period is imposed in all networks given unless otherwise noted. NIDA networks can also be classified as types of spiking neural networks.

Charges in neurons and delays in the synapses allow for networks to store information in a distributed way over the course of a single simulation; thus, both NIDA neurons and NIDA synapses have "memory storage" mechanisms. Using variable length delays and cycles, information can be stored and accumulated for any period of time required for a particular task.

The architectural formulation admits many roughly equivalent optimal solutions for a task. Location of neurons and synapses plays a role in our network design, but the operation is invariant to translation and rotation in the three-dimensional space because the delays in the network are maintained in these cases. This is not true for spatial scaling. Scaling the network fundamentally changes the network's operation because the delays in the network are changed; we can achieve a similar effect by altering the propagation velocity in the network.

3.2 Design Method

It is possible to design NIDA networks by hand to perform useful tasks. For example, a two neuron NIDA network can be hand-tooled as a simple oscillator (Figure 3.4). Similarly, central pattern generators [118, 158] can be hand-tooled using our networks. Most tasks, however, require networks that are too complicated to be hand-tooled by a user. Thus, an automated design method to produce network architectures that solve tasks reasonably well is preferable. We selected an evolutionary optimization method to design our networks.

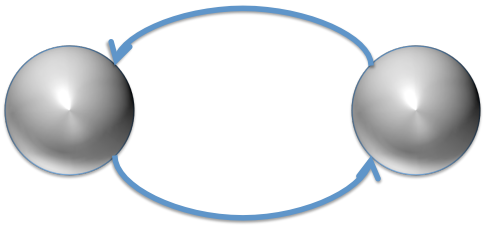


Figure 3.4: Two neuron oscillator, connected with excitatory synapses. By setting the weights and thresholds correctly, one may design this network so that the firing of one neuron causes an interleaved firing of both neurons at regular intervals.

There are several issues that must be addressed when considering the use of evolutionary optimization for neural networks in general and NIDA networks in particular. These issues include representation, measures of performance, and creation of offspring.

3.2.1 Representation

The first issue that must be addressed when using evolutionary optimization to design NIDA networks is how the network will be represented. We chose a direct representation of the network structure, rather than an indirect representation, meaning that every parameter in the network is directly represented in the representation. We chose the direct representation because it is easier to define, but future implementations may use an indirect representation. The importance of representation is that it must maintain some link to the functionality of the network. With traditional neural network structures, it can be difficult to tell whether two networks are very similar (perhaps off by just a few neurons or synapses), or entirely different. Since the NIDA network structure is embedded into a topological space, the representation is entirely unambiguous. That is, there either is a neuron at (0,0,0) or there is not a neuron at (0,0,0). Similarly, if there is a neuron at (0,0,0) and a neuron at (0,0,1), then there either is a synapse between them or there is not. Since the functionality of the network

depends not only on the structures, but also on the distances between the structures, the inclusion of this topological property in the representation is important.

It is important to note that it is possible for two networks to have the same functionality while being represented by two distinct neural network structures (a version of the competing conventions problem). This is because the behavior of the network is invariant to translation and rotation; for example, the same network shifted two units over and rotated by ninety degrees will have the same functionality as the original network. To combat the issue with invariance to translation, for each task we fix the positions of the input and output neurons in the network; so, every network in the population has inputs and outputs at exactly the same position. In general, this does not address the issue of invariance to rotation (as the axis of rotation might be along the line between a single input and output neuron). We argue that this is not as much of an issue, particularly in networks where there are multiple input or output neurons. In fact, it may very well help the functionality of the evolutionary algorithm, as it means that there are potentially many optimal network structures.

3.2.2 Performance

The second issue that must be addressed when using evolutionary optimization for network design is how the fitness or score of each network will be determined. The concept of fitness of a network is fundamental to the use of evolutionary optimization. The selection of members of the population to reproduce for the next generation is entirely determined by the fitness of a network.

Fitness is an application-specific issue. That is, the fitness of a network depends on what problem the network is attempting to solve. When working with our software package, there are two functions that the user must define for each usage: an initialization function and a fitness function. The initialization function provided by the user defines how the initial population should be defined. The most important part of the initialization function provided by the user is where input and output

neurons should appear in the networks (as this will be the same for each network in the population). This initialization function also allows users to impart any prior knowledge they may have about the problem structure; for example, if a user knows that the network will require at least 100 neurons in order to solve the problem at hand, he or she can initialize all of the networks in the initial population to have 100 or more neurons.

The second function the user must define is a fitness function. The fitness function takes as an argument a network structure and returns a numerical value associated with that network. There is no restriction on what that numerical value may be; it can be negative, positive, large or small. The only requirement of the return value is that higher values are associated with higher performing networks and lower values are assigned to networks that perform poorly on the task. These values are only relative to what occurs in the application. For example, when considering a classification problem, the typical goal of the training process is to minimize the error on a training set (assuming the error used is a positive value, such as mean-squared error). For this type of problem, the fitness function may return the negative of the error, so that functions with lower errors will have higher fitness values. For a reinforcement learning task, the goal is to maximize the reward received. In this case, the fitness function might return the reward received over the course of one episode. An equivalent formulation of the fitness function would be minimizing a function value, in which case, we could consider the inverse of the reward values so that higher rewards give lower values. The choice of fitness function makes a profound difference on the performance of the evolutionary optimization for any given task. Selecting a proper fitness function for evolutionary optimization is oftentimes as much an art as it is a science.

3.2.3 Creation of Offspring

Evolutionary optimization relies on the ability of well-performing networks to "reproduce." That is, well-performing networks (as determined by the fitness function) should be selected to create children networks that share some or all of their functionality (see Section 3.2.5 for how these reproduction operations are used to repeatedly create child populations from parent populations).

As in many evolutionary or genetic algorithms, we have two operations that create offspring: mutation and crossover. Both mutation and crossover have associated rates at which they may occur. However, with the direct representation of the network, neither mutation nor crossover is straightforward.

Mutation

We define several mutation operations in our genetic algorithm. Each mutation operation has an associated probability of occurrence; there is a default probability of each operation occurring, but these probabilities can be specified by the user. The mutation operations in our system are:

- Change the sign of the weight of a randomly selected synapse in the network, which is the same as switching a synapse from inhibitory to excitatory or vice versa.
- Randomly change the weight of a randomly selected synapse in the network.
- Add a synapse between two randomly selected neurons in the network where a synapse did not exist previously.
- Delete a randomly selected synapse in the network.
- Add a neuron to the network at a randomly selected vacant location. This operation also adds a synapse to a randomly chosen existing neuron in the network and a synapse from a randomly chosen existing neuron in the network to the new neuron.

- Change the threshold of a randomly selected neuron.
- Delete a randomly selected neuron in the network. This neuron cannot currently be an input or output neuron, though this may be changed in the future. For example, we may allow input neurons to be deleted in the future. This way, the network structure can decide which input is required in order to solve the problem. This operation also deletes all synapses associated with the deleted neuron.

Since our mutations are often very small changes to the network and only one mutation is applied to a network at a time, we typically select our mutation rate to be very high (greater than or equal to 0.9, or that mutations occur in approximately 90 percent of the child networks). This may be changed in future iterations of the software. For example, we may update the mutations so that many small changes are made and decrease the selected mutation rate.

Crossover

Crossover is a much more complicated operation for complex representations. One of the goals of the crossover operation is that some of the functionality of both parents should be retained in each of the children. The functionality of our network structure is linked to the number of neurons and synapses, the placement of the neurons and synapses, and delay, which is determined by the distance between two connected neurons. With this in mind, crossover is implemented as follows. A random plane through the cube that contains the network is chosen. This plane is determined by randomly choosing two neurons in one of the parent networks, letting one be a point in the plane, and letting the vector between the two neurons be the normal vector of the plane. Then, the plane is placed into both networks, splitting each of the parent networks into two parts, part A and part B. If the neuron that is normal to the plane appears in part A, then the neuron on the plane appears in part B, and vice versa.

One child is constructed by taking all of the neurons in part A of parent 1 and all of the neurons in part B of parent 2, and placing them in the network. Similarly, all of the neurons in part B of parent 1 and all of the neurons in part A of parent 2 are placed in the network of the second child. Determining where synapses should go in the network is a slightly more complicated process. All of the neurons from part A of parent 1 are in child 1. For each synapse going from a neuron in part A of parent 1, the following process is followed in child 1: if the second neuron also exists in child 1, then the synapse is formed between those two neurons with the same weight as the original synapse from the parent. If the second neuron does not exist in child 1, then the neuron in the child spatially closest to the location where the corresponding neuron appears in the parent is chosen. A synapse is created between this neuron and the original neuron with the same weight as the original synapse. A similar process is followed for part B of parent 2 in the first child, part B of parent 1 in the second child, and part A of parent 2 in the second child. This attempts to preserve as much of the structure that exists in the parents in the children, while also attempting to preserve the delay properties of the parents in the children. A crossover example is shown in Figure 3.5. A flow chart detailing this crossover operation is given in Figure 3.6

There are other possible crossover operations. One possible operation is a "random crossover," in which, for every neuron in each parent network, that neuron is randomly passed to child 1 or child 2. In this operation, synapses would be passed down in the same way as in the plane crossover.

3.2.4 Advantages and Disadvantages of Evolutionary Optimization

The search space for NIDA architectures is extremely large, encompassing both continuous parameters and discrete structure, and the fitness function is non-convex. Constructing an optimal or near-optimal solution is both a combinatoric problem and

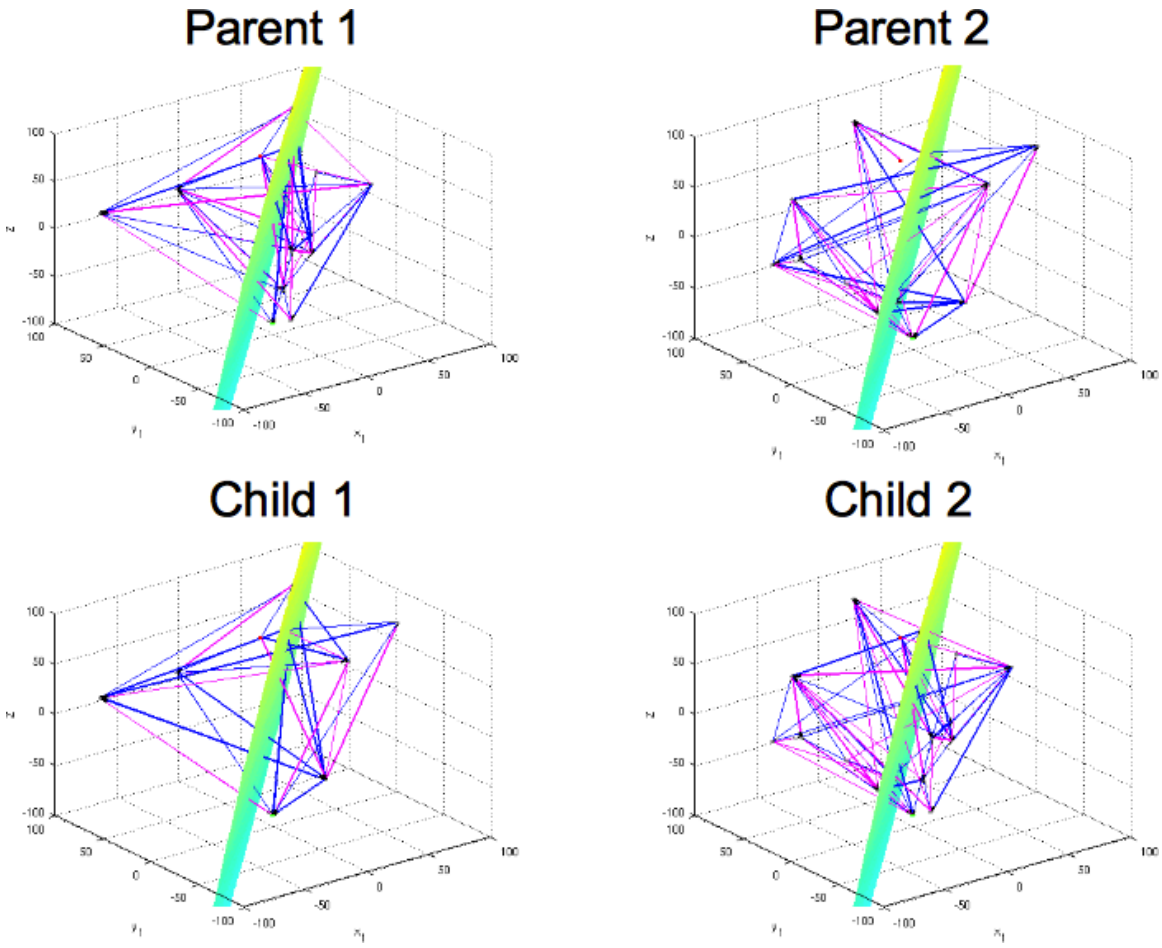


Figure 3.5: Example plane crossover parent and child networks. In this example, the part of parent 1 to the left of the plane goes to child 1, and the part to the right of the plane goes to child 2. Similarly, the part to the left of the plane of parent 2 goes to child 2, and the part to the right of the plane goes to child 1.

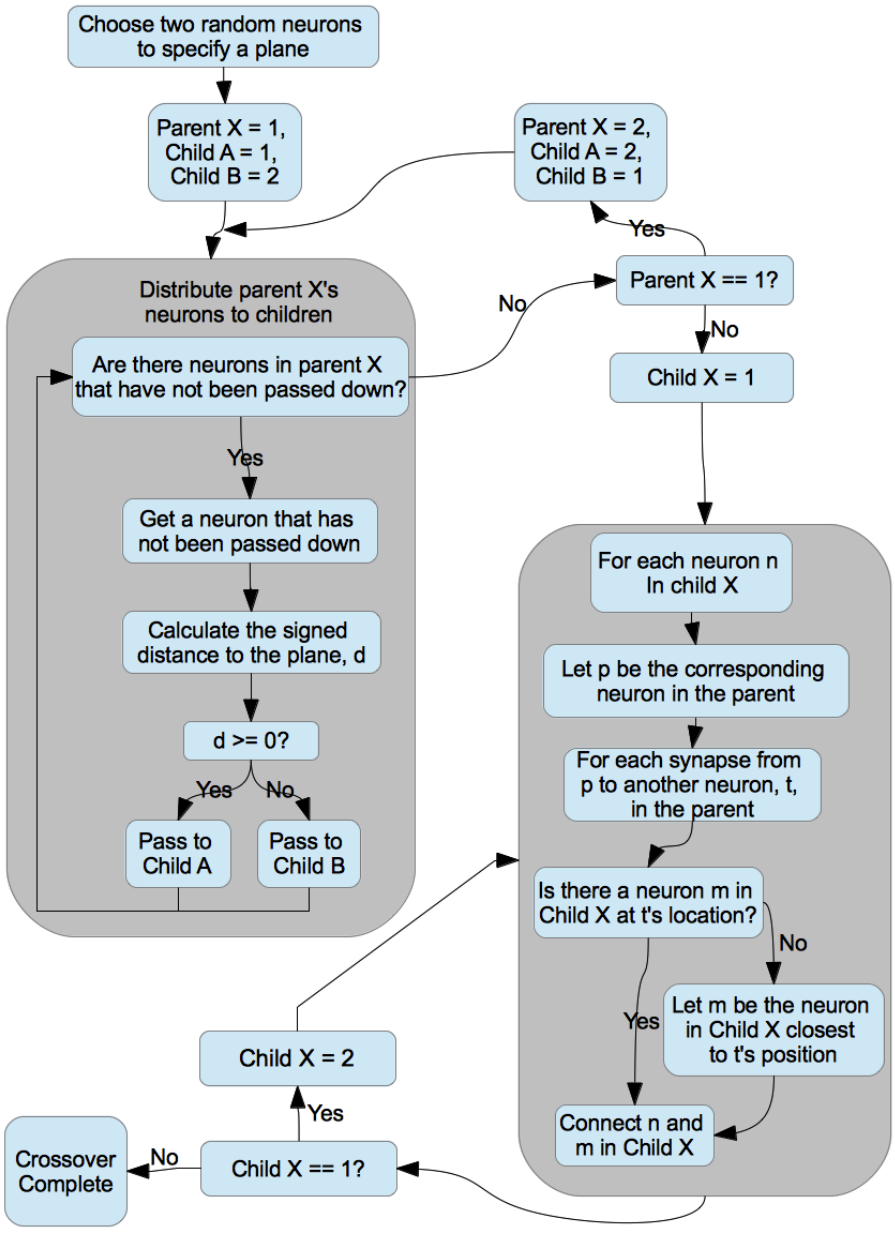


Figure 3.6: Flow chart depicting the crossover operation.

a traditional (continuous parameter) optimization problem. Traditional, non-linear optimization techniques are not useful in this context unless the starting random initialization of the network is both structurally correct and close to a local minimum. Evolutionary optimization (EO) is a randomized search method and tends to avoid issues associated with bad initial starting points.

One reason we chose EO is its flexibility. By selecting an appropriate flexible fitness function, we can use the same method when designing a NIDA network for a supervised task (in which there is an error that is being minimized), a reinforcement learning task (in which there is a reward that is being maximized), or any other task in which we can compare the performance of two NIDAs. It is worth noting that the efficiency of EO is highly dependent on the efficiency of the evaluation of the fitness function, since every member of the population is evaluated using the fitness function many times.

Because EO discovers the parameters, structure, and dynamics of the network, the user is not required to know any information about the NIDA network (other than interfaces to its environment). In contrast, the user of traditional ANNs often has to pre-specify some characteristic of the network (such as number of neurons and connections) before training. Selection of the number and placement of neurons is known to have a significant effect on the performance of traditional ANNs [117]. Determining the structure of the network may be a difficult task for the user, especially if the user has no intuition about what type of structure is required for a particular task. In the case of problems in which the user knows something significant about the number and/or placement of neurons and/or synapses, our design method can easily accommodate this information and force all networks in the EO population to have these characteristics.

Our NIDA networks have a defined neuron and synapse structure. However, the design method does not depend on this defined structure. For example, our neuron model could be replaced with a more complex neuron model, but the design method

would not have to be changed; however, if the neuron model is more complex, the performance of the EO may suffer.

Another major advantage of using EO is that the user can include as few or as many restrictions on the types of NIDA networks that are produced as required by the current application. For example, depending on the application, the user may wish to restrict networks to two-dimensional representations, rather than three. This is a special advantage when considering a network implemented in hardware. Several neuroscience-inspired computational elements are currently being implemented in hardware (as discussed in Section 2.6), and the simplicity of our NIDA architecture lends itself to a hardware implementation. A hardware definition of a NIDA network may require more restrictions than a simulated NIDA network (see Chapter 6). For example, number and placement of wires in the network can be an issue. Using EO to design the network allows these placements to be optimized in simulation and then implemented in hardware.

There are many good reasons to use EO as the design mechanism for our NIDA network. However, the major downside of using EO, as opposed to a specially tailored algorithm for a particular network structure or definition is that EOs can converge much more slowly to a solution. In essence, by choosing to use EO, we are trading off programmer time required to select a reasonable structure for computer time.

3.2.5 Sequential Implementation

Figure 3.7 is a flow chart of the current training algorithm. The selection of two parent networks is done using either roulette selection or tournament selection. For roulette selection, two random networks are chosen, where networks with higher fitness are more likely to be chosen than networks with lower fitness. Tournament selection has an associated parameter N, where N is less than the population size. For tournament selection, N randomly selected members are chosen uniformly from the population. Then, the member with the highest fitness in this group of N is chosen. Each new

generation of networks constitutes an epoch. Number of epochs to converge is one metric that can be used to compare how quickly convergence occurs. This measure of convergence makes sense only in the context of comparing two algorithms that do something comparable within a single epoch (e.g., when comparing two instances of the same algorithm with different parameter values). Since the computation that takes place in a single epoch includes evaluating the fitness for each member of the population, the number of members of the population can radically affect the training time.

3.2.6 Multi-Process Implementations

In order to deal with larger populations, larger networks, and more complex simulations, a more complex software model of the genetic algorithm is required. The goals of the new models (GAv2 and GAv3) are to be scalable and parallel, but they must also work within the confines of the resources available.

GAv2: One Process per Network

GAv2 is composed of a single server process and many client processes. Each client process handles a single network from the population, calculates fitness values for that network, and runs a single network training process similar to simulated annealing (see Appendix C for information about simulated annealing). The server process handles all of the computations associated with the genetic algorithm. Sockets are used for process-to-process communication. Each client communicates with the server using a socket, and there is no communication between client processes except through the server process. The server process provides a socket to which client processes can connect. Both the server process and the client processes are multi-threaded in order to handle communication. An overview of the server and client processes and their associated threads is shown in Figure 3.8.

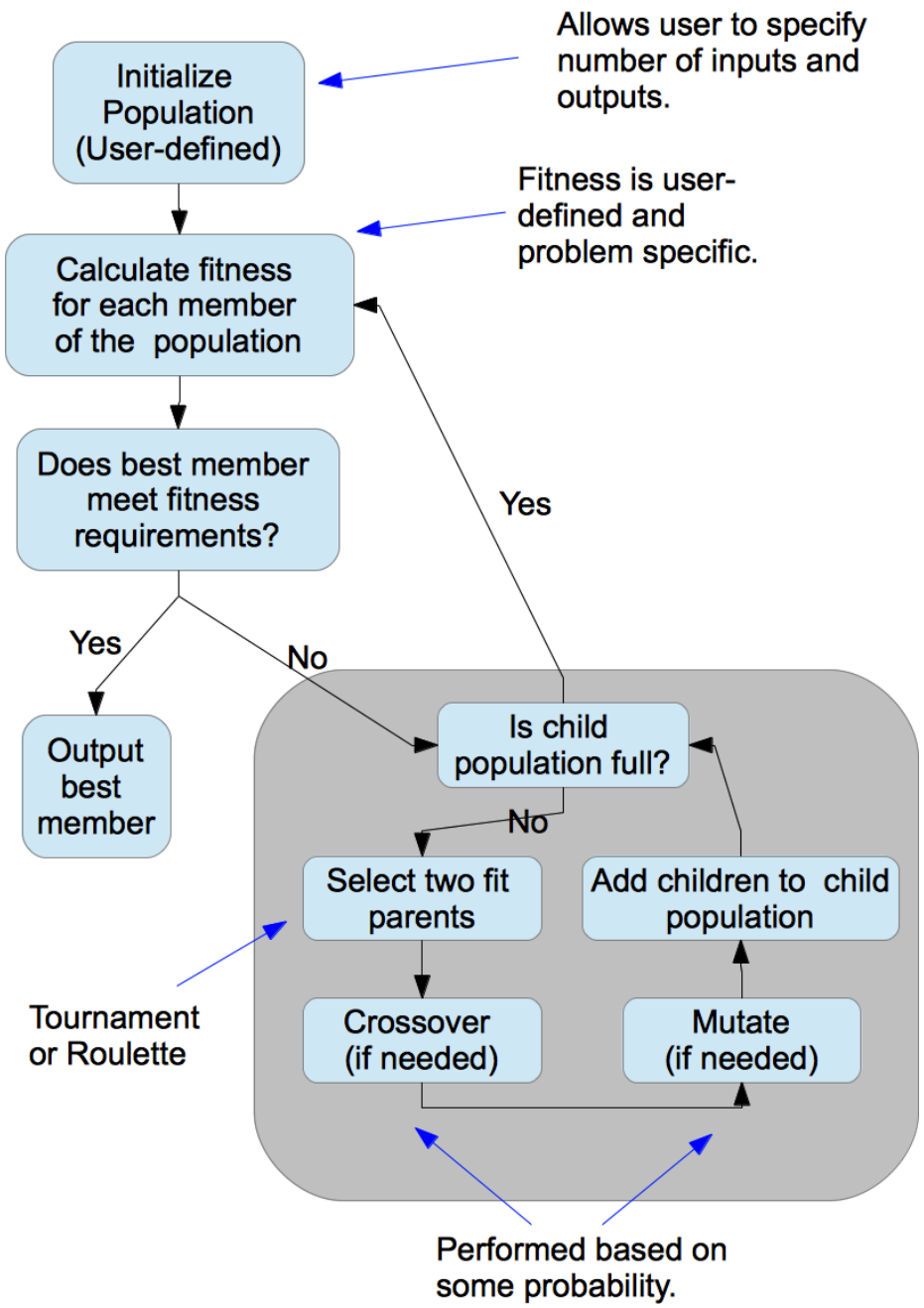


Figure 3.7: Flow chart depicting the evolutionary optimization used.

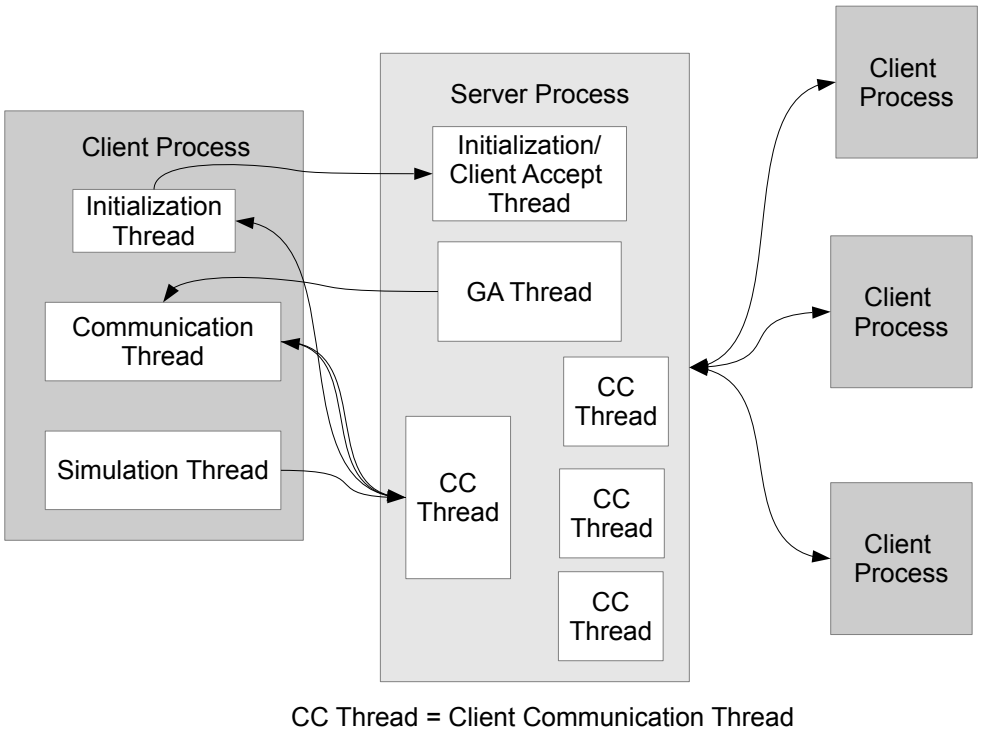


Figure 3.8: Overview of the client and server processes and their associated threads. One client process is shown in expanded view so that the threads within it can be seen. Communication between threads is shown with arrows.

The server process first initializes a population of networks and adds those networks to a "pending queue." The pending queue contains all of the networks that need to be sent to client processes to be simulated. Networks are added to this queue only upon initialization of the server and upon death of a client process (discussed later in this section). The initialization of networks is application-specific, as the initialization of the network defines the number and placement of the input and output neurons, which differs for each application. Since the server process is application specific, it can only handle one application at a time; one can create multiple server processes in order to simultaneously process multiple applications.

The server process initializes the genetic algorithm (GA) thread, which will be discussed shortly. The server process creates multiple client processes (one for each member of the population, up to the desired population size). The creation of these client processes may be done by executing a script that either launches new processes on the same machine, submits a request for jobs on a cluster, or launches processes on a different machine. The server process then serves a single socket to which clients may connect. The server process then waits for a client process to connect. When the server accepts a connection from a client, the server creates a client-communication thread, so each client process has its own client-communication thread. This process is shown in Figure 3.9.

The server process creates a client-communication thread when it accepts a connection from a client process (Figure 3.10). Each client-communication thread is responsible for communicating with its corresponding client process. The number of client threads in the server process is governed by the number of client processes the server creates. The first action of a client-communication thread within the server process is to remove a network from the front of the pending queue of networks and to send that network to the client thread. Once a network has been sent to the client, the client-communication thread waits to receive a mode from the client process. Those modes (fitness, parent, dying, and best) will be discussed further below.

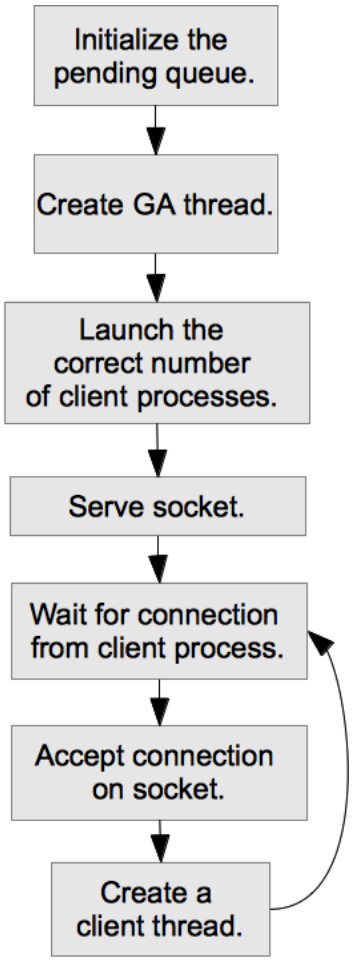


Figure 3.9: Flow chart for the initialization/client accept thread of the server process.

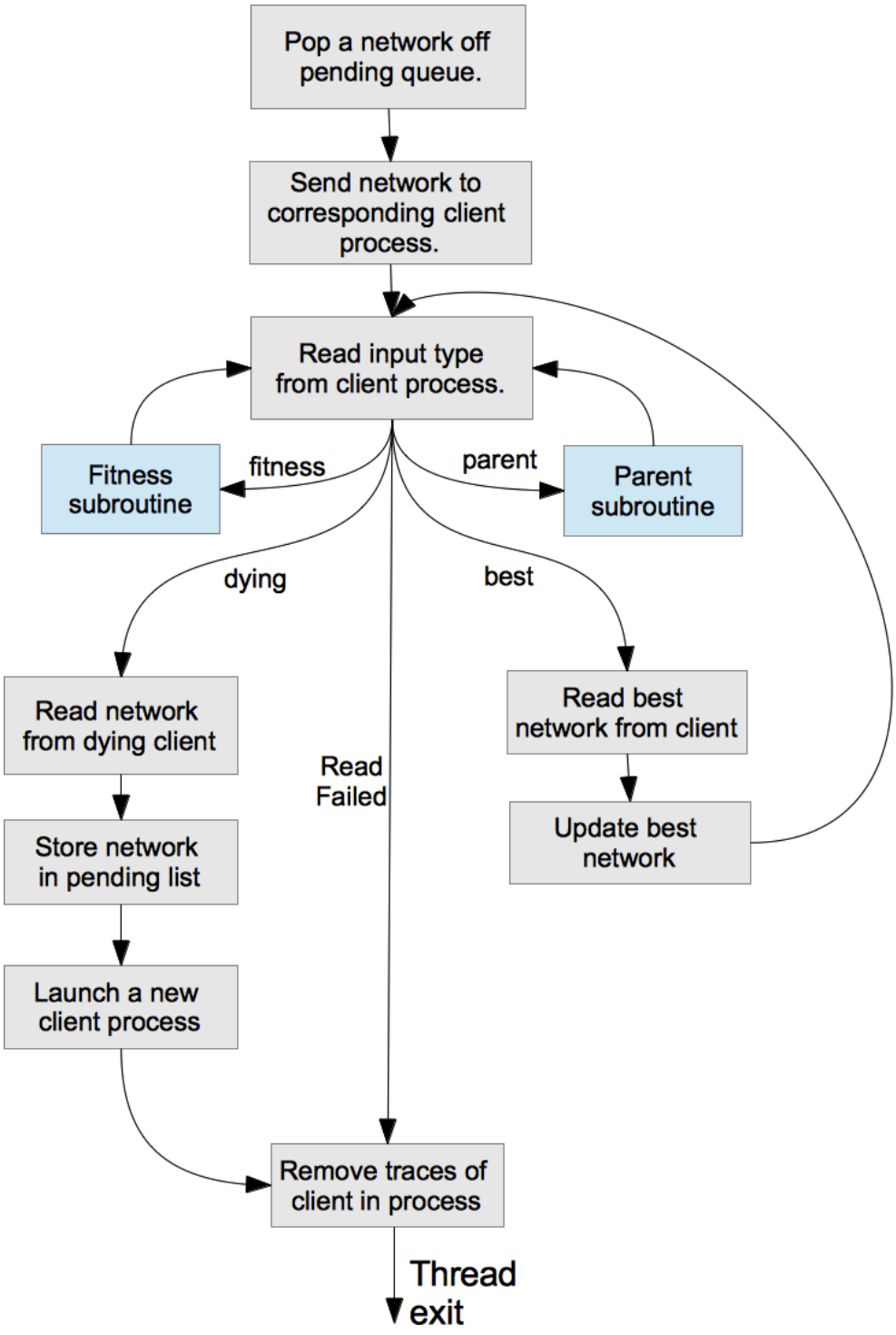


Figure 3.10: Flow chart for the client-communication thread of the server process. The fitness subroutine is shown in Figure 3.13, and the parent subroutine is shown in Figure 3.16.

A client process is created either directly by the server process or through some intermediate script launched by the server process. Each client process will be given a host name and port number. The host name defines the machine on which the server process runs, and the port number is the port on which the socket was made available by the server process. As noted above, this connection to the server process causes the server process to launch a client-communication thread, which immediately sends a network to the client process. The client process then reads and stores this network.

The client process is responsible for simulating the network and interacting with the environment for a particular task or application. The client process deals with a new event type: the "interact-with-environment" event type. The interact-with-environment event causes a new input to be applied to the network from the environment and may also include processing an output from the network or updating a simulation of the environment. (Optionally, multiple inputs and/or outputs could be handled by this type of event.) The functionality of the interact-with-environment event type is application specific. For example, for a pole balancing task, the interact-with-environment event will check the output of the network, update the simulation of the environment (the cart and pole) based on the output of the network (by applying forces based on the output of the environment), and add new inputs to the network simulation based on the updated environment. The simulation of the environment may be carried out by a separate process or thread with special purpose hardware or implemented by a physical (or virtual) system.

Immediately after the client process reads its initial network from the server, it will add an initial interact-with-environment event to the network's event queue to be processed, so that it will trigger the first set of inputs to the network's simulation. Once all of these initializations are complete, two threads are launched in the client: the communication thread, which is waiting to receive input from the server, and the network simulation thread (hereafter refereed to as "simulation thread"), which does all of the computation and simulation in the client thread. The creation of these two threads is the end of the initialization thread of the client process, so the initialization

thread will exit. Figure 3.11 shows the operations of the initialization thread of the client process.

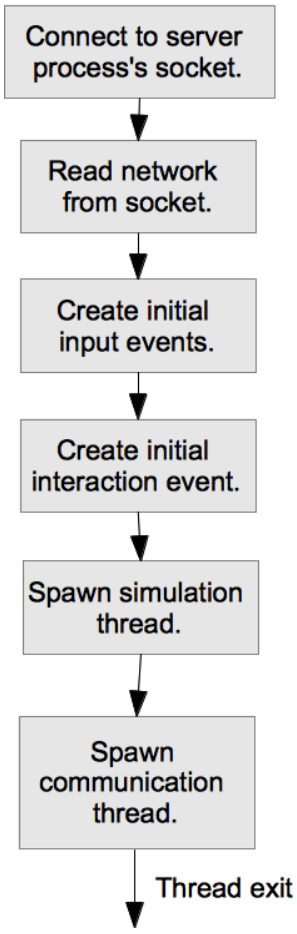


Figure 3.11: Flow chart for the initialization thread of the client process.

The simulation thread of the client process is where the computation and simulation of the network is performed. This thread conducts the simulation of the network and also deals with interact-with-environment events. In particular, the thread deals with the network's event queue. It removes events off the front of the queue and processes those events. All events except for the interact-with-environment events are processed as normal for simulating the network. These events include input pulses, change-in-charge events and fire events. The primary computation in the simulation thread occurs when the interact-with-environment event is encountered.

On an interact-with-environment event, the environment is updated based on the network's state, and new input events are generated from the updated environment. At this point, the network may or may not be ready to be scored. For example, in some control tasks, the environment will provide inputs at several different times, and the output of the network will change the state of the environment several times before a fitness score can be obtained. If the task is not complete or an end state has not been reached when the interact-with-environment event is being processed, a new interact-with-environment event is scheduled (i.e., added to the event queue), and the network and environment simulations continue. If the end of the task or some end state of the environment (or an intermediate state for ongoing, online or real-time applications) has been reached, the network is ready to be scored. The network that is simulated is called the "working" network. The client process maintains the best fitness score it has seen over its lifetime, as well as the network associated with that score; thus, if the working network's score is the new best score, the best score and network are updated accordingly.

To avoid simulating the same network many times and to cut down on communication with the server process, some training is done in the client process. In particular, small-scale mutations are made to the working network in order to attempt to improve the performance of the network. Since these changes may hurt the performance of the network, the working network is occasionally replaced with the best network. This replacement occurs randomly with respect to a "temperature" value (similar to a temperature value in simulated annealing). The temperature value starts high relative to its final value. As more and more changes occur and the fitness value increases, the temperature value decreases, making the simulation thread less likely to accept a poorer performing network.

Each calculation of a new score constitutes an epoch in the client thread. Again, to cut down communication with the server socket, the current best score in the simulation is sent to the server process during the interact-with-environment event every N epochs, where N is defined by the user. Only the score is sent to the server

process. If the server process needs the network associated with that score, it will prompt the client process for that network (either with the "best" mode or "parent" mode). This completes the processing of the interact-with-environment event when the network needs to be scored. At this point, the simulation may or may not reset. If the simulation is reset, new input events are created for the new simulation, along with a new interact-with-environment event, scheduled for some point in the future of the simulation. The complete functionality of the simulation thread for the client is given in Figure 3.12.

As noted above, the simulation thread of the client process intermittently sends fitness scores to the server process. The fitness score package includes a mode, in this case "fitness" mode, which tells the server process what to read from the client and what to do with that input. For fitness mode, the server process knows to expect a fitness score. Once the server process has read the score, it proceeds to the fitness subroutine (Figure 3.13). The server maintains a listing of client ID numbers and scores associated with those IDs. When a fitness score is received from the client process, the server process checks to see if the listing of client IDs includes the current client's ID. If it does not, this means that this the first time that client has sent a score associated with its current network. We call this client "newly mature" because it is the first time that client has sent a score to the server process. We add the associated client ID and score to the listing of clients. The server process does not initiate a reproduction subroutine until enough clients have reached a "mature" state. If the newly added client ID and score causes the size of the listing of client IDs of mature networks to reach a predefined size, then a signal is sent to the GA thread of the server process. This signal will initiate a reproduction process, which will be discussed in more detail shortly.

If the client's number already appears in the listing stored by the server, the score for that client is simply updated based on the fitness score received from that client. In both cases (if the client was already in the listing or not), the new score received from the client is compared with the current best score maintained by the server

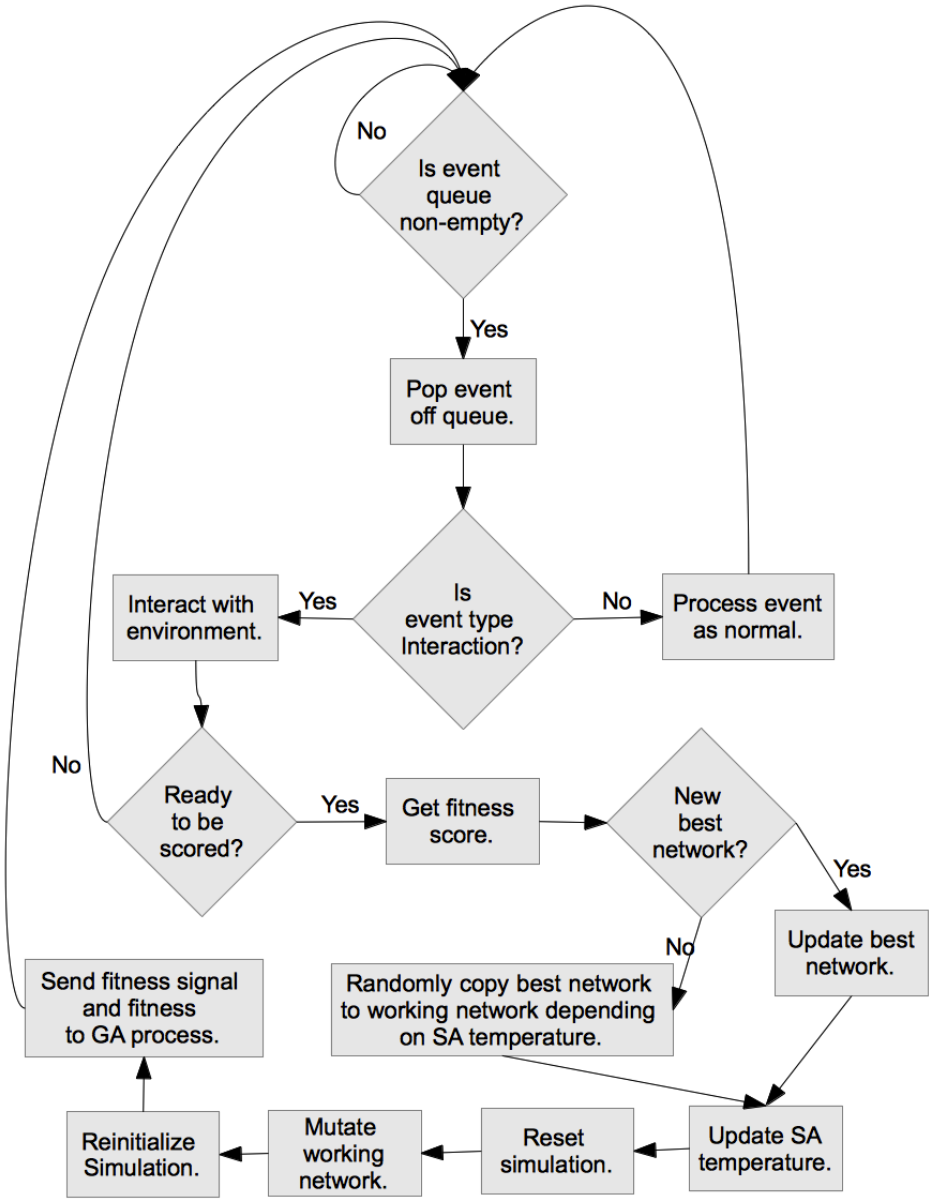


Figure 3.12: Flow chart for the simulation thread of the client process.

process. If the new score is higher than the current best score, the server process requests the current best network from the client process. This request is processed by the communication thread in the client process, which will be discussed shortly.

The server process has another type of thread that is executing in parallel with the client-communication threads and the initialization thread. This thread performs the selection and reproduction operations associated with the evolutionary optimization or genetic algorithms (GA) and as such, is called the GA thread. The GA thread is created by the initialization thread of the server process. However, the GA thread remains inactive until enough "mature" networks have been received by the client threads in the server process. The GA thread "sleeps" until it receives the signal that this threshold has been reached (this signal is sent as part of the fitness subroutine, as discussed above). This signal wakes up the GA thread. Upon waking, the GA thread has access to the full list of client IDs and associated scores. Using these scores, the GA thread can use a selection algorithm such as tournament selection to select two parents.

At this point in execution the server process does not have access to the parent networks, as the networks are not stored in the server process. These parent networks are required to perform reproduction, so the GA thread sends requests to the client process for parents and waits (or sleeps) until both parents' networks have been received by the two client-communication threads that correspond to those parents. Once these parent networks have been received, crossover and mutation can be used to produce two child networks. The GA thread will again use the listing of client IDs and associated scores to select two lower performing clients/networks. The GA thread will send a deletion signal to the corresponding client process, along with the new child network to replace the old network. This deletion signal will be processed by the communication thread in the client process. At this point, the GA thread has completed one round of reproduction, so it will again wait for the size of the pool of mature networks to reach a certain threshold before repeating the reproduction process (Figure 3.14).

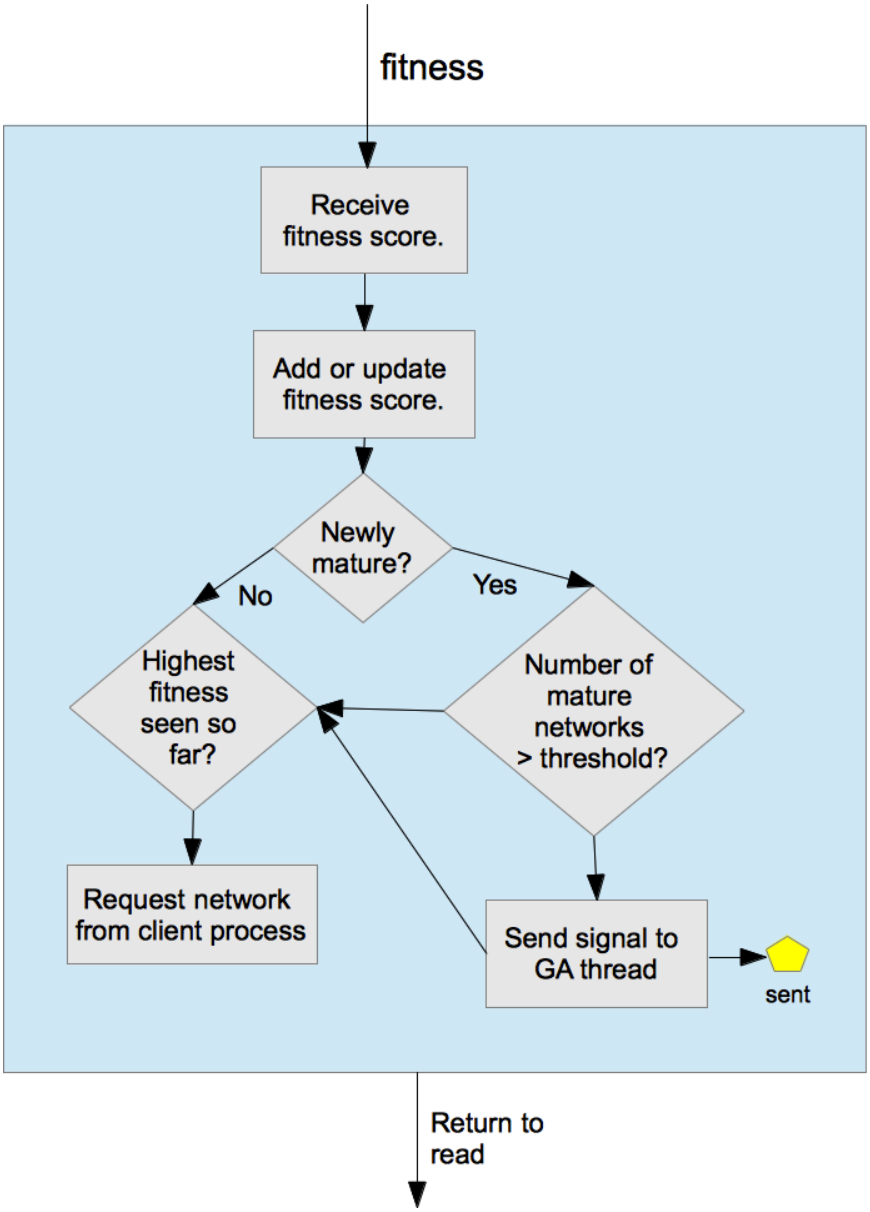


Figure 3.13: Flow chart for the fitness subroutine of the client thread of the server process.

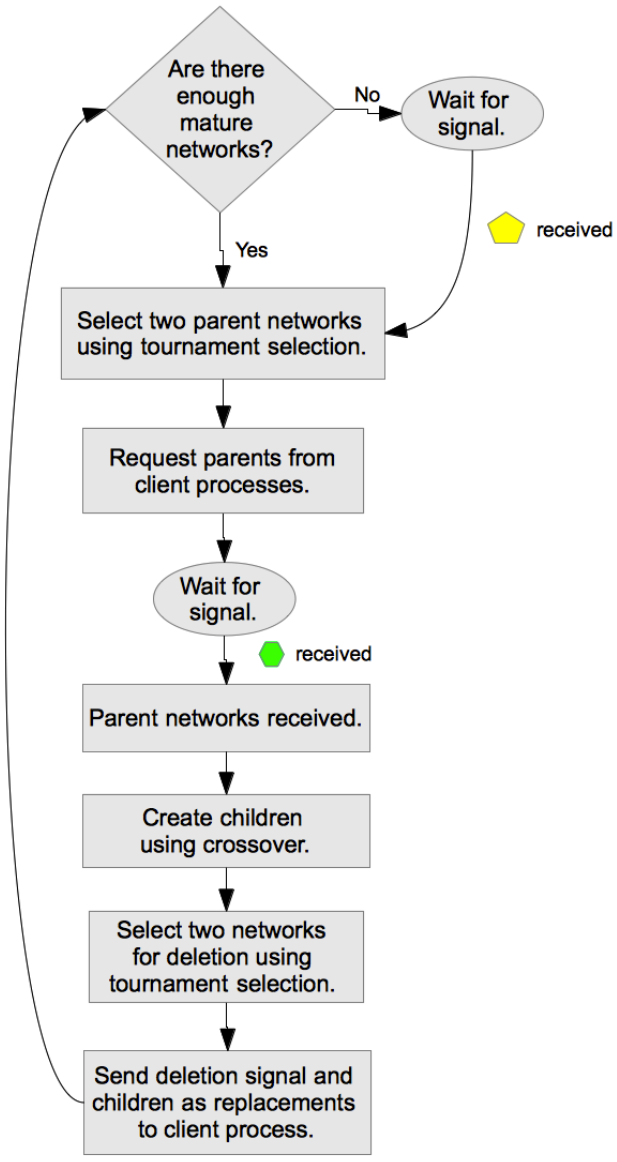


Figure 3.14: Flow chart for the GA thread of the server process.

We will now discuss the communication thread in the client process. As noted above, the communication thread in the client process waits for prompts from the server process. In the previous paragraphs, we have seen that the various types of prompts that come from the server to a client process occur when: (1) the client process's network has been selected to be a parent network by the GA thread in the server process, (2) the client process's network is the current best performing network across all clients, or (3) the client process's network has been selected for deletion. In cases (1) and (2), the communication thread in the client process simply sends the network requested to the server process to be used as a parent or to replace the current best network overall. When the client process's network is selected for deletion the client process stops simulation of its working network and deletes its current best network and its working network. One of the new child networks that has been created by the GA thread in the server process is sent to the client process to replace its old working network. Since the network is entirely new, a new simulation is created by creating new initial input events and an initial interact-with-environment event. A flow chart for the operation of the communication thread in each client process is given in Figure 3.15.

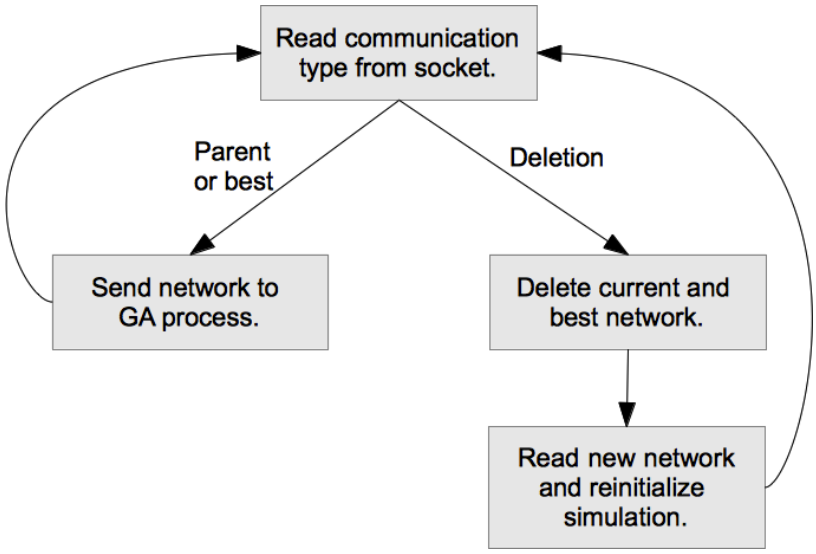


Figure 3.15: Flow chart for the communication thread of the client process.

Now, the remaining operations of the client-communication thread (Figure 3.10) in the server process can be enumerated. We have already seen that one possible mode received from the client is the "fitness" type. Two of the other possible modes from the client are "best" and "parent." Best mode from the client is received when the client process has been prompted to send its network to the server process because the client's network is the current best performing network among all the client processes (as noted above). In this case, the network is read from the client, and the best network is replaced in the server process. The client sends the "parent" mode when the client has been selected to be a parent by the GA thread in the server process. In this case, the parent subroutine is entered (Figure 3.16). In this subroutine, the network that is to be used as a parent is received from the client process. If both parents have been received, the client-communication thread signals the GA thread to continue the reproduction process. Otherwise, the parent subroutine returns to the client-communication thread and waits for another mode to be read from its corresponding client process.

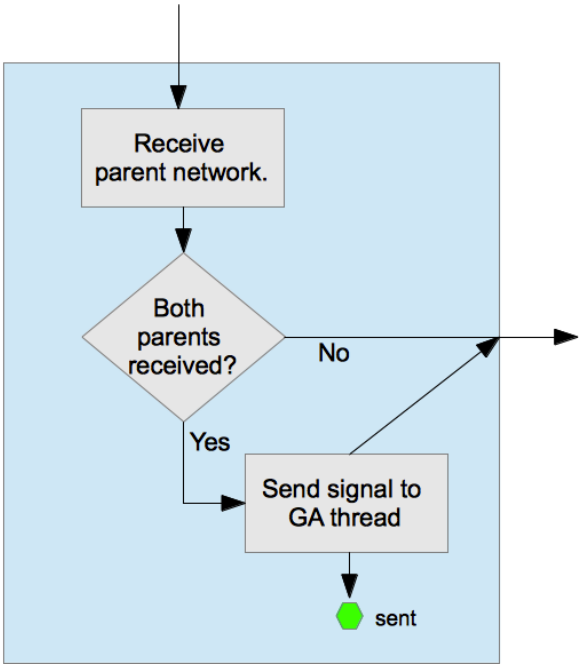


Figure 3.16: Flow chart for the parent subroutine of the client thread of the server process.

The final mode type that may be received from a client process in the client-communication thread in the server is that the corresponding client process is dying. The client process may be running on a cluster in which each process has an enforced time limit. In this case, client processes will intermittently die due to time constraints. To deal with this constraint, client processes have "alarms" so that the process is alerted shortly before it is to be killed. When the alarm is received, the client process enters its dying phase. During the dying phase, the client process sends the dying mode to the server process and sends its current best network to the server process before dying. The server process receives this network from the dying process and adds the corresponding network to the pending queue, so that a newly created client process can receive that network later as its working network. After adding the network to the pending queue, the server process launches a new client process to replace the old client process. As before, this can be done using an intermediate script that the server process launches.

The final case of the client-communication thread is if the client process fails for some other unknown reason. In this case, the server process is unable to read from that client process, so the client-communication thread removes all traces of that client's existence in the server process and exits the client-communication thread.

An example of GAv2 is shown in Figure 3.17, in which there are twelve clients and the threshold of mature networks required by the server process to reproduce is ten. This approach has a scaling limit in the number of threads allowed and the number of socket connections permitted per machine. This approach goes to one extreme of parallel processing in that each client process only maintains a single best network and does not contain any other members of a population. This may be required for certain tasks, such as those with large scale simulations or those that may require too much memory to maintain more than one network at a time. However, many tasks will not exceed this limit, so this approach, while available to the user, is not the best use of a large scale cluster for tasks with less complicated simulations.

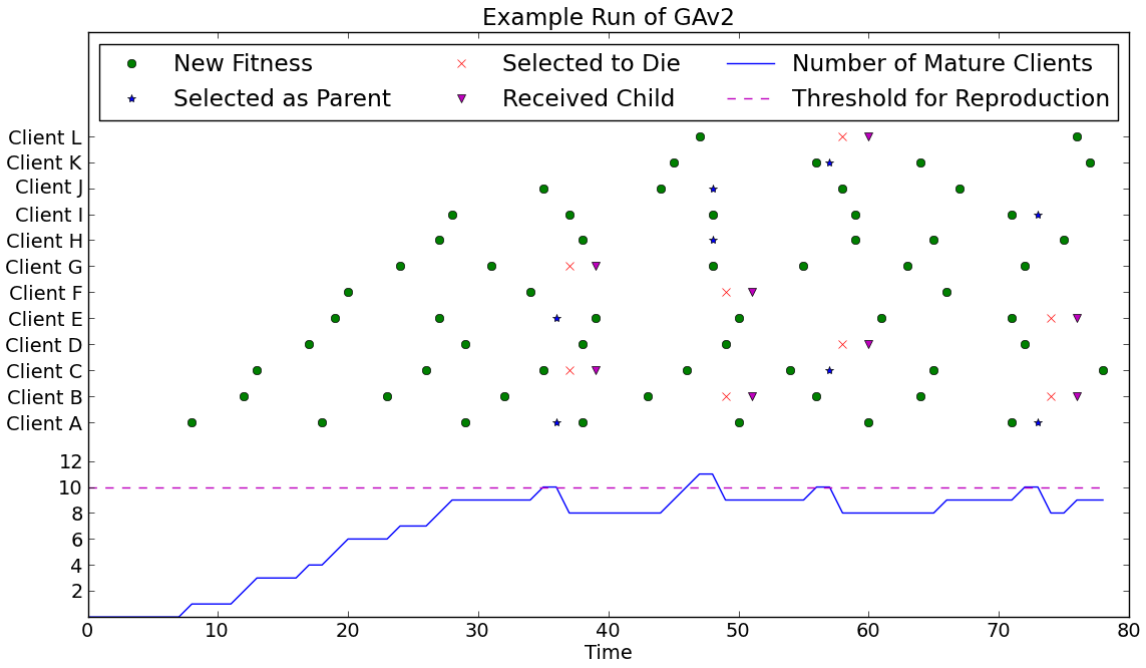


Figure 3.17: Example of a run of GAv2 in which there are 12 client processes. The server process has a threshold of mature networks set to 10. It is noted when each client process sends a new fitness value to the server process, when that process is selected for deletion, when it is selected to be a parent, and when it receives a new child network. The current number of mature networks in the server process is shown with a blue line, and the threshold of mature networks required to initiate the reproduction process is shown as well.

GAv3: Mini-Populations per Process

This implementation is a compromise between the two extremes of the sequential version and one network per process multi-process version (GAv2). For this implementation, there are again one server process and many client processes. Each client process maintains a population, rather than a single network.

Upon initiation of a client process, the client process establishes a connection (via a socket) to the server process. The server process sends some number of networks (possibly zero) to that client process and severs the connection to the client. The client process includes the networks it receives from the server in its initial population and fills out the remainder of the population (whose size may be determined by the user) with randomly initialized networks. The client process performs the sequential evolutionary optimization method (described in Section 3.2.5) based on this population and continues operating until either a performance threshold is reached (which is task specific) or until its time or number of epochs has exceeded a pre-defined threshold. When one of these thresholds is met, the client process initiates another connection with the server process and sends its current best network to the server, along with the score for that network. An alternative is to send multiple highly ranked networks, as determined by a fitness function. Initial client processes are launched independently of the server process; each client process may initiate another client process when its time limit expires or a number of epochs is exceeded. If the client process ends because the performance threshold has been reached, the client process sends a "completed" signal to the server process along with the best network (or, alternately, a set of highly ranked networks).

The server consists of a thread that accepts connections from clients and creates new client processing threads each time a connection occurs. The client processing threads in the server enter one of three modes based on the signal sent from the client: sending mode, receiving mode, and completed mode (see Figure 3.18). In sending mode, the client has requested that the server send its networks. In this

mode, the server sends all of the networks it has maintained to the client and then severs the connection to the client. In receiving mode, the server receives a network (alternatively, multiple networks) and a score from the client. The server stores the networks it receives from its clients and their associated scores. The server maintains a fixed number of networks and stores only the best performing networks that it receives. Thus, each time a new network and score is received from a client, the server process stores that network and score. If the number of networks stored exceeds a user-defined threshold, the server process removes the lowest performing networks until the number of networks is at or below the user-defined threshold. In completed mode, the server receives the completed signal from the client, prints the completed network and exits. The active clients for that server will exit when they fail to write to the server, and queued client jobs that have not yet started will exit when they fail to connect to the server after a fixed number of attempts.

This approach allows for the evolutionary optimization to maintain a much larger total population size by maintaining many mini-populations. The best networks from each mini-population are used to create other mini-populations. This also offloads the evolutionary optimizations to the clients rather than placing that burden on the server, which allows the server to be much less complicated and task-independent.

3.3 Results

In this section, we demonstrate the utility of NIDA networks and the associated design method based on evolutionary optimization on tasks from three domains: anomaly detection, control, and classification. One of the major goals of the architecture and design method is to make it as general as possible so that it may solve varying problem types without any major alterations to the way the network or optimization algorithm operates.

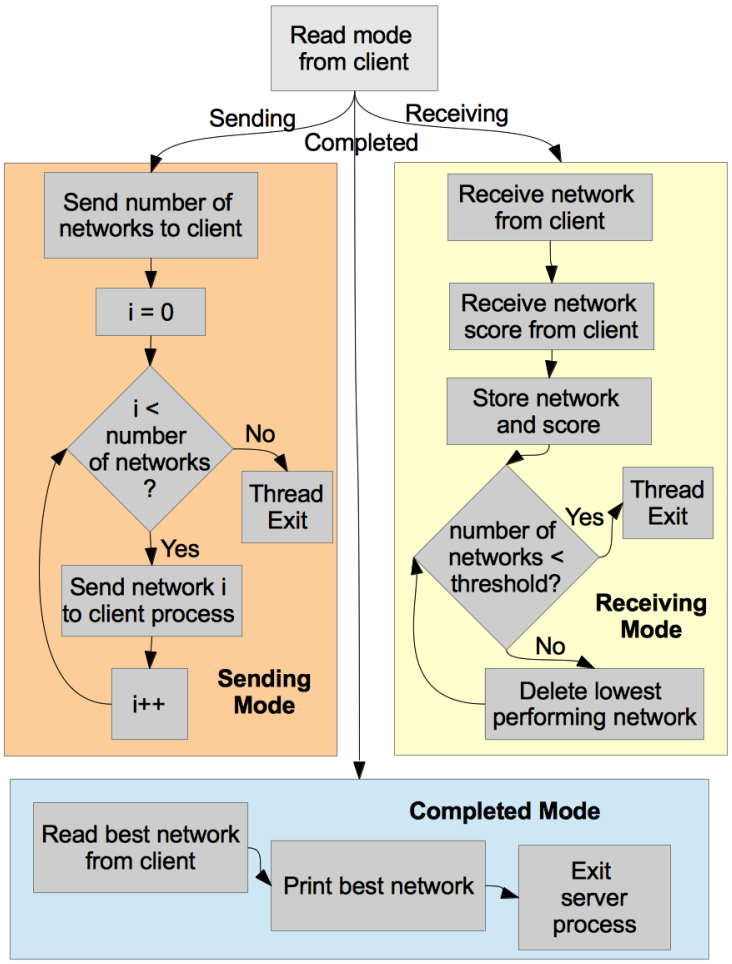


Figure 3.18: Flow chart for the client thread of the server process, which enters one of three modes, the operations of which are encapsulated in colored boxes: sending (orange), receiving (yellow), or completed (blue).

3.3.1 Anomaly Detection: Network Traffic

The description of the problem and many of the results in this section come directly from [228].

Task Description

The problem of interest relates to an application in cyber security, which is the observation of packet arrivals at a node in a packet–switched communication network and detection of changes in the statistics of the arrival rate. These statistics are typically not known *a priori* and would require estimation in a conventional detection scheme and assumptions about the underlying process that governs the statistics. It is realistic to assume the statistics are piecewise constant but unknown, with a distribution on a finite interval of real-valued arrival rates, with jump discontinuities. The distribution of time intervals between jumps is typically not known. This corresponds to a mix of software applications that each generates network traffic at a more or less constant rate that is destined for a monitored network address. Some applications may be malware, in which case an objective is detection of the start and end of packet streams produced by the malware.

This scenario can be modeled as an observation of a discrete–event process that can be characterized by its arrival times $\{t_k | t_0 = 0, t_{k+1} > t_k, k \in I^+\}$. This process can be represented as a discrete time real–valued random process $\{x_k = t_k - t_{k-1} | k \in I^+ - \{0\}\}$, where the x_k are in $\mathbb{R}^+$ and $x_k \neq 0$. The x_k are the time intervals between event arrivals and, with the additional knowledge of the time of the first event, t_0, fully characterize the discrete–event process. A well–known statistical detection problem assumes that x_k is a random process sampled from one of two known distributions, characterized by probability spaces $(\Omega_i, \Lambda_i, P_i)$, for $i = 0, 1$. Optimal detectors are known that minimize a linear combination of (i) the probability of detection p_0, (ii) the false alarm probability p_1, and (iii) the expected time of detection (decision) $E\{T\}$. The optimal algorithm processes received events sequentially, and after each receipt

decides (i) not to make a decision until additional information is received, or (ii) that the inputs correspond to process 0 or 1. In the second case, the algorithm outputs the determined process type and stops. A slightly more challenging problem assumes that x_k is a random process whose statistics can change from sample time to sample time between the two probability spaces. In both cases, the problem is well-defined in the field and has an optimal solution when the parameters of both distributions are completely specified ([197]). There are also algorithms for this problem when the second distribution has some unknown parameters ([144]).

Results

For the networks in this example, the LTP/LTD refractory period (the time between updates to a synapse's weight value caused by LTP and LTD) used was 100 time units, and the amount the synapses were changed in LTP/LTD was 0.001. For the evolutionary algorithm used in this example, we defined the population size to be 100, the crossover rate to be 0.9, and the mutation rate to be 0.99. The approximate probabilities defined for each mutation are given in Table 3.1.

Table 3.1: Approximate probabilities for each mutation type for the anomaly detection problem.

Mutation	Approximate Probability
Change sign of a synapse	0.3
Change the value of the weight of a synapse	0.3
Add a synapse	0.15
Remove a synapse	0.15
Add a neuron	0.03
Delete a neuron	0.03
Alter threshold	0.03

In our setup, the network to be designed has one input neuron and one output neuron. The network receives a pulse each time a packet arrives. Firing of the output neuron corresponds to a change in behavior. We allow for a window of 100 time steps after the change in behavior. We also define a threshold value, τ, that determines how

Table 3.2: Estimated probability of detection (P_d), probability of false alarm (P_{fa}), and probability of missed detection (P_m) of increases and decreases for N_+ and N_-, respectively, for three different test types (large, medium, and small changes in mean arrival rate).

	Large			Medium			Small		
Network	P_d	P_{fa}	P_m	P_d	P_{fa}	P_m	P_d	P_{fa}	P_m
N_+	0.90	0.10	0.02	0.82	0.18	0.08	0.78	0.22	0.35
N_-	0.87	0.13	0.01	0.87	0.13	0.14	0.78	0.22	0.45

many output firings constitute a detection. If the mean arrival rate changes at time t, then τ firings of the output neuron at any point between t and $t+100$ is considered a true positive. If the output node fires τ times in a 100 time step window at any other time than 100 time steps following a change in mean arrival rate, it is categorized as a false alarm. For training, $\tau = 1$ is used, and the fitness of the network is a function of the the number of correct detections and false alarms.

We ran our training algorithm for 10000 epochs. The results shown below were produced by two networks, N_+ (65 neurons and 187 synapses) and N_- (47 neurons and 148 synapses), that were able to detect, respectively, increases (+) and a decreases (–) in the mean arrival rate. We ran three types of tests: tests with large changes in the mean arrival rate (changes of at least 0.38), medium changes in the mean arrival rate (changes of at least 0.2) and small changes in the mean arrival rate (changes of at least 0.1). We estimated the probability of detection (P_d) of increases (decreases), the probability of false alarms (P_{fa}) for increases (decreases), and the probability of missed detection (P_m) of increases (decreases) by the frequency of detection and missed detection events over 100 test runs for both N_+ and N_-. P_d is the probability that a given event constitutes a true detection, P_{fa} is the probability that a given event is false alarm, and P_m is the probability that a change in mean arrival rate occurred, but the detector did not detect it. All runs (training and evaluation) utilized independently generated random input event sequences. The results are shown in

Table 3.2. Figure 3.19 shows the changes in mean arrival rate of one example test run, as well as when N_+ and N_- fired in that test run.

Our networks performed well when detecting large changes in the mean arrival rate, but the performance decreased as the size of the change in the arrival rate decreased. This is expected and consistent with the behavior of optimal detectors for problems where they are known. Performance decreases as the region of overlap increases between the probability functions of the event observables conditioned upon the hypotheses.

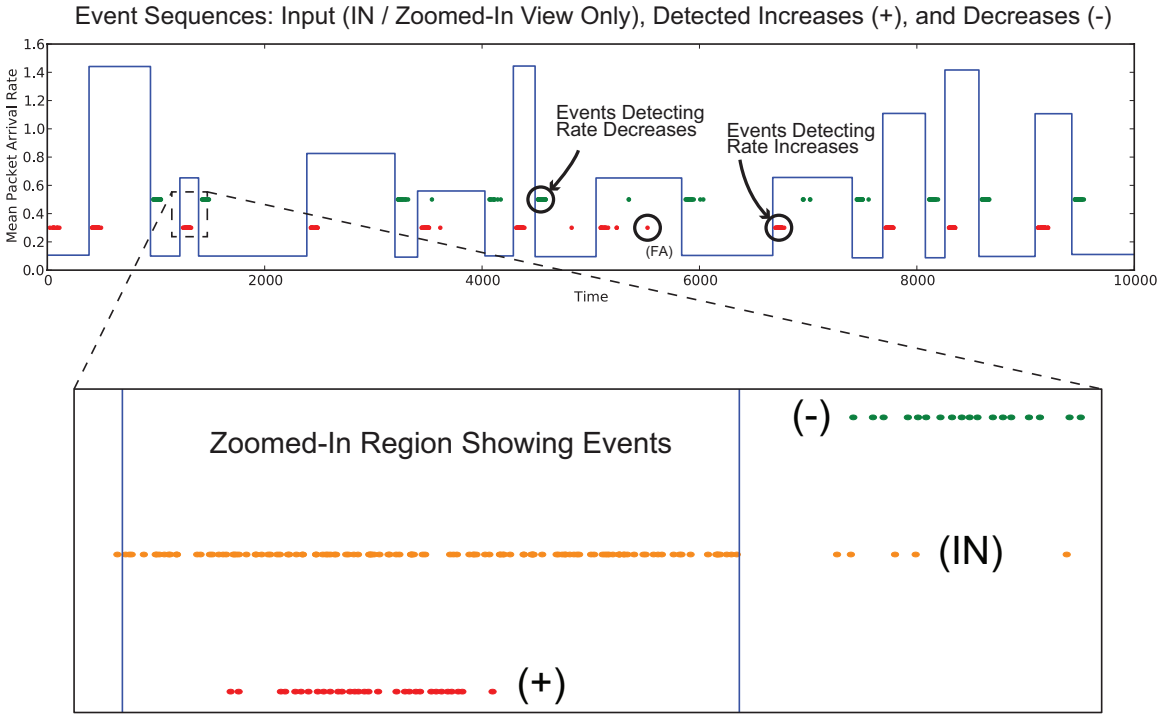


Figure 3.19: Simulation results with each detection network. The mean arrival rate of the test run is shown as the blue line. The output firing events for N_+ and N_- are shown in red (+) and green (–). The events are discrete, and only their times are meaningful (see zoomed–in region showing both inputs and outputs); their vertical placement is for convenience. Example false alarm events (FA) are indicated.

A simple fitness function was used: the difference between the numbers of correct detections and false alarms. The fitness function favored networks that could detect any change in mean activity rate, but all of our training examples produced networks that detected either positive or negative changes, but not both. The networks that

were produced had many recurrent connections, as would be expected for this type of problem.

Comparison with Optimal Detector

The performance characteristics of the NIDA networks can be compared against the performance of an optimal detector for a simplified problem where the solution is known. We consider a classic example where the input event process has a constant mean arrival rate λ that is one of two values $\{\lambda_0, \lambda_1\}$ and is observed over a time interval ΔT. We assume $\lambda_1 > \lambda_0$. The optimal probabilities of detection and error, suitably defined, can be computed without regard to the observed sequence of events in this case.

The number of received events n is a Poisson random variable with distribution

$$p(n) = \frac{(\lambda \Delta T)^n}{n!} e^{-\lambda \Delta T} \tag{3.1}$$

where $\lambda \Delta T$ is the mean number of observed events in the time interval. The problem is to decide which of two hypotheses is correct: $\{H_0 : \lambda = \lambda_0\}$ or $\{H_1 : \lambda = \lambda_1\}$. Assuming the *a priori* probability of hypothesis H_0 is 0.5 and the costs assigned to correct (detection) and incorrect (false alarm or failure to detect) identification of the true hypothesis are equal, the optimal decision rule is, given an observed number of events n in the time interval, and defining the function

$$f(\lambda_0, \lambda_1) = \frac{(\lambda_1 - \lambda_0)\, \Delta T}{\ln \lambda_1 - \ln \lambda_0}, \tag{3.2}$$

$$h = \begin{cases} H_1 \text{ if } n > f(\lambda_0, \lambda_1) \\ H_0 \text{ if } n < f(\lambda_0, \lambda_1) \end{cases} \tag{3.3}$$

with no (or random) choice in the case of equality [254]. The probability of detection (correct classification) is the sum of the probabilities that H_0 is true and the number of observed events is less than $f(\lambda_0, \lambda_1)$, and that H_1 is true and the number of

observed events in greater than $f(\lambda_0, \lambda_1)$ (assuming the function's value is not an integer):

$$p_d = \sum_{\substack{n \geq 0 \\ n < f(\lambda_0, \lambda_1)}} \frac{(\lambda_0 \Delta T)^n}{n!} e^{-\lambda_0 \Delta T} + \sum_{n > f(\lambda_0, \lambda_1)} \frac{(\lambda_1 \Delta T)^n}{n!} e^{-\lambda_1 \Delta T} \tag{3.4}$$

The probability of error is expressed similarly:

$$p_e = \sum_{\substack{n \geq 0 \\ n < f(\lambda_0, \lambda_1)}} \frac{(\lambda_1 \Delta T)^n}{n!} e^{-\lambda_1 \Delta T} + \sum_{n > f(\lambda_0, \lambda_1)} \frac{(\lambda_0 \Delta T)^n}{n!} e^{-\lambda_0 \Delta T} \tag{3.5}$$

This classic detector is predicated upon the assumptions that either hypothesis H_0 or H_1 is valid for the duration of the time interval, that *a priori* statistics are known, and that costs can be assigned. When one of the hypotheses is not valid for the entire interval, as is the case for the application of interest, the mathematics become more challenging. One approach is the assumption of a Markov process that generates the arrival statistic as a function of time, and the methods of quickest detection, discussed previously, can be applied in some cases.

