Topology Modification and Training for Spiking Neural Networks

A Dissertation Presented for the

Doctor of Philosophy

Degree

The University of Tennessee, Knoxville

Nicholas Dane Skuda

December 2024

Abstract

Spiking Neural Networks are a promising field with potential for performing machine learning tasks using hardware with very low SWaP costs. These systems have an inherently temporal nature using event-based information transfer with a dynamic state, making Spiking Neural Networks well suited for dynamic, temporal tasks. However, training a spiking neural network to take advantage of the capabilities of such a system is challenging. Existing Evolutionary Optimization algorithms work well to solve small problems very efficiently but struggle to learn larger problems. This work begins by introducing a network compilation process to assemble these small networks into a more complex whole to potentially operate for more complex problems. Further, this work introduces a means of reducing the size of assembled networks through a process called time multiplexing. Finally a modified version of the EONS algorithm is introduced to operate on problems with larger input spaces by creating more densely-connected networks.

Table of Contents

1 Introduction **1**

1.1 Statement of the Problem 1

1.2 What is a Neural Network 2

1.3 What is a Spiking Neural Network 3

1.4 Purpose 4

2 Literature Review **6**

2.1 Neuromorphic Computing and Spiking Neural Networks 6

2.2 TENNLab Exploratory Neuromorphic Computing Framework 8

2.2.1 Neuroprocessors 9

2.3 Datasets 10

3 Genetic Algorithms and Neuromorphic Computing **11**

4 Network Compiler **13**

4.1 Network Compiler Example 15

4.2 Compiler Instructions 19

4.3 Pre-processing 27

4.4 Network Instance Generation 30

4.5 Inter-network Synapse Generation 31

5 Compiler-Assisted EONS **33**

5.1 Compiler-Assisted EONS Simple Demonstration 34

5.2 Evaluation 36
5.2.1 Asteroids 37
5.2.2 Space Invaders 39
5.2.3 MNIST 44
5.3 Larger Subpopulation Test 46
5.4 Future Work 48

6 Time Multiplexing **51**
6.1 Building Time Delay 53
6.2 Time Compressing Outputs 55

7 ConnectEONS **60**

8 Lessons Learned **71**

9 Conclusion **75**

Appendices **82**

A Path Generation **82**

B Simple Network Generation **88**

C Clustering Network Generation **89**

D Counting Network Generation **100**

E Network Compiler **111**

Vita **158**

List of Tables

4.1 Network Compiler Instruction Format 22

6.1 Time Multiplexing example network sizes 54

7.1 MNIST training run data 67

7.2 Initial ConnectEONS Hyperparameters 69

7.3 Hyperparameter Grid Search Configuration Members 69

List of Figures

4.1 A simple example of the network compiler with two subnetworks for an application requiring 8 inputs and 3 outputs. 14

4.2 A simple compiled network that detects if there were 4 or more fires on either the left or right half of the inputs (top row). 16

4.3 The 3 subnetworks for the compilation example including: a) A simple 12-neuron input layer b) A 6-input counting network detecting 4 inputs over 20 cycles with a 5-cycle reset c) RISP xor network 18

4.4 The process of creating inter-subnetwork connections for the example's first instruction (a) and second instruction (b) 20

4.5 The compiler's instructions (a) can connect multiple differents input subnetworks together in by either spanning inputs (b) or overlapping into the same instance with the input mixing flag (c) 22

4.6 A 3x3 spiking counting network, courtesy Charlie Rizzo [1] 24

4.7 Example of network compilation to an input set to a 3x4 input grid . 26

5.1 A example of compiler-assisted EONS for a binary classification problem with fixed subpopulations (a) and the demonstration of how evaluation networks are generated (b) as well as scoring for the compiler EONS example, demonstrating how scores are determined for the evaluation networks and the max scores for individual subnetworks (c) 35

5.2 The application visualiation for the TENNLab Asteroids application (a) and fitness scores for the EONS runs with and without compiler-assistance (b) 38
5.3 The application visualiation for the TENNLab Space Invaders application (a) and compiler instructions for space invaders training (b) . . 40
5.4 SpaceInvaders Score 43
5.5 MNIST compiled configuration 45
5.6 The training fitness (a) and speed (b) for EONS with and without compiler-assistance 47
5.7 The training fitness (a) and speed (b) for compiler assisted EONS on larger-population asteroids runs 49

6.1 Compiling 3 networks using instances versus time-multiplexing 52
6.2 Compiling with (right) and without (left) time multiplexing with simple time compression for periods of 5 (top) or 20 (bottom) cycles for an XOR of counting networks compilation task. 54
6.3 Time Compression support structures to allow for more flexible output options and better time restrictions. The time multiplexed network is in red, a single instance of the time compressed output is in blue with its support neurons in green. 6.3a shows a network which relies on high maximum allowed thresholds and 6.3b shows a network requiring high refractory periods and delays. 57

7.1 An example of a connection pathway with both a merge from the bottom input and split to the bottom output, from inputs (left) to outputs (right). 61
7.2 a) Demonstration of adding features while creating a connection path b) ConnectionEONS traceback operations using geometric distribution to select a neuron to return c) The merge and split operations which occur during path creation d) ConnectEONS path mutation operations 63

7.3 Results for Space Invaders (left) and Asteroids (right) for EONS, ConnectEONS, and Compiler assistance. For the Asteroids results, the ConnectEONS results are the right for each configuration. 65
7.4 Results for MNIST using EONS, ConnectEONS, and Compiler assistance, a) training f1 scores b) top ConnectEONS test confusion matrix c) training times 67
7.5 ConnectEONS Hyperparameter exploration asteroids score and training times. 69
7.5 (continued) Results for ConnectEONS Asteroids grid search 70

Nomenclature

CNN	Convolutional Neural Network
ConnectEONS	Connection-driven Evolutionary Optimization for Neuromorphic Systems, a training algorithm proposed in this work
DNN	Deep Neural Network
EONS	Evolutionary Optimization for Neuromorphic Systems, a training algorithm for SNN's [2]
GA	Genetic Algorithm
IAF	Integrate And Fire, a simple, fast spiking neuronal model
LIF	Leaky Integrate and Fire, another simple, fast spiking neuronal model where neurons lose charge over time
MOA	Mother of All (... random number generators), a flexible psuedo-random number generator used frequently in the TENNLab Framework
run	One execution of a genetic algorithm, training a population of SNN's for a set time or number of epochs
SNN	Spiking Neural Network
TENNLab	Tennesseans Enganged in Neural Networks Labratory, an SNN-focused research group at the University of Tennessee, Knoxville

Chapter 1

Introduction

1.1 Statement of the Problem

With the end of Moore's law and Denard scaling, non-standard computing architectures with low Size Weight and Power (SWaP) needs represent an intriguing future [3]. In addition, as machine learning becomes increasingly popular, the desire for machine learning accelerators and novel machine learning techniques is increasingly important [4]. Spiking neural networks (SNN) are a promising machine learning model with great potential for low SWaP needs, however training SNNs is challenging [5]. Simulating an SNN can be slow due to their time-based mechanisms and dynamic state. Furthermore the extra parameters, compared to simpler DNNs, leads to a larger search space in training the same size of network. While it has been claimed that SNNs have supra-Turing Machine capabilities, large but simple feed-forward networks have not been demonstrated to exhibit this behavior [6]. Small, sparse, recurrent spiking networks have been shown to be capable of solving problems equal to that of a much larger deep or convolutional neural network [7]. However allowing for this kind of topological freedom has extreme costs in terms of training time. Slight changes in network topology can cause some or all of the parameters to have a different "meaning" in the resulting function approximation. This kind of training has

traditionally been performed using genetic algorithms, such as EONS[2]. By casting both small-scale topological and parameter changes both as mutation operations, a single algorithm can both search for a usable network topology and converge towards an ideal parameter set. Unfortunately the power to find all possible topologies within the constraints of the learning process also means that both the topological and parameter search spaces become unrealistically large for warehouse scale computing systems to explore fairly modest network sizes. Spiking neural networks use labeled, directed graphs, so just in terms of inter-neuron synapse existence, the number of edge topologies is a permutation problem with 2^{n^2-n} possibilities for n neurons.

1.2 What is a Neural Network

A neural network is a concept based on observations of animal brains which are organized into networks of neurons connected by synapses [8][5]. Artificial neural networks (ANN) take inspiration from this natural design to perform computation or learning in computer systems. Artificial neural networks mathematically represent a function approximator, with individual neurons and their incoming synapses representing a smaller, simpler function. The usual way neural networks are viewed is as a directed network graph representing neurons with nodes, synapses with edges, and external inputs and outputs represented as special, non-synaptic connections. The earliest neuron design is the McCulloch-Pitts model, in which synapses represented a boolean input or its inverse, and neurons use the heaviside step function for their activation. The McCulloch-Pitts model of neural networks showed use in early perceptrons [9], feed-forward neural networks where each input was connected to each output with variable weights to process imaging data. There have been many neural network designs since this time, and many modern designs have synapses modify neuronal signals with a real-valued weight connecting to a neuron that may have any number of activation functions. Neural networks must be 'trained' using example data for the task the network is expected to perform. Training

usually involves modifying the weights of synapses within a network, but may involve altering other parameters or even changing the structure of a network.

Deep neural networks (DNN) are neural networks where the simple input and output layers of a perceptron are augmented with 'hidden' layers [10]. These hidden layers can be other fully-connected layers, such as in the multilayer perceptron, or can be augmented with various other layer designs, such as convolutional layers, pooling layers, and other designs. Training DNNs is most frequently performed using some variation of error backpropogation via stochastic gradient descent. This technique is very powerful, but amongst other drawbacks, it requires a continuously differentiable activation function in the neuron, which will be relevant shortly.

1.3 What is a Spiking Neural Network

A SNN is a biologically inspired extension of artificial neural networks where rather than a neuron necessarily outputting a value, instead, a neuron may or may not emit 'spike' events depending on its inputs [5]. A spike is a signal sent from a pre-synaptic neuron to a post-synaptic neuron across a synapse with a value and arrival time modified by the parameters of that synapse. This makes SNNs inherently temporal with a dynamic state that leads to their reference of being more 'biologically plausible' than traditional ANNs. While biologically-plausible models using systems of differential equations such as the Hodgkin-Huxley model[11] or the Izhikevich model[12] have popular uses in and adjacent to biological neuroscience fields, simpler models such as the Integrate and Fire (IAF) or Leaky Integrate and Fire (LIF) are popular for their high simulation speed. In IAF neuron models, spiked values that do not exceed the neuron's threshold will remain in the neuron over time as 'charge' and is used in several high speed neuroprocessor simulators such as RISP[13]. Many spiking models are LIF models which use some form of 'leak' where accumulated charge will decay over time, which may help ensure that the most recent spikes have a higher effect on a neuron's fire such as the RAVENS [14] or MRDANNA [15] neuroprocessors.

When a neuron does fire, it fires with a static value into its outgoing synapses, usually 1. The qualities of SNNs allows for tremendous potential for problem solving and makes them excellent candidates for novel analog and mixed-signal electronics, but they also lead to many challenges in learning and application. The static threshold function that LIF SNN's use is not continuously differentiable, which means that the error backpropagation via stochastic gradient descent will not work for these spiking neural networks without some modification. The nature of a neuron always firing with a fixed value means that the neuron's output cannot differentiate between slight or extreme super-threshold accumulation. Finally a neuron that does not fire does not induce activity later in the network, which can cause problems in both input encoding and can cause issues in very sparse and/or inhibitory networks.

1.4 Purpose

The purpose of this research is to analyze the effect of altering the graph techniques used in genetic algorithms for training sparse, recurrent spiking neural networks. There is existing research involving the use of genetic algorithms to build spiking neural networks, but many of these algorithms struggle to construct and optimize larger networks and thus can fail or be computationally expensive on more complex tasks. This work focuses on examining the way that graphs are generated in the process of training sparse SNNs using evolutionary algorithms.

I plan to address these challenges by dividing the workload of this process into smaller problems. First, my goal is to explore ways of combining neural networks to meaningfully approximate more complex functions within a spiking context. Second, I plan to integrate this network connection into an evolutionary optimization process to allow training to operate on smaller subnetworks, reducing the size of the search space for the problem. Finally, I propose to further limit the search space by eliminating networks with non-connected neurons, as additional neurons with no

connections exponentially increases the network search space without altering the network functionality.

Chapter 2

Literature Review

2.1 Neuromorphic Computing and Spiking Neural Networks

Spiking Neural Networks have become a much more active field of research as novel forms of computing have become more interesting in a Post-Moore's Law world [5]. Neuromorphic research has found many ways of training spiking neural networks. SNN research frequently is based on Lapicque's Leaky-Integrate-and-Fire model of the neuron, and frequently includes some form of Hebbian-like learning mechanism, especially in hardware implementations. Perhaps the most natural way of approaching training brain-inspired neural networks is using brain-inspired learning mechanisms. One such approach that has been historically popular is a method of Hebbian-like learning, called Spike-Timing Dependant Plasticity (STDP), where spike activity is used directly to modify network parameters (usually synaptic weights). Hebbian-like learning is concerned specifically with the connections of a network, commonly referred to with the aphorism "Neurons that fire together, wire together" [16]. Hebbian-like learning can take the form of either supervised or unsupervised learning for pattern recognition or classification. As these methods are concerned

with parameter optimization rather than topology, users must start by selecting a network topology manually.

Another natural approach is to use the gradient descent methods that have served DNNs and CNNs so well. Rather than try to optimize all of the spiking parameters, directly training the weights for a DNN is a relatively straight-forward process compared to SNNs [17]. Feed-forward Neural networks can train using a differentiable activation function to estimate a contribution to an inference error. Spiking neural networks traditionally use a discrete spike function which is non-differentiable, however, so some clever tricks must be used to use error backpropagation to train a spiking neural network. The Whetstone algorithm trains a traditional DNN using a bounded rectified linear unit, and "sharpens" the activation function, over the course of training to further approach a spiking threshold in behavior [18].

The Slayer algorithm gets around the non-differentiability of a deterministic SNN by backpropagating error as if the neuron were stochastic [19]. By also comparing outputs to a target spike train rather than just a particular network output, the timing of the outputs can be used allowing SLAYER to train delay parameters as well. These processes both involve the architecture of a traditional DNN and either altering the behavior of a DNN to appear spiky or an SNN to appear less so. The different dynamics of the SNN necessarily act differently as a function approximator than the original, so in addition to making very large networks, the accuracy is necessarily less than the originally trained DNN. Training these large networks requires numerous iterations through a dataset, which makes searching different topologies extremely costly.

Rather than just training parameters, genetic algorithms can train both parameters and network topology through genetic operators [5]. Unlike the previously listed methods which use a single network that is trained through numerous iterations, genetic algorithms generally use a population of networks which are evaluated each epoch. After being evaluated for their fitness (or error for classification problems), the genetic algorithm performs operations to alter the population for an ideally better

fitness. Mutation operations generally enact small changes on an individual member of the population whereas crossover operations usually have some way of combining two individuals to make one or more new individual based on the two parents. The EONS algorithm operates on a randomly generated population of networks, usually with a small number of hidden neurons and internal synapses [2] Mutation operations in EONS include adding and removing neurons or synapses, as well as randomizing the parameters of these objects. The neurons and synapses within an EONS based network have position in 1+ dimension space, and crossover combines two networks by splitting the hidden neurons and their synapses across a random boundary in that space into two different children. By allowing mutation and crossover operations to make small topology changes, such as adding and removing individual neurons and synapses, these genetic algorithms may search the entire space of possible topologies. Unfortunately this also means that the algorithm has a massive search space and can struggle solving problems with larger networks. While there are challenges in examining the space of all possible networks, this may be the best way of fully leveraging all of the possible features of spiking neural networks. EONS can take advantage of parallelism using a tool called Islands [20] Islands allows multiple runs of EONS to happen simultaneously across different computing nodes with automated management tools to stop non-improving eons runs or to allow high performing networks to 'migrate' to different eons runs.

2.2 TENNLab Exploratory Neuromorphic Computing Framework

The TENNLab software framework represents an end-to-end software solution for simulating, learning, and applying SNN's [21]. The framework consists of several modules which may be utilized together to train and simulate an SNN for a particular task. Applications in the framework include a suite of LIDAR control applications

[22], integration for the Arcade Learning Environment [23][1], and support for classification tasks such as the Wine [24] and Breast Cancer [25] datasets from the UC Irvine Machine Learning Repository. To assist the applications in converting information between numeric and spike formats, the framework has a robust suite of tools for encoding and decoding values as spike and back [26]. Application control is handled through an application agent, which handles hyperparameter management; establishing encoders and decoders to construction the spiking observation and action spaces; manage population of networks; and communicating between neuroprocessors, learning algorithms, encoders/decoders, and neuromorphic applications.

Training can be accomplished using algorithms such as EONS [2], Whetstone [18], decision trees, or reservoir computing. Learning algorithms can operate at high speed through the C++ interface, but integration of many different learning application both within and from outside the framework is possible through the python interface. To help training algorithms, the framework also has a Bayesian Optimization tool for hyper-parameter improvement [21].

2.2.1 Neuroprocessors

Neuroprocessors perform the actual SNN simulation within the framework. The TENNLab framework includes several neuroprocessors such as Caspian [27], RISP [13], RAVENS [14], as well as external neuroprocessors such as NEST [28]. Neuroprocessors in the TENNLab framework use a standardized interface, so that applications and training algorithms work consistently regardless of the chosen neuroprocessor.

The RAVENS neuroprocessor is a configurable SNN system with LIF or IAF neurons developed by TENNLab [14]. RAVENS can take the form of a software simulation, an FPGA, and both analog and digital ASIC designs. The neuroprocessor is highly configurable, with controllable leak, table-based STDP, and two controllable refractory periods. RAVENS can also be configured with a range of maximum delays, weight resolution, maximum neuron in-degree, maximum neuron threshold,

and maximum refractory periods. These parameters give RAVENS a great deal of control, but have a strong impact on the data size of neurons which can greatly impact the number of neurons that can fit onto digital implementations such as FPGAs.

The RISP neuroprocessor is a high speed SNN simulator with a simple feature set developed by TENNLab [13]. Leak is configured on a per-neuron level with leaking neurons depleting all stored charge each cycle if they don't fire, and all synapses only have a single-cycle delay. These reduced temporal dynamics allow for RISP to maintain a high simulation speed, and this will be the neuroprocessor used for evaluation for this project.

2.3 Datasets

The TENNLab framework has a useful classification application which allows users to import datasets of their own for training. The MNIST dataset [29] consists of 28x28 greyscale images of handwritten digits collected and modified from the NIST Special Database 19 NIST Handprinted Forms and Characters Database [30]. This collection of images is a popular dataset for machine learning training [5]. Another collection using the same format of 28x28 greyscale images is the Fashion-MNIST dataset [31]. Intended to be a more challenging form of the MNIST dataset using the same data format, Fashion-MNIST contains images of clothing and accessories to identify from 10 different classes.

Chapter 3

Genetic Algorithms and Neuromorphic Computing

Each of the genetic algorithm learning processes involves many of the same steps, the details of which vary from algorithm to algorithm but the overall process remains similar. Unlike training of a DNN, evolutionary optimization involves a population of multiple starting networks rather than a single network with small initial parameters. The initial population of networks is generally random, with later generations taking the performance of the population into account to better exploit the best performing networks. However for any of these algorithms it is important to keep introducing random networks regularly into the population to maintain some level of diversity and avoid over-fitting into local optima.

Each epoch of optimization begins with the population being evaluated at the desired task. Evaluations of spiking neural networks involve taking some kind of input, usually numeric, and converting them into input spikes. In the TENNLab framework, this involves the use of input encoders, which turn a single input value into one or more spikes across one or more input neurons. With this initial activity pre-programmed, the network is run for a user-specified time period. During this time the activity of the chosen output neurons is tracked, and the network utilizes an

output decoder to help the application convert spike activity into a choice or action. Depending on the application, the internal network state — neuronal charge, delayed spikes, etc — may or may not be wiped out in between each run of the network.

Classification tasks generally take a pre-labelled data set, divided in between training, testing, and verification subsets. Each epoch's evaluation involves the population being evaluated on the data in the training set, and the networks are scored in how accurately the output matches the existing data labels. This accuracy data is used to improve networks each generation. Once the final epoch of training is reached the performance of the best network(s) is compared to the testing dataset, to see how the network performs against novel data; poor performance during testing may indicate that a network has "over-fit" the training data, and will generalize to real-world data poorly. In execution for SNNs in a genetic algorithm, a single set of inputs being passed to the network and each run's output are used for a single classification example.

Control tasks usually cannot be solved with a single set of outputs from a network, representing tasks such as automation, robotic operation, and game playing. Control problems usually involve many, related network executions, with each set of inputs corresponding to the application state at a timestep and application state, and the outputs of the network corresponding to an action taken, similar to reinforcement learning problems. However, the evaluation of the network operates as normal for TENNLab applications, and the network is not evaluated based on individual state-action pairs but instead on the performance across one or more episodes.

Chapter 4

Network Compiler

Combining smaller subnetworks to form into a larger network is an essential activity in building neural networks for a task. From simple linear neuron layers with 'fully-connected' synapses into another neuron layer in multilayer perceptrons to the complicated stacks of connections in modern convolutional networks, solving applications with neural networks frequently involves deliberate choice in the construction and assembly of the final, problem-solving network out of smaller subnetworks. This approach is used in neuromorphic computing with techniques such as Whetstone or SLAYER that convert conventionally trained DNNs or CNNs into spiking networks, but there may be room to test how spiking-native algorithms can integrate a super-network structure into their workflow. In this section I propose a network compilation tool that can be integrated into the EONS evolutionary algorithm.

The compiler is a C++ function with a pair of support structures to format the compiler instructions. It operates on these user instructions to connect subnetworks together starting from the inputs of the final network (see fig 4.1). Following the TENNLab framework, only a single neuron may be associated with a given input value in the final network, so only a single network may be given overall network inputs in its instructions. The network compiler generates additional instances

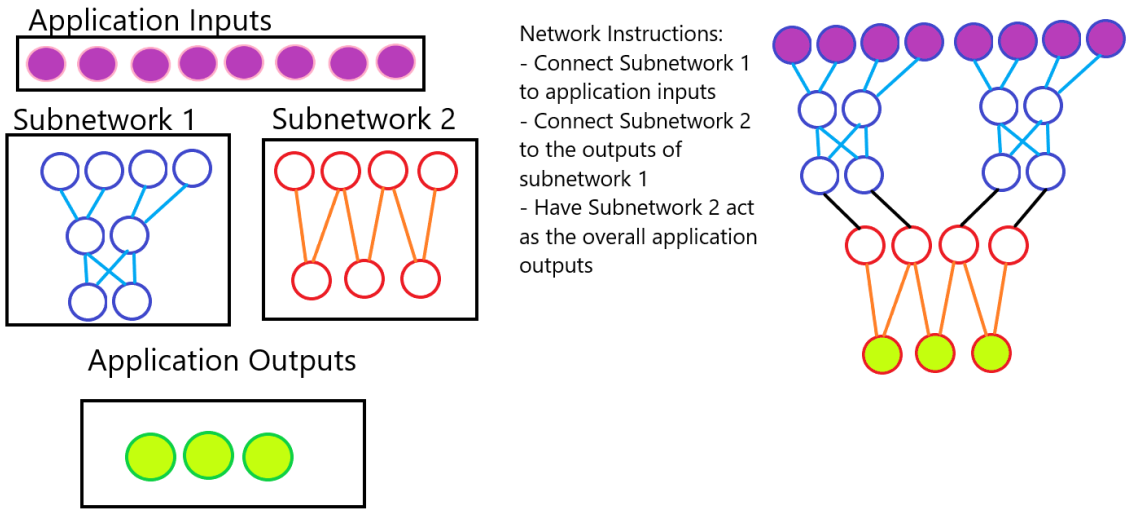


Figure 4.1: A simple example of the network compiler with two subnetworks for an application requiring 8 inputs and 3 outputs.

of a network as needed to span all inputs, including the user-specified overall network inputs. The network compiler works in three steps which will be detailed in later sections: pre-processing, network replication, and network connection.

4.1 Network Compiler Example

Figure 4.2 is an example of a compiled network. This RISP network uses a few subnetworks to detect if there are 4+ input spikes in either the left (input neurons 0-5) or right (input neurons 6-11) half of the inputs, but not both. The network has 12 input neurons and a single output neuron. The network consist of different subfunctions (input handling, counting spikes, xor) and the spike counting must run twice, making this an ideal case for using the network compiler.

For the Network Compiler to make a TENNLab framework legal network, there must be a PropertyPack to use as the basis of the network. In this case, the PropertyPack of a RISP building-block network contains the PropertyPack in question [32].

The inputs in this case are simple neurons with a single connection each. Each input neuron is connected exactly once, which means that in this case there is no need for a separate input layer, but in many cases inputs are reused such as in grid-connection schemes, so the layer is demonstrated for example rather than practicality. The input layer has no intra-network synapses, and each neuron has the minimum threshold for property pack.

The counting network is based on Rizzo's spike counting networks [1]. In this case, these are 6-neuron counting networks with a counting threshold of 4 spikes . The counting network is set to operate over 20 cycles with a 2 cycle reset period. RISP lacks the ability to set a long refractory period as described in Rizzo's paper, so instead the network compiler builds a parallel structure (neurons 33-39) to the threshold neuron (neuron 32). The reset structure has a minimum threshold value, and uses a chain of single neurons connected by synapses with controlled delays

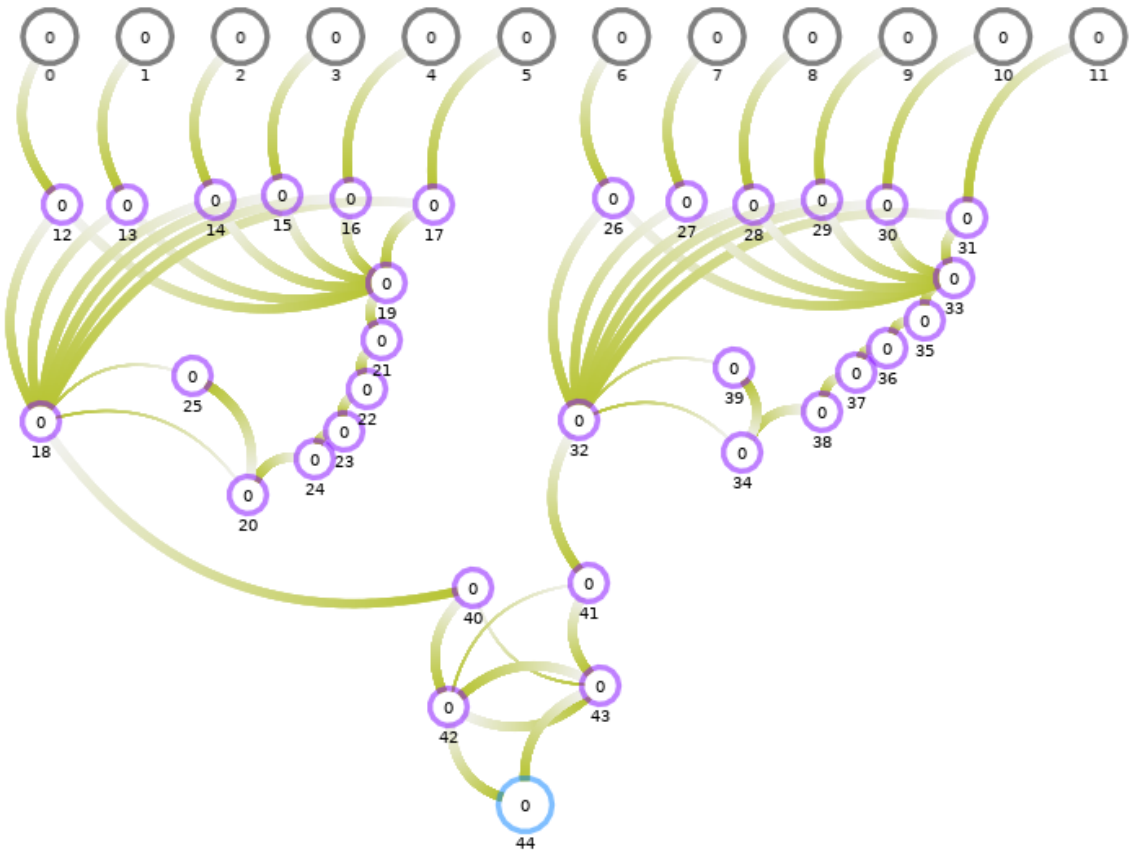


Figure 4.2: A simple compiled network that detects if there were 4 or more fires on either the left or right half of the inputs (top row).

to ensure that the inhibitory connection from neuron 34 and the excitatory connection from neuron 39 will reset neuron 32 to its default charge state if neuron 32 did not fire.

