# Neuron Clustering for Mitigating Catastrophic Forgetting in Supervised and Reinforcement Learning

A Dissertation Presented for the

Doctor of Philosophy

Degree

The University of Tennessee, Knoxville

Benjamin Frederick Goodrich

December 2015

© by Benjamin Frederick Goodrich, 2015
All Rights Reserved.

# Acknowledgements

There have been a lot of people over the years who have encouraged me, and who opened doors and provided opportunities to make this work possible. First of all, I want to thank Dr. Michael J. Roberts who taught several of my courses during my undergraduate years at The University of Tennessee. He was one of the strongest influences in encouraging me to go on to graduate school.

The last few years in my graduate program, I've had many colleagues and friends who I want to thank. I want to thank everyone in my lab who have been there to bounce around research ideas and who have been very supportive. Some of the ideas presented in this dissertation began during discussions with these people. This includes Aaron Mishtal, Andrew Davis, Derek Rose, Steven Young, Tomer Lancewicki, Benjamin Martin, Nicole Pennington, and Bobby Coop. I also want to thank Dr. Harry Richards, who was the program manager of the SCALE-IT program and who has been an excellent mentor and friend. I also want to thank my gym trainer Mariah Melancon who helped me to eat properly and exercise, which turned out to be absolutely critical for my mental and physical health during the last year especially.

Finally, perhaps most importantly, I want to thank my advisor Dr. Itamar Arel who has been the most helpful and influential person in encouraging me to finish for the past few years.

# Abstract

Neural networks have had many great successes in recent years, particularly with the advent of deep learning and many novel training techniques. One issue that has afflicted neural networks and prevented them from performing well in more realistic online environments is that of catastrophic forgetting. Catastrophic forgetting affects supervised learning systems when input samples are temporally correlated or are non-stationary. However, most real-world problems are non-stationary in nature, resulting in prolonged periods of time separating inputs drawn from different regions of the input space.

Reinforcement learning represents a worst-case scenario when it comes to precipitating catastrophic forgetting in neural networks. Meaningful training examples are acquired as the agent explores different regions of its state/action space. When the agent is in one such region, only highly correlated samples from that region are typically acquired. Moreover, the regions that the agent is likely to visit will depend on its current policy, suggesting that an agent that has a good policy may avoid exploring particular regions. The confluence of these factors means that without some mitigation techniques, supervised neural networks as function approximation in temporal-difference learning will be restricted to the simplest test cases.

This work explores catastrophic forgetting in neural networks in terms of supervised and reinforcement learning. A simple mathematical model is introduced to argue that catastrophic forgetting is a result of overlapping representations in the hidden layers in which updates to the weights can affect multiple unrelated regions

of the input space. A novel neural network architecture, dubbed "cluster-select," is introduced which utilizes online clustering for the selection of a subset of hidden neurons to be activated in the feedforward and backpropagation stages. Cluster-select is demonstrated to outperform leading techniques in both classification and regression. In the context of reinforcement learning, cluster-select is studied for both fully and partially observable Markov decision processes and is demonstrated to converge faster and behave in a more stable manner when compared to other state-of-the-art algorithms.