An alternative approach using Neyman–Pearson detectors [254], which compare a computed likelihood ratio against a threshold, is used here to explore how the probability of detection changes with a constraint on the maximum allowed probability of error, expressed graphically as receiver operating characteristic (ROC) curves. If the probabilities of observation of a signal S given hypotheses H_0 and H_1, $p(S|H_0)$ and $p(S|H_1)$, are known, the likelihood ratio (LR)

$$\Lambda(S) = \frac{p(S|H_1)}{p(S|H_0)} \tag{3.6}$$

can be compared against a threshold η determined by the solution of a constrained optimization problem, yielding a decision that H_1 is true if the LR $\Lambda(S) > \eta$ and that H_0 is true if it is less. Figure 3.20 shows representative ROC curves for Neyman–Pearson detectors and a NIDA network detector for different values of λ_0 and λ_1. The Neyman–Pearson optimal detector, which is a function of the maximum

allowed probability of error, is used to generate the curves on the left for each pair of mean arrival rates. In contrast, the same NIDA network detector structure is used for all pairs on the right. In order to evaluate the NIDA network detector in a similar manner, the detector's output events within intervals $[t - \Delta t, t]$ are counted and compared against a threshold. A detection at time t corresponds to the count exceeding the threshold at that time, and frequencies of detection and error are computed for a range of thresholds and graphs. The salient point here is the

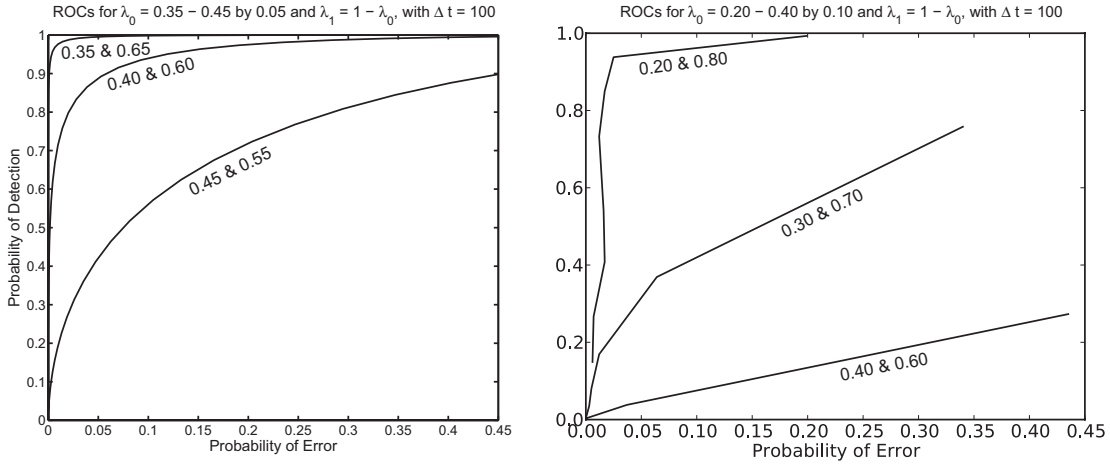


Figure 3.20: ROC curves for Neyman–Pearson optimal detectors (left) and a NIDA network that detects increases in mean arrival rate. Note that the optimal detectors assume equal *a priori* probability of the hypotheses and arriving events are from one of two processes having the indicated mean arrival rates, while the NIDA network detector was trained to detect *any* increase in arrival rate within a specified range and made no assumptions about *a priori* statistics.

NIDA network detector is providing a (probably suboptimal) solution to a much more challenging detection problem than can be solved mathematically, where the statistics of the underlying processes are not known and must be learned (along with the solution) by observing the input event sequence. The learning problem is supervised, as an oracle is assumed that allows evaluation of the fitness function during training, but this is not sufficient to drive an optimal detector. It is sufficient at this point to recognize that the NIDA network detector's performance has similar behavior to an (over-simplified) optimal detector, exhibiting increasing detection probability with

increasing allowable probability of error, and increasing probability of detection with an increasing difference in mean arrival rate of the events. The two types of detectors are qualitatively, but not quantitatively, comparable.

Different Fitness Functions

The fitness function used for this application when training was the numerical difference between the number of true positives and the number of false alarms. The networks described in the previous sections were trained using random step changes, and the fitness function checked to see if the network was detecting both positive and negative changes. For comparison, we tried training with two alternative fitness functions:

- One that had random step changes in mean arrival rate, but only checked for positive changes.
- One that had fixed-sized step changes in mean arrival rate, but checked for both positive and negative changes.

Networks were trained using these fitness functions for 10000 epochs. We applied the same three tests from the previous sections (large positive steps, medium positive steps, and small positive steps in mean arrival rate), and measured probability of detection (P_d), probability of false alarm (P_{fa}), and the probability of a missed detection (P_m). Figure 3.21 shows networks that performed well for each of the three fitness test cases.

This figure shows that networks that performed reasonably well emerged from all three test cases. It also shows that this problem is fairly resilient to the type of training completed. Of particular note is the network that performs well on random positive step changes, despite only being trained to recognize fixed step changes. This result demonstrates that this network structure can generalize learned results.

Another interesting note about this problem is that, out of 20 networks we trained to detect both positive and negative step changes, nineteen detected only positive

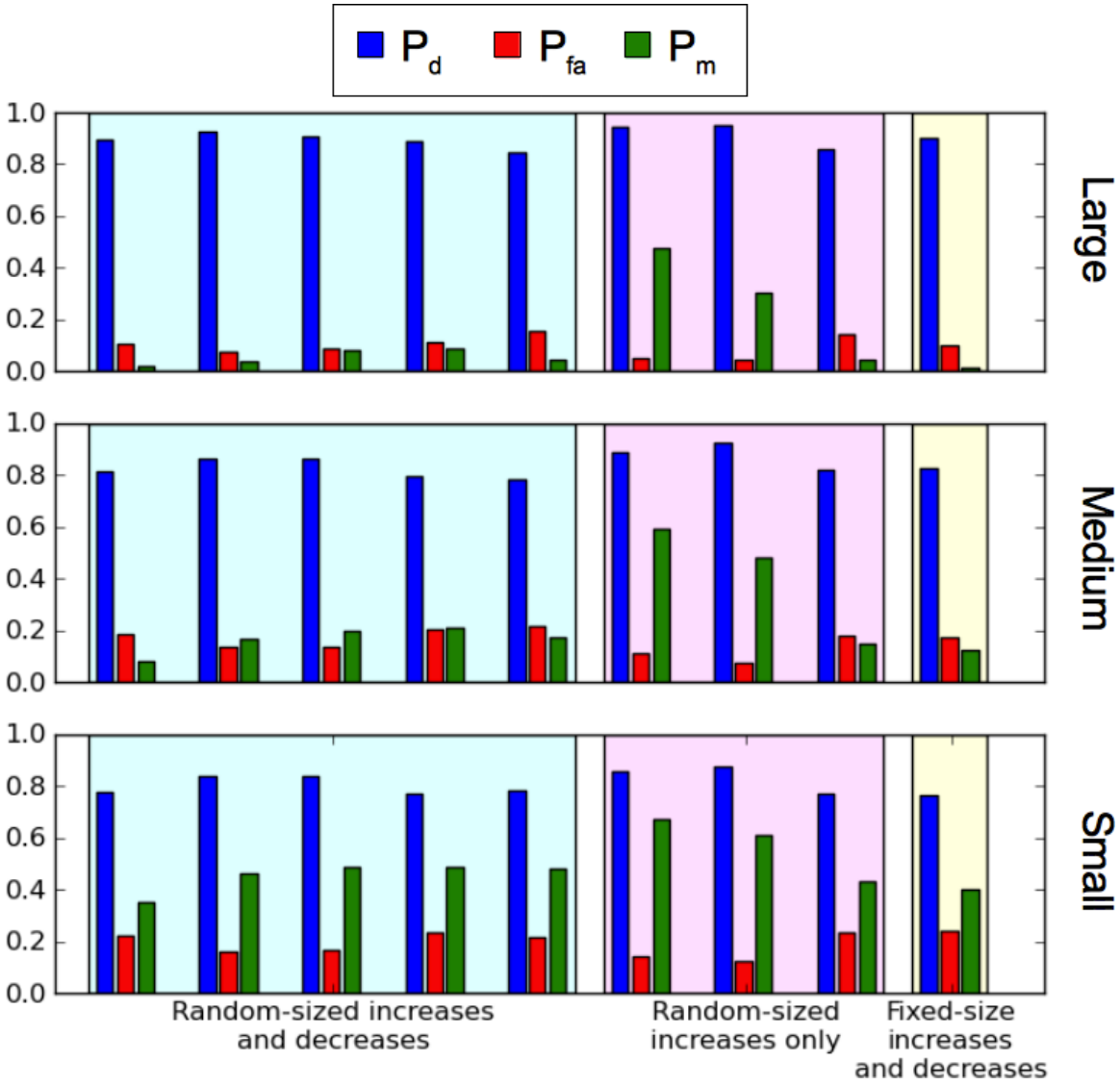


Figure 3.21: Probability of detection (P_d), probability of false alarm (P_{fa}), and the probability of a missed detection (P_m) results for nine different networks that performed reasonably well on detecting positive changes for the three test cases: large, medium, and small positive changes in mean arrival rate. The networks are grouped based on the fitness function they were trained under.

step changes and one detected only negative step changes. It is clear the we could construct a network that detects both by combining one network that detects positive changes and one that detects negative changes, so there exists at least one network that detects both types of changes (this combined network). It appears that in the space of network structures that perform well on this problem, there are many more locally optimal networks that detect positive changes than negative changes, and many more locally optimal networks that detect one or the other than detect both.

Conclusions

This section demonstrates that our network structure can detect anomalies in the input from the environment by demonstrating that the network can detect changes in the mean arrival rate of pulses to the network. This problem requires time-dependent information, so the recurrent connections established in our network are necessary. Furthermore, it requires long-term information about the environment (e.g., when the change occurred) to remain in the system. We also demonstrated that the network structure can generalize well, by training a network using a specific type of change in mean arrival rate (a switch between two set values) and testing on random changes.

The fitness functions used in training were very simple; more complex fitness functions could be used to enhance performance. For example, one of the values we measured in testing our networks was P_m, the probability of a missed detection. This value was not considered when training the networks. To produce networks with fewer missed detections, we could develop a fitness function that includes this value by penalizing networks for missing detections. This fitness function may be explored in future work.

3.3.2 Control: Pole Balancing

In this version of the pole balancing problem, a cart is assumed to be on a track so that it can move in only one dimension; that is, the cart can only be moved left

or right. The track is assumed to be finite in that the cart must be kept between two points on the track. Attached to the cart is a pole. The pole is initially in some upright position. The goal of the pole balancing problem is to apply forces to the cart in order to keep the pole balanced and to keep the cart between the two endpoints of the track. The pole balancing problem is discussed in detail (along with the equations described below) in [9, 80].

The state of the pole balancing problem is described by four variables:

- x = the position of the center of the cart on the track in meters.
- $\dot{x}$ = the velocity of the cart in meters/sec.
- θ = the angle of the pole from vertical in radians.
- $\dot{\theta}$ = the angular velocity of the pole in radians/sec.

Only two control actions were considered for this problem, a negative force or a positive force of equal magnitude applied to the cart in a horizontal direction. This is referred to as the bang-bang control version of the inverted pendulum or pole-balancing problem. A diagram of the environment with the various state variables labeled is shown in Figure 3.22.

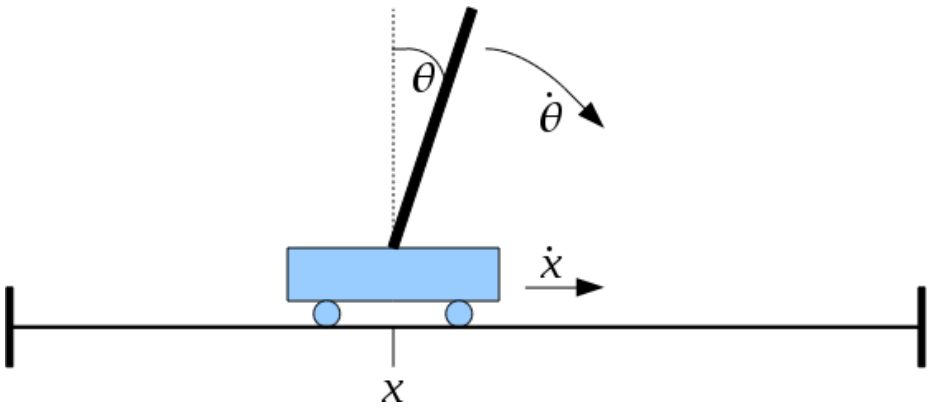


Figure 3.22: A diagram of the environment with the state variables labeled (not drawn to scale).

Table 3.3: Parameter values used in single pole balancing application.

Parameter	Value
Mass of cart m_c	1 kg
Mass of pole m_p	0.1 kg
Cart length	1 m
Pole length l	1 m
Track length	5.8 m

Table 3.3 gives the parameter values used in the single pole balancing simulation. The track ranges from -2.9 meters to 2.9 meters, so the maximum value of the x variable is 2.4 and the minimum value of the x variable is -2.4. The two actions are $-10N$ and $10N$. In this work, a force will be applied and the state will be updated every 0.02 seconds.

Given x, $\dot{x}$, θ, $\dot{\theta}$ and the force applied, the new state can be determined with a set of equations. If $\dot{\theta} = \omega$, the following equation ([9]) can be used to determine the angular acceleration, where F is the force, g is the gravitational constant ($9.8m/sec^2$):

$$\dot{\omega} = \frac{g\sin(\theta) + \cos(\theta)\left(\frac{-F - m_p l\dot{\theta}^2\sin(\theta)}{m_c + m_p}\right)}{l\left(\frac{4}{3} - \frac{m_p\cos^2(\theta)}{m_c + m_p}\right)} \tag{3.7}$$

Then, if $\dot{x} = v$, the acceleration of the cart can be determined with the following equation ([9]):

$$\dot{v} = \frac{F + m_p l[\dot{\theta}^2\sin(\theta) - \dot{\omega}\cos(\theta)]}{m_c + m_p} \tag{3.8}$$

The dynamic behavior of the cart and pole system is approximated using Euler's first-order numerical integration rule. Using this rule, the new cart position can be approximated using the following equation, where $\tau = 0.02$:

$$x(t + \tau) = x(t) + \tau\dot{x}(t) \tag{3.9}$$

The new angle of the pole can be determined with the following equation:

$$\theta(t+\tau) = \theta(t) + \tau\dot{\theta}(t) \tag{3.10}$$

Then, the new velocity of the cart, $v = \dot{x}$, can be determined with the following equation:

$$\dot{x}(t+\tau) = v(t+\tau) = v(t) + \tau\dot{v}(t) \tag{3.11}$$

The new angular velocity of the cart, $\omega = \dot{\theta}$, can be determined with the following equation:

$$\dot{\theta}(t+\tau) = \omega(t+\tau) = \omega(t) + \tau\dot{\omega}(t) \tag{3.12}$$

The ranges of possible values for each of these parameters is continuous, so there are infinitely many possible input values. Choosing how to encode the input in the system and how to interpret the output is a difficult problem for these types of networks. The output values for this problem are fairly straightforward, since there are three possible actions (apply a force of -10 N, apply a force of 10 N, and apply no force at all to the cart). In both examples, this is encoded using two output neurons. One corresponds to $-10N$ and the other corresponds to $10N$. The output neuron that fires the most is the chosen action. If neither of the output neurons fire in the desired window, then no force is applied.

In our networks, one way to encode continuous values is to relate those values to a frequency of pulse inputs. Another way to approach the problem is to discretize the possible input space by splitting the space into ranges of values and having one input neuron correspond to each range. Both of these approaches are applied in this work and are described in detail and compared below.

The fitness function used can also have a major effect on the convergence rate of the system. For this problem, networks that are able to balance the pole longer should be maintained. That is, the amount of time the network can balance the pole is a straightforward fitness function for this problem. However, ideally we would like for the system to be able to balance the pole from a variety of starting conditions

Table 3.4: Test cases used in f_2 for the single pole balancing application.

Test Case	x	$\dot{x}$	θ	$\dot{\theta}$
1	0	0	0.15	0
2	0	0	-0.15	0
3	1.2	0	0.15	0
4	1.2	0	-0.15	0
5	-1.2	0	0.15	0
6	-1.2	0	-0.15	0

(i.e., the pole in a various positions and the cart in various positions). Specifically, we would like for the cart to be able to start in any position between -1.2 and 1.2 meters and for the pole to start from -0.1045 to 0.1045 radians from vertical. The simplest possible fitness function would be to start in one state, measure how long the network is able to keep the pole balanced, and return that time as the fitness. However, this does not measure the network's ability to generalize. That is, a network that performs well starting from one state may perform horribly when starting from another state. To overcome this, a set of state values can be used for each fitness test, and an average over their balancing times should be taken.

From this observation, we defined two fitness functions. One fitness function chooses ten random initial x and θ values (where $x \in [-1.2, 1.2]$ and $\theta \in [-0.1045, 0.1045]$), and $\dot{x} = \dot{\theta} = 0$.

In this way, the fitness function is different, not only for each network, but also for each epoch. This fitness function will be referred to as f_1. The second fitness function has six tests cases that are tried, given in Table 3.4. This fitness function is referred to as f_2. Results for both are given below.

Results - Single Pole - Frequency-Based Version

As noted above, one way to encode a continuous value of input is to have the values correspond to frequency of pulses on the input neuron. For example, a higher value of x may correspond to a higher frequency of pulses. In this problem, for all four variables (x, $\dot{x}$, θ, and $\dot{\theta}$), the ranges of possible values include positive and negative values.

As such, we have chosen to include two input neurons for each variable, resulting in eight total input neurons, where each variable has an input neuron corresponding to positive values and an input neuron corresponding to negative values. Then, for higher magnitudes of each value, more pulses are applied to the corresponding neuron.

Using this framework, for each set of values $(x, \dot{x}, \theta, \dot{\theta})$ at time step t, corresponding pulse frequencies are calculated. Specifically, the pulse frequencies used are the magnitudes of the values themselves. Lower (in magnitude) values of the $(x, \dot{x}, \theta, \dot{\theta})$ correspond to fewer pulses to the network. Lower (in magnitude) values of these variables also correspond to the network doing well in the simulation. Thus, the network is excited more when, for example, the cart is getting close to the track edge, or the pole is further away from vertical.

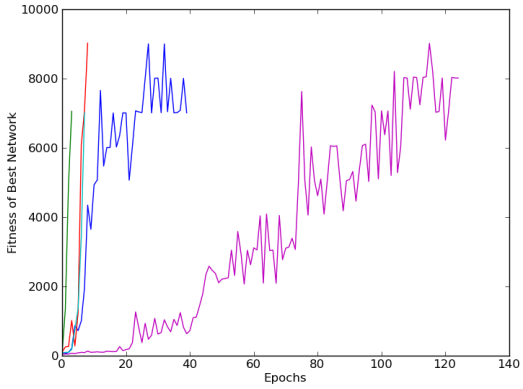
Pulses are applied at those frequencies to the corresponding input neurons for 200 units of simulated network time. At the end of these 200 time steps, the number of firings in that window for both output neurons is measured and the action that corresponds to the output neuron that fired the most is applied. The state of the system $(x, \dot{x}, \theta, \dot{\theta})$ is then updated based on the force applied, and the process is repeated until a failure state is reached (the cart hits the end of the track or the pole falls) or the pole is kept up and the cart is kept between the tracks for some predefined time period. For this experiment, we have defined this time period to be 10000 update steps, which corresponds to 200 seconds.

The population size for this test was 100, which allowed for fairly fast epoch evaluations. Larger population sizes could be used, but as the population size grows, so does the time per epoch or generation, resulting in potentially longer training times. The mutation and crossover rates were both set to 0.9. The approximate mutation probabilities are given in Table 3.5.

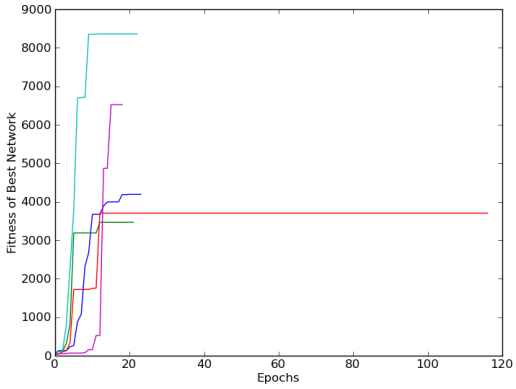
As a sample set, the results of five tests runs are shown below for both fitness functions f_1 and f_2. Some of these five networks are not fully trained; this can be seen for the networks in which the fitness function did not approach 10000. The results are given in Figure 3.23.

Table 3.5: Approximate probabilities for each mutation type for the pole balancing problem.

Mutation	Approximate Probability
Change sign of a synapse	0.278
Change the value of the weight of a synapse	0.278
Add a synapse	0.139
Remove a synapse	0.139
Add a neuron	0.069
Delete a neuron	0.069
Alter threshold	0.028



Fitness function f_1.



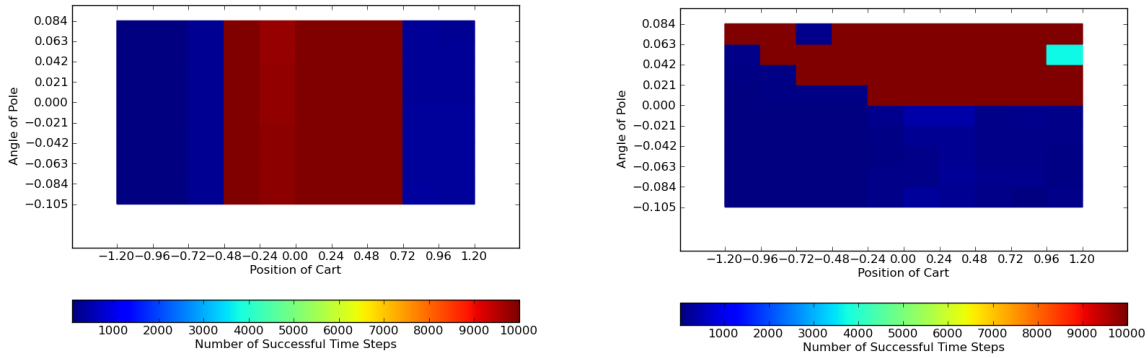
Fitness function f_2.

Figure 3.23: Fitness value (roughly the number of time steps the network can keep the pole balanced) vs. epoch for the best network in the population. Results are shown for five networks for each fitness function.

As can be noted in the graphs, for f_1, the best performing network's fitness can actually decrease across epochs; this is because random input positions are tried at each step. Fitnesses of networks trained using f_2, however, typically increase or stays the same as training continues. It can, however, stagnate at a particular value. It may be of interest to introduce more randomness into the evolutionary algorithm in order to prevent this stagnation.

The two best performing networks for f_1 and f_2 had comparable sizes. The best performing network trained using f_1 had 22 neurons and 85 synapses, while the best performing network using f_2 had 21 neurons and 106 synapses.

A high fitness measured using f_1 for a network does not necessarily indicate that that network performs well in general. To test how well the networks perform in general, 110 test cases were applied. For f_1, the networks that performed well typically fell into one of two categories: either it performed well for a subset of cart positions or it performed well for a subset of pole positions. Figure 3.24 shows two such networks, one of which performs well for positive θ values, and the other that performs well for small values of x.

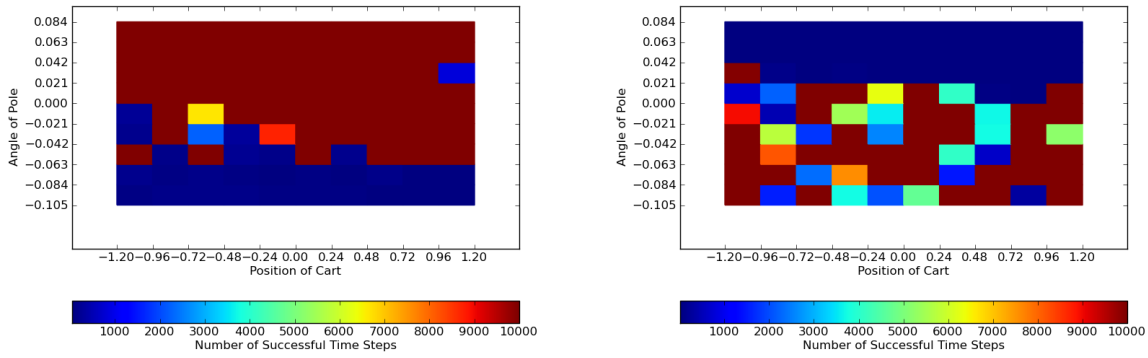


Performs well for small magnitudes of x.

Performs well for $\theta > 0$.

Figure 3.24: Results for two networks trained using f_1 when tested on 110 different start conditions.

Similar results occurred for f_2. Figure 3.25 shows the results for two networks, one of which does well just about everywhere except negative values of θ, and the other that is beginning to do well everywhere except for positive values of θ.



Performs well for $\theta > 0$. Performs well for $\theta < 0$.

Figure 3.25: Results for two networks trained using f_2 when tested on 110 different start conditions.

The stopping conditions of 10000 time steps is a relatively small amount of time for the pole to be balanced (it corresponds to 200 seconds or three minutes and 20 second). A second test of performance of these networks is whether they can continue to balance the pole for a longer time period. Instead of placing a bound of 10000 time steps, we now place a bound of 100000 time steps (2000 seconds or 33 minutes and 20 seconds), and test the networks on the same six starting conditions that are used in f_2. Table 3.6 shows the results for a network trained using f_1 and a network trained using f_2.

By this measure, the network produced using f_2 does much better. However, networks trained using f_2 are trained specifically to perform well on these starting points, so this is not a surprise.

Results - Single Pole - Discretized Version

Another option for encoding continuous valued input is to discretize the possible input space by splitting the space into ranges of values and having one input correspond to

Table 3.6: Number of time steps two networks are able to continue balancing the pole. The cut-off point for these networks was 100000 time steps.

Test Case	f_1	f_2
$x = 0$, $\theta = 0.15$	71007	100000
$x = 0$, $\theta = -0.15$	41	100000
$x = 1.2$, $\theta = 0.15$	61932	65581
$x = 1.2$, $\theta = -0.15$	118	100000
$x = 1.2$, $\theta = 0.15$	74	100000
$x = 1.2$, $\theta = -0.15$	21	158

each range. To accomplish this, each of the four state space variables were split into three ranges, as described in Table 3.7.

Table 3.7: Input Encoding Ranges. Each of the four state space variables were split into three ranges. There is a corresponding input neuron for each of these ranges. When the state space is updated (or set initially), the range that each value belongs to is calculated, resulting in four ranges. Then, a pulse is applied to each of the four neurons.

Parameter	Ranges for Encoding
x	$(-\infty, -0.8], (-0.8, 0.8], (0.8, \infty)$
$\dot{x}$	$(-\infty, -0.66], (-0.66, 0.66], (0.66, \infty)$
θ	$(-\infty, -0.0697], (-0.0697, 0.0697], [0.0697, \infty)$
$\dot{\theta}$	$(\infty, -0.667], (-0.667, 0.667], (0.667, \infty)$

There is a corresponding input neuron for each of these ranges. When the state space is updated (or set initially), the range that each value belongs to is calculated, resulting in four ranges, one for each parameter value. Then, a single pulse is applied to each of the four neurons corresponding to their ranges. The network is then simulated for 200 units of simulated network time. As in the frequency based version, at the end of these 200 time steps, the number of firings in that window for both output neurons is measured and the action that corresponds to the output neuron that fired most is applied. The state of the system $(x, \dot{x}, \theta, \dot{\theta})$ is then updated based on the force applied and the process is repeated until a failure state is reached (the

cart hits the end of the track or the pole falls) or the pole is kept up and the cart is kept between the tracks for some predefined time period.

The population size for this test was 500. The mutation and crossover rates were both set to 0.9. The approximate mutation probabilities are given in Table 3.8. The allocations were chosen to favor small-scale changes (such as changing the weight value) over more large-scale changes (adding or deleting a neuron) because mutations are usually meant to be small steps in the state space, whereas crossover corresponds to larger steps in the state space.

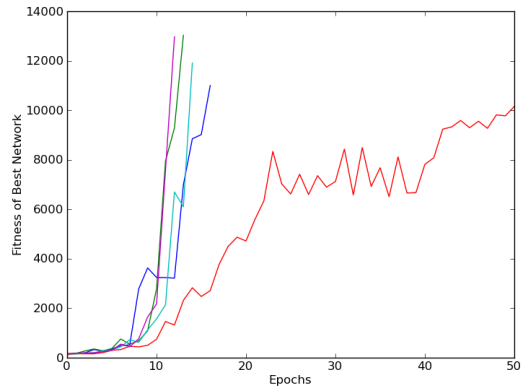
Table 3.8: Approximate probabilities for each mutation type for the pole balancing problem.

Mutation	Approximate Probability
Change sign of a synapse	0.317
Change the value of the weight of a synapse	0.317
Add a synapse	0.159
Remove a synapse	0.159
Add a neuron	0.032
Delete a neuron	0.016

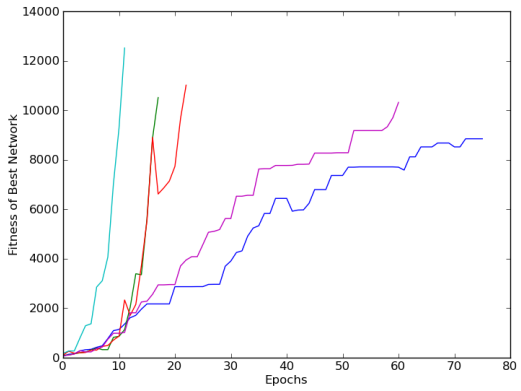
As a sample set, the results of five tests runs are shown in Figure 3.26 for both fitness functions f_1 and f_2.

Fitness function f_1 usually converges in less time. However, as noted above, a high fitness measured using f_1 for a network does not necessarily indicate that that network performs well in general. To test how well the networks perform in general, 110 test cases were applied. The results for the two networks with the highest average performance on these test cases are shown in Figure 3.27. The figure shows that the network for f_2 generalizes better than the network for f_1. In particular, f_2 forces the networks to perform well in a few set conditions, whereas for f_1, it is possible that the resulting network performs very well in some cases and not well in others.

These two networks had comparable size, with the network produced by fitness function f_1 having 28 neurons and 308 synapses and the network produced by f_2 having 27 neurons and 266 synapses.

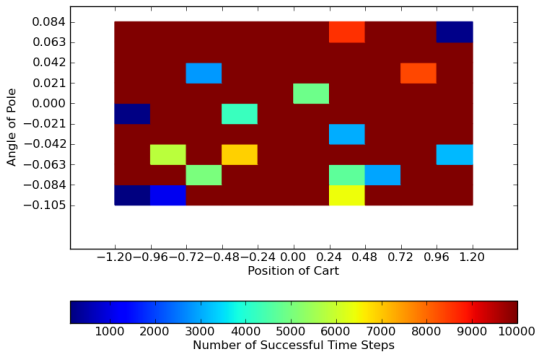


Fitness function f_1.

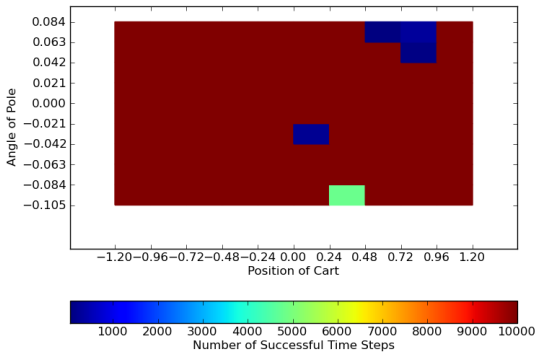


Fitness function f_2.

Figure 3.26: Fitness value (roughly the number of time steps the network can keep the pole balanced) vs. epoch for the best network in the population. Results are shown for five networks for each fitness function.



Fitness function f_1.



Fitness function f_2.

Figure 3.27: Results for the best performing networks for f_1 and f_2 when tested on 110 different start conditions.

Table 3.9: Number of time steps four networks are able to continue balancing the pole. The cut-off point for these networks was 100000 time steps.

	f_1	
Test Case	**Network 1**	**Network 2**
$x = 0$, $\theta = 0.15$	34434	100000
$x = 0$, $\theta = -0.15$	52500	100000
$x = 1.2$, $\theta = 0.15$	64701	100000
$x = 1.2$, $\theta = -0.15$	12105	100000
$x = 1.2$, $\theta = 0.15$	37309	13
$x = 1.2$, $\theta = -0.15$	40767	101

	f_2	
Test Case	**Network 1**	**Network 2**
$x = 0$, $\theta = 0.15$	93964	97147
$x = 0$, $\theta = -0.15$	18168	100000
$x = 1.2$, $\theta = 0.15$	21792	31546
$x = 1.2$, $\theta = -0.15$	100000	98264
$x = 1.2$, $\theta = 0.15$	100000	79678
$x = 1.2$, $\theta = -0.15$	100000	100000

As in the frequency-encoded networks, a second test of performance of these networks is whether they can continue to balance the pole for a longer time period. Instead of placing a bound of 10000 time steps, we now place a bound of 100000 time steps (2000 seconds or 33 minutes and 20 seconds), and test the networks on the same six starting conditions that are used in f_2. The results for two networks for each fitness function are shown in Table 3.9.

The first network for f_1 and f_2 shown in the table are the same networks presented in Figure 3.27. As can be seen, for f_1, the first network does better generally for the six test cases, but the second network performs extremely well for four out of the six test cases and extremely poorly for the other two test cases. This is the hazard of using a fitness function such as f_1, in which random simulations are used for each training step; just because the network performs well on some of the cases does not mean that it will perform well in general.

Conclusions

For this problem, in terms of overall computation time, networks with discrete-encoded input were trained more quickly than the networks with frequency-encoded input. This is because there are many more events occurring in the frequency-encoded networks rather than the discrete-event networks, so each simulation takes much longer. For f_1 each fitness evaluation requires 10 simulations, and for f_2 each fitness evaluation requires six simulations. Also, because the simulation times for the discrete-encoded input networks are shorter, larger population sizes could be used. One approach to alleviate this would be to scale the frequencies of input pulse events to reduce the amount of firing in the network.

3.3.3 Classification: Handwritten Digit

The description of the problem and many of the results in this section are available in [230].

Task Description

We apply our networks to the MNIST handwritten digit classification task [142]. In this task, 28 by 28 pixel images of handwritten digits (0-9) are given as input, and the goal of the task is to produce the correct digit corresponding to the image as output.

This task is entirely static; there is no time component. Thus, simply feeding the image as input to a NIDA network does not take advantage of the dynamic components of our network. To take advantage of the information content that can be stored in a network via synaptic delays and neuronal charges, we chose to add a time component to the task. In particular, rather than feeding the entire image into the network at once, the network "scans" the image in one of three ways: (1) a row at a time, (2) a column at a time, or (3) both a row and a column at a time (Figure 3.28). This allows the task to take advantage of the inherent dynamic properties of NIDA networks. This approach also significantly reduces the size of the network (by

reducing the number of input neurons from 784 to 28 or 56). There are several ways

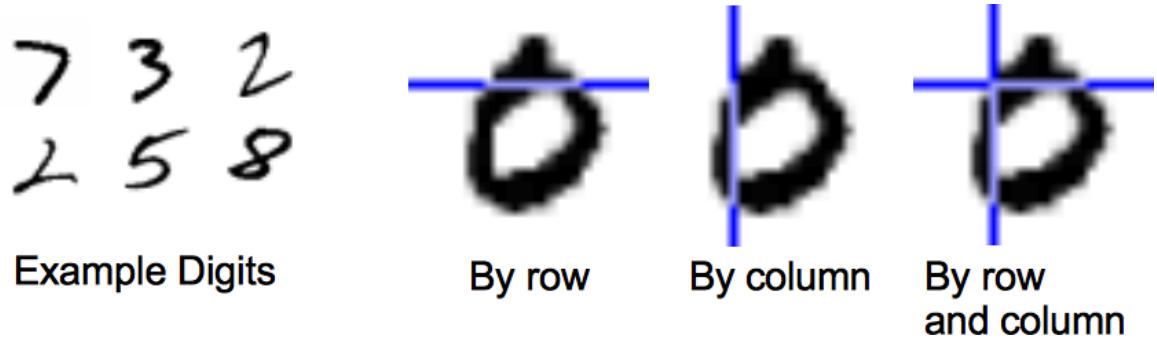


Figure 3.28: Example digits from the MNIST handwritten database [142]. Rather than give an entire image as input to a NIDA network, input is given in one of three ways: (1) a row at a time, (2) a column at a time, or (3) both a row and a column at a time.

one might use our networks to solve this task. A single network could be trained that takes the image as input (in the scanning way as described above) and has 10 output neurons (one corresponding to each digit). Then, based on the activity of the network a guessed digit or digits can be produced, for example, by choosing the digit that corresponds to the output neuron that fires the most during simulation. This is the most straightforward approach; however, because the network is required to recognize each digit type, the resulting networks may be very complex. We instead use many small networks that contribute to the final solution.

In particular, we propose training networks to recognize a single digit at a time. The results of these networks can then be combined via a winner-take-all (WTA) scheme to produce the guessed digit for any given test case. There are multiple fitness or scoring functions that can be defined for this task. We explored two different fitness functions.

The first fitness function takes 500 randomly selected images at a time from the training set (50 representing each digit type). Then, of those 500, the fitness function takes 10 at a time (one for each digit type), and simulates the network on each of those images. Suppose we are training a network to recognize images of the digit d (where $d \in \{0, ..., 9\}$). If the network correctly classifies an image of type d and

correctly identifies that the other nine are not of digit type d, the network's score is increased by 10. If the network fails on any of those ten classifications, its score is unchanged. The goal of this fitness function is to produce networks that are able to identify a particular digit and only digits of that type.

The second fitness function is based on entropy. Again, suppose we are training a network to recognize images of digit type d. For each network, we evaluate that network on all of the training set images to produce a listing of the number of times that network fires (in some window or over the course of the entire simulation). Once all of the training images have been evaluated, a set of 2-tuples is produced for the network, one for each training image: $S = \{(f_1, d_1), (f_2, d_2), ..., (f_n, d_n)\}$, where f_i is the number of times the network fired for training image i and d_i is the correct digit associated with training image i. Define a threshold parameter $T \in \{f_i\}_{i=1}^{n}$. If $f_i \geq T$,let $(f_i, d_i) \in S_{yes}$; otherwise, let $(f_i, d_i) \in S_{no}$. Let $n_{yes,c} = |\{(f_i, d_i) \in S_{yes} : d_i = d\}|$, and let $n_{yes,w} = |\{(f_i, d_i) \in S_{yes} : d_i \neq d\}|$. Let $n_{no,c} = |\{(f_i, d_i) \in S_{no} : d_i \neq d\}|$, and let $n_{no,w} = |\{(f_i, d_i) \in S_{no} : d_i = d\}|$.

The entropy for a particular threshold value T is calculated as follows:

$$H(T) = -\sum_{r \in \{yes,no\}} \frac{|S_r|}{|S|} \sum_{k \in \{c,w\}} \frac{n_{r,k}}{|S_r|} \log_2 \frac{n_{r,k}}{|S_r|} \tag{3.13}$$

The fitness function for training is defined as:

$$F(\text{Network}) = \min_{T \in \{f_i\}_{i=1}^{n}} H(T) \tag{3.14}$$

The evolutionary optimization attempts to find the network (and associated threshold T) that minimizes the fitness function.

In both fitness functions, rather than training over the entire set of training images at a time, 5,000 of the 60,000 are randomly chosen, where 500 of each digit type are represented in the 5,000. This is an attempt to avoid over fitting to the training set. Then, rather than using only one network for each digit, many networks are

chosen. Using one or both of these fitness functions, we assume that the networks that are being produced are recognizing particular features of each digit type, but not necessarily recognizing every feature of a particular digit.

Results

For the networks in this example, the LTP/LTD refractory period used was 100, and the amount the synapses were changed in LTP/LTD was 0.001. For the evolutionary algorithm used in this example, we defined the population size to be 100, the crossover rate to be 0.7, and the mutation rate to be 0.99. The approximate probabilities defined for each mutation are given in Table 3.10.

Table 3.10: Approximate probabilities for each mutation type for the handwritten digit problem.

Mutation	Approximate Probability
Change sign of a synapse	0.244
Change the value of the weight of a synapse	0.244
Add a synapse	0.122
Remove a synapse	0.122
Add a neuron	0.122
Delete a neuron	0.024
Alter threshold	0.122

Each network produced by the evolutionary optimization method scans an input image, receiving one row, one column, or both a row and a column at a time. Each network in the ensemble receives each image as input. We have defined two methods by which a network may identify a digit d. In both cases, there is a single output neuron of the network and only the firings that occur in the last 50 time steps of a 500 time step simulation are considered. The network may then be trained to fire in the last fifty time steps if the image read in is an image of the digit d (we call this the "firing" method), or it may be trained to fire if the image read in is not an image of the digit d (we call this the "not firing" method). Networks trained to recognize the digit d with the firing method cast a vote for digit d when they fire in the last 50 time

steps, and networks trained to recognize the digit d with the not firing method cast a vote for digit d when they do not fire in the last 50 time steps. The digit receiving the most votes is the decision of the ensemble (Figure 3.29).

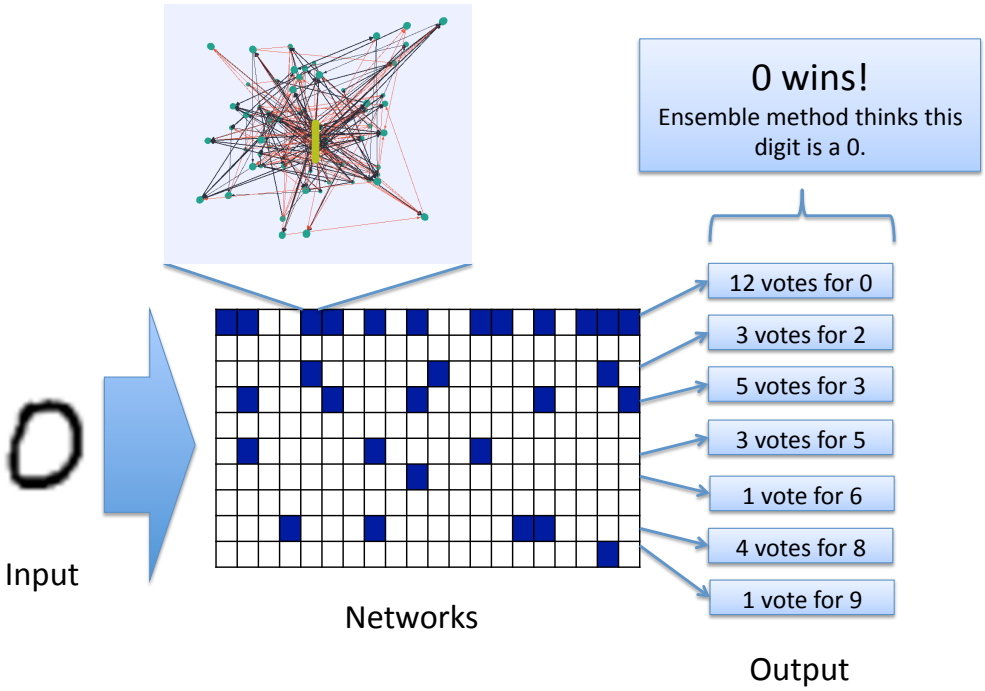


Figure 3.29: A voting scheme amongst networks of an ensemble is used to determine the digit for a particular image. Each network in the ensemble (represented by a square on the grid) receives an image on input and simulates activity within the network. Based on that activity, the network may or may not cast a vote (casting a vote is represented by shading in the grid). The digit with the most votes is the guessed digit for the ensemble.

The first fitness function trained with the firing method produced an ensemble of networks that resulted in 88.3 percent accuracy on the testing set of handwritten digits. This results ensemble was made up of 1200 networks, 40 for each digit for each scanning type. The accuracies for each scanning type were as follows: 81.0 percent accuracy by row, 86.1 percent accuracy by column, and 85.8 percent accuracy for both

row and column. Figure 3.30 shows the results for each digit type for the ensemble of 1200 networks.

This figure gives some insight as to how the networks are operating. For example, nines were often mis-classified as either fours or sevens, which we categorize as a valid mis-guess, as some nines may often appear very similar to fours or sevens. Similarly, threes are often mis-classified as fives. Again, threes have many similar features as fives, so the mis-guess makes sense within the construct of the problem. It also indicates that this fitness function may be inadvertently producing networks that recognize features, rather than the entire digit, so the second, more complex fitness function may not be required.

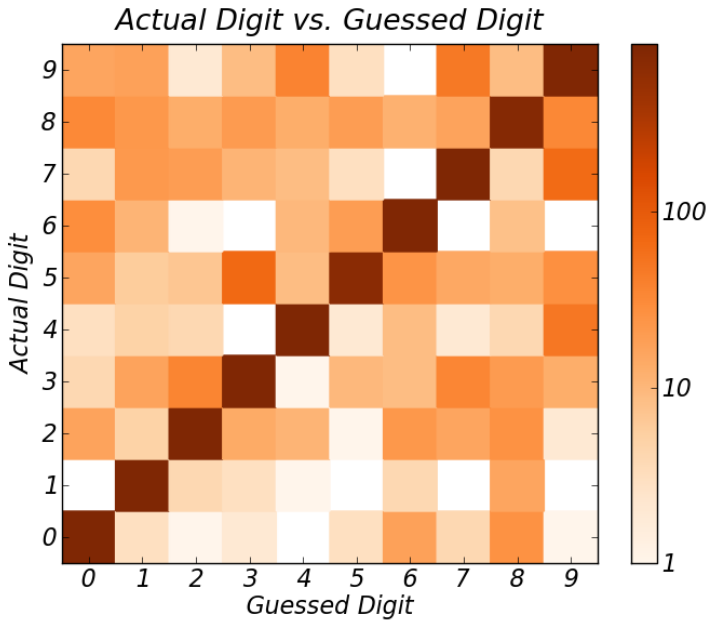


Figure 3.30: Classification results for the first fitness function trained with the firing method. Elements along the diagonal indicate correctly classified digits. All other values are incorrectly classified. Note that the coloring is on a log scale, which was used so that incorrect classifications can be more easily observed.

The first fitness function trained with the not firing method produced an ensemble of networks that resulted in 89.3 percent accuracy on the testing set of handwritten digits. This results ensemble was made up of 1200 networks, 30 for each digit for

each scanning type, plus an additional 30 for each digit, scanning by column, as those networks produced the best results. The accuracies for each scanning type were as follows: 77.9 percent accuracy by row, 86.6 percent accuracy by column, and 84.1 percent accuracy for both row and column. Figure 3.31 shows the results for each digit type for the ensemble of 1200 networks. Again, many of the same misclassifications are valid, as in the firing method. It is interesting to note the difference in performance between the firing method and the not firing method. Table 3.11 shows a breakdown of accuracies by digit for networks scanning by column. The not firing method outperforms on every digit except for one and nine. Ones are relatively simple to recognize, and their performance is comparable. The firing method, however, outperformed the not firing method by a relatively large margin on nines. Figure 3.11 shows that the not firing ensemble misclassified nines as every other digit.

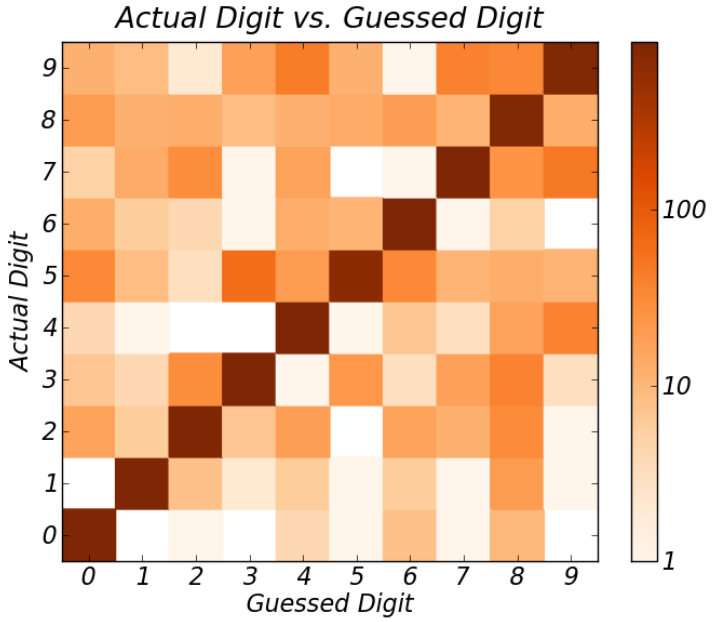


Figure 3.31: Classification results for the first fitness function trained with the not firing method. Elements along the diagonal indicate correctly classified digits. All other values are incorrectly classified. Note that the coloring is on a log scale, which was used so that incorrect classifications can be more easily observed.

Table 3.11: Accuracy breakdown for networks scanning by column trained with firing and not firing methods

Digit	Accuracy for Firing	Accuracy for Not Firing
0	93.5	96.1
1	96.6	96.4
2	84.7	86.6
3	78.6	80.3
4	86.3	87.7
5	65.6	71.1
6	91.0	92.5
7	81.2	86.1
8	75.9	78.1
9	82.7	76.3

The second fitness function produced an ensemble of networks that results in 83.5 percent accuracy on the testing set. An important feature of the second fitness function is that it is just as likely to produce networks to recognize a particular digit type that fire when the image is *not* of that digit type (that is, the firing and not firing method are combined for this fitness function). For example, if the network is supposed to recognize threes, this fitness function may produce networks that fire when an image is not of a three, rather than firing when recognizing a three. We tested each network's classification accuracy on the training set to determine if the network fired when recognizing the digit it was supposed to classify or if the network fired when it determined the current image was not the digit it was meant to recognize. The results ensemble was made up of 200 networks, 20 for each digit. For this fitness function, only networks scanning by column were produced, as those networks resulted in the best individual accuracy. Figure 3.32 shows the results for each digit type.

We also combined all three sets of ensembles (first fitness function with firing method, first fitness function with not firing method, and second fitness function) to produce one large ensemble of 2600 networks. This ensemble produced a testing classification accuracy of 90.6 percent. Figure 3.33 shows the results for each digit

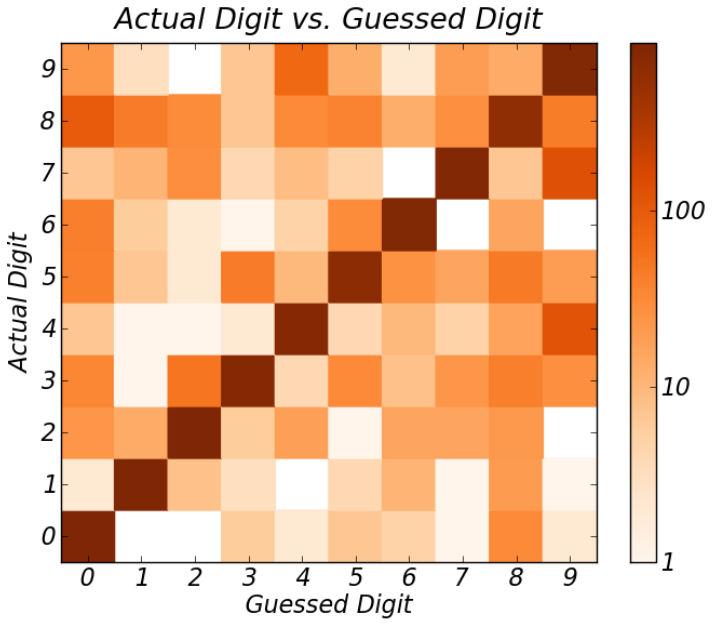


Figure 3.32: Classification results for the second fitness function. Elements along the diagonal indicate correctly classified digits. All other values are incorrectly classified. Note that the coloring is on a log scale, which was used so that incorrect classifications can be more easily observed.

type. The sizes of the networks for each digit type are shown in Figure 3.34. Most of the networks were very similar in size, with networks scanning both by row and column being larger in number of neurons. This inflation is mostly due to the fact that these networks had 56 input neurons rather than 28 as the other network types did.

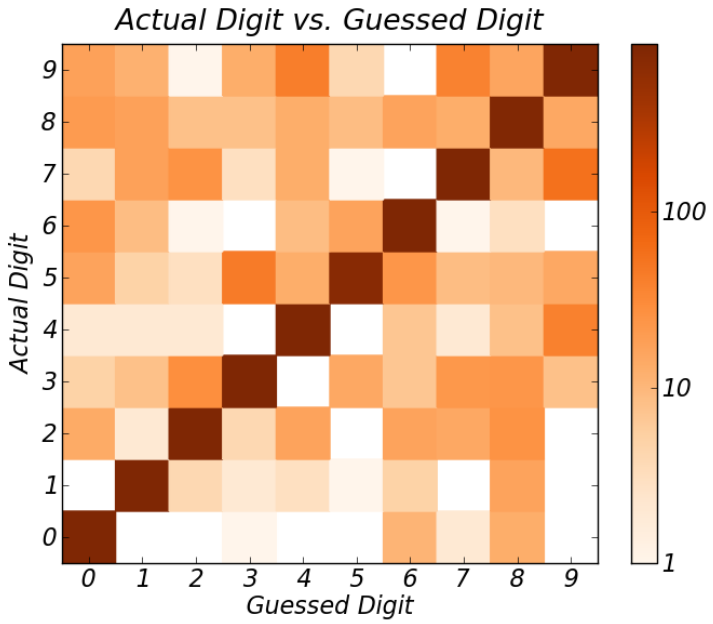


Figure 3.33: Classification results for combined ensemble of networks trained with the first fitness function with the firing method, the first fitness function with the not firing method, and the second fitness function (entropy-based). Elements along the diagonal indicate correctly classified digits. All other values are incorrectly classified. Note that the coloring is on a log scale, which was used so that incorrect classifications can be more easily observed.

Each of the networks produced was generated in 24 hours or less. No pre-processing was done on the handwritten images. Some pre-processing on the images may also improve performance. These results are not yet comparable with the state-of-the-art results on classification of handwritten digits, which are available in [142], and which have less than 1 percent error in classification of the testing set. However, many of the other methods used in solving this problem are tailored specifically for problems such as this one, whereas NIDA networks are meant to perform relatively

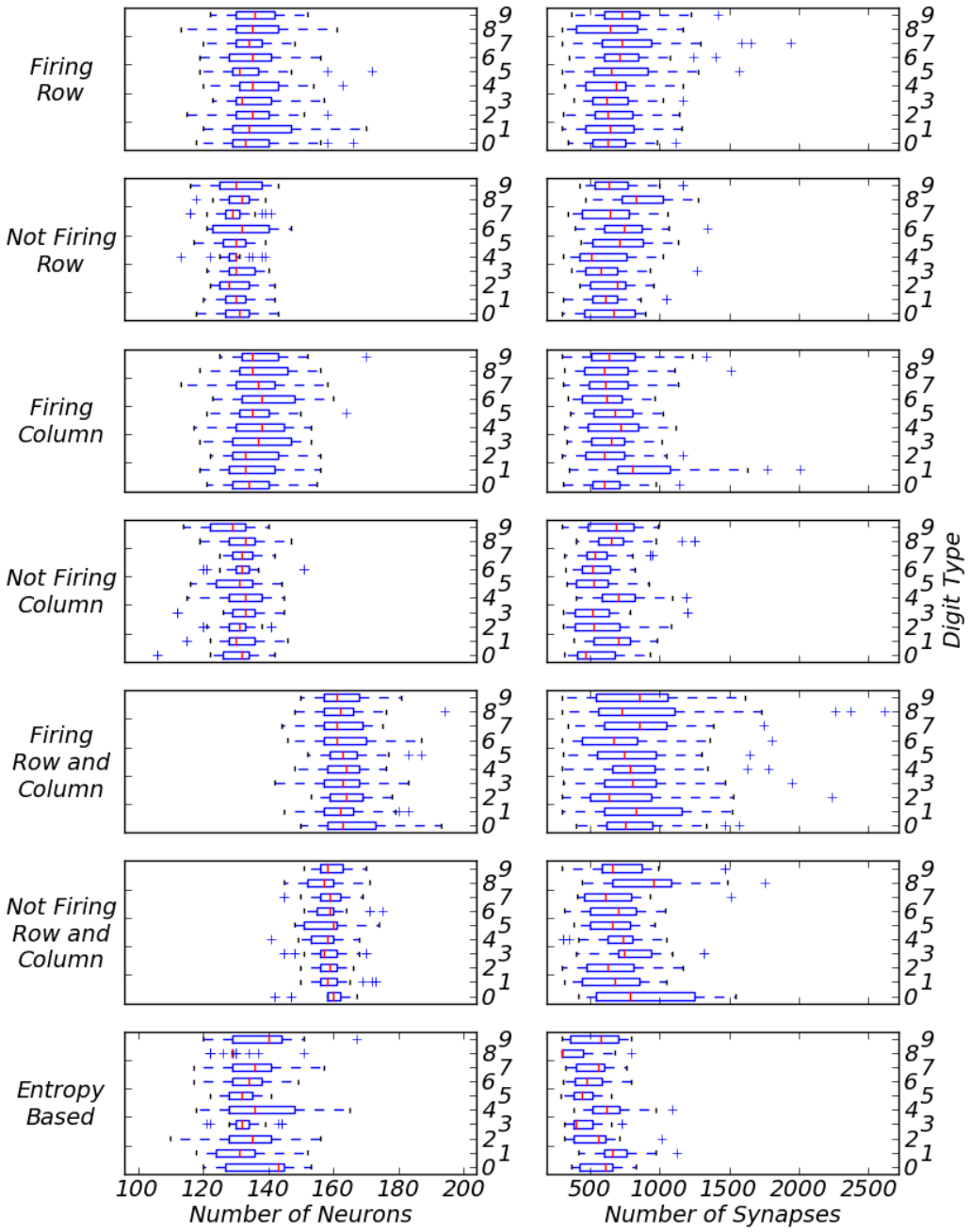


Figure 3.34: Network size information for the networks produced. The first six rows of the plots correspond to networks produced using the first fitness function (alternating firing and not firing method), while the last row of networks corresponds to networks produced using the second fitness function (based on entropy). The box and whisker plots show the lower to upper quartile values, with a red line at the median. The whiskers indicate the range of the data, with outliers also shown. Numbers of neurons (first column) and synapses (second column) are given for each digit type and each scanning type.

well on a wide variety of problem types, including those with temporal components. NIDA has comparable performance on this task with Beyeler's spiking neural network model [34], which achieved 91.6 percent accuracy on the testing set.

Conclusion

NIDA networks are most suitable to dynamic tasks (tasks in which there is a time component). Many static tasks can be adapted to include a time component, as we did with the "scanning" of the handwritten digit images, thus taking advantage of the dynamic properties of our network. The main advantage of our network architecture and design method is that it can produce networks to solve a wide variety of tasks, including anomaly detection tasks, control tasks, and classification tasks. A primary disadvantage of using evolutionary optimization as the training method is that it can be relatively slow to adapt for some problem types. However, we believe that the flexibility that evolutionary optimization provides justifies its use [229].

Instead of producing a single network to solve the entire handwritten digit classification problem, we produce many networks solving a subproblem of the classification problem (identifying a particular digit) and combine the results to produce a solution. This approach produces useful networks that can be combined into one larger network to produce the solution. In particular, the approach produces useful subnetworks to be included in the network that solves the larger problem of taking an image as input and identifying the digit represented in that image. By breaking a problem into smaller subproblems and solving those, we can then combine the networks into a single larger network and use evolutionary optimization to discover the connections required between the subnetworks. A similar approach can be applied in many anomaly detection and control problems in which there are subproblems to be solved. In fact, many problems may reuse the same subnetwork many times. Furthermore, we suspect that networks that are useful for one problem may also be useful for other problems, as is observed in biology [10, 199]. By building a library of networks that solve subproblems, we can reuse these networks in future

applications and improve learning time by tweaking existing networks instead of generating networks from scratch for each problem (see Chapter 4).

We present two fitness functions to develop networks that classify handwritten digits. We believe that the second fitness function or similar functions based on entropy are worth pursuing for classification tasks. In particular, the second fitness function may be able to produce networks to be used in a decision tree framework, used in combination with algorithms such as ID3 [201] or CART [45]. Either of these approaches could be used to select a tree of classifiers that utilize the entropy-based feature extraction methods in order to maximize information gain.

3.4 Fixed Structure Weight Training

We explored the capability of our NIDA architecture with fixed structure to determine whether structural evolution is necessary. The results of this exploration are important to demonstrate that structural evolution is necessary; otherwise, we may pre-define a structure and not add or delete neurons and synapses to suit the application. We found that structural evolution was necessary in general in order to achieve a good result.

For this test, we used the discrete-encoding pole balancing problem with fitness function f_2. A network with N neurons and S synapses is instantiated, where $N-14$ of the neurons are randomly placed hidden neurons and the remaining fourteen are the twelve input and two output neurons, which are in the same location for all networks. The S synapses are also randomly generated in that two random neurons are chosen (those neurons may be input, output, or hidden), one is chosen to be the pre-synaptic neuron, while the other is chosen to be the post-synaptic neuron, and the weight is randomly initialized. For fixed structure training, we made random changes to the weight values of the S synapses in the network, and if the new network outperformed the old network, we kept the new network (Figure 3.35). For this task,

we tried $N \in \{20, 30, 40, 50, 60, 70, 80, 90\}$ and $S = s * N$, where $s \in \{5, 10, 15, 20, 25\}$ and $s < N$. We call s the "synapse multiplier."

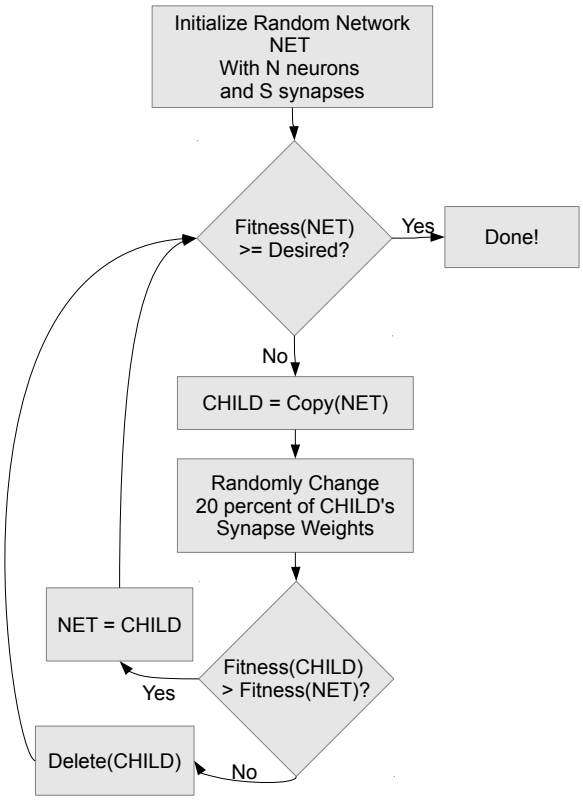


Figure 3.35: Flow chart showing the fixed-structure weight training method used.

For each of these N and S values, we ran 60 test runs with different random seeds, so each of the 60 networks for a particular N and S values had a different random network. The fixed-structure weight training method was applied to each of these networks and allowed to run for up to 24 hours or up to 20,000 iterations of the method. Figure 3.36 shows the resulting average fitness value, maximum fitness value, and minimum fitness value achieved for varying values of N and S.

These results show that the smaller networks were more likely to find an adequate solution than the more complex networks for this application. This is likely because it is more difficult to find a set of weight values that interacts well for more

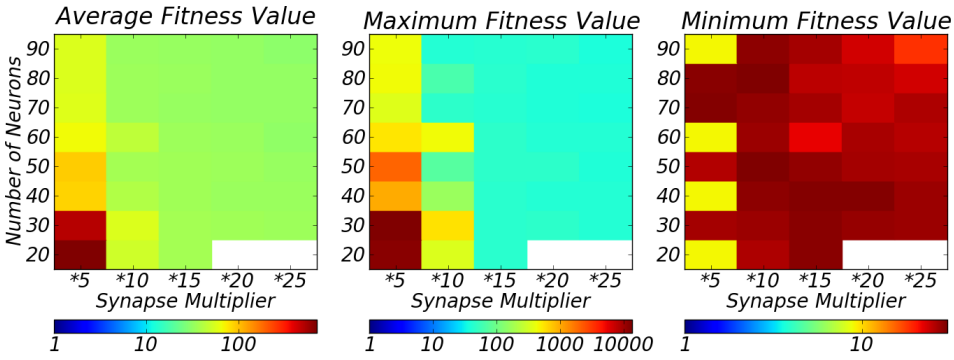


Figure 3.36: Heat maps showing the average, maximum, and minimum fitness values for varying values of N and varying s multiplier values, where $S = s * N$. The values where $N <= s$ are displayed as white squares in the heat map. Higher values indicate better performing networks.

complex networks than for smaller, less complex networks. Interestingly, however, the minimum fitness values were worst for the smaller, less complex networks. This indicates that there is a set of connections that are necessary for this problem to be solved adequately, and if these connections are not present, the problem is unable to be solved. Based on the averages and maximum values, however, the smaller networks were by far the most successful.

Though it may be possible for a random initial structure or a structure that we choose to have a good solution found through weight training only, we are unlikely to stumble across this structure by randomly initializing the network. We contrast this with the results given in Section 3.3.2, in which networks converged for this problem within 100 epochs using structural and parameter evolution. By allowing the optimization algorithm to discover structure as well as parameters, we improve the learning rate for these networks. Thus, the main point of these results is the importance of structural evolution for NIDA networks.

3.5 Effect of Mutation and Crossover Rates on Training

Two parameters of the design method are the crossover rate, r_c, and the mutation rate, r_m. In the evolutionary optimization design method, two parents are always selected at a time to produce children. The crossover rate specifies how often these two parent networks are combined using crossover to produce two children. If they are not "crossed over," the children are duplicates of the parents. The mutation rate specifies how often the resulting child networks are then mutated. If a mutation is chosen for one or both of the children, then one of several mutations is selected randomly, with the associated probability.

In this section, we look briefly into the effect of the crossover and mutation rates on training of the network. We found that consistent values that gave the best performance for these rates were not observed. This motivates future work in which we may use an adaptation strategy to select values that give better results for each task.

We explored two problems: the pole balancing problem and the handwritten digit problem. For each of these problems, we looked at the following values for both the crossover and mutation rates: $\{0, 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1\}$. For each combination of crossover and mutation rates, we ran 10 tests with 10 different random seeds. However, the same 10 seeds were used for all crossover/mutation rate combinations, to avoid variations in results due to the random populations used. Each network was allowed to be trained for two hours and the best fitness value obtained in that two hour time period was used. In all of the results shown below, the average of the best fitness value over the 10 runs is given.

Figure 3.37 shows the results for the pole balancing example. This figure indicates that crossover is required in order for success in all tests. Crossover allows for large scale structural changes. In the previous section we showed that structural evolutions are important for success, so these results confirm that large-scale structural evolution

is also important. Mutation rates had little or no effect on the performance of the resulting networks. Figures 3.38a through 3.38j show the results for handwritten digit recognition, where the figures correspond to the results for different digits (0-9). Figure 3.38b shows that the mutation rate was very important for developing recognizers of the digit 1, while the crossover rate was much less important. Most of the results for other digits indicate that a higher crossover rate can improve performance and that very high mutation rates can result in lower performing networks.

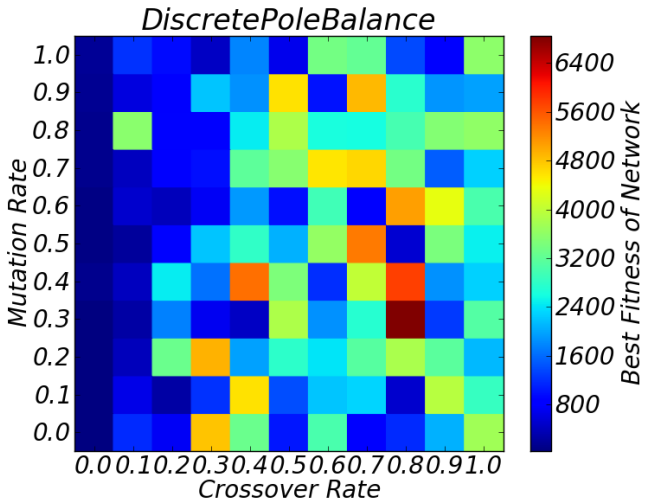
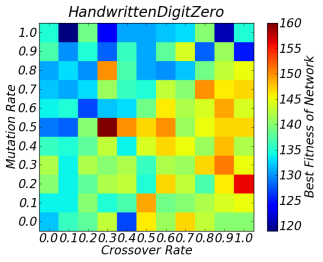
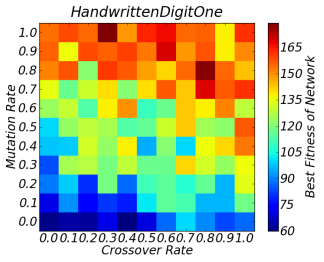


Figure 3.37: Results from the pole balancing problem (discrete encoding and fitness function f_2) for varying crossover and mutation rates. Each block in the map represents the average fitness value over 10 runs of the best network after training for two hours for that crossover and mutation rate combination.

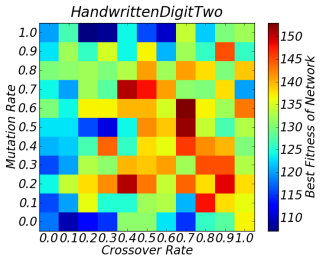
Crossover brings about a large-scale structural change on the networks and provides more diversity of structure in the population; mutations make relatively low-level changes to either parameters of neurons or synapses or the structure of the network. It is clear from some of our testing that even small-scale changes may have a major effect on performance (as demonstrated in Section 3.4), as parts of the network affected by mutations may be integral to performance. Some tasks may exhibit a large number of optimal or near optimal network structures and require only small-scale changes to reach a good solution to a problem, in which case a high mutation rate



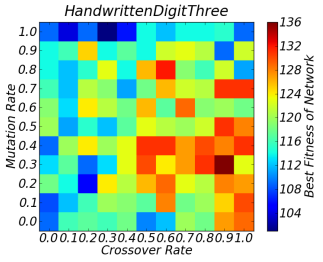
(a) Recognizer of 0



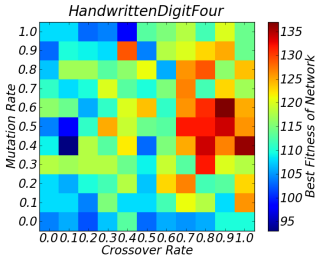
(b) Recognizer of 1



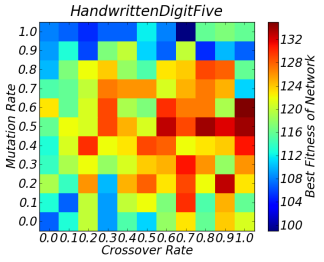
(c) Recognizer of 2



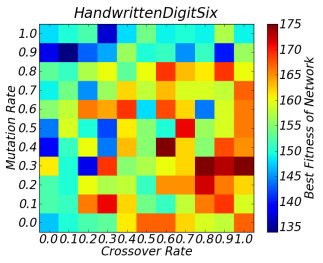
(d) Recognizer of 3



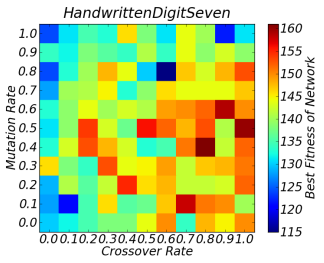
(e) Recognizer of 4



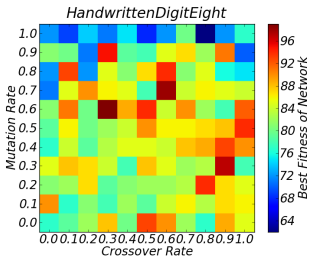
(f) Recognizer of 5



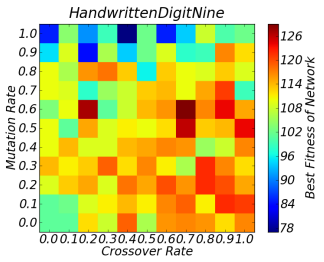
(g) Recognizer of 6



(h) Recognizer of 7



(i) Recognizer of 8



(j) Recognizer of 9

Figure 3.38: Results from the digit recognizers for the handwritten digit problem (first fitness function, scanning by column) for varying crossover and mutation rates. Each block in the map represents the average fitness value over 10 runs of the best network after training for two hours for that crossover and mutation rate combination.

and low crossover rate may be sufficient. Other problems may have very restrictive structural requirements, requiring a large amount of structural exploration, in which case a high crossover rate may be required. It may not be clear *a priori* into which type a task belongs.

The key insight from this set of tests is that while the mutation and crossover rates clearly have an impact on training, there is not one set of values that will work well for every problem (or even versions of the same problem). One possible approach for determining crossover and mutation rates for a particular task is to allow the crossover and mutation rates to adapt over the course of training to the problem. For example, a simulated annealing approach might be useful, in which the rates decrease as results for the particular problem improve. It might also be beneficial to increase the crossover and mutation rates when results stagnate, in order to stimulate new innovation in the structure and parameters. In other words, this motivates the use of an adaptation strategy to determine the most applicable crossover and mutation rates.

3.6 Comparison to Standard Benchmarks

We selected three of the most popular data sets from the UCI Machine Learning Repository [17] in order to compare performance of NIDA networks with other network architectures. The architectures and methods that we compare against are described in more detail in Section 2. In particular, we chose to compare against published results on these datasets using spiking neural network architectures trained with genetic algorithms or evolutionary strategies (discussed in Section 2.3.2) and traditional neural networks trained with genetic algorithms (discussed in Section 2.1). The three data sets we chose were the Wisconsin breast cancer data set, the Pima Indian diabetes data set, and the iris data set. Results were reported for one or more of these data sets using the following methods.

Methods that use neuroevolution to train traditional artificial neural networks:

- **MPANN**: Memetic pareto ANN, a neuroevolution method described in Section 2.1 and in [1].
- **Fogel** - A neuroevolution method developed by Fogel *et al.* described in Section 2.1 and in [81, 82].
- **Alba-GA**: Traditional artificial neural networks trained with a genetic algorithm as described in [5].
- **Alba-GALM**: Hybrid of genetic algorithm and Levenberg-Marquardt algorithm used to train traditional artificial neural networks as described in Section 2.1 and in [5].
- **Alba-GABP**: Hybrid of genetic algorithm and back-propagation algorithm used to train traditional artificial neural networks as described in Section 2.1 and in [5].
- **GP-StdX**: Evolutionary algorithm using standard crossover used to train traditional artificial neural networks, described in Section 2.1 and in [92].
- **GP-GaX**: Evolutionary algorithm using crossover based on the genetic algorithm used to train traditional artificial neural networks, described in Section 2.1 and in [92].
- **GP-SaX**: Evolutionary algorithm using crossover based on simulated annealing used to train traditional artificial neural networks, described in Section 2.1 and in [92].
- **MGNN-roul**: Mutation-based genetic neural network using roulette-wheel selection, as described in Section 2.1 and in [186].
- **MGNN-rank**: Mutation-based genetic neural network using rank-based selection, as described in Section 2.1 and in [186].