The XOR network is a pre-build building-block network for RISP [32]. Every network compilation task requires at least one existing network to serve as the base PropertyPack for the final network product. In addition to this task, this small 2-input, 1-output network also acts as the output layer for the final network, with the equivalent of neuron 4 being the sole output neuron in the final network (see fig 4.3c).

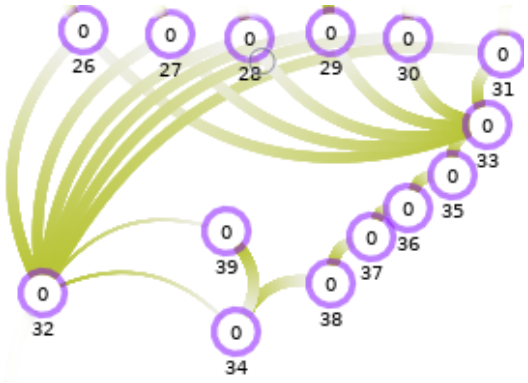
Each of these subnetworks includes an instruction that tells the compiler how to incorporate the subnetwork into the final network. The input and counting subnetworks are also generated using the compiler instruction. Combining the loaded network and its property pack, the 3 instructions, and the call for 12 input neurons, the compiler can begin assembling the final network.

The first instruction creates a simple, one neuron subnetwork. That neuron is both an input and output for its subnetwork. The other part of the first instruction says that the subnetwork will fill the role of the input neurons for the final network. The final network is to have at least 12 input neurons, so 12 instances of the 1-input subnetwork will be generated as per figure 4.3a, each instance being a copy of the original network's properties with sequentially increasing neuron ids. These 12 neurons, which act as both input id 0 and output id 0 within the subnetwork are instead labeled as an input id based on the instance id, the fact that the neurons are outputs is tracked by the compiler for connectivity but are not specially noted in the final network.

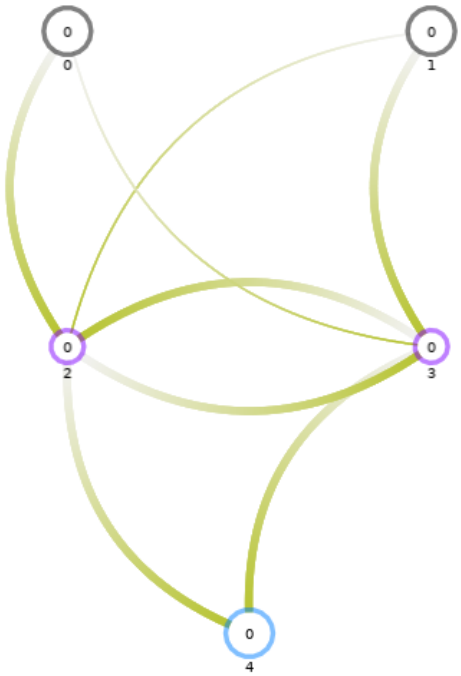
The second instruction calls for the generation of a simple-type counting network with 6 inputs, a counting threshold of 4 spikes, a collection period of 20 cycles, and a reset of 5 cycles. The instruction also calls for the second subnetwork's inputs to be connected to the outputs of the input subnetwork. The compiler generates the 14 neuron subnetwork and compares its 6 inputs to the 12 total outputs of the input subnetwork across all instances. In this case it is necessary to make 2 instances of the



(a)



(b)



(c)

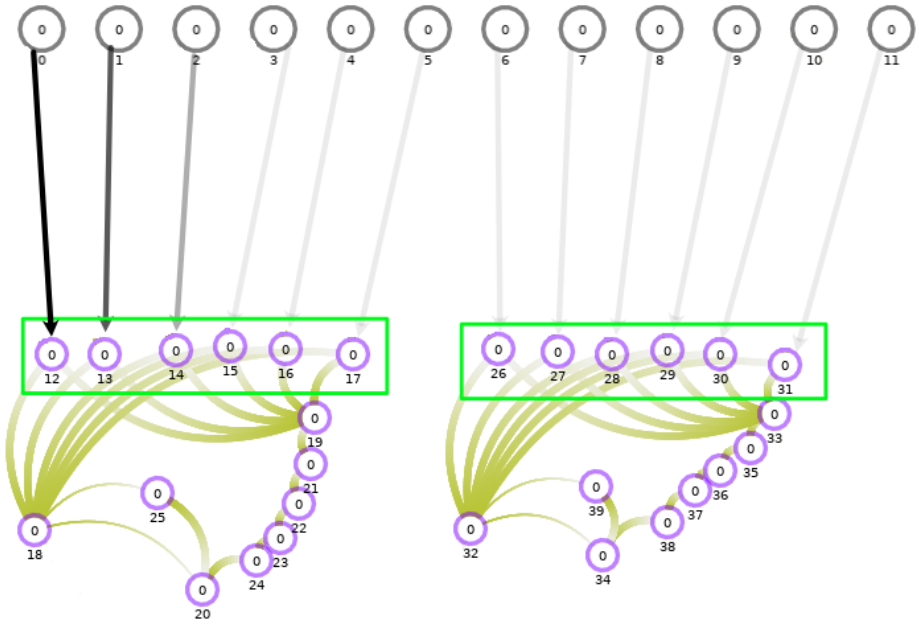
Figure 4.3: The 3 subnetworks for the compilation example including:
a) A simple 12-neuron input layer
b) A 6-input counting network detecting 4 inputs over 20 cycles with a 5-cycle reset
c) RISP xor network

subnetwork, for a total of 12 inputs and 2 outputs. The inputs for this instruction are another subnetwork rather than the final network inputs, so additional synapses need to be created to connect the networks. Subnetwork instances are created in order of their id, so neuron id's for a subnetwork increase as the instance id increases. By default the compiler creates a single synapse from each output neuron in the target subnetwork with minimum delay and maximum weight, with each further output neuron connecting to the next available input neuron of the current subnetwork. Following this instruction the compiler creates these simple synapses starting with neuron 0 connecting to neuron 12, continuing to the second subnetwork connecting neuron 6 to neuron 26, and ending by connecting neuron 11 to neuron 31 (see figure 4.4a).

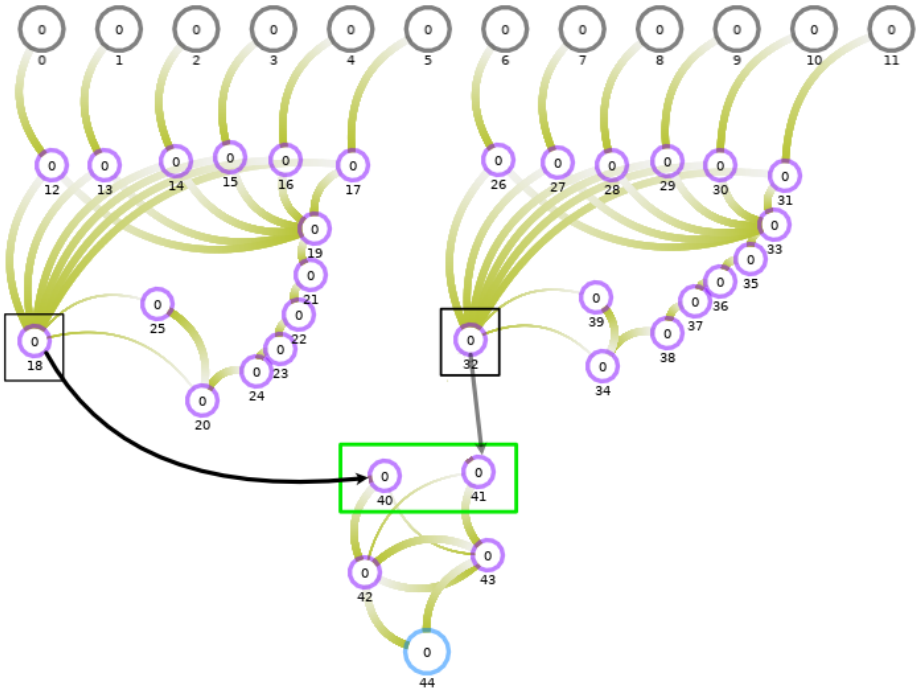
The third and final instruction concerns the placement and connections for the loaded XOR network. The network is set to use the outputs from the counting subnetwork as its inputs. An additional flag in the instruction tells the compiler to use this subnetwork's output neuron(s) as the output neurons for the final network. The compiler instructions are input-facing, so this topology information is separate than other connectivity information. In this case there are two total output neurons from the counting subnetwork and two input neurons in the XOR network so only a single instance is needed, with parameters copied from the original into the final subnetwork. Neuron 44 is set to be output id 0, finalizing the 12-input 1-output network. Connections once again occur in increasing neuron ID so first one synapse is created from neuron 18 to neuron 40, and the compiler creates the final synapse connecting neuron 32 to neuron 41 (see figure 4.4b).

4.2 Compiler Instructions

The most unique of the network compiler's inputs are the instructions. These can take the form of a .json file, terse strings, or a simple structure used internally within the compiler. Each instruction applies to a single subnetwork, and may indicate that



(a)



(b)

Figure 4.4: The process of creating inter-subnetwork connections for the example's first instruction (a) and second instruction (b)

the compiler should generate the subnetwork in question. The compiler builds the final vector of networks by examining instructions one at a time in order and either pairs that instruction with the lowest-index user-provided network or generates a network. Assuming the inputs to the compiler are well-formed, the result will be matched-length vectors of instructions and networks. Any instruction operates on the network with the matching index value, referred to as the current subnetwork while the compiler operates on that instruction. Instructions reference other subnetworks by using the vector index of the instruction that applies to that subnetwork. The list of instruction commands can be seen in table 4.1.

The primary structural component of an instruction is the list of subnetwork indices that the compiler should connect as inputs to this subnetwork. If the current subnetwork is to serve as the inputs in the final network, then the instruction will include a sentinel value of -1 in the list of inputs. The input command is modified by the Input Mixing flag, which causes the compiler to connect same-numbered outputs from the target networks to the same-numbered input in the current subnetwork, generating fewer instances of the current subnetwork when there are more than one input (see fig 4.5). When input mixing is disabled, the compiler will instead Span Input mode, where each input neuron connects to exactly one output neuron from the worklist of target connections. The delay parameter sets all incoming inter-subnetwork synapses to have the specified delay value, limited to the legal range within the PropertyPack. The output parameter indicates that the output neuron of the current subnetwork should act as the outputs of the final network.

The manual instance count instruction creates only the specified number of instances of the current network, which can be used to solve circular dependency issues for recurrent networks. Manual instances are also required for fully-connected mode. While fully connected layers are common in DNNs, they are much more rare in SNNs. The high activity factor possible with fully connected synaptic connections means that it is unlikely for maximum weight synapses to be of much use. Because of this the fully-connected instruction comes with a parameter to seed a MOA random

Table 4.1: Network Compiler Instruction Format

Command	JSON key	JSON type	string	internal format
Input	input	array	i [] /i	vector of integers
Delay	delay	integer	d []	unsigned integer
Input Mix	mix	integer	mix	boolean
Generate	generate	integer	g []	boolean
Counting	counting_network	object	gcn []	boolean
Output Grid	output_grid	object	og []	boolean
Input Grid	input_grid	object	ig []	boolean
Fully Connected	fully_connected	object	fc []	boolean
Manual Instances	instances	integer	n []	unsigned integer
Time Multiplexing	time_multiplex	boolean	tm []	boolean
Override Link	override_link	array	ol []	vector of objects
Output	outputs	boolean	o	boolean

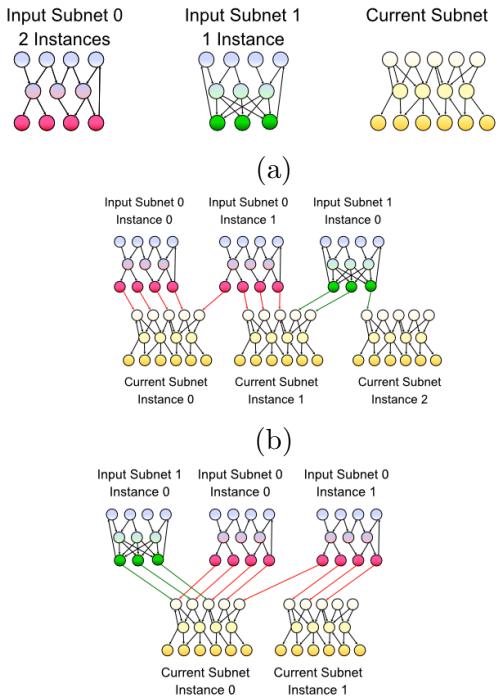


Figure 4.5: The compiler's instructions (a) can connect multiple differents input subnetworks together in by either spanning inputs (b) or overlapping into the same instance with the input mixing flag (c)

number generator to setup the weights for the connections.

Time multiplexing causes the compiler to create a single instance, and connect the different instances of the inputs with long delay values instead. Time compression works on time-multiplexed inputs by inverting the delays of the output synapses to bring simultaneous spikes from a time-multiplexed subnetwork instance into the instances of the current subnetwork. Time multiplexing and time compression are described more in Chapter 6.

The 'generate simple network' command creates a network made up of unconnected neurons each of which has the minimum allowed threshold and is both an input and output neuron of matching ids. The user may specify the number of neurons per instance; for basic and fully-connected modes there is no difference between large simple networks and large instance counts of a single neuron each. However, in time-multiplexing and grid connection modes network instance counts are meaningful, and a user must use care to ensure that the connection type works as intended.

The 'counting' command creates a counting network based on Charlie Rizzo's work on spiking counting networks [1]. In these structures, a large number of neurons feed simple synapses to a single counting neuron (see fig 4.6). Counting networks rely on controllable neuron thresholds, a high in-degree of synapses into certain neurons, and a controllable neuron refractory period. Out of the TENNLab NeuroProcessors, RAVENS is the best candidate for counting neurons, but particularly in hardware-equivalent configurations the simulator has limits on these parameters. The compiler can attempt to compensate for this design through a loss of strict counting accuracy by building a pyramid of neurons which accumulate to a user-controlled 'subcounting threshold'. Then the final counting output neuron uses its own counting threshold. This allows for a network which may produce similar output when the max in-degree of a spiking neuron is limited. The ability to generate counting networks is useful as otherwise every different counting threshold or number of neurons to be counted would require manual creation or editing of an existing network which is both time-consuming and prone to error. The counting network generation mode has parameters

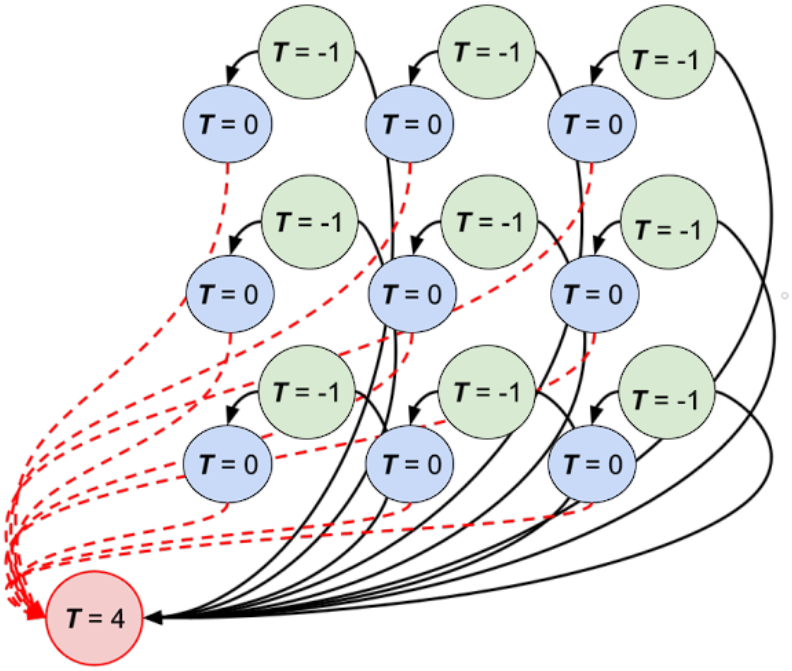


Figure 4.6: A 3x3 spiking counting network, courtesy Charlie Rizzo [1]

for the number of input neurons or counting size, the number of cycles to count over, the counting threshold, and the sub-counting threshold. The counting network also builds a 'reset structure'; a parallel counting network with minimum thresholds which then use inhibitory spikes into the counting and subcounting neurons to reset the operation as described in the paper, and a user specified time period for the reset to occur over. Finally, a user can also instruct the compiler to use their choice of the three counting network designs presented in the paper.

The 'output grid' instruction specifies how its output neurons are to be organized when combined with the input grid command on a subnetwork that takes the current network as input. The output grid takes the dimensions of a 2d-grid, the product of which must match the number of output neurons of the current subnetwork exactly. The output can be visualized as forming a grid, with output 0 in the first position, and proceeding through the output neurons in row-major order.

The 'input grid' command specifies the tiling of network instances across the formatted outputs of one or more of the current network's inputs (see fig 4.7). The user specifies the dimensions of a 2d-grid, which must exactly match the number of input neurons. The user may specify the desired boundry for the input grid per border, with negative values expanding the size of the input grid, allowing for overhang. The user then specifies the stride distance and direction, allowing row-major or column major traversal along with the stride size. The user may specify a stride smaller than the grid itself, which will result in overlap; the input grid / output grid pairing ensures that the input grid cannot work directly on the final network inputs, circumventing the TENNLab framework limitation on an input connecting to only one neuron. By default if the dimensions of the input grid are not a multiple of the padded output grid dimension, then no partially-connected instances will be generated. However the user may allow overhang at the end of each dimension by disabling the fitgrid parameter.

Finally, override links allow individual, manual connections between subnetworks allowing more intricate connections. An overridden connection describes a connection

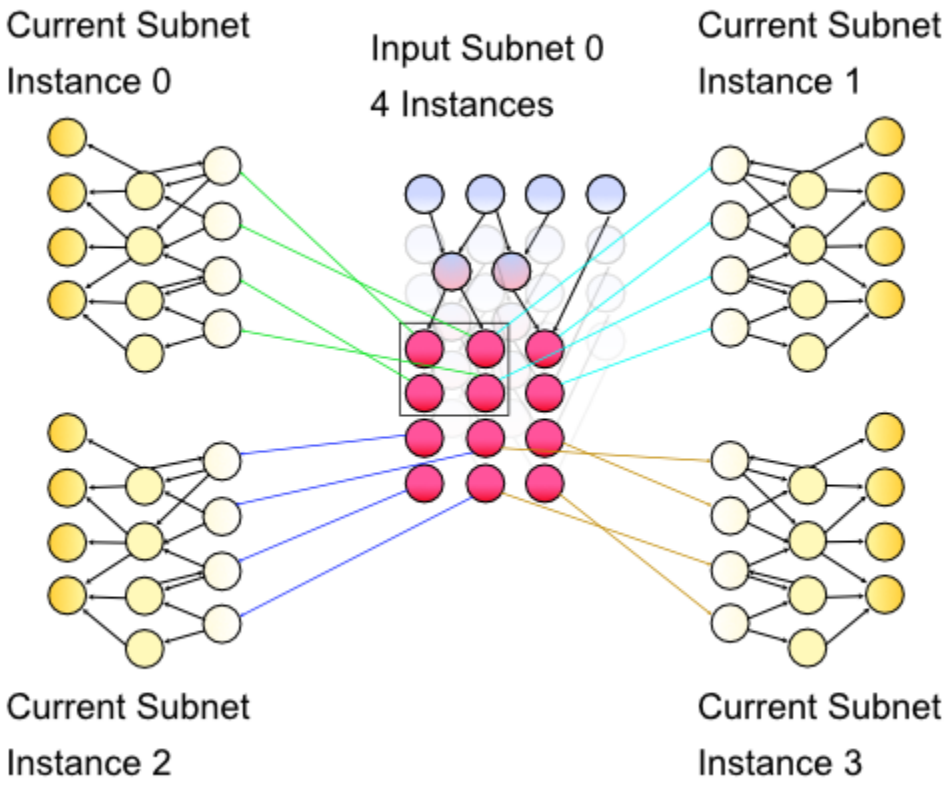


Figure 4.7: Example of network compilation to an input set to a 3x4 input grid

to a specific input neuron from a specific output neuron for each instances of the current subnetwork. If the number of instances between the current subnetwork and the target subnetwork are different, an override instance mapping must be provided by the user for each instance of the current subnetwork. The user may specify the delay value of the overriden connections, and may choose to mix with or block other connections to that input neuron. Each input that is blocked reduces the number of input neurons available for standard connections. Without careful design, mixing manual and automatic connections can increase the number of instances of a subnetwork required.

4.3 Pre-processing

The pre-processing step of network compilation involves choosing the PropertyPack and parameter ranges for the final network, determining the number of instances that are required of each subnetwork, mapping the connections between all of the subnetworks, creating an instance of generated networks requested by the user, and organizing overloaded connections. This is also the step where the user instructions are interpreted from either a string or json format.

In the TENNLab exploratory framework, a PropertyPack describes to the network, runtime agent, and learning algorithm what the parameters can be altered in the neuroprocessor, what value type they take, and the range of legal values for those parameters. The network compiler examines the vector of input networks for the lowest-indexed network and uses that PropertyPack for the final network. For the purpose of generated networks, the minimum and maximum values of synaptic weights and delays as well as neuronal thresholds and refractory periods are extracted.

The user may request the generation of a "Simple Network": a subnetwork with one or more neurons and no synapses. The neurons in the network have the minimum threshold allowed for the network, and any other neuron proprties will be set to the maximum value in the PropertyPack. Each neuron in the simple network is set to

be the input and output neuron of its index. One of the uses of these networks is to re-use input signals; the TENNLab framework does not allow an input to map to more than one neuron or for a neuron to map to more than one input. This limitation means that a separate input neuron needs to be included for either time-multiplexed inputs or overlapping grid kernels. The user can also request generation of a counting network.

The next preprocessing step is creating a connection map. Each instruction must have one or more input connections specified, even if all of the inputs are overridden. A single pass through the network instructions allows for a simple map of all inputs leading to the target subnetwork. This pass also collects the list of input and output neurons for each subnetwork, which is needed for determining the number of instances.

The network compiler uses a system of overrides within user instructions to permit the connection of single neurons in ways different than that of the overall instructions. Overrides have a flag to allow mixing overrides with the standard, automatic connection synapses. If this setting is false, then the input neuron targeted by the override is removed from the input neuron list. In either case the entry for the overridden input connection is saved for later. Finally, if all of a subnetwork's inputs are overridden without mixing and the network does not have a given number of instances, the network is set to a single instance and is considered solved for number of instances. During the pass for overrides if any instruction specifies a user-set number of instances, those networks are also considered solved.

The compiler then begins to solve the remaining number of instances. The remaining networks to be solved are those that are not fully overridden nor do they have a set number of instances set by the user. The compiler will make one or more passes across the list of subnetworks with unsolved instance counts. If no progress happens during a pass, the compiler detects a circular dependency and quits with an exception to the user. When a subnetwork that is unsolved has only solved subnetworks as inputs, then its input counts can be determined. Simple connections between subnetworks can allow input mixing or not. When input mixing is allowed,

the number of instances is based on $\lceil(\frac{\max(\text{output neurons})}{\text{input neurons}})\rceil$ for all outputs of all input subnetworks where as if input mixing is not allowed, then the number of instances for the subnetwork is based on $\lceil(\frac{\sum(\text{output neurons})}{\text{input neurons}})\rceil$ for the output nuerons of all instances of each of the input subnetworks. If a network's only inputs are those of the final network (set in the instruction as -1) or a combination of final network instances and already solved networks, then the number of instances of that subnetwork can be determined treating the user-specified number of final network input neurons as another count of output neurons for the above determinations.

When the user specifies an input grid configuration and the current subnetwork has a target subnetwork with a valid output grid specification, then grid-type connections are used instead. The grid connection mode has many options to customize the layout of instances across the output grid of the target subnetwork, described above. In the strict, fitgrid mode, the number of instances may be determined as:

$$(1 + \lfloor\frac{x_o - x_i - b_l - b_r}{s_x}\rfloor) * (1 + \lfloor\frac{y_o - y_i - b_t - b_b}{s_y}\rfloor)$$

where the output grid dimensions are x_o by y_o; the input grid dimensions are x_i by y_i; the left, right, top, and bottom boundries are set by b_l, b_r, b_t, and b_b; and the x and y stride are s_x and s_y respectively. If fitgrid is disabled, then the instances are instead calculated as:

$$(1 + \lceil\frac{x_o - x_i - b_l - b_r}{s_x}\rceil) * (1 + \lceil\frac{y_o - y_i - b_t - b_b}{s_y}\rceil)$$

These instance counts are for the specific connection to the target subnetwork. If input mixing is allowed and the current subnetwork has more than one input, then these grid connections will share instances with other inter-network connections, grid type or not. If input mixing is disabled and the current subnetwork has more than one input instruction, then these grid instances will be in addition to any other connected instances of the current subnetwork.