# Table of Contents


**1 Introduction** **1**

**2 Background and Related Work** **4**

2.1 Neural Networks . . . . . 4

2.1.1 Artificial Neuron Model . . . . . 5

2.1.2 FeedForward Network . . . . . 6

2.1.3 Recurrent Neural Networks . . . . . 9

2.2 Reinforcement Learning . . . . . 10

2.2.1 Fully Observable MDPs . . . . . 11

2.2.2 Reinforcement Learning Problem . . . . . 12

2.2.3 Partially Observable MDPs . . . . . 12

2.2.4 Q-Learning . . . . . 13

2.2.5 SARSA(0) . . . . . 16

2.3 Catastrophic Forgetting Overview . . . . . 17

2.3.1 Defining Catastrophic Forgetting . . . . . 18

2.3.2 Existing Network Architectures . . . . . 22

2.3.3 Catastrophic Forgetting in Control Problems . . . . . 28

**3 A Neuron Clustering Approach** **31**

3.1 Motivation . . . . . 31

3.2 Analysis . . . . . 33

3.3 The Cluster-Select Approach . . . . . 36

3.3.1 Feedforward Implementation Details . . . 38
3.3.2 Covariance Estimation . . . 41

**4 Mitigating Catastrophic Forgetting in Classifier and Regression Problems 43**

4.1 Feedforward with Cluster-Select . . . 43
4.2 Recurrent Network . . . 46
4.2.1 Online Non-stationary Task . . . 46
4.2.2 Recurrent Training Details . . . 47
4.3 Simulation Results and Analysis . . . 49
4.3.1 MNIST Experiment . . . 50
4.3.2 MNIST Experiment with Noise . . . 53
4.3.3 20 Newsgroups Experiment . . . 55
4.3.4 Autoassociative Encoder Experiment . . . 56
4.3.5 Reduced MNIST Experiment . . . 58
4.3.6 Experiment with Gas Sensor Array Dataset . . . 60
4.3.7 Pendulum Experiment . . . 61

**5 Mitigating Catastrophic Forgetting in Reinforcement Learning Environments 68**

5.1 Forgetting in MDPs . . . 69
5.1.1 Cart-Pole Experiment . . . 70
5.2 POMDP Environment . . . 75
5.2.1 Partially Observable Cart-Pole Experiment . . . 75
5.2.2 Arcade Learning Environment Experiment . . . 77

**6 Conclusions 87**

6.1 Summary of Contributions . . . 87
6.2 Future Work . . . 88
6.3 Concluding Remarks . . . 90

6.4 Publications . . . . . 90

**Bibliography** **92**

**Vita** **102**

# List of Tables

5.1 Constants for Cart-Pole Test . . . . . 70
5.2 Summary of Results . . . . . 75

# List of Figures


2.1 Artificial Neuron Model . . . . . 6

2.2 Example model of a neural network with a single hidden layer. $X$ is the matrix of inputs fed in, $Y$ is the matrix of hidden outputs, and $Z$ is the matrix containing the final network outputs. The two weight matrices $W^{(1)}$ and $W^{(2)}$ correspond to the input to hidden and hidden to output weights respectively. . . . . 7

2.3 The popular mini-batch gradient descent algorithm, used for supervised training on large datasets . . . . . 9

2.4 Model of an Elman Network. The hidden outputs from the previous time step are provided as inputs for the next time step. Note that the number of inputs, hidden, and output nodes can vary from the number depicted here. . . . . 11

2.5 Illustration of Catastrophic Forgetting . . . . . 21

3.1 Illustration of the cluster-select process . . . . . 37

4.1 Training algorithm used for mini-batch training with cluster-select . . 46

4.2 How MNIST was split into $P1$ and $P2$ . . . . . 50

4.3 P1 miss rate vs. P2 miss rate possibilities frontiers for MNIST forgetting task . . . . . 51

4.4 Results for MNIST Test with noise . . . . . 54

4.5 P1 miss rate vs. P2 miss rate possibilities frontiers for 20 newsgroups dataset forgetting task . . . . . 57
4.6 $P1$ Miss Rate vs. $P2$ Miss Rate Possibilities Frontiers for Autoassociative Encoder Forgetting Task . . . . . 58
4.7 Cluster-Select MNIST Result . . . . . 60
4.8 Cluster-Select Gas Sensor Array Dataset Result . . . . . 61
4.9 Sample of $P1$ and $P2$ datasets. This plot shows $sin(\theta)$ and $-cos(\theta)$ which is the data that is fed to the neural network. The data shown indicates the relative x and y position of the pendulum if it is swinging around the origin. In 4.9(a) the pendulum swings completely around in circles. In 4.9(b) however, the pendulum is swaying back and forth. To see that the pendulum is not swinging around, notice the y position $-cos(\theta)$ does not swing higher than $-0.5$ units. . . . . 64
4.10 Illustration of $P1$ and $P2$ error rates during training on the recurrent simple pendulum task . . . . . 65
4.11 Selected Neurons During the Regime Change . . . . . 66

5.1 Result for a Tabular $Q_{s,a}$ Estimator . . . . . 73
5.2 Result for Cluster-Select Neural Net $Q_{s,a}$ Estimator . . . . . 73
5.3 Result for a Neural Net $Q_{s,a}$ Estimator with Linear Rectified Activations 74
5.4 Result for a Neural Net $Q_{s,a}$ Estimator with Sigmoid Activations . . . 74
5.5 Result for a Neural Net $Q_{s,a}$ Estimator with Hyperbolic Tangent Activations . . . . . 74
5.6 Result for Cluster-Select Neural Net $Q_{s,a}$ Estimator on a POMDP Cart-pole Test Case . . . . . 77
5.7 Modified algorithm from deep Q-learning with experience replay. . . . 79
5.8 Plot of Number of Episodes Before Achieving a Score (Convergence Speed) . . . . . 85