- **MGNN-ep**: Mutation-based genetic neural network with evolutionary programming-based scheduled mutation policy and rank-based selection, as described in Section 2.1 and in [186].

- **EPNet**: Evolutionary system for evolving the architecture and weights of feed-forward neural networks, as described in Section 2.1 and in [273].

Spiking neural network methods:

- **SRM SNN**: Spike response model spiking neural networks as described in Section 2.3.2 and in [26].

- **DSSNN**: Dynamic synapse spiking neural networks as described in Section 2.3.2 and in [26].

- **SpikeProp**: Spiking neural networks trained with a back-propagation like method, described in [41]; results reported in [26].

- **Jin**: Spiking neural network trained with Pareto optimization as described in Section 2.3.2 and in [123].

- **SNN-PARDE**: Parallel differential evolution algorithm for designing spiking neural networks as described in Section 2.3.2 and in [193].

Other methods:

- **MDT**: Multivariate decision trees, described in [1, 2].

- **MatlabBP**: Traditional artificial neural networks trained with back-propagation as implemented in MATLAB; results reported in [26].

- **MatlabLM**: Traditional artificial neural networks trained with the Levenberg-Marquardt algorithm, as implemented in MATLAB; results reported in [26].

- **Alba-BP**: Traditional artificial neural network trained with back-propagation, with results as reported in [5].

- **Alba-LM**: Traditional artificial neural network trained with the Levenberg-Marquardt algorithm, with results as reported in [5].
- **HDANNs**: Hand designed traditional feed-forward neural networks, with results reported in [273].
- **FNNCA**: Feed-forward neural networks trained with a quasi-Newton method, as described in [234], with results reported in [273].
- **Perceptron**: Perceptron network trained using the thermal perceptron algorithm, with results reported in [189].
- **MPyramid-real**: Constructive algorithm described in Section 2.1 for neural networks using a pyramid building algorithm, as described in [189].
- **MTiling-real**: Constructive algorithm described in Section 2.1 for neural networks using a tiling algorithm, as described in [189].
- **Pav-BP**: Multilayer perceptron network trained using back-propagation, as reported in [193].
- **Pav-MBP**: Multilayer perceptron network trained using back-propagation with momentum, as reported in [193].
- **Pav-SMBP**: Multilayer perceptron network trained using back-propagation with second order momentum, as reported in [193].

Results for the diabetes data set were also given for a set of statistical methods as described and reported in [167]:

- **Logdisc**: Logistic discrimination.
- **Cal5**: Decision tree method.
- **ITrule**: Decision tree method.

- **DIPOL92**: Hybrid algorithm that shares characteristics with logistic discrimination and nonparametric statistical methods.
- **Discrim**: Linear discriminant analysis.
- **Quadisc**: Quadratic discriminant analysis.
- **CART**: Classification and regression tree.
- **RBF**: Radial basis function.
- **CASTLE**: Causal structures from inductive learning.
- **NaiveBay**: Naive Bayesian classifier.
- **IndCart**: An extension of CART.
- **SMART**: Classification and regression type algorithm.

The acronyms in bold will be used to reference that method in the results presented below. In these results, the metric for comparison used is testing error, the error rate of each classifier on the testing set. One common metric for comparison is training time; however, training times were reported inconsistently in the literature, so an accurate comparison could not be made.

3.6.1 Breast Cancer

The Wisconsin breast cancer data set is composed of 699 instances of patient information related to diagnostics of breast cancer [17, 156].* There are nine relevant attributes per instance, each of which is assigned a value between 1 and 10: clump thickness, uniformity of cell size, uniformity of cell shape, marginal adhesion, single epithelial cell size, bare nuclei, bland chromatin, normal nucleoli, and mitoses. These nine attributes serve as inputs to the networks. The output of the network corresponds

*The Wisconsin breast cancer data set is available at https://archive.ics.uci.edu/ml/datasets/Breast+Cancer+Wisconsin+(Original).

to one of two values: benign or malignant. There are missing values in the data set, which means that some of the instances are incomplete. The 699 instances can be divided into two sets: training and testing, where the training instances are used to train networks to derive the optimal NIDA network, and the testing set is used to verify how well the network was trained to perform the classification. Results are reported in terms of classification error on the testing set, where the classification error, E, is defined as:

$$E = \frac{\text{Number of incorrect classifications}}{\text{Total number of test instances}} * 100 \tag{3.15}$$

The networks used in this comparison had nine inputs neurons, one corresponding to each attribute. There is a single output neuron. If that output neuron fires in the last 50 time steps of a 500 time step simulation, then the network classifies that instance as malignant; if the output neuron does not fire in that window, the network classifies that instance as benign. The values of each attribute are integer values and thus are easily converted into pulse streams. If the value is x, then x pulses were applied to the corresponding input neuron, starting at time 0, and spaced five simulated time steps apart. Thus, if the value is 3, then three pulses are applied to that neuron, one at time 0, one at time 5, and one at time 10. The missing values were set as 0 and thus ignored by the network. Other encoding schemes can be used.

A population of size 100 was used, with a crossover rate of 0.9 and a mutation rate of 0.99. The evolutionary optimization was allowed to proceed for 100 epochs or until the best fitness was achieved. The fitness function ran 150 randomly selected training instances of both malignant and benign types and evaluated how the network performed on those training instances. A training instance, in this context, is one complete set of input information (nine values in this case) and the classification of benign or malignant to be used in evaluation. The fitness value was the fraction correctly classified. A perfect fitness function score corresponds to correctly classifying

all 300 randomly selected instances, which is the second stopping condition of the evolutionary optimization.

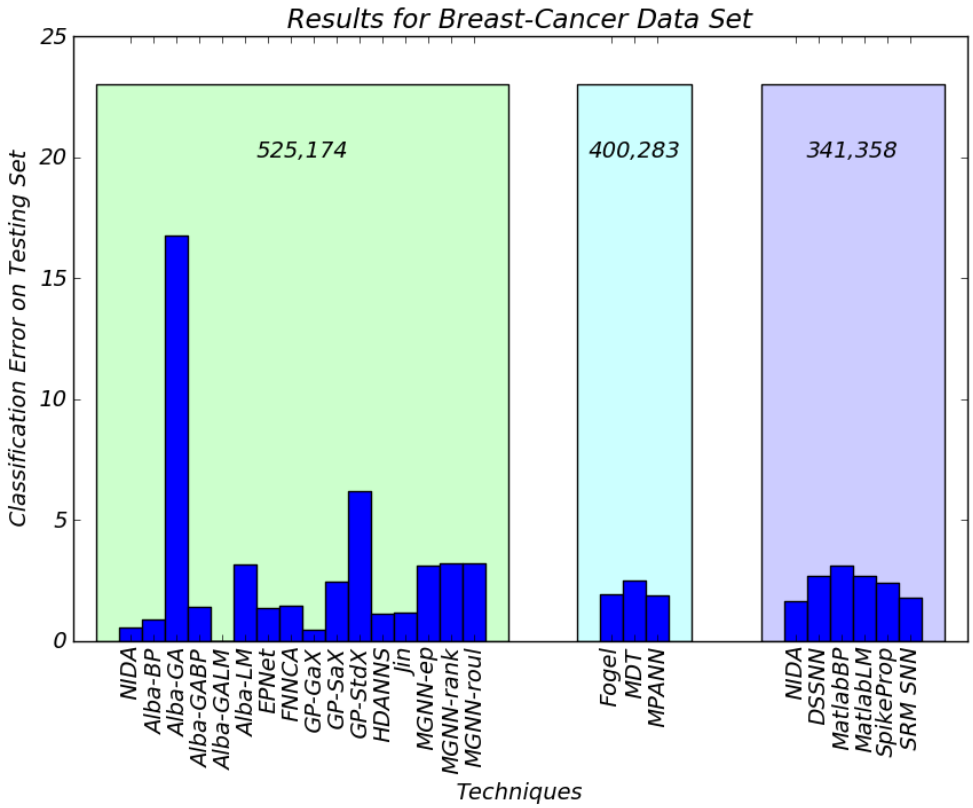


Figure 3.39: Bar chart depicting results for the breast cancer classification task. Results are grouped by the number of training/testing instances, where the number of training instances is listed first and the number of testing instances is listed second. Each method is labeled.

Table 3.12 and Figure 3.39 show the results, listed by increasing testing error rate for a variety of methods on the breast cancer data set. Various methods used different sized training/testing sets. NIDA's results are bolded and are shown for two sizes of training/testing set. NIDA had the third best overall results amongst these comparisons and the best results for the smaller training size. The two methods with higher results than NIDA are Alba-GALM and GP-GaX, both neuroevolution methods that train traditional artificial neural networks. It is important to note that Alba-GALM is a method that made use of both a genetic algorithm and the

Table 3.12: Classification error rate on the testing set for the breast cancer classification problem. The number of training and testing instances used are listed; when unknown they are left blank. The number of testing instances listed with the (*) indicate that the instances with unknown data were not included as part of the testing or training set. The results are listed in increasing order of testing error rate. NIDA results are bolded.

Technique	Number Training Examples	Number Testing Examples	Testing Error
Alba-GALM	525	174	0.02
GP-GaX	525	174	0.46
NIDA	**525**	**174**	**0.58**
Alba-BP	525	174	0.91
HDANNS	525	174	1.15
Jin	525	174	1.2
EPNet	525	174	1.376
Alba-GABP	525	174	1.43
FNNCA	525	174	1.45
NIDA	**341**	**358**	**1.68**
SRM SNN	341	358	1.8
MPANN	400	283*	1.9
Fogel	400	283*	1.95
SpikeProp	341	358	2.4
GP-SaX	525	174	2.48
MDT	400	283*	2.5
DSSNN	341	358	2.7
MatlabLM	341	358	2.7
MatlabBP	341	358	3.1
MGNN-ep	525	174	3.14
Alba-LM	525	174	3.17
MGNN-rank	525	174	3.22
MGNN-roul	525	174	3.23
GP-StdX	525	174	6.19
Alba-GA	525	174	16.76

Levenberg-Marquardt method to train the neural network for this task. We speculate that this method outperforms NIDA because Levenberg-Marquardt is particularly suited for solving this type of problem for feed-forward neural networks. GP-GaX, on the other hand, uses an advanced crossover operation based on genetic algorithms to avoid the competing conventions problem, so it is essentially a nested genetic algorithm. Both of these methods are restricted to feed-forward neural network architectures, unlike NIDA, which works on recurrent architectures and with spiking neural networks. NIDA outperforms all of the spiking neural network methods on this task in terms of classification error.

3.6.2 Diabetes

The Pima Indian diabetes data set is composed of 768 instances of patient information related to diagnosing diabetes [17].[†] There are eight attributes per instance: number of times pregnant, plasma glucose concentration, diastolic blood pressure, triceps skin fold thickness, 2-hour serum insulin, body mass index, diabetes pedigree function, and age. These attributes serve as input to the network. The output of the network corresponds to whether or not the patient has diabetes, based on those attributes. There are missing values in the data set, which means that some of the instances are incomplete.

The networks used in this comparison had eight input neurons, one corresponding to each attribute. There is a single output neuron. If that output neuron fires in the last 50 time steps of a 500 time step simulation, then the network classifies that instance as positive for diabetes; if the output neuron does not fire in that window, the network classifies that instance as negative. The input values were real-valued and not easily translatable to pulse streams, so, as was done in many of the other works using this data set, the values for each attribute were scaled to between 0 and 10 and rounded to the nearest integer. As in the breast cancer test case, if the scaled

[†]The Pima Indian diabetes data set is available at https://archive.ics.uci.edu/ml/datasets/Pima+Indians+Diabetes.

value is x, then x pulses were applied to the corresponding input neuron, starting at time 0, and spaced five simulated time steps apart. The missing values were set as 0 and thus ignored by the network. Other encoding schemes can be used.

A population of size 100 was used, with a crossover rate of 0.9 and a mutation rate of 0.99. The evolutionary optimization was allowed to proceed for 100 epochs or until the best fitness was achieved. The fitness function ran 75 random training instances of both class types and evaluated how the network performed on those training instances. The fitness value was the fraction correctly classified. A perfect fitness function value corresponds to correctly classifying all 150 randomly selected instances, which is the second stopping condition of the evolutionary optimization.

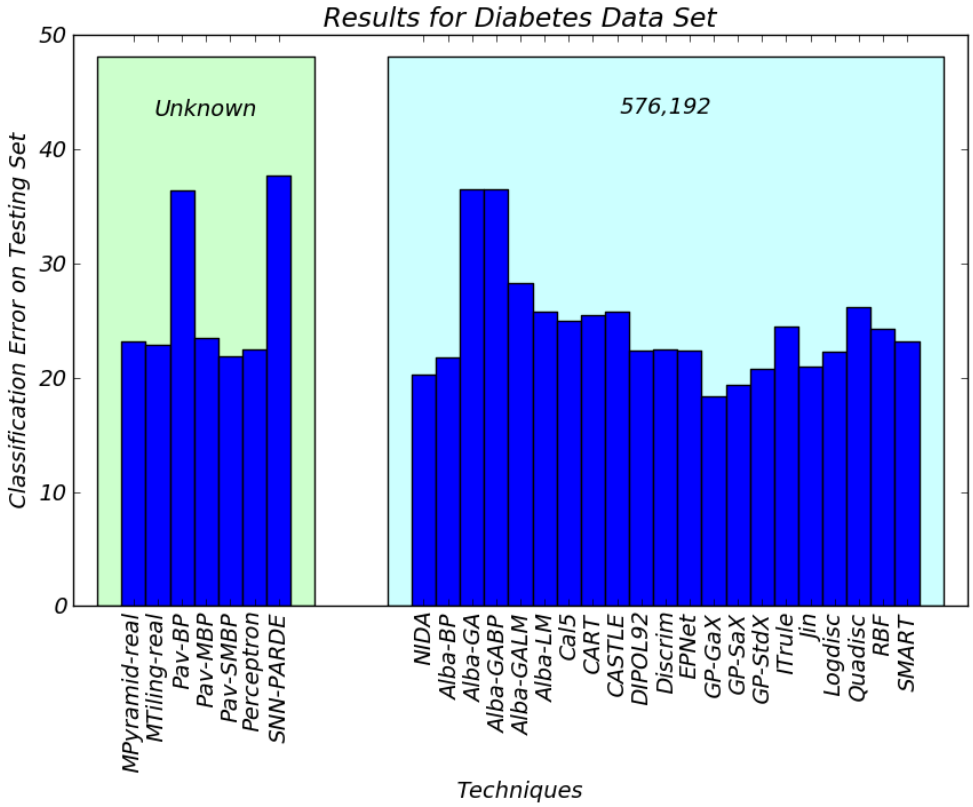


Figure 3.40: Bar chart depicting results for the diabetes classification task. Results are grouped by the number of training/testing instance, where the number of training instances is listed first and the number of testing instances is listed second. It is also noted if the number of training/testing instances is unknown. Each method is labeled.

Table 3.13: Classification error rate on the testing set for the diabetes classification problem. The number of training and testing instances used are listed; when unknown they are left blank. The results are listed in increasing order of testing error rate. NIDA results are bolded.

Technique	Number Training Examples	Number Testing Examples	Testing Error
GP-GaX	576	192	18.4
GP-SaX	576	192	19.39
NIDA	**576**	**192**	**20.31**
GP-StdX	576	192	20.82
Jin	576	192	21
Alba-BP	576	192	21.76
Pav-SMBP			21.88
Logdisc	576	192	22.3
EPNet	576	192	22.4
DIPOL92	576	192	22.4
Perceptron			22.5
Discrim	576	192	22.5
MTiling-real			22.9
SMART	576	192	23.2
MPyramid-real			23.2
Pav-MBP			23.43
RBF	576	192	24.3
ITrule	576	192	24.5
Cal5	576	192	25
CART	576	192	25.5
Alba-LM	576	192	25.77
CASTLE	576	192	25.8
Quadisc	576	192	26.2
Alba-GALM	576	192	28.29
Pav-BP			36.45
Alba-GA	576	192	36.46
Alba-GABP	576	192	36.46
SNN-PARDE			37.69

Table 3.13 and Figure 3.40 give the classification error results for the diabetes classification task. The results in the table are listed in increasing order by testing error. This task has a much higher testing error for all methods than both the breast cancer and iris testing sets, indicating that this task is particularly difficult to perform. For this method, NIDA again achieved the third lowest testing error. GP-GaX and GP-SaX outperformed NIDA on this task. GP-SaX uses simulated annealing instead of genetic algorithms to perform crossover, but again, it is a nested optimization algorithm. The same weakness applies to GP-SaX as GP-GaX; it is a method that is restricted to traditional artificial neural network architectures, specifically networks with a single hidden layer, whereas NIDA is a more flexible architecture and training method. Again, NIDA outperformed the spiking neural network methods given, as well as all of the statistical methods reported in [167].

3.6.3 Iris

The iris data set is composed of 150 instances of measurement information of irises [17].[‡] There are four attributes for each instance: sepal length, sepal width, petal length, and petal width. These attributes serve as input to the network. Each of the instances belongs to one of three classes of irises: Iris Setosa (class 0), Iris Versicolour (class 1), or Iris Virginica (class 2). The output of the network corresponds to one of these three class types.

The networks used in this comparison had four input neurons, one corresponding to each attribute. There is a single output neuron. The number of fires of the output neuron in the last 100 times steps of a 500 time step simulation was used to determine class. If the output neuron fired 10 times or less in this time window then the network classified the iris as class 0. If it fired between 10 and 20 times (including 20) then the network classified the iris as class 1. If it fired more than 20 times, then the iris was classified as class 2. The input values were real-valued and not easily translatable

[‡]The iris data set is available at https://archive.ics.uci.edu/ml/datasets/Iris.

to pulse streams, so, as was done in many of the other works using this data set, the values for each attribute were scaled to between 0 and 10 and rounded to the nearest integer. As in the breast cancer and diabetes test cases, if the scaled value is x, then x pulses were applied to the corresponding input neuron, starting at time 0, and spaced five simulated time steps apart. Other encoding schemes can be used.

A population of size 100 was used, with a crossover rate of 0.9 and a mutation rate of 0.99. The evolutionary optimization was allowed to proceed for 100 epochs or until the best fitness was achieved. The fitness function ran all 75 training instances and evaluated how the network performed on those training instances. The fitness value was the fraction correctly classified. A perfect fitness function value corresponds to correctly classifying all instances, which is the second stopping condition of the evolutionary optimization. We note that using all of the training data may lead to over fitting, but because of the small training size, we wanted to utilize as much information as possible during training.

Table 3.14 and Figure 3.41 give the results for the iris classification task. The table lists the results in increasing order by testing error. NIDA achieved the best results for this task, outperforming both GP-GaX and GP-SaX, even though both of those methods used a larger training set. NIDA also outperformed all of the spiking neural network methods. NIDA often achieved perfect classification accuracy for the training set over the course of training, which may be one reason it was able to achieve a higher testing accuracy. Other methods have less restrictive stopping conditions for their training algorithms. This stopping condition is not good to use in general because it may lead to over fitting to the training set. However, NIDA may be less susceptible to that problem than methods using gradient methods for training because of the use of the genetic algorithm.

Table 3.14: Classification error rate on the testing set for the iris classification problem. The number of training and testing instances used are listed; when unknown they are left blank. The results are listed in increasing order of testing error rate. NIDA results are bolded.

Technique	Number Training Examples	Number Testing Examples	Testing Error
NIDA	**75**	**75**	**1.33**
Pav-BP			2.66
SNN-PARDE			2.7
DSSNN	75	75	2.7
SRM SNN	75	75	2.7
GP-GaX	111	39	2.84
GP-SaX	111	39	2.95
GP-StdX	111	39	3.24
SpikeProp	75	75	3.8
SRM SNN (sparse)	75	75	4
MatlabLM	75	75	4
Pav-MBP			4
Pav-SMBP			4
MGNN-ep	111	39	4.68
MatlabBP	75	75	5.4
MGNN-roul	111	39	7.35
MGNN-rank	111	39	7.46

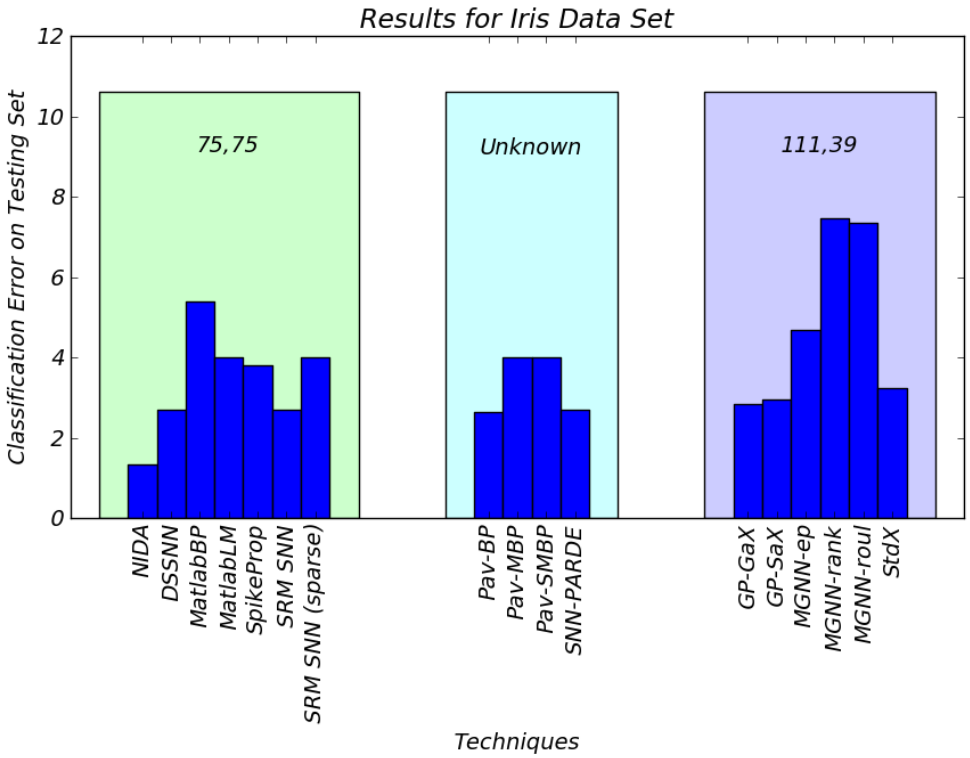


Figure 3.41: Bar chart depicting results for the iris classification task. Results are grouped by the number of training/testing instances, where the number of training instances is listed first and the number of testing instances is listed second. It is also noted if the number of training/testing instances is unknown. Each method is labeled.

3.6.4 Summary

Overall, NIDA performed well on these standard benchmark applications in comparison to other published results that use related architectures or training methods. These applications do not require NIDA's dynamical properties, structural evolution, or even neural networks in order to arrive at satisfactory results. However, we wanted to demonstrate that NIDA could perform well even on these types of static tasks. We found that NIDA performed relatively well overall, consistently placing in the top three best methods for each task. The results also indicate that neuroevolution methods tended to perform better on these tasks than methods that did not allow for structural change. NIDA often outperformed methods in terms of testing accuracy even with a smaller training set. NIDA also performed well with respect to spiking neural network methods. In particular, NIDA outperformed both SpikeProp (a training method for spiking neural networks based on back-propagation) and genetic/evolutionary algorithm-based training methods for spiking neural networks on these tasks.

3.7 Summary

This section describes the structure, design, and performance of neuroscience-inspired dynamic architecture (NIDA) networks. Two multi-process implementations of the design method are described, both of which rely on a server-client system of interaction. We explored the results of NIDA networks on problems from three domains: anomaly detection (recognition of change in mean arrival rate of packets), control (pole balancing), and classification (handwritten digit recognition). An important point to emphasize for NIDA networks is that the same network architecture and design method were used for all three problem types, and relatively little programming was required to develop each individual application. That is, the NIDA software library is built to accommodate many different problem types

with relative ease. This is important to note, because the selection of architecture (e.g., feed forward or recurrent) and network type (e.g., traditional or spiking neural network) for a particular task can often be difficult for a novice user to determine.

We presented a simple training method that optimizes over parameters of neurons and synapses only and found that though good performance is possible, we are much more likely to find a good solution quickly if we allow for structural evolution. We also present results on the effect of the selection of crossover and mutation rates on the performance of the design method on two tasks. We found that the selection of the rate was highly task specific. This indicates that some adaptation mechanism may be useful in finding appropriate crossover and mutation rates for each task, rather than relying on the user to select good rates. We also tested NIDA networks on three standard benchmark classification tasks and compared the results with other published results on spiking neural networks and networks trained with genetic algorithms or evolutionary strategies. We found that our networks performed competitively with these other methods.

There is significantly more analysis that could be done to analyze the architecture and design method. More parallel techniques could be developed, and more parameter analysis could be performed. Different architecture characteristics may be explored, and different tasks may be tested. These future directions and others are discussed further in Section 7.1. In the next two chapters, we explore possible extensions to the NIDA architecture and design method: recognition and reuse of useful substructures (Chapter 4) and the inclusion of affective systems (Chapter 5).

4

Recognition and Reuse of Useful Structures

"There is nothing like looking, if you want to find something. You certainly usually find something, if you look, but it is not always quite the something you were after."

– J.R.R. Tolkien, *The Hobbit*

As noted in Section 2.4, there is evidence of repeated and reused structure in the biological brain [10, 199]. There is also evidence that modularity is one mechanism for the evolution of complex biological organisms [108, 131, 190, 261, 262]. Moreover, modularity has been shown to be beneficial in machine learning [145] (see Section 2.4 for more examples).

We speculated that our evolutionary optimization method would produce networks with useful substructures, where a useful substructure is defined as a subset of neurons and synapses in a network that is integral in some way to the behavior of a network and that network's ability to succeed at the given task. Moreover, we speculated

that identifying these substructures and reusing them in other networks with similar characteristics may improve or escalate the learning process [37]. For this aspect of the work, two major objectives are considered: (1) discover useful substructures in networks and (2) reuse these substructures in the learning process over the course of the evolutionary optimization. This chapter is divided into two sections that discuss these two objectives respectively, the work completed on each, and the results obtained.

The results we have obtained so far for the automatic recognition of useful substructures based on activity and the inclusion of those substructures during the design method as mutations do not support our hypothesis; that is, they did not improve the learning process. We have used small networks that accomplish a subtask as building blocks for larger, more complex networks. Hand-tooling or training networks on a subtask and replicating those structures to perform a larger task may be useful in image and video processing.

4.1 Recognizing Useful Substructures

The first objective was to identify useful substructures in networks produced by the evolutionary optimization. We proposed three major approaches for identifying useful structures: (1) common structures, (2) activity-based structures, and (3) causality paths.

4.1.1 Common Structures

One way of identifying useful structures for a task is to look for substructures that occur in many networks that are successful for a particular problem. If a structure appears in many of these networks, it is likely useful (or perhaps necessary) in order to solve the task at hand.

Each of these networks that performed well in our anomaly detection task (described in Section 3.3.1) has two neurons that are fixed: input and output. The structures we chose to search for were one of three types: (1) paths from input to output, (2) cycles including the input neuron, and (3) cycles including the output neuron. We chose paths from input to output as one structure of interest because these are the paths that are followed in order for meaningful output to be produced. We chose to search for cycles because recurrent connections are very important in this application. Since the input and output neurons are the two fixed neurons in our system, we search for cycles involving at least one of those neurons. Some cycles will involve both input and output.

In searching for commonalities among the structures, we categorized the paths and cycles based on their pattern of inhibitory and excitatory connections. When looking at a cycle from input to input (similarly, output to output), we further categorize the cycles by the position, if any, of the output (input) neuron in that cycle. We discovered that there were many of these structures that were common across several of the networks.

Figures 4.1 and 4.2 show all of the structures that appeared in at least eight of nine networks. Figure 4.3 shows all of the structures that appeared in seven of nine networks. For each structure, a histogram showing the distances of the paths/cycles fitting this structure found in each network. Since the substructure may appear more than once in each network, all substructures that fit these criteria from each network are included in the histogram.

These commonalities across networks imply that these structures likely play a large role in the function of the network. For example, the last cycle given in Figure 4.3 shows that, in seven of the nine networks, there is an excitatory connection from the output neuron to another neuron, followed by a second excitatory connection, followed by an inhibitory connection back to the output neuron. Many of the subnetworks that fall into this type have a cycle-length or delay of between 100 and 150. Our fitness function specified that an output neuron should only fire during a period of 100 time

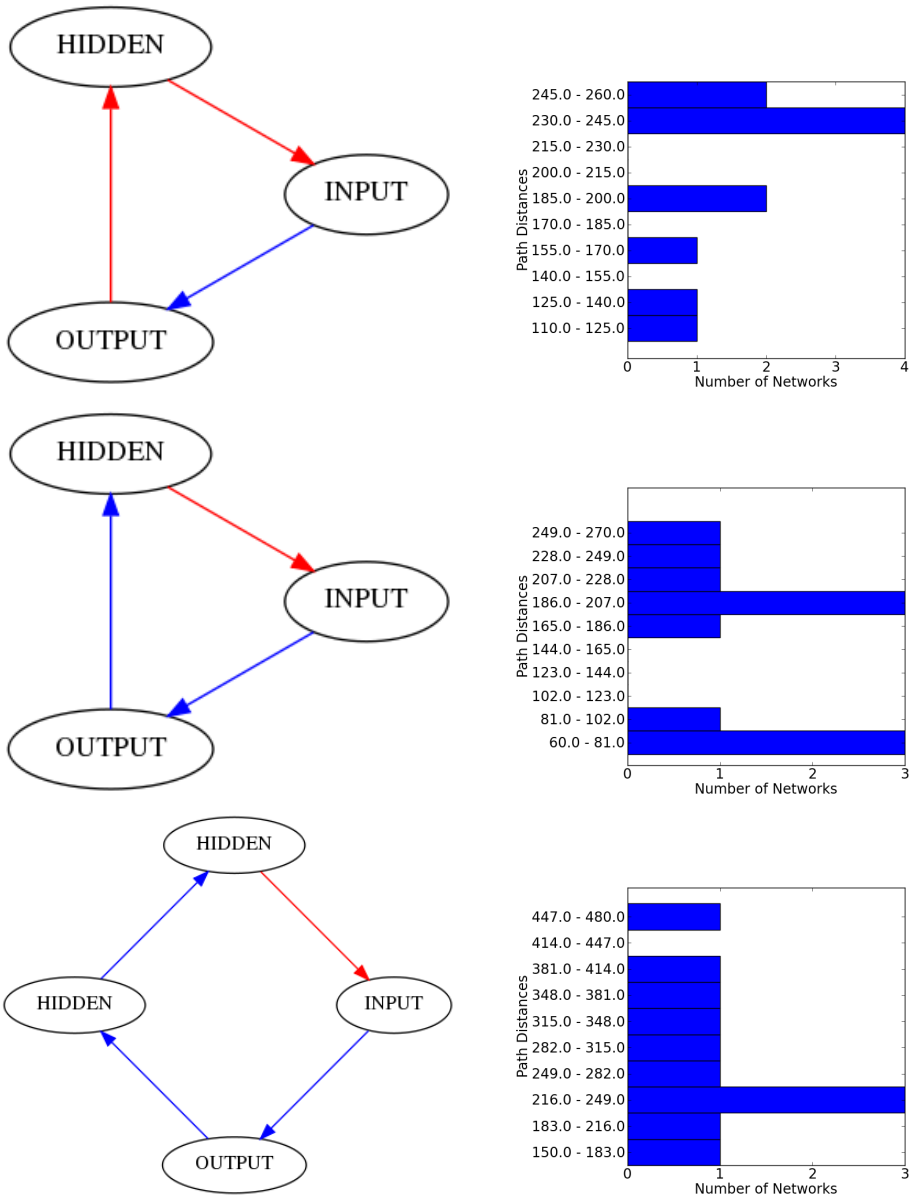


Figure 4.1: Common structures (cycles or paths) appearing in at least eight of nine networks that performed well in detecting increases in mean packet arrival rate. Red synapses are inhibitory and blue synapses are excitatory. A histogram showing the distances of the path/cycle is also given for each common path/cycle. More common paths are in Figure 4.2.

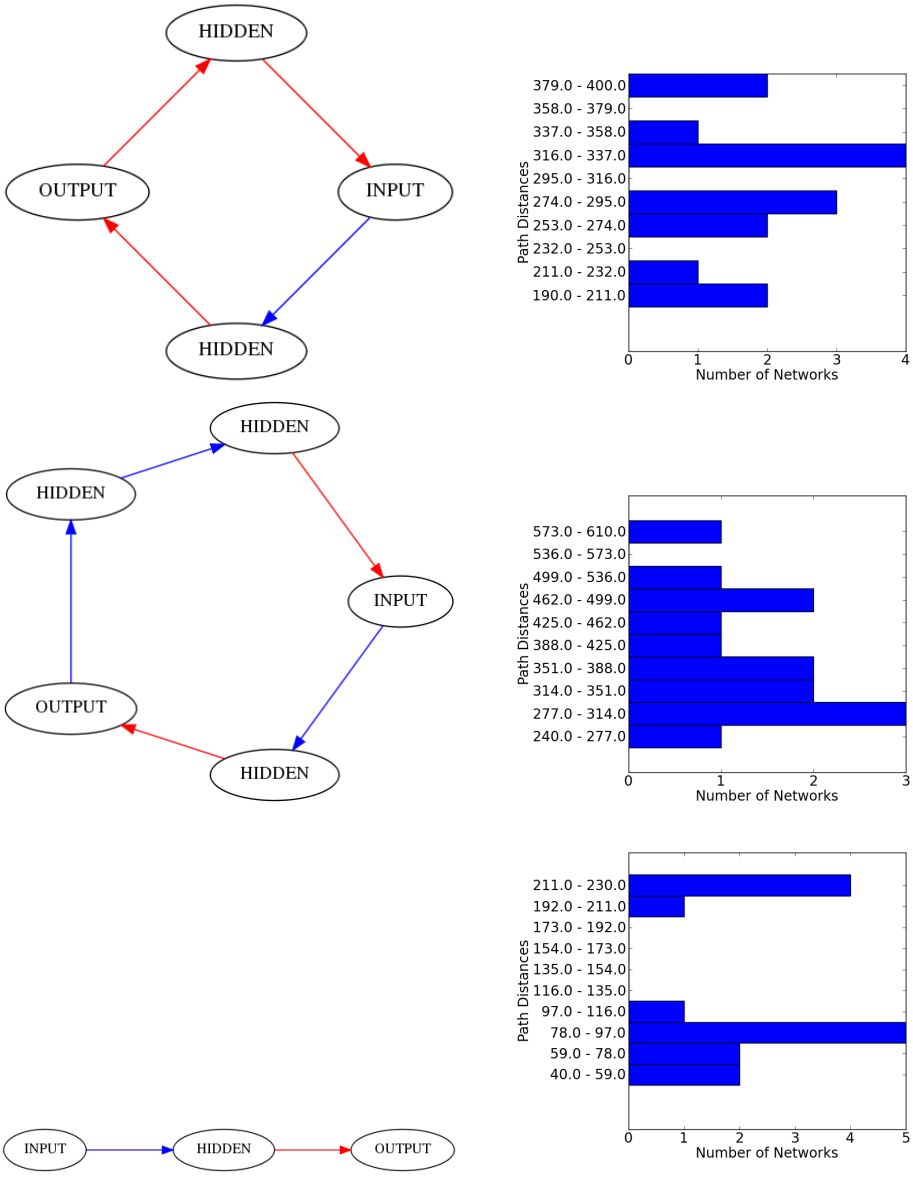


Figure 4.2: Common structures (cycles or paths) appearing in at least eight of nine networks that performed well in detecting increases in mean packet arrival rate. Red synapses are inhibitory and blue synapses are excitatory. A histogram showing the distances of the path/cycle is also given for each common path/cycle. More common paths are in Figure 4.1.

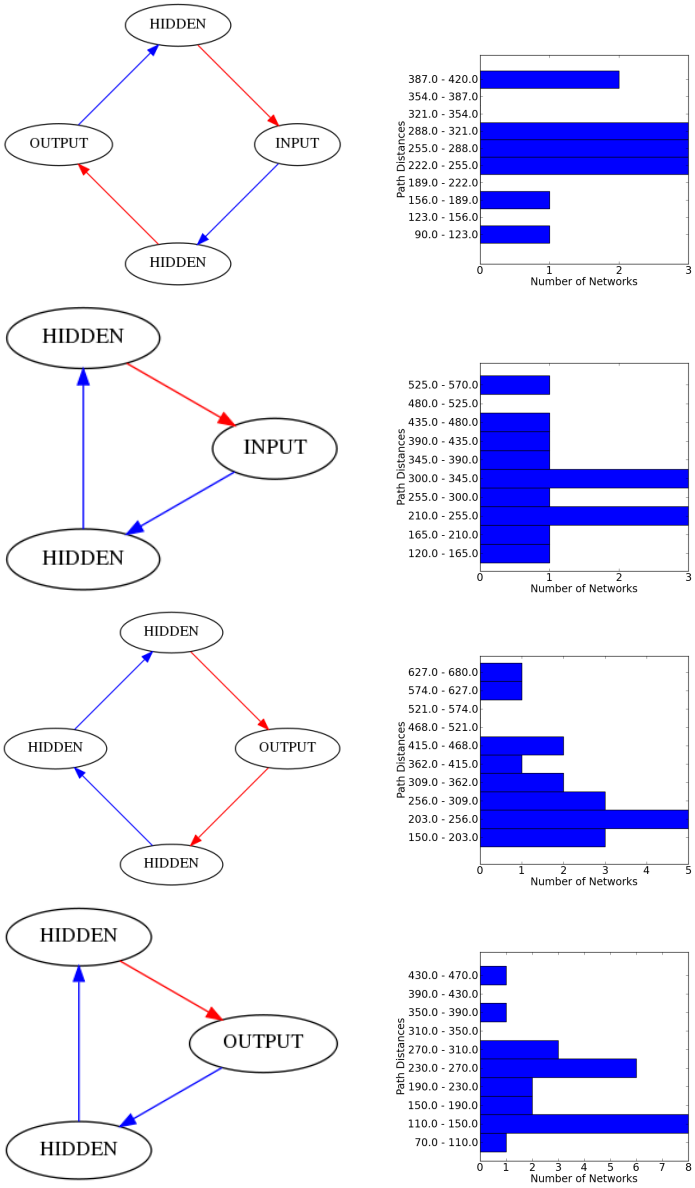


Figure 4.3: Common structures (cycles or paths) appearing in seven of nine networks that performed well in detecting increases in mean packet arrival rate. Red synapses are inhibitory and blue synapses are excitatory. A histogram showing the distances of the path/cycle is also given for each common path/cycle.

steps after an increase of mean arrival rate. For this structure, this means that if the output neuron fires, it increases the charge on the first hidden neuron, which, in turn, may fire and increase the charge on the second hidden neuron. If that neuron fires, it *decreases* the charge on the output neuron, making it less likely to fire.

Beyond those shown in Figures 4.1, 4.2, and 4.3, there were six unique structure types that appeared in exactly six of these networks, eight structures that appeared in exactly five of the networks, 17 that appeared in exactly four of the networks, and 51 that appeared in exactly three of these networks.

The distributions of distances for the substructures implies that it may be necessary to further refine the search for common structures by distance. We also did not consider the magnitudes of the weight on the synapses in our useful structure search. It may be helpful to refine the search using this mechanism. It would also be useful to look for more types of structures (other than paths and cycles) in the network.

We performed similar structural analyses on the handwritten digit networks that scan the images by column (described in Section 3.3.3). We allowed 500 instances of the evolutionary optimization for each digit type to run for two hours and then compared the best performing networks from each of these instances. In the handwritten digit optimization, we allowed for mutations of the thresholds of the neurons; this makes it more difficult to compare structures in the network, as the weight values on synapses no longer have the same meaning when the thresholds of the neurons differ. To accommodate for this change, we adjusted all thresholds to zero and adjusted weights in the network as well. For example, suppose neuron n has threshold t. We calculated d such that $t+d=0$ (or $d=-t$). Then, we added d to the threshold of neuron n and also added d to all of the weights on the synapses coming into n. This way, we know that negative scaled weights correspond to synapses that do not reach the network's threshold and positive scaled weights are synapses that would cause the neuron's threshold to be exceeded. For these networks, we found all paths from each input neuron to the output neuron of the network such that the delay of the

path is less than 500 (the number of simulated time steps the network runs for each image). We then looked at the scaled weight values of each synapse for all of these networks and looked for commonalities in patterns. For example, an input/output pair is (Input 0, Output) and one possible path type for this input/output pair is PN, which is a path that consists of two synapses, where the scaled weight values for the synapses on that path are positive (P) for the first and negative (N) for the second.

Figures 4.4 through 4.13 show the most common directed paths from input to output for each digit type, along with a histogram of the delays of the paths. Blue lines between nodes indicate positive scaled weight values and red lines indicate negative synapse weight values. For networks recognizing 0's, the two most common path types had very similar delay distributions and the same path structure, but for two different (though relatively close) input neurons, indicating that those two paths may play the same role in these networks. Each of the paths in these figures contain three neurons: an input neuron, a hidden neuron, and an output neuron. Longer common paths with delays of less than 500 that appeared in more than five percent of the networks were not found.

Networks recognizing 1's had the most commonalities in paths (shown in Figure 4.5); there were many more paths that appeared in at least 5 percent of the networks recognizing 1's, but only the top four are shown. Three of the top four paths have negative scaled weights followed by positive scaled weights and had similar delay distributions. These paths are likely indicators of non-one images, as they inhibit a hidden neuron that, upon firing, would have cause the output neuron to fire.

Figure 4.9 shows that, in 33 of the networks, input neuron 6 firing is likely to also cause the output neuron to fire (as indicated by two positive scaled weights or blue synapses). Since a single pixel value at any given point is not likely to specify one digit or another, it is probable that there is extra inhibitory structure in these networks (though different) that keeps the hidden neuron from firing in the presence of other pixels.

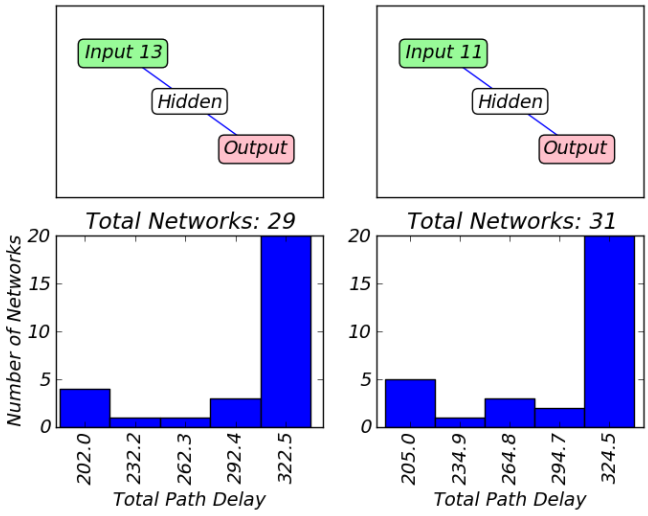


Figure 4.4: Common paths from input to output for networks recognizing the digit 0 in images. The total number of networks in which that path appears is also given, along with a histogram of the delays for that path.

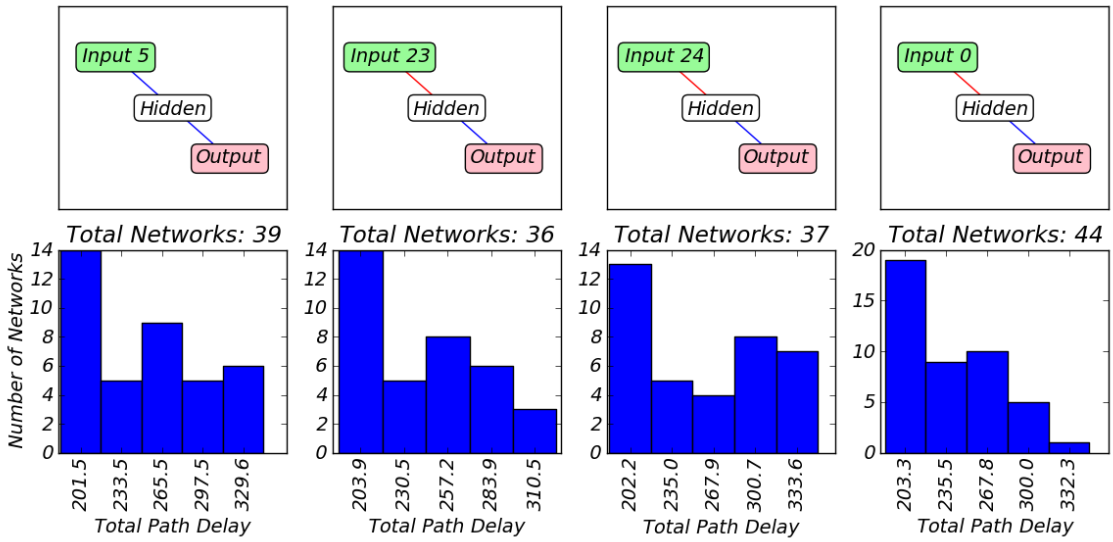


Figure 4.5: Common paths from input to output for networks recognizing the digit 1 in images. The total number of networks in which that path appears is also given, along with a histogram of the delays for that path.

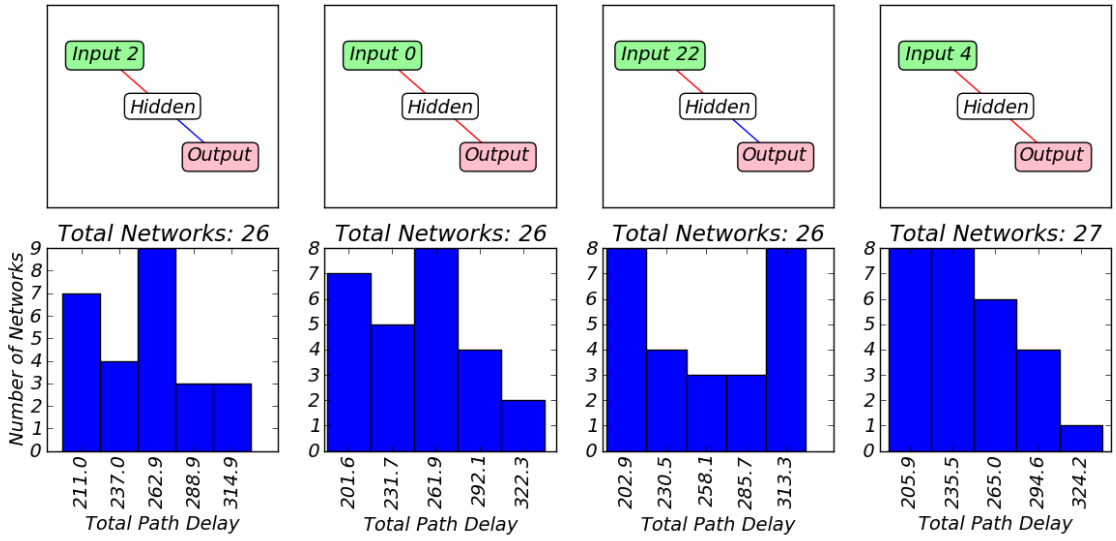


Figure 4.6: Common paths from input to output for networks recognizing the digit 2 in images. The total number of networks in which that path appears is also given, along with a histogram of the delays for that path.

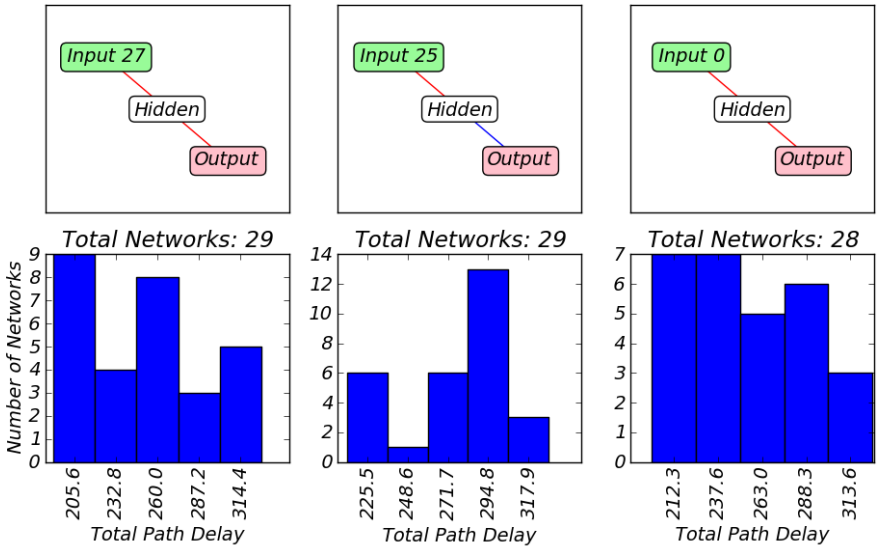


Figure 4.7: Common paths from input to output for networks recognizing the digit 3 in images. The total number of networks in which that path appears is also given, along with a histogram of the delays for that path.

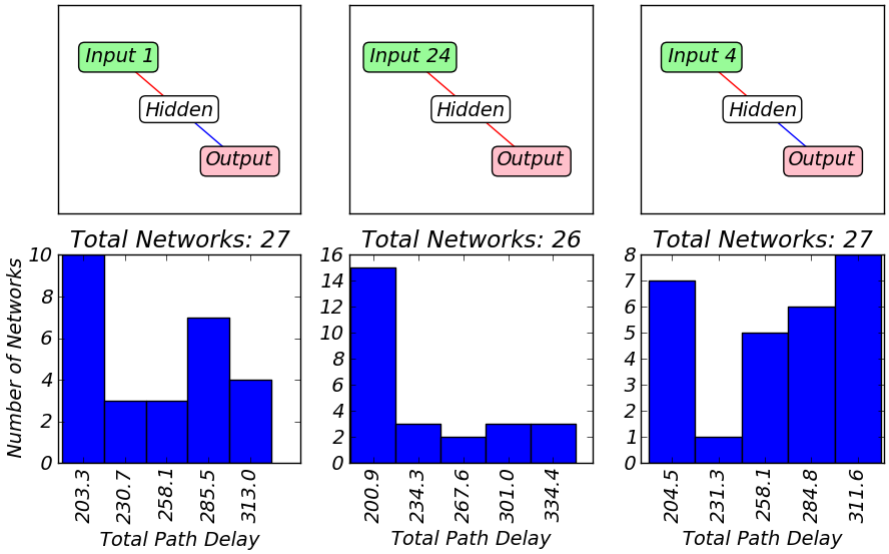


Figure 4.8: Common paths from input to output for networks recognizing the digit 4 in images. The total number of networks in which that path appears is also given, along with a histogram of the delays for that path.

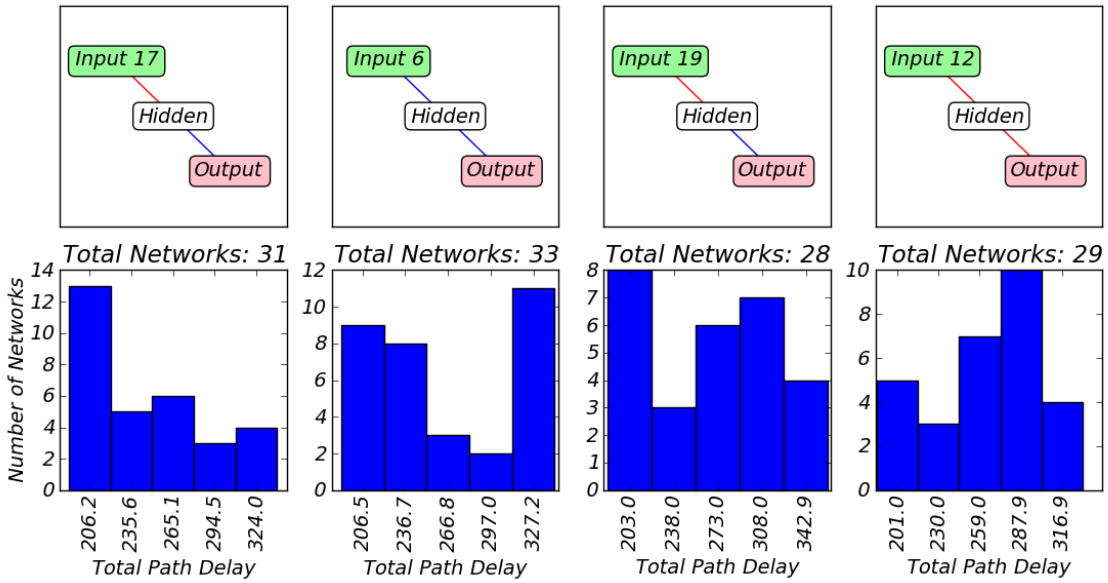


Figure 4.9: Common paths from input to output for networks recognizing the digit 5 in images. The total number of networks in which that path appears is also given, along with a histogram of the delays for that path.

There was only one path that appeared in more than 5 percent of the networks recognizing 6's. This is another path for which the hidden neuron firing would cause the output neuron to fire and the input neuron firing decreases the likelihood of the hidden neuron to fire, thus reducing the likelihood that the output neuron would fire. The input appearing in this path is input 1, which corresponds to pixels in the first row (in which the rows are 0-indexed). Since there should only be one or two pixels in the first row for a six image, it is likely that this path is active for non-six images where there would be more pixels in the first row (for example, any digit in which there is a curve or line across the top, such as 0, 3, 5, 7, 8, and 9).

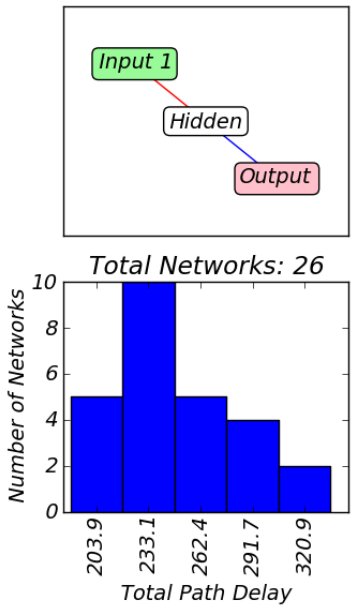


Figure 4.10: Common path from input to output for networks recognizing the digit 6 in images. The total number of networks in which that path appears is also given, along with a histogram of the delays for that path.

There are a variety of issues with finding common structures through graph analysis. One issue is that, to get truly meaningful network structures, a large number of networks need to be analyzed. Graph analysis is notoriously computationally difficult, and many of the algorithms required to search for certain substructures in graphs are NP. That is, using the known algorithms, the time required to find these

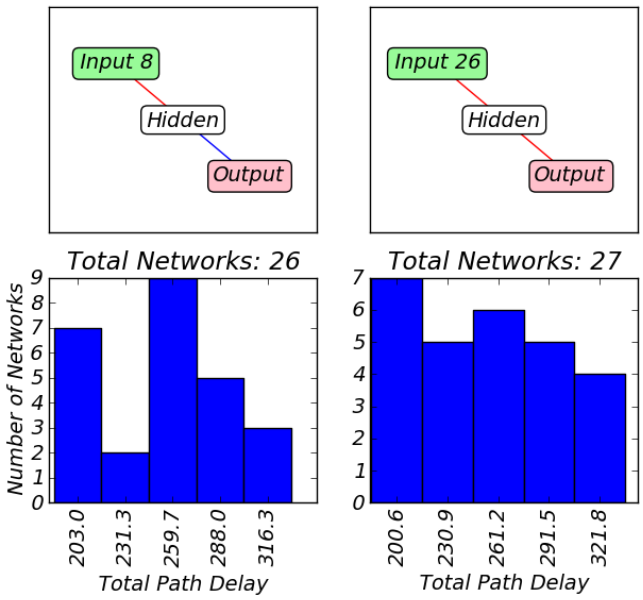


Figure 4.11: Common paths from input to output for networks recognizing the digit 7 in images. The total number of networks in which that path appears is also given, along with a histogram of the delays for that path.

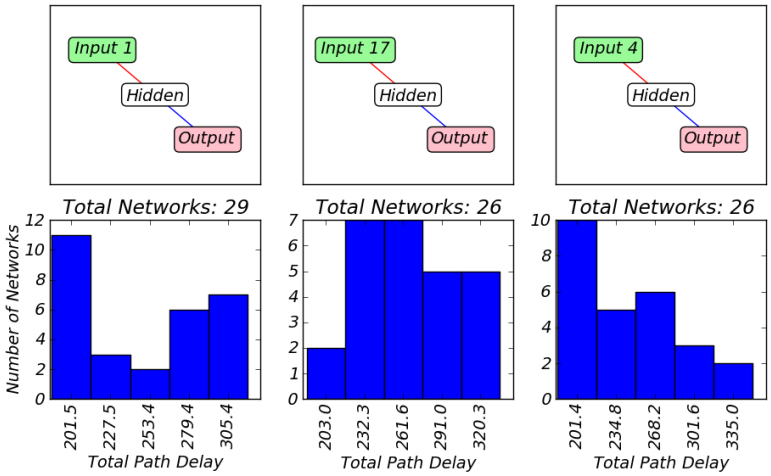


Figure 4.12: Common paths from input to output for networks recognizing the digit 8 in images. The total number of networks in which that path appears is also given, along with a histogram of the delays for that path.

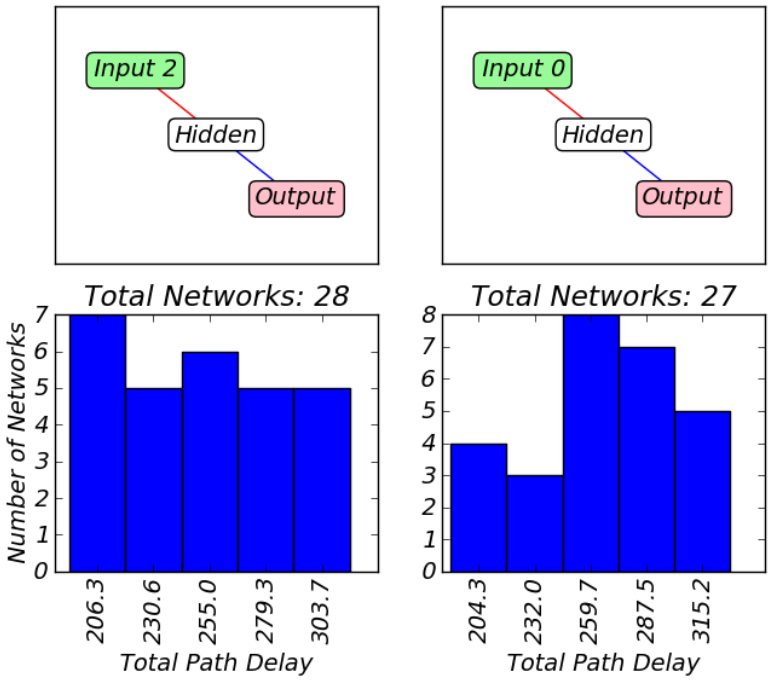


Figure 4.13: Common paths from input to output for networks recognizing the digit 9 in images. The total number of networks in which that path appears is also given, along with a histogram of the delays for that path.

substructures grows at least exponentially with the size of the graph. Moreover, it is unclear what types of substructures should be analyzed for each problem type.

4.1.2 Activity-Based

The previous approach for determining useful substructures for a task requires many networks that perform well on a particular task. Another approach we propose for finding useful substructures requires a single network. For this approach, we simulate the network on one or more instances of the task and determine if there are different parts of the network that contribute to the network's operation. For the anomaly detection task (Section 3.3.1), we note that the environment can be in one of several states. For example, a few possible states are: a change in mean arrival rate has not occurred, a change in mean arrival rate has occurred in the last 100 time steps, and a change in mean arrival rate has occurred but it occurred more than 100 times steps previously.

If we can isolate parts of the network that are active during these various environmental states, we may be able to determine which part of the network is useful for which portion of the task. To accomplish this, the network must be simulated on an instance of the task, and every firing event and addition of charge event in the network must be recorded over the course of simulation.

Figures 4.14 through 4.19 show one simulation of an instance of the anomaly detection task and one network that performs well on that task at various points in the simulation. The behavior of the network is recorded for various environmental states (shaded in the plot of the mean arrival rate of pulses). The neurons are shaded based on how often they fire in the time window, and synapses are highlighted if charge travels along that synapse during the specified time window. The plot of the mean arrival rate also shows when the output neuron fires in the simulation.

We are able to see that different parts of the network contribute to the behavior of the network for different states of the environment. Specifically, different parts of the

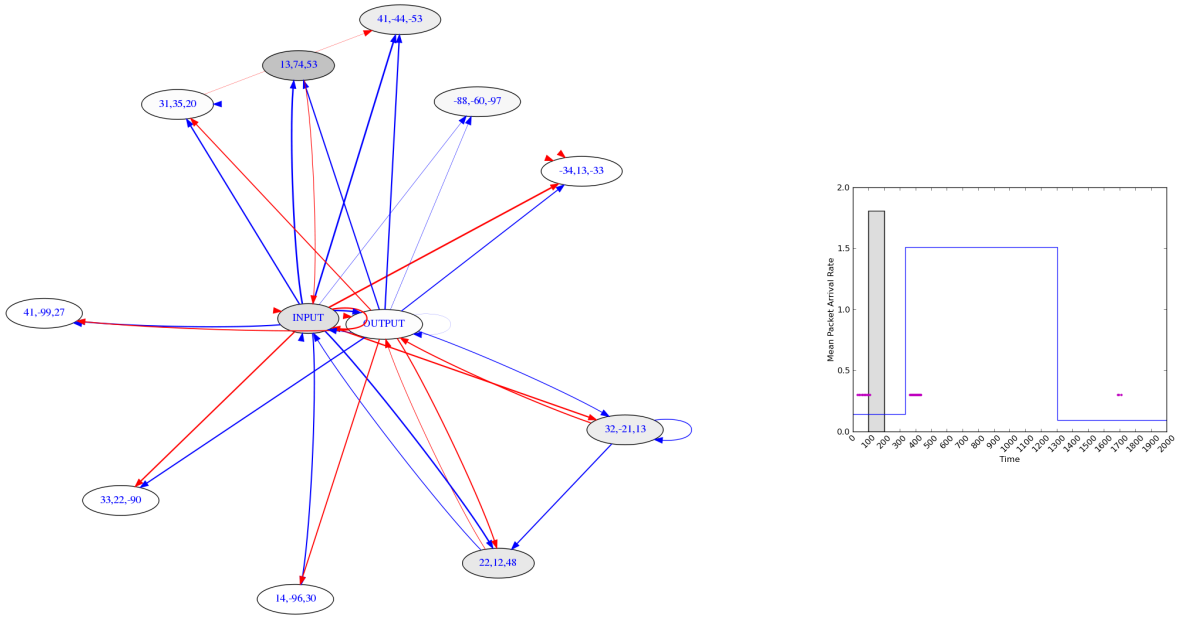


Figure 4.14: Network activity and simulation plot for time steps 100 to 200, when the simulation has just started.

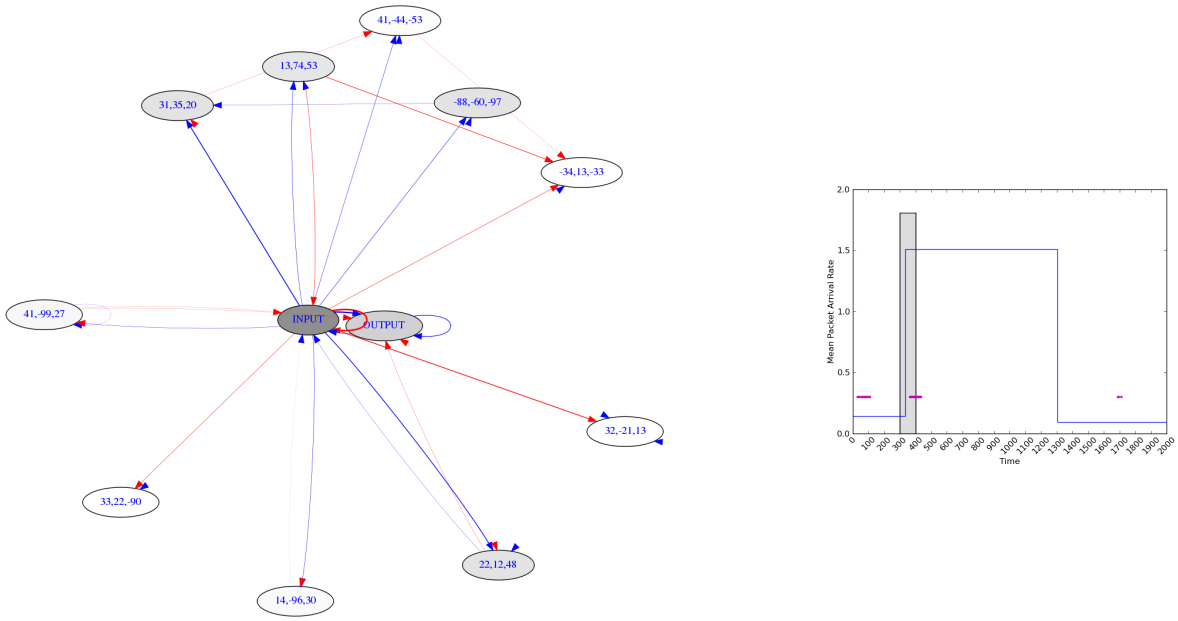


Figure 4.15: Network activity and simulation plot for time steps 300 to 400, when an increase in mean arrival rate has occurred.

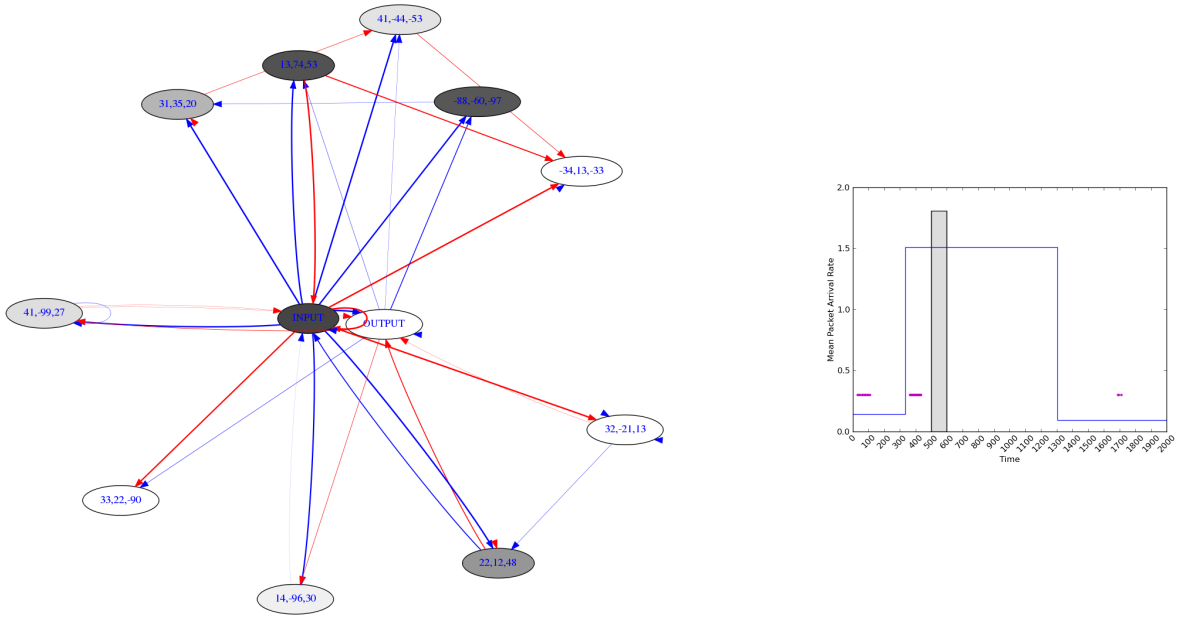


Figure 4.16: Network activity and simulation plot for time steps 500 to 600, when an increase in mean arrival rate has occurred more than 100 time steps previously.

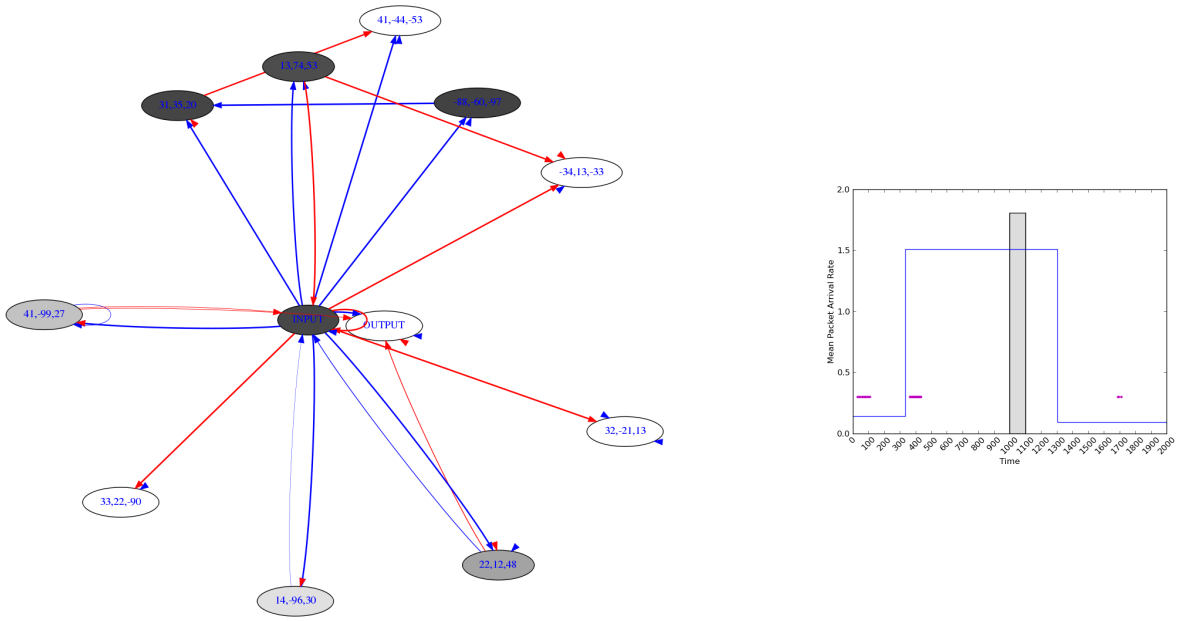


Figure 4.17: Network activity and simulation plot for time steps 1000 to 1100, when an increase in mean arrival rate has occurred more than 100 time steps previously.

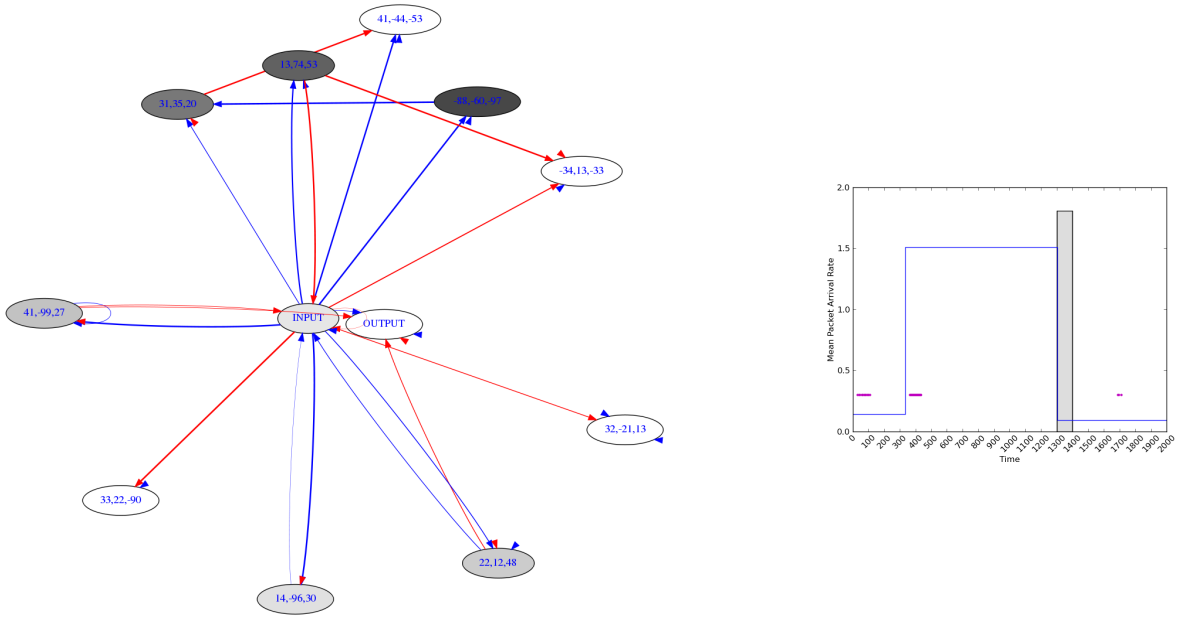


Figure 4.18: Network activity and simulation plot for time steps 1300 to 1400, when a decrease in mean arrival rate has just occurred.

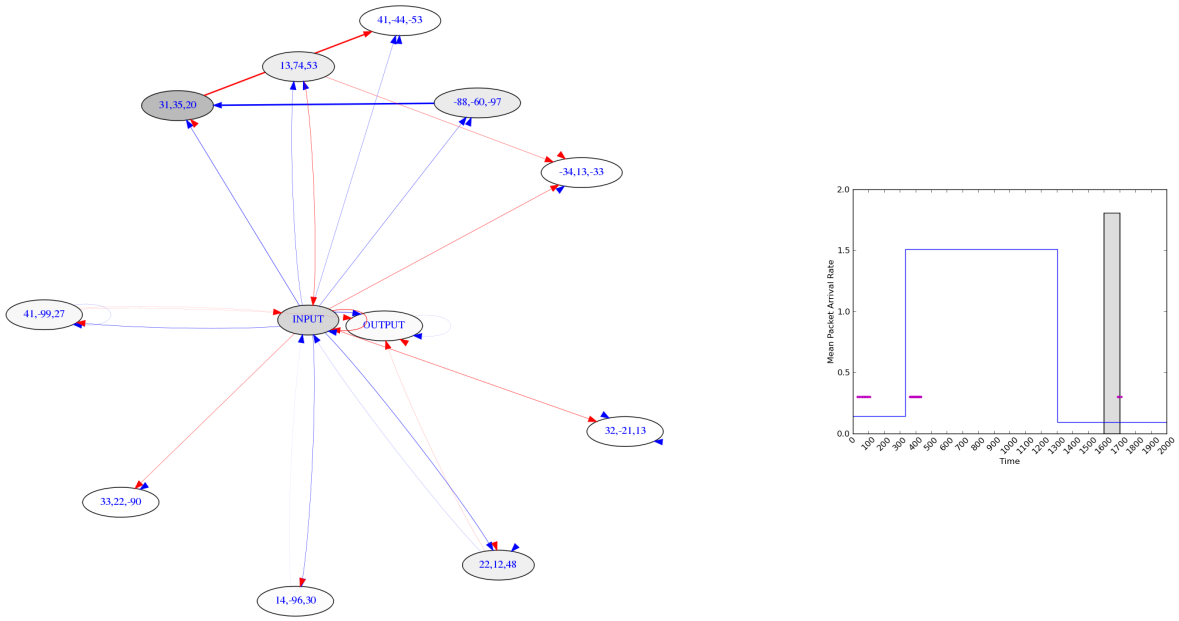


Figure 4.19: Network activity and simulation plot for time steps 1600 to 1700, when a decrease in mean arrival rate has occurred more than 100 time steps previously.

network are controlling different aspects of the network's behavior. Similar results occur for other networks that perform well on this task. Any of these behaviors may be useful for other tasks, so the automatic extraction of these substructures may be useful.