4.4 Network Instance Generation

After the instance count for each network is determined, the number of total nodes that all instances of each subnetwork are calculated along with the starting node index for each subnetwork. Neurons may not share an ID, so no attempt at maintaining the exact node id is attempted. Rather, neurons are given sequential ID value, with the lowest indexed neuron in the sorted node vector of instance 0 of subnetwork 0 being given id 0 and going upward by neuron, subnetwork index, and finally subnetwork. The node count of a network instance is simply the product of the number of nodes per instance and the number of instances. The starting node id of the a subnetwork is 0 for subnetwork 0, otherwise $\sum_{i=0}^{c-1} \mathrm{nc}_i$ where nc_i is the node count of a subnetwork at index i and c is the index of the subnetwork in question.

With the necessary information to label the neurons, the neurons of all subnetwork instances are generated. The parameters of the original neuron are copied to its copies in each instance of current subnetwork. When the new neuron is an input neuron of the subnetwork marked to act as the inputs in the final network is added, it is given a sequentially increasing input id starting at 0. A mirrored process occurs for output neurons on the subnetwork marked to act as the final network's output with output ids starting at 0.

After the neurons are created, the compiler creates the internal subnetwork synapses for each instance of each subnetwork. Synapses in the TENNLab framework are stored in a map indexed by the pre- and post-synaptic neuron IDs. With no ID necessary it is a simple process to use the node count and starting ID for the subnetwork in question to convert from the subnetwork ID of a synapse's neurons to the ID in the final network.

With all network instances being recreated, the compiler then sets up overridden network connections before moving on the general inter-subnetwork connections. It is at this point that the override instructions are checked for validity, to avoid wasting time before more serious failure points can be identified. Each overridden connection

instruction contains all of the specific details governing the rules of connection including subnetwork instance mapping, so this process is very simple.

4.5 Inter-network Synapse Generation

The process of connecting the instances of the various subnetworks together occurs in ascending order of instruction indices. Unless specified otherwise, inter-subnetwork synapses will have the maximum weight allowed by the PropertyPack. By default, inter-subnetwork synapses have the minimum delay allowed in the PropertyPack, but a user can specify a manual delay instead. The output neurons from each instance of a single target subnetwork are grouped together into a single-dimensional array. This array acts as a worklist for the target subnetwork. In the case of input mixing, the instance of the current network will begin at 0 for each input, but when input mixing is disabled (span mode) the current network instance will continue incrementing for each new target subnetwork.

For standard inter-network connections, the worklist is consumed in order. For each entry, the output neuron in question is connected to the next input neuron in sequence in the current subnetwork. The compiler does not attempt to specifically match instance to instance; for example if the number of non-overridden inputs in the current subnetwork is less than the number of outputs in the target subnetwork, the outputs of the target subnetwork will connect to multiple instances of the current subnetwork.

For the fully-connected mode, each output neuron in the worklist is connected via synapse to each input neuron in every instance of the current subnetwork. Using the seed value tied to fully-connected mode, each synapse is given a random weight value in the legal range allowed by the PropertyPack, along with the delay as described above.

In grid connection mode, the compiler uses the outputs of all instances of the target and fills a grid of the dimensions set as the output grid and considers the

grid to be filled with the output neurons in row-major order in order of $(j * o_c) + i$ where j is the instance, o_c is the number of outputs in the target subnetwork, and i is the output neuron id. The compiler then loops through all grid starting coordinates based on padding, input grid size, and step size following either row-major or column-major order based on the instructions. Each starting point indicates an instance of the current subnetwork whereas the output grid spans all instances of the target subnetwork. The compiler loops through the size of the input grid, connecting the next output within the input grid boundries following row-major order to the next input neuron of the instance. This process is repeated for all of the instances of the current subnetwork. If the boundries of the input grid are set to negative value or the fitgrid parameter is disabled, an instance of the current subnetwork may not have connections to each of its input neurons. Recurrence may still allow those inputs to be valid connections, however pruning is recommended to reduce unnecessary network size.

With the subnetworks connected, the network is complete and is returned. The TENNlab framework includes a pruning tool for situations where inputs may not be connected. The resulting network is otherwise connected, and while the neuron id's may be largely unrelated to their host subnetwork, the topology of the duplicated subnetworks should be identical.

Chapter 5

Compiler-Assisted EONS

The compiler can be integrated into the EONS training regime but some modifications to the algorithm are required. In the compiler-assisted mode of EONS, multiple subpopulations can be managed by the app agent. Each subpopulation has matching input and output counts, and all fill the same subnetwork role in the compiled network. Evaluation is performed on the power set of all combinations of subnetworks through the compiler. The scores for each compiled network are kept, but each subnetwork keeps a list of all scores for compiled networks it is part of this epoch. Instead of the normal population size hyperparameter, compiler-assisted EONS takes a list of instructions, subpopulation sizes, subnetwork input counts, and subnetwork output counts. The user does not need to specify an input size like the standalone compiler tool as this is determined by the encoder array and application.

Compilation is performed in the worker thread, to prevent bottlenecks in the main thread. By the rule of product, the size of the work queue in compiler-assisted EONS is the product of all subpopulation sizes. In the work threads of the app agent, the work queue is converted back and forth from a linear integer representation and a multi-dimensional array representation, indicating which combination of subnetworks from each subpopulation is used. The compiled network is then used for network evaluation as normal in a worker thread in the TENNLab framework. In addition to

keeping the score for the overall network, each subnetwork keeps an array of scores from this epoch. Compiled networks are not kept, but scores are instead associated with their work queue id allowing them to be recreated as needed.

The app agent records the best fitness for each subpopulation and passes that fitness value to EONS to evolve for the next epoch. Within EONS, each subpopulation is treated normally each epoch, and each subpopulation evolves separately. When EONS ends, the best performing network is recompiled and saved, and each of the subnetworks are saved with a filename extension indicating the subpopulation index.

5.1 Compiler-Assisted EONS Simple Demonstration

An example of compiler-assisted eons is to solve a binary classification task where the solution output is the XOR of the AND of two pairs of inputs. The compiler's instructions are to have a 2-input 1-output subnetwork (subnetwork 0) act as the input layer, and another 2-input 1-output subnetwork (subnetwork 1) take the outputs from the two instances of the input subnetwork and provide the output neuron for the final network. In this example, EONS is instructed to load a set initial population of binary operation building block networks [32] instead of generating populations. Specifically, each subpopulation will begin with a RISP XOR, OR, and AND subnetwork, instead of each subpopulation being filled with its own vector of networks by EONS (see figure 5.1a). The network instructions are always shared across an EONS run. In this case, the first instruction is to act as the final network inputs, the second instruction is to take the subnetwork 0 as inputs and act as the final network outputs.

Compiler assisted EONS creates a population of valid evaluation networks by compiling every combination of subnetworks (see figure 5.1b). The evaluation worker threads within EONS retrieve a subnetwork from each subpopulation based on a work queue and compile the networks together. The final network runs as normal, applying

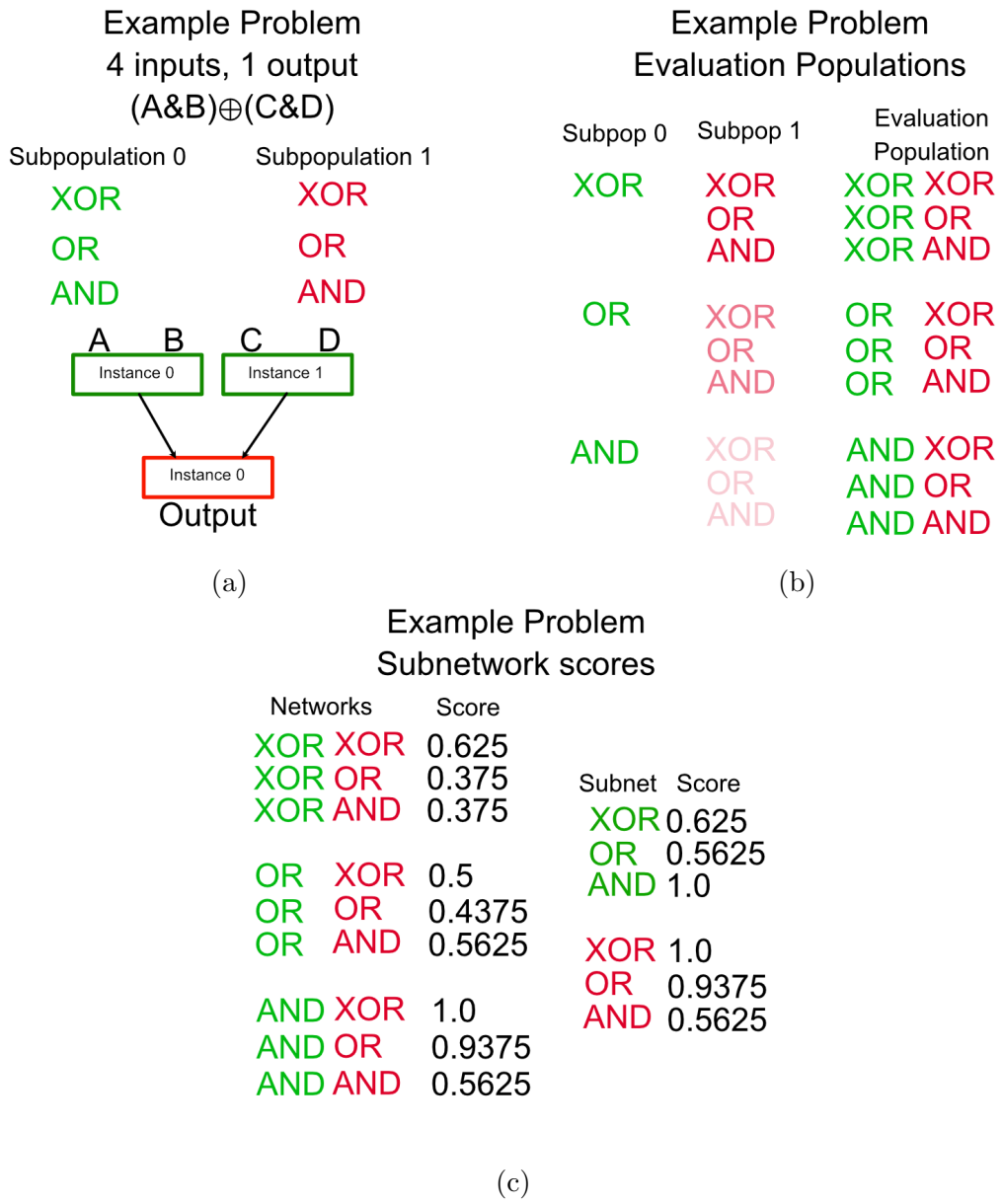


Figure 5.1: A example of compiler-assisted EONS for a binary classification problem with fixed subpopulations (a) and the demonstration of how evaluation networks are generated (b) as well as scoring for the compiler EONS example, demonstrating how scores are determined for the evaluation networks and the max scores for individual subnetworks (c)

inputs from the application through an input encoder, producing outputs that are passed through the desired output encoder to the application.

The application returns an accuracy score for each evaluation network based on the 16 possible input configurations (see figure 5.1c). In addition to keeping the best overall network, the compiler-assisted EONS mode keeps the best score for each subnetwork. In this case, maximum fitness was achieved so no population operations are needed. If this were not the case, each subnetwork within a subpopulation uses the best fitness of any network it was included in to determine its fitness within the subpopulation (see figure 5.1c). These fitness values would then affect selection operations performed on that subpopulation.

5.2 Evaluation

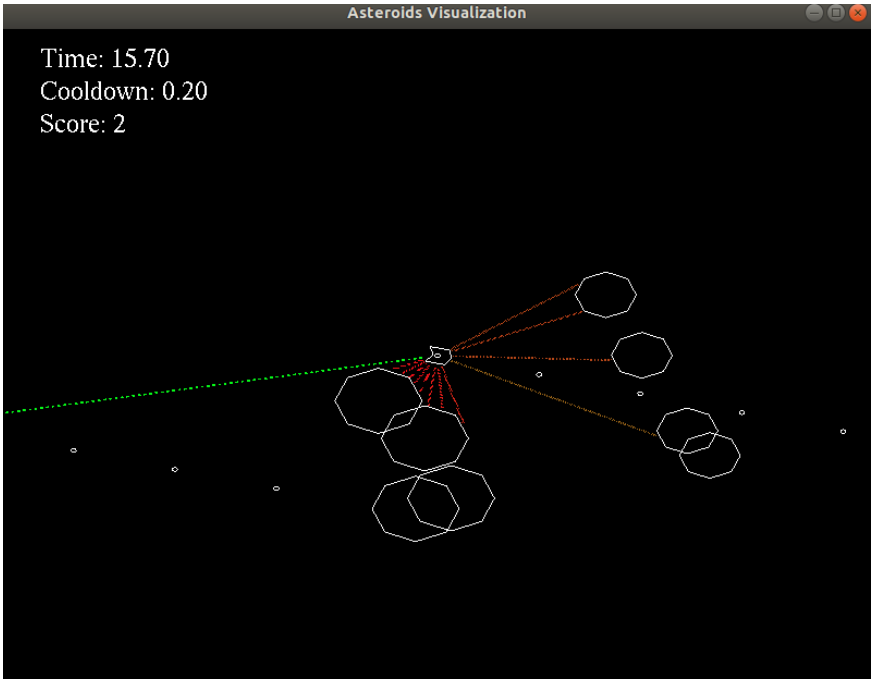
The motivation of this work is to enhance the training process, so evaluation is based on the comparison between EONS and compiler-assisted EONS. Multiple application choices on applications that EONS cannot trivially solve are important to establish the differences with and without compiler support. Neither EONS nor the compiler are directly tied to the type of application, so it will also be helpful to have both control and classification applications. For this evaluation I will be testing with the TENNLab Asteroids control application [22], the TENNLab Space Invaders control application, and the MNIST image classification dataset [29]. The asteroids control problem has a large number of identical, lidar-based inputs which gives a large, uniform input space which may be ideal for simple compiler instructions. The space invaders control task on the other hand groups input lidars which are non-adjacent as well as adding bias and position information, which serves to test the compiler on non-uniform inputs with more complicated instructions. The MNIST dataset has 784 inputs and 10 output classes which should serve to examine success on a larger dataset.

5.2.1 Asteroids

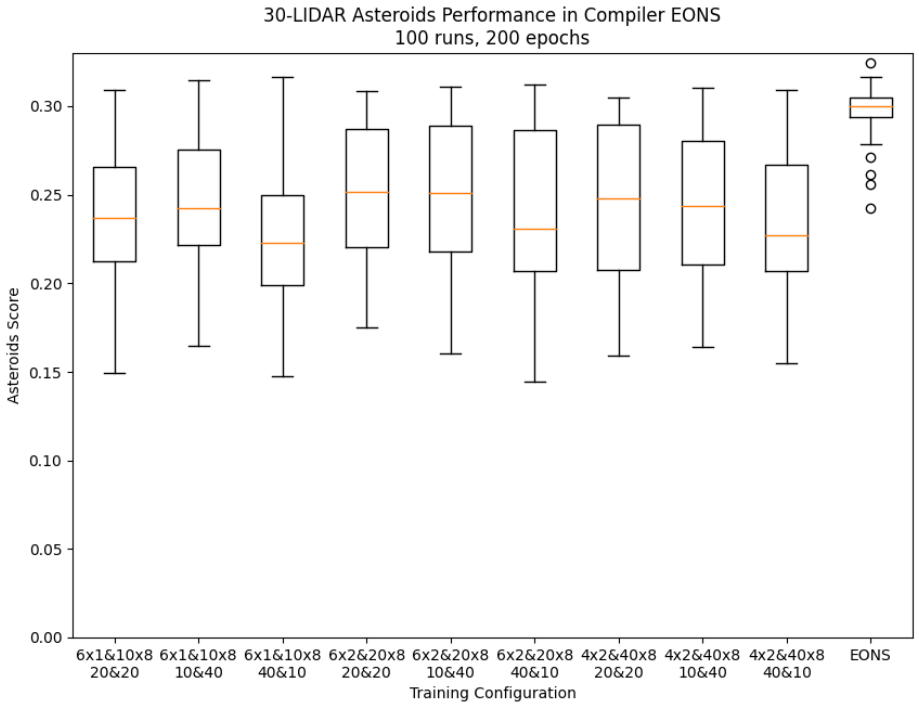
The TENNLab framework includes an application similar to Atari's Asteroids game and uses the same name (see fig 5.2). The application consists of a circular agent represented by a spaceship which can move around the screen and shoot projectiles, and multiple round, moving, asteroid-like obstacles that can impact the player ship and be shot. The player has actions to turn left, turn right, thrust forward, or 'warp' which instantly transports the player to a random area of the screen. The inputs to the player come in the form of 30 lidar inputs which are evenly spaced around the player ship. The fitness score is determined by the number of asteroids which were destroyed before the player ship collides with an asteroid.

The lidar in this application are assigned at incremental angles, which means that grouping adjacent input neurons may create usefully related subnetworks. Using a 2-bin flip-flop encoder, solution networks have 60 input neurons, which is higher than many of the tasks that EONS excels at, however asteroids is an application that EONS has successfully trained on in the past. This experiment will examine the training differences between EONS runs against the compiler-enhanced EONS workflow.

The compiler configurations consist of 9 runs, divided by 3 different sets of subnetwork sizes, each with 3 different subpopulation sizes. Runs 1-3 have an input subnetwork with 6 input neurons and 1 output neuron, and the 10 instances of that subnetwork connect to an output subnetwork with 10 inputs and 8 outputs. Runs 4-6 are similar in having 10 instances of an input with 6 inputs, but have 2 outputs each, with the output subnetwork having 20 inputs and 8 outputs. Finally runs 7-9 use 15 instances of a 4-input, 2-output input subnetwork connecting to a single 40-input, 8-output subnetwork. The first run in each subnetwork set has equally sized subpopulations for each subnetwork type with 20 members each. The second run in each set has 10 members in the input subpopulation and 40 members in the output subpopulation, whereas the third run inverts this to have 40 members in the input subpopulation to 10 in the output subpopulation.



(a)



(b)

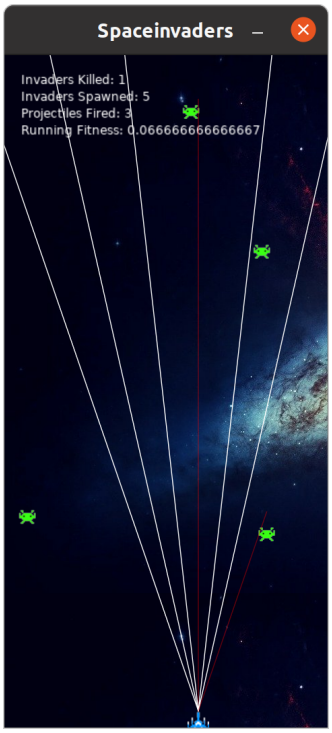
Figure 5.2: The application visualiation for the TENNLab Asteroids application (a) and fitness scores for the EONS runs with and without compiler-assistance (b)

The runs were completed on the TENNLab neuro cluster, with the best performance represented by EONS without compiler assistance (see fig 5.2b). Out of the compiler runs, there appears to be a drop in accuracy for the three configurations with the small subpopulations allocated to the larger output subnetwork. All of the compiler-assited configurations generated top networks much larger than the EONS networks. Overall, the differences between configurations may not have been large enough to extra meaningful differences. Future runs for this application should focus on larger variation between compiler configurations to see if any useful patterns emerge.

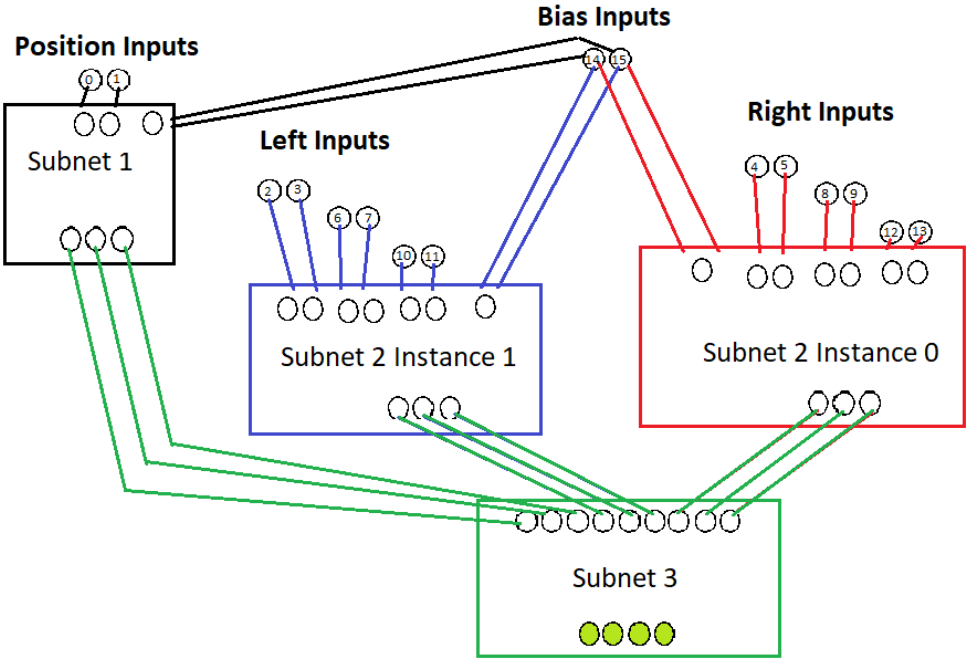
5.2.2 Space Invaders

The TENNLab framework includes an application loosely inspired by Taito's Space Invaders game and uses the same name. The application consists of a player object at the bottom of the screen and 'invader' sprites which come downward from the top of the screen (see figure 5.3a). The player has actions to move left, move right, and fire a projectile upward. The inputs to the player come in the form of lidar signals along with an indication of where the player is located in the x-dimension. The application has a maximum runtime and spawns invaders regularly through the runtime. The score for the application is the product of destroyed invaders and runtime divided by the product of the number of spawned invaders and the maximum runtime.

The lidar in this app are placed on the player agent in incremental angles from the vertical position. Odd counts of LIDAR include a vertical lidar, whereas even counts do not. Previous experiments with this application have resulting in networks that scan the field and fire upwards once an invader crosses the central lidar, resulting in a guaranteed kill. To make this experiment more challenging, I have set the parameters to not have a central lidar. This will create 6 lidar alternating sides: lidar 0 is be the center-left lidar, 1 is the center-right lidar, 4 is the far-left lidar, and so on. The application sets up the inputs with the position information first, then the lidars



(a)



(b)

Figure 5.3: The application visualiation for the TENNLab Space Invaders application (a) and compiler instructions for space invaders training (b)

in order, and finally a bias value to be spiked in regularly. For this experiment, a multiple spike encoder is used, with 2 flip-flop inputs with spike trains up to 12 spikes long, and minimum-valued input spikes are not ignored. EONS is run with typical hyperparameters, using tournament selection to choose the top 4 networks, a 90% mutation rate with weights having roughly double the mutation weight as delay.

This setup is simple but challenging for compiler-assisted eons to use easy instructions as 'adjacent' data is largely unrelated. Instead, I chose to use a more complex set of instructions utilizing override links to reroute connections (see fig 5.3b). This experiment uses 4 instructions: a simple generated subnetwork to act as input neurons, a positioning subnetwork with 1 instance, a lidar subnetwork with 2 instances, and an output subnetwork. This configuration should also test the effect of a smaller search space and restrictions to the network topology.

The generated input subnetwork 0 will span the 16 inputs and will allow for inputs information to be repeated to multiple subnetworks. As normal for simple generated subnetworks, these neurons will have the minimum threshold for the PropertyPack which in the case of the basic RISP parameters used is -1.0. These neurons are each input and output neurons within the subnetwork and this subnetwork 0 contains no synapses. This subnetwork, like all generated subnetworks, fulfills a completely structural role and is not evolved. The string form of the command for this instruction is "i -1 /i g".

Subnetwork 1 processes the positional information in the overall network and serves as subpopulation 0 in EONS, and has three inputs and outputs. The first two input neurons of the network are connected to the two, binned, positional inputs from the application (0 and 1). The last input is connected to both of the bias neurons passed from the application to the input sublayer(14 and 15). There is only one pair of positional data inputs, so subnetwork 1 only will use a single instance. The string form of the instructions for subnetwork 1 is "i 0 /i n 1 ol 0 0 0 0 ol 0 0 1 1 ol 0 0 2 14 ol 0 0 2 15 ".

Subnetwork 2 handles the lidar inputs, and is split into two instances, one for each side of the player ship and acts as subpopulation 1 in EONS. The subnetwork has 7 inputs matching the 3 pairs of lidar input neurons and a bias neuron input. Subnetwork 2 serves 3 outputs, like subnetwork 1, chosen arbitrarily. The inputs are connected using override links, using instance mapping to ensure that connections go appropriately between the left and right networks. The string form of the fully-overriden instructions for subnetwork is "i 0 /i n 2 ol 0 0 0 2 4 ol 0 0 1 3 5 ol 0 0 2 6 8 ol 0 0 3 7 9 ol 0 0 4 10 12 ol 0 0 5 11 13 ol 0 0 6 14 14 ol 0 0 6 15 15 ".

Subnetwork 3 is the final EONS subpopulation and takes the processed input information and handles the final network's 4 outputs back to the application. The 9 application outputs from subnetworks 1 and 2 are passed into the 9 matching inputs of the sublayer. The inputs from the two instances of the lidar-handling subnetwork are handled manually; handling with standard inputs is now possible but at the time of testing input mixing could not be disabled. The string form of the final instruction is "i 1 /i n 1 ol 2 0 3 0 ol 2 1 4 0 ol 2 2 5 0 ol 2 0 6 1 ol 2 1 7 1 ol 2 2 8 1 o".

Early testing of 6-lidar space invaders proved the problem to be challenging. Given the challenge, a great deal of compute time was allocated to the project, and sets of 200 runs with a population of 256 evaluation networks for 1000 epochs. EONS was performed on the TENNLab cluster of Dell PowerEdge C6145's and the runtime and performance of each run of each epoch was captured, along with the best network of each run. For compiler-assisted runs, the best subnetworks were also captured. For this experiment, two sets of runs will be tested: one with regular EONS and one with the compiler-assisted EONS. These sets of runs have been completed but the initial EONS runs only had a population of 100 and will need to be re-run to match the later compiler-based runs.