5.9 Number of Consecutive Episodes which were Greater Than a Particular Score (Stability) . . . . . 86

# Chapter 1

# Introduction

Catastrophic forgetting is a problem that affects artificial neural networks as well as other learning systems [1]. When a network with a global shared pool of parameters is trained on one task, then trained on a second task, it will rapidly exhibit degraded performance on the first task. This problem significantly impacts application domains in which neural networks can be employed, as it exposes difficulties in operating when online or non-stationary settings are considered.

Traditionally, the data must be selected in a way that makes it appear stationary such that samples are independently and identically distributed (i.i.d.) Training data is generally shuffled and presented in a random order [2]. Should the data be presented in a non-stationary manner, the network may not adequately capture the representations pertaining to all samples due to temporal bias associated with a particular subset of samples.

While training offline allows for drawing samples in an i.i.d. manner, there are many online learning scenarios in which one does not have the convenience of determining the order of the training samples a priori. An example of such online task is reinforcement learning with a large state space. When using neural networks for value function approximation, sequences of samples presented to the network typically pertain to a small region of the state space. Should there be a region that is

not visited as often as other regions, the network based representation for that region will rapidly degrade. Limitations of neural networks when applied to reinforcement learning have been recognized and explored in the past with limited success [3] [4] [5] [6] [7]. Recent work has introduced techniques aimed at mitigating catastrophic forgetting which have been successfully applied to deep neural networks in playing Atari games [8]. However, the solutions proposed have notable scalability limitations. One major goal of this dissertation is to investigate reinforcement learning as a test case for studying techniques that mitigate catastrophic forgetting.

Environments in which actions are taken, and observations and rewards are generated sequentially are common to living organisms. Similar to situations faced by agents in a reinforcement learning scenario, biological creatures typically receive sequences of observations that are correlated as they pertain to a common underlying state of the environment. This stands in contrast to the common way in which neural networks are trained where consecutive samples are assumed to be uncorrelated.

Catastrophic forgetting may contribute to degraded performance even in stationary settings, particularly in the context of deep learning systems involving very large datasets. A network that captures characteristics and features of a large dataset must also be made large, suggesting more neurons or layers must be allocated. In [9] a law of diminishing returns is demonstrated, whereby adding capacity to the network ceases to contribute capturing of new representations. Catastrophic forgetting may be at play, since for large datasets, key characteristics will be presented less frequently, and the network will not be able to adequately fit to those features before they are lost.

In the past, substantial research was aimed at addressing catastrophic forgetting [10] [11] [12] [13] [14]. The topic has grown in interest recently [15] [16] [17]. In this work, a technique will be devised which will be demonstrated to mitigate catastrophic forgetting in non-stationary classification and regressing settings, as well as the more challenging case of reinforcement learning.

An outline for the rest of this dissertation is as follows. Chapter 2 will introduce background information necessary for the rest of the dissertation. A brief review of

neural networks will be given in order to establish notation and context for discussing catastrophic forgetting. A review of reinforcement learning will set the stage for exposing catastrophic forgetting. Finally, a brief review of the existing techniques that have been proposed as means of reducing the impact of catastrophic forgetting will be provided.

Chapter 3 begins by describing a mathematical model used to derive insight into the catastrophic forgetting phenomenon. The main theoretical contribution reveals how overlapping representations trigger catastrophic forgetting. This is followed by the introduction of the main technique presented in this dissertation, dubbed "cluster-select." The latter uses online clustering as a form of unsupervised learning to select (or mask) neurons during the feedforward phase. Each neuron is associated with a centroid in addition to its weights and only neurons that have centroids that are nearest to the sample point are selected. This effectively creates overlapping sub networks out of a large network and reduces overlap between samples that belong to different regions.

Chapter 4 explores applying cluster-select to several classification and regression test cases. For classification, the data samples are constructed in a non-stationary manner by switching class labels during training. The regression test case involves a time series prediction problem for a pendulum. The pendulum behaves differently depending on the speed at which it swings which yields non-stationary behavior and corresponding samples.