A new visualization tool was developed by Margaret Drouhard to better observe the activity of NIDA networks [35, 70]. Using the new visualization tool, we are also able to identify useful substructures based on activity. For the handwritten digit task described in Section 3.3.3, we observed a network designed to recognize zeros on several different zero images. We were able to observe a substructure that fired noticeably more than other neurons in the network. This substructure is highlighted in Figures 4.20 and 4.21.

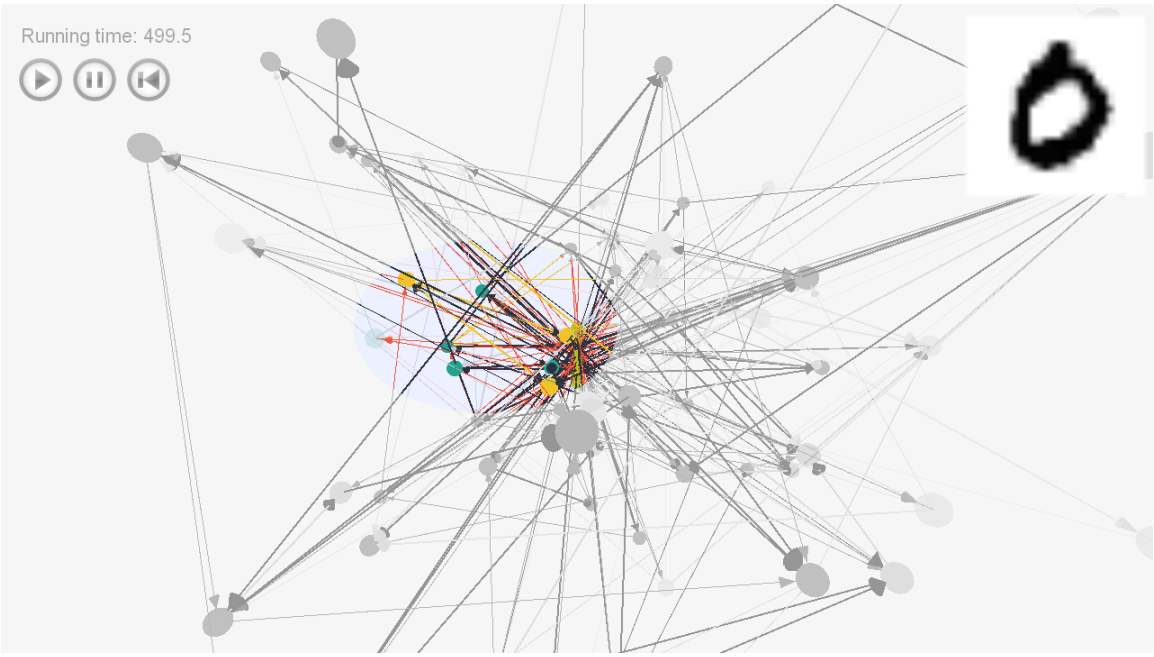


Figure 4.20: A substructure (shown in yellow in the highlighted portion of the network) that is noticeably more active than other neurons in the network when processing the shown handwritten image of a zero. (Figure created by Margaret Drouhard and included with her permission.)

To obtain a better idea of how often neurons in this network were firing, an analysis of the number of times each hidden neuron in the network fires over all testing images

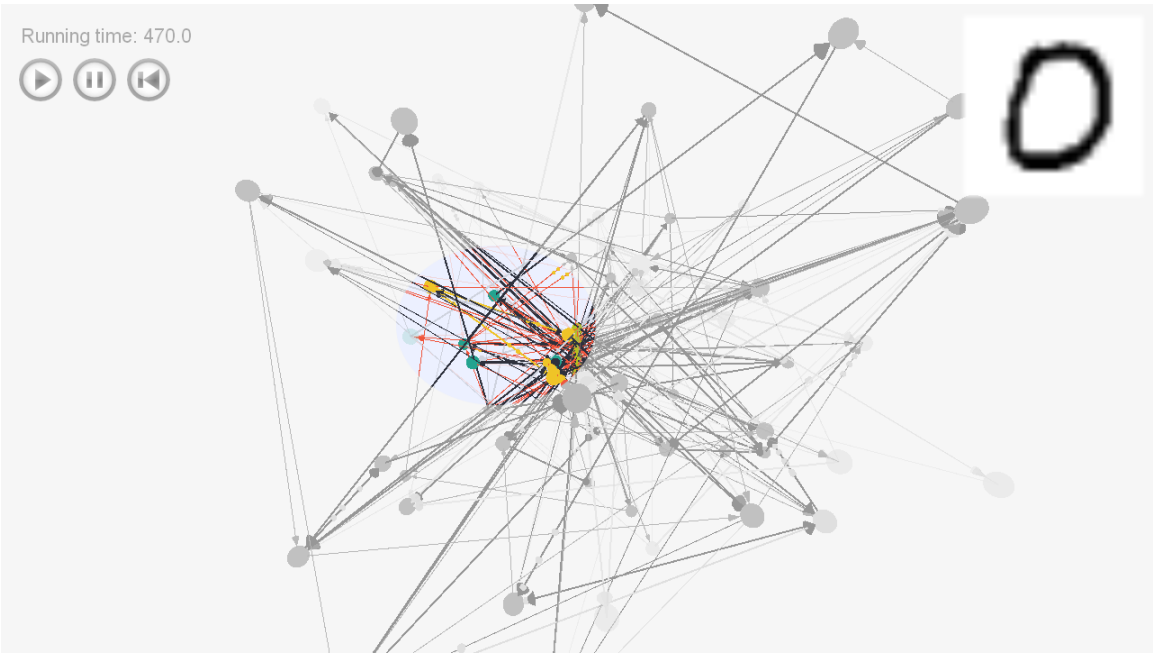


Figure 4.21: The same substructure as in Figure 4.20 that is, again, noticeably more active than other neurons in the network when processing the shown handwritten image of a zero. (Figure created by Margaret Drouhard and included with her permission.)

of zeros was performed. The results of this analysis are given in Figure 4.22. The three neurons that fire noticeably more than any other neurons in the network for images of zeros are the same three neurons highlighted in Figures 4.20 and 4.21. As such, it is likely that these three neurons are integral to the success of the network in processing images of zeros.

This result is very encouraging, as it is much easier computationally to identify structures based on firing than it is to look for common substructures. This computation can be completed using a single network (rather than many networks) and can be done over the course of a single evolutionary optimization to identify useful substructures. We may also look across multiple networks to see if a similar structure exists within those networks.

4.1.3 Causality Paths

Each fire event or change-in-charge event in the network is created as a result of one or more other fire events, change-in-charge events or pulse events in the simulation. Pulse events are only visible at the input neuron and are only generated by the environment. For any fire or change-in-charge event in the network, we can trace back to find the pulse event(s) that caused that behavior. Since each fire event has an associated neuron and each change-in-charge event has an associated synapse, we can form a path containing the neurons and synapses that are associated with the precipitating events for any particular fire or change-in-charge event. We call these paths "causality paths." Much of the results in this section are documented in [70].

We have explored causality paths with respect to the networks generated for the handwritten digit problem. Figure 4.24 is a path extracted from the activity of network N, a handwritten digit recognizer trained to recognize the digit 7 during the processing of an input image of a 7. The figure shows the first firing of the output neuron during the final time window, signaling recognition of the digit 7. In contrast, Figure 4.25 shows a path drawn from the same network during the processing of an

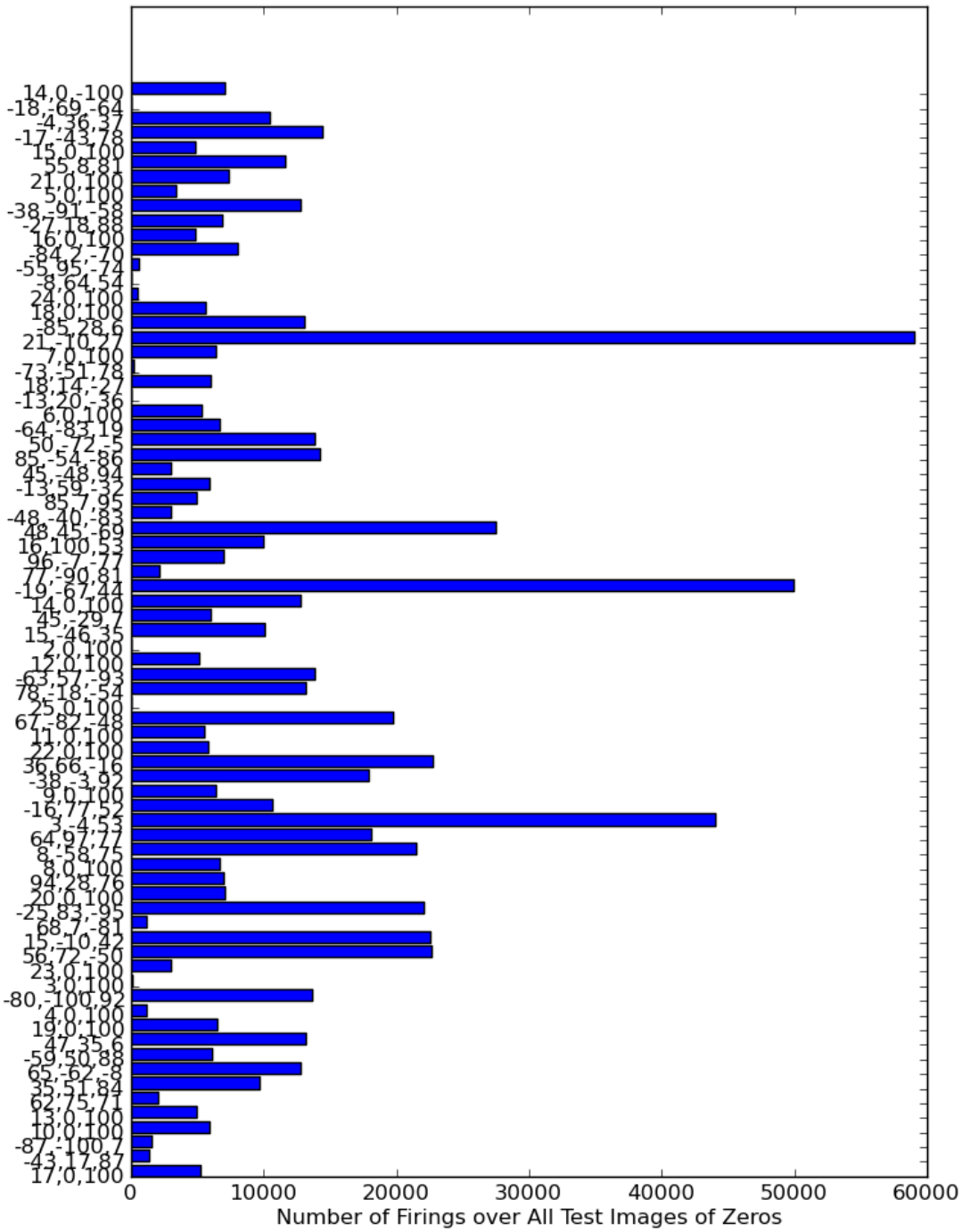


Figure 4.22: Activity of all neurons in the network depicted in Figures 4.20 and 4.21. The y-axis gives the neurons labeled by position in a three-dimensional cube. The three most active neurons are those contained in substructure s, highlighted in 4.20 and 4.21.

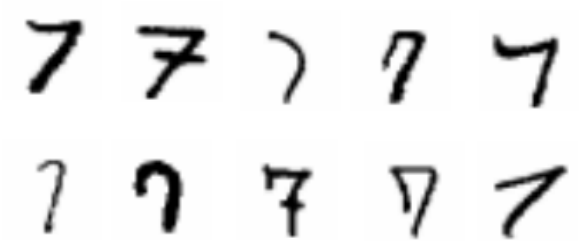


Figure 4.23: Example images of written 7's. The variations in the way 7's are drawn may contribute to the variation in causality paths that occur.

input image of the digit 2, which has similar features to images of sevens. The path in Figure 4.25 includes the final firing of the output neuron at time 391.389. The occurrence of this last fire prior to the beginning of the final time window indicates that the network recognizes that the input image is not a 7. The network behavior was similar for multiple input images of the digit 2. The final firing of the output neuron could be traced back to the input pulse along the same relatively short two-segment path. The final firing propagated charge along inhibitory synapses. Other input images, such as those of the digit 1, triggered different activity, but the paths to the final firing tended to be short and to trigger more inhibitory behavior. The paths for correct recognition of the digit 7 tended to vary more, but were longer overall, as could be expected since the fire to indicate recognition of d was designed to occur within the final 50 time units. Some of these paths were cyclical, unlike the paths observed for non-d digits. The variation in paths for images of 7 may be attributable to the variations in ways 7's can be written, as shown in Figure 4.23.

The causality paths appearing in Figures 4.24 and 4.25 provide further intuition about how networks of this type operate. Based on these results, we can speculate that shorter paths to the final firing of the output neuron result from the relative ease of identifying an image as a non-d digit as compared to the paths that result when identifying an image as a d digit. That is, it is easier (and requires less complicated structure) for the network to determine that an image is not of a d than it is for the network to definitively say that the image is of a d. Table 4.1 gives the classification results of one of these networks in isolation (a network trained to recognize images

Table 4.1: Accuracy breakdown for a network trained to recognize images of the digit 7.

Digit	Accuracy
0	99.4898
1	99.9119
2	97.6744
3	90.8911
4	97.4542
5	92.9372
6	99.791
7	79.3574
8	94.5585
9	77.106

of 7's). In particular, this table shows that for non-7 images of digits other than 9's, the network achieves higher than 90 percent accuracy (that is, does not fire in the last time window for these images), whereas it only achieves around 80 percent for images of sevens. The low accuracy rate for 9's may be attributed to the similarities in the ways 7's and 9's are written.

We further explored the use of causality paths by examining the best performing network for each digit in the handwritten digit task (scanning by column and trained using fitness function 1 as described in Section 3.3.3). We found causality paths for each firing event in the last time window for the first 100 testing set images of the desired digit for each network. Figures 4.26 through 4.28 show these causality paths in the networks where the size of the neuron/synapse represents how often that neuron/synapse appeared in a causality path. In these figures, input neurons are colored yellow-green, the output neuron is colored red-orange, and the hidden neurons are teal. The inhibitory synapses are red and the excitatory synapses are navy. These causality paths show a much greater level of complexity than those shown in Figure 4.24, partially because they represent all fires in the last window and a larger number of inputs. This indicates that there are many different potential paths that contribute to firing of the output neuron in the networks, which are more

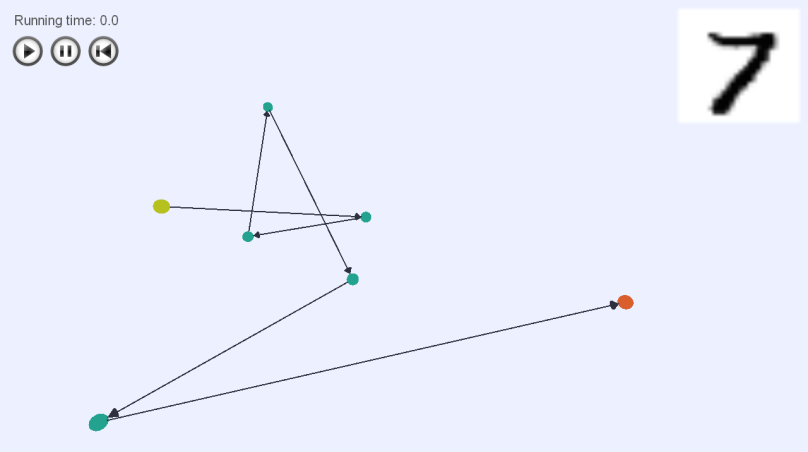


Figure 4.24: Event path from a network trained to recognize the digit 7 processing an input image of a 7. The event traced by the path is the first firing of the output neuron within the final time window, which signals that the network correctly identifies the image as a 7. (Figure created by Margaret Drouhard and included with her permission.)

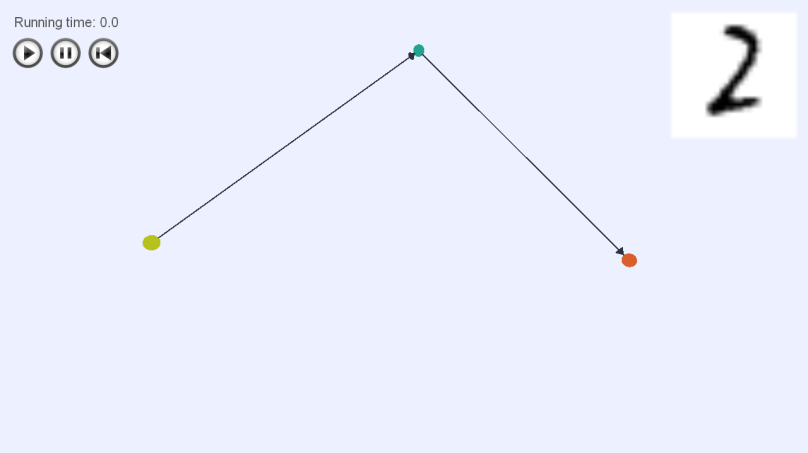
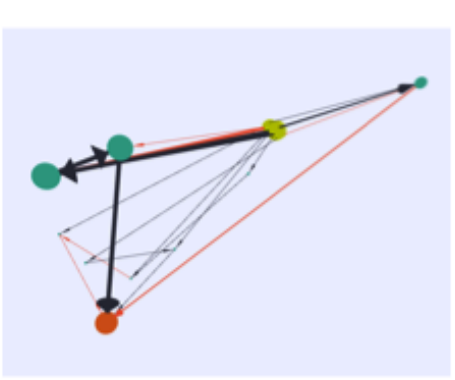


Figure 4.25: Event path from a network trained to recognize the digit 7 processing an input image of the digit 2. The event traced by the path is the last output neuron fire. The fire occurs prior to the final time window, indicating that the network recognizes that the image does not show the digit 7. (Figure created by Margaret Drouhard and included with her permission.)

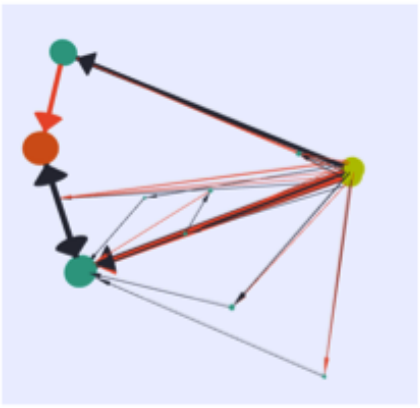
consistent with our expectations. Moreover, it is worth noting that many of the neurons/synapses in these networks appear in a large number of paths (as indicated by the size of the neurons/synapses in the figures). Thus, there are a relatively small set of neurons and synapses in the network that contribute to the recognition of the digit.

These figures do not, however, give any indication of how the network works when an image is not recognized as a particular digit (i.e., the output neuron does not fire in the final time window). One could build causality paths for these types of inputs by tracing all inhibitory events on the output neuron. However, these types of events may not occur in the course of a simulation or if they do occur, they may not be contributing to the neuron's lack of firing. For example, Figure 4.28(b) shows a network that recognizes 9's in which the two main synapses coming into the output neuron are inhibitory. In this network, the threshold of the output neuron is negative, so these inhibitory synapses actually cause the output neuron to fire. It may also be the case that there are no inhibitory events on the output neuron. That is, it may be that the inhibitory events occur at hidden neurons in the network, causing those neurons not to fire and thus causing the output neuron not to fire. In any case, searching for causality paths for a lack of events is much harder than the alternative. We also note that even the some of the causality paths in Figures 4.26 through 4.28 are relatively complex network structures, which means that there is even more structure in the full network that contributes to a lack of firing of the output neuron.

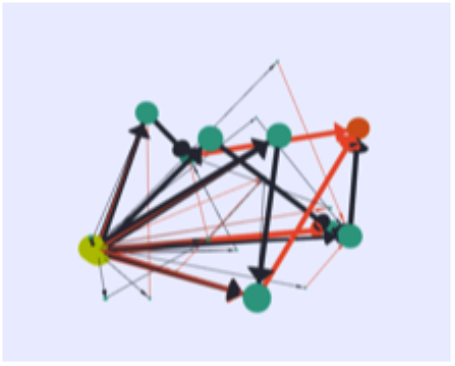
Causality paths are helpful in understanding what structure in the network is important in producing the functionality of the network. They are another automated way to track useful substructures that may be exploited during the evolutionary optimization method. However, it can be computationally and memory intensive to maintain every event and backtrace to develop causality paths during optimization. Moreover, a causality path is obtained by choosing one or more events to backtrace. This choice is usually task-specific and would have to be specified by the user, who may not know what events are important for a particular task.



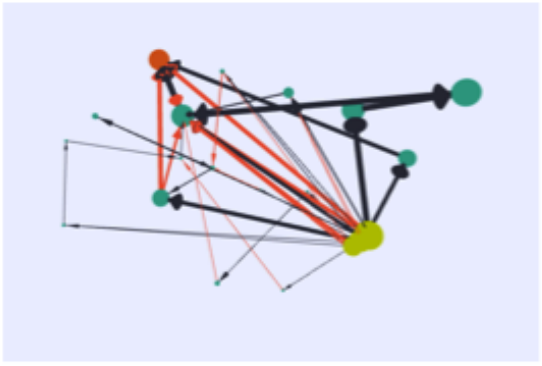
(a) 0 Recognizer



(b) 1 Recognizer



(c) 2 Recognizer



(d) 3 Recognizer

Figure 4.26: Subnetworks of networks trained to recognize various digits (0-3) in handwritten digit images. In these subnetworks, the neurons and synapses are sized based on the number of times they appear in a causality path for a firing event in the last 50 time steps of the simulation, which signifies a detection of a particular image. (Figure created in part by Margaret Drouhard and included with her permission.)

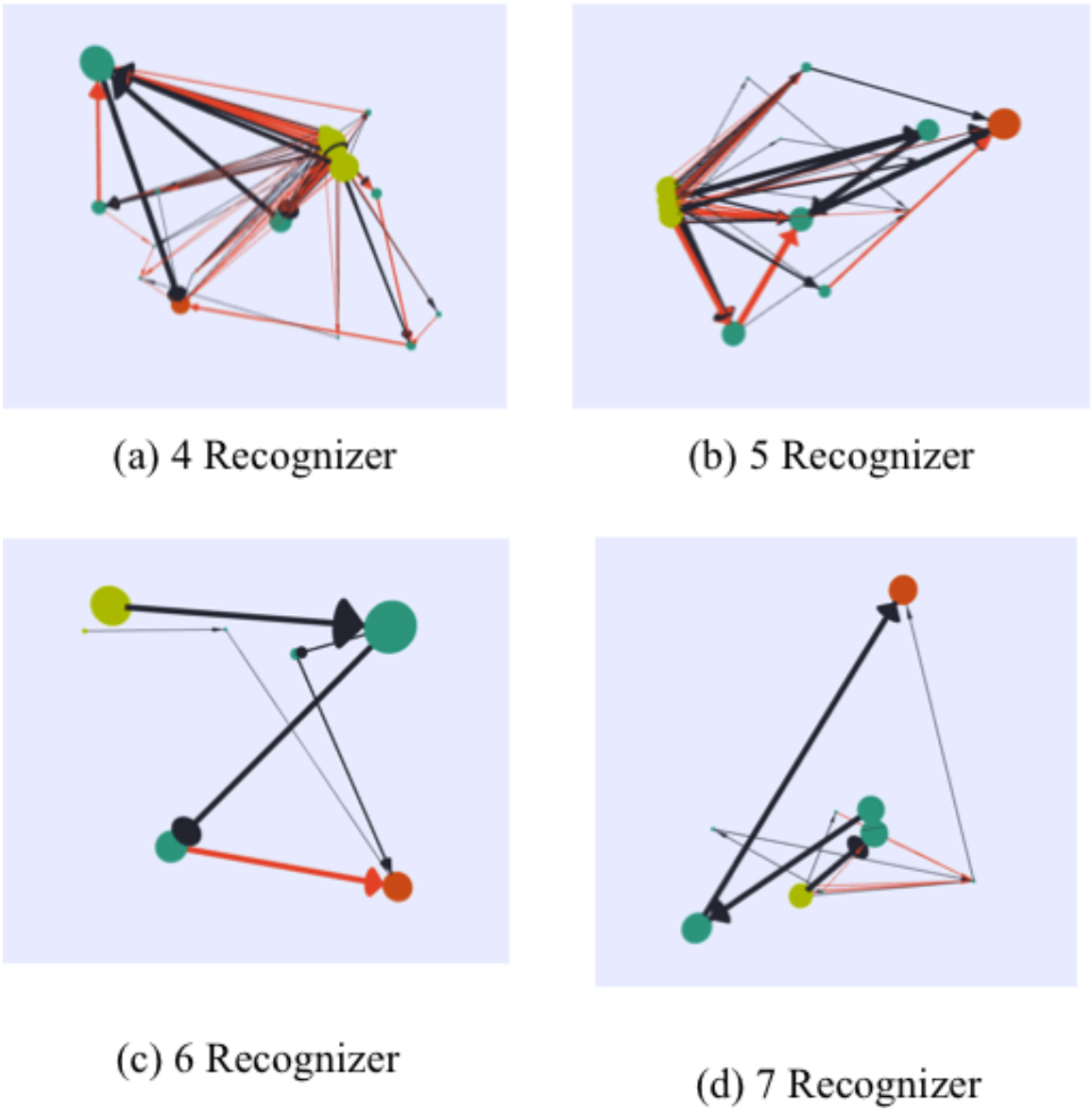
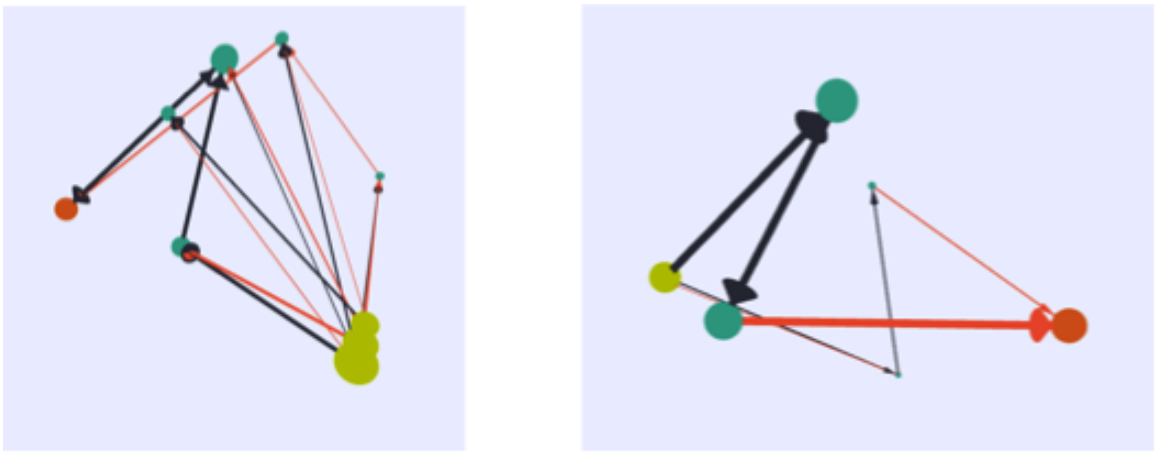


Figure 4.27: Subnetworks of networks trained to recognize various digits (4-7) in handwritten digit images. In these subnetworks, the neurons and synapses are sized based on the number of times they appear in a causality path for a firing event in the last 50 time steps of the simulation, which signifies a detection of a particular image. (Figure created in part by Margaret Drouhard and included with her permission.)



(a) 8 Recognizer (b) 9 Recognizer

Figure 4.28: Subnetworks of networks trained to recognize various digits (8,9) in handwritten digit images. In these subnetworks, the neurons and synapses are sized based on the number of times they appear in a causality path for a firing event in the last 50 time steps of the simulation, which signifies a detection of a particular image. (Figure created in part by Margaret Drouhard and included with her permission.)

4.2 Reuse of Useful Structures

One goal of identifying useful substructures in networks is to reuse those structures in future networks. To facilitate the reuse of structures in the network, several features were added. These features are the ability to translate an existing neuron in the network, translate a section of an existing network, translate the entire network, dropping in an existing network into another network, and rotating an existing subnetwork. Using these mechanisms, we can take an existing useful substructure (described in a network file), rotate or translate it in its own three-dimensional space so that it ends up in the location in which we want it to exist in a new network, and then drop that network into the new network. This way, one small structure can be stored in a canonical way (for example, oriented around the origin (0,0,0)), but manipulated as needed to be used many times in a network structure (see Figure 4.29). These operations are useful when initializing networks using a subnetworks from a "toolbox" of useful networks. They are also useful when adding subnetworks as

mutations to an existing network. The location in which we add the new substructure may be dependent upon locations of existing neurons and synapses in the network, such as particular input and output neurons.

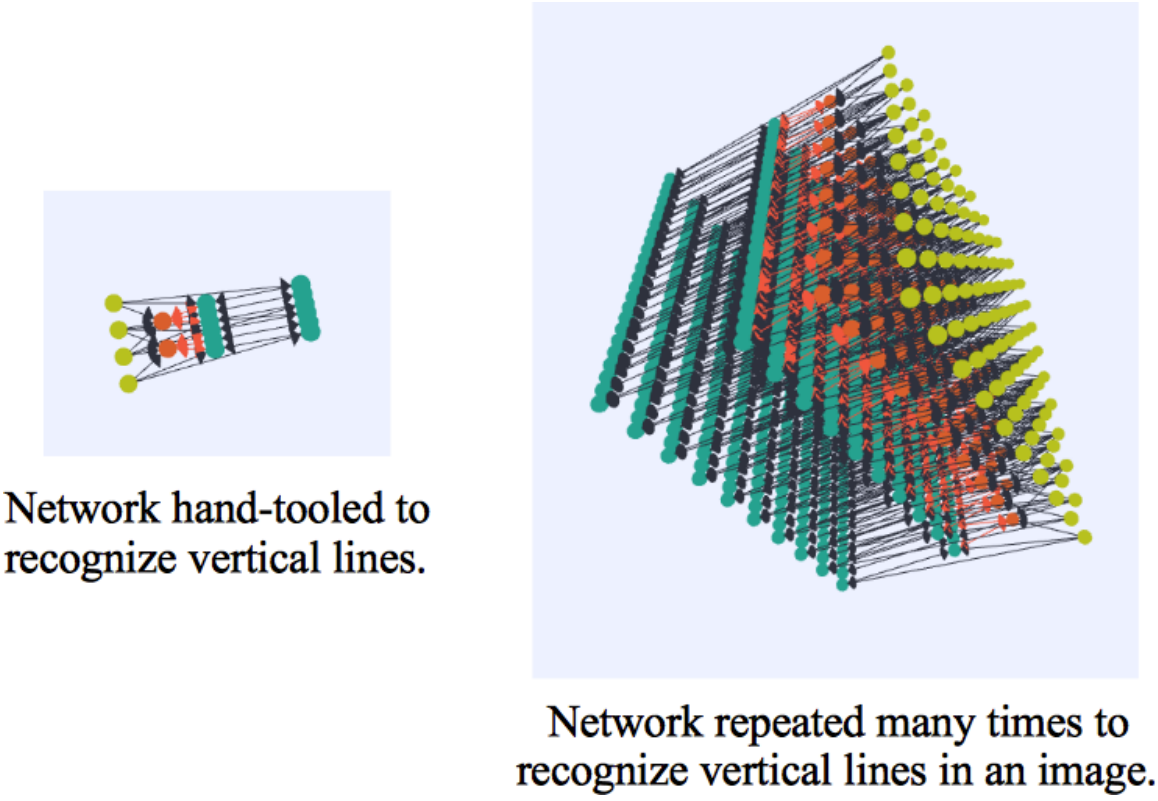


Figure 4.29: An example substructure, hand-designed to recognize a vertical line is given on the left. On the right is a network that is built from many of these substructures so that it will detect a vertical line in any location in a large grid. (Figure created in part by Margaret Drouhard and included with her permission.)

These operations rely on the existence of the toolbox of useful networks when initializing networks or over the course of the operation of the evolutionary optimization. These networks may have been extracted from networks trained for the same or related tasks by the mechanisms specified in Section 4.1, or these networks may have been designed by hand, as in Figure 4.29.

An alternative or companion to using subnetworks that were determined as useful for other tasks is to identify useful substructures over the course of a single

optimization and reuse these structures during the optimization. Unlike networks in the toolbox, these subnetworks are directly related to the task at hand and are created during the optimization. The common substructures approach to finding useful subnetworks does not apply to this case. For a single optimization, the same substructure existing in multiple networks in the population may be an artifact of the reproduction process and not indicative of the utility of that substructure. Moreover, the detection of common substructures may be NP-hard, so it may not be feasible to complete during the evolutionary optimization. Active substructures, however, are easier to detect and isolate; as such, this is the method that was implemented.

An extension to the evolutionary optimization software was added to accommodate the ability to reuse structures based on activity. Several small changes to the evolutionary optimization software were implemented. Each time a new best network is found for the task, the simulations required for the task are completed, and firing statistics for each neuron in that network are recorded (not all of these records are maintained during normal simulations). Based on these results, the most active hidden neurons in the network are recorded (the input and output neurons are not included because they exist in every network in the population). A user-specified percentage (the default value is five percent) of these neurons are extracted from the network to be included in the useful substructure; let us denote the set of these neurons as N. We use a default value of five percent to avoid overly large structures in the library. In particular, we want our population of networks to maintain diversity. Thus, we do not want multiple networks that are very similar, which could result from larger extracted substructures in the library. Any synapses that connect neurons in N are also included in the useful substructure. A minimum-hop path from each neuron in N from an input neuron in the network is also included (where a path includes both neurons and synapses along that path, but we do not include the input neuron). Minimum-hop paths are defined as the paths with the fewest number of synapses (as opposed to the shortest path). These paths go from the input neuron to a neuron in N. Similarly, a minimum-hop path from each neuron in N to an output neuron

in the network is also included in the substructure. Figure 4.30 gives an example of a network and the associated useful substructure that is extracted from the network using this method.

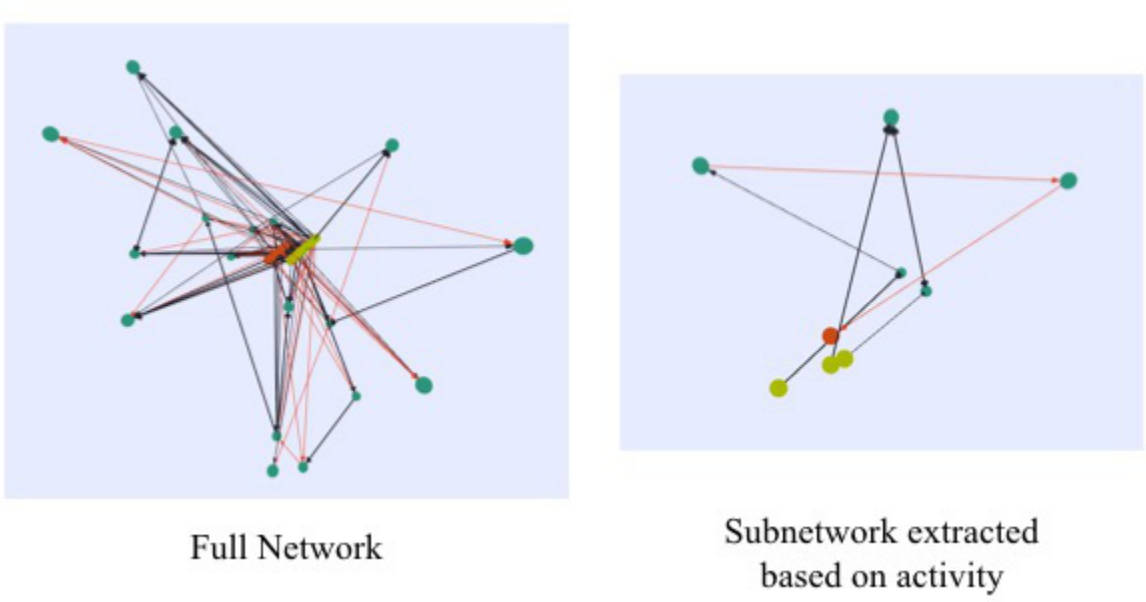


Figure 4.30: An example network is shown, along with the useful substructure extracted from the network based on N for that network is also given. (Figure created in part by Margaret Drouhard and included with her permission.)

A user-specified number of useful substructures is maintained as part of the evolutionary optimization, along with the fitness value of the network from which each substructure was obtained. The mutation operation is expanded so that one possible mutation is the inclusion of one of these substructures in the network. This mutation includes either the subnetwork from the most successful network thus far or randomly selects one of the other useful substructures. It randomly selects between the two, but weights the selection based on user-defined parameters. For example, the user could specify that the subnetwork from the most successful network should be included in 90 percent of the instances in which this mutation occurs, and another subnetwork from the list should be included in 10 percent of the instances.

4.2.1 Results

The extension of the evolutionary optimization to include recognition of active substructures and reuse of those substructures in the network was compared for two different tasks: pole balancing (see Section 3.3.2) and handwritten digit identification (see Section 3.3.3). For the handwritten digit identification, the ensemble method was not used for comparisons; only the results for individual networks were used.

Comparison to Original Evolutionary Optimization

For these tasks, the same initial seeds were used to produce initial populations and to produce the same choices. The only difference between the two instances of the evolutionary optimization occurs when the inclusion of a substructure mutation is selected. In this case, no mutation is made for the original evolutionary optimization. A library is maintained in the original evolutionary optimization instance, but it is not used. For each of the test cases given below training was done for two hours. At the end of two hours, each test case reached a given epoch while training or completed training by reaching the predefined maximum fitness value. The minimum epoch reached between the run using the library and the run not using the library is determined, and the corresponding fitness value of the best performing network for that epoch for both runs is recorded.

Figures 4.31 and 4.32 demonstrate the effect of the inclusion of the library of active substructures on the performance of the networks over the course of training for the pole balancing problem. For this task, 2000 different random seeds were used to generate a test run with the library of active structures and a test run without the library of active structures of the evolutionary algorithm. Figure 4.31 gives an overview of the difference in performance between the evolutionary algorithm using the library of active structure and the evolutionary algorithm that does not make use of that library. We can see that the median and average indicate that, overall, the inclusion of the library had no effect on the performance. We see similar results in

Figure 4.32, which shows that the library hurt the performance of the evolutionary optimization in more cases than it helped, but the number of test cases is comparable for helping and hurting, again indicating that the inclusion of the library is equally likely to help or hurt for this problem type.

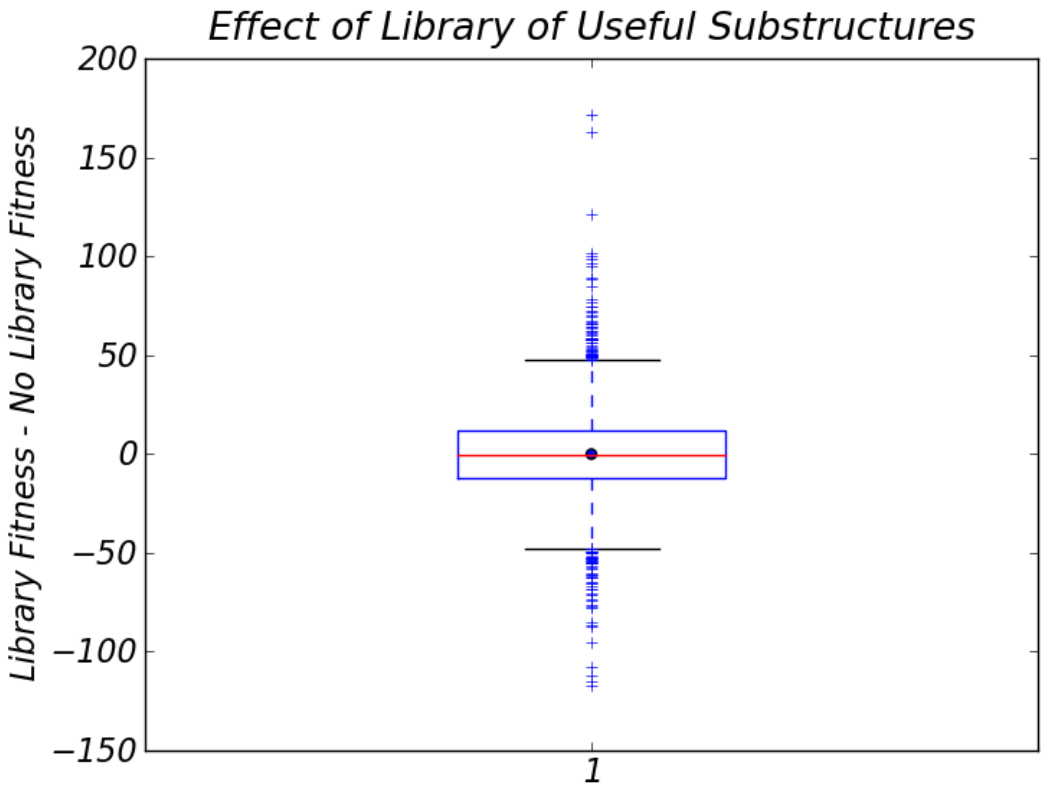


Figure 4.31: A box plot showing the effect of the inclusion of the library on the double pole balancing task. The y-axis gives the difference in performance between the same seeded population when using the library and when not using the library. If the result is positive, it means the library helped, and if the result is negative it means the library was detrimental to the performance of the evolutionary algorithm. The median is shown as the red line for each digit, the box gives the quartiles, and outliers are also shown. The average is shown as a dot.

For the handwritten digit task, 1000 different random seeds were used to generate a test run with the library of active structures and a test run without the library of active structures of the evolutionary algorithm for each digit type. Figure 4.33

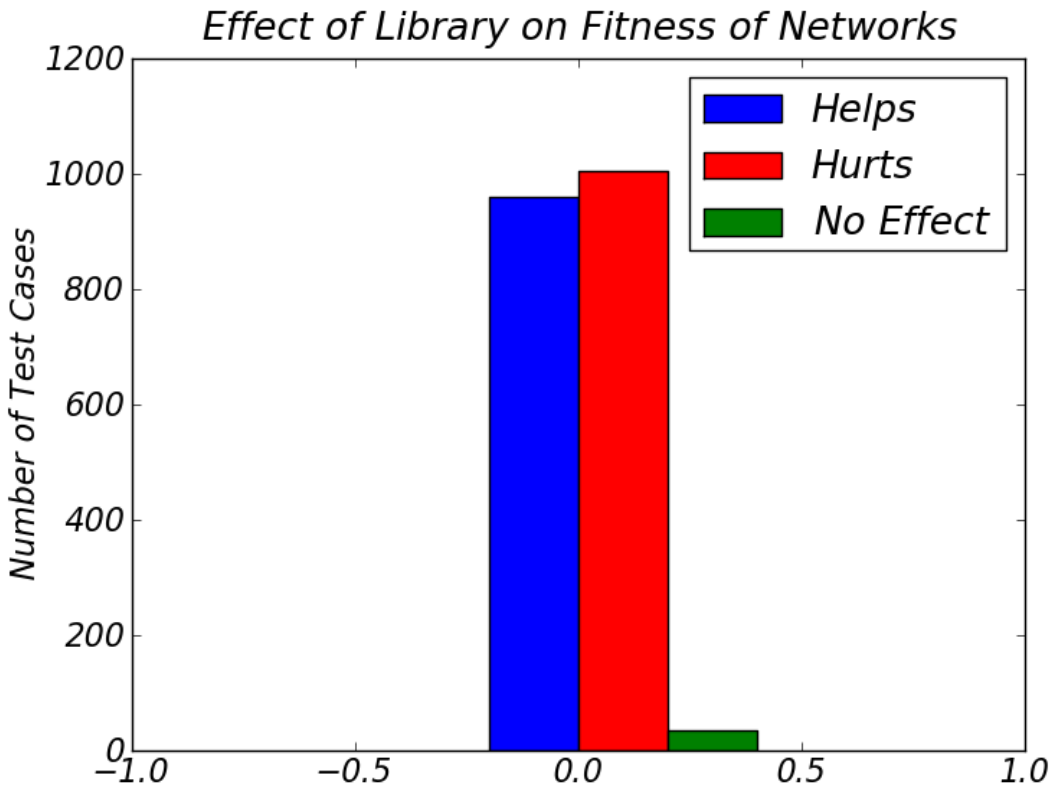


Figure 4.32: A bar graph showing the effect of the inclusion of the library on the double pole balancing task. The three bars shows the number of tests (out of 2000) in which the inclusion of the library helped performance, hurt performance, and had no effect on performance, respectively.

shows a box plot of the differences in performance between these two test runs. This figure shows that, in general, the use of the library did not have a major effect on performance, though there are many outliers. The median of the difference in performance between test runs was fairly consistently at or near zero. Similarly, the average difference in performance was positive in half of the cases and negative in half of the cases, meaning that the library helped for half of the digits and hurt the performance for the other digits.

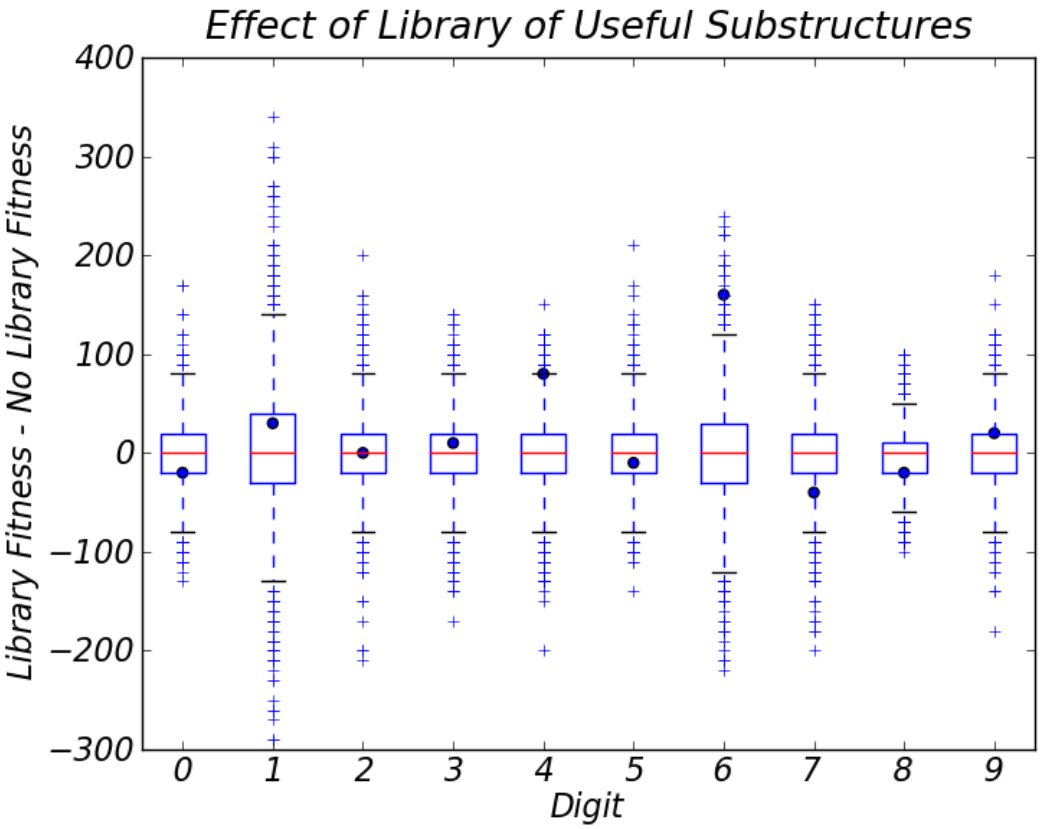


Figure 4.33: A box plot showing the effect of the inclusion of the library on the handwritten digit task, with a box for each digit type (0-9). The y-axis gives the difference in performance between the same seeded population when using the library and when not using the library. If the result is positive, it means the library helped and if the result is negative it mean the library was detrimental to the performance of the evolutionary algorithm. The median is shown as the red line for each digit, the box gives the quartiles, and outliers are also shown. The average is shown as a dot.

Figure 4.34 shows the breakdown of the performance on each of the 1000 tests for each digit; in particular, it shows the number of tests in which the library helped, hurt, or had no effect. In this case, we see that for six of the ten digits, there were more test cases in which the library helped, and for four of the ten digits, there were more test cases where the library hurt. In a fairly large number of test cases, the inclusion of the library had no effect on the performance.

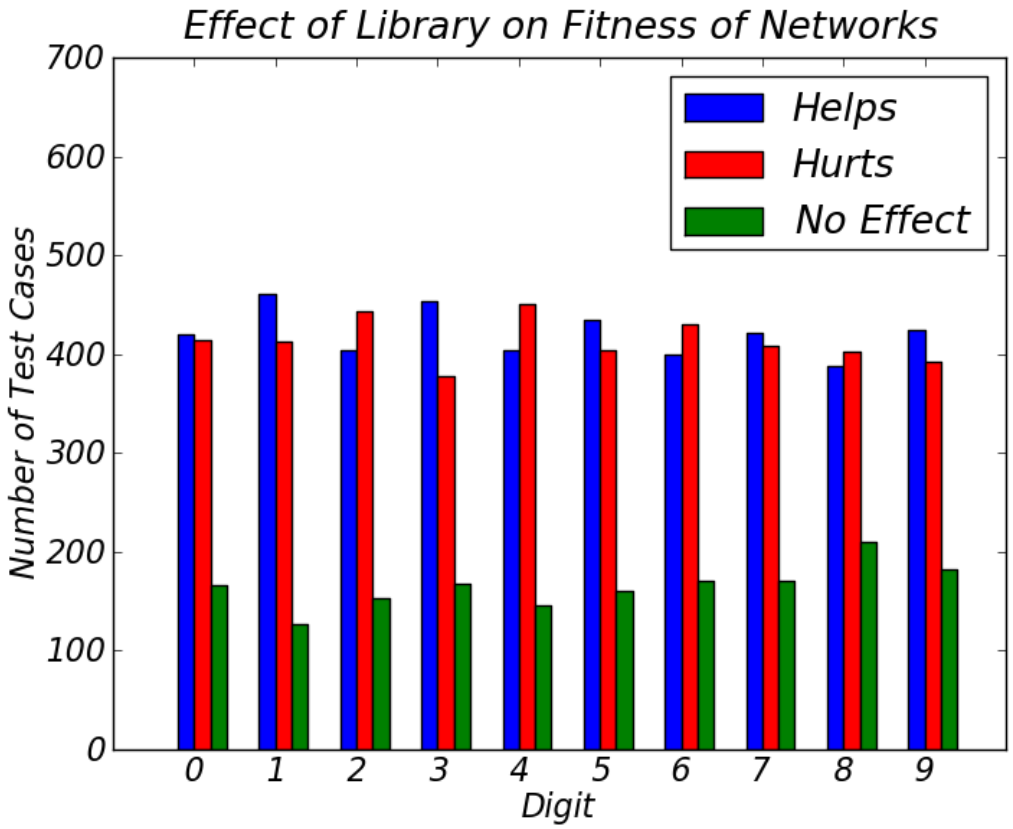


Figure 4.34: A bar graph showing the effect of the inclusion of the library on the handwritten digit task. The three bars shows the number of tests (out of 1000) in which the inclusion of the library helped performance, hurt performance, and had no effect on performance, respectively. Results are given for each digit type.

These results indicate that, for these two problems, selecting useful substructures based on points of high activity does not have a major effect on training performance. It may be that the structures that are highly active in these networks do not have a

major effect on the performance of the network because a large amount of activity is not required to solve either of these problems. In both cases, firing of the output neuron(s) corresponds to a choice, but a single fire at the output neuron can have the same effect as 10 or more fires for both of these problems. In the case of the handwritten digit recognition task, only a single fire is required for recognition, while in the pole balancing problem, a single firing on one of the output neurons can lead to a selection of that force. This type of useful substructure may not be suitable for problems where this is true.

The changes in the software that allowed for extraction of useful substructures and inclusion of those structures in future networks can be easily altered to extract substructures based on other factors. For example, in the future, we may want to extract causality paths or some other structure based on activity.

4.3 Summary

This chapter explores the recognition and reuse of useful substructures in NIDA networks. Potentially useful substructures were recognized through three mechanisms: common substructure graph analysis, firing activity analysis, and causality path analysis. There are many ways to examine activity within the network that may be more illustrative than firing analysis, including detection of causality paths. It may also be useful to examine the activity in the network as a directed graph, where nodes in the graph are events (such as firing of neurons and change in charge events on neurons), as shown in Figure 4.35. Comparing these types of graphs may give more information than attempting to compare the structure of networks using graph analysis. It would also be useful to define a set of similarity metrics so that two networks can be compared. For example, we may want to compute a similarity metric based simply on graph structure, or we may want to compute a similarity metric based on activity in two networks based on a particular set of inputs. It is not immediately clear how to define these metrics for NIDA networks, though it is clear

that they would be useful in the identification of useful substructures within NIDA networks.

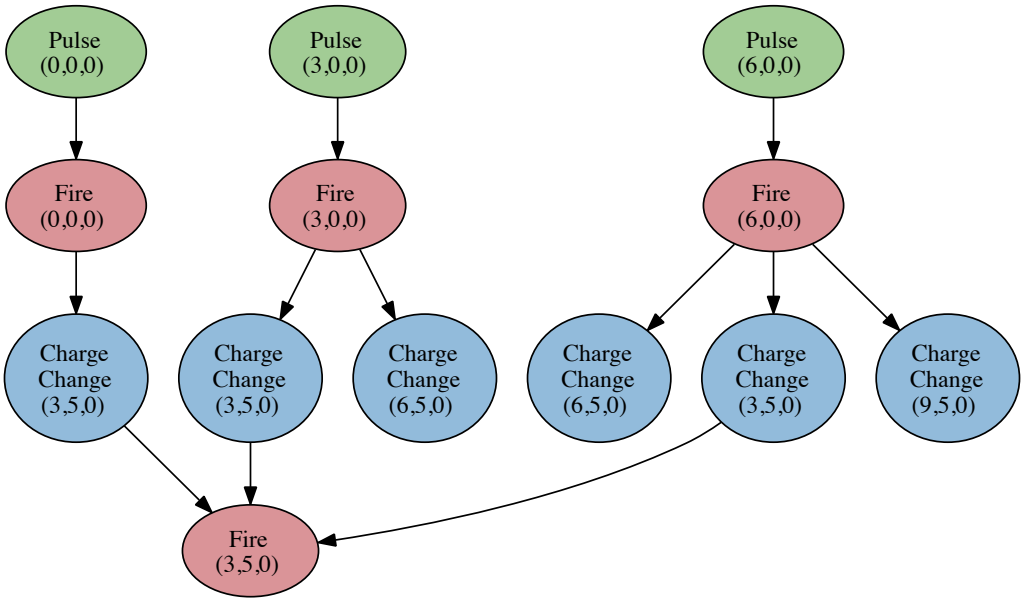


Figure 4.35: Graph showing activity in a single network, in which the nodes are events and the edges indicate causality.

We explored two ways of reusing useful substructure. We hand-designed a small network to solve a subtask and developed tools to allow for ease of replication of that structure across the network. We could develop a "toolbox" of networks that solve simple problems that may be useful for various tasks (e.g., networks that compute bit-wise operations like AND, OR, and XOR) by hand-designing them or designing them using the evolutionary optimization method. Then, users could build larger, more complex networks by using these networks as building blocks.

The second method for reusing useful substructures was automatic detection and extraction of high-activity subnetworks within networks over the course of evolutionary optimization and the inclusion of these networks as mutations. We applied this method to two tasks: a pole balancing task and the handwritten digit

recognition task. With this simple method of substructure detection, we saw, on average, very little change in the performance of the evolutionary optimization, indicating that this method may not be the best method for finding useful substructures. We speculate that the detection, extraction, and inclusion of causality paths is more likely to produce better results. However, as noted in the previous sections, it may be too computationally intensive to detect causality paths during the evolutionary optimization. A better method may be developing a parallel optimization algorithm in which causality paths of the best performing networks are determined in a separate thread or process from the optimization algorithm. We discuss several other future directions for the recognition and reuse of useful substructures in Section 7.2.

5

Affective Systems and Multiple Interacting Networks

"I cannot fix on the hour, or the spot, or the look or the words, which laid the foundation. It is too long ago. I was in the middle before I knew that I had begun."

– Jane Austen, *Pride and Prejudice*

In this chapter we describe artificial affective systems that may be included as part of our NIDA networks. A very simple affective system based on a set of equations and a more complex affective system based on NIDA subnetworks are described. It is noted that these affective systems are best utilized when they are trained alongside the NIDA network. As such, several adaptations to the design method are made to accommodate for both the simple and complex affective systems. We also present results on the effect of the inclusion of artificial affective systems on the design method and the resulting networks. Finally, it is noted that the adaptations to the design method are also suitable for training a team of multiple interacting NIDA networks,

and that, in fact, affective systems within NIDA networks are a special case of multiple interacting networks. Many of the results in Section 5.1 come directly from [227].

5.1 Affective System Description

As noted in Section 2.5, in comparing artificial neural networks (ANNs) to their biological counterparts, we note that ANNs typically perform work that is accomplished in the cerebral cortex. ANNs, unlike the cerebral cortex, are trained to perform a highly specific, usually well-defined task. However, the role of an ANN can be thought of as analogous to a very small network in the cerebral cortex. An ANN, like the cerebral cortex, receives and transmits information to its environment, often through other mechanisms that play the roles of sensory systems and motor systems. The interaction of the cerebral cortex with the basal ganglia and the limbic system, however, is typically not included in a neural network architecture. Among their many functions, parts of the limbic system are responsible for the affective system. The affective system deals with emotions. Emotions are known to have an effect on the way we learn and reason [63]. Panksepp, in his work on affective neuroscience, defines four major emotional operating systems with their roots in the limbic system: seeking (curiosity), fear, panic, and rage [188]. Different parts of the limbic system are associated with different emotions. These affective systems communicate with the cerebral cortex through neurotransmitter pathways (Figure 5.1). All of these systems have an effect on learning; for example, a person with damage to the pathways that produce seeking behavior will be less curious, and thus, less likely to seek out new information about their environment. The neurotransmitters associated with each system also differ, and thus have a different effect on the operation of the cerebral cortex.

In our effort to move ANNs more towards biological neural networks, we propose the inclusion of an artificial affective system in the ANN, so the ANN has two coupled

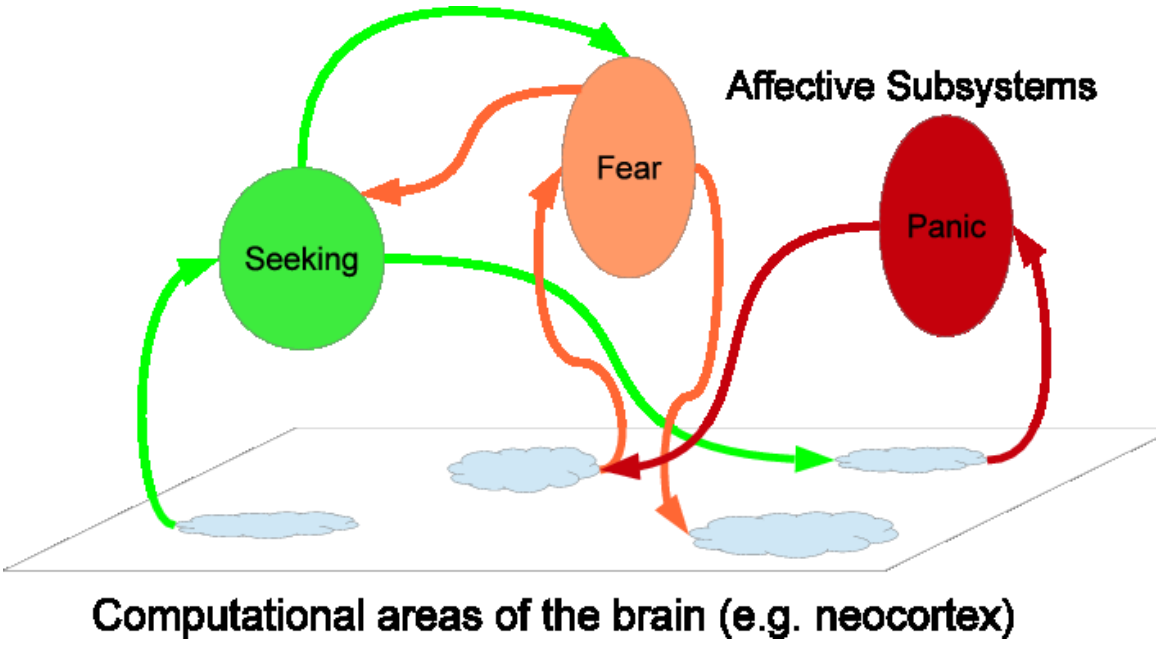


Figure 5.1: Simplistic view of affective systems interacting with each other, the neocortex, and other parts of the brain using neurotransmitters.

parts: an affective system analogous to part of the limbic system and a computational system analogous to part of the cerebral cortex.

In the race for human-level machine intelligence, there is some argument as to whether affective systems are necessary. Many works in the field are simply attempting to emulate the neocortex, or higher level functionality. However, there are some works that are attempting to create emotion or affective behaviors in the machine learning community, discussed in Section 2.5. We propose that, rather than using a model of a particular emotion, the emotional or affective systems should learn or evolve along with the computational system. There should not be a pre-defined behavior associated with an emotion.

We define an emotional or affective system with respect to our NIDA networks. An artificial affective system (AAS) has three main characteristics:

1. The affective system has an inhibitory or excitatory effect on the computational system. For example, in NIDA networks, the inhibitory or excitatory effect is

on a neuron or neurons in the network, making the neurons more or less likely to fire.

2. The affective system's effects on the computational system may be short-term or long-term.

3. The affective system evolves or learns alongside the computational network and possibly one or more other affective systems. That is, it has parameters that can be altered as part of training or learning.

Neurotransmitters are the inspiration for the first and second components of our AASs; just as biological affective systems release neurotransmitters within different regions of the brain, our AASs have similar effects on the NIDA networks. Designing or training the AAS alongside the NIDA network requires some flexibility in the definition of the AAS; it should have associated parameters that can be modified by an optimization or adaptation strategy.

5.1.1 Simple Affective System

In this section, we define two simple affective systems. The goal of these simple affective systems is to regulate the firing rate of neurons in a NIDA network. Our simple affective system is determined by the following equations. Parameter f_t is the firing rate of the network, measured over a certain window, at time t. This is the input provided to the affective system from the network. Parameter d_t is the desired firing rate at time t. This desired firing rate is provided by the environment and can be changed by a desired firing rate event; in preliminary tests, these are pre-defined values, but the desired firing rate may also be altered over the course of optimization for a particular task. The error at time t, e_t, is calculated as:

$$e_t = f_t - d_t \tag{5.1}$$

This equation is such that the error will be positive if the current firing rate f_t is higher than the desired firing rate d_t. In this case, we want to lower the firing rate by increasing the thresholds. Similarly, e_t is negative when we want to increase the firing rate by decreasing threshold values. Thus, positive e_t implies a positive change in threshold, and negative e_t implies a negative change in threshold. We have two simple affective systems: an affective system with two parameters and an affective system with three parameters. Both affective systems have the parameter $w > 0$, which is the window size of the system, and w specifies how often the error is recalculated. In the first simple affective system, the change in the threshold at time t is calculated:

$$\Delta\tau_t = \alpha e_t \tag{5.2}$$

The parameter α is a weighting term and is always non-negative, and the change in the threshold at each time step is proportional to the firing rate error. $\Delta\tau_t$ is the amount that every threshold in the network is changed at time t. This result is passed back to the network, and the change is applied to *all* of the neurons in the network; since all of the neurons have the same initial threshold value of 0.5, all neurons in the network maintain the same threshold value throughout the simulation (unless otherwise noted). The threshold is restricted to the interval $[-1, 1]$ and equation (5.2) has no effect if it would cause either bound to be violated.

In the second simple affective system, a second parameter, λ, a tuning parameter, is added. A geometrically averaged error at time t, E_t is calculated:

$$E_t = \lambda E_{t-w} + (1 - \lambda)e_t \tag{5.3}$$

The parameter λ is a decay rate, and λ is bounded such that $0 \leq \lambda \leq 1$. It defines how errors at times 0 through $t - 1$ will affect the change in the threshold at time t.

With this second affective system, the change in the threshold at time t is calculated:

$$\Delta\tau_t = \alpha E_t \tag{5.4}$$

where, again, α is a weighting term. In both cases, the result $\Delta\tau$ is passed back to the network, and the change is applied to all of the neurons in the network. Note that the first and second systems are equivalent if $\lambda = 0$. The same boundary logic applies as with equation (5.2).

5.1.2 Results for Simple Affective System

We want to demonstrate that the defined simple affective systems interacting with an artificial neural network can have a noticeable effect and can stabilize the average firing rate at desired levels. All networks discussed in this section have 1000 neurons and 10000 synapses, where $M_x = M_y = M_z = 100$ and $m_x = m_y = m_z = -100$. This is a relatively large artificial neural network, but compared to the human brain, this is a very small network. It is important to note, however, that we are not attempting to model a biological neural system with our artificial neural networks; our artificial neural networks are merely motivated by biology. The tasks these artificial networks are applied to are specific and well-defined. As such, they can be thought of as analogs to the small portions of the neocortex that implement specific functionalities. Networks with different numbers of neurons and synapses yield similar results, though they are not shown in this work.

The initial neuron placements in the network are random, and the distribution of the synapses is random, but with a higher likelihood of connectivity between spatially close neurons than neurons that are farther apart. In this network structure, there are 200 possible x-coordinate values, 200 possible y coordinate values and 200 possible z coordinate values, resulting in 8×10^6 possible locations for neurons in the network. A specific instance or realization of a network has neurons at 1000 of these locations,

randomly selected according to a uniform distribution, except no two neurons are allowed to occupy the same location.

Each network has a single input neuron that receives information from the environment. The "environment" in our setup consists of two things: pulses sent to the input neuron at exponentially-distributed random intervals, with a mean firing rate of 0.1 firings per unit time, and an input to the affective system that sets the current desired firing rate. This input plays the role of a persistent external excitation used to initiate and promote firing events in the network. This is an extremely simple environment; more complex tasks have richer environments that provide meaningful information to the network and receive signals produced by the network, as seen in other sections in this work. The affective system monitors the behavior of the network and applies the threshold changes to the network every w (the window size) units of simulation time. For all of the tests in this work, $w = 10$.

All neurons in the network have a refractory period of one, which means that there is an upper limit on the firing rate of the network; since each neuron can fire at most once in a single simulated time step, the maximum firing rate of the network per time step is 1000. This assumes that the network is fully connected, which is not a requirement placed on the random initialization of the networks. There may be neurons that have no incoming synapses or neurons with no outgoing synapses, which would further limit the maximum firing rate of the network, and the network is not necessarily connected.

In our preliminary experiments, we set the parameters of the affective system to be $\alpha = 0.001$ and $w = 10$. The long term potentiation/long term depression refractory periods are set to be 10, and the weights are adjusted up (for LTP) and down (for LTD) by 0.001.

Single Desired Firing Rate

The first goal is to demonstrate that, even with these two very simple controlling affective systems, the network's firing rate can be adjusted and stabilized. The

environment provides the single firing rate (a value between 50 and 950) to the affective system at the beginning of the simulation. The simulation is allowed to run for 10000 simulated units of time, where the input pulses are randomly applied as described above, and the affective system updates the threshold of the network every ten simulated time units ($w = 10$).

We ran the same tests for 100 random network structures (each with 1000 neurons and 10000 synapses). Similar results were seen in each case. Figure 5.2 shows results for both of the simple systems, $\lambda = 0$ and $\lambda = 0.9$, where $\alpha = 0.001$, for one representative random network structure with 1000 neurons and 10000 synapses. The results are shown for a range of desired firing rates (150 to 900 in increments of 150). A control trace is also shown, demonstrating how the network behaves with the same set of input pulses, but without an associated affective system.

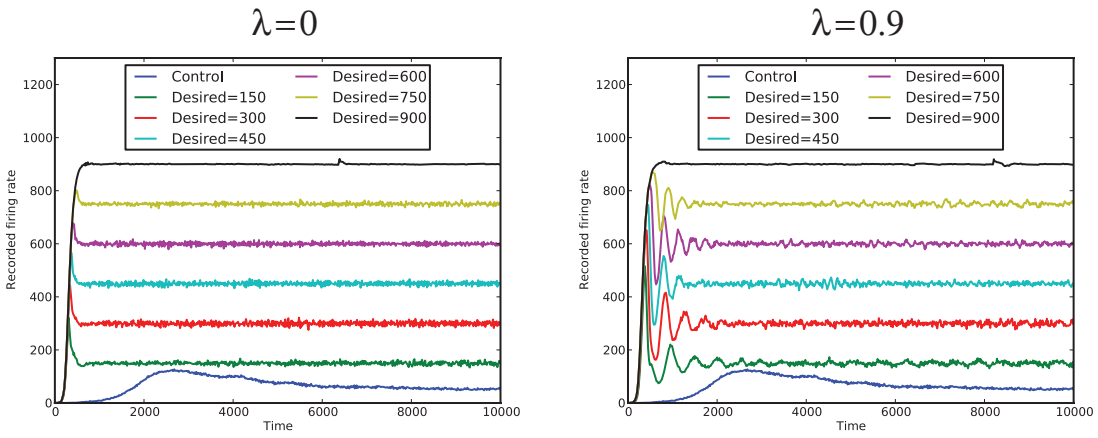


Figure 5.2: Simulation results for a single desired firing rate. Results for both $\lambda = 0$ and $\lambda = 0.9$ are shown.

Figure 5.3 shows a box and whisker plot of the final threshold values after simulation has occurred, for the range of desired firing rates. The box and whisker plot shows these threshold values over the 100 random network structures. The resulting values of the thresholds are almost identical for both $\lambda = 0$ and $\lambda = 0.9$, so only the results for $\lambda = 0.9$ are shown. As expected, the resulting value of the threshold decreases as the desired firing rate increases. This is because neurons with

low threshold values are more likely to fire than neurons with high threshold values. It is also important to note that when the neurons fire, their charge is set to neutral (0). This does not automatically force neurons with negative thresholds to fire. The neuron must receive some charge before it may fire again, and it must be outside of its refractory period to fire again.

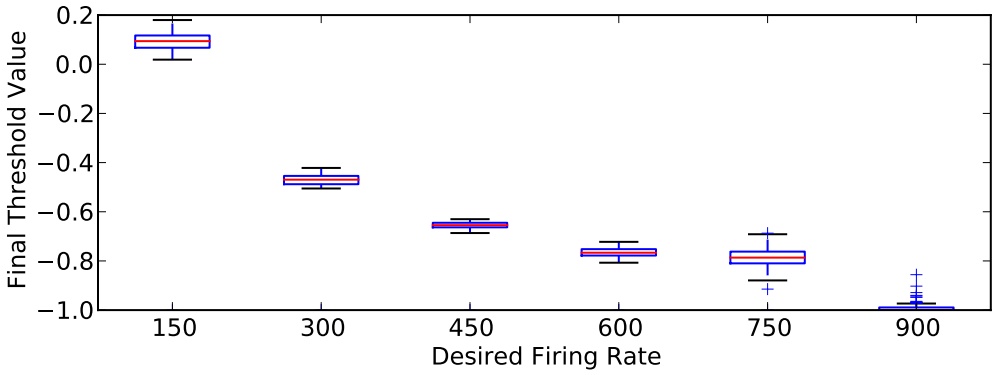


Figure 5.3: Box and whisker plots for the final threshold value. Plots shown for 100 networks at simulation completion for various desired firing rates. Results shown are for $\lambda = 0.9$; results for $\lambda = 0$ are nearly identical.

As is evident from Figure 5.2, the simple affective system can successfully bring the firing rate close to the desired firing rate and maintain firing activity near the desired firing rate. However, there is oscillation around the desired firing rate in most cases, more so than is seen in the control case. The system with $\lambda = 0.9$ has more oscillation around the desired firing rate before it settles to the desired firing rate, and in fact, as λ increases from 0 to 1, the amount of oscillation increases, which is why the extreme value of $\lambda = 0.9$ is shown in contrast to $\lambda = 0$. In many cases this is undesirable behavior. However, as will be seen in later sections, for different values of α, the affective system with $\lambda \neq 0$ may be required in order for the system to stabilize.

Multiple Desired Firing Rates

The second experiment seeks to demonstrate that not only can the controlling affective systems adjust the network firing rate to a particular level, but they can *readjust* the network firing rate level as needed over the course of simulation.

This test is important because the affective system is not the only regulatory system at work in our network architecture. LTP and LTD are also affecting the behavior of the network. Specifically, they are adjusting the weights in the network, and this adjustment is based on the firing rate of the network. There is no guarantee that the new weight distribution will allow for the affective system to adjust the firing level. This test is to determine whether the affective system can continue to adjust the firing level despite the changes made by LTP and LTD.

In this experiment, the network continues to receive input pulses at exponentially distributed random intervals with a mean of 10, but the desired firing rate of the network is changed partway through the simulation, via a change in desired firing rate event sent by the environment and received by the affective system. In this section, results are shown for one network structure. Similar results occur for different network structures.

Figure 5.4 shows the simulation results for $\lambda = 0$ for cases where the desired firing rate of the system increases partway through the simulation; two different sizes of increases are shown (increases of 150 and 600). For both types of increases the affective system is able to adjust the thresholds to reach the first firing rate, and then readjust the thresholds to achieve the new firing rate, within a relatively short time period. Figure 5.5 show the simulation results for the first simple system ($\lambda = 0$) for cases where the desired firing rate of the system decreases partway through the simulation; two different sizes of decreases are shown (decreases of 150 and 600). Again, for both types of decreases the affective system is able to adjust the thresholds so that the network can achieve the original desired firing rate, and then readjust the thresholds to achieve the new firing rate. For all changes, there is a period of

overcorrection of the affective subsystem; that is, for increases, the affective system adjusts the thresholds such that the firing rate usually ends up going higher than the desired firing rate (similarly for decreases). However, for smaller changes in the desired firing rate, there is less of an overcorrection by the affective system than there is for the higher firing rate. This is easily attributable to the design of the affective system; a larger change corresponds to a larger error, and thus a larger correction to the threshold value.