The runs were completed on the TENNLab cluster, with the best performance by non-compiler assisted EONS (see fig 5.4). Similarly, the training time and final network size are substantially larger in the compiler-assisted EONS. Network accuracy differences appear to be more pronounced than the asteroids results. It is possible

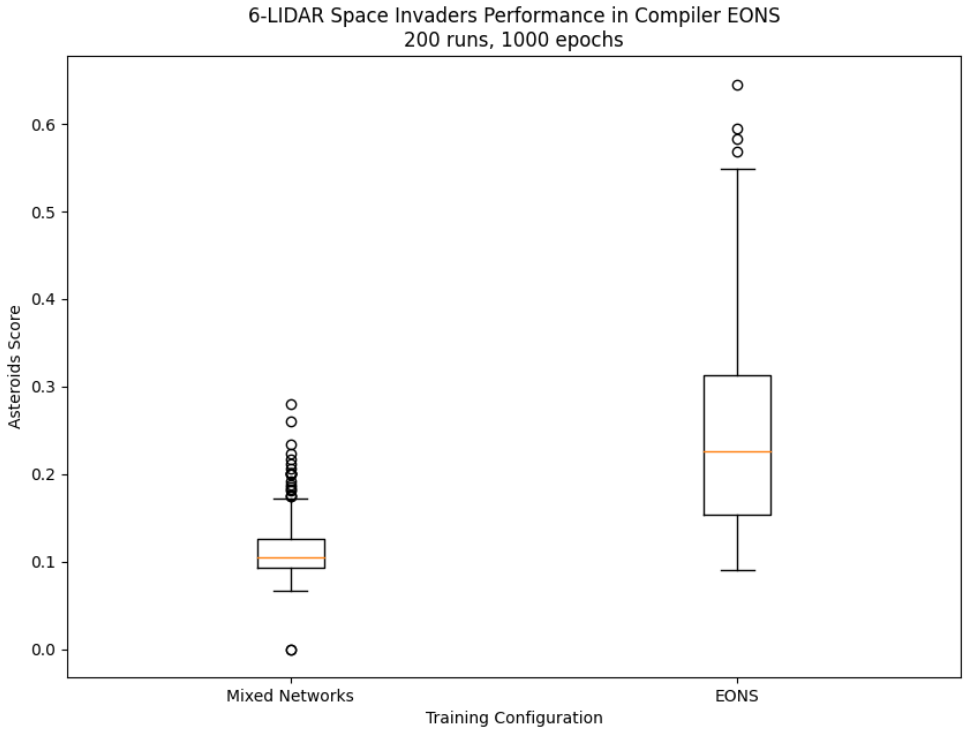


Figure 5.4: SpaceInvaders Score

that the extremely small subpopulation size of 8 had a more noticable effect than in the asteroids runs. It may be worthwhile to examine subnetworks sizes that are more directly comparable for future runs, rather than keeping final network evaluations the same across configurations.

5.2.3 MNIST

The MNIST handwritten digits dataset is a public domain dataset made up of images of handwritten digits [29]. Each entry represents a 28 by 28 pixel 8-bit greyscale image of a single hand-written digit. The dataset is available in many different formats, and in this case a .csv converted version was chosen for easy compatibility with the TENNLab classify app. This collection is split into two .csv files, a training set with 60,000 labelled entries and a testing set with 10,000 labelled entries. Each entry consists of a 0-9 valued label and the 784 pixel values ranging from 0-255. For training and validation, I split the training data into two files, one with the labels, and the other with the pixel values for each entry.

For training, the same encoding scheme of 2 bin flip-flop per input with up to 12 spikes per bin is used, making a total of 1,568 inputs per network. Each configuration of run uses 200 epochs and a total evaluation population of 400 networks, with 250 runs per configuration. Being an image classification problem, this seems an ideal environment to test the grid mode for the compiler. As with the above examples, the experiment involves at least one compiler configurations and a non-compiler configuration for EONS. In this case for training only the training data file is used, and a 10%/90% train/test split is used to speed up training. In this case due to the image-based nature of the dataset, I am using a grid-based connection scheme for the compiler-assisted EONS runs in the experiment (see fig 5.5), so each of the input-processing subnetworks acts to examine a square rather than linear section of the image. The asteroids experiment already covered differing subpopulations sizes for this scale of total evaluation populations, so this experiment only uses a

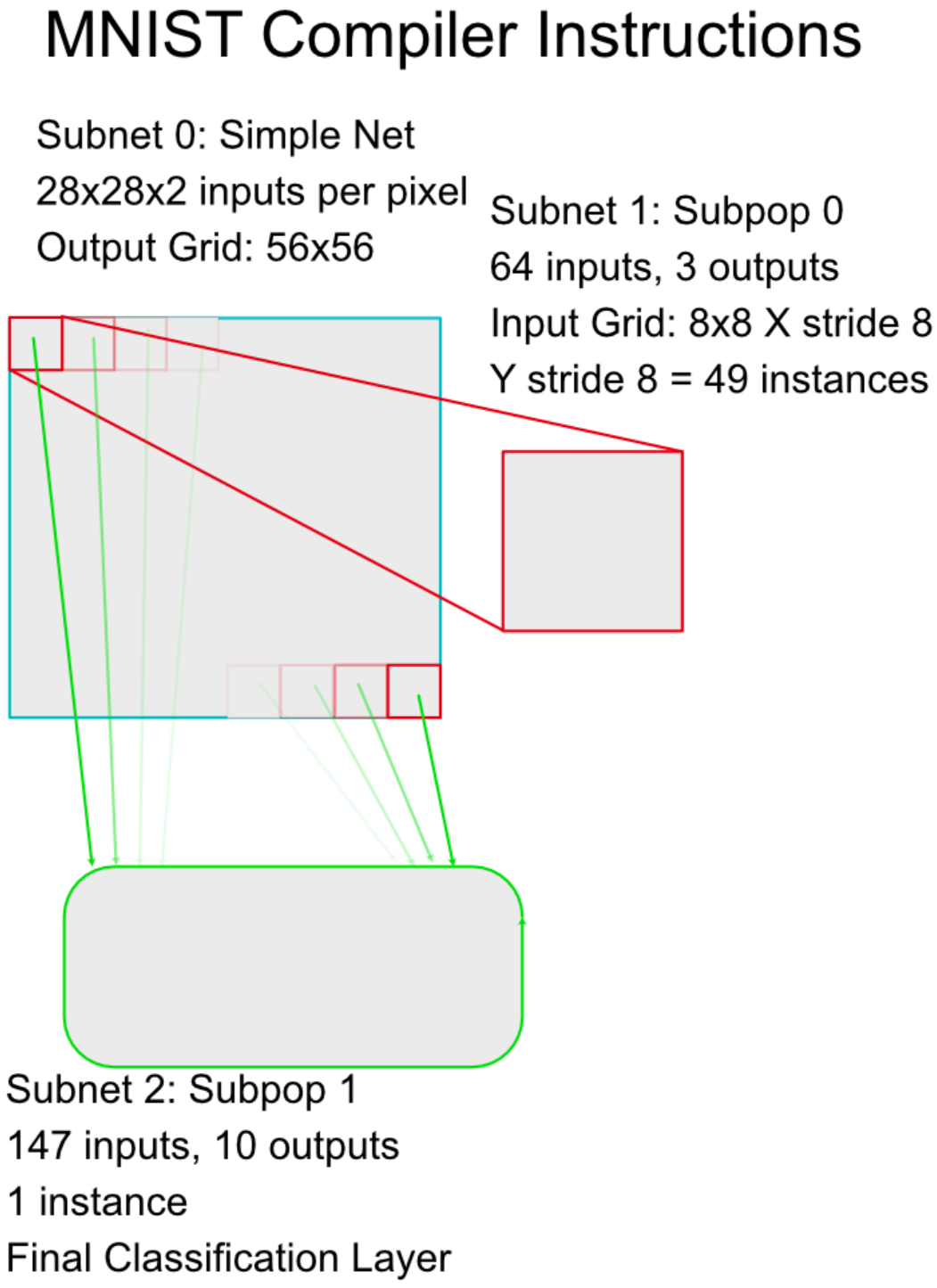


Figure 5.5: MNIST compiled configuration

subpopulation size of 20 for each subnetwork.

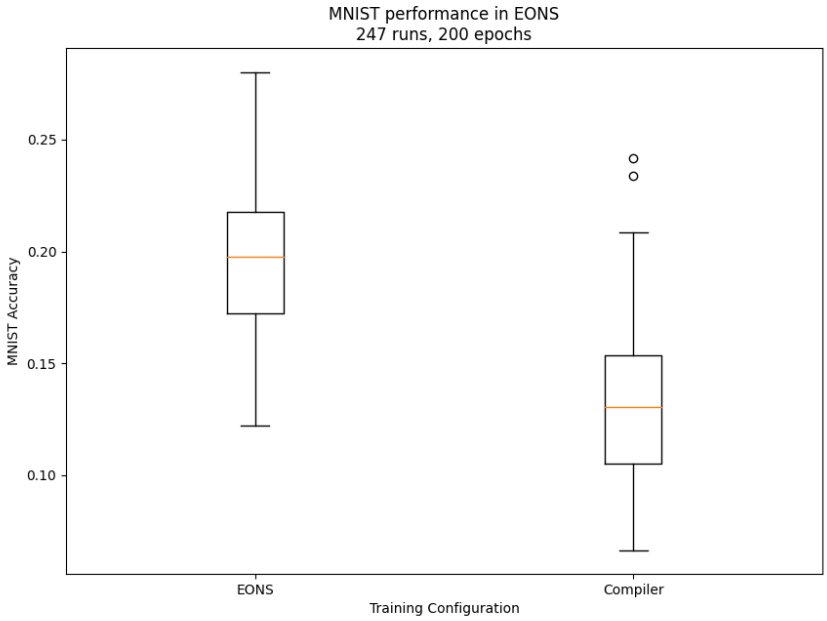
During evaluation, some runs were not successful either due to failures in the aging hardware used or rare code failures, so the evaluation is performed on 247 total runs. Like the previous tests, the compiler-assisted EONS performed worse than standard EONS runs (see fig 5.6) in both training time required and training accuracy.

5.3 Larger Subpopulation Test

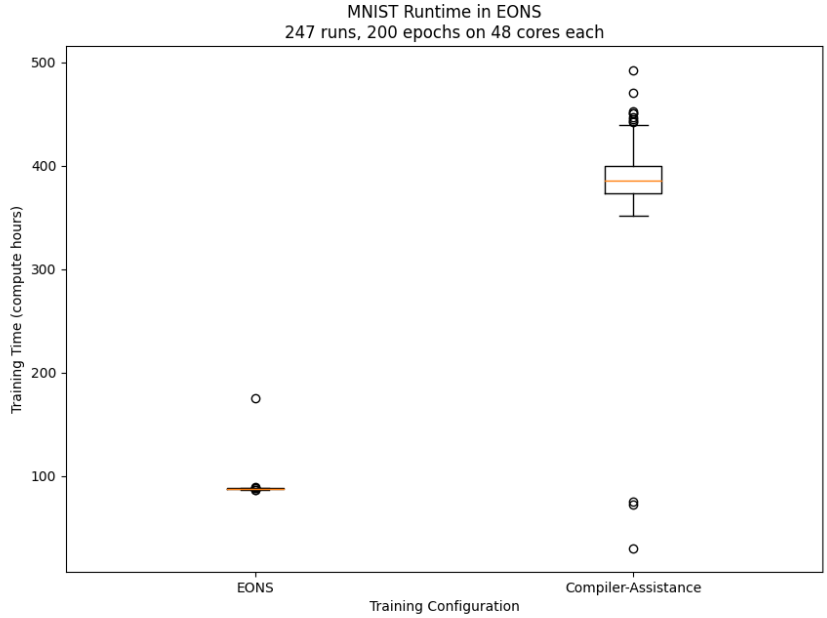
One likely cause for limited performance in these experiments may be the limited size of the subpopulations used for training multiple subnetworks at a time using EONS. The way that EONS generates networks using the settings of these experiments, the initially-created networks tend to be extremely sparse. SNN's do not guarantee the transmission of information due to the nature of the binary threshold function within the LIF neuron. If there are no spikes from some of the output neurons in the input subnetwork, there are sharp limits on the activity that can occur from the output subnetwork if it is also sparse. These sparse networks are extremely efficient in terms of SWaP costs, but in situations where information needs to flow to a second network it may be more challenging to find a candidate. In this case, a subpopulation size of 100 instead of 20 allows for more information for EONS operations to train subpopulations with. This increases the evaluation population for 400 to 10,000.

This experiment uses the Asteroid application, with value with a larger input space which can still be trained in a reasonable span of time. Each experimental run uses 200 epochs. In addition, this experiment also measures more configurations of subnetwork size. 7 different subnetwork 0 input sizes and 5 different subnetwork 0 output sizes across 9 different compiler configurations provides more opportunities for different results than the limited initial test.

Due to the long training time needed for these experimental runs, this experiment only uses 5 EONS training runs for each of the 9 different configurations tested. Compared to the experiment in section 5.2.1, the evaluation population is 25x larger,



(a)



(b)

Figure 5.6: The training fitness (a) and speed (b) for EONS with and without compiler-assistance

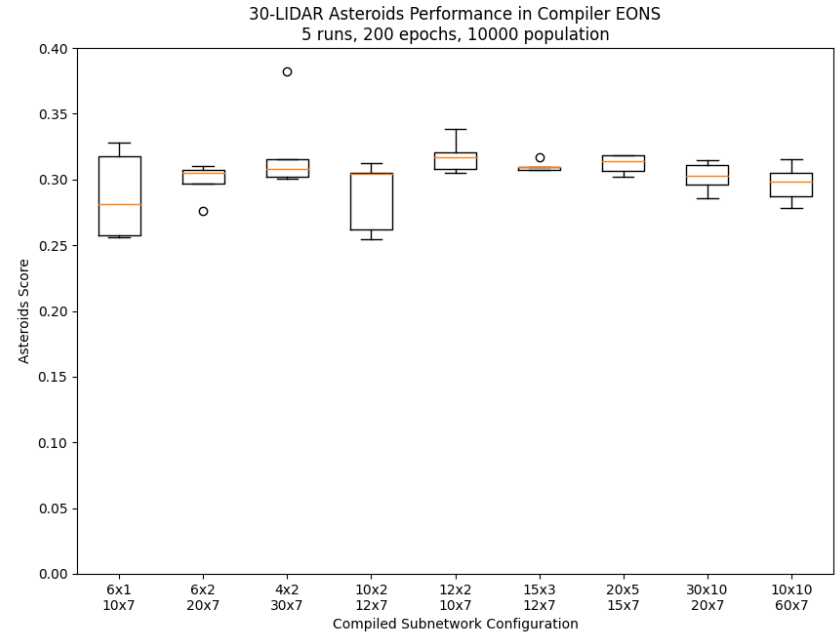
but with only 1/20th the number of runs and 9 parameters compared to the original 10, the total experimental time should be only approximately 12.5% longer, which is likely within the margins of error given how extreme outliers have been in previous experiments.

The fitness values in these larger training runs were largely better on average than the 400 population runs (see fig 5.7a). Average runtime per parameter was more than 25 times greater, however with higher scores it is possible that the agent survived longer per run which could have influenced runtime as well. Further testing is required to determine the contribution of network size, network success, and evaluation population to the total runtime to determine the relationship of each metric. In any case, for a problem such as this, there appears to be value in fewer total runs with larger evaluation populations.

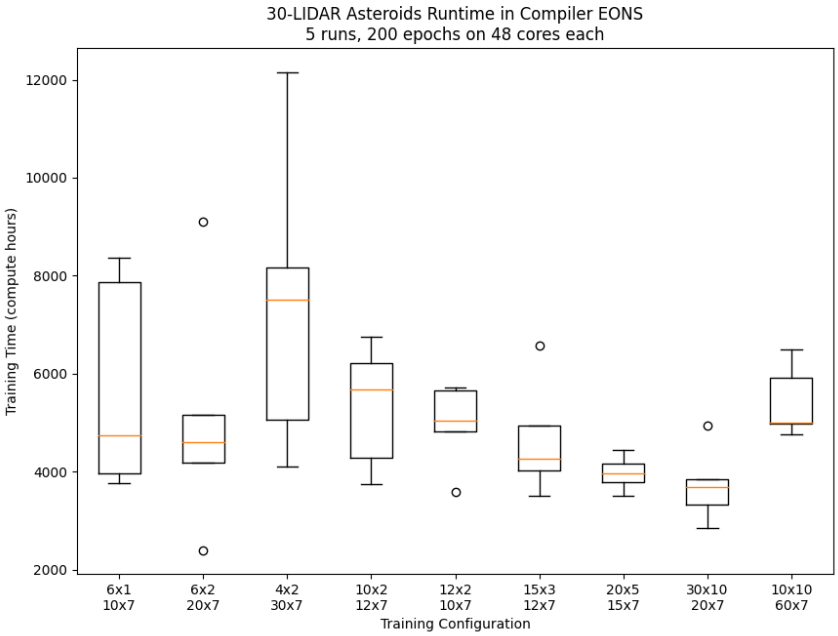
5.4 Future Work

One challenge for compiler-assisted EONS is that maintaining a sufficiently large population for each subnetwork results in many evaluations that must be performed each epoch, many of which may be duplicates from previous epochs. It may be possible to reduce some of the redundant work by generating a hash or unique identifier of each network and caching previously solved work. The subpopulations in these tests have been very small which dramatically increases the number of redundant evaluations across epochs. This should be a fairly easy experiment to run, and comparing with traditional EONS may reveal information about the existing algorithm.

The compiler can also improved by improving more network generation types. One example I am planning on including is a modified form of DB-Scan clustering ([33]) designed for spiking neural networks without differentiating between different clusters. This algorithm works similar to Rizzo's spiking counting networks ([1]), but instead connects all groups of inputs to neighbors with a set spacing. As such



(a)



(b)

Figure 5.7: The training fitness (a) and speed (b) for compiler assisted EONS on larger-population asteroids runs

this network type has some feature similarity to counting networks and time-multiplexing options but actually forms one large, inter-connected network which would be tedious to route for the user and must be re-generated for every different size of network. This kind of network is an example of the layer usage of the compiler without needing instancing features – though the grid options are likely still useful despite only a single actual instance being created.

Chapter 6

Time Multiplexing

Part of the draw of neuromorphic computing is the ability to solve problems and approximate functions with very limited SWaP resources. The network compiler may aid in increasing the performance of training, but by combining networks by adding in even more inter-network synapses, it expands the amount of computing resources required by the finished network. To constrain the space of a problem, the temporal nature of neuromorphic devices allows us to shift spatial resources into temporal ones, solving the same problems slower but with less resources. Similar to reducing parallelism in standard computing resources to solve problems sequentially, eliminating duplicate subnetworks in the compilation process and using synaptic delay to push spike trains further in time may allow for reuse of the same structure repeatedly without a change in final network results (see fig 6.1).

The output of a time multiplexed instance is necessarily spread across time in the same way that the input values are. However, in instances where a larger network would normally spread multiple instances across the inputs of a single subnetwork, an inverse multiplexing operation would be required to reverse the effects of the input delays. This should result in roughly simultaneous input in the subsequent network. If that network also demands inputs from other networks, then the user should specify sufficient delays for those other networks as well. In this case, the process will be

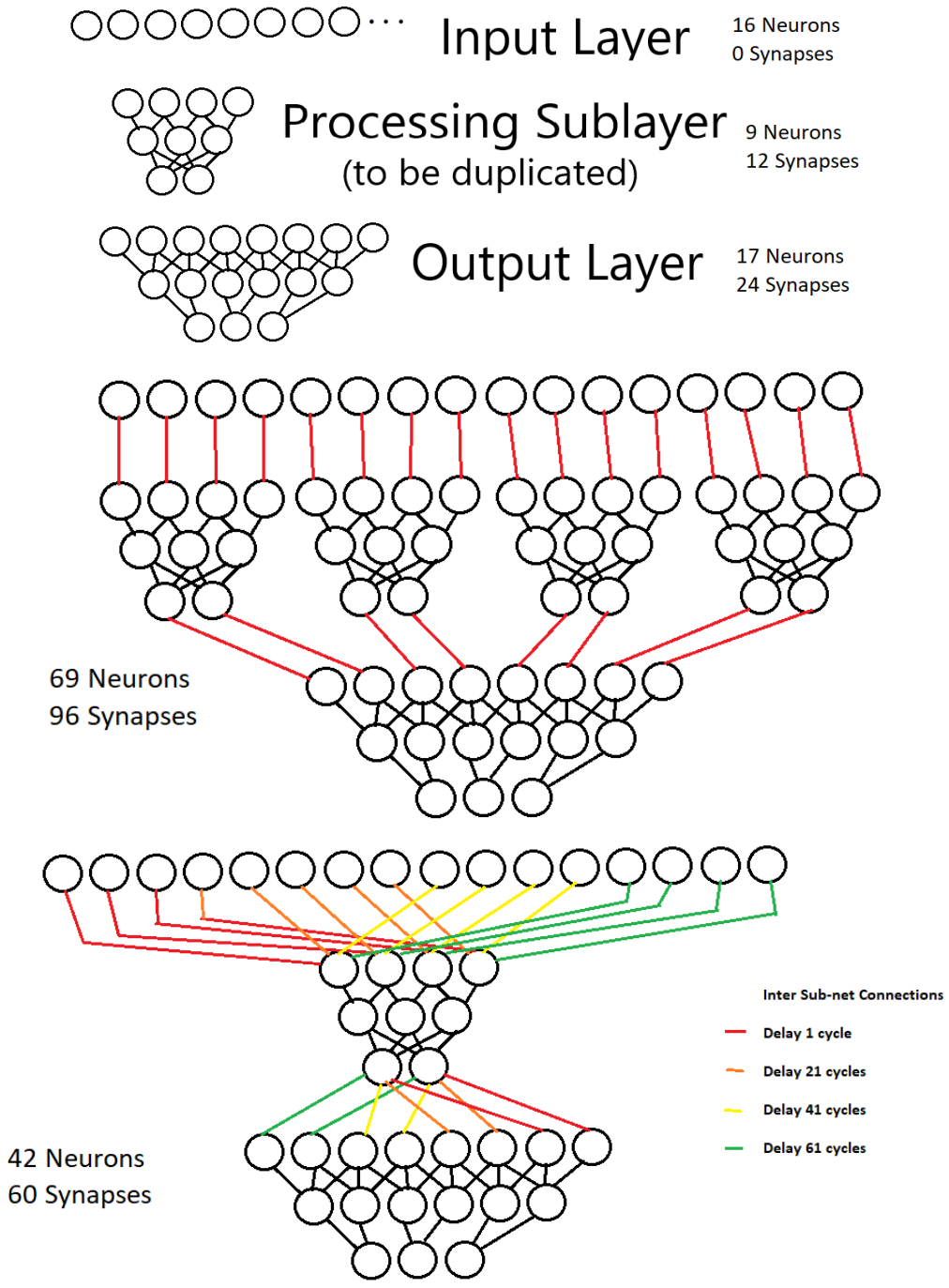


Figure 6.1: Compiling 3 networks using instances versus time-multiplexing

referred to as time compression.

One additional challenge of time-multiplexing is that the long delays required can easily exceed the maximum synaptic delay of many neuroprocessors. While the compiler can easily correct this by adding additional neurons and synapses in a chain or other structure, it is easy to construct a situation where the number of graph elements exceeds that of a non-time-multiplexed network due to the long delay chains (see fig 6.2). Conversely, an insufficient per-run delay value may result in merged activity between two temporal instances, which is unlikely to be desired behavior. Time multiplexing can reduce the space needed by a network but careful planning by the user is required (see table 6.1).

The overall effectiveness of time multiplexing is in reducing the number of neurons and synapses within a subnetwork. To achieve noticeable benefits, the subnetwork must be large. If the neuroprocessor design puts limits on maximum delay or maximum connectivity per neuron, time multiplexing will not only increase the runtime but also be less efficient in terms of network size. Understanding the nature of the neuroprocessor used is essential in using time multiplexing to increase efficiency.

6.1 Building Time Delay

The time multiplexing process determines the number of instances that would normally be created (see Chapter 5) as usual for network compilation, but only creates a single copy of the current network. When the compiler begins to generate inter-network connections for the time-multiplexed subnetwork, new connections are created and added to the same input neuron as normal, but all on the same subnetwork instance. Instead of 1-delay simple synapses, a connection pathway with a total delay of the subnetwork instance id that the connection would normally go to multiplied by a user-specified time-delay. The time delay chosen should be a sufficient number of cycles for the network to perform its operation. For example, given a chosen time delay of 20 cycles, connections assigned to the first time-multiplexed

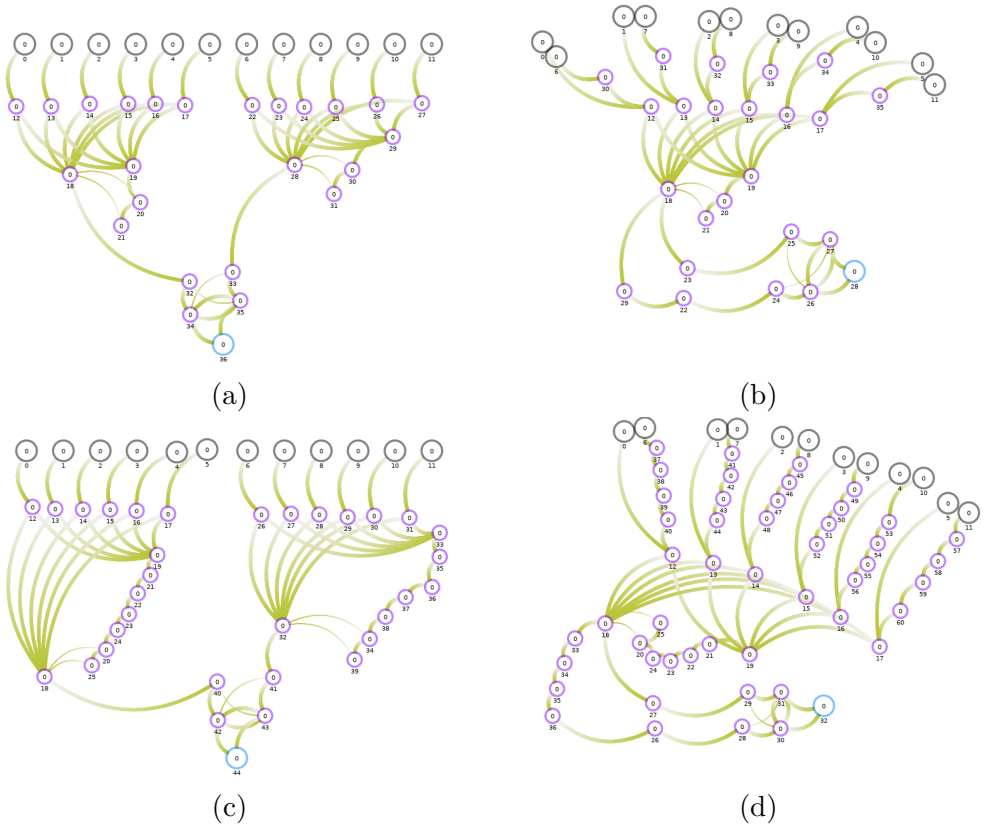


Figure 6.2: Compiling with (right) and without (left) time multiplexing with simple time compression for periods of 5 (top) or 20 (bottom) cycles for an XOR of counting networks compilation task.