Chapter 5 addresses the more interesting test case of reinforcement learning. Cluster-select is first applied to a cart-pole balancing reinforcement learning task in both fully and partially observable settings. Cluster-select is shown to perform better than alternative neural network architectures on the cart-pole test case. Next, the pong video game of the arcade learning environment [18] is investigated where we compare cluster-select to using replay buffer which was successfully demonstrated in recent work [8]. Finally, chapter 6 provides a summary of dissertation contributions along with concluding thoughts.

# Chapter 2

# Background and Related Work

The purpose of this chapter is to introduce background material which will be necessary to provide context for the rest of this dissertation. Three main topics are covered, the first being an overview of artificial neural networks, since it is necessary to understand neural networks before discussing catastrophic forgetting. Next reinforcement learning will be reviewed because it provides an environment in which catastrophic forgetting naturally occurs. The final topic of discussion is catastrophic forgetting. Prior work on catastrophic forgetting will be outlined where existing approaches that mitigate forgetting will be covered. The chapter concludes with a discussion of related work that encountered forgetting in reinforcement learning in addition to related work that examined forgetting in biological systems.

## 2.1 Neural Networks

Before we can begin to discuss catastrophic forgetting in supervised learning, it is necessary to provide a brief overview of neural networks. This section will serve as a brief refresher, and to establish some notation that will be used. By no means is this meant to be a comprehensive overview of the field of neural networks. More in depth discussion can be found in [19], [20], and [2].

As a general overview, neural networks provide a way to approximate a nonlinear function, provided training data. That is, when shown a set of input and output pairs, an artificial neural network can adjust its internal weights to produce the desired input output pair. The real power of neural networks is in their ability to generalize. If a neural network is trained on sufficient data, it can learn to produce useful outputs for input data that has not been presented during training.

### 2.1.1 Artificial Neuron Model

The perceptron [21] is the most common type of artificial neuron. Figure 2.1 shows the typical behavior of this type of neuron. Mathematically, it takes an input vector $X = [x_0, , x_1, ..., x_{n-1}, x_n = 1.0]$ and computes an activation value simply by taking the dot product of the input vector with internal weights. $a = \vec{x} \cdot \vec{w}$. This activation value is passed through a nonlinear activation function.

Most activations functions are sigmoidal in nature, meaning the function is 'S' shaped. An activation function that was very popular in the past, and remains commonly used is the logistic function $f(a) = \frac{1}{1+e^{-a}}$. For multiclass problems, a popular extension of this activation function is softmax, which can be defined as $f(a_i) = \frac{e^{a_i}}{\sum_j^C e^{a_j}}$. Softmax is used for the output layer, and it builds an output probability distribution for each of the $j$ classes that have activation values $a_j$. Another popular activation function is $a \times tanh(b \times x)$ where $a = 1.7159$ and $b = 2/3$ were recommended in [2]. In recent years, rectified linear activations have been found to perform well in deep neural networks. Rectified linear is defined as $f(a) = a$ if $(a > 0), 0$ otherwise. Rectified linear activation functions are not sigmoidal, but they encourage a more sparse representation, and the gradient is signal is stronger during the backpropagation phase, making them ideal for deeper networks.

Note that the input vector is internally augmented to contain a "1", this is an internal input, known as the bias input and allows the activation to be nonzero for

an input that consists of zeros. This perceptron neuron model is very common and is sometimes referred to simply as an "Artificial Neuron."

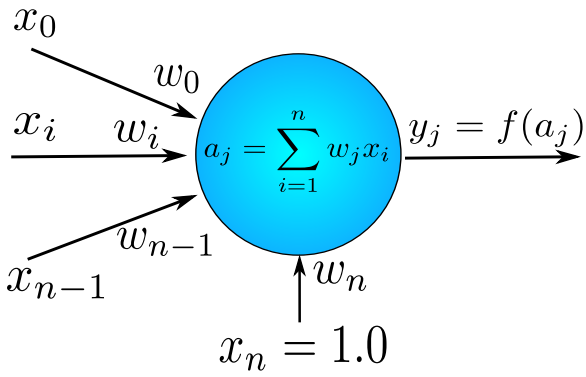



**Figure 2.1:** Artificial Neuron Model

### 2.1.2 FeedForward Network

Artificial neurons can be linked together in a directed weighted graph to form a feedforward neural network. The most common type of feedforward neural network is a multilayer perceptron *.