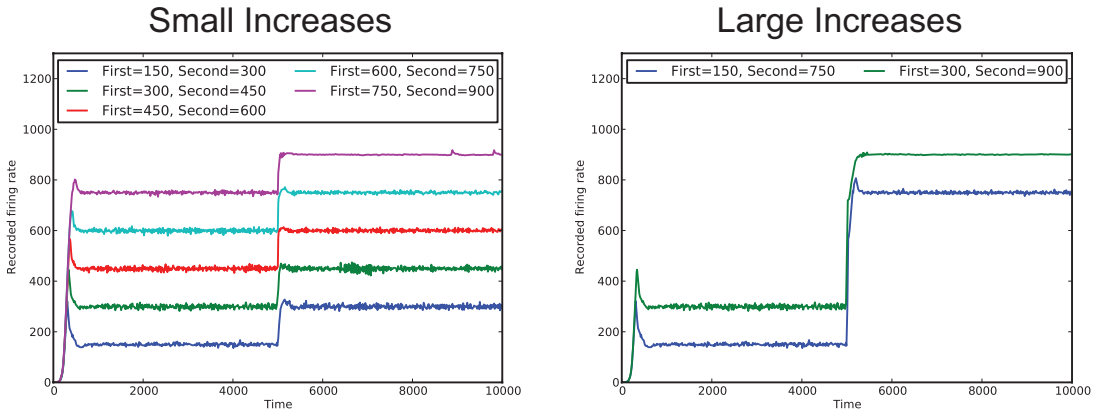


Figure 5.4: Simulation results for an increase in desired firing rates. Results for both small increases of 150 and large increases of 600 in the firing rate are shown.

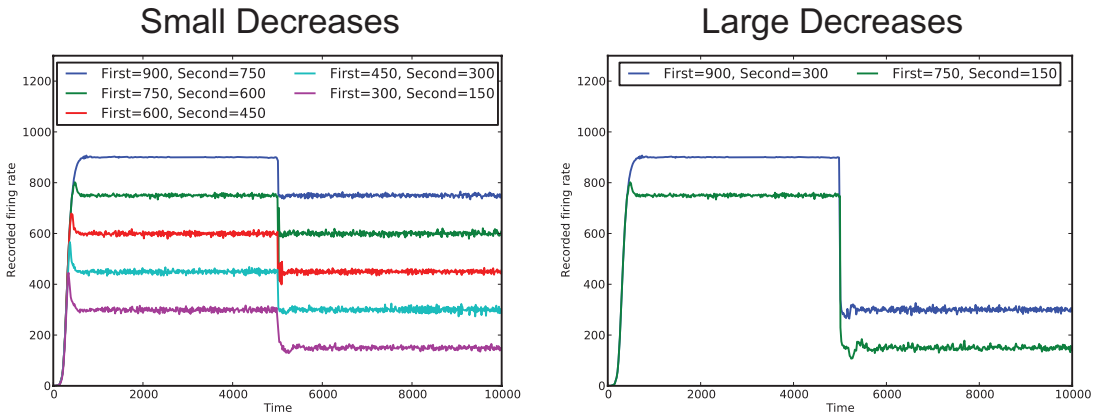


Figure 5.5: Simulation results for a decrease in desired firing rates. Results for both small decreases of 150 and large decreases of 600 in the firing rate are shown.

For the second simple affective system ($\lambda = 0.9$), the results are similar, except, as in the single desired firing rate case, when adapting to a new desired firing rate, there is initially more oscillation around the desired firing rate. We have also seen, for these systems, that large decreases can cause unstable behavior (that is, the affective system cannot tune the thresholds to achieve the new desired firing rate).

Figure 5.6 shows a simulation over 50000 simulated time units in which the desired firing rate was adjusted several times for $\lambda = 0$. It demonstrates that the network can adjust to multiple firing rates over the course of simulation. However, there are some issues with instability when adjusting to later firing rates. These issues are related to LTP and LTD and are addressed in a later section.

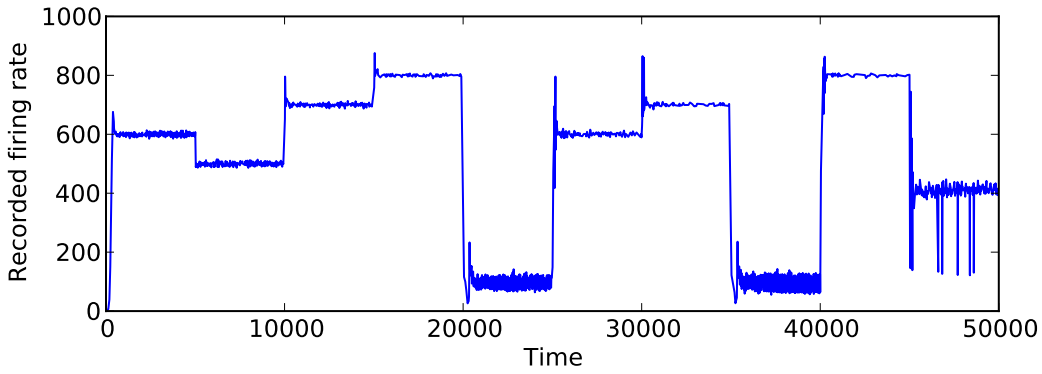


Figure 5.6: Simulation results for many changes in the firing rate. Results shown are for a network with $\lambda = 0$.

Affective System Parameters

The second simple affective system has three parameters: α (a step-size parameter), λ (a decay rate, which may be 0), and w (the window size). For all experiments we have conducted, we have used $w = 10$. We chose this value because it averages out variations in firing rate in a small time period, but it also allows the affective system to update the thresholds frequently. Different window sizes may be considered in future work.

We define settling time as the time required for the network to achieve a desired firing rate and remain within 50 units of that desired firing rate. In the previous sections, we have shown how the results vary for two extreme values of λ (0 and 0.9). In particular, we have found that for larger values of λ, it takes longer for the network to settle at the desired firing rate, and there is more oscillation around the desired firing rate before the network settles. This behavior, however, is not consistent across a variety of desired firing rates; for instance, there is less oscillation at higher firing rates than there is for lower firing rates.

Thus far, we have only presented results for a single value of α ($\alpha = 0.001$). We chose this value of α because the absolute maximum firing rate of the network is 1000, which means that the error between the desired firing rate and the current value of the network is at most (in the extreme case) 1000. When $\alpha = 0.001$ and $\lambda = 0$, this bounds the change in thresholds to a magnitude of one. However, we also tried a variety of values of α, from 0.00025 to 0.005 in increments of 0.00025. Figure 5.7 shows how the different values of α affect the percent overshoot of the firing rate (how high above the desired firing rate the recorded firing rate is, displayed as percent of the desired firing rate) and the time to settle (how long it takes the affective system to adjust the thresholds to achieve the desired firing rate).

Instability in this figure corresponds to a time to settle of 10000; in other words, settling is not achieved. This figure shows that, for higher values of α, a high value of λ may be required to achieve stability. However, the percent overshoot increases as λ increases (except in the cases of instability). If λ is positive, then α also affects the period of oscillation of the measured firing rate, prior to settling. Higher values of α correspond to shorter periods of oscillations, as is shown in Figure 5.8. It is important to note that these figures only show the results for one network and one desired firing rate. Although the general conclusions still hold, the specific values of α where instability becomes an issue differs for different networks.

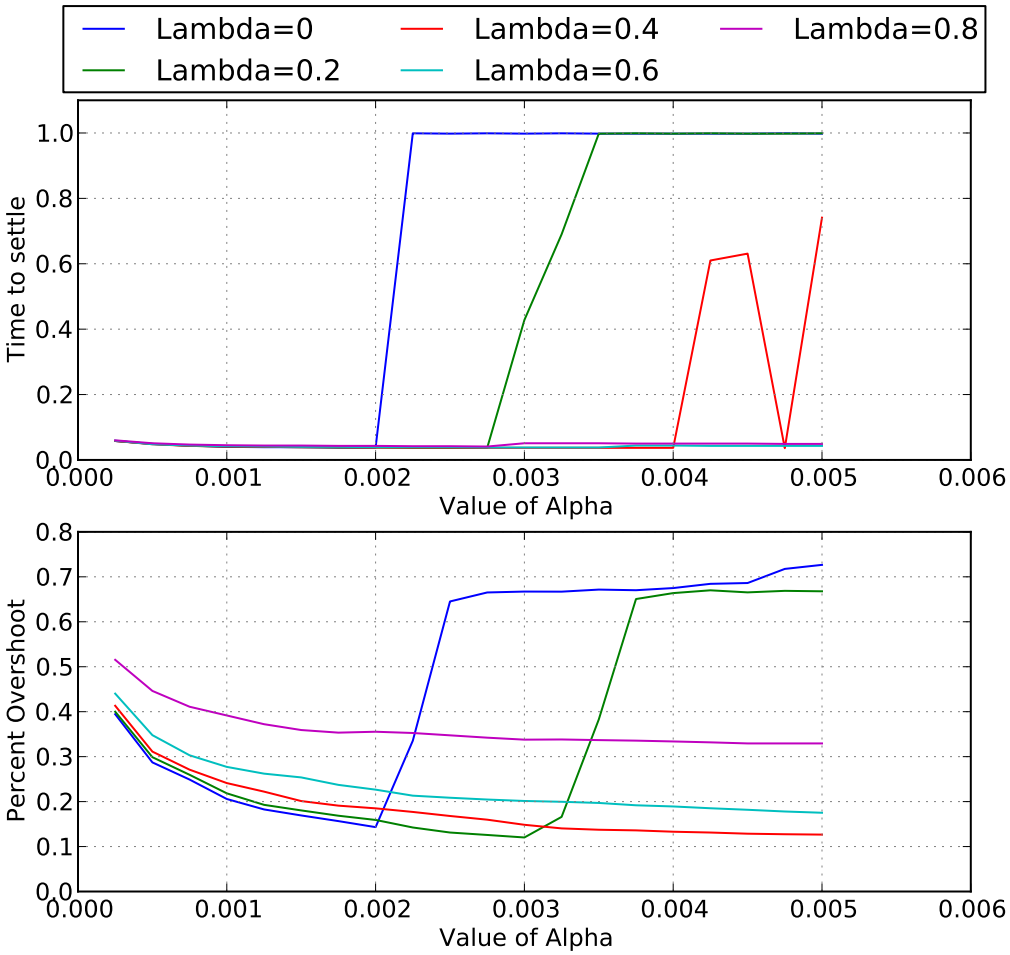


Figure 5.7: The effect of α and λ on time to settle and percent overshoot. Results shown are for one network structure and one desired firing rate.

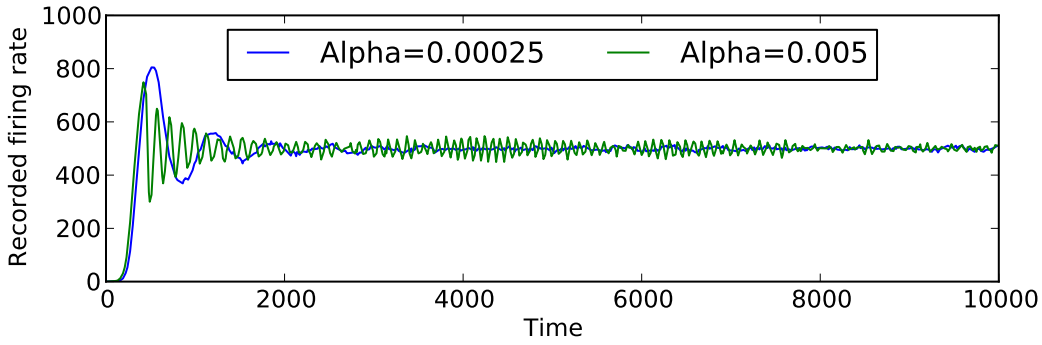


Figure 5.8: Simulation results for two extreme values of α. Results shown are for one network structure and one desired firing rate with $\lambda = 0.9$. The two values of α show are $\alpha = 0.00025$ and $\alpha = 0.005$.

Structure of the Network

Over the course of each simulation, the network topology is fixed; that is, the placement of neurons and synapses in the network is constant. However, the weights of the synapses change during the simulation due to the processes of LTP and LTD. Initially, for each network, the weights of the synapses are uniformly distributed between -1 and 1. In the following figures, we compute a histogram of the network weights across 20 bins. For a weight value of W, each bin is of the form $a \leq W < a + 0.1$, for $a = -1, -0.9, ..., 0.8$, except for the last bin, which is $0.9 \leq W \leq 1$. Figure 5.9 shows the histogram of the final weight values in the network, after the simulation is completed. This figure shows that, in general, most weights fall into either the $-1.0 \leq W < -0.9$ bin, with more weights falling into this bin as the desired firing rate increases. In all other bins up to the $0.9 \leq W \leq 1$, there are relatively few weights, and for smaller values of the desired firing rate, there is an increase in weights in the $0.9 \leq W \leq 1$ bin. This figure, along with Figure 5.3 indicates that, to achieve a higher firing rate, the threshold needs to be decreased, but the weights on the synapses also need to decrease, making most synapses in the network inhibitory synapses.

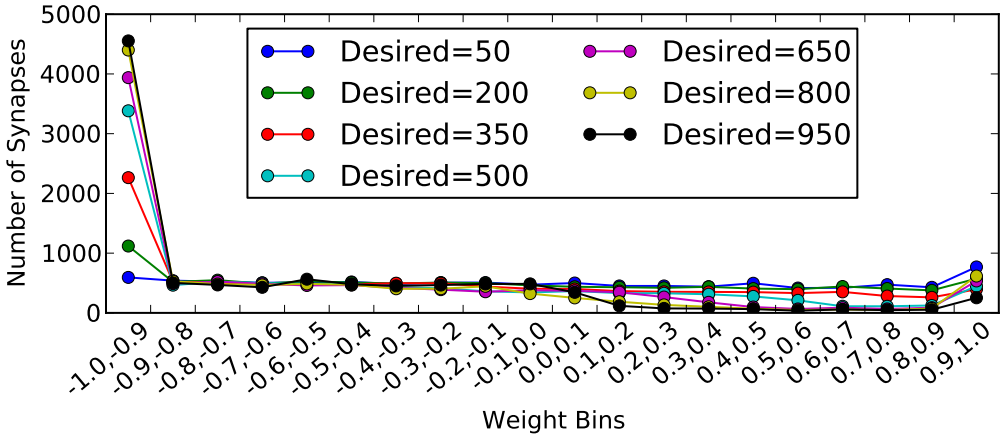


Figure 5.9: Histograms for the final synapse weights in the network at simulation completion. Results shown are for varying desired firing rates and one network structure with $\lambda = 0.9$.

These tests were run using 100 different randomly selected network structures. Figure 5.10 shows box and whisker plots of the histograms of the weights in the final network structures for one desired firing rate (firing rate of 900). This figure shows that the results are consistent across these different network structures.

For the purposes of these simulations, LTP and LTD are not required. The affective system can tune the desired firing rate in the networks without these processes. However, we believe that, since LTP and LTD are expected to play an important role in day-to-day learning in the biological brain, it is prudent to include them in simulations of networks. The exclusion of LTP and LTD can lead to improved stability characteristics. Also, higher LTP and LTD refractory periods can result in lower instability. Figure 5.11 shows the simulation results for a network with no LTP/LTD, with LTP/LTD refractory period at 10 simulated time steps, and with LTP/LTD refractory period of 50 simulated time steps. As noted, the network without LTP/LTD was able to achieve its desired firing rates easily; however, the network with a higher LTP/LTD refractory period had similar results. This indicates that for these networks, learning (that is, any process that affects the structure of

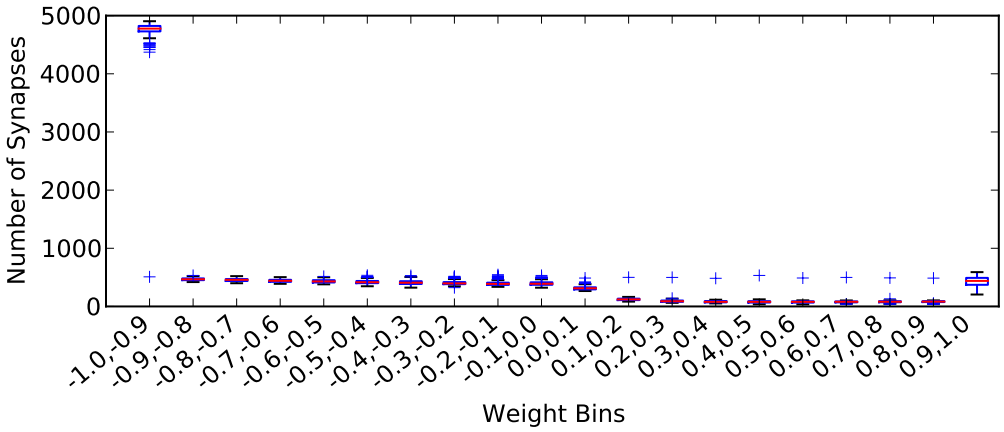


Figure 5.10: Box and whisker plots for the final synapse weights of 100 networks at simulation completion. Results shown are for desired firing rate of 900 and $\lambda = 0.9$.

the network) should occur on a slower time scale than the affective system controller process.

Figure 5.12 shows how the weights change over the course of the simulation for one network structure and one desired firing rate. Similar results are obtained for different firing rates, but the rate at which the weights change is different for each desired firing rate. As expected by the results in Figure 5.9, the end result is that most weights fall into the $-1.0 \leq W < -0.9$ bin. As the simulation continues, more and more weights in the network fall into this bin. However, we also see an increase in the weights that fall into $0.9 \leq W \leq 1$ bin. This indicates that most weights in the network become either nearly fully inhibitory or excitatory. We can also see how different desired firing rates affect the synapse weights. For low firing rates (such as those at simulation times 20000-25000 or 35000-40000), the weight values of the synapses stay mostly constant. Again, this is consistent with what we saw in Figure 5.9. Figure 5.13 shows how the weights changed over the course of the simulation when the LTP/LTD refractory period was set to 50 instead of 10. As expected, the weight values change much more slowly; however, the same basic trends occur.

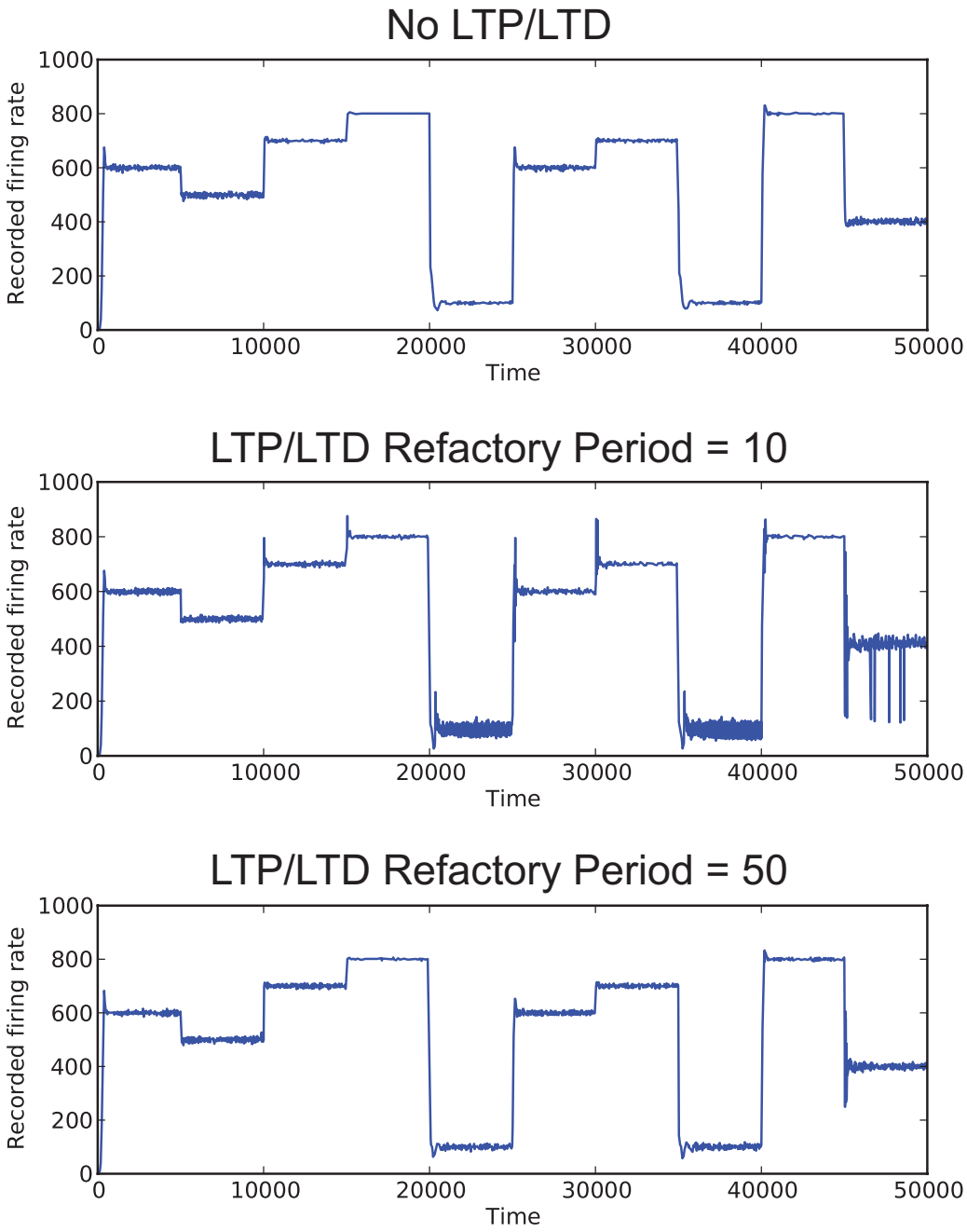


Figure 5.11: Simulation results for different LTP/LTD parameters. Results shown are for $\lambda = 0$.

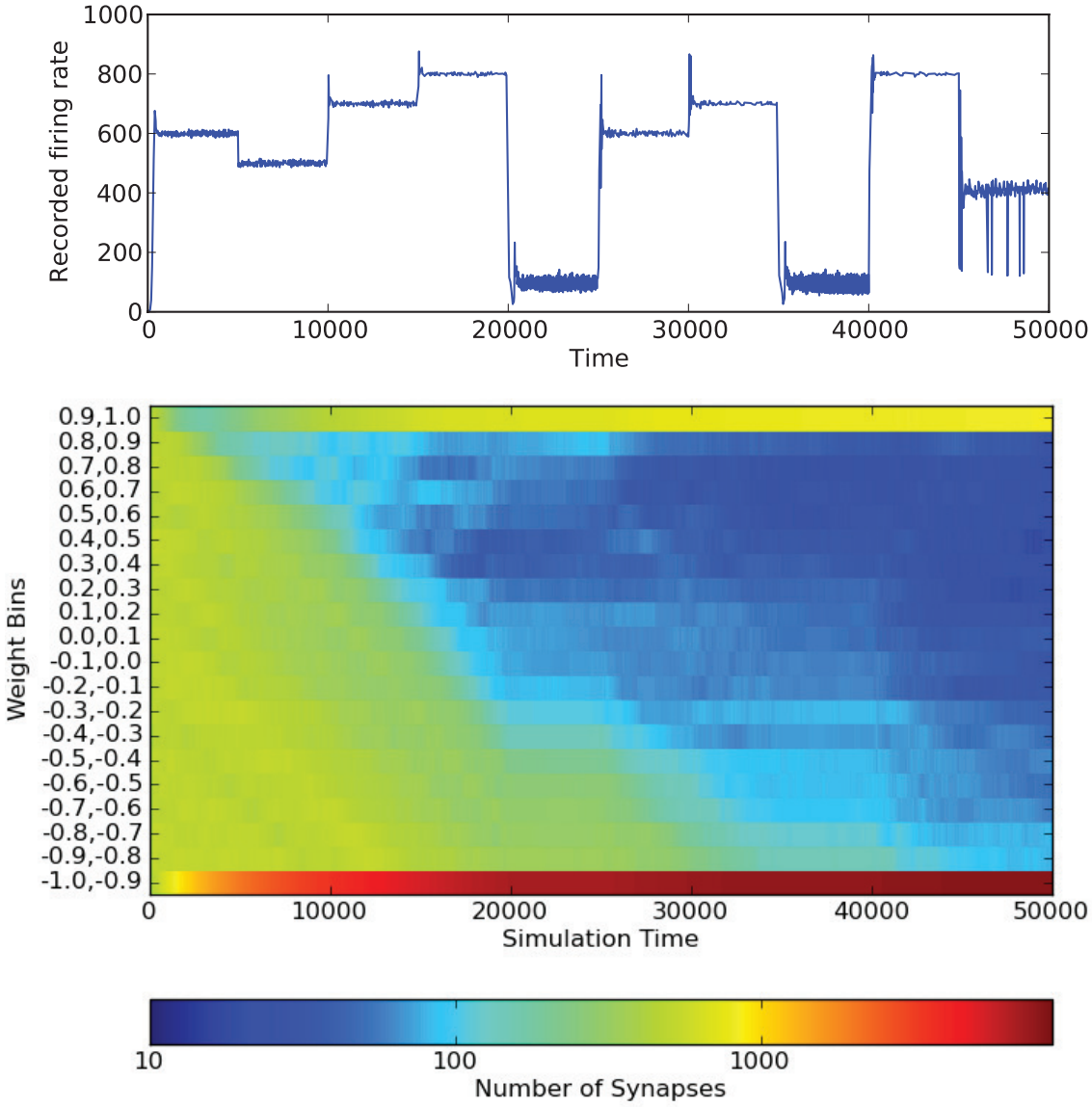


Figure 5.12: Color map showing the changes in synapse weights over a single simulation. Results shown are for $\lambda = 0$. Note that the color bar is on a log scale. The firing rates over the course of the simulation are also shown. The LTP/LTD refractory period used is 10.

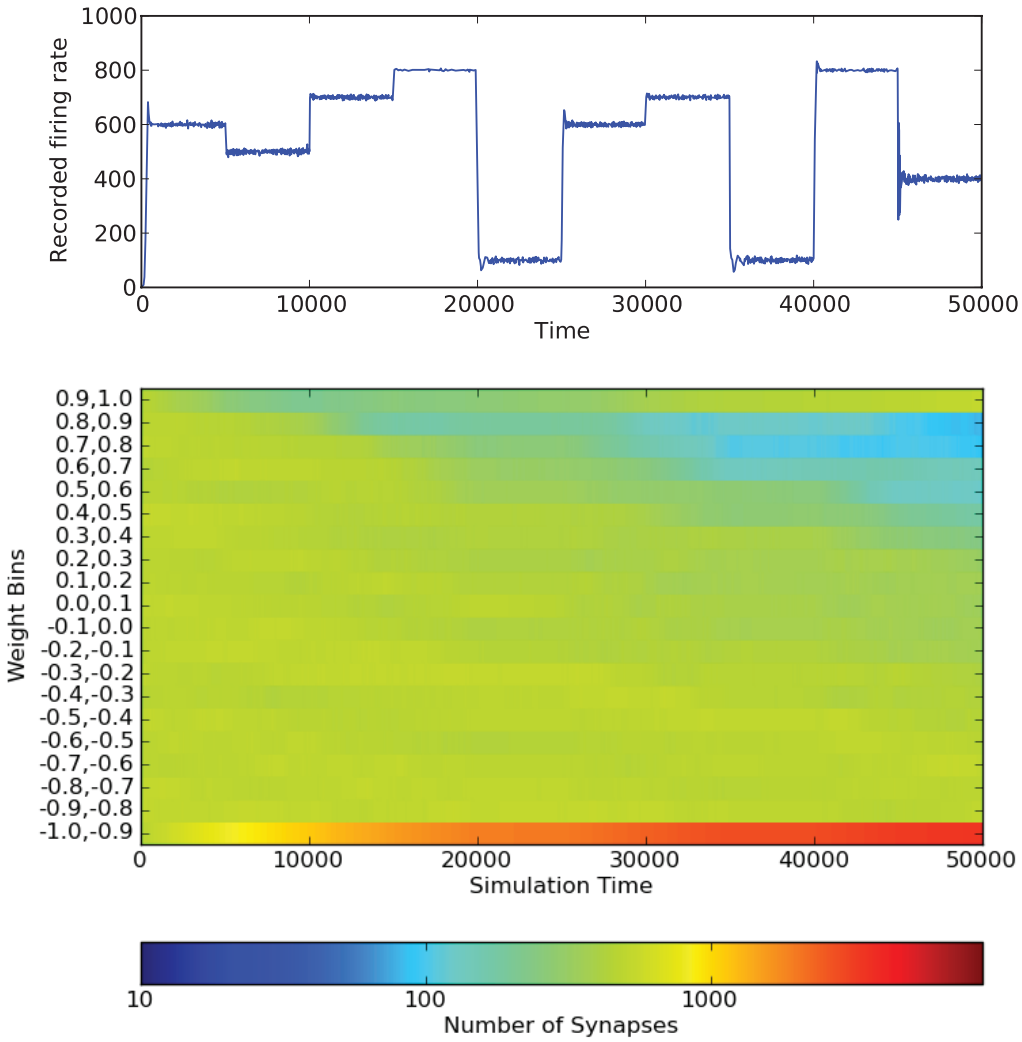


Figure 5.13: Color map showing the changes in synapse weights over a single simulation. Results shown are for $\lambda = 0$. Note that the color bar is on a log scale. The firing rates over the course of the simulation are also shown. The LTP/LTD refractory period used is 50.

Figures 5.12 and 5.13 show how the weights of synapses of changes over time due to LTP and LTD and how the LTP/LTD refractory period affects those changes. Both of these figures show the trend that most of the synapses in the network become either fully excitatory or fully inhibitory. These figures also illustrate that the primary effect of the LTP/LTD refractory period is to slow the learning process. In some cases, we do want the learning process to occur more slowly, as it results in more stable behavior (as shown in Figure 5.11). Slower learning processes also potentially prevent the networks from learning transient details in the environment.

5.1.3 Complex Affective System

In our network structure implementation, we allowed for a neuron to be defined at any point in the grid. We can extend this implementation so that a subnetwork (another NIDA network) may also be defined at any point in the grid. These subnetworks can be defined such that they exist within the space that a single neuron would occupy and not overlap outside neurons. This allows for activity within the network to take place at differing time scales, as activity within the subnetwork will, by nature of how close the neurons are to one another, happen much faster than interactions between neurons in the larger network. We may also alter the propagation velocity of the subnetwork so that the activity in the subnetwork occurs at a similar time scale to activity in the larger network. Moreover, we wanted these subnetworks to have connectivity to the greater network. An example computational network with an embedded subnetwork is shown in Figure 5.14.

This hierarchical representation of the network lends itself to our affective network representations. For ease of optimization, we define an affective network as one that exists as a subnetwork within a larger computational network. This subnetwork may exist such that any neuron within the computational network can be connected to any neuron within the subnetwork and vice versa. However, we may restrict this connectivity for use in our evolutionary optimization.

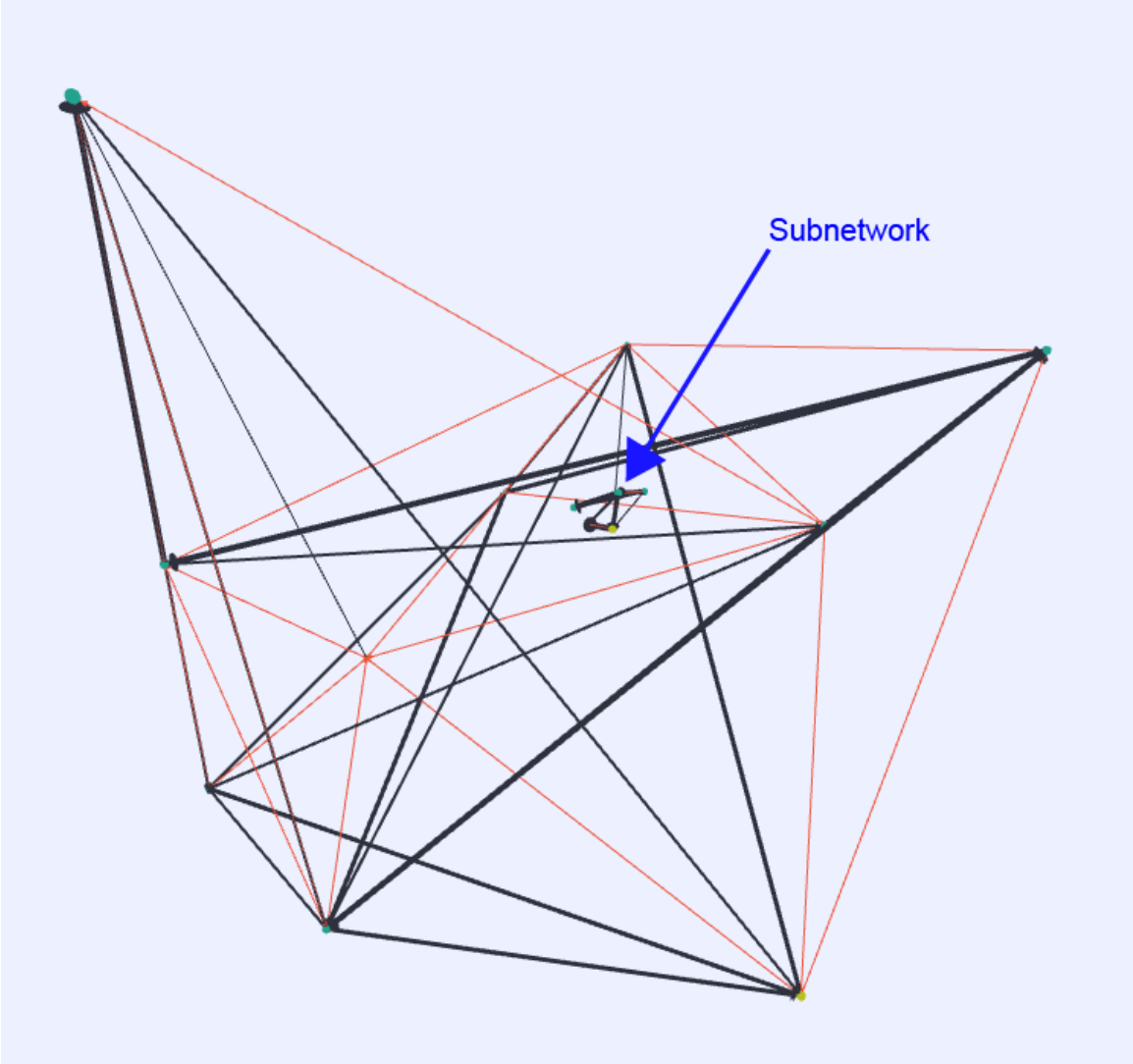


Figure 5.14: Example NIDA network with an embedded subnetwork. (Figure created in part by Margaret Drouhard and included with her permission.)

For example, we may define an affective network to have one input neuron and one output neuron. Within this definition, all neurons in the larger computational network connected to the subnetwork are connected to the input neuron, and all output from the subnetwork to the larger computational network is done via the output neuron. The synapses that extend from the output neuron of the subnetwork to neurons within the larger computational network may have a special property in that, rather than increasing or decreasing charge on a neuron, they increase or decrease threshold on a neuron. We may allow for outgoing synapses of affective systems that affect both threshold and charge, and let the optimization method decide which synapse type is appropriate, or we may restrict the optimization so that only one of the two types of synapses is allowed.

A second possibility for the subnetwork is to exclude connections from within the subnetwork to outside the subnetwork. For this case, we may define multiple output neurons that have different effects on neurons in the neighborhood of the subnetwork in the computational network. For example, suppose we define a neighborhood as neurons within a 10-unit radius of the subnetwork. Then, when one of the output neurons fire, rather than sending a pulse along one or more synapses, this pulse causes the thresholds of neurons, charge of neurons, or weights of synapses within that neighborhood to change by a small amount (defined by the output neuron).

5.2 Adaptation to Design Method

The original design method presented in Section 3.2 does not take into account the presence of one or more affective systems. The design method must be altered in order to allow affective systems to adapt along with the network structure. This section describes the changes that were made to the design method to accommodate the presence of either simple or complex affective systems.

5.2.1 Simple Affective System

Both crossover and mutation operations are altered slightly to compensate for the inclusion of the simple affective systems. In particular, the desired firing rate is included as part of the training. In the crossover operation, the average desired firing rate of the parents is taken to be the desired firing rate in both children produced. A new mutation that sets the desired firing rate to a randomly selected value between zero and the number of neurons in the network is added to the mutation operation. Parameters such as α and λ may also be altered through mutation and/or crossover as part of the adaptation to the affective system.

5.2.2 Complex Affective System

The inclusion of complex affective systems requires a major overhaul to the evolutionary optimization method used in designing the networks. In particular, for the inclusion of complex affective systems, we wanted to allow the user the ability to specify an objective for each affective system, which may be different from the objective of the entire network, which includes the computational network and all affective systems. Thus, there is an overall goal for the system to accomplish, but each affective system has a sub-goal that is related to the overall goal.

The first major change to the evolutionary optimization is that multiple populations are maintained. For any given task, a set of one or more affective systems is defined. Each of these sets of affective systems will have an associated goal. Then, the evolutionary optimization maintains a population of computational networks and a population of subnetworks for each affective system type. Each computational network contains zero or more placeholders for each affective system type. These placeholders exist at a point in the three-dimensional space in which the network exists and may be connected to other neurons in the network via synapses (Figure 5.15).

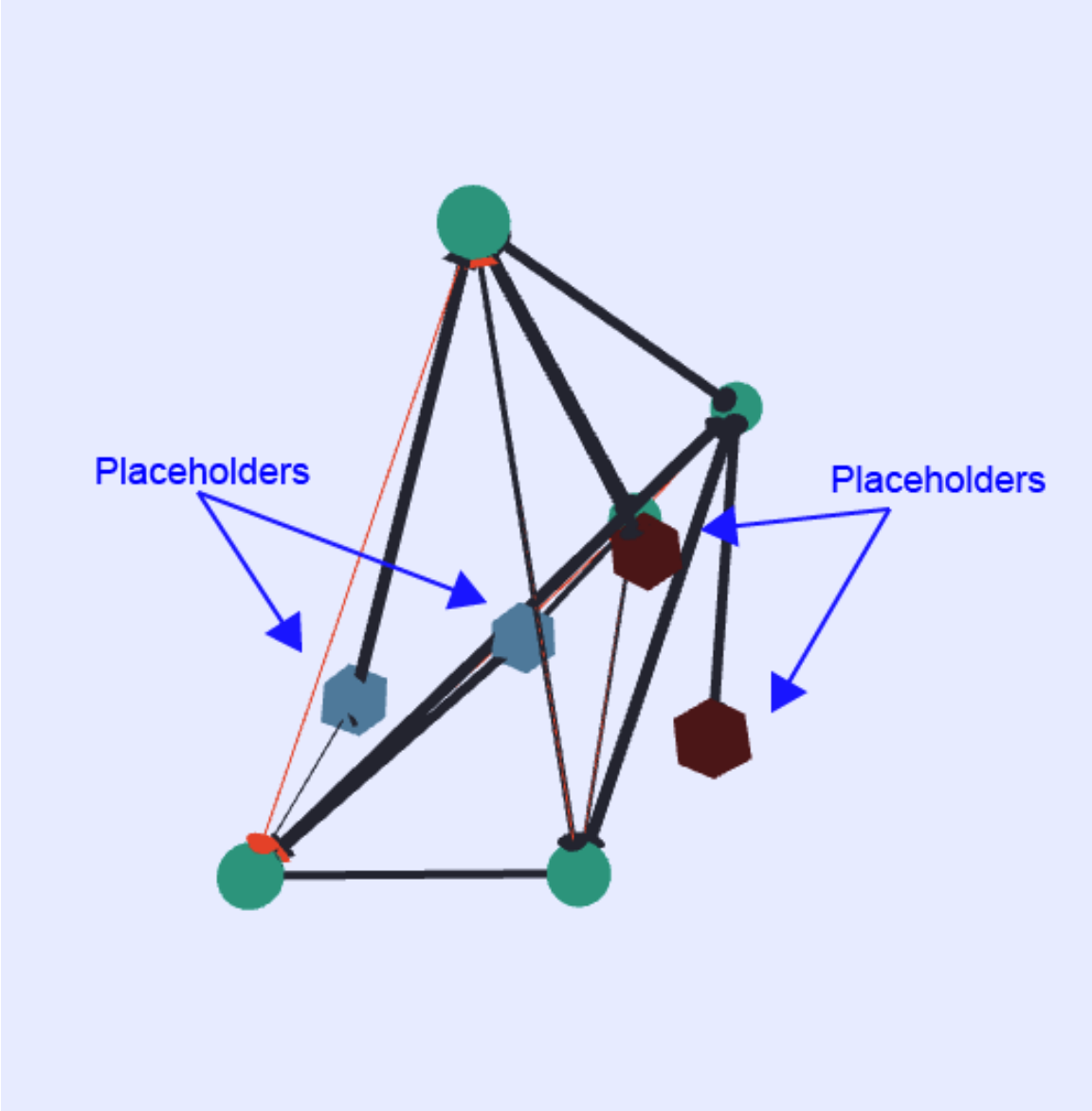


Figure 5.15: Example of a computational network with placeholders for two types of affective systems (shown as dark red and light blue cubes in the network). (Figure created in part by Margaret Drouhard and included with her permission.)

Two initialization methods are required for each task. One initialization method returns a computational network with appropriate placeholders for affective systems. The other initialization method returns a vector of affective subnetworks, one for each type of affective system. These initialization methods are used to initialize the populations of the new evolutionary optimization.

The selection method of the evolutionary optimization is also changed significantly to deal with multiple populations. In order to score a network using the fitness function, a "combined" network must be formed. This combined network consists of a computational network, which has placeholders for affective networks, and one of each type of affective network. Suppose for a task we have defined two affective types: seeking and fear. Then, there are three populations: a computational population, a seeking population, and a fear population. Each network in the computational population may have zero or more placeholders for seeking networks and zero or more placeholders for fear networks. To form a combined network, we select one network from each population: C from the computational population, S from the seeking population and F from the fear population. For each placeholder in C for a seeking network, we replace that placeholder with S and connect all synapses that go to that placeholder to the input neuron of S and all of the synapses that come from that placeholder to the output neuron of S. A similar process is completed for all fear placeholders in C, but we replace those with F. Figure 5.16 shows a combined network.

For the evolutionary optimization, there is a user-defined parameter ψ that determines the number of times each computational network will appear in a combined network. This parameter also determines (based on the relative population sizes of the affective populations) the number of times each affective network will appear in a computational network. For example, suppose the computational population size is 100, the seeking and fear populations both have size 20 ,and $\psi = 2$. Then, there will be a total of 200 combined networks. Each computational network appears in exactly two combined networks, and each affective network appears in exactly 10 combined

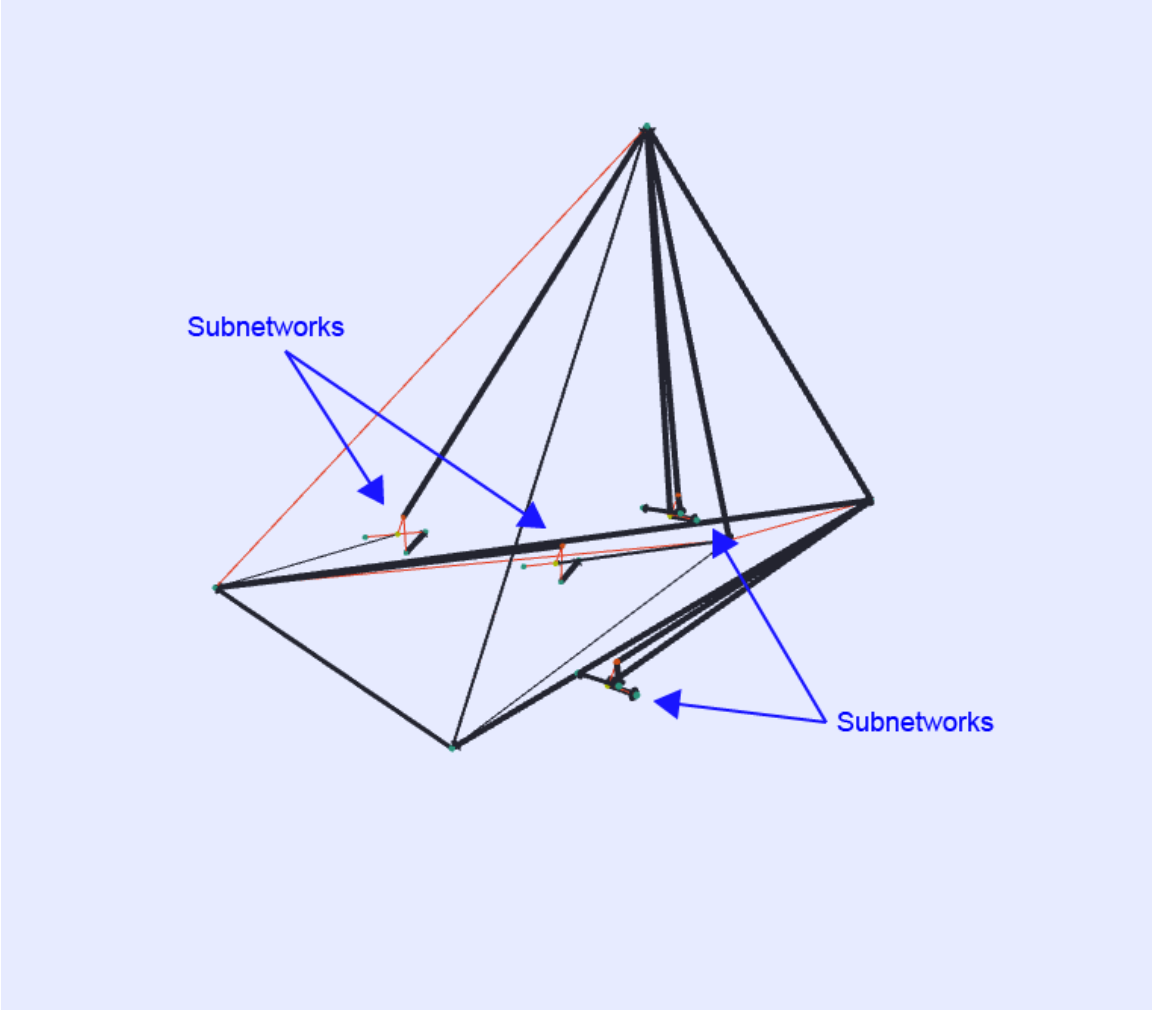


Figure 5.16: A combined network, comprised of a computational network and affective system networks. This is the same computational network shown in Figure 5.15, but an affective network from each population has replaced the placeholders shown in that figure. (Figure created in part by Margaret Drouhard and included with her permission.)

networks. To achieve this, each network in each affective system is given an ID, and a vector is formed for each affective system so that the ID of each network appears as many times as each network should appear in a combined network. In the previous example, the ID of each affective system would appear in the vector 10 times. Then, this vector of IDs is shuffled. The shuffled vectors are used to construct the combined networks (Figure 5.17).

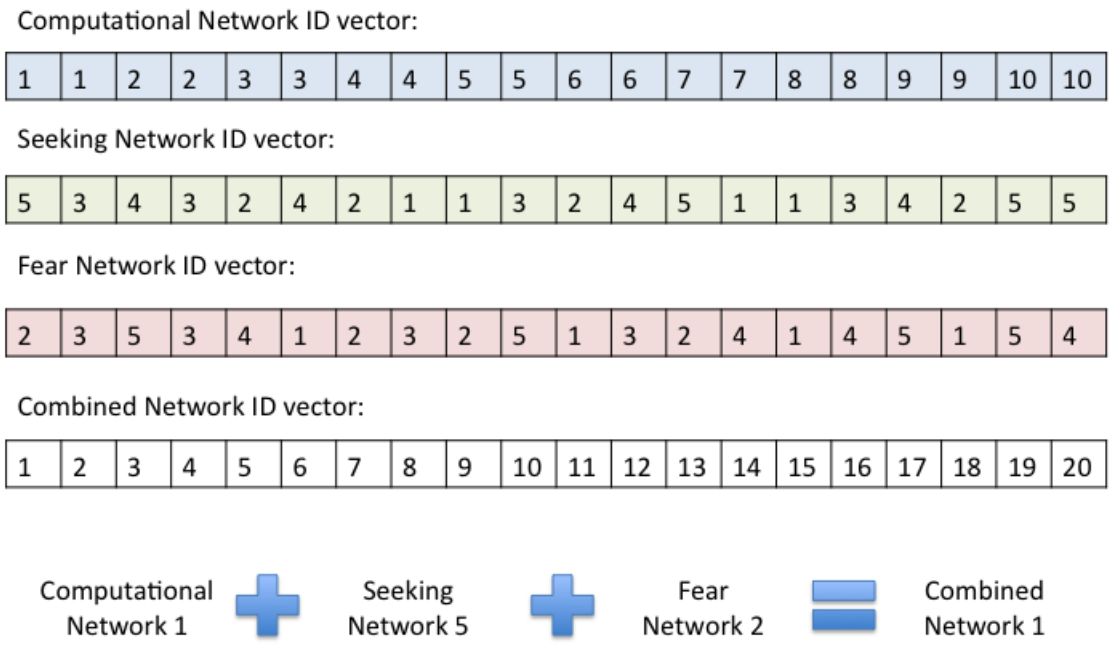


Figure 5.17: Example of a combined network building method. Vectors are formed of IDs that represent networks in each population. The affective vectors are shuffled. These vectors are then used to decide how to construct the combined networks.

As part of the selection method, each combined network is formed as described above and scored using a fitness function. The fitness function must also be altered slightly to accommodate affective systems. In this case, the fitness function returns a vector of scores. The first value in the vector is the overall score of the combined

network on the task or may also be interpreted as the score of the computational network included in the combined network. Each subsequent value in the vector corresponds to the score of that combined network with respect to a particular goal associated with each affective system. That is, the rest of the scores can be interpreted as how well the affective networks within the computational network are performing in the task. These scores are maintained for each computational network and affective network in each of the populations. An average score for each of the networks is determined by averaging the fitness scores of each combined network in which they appear. This average score is the overall fitness score for each network in each population.

Once the final scores are obtained, crossover and mutations are used to produce the child populations for each population in the evolutionary optimization. Crossover is altered slightly for computational networks in order to deal with placeholders. New mutations are added to insert or delete placeholders and connections to or from placeholders in the networks. The overall selection method is given as a flowchart in Figure 5.18.

5.3 Results

The inclusion of either a simple affective system or one or more complex affective systems has an effect on the performance of a NIDA network on tasks. This section presents the results of NIDA networks with simple affective systems and complex affective systems on the pole balancing task.

5.3.1 Effect of Simple Affective System

In this section, we describe the results associated with the inclusion of a simple affective system on the design method for the pole balancing problem. We found that the inclusion of the simple affective system had a small, positive effect on the overall

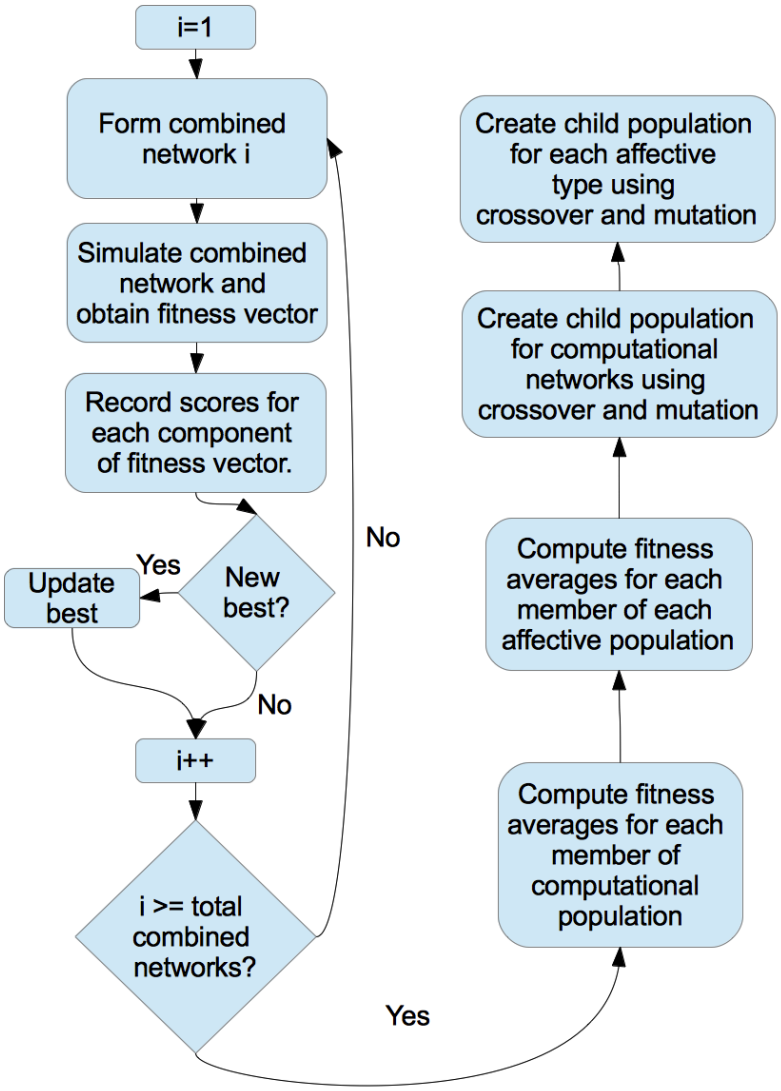


Figure 5.18: Selection method used to produce child populations from the current populations in the evolutionary optimization method updated to deal with complex affective systems.

performance of the networks. The parameters of the evolutionary optimization are given in Table 5.2. We chose the parameters of the affective system for this application based on the results from previous sections. Those parameters are also given in Table 5.1.

Table 5.1: Network and affective system parameters.

Parameter	Value
M_x	100
M_y	100
M_z	100
Network granularity δ	1
α	0.001
λ	0
LTP/LTD refractory	100 steps of simulated time
Amount LTP/LTD adjusted	0.001
Window size w	20

Table 5.2: Evolutionary algorithm parameters.

Parameter	Value
Population size	500
Mutation rate	0.9
Crossover rate	0.9
Tournament size	20

The application used to demonstrate the effect of the inclusion of an affective system is the single pole balancing problem. Specifically, the discretized version of the pole balancing problem with fitness function f_2, as described in Section 3.3.2, is used. The evolutionary algorithm is trained until some pre-defined level of performance is achieved. We define a network as fully-trained in the pole balancing problem if that network can keep the pole balanced and the cart between the endpoints for 300 seconds for each of the six test cases shown in Figure 5.19, where the linear and angular velocities are zero. These are the same test cases given in Table 3.4.

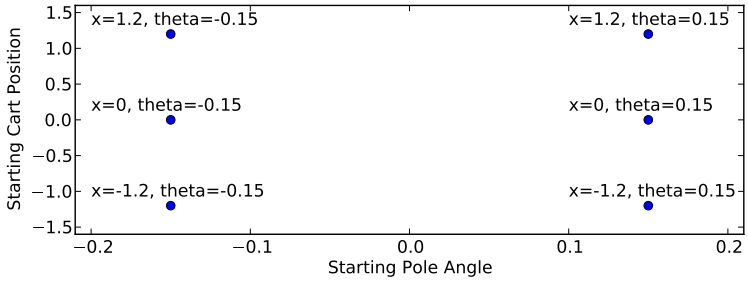


Figure 5.19: Six starting conditions for the fitness function. This graph shows the six starting conditions that are tested as part of the fitness function for the simple pole balancing problem. $\dot{x}$ and $\dot{\theta}$ are assumed to be zero.

Using this measure of performance, sixteen networks were evolved that satisfy the performance requirement. As part of training, the desired firing level was also found. We also obtained networks using the same random seeds (and thus the same initial populations) as the networks with the affective systems, and trained them without affective systems. Using these sixteen random seeds, only twelve of the networks converged in 100 epochs (whereas the average convergence time in epochs for networks with affective systems was 33.88). Since the other four did not converge in 100 epochs, they are not included in the results given below. It is worth noting that the results presented here are not conclusive because the number of networks found at this point is relatively small. Future work may expand upon the results shown here.

Box plots showing the sizes of the networks trained with affective systems (in terms of neurons and synapses), as well as the resulting desired firing rates for those networks are given in Figure 5.20. The average number of neurons in a network trained with an affective system is 24.83, while the average number of neurons in a network trained without an affective system is 25.25. Similarly, the average number of synapses in a network trained with an affective system is 223, while it is 239.25 in a network trained without an affective system. We do not view these differences as representative because of the small sample size. However, we expect the networks

with affective systems to be smaller than networks without affective systems, because of the extra level of complexity and resulting flexibility in operation that the affective systems add.

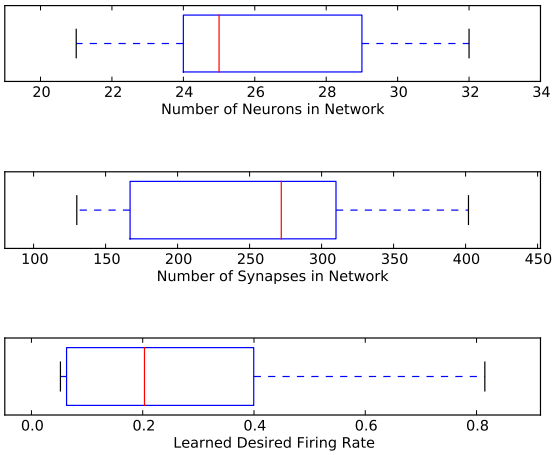


Figure 5.20: Box and whisker plots summarizing the complexity of the networks. The number of neurons, the number of synapses and the desired firing rate are shown for the sixteen networks trained with affective systems.

The resulting networks were tested to determine how well they generalized their solutions to the pole balancing task. To determine this, 10 random starting conditions were chosen from each of the following sets of ranges: $x \in [x_0, x_0 + 0.2)$, $\theta \in [\theta_0, \theta_0 + 0.05)$, where x_0 started as -2.0 and continued to 1.8 in increment of 0.2, and θ started at -0.15 and continued to 0.1 in increments of 0.05, resulting in a total of 1260 total test runs. On the contour plots, the points shown are the averages of the 10 random starting conditions in $x \in [x_0, x_0 + 0.2)$, $\theta \in [\theta_0, \theta_0 + 0.05)$.

We ran this test on three different network types: the networks trained with the affective systems; the same networks trained with the affective system, but not including the affective system; and networks trained without the affective system.

Figure 5.21 shows the results when the sixteen networks that were trained with the affective system underwent the generalization tests with and without the affective system operating during the tests. As can be seen in this figure, the operation of the affective system has an effect on the performance of the system. In particular, the

networks with the affective system generalized their solutions better than the networks without affective systems. This is to be expected, because the networks were trained to operate with the affective system. However, we saw two types of networks emerge in that group of sixteen: one group in which the affective system was vital for good performance and one group in which the affective system was not vital. The affective system was absolutely required for more than half of the sixteen networks in order to have good performance. It is likely that the inclusion of the affective system in training but not in operation helped find better thresholds, even in those systems in which the affective system was not vital for performance.

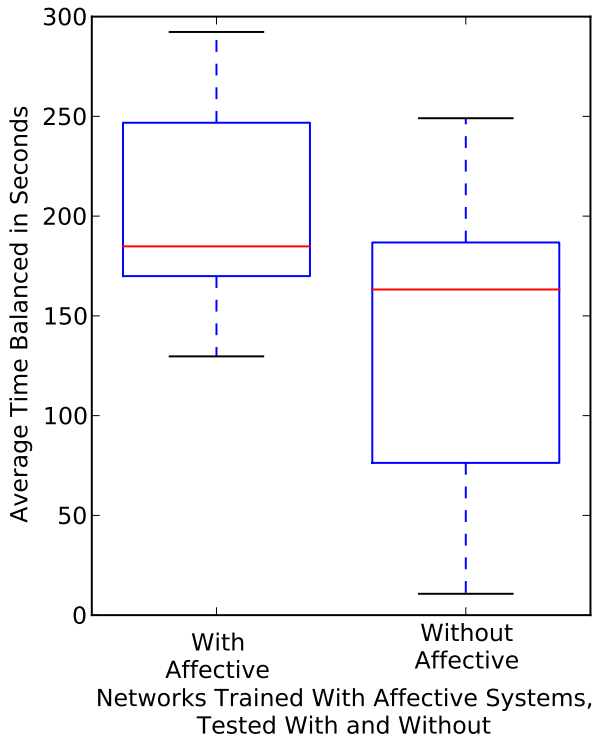


Figure 5.21: Box and whisker plots showing the results for networks tested with and without affective systems. The results shown in the box plots are the overall averages of the sixteen networks over 1260 test cases (10 random points chosen in each of 126 boxes).

Figure 5.22 shows the results of the twelve networks trained without affective systems (of the 16 that satisfied the criterion for completion of training), when compared with the same seeded twelve networks trained with affective systems. Again, it appears as though the networks trained with the affective systems perform better than those trained without affective systems. However, the difference is relatively small, especially when considering the small sample size. Also, only 75 percent of the training tasks completed without the affective system, whereas 100 percent of the training tasks that incorporated the affective subsystems completed.

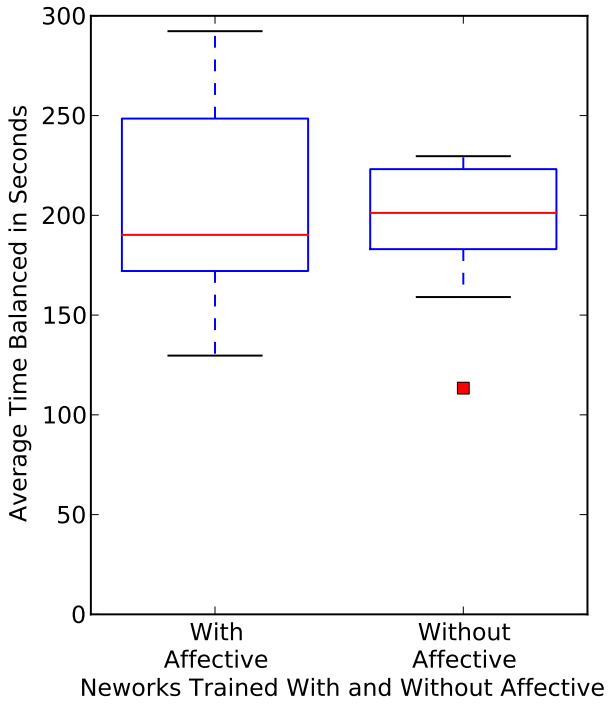


Figure 5.22: Box and whisker plots showing the results for networks trained with and without affective systems. The results shown in the box plots are the overall average of the networks over 1260 test cases. Since only twelve of the networks trained without affective networks converged, the results are shown only for those twelve networks, as well as the twelve networks with affective systems with the corresponding random seeds. The square in the plot is an outlier.

Figures 5.23, 5.24, and 5.25 show, respectively, the results for one network that was trained with the affective system and tested with the affective system, the same network tested without the affective system, and a network that was trained without the affective system, but started with the same initial evolutionary algorithm population. Each point on the contour map is the average of the 10 runs that correspond to that range. For this example, the network with the affective system performed very well on the generalization test, except for a small region in which the cart is very far to the right of the track and the pole is leaning very far to the left. The network trained with the affective system but tested without it performed second best; however, it is clear that the affective system was vital for the good performance of this network. Finally, the network trained without the affective system performed worst. In this case, the inclusion of the affective system in training vastly improved the generalization performance of the network.

5.3.2 Effect of Complex Affective System

We also used the pole balancing application with discretized inputs to demonstrate the use and the effect of the complex affective systems. In this section, we describe the results for our new design method. We found that the inclusion of the affective system during training had the unexpected result of producing networks that are more easily able to deal with noisy input. However, we also found that many of the resulting networks produced had evolved to exclude the complex affective systems. For the networks that did include affective systems, we found that the removal of those systems from the network caused little or no change in the performance of the network, indicating that for this task, the complex affective systems are not essential for success.

The type of complex affective system used for this task is a subnetwork that has only incoming synapses from the network to its input neuron. There are two output neurons in each subnetwork. When either one of those output neurons fire, they cause

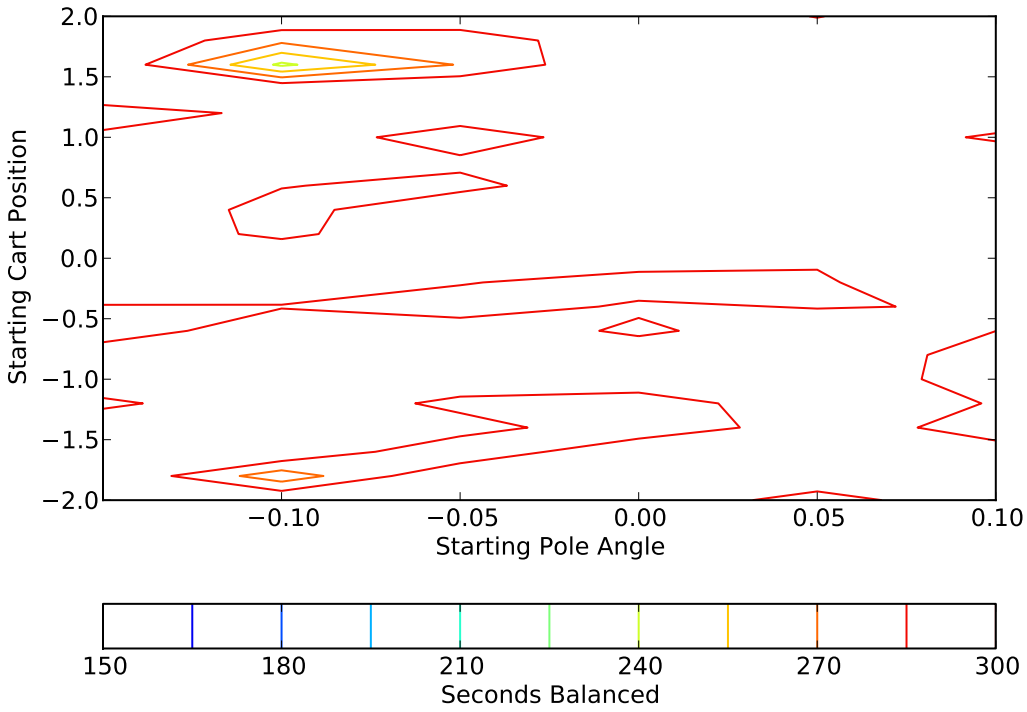


Figure 5.23: Plot showing the generalization results on a network with an affective system. Each point on the contour plot is the average of 10 test cases.

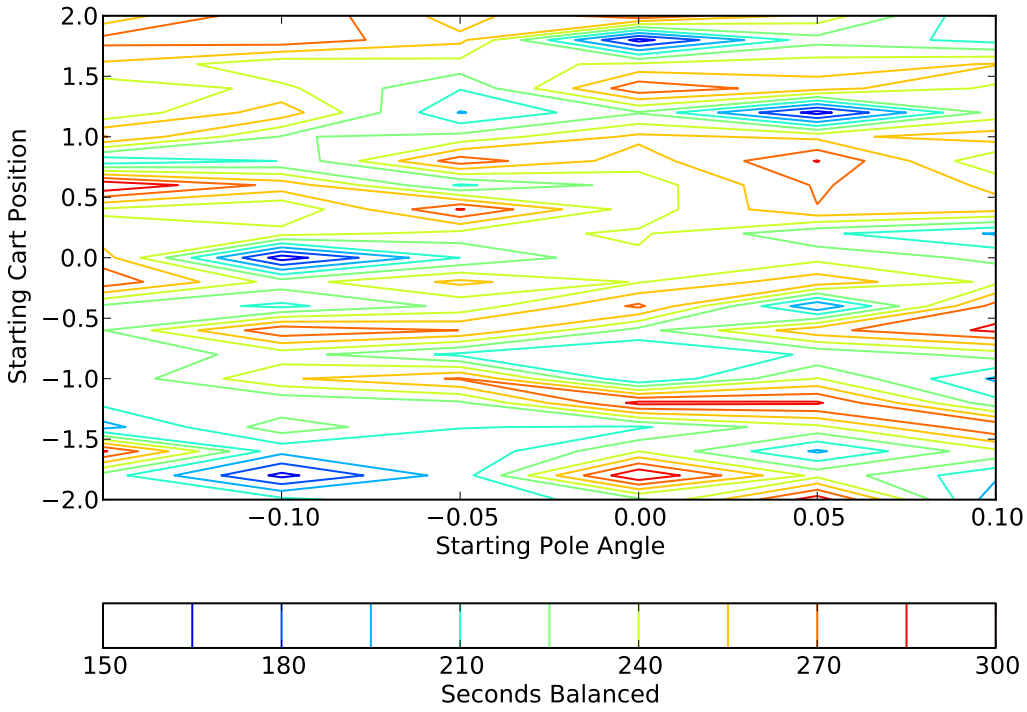


Figure 5.24: Plot showing the generalization results on a network tested without an affective system. The network in this case is the same network as in Figure 5.23, but the affective system is not operating during the tests. Each point on the contour plot is the average of 10 test cases.

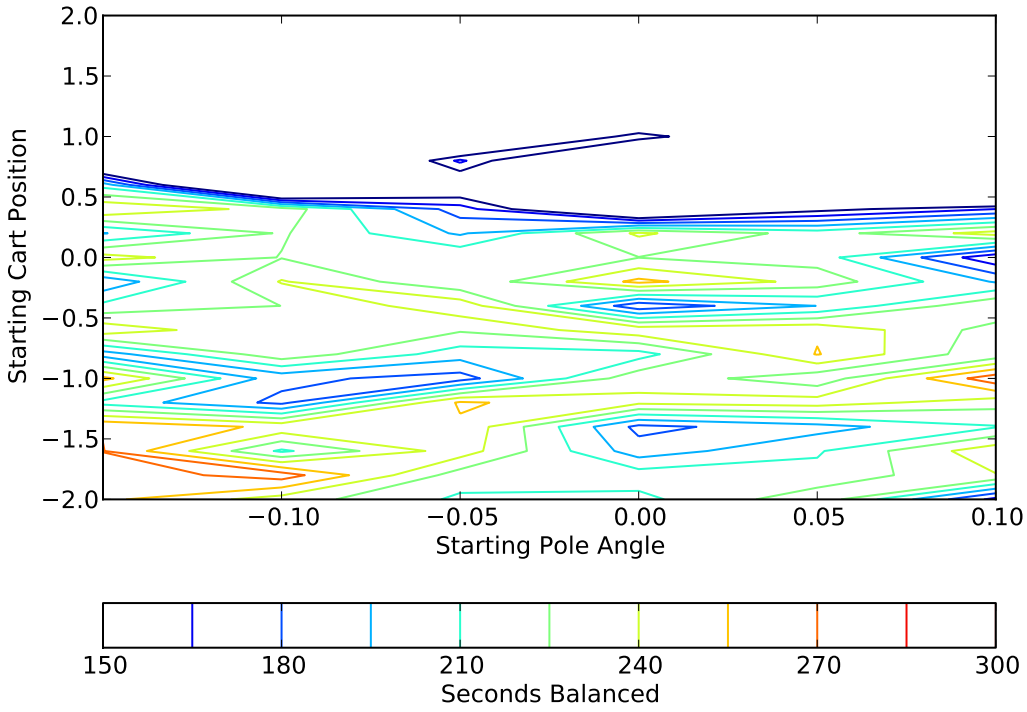


Figure 5.25: Plot showing the generalization results on a network trained without an affective system. The network was trained without an affective system, but the original population on the evolutionary algorithm was the same as the population for Figure 5.23 (they used the same random seed). Each point on the contour plot is the average of 10 test cases.

a change in charge on all neurons in the parent network that are within a 10 unit radius of the subnetwork. One output neuron corresponds to a positive change in charge, and the other output neuron corresponds to a negative change in charge.

In this case, there is a single affective system type,"error." The fitness function returns a vector of scores. The score for the combined network is the average of the number of time steps successfully balanced from six fixed starting conditions (fitness function f_2). The score associated with the affective system is related to the angle of the pole over the course of the simulation. The pole is considered to have fallen at time t if $|\theta_t| \geq 0.209 = \theta_{fail}$. Let T be the maximum number of time steps the pole was balanced for a single test case. If $T > 500$, then the score associated with the "error" affective system for that test case is:

$$\frac{1}{\theta_{fail}} \sum_{t=501}^{T} \theta_t \tag{5.5}$$

Otherwise, if $T \leq 500$, then the score associated with the "error" affective system for that test case is 100. We call this score $g_{affective}(Net, test)$, where Net is the network and $test$ is the current test case. The overall affective score of the network is then:

$$f_{affective}(Net) = \frac{1}{|TestCases|} \sum_{test \in TestCases} \frac{-g_{affective}(Net, test)}{f_2(Net)} \tag{5.6}$$

Where $TestCases$ is the set of the six test cases shown in Figure 5.19. Thus, the score is higher if the pole is kept closer to vertical after 500 time steps and lower if the pole is further from vertical after 500 time steps. The parameters for the network are given in Table 5.3. The mutation rates for this task are shown in Table 5.4. The parameters for the optimization method are shown in Table 5.5.

We ran 1000 tests for 24 hours each using these fitness functions and the new optimization method described in Section 5.2.2. The highest performing combined network over the course of optimization was determined for each of these 1000 tests.

Table 5.3: Network parameters.

Parameter	Value
M_x	100
M_y	100
M_z	100
Network granularity δ	1
LTP/LTD refractory	100 steps of simulated time
Amount LTP/LTD adjusted	0.001
Window size w	10

Table 5.4: Mutation rates.

Mutation Type	Approximate Mutation Rate
Update synapse weight	0.2532
Update threshold	0.2532
Add synapse	0.1266
Remove synapse	0.1266
Add neuron	0.1266
Remove neuron	.0633
Add subnetwork/placeholder	.0253
Remove subnetwork/placeholder	.0253

Table 5.5: Optimization parameters.

Parameter	Value
Computational network population size	100
Affective network population size	20
Crossover rate	0.99
Mutation rate	0.99
Computational network tournament size	10
Affective network tournament size	4
ψ (defined in Section 5.2.2)	2

Of these 1000 networks, 67 were able to balance for five minutes from all six starting positions. Only five of those 67 contained affective subnetworks; affective subnetworks had been trained out of the remaining 62 networks. For each of these networks, we performed similar generality tests on the networks as we did in Section 5.3.1.

For the five networks that contained at least one subnetwork, we removed the subnetworks to see how the performance would be affected. For all five of these networks, the performance on the generality tests changed very little or not at all upon removal of the subnetworks, indicating that the subnetworks had very little, if any, effect on the performance of the networks.

To discover whether the inclusion of the affective systems had any effect on the training and ultimate performance of the network, we ran 67 instances of the original design method (described in Chapter 3) with the same seeds as those used to generate the 67 networks described above, where the seed value is used to seed the random number generator. We imposed the same stopping conditions on these instances as in the complex affective system design method (24 hour time limit or balanced the pole from six starting conditions for five minutes), as well as one additional stopping condition. The additional stopping condition was based on the number of epochs required for the complex affective networks to train; in particular, if the original design method continued to train for more than 10 epochs beyond the training epochs for the complex affective system, the optimization was ended and the best network achieved up until that point was used. We were most interested in how the networks performed on the generality tests with respect to both how long the network was able to balance the pole, as well as the error of the pole (how far away the pole is from vertical) for each time step (0.02 seconds).