Table 6.1: Time Multiplexing example network sizes

	5 cycle		20 cycle	
	Standard	Tmux	Standard	Tmux
Neurons	37	36	45	61
Synapses	54	47	62	72

pseudo-instance would default to the minimum delay allowed by the neuroprocessor. The connection pathways for the second pseudo-instance would have an additional 20 cycles of delay and so on for each pseudo-instance.

These connections are referred to as connection pathways instead of just synapses as time multiplexing frequently exceeds the maximum delay permitted within a neuroprocessor. In these cases, the connection is divided into $\lceil \frac{t}{md+1} \rceil$ synapses connecting the same number minus 1 simple neurons (minimum threshold, no leak if optional), where t is the total delay of the connection pathway and md is the maximum delay permitted.

This will result in only a single set of the subnetwork outputs, and the output spikes are necessarily spread across time following a similar delay scheme to the inputs. However, the time multiplexing command does not build any sort of 'reset' structure to ensure that the internal state of the time-multiplexed subnetwork is brought to an expected state before each temporal pseudo-instance. It is important to understand the behavior of the subnetwork that is being time-multiplexed for a variety of inputs, and how that state may persist across time. It may be desirable to alter the design of the input subnetwork to include reset functionality.

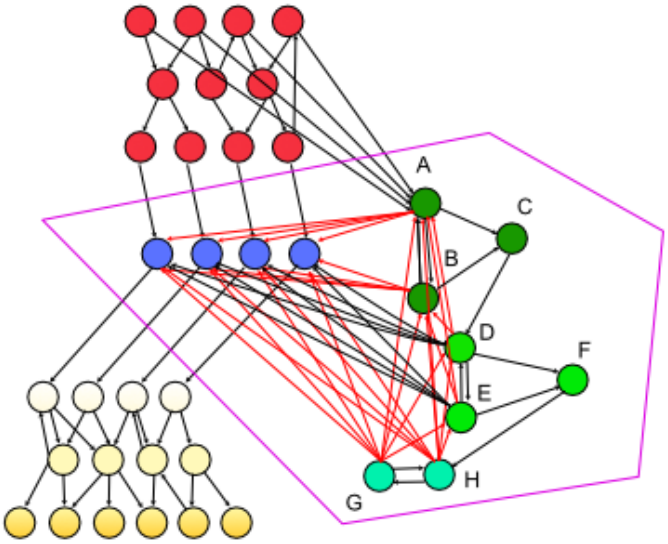
6.2 Time Compressing Outputs

Rather than space the outputs of the network over an extended time period on a small number of neurons, the compiler can instead compress the timeframe of the resulting spikes onto a different set of output neurons from each source pseudo-instance. With the addition of the time compression flag, the compiler alters the original subnetwork by adding additional output neurons equal to the number of original output neurons multiplied by the number of time multiplexed pseudo-instances the compiler is using (see fig 6.2). These outputs are set to be the outputs of the network, rather than the original neurons. The threshold of the these new output neurons is equal to 1 + the pseudo-instance they are part of.

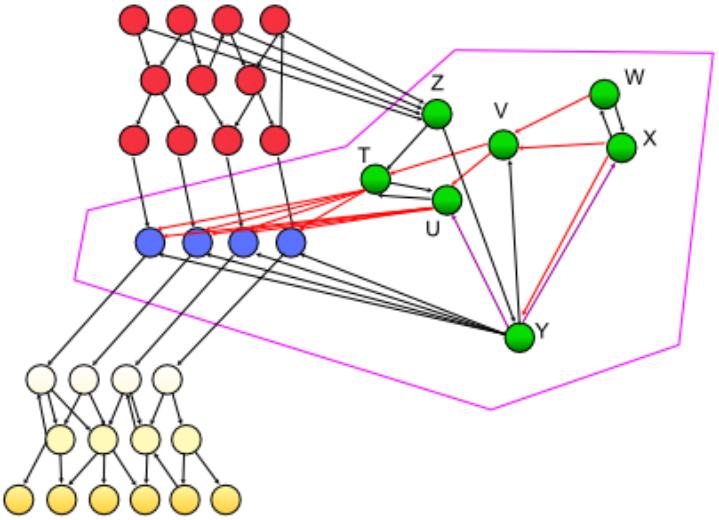
To connect these new outputs to the network, connection pathways are generated like the time multiplexed connections above. However now the longest delays are connected to the first psuedo-instance, and the synapses with minimum delay are connected to the final pseudo-instance. Like most connective synapses, these are also generated with maximum weights.

With the only means of separating fires in this design being the threshold of the neuron, this model of time compression only works when each temporal instance would only result in a single fire per neuron, such as in binary applications. This also would only function correctly when the neuron fires exactly once per temporal instance and from the same outputs each time, which only leaves the simplest set of spikes as output. It is possible to build a more powerful time compression layer abstracted to the settings of most TENNLab-compatible LIF neuroprocessors which adapts to the PropertyPack in use, but the implementation is beyond the time constraints of this work.

A more powerful time compression scheme would, in most cases, require delays, thresholds, and/or refractory periods based on the time multiplexing runtime. These values are far outside the standard values used in TENNLab network configurations, but should still be possible for simulated neuroprocessors. To capture an exact count of time within a network, cycle counting substructures will be necessary (see figure 6.3). For example, in 6.3a neurons A and B are simple neurons with a threshold of 1 and max-weighted synapses connecting them in both directions. These neurons will alternate firing, with one or the other firing each cycle once they have begun firing. Each is connected to neuron C with a 1.0 weight synapse, adding 1 charge to neuron C each cycle. Neuron C has a threshold equal to the delay until the associated time-compression nodes should start accepting fires, minus any necessary delays for the mechanism to work. This small subnetwork is disabled simply via inhibitory connections from neuron D after being signalled by neuron C. This simple setup allows for precise timekeeping in the design, at the cost of very dense network activity and thus energy.



(a)



(b)

Figure 6.3: Time Compression support structures to allow for more flexible output options and better time restrictions. The time multiplexed network is in red, a single instance of the time compressed output is in blue with its support neurons in green. 6.3a shows a network which relies on high maximum allowed thresholds and 6.3b shows a network requiring high refractory periods and delays.

The design shown in figure 6.3a uses multiple of these time keeping subnetworks, and would require potentially very high maximum allowed thresholds to function. In this network, neurons A and B trigger time-counting in neuron C, as well as inhibiting the 4 associated time compression outputs to prevent their fire during another time multiplexed temporal instance. Neuron C has a threshold set to the delay that would normally be applied to the time-compression output neurons (blue) minus 6 before it fires into neuron D. Neuron D and E have another time accumulating purpose into neuron F, signaling the switch to active mode where the time compressed outputs should allow accumulation and fire into the outputs. Fires from these two neurons are intended to fight against leak in the blue neurons, but the connections from D and E into the time compression outputs can be removed depending on the neuroprocessor design. Once neuron F crosses its threshold the runtime period for this time multiplexed instance has passed, and neurons G and H inhibit the outputs as well as both other counting pairs.

The design shown in figure 6.3b uses delay and refractory periods to manage time within the support subnetwork. Neuron Z fires at the start of the first runtime with a refractory period equal to the time multiplexed delay until the associated temporal instance will start running. Firing into neuron T starts neurons T and U inhibiting the time compression outputs to prevent fires, and a fire into neuron Y does nothing as it has a threshold of 2. The second time neuron Z fires, during the desired activation period, it should have no effect on neurons T and U but will trigger neuron Y to fire. Neuron Y sends an excitatory spike to the time compressed outputs with a weight to bring the outputs to neutral charge with enough delay to wait for neurons T and U to stop their cycle. A spike to neuron V will result in inhibitory spikes to stop neurons T and U, where a excitatory spike to neuron U will restart their cycle with a delay equal to the time multiplexed runtime. a similarly delayed spike to neurons W and X will start another cycle to inhibit the other support neurons to prevent a much later reactivation of the output neurons.

The exact nature of the neuroprocessor and its PropertyPack will determine how best to construct support structures such as those in figure 6.3. If self-loops are possible in the neuroprocessor, then the pairs of cycling neurons can be replaced with a single neuron in most cases. Small maximum thresholds may restrict the construction of the 6.3a design, but alternately by building a structure of lower-threshold neurons can be assembled to act as a single, high threshold neuron at a potentially high activity cost. Similarly the long delays used in 6.3b design can be replaced by delay chains and the refractory period can be simulated using a self-disabling, inhibitory time count structure like in 6.3a. All of these designs are explicitly for configurations of IAF neurons used in a similar manner to those in the TENNLab neuroprocessors and may not prove compatible with other neuroprocessor designs.

Chapter 7

ConnectEONS

EONS builds networks by adding neurons and randomly connecting pairs of neurons, both in initial network generation and through mutation [2]. However, these mutations are random, meaning it is a common occurance for EONS to operate across the entire span of epochs with many neurons lacking incoming and/or outgoing synapses. With an event-based simulator this is not much of a performance concern, but it does mean that many genetic operations that EONS performs do not actually affect the performance of the network. By modifying the EONS network generation and genetic operations to work on synaptic connection pathways, it may be possible to reduce the number of epochs needed to learn, especially on larger problems(see fig 7.1). I will refer to this as ConnectEONS.

ConnectEONS creates connection path features, which consist of a series of neurons connected sequentially by single synapses. To allow users to have some control over the size of features without having to constrain the algorithm to specific size ranges, the creation of these features follows a geometric distribution. ConnectEONS generates pseudo-random numbers in the range $[0, 1.0)$ and adds to the size of the feature (usually a neuron and a synapse leading to that neuron) and generates a new number until the generated number is greater than a user-specified threshold. In this way, the pseudo-random number generator generates Bernoulli trials, adding features

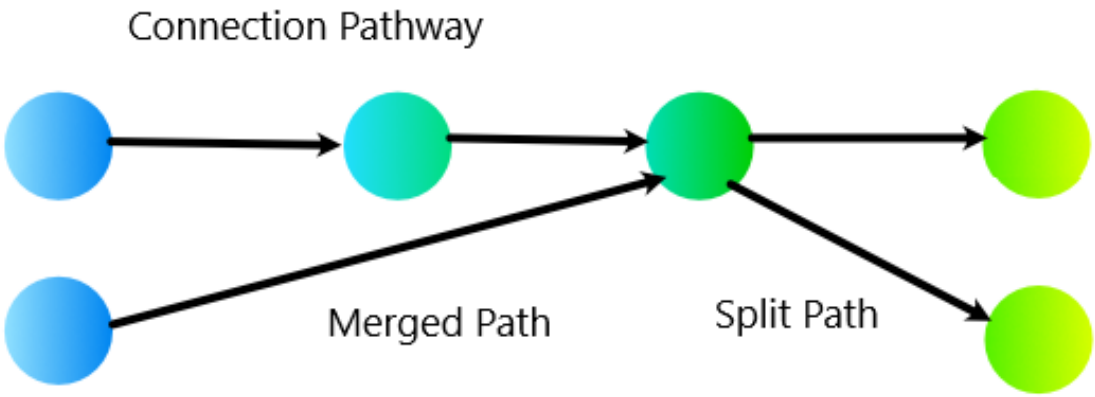


Figure 7.1: An example of a connection pathway with both a merge from the bottom input and split to the bottom output, from inputs (left) to outputs (right).

until a success is generated in a trial, indicating it is time to stop adding to the feature. The user specified rate p guides the size of these features, with an expected size of $\frac{1}{p}$.

ConnectEONS creates connection path features, which consist of a series of neurons connected sequentially by single synapses. To allow users to have some control over the size of features without having to constrain the algorithm to specific size ranges, the creation of these features follows a geometric distribution. ConnectEONS generates pseudo-random numbers in the range $[0, 1.0)$ and adds to the size of the feature (usually a neuron and a synapse leading to that neuron) and generates a new number until the generated number is greater than a user-specified threshold. In this way, the pseudo-random number generator generates Bernoulli trials, adding features until a success is generated in a trial, indicating it is time to stop adding to the feature. The user specified rate p guides the size of these features, with an expected size of $\frac{1}{p}$.

ConnectEONS builds and modifies network topology through connection paths. A path between two neurons consists of n synapses and $n-1$ neurons where $n >= 1$. A connection path from neuron A and neuron B is generated by generating neurons following the geometric distribution described above, and connecting them between A and B with synapses. Thus if the first Bernoulli trial is successful, a single synapse is added from neuron A to neuron B (see fig 7.2a).

ConnectEONS also uses a process called traceback. The traceback process operates starting on a temporary neuron T, and has a user-specified backtrace rate. The amount of steps the backtrace takes follows the geometric distribution described above using the user's backtrace rate. For each failed trial, one of the incoming synapses to the current neuron is selected pseudo-randomly, and the pre-synaptic neuron for that synapse becomes the new current neuron. This process continues until a successful trial, at which point the current neuron is returned (see fig 7.2b).

ConnectEONS begins network generation as normal by generating input and

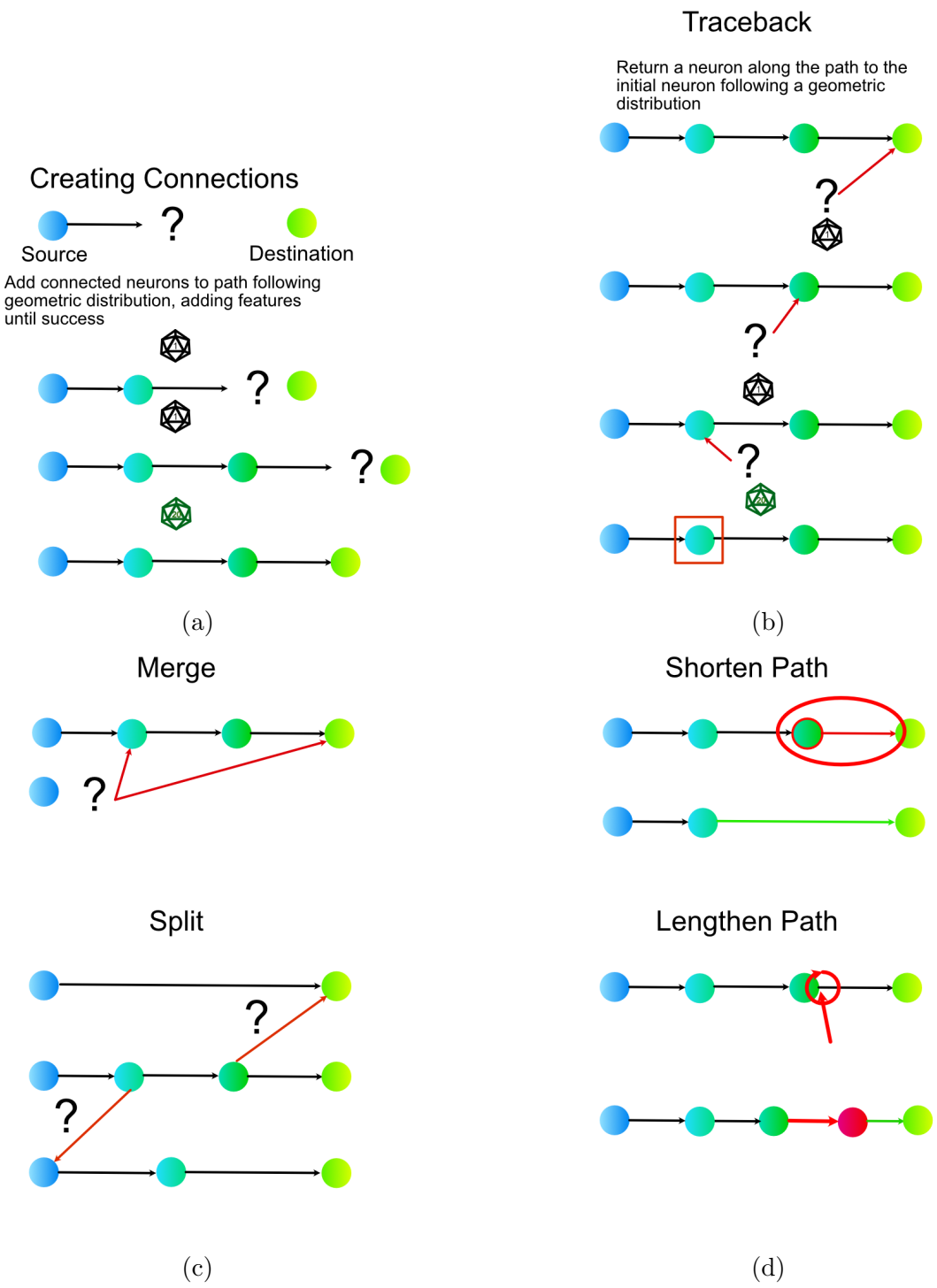


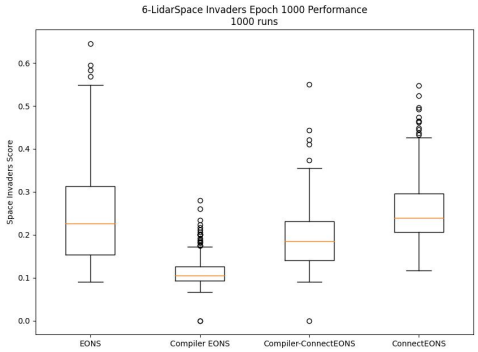
Figure 7.2: a) Demonstration of adding features while creating a connection path b) ConnectionEONS traceback operations using geometric distribution to select a neuron to return c) The merge and split operations which occur during path creation d) ConnectEONS path mutation operations

output neurons. To generate initial connections, for each possible connection from an input to an output ConnectEONS may create a connection path with probability set by the user's connect rate. Then for each connection, forks may be generated following a geometric distribution using the users fork rate. A fork is generated by selecting another output neuron, following a backtrace through its existing connections, and creating a connection path to that selected neuron. Once other connection paths have been generated for the target output neuron, connectEONS may join an existing path with probability based on the user specified join rate (see fig 7.2c) instead of creating an all-new path.

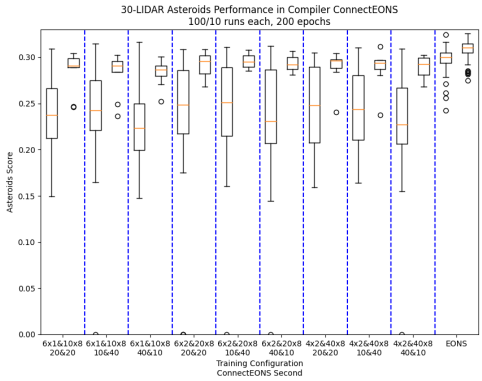
ConnectEONS does not utilize merge or crossover operations. Instead, only mutations are utilized for modifying networks. Instead of mutations to add a new random neuron or synapse, this process instead adds three new mutations: create path, lengthen path, and shorten path. Create path simply creates a path between two random neurons, allowing for additional feed-forward or recurrent connections. Lengthen path finds a random path into a random neuron, and adds an additional neuron to that path. Shorten path also finds a random path into a random neuron and shortens it by one neuron; if there were no neurons on that path, the synapse is removed instead (see fig 7.2d). These mutations alongside mutations to randomize or perturb node or edge parameters make up the sum of all operations to modify networks.

To evaluate the performance of ConnectEONS, the same applications are used as the experiments with the Compiler-assisted EONS. These first experiments exist both with and without the compiler assistance. Figure 7.3 shows the results of the experiment for Asteroids and Space Invaders. The Asteroids results are limited to only 10 ConnectEONS runs due to limits on resources at the time of the experiment.

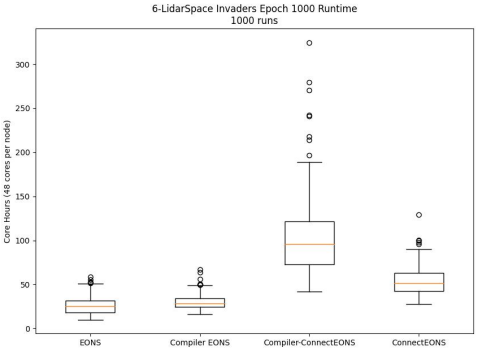
In both cases the fitness values were higher on average for Compiler-assisted ConnectEONS than Compiler-assisted EONS, while training time and network sizes were substantially higher in every configuration for ConnectEONS over EONS. ConnectEONS achieved slightly lower peak fitness than EONS for Space Invaders,



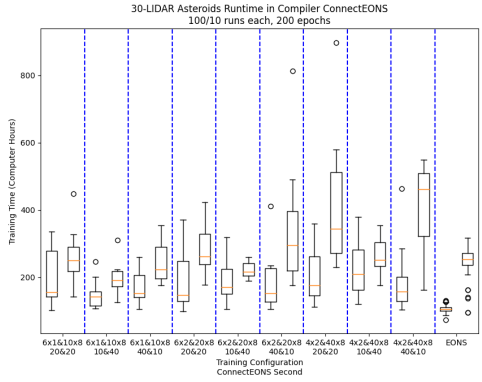
(a)



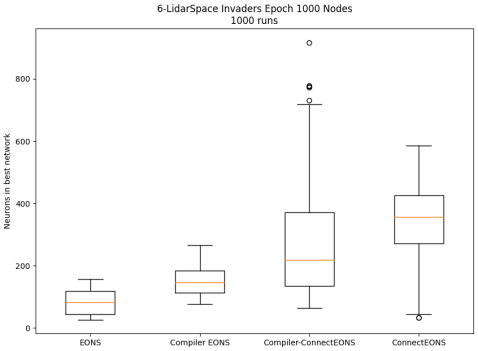
(b)



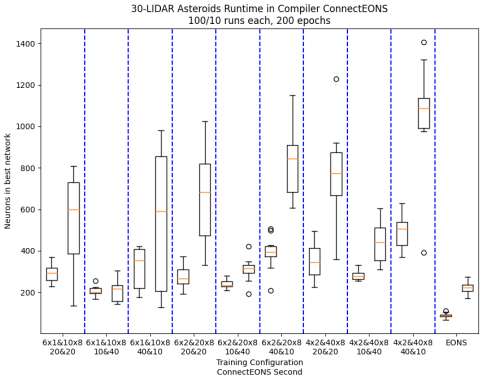
(c)



(d)



(e)



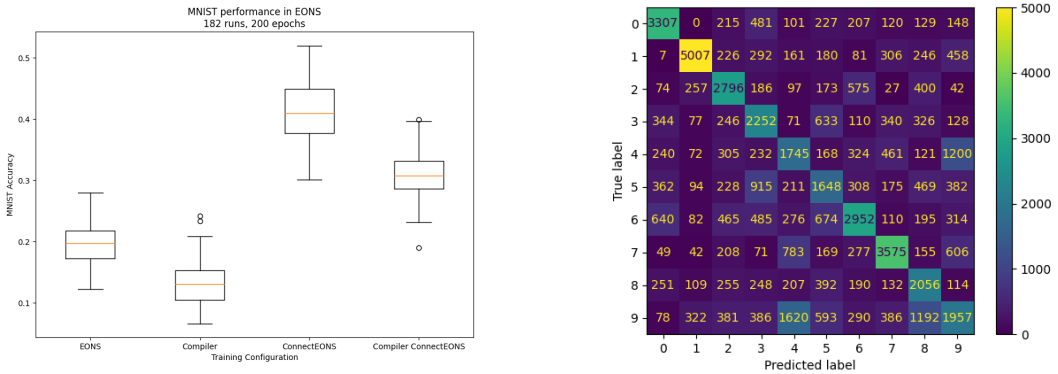
(f)

Figure 7.3: Results for Space Invaders (left) and Asteroids (right) for EONS, ConnectEONS, and Compiler assistance. For the Asteroids results, the ConnectEONS results are the right for each configuration.

but achieved slightly higher for Asteroids despite the lower number of runs.

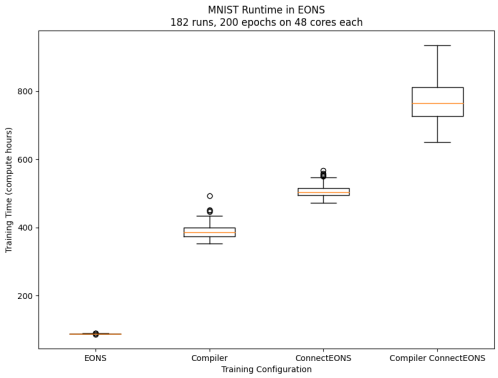
The goal of ConnectEONS is to use the general workflow of EONS but generating larger, more connected networks. This configuration of the TENNLab Space Invaders application only uses 16 input neurons, compared to the 60 inputs used in this configuration of the TENNLab Asteroids application. ConnectEONS seems to perform better relative to EONS on the larger problem size, but also with compiler assistance. More experiments are necessary to pinpoint the cause for a performance improvement using ConnectEONS with the compiler assistance but it may have to do with the small population sizes, where the more dense initial connection of ConnectEONS make more usable data transmission between subnetworks.

Completing the set of previous applications are the experimental results for the TENNLab classification application using the MNIST dataset (see fig 7.4). Following the trend seen in Space Invaders and Asteroids, the 1568 input neurons used in the MNIST classification using the encoding settings chosen for the experiment might show a further difference in ConnectEONS accuracy over EONS with a correspondingly larger runtime. As shown in figures 7.4a and 7.4c, those trends do continue. As in previous application runs, compiler-assisted ConnectEONS also continues to outperform compiler-assisted EONS. In this case the difference in accuracy and runtime are both substantial as are the differences in average network size (table 7.1). The EONS networks achieving an f1 score of nearly 0.3 with a mean synapse count of 55 is interesting, but the result serves to showcase the challenges of the current implementation of EONS on such large problems. The ConnectEONS f1 score is much higher than EONS, but at a cost of nearly six times the compute time and these results are still much less than state of the art performance for the problems [34] [35]. This experiment shows a positive progress for this type of relatively small, unorganized SNN for solving larger problems but there is a substantial potential for improvement. These experiments also serve to highlight the basic capabilities of ConnectEONS however the hyperparameters are based on default values that are



(a)

(b)



(c)

Figure 7.4: Results for MNIST using EONS, ConnectEONS, and Compiler assistance, a) training f1 scores b) top ConnectEONS test confusion matrix c) training times

Table 7.1: MNIST training run data

Configuration	Mean Neurons	Mean Synapses	Mean Core Hours	Max f1 Score
EONS	1596	55	88	0.297
ConnectEONS	4828	7511	506	0.530

entirely untested.

Each of these experiments use the same ConnectEONS hyperparameters as shown in table 7.2. The initial values were chosen at a whim to get initial experimental data, but the next natural experiment is to examine the effect of varying these hyperparameters. The final ConnectEONS experiment in this work is a grid search of four of these values at values of either 0.1 or 0.5 as shown in table 7.3.