Multilayer perceptrons consist of a layer of input neurons, one or more hidden layers, and an output layer. Figure 2.2 shows a typical fully connected arrangement with a single hidden layer. The input layer is not technically a layer of neurons, but is simply a placeholder for the inputs to the network.

#### Gradient Descent Training

Typically after a set of inputs is fed forward to produce a set of outputs, an error or cost function $e$ is computed. Mean squared error (MSE) is a natural choice for the cost function because it yields the maximum likelihood estimator. If $\vec{z}$ is the vector of outputs from the network, and $\vec{t}$ is the target, then the mean squared error gives $e = (t - z)^2$. An algorithm known as backpropagation allows computing of

*In literature, the terminology is often mixed. Some papers refer to multilayer perceptrons as feedforward networks, or even simply as artificial neural networks.

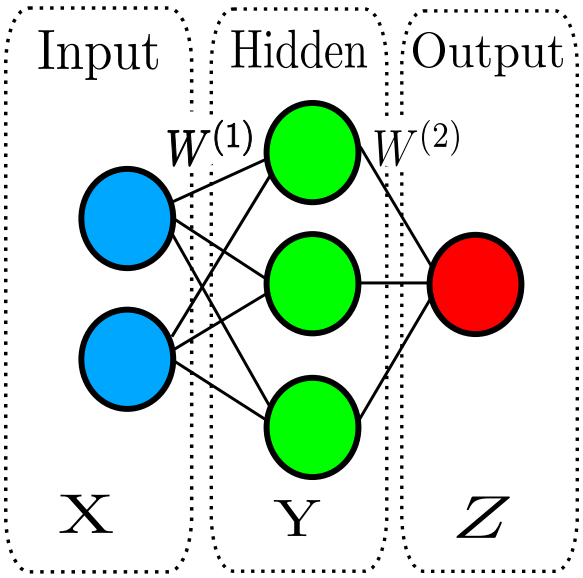



**Figure 2.2:** Example model of a neural network with a single hidden layer. $X$ is the matrix of inputs fed in, $Y$ is the matrix of hidden outputs, and $Z$ is the matrix containing the final network outputs. The two weight matrices $W^{(1)}$ and $W^{(2)}$ correspond to the input to hidden and hidden to output weights respectively.

the gradient of each weight with respect to this error function. This allows gradient descent based weight updates to occur during training. To find more information on gradient descent and backpropagation, see [19]

### Matrix Computation for Neural Networks

Neural network computations can be reduced to large matrix operations if one is careful in setting up the matrices. This allows obtaining multiple samples at once. For computing the feedforward pass, the samples can be placed in a matrix denoted $X \in \mathbb{R}^{n \times k}$ with $n$ being number of inputs and $k$ the number of samples (i.e. each sample is a column vector in this matrix). The neural network weights from the input layer to hidden layer can be placed in a matrix denoted $W^{(1)} \in \mathbb{R}^{h \times n}$ where $h$ is number of hidden neurons and $n$ is number of inputs (i.e. each neuron weight vector is a row in this matrix). If the inputs and weights are set up in this manner, then a feedforward operation from input to hidden can be computed as $Y = f(W^{(1)}X)$ where $f(\cdot)$ performs the nonlinear activation function on each element of the input

matrix. This hidden output matrix $Y$ can be further fed-forward again to the output layer by performing $Z = f(W^{(2)}Y)$. Note the $W^{(2)}$ matrix is the weights from the hidden to output layer. This will produce an output $Z$ that contains the network output for all of the samples. These matrices, $X$, $Y$ $Z$, $W^{(1)}$, and $W^{(2)}$ are labeled in figure 2.2.

It is also possible to define backpropagation as a series of matrix operations as well, see [2] for details. This allows computing a gradient that reduces the error for all of the samples that are fed through. Highly optimized CPU and GPU libraries exist for matrix operations. The ability to scale to parallel architectures makes neural networks very attractive from a research standpoint. This attractiveness is part of the reason that neural network are now popular computational tools to use.