Figure 5.26 shows box plots of the average time the pole was balanced (over 1260 test cases) and the average angle from vertical of the pole (over 1260 test cases) for the 67 networks trained with affective systems and the 67 networks trained without affective systems. This figure shows that the networks trained with the affective systems performed consistently better on the generality tests and appears to have

achieved slightly lower (on average) pole error values, though these values are too close for any conclusions to be drawn. Thus, the inclusion of the affective systems in training may improve the overall generality performance of the networks.

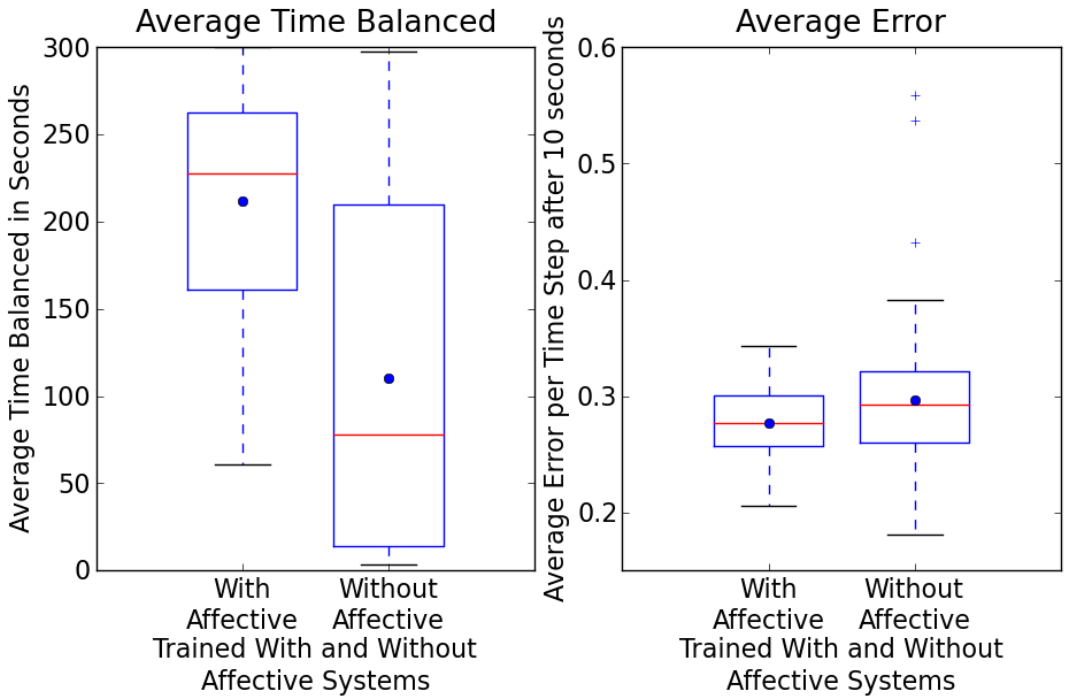


Figure 5.26: Box plot showing the pole balancing performance of networks trained with and without complex affective networks. The averages plotted are over 1260 test cases (as described in Section 5.3.1). Average fitness value and average error of the pole from vertical per time step are shown. The box extends from the lower to upper quartile, with whiskers showing the range of data. Outliers are also plotted, and the average is plotted as a dot.

Three of the 67 networks trained with affective systems were able to balance the pole for at least five minutes from all 1260 starting conditions, which we had not yet seen for any of the pole balancing tests for any other conditions (results for original design method given in Section 3.3.2 and results for simple affective systems given in Section 5.3.1). Figure 5.27 shows heat maps for both the time balanced and the pole errors for the best performing network in terms of balancing time for all

networks trained with the complex affective system. This figure shows that not only is the network very consistent in how long it can balance a pole, but it is also very consistent in how the pole is kept upright in terms the average error from vertical. We contrast this with the best performing network (of the 67) trained without complex affective systems, shown Figure 5.28. In this case, the network still performs fairly well, but does not achieve the consistency across all 1260 test cases exhibited by the network trained with the complex affective system.

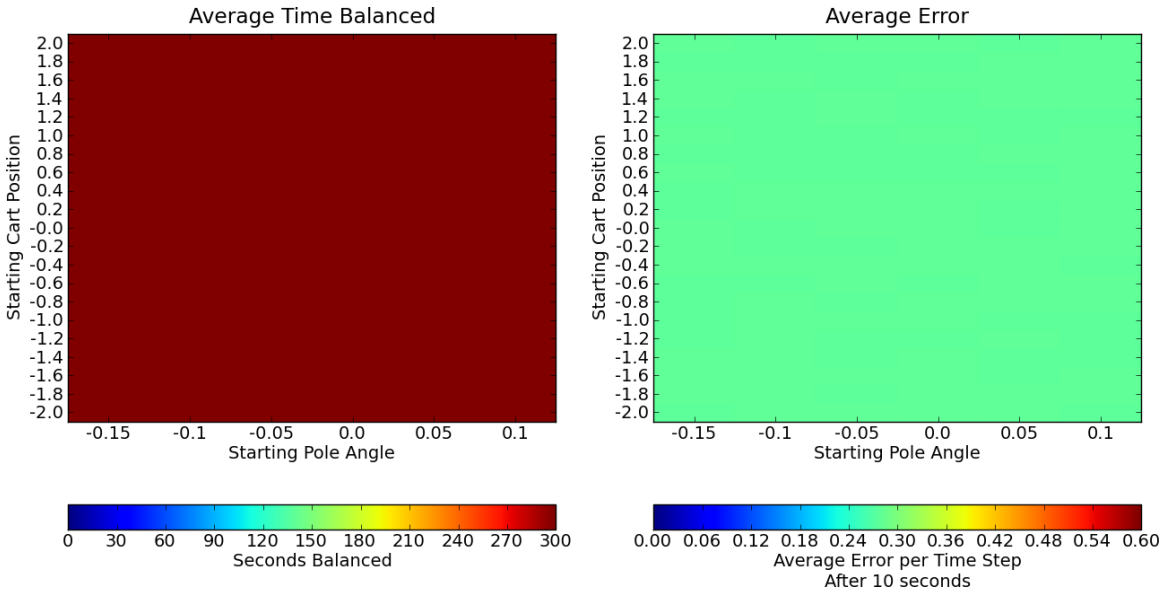


Figure 5.27: Heat maps showing the performance of the best performing network (with respect to time balanced from each starting condition) trained with complex affective systems. Each square in each heat map is the average of 10 test cases.

The best networks with respect to average error from vertical of the pole for the 1260 test cases are shown in Figure 5.29 and 5.30, trained with and without complex affective systems respectively. For both networks, they did not demonstrate the consistency seen in Figures 5.27 and 5.28. In this case, the average error from vertical of the pole per time step was lower for the network trained without the affective system, but that network also was not able to balance the pole on average.

This tradeoff between the two goals of the network was an outcome we were expecting. We did not expect the affective subnetworks to be almost completely

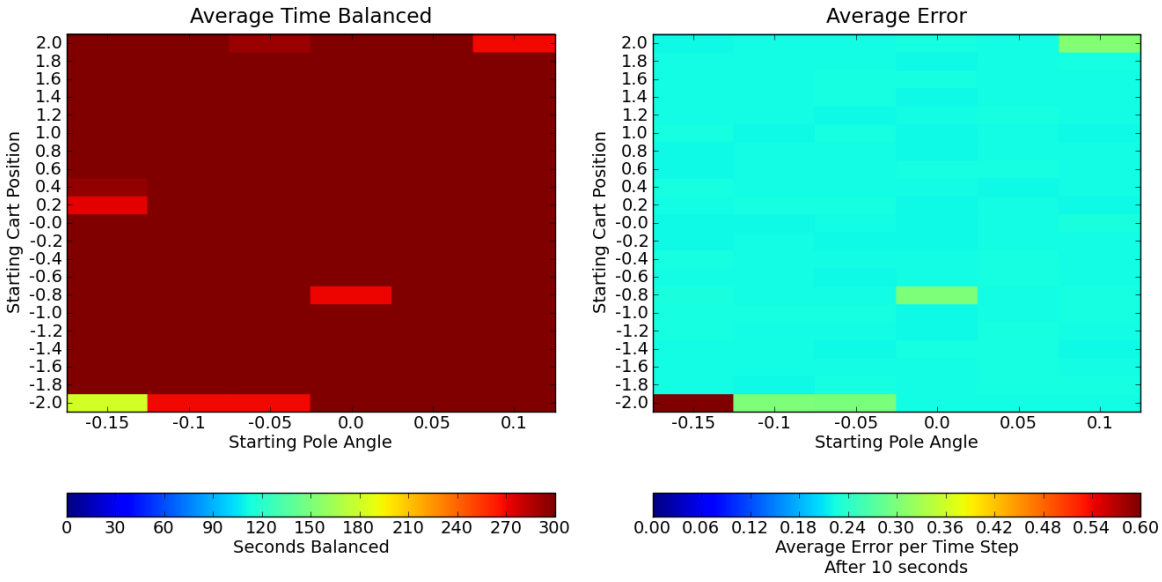


Figure 5.28: Heat maps showing the performance of the best performing network (with respect to time balanced from each starting condition) trained without complex affective systems. Each square in each heat map is the average of 10 test cases.

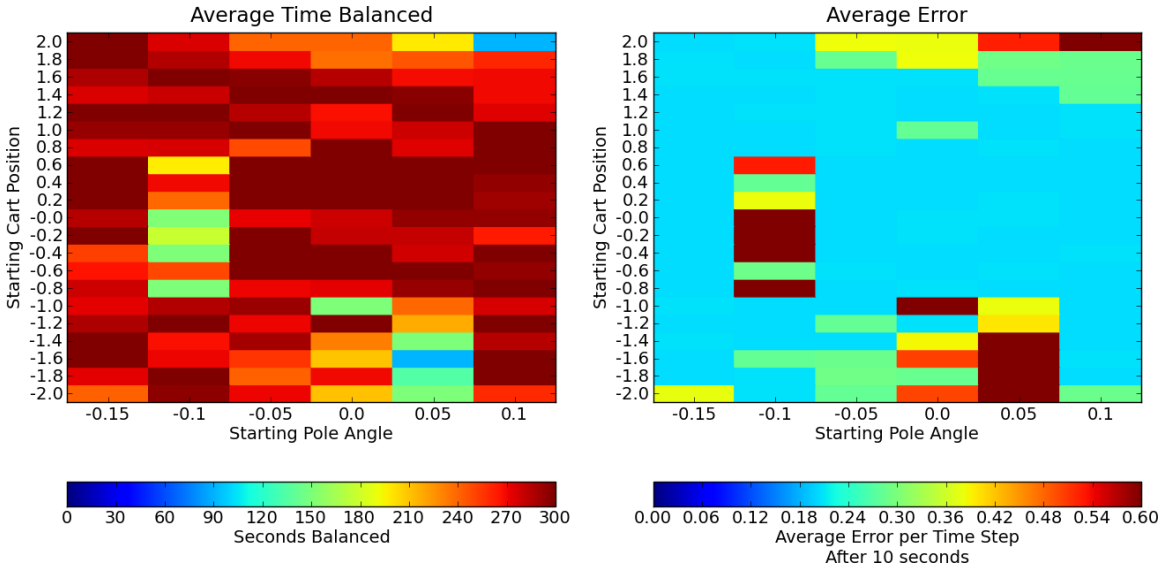


Figure 5.29: Heat maps showing the performance of the best performing network (with respect to error from vertical of the pole) trained with complex affective systems. Each square in each heat map is the average of 10 test cases.

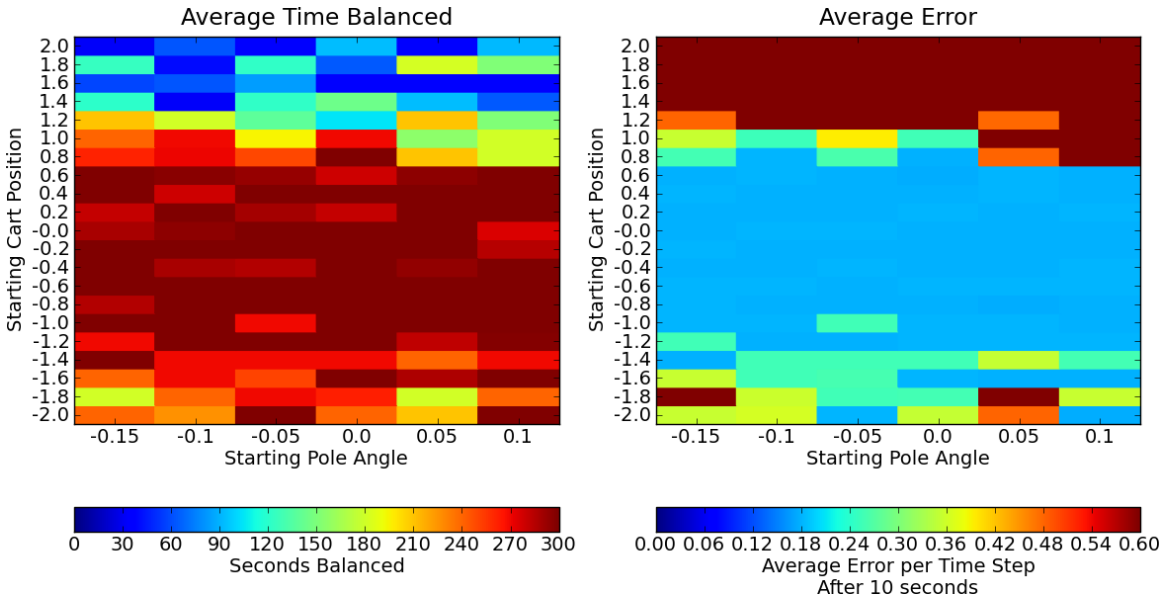


Figure 5.30: Heat maps showing the performance of the best performing network (with respect to error from vertical of the pole) trained without complex affective systems. Each square in each heat map is the average of 10 test cases.

eradicated over the course of training for our best performing networks. However, keeping the pole close to vertical is not stipulated and may actually be a competing goal to the overall goal of the system, which may be one factor as to why those subnetworks were trained out of the system.

Another unexpected occurrence with these types of affective systems was the production of the best generalizing networks of any training method and affective system inclusion we have seen so far on the pole balancing problem. It is possible that the inclusion of the affective subnetworks in the population of networks allowed for more robust networks to be developed over the course of training, as the parameters and structure of the network had to adapt to deal with occasional unexpected changes in charge (or noise) due to the affective systems. Since variation in inputs lead to changes in charge, by adding affective systems in training, we may have inadvertently produced networks that are able to generalize better to unexpected or unknown

inputs. This is certainly a valuable result, one that may be exploited in the future for other types of problems where noisy input values need to be recognized.

5.4 Multiple Interacting Networks

This hierarchical definition of the network structure and the multiple-objective evolutionary optimization method defined may be extended to train multiple interacting networks for a task. In this case, the computational network is actually multiple team member networks that exist and are not connected to one another, but operate on the same time scale and interact through some mechanism (e.g., firing of certain output neurons correspond to communications sent to other networks). Each affective type represents a role in the team. A computational network in this case is comprised only of placeholders; there are no neurons within the computational network itself nor are there connections between the placeholders (see Figure 5.31). The computational network optimization is then finding the number of members per role and the total number of members required to complete the task.

For a team of networks, as in the computational network with affective systems case, there is an ultimate goal of the system. In this case, the goal is related to the task that the team is attempting to complete. Each role in the team (which may be completed by multiple team members or networks) has an associated fitness score (just as each affective system has its own fitness score). Thus, the same (or very similar) changes to the evolutionary optimization described in Section 5.2.2 also apply in this case. The changes required to adapt to the multiple-interacting networks case are those that restrict the types of structural changes that can occur at the computational network (or whole team) level, in that no neurons or synapses exist at that level.

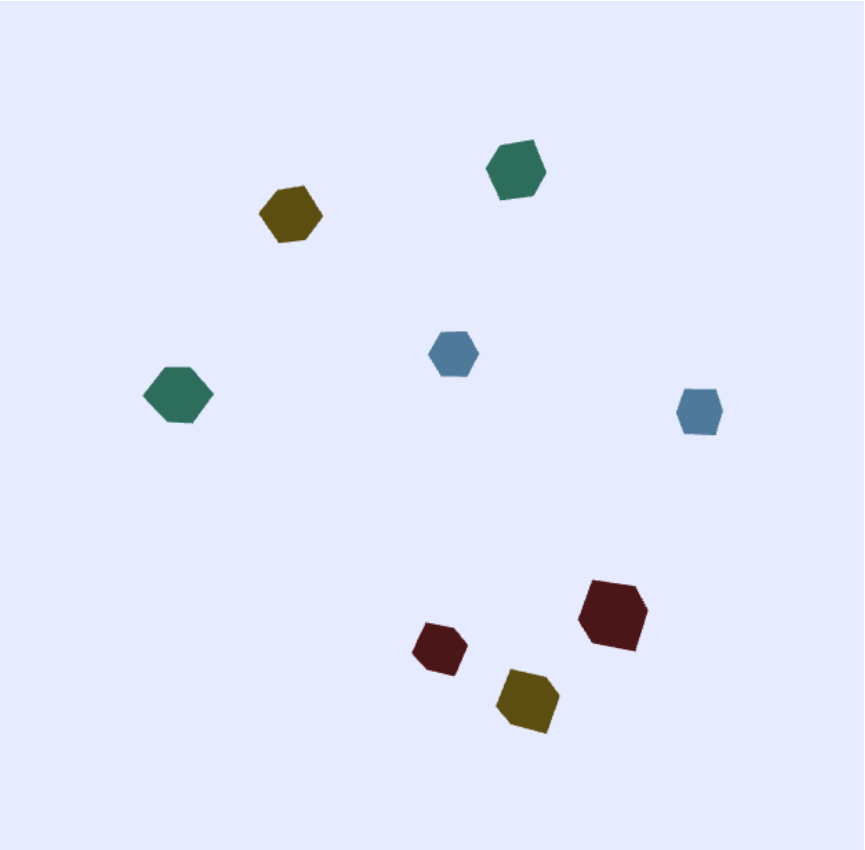


Figure 5.31: A computational network in the multiple interacting networks case is comprised only of placeholders, not neurons or synapses. In this example, there are four affective types, which correspond to four possible roles in the team. (Figure created in part by Margaret Drouhard and included with her permission.)

5.5 Summary

This chapter describes affective systems within the context of NIDA networks. It describes both a simple affective system (based on a set of equations) and a complex affective system (based on nested subnetworks). We examine the simple affective system in detail and give results on how it affects the behavior and parameters of general NIDA networks. We describe a version of the design method that incorporates affective systems and sub-goals, and we describe how this method may be extended to deal with training multiple interacting NIDA networks on a team-based task. We also describe how the inclusion of simple and complex affective systems affected the performance of networks on the pole balancing task.

This was the most exploratory portion of this work. We found that the inclusion of both simple and complex affective systems improved generalization ability of NIDA networks on a pole balancing task. For complex affective systems, however, we have not obtained useful results on how well the new design method incorporating sub-goals performs. In the application described, all of the subnetworks responsible for the sub-goals were either evolved out of the network or were "vestigial" (were of no use in the final network). However, we believe that the results obtained indicate that the inclusion of these affective-like systems over the course of training may be very useful in producing more robust NIDA networks. Future directions for the development of affective systems and associated design methods are described in Section 7.3.

6

Hardware Implementation

"If you've never programmed a computer, you should. There's nothing like it in the whole world. When you program a computer, it does exactly what you tell it to do. It's like designing a machine – any machine, like a car, like a faucet, like a gas-hinge for a door – using math and instructions. It's awesome in the truest sense: it can fill you with awe."

– Cory Doctorow, *Little Brother*

This work inspired a hardware implementation of the NIDA architecture: Dynamic Adaptive Neural Network Array (DANNA). The hardware implementation of the array element was written in VHDL by Dr. Mark Dean and implemented in a field programmable gate array (FPGA) by a senior design team in Spring, 2014. This chapter gives a brief overview of DANNA networks and the optimization method used to design them, and is given from the software perspective. A more thorough description of the hardware implementation is given in [36, 38, 39, 66].

6.1 Dynamic Adaptive Neural Network Array

DANNA networks are composed of a grid (array) of elements. Elements are programmable as at least either a neuron or synapse. Each element is connected to eight of its neighbors via connection ports (labeled 0-7 in Figure 6.1). This was later extended to support 16 connections.

Neurons have a threshold parameter. In the current implementation, neurons may either be input neurons or internal (hidden) neurons. Neurons are connected to other neurons via synapses. The connections are directed, so neurons have a set of synapses that go to that neuron and a set of synapses that come from that neuron. The primary actions associated with neurons are changes in charge and firing. Changes in charge at a neuron occur when one of the synapses connected to that neuron emits an event (described below), and the charge changes by an amount determined by the weight of that synapse. When a neuron's threshold is reached, the neuron fires, emitting pulses to all of the synapses that receive signals from that neuron. Upon firing, the charge on the neuron is set to a bias level, which is defined as a parameter of the network. Just as in NIDA networks, neurons in a DANNA enter a refractory period after firing, during which their charge may change but they may not fire. Each connection port on a neuron is either enabled or disabled so that input from that port is included in an update of the neuron's state or ignored, respectively.

Synapses have two parameters: weight and delay. Weight represents the strength of the connection and may be positive or negative so that the synapse has an excitatory or inhibitory effect, respectively. Delay governs how long it takes for charge to travel along a synapse. Delay is implemented in the element as a first-in, first-out (FIFO) shift register. Processes similar to long-term potentiation (LTP) and long-term depression (LTD) are implemented in synapses in DANNA. Synapses have only one input port enabled and one output port enabled, and both of these ports must be connected to neurons for functionality. Alternatively, synapses may be chained together, in which case only the last synapse in the chain (the one with

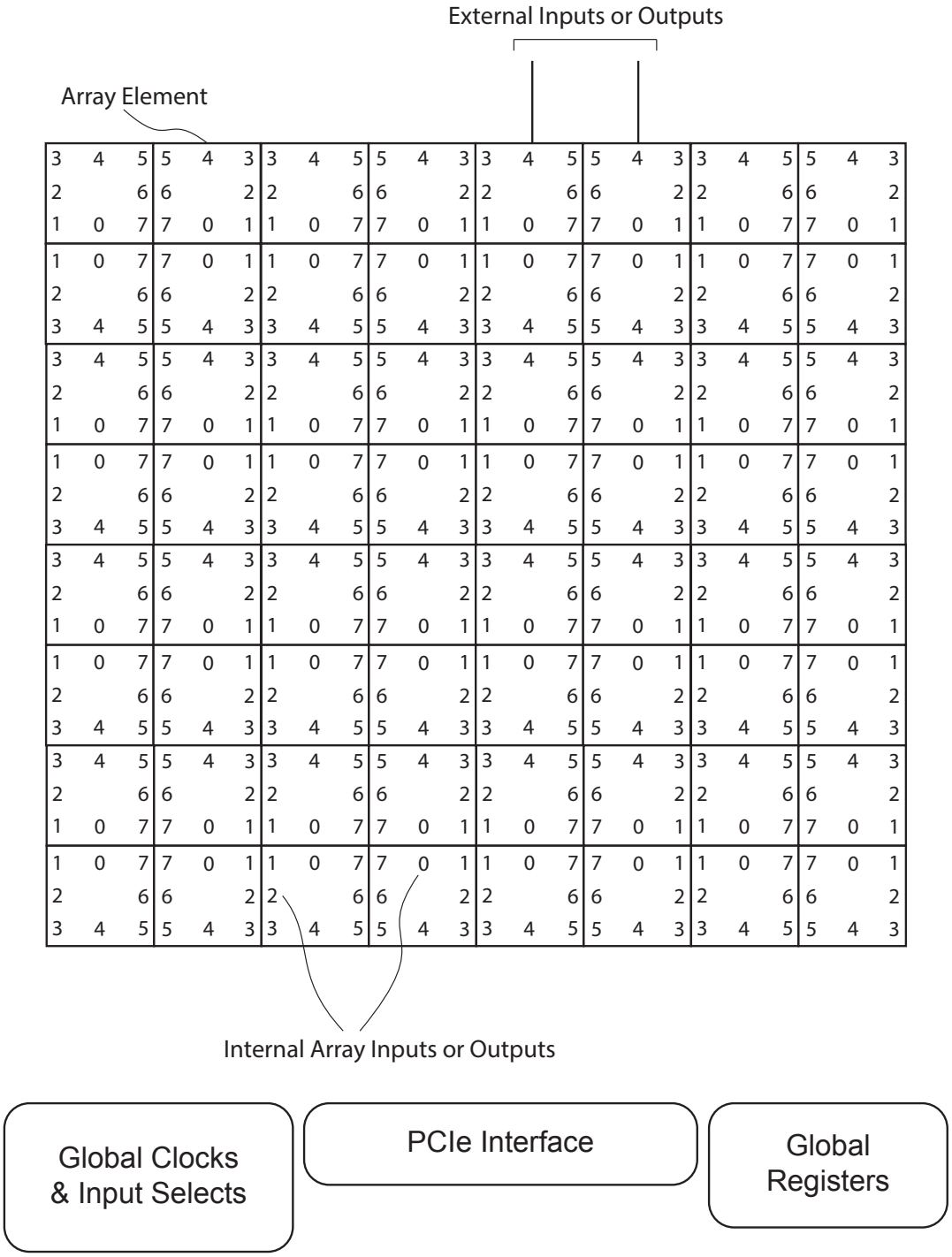


Figure 6.1: Grid structure of the hardware implementation based on the NIDA architecture. (Figure created by Dr. Doug Birdwell and Dr. Mark Dean and included with their permission.)

its output connected to a neuron) is affected by LTP/LTD. When the input neuron to the synapse is enabled, a firing event is added to the shift register. This firing event will be emitted from the synapse when it reaches the end of the shift register, implementing the delay. Upon reaching the end, the synapse is turned on and the weight of the synapse is added to the charge of the neuron connected to the output of that synapse (if the neuron's input connection to that element is enabled). If the output neuron is in its refractory period when this occurs, a depression of the weight value of the synapse occurs (in a process similar to LTD). If the change in charge on the output neuron causes that neuron to fire, then a potentiation of the weight value of the synapse occurs (in a process similar to LTP). The synapse also has a programmable LTP/LTD refractory period to limit the rate of change of the weight value over the course of execution. Upon depressing or potentiating, the synapse enters a refractory period in which all operation continues as normal except the LTP/LTD-like operations do not occur. Synapses operate as the output elements for the DANNA networks so that the output of the network is a value (the weight of the synapse), as opposed to the neurons, which only output fire events and do not have an associated value.

DANNA networks operate on two clocks, the global simulation clock (or network clock) and the element clock. The element clock operates at eight times the speed of the global simulation clock (or, in general, the element clock operates at *number of connection ports* times the speed of the global simulation clock). The element clock corresponds to the sampling of the eight connections ports for each element. At each step of the element clock, if the corresponding input for each neuron element is enabled, then that input is sampled, and the state of the element is updated. The starting connection port number is randomized at each step to avoid potentiating and depressing the same synapses repeatedly [38]. For example, for global network clock cycle 0, the eight corresponding element clock cycles may sample inputs 4, 5, 6, 7, 0, 1, 2 and 3 in sequence, but in global network clock cycle 1, the eight corresponding element clock cycles may sample 7, 0, 1, 2, 3, 4, 5, and 6. The

global input select is a network parameter that specifies the starting connection port number for each network clock cycle and is updated randomly before the beginning of each network clock cycle. Synapses sample their one enabled input port at the beginning of the network clock cycle and update the state of the element at that time by updating the shift register representing the delay as needed.

A complete software model of the hardware implementation was developed for testing and training purposes. Though this model is similar in nature to and inspired by NIDA networks, there are several fundamental differences, where the key difference is the hardware model is clock-driven, while the NIDA network model is event-driven. The software simulation of the hardware implementation operates sequentially, but must be designed so that it appears that all elements are updated simultaneously on the element clock. To do this, two full grids are maintained, an "old" grid of elements and a "new" grid of elements, where the new grid is updated based on the state of the old grid. Moreover, the timing of each interaction between elements has to match the interaction of those same components in the hardware exactly in order to correctly simulate the behavior of these networks. For example, upon detecting that the threshold has been reached, a neuron element's output in the hardware array does not turn on for three element cycles (because of the computation that occurs in the element). Thus, these three element cycles required for computation have to be included in the software simulation in order to mimic the hardware implementation.

6.2 DANNA Design Method

A design method based on evolutionary optimization was developed to accommodate the new architecture and its restrictions. Mutations for this architecture include changing parameter values on elements, setting an element to be a neuron or a synapse, making a new path of elements between two neurons, or deleting neurons or synapses in the grid. Crossover was adapted to deal with the grid structure rather than a three dimensional cube. Two types of crossover were implemented: line

crossover (similar to plane crossover described in Section 3.2.3) and random crossover. Line crossover randomly selects a row (column) in the grid. Then, child 1 receives everything above the line (to the left of the line) from parent 1 and receives everything below the line (to the right of the line) from parent 2, while child 2 receives everything above the line (to the left of the line) from parent 2 and everything below the line (to the right of the line) from parent 1. Random crossover iterates over all elements of the grid for each child and randomly selects (with equal likelihood) the corresponding element from either parent 1 or parent 2. The crossover operation is more difficult for DANNA than NIDA because of the connection lines between elements and how inputs and outputs are enabled to make connections between elements. For example, it is possible that, using either random crossover or line crossover, a neuron element is completely surrounded by other neuron elements, making that neuron unable to connect to any other neuron via a synapse (see Figure 6.2).

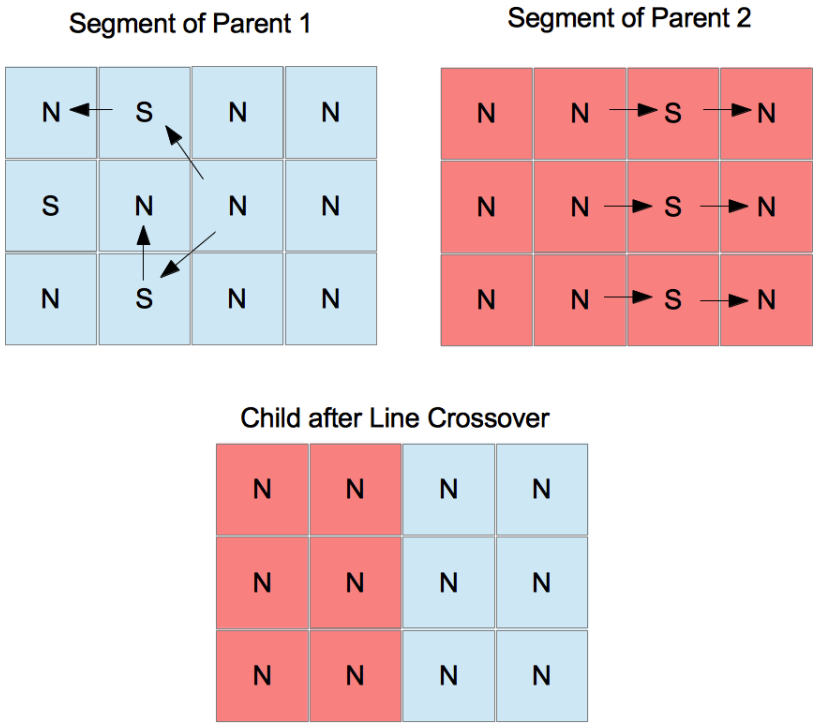


Figure 6.2: Segments of a grid of two networks in which line crossover produces a child with isolated neurons (neurons that are unable to connect to another element because they are surrounded by neurons instead of synapses).

There are several issues with developing the structure of the network beyond the issues with crossover given above. In the original implementation, each element can connect to eight of its neighbors. Depending on the type of element, this connectivity may be restricted. For example, one definition of a synapse only has one enabled connection from another element and only connects to one element (which may be the same element). Thus, the other six or seven output ports are not in use. Because of this definition of a synapse, arbitrarily defining a path in the grid may inadvertently cut off a portion of the grid from the current network, as in Figure 6.3.

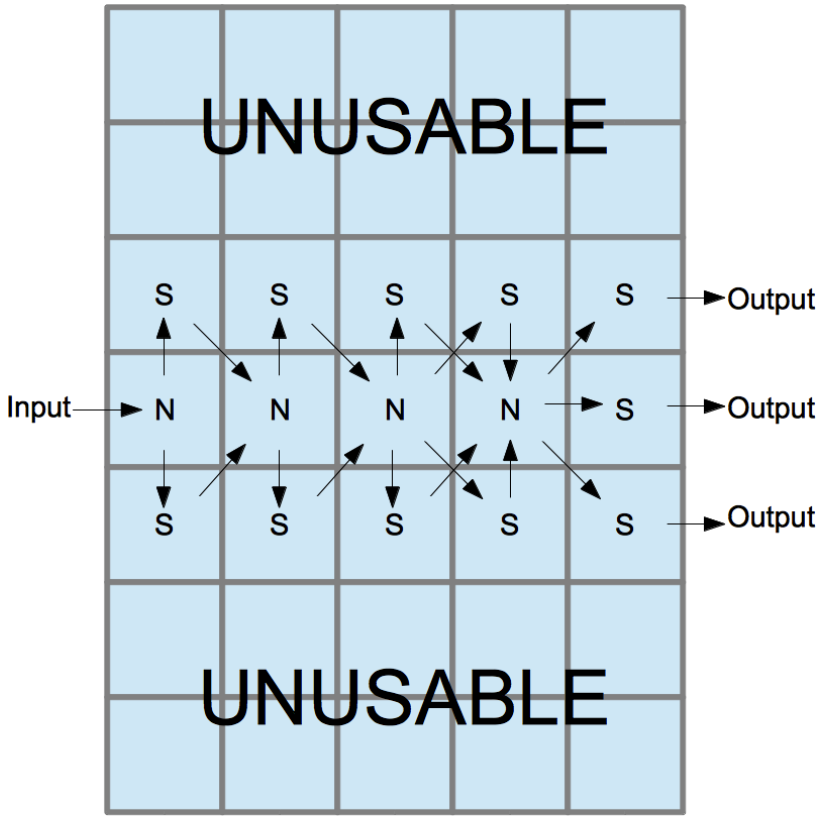


Figure 6.3: An example of a network in which the paths defined make a large portion of the potential grid elements unusable.

Moreover, there are many structures that do not utilize as much of the connectivity of the network as may be needed to solve many complex problems. Thus, it may be advantageous to pre-define a grid structure, so that when designing the network, only parameters are optimized (thresholds of the neurons and delays and weights of the

synapses). We define one grid structure in terms of a 4x4 element block. An example of a 4x4 element block is given in Figure 6.4.

N	←	N	←
↑	↙	↓	↖
N	→	N	→
↑	↗	↓	↘

Figure 6.4: An example of a 4x4 element block, where elements labeled N are neurons and elements labeled with ↑, →, ↓, or ← are synapses where the direction of the arrow indicates the direction of the synapse.

These 4x4 blocks of elements may be tiled to form arbitrarily large networks. These larger network blocks have built-in cycles (Figure 6.5) that are likely to be useful in many computational tasks, as we know that cyclical connections play important roles in the mammalian brain.

At present, the simulation software for the DANNA networks is too slow to be used as part of the evolutionary optimization to obtain meaningful results. We have implemented the anomaly detection task described in Section 3.3.1 in the software simulation for DANNA and have seen promising results, but training was too slow to give a complete solution. The best network produced for the software utilized the repeating grid structure given above and is shown in Figure 6.6. The corresponding output graph for that network is shown in Figure 6.7, showing the network is beginning to detect decreases in firing rate.

N	←	N	←	N	←	N	←
↑	↙	↓	↖	↑	↙	↓	↖
N	→	N	→	N	→	N	→
↑	↗	↓	↘	↑	↗	↓	↘
N	←	N	←	N	←	N	←
↑	↙	↓	↖	↑	↙	↓	↖
N	→	N	→	N	→	N	→
↑	↗	↓	↘	↑	↗	↓	↘

Figure 6.5: Example cycles in a tiled example of the 4x4 element blocks.

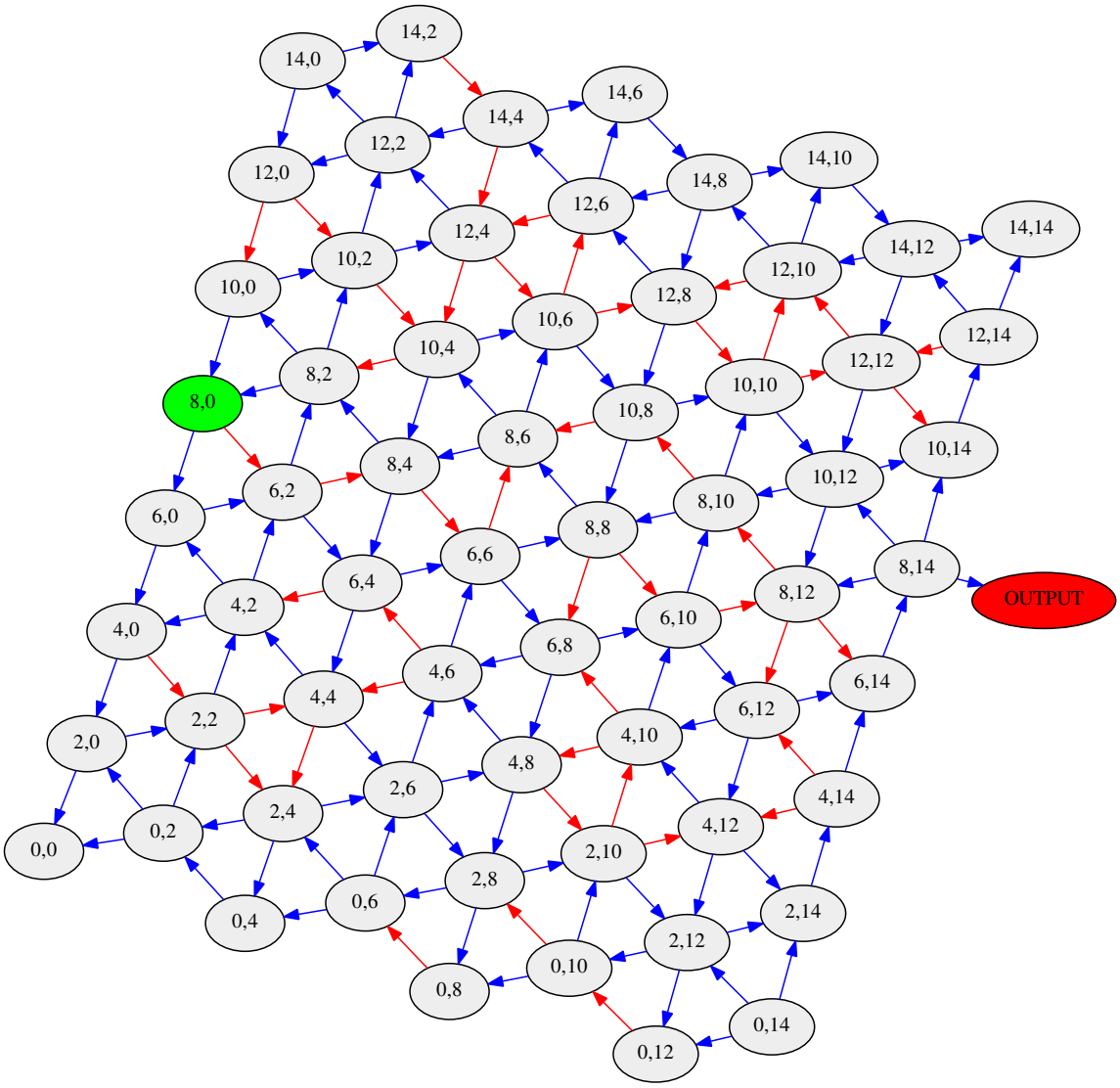


Figure 6.6: A network that was partially trained to recognize changes in input firing rate. This network is made up of 4x4 blocks similar to those given in Figure 6.4. The parameters of the network (thresholds of neurons and weights and delays of synapses) were trained using evolutionary optimization. The neurons are represented by ovals and the synapses are represented by arrows. The red oval for the output indicates which synapse corresponds to the output synapse. The green oval is the input neuron. Neurons are labeled by their row and column in the grid. Excitatory synapses are blue, and inhibitory synapses are red.

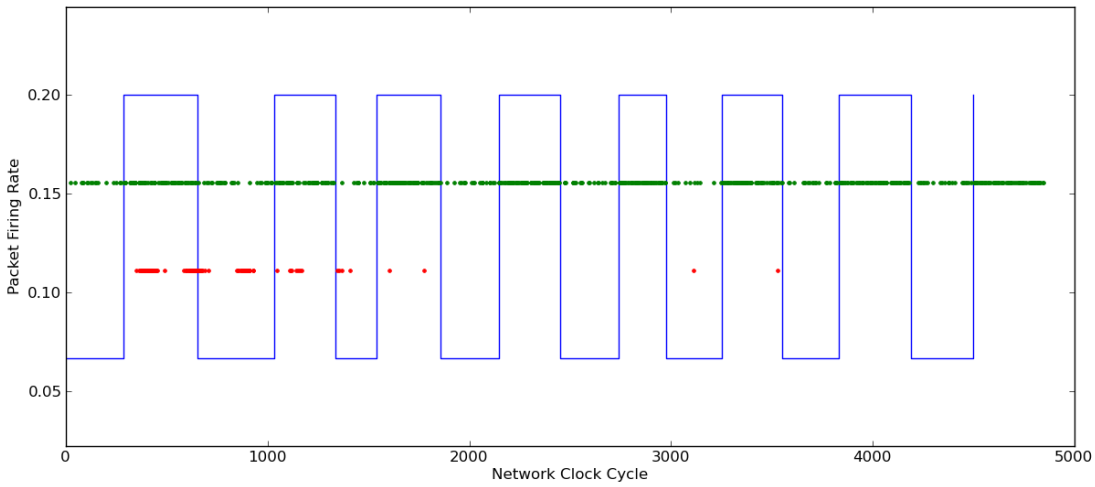


Figure 6.7: An example run of the network given in Figure 6.6. The green dots correspond to input pulses, the red dots correspond to output pulses and the blue line shows the packet firing rate.

6.3 Summary

A hardware implementation based on NIDA networks called the Dynamic Artificial Neural Network Array (DANNA) has been developed. This chapter describes in general terms how the operation of DANNA differs from NIDA, and it also briefly describes the software simulation of the hardware that was developed alongside the hardware implementation. An adaptation to the evolutionary optimization design method was also developed to work with the DANNA array, and some very preliminary results are given.

This work differs from the neuromorphic implementations discussed in Section 2.6 in several ways. First, we implement a simple neuron model with two parameters (threshold and refractory period). This is contrasted with more complicated neuron models, some of which have as many as 14 parameters. There is some question as to how to select these parameters and which are important for functionality of the system. We speculate that there can be a tradeoff between complexity of neurons

and number of neurons in the network in order to account for any functionality that may be lost by reducing the complexity of the neuron.

A second major difference between our neuromorphic implementation and other implementations is the connectivity of the synapses. At present, we allow our neurons to only be directly connected to a small number of other neurons (in the described implementation, the maximum allowable number of connections is eight, but this has recently been extended to 16). In most neuromorphic implementations, there are a very large number of connections between neurons, sometimes as many as to allow for a fully connected network. We speculate that this complexity is not required to solve many problems. Moreover, because we allow for fewer synapses, we can make the synapses more complex in that they implement programmed delays.

Finally, our implementation discovers weights, thresholds and delays through an evolutionary algorithm training process. It also allows for structural modifications to the network, though they are harder to update. This is another major difference from most neuromorphic implementations, some of which select connectivity based on what is known about biology or use traditional neural network architecture restrictions (such as a feed-forward neural networks). Parameters are also often selected based on biology, or the weights are trained primarily through spike-time dependent plasticity. This opens a major question for even biology-driven neuromorphic implementations. In many cases, it is difficult to determine values for parameters to produce some behavior. We speculate that it may be possible to use an evolutionary optimization on these other architectures to determine good parameter values.

The hardware implementation has been updated to allow each element to be connected to 16 other elements, rather than eight. This reduces and may eliminate the problem with blockage in array due to synapses as seen in Figure 6.3. In parallel, a new software simulation to reflect the hardware changes is being developed. We are exploring the possibility of loading potential networks in the evolutionary optimization population onto an FPGA, running the simulation there (at a tremendous speed-up over the software simulation) and using the results from

the simulation to continue the optimization process. The computationally heavy component of the evolutionary optimization is the simulation itself, so it is possible that evolutionary optimization may be able to take place on a small on-chip processor (such as an ARM processor). An ASIC implementation of the VLSI design is also planned for the near future. We are also working on a mapping procedure between NIDA networks and DANNA networks so that existing NIDA networks can be implemented in DANNA networks.

7

Future Directions

"Going outside is highly overrated."

– Ernest Cline, *Ready Player One*

We have introduced a neuroscience-inspired dynamic architecture (NIDA) and its associated design method based on evolutionary optimization. We have explored extensions to this architecture based on the recognition and reuse of useful substructures and on affective systems in biology. We have also introduced an implementation of NIDA networks suitable for hardware, Dynamic Adaptive Neural Network Arrays (DANNA). There are many future directions for each of these topics, some of which we describe in this chapter. We also discuss the potential for a real-time learning method for NIDA networks using evolutionary optimization.

7.1 Architecture and Method

NIDA networks were designed to be simple but flexible. NIDA neurons have very few parameters: position, threshold and refractory period. Both the positions of

neurons and thresholds of neurons are changeable parameters in many of the tests given in this work. However, refractory period changes have not yet been explored. Refractory periods were imposed on the neurons to maintain an upper limit on the amount of possible activity in the networks. It may be worth exploring how altering this refractory period (while maintaining some refractory period in order to bound activity) during training affects performance of the training method. Our neurons also maintain accumulated charge over the course of the simulation, until either the neuron fires or the simulation ends. It may be worthwhile to explore a more biologically realistic neuron in which neurons slowly return to a neutral state (i.e., "leaky" neurons).

Synapses in this design have delays entirely based on distance between neurons. This is not biologically realistic, as synapses (or more accurately axons and dendrites, along with synapses) often have myelination, which can speed up the propagation of charge along the synapses. It may be worthwhile to include a concept of myelination in the networks to allow for changes in propagation speed in the network. This could be implemented by imposing another parameter on each synapse, delay. This delay could maintain a relationship with the distance of the synapse, or it could be entirely independent of the distance of the synapse. It also provides another adaptation mechanism. We did not include this concept in the original NIDA networks because we wanted two networks that are similar structurally to have similar behavior. By allowing for arbitrary delays on synapses, we lose this link between structure and functionality. It may be beneficial instead to include special types of synapses between neurons that are far apart allowing for rapid transfer of information between those neurons. These types of pathways have been observed in the brain. In this method, most synapses in the network maintain the relationship between delay and distance, but the special types of synapses allow for additional flexibility. This could be generalized to allow for multiple classes of synapses having different propagation velocities.

It may be worthwhile to explore different implementations of weight-change learning mechanisms, in addition to our simple implementation based on long-term potentiation (LTP) and long-term depression (LTD). For example, we may include more biologically realistic LTP/LTD, or we may explore alternative implementations, such as the spike-time dependent plasticity and other forms of Hebbian learning.

There are several extensions that may be made to the design method as well. We have explored a limited number of mutation operations and crossover types. There are certainly a wider variety that could be added to extend the functionality of the evolutionary optimization. For example, we have implemented but not explored random crossover, in which each neuron of each of the parent networks is randomly selected to appear in one of the two child networks and synapses are created between neurons as they are created in plane crossover (the crossover described in Section 3.2). An example mutation that could be included is neuron migration. In this mutation, an existing neuron in the network would be moved to a new location but maintain all of its connectivity with other neurons in the network, thus altering the set of delays to and from that neuron.

A mutation type that we did not explore was pruning. We implemented structural pruning of the network in which all "dead cells" (hidden neurons without any incoming connections or without any outgoing connections) and their existing connections were pruned out. Including dead cell pruning as a mutation may be used to limit the complexity of the generated network. However, it may also hinder the development of more complex structures that may be required for some problems. We may also consider implementing pruning based on activity in the network by pruning out neurons and synapses that are not very active or inactive over the course of simulation (potentially a much more drastic pruning than simple structural pruning). As discussed in Section 2.1 simpler structures may generalize better than more complex structures, so pruning (either over the course of simulation or after training takes place) may improve performance of the network.

We have used a simple version of tournament selection as the selection method for evolutionary optimization because of its flexibility and ease of implementation. Other possible selection methods may be used, including fitness proportion, rank, kernel, and roulette wheel selection [138]. Any of these selection methods may significantly change the performance of the evolutionary optimization method.

Three implementations of the design method have been described: a sequential implementation, a parallel implementation making use of one network per process and a parallel implementation in which a mini-population is maintained by each process. Parallel genetic algorithms have also been widely studied, so there are a variety of approaches that may be suitable for this implementation as well [251]. For example, maintaining mini-populations but passing along the best networks from each population to other mini-populations (or allowing for "drift" between mini-populations) is another example parallel implementation for evolutionary optimization that may be worth exploring.

There may be a faster training mechanism for NIDA networks (analogous to back-propagation or based on steepest descent approaches) that could be used to refine the parameters of a network once a good structure is discovered. Many approaches discussed in Section 2.1 use a hybrid learning approach, where structural learning occurs at a slower rate than weight training. We have proposed a tiered training system using simulated annealing as a training mechanism, but there may be a better, non-randomized optimization method to use for these networks in order to improve convergence rates.

We have used a direct representation of the network in implementing evolutionary optimization. There are many ways that a NIDA network may be represented directly (e.g., as a vector of real-values) or indirectly (e.g., as a genome that "unfolds" into a network). It may be prudent to explore alternative representations of the network and how these different representations alter the performance of the training method. For example NEAT ([245, 246]), HyperNEAT [244] and other methods discussed in Section 2.1 use indirect representations of network structures. These network

structures were not spiking neural networks, which have different characteristics than traditional neural network structures, and though HyperNEAT included a notion of geometry, it placed neurons in the space based on functions. Both of these characteristics of NIDA networks (dynamics and geometric space) would have to be accommodated when developing an indirect representation.

We have used fixed crossover rates and mutation rates in our training methods. It may be prudent to change mutation and crossover rates based on various metrics over the course of training. For example, we may want to selectively mutate neurons and synapses based on how active (or inactive) they are in simulation, or we may want to make crossover and mutations less likely as performance improves.

We have explored a small selection of tasks in three domains: classification, control, and anomaly detection. There are many other tasks to which NIDA networks may be applicable, including audio processing, signal processing, biological simulations (such as a C. elegans simulation [187]), synchronization of processes, system control, and many more. We believe that we have only scratched the surface of possible applications of this tool. One of the tasks worth exploring is using a NIDA network to realize an associative memory. In this task, a set of patterns would be presented to the network, and the network would be trained to recognize similar patterns or to produce whole patterns from partial patterns.

We presented a fitness function in Section 3.3.3 that may be applicable to a large set of classification problems. In particular, we introduced a fitness function based on entropy that could be used to extract features of a problem space. We propose using a similar fitness function to build nodes in a decision tree. That is, this fitness function could be used to split a set of inputs into two (or more) groups and to train NIDA networks on each new group as needed, until each group consists of only one classification type, as shown in Figure 7.1.

One major question regarding biological neural systems is how they are able to learn so quickly. We speculate one reason is that structures were evolved in the brain that are able to learn or adapt quickly to new tasks through Hebbian learning-type

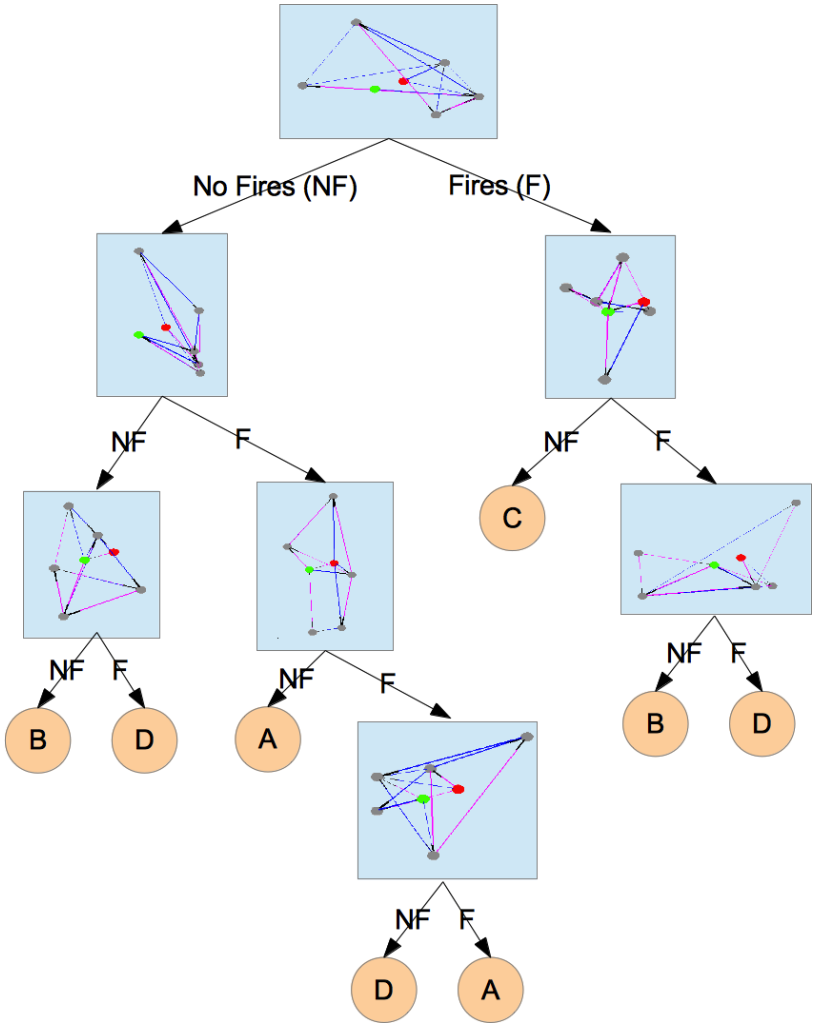


Figure 7.1: NIDAs can be used to make decisions at each node of a decision tree. For example, the decision can be based on whether the output neuron of the network fires in a particular time window. Based on the decision, a new NIDA network can be simulated on the input to further classify it.

mechanisms. One possible task for our networks is to develop network structures that can learn quickly using on-line learning mechanisms such as Hebbian learning or spike-time dependent plasticity in order to learn to do a set of tasks. That is, the goal would not be to develop one network to solve one task, but to develop a network structure with weights that can be adjusted using one of these learning mechanisms to solve one of a set of tasks.

We have explored some theoretical implications of NIDA networks and our design method in Appendix E. There has been significant theory developed on spiking neural networks and their properties. It may be worthwhile to further develop theoretical results that might later be exploited to improve the architecture and/or the design method.

There is significant potential for developing a user-friendly environment with which to design tasks for NIDA networks to solve, run existing NIDA networks on tasks and varying inputs, and explore the activity of NIDA networks. A visual analytics tool is in the process of being developed [70]. This tool includes the ability to visualize activity in a network, as well as to find and trace causality paths in the network (see Section 4.1.3).

In the future, we envision a tool in which a user can select from various problems (e.g., the benchmark problems described in Section 3.6), create jobs associated with those problems to be run on whatever resources are available (for example, within a cluster), and view the results of those job runs (Figure 7.2). Examples of jobs may be instances of the evolutionary optimization on that task, or they may be analysis jobs associated with that task, such as structural or activity analysis to extract a useful substructure.

Overviews of potential future directions discussed in this section are given for the architecture in Figure 7.3, for the design method in Figure 7.4, and for tasks and usability in Figure 7.5.

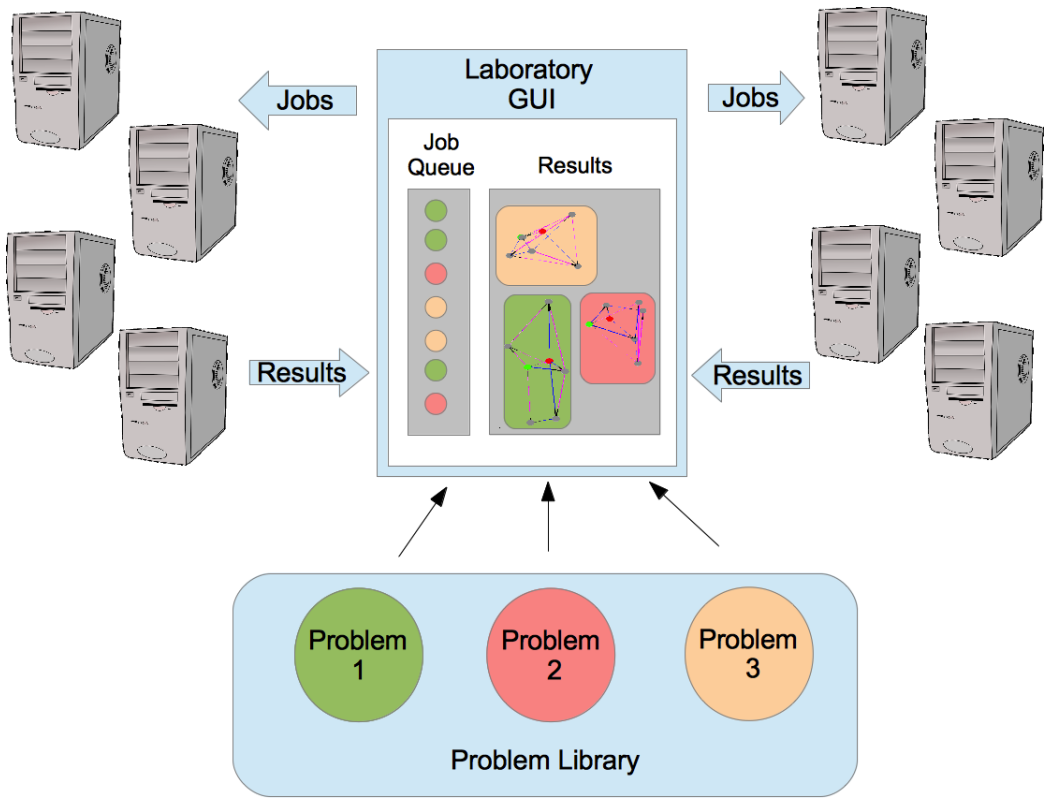


Figure 7.2: A GUI can be useful in which the user may select from various problems/tasks, which will create and distribute jobs to any available resources (e.g., nodes in a cluster), gather results, and display results in a meaningful manner. The current visual analytics tool would be included as a way to analyze networks and to display results meaningfully.

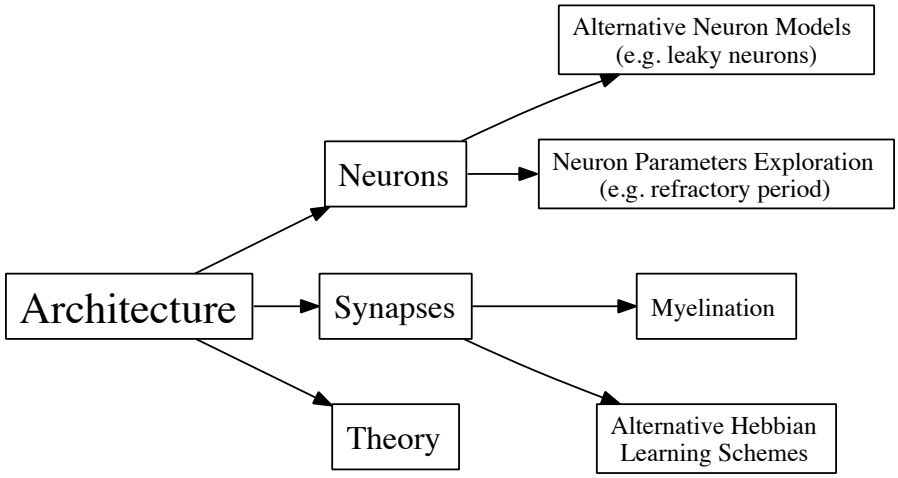


Figure 7.3: Graphic overviewing potential future directions for neuroscience-inspired dynamic architectures (NIDA).

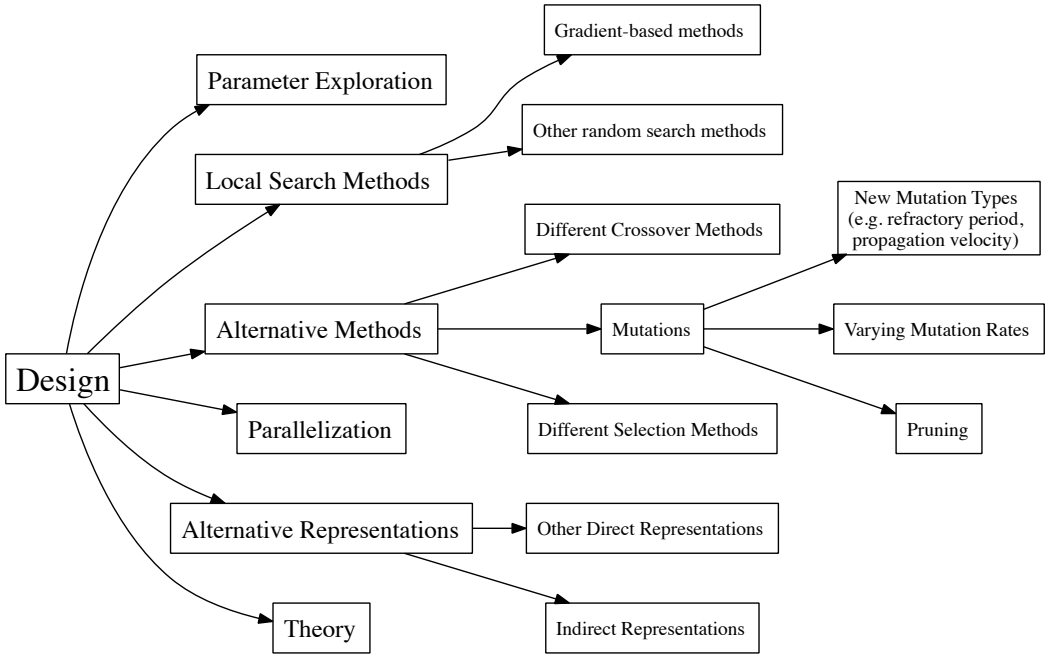


Figure 7.4: Graphic overviewing potential future directions for the NIDA design method.

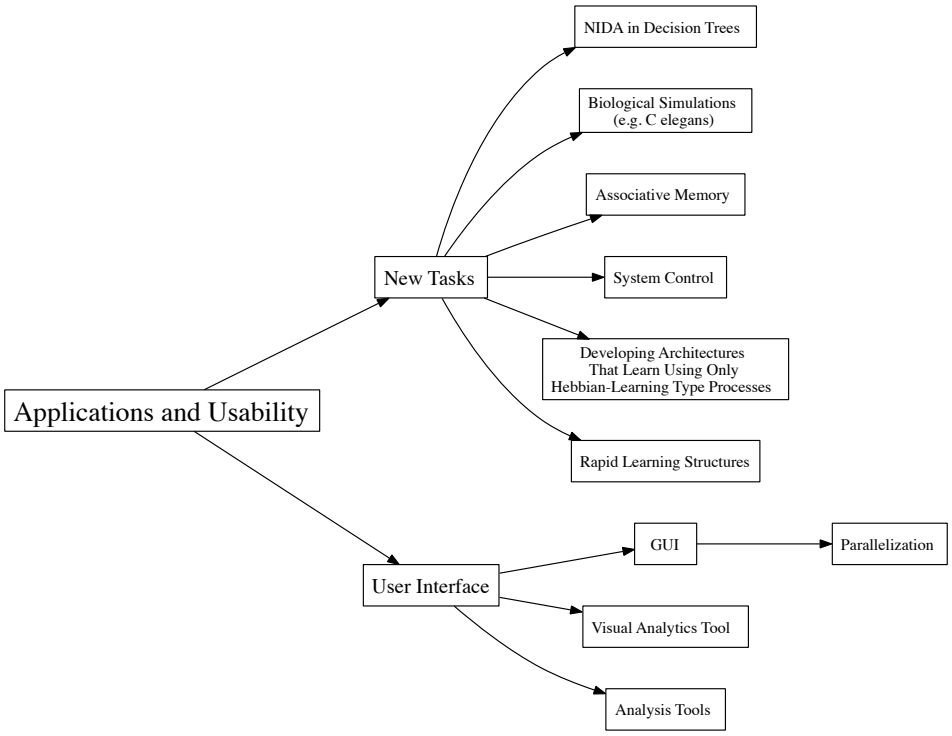


Figure 7.5: Graphic overviewing potential future directions for developing tasks and interaction tools for NIDA networks and the associated design method.

7.2 Useful Substructures

In this work, we have presented three mechanisms for identifying useful substructures. Those mechanisms are: looking for common substructures across many networks that solve the same task, looking for structures within a single network based on activity, and tracing causality paths through the network. We have also created networks for simpler tasks that can be combined and used in a more complicated task. Finally, we have extracted activity-based substructures from networks during training and incorporated those structures into the networks, with middling results. We have only explored the inclusion of activity-based structures over the course of training. It may be beneficial to extract and include causality-based structures over the course of training as well.

There are potentially many more "useful substructure" types that exist in these networks that we have not yet recognized. The common substructure analysis that we have performed is fairly primitive; more graph analysis tools could be used to identify important structures in a network. It would be useful to develop a canonical network representation that abstracts out position of nodes in the network while maintaining labels as to which neurons correspond to which input and output information in order to compare structure of the network.

We have only scratched the surface of what is possible for activity analysis. In particular, we have only looked at active structures within a single network and extracted substructures that exhibit the most activity. One possible way to compare networks is to look at their activity (rather than their structure). For example, a similarity metric could be developed in which networks that perform similarly (or produce the same result, if not the same firing patterns) for a set of input values are defined as similar. It is difficult to compare firing patterns directly between networks because two sets of firing patterns may produce the same behavior, depending on delays in the network. We propose developing a graph of activity in the network and using that graph to compare activity across networks. However, this has the

same pitfall as comparing networks directly, in that graph analysis quickly becomes intractable as many graph analysis techniques are NP-complete or NP-hard problems.

Once a set of tools has been established to extract subnetworks from network structures, we may maintain an external library of useful substructures. Currently, a library is maintained over the course of a single learning instance; thus, it is "internal" to a particular task and particular learning instance. We believe it would be useful to maintain a library external to a task and an instance. That is, the external library would include substructures from not only many instances of a single task (e.g., many different pole balancing networks) but many tasks as well (e.g., subnetworks from anomaly detection and handwritten digit recognition). As we noted in Section 2.4, substructures in the brain are often useful for more than one task. We speculate that our substructures may also be useful for more than one task.

This external library may be maintained within a database system, in which information about the structure of that subnetwork may be stored. We may also track further information about each structure in the network, such as an overall score for how that network performs within various tasks or specific scores for each task (for example, a specific score for how helpful that subnetwork was for the task solved by the network from which it was extracted). These scores may be used to decide which substructures should be selected for inclusion during learning in another task.

It may be prudent to conduct an analysis of structures over the course of learning in a separate thread or process from the optimization algorithm. For example, we may want to analyze the activity of the best network in the population (as we did in Section 4.2). This was done serially in the implementation described in that section. It would be useful to develop a parallel version of the optimization algorithm that performs this analysis separately from the optimization, so that the optimization is not slowed. Using this method, more complex analyses can be performed and utilized to improve the training or learning rates, without incurring as much of a penalty in training time.

We explored building a structure to solve a small problem and repeating that structure to tackle a larger problem in our network structure in Section 4.2. It may also be useful to design an existing structure for one problem and tweak that structure or weights in that structure for a similar, though slightly different problem. For example, one task might be recognizing piano notes. The structure to recognize one note (e.g., F) may be very similar to the structure required to recognize another note (e.g., F#), but the distances or delays may need to be scaled. This general idea has wider applications as well, such as frequency recognition, edge detection, and character recognition.

It would probably be useful to investigate the development of a library of components that are analogous to components needed in signal processing, communication systems or controls. Examples include oscillators, band pass and band stop filters, feedback regulators, and pattern generators. Such a library can be utilized to select possibly useful subnetworks as discussed above using evolutionary optimization. One could use the library to handcraft solutions to particular applications.

An overview of the potential future directions for exploring useful substructures in NIDA networks is given in Figure 7.6.

7.3 Affective Systems and Multiple Interacting Networks

We have introduced a simple affective system that alters the thresholds of neurons in the computational network in order to stabilize firing rate, and a more complex affective system that uses small-scale subnetworks embedded in the network to adjust thresholds, charge, or weights of synapses on neurons and synapses that exist in a small neighborhood of that subnetwork or that are connected to that subnetwork. We have also described how the optimization method can be altered to accommodate

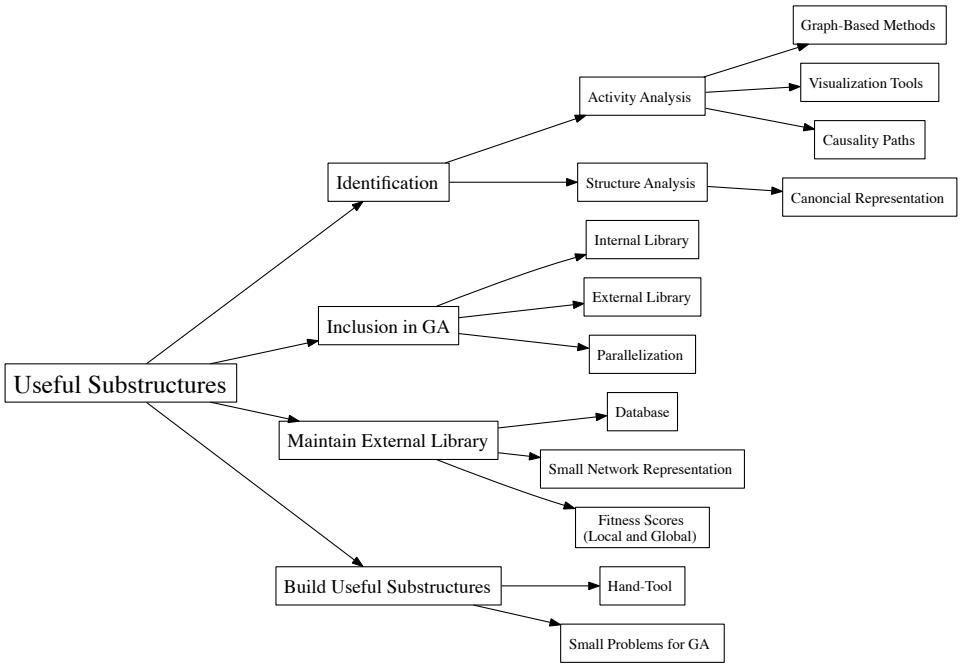


Figure 7.6: Graphic overviewing potential future directions for exploring useful substructures in NIDA networks.

the simple affective system and a more complex affective system, as well as how these alterations also accommodate training of multiple interacting networks.

Subnetworks in computational networks currently implement a "dispersal" mechanism that causes a change in charge or thresholds on nearby neurons or a change in weight values on nearby synapses. These changes currently occur instantaneously with the firing of the output neurons within the subnetwork. More complicated dispersal could be implemented in which the changes in charge, threshold or weight value are scheduled events in the event simulation and occur at some point in the future after the output neuron has fired (perhaps relative to the distance from the subnetwork). Taking a cue from biological systems, this mechanism can also modify localized energy available to neurons, modulating or suppressing the firing rates of neurons in a localized neighborhood of the network.

An alternative multi-network description may be defined so that multiple network structures that are represented in the same space overlap and interact but evolve separately. This is similar to the notion of subsystems in the network, but instead of the subnetworks occupying a smaller space than the original network, all of the networks would exist on the same scale, but each network would have a different goal that may conflict with the goals of other networks. A similar optimization method to the one presented in Section 5.2.2 could be implemented to accommodate this change.

There are many multi-objective optimization algorithms that might be worth exploring for this task. However, in contrast to these methods, we are simultaneously evolving multiple interacting networks, each of which has its own objective and for which there may be an overall objective of the system of networks. Moreover, we are optimizing over the connections between these networks. In many examples in which these would be useful, these networks may not operate in isolation; that is, they must exist as part of a system or team of networks in order to be evaluated. For example, consider an exploration/foraging task in which the agent must explore an environment to find food but while also avoiding predators. In this example, the agent may be composed of a computational network that receives sensory information

and outputs an action for the agent, as well as subnetworks that are trained to be "seeking" and "fear" networks, which encourage exploration for food and avoidance of predators, respectively. A seeking subnetwork cannot be evaluated alone on its task; it must be embedded in an agent in order to be evaluated. That is, we cannot simulate the seeking network in isolation and assign it a score; we must see how the seeking network influences the larger network to behave in the environment and assign it a score based on the agent's behavior. This is much harder to tackle, and we do not have conclusive evidence that the algorithm presented here effectively addresses this problem. Therefore, there is ample opportunity for further work on developing optimization algorithms that work in this realm.

In Section 5.4, we discussed how the inclusion of affective systems within a computational network naturally extends to a team of interacting networks that are trained to work together on a task, but in which different team members have different objectives. In this case, there may not be direct connections (synapses) between subnetworks (in this case, between interacting networks), but it may be that these networks can be simulated simultaneously so that they may communicate in some other way. For example, suppose we want to evolve a team of simulated ant-like agents, in which the role of some agents is to explore new environments and find food (type A agents), while the role of other agents (type B) is to discover when a type A agent has found food and to collect food from that food source until it is exhausted. In this situation, it would be possible to evolve type A agents without the type B agents, but it might not be possible to evolve type B agents without type A agents because type B agents rely on communications from type A agents. We may attempt to simulate how a type A agent will behave when training type B agents, but there is no guarantee that the evolved type A agents will behave in this way. By evolving two sets of agents simultaneously, we can evolve agents that learn to work together. We may also simultaneously evolve a team of agents in which each agent is composed of multiple networks, where the agents may all have the same (or different) goals and where the subnetworks per agent may each have their own goal.

There is ample opportunity to further explore the effects of simple affective systems and complex affective systems on the dynamics and behavior of the network, as we did in Section 5.1.2. For example, we did not examine how the complex affective systems affect LTP and LTD and how those parameters interact. More studies on the general properties of these affective systems would be worthwhile so that we may better understand their utility in the scope of NIDA networks.

We initially proposed the inclusion of the affective networks because we felt that they were an important part of biological neural systems that were, for the most part, not considered in artificial neural network systems. It is clear that one potential role of affective systems is for modulation of behavior of a network based on sensory information (as also noted by Krichmar in [140]). By evolving affective systems alongside network structures, we allow for a possibly different set of potentially important input information to be processed in an alternate framework (using different objectives) and to affect the behavior of an agent. Again, consider an agent in an environment seeking food and avoiding predators. By including notions of affective systems, a fear subnetwork may be trained to modulate behavior only when a predator is near, while a seeking subnetwork may be trained to modulate behavior only when the agent is "hungry." These affective controls allow for an on-line mechanism to change behavior based on environmental cues that may not be directly related to the task.