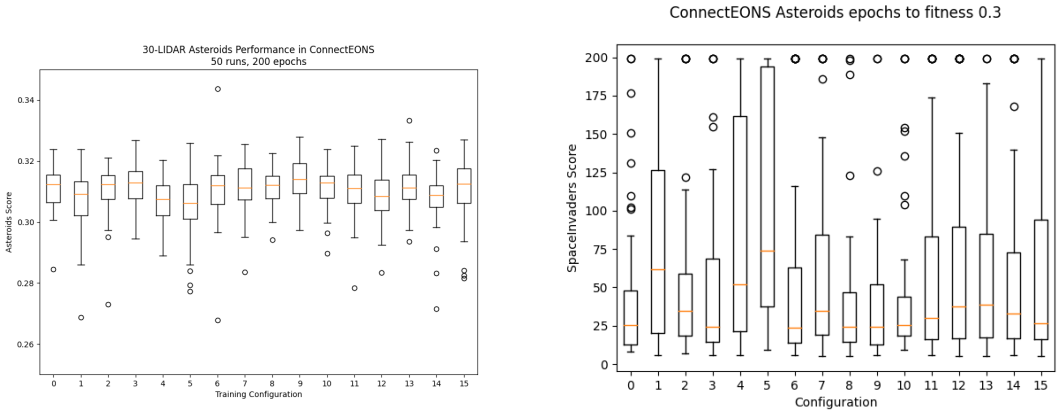
Each run used the same encoding, decoding, and application settings as previous runs. The experiment used 100 runs using 200 epochs per run as shown in figure 7.5. The fitness results are very inconclusive (see fig 7.5a) but the number of epochs each run took to reach a fairly high score of 0.3 (see fig 7.5b) shows that the 200 epochs were likely far too many. The highest neuron counts (7.5c) appear to correspond primarily to a high split rate especially when paired with a high connection rate. Conversely a high synapse count (7.5d) ties most directly with a high connect rate, with secondary consideration to a high split rate.

Table 7.2: Initial ConnectEONS Hyperparameters

Hyperparameter	Value
Connection Rate	0.25
Connection Fork Rate	0.1
Connection Split Rate	0.5
Connection Merge Rate	0.2
Connection Backtrace Rate	0.5

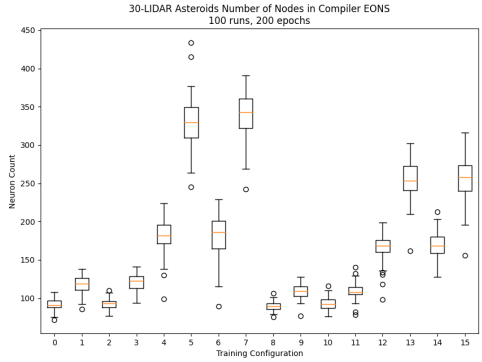
Table 7.3: Hyperparameter Grid Search Configuration Members

Hyperparameter	Low Members	High Members
Connection Rate	0,2,4,6,8,10,12,14	1,3,5,7,9,11,13,15
Connection Fork Rate	0,1,4,5,8,9,12,13	2,3,6,7,10,11,14,15
Connection Split Rate	0,1,2,3,8,9,10,11	4,5,6,7,12,13,14,15
Connection Merge Rate	0,1,2,3,4,5,6,7	8,9,10,11,12,13,14,15

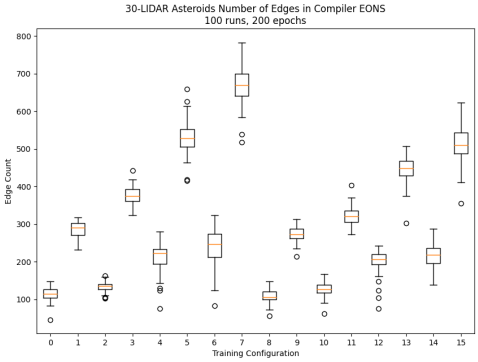


(a) (b)

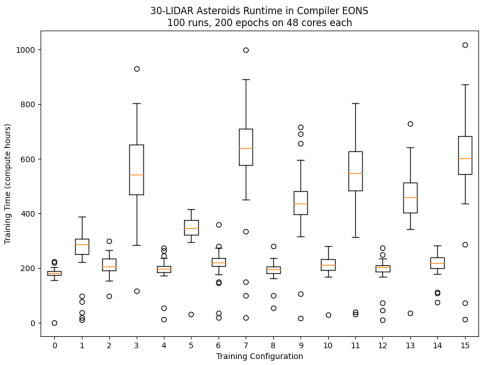
Figure 7.5: ConnectEONS Hyperparameter exploration asteroids score and training times.



(c)



(d)



(e)

Figure 7.5: (continued) Results for ConnectEONS Asteroids grid search

Chapter 8

Lessons Learned

This work explores the beginnings of multiple research angles in neuromorphic computing. The process of this exploratory work results in many lessons that may not be visible within the results of experiments or the outputs of a process. Some of what I have learned may hopefully be useful to those investigating these techniques in the future.

The Network Compiler is a powerful tool that tugs at the problem-solving and creative side of the user. In exploring the uses of the tool, I imagined many different features that time did not permit me to include within the compiler itself. Time multiplexing demonstrated a real need for a subnetwork modification instruction for the compiler to alter the loaded network to include a periodic reset feature. Such a feature could involve even a single neuron for neuroprocessors with permissive connectivity, reading input signals and sending a sequence of inhibitory and excitatory spikes to bring a network to default threshold and hopefully preventing any internal neuron fires during the reset process.

Features such as the proposed reset or time compression would benefit from a meta-network-scale input, bias, or timing command. The intent is to create a tiny subnetwork consisting of a pair of neurons connected to one another for each desired timing period, which take all of the final network's input neurons from synapses

and connects to the desired subnetwork inputs. Such an instruction could be of the format 'allin 2-0-n'. The first integer reflects the timing period, and the delay of the timing pair of neurons can accomodate any period (longer periods may require multiple neurons depending on max delay). The output synape(s) then connect to the desired input neurons, marked by the second integer in the command. The final command indicates if this should be considered an overriden neuron not to connect via standard methods (indicated by char 'y') or not (indicated by char 'n'). While this kind of network still does not perfectly mark runtime if there are not spikes in the first cycle, it would provide much more consistent spikes to individual subnetworks. Each subnetwork that uses the same allin period could share the same timing pairs, but different sets are needed for each demanded period (though even multiples can be accomplished with a carefully thresholded neuron on the connection path if needed).

One of the next steps that should be done in advancing the work on the network compiler, especially in the interest of Compiler-assisted EONS, is to replace the input structure from a vector of networks and a vector of instructions to just use a vector of instructions. By adding a new instruction mode to load a network file (or even to include a network json in a very long instruction), no functionality is lost. This makes to process of loading a network and generating a network more consistent, and allows for Compiler-assisted EONS to take advantage of pre-trained networks.

Designing network components for time compression showed how complex and expensive these components would be. In addition to higher power consumption, the number of additional neurons and synapses would likely eliminate the size benefits for smaller networks for time multiplexing. Some of these costs may be able to be reduced using a modified version of the all-in components described above but not many. However, a subnetwork that converts spikes from different time slices into a condensed layer to do neuromorphic work simultaneously may still be valuable for operating on time-series tasks. For such tasks, these fairly large, complex subnetworks may save EONS or other algorithms the effort of learning these highly complex topologies.

Compiler-assisted EONS is a powerful tool that has only begun being explored. In all of the small-population tests ConnectEONS outperformed EONS on average; this may indicate that building the necessary connection pathways for information flow is one of the larger hurdles in this kind of learning. The larger subpopulations tests showed how even with standard EONS for learning, trading number of runs for subpopulation size seems an effective solution style for more complex problems.

Time multiplexing is a potentially powerful optimization tool in reducing network sizes but it must be used very carefully. Larger networks, such as those trained in ConnectEONS, have more components to save per instance and provide better savings using time multiplexing. The main application of this feature is for resource-constrained neuromorphic devices which are operating at a much faster cycle speed than the rate of input data is arriving. The primary use case of repeated subnetworks thus far has been for input pre-processing. Reducing the number of subnetwork instances earlier in the final network will reduce the input space on subsequent subnetworks as well. While the time multiplexing process cascades into later subntworks, the outputs of the final network is also time multiplexed which may need to be processed differently than parallel outputs.

ConnectEONS shows promise as a useful training algorithm where the input space is larger than approximately 60 neurons. Based on experimentation, keeping the hyperparameters that grow the network quite small gives the best results in training accuracy and speed. The algorithm's main strength over standard EONS appears to be its inital network generation rather than parameter evolution. It seems likely that the best use of available computing resources for ConnectEONS is to train using a shorter number of epochs than what would normally be used in EONS.

Beyond the actual experiments of the work, some additional lessons are apparent. Implementing a small network that performs a simple task becomes challenging when the exact details of the LIF neuroprocessor the network can be installed on becomes variable. For other kinds of SNN component where the entire behavior of a neuron or synapse can change completely, a general way to implement the same functionality

may not even be possible. If catalogues of mini-networks or network components prove a fruitful path forward for the field, there will likely be necessary segmentation amongst those working with different component designs.

Chapter 9

Conclusion

This work explores some novel techniques for building and training spiking neural network topologies. The network compiler gives a user control over building larger network topologies from smaller pieces. Compiler-Assisted EONS extends that kind of control to training tasks where the pieces may or may not be known. ConnectEONS works in the EONS paradigm but generates more highly connected networks at generation for larger problems. Time multiplexing can take the complex networks and exchange resource costs for time. Each of these techniques are specialized tools but are useful for the correct task.

The network compiler revealed numerous uses to utilizing small networks for simple yet conventional computing tasks which have been developed in other works [32] [1] and combining them to larger operations. The field of neuromorphic computer cannot be divorced from the development of neuromorphic hardware. The ability to keep as many processing tasks natively within a neuromorphic chip may be useful to future developers looking to reduce communication overhead between neurmorphic and traditional compute resources. Training multiple subpopulations within EONS requires multiplicatively higher runtimes than training a single network at a time, but in some cases these more complex networks can achieve higher fitnesses than an equivilent compute-time of runs without. The network compiler is a powerful tool,

and many of the uses of the tool have still scarcely been explored.

Time multiplexing alters how the network compiler handles a large number of inputs into a subnetwork by reusing a single instance over a longer timeframe. Depending on the costs and limitations of components within a neuroprocessor it is possible to construct scenarios where the time multiplexing process makes a network larger as well as slower. However, in many cases the reduction of so many subnetwork instances does reduce total computing resource costs. In situations where any additional processing done the outputs of the target subnetwork are not linked across instances, time multiplexing naturally applies to such subsequent subnetworks as well. In cases where the time multiplexing operation has to be undone, this work includes two time compression subnetwork designs which are not yet implemented.

ConnectEONS is an alternative to EONS that can provide better training performance on problems with an input neuron count of around 60 or greater. Experimentally ConnectEONS was demonstrated to match or outform the EONS algorithm on these problem sizes. Hyperparameter exploration suggests that reducing the count and size of connectivity features increases accuracy and performance. ConnectEONS seems very promising for approximating solutions to more complex problems without requiring millions of parameters or expensive hardware.

Building a topology for an SNN is difficult problem that scales exponentially with the size of the network needed. These challenges are exacerbated by resource constraints, timing requirements, and power limitations. The techniques in this work, and those they might inspire, are steps in solving more complex problems at lower costs.

Bibliography

[1] C. P. Rizzo, C. D. Schuman, and J. S. Plank, "Neuromorphic downsampling of event-based camera output," in *Proceedings of the 2023 Annual Neuro-Inspired Computational Elements Conference*, ACM, April 2023. vi, 9, 15, 23, 24, 48, 75

[2] C. D. Schuman, J. P. Mitchell, R. M. Patton, T. E. Potok, and J. S. Plank, "Evolutionary optimization for neuromorphic systems," in *NICE: Neuro-Inspired Computational Elements Workshop*, 2020. ix, 2, 8, 9, 60

[3] J. L. Hennessy and D. A. Patterson, "A new golden age for computer architecture: Domain-specific hardware/software co-design, enhanced security, open instruction sets, and agile chip development," in *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*, pp. 27–29, 2018. 1

[4] J. S. Plank, C. D. Schuman, G. Bruer, M. E. Dean, and G. S. Rose, "The TENNLab exploratory neuromorphic computing framework," *IEEE Letters of the Computer Society*, vol. 1, pp. 17–20, July-Dec 2018. 1

[5] C. D. Schuman, T. E. Potok, R. M. Patton, J. D. Birdwell, M. E. Dean, G. S. Rose, and J. S. Plank, "A survey of neuromorphic computing and neural networks in hardware." arXiv:1705.06963, May 2017. 1, 2, 3, 6, 7, 10

[6] H. T. Siegelmann, "Neural and super-turing computing," *Minds and Machines*, vol. 13, pp. 103–114, 2003. 1

[7] C. D. Schuman, T. E. Potok, S. Young, R. Patton, G. Perdue, G. Chakma, A. Wyer, and G. S. Rose, "Neuromorphic computing for temporal scientific data classification," in *Neuromorphic Computing Symposium*, July 2017. 1

[8] W. S. McCulloch and W. Pitts, "A logical calculus of the ideas immanent in nervous activity," *The bulletin of mathematical biophysics*, vol. 5, no. 4, pp. 115–133, 1943. 2

[9] F. Rosenblatt, "The perceptron: a probabilistic model for information storage and organization in the brain.," *Psychological review*, vol. 65, no. 6, p. 386, 1958. 2

[10] Y. LeCun, Y. Bengio, and G. E. Hinton, "Deep learning," *Nature*, vol. 521, pp. 436–444, 2015. 3

[11] A. L. Hodgkin and A. F. Huxley, "A quantitative description of membrane current and its application to conduction and excitation in nerve," *The Journal of Physiology*, vol. 117, no. 4, pp. 500–544, 1952. 3

[12] E. Izhikevich, "Simple model of spiking neurons," *IEEE Transactions on Neural Networks*, vol. 14, no. 6, pp. 1569–1572, 2003. 3

[13] J. S. Plank, C. Zheng, B. Gullett, N. Skuda, C. Rizzo, C. D. Schuman, and G. S. Rose, "The case for RISP: A reduced instruction spiking processor." arXiv:2206.14016, 2022. 3, 9, 10

[14] A. Z. Foshie, J. S. Plank, G. S. Rose, and C. D. Schuman, "Functional specification of the RAVENS neuroprocessor." arXiv:2307.15232, 2023. 3, 9

[15] G. Chakma, M. E. Dean, G. S. Rose, K. Beckmann, H. Manem, and N. Cady, "A hafnium-oxide memristive dynamic adaptive neural network array," in *International Workshop on Post-Moore's Era Supercomputing (PMES)*, (Salt Lake City, UT), Nov. 2016. 3

[16] S. Lowell and W. Singer, "Selection of intrinsic horizontal connections in the visual cortex by correlated neuronal activity.," *Science*, vol. 255, no. 5041, p. 209, 1992. 6

[17] D. E. Rumelhart, G. E. Hinton, and R. J. Williams, "Learning representations by back-propagating errors," *Nature*, vol. 323, pp. 533–536, 1986. 7

[18] W. Severa, C. M. Vineyard, R. Dellana, S. J. Verzi, and J. B. Aimone, "Training deep neural networks for binary communication with the whetstone method," *Nature Machine Intelligence*, vol. 1, pp. 86–94, Jan. 2019. 7, 9

[19] S. B. Shrestha and G. Orchard, "Slayer: Spike layer error reassignment in time," 2018. 7

[20] C. Zheng, "The islands project for managing populations in genetic training of spiking neural networks, master's thesis," 2022. 8

[21] C. D. Schuman, J. S. Plank, M. Parsa, S. R. Kulkarni, N. Skuda, and J. P. Mitchell, "A software framework for comparing training approaches for spiking neuromorphic systems," in *IJCNN: The International Joint Conference on Neural Networks*, pp. 1–10, July 2021. 8, 9

[22] J. S. Plank, C. Rizzo, K. Shahat, G. Bruer, T. Dixon, M. Goin, G. Zhao, J. Anantharaj, C. D. Schuman, M. E. Dean, G. S. Rose, N. C. Cady, and J. Van Nostrand, "The TENNLab suite of LIDAR-based control applications for recurrent, spiking, neuromorphic systems," in *44th Annual GOMACTech Conference*, (Albuquerque), Mar. 2019. 9, 36

[23] M. G. Bellemare, Y. Naddaf, J. Veness, and M. Bowling, "The arcade learning environment: An evaluation platform for general agents," *Journal of Artificial Intelligence Research*, vol. 47, pp. 253–279, 2013. 9

[24] S. Aeberhard and M. Forina, "Wine." UCI Machine Learning Repository, 1991. DOI: https://doi.org/10.24432/C5PC7J. 9

[25] M. Zwitter and M. Soklic, "Breast Cancer." UCI Machine Learning Repository, 1988. DOI: https://doi.org/10.24432/C51P4M. 9

[26] C. D. Schuman, J. S. Plank, G. Bruer, and J. Anantharaj, "Non-traditional input encoding schemes for spiking neuromorphic systems," in *IJCNN: The International Joint Conference on Neural Networks*, (Budapest), pp. 1–10, 2019. 9

[27] J. P. Mitchell, C. D. Schuman, R. M. Patton, and T. E. Potok, "Caspian: A neuromorphic development platform," in *NICE: Neuro-Inspired Computational Elements Workshop*, ACM, March 2020. 9

[28] M.-O. Gewaltig and M. Diesmann, "Nest (neural simulation tool)," *Scholarpedia*, vol. 2, no. 4, p. 1430, 2007. 9

[29] Y. LeCun and C. Cortes, "MNIST handwritten digit database." http://yann.lecun.com/exdb/mnist/, 2010. timestamp: 2016-07-12T19:25:30.000+0200. 10, 36, 44

[30] P. Grother, "Nist special database 19 handprinted forms and characters database," 1995. 10

[31] H. Xiao, K. Rasul, and R. Vollgraf, "Fashion-mnist: a novel image dataset for benchmarking machine learning algorithms," *arXiv preprint arXiv:1708.07747*, 2017. 10

[32] J. S. Plank, C. Zheng, C. D. Schuman, and C. Dean, "Spiking neuromorphic networks for binary tasks," in *International Conference on Neuromorphic Computing Systems (ICONS)*, pp. 1–8, ACM, 2021. 15, 17, 34, 75

[33] M. Hahsler, M. Piekenbrock, and D. Doran, "dbscan: Fast density-based clustering with R," *Journal of Statistical Software*, vol. 91, no. 1, pp. 1–30, 2019. 48

[34] A. Byerly, T. Kalganova, and I. Dear, "A branching and merging convolutional network with homogeneous filter capsules," *CoRR*, vol. abs/2001.09136, 2020. 66

[35] D. Hirata and N. Takahashi, "Ensemble learning in CNN augmented with fully connected subnetworks," *CoRR*, vol. abs/2003.08562, 2020. 66

Appendices

Appendix A

Path Generation

```
vector<Node *> makePathWithDelay(Network *n, Node *pre, Node
*post, double delay, double maxd, double mind, double mint,
double maxw, double outw, int startingid, int minr, string
refkey, int nodedelay, bool forcenewnode)
 {
   vector<Node *> rv;
   size_t id;
   double rd, td;
   Edge *tedge;
   Node *tnode, *pnode;

   id = startingid;
```

```
pnode = pre;

rd = delay;

if(delay < mind)
{
  throw std::runtime_error("Error: desired path less
  than minimum allowed delay");
}
if((forcenewnode == true && delay < ((2*mind)
  +nodedelay))||(delay < mind))
{
  throw std::runtime_error("Error: desired path with an
  extra node does not have sufficient delay");
}
else if((forcenewnode == true && delay >= ((2*mind)
  +nodedelay)))
{
  tnode = n->add_node(id);
  tnode->set("Threshold", mint);
  if(refkey != "")
  {
    tnode->set(refkey, minr);
  }
  id++;
  rv.push_back(tnode);
  tedge = n->add_edge(pnode->id, tnode->id);
  tedge->set("Weight", maxw);
  tedge->set("Delay", mind);
```

```
  pnode = tnode;
  rd -= mind + nodedelay;
}
if(forcenewnode == false && delay <= maxd && delay >=
  mind)
{
  tedge = n->add_edge(pre->id, post->id);
  tedge->set("Weight", outw);
  tedge->set("Delay", delay);
}
else
{
  if(forcenewnode == false)
  {
    pnode = pre;
  }
  id = startingid;
  while(rd > 0)
  {
    if(rd <= maxd && rd >= mind)
    {
      tedge = n->add_edge(pnode->id, post->id);
      tedge->set("Weight", maxw);
      tedge->set("Delay", rd);
      rd = 0;
    }
    else if(rd <= (2* maxd + nodedelay) && rd >= (2*
      mind + nodedelay))
    {
```

```
  tnode = n->add_node(id);
  tnode->set("Threshold", mint);
  tedge = n->add_edge(pnode->id, tnode->id);
  tedge->set("Weight", maxw);
  td = floor((rd -nodedelay) / 2);
  tedge->set("Delay", td);
  tedge = n->add_edge(tnode->id, post->id);
  tedge->set("Weight", maxw);
  tedge->set("Delay", rd - td - nodedelay);
  id++;
  rv.push_back(tnode);
  pnode = tnode;
  rd = 0;
}
else if(rd > (2 * maxd + nodedelay) && rd < (2 *
  maxd + mind + nodedelay))
{
  tnode = n->add_node(id);
  tnode->set("Threshold", mint);
  tedge = n->add_edge(pnode->id, tnode->id);
  tedge->set("Weight", maxw);
  td = floor((rd - 2 * nodedelay) / 3);
  tedge->set("Delay", td);
  id++;
  rv.push_back(tnode);
  pnode = tnode;
  rd -= (td + nodedelay);

  tnode = n->add_node(id);
```

```
      tnode->set("Threshold", mint);
      tedge = n->add_edge(pnode->id, tnode->id);
      tedge->set("Weight", maxw);
      td = floor((rd - nodedelay) / 2);
      tedge->set("Delay", td);
      rd -= (td + nodedelay);
      id++;
      rv.push_back(tnode);
      pnode = tnode;

      tedge = n->add_edge(pnode->id, post->id);
      tedge->set("Weight", maxw);
      tedge->set("Delay", rd);
      rd = 0;
    }
    else
    {
      tnode = n->add_node(id);
      tnode->set("Threshold", mint);
      tedge = n->add_edge(pnode->id, tnode->id);
      tedge->set("Weight", maxw);
      tedge->set("Delay", maxd);
      id++;
      rv.push_back(tnode);
      pnode = tnode;
      rd -= (maxd + nodedelay);
    }
  }
}
```

```
  return rv;
}
```

Appendix B

Simple Network Generation

```
  Network *makeSimpleNetwork(PropertyPack &pp, double mint,
size_t neuron_count)
  {
    Network *newnet;
    size_t i;
    newnet = new Network();
    newnet->set_properties(pp);
    for(i = 0; i < neuron_count; i++)
    {
      newnet->add_node(i)->set("Threshold", mint);
      newnet->set_input(i,i);
      newnet->set_output(i,i);
    }
    return newnet;
  }
```