Neural networks lost their popularity in the late 90s due to greater success when using support vector machines. However, they have enjoyed a renewed popularity starting in the mid 2000s. There are several reasons for this renaissance. The first reason is the emergence of big data, that is very large datasets that are the result of cheaper storage and the fact that in many cases data collection is much easier to perform. The second reason is the emergence of deep learning. Neural networks that are much deeper in the sense that they have many hidden layers can be trained that are capable of making more powerful generalizations. All of this is ultimately due to the final reason which is the resurgence of parallel architectures and the fact that neural network training can be performed as a series of parallel operations. This has led to a computational revolution in the speed at which training deep networks can be performed on large datasets, making neural networks very powerful tools that can be applied to many problem domains.

### Mini-batch Gradient Descent

The most common training algorithm for a large dataset is mini-batch gradient descent shown in Figure 2.3 below. This algorithm iterates through the full dataset on each epoch. On each iteration, it divides the full dataset up into mini-batches, each

**Algorithm 1:** Mini-batch Gradient Descent

initialize mini-batch size hyperparameter $m$
initialize training dataset $\hat{X}$
**for** *each epoch of training* **do**
    divide training data $\hat{X}$ into smaller portions $\hat{X}_i$, each containing $m$ samples.
    **for** *each mini-batch* **do**
        select a mini-batch $\hat{X}_i$ and put it into $X$
        feedforward $X$ through the network
        compute output error
        backpropagate to compute weight updates
        apply weight updates
    **end**
    shuffle training data
**end**

**Figure 2.3:** The popular mini-batch gradient descent algorithm, used for supervised training on large datasets

containing $m$ samples. The training algorithm feeds a mini-batch in as a whole and applies a weight update to minimize the error for that mini-batch. The training data is shuffled to randomize the order that each sample is presented to the network which prevents oscillations in convergence. Shuffling the training data may potentially be a slow operation, hence it may be preferable to only perform the shuffling every n epochs where n is a small integer.

This algorithm naturally helps prevent catastrophic forgetting primarily because the data is shuffled and presented in batches. This ensures that each weight update minimizes an independent and identically distributed collection of samples and thus guarantees that the error gradient does not favor a set of samples that are correlated in any way.

### 2.1.3 Recurrent Neural Networks

Some training tasks require a network to capture temporal dependencies in the training data. Time series prediction, for example, may require future predictions to

be made based on past observations. To accomplish this, a recurrent neural network is used. Recurrent networks are designed to capture temporal regularities in the data.

### Elman networks

One common type of recurrent neural network is known as an Elman network [22]. The latter achieves recurrence by taking the outputs from the hidden layer and passing them in as inputs for the subsequent time step. See Figure 2.4 for an illustration of an Elman recurrent network.

Elman networks can be trained by stepping through a temporal dataset and calculating errors at each time step. During training, it is also possible to unfold the network through time and backpropagate through multiple time steps into the past. This leads to a technique known as backpropagation through time [23]. Unfortunately, backpropagation through time for multiple time steps leads to a problem known as the vanishing gradient problem [24], where the gradients for the weight updates rapidly shrink for each layer of backpropagation. Another issue is the exploding gradient problem, where gradients rapidly grow in size, causing a very large weight update that can corrupt the network weights if it isn't treated properly [25].

## 2.2 Reinforcement Learning

This section introduces reinforcement learning which provides an environment that is naturally vulnerable to catastrophic forgetting. We begin by defining reinforcement learning in terms of fully observable Markov decision processes (MDPs), then we cover partially observable MDPs (POMDPs) which reflect the more realistic setting of partial observability of the environment. Next we cover two popular reinforcement learning algorithms: SARSA and $Q$-learning which will be used in Chapter 5. Finally we review existing work on catastrophic forgetting in the reinforcement learning context.

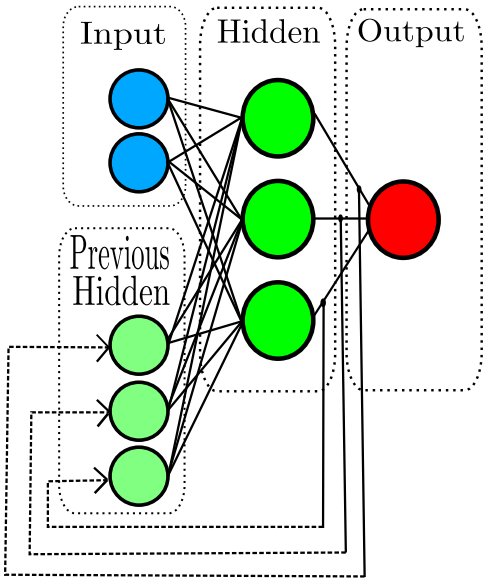



**Figure 2.4:** Model of an Elman Network. The hidden outputs from the previous time step are provided as inputs for the next time step. Note that the number of inputs, hidden, and output nodes can vary from the number depicted here.