An overview of the potential future directions for exploring affective networks in NIDA networks and training and using multiple interacting NIDA networks is given in Figure 7.7.

7.4 Hardware Implementations

We have presented a preliminary hardware implementation that has been designed in VHDL and implemented on an FPGA. An accompanying simulation has been developed, though it is primitive and was developed only as a testing mechanism to

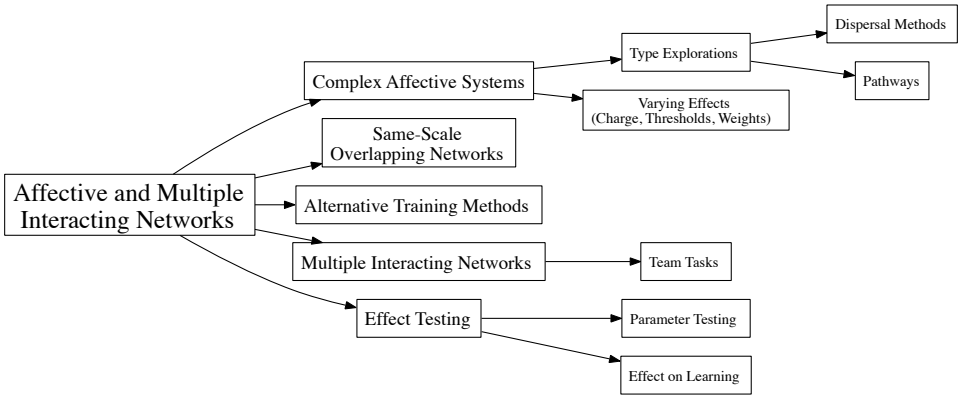


Figure 7.7: Graphic overview of potential future directions for affective networks and multiple-interacting NIDA networks.

compare against the results obtained on the hardware. There are many improvements and extensions that could be made to this implementation.

An updated simulation that makes use of graphics processing units (GPUs) could significantly speed up performance. This type of simulation is suited to GPUs because of its synchronous nature and the division of computation into elements within a grid that only communicate with their neighbors (or with elements relatively near to them). A GPU implementation would allow many (if not all) elements to be updated in parallel, avoiding serial updates of elements. We may also simulate the network on the FPGA or custom chip.

The software simulation could also be significantly improved by developing an event-driven version, rather than a clocked version. In the current implementation, the state of every element is checked and updated at every element-clock cycle, which requires a significant amount of computation, even if there is no activity on that element. A better version would be similar to our NIDA simulation, which is event-driven. In this simulation, events would be scheduled in an event queue, and the state of the system would be updated only when an event occurs. It may also be possible to restrict state updates to a subset of the array at each event, rather than the entire array, improving performance of the simulation. Many of the hardware

implementations discussed in Section 2.6 have developed event-driven (or partially event-driven) simulations because of the slow nature of clock-driven simulations.

The design method currently relies on the old simulation code and is thus very slow. By replacing the simulation code with a new, faster simulation, whether it is the GPU or event-driven simulation or running directly on the hardware, the performance of the design method will improve. As discussed in Section 7.1 with respect to the evolutionary optimization for NIDA networks, there are many facets of evolutionary optimization for the hardware platform that may be explored, such as selection methods, alternative reproduction operators, and parallelization methods.

The current design method was implemented to work with the 8-connection element. For a newer 16-connection element (shown in an array in Figure 7.8), more structural alterations may be included, as the 16-connection element allows for more complex structures without fear of blocking. The 16-connection element scheme can be extended arbitrarily to allow for more and more distant connectivity among elements, though there is a penalty incurred for additional complexity within each element. Since the ability to include major structural flexibility is one of the notable features of our NIDA networks, we want to allow for as much structural flexibility within our hardware networks as well, while maintaining a reasonable sized element.

There is potential for implementing the design method on a small processor (such as an ARM processor) that does all simulations on hardware (networks on an FPGA or on a custom chip). The design method requires the storage of a population of networks (both parents and children), as well as scores for those networks. The design method may use a very simple algorithm to select parents to produce children and to create children using simple reproduction methods, all of which may be implemented on the ARM processor. Depending on the size of the networks and the hardware (GPU, FPGA, or ASIC/custom VLSI), it may be possible to simulate multiple networks at once, allowing for parallelization of the design method. An issue with implementing

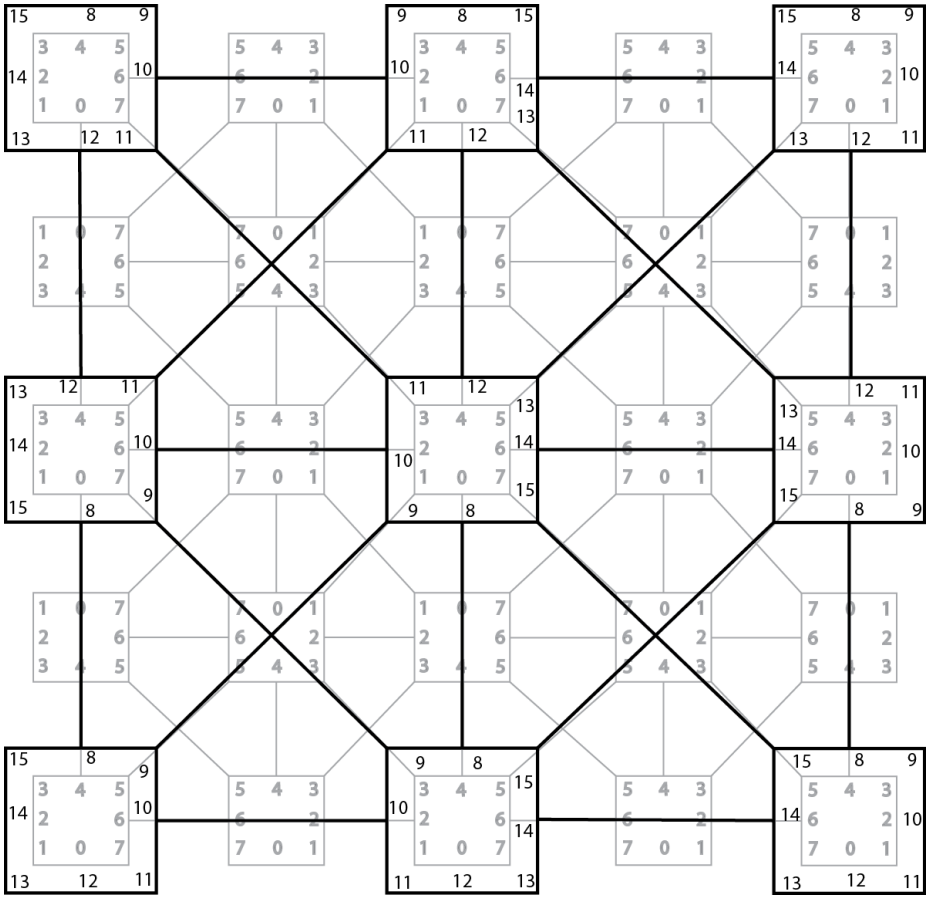


Figure 7.8: New element connectivity so that each element is connected to 16 other elements in its neighborhood. Four (offset) copies of the overlay (shown in bold) can be utilized to provide complete coverage of the network's elements. Alternatively, only a subset of the elements may implement larger-distance connectivity. The structure can be repeated with greater spacing between connected elements (and more copies for complete coverage) to provide even longer distance direct connections. (Figure created by Dr. Doug Birdwell and included with his permission.)

the design method on an ARM or other small processor is the implementation of task-specific fitness functions. Some fitness functions may require off-chip interactions (for example, interacting with a real cart and pole for the pole balancing problem) or may require large scale simulations or large amounts of input/output information (such as in the handwritten digit problem). One approach is to use additional hardware components, or a combination of hardware and software, to implement either a simulation/emulation of a system or process interfaced to the network, or to provide such an interface to a physical system or external process. This issue will have to be addressed when implementing the design method on alternative processors, whether small or not, to gain increased learning/adaptation performance.

In addition to developing a design method specifically for DANNAs, it may be useful to develop a mapping mechanism that converts NIDA networks directly into DANNA networks and to use the DANNA design method to refine parameter values to produce desired operations. Direct mappings have been attempted by hand for small networks (e.g., 5-10 neurons and 10-30 synapses), but this mapping process is far too complicated to be done by hand for larger networks. It may be possible to adapt existing mapping software (e.g., circuit board layout software) to perform the mappings between NIDA and DANNA. Otherwise, a stochastic optimization approach, such as a genetic/evolutionary method or simulated annealing can be utilized. This problem is similar to circuit and component layout problems in electronics and digital system (e.g., VLSI) design.

As discussed, the VHDL code for an array element has been designed and implemented on an FPGA. Arrays of 16 by 16 elements and 50 by 50 elements have been constructed and tested on the FPGA. There are plans to test larger arrays on FPGAs, while simultaneously developing a VLSI implementation that can be used to create a custom chip for our arrays.

One feature that we are considering including in our arrays are "rich-club" or hub neurons, which are neurons that are highly connected locally and have long-range connections to other rich-club or hub neurons in the network. Rich-club neurons

have been observed in biological neural networks [107]. In our networks, these would be developed as elements that have long-range connections to other elements in the network (rather than just near-neighbor or local connectivity). There is also biological evidence that there is significant feedback to sensory neurons, or other neurons near input processes, from the equivalent of "hidden neurons" in the brain [6]. It may be useful to include pathways from many elements (if not all elements) in the grid to the input neurons to mimic these feedback patterns. In all cases, the evolutionary optimization method can be adapted to utilize the features and restrictions of a particular hardware or combination hardware/software implementation of an array.

As in the software implementation, there is significant room to develop a user-friendly development and testing environment. Ideally, this environment would be implemented as a GUI that provides the user with easy interface to both hardware and software simulations of the network, analysis and visualization tools, and debugging tools. The goal would be to develop an environment that allows users to load a network (either to simulation, to an FPGA, or to a custom chip), and allows the user to provide some form of input to that network (e.g., input from a camera, audio device, or other type of sensor), and to output information in a form that the user selects for some type of output device (e.g., motors or actuators in a robot). This interface between the GUI and some form of the network should be abstracted so that it is capable of connecting to a software simulation (which may be done on a GPU), to an FPGA or to a custom chip, and able to run seamlessly, treating each of these as if they are identical except for differences in size and features as discussed above.

An overview of the potential future directions for developing a hardware implementation is given in Figure 7.9.

7.5 Real-Time Learning

One potential future direction of this work is to develop software that will simultaneously produce output for a real-time system, while also continuing to learn. For

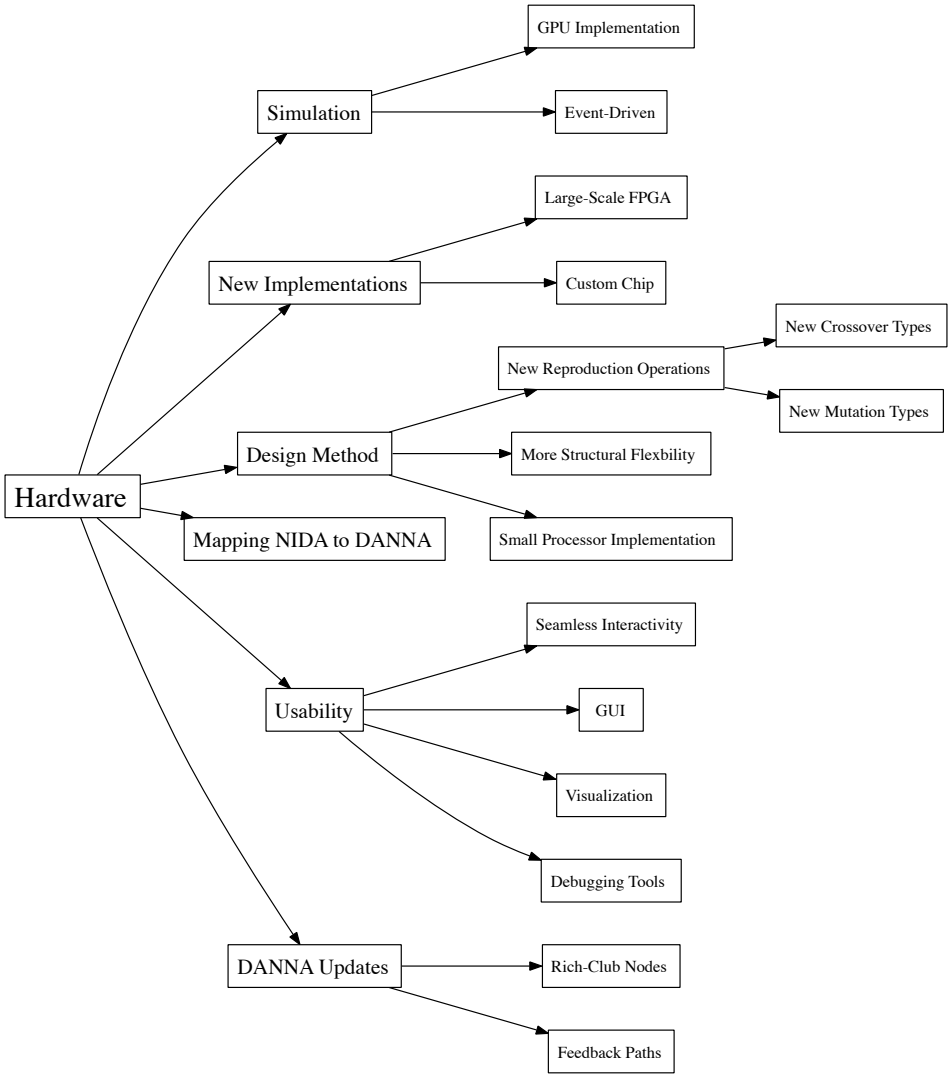


Figure 7.9: Graphic overviewing potential future directions related to the hardware implementation of NIDA networks, DANNA.

this type of system, we may allow an initial off-line learning process until a predefined performance threshold is reached. Then, learning may be converted to on-line, so that the system is producing usable output while adapting to changes in its environment. This is a biological model in that humans are continually learning and improving beginning as infants and accomplish tasks on a day-to-day basis beginning in childhood and continuing throughout their lives.

To accomplish this on-line learning, one potential approach is to split an evolutionary algorithm population pool into two teams: a production team and a development team (Figure 7.10). The production team, for example, may be made up of networks that perform well in the population. The output of the system will be an aggregate, selection, or weighted combination of the outputs of the networks in the production team. The development team will be lower performing networks that are maintained as part of the population for diversity. The entire population will be iterated over, except the production team will stay mostly in place (with fewer changes to the members), while the development team will change much more rapidly. Members will be promoted to the production team from the development team depending on their performance. Similarly, members of the production team may be demoted to the development team or die off from the population entirely depending on their performance.

The intent of this type of organization is that the production networks keep the overall system working regardless of the learning that is happening in the development networks. With this system, safeguards would be required to keep the production team's performance above a certain threshold. However, this threshold may not be explicitly given. For this type of system, an approximate value function, such as that used in reinforcement learning and approximate dynamic programming, may be useful.

Using this system, the overall performance of the algorithm should continue to improve. Also, using this method, the system should be adaptable to changes in the task at hand, if the task is changing over time. There are several issues associated

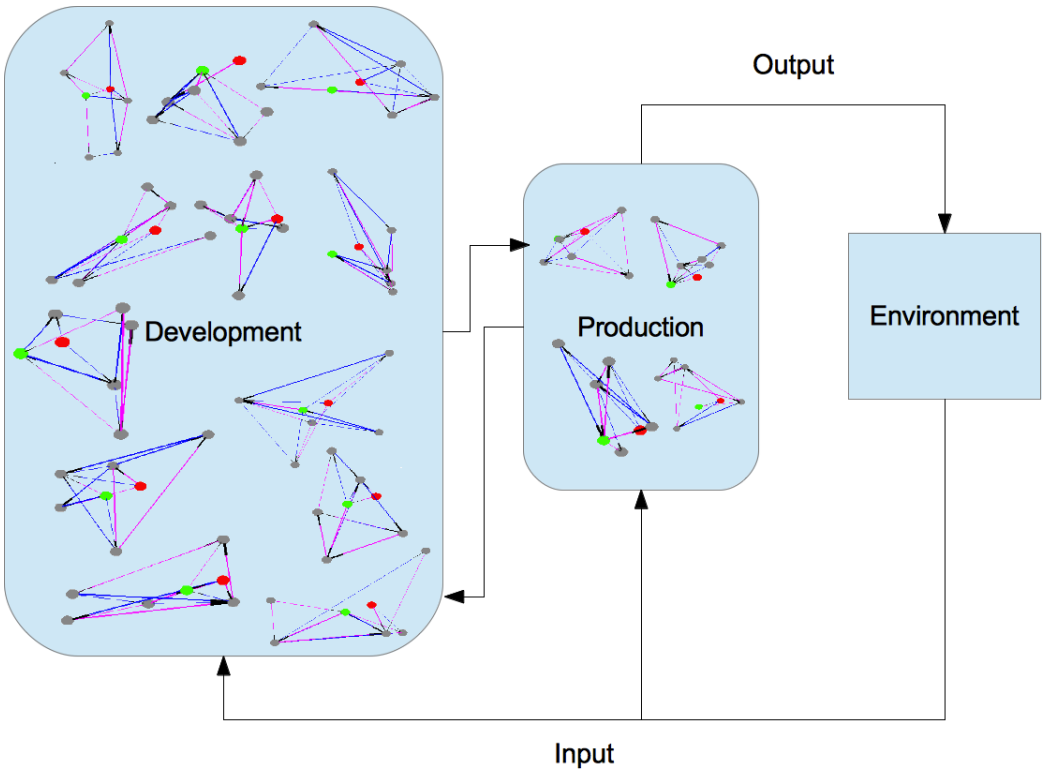


Figure 7.10: General real time learning scheme.

with this approach that must be addressed. One issue is how to score the networks that are part of the development population. The output of these networks is never reaching the environment and thus they have no effect on the performance of the task. Similarly, it may be difficult to score individual production-level networks as well, if there are many production networks and the result presented to the environment is some aggregate of the output calculated by the networks.

One approach is to maintain a model of the environment that is being updated based on the environment's behavior and to use the model of the environment to calculate the scores of the networks. This approach is similar to model predictive control or model reference adaptive control schemes reported in the systems and control literature. The concept is to preferentially utilize models that accurately predict an observed process's behavior to decide what control action or decision to apply. A typical method is to monitor an error process (or innovations process) as

a measure of a system's ability to surprise a model-based prediction of its behavior [8, 16, 43, 89, 94, 175, 178, 181]. Although the model may be out-of-date with the current environment, it may still give some information about how the networks perform. It may also be very difficult to maintain an accurate (or even semi-accurate) model of a complex environment.

7.6 Summary

This work has the potential to generate a large number of research projects and has already motivated the creation of several, including the hardware implementation presented in [66] and the visual analytics tool presented in [70].

This chapter presents a host of other potential research topics within this field. One goal of this work is to serve as a starting point for these future research projects, meant to introduce the architecture and method and demonstrate its utility. The immediate priorities for the future fall into two categories: relatively easy extensions and difficult, exploratory work that has the potential for high payoff.

The relatively easy extensions that could significantly improve performance and usability are:

1. Graphical user interface (GUI) for ease of use, facilitating ease of creation of new applications and testing/analyzing existing applications.

2. Improved parallelization techniques that can work across multiple machine types, including shared memory systems.

3. Causality path recognition and reuse integration into design method. This is one potential existing substructure type we have recognized that may significantly improve training performance.

4. Running DANNA simulations during training on the FPGA (and/or on GPUs), in order to speed up training.

The more difficult, exploratory work includes:

1. Local search methods (perhaps based on gradient methods) for parameter searches (for parameters such as weights of synapses and thresholds of neurons). This could facilitate faster training on a structure that is determined by the evolutionary optimization design method.

2. New affective system implementations. We have explored embedded sub-networks as complex affective systems, but we have also proposed several alternative implementations.

3. New affective system/multiple interacting network training methods.

4. Custom chip implementations of DANNA networks. These chips may offer considerable performance increases and operate at lower power costs.

8

Conclusion

"What is the point of being alive if you don't at least try to do something remarkable?"

– John Green, *An Abundance of Katherines*

We gave an overview of the current landscape in neuroscience and computer science in Chapter 1. We noted that neuroscience is perhaps the "last frontier" of science, and that neuroscience results have inspired computer scientists to explore novel neural network architectures and develop neuromorphic hardware to combat our growing computational needs and the end of Moore's Law.

Chapter 2 provides an overview of some of the previous work that is related to or that has inspired this work. This previous work includes evolutionary optimization methods for designing neural networks, neural networks embedded in a three-dimensional space, continuous time recurrent neural networks, and spiking neural networks. We also briefly discuss work related to the identification and reuse of useful substructures in the literature, which includes works on modular neural networks and neural networks that exploit hierarchy and regularity. A brief overview of artificial

affective systems in machine learning is also included, along with an overview of some of the major neuromorphic hardware efforts.

We introduced our neuroscience-inspired dynamic architecture (NIDA) in Chapter 3. This architecture takes inspiration from both existing neural network architectures and from neuroscience. It exhibits dynamic behaviors and is embedded in geometric space. We presented a design method based on evolutionary optimization and genetic algorithms. This design method provides the flexibility to work with a variety of applications/tasks and can be expanded to work with alternative network architectures. We demonstrated the flexibility of the architecture and design method by demonstrating their utility for tasks in anomaly detection, control, and classification. We showed that we can produce results competitive with traditional architectures on three classic benchmark problems.

In Chapter 4 we explored the recognition and reuse of useful substructures. We noted that the recognition of these structures enhances our understanding of these architectures and provides a potentially valuable tool for exploring the behavior of these network types. We also explored the reuse of useful substructures to aid in improving evolutionary optimization. Though these results were not conclusive, we believe that there is potential for further work in identifying and reusing subnetworks in order to improve learning.

We explored the use of several artificial affective systems in Chapter 5. We explored both simple and complex affective systems, and we proposed updates to the optimization method in order to accommodate the inclusion of these affective systems. We tested the effects of affective system inclusion on the pole balancing problem. For both simple and complex affective systems, we found that the inclusion of these affective systems in training improved the performance of the networks by a small amount in testing, but overall, the results were mostly inconclusive. However, we did observe some interesting characteristics of the behavior of the simple affective system with a single network. We believe it is worthwhile to further explore both simple and complex affective systems with NIDA networks.

Chapter 6 provides an overview of a neuromorphic hardware implementation based on NIDA networks, Dynamic Adaptive Neural Network Arrays (DANNA). We discuss preliminary implementations of simulation and design software for DANNAs. These results are presented primarily from the software perspective. However, the DANNA project has become highly collaborative, in which a team of electrical engineers, computer engineers, and computer scientists continue to develop better implementations.

The work in Chapters 3 through 6 is only the tip of an iceberg of potential future research on this topic. Some of the future directions for this topic are detailed in Chapter 7. We cannot conclusively say that our method is better than other proposed neural network or neuromorphic architectures, but we strongly believe that this work provides justification that both NIDA and DANNA are worth further exploration. This work provides a foundation upon which to build these future research efforts.

"Somewhere, something incredible is waiting to be known."

– Carl Sagan

It is clear that a major sea change in computer science and computer engineering is imminent. With the decline of Moore's Law, a new computing paradigm is required to serve as a complement to traditional von Neumann architectures. We are gathering data in larger volumes than ever before. Not only is there a tremendous amount of data that, ideally, would be processed in real-time, but there are also a wide variety of data types: text, audio, images, video and so on. Much of the data are easily processed by humans, but not as easily processed by existing paradigms.

Neuroscience-inspired architectures and methods are one approach to technologies that can more easily process these types of data. Moreover, these architectures often come hand-in-hand with neuromorphic hardware implementations, one proposed alternative/complement to traditional von Neumann architectures. Thus, by exploring these types of architectures we may be able to simultaneously deal with processing

difficult data types and address existing hardware limitations. The neuroscience-inspired dynamic architecture presented in this work is one step in this direction.

This work falls squarely in the realm of exploratory research. It is clear that there are a tremendous number of different types of neural network architectures, design methods and neuromorphic hardware implementations. It is entirely possible that this architecture, design method, and hardware implementation will not result in any groundbreaking achievements or be competitive with other, similar methods as they continue to emerge. We believe that this work has, at the very least, demonstrated that NIDA and DANNA are not a dead end; that is, we believe that we have demonstrated that this approach is worth further exploration. It is impossible to know which outcome will occur without at least trying.

> *"If we knew what it was we were doing, it would not be called research, would it?"*
>
> – Albert Einstein

Bibliography

[1] Hussein A. Abbass. An evolutionary artificial neural networks approach for breast cancer diagnosis. *Artificial Intelligence in Medicine*, 25(3):265–281, 2002. 17, 22, 27, 145, 146

[2] Hussein A. Abbass, Michael Towsey, and G. Finn. C-Net: A method for generating non-deterministic and dynamic multivariate decision trees. *Knowledge and Information Systems*, 3(2):184–197, 2001. 146

[3] David H. Ackley, Geoffrey E. Hinton, and Terrence J. Sejnowski. A learning algorithm for Boltzmann machines. *Cognitive Science*, 9(1):147 – 169, 1985. 350

[4] Hyungil Ahn and Rosalind Picard. Affective-cognitive learning and decision making: A motivational reward framework for affective agents. In Jianhua Tao, Tieniu Tan, and Rosalind Picard, editors, *Affective Computing and Intelligent Interaction*, volume 3784 of *Lecture Notes in Computer Science*, pages 866–873. Springer Berlin / Heidelberg, 2005. 48

[5] Enrique Alba and J. Chicano. Training neural networks with GA hybrid algorithms. In Kalyanmoy Deb, editor, *Genetic and Evolutionary Computation GECCO 2004*, volume 3102 of *Lecture Notes in Computer Science*, pages 852–863. Springer Berlin / Heidelberg, 2004. 23, 27, 145, 146, 147

[6] Henry J. Alitto and W. Martin Usrey. Corticothalamic feedback and sensory processing. *Current Opinion in Neurobiology*, 13(4):440 – 445, 2003. 283

[7] R. Ananthanarayanan, S.K. Esser, H.D. Simon, and D.S. Modha. The cat is out of the bag: cortical simulations with 109 neurons, 1013 synapses. In *High Performance Computing Networking, Storage and Analysis, Proceedings of the Conference on*, pages 1–12, Nov 2009. 55

[8] Brian Anderson, Thomas Brinsmead, Daniel Liberzon, and A. Stephen Morse. Multiple model adaptive control with safe switching. *International Journal of Adaptive Control and Signal Processing*, 15(5):445–470, 2001. 287

[9] C.W. Anderson. Learning to control an inverted pendulum using neural networks. *Control Systems Magazine, IEEE*, 9(3):31 –37, April 1989. 114, 115

[10] Michael L. Anderson and Marcie Penner-Wilger. Neural reuse in the evolution and development of the brain: Evidence for developmental homology? *Developmental Psychobiology*, 55(1):42–51, 2013. 40, 137, 161

[11] P.J. Angeline, G.M. Saunders, and J.B. Pollack. An evolutionary algorithm that constructs recurrent neural networks. *Neural Networks, IEEE Transactions on*, 5(1):54 –65, January 1994. 17, 20, 27, 41

[12] David W. Arathorn. *Map-seeking circuits in visual cognition: A computational mechanism for biological and machine vision*. Stanford University Press, 2002. 28

[13] Alain Artola and Wolf Singer. Long-term depression of excitatory synaptic transmission and its relationship to long-term potentiation. *Trends in Neurosciences*, 16(11):480 – 487, 1993. 354

[14] Derrik E. Asher, Jeffrey L. Krichmar, and Nicolas Oros. Evolution of biologically plausible neural networks performing a visually guided reaching task. In *Proceedings of Genetic and Evolutionary Computation Conference*. ACM, 2014. 37

[15] M. Athans, R. Ku, and S. Gershwin. The uncertainty threshold principle: Some fundamental limitations of optimal decision making under dynamic uncertainty. *Automatic Control, IEEE Transactions on*, 22(3):491–495, 1977. 361

[16] Michael Athans, David Castanon, K.P. Dunn, C. Greene, Wing Lee, N. Sandell Jr., and Alan S. Willsky. The stochastic control of the F-8C aircraft using a multiple model adaptive control (MMAC) method–part i: Equilibrium flight. *Automatic Control, IEEE Transactions on*, 22(5):768–780, 1977. 287

[17] K. Bache and M. Lichman. UCI machine learning repository, 2013. 144, 148, 152, 155

[18] Arunadevi Baladhandapani and Deepa Subramaniam Nachimuthu. Evolutionary learning of spiking neural networks towards quantification of 3D MRI brain tumor tissues. *Soft Computing*, pages 1–14, 2014. 36

[19] Sajal Chandra Banik, Keigo Watanabe, Maki K. Habib, and Kiyotaka Izumi. Affection based multi-robot team work. In S.C. Mukhopadhyay and R.Y.M. Huang, editors, *Sensors*, volume 21 of *Lecture Notes in Electrical Engineering*, pages 355–375. Springer Berlin Heidelberg, 2008. 51

[20] Andrew G. Barto, Satinder Singh, and Nuttapong Chentanez. Intrinsically motivated learning of hierarchical collections of skills. In *Proceedings of the 3rd international conference on development and learning (ICDL 2004)*, pages 112–119, 2004. 47

[21] R. Batllori, C.B. Laramee, W. Land, and J.D. Schaffer. Evolving spiking neural networks for robot control. *Procedia Computer Science*, 6(0):329 – 334, 2011. Complex adaptive sysytems. 35

[22] Randall D. Beer. On the dynamics of small continuous-time recurrent neural networks. *Adaptive Behavior*, 3(4):469–509, 1995. 31

[23] Randall D. Beer. The dynamics of adaptive behavior: A research program. *Robotics and Autonomous Systems*, 20(2-4):257 – 289, 1997. Practice and Future of Autonomous Agents. 31

[24] Randall D. Beer. The dynamics of active categorical perception in an evolved model agent. *Adaptive Behavior*, 11(4):209–243, 2003. 31

[25] Randall D. Beer and John C. Gallagher. Evolving dynamical neural networks for adaptive behavior. *Adapt. Behav.*, 1(1):91–122, June 1992. 14, 27

[26] Ammar Belatreche, Liam P Maguire, and Martin McGinnity. Advances in design and application of spiking neural networks. *Soft Computing*, 11(3):239–248, 2007. 34, 146

[27] Y. Bengio, P. Simard, and P. Frasconi. Learning long-term dependencies with gradient descent is difficult. *Neural Networks, IEEE Transactions on*, 5(2):157–166, 1994. 14, 347

[28] Yoshua Bengio. Learning deep architectures for AI. *Found. Trends Mach. Learn.*, 2(1):1–127, January 2009. 3, 28, 349

[29] B.V. Benjamin, Peiran Gao, E. McQuinn, S. Choudhary, A.R. Chandrasekaran, J.M. Bussat, R. Alvarez-Icaza, J.V. Arthur, P.A Merolla, and K. Boahen. Neurogrid: A mixed-analog-digital multichip system for large-scale neural simulations. *Proceedings of the IEEE*, 102(5):699–716, May 2014. 60

[30] James Bergstra and Yoshua Bengio. Random search for hyper-parameter optimization. *The Journal of Machine Learning Research*, 13(1):281–305, 2012. 16

[31] Dimitri P. Bertsekas. *Nonlinear Programming*. Athena Scientific, Belmont, Massachusetts, second edition, 1999. 13

[32] Dimitri P. Bertsekas. *Dynamic Programming and Optimal Control*. Athena Scientific, Belmont, Massachusetts, second edition, 2000. 362, 363

[33] Dimitri P. Bertsekas and John N. Tsitsiklis. *Neuro-Dynamic Programming*. Athena Scientific, Belmont, Massachusetts, 1996. 363

[34] Michael Beyeler, Nikil D Dutt, and Jeffrey L Krichmar. Categorization and decision-making in a neurobiologically plausible spiking network using a STDP-like learning rule. *Neural Networks*, 48:109–124, 2013. 137

[35] J. Douglas Birdwell, Mark E. Dean, Margaret Drouhard, and Catherine Schuman. Method and apparatus for constructing a neuroscience-inspired artificial neural network with visualization of neural pathways. U.S. Patent Application 14/513,497, filed October 14, 2014. 179

[36] J. Douglas Birdwell, Mark E. Dean, and Catherine Schuman. Method and apparatus for constructing a dynamic adaptive neural network array (DANNA). U.S. Patent Application 14/513,297, filed October 14, 2014. 250

[37] J. Douglas Birdwell, Mark E. Dean, and Catherine Schuman. Method and apparatus for constructing, using and reusing components and structures of an artificial neural network. U.S. Patent Application 14/513,388, filed October 14, 2014. 162

[38] J. Douglas Birdwell, Mark E. Dean, and Catherine Schuman. Method and apparatus for providing random selection and long-term potentiation and depression in an artificial network. U.S. Patent Application 14/513,334, filed October 14, 2014. 250, 253

[39] J. Douglas Birdwell, Mark E. Dean, and Catherine Schuman. Method and apparatus for providing real-time monitoring of an artificial neural network. U.S. Patent Application 14/513,447, filed October 14, 2014. 250

[40] J. Douglas Birdwell and Catherine Schuman. Method and apparatus for constructing a neuroscience-inspired artificial neural network. U.S. Patent Application 14/513,280, filed October 14, 2014. 66

[41] Sander M. Bohte, Joost N. Kok, and Han La Poutr. Error-backpropagation in temporally encoded networks of spiking neurons. *Neurocomputing*, 48(14):17 – 37, 2002. 33, 146

[42] Olaf Booij and Hieu tat Nguyen. A gradient descent rule for spiking neurons emitting multiple spikes. *Information Processing Letters*, 95(6):552 – 558, 2005. Applications of Spiking Neural Networks. 33

[43] Stephen P. Boyd, Venkataramanan Balakrishnan, Craig H. Barratt, Nasser M. Khraishi, Xiaoming Li, David G. Meyer, and Stephen A. Norman. A new CAD method and associated architectures for linear controllers. *Automatic Control, IEEE Transactions on*, 33(3):268–283, 1988. 287

[44] Jürgen Branke. Evolutionary algorithms in neural network design and training – A review. In Jarmo T. Alander, editor, *Proc. of the First Nordic Workshop on Genetic Algorithms and their Applications (1NWGA)*, volume 95-1, pages 145–163, Vaasa, Finnland, 1995. 17

[45] Leo Breiman, Jerome Friedman, Charles J. Stone, and Richard A. Olshen. *Classification and regression trees*. CRC press, 1984. 138

[46] Per Brodal. *The Central Nervous System: Structure and Function*. Oxford University Press, fourth edition, 2010. 331, 334, 335, 336, 337, 338

[47] Daniel Brüderle, Mihai A. Petrovici, Bernhard Vogginger, Matthias Ehrlich, Thomas Pfeil, Sebastian Millner, Andreas Grübl, Karsten Wendt, Eric Müller, Marc-Olivier Schwartz, DanHusmann de Oliveira, Sebastian Jeltsch, Johannes Fieres, Moritz Schilling, Paul Müller, Oliver Breitwieser, Venelin Petkov, Lyle Muller, AndrewP. Davison, Pradeep Krishnamurthy, Jens Kremkow, Mikael Lundqvist, Eilif Muller, Johannes Partzsch, Stefan Scholze, Lukas Zühl, Christian Mayr, Alain Destexhe, Markus Diesmann, TobiasC. Potjans, Anders Lansner, René Schüffny, Johannes Schemmel, and Karlheinz Meier.

A comprehensive workflow for general-purpose neural modeling with highly configurable neuromorphic hardware systems. *Biological cybernetics*, 104(4-5):263–296, 2011. 59

[48] John Byrne. Cellular analysis of associative learning. *Physiol. Rev.*, 67(2):329–439, 1987. 354

[49] T. Caelli, Ling Guan, and W. Wen. Modularity in neural computing. *Proceedings of the IEEE*, 87(9):1497 –1518, September 1999. 39, 40

[50] Mario D. Capuozzo and David L. Livingston. A compact evolutionary algorithm for integer spiking neural network robot controllers. In *Southeastcon, 2011 Proceedings of IEEE*, pages 237–242. IEEE, 2011. 35

[51] K.D. Carlson, M. Beyeler, N. Dutt, and J.L. Krichmar. GPGPU accelerated simulation and parameter tuning for neuromorphic applications. In *Proceedings of the 19th Asia and South Pacific Design Automation Conference (ASP-DAC'14)*, 2014. 37

[52] Kristofor D. Carlson, Jayram Moorkanikara Nageswaran, Nikil Dutt, and Jeffrey L. Krichmar. An efficient automated parameter tuning framework for spiking neural networks. *Frontiers in neuroscience*, 8, 2014. 37

[53] Kristofor D. Carlson, Micah Richert, Nikil Dutt, and Jeffrey L. Krichmar. Biologically plausible models of homeostasis and STDP: Stability and learning in spiking neural networks. In *Neural Networks (IJCNN), The 2013 International Joint Conference on*, pages 1–8. IEEE, 2013. 36

[54] Dmitri B. Chklovskii and Alexei A. Koulakov. Maps in the brain: What can we learn from them? *Annual Review of Neuroscience*, 27(1):369–392, 2004. 6, 28

[55] Sung-Bae Cho. Combining modular neural networks developed by evolutionary algorithm. In *Evolutionary Computation, 1997., IEEE International Conference on*, pages 647 –650, April 1997. 40

[56] Sung-Bae Cho and K. Shimohara. Modular neural networks evolved by genetic programming. In *Evolutionary Computation, 1996., Proceedings of IEEE International Conference on*, pages 681 –684, May 1996. 40

[57] Sung-Bae Cho and Katsunori Shimohara. Evolutionary learning of modular neural networks with genetic programming. *Applied Intelligence*, 9(3):191–200, 1998. 40

[58] Swadesh Choudhary, Steven Sloan, Sam Fok, Alexander Neckar, Eric Trautmann, Peiran Gao, Terry Stewart, Chris Eliasmith, and Kwabena Boahen. Silicon neurons that compute. In *Artificial Neural Networks and Machine Learning–ICANN 2012*, pages 121–128. Springer, 2012. 60

[59] B Cox and Jeffrey L Krichmar. Neuromodulation as a robot controller. *Robotics & Automation Magazine, IEEE*, 16(3):72–80, 2009. 52

[60] Özgür Şimşek and Andrew G. Barto. An intrinsic reward mechanism for efficient exploration. In *Proceedings of the 23rd international conference on Machine learning*, ICML '06, pages 833–840, New York, NY, USA, 2006. ACM. 49

[61] Yann Le Cun, John S. Denker, and Sara A. Solla. Optimal brain damage. In *Advances in Neural Information Processing Systems*, pages 598–605. Morgan Kaufmann, 1990. 17

[62] James Daly, Jacob Brown, and Juyang Weng. Neuromorphic motivated systems. In *Neural Networks (IJCNN), The 2011 International Joint Conference on*, pages 2917–2924. IEEE, 2011. 52

[63] Antonio Damasio. *Descartes' Error: Emotions, Reason, and the Human Brain*. Penguin Books, 1994. 8, 44, 202

[64] D. Dasgupta and D.R. McGregor. Designing application-specific neural networks using the structured genetic algorithm. In *Combinations of Genetic Algorithms and Neural Networks, 1992., COGANN-92. International Workshop on*, pages 87 –96, June 1992. 19, 27

[65] Peter Dayan and L.F. Abbott. *Theoretical Neuroscience: Computational and Mathematical Modeling of Neural Systems*. The MIT Press, Cambridge, Massachusetts, 2001. 68, 334, 338, 339

[66] Mark E. Dean, Catherine D. Schuman, and J. Douglas Birdwell. Dynamic adaptive neural network array. In Oscar H. Ibarra, Lila Kari, and Steffen Kopecki, editors, *Unconventional Computation and Natural Computation*, Lecture Notes in Computer Science, pages 129–141. Springer International Publishing, 2014. 250, 287

[67] S. Dominic, R. Das, D. Whitley, and C. Anderson. Genetic reinforcement learning for neural networks. In *Neural Networks, 1991., IJCNN-91-Seattle International Joint Conference on*, volume 2, pages 71 –76, July 1991. 15, 27

[68] John Doyle. Analysis of feedback systems with structured uncertainties. *IEE Proceedings D (Control Theory and Applications)*, 129(6):242–250, 1982. 31

[69] Bogdan Draganski, Christian Gaser, Volker Busch, Gerhard Schuierer, Ulrich Bogdahn, and Arne May. Neuroplasticity: Changes in grey matter induced by training. *Nature*, 427(6972):311–312, January 2004. 339

[70] Margaret Drouhard, Catherine D. Schuman, J. Douglas Birdwell, and Mark E. Dean. Visual analytics for neuroscience-inspired dynamic architectures. *IEEE Symposium Series on Computational Intelligence*, 2014. In press. 179, 181, 268, 287

[71] P. Durr, C. Mattiussi, and D. Floreano. Genetic representation and evolvability of modular neural controllers. *Computational Intelligence Magazine, IEEE*, 5(3):10–19, Aug 2010. 44

[72] Matthias Ehrlich, Christian Mayr, Holger Eisenreich, Stephan Henker, Andre Srowig, Andreas Grübl, Johannes Schemmel, and René Schüffny. Wafer-scale vlsi implementations of pulse coupled neural networks. In *Proceedings of the International Conference on Sensors, Circuits and Instrumentation Systems*, 2007. 58

[73] Howard Eichenbaum. Hippocampus: Cognitive processes and neural representations that underlie declarative memory. *Neuron*, 44(1):109 – 120, 2004. 338

[74] Scott E. Fahlman and Christian Lebiere. The cascade-correlation learning architecture. In David S. Touretzky, editor, *Advances in neural information processing systems 2*, pages 524–532. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1990. 17

[75] J.E. Fieldsend and S. Singh. Pareto evolutionary neural networks. *Neural Networks, IEEE Transactions on*, 16(2):338 – 354, March 2005. 23, 27

[76] Gary William Flake. *The Computational Beauty of Nature: Computer Explorations of Fractals, Chaos, Complex Systems, and Adaptation*. MIT Press, 1998. 354, 356

[77] Dario Floreano, Peter Dürr, and Claudio Mattiussi. Neuroevolution: from architectures to learning. *Evolutionary Intelligence*, 1(1):47–62, 2008. 17

[78] Dario Floreano, Yann Epars, Jean-Christophe Zufferey, and Claudio Mattiussi. Evolution of spiking neural circuits in autonomous mobile robots. *International Journal of Intelligent Systems*, 21(9):1005–1024, 2006. 33

[79] Dario Floreano and Claudio Mattiussi. Evolution of spiking neural controllers for autonomous vision-based robots. In *Evolutionary Robotics. From Intelligent Robotics to Artificial Life*, pages 38–61. Springer, 2001. 33

[80] Razvan V Florian. Correct equations for the dynamics of the cart-pole system. *Center for Cognitive and Neural Studies (Coneural), Romania*, 2007. 114

[81] D. Fogel, L. Fogel, and V. Porto. Evolving neural networks. *Biological Cybernetics*, 63(6):487–493, 1990. 14, 27, 145

[82] David B. Fogel, Eugene C. Wasson III, and Edward M. Boughton. Evolving neural networks for detecting breast cancer. *Cancer letters*, 96(1):49–53, 1995. 145

[83] Frauke Friedrichs and Christian Igel. Evolutionary tuning of multiple SVM parameters. *Neurocomputing*, 64(0):107 – 117, 2005. Trends in Neurocomputing: 12th European Symposium on Artificial Neural Networks. 16

[84] Kenichi Funahashi and Yuichi Nakamura. Approximation of dynamical systems by continuous time recurrent neural networks. *Neural Networks*, 6(6):801 – 806, 1993. 31

[85] S.B. Furber, D.R. Lester, L.A Plana, J.D. Garside, E. Painkras, S. Temple, and AD. Brown. Overview of the SpiNNaker system architecture. *Computers, IEEE Transactions on*, 62(12):2454–2467, Dec 2013. 63

[86] Steve Furber and Steve Temple. Neural systems engineering. *Journal of the Royal Society interface*, 4(13):193–206, 2007. 61

[87] John C. Gallagher and Saranyan Vigraham. A modified compact genetic algorithm for the intrinsic evolution of continuous time recurrent neural networks. In *Proceedings of the Genetic and Evolutionary Computation Conference*, GECCO '02, pages 163–170, San Francisco, CA, USA, 2002. Morgan Kaufmann Publishers Inc. 31

[88] Peiran Gao, B.V. Benjamin, and K. Boahen. Dynamical system guided mapping of quantitative neuronal models onto neuromorphic hardware. *Circuits and Systems I: Regular Papers, IEEE Transactions on*, 59(10):2383–2394, Oct 2012. 60

[89] Carlos E. Garcia and A.M. Morshedi. Quadratic programming solution of dynamic matrix control (QDMC). *Chemical Engineering Communications*, 46(1-3):73–87, 1986. 287

[90] N. Garcia-Pedrajas, C. Hervas-Martinez, and J. Munoz-Perez. COVNET: a cooperative coevolutionary model for evolving artificial neural networks. *Neural Networks, IEEE Transactions on*, 14(3):575 – 596, may 2003. 41

[91] N. Garcia-Pedrajas, C. Hervas-Martinez, and D. Ortiz-Boyer. Cooperative coevolution of artificial neural network ensembles for pattern classification. *Evolutionary Computation, IEEE Transactions on*, 9(3):271 – 302, June 2005. 41

[92] Nicolás García-Pedrajas, Domingo Ortiz-Boyer, and César Hervás-Martínez. An alternative approach for neural network evolution with a genetic algorithm: Crossover by combinatorial optimization. *Neural Networks*, 19(4):514 – 528, 2006. 24, 27, 145

[93] C. Gaser and G. Schlaug. Brain structures differ between musicians and non-musicians. *The Journal of neuroscience: the official journal of the Society for Neuroscience*, 23(27):9240, 2003. 339

[94] Gangadhar Gattu and Evanghelos Zafiriou. Nonlinear quadratic dynamic matrix control with state estimation. *Industrial & engineering chemistry research*, 31(4):1096–1104, 1992. 287

[95] Jason Gauci and Kenneth O. Stanley. Autonomous evolution of topographic regularities in artificial neural networks. *Neural Computation*, 22(7):1860–1898, 2010. 28, 29

[96] David C. Geary. *The Origin of Mind: Evolution of Brain, Cognition, and General Intelligence*. American Psychological Association, 2005. 353

[97] Samanwoy Ghosh-Dastidar and Hojjat Adeli. A new supervised learning algorithm for multiple spiking neural networks with application in epilepsy and seizure detection. *Neural Networks*, 22(10):1419 – 1431, 2009. 33

[98] Samanwoy Ghosh-Dastidar and Hojjat Adeli. Spiking neural networks. *International Journal of Neural Systems*, 19(04):295–308, 2009. 33

[99] Samanwoy Ghosh-Dastidar and Hojjat Adeli. Third generation neural networks: Spiking neural networks. In *Advances in Computational Intelligence*, pages 167–178. Springer, 2009. 3

[100] David E. Goldberg. *Genetic Algorithms in Search, Optimization, and Machine Learning*. Addison-Wesley Publishing Company, Inc., 1989. 355, 356

[101] Faustino Gomez and Risto Miikkulainen. 2-d pole balancing with recurrent evolutionary networks. In *in Proceeding of the International Conference on Artificial Neural Networks (ICANN)*, pages 425–430, 1998. 21, 27

[102] Faustino Gomez, Jürgen Schmidhuber, and Risto Miikkulainen. Efficient non-linear control through neuroevolution. In Johannes Fürnkranz, Tobias Scheffer, and Myra Spiliopoulou, editors, *Machine Learning: ECML 2006*, volume 4212 of *Lecture Notes in Computer Science*, pages 654–662. Springer Berlin / Heidelberg, 2006. 15, 27

[103] Faustino Gomez, Jürgen Schmidhuber, and Risto Miikkulainen. Accelerated neural evolution through cooperatively coevolved synapses. *J. Mach. Learn. Res.*, 9:937–965, June 2008. 15, 27

[104] Daniel Graupe. *Principles of artificial neural networks*. World Scientific, 2nd edition, 2007. 343

[105] H. Hagras, A Pounds-Cornish, M. Colley, V. Callaghan, and G. Clarke. Evolving spiking neural network controllers for autonomous robots. In *Robotics and Automation, 2004. Proceedings. ICRA '04. 2004 IEEE International Conference on*, volume 5, pages 4620–4626 Vol.5, April 2004. 33

[106] Tim Harford. *Adapt: Why success always starts with failure*. Macmillan, 2011. 7

[107] Logan Harriger, Martijn P. van den Heuvel, and Olaf Sporns. Rich club organization of macaque cerebral cortex and its role in network communication. *PLoS ONE*, 7(9):e46497, 09 2012. 6, 283

[108] Leland H. Hartwell, John J. Hopfield, Stanislas Leibler, and Andrew W. Murray. From molecular to modular cell biology. *Nature*, 402(6761):C47–C52, December 1999. 39, 161

[109] M. Hausknecht, J. Lehman, R. Miikkulainen, and P. Stone. A neuroevolution approach to general Atari game playing. *Computational Intelligence and AI in Games, IEEE Transactions on*, PP(99):1–1, 2014. 25

[110] Jeff Hawkins and Dileep George. Hierarchical temporal memory: Concepts, theory and terminology. *Whitepaper, Numenta Inc*, 2006. 29

[111] Etay Hay, Sean Hill, Felix Schürmann, Henry Markram, and Idan Segev. Models of neocortical layer 5b pyramidal cells capturing a wide range of dendritic and perisomatic active properties. *PLoS computational biology*, 7(7):e1002107, 2011. 32

[112] Simon Haykin. *Neural Networks: A Comprehensive Foundation*. Macmillan College Publishing Company, 1994. 341, 348, 349, 350

[113] D.O. Hebb. *The Organization of Behavior: A Neuropsychological Theory*. John Wiley & Sons Inc., 1949. 5, 30, 338, 348, 349

[114] J.H. Holland. *Adaptation in Natural and Artificial Systems*. The University of Michigan Press, 1975. 359

[115] J.J. Hopfield. Neural networks and physical systems with emergent collective computational abilities. *Proceedings of the National Academy of Sciences*, 79(8):2554–2558, 1982. 350

[116] Gregory S. Hornby. Measuring, enabling and comparing modularity, regularity and hierarchy in evolutionary design. In *Proceedings of the 2005 conference on Genetic and evolutionary computation*, GECCO '05, pages 1729–1736, New York, NY, USA, 2005. ACM. 42

[117] C. Igel. Neuroevolution for reinforcement learning using evolution strategies. In *Evolutionary Computation, 2003. CEC '03. The 2003 Congress on*, volume 4, pages 2588 – 2595 Vol.4, December 2003. 14, 16, 27, 81

[118] Auke Jan Ijspeert. Central pattern generators for locomotion control in animals and robots: A review. *Neural Networks*, 21(4):642 – 653, 2008. Robotics and Neuroscience. 72

[119] Masao Ito. Long-term depression. *Ann. Rev. Neurosci.*, 12:85–102, 1989. 354

[120] K. Izumi, S.C. Banik, and K. Watanabe. Behavior generation in robots by emotional motivation. In *Industrial Electronics, 2009. ISIE 2009. IEEE International Symposium on*, pages 455 –460, July 2009. 51

[121] Bryan L. Jackson, Bipin Rajendran, Gregory S. Corrado, Matthew Breitwisch, Geoffrey W. Burr, Roger Cheek, Kailash Gopalakrishnan, Simone Raoux, Charles T. Rettner, Alvaro Padilla, Alex G. Schrott, Rohit S. Shenoy, Bülent N. Kurdi, Chung H. Lam, and Dharmendra S. Modha. Nanoscale electronic

synapses using phase change devices. *J. Emerg. Technol. Comput. Syst.*, 9(2):12:1–12:20, May 2013. 56

[122] Xin Jin, M. Luján, L.A Plana, S. Davies, S. Temple, and S.B. Furber. Modeling spiking neural networks on spinnaker. *Computing in Science Engineering*, 12(5):91–97, Sept 2010. 62

[123] Yaochu Jin, Ruojing Wen, and Bernhard Sendhoff. Evolutionary multi-objective optimization of spiking neural networks. In *Artificial Neural Networks–ICANN 2007*, pages 370–379. Springer, 2007. 34, 146

[124] Jae-Yoon Jung and J.A. Reggia. Evolutionary design of neural network architectures using a descriptive encoding language. *Evolutionary Computation, IEEE Transactions on*, 10(6):676 –688, dec. 2006. 42

[125] Jon H. Kaas. Topographic maps are fundamental to sensory processing. *Brain research bulletin*, 44(2):107–112, 1997. 6, 28

[126] Takumi Kamioka, Eiji Uchibe, and Kenji Doya. Neuroevolution based on reusable and hierarchical modular representation. In *Proc. 15th Int. Conf. Advances in Neuro-Information Processing, Part I (ICONIP'08)*, pages 22–31. Berlin, Heidelberg: Springer-Verlage, 2009. 43

[127] Eric Kandel, James Schwartz, and Thomas Jessell. *Principles of Neural Science*. McGraw-Hill Medical, 4 edition, January 2000. 28, 45

[128] Frédéric Kaplan and Pierre-Yves Oudeyer. Intrinsically motivated machines. In Max Lungarella, Rolf Pfeifer, Fumiya Iida, and Josh Bongard, editors, *50 years of artificial intelligence*, pages 303–314. Springer-Verlag, Berlin, Heidelberg, 2007. 49

[129] Nikola Kasabov. Evolving spiking neural networks and neurogenetic systems for spatio-and spectro-temporal data modelling and pattern recognition. In *Advances in Computational Intelligence*, pages 234–260. Springer, 2012. 33

[130] Nikola Kasabov, Valery Feigin, Zeng-Guang Hou, Yixiong Chen, Linda Liang, Rita Krishnamurthi, Muhaini Othman, and Priya Parmar. Evolving spiking neural networks for personalised modelling, classification and prediction of spatio-temporal patterns with a case study on stroke. *Neurocomputing*, 134(0):269 – 279, 2014. Special issue on the 2011 Sino-foreign-interchange Workshop on Intelligence Science and Intelligent Data Engineering (IScIDE 2011) Learning Algorithms and Applications Selected papers from the 19th International Conference on Neural Information Processing (ICONIP2012). 37

[131] Nadav Kashtan, Uri Alon, and Jr. Callan, Curtis G. Spontaneous evolution of modularity and network motifs. *Proceedings of the National Academy of Sciences of the United States of America*, 102(39):pp. 13773–13778, 2005. 7, 39, 161

[132] Y. Kassahun. *Towards a Unified Approach to Learning and Adapation*. PhD thesis, Christian-Albrechts-University, Kiel, Germany, February 2006. 24, 27

[133] Mitsuo Kawato. Internal models for motor control and trajectory planning. *Current Opinion in Neurobiology*, 9(6):718 – 727, 1999. 39

[134] Eric B. Keverne and James P. Curley. Epigenetics, brain evolution and behaviour. *Frontiers in Neuroendocrinology*, 29(3):398 – 412, 2008. 353

[135] M.M. Khan, D.R. Lester, L.A. Plana, A. Rast, X. Jin, E. Painkras, and S.B. Furber. Spinnaker: Mapping neural networks onto a massively-parallel chip multiprocessor. In *Neural Networks, 2008. IJCNN 2008. (IEEE World Congress on Computational Intelligence). IEEE International Joint Conference on*, pages 2849–2856, June 2008. 61

[136] S. Kirkpatrick, C. D. Gelatt, and M. P. Vecchi. Optimization by simulated annealing. *Science*, 220:671–680, 1983. 357, 359

[137] Harold Köndgen, Caroline Geisler, Stefano Fusi, Xiao-Jing Wang, Hans-Rudolf Lüscher, and Michele Giugliano. The dynamical response properties of neocortical neurons to temporally modulated noisy inputs in vitro. *Cerebral Cortex*, 18(9):2086–2097, 2008. 32

[138] John R. Koza. *Genetic Programming: On the Programming of Computers by Means of Natural Selection*. MIT Press, Cambridge, Massachusetts, 1992. 265, 356

[139] J.R. Koza and J.P. Rice. Genetic generation of both the weights and architecture for a neural network. In *Neural Networks, 1991., IJCNN-91-Seattle International Joint Conference on*, volume ii, pages 397 –404 vol.2, jul 1991. 19, 27

[140] Jeffrey L. Krichmar. The neuromodulatory system: A framework for survival and adaptive behavior in a challenging world. *Adaptive Behavior*, 16(6):385–399, 2008. 51, 278

[141] Jeffrey L Krichmar. A biologically inspired action selection algorithm based on principles of neuromodulation. In *Neural Networks (IJCNN), The 2012 International Joint Conference on*, pages 1–8, 2012. 52

[142] Yann Lecun and Corinna Cortes. The MNIST database of handwritten digits, 2009. xix, 126, 127, 135

[143] F.H.F. Leung, H.K. Lam, S.H. Ling, and P.K.S. Tam. Tuning of the structure and parameters of a neural network using an improved genetic algorithm. *Neural Networks, IEEE Transactions on*, 14(1):79 – 88, January 2003. 23, 27

[144] Chengzhi Li, Huaiyu Dai, and Husheng Li. Adaptive quickest change detection with unknown parameter. In *Acoustics, Speech and Signal Processing, 2009. ICASSP 2009. IEEE International Conference on*, pages 3241–3244, April 2009. 105

[145] Hod Lipson. Principles of modularity, regularity, and hierarchy for scalable systems. *The Journal of Biological Physics and Chemistry*, 7(4):125–128, 2007. 42, 161

[146] Yong Liu and Xin Yao. A population-based learning algorithm which learns both architectures and weights of neural networks. *Chinese Journal of Advanced Software Research (Allerton)*, 10011:54–65, 1996. 20, 27

[147] Hongtao Lu. On stability of nonlinear continuous-time neural networks with delays. *Neural Networks*, 13(10):1135 – 1143, 2000. 31, 32

[148] David Luenberger. Introduction to dynamic systems: theory, models, and applications, 1979. 31

[149] Wolfgang Maass. Networks of spiking neurons: The third generation of neural network models. *Neural Networks*, 10(9):1659 – 1671, 1997. 3, 32

[150] Wolfgang Maass and Thomas Natschläger. Networks of spiking neurons can emulate arbitrary hopfield nets in temporal coding. *Network: Computation in Neural Systems*, 8(4):355–371, 1997. 33

[151] Wolfgang Maass and Thomas Natschläger. Emulation of hopfield networks with spiking neurons in temporal coding. In *Computational Neuroscience*, pages 221–226. Springer, 1998. 33

[152] Shingo Mabu, Kotaro Hirasawa, and Jinglu Hu. A graph-based evolutionary algorithm: Genetic network programming (gnp) and its extension using reinforcement learning. *Evol. Comput.*, 15(3):369–398, September 2007. 43

[153] Eleanor A. Maguire, David G. Gadian, Ingrid S. Johnsrude, Catriona D. Good, John Ashburner, Richard S. J. Frackowiak, and Christopher D. Frith. Navigation-related structural change in the hippocampi of taxi drivers. *Proceedings of the National Academy of Sciences*, 97(8):4398–4403, April 2000. 339, 353

[154] M. Malfaz, A. Castro-Gonzalez, R. Barber, and M.A. Salichs. A biologically inspired architecture for an autonomous and social robot. *Autonomous Mental Development, IEEE Transactions on*, 3(3):232 –246, September 2011. 51

[155] Udayan Mallik, R. Jacob Vogelstein, Eugenio Culurciello, Ralph Etienne-Cummings, and Gert Cauwenberghs. A real-time spike-domain sensory information processing system [image processing applications]. In *Circuits and Systems, 2005. ISCAS 2005. IEEE International Symposium on*, pages 1919–1922, 2005. 58

[156] Olvi L. Mangasarian, R. Setiono, and W.H. Wolberg. Pattern recognition via linear programming: Theory and application to medical diagnosis. *Large-scale numerical optimization*, pages 22–31, 1990. 148

[157] V. Maniezzo. Genetic evolution of the topology and weight distribution of neural networks. *Neural Networks, IEEE Transactions on*, 5(1):39 –53, January 1994. 19, 27

[158] Eve Marder and Dirk Bucher. Central pattern generators and the control of rhythmic movements. *Current Biology*, 11(23):R986 – R996, 2001. 72

[159] Henry Markram. The blue brain project. In *Proceedings of the 2006 ACM/IEEE conference on Supercomputing*, SC '06. ACM, 2006. 32

[160] Henry Markram, Wulfram Gerstner, and Per Jesper Sjöström. A history of spike-timing-dependent plasticity. *Frontiers in Synaptic Neuroscience*, 3:4, 2011. 30

[161] Warren McCulloch and Walter Pitts. A logical calculus of the ideas immanent in nervous activity. *Bulletin of Mathematical Biology*, 5:115–133, 1943. 349

[162] S. McKennoch, Dingding Liu, and L.G. Bushnell. Fast modifications of the SpikeProp algorithm. In *Neural Networks, 2006. IJCNN '06. International Joint Conference on*, pages 3970–3977, 2006. 33

[163] P. Merolla, J. Arthur, F. Akopyan, N. Imam, R. Manohar, and D.S. Modha. A digital neurosynaptic core using embedded crossbar memory with 45pj per spike in 45nm. In *Custom Integrated Circuits Conference (CICC), 2011 IEEE*, pages 1–4, Sept. 2011. 55

[164] Paul A. Merolla, John V. Arthur, Rodrigo Alvarez-Icaza, Andrew S. Cassidy, Jun Sawada, Filipp Akopyan, Bryan L. Jackson, Nabil Imam, Chen Guo, Yutaka Nakamura, Bernard Brezzo, Ivan Vo, Steven K. Esser, Rathinakumar Appuswamy, Brian Taba, Arnon Amir, Myron D. Flickner, William P. Risk, Rajit Manohar, and Dharmendra S. Modha. A million spiking-neuron integrated circuit with a scalable communication network and interface. *Science*, 345(6197):668–673, 2014. 56

[165] Nicholas Metropolis, Arianna W. Rosenbluth, Marshall N. Rosenbluth, Augusta H. Teller, and Edward Teller. Equation of State Calculations by Fast Computing Machines. *The Journal of Chemical Physics*, 21(6):1087–1092, 1953. 358

[166] F. Michaud, E. Robichaud, and J. Audet. Using motives and artificial emotions for prolonged activity of a group of autonomous robots. *Emotional and Intelligent II: The Tangled Knot of Social Cognition-AAAI Fall Symposium, Technical Report FS-01-02*, pages 85–90, Nov. 2001. 50

[167] Donald Michie, David J. Spiegelhalter, and Charles C. Taylor. Machine learning, neural and statistical classification, 1994. 147, 155

[168] Sebastian Millner, Andreas Grübl, Karlheinz Meier, Johannes Schemmel, and Marc olivier Schwartz. A VLSI implementation of the adaptive exponential integrate-and-fire neuron model. In J.D. Lafferty, C.K.I. Williams, J. Shawe-Taylor, R.S. Zemel, and A. Culotta, editors, *Advances in Neural Information Processing Systems 23*, pages 1642–1650. Curran Associates, Inc., 2010. 59

[169] M.L. Minksky. *Theory of neural-analog reinforcement systems and its application to the brain-model problem.* PhD thesis, Princeton University, Princeton, NJ, 1954. 350

[170] Marvin Minsky and Seymour Papert. Perceptron (expanded edition), 1969. 350

[171] M.L. Minsky. Steps toward artificial intelligence. *Proceedings of the IRE*, 49(1):8 –30, January 1961. 350

[172] Melanie Mitchell. *An Introduction to Genetic Algorithms.* The MIT Press, Cambridge, Massachusetts, 1998. 354

[173] Don Monroe. Neuromorphic computing gets ready for the (really) big time. *Communications of the ACM*, 57(6):13–15, 2014. 5

[174] David J. Montana and Lawrence Davis. Training feedforward neural networks using genetic algorithms. In *Proceedings of the 11th international joint conference on Artificial intelligence - Volume 1*, pages 762–767, San Francisco, CA, USA, 1989. Morgan Kaufmann Publishers Inc. 14, 27

[175] Manfred Morari and Jay H. Lee. Model predictive control: past, present and future. *Computers & Chemical Engineering*, 23(4):667–682, 1999. 287

[176] David E. Moriarty and Risto Miikkulainen. Forming neural networks through efficient and adaptive coevolution. *Evol. Comput.*, 5(4):373–399, December 1997. 21, 27

[177] David E. Moriarty and Risto Mikkulainen. Efficient reinforcement learning through symbiotic evolution. *Machine Learning*, 22(1):11–32, 1996. 21, 27

[178] A.M. Morshedi, C.R. Cutler, and T.A. Skrovanek. Optimal solution of dynamic matrix control with linear programing techniques (LDMC). In *American Control Conference, 1985*, pages 199–208. IEEE, 1985. 287

[179] J.B. Mouret and S. Doncieux. Evolving modular neural-networks through exaptation. In *Evolutionary Computation, 2009. CEC '09. IEEE Congress on*, pages 1570 –1577, May 2009. 43

[180] Martin Fodslette Mller. A scaled conjugate gradient algorithm for fast supervised learning. *Neural Networks*, 6(4):525 – 533, 1993. 13

[181] Kumpati S. Narendra and Jeyendran Balakrishnan. Adaptive control using multiple models. *Automatic Control, IEEE Transactions on*, 42(2):171–187, 1997. 287

[182] Thomas Natschläger, Berthold Ruf, and Michael Schmitt. Unsupervised learning and self-organization in networks of spiking neurons. In *Self-Organizing neural networks*, pages 45–73. Springer, 2002. 33

[183] Javier Navaridas, Mikel Luján, Jose Miguel-Alonso, Luis A. Plana, and Steve Furber. Understanding the interconnection network of SpiNNaker. In *Proceedings of the 23rd International Conference on Supercomputing*, ICS '09, pages 286–295, New York, NY, USA, 2009. ACM. 62

[184] Martin O'Halloran, Brian McGinley, Raquel Cruz Conceicao, Fearghal Morgan, Edward Jones, and Martin Glavin. Spiking neural networks for breast cancer classification in a dielectrically heterogeneous breast. *Progress In Electromagnetics Research*, 113:413–428, 2011. 36

[185] National Research Council (US). Committee on Innovations in Computing, Communications, and Lessons from History. *Funding a revolution: government support for computing research*. National Academies Press, 1999. 350

[186] P.P. Palmes, T. Hayasaka, and S. Usui. Mutation-based genetic neural network. *Neural Networks, IEEE Transactions on*, 16(3):587 –600, May 2005. 24, 27, 145, 146

[187] Andrey Palyanov, Balazs Szigeti, Giovanni Idili, Jim Hokanson, Matteo Canatrelli, Michael Currie, Padraig Gleeson, Sergey Khayrulin, and Stephen Larson. Openworm. http://www.openworm.org/, 2014. 266

[188] Jaak Panksepp. *Affective Neuroscience: The Foundations of Human and Animal Emotions*. Oxford University Press, 1998. xv, 45, 202, 332, 333, 334, 335, 354

[189] R. Parekh, J. Yang, and V. Honavar. Constructive neural-network learning algorithms for pattern classification. *Neural Networks, IEEE Transactions on*, 11(2):436 –451, March 2000. 17, 147

[190] Merav Parter, Nadav Kashtan, and Uri Alon. Facilitated variation: how evolution learns from past environments to generalize to new environments. *PLoS computational biology*, 4(11):e1000206+, November 2008. 39, 161

[191] Stephen Paslaski, Courtland VanDam, and Juyang Weng. Modeling dopamine and serotonin systems in a visual recognition network. In *Neural Networks (IJCNN), The 2011 International Joint Conference on*, pages 3016–3023. IEEE, 2011. 52

[192] Ole Paulsen and Terrence J Sejnowski. Natural patterns of activity and long-term synaptic plasticity. *Current Opinion in Neurobiology*, 10(2):172 – 180, 2000. 349

[193] N.G. Pavlidis, D.K. Tasoulis, V.P. Plagianakos, G. Nikiforidis, and M.N. Vrahatis. Spiking neural network training using evolutionary algorithms. In *Neural Networks, 2005. IJCNN '05. Proceedings. 2005 IEEE International Joint Conference on*, volume 4, pages 2190–2194 vol. 4, July 2005. 34, 146, 147

[194] Israel Tabarez Paz, Neil Hernández Gress, and Miguel González Mendoza. Pattern recognition with spiking neural networks. In *Advances in Soft Computing and Its Applications*, pages 279–288. Springer, 2013. 33

[195] Rodrigo Perin, Thomas K. Berger, and Henry Markram. A synaptic organizing principle for cortical neuronal groups. *Proceedings of the National Academy of Sciences*, 108(13):5419–5424, 2011. 32

[196] Thomas Pfeil, Andreas Grübl, Sebastian Jeltsch, Eric Müller, Paul Müller, Mihai A. Petrovici, Michael Schmuker, Daniel Brüderle, Johannes Schemmel, and Karlheinz Meier. Six networks on a universal neuromorphic computing substrate. *Frontiers in neuroscience*, 7, 2013. 60

[197] H. Vincent Poor and Olympia Hadjiliadis. *Quickest Detection*. Cambridge University Press, Cambridge, 2009. 105

[198] Robert Preissl, Theodore M. Wong, Pallab Datta, Myron Flickner, Raghavendra Singh, Steven K. Esser, William P. Risk, Horst D. Simon, and Dharmendra S. Modha. Compass: a scalable simulator for an architecture for cognitive computing. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, SC '12, pages 54:1–54:11, Los Alamitos, CA, USA, 2012. IEEE Computer Society Press. 56

[199] M. Ptito, J.-F. Gigure, D. Boire, D.O. Frost, and C. Casanova. When the auditory cortex turns visual. In M. Ptito C. Casanova, editor, *Vision: From Neurons to Cognition*, volume 134 of *Progress in Brain Research*, pages 447 – 458. Elsevier, 2001. 40, 137, 161

[200] Joo Carlos Figueira Pujol and Riccardo Poli. Evolving the topology and the weights of neural networks using a dual representation. *Applied Intelligence*, 8:73–84, 1998. 22, 27

[201] J.R. Quinlan. Induction of decision trees. *Machine Learning*, 1(1):81–106, 1986. 138

[202] Mikhail I. Rabinovich, Pablo Varona, Allen I. Selverston, and Henry D. I. Abarbanel. Dynamical principles in neuroscience. *Rev. Mod. Phys.*, 78:1213–1265, Nov 2006. 338, 348

[203] Rudolf A. Raff. *The Shape of Life: Genes, Development, and the Evolution of the Animal Form*. The University of Chicago Press, 1996. 38, 39

[204] B. Rajendran, Yong Liu, Jae sun Seo, K. Gopalakrishnan, L. Chang, D.J. Friedman, and M.B. Ritter. Specifications of nanoscale devices and circuits for neuromorphic computational systems. *Electron Devices, IEEE Transactions on*, 60(1):246–253, Jan. 2013. 56

[205] A.D. Rast, F. Galluppi, X. Jin, and S.B. Furber. The leaky integrate-and-fire neuron: A platform for synaptic model exploration on the spinnaker chip. In *Neural Networks (IJCNN), The 2010 International Joint Conference on*, pages 1–8, July 2010. 62

[206] Alexander Rast, Francesco Galluppi, Sergio Davies, Luis Plana, Cameron Patterson, Thomas Sharp, David Lester, and Steve Furber. Concurrent heterogeneous neural model simulation on real-time neuromimetic hardware. *Neural Networks*, 24(9):961 – 978, 2011. Multi-Scale, Multi-Modal Neural Modeling and Simulation. 62

[207] Alexander D. Rast, Xin Jin, Francesco Galluppi, Luis A. Plana, Cameron Patterson, and Steve Furber. Scalable event-driven native parallel processing: The spinnaker neuromimetic system. In *Proceedings of the 7th ACM International Conference on Computing Frontiers*, CF '10, pages 21–30, New York, NY, USA, 2010. ACM. 62

[208] I. Rechenberg. Cybernetic solution path of an experimental problem. In *Royal Aircraft Establishment Translation No. 1122, B. F. Toms, Trans.* Ministry of Aviation, Royal Aircraft Establishment, Farnborough Hants, August 1965. 359

[209] I. Rechenberg. *Evolutionsstrategie – Optimierung technisher Systeme nach Prinzipien der biologischen Evolution*. Frommann-Holzboog, Stuttgart, GER, 1973. 359

[210] W. Scott Reilly and Joseph Bates. Building emotional agents. Technical Report CMU-CS-92-143, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA, May 1992. 50

[211] Joseph Reisinger, Kenneth O. Stanley, and Risto Miikkulainen. Evolving reusable neural modules. In *Proceedings of the Genetic and Evolutionary Computation Conference*, 2004. 41

[212] Rupert Riedl. *Order in living organisms: a systems analysis of evolution*. Wiley, 1978. 38

[213] F. Rosenblatt. The perceptron: A probabilistic model for information storage and organization in the brain. *Psychological Review*, 65(6):386–408, 1958. 350

[214] F. Rosenblatt. On the convergence of reinforcement procedures in simple perceptors. Technical Report VG-1196-G-4, Cornell Aeronautical Laboratory, Inc., Buffalo, NY, 1960. 350

[215] Cyrille Rossant, Dan F.M. Goodman, Bertrand Fontaine, Jonathan Platkiewicz, Anna K. Magnusson, and Romain Brette. Fitting neuron models to spike trains. *Frontiers in neuroscience*, 5:9, 2011. 36

[216] D.E. Rumelheart, G.E. Hinton, and R.J. Williams. Learning representations by back-propagating errors. *Nature*, 323:533–536, 1986. 13, 350

[217] A. Russell, G. Orchard, Yi Dong, S. Mihalas, E. Niebur, J. Tapson, and R. Etienne-Cummings. Optimization methods for spiking neurons and networks. *Neural Networks, IEEE Transactions on*, 21(12):1950–1962, Dec 2010. 35

[218] N. Saravanan and D.B. Fogel. Evolving neural control systems. *IEEE Expert*, 10(3):23 –27, June 1995. 14

[219] M. Schembri, M. Mirolli, and G. Baldassarre. Evolving internal reinforcers for an intrinsically motivated reinforcement-learning robot. In *Development and Learning, 2007. ICDL 2007. IEEE 6th International Conference on*, pages 282 –287, july 2007. 50

[220] J. Schemmel, D. Brderle, A Grbl, M. Hock, K. Meier, and S. Millner. A wafer-scale neuromorphic hardware system for large-scale neural modeling. In *Circuits and Systems (ISCAS), Proceedings of 2010 IEEE International Symposium on*, pages 1947–1950, May 2010. 59

[221] J. Schemmel, J. Fieres, and K. Meier. Wafer-scale integration of analog neural networks. In *Neural Networks, 2008. IJCNN 2008. (IEEE World Congress on Computational Intelligence). IEEE International Joint Conference on*, pages 431–438, June 2008. 58