Appendix C

Clustering Network Generation

```
Network *makeClusteringNetwork(PropertyPack &pp, int range,
int clusterRangetype, double threshold, vector<size_t> area,
vector<size_t> stride, size_t buffer, double minw, double
mint, double maxt)
  {
    Network *newnet;
    size_t i, j, k, l, id, oid;
    vector<Node *> corenodes;
    vector<Node *> bordernodes;
    vector<Node *> resetnodes;
    vector<vector<int> >lazygrid;
    bool (*distancefunc) (int x1, int y1, int x2, int y2,
      int range);
    Node *tnode;
    int minx;
    int maxx;
    double tweight;
    Edge *tedge;
```

```
if(clusterRangetype < 1 || clusterRangetype > 3)
{
  throw std::runtime_error("makeClusteringNetwork error:
    range type must be integer in [1,3]");
}
if(clusterRangetype == 1) distancefunc =
  clusterchebydist;
else if (clusterRangetype == 2) distancefunc =
  clustermandist;
else distancefunc = clusterrealdist;
if(area.size() < 1 || area.size() > 2)
{
  throw std::runtime_error("makeClusteringNetwork error:
    area must be 1d or 2d");
}
if(area.size() != stride.size())
{
  throw std::runtime_error("makeClusteringNetwork error:
    area and stride array size mismatch");
}
if(maxt < threshold)
{
  throw std::runtime_error("makeClusteringNetwork error:
   propertypack threshold below requested threshold");
}
if(maxt < 2 && minw > 0.5)
{
  throw std::runtime_error("makeClusteringNetwork error:
   propertypack threshold must be at >= 2 or min weight
```

```
   must be <= 0.5");
}
newnet = new Network();
newnet->set_properties(pp);
//1d mode does not use range types
if(area.size() == 1)
{
  if(buffer*2 >= area[0])
  {
    throw std::runtime_error("makeClusteringNetwork
    error: buffer too large for 1d area");
  }
  //make input layer
  for(i = 0; i < area[0]; i++)
  {
    tnode = newnet->add_node(i);
    newnet->set_input(i,i);
    tnode->set("Threshold", mint);
  }
  //make core detection nodes
  for(i = 0; i < ((area[0]-2*buffer)/stride[0]); i++)
  {
    corenodes.push_back(newnet->add_node(area[0]+i));
    newnet->set_output(2*i, area[0]+i); //core node out
    corenodes[i]->set("Threshold", threshold);
    minx = (int) buffer + (int) (i*stride[0]) - range;
    if(minx < 0) minx = 0;
    maxx = (int) buffer + (int) (i*stride[0]) + range;
    if(maxx > (int)(area[0] -1)) maxx = area[0] - 1;
```

```
  //connect to input nodes within area of detection
  for(j = (size_t) minx; j < (size_t) maxx; j++)
  {
    tedge = newnet->add_edge(j, i);
    tedge->set("Weight", 1);
    tedge->set("Delay", 1);
  }
}
//make hinterland detection nodes
for(i = 0; i < ((area[0]-2*buffer)/stride[0]); i++)
{
  bordernodes.push_back(newnet->add_node(
      area[0]+corenodes.size()+3*i));
  bordernodes.push_back(newnet->add_node(
      area[0]+corenodes.size()+3*i+1));
  bordernodes.push_back(newnet->add_node(
      area[0]+corenodes.size()+3*i+2));
  //if activity node
  bordernodes[3*i]->set("Threshold", 1.0);
  //if adj to core node
  bordernodes[3*i+1]->set("Threshold",1.0);
  //AND node
  newnet->set_output(2*i+1, area[0]+corenodes.size()
    +3*i+2);
  if(maxt < 2)
  {
    bordernodes[3*i+2]->set("Threshold", 1.0);
    tweight = 0.5;
```

```
}
else
{
  bordernodes[3*i+2]->set("Threshold", 2.0);
  tweight = 1.0;
}
//connect to input nodes within area of detection
  for activity detection
minx = (int) buffer + (int) (i*stride[0]) - range;
if(minx < 0) minx = 0;
maxx = (int) buffer + (int) (i*stride[0]) + range;
if(maxx > (int)(area[0] -1)) maxx = area[0] - 1;
for(j = (size_t) minx; j < (size_t) maxx; j++)
{
  tedge = newnet->add_edge(j, area[0]
    +corenodes.size()+3*i);
  tedge->set("Weight", 1);
  tedge->set("Delay", 2);
}
//connect to core points within radius
minx = i - (range / stride[0]);
if(minx < 0) minx = 0;
maxx = i + (range/stride[0]);
if(maxx > (int)((area[0]-2*buffer)/stride[0] -1))
maxx = ((area[0]-2*buffer)/stride[0])-1;
for (j = minx; (int)j < maxx; j++)
{
  if(i != j)
  {
```

```
        tedge = newnet->add_edge(corenodes[j]->id,
          bordernodes[3*i+1]->id);
        tedge->set("Weight", 1);
        tedge->set("Delay", 1);
      }
    }
    //perform the AND connection
    tedge = newnet->add_edge(bordernodes[3*i]->id,
      bordernodes[3*i+2]->id);
    tedge->set("Weight", tweight);
    tedge->set("Delay", 1);
    tedge = newnet->add_edge(bordernodes[3*i+1]->id,
      bordernodes[3*i+2]->id);
    tedge->set("Weight", tweight);
    tedge->set("Delay", 1);
  }
}
else if (area.size() == 2)
{
  lazygrid.resize(area[0], vector<int>(area[1],-1));
  for(i = 0; i < area.size(); i++)
  {
    if(buffer*2 >= area[i])
    {
      throw std::runtime_error("makeClusteringNetwork
        error: buffer too large for 2d area");
    }
  }
  //make input layer
```

```
for(i = 0; i < area[0]; i++)
{
  for(j = 0; j < area[1]; j++)
  {
    l = j * area[0] + i;
    tnode = newnet->add_node(l);
    newnet->set_input(l,l);
    tnode->set("Threshold", mint);
  }
}
//core detection
id = area[0] * area[1];

for(i = buffer; i < area[0] - buffer; i+= stride[0])
{
  for(j = buffer; j < area[1] - buffer; j+= stride[1])
  {
    tnode = newnet->add_node(id);
    lazygrid[i][j]=id;
    newnet->set_output(2*corenodes.size(), id);
    tnode->set("Threshold", threshold);
    corenodes.push_back(tnode);
    for(k = std::max((int)i - range, 0); k < std::min(
        i + range+1, area[0]); k++)
    {
      for(l = std::max((int)j - range, 0); l <
        std::min(j + range+1, area[1]); l++)
      {
        if (distancefunc(i,j,k,l,range) == true)
```

```
        {
          tedge = newnet->add_edge(l*area[0]+k,id);
          tedge->set("Weight", 1);
          tedge->set("Delay", 1);
        }
      }
    }
    id++;
  }
}
//hinterlands detection
oid = 1;
for(i = buffer; i < area[0] - buffer; i+= stride[0])
{
  for(j = buffer; j < area[1] - buffer; j+= stride[1])
  {
    //adj core detection
    tnode = newnet->add_node(id);
    tnode->set("Threshold", 1);
    bordernodes.push_back(tnode);
    //activity detection
    tnode = newnet->add_node(id+1);
    tnode->set("Threshold", 1);
    bordernodes.push_back(tnode);
    //AND node
    tnode = newnet->add_node(id+2);
    newnet->set_output(oid,id+2);
    oid+=2;
    if(maxt < 2 && minw <= 0.5)
```

```
{
  tnode->set("Threshold", 1);
  tedge = newnet->add_edge(id, id+2);
  tedge->set("Weight",0.5);
  tedge->set("Delay",1);
  tedge = newnet->add_edge(id+1,id+2);
  tedge->set("Weight", 0.5);
  tedge->set("Delay", 1);
}
else
{
  tnode->set("Threshold", 2);
  tedge = newnet->add_edge(id, id+2);
  tedge->set("Weight",1);
  tedge->set("Delay",1);
  tedge = newnet->add_edge(id+1,id+2);
  tedge->set("Weight", 1);
  tedge->set("Delay", 1);
}
bordernodes.push_back(tnode);
//get neighbors core nodes to the core detection
  and inputs to the activity node
for(k = std::max((int)i - range, 0); k < std::min(
    i + range+1, area[0]); k++)
{
  for(l = std::max((int)j - range, 0); l <
    std::min(j + range+1, area[1]); l++)
  {
    if (distancefunc(i,j,k,l,range) == true)
```

```
            {
              //edges from inputs
              tedge = newnet->add_edge(l*area[0]+k,id+1);
              tedge->set("Weight", 1);
              tedge->set("Delay", 2);
              //edges from core nodes
              if(lazygrid[k][l] != -1 &&
                (i != k || j != l))
              {
                tedge= newnet->add_edge(
                    lazygrid[k][l],id);
                tedge->set("Weight", 1);
                tedge->set("Delay", 1);
              }
              if(lazygrid[k][l] != -1 &&
                (i == k && j == l))
              {
                tedge= newnet->add_edge(
                    lazygrid[k][l],id);
                tedge->set("Weight", -1);
                tedge->set("Delay", 1);
              }
            }
          }
        }
        id+=3;
      }
    }
  }
```

```
  return newnet;
}
```

Appendix D

Counting Network Generation

```
  Network *makeCountingNetwork(PropertyPack &pp, double minw,
  double maxw, double mint, double maxt, int mind, int maxd,
  string refkey, int minr, int maxr,
  compilerInstructions &direction)
{
  Network *newnet;
  size_t maxin, xx, yy;
  size_t clayer, nlayer, znodes;
  Node *node, *tnode, *rnode, *onode;
  Edge *tedge;
  size_t k, l, m;
  vector<size_t> tnodelist, instancecount,
    layerresetpathnodes;
  vector<Node *> pathnodes, curlayer, reslayer;
  //generate a counting network
  newnet = new Network();
  newnet->set_properties(pp);
  clayer = direction.countingSize;
```

```
if(pp.networks.find("max_input_edges_per_node") !=
  pp.networks.end())
{
  maxin = newnet->values[newnet->get_network_property(
    "max_input_edges_per_node")->index];
}
else
{
  maxin = clayer;
}
l = 0;
//make the input layer
for(k = 0; k < clayer; k++)
{
  node = newnet->add_node(k);
  curlayer.push_back(node);
  node->set("Threshold", mint);
  if(direction.countingType == PIXELCOUNT
    && refkey != "")
  {
    if(direction.countingPeriod > (maxr))
    {
      throw std::runtime_error("Error: desired counting
        threshold is higher than is permitted");
    }
    node->set(refkey, direction.countingPeriod);
  }
  newnet->set_input(k,k);
  l++;
```

```
}
nlayer = ceil((double)(clayer) / (double)maxin);
znodes = 0;
while(nlayer > 1){
  curlayer.clear();
  for(k = 0; k < nlayer;k++)
  {
    node = newnet->add_node(l);
    curlayer.push_back(node);
    node->set("Threshold", std::max(
      direction.countingSubthreshold,mint));
    yy = std::min(ceil(((double)clayer) /
      ((double)nlayer)), (double)maxin);
    if(k < (nlayer - 1))
    {
      for(m = 0; m < yy; m++)
      {
        xx = znodes+(k*yy)+m;
        if(xx < clayer)
        {
          tedge = newnet->add_edge(xx, l);
          tedge->set("Weight", 1.0);
          tedge->set("Delay", mind);
        }
      }
    }
    else
    {
      for(m = znodes+(k*yy); m < clayer; m++)
```

```
    {
      tedge = newnet->add_edge(m, l);
      tedge->set("Weight", 1.0);
      tedge->set("Delay", mind);
    }
  }
  l++;
}
//build the reset structure, parallel to the counting
  structure
/* NOTE: this structure makes the assumption that a
  max weight synapse firing into a
minimum threshold neuron will always produce a
  neuronal firing event */
/* Additionally this structure is only used in
  software, RAVENS can perform a 1-cycle
reset on individual RNAs, so this can all be removed
  from networks loaded onto hardware*/
if(direction.countingType == SIMPLECOUNT && minw < 0)
{
  reslayer.clear();
  for(k = 0; k < nlayer; k++)
  {

    layerresetpathnodes.push_back(0);
    node = newnet->add_node(1);
    reslayer.push_back(node);
    node->set("Threshold", mint);
    node->name = "Reset Node"; //should make it easier
```

```
   to find these nodes to remove for hardware
yy = std::min(ceil(((double)clayer) /
  ((double)nlayer)), (double)maxin);
if(k < (nlayer - 1))
{
  for(m = 0; m < yy; m++)
  {

    xx = znodes+(k*yy)+m;
    if(xx < clayer)
    {
      tedge = newnet->add_edge(xx, l);
      tedge->set("Weight", maxw);
      tedge->set("Delay", mind);
    }
  }
}
else
{
  for(m = znodes+(k*yy); m < clayer; m++)
  {
    tedge = newnet->add_edge(m, l);
    tedge->set("Weight", maxw);
    tedge->set("Delay", mind);
  }
}
l++;
//build horizontal reset with a delay based on
  the desired counting period
```

```
      pathnodes = makePathWithDelay(newnet, node,
        newnet->get_node(l-nlayer-1),
        direction.countingPeriod, maxd, mind, mint,
        maxw, minw, l, minr, refkey, 0);
      l += pathnodes.size();
      layerresetpathnodes[k] += pathnodes.size();
      pathnodes.clear();
    }
    for(k = 0; k < nlayer; k++)
    {
      znodes+= layerresetpathnodes[k];
    }
  }
  znodes += clayer;
  clayer = nlayer;
  nlayer = ceil((double)(clayer) / (double)maxin)-1;
}

//add the output neuron
onode = newnet->add_node(l);
onode->set("Threshold", std::min(
  direction.countingThreshold,maxt));
if(direction.countingType == SIMPLECOUNT
  && refkey != "")
{
  if(maxr < direction.countingPeriod)
  {
    throw std::runtime_error("Compiler Error: counting
      period cannot be longer than neuroprocessor's
```

```
      maximum refractory period");
  }
  onode->set(refkey, direction.countingPeriod);
}
else if(refkey != "" &&
  (direction.countingType == PIXELCOUNT
  || direction.countingType == OPTCOUNT))
{
  onode->set(refkey, minr);
}
newnet->add_output(l);
for(m = 0; m < curlayer.size(); m++)
{
  tedge = newnet->add_edge(curlayer[m]->id, onode->id);
  tedge->set("Weight", 1.0);
  tedge->set("Delay", mind);
}
l++;

//build a false refractory structure to suppress the
  output neuron
if(refkey == "" && onode->incoming.size() < maxin
  && minw < mint)
{
  for(k = 0; k < (maxin - onode->incoming.size()); k++)
  {
    node = newnet->add_node(l);
    tnode = newnet->add_node(l+1);
    l+=2;
```

```
node->set("Threshold", mint);
tnode->set("Threshold", mint);

tedge = newnet->add_edge(onode->id, node->id);
tedge->set("Weight", maxw);
tedge->set("Delay", mind);
tedge = newnet->add_edge(node->id, tnode->id);
tedge->set("Weight", maxw);
tedge->set("Delay", mind);
tedge = newnet->add_edge(tnode->id, node->id);
tedge->set("Weight", maxw);
tedge->set("Delay", mind);

tedge = newnet->add_edge(tnode->id, onode->id);
tedge->set("Weight", minw);
tedge->set("Delay", mind);
tedge = newnet->add_edge(node->id, onode->id);
tedge->set("Weight", minw);
tedge->set("Delay", mind);

if(direction.countingPeriod < maxd)
{
  pathnodes.clear();
  pathnodes.push_back(newnet->add_node(l));
  l++;
  pathnodes[0]->set("Threshold", mint);
  tedge = newnet->add_edge(onode->id,
    pathnodes[0]->id);
  tedge->set("Weight", maxw);
```

```
      tedge->set("Delay", mind);
      tedge = newnet->add_edge(pathnodes[0]->id,
        node->id);
      tedge->set("Weight", minw);
      tedge->set("Delay", direction.countingPeriod);
      tedge = newnet->add_edge(pathnodes[0]->id,
        tnode->id);
      tedge->set("Weight", minw);
      tedge->set("Delay", direction.countingPeriod);
      pathnodes.clear();
    }
    else
    {
      pathnodes = makePathWithDelay(newnet, onode, node,
        direction.countingPeriod, maxd, mind, mint,
        maxw, minw, l, minr, refkey, 0);
      l+= pathnodes.size();
      tedge = newnet->add_edge(
        pathnodes[pathnodes.size() -1]->id, tnode->id);
      tedge->set("Weight", minw);
      tedge->set("Delay", direction.countingPeriod
        -(maxd * floor(direction.countingPeriod/maxd)));
      pathnodes.clear();
    }
  }
}
//reset components -- only part of the simplecount
//reset trigger neuron at the cap of the reset pyramid
  along with its incoming synapses
```

```
if(direction.countingType == SIMPLECOUNT)
{
  rnode = newnet->add_node(l);
  rnode->set("Threshold", mint);
  if(refkey != "")
  {
    rnode->set(refkey, minr);
  }
  for(m = 0; m < curlayer.size(); m++)
  {
    tedge = newnet->add_edge(curlayer[m]->id,
      rnode->id);
    tedge->set("Weight", 1.0);
    tedge->set("Delay", mind);
  }
  l++;

  //build the reset path for the output neuron
  if(minw < 0)
  {
    tnode = newnet->add_node(l);
    l++;
    pathnodes = makePathWithDelay(newnet, rnode, tnode,
      direction.countingPeriod, maxd, mind, mint, maxw,
      maxw, l, minr, refkey, 0);
    l += pathnodes.size();
    tedge = newnet->add_edge(tnode->id, onode->id);
    tedge->set("Weight", minw);
    tedge->set("Delay", mind);
```

```
      for(k = -1 * minw; k < (onode->get("Threshold")
        - mint)/(-1 * minw); k+=(-1 * minw))
      {
        node = newnet->add_node(l);
        node->set("Threshold", mint);
        if(refkey != "")
        {
          node->set(refkey, minr);
        }
        tedge = newnet->add_edge(l, onode->id);
        tedge->set("Weight", minw);
        tedge->set("Delay", direction.countingReset);
        tedge = newnet->add_edge(tnode->id, node->id);
        tedge->set("Weight", maxw);
        tedge->set("Delay", mind);
        l++;
        tnode = node;
      }
    }
  }
  return newnet;
}
```

Appendix E

Network Compiler

```
Network *netcompile(vector<Network *> networks,
vector<compilerInstructions> directions, size_t in)
{
  Network *rv;
  size_t i, j, k, l, m,n, trycount, maxtries, maxin,
    hostnet;
  int x, y, innerx, innery, overflowx, overflowy;
  std::vector<compilerInstructions> ins;
  compilerInstructions tin;
  vector<vector<size_t> >layers;
  vector<vector<int> >insources;
  std::ostringstream oss;
  size_t tnet, tlayer, pre, post, apre, apost;
  Edge *tedge;
  vector<Node *> pathnodes;
  EdgeMap::iterator eitr;
  vector<vector<vector<Node *> > > nodes;
    //network, instance, nodes
```

```
vector<vector<vector<Edge *> > > edges;
  //network, instance, edges
vector<map<size_t, size_t> >nodemap, rnodemap;
  //each network's id to an index
vector<size_t> tnodelist, instancecount,
  layerresetpathnodes;
vector<size_t> neto, neti, netinstances, netnodes,
  cnetnodes;
vector<vector<size_t> > inputinstances;
vector<size_t> tmcycles, tminstances;
vector<Network *>nets;
Network *newnet;
std::set<size_t> iq;
std::set<size_t>::iterator iqitr;
std::set<size_t> inputoverrides;
vector<std::set<size_t> >noveride;
vector<std::pair<size_t, size_t> >toutputs;
vector<std::string> datakeys;
vector<vector<Node *> >tmuxpre, tmuxpost;
vector<Node *> temppath;
PropertyPack pp;
networkLink tnl;
MOA rng;
bool progress, propfound;
int xx, yy, mind, maxd, minr, maxr;
double rngw;
double maxt, mint, maxw, minw;
string refkey;
json js;
```

```
json otherdata;
//get the propertypack from the first non-null network
for(i = 0; i < networks.size(); i++)
{
  if(networks[i] == nullptr) continue;

  pp = networks[i]->get_properties();
  otherdata = json::object();
  try{
    otherdata["app_params"] =
      networks[i]->get_data("app_params");
  }
  catch(std::exception &e) {}
  try{
    otherdata["proc_params"] =
      networks[i]->get_data("proc_params");
  }
  catch(std::exception &e) {}
  try{
    otherdata["encoder_array"] =
      networks[i]->get_data("encoder_array");
  }
  catch(std::exception &e) {}
  try{
    otherdata["decoder_array"] =
      networks[i]->get_data("decoder_array");
  }
  catch(std::exception &e) {}
  try{
```

```
  otherdata["other"] = networks[i]->get_data("other");
}
catch(std::exception &e) {}
hostnet = i;
/*get the keys so we can get the associated_data which
  is protected*/
datakeys = networks[i]->data_keys();
js = networks[i]->get_properties().as_json();
maxw = networks[i]->get_edge_property("Weight")
  ->max_value;
minw = networks[i]->get_edge_property("Weight")
  ->min_value;
mind = networks[i]->get_edge_property("Delay")
  ->min_value;
maxd = networks[i]->get_edge_property("Delay")
  ->max_value;
maxt = networks[i]->get_node_property("Threshold")
  ->max_value;
mint = networks[i]->get_node_property("Threshold")
  ->min_value;
minr = maxr = -1;
refkey = "";
propfound = false;
if(js.contains("node_properties") &&
  js["node_properties"].is_array() == true)
{
  for(j = 0; j < js["node_properties"].size(); j++)
  {
    if(js["node_properties"][j].is_object() == true
```

```
    && js["node_properties"][j].contains("name")
      == true && js["node_properties"][j]["name"]
      == "Refractory")
  {
    propfound = true;
    break;
  }
}
if(propfound == true)
{
  minr = networks[i]
    ->get_node_property("Refractory")->min_value;
  maxr = networks[i]
    ->get_node_property("Refractory")->max_value;
  refkey = "Refractory";
}
else
{
  for(j = 0; j < js["node_properties"].size(); j++)
  {
    if(js["node_properties"][j].is_object() == true
      && js["node_properties"][j].contains("name")
      == true && js["node_properties"][j]["name"]
      == "AbsRefrPeriod")
    {
      propfound = true;
      break;
    }
  }
```

```
      if(propfound == true)
      {
        minr = networks[i]->get_node_property
          ("AbsRefrPeriod")->min_value;
        maxr = networks[i]->get_node_property
          ("AbsRefrPeriod")->max_value;
        refkey = "AbsRefrPeriod";
      }
    }
  }
  break;
}
if(i == networks.size())
{
  oss.clear();
  oss << "Error: NetworkCompiler::compile() -- compiler
    requires at least 1 non-null network\n";
  throw std::runtime_error(oss.str());
}
j=0;
for(i = 0; i < directions.size(); i++)
{
  if(directions[i].generateNet == true)
  {
    /*generate a simple, single neuron network with
      minimum threshold*/
    newnet = makeSimpleNetwork(pp, mint,
      directions[i].generateSize);
    nets.push_back(newnet);
```

```
}
else if(directions[i].countingSize > 0)
{
  newnet = makeCountingNetwork(pp, minw, maxw, mint,
    maxt, mind, maxd, refkey, minr, maxr,
    directions[i]);
  nets.push_back(newnet);
}
else if(directions[i].clusterRange > 0)
{
  if(directions[i].clusterYsize > 0)
  {
    newnet = makeClusteringNetwork(pp,
      directions[i].clusterRange,
      directions[i].clusterRangetype,
      directions[i].clusterThreshold,
    vector<size_t>{(size_t)directions[i].clusterXsize,
      (size_t)directions[i].clusterYsize},
      vector<size_t> {(size_t)directions[i]
      .clusterXstride, (size_t)directions[i]
      .clusterYstride},
    directions[i].clusterBuffer, minw, mint, maxt);
  }
  else
  {
    newnet = makeClusteringNetwork(pp, directions[i]
      .clusterRange, directions[i].clusterRangetype,
      directions[i].clusterThreshold,
    vector<size_t>{(size_t)directions[i].clusterXsize}
```

```
      , vector<size_t> {(size_t)directions[i]
      .clusterXstride},
      directions[i].clusterBuffer, minw, mint, maxt);
    }
    nets.push_back(newnet);
  }
  else
  {
    if(j >= networks.size())
    {
      throw std::runtime_error("Error: NetworkCompiler::
        compile() -- network / direction
        size mismatch\n");
    }
    nets.push_back(networks[j]);
    j++;
  }
}
if(nets.size() != directions.size())
{
  throw std::runtime_error("Error: NetworkCompiler::
    compile() -- network / direction size mismatch\n");
}
k = hostnet;
rv = new Network();
rv->set_properties(pp);
if(otherdata.contains("app_params") == true)
{
  rv->set_data("app_params", otherdata["app_params"]);
```

```
}
if(otherdata.contains("proc_params") == true)
{
  rv->set_data("proc_params", otherdata["proc_params"]);
}
if(otherdata.contains("encoder_array") == true)
{
  rv->set_data("encoder_array",
    otherdata["encoder_array"]);
}
if(otherdata.contains("decoder_array") == true)
{
  rv->set_data("decoder_array",
    otherdata["decoder_array"]);
}
if(otherdata.contains("other") == true)
{
  rv->set_data("other", otherdata["other"]);
}
for(i = 0; i < datakeys.size(); i++)
{
  rv->set_data(datakeys[i], networks[k]
    ->get_data(datakeys[i]));
}
for(i = 0; i < directions.size(); i++)
{
  if(directions[i].inputdelay < rv->get_edge_property
    ("Delay")->min_value)
  {
```

```
    fprintf(stderr, "Warning: Instruction %lu instructs
      for illegal delay value %lu\n", i,
      directions[i].inputdelay);
    directions[i].inputdelay = rv->
      get_edge_property("Delay")->min_value;
  }
  if(directions[i].inputdelay > rv->
    get_edge_property("Delay")->max_value)
  {
    fprintf(stderr, "Warning: Instruction %lu instructs
      for illegal delay value %lu\n", i,
      directions[i].inputdelay);
    directions[i].inputdelay = rv->get_edge_property
      ("Delay")->max_value;
  }
}
/*Get the instruction settings for the networks, and
  figure out the layer structure*/
for(i = 0; i < directions.size(); i++)
{
  for(j = 0; j < directions[i].inputs.size(); j++)
  {
    if(directions[i].inputs[j] >= (int)insources.size())
    {
      insources.resize(directions[i].inputs[j] +1);
    }
    /* build the network connections */
    if(directions[i].inputs[j] != -1)
    {
```

```
      tnet = directions[i].inputs[j];
      insources[tnet].push_back(i);
    }
  }
  if(directions[i].outputs == true && i >=
    insources.size())
  {
    insources.resize(i+1);
  }
  //check for input grid sizes
  if(directions[i].inputgrid == true)
  {
    if(nets[i]->num_inputs() != (int)(directions[i]
      .igrid_x * directions[i].igrid_y))
    {
      oss.clear();
      oss << "Error: NetworkCompiler::compile() --
        network " << i << " invalid grid of " <<
        directions[i].igrid_x << "x" <<
        directions[i].igrid_y << " does not match
        network inputs: " << nets[i]->num_inputs()
        << std::endl;
      throw std::runtime_error(oss.str());
    }
  }
}
if(insources.size() != nets.size())
{
  oss.clear();
```

```
  oss << "Warning: NetworkCompiler::compile() -- nets
    are missing connections or connection go to non-
    existant layers\n";
  std::cerr << oss.str();
}
rnodemap.resize(nets.size());
nodemap.resize(nets.size());
/*get the size of each nets IO and build the reverse
  nodemap*/
for(i = 0; i < nets.size(); i++)
{
  neti.push_back(nets[i]->num_inputs());
  neto.push_back(nets[i]->num_outputs());
  j = 0;
  tnodelist.clear();
  nets[i]->make_sorted_node_vector();
  for(j = 0; j < nets[i]->sorted_node_vector.size();
    j++)
  {
    nodemap[i][nets[i]->sorted_node_vector[j]->id] = j;
    rnodemap[i][j] = nets[i]->sorted_node_vector[j]->id;
  }
}
netinstances.resize(nets.size(), 0);
inputinstances.resize(nets.size());
tminstances.resize(nets.size());
netnodes.resize(nets.size(), 0);
noveride.resize(nets.size());
/*look at network links and reduce the inputs by that
```

```
  amount -- network links do not reduce the outputs for
  now*/
/*also build an instance queue (iq) to process the
  number of instances*/
for(i = 0; i < directions.size(); i++)
{
  if(directions[i].inputs.size() == 0)
  {
    throw std::runtime_error("Error: Network cannot
      have no inputs");
  }
  inputinstances[i].resize(directions[i].inputs.size(),
    0);
  if(nets[i] != nullptr)
  {
    for(j = 0; j < (size_t) nets[i]->num_inputs(); j++)
    {
      noveride[i].emplace(j);
    }
    /*in case the same input has multiple manual
      overrides, find an accurate count of overridden
      inputs*/
    inputoverrides.clear();
    for(j = 0; j < directions[i].overrides.size(); j++)
    {
      if(directions[i].overrides[j].allowmix == false)
      {
        inputoverrides.emplace(directions[i]
          .overrides[j].inputNum);
```

```
    if(noveride[i].find(directions[i].overrides[j]
      .inputNum) != noveride[i].end())
    {
      noveride[i].erase(directions[i]
        .overrides[j].inputNum);
    }
    else if (directions[i].overrides[j].inputNum >=
      (size_t) nets[i]->num_inputs())
    {
      throw std::runtime_error("Error:
        NetworkCompiler::compile() -- Override
        requested for non-existant input.\n");
    }
  }
}
neti[i] -= inputoverrides.size();
/*if all inputs are overriden, this network has a
  single instances and is already solved*/
if(neti[i] <= 0 && directions[i].instances == 0)
{
  netinstances[i] = 1;
  netnodes[i] = nets[i]->num_nodes();
  for(j = 0; j < inputinstances[i].size(); j++)
  {
    inputinstances[i][j] = 0;
  }
}
else if (directions[i].instances > 0 ||
  directions[i].tmux == true)
```

```
{
  if(directions[i].tmux == false)
  {
    netinstances[i] = directions[i].instances;
    netnodes[i] = netinstances[i] * nets[i]
      ->num_nodes();
    neto[i] *= netinstances[i];
    neti[i] *= netinstances[i];
  }
  else
  {
    /*if time compression, delay netnodes until
      inputs are solved to build the new time
      compression outputs*/
    if(directions[i].tcom == false)
    {
      netinstances[i] = 1;
      netnodes[i] = nets[i]->num_nodes();
    }
    else
    {
      iq.emplace(i);
    }


  }
}
//other cases must be solved based on inputs
else
```

```
  {
    iq.emplace(i);
  }
}
/*for null networks handle as if using generate simple
  network command*/
else
{
  if(directions[i].overrides.size() > 0)
  {
    for(j = 0; j < directions[i].overrides.size();
      j++)
    {
      if(directions[i].overrides[j].instances.size()
        > 1)
      {
        throw std::runtime_error("Error:
          NetworkCompiler::compile() -- Instance map
          larger than 1 for overridden empty network.
          \n");
      }
    }
    netinstances[i] = 1;
    netnodes[i] = 1;
  }
  else
  {
    noveride[i].emplace(0);
    iq.emplace(i);
```

```
    }
  }
}
// Build the number of network instances required
maxtries = nets.size() * 10; /*overly permissive, but
  just in case*/
trycount = 0;
while(iq.empty() != true)
{
  /*Just in case of some case not caught earlier,
    prevent infinite loops*/
  trycount++;
  progress = false;
  if(trycount >= maxtries)
  {
    throw std::runtime_error("Error: NetworkCompiler::
      compile() -- Network Instance generation timeout.
      Verify network connections are valid.\n");
  }
  for(iqitr = iq.begin(); iqitr != iq.end(); iqitr++)
  {
    i = *iqitr;
    /*if user-specified number of instances -- all
      fully-connected networks should be as well as a
      time-multiplexed network*/
    if(directions[i].instances > 0)
    {
      netnodes[i] = nets[i]->num_nodes()
        * netinstances[i];
```

```
  neto[i] *= netinstances[i];
  neti[i] *= netinstances[i];
  iq.erase(iqitr);
  progress = true;
  break;
}
else
{
  instancecount.clear();
  for(j = 0; j < directions[i].inputs.size(); j++)
  {
    //Case: network connected to the overall inputs
    if(directions[i].inputs[j] == -1)
    {
      if(directions[i].tmux == true)
      {
        instancecount.push_back(1);
      }
      else
      {
        instancecount.push_back(in / neti[i] +
          ((in % neti[i]) > 0 ? 1 : 0));
      }
      inputinstances[i][j] = in / neti[i] +
        ((in % neti[i]) > 0 ? 1 : 0);
    }
    /*Case: fully-connected with no specified number
      of instances*/
    else if(directions[i].fullyConnected == true)
```

```
{
  throw std::runtime_error("Error: Fully
    connected networks must have a count of
    instances");
}
// Case: Input grid
else if(directions[i].inputgrid == true)
{
  if(directions[i].inputs[j] >= 0 &&
    directions[directions[i].inputs[j]]
    .outputgrid == true)
  {
    l = directions[i].inputs[j];
    x = (directions[l].ogrid_x -directions[i]
      .igrid_x - directions[i].leftborder
      - directions[i].rightborder)
      / directions[i].xstride  +1;
    if(directions[i].fitgrid != true &&
      ((directions[l].ogrid_x -directions[i]
      .igrid_x - directions[i].leftborder
      - directions[i].rightborder)
      % (directions[i].xstride)  +1) > 0)
    {
      x++;
    }
    y = (directions[l].ogrid_y -directions[i]
      .igrid_y - directions[i].topborder
      - directions[i].botborder)
      / directions[i].ystride  +1;
```

```
    if(directions[i].fitgrid != true &&
      ((directions[l].ogrid_y -directions[i]
      .igrid_y - directions[i].topborder
      - directions[i].botborder)
      % (directions[i].ystride)  +1) > 0)
    {
      y++;
    }
    if(directions[i].tmux == true
      && directions[i].tcom == true)
    {
      instancecount.push_back(1);
    }
    else
    {
      instancecount.push_back(x*y);
    }
    inputinstances[i][j] = x*y;
  }
}
/*Case: network connected to another network's
  outputs which have a known number of
  instances */
else if(netinstances[directions[i].inputs[j]]
  != 0)
{
  if (directions[i].tcom == true
    && directions[directions[i].inputs[j]]
      .tmux == true)
```

```
    {
      instancecount.push_back(1);
      inputinstances[i][j] = neto[directions[i]
        .inputs[j]] / neti[i] + ((
          neto[directions[i].inputs[j]]
          % neti[i]) > 0 ? 1 : 0);
    }
    else if(directions[i].tcom == false
      || directions[directions[i].inputs[j]].tmux
      == false)
    {
      instancecount.push_back(
        neto[directions[i].inputs[j]] / neti[i]
        + ((neto[directions[i].inputs[j]]
        % neti[i]) > 0 ? 1 : 0));
      inputinstances[i][j] =
        neto[directions[i].inputs[j]] / neti[i]
        + ((neto[directions[i].inputs[j]]
        % neti[i]) > 0 ? 1 : 0);
    }
  }
}
if(instancecount.size() ==
  directions[i].inputs.size())
{
  if(directions[i].inMix == true)
  {
    maxin = instancecount[0];
    for(j = 1; j < instancecount.size(); j++)
```

```
{
  if(instancecount[j] > maxin)
  {
    maxin = instancecount[j];
  }
}
if(maxin == 0)
{
  throw std::runtime_error("Error: A network
    must have at least 1 instance");
}
if(directions[i].tmux == true && directions[i]
  .tcom == true)
{
  netinstances[i] = 1;
  neto[i] *= maxin;
  netnodes[i] = nets[i]->num_nodes()
    + neto[i];
  iq.erase(iqitr);
  progress = true;
  break;
}
else
{
  if(directions[i].outputgrid == true)
  {
    x = directions[i].ogrid_x;
    y = directions[i].ogrid_y;
    l = x*y;
```

```
      m = l / nets[i]->num_outputs();
      if(m < maxin)
      {
        fprintf(stderr, "Warning: network for
          instruction %lu to generate
          insufficient instances (%lu) for grid
          size %dx%d, generating %lu instead\n"
          , i, maxin, x, y, m);
        maxin = m;
      }
    }
    netinstances[i] = maxin;
    netnodes[i] = netinstances[i]
      * nets[i]->num_nodes();
    neto[i] *= netinstances[i];
    neti[i] *= netinstances[i];
    iq.erase(iqitr);
    progress = true;
    break;
  }
}
else
{
  maxin = 0;
  for(j = 0; j < inputinstances[i].size(); j++)
  {
      maxin += inputinstances[i][j];
  }
  if(maxin == 0)
```

```
      {
        throw std::runtime_error("Error: A network
          must have at least 1 instance");
      }
      if(directions[i].tmux == true
        && directions[i].tcom == true)
      {
        netinstances[i] = 1;
        neto[i] *= maxin;
        netnodes[i] = nets[i]->num_nodes()
          + neto[i];
        iq.erase(iqitr);
        progress = true;
        break;
      }
      else
      {
        netinstances[i] = maxin;
        netnodes[i] = netinstances[i]
          * nets[i]->num_nodes();
        neto[i] *= netinstances[i];
        neti[i] *= netinstances[i];
        iq.erase(iqitr);
        progress = true;
        break;
      }
    }
  }
}
```

```
  }
  if(progress == false)
  {
    throw std::runtime_error("Error: instructions are
      ambiguous, try declaring manual instance counts");
  }
}
j = 0;
/*build the cumulative number of neurons "before" this
  network, will be used in figuring out net ids*/
for(i = 0; i < netnodes.size(); i++)
{
  cnetnodes.push_back(j);
  j += netnodes[i];
}
/*build the number of time-multiplexed psuedo-instances
  and figure out their cycle delay*/
tmcycles.resize(nets.size());
tmuxpre.resize(nets.size());
tmuxpost.resize(nets.size());
for(i = 0; i < nets.size(); i++)
{
  if(directions[i].tmux == true)
  {
    if(directions[i].tmuxPeriod < 0)
    {
      throw std::runtime_error("Error: time multiplexing
        period must be greater than 0");
    }
```

```
    tmcycles[i] = 0;
    if(directions[i].inMix == true)
    {
      for(j = 0; j < directions[i].inputs.size(); j++)
      {
        if(directions[i].inputs[j] != -1
          && instancecount[directions[i].inputs[j]]
          > tmcycles[i])
        {
          tmcycles[i] =
            instancecount[directions[i].inputs[j]];
        }
      }
    }
    else
    {
      for(j = 0; j < directions[i].inputs.size(); j++)
      {
        if(directions[i].inputs[j] != -1)
        {
          tmcycles[i]
            += instancecount[directions[i].inputs[j]];
        }
      }
    }
  }
}
nodes.resize(nets.size());
edges.resize(nets.size());
```

```
/*using the number of instances, build the neurons of
  all the nets, sorted as noted in the nodes table*/
for(i = 0; i < nets.size(); i++)
{
  if(nodes[i].size() == 0)
  {
    nodes[i].resize(netinstances[i]);
    edges[i].resize(netinstances[i]);
  }
  for(j = 0; j < netinstances[i]; j++)
  {
    for(k = 0; k < nets[i]->num_nodes();k++)
    {
      apost = cnetnodes[i] + j * nets[i]->num_nodes()
        + k;
      if(apost != rv->num_nodes())
      {
        fprintf(stderr, "error: node mismatch\n");
      }
      nodes[i][j].push_back(rv->add_node(apost));
      nodes[i][j][k]->set("Threshold",
        nets[i]->sorted_node_vector[k]
        ->get("Threshold"));
      for(l = 0; l < directions[i].inputs.size(); l++)
      {
        if(directions[i].inputs[l] == -1 && (nets[i]
          ->sorted_node_vector[k]->input_id >= 0))
        {
          rv->add_input(apost);
```

```
      break;
    }
  }
  if(directions[i].outputs == true && (nets[i]
    ->sorted_node_vector[k]->output_id >= 0)
      && (directions[i].tmux == false
      || directions[i].tcom == false))
  {
    rv->add_output(apost);
  }

}
if(directions[i].tmux == true
  && directions[i].tcom == true)
{
  for(k = 0; k < neto[i]; k++)
  {
    apost = cnetnodes[i] + (j+1)
      * nets[i]->num_nodes() + k;
    if(apost != rv->num_nodes())
    {
      fprintf(stderr, "error: node mismatch\n");
    }
    nodes[i][j].push_back(rv->add_node(apost));

    nodes[i][j][nets[i]->num_nodes() + k]
      ->set("Threshold", (double)k);
    if(directions[i].outputs == true && (nets[i]
      ->sorted_node_vector[k]->output_id >= 0)
```

```
            && directions[i].tmux == false)
        {
          rv->add_output(apost);
        }
      }
    }
  }
}
//build the internal synapses of each instance of
  each network
for(i = 0; i < nets.size(); i++)
{
  k = 0;
  for(eitr = nets[i]->edges_begin(); eitr
    != nets[i]->edges_end(); eitr++)
  {
    tedge = eitr->second.get();
    pre = nodemap[i][tedge->from->id];
    post = nodemap[i][tedge->to->id];
    for(j = 0; j < netinstances[i]; j++)
    {
      apre = cnetnodes[i] + j * nets[i]->num_nodes()
        + pre;
      apost = cnetnodes[i] + j * nets[i]->num_nodes()
        + post;
      edges[i][j].push_back(
        rv->add_edge(apre, apost));
      edges[i][j][k]->set("Weight",
        tedge->get("Weight"));
```

```
      edges[i][j][k]->set("Delay",
        tedge->get("Delay"));
    }
    k++;
  }
  if(directions[i].tmux == true && directions[i].tcom
    == true)
  {
    for(j = 0; j < (neto[i] / nets[i]->num_outputs());
      j++)
    {
      for(k = 0; (int)k < nets[i]->num_outputs(); k++)
      {
        pre = nodemap[i][nets[i]->get_output(k)->id];
        post = nets[i]->num_nodes() + j*nets[i]
          ->num_outputs() + k;
        apre = cnetnodes[i] + pre;
        apost = cnetnodes[i] + post;
        makePathWithDelay(rv, rv->get_node(apre),
        rv->get_node(apost), directions[i].tmuxPeriod
        * ( (neto[i] / nets[i]->num_outputs())- j-1)+1,
        maxd, mind, mint, maxw, maxw, rv->num_nodes(),
        minr, refkey, 0, false);
      }
    }
  }
}
/*create the network link (override) inter-network
  synapses*/
```

```
for(i = 0; i < nets.size(); i++)
{
  for(k = 0; k < directions[i].overrides.size(); k++)
  {
    tnl = directions[i].overrides[k];
    if(tnl.source > (int) nets.size())
    {
      oss.clear();
      oss << "Error: NetworkCompiler::compile() --
        Invalid override source: " << tnl.source;
      throw std::runtime_error(oss.str());
    }
    if(nets[tnl.source] == nullptr)
    {
      pre = 0;
      xx = 1;
      if((int) tnl.inputNum > nets[i]->num_inputs())
      {
        oss.clear();
        oss << "Error: NetworkCompiler::compile() --
          Invalid override input: " << tnl.outputNum;
        throw std::runtime_error(oss.str());
      }
      post = nodemap[i][nets[i]->
        get_input(tnl.inputNum)->id];
      yy = nets[i]->num_nodes();
    }
    else
    {
```

```
  if((nets[tnl.source] == nullptr && tnl.outputNum
    > 0) || (nets[tnl.source] != nullptr
    && tnl.outputNum > nets[tnl.source]
    ->num_outputs()))
  {
    oss.clear();
    oss << "Error: NetworkCompiler::compile() --
      Invalid override output: " << tnl.outputNum;
    throw std::runtime_error(oss.str());
  }
  if((nets[i] == nullptr && tnl.inputNum > 0)
    || (nets[i] != nullptr && (int) tnl.inputNum
    > nets[i]->num_inputs()))
  {
    oss.clear();
    oss << "Error: NetworkCompiler::compile()
      -- Invalid override input: " << tnl.outputNum;
    throw std::runtime_error(oss.str());
  }
  pre = nodemap[tnl.source][nets[tnl.source]
    ->get_output(tnl.outputNum)->id];
  xx = nets[tnl.source]->num_nodes();

  post = nodemap[i][nets[i]
    ->get_input(tnl.inputNum)->id];
  yy = nets[i]->num_nodes();
}
/*if overridden connections do not specify an
  instance mapping, then the instances map 1 to 1*/
```

```
if(tnl.instances.size() == 0)
{
  if(netinstances[i] != netinstances[tnl.source])
  {
    oss.clear();
    oss << "Error: NetworkCompiler::compile()
      -- Override instruction from network: "
      << tnl.source << " ("
      << netinstances[tnl.source] << " instances) to
      network: " << i << " (" << netinstances[i] <<
      " instances) has no instance map despite a
      different number of instances";
    throw std::runtime_error(oss.str());
  }
  for(j = 0; j < netinstances[i]; j++)
  {
    apre = cnetnodes[tnl.source] + j
      * nets[tnl.source]->num_nodes() + pre;
    apost = cnetnodes[i] + j
      * nets[i]->num_nodes() + post;
    edges[i][j].push_back(rv->add_edge(apre, apost));
    edges[i][j][k]->set("Weight",
      rv->get_edge_property("Weight")->max_value);
    edges[i][j][k]->set("Delay",
      rv->get_edge_property("Delay")->min_value);
    /* I store the internetwork edges with the post-
      synaptic neuron*/
  }
}
```

```
else //use the defined instance mapping
{
  if(tnl.instances.size() != netinstances[i])
  {
    oss.clear();
    oss << "Error: NetworkCompiler::compile() --
      Override instruction from network: "
        << tnl.source << " ("
        << netinstances[tnl.source] << " instances)
        to network: " << i << " ("
        << netinstances[i] << " instances) has
        incorrect instance map size: "
        << tnl.instances.size();
    throw std::runtime_error(oss.str());
  }
  for(j = 0; j < netinstances[i]; j++)
  {
    if(tnl.instances[j] >= netinstances[tnl.source])
    {
      oss.clear();
      oss << "Error: NetworkCompiler::compile() --
        Override instruction from network: "
        << tnl.source << " ("
        << netinstances[tnl.source]
        << " instances) to network: " << i << " ("
        << netinstances[i] << " instances) has
        invalid instance map source: "
        << tnl.instances[j];
      throw std::runtime_error(oss.str());
```

```
        }
        apre = cnetnodes[tnl.source] + tnl.instances[j]
          * xx + pre;
        apost = cnetnodes[i] + j * yy + post;
        edges[i][j].push_back
          (rv->add_edge(apre, apost));
        edges[i][j][k]->set("Weight",
          rv->get_edge_property("Weight")->max_value);
        edges[i][j][k]->set("Delay",
          rv->get_edge_property("Delay")->min_value);
      }
    }
  }
}
k = 0;
//create the general inter-network synapses
for(i = 0; i < nets.size(); i++)
{
  /* n represents which instance of the current
    subnetwork to look at*/
  n = 0;
  for(l = 0; l < directions[i].inputs.size(); l++)
  {
    if(directions[i].inMix == true)
    {
      n = 0;
    }
    /*if the inputs are the overall network inputs they
      are already handled*/
```

```
if(directions[i].inputs[l] == -1) continue;
tlayer = directions[i].inputs[l];
toutputs.clear();
/*build linear list of outputs across all instances
  for convenience*/
for(j = 0; j < netinstances[tlayer]; j++)
{
  if(nets[tlayer] == nullptr)
  {
    toutputs.push_back(std::make_pair(j,0ul));
  }
  else
  {
    if(directions[tlayer].tmux == true
      && directions[tlayer].tcom == true)
    {
      for(k = 0; k < neto[tlayer]; k++)
      {
        toutputs.push_back(std::make_pair(j,
          nets[tlayer]->num_nodes()+k));
      }
    }
    else
    {
      for(k = 0; k < (size_t) nets[tlayer]
        ->num_outputs(); k++)
      {
        toutputs.push_back(std::make_pair(j,
          nodemap[tlayer][nets[tlayer]
```

```
          ->get_output(k)->id]));
      }
    }
  }
}
//standard (linear) input mode
if(directions[i].inputgrid == false
  && directions[i].fullyConnected == false)
{
  //map to non-overridden inputs
  if(directions[i].tmux == false)
  {
    k = 0;
    for(j = 0; j < netinstances[i]; j++)
    {
      for(iqitr = noveride[i].begin(); iqitr
        != noveride[i].end(); iqitr++)
      {
        if(k >= toutputs.size())
        {
          /*the inputs of the layer were not an even
            multiple, and so there is some overhang*/
          continue;
        }
        pre = toutputs[k].second;
        post = *iqitr;
        if(nets[tlayer] == nullptr)
        {
          tnet = 1;
```

```
      }
      else
      {
        tnet = nets[tlayer]->num_nodes();
      }
      apre = cnetnodes[tlayer] + toutputs[k].first
        * tnet + toutputs[k].second;
      tnet = nets[i]->num_nodes();
      apost = cnetnodes[i] + n * tnet + *iqitr;
      edges[i][n].push_back(
          rv->add_edge(apre, apost));
      edges[i][n][edges[i][n].size() -1]
        ->set("Weight", rv
        ->get_edge_property("Weight")
        ->max_value);
      edges[i][n][edges[i][n].size() -1]
        ->set("Delay", directions[i].inputdelay);
      k++;
    }
    n++;
  }
}
else
{
  l = 0;
  while((l < toutputs.size()))
  {
    for(iqitr = noveride[i].begin();
      iqitr != noveride[i].end(); iqitr++)
```

```
      {
        if(l > toutputs.size()) break;
        pre = toutputs[l].second;
        post = *iqitr;
        tnet = nets[tlayer]->num_nodes();
        apre = cnetnodes[tlayer] + toutputs[l].first
          * tnet + pre;
        apost = cnetnodes[i] + *iqitr;

        makePathWithDelay(rv, rv->get_node(apre),
          rv->get_node(apost),
          (n* directions[i].tmuxPeriod)+mind, maxd,
          mind, mint, maxw, maxw, rv->num_nodes(),
          minr, refkey, 0, false);
        l++;
      }
      k++;
      n++;
    }
  }

}
//fully connected mode
else if (directions[i].fullyConnected == true)
{
  rng.Seed(directions[i].fcseed, COMPILER_SEED);
  //map to non-overridden inputs
  for(j = 0; j < netinstances[i]; j++)
  {
```

```
for(iqitr = noveride[i].begin(); iqitr
  != noveride[i].end(); iqitr++)
{
  for(k = 0; k < toutputs.size(); k++)
  {
    pre = toutputs[k].second;
    post = *iqitr;
    if(nets[tlayer] == nullptr)
    {
      tnet = 1;
    }
    else
    {
      tnet = nets[tlayer]->num_nodes();
    }
    apre = cnetnodes[tlayer] + toutputs[k].first
      * tnet + toutputs[k].second;
    tnet = nets[i]->num_nodes();
    apost = cnetnodes[i] + n * tnet + *iqitr;
    edges[i][j].push_back(rv->add_edge(apre,
      apost));
    rngw = rng.Random_Double() *
      (rv->get_edge_property("Weight")
      ->max_value - rv
      ->get_edge_property("Weight")
      ->min_value) + rv
      ->get_edge_property("Weight")->min_value;
    edges[i][j][edges[i][j].size() -1]
      ->set("Weight", rngw);
```

```
        edges[i][j][edges[i][j].size() -1]
          ->set("Delay", directions[i].inputdelay);
      }
    }
  }
}
//grid connection mode
else if (directions[i].inputgrid == true)
{
  /*ignore inputs subnetworks whose instructions
    lack a configured output grid*/
  if(directions[tlayer].outputgrid == false) continue;

  l = 0;
  if(directions[i].xfirst == false)
  {
    /*if instructions are set to 'fit grid' then
      some inputs may not be connected
    otherwise some instances of the target network
      may be partially connected instead, if sizes
      do not match up*/
    overflowx = directions[tlayer].ogrid_x
      - directions[i].leftborder
      - directions[i].rightborder;
    if(directions[i].fitgrid== true
      && (overflowx % directions[i].xstride > 0))
    {
      overflowx += directions[i].xstride;
    }
```

```
for(x = 0 + directions[i].leftborder; x
  < (overflowx + directions[i].leftborder);
  x+=directions[i].xstride)
{
  overflowy = directions[tlayer].ogrid_y
    - directions[i].topborder
    - directions[i].botborder;
  if(directions[i].fitgrid == true)
  {
    overflowy += directions[i].ystride;
  }
  for(y = 0 + directions[i].topborder; y
    < (overflowy + directions[i].topborder);
    y+=directions[i].ystride)
  {
    m = 0;
    for(innery = y; innery < (y
      + directions[i].igrid_y); innery++)
    {
      if(innery < 0 || innery
        >= directions[tlayer].ogrid_y) continue;
      for(innerx = x; innerx < (x
        + directions[i].igrid_x); innerx++)
      {
        k = (innery *
          directions[tlayer].ogrid_x) + innerx;
        if(innerx < 0 || innerx
          >= directions[tlayer].ogrid_x
```

```
  || k >= toutputs.size())
  continue;
pre = toutputs[k].second;
post = m;
apre = cnetnodes[tlayer]
  + toutputs[k].first
    * nets[tlayer]->num_nodes() + pre;
if(directions[i].tmux == false)
{
  apost = cnetnodes[i] + n
    * nets[i]->num_nodes()
      + nets[i]->get_input(post)->id;
  edges[i][n].push_back(
      rv->add_edge(apre, apost));
  edges[i][n][edges[i][n].size() -1]
    ->set("Weight",
    rv->get_edge_property("Weight")
    ->max_value);
  edges[i][n][edges[i][n].size() -1]
    ->set("Delay",
    directions[i].inputdelay);
}
else
{
  apost = cnetnodes[i] + post;
  makePathWithDelay(rv,
    rv->get_node(apre),
    rv->get_node(apost), n*
    directions[i].tmuxPeriod +1+mind*2,
```

```
                maxd, mind, mint, maxw, maxw,
                rv->num_nodes(), minr, refkey, 0,
                false);
            }
            m++;
          }
        }
        l++;
        n++;
      }
    }
  }
  else
  {
    /*if instructions are set to 'fit grid' then
      some inputs may not be connected
    //otherwise some instances of the target network
      may be partially connected instead, if sizes
      do not match up*/
    overflowy = directions[tlayer].ogrid_y
      - directions[i].topborder
      - directions[i].botborder;
    if(directions[i].fitgrid == true)
    {
      overflowy += directions[i].ystride;
    }
    //Warning that overflow may be uninit
    for(y = 0 + directions[i].topborder; y
      < (overflowy + directions[i].topborder);
```

```
  y+=directions[i].ystride)
{
  overflowx = directions[tlayer].ogrid_x
    - directions[i].leftborder
    - directions[i].rightborder;
  if(directions[i].fitgrid== true
    && (overflowx % directions[i].xstride > 0))
  {
    overflowx += directions[i].xstride;
  }
  for(x = 0 + directions[i].leftborder; x
    < (overflowx + directions[i].leftborder);
    x+=directions[i].xstride)
  {
    m = 0;
    for(innery = y; innery < (y
      + directions[i].igrid_y); innery++)
    {
      if(innery < 0 || innery
        >= directions[tlayer].ogrid_y) continue;
      for(innerx = x; innerx < (x
        + directions[i].igrid_x); innerx++)
      {

        k = (innery *
          directions[tlayer].ogrid_x) + innerx;
        if(innerx < 0 || innerx
          >= directions[tlayer].ogrid_x
        || k >= toutputs.size())
```

```
  continue;
pre = toutputs[k].second;
post = m;
apre = cnetnodes[tlayer]
  + toutputs[k].first * nets[tlayer]
  ->num_nodes() + pre;
if(directions[i].tmux == false)
{
  apost = cnetnodes[i] + n * nets[i]
    ->num_nodes()
    + nets[i]->get_input(post)->id;
  edges[i][n].push_back(
      rv->add_edge(apre, apost));
  edges[i][n][edges[i][n].size() -1]
    ->set("Weight", rv
    ->get_edge_property("Weight")
    ->max_value);
  edges[i][n][edges[i][n].size() -1]
    ->set("Delay", directions[i]
    .inputdelay);
}
else
{
  apost = cnetnodes[i] + post;
  makePathWithDelay(rv, rv
    ->get_node(apre), rv->
    get_node(apost), n* directions[i]
    .tmuxPeriod +1+mind*2, maxd, mind,
    mint, maxw, maxw, rv->num_nodes(),
```

```
                        minr, refkey, 0, false);
                    }
                    m++;
                  }
                }
                l++;
                n++;
              }
            }
          }
        }
      }
    }
    for(i = 0; i < directions.size(); i++)
    {
      if(directions[i].generateNet == true)
      {
        delete(nets[i]);
      }
    }
    return rv;
}
```

Vita

Originally from Florida, Nicholas Skuda grew up in Knoxville, Tennessee. After high school, he attended the University of Tennessee, Knoxville and received a Bachelor of Science degree in Computer Science. After working as a undergraduate research assistant with the TENNLab neuromorphic computing research group, he knew he wanted to continue into graduate school. He chose to continue his work at the University of Tennesse, Knoxville to pursue a Dcotor of Philosophy degree in Computer Science with a concentration in Intelligent Systems and Machine Learning. His research interests include neuromorphic computing, evolutionary and genetic algorithms, and device simulation.