Reinforcement learning [26] is a branch of machine learning that is concerned with training an agent to learn from an environment by collecting rewards. The agent observes the environment to obtain a state, or in some cases a partial state known as an observation. From these observations the agent must take an action to collect a reward and achieve a new state. By learning to take the action that leads to the greatest overall reward, the agent achieves the desired behavior.

### 2.2.1 Fully Observable MDPs

Formally, the fully observable case of reinforcement learning (RL) can be framed as a Markov decision process (MDP) which is defined by the tuple $(S, A, P, R)$ where $S$ is the state space, $A$ is the space of possible actions, $P : S \times A \times S \mapsto [0, 1]$ is the state transition probability function, $R : S \times A \mapsto \mathbb{R}$ is the reward function. A policy $\pi$ maps states to actions $\pi : S \mapsto A$.

### 2.2.2 Reinforcement Learning Problem

Let the state space $S$ be $S = (s_1, s_2, ..., s_n)$ and, accordingly, the action space $A$ be $A = (a_1, a_2, ..., a_n)$. Suppose at episode $k$, the agent detects $S_k = s \in S$, the agent chooses an action $a_k = a \in A(s_k)$ according to policy $\pi$ in order to interact with the environment. Next, the environment transitions into a new state $s_{k+1} = s' \in S$ with the probability $P_{ss'}(a)$ and provides the agent with a feedback reward denoted by $r_k(s, a)$. The process is then repeated. The goal for the RL agent is to maximize the expected discounted reward, or state-value, which is represented as:

$$V^{\pi}(s) = E_{\pi}\left\{\sum_{k=0}^{\infty}\gamma^{k} r_k(s_k, \pi(s_k))|s_0 = s\right\} \tag{2.1}$$

where $\gamma(0 \leq \gamma < 1)$ is the discount factor and $E_{\pi}\{\}$ denotes the expected return when starting in $s$ and following policy $\pi$ thereafter. The equation above can be rewritten as

$$V^{\pi}(s) = R(s, \pi(s)) + \gamma \sum_{s' \in S} P_{ss'}(\pi(s)) V^{\pi}(s'), \tag{2.2}$$

where $R(s, \pi(s)) = E\{r(s, \pi(s))\}$ is the mean value of the reward $r(s, \pi(s))$.

The reinforcement learning problem involves finding a policy $\pi$ that maximizes discounted reward according to equation 2.2. In practical situations, many of the above parameters are unknown, and must be somehow learned or estimated. For example, the transition probability $P_{ss'}(a)$ and the reward function $R(s, \pi(s))$ are unknown, making the problem more difficult. Moreover, in more challenging situations as discussed next, the state construct may not even be fully known.

### 2.2.3 Partially Observable MDPs

Thus far an unrealistic assumption has been made, that the environment is fully observable. By modeling the state as an explicit variable we are assuming that the agent can instantaneously know everything about its current environment. For

example, consider a robot maneuvering in an environment, full observability presumes the robot would have full knowledge of where all objects of interest are located. In a more practical setting, the agent can only know what is presently being observed from the environment. Observations could include for example, sensor readings or other measurements of the environment. This implies a partially observable Markov decision process (POMDP) where instead of a state $s$ we are given an observation $o$. Therefore, the state must somehow be inferred from a sequence of observations.

State inference necessitates some type of machine learning algorithm, or a neural network that can learn to infer a model of the environment. One simple way to model the environment is to train a recurrent neural network to predict the next observation given past observations. If the network can successfully learn to generate such predictions, it suggests the hidden layers of the network represent the current belief about the state of the environment. Under this type of framework, a recurrent network is used to derive the true state $S$ by learning to predict observations $o \in \Omega$.

The POMDP Model can formally be described as a tuple $(S, A, P, R, O, \Omega)$. It can be assumed that