[222] Johannes Schemmel, Andreas Grubl, Karlheinz Meier, and Eilif Mueller. Implementing synaptic plasticity in a VLSI spiking neural network model. In *Neural Networks, 2006. IJCNN'06. International Joint Conference on*, pages 1–6. IEEE, 2006. 58, 60

[223] Stefan Schliebs, Michaël Defoin-Platel, Sue Worner, and Nikola Kasabov. Integrated feature and parameter optimization for an evolving spiking neural network: Exploring heterogeneous probabilistic models. *Neural Networks*, 22(5):623–632, 2009. 34

[224] J. Schmidhuber. Curious model-building control systems. In *Neural Networks, 1991. 1991 IEEE International Joint Conference on*, volume 2, pages 1458 – 1463, November 1991. 47, 48

[225] J. Schmidhuber. Developmental robotics, optimal artificial curiosity, creativity, music, and the fine arts. *Connection Science*, 18(2):173–187, 2006. 48

[226] J. Schmidhuber. Formal theory of creativity, fun, and intrinsic motivation (1990-2010). *Autonomous Mental Development, IEEE Transactions on*, 2(3):230 –247, sept. 2010. 48

[227] Catherine D. Schuman and J. Douglas Birdwell. Dynamic artificial neural networks with affective systems. *PLoS ONE*, 8(11):e80455, 11 2013. 202

[228] Catherine D. Schuman and J. Douglas Birdwell. Variable structure dynamic artificial neural networks. *Biologically Inspired Cognitive Architectures*, 6(0):126 – 130, 2013. 66, 104

[229] Catherine D. Schuman, J. Douglas Birdwell, and Mark Dean. Neuroscience-inspired dynamic architectures. In *Biomedical Science and Engineering Center Conference (BSEC), 2014 Annual Oak Ridge National Laboratory*, pages 1–4, May 2014. 66, 137

[230] Catherine D Schuman, J Douglas Birdwell, and Mark E Dean. Spatiotemporal classification using neuroscience-inspired dynamic architectures. *Procedia Computer Science*, 41:89–97, 2014. 66, 126

[231] H.P. Schwefel. *Evolutionsstrategie und numerische optimierung*. PhD thesis, Technische Universität Berlin, 1975. 359

[232] H.P. Schwefel. *Numerische Optimierung Von Computer-Modellen Mittels Der Evolutionsstrategie*. Birkhäuser, Basel, 1977. 359

[233] J. Seo, B. Brezzo, Yong Liu, B.D. Parker, S.K. Esser, R.K. Montoye, B. Rajendran, J.A Tierno, L. Chang, D.S. Modha, and D.J. Friedman. A 45nm cmos neuromorphic chip with a scalable architecture for learning in networks of spiking neurons. In *Custom Integrated Circuits Conference (CICC), 2011 IEEE*, pages 1–4, Sept 2011. 55

[234] Rudy Setiono and Lucas Chi Kwong Hui. Use of a quasi-newton method in a feedforward neural network construction algorithm. *Neural Networks, IEEE Transactions on*, 6(1):273–277, 1995. 147

[235] Thomas Sharp, Francesco Galluppi, Alexander Rast, and Steve Furber. Power-efficient simulation of detailed cortical microcircuits on spinnaker. *Journal of neuroscience methods*, 210(1):110–118, 2012. 62

[236] Nils T. Siebel and Gerald Sommer. Evolutionary reinforcement learning of artificial neural networks. *Int. J. Hybrid Intell. Syst.*, 4(3):171–183, August 2007. 24, 27

[237] N.T. Siebel, J. Botel, and G. Sommer. Efficient neural network pruning during neuro-evolution. In *Neural Networks, 2009. IJCNN 2009. International Joint Conference on*, pages 2920 –2927, June 2009. 24, 27

[238] Allan Siegel and Hreday N. Sapru. *Essential Neuroscience*. Lippencott Williams and Wilkins, 2nd edition edition, 2011. xv, 330, 332, 333, 334, 353

[239] S. Singh, A. Barto, and N. Chentanez. Intrinsically motivated reinforcement learning. In *18th Annual Conference on Neural Information Processing Systems (NIPS)*, 2004. 47

[240] S. Singh, R.L. Lewis, A.G. Barto, and J. Sorg. Intrinsically motivated reinforcement learning: An evolutionary perspective. *Autonomous Mental Development, IEEE Transactions on*, 2(2):70 –82, june 2010. 47

[241] Andrea Soltoggio and Kenneth O Stanley. From modulated hebbian plasticity to simple behavior learning through noise and weight saturation. *Neural Networks*, 34:28–41, 2012. 53

[242] Olaf Sporns. Networks analysis, complexity, and brain function. *Complex.*, 8(1):56–60, September 2002. 6, 28

[243] Kenneth Stanley. Compositional pattern producing networks: A novel abstraction of development. *Genetic Programming and Evolvable Machines*, 8(2):131–162, 2007. 39, 42

[244] Kenneth O. Stanley, David B. D'Ambrosio, and Jason Gauci. A hypercube-based encoding for evolving large-scale neural networks. *Artificial Life*, 15(2):185–212, 2009. 25, 27, 28, 265

[245] Kenneth O. Stanley and Risto Miikkulainen. Evolving neural networks through augmenting topologies. *Evolutionary Computation*, 10(2):99–127, 2002. 22, 27, 265

[246] K.O. Stanley, B.D. Bryant, and R. Miikkulainen. Evolving adaptive neural networks with and without adaptive synapses. In *Evolutionary Computation, 2003. CEC '03. The 2003 Congress on*, volume 4, pages 2557 – 2564, dec. 2003. 23, 27, 265

[247] K.O. Stanley, J. Reisinger, and R. Miikkulainen. Exploiting morphological conventions for genetic reuse. In *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO-2004) Workshop Program*, New York, NY, 2004. Springer-Verlag. 41

[248] Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning: An Introduction*. MIT Press, 1998. 13, 47, 347, 361, 362, 363, 364

[249] J. Takeuchi, O. Shouno, and H. Tsujino. Modular neural networks for reinforcement learning with temporal intrinsic rewards. In *Neural Networks, 2007. IJCNN 2007. International Joint Conference on*, pages 1151 –1156, aug. 2007. 50

[250] K.S. Tang, C.Y. Chan, K.F. Man, and S. Kwong. Genetic structure for nn topology and weights optimization. In *Genetic Algorithms in Engineering*

Systems: Innovations and Applications, 1995. GALESIA. First International Conference on (Conf. Publ. No. 414), pages 250 –255, sep 1995. 20, 27

[251] Marco Tomassini. *Spatially structured evolutionary algorithms*. Springer, 2005. 265

[252] Hiroshi Tsujino, Johane Takeuchi, and Osamu Shouno. Basal ganglia models for autonomous behavior learning. In Bernhard Sendhoff, Edgar Krner, Olaf Sporns, Helge Ritter, and Kenji Doya, editors, *Creating Brain-Like Intelligence*, volume 5436 of *Lecture Notes in Computer Science*, pages 328–350. Springer Berlin / Heidelberg, 2009. 50

[253] Nakaakira Tsukahara. Synaptic plasticity in the mammalian central nervous system. *Ann. Rev. Neurosci.*, 4:351–379, 1981. 354

[254] Harry L. Van Trees. *Detection, Estimation, and Modulation Theory, Part I: Detection, Estimation, and Linear Modulation Theory*. Wiley, New York, 1968. 108, 109

[255] R. Jacob Vogelstein, Udayan Mallik, Eugenio Culurciello, Gert Cauwenberghs, and Ralph Etienne-Cummings. Saliency-driven image acuity modulation on a reconfigurable array of spiking silicon neurons. In *Advances in neural information processing systems*, pages 1457–1464, 2004. 57, 58

[256] R. Jacob Vogelstein, Udayan Mallik, Eugenio Culurciello, Gert Cauwenberghs, and Ralph Etienne-Cummings. A multichip neuromorphic system for spike-based visual information processing. *Neural computation*, 19(9):2281–2300, 2007. 58

[257] R .Jacob Vogelstein, Udayan Mallik, Eugenio Culurciello, Ralph Etienne-Cummings, and Gert Cauwenberghs. Spatial acuity modulation of an address-event imager. In *Electronics, Circuits and Systems, 2004. ICECS 2004.*

Proceedings of the 2004 11th IEEE International Conference on, pages 207–210. IEEE, 2004. 58

[258] R.J. Vogelstein, U. Mallik, and G. Cauwenberghs. Silicon spike-based synaptic array and address-event transceiver. In *Circuits and Systems, 2004. ISCAS '04. Proceedings of the 2004 International Symposium on*, volume 5, pages V–385–V–388 Vol.5, May 2004. 57

[259] R.J. Vogelstein, U. Mallik, J.T. Vogelstein, and G. Cauwenberghs. Dynamically reconfigurable silicon array of spiking neurons with conductance-based synapses. *Neural Networks, IEEE Transactions on*, 18(1):253–265, Jan 2007. 57, 58

[260] John Von Neumann. *The computer and the brain*. Yale University Press, 1958. 2

[261] Gunter P. Wagner. Homologues, natural kinds and the evolution of modularity. *American Zoologist*, 36(1):pp. 36–43, 1996. 39, 161

[262] Gunter P. Wagner and Lee Altenberg. Perspective: Complex adaptations and the evolution of evolvability. *Evolution*, 50(3):pp. 967–976, 1996. 39, 161

[263] Van J. Wedeen, Douglas L. Rosene, Ruopeng Wang, Guangping Dai, Farzad Mortazavi, Patric Hagmann, Jon H. Kaas, and Wen-Yih I. Tseng. The geometric structure of the brain fiber pathways. *Science*, 335(6076):1628–1634, 2012. 337

[264] David White and Panos Ligomenides. GANNet: A genetic algorithm for optimizing topology and weights in neural network design. In José Mira, Joan Cabestany, and Alberto Prieto, editors, *New Trends in Neural Computation*, volume 686 of *Lecture Notes in Computer Science*, pages 322–327. Springer Berlin / Heidelberg, 1993. 19, 27

[265] D. Whitley, T. Starkweather, and C. Bogart. Genetic algorithms and neural networks: optimizing connections and connectivity. *Parallel Computing*, 14(3):347 – 361, 1990. 19, 27

[266] A.P. Wieland. Evolving neural network controllers for unstable systems. In *Neural Networks, 1991., IJCNN-91-Seattle International Joint Conference on*, volume ii, pages 667 –673 vol.2, jul 1991. 14, 27

[267] Gordon Winocur, Morris Moscovitch, and Melanie Sekeres. Memory consolidation or transformation: context manipulation and hippocampal representations of memory. *Nat Neurosci*, 10(5):555–557, May 2007. 338

[268] H.S. Witsenhausen. Separation of estimation and control for discrete time systems. *Proceedings of the IEEE*, 59(11):1557–1566, 1971. 361

[269] Daniel M. Wolpert, R. Chris Miall, and Mitsuo Kawato. Internal models in the cerebellum. *Trends in Cognitive Sciences*, 2(9):338 – 347, 1998. 39

[270] Simei Gomes Wysoski, Lubica Benuskova, and Nikola Kasabov. On-line learning with structural adaptation in a network of spiking neurons for visual pattern recognition. In *Artificial Neural Networks–ICANN 2006*, pages 61–70. Springer, 2006. 34

[271] Simei Gomes Wysoski, Lubica Benuskova, and Nikola Kasabov. Fast and adaptive network of spiking neurons for multi-view visual pattern recognition. *Neurocomputing*, 71(1315):2563 – 2575, 2008. Artificial Neural Networks (ICANN 2006) / Engineering of Intelligent Systems (ICEIS 2006). 33, 34

[272] Jianguo Xin and M.J. Embrechts. Supervised learning with spiking neural networks. In *Neural Networks, 2001. Proceedings. IJCNN '01. International Joint Conference on*, volume 3, pages 1772–1777, 2001. 33

[273] X. Yao and Y. Liu. A new evolutionary system for evolving artificial neural networks. *Neural Networks, IEEE Transactions on*, 8(3):694 –713, May 1997. 21, 22, 27, 41, 146, 147

[274] Xin Yao. Evolving artificial neural networks. *Proceedings of the IEEE*, 87(9):1423 –1447, September 1999. 13, 17

[275] Xin Yao and M.M. Islam. Evolving artificial neural network ensembles. *Computational Intelligence Magazine, IEEE*, 3(1):31 –42, February 2008. 43

[276] Xin Yao and Yong Liu. Towards designing artificial neural networks by evolution. *Applied Mathematics and Computation*, 91(1):83 – 90, 1998. 22, 27

Appendix

Appendix A

A Brief Introduction to the Brain

This research is largely inspired by topics from neuroscience. A brief introduction to the biological systems in the brain is warranted and is given below.

A.1 Neurons and Synapses

Neurons are the nerve cells present in the brain. The human brain consists of about 10^{11} neurons, each of which operates in parallel with the others. A *process* in neuroscience usually refers a physical feature. The various processes of the neuron are called *neurites*; henceforth, the term neurite will be used rather than process to avoid confusion with the computer science notion of process. The neuron receives information through neurites called *dendrites*, which also communicate the information to the neuron's cell body. The neurite that transmits information out of the neuron to other targets is called the *axon*. Signals between neurons are usually transferred across *synapses*, although direct connections that allow ion exchange have been observed. Typically, the communication is done chemically via *neurotransmitters* [238].

Neurons can be categorized in several different ways. One way to categorize the neurons is by the number of axons and dendrites [238]. These categories are:

- **Multipolar neurons**: These are the most common neurons in the brain. They possess two or more dendrites and one long axon.

- **Bipolar neurons**: Two neurites, one on each end, emerge from the cell body. One of the neurites ends in dendrites, and the other ends in an axon with terminals in the central nervous system. These neurons typically have sensory functions that receive signals from other neurons, and they transmit information received by the dendrites within the central nervous system.

- **Pseudo-unipolar neurons**: For these neurons, a single neurite emerges from the cell body and divides into two branches, each of which has the structural and functional characteristics of an axon.

- **Unipolar neurons**: These neurons are relatively rare. In these neurons, dendrites arise from one end of the neuron, and an axon arises from the site where the dendrites are located.

Another way to categorize neurons is by the neurons with which they are connected [46]. Using this categorization, there are two main kinds of neurons in the brain:

- **Projection neurons**: These neurons can influence neurons or other cells across relatively large distances because they have very long axons. These neurons can have axons more than one meter long. Projection neurons are the type of neurons that convey information from the spinal cord to muscles.

- **Interneurons**: Interneurons have relatively short axons that can branch significantly within the immediate area of the neuron. The term usually refers to neurons with short axons that do not leave a particular group of neurons.

Dendrites are usually shorter than axons and arise from the cell body of the neuron. They generally branch off into dendritic spines, which receive information from axons from other neurons. The dendritic spines are typically where the

communication between neurons across synapses and from axons takes place, although sometimes communication is direct from cell body to cell body, or between dendrites.

Although information is transmitted from an axon to a dendrite in a typical synapse, there are also synapses between two axons, two dendrites, and synapses and from axons in which information travels from dendrite to axon. Because of these differences, connections between neurons in the artificial neural networks defined in this work will all be referred to only as synapses, with no distinction between dendrites and axons. The synapses in this work are directed in that information travels from one neuron to another, but not the opposite direction along that synapse.

There are two ways for synaptic transmission to take place in the brain: electrical transmission and chemical transmission. Electrical transmission occurs when the current generated by one neuron spreads to another neuron on a pathway of low electrical resistance. Electrical synapses are relatively rare in the mammalian brain; evidence suggests that they occur in regions where the activities of neighboring neurons need to be highly synchronized. In chemical transmissions, neurotransmitters are transmitted from one neuron to another [238].

A *neurotransmitter* is a chemical substance that is synthesized in a neuron and is released at a synapse following depolarization of the nerve terminal. The neurotransmitter then binds to receptors at a postsynaptic cell and/or postsynaptic terminal to elicit a response. This response may excite or inhibit the neuron, meaning neurotransmitters play a major role in the way the brain operates. A list of some of the known neurotransmitters and their effects is given in Table A.1; more complete lists are available in [188, 238].

Neurotransmitters are released according to action potentials in the neuron. An action potential is a potential that develops across the membrane of a neuron. Neurons have a particular membrane potential in which they are at rest. Typically, a neuron is "at rest" when the potential inside the neuron's cell wall is approximately -70 mV compared to the outside of the neuron. When positively charged ions flow out of the cell, the membrane potential becomes more negative, while current flowing into

Table A.1: Neurotransmitters and their effects [188, 238].

Neurotransmitter	Effect
Acetylcholine	Implicated in regulation of forebrain activity during cycles of sleep and wakefulness. Also involved in learning and memory. Mediates attention and arousal in secondary systems. Can promote the impulse of anger and aggression.
Glutamate	Excitatory neurotransmitter. May be involved in learning and memory processes. Prolonged stimulation of neurons by excitatory amino acids can lead to neuronal death or injury. Can promote the impulse of anger and aggression. Key transmitter for generating the unconditioned response of fear.
Gamma aminobutyric acid (GABA)	Inhibitory amino acid. Capable of suppressing fear as well as many other emotional and motivational processes.
Glycine	Inhibitory amino acid. In the brain stem, exerts inhibition over a wide range of motor processes.
Dopamine	Excitatory at some sites, inhibitory at others. Maintains psychomotor and motivational focus and arousal. Plays a major role in the SEEKING system, discussed in Section 2.5.
Norepinephrine	Excitatory at some sites, inhibitory at others; believed to play a role in depression. Sustains high signal/noise ratios in sensory processing areas.
Serotonin	Plays a role in mediating affective processes such as aggression and arousal. Important role in depression. Excitatory at some, inhibitory at others. Reduces impact of incoming information and cross-talk between sensory channels. Low levels of serotonin can lead to more aggressive behavior.
Histamine	Excitatory at some sites, inhibitory at others. Present in almost all areas of the brain.

the cell changes the membrane potential to a less negative or positive value. Each neuron has an associated threshold level. If the membrane potential rises above this threshold level, the neuron generates an action potential [65]. The generation of the action potential is called a "firing" of the neuron.

The generation of an action potential relies not only on the threshold of the neuron but also on the recent firing history. Each neuron has an associated refractory period. For a short period of time after a neuron has fired, it is highly unlikely that that neuron will fire again. This period is called the absolute refractory period. For a slightly longer period of time after the absolute refractory period, it is difficult, but more likely, for the neuron to fire again. This period is called the relative refractory period [65].

In the central nervous system, multiple types of cells provide myelin sheaths along axons.* Myelin is a fat that provides an insulating layer for the axon. The thickness of the myelin sheath controls the propagation delay of signals along the axon. Myelin sheaths are separated along the axon by *nodes of Ranvier*. The action potential traveling along the axon is regenerated at each of the nodes of Ranvier.

Figure A.1 shows a typical neuron.

A.2 Parts of the Brain

The basal ganglia is one of the most important layers of the brain for emotion processing and generation; it is also known as the reptilian brain. The basal ganglia is the portion of the brain that contains innate behavioral knowledge, including motor functions and primal emotions such as fear, anger, and sexuality [188]. It is also responsible for motor integration in the cerebral cortex [238].

*Nervous tissue in the brain is categorized into two types: gray matter and white matter. Gray matter consists mainly of the neuron cell bodies (which gives gray matter its color) and the dendrites and non-myelinated axons, while white matter is predominantly myelinated axons. The color of white matter is due to the whitish color of myelin [46].

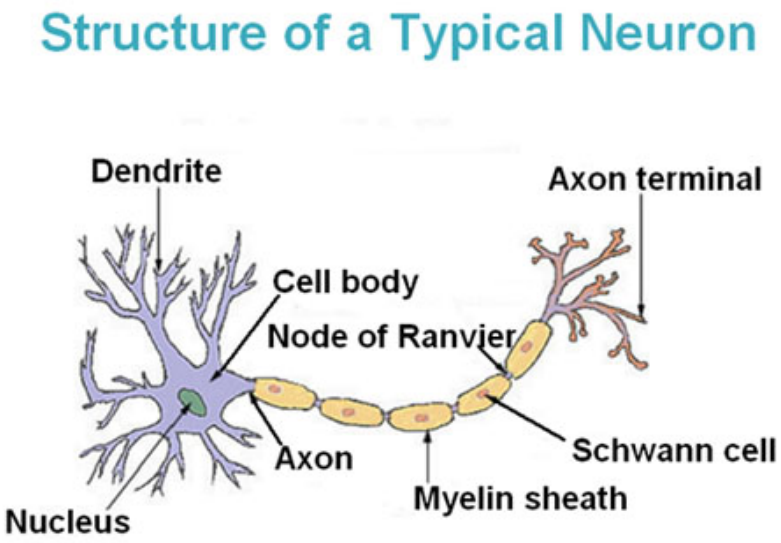


Figure A.1: A typical biological neuron[†].

The next layer of the brain, known as the limbic system or the visceral brain, is where many of the various social emotions are processed. It processes most affective knowledge, generating more sophisticated emotional responses [188]. The limbic system also appears to mediate or control memory processes. Both the amygdala and the hippocampus are part of the limbic system. The hippocampus plays an important role in memory formation in the brain, particularly short term memory. The amygdala is important for learning associations between stimuli and emotional value [46]. For example, the amygdala may associate fear with a stimulus that causes pain.

The neocortex is the structure in the brain that is more evolved in human brains than in other mammal brains. The neocortex is responsible for associating a diversity of sensations and innate ideas, such as a sense of causality and spatial referencing, into perception, concepts and attributions. This is the portion of the brain that contains what we think of as the rational mind and the imagination and the part of the brain that generates ideas [188]. The neocortex in humans is organized in six layers, which are parallel to the surface of the cortex. The neurons in the neocortex are organized

[†] "Neuron". Licensed under Public domain via Wikimedia Commons - http://commons.wikimedia.org/wiki/File:Neuron.jpg#mediaviewer/File:Neuron.jpg

in cylindrical columns (cortical columns), which are perpendicular to the cortical surface. Axons that traverse vertically in the neocortex typically form connections to neurons within a column, but among the neurons in different layers. Axons that traverse horizontally in the neocortex allow communication between neurons in different columns [46]. Figure A.2 shows the general structure of the brain.

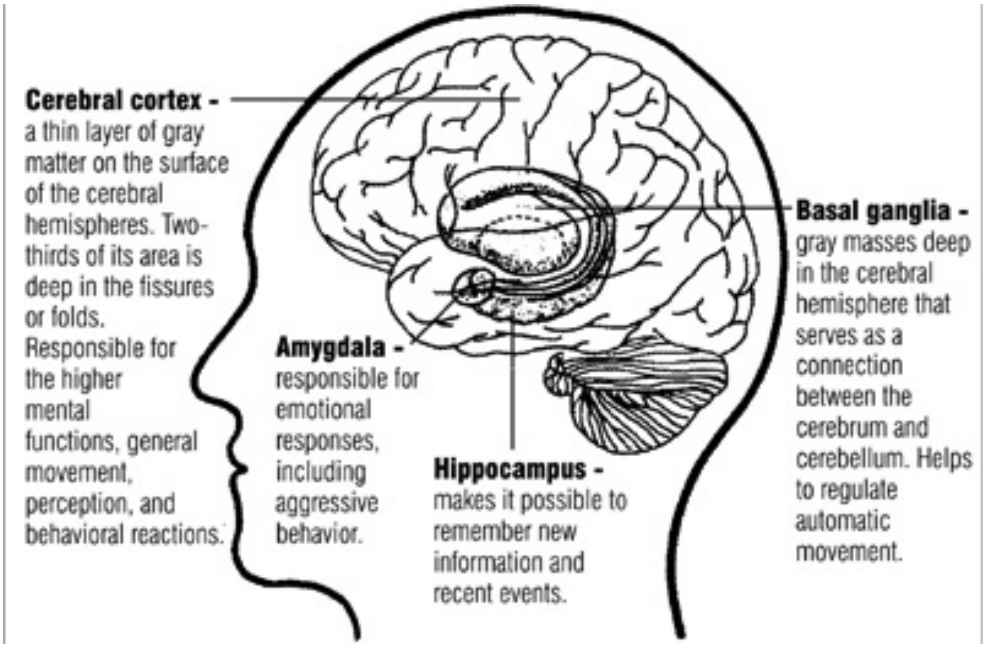


Figure A.2: The human brain[‡].

A.3 Development of the Brain

The shape of the brain, the position of neuronal groups, and the main neural connections all arise prior to birth. These factors are all determined by the individual's DNA, although methylation of specific sites in the DNA can cause the genes to be expressed differently, without a change to the DNA itself having taken place [46].

Initial axon growth is determined by genetics, but most axonal growth and connections are determined by signals in the environment during development.

[‡]This work is a derivative of an image by National Institutes of Mental Health, National Institutes of Health [Public domain], via Wikimedia Commons.

Trophic factors are substances that promote growth in the axons. Axons can have *neuronal-cell adhesion molecules* (N-CAMs), which make them sticky, and axons with similar N-CAMs are kept together. These N-CAMs also repel or inhibit the growth of other axons. Trophic factors operate on a part of the axon known as the *growth cone*. The growth cone is on the tip of the axon, and it sends out extensions called *filopodia* to explore its surroundings. The filopodia that find the proper chemicals in the surrounding tissue are stabilized, and the filopodia that do not find the proper chemicals retract. This directs the growth of the axon in the correct direction [46].

In Weeden *et al.*, it was shown that connectivity in the brain formed a grid structure that was maintained at varying scales in the brain. They suggest that grid structure increases the efficacy of path orientation to aid axonal path-finding in the brain. That is, grid structure simplifies axonal path-finding, compared with models that allow less constrained connectivity (connections to form in any direction). They speculate that the grid organization of the brain may represent a default connectivity, which is elaborated through evolution and development [263].

The number and density of synapses in the brain undergo significant changes during human growth until the end of puberty. The number of synapses in the brain increases until shortly after birth (within the first year of life). Then, the number of synapses begins to decrease significantly until the end of puberty. It is speculated that the decrease occurs in synapses that are not used frequently [46].

The structure of the brain can be changed due to how it is used. Most of the use-dependent structural changes occur at the synaptic level. Experiments have shown that an increase of synaptic density is related to an improvement in performance in learning and memory tasks [46], as discussed in the next section.

A.4 Memory, Learning, and Plasticity in Biological Brains

There are two types of memory in the brain: declarative memory and nondeclarative memory. Declarative memory is explicit memory and typically depends on the hippocampus and other areas of the brain [73]. Declarative memory includes episodic memory (memory of events from one's life) and semantic memory (general knowledge of the world). The hippocampus retains context-dependent memories until they are consolidated in neocortical structures, but there is evidence that these memories are stored differently in the two structures [267].

Nondeclarative memory is implicit, procedural memory and depends mostly on the basal ganglia and parts of the cerebral cortex (including the neocortex). Nondeclarative memory is needed to learn skills, such as swimming [46].

For the most part, it is still unclear precisely how learning and memory work in the human brain. However, it is clear that in order for the brain to learn, the structure of the brain must be somewhat plastic; that is, the structure must be able to adapt.

Synaptic plasticity dependent on the activity of the synapses is widely thought to be the mechanism through which learning and memory take place. Hebb proposed the idea that if the action potential from one neuron caused another neuron to fire, then the synapse along which the action potential travelled should be strengthened [113]. This relationship is called the Hebb rule. It has been generalized to include decreases in strength as well. These decreases take place when a particular synapse repeatedly fails to be involved in the firing of a neuron and are supported by experiment [65, 202].

The effects of these increases and decreases of strength in the synapses can be both short-term and long-term. If the effects last a significant period of time, they are called long-term potentiation (LTP) and long-term depression (LTD). The basic Hebb rule takes the following form:

$$\tau_w \frac{d\mathbf{w}}{dt} = v\mathbf{u} \tag{A.1}$$

where τ_w is a time constant that controls the rate at which rates change, $\mathbf{w}$ is the vector representing all of the synaptic weights coming into a particular neuron, $\mathbf{u}$ is a vector of the inputs from the synapses coming into the neuron, and v is the activity evoked in the neuron by $\mathbf{u}$ [65]. This equation implies that if activity along a synapse causes a neuron to fire, then the weight of that synapse will be increased. That is, if the firing of one neuron leads to the firing of another neuron, then the synapse between those two neurons should be strengthened. Synaptic plasticity is seen as a slow process that occurs gradually over time, and the rate of the change is specified by the time constant τ_w.

Hebb's rule applies to synaptic plasticity only. There is also evidence that more large-scale changes to the structure of the brain can occur in fully developed brains.

In [153], Maguire *et al.* found that the posterior hippocampi, the region of the brain that stores a spatial representation of the environment, was significantly larger in the brains of London taxi drivers than the same region in subjects who were not taxi drivers. That is, they found that there was a capacity for local plastic change in the structure of an adult human brain in response to the environment. In this case, the taxi drivers had a high dependence on superior navigation skills, so the area responsible for those abilities in the brain was more developed.

In [69], Draganski *et al.* compared the structure of the brain in subjects before and after they were taught to juggle. Their results showed temporary structural changes in areas of the subjects' brains. Comparing the subjects with non-jugglers, Draganski *et al.* were able to show that the structural changes were related to the subjects' juggling training. They were able to show that training can affect structure in the brain. The alterations were such that they were able to observe them macroscopically using three-dimensional magnetic-resonance imaging (MRI). They speculated that the structural changes may be related to changes in synaptic bulk and neuritis or they might include increased cell genesis.

Gaser and Schlaug investigated structural differences in the brains of musicians and non-musicians in [93]. They were able to demonstrate significant structural

differences in the brains of professional musicians, amateur musicians, and non-musicians. Though they note that some structural differences may be innate, the strong associations that they found between the structural differences, the status of the musicians, and the amount of time the musicians practiced led them to believe that the majority of the structural differences were acquired through structural plasticity.

Appendix B

Artificial Neural Networks

Artificial neural networks can be thought of as directed weighted graphs, where the neurons are the nodes and the synapses are the directed edges. All neural networks referred to in this section are artificial networks unless otherwise specified.

B.1 Neural Network Architectures

Network architectures come in several flavors [112]. Networks are made up of input neurons, output neurons, and "hidden" neurons. The hidden neurons are those that are neither input nor output neurons. There are multiple structures of networks that are used in neural network literature. These structural types include:

- Feed-forward neural networks: There is a layer of input neurons, zero or more layers of hidden neurons, and an output layer. Input layers only contain outgoing edges, and the edges of one layer are only connected to the next layer (whether it be a hidden layer or the output layer). Networks may either be fully connected, in the sense that every neuron in a layer has a directed edge to every neuron in the next layer, or they may only be partially connected, where some of these edges are missing. An example of a feed-forward neural network is shown in Figure B.1.

- Recurrent neural networks: Recurrent neural networks contain at least one loop, cycle, or feedback path. A loop in a directed graph is when there is an edge from a node to itself. Cycles in a directed graph occur when there is a path from a node to itself that contains other nodes. Feedback loops and paths typically involve delay elements. Feedback allows for storage to take place in the neurons; it gives the network a sense of memory from one instance to the next. An example of a recurrent neural network is shown in Figure B.2.

- Modular neural networks: A neural network is modular if the computation performed by the network can be decomposed into two or more subsystems that operate on distinct inputs without communication. The outputs of these modules are then combined to form the outputs of the network.

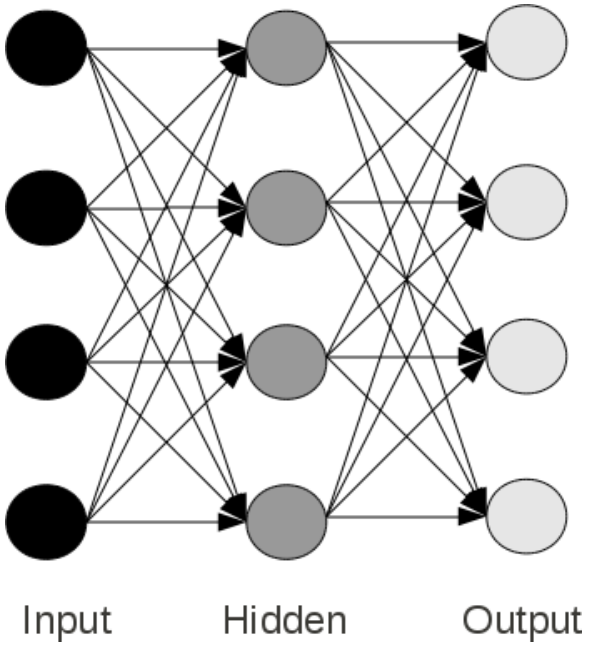


Figure B.1: An example of a fully-connected feed-forward neural network.

Recurrent neural networks can be divided further into discrete-time neural networks and continuous-time neural networks. In discrete time neural networks, time $t \in \{0, 1, 2, ...\}$ or a set isomorphic to this set. Charge is applied at the inputs at time t and propagates through the network, producing output no earlier than time

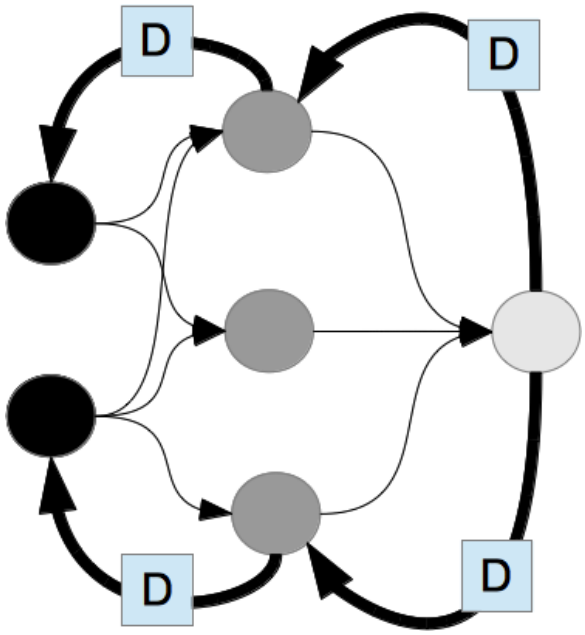


Figure B.2: An example of a recurrent neural network. The recurrent connections are bolded in the network. The delay elements are indicated with boxes labeled with D.

t. For recurrent neural networks, the outputs at time step t of neurons that have recurrent connections are applied to the recipient neurons at time step t or later. Continuous-time neural networks model behaviors such as spikes in the network at infinitesimally small time steps. These spikes are typically modeled using a differential equation rather than discrete events and may not have a stable solution [104].

B.2 Neuron Models

Neurons in neural networks are the information processing units of the network. Neurons usually sum signals they receive from their connections, and an activation function is applied to the result (as shown in Figure B.3). This neuron model is a static map in that there is no memory in the neuron. Neurons described using differential or difference equations have memory.

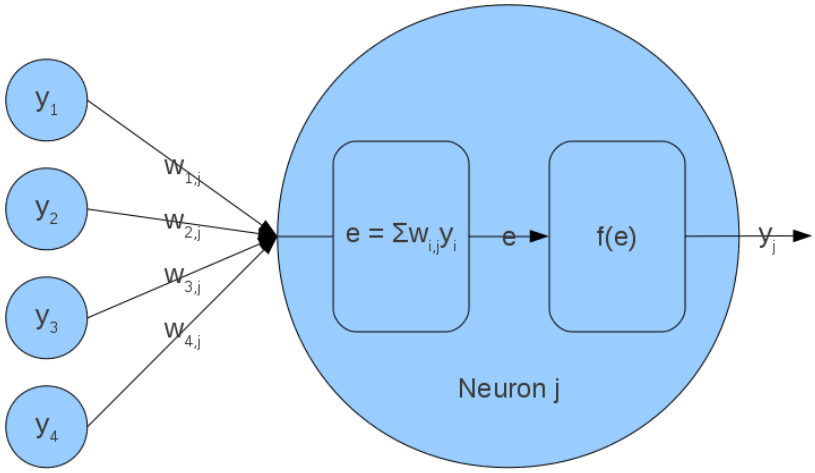


Figure B.3: A typical neuron model.

A neuron in the network is said to *fire* if the output value y_j is non-zero. Several different activation functions are commonly used. Some are listed below, with examples and corresponding graphs:

- The threshold function:

$$f(e) = \begin{cases} 1 & \text{if } e \geq 0 \\ 0 & \text{if } e < 0 \end{cases} \tag{B.1}$$

- Piecewise-Linear function (also called saturation):

$$f(e) = \begin{cases} 1 & \text{if } e \geq 1/2 \\ e + 1/2 & \text{if } -1/2 < e < 1/2 \\ 0 & \text{if } e \leq -1/2 \end{cases} \tag{B.2}$$

- Sigmoid function (where a determines the slope at $e = 0$):

$$f(e) = \frac{1}{1 + \exp(-ae)} \tag{B.3}$$

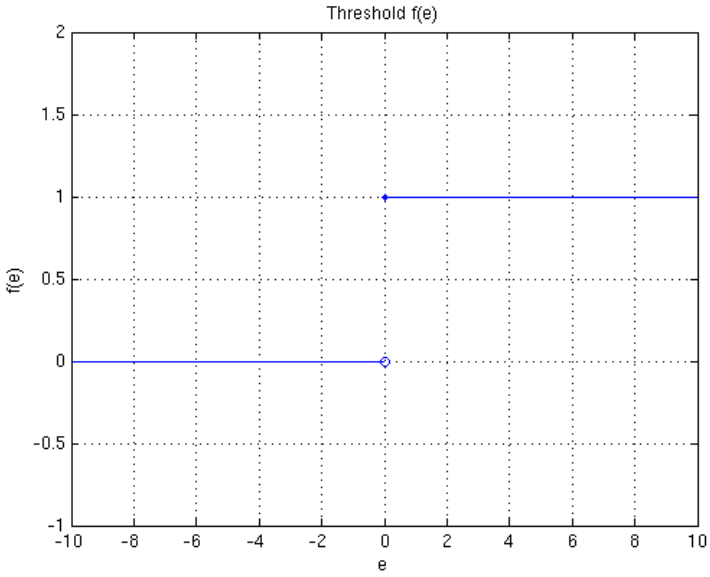


Figure B.4: Threshold activation function.

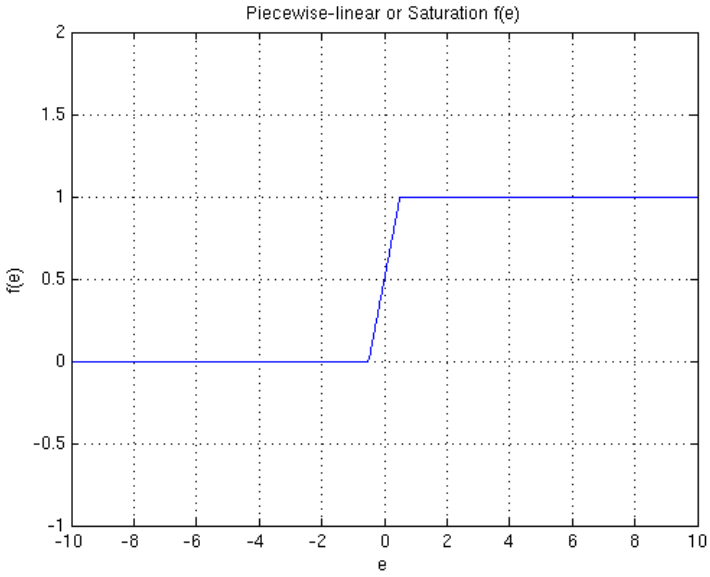


Figure B.5: Piecewise-linear activation function.

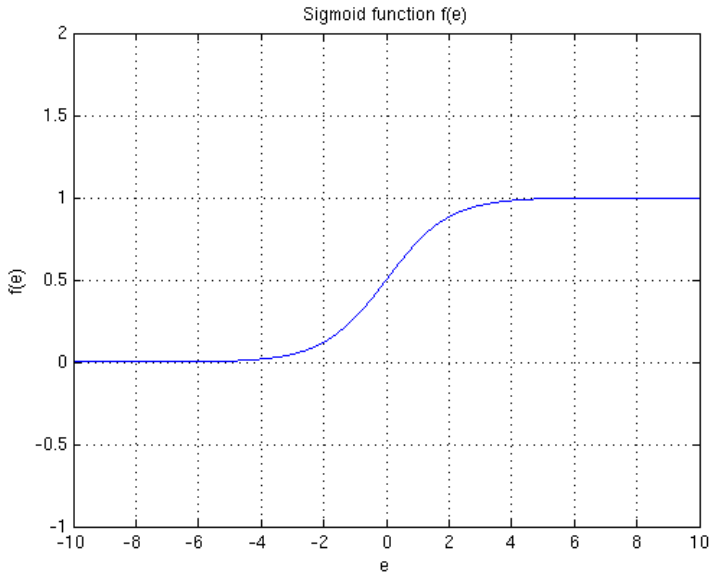


Figure B.6: Sigmoid activation function.

B.3 Training a Neural Network

Training in a neural network has canonically meant changing the weights of the connections and/or the threshold values. Relatively recently, training has also referred to changes in the architecture of the network. We categorize neural networks with training algorithms that cannot change the architecture of networks as **fixed-structure**. Similarly, we will categorize networks with training algorithms that can change the architecture as **variable-structure**.

There are two main methods of training: gradient-based methods and evolutionary methods (discussed in Appendix C). A discussion of evolutionary methods as applied to neural networks and methods that change the architecture of the network is given in Section 2.1.

Back-propagation is the most widely used algorithm for training neural networks in a supervised way. The algorithm is supervised because it requires a set of inputs and their corresponding outputs, called a training set. Back-propagation has two distinct phases: a forward pass and a backward pass. In the forward pass, input signals are propagated through the network, to produce output signals. This output is compared

with the expected output, producing an error. The error signals are then propagated backwards through the network, where the weights of the networks are adjusted in order to minimize the mean-squared error. Back-propagation can also be used to train a neural network when the error is a temporal difference error, obtained from a temporal difference method such as SARSA(λ) or Q(λ) in reinforcement learning [248]. Back propagation is a gradient-based optimization technique. It makes use of the gradient of an error function, evaluated using a training data set, with respect to the weights in the network. That is, back propagation uses the gradient of an error to determine how the weights in the network should be changed to reduce the error.

One of the known limitations of back propagation and other supervised learning algorithms is that they typically do not scale well. Gradient-based optimization algorithms have several known limitations as well. Because the weights are changed so that the error follows the steepest direction (in the space of weights) of descent, the results and speed of convergence of the optimization algorithm depend largely on the initial starting point. If the initial starting point is located near local optima and far from the global optimum, the back-propagation algorithm will likely converge to one of the local optima. This is a drawback for the back propagation algorithm because complex systems often have many local optima with significantly different (poorer) performance than a global optimum. Other limitations to back propagation are noted in Section 2.1.

It is also known that gradient-based optimization does not work well for recurrent neural networks. Bengio *et al.* proposed several alternative training methods such as simulated annealing, multi-grid random search, and time-weighted pseudo-Newton optimization for training [27]. Neuroevolution methods have also been proposed and are discussed in detail in Section 2.1.

For unsupervised learning in neural networks, a competitive learning rule is usually used. In competitive learning, the output neurons of a neural network compete among themselves to determine which one should be fired. A neuron learns by decreasing

synaptic weights of inactive input nodes and increasing the weights of active input nodes.

Another type of learning in neural networks is Hebbian learning, as discussed in A.4. Hebb's rule for synapse learning is given in two parts in [113], as described in [112]:

1. If two neurons on either side of the synapse (connection) are activated simultaneously, the strength of that synapse should be increased.

2. If two neurons on either side of a synapse are activated asynchronously, then the synapse should be weakened or eliminated.

Hebbian learning is analogous to long-term potentiation (LTP) and long-term depression (LTD) that occurs in the brain [202]. In LTP, if the firing of one neuron occurs before the firing of a receiving neuron, then the synapse between these two is strengthened. That is, in LTP, the possibility of a causal relationship between the two neurons (i.e., that the firing of one directly leads to the firing of another), influences how synaptic changes are made. In LTD, the strength of the synapse is decreased when the firing of one neuron does not lead to the firing of its connected neurons. In LTD, the possibility of a non-causal relationship between the two neurons influences how synaptic changes are made. For example, if a receiving neuron fired immediately prior to the firing of a transmitting neuron, it may be appropriate to decrease the strength of the synapse [202].

There are four characteristics of Hebbian synapses:

1. Time-dependency: Modifications to a Hebbian synapse depend heavily on time in that increases are made if neurons are activated at the same time, and decreases are made if two neurons are activated at different times.

2. Localness: All information required to determine if a change to a Hebbian synapse should be made is local information. That is, the only information

required to know if a synapse should change is the activity of the neurons that are connected by that synapse.

3. Interactiveness: Changes in the weight of a Hebbian synapse are determined by the firing patterns of the two neurons connected by the weight.

4. Conjunctional or correlational synapse: An increase in the strength of the synapse is caused by the conjunction of presynaptic and postsynaptic activity.

Hebbian learning has been observed in biological neural networks [192]. However, as noted in Appendix A.4, learning in biological systems is significantly more complicated than implied by these four characteristics.

Another type of learning in neural networks is Boltzmann learning [112]. In Boltzmann learning, each neuron is binary; that is, each neuron in a Boltzmann neural network (also called a Boltzmann machine) is either on or off. The machine then flips a neuron at random during the learning process, with some probability that is dependent on the "energy" of the machine, where energy is a function of the current values of each neuron and the weights between them. A specific type of Boltzmann machines, restricted Boltzmann machines, is often used in deep machine learning [28].

B.4 A Brief History of Artificial Neural Networks

Much of the work on artificial neural networks stems from McCulloch and Pitts [161]. In this paper, the authors described the relationships between biological neural network events using propositional logic, showing that for any logical expression satisfying certain conditions, there is a net that behaves in the way the logical expression describes.

Hebb described a physiological learning rule for synapses, as well as his postulate of learning, as described in Appendix B.3 in [113]. He proposed that the connectivity of the brain is changed as an organism learns new tasks.

Minsky's dissertation [169] discussed what are now known as artificial neural networks. In [171], Minsky discussed what was then the state of the art in artificial intelligence, including a discussion on artificial neural networks and their limitations.

In [213], Rosenblatt introduced the perceptron, the simplest form of a neural network used for the classification of linearly separable patterns. This work also introduced a learning algorithm for the perceptron. In [214], Rosenblatt proved the perceptron convergence theorem, which states that if there is a set of weights for the network such that the network correctly classifies the training patterns, then the learning algorithm will find one such weight set in a finite number of iterations.

In [170], Minsky and Papert discussed the limitations of perceptrons, specifically discussing the limitations of one layer perceptron networks and extending these limitations to multi-layer networks. The downturn of work on artificial neural networks in the 1970's is partially attributed to this work [112]. Another contributing factor to the decline in research on artificial neural networks was DARPA's (then ARPA) change in funding requirements in 1969. With the Mansfield Amendment, all DARPA-funded projects had to demonstrate immediate utility to a military mission. This led to a decline in not only artificial neural network work, but artificial intelligence work in general [185].

In [115], Hopfield introduced Hopfield networks, which are recurrent networks. Hopfield established an isomorphism between these types of networks and the Ising model in physics. This led to a significant amount of work on neural networks in physics. However, as noted in [112], these models are likely not a realistic model of neurobiological behaviors. This is one of the places in history in which artificial neural networks diverged from biological systems.

Ackley *et al.* introduced Boltzmann learning for neural networks in [3]. This algorithm exploited simulated annealing, which is briefly discussed in Appendix C, in the development of a learning algorithm for neural networks. The back-propagation algorithm as described above was introduced by Rumelhart *et al.* in [216].

As neural network research developed, there was a significant divergence from the original biological inspiration, with the development of Hopfield networks and algorithms such as back-propagation and Boltzmann learning. An alternative to the use of the algorithms described above for training is to use evolutionary algorithms for training the network. This is arguably a more biological approach, as biological brains evolved over time to become the complex structures they are. A discussion of evolutionary algorithms is given in Appendix C and a specific discussion of these algorithms as applied to neural networks is given in Section 2.1.

Appendix C

Evolutionary Algorithms

The evolution of the structure of the brain and evolution within the brain can be categorized in four forms.

First, at the highest level, there is evolution via speciation, or evolution over the brain structure, which has occurred over millions of years. This **long-term evolution** has affected every aspect of the brain, but most notably, it is the level of evolution where the gross structure of the brain has developed. Following typical evolutionary theory, the complex structures from the human brain evolved from simpler structures that underwent three evolutionary mechanisms: mutation, the introduction of new structures or pieces of structures; recombination, the combination or re-use of existing structures in novel ways; and natural selection, the dying off of unsuccessful structures.

The general structure of the brain does not differ greatly from person to person; there are certain parts of the brain that are present in nearly every individual, though as the evolution of species has occurred these structures have become more complex. These are the types of structures that are of concern at the level of **long-term evolution**.

A shorter term evolution of the brain, what will be referred to in this work as **moderate-term evolution**, has been recently discovered. This evolution, referred

to as epigenesis, also affects the structure of the brain, but at a finer level. Epigenesis is caused by modifications to the structure of proteins that regulate the transcription of genes; these modifications are often caused by the environment, but unlike other environmental effects, these modifications can be inherited by future generations through methylation of DNA. The modifications can lead to changes in the structure of the brain and thus far, have been seen to primarily affect the social and affective aspects of the brain [134].

The evolution (or perhaps more aptly, development and adaptation) that occurs within a single human's brain over the course of a lifetime, from conception through adulthood, will be referred to in this work as **short-term evolution**. The morphology of the brain is shaped partly through genetics, influenced by both long-term and moderate-term evolution, but also through experience (or by environmental effects). Neurons proliferate and die over the course of an individual's development. One of the factors that affects the formation and survival of neurons in this stage is the way connections are formed, that is, the types of neurons that a particular neuron's axon connects during development [238]. The connections of a neuron affect the way that neuron behaves and operates in the future, and these connections are initially determined during this short-term evolutionary stage. An example of this type of evolution is found in London taxi drivers (see Appendix A.4 and [153]).

There is a certain amount of plasticity during development that allow an individual to adapt the different parts of the brain (determined by long-term evolution) to his or her particular role. There are certain portions of the brain, such as the neocortex, in which the local structure (i.e., connection strengths) appears to mostly depend on the environment, rather than genetics [96].

Another major structural aspect of the brain that is evolved or developed over the course of single person's lifetime is myelination. Myelination, as discussed in Appendix A.1, affects the efficiency and rapidity of transmissions of signals in the brain. Myelination continues well into the second decade of life [238].

Finally, **very short term evolution** (development or learning, in this case) occurs on a day-to-day basis in the brain. This evolution affects synapses; this type of evolution is what is typically referred to as plasticity in the brain. There are four known major types of synaptic plasticity: long-term potentiation, long-term depression, sensitization, and axonal sprouting and formation of new synapses. Long-term potentiation and long-term depression were discussed briefly in Appendix A.4 within the context of Hebb's rule. Long-term potentiation is a permanent change in the way a neuron fires and is caused by repeated activation with stimulation; it is associated with memory in the brain [188]. Long-term depression refers to any form of depression in synaptic transmission, such as the lowering of signal transmission efficacy [119]. Long-term potentiation occurs only when a synapse is active, but long-term depression can occur whether a synapse is active or inactive [13].

Sensitization refers to enhancement of a response as a result of applying a novel stimulus [48]. Finally, axons can sprout (grow new connections), both during initial formation and after transection, in the brain. Axon sprouting occurs most commonly during neonatal development, but it also can occur in adulthood [253].

Evolutionary algorithms are optimization algorithms that are often used in large, complex state spaces. As pointed out by Mitchell in [172], biological evolution is a method for searching a huge number of possibilities for solutions, where solutions are the organisms themselves. The biological inspiration of evolutionary algorithms is described in [76] as follows:

$$Adaptation = Variation + Selection + Heredity \tag{C.1}$$

In evolutionary algorithms, a population of potential solutions is maintained. The members of the population are usually distinct and maintain variety. Evolutionary algorithms are inherently random, and the random influences contribute to the variety in the population. Selection is perhaps the most important component of the formula given above. Selection refers to the concept of "survival of the fittest." For

evolutionary algorithms, some concept of fitness must exist, where fitness is typically a function or algorithm mapping members of the population to numerical values. It is worth noting that the fitness function can be based on simulated values, so it may generate a different value each time it is applied to a member of the population. The fitness of a member of a population should represent the relative ability of that member of the population to perform a particular task. The most fit members of the population are those that are selected to reproduce and express traits that are kept over multiple generations. Members of the population that are the least fit are those that are allowed to die off.

Heredity is emulated in evolutionary algorithms by producing "offspring" from existing members of a population. The offspring can be produced in a variety of algorithm-specific ways. The sequence of typical operations for producing offspring, as discussed in [100], are:

- Reproduction
- Crossover
- Mutation

For reproduction, one or more relatively fit members of the population are chosen to reproduce. Members of the population that have a higher fitness level are more likely to have offspring in the next generation of the population. The selection of these members of the population can be done in a variety of ways. One of the ways this is done is using Roulette selection. In Roulette selection, a member of the population is randomly selected, where the probability that a given member of the population is selected is based on that population member's fitness. That is, if a member has a high fitness, it is more likely to be selected. Another selection algorithm is tournament selection. In tournament selection, a fixed percentage of the population is randomly selected. From that smaller group, the member with the highest fitness is selected. The percentage selected from the original population is a parameter of this method

[138]. For example, if this parameter is set to be 100 percent, then the most fit member of the population would always be selected. However, if the population size is 100 and the parameter was set as one percent of the population, then the selection would be entirely random (i.e., not based on fitness at all).

In crossover, attributes of two or more members of the population are combined to form a new member of the population. Different forms of crossover allow for multiple ways in which this occurs, and how the children are formed is highly dependent on the representation of the members of the population. Finally, mutation can occur, in which some attribute of the new member is randomly changed in some way. Different types of mutations can be employed, depending upon the complexity of the representation of each member of the population. Both crossover and mutation have associated rates in an evolutionary algorithm. The crossover rate is the percentage of time in which selected members of the parent population are crossed over to produce members of the child population, whereas the mutation rate is the rate at which members of the parent population are mutated to produce members of the child population. Assuming neither of these rates is 1 (in which case both mutation and crossover would be forced to occur at each step), there is some propagation of identical members of the parent population to the child population.

Four basic types of evolutionary algorithms have been discussed in the literature: genetic algorithms, genetic programming, evolutionary programming, and evolutionary strategies [76, 100]. A brief description of each of those algorithms is given below:

- Genetic algorithms: In genetic algorithms, populations are composed of strings. Genetic algorithms use crossover, mutation, and selection to obtain offspring. They efficiently exploit historical information to speculate on new search points.

- Genetic programming: Genetic programming techniques are similar to genetic algorithms, but differ in that program fragments are used instead strings. For example, a parse tree of a program may be used. If the program is represented as a string, then genetic algorithm methods can be used. In genetic programming

methods using other representations, such as parse trees, crossover is still possible. The programming language Lisp and variants of Lisp have been used for genetic programming operations. It is important to note that these types of structures can be represented as strings, so in this sense, genetic algorithms and genetic programming are the same.

- Evolutionary programming: Evolutionary programming techniques differ from genetic algorithms in three ways: representation of solution in evolutionary programming follows directly from the problem and is not constrained to be in the form of a string of characters, as in genetic algorithms. Evolutionary programming techniques do not attempt to closely model the genetic operations of living organisms, so crossover is not used in evolutionary programming. Mutations usually take a different form in evolutionary programs, typically dependent on the structure of the solutions. Finally, the rate of mutations is typically reduced as the optimal stage is approached with evolutionary programs.

- Evolutionary strategies: Evolutionary strategies are similar to evolutionary programming techniques, but differ slightly in the way selections are made and how mutations are performed. Recombination is also possible in evolutionary strategies. Evolutionary strategies and evolutionary programming are similar in spirit to simulated annealing.

All of these algorithms are essentially equivalent. The main difference is the representation of the members of the population, which may depend on the application at hand. Evolutionary algorithms have three essential characteristics: the representation of the members of the population, the types of operations performed on the population, and the number of members of the population. They can be categorized using any of these three mechanisms. Simulated annealing [136], for example, is an evolutionary algorithm in which the population size is one, while genetic algorithms have larger populations. Some evolutionary algorithms only use

mutation operations to produce new members of the population, while others use crossover operations as well.

One aspect we are interested in exploring is the instance of an evolutionary algorithm in which there are multiple (interacting) subproblems that are each optimized with respect to their own fitness function. Specifically, since our populations will be made of up networks, we are interested in exploring evolutionary algorithms that will optimize multiple interacting networks simultaneously. For example, one population of networks may represent limbic system type networks, while another population of networks would represent computational, cortical type networks. In this case the two types of networks should interact by exchanging information. The evolutionary algorithm must be able to deal with these interacting, simultaneous optimizations. See Chapter 5 for more details.

C.1 A Brief History of Evolutionary Algorithms and Related Methods

Evolutionary methods are types of optimization algorithms that attempt to explore a large state space in a somewhat random way, as opposed to gradient-based or more general descent-based algorithms, which explore the state space in a fixed and local way. There are several other types of optimization algorithms that are similar to evolutionary methods.

In [165], Metropolis *et al.* described the Metropolis algorithm, which is a Markov Chain Monte Carlo method. The Metropolis algorithm starts at an arbitrary point in the search space and generates a sequence of points in the search space, where each point depends only on the previous point. Whether a jump is made from point to point is dependent on the acceptance probability. This algorithm is an evolutionary algorithm with a population size of one. In some cases, this algorithm is applied to many populations of size one, but there is no interaction between the populations.

Simulated annealing is a similar method, first described by Kirkpatrick *et al.* in [136], in which the Metropolis algorithm is used to choose points in the space, but the acceptance probability is decreased in order to improve the convergence rate of the algorithm. Simulated annealing methods consider only one solution at a time whereas evolutionary algorithms may maintain a population of solutions. Simulated annealing uses only a form of mutation, while evolutionary algorithms may use mutation, selection, and crossover to produce new solutions.

Evolutionary strategies were first discussed by Rechenberg in [208]. He went on to describe them in [209]. Schwefel expounded on Rechenberg's ideas in [231, 232]. Genetic algorithms were described by Holland in [114]. In this work, Holland described a genetic algorithm in which the chromosomes (the members of the population) were binary strings. The operations he used to form new chromosomes were crossover, mutation, and inversion, in which a section of the chromosome is reordered.

Appendix D

Reinforcement Learning

A third subject of machine learning that is relevant to this topic is reinforcement learning. In reinforcement learning, an agent is required to learn a task. Unlike supervised learning, the agent is not told what to do or what not to do; the agent must discover how to behave on its own. That is, there is no training set in which there are inputs and associated desired outputs that the agent is required to learn. In reinforcement learning, the agent is guided by rewards. If a behavior is deemed good, the agent is rewarded. However, the agent usually has no concept of how much reward is possible in a given environment; typically, the agent's only knowledge is what it has discovered so far.

The goal in reinforcement learning tasks is to maximize the rewards received over time in a task. Reinforcement learning methods require a balance in exploration and exploitation. In other words, an agent wants to exploit its current knowledge to obtain rewards while simultaneously exploring the environment to discover if there is a greater source of reward in a part of the environment the agent has not yet discovered. How the tradeoff between exploration and exploitation is managed is often a major issue in reinforcement learning methods. This type of tradeoff between exploration and exploitation traces back to the work on dual control. The same issues arise in dual

control and are discussed in Athans and Gershwin's work on the uncertainty principle [15] and in Witsenhausen's work on the separation of estimation and control [268].

Reinforcement learning problems can be divided into two groups. Some tasks are naturally divided into episodes (e.g., trips through a maze). These tasks are called episodic tasks. Other tasks, continuing tasks, have no clear ending (e.g., many control theory applications). Some reinforcement learning methods require tasks to be episodic; however, most continuing tasks can be adapted to be episodic tasks. Some tasks (e.g., the pole balancing task) can be formulated as either episodic tasks or continuing tasks.

Reinforcement learning problems are characterized by an agent's interaction with its environment. An agent's actions affect the environment, and changes in the environment affect the agent's actions. The environment provides a reward or punishment to the agent for each action taken. The reward received is characterized by the *reward function*, which maps a state or a state-action pair to the associated reward. Typically, a *value function* is used to determine what action should be taken. The value function depends on the state of the agent in the environment and defines, in general terms, how much reward the agent can expect from that state onward. A variant of the value function, the *action-value function*, defines, in general terms, how much reward an agent can expect when taking a particular action in a given state. The interaction between an agent and its environment is depicted in Figure D.1. Over the course of learning, the agent develops a *policy*. The policy determines the action an agent will take based on its current state. In these terms, the goal of reinforcement learning tasks is to develop a policy that will maximize the rewards received [248].

In [248], Sutton and Barto classify optimal control as a reinforcement learning topic. In this context, optimal control refers to any problem in which a controller is designed to optimally solve a problem. The optimization is usually characterized by the minimization or maximization of some measure of the system's behavior over time. The Bellman equation, defined by Richard Bellman in the 1950s, is

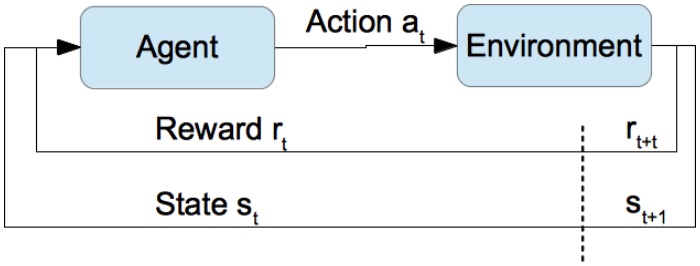


Figure D.1: An agent decides its action based on the current state of the system. An action of the agent affects both the reward for the agent and the state of the environment.

an equation that maps the current state and input to expected reward given that after the current input, an optimal policy will be used. The methods used to solve this type of equation by determining an optimal policy for the agent to follow are called dynamic programming methods. Approximate dynamic programming methods iterate approximations of the policy and value functions in order to determine the near optimal or optimal policy for an environment. Each policy approximation relies on previous approximations of the value function, and approximations of the value function rely on previous approximations of the policy. Sutton and Barto call this reliance on earlier approximations *boot-strapping* [248].

Nearly every dynamic programming method requires complete knowledge about every aspect of the system, including the environment. In reality, this knowledge is not likely to be known. In [32], Bertsekas notes that the dynamic programming methods that do not require complete knowledge of the environment are significantly more computationally intensive than methods that require perfect state knowledge. Methods for overcoming a lack of knowledge about the environment include a reduction of the system to one with perfect state knowledge and using sufficient statistics to describe the environment.

Perhaps the most significant issue associated with dynamic programming methods is the *curse of dimensionality*. In general, the curse of dimensionality refers to methods in which computation time increases exponentially as the size of the problem increases. As computation time is often an important factor in problem solving,

approximation methods are required. These types of methods trade an optimal policy for a suboptimal policy that can be obtained in a reasonable amount of time. In [32], Bertsekas notes several methods that can be used to obtain suboptimal policies in control problems. Those methods include certainty equivalence controllers, open-loop feedback controllers, limited lookahead policies, and rollout algorithms. Suboptimal control methods approximate the value function of the current state. Methods differ based on how the approximation takes place. Bertsekas and Tsitsiklis explore the use of neural networks to approximate the value function in [33]; they refer to these methods as neuro-dynamic programming. There are close parallels between the methods of reinforcement learning [248] and approximate dynamic programming [32, 33].

One group of methods that solve reinforcement learning problems without complete knowledge of the environment is the Monte Carlo methods. Monte Carlo methods rely on experience rather than knowledge. These methods learn about the environment over the course of *sample episodes*. Monte Carlo methods can learn via direct interaction with the environment (on-line experience) or through a simulated environment. Simulations of the environment can often be done without complete knowledge of the environment, and simulation results can often be obtained more quickly than on-line experience. Sutton and Barto note that one of the advantages of Monte Carlo methods is that they can be focused on a relevant subset of states, reducing the exploration required in other methods [248]. This is also a disadvantage of Monte-Carlo methods, because they will not sufficiently explore the environment on their own. In order to force the methods to explore properly, random starting conditions in the environment and/or random perturbations (or noise) in the exploration should be used. Unlike dynamic programming methods, Monte Carlo methods do not rely on previous estimates to obtain new estimates; that is, Monte Carlo methods do not bootstrap.

Perhaps the most important methods in reinforcement learning are temporal difference methods. Sutton and Barto note that temporal difference methods combine

the experience-based learning of Monte Carlo methods with the bootstrapping of dynamic programming methods. They assert that in practice temporal difference methods converge faster than Monte Carlo methods on stochastic tasks; however, one method has not been proved to converge faster than the other [248]. Temporal difference methods update the value function based on a particular policy and experience using that policy. The simplest version of temporal difference learning is called TD(0). It uses information from time $t+1$ to update the value information about time step t, as does dynamic programming. Monte-Carlo methods, on the other hand, usually work on information from an entire episode, not just a single time step.

SARSA is one temporal difference method. SARSA is so called because the updates to the action-value function made during learning require information about the state at time t (S), the action at time t (A), the reward at time t (R), the state at time $t+1$ (S), and the action at time $t+1$ (A). SARSA estimates the action-value function for the current policy and updates the policy based on the action-value function. SARSA is an on-policy algorithm, which means that it makes updates based on the policy it is using to determine the agent's behavior. Another temporal difference method, Q-learning, is an off-policy algorithm, which means that updates to the action-value function estimate are made based on the optimal policy, rather than the policy being followed [248].

Temporal difference algorithms have been adapted to make use of neural networks. In these applications, just as in neuro-dynamic programming, neural networks are used to approximate the action-value function (or the cost-to-go function in dynamic programming). The neural networks are typically trained using back-propagation. The error used in the back-propagation algorithm is the error determined by the Bellman equation used. Since the Bellman equation used is slightly different for different methods (e.g., SARSA or Q-learning), different errors can be used to train the network. Methods perform differently for different problems, so it is usually difficult to determine which method is best suited for a problem *a priori*. Similarly, there are several parameters for each of the different methods, and all of

the parameters are known to have an effect on the convergence rate. The user is often left to to experimentally determine the proper parameters for these methods, and the appropriate parameter values are typically specific to the application.

Appendix E

Theoretical Results

This section provides some theoretical results related to NIDA networks. These results (and similar results as the theory continues to be developed) may be of use in developing heuristics for exploring possible network structures and parameter values over the course of the evolutionary optimization.

Suppose there is a grid that has the ability to accommodate N neurons (and thus N^2 synapses). Let X be the set of all binary strings of length $N+N^2$ such that the i^{th} element in the binary string for $i = 1, ..., N$ corresponds to the i^{th} neuron in the grid and the $j + N^{th}$ element corresponds to the j^{th} possible synapse in the grid. Then, each string x represents a fixed network structure. The binary representation does not take into account the parameters of the network (i.e., the weights of the synapses and the thresholds of the neurons). Note that some binary strings may have synapses that are "on" but cannot exist in the structure because one or both of its neurons is not represented in the network. Then, the structure will not include the synapse. Thus, there are many potential binary strings that represent the same structure. We can let $[x]$ represent the string with the minimum number of 1's that uniquely represents the network structure x. Thus, $[x]$ represents the class of strings that represents the same network structure.

Let P_x be the set of all possible parameter values for a network structure represented by the binary string x. There is a finite (though large) set of possible values for both thresholds and weights because the parameters are bounded (e.g., between -1 and 1) and the set of possible values is restricted by the precision of a given machine. A complete network is defined then by x and p where $x \in X$ and $p \in P_x$.

Let f be the fitness function for a particular task. f takes a complete network and gives a corresponding fitness score.

Now, define $f^* : X \to \mathbb{R}$ as follows:

$$f^*(x) = \max_{p \in P_x} f(x, p) \tag{E.1}$$

Thus, $f^*(x)$ is the maximum possible fitness score attainable by the structure defined by x.

Let $x, y \in X$ and define $\leq$ as follows: $x \leq y$ if $x_i = 1 \implies y_i = 1$.

Theorem E.1. $(X, \leq)$ *is a lattice.*

Proof. Note that a lattice has the following properties:

1. $x \leq x$
2. If $x \leq y$ and $y \leq x$, then $x = y$.
3. If $x \leq y$ and $y \leq z$, then $x \leq z$.
4. x and y have a unique least upper bound, $x \vee y$.
5. x and y have a unique greatest lower bound, $x \wedge y$.

First note that $x_i = 1 \implies y_i = 1 \forall i$ is equivalent to $x_i \leq y_i \forall i$. This is because $x_i, y_i \in \{0, 1\}$. $x_i = 0 \implies y_i$ may be either 0 or 1, but $x_i = 1 \implies y_i = 1$. Thus, $x_i \leq y_i$.

Proof(1): Let $x \in X$. It is trivial that $x_i = 1 \implies x_i = 1$, so $x \leq x$ by definition.

Proof(2): Let $x, y \in X$ and suppose that $x \leq y$ and $y \leq x$. Note that $x_i \in \{0, 1\}$ and $y_i \in \{0, 1\}$ $\forall i \in \{1, ..., N + N^2\}$. Thus, $x \leq y \implies x_i \leq y_i \forall i \in \{1, ..., N + N^2\}$ where $\leq$ is the operator on $\mathbb{R}$. Similarly, $y \leq x \implies y_i \leq x_i \forall i \in \{1, ..., N + N^2\}$. Thus, $x_i \leq y_i \forall i \in \{1, ..., N + N^2\}$ and $y_i \leq x_i \forall i \in \{1, ..., N + N^2\} \implies x_i = y_i \forall i \in \{1, ..., N + N^2\}$. Thus, $x = y$.

Proof(3): Let $x, y, z \in X$ such that $x \leq y$ and $y \leq z$. Then, $x_i \leq y_i$ and $y_i \leq z_i$. By transitive property of real numbers, $x_i \leq z_i \implies x \leq z$.

Proof(4): Let $x \vee y$ be defined such that $(x \vee y)_i = x_i \vee y_i$ where $\vee$ is the binary operator OR.

Proof(5): Let $x \wedge y$ be defined such that $(x \wedge y)_i = x_i \wedge y_i$ where $\wedge$ is the binary operator AND.

Since binary operator OR and binary operator AND are idempotent, commutative, and associative, and satisfy the absorption law, so do the $\vee$ and $\wedge$ operators on elements from X.

Note that $x \wedge y = x$ implies that $(x_i \wedge y_i) = x_i$. Thus, $x_i = 1 \implies x_i \wedge y_i = 1 \implies y_i = 1 \implies x \leq y$.

Thus, $(X, \leq)$ defines a lattice.

□

Theorem E.2. *Let x be in X so that there exists i such that $x_i = 0$. Let $\tilde{x}$ be such that $\tilde{x_j} = x_j$ for $j \neq i$ and $\tilde{x_i} = 1$. Then $f^*(x) \leq f^*(\tilde{x})$.*

Proof. Note that $\tilde{x}$ is identical to x except that either a neuron or a synapse is added to the structure. If a neuron is added, it is possible that some synapses going to that neuron and from that neuron are now legal and are also added to the structure.

Suppose i is such that setting $\tilde{x_i} = 1$ adds a synapse to the network represented by x. Then, there exists $p' \in P_{\tilde{x}}$ such that the weight of the synapse added to $\tilde{x}$ is 0. Thus, there exists $p' \in P_{\tilde{x}}$ such that $f(\tilde{x}, p') = f^*(x)$. By definition, $f(\tilde{x}, p') \leq f^*(\tilde{x})$. Then:

$$f^*(x) = f(\tilde{x}, p') \leq f^*(\tilde{x}) \tag{E.2}$$

Suppose i is such that setting $\tilde{x}_i = 1$ adds a neuron to the network. Then, there exists $p'' \in P_{\tilde{x}}$ such that all synapses exiting neuron i have weight 0. Thus, there exists $p'' \in P_{\tilde{x}}$ such that $f(\tilde{x}, p'') = f^*(x)$. By definition, $f(\tilde{x}, p'') \leq f^*(\tilde{x})$. Then:

$$f^*(x) = f(\tilde{x}, p'') \leq f^*(\tilde{x}) \tag{E.3}$$

In both cases, $f^*(x) \leq f^*(\tilde{x})$.

□

Theorem E.3. *Let $x, y \in X$ such that $x \leq y$. Then, $f^*(x) \leq f^*(y)$.*

Proof. $x \leq y \implies \exists J \subseteq \mathbb{N}$ such that $x_j = 0$ and $y_j = 1$ for all $j \in J$. If $J = \varnothing$, then $x = y$ and $f^*(x) = f^*(y)$.

If $|J| = 1$, then this falls into the case of Theorem 2, so, $f^*(x) \leq f^*(y)$.

Otherwise, $|J| \geq 2$. Then, construct a sequence of binary strings in X as follows. $\{x^i\}_{i=0}^{|J|}$, where $x^0 = x$ and $x^{|J|} = y$. Construct an ordering on J, $\{j_i\}_{i=1}^{|J|}$. Then, for $j = 1, ..., |J|$, let:

$$x_k^i = \begin{cases} x_k^{i-1} & : k \neq j_i \\ 1 & : k = j_i \end{cases} \tag{E.4}$$

So, $\{x^i\}_{i=0}^{|J|}$ is a sequence of binary strings in X that begins at x and each subsequent string in the sequence is the same as the one before, except with one 0 in the binary string changed to a 1, in a place where y has a 1 but x does not. By Theorem 2, $f^*(x^i) \leq f^*(x^{i+1})$ for $i = 0, ..., |J| - 1$.

$$f^*(x) = f^*(x^0) \leq f^*(x^1) \leq ... \leq f^*(x^{|J|-1}) \leq f^*(x^{|J|}) = f^*(y) \tag{E.5}$$

□

Every fitness calculation we make for a network specified by (x, p), we are determining a lower bound for $f^*(x)$ and a lower bound for $f^*(y) \forall y \in X$ such that $x \leq y$. We may be able to exploit these properties in order to eliminate a

portion of the search space in which we are attempting to find structures that solve problems. For example, suppose we have (x_1, p_1) and (x_2, p_2) such that $x_1 \leq x_2$ and $f(x_1, p_1) \leq f(x_2, p_2)$. If we were able to determine that $f(x_1, p_1) = f^*(x_1)$, then we could eliminate all searches with structures y such that $y \leq x$, because there is no $y \leq x$ such that $f^*(y) > f^*(x)$. This is a theoretical way in which a portion of the search space may be eliminated. By further developing similar theory and exploring the properties of the network space and parameter space, we may be able to further restrict our search space and improve the evolutionary optimization.

Vita

Catherine (Katie) Schuman was born in Harriman, Tennessee, to John and Sherry Schuman. She is the youngest of three children; her siblings are Tracey Lawson and Alan Oliver. Katie chose to study computer science after completing an internship at Scripps Networks during the summer between her junior and senior year of high school. During her time as an undergrad, Katie worked as an undergraduate research assistant with Dr. Jim Plank, studying erasure coding, which inspired her to pursue a graduate degree in computer science. She earned her Bachelor of Science degree with majors in Computer Science and Mathematics from the University of Tennessee in May 2010. Katie joined the doctoral program in computer science at the University of Tennessee in August 2010. Katie worked as a research assistant under Dr. Doug Birdwell and Dr. Mike Langston, and also served as a graduate teaching assistant for Dr. Jim Plank. Katie received the National Science Foundation Graduate Research Fellowship in 2012. Katie graduated with her Doctorate of Philosophy in Computer Science in May 2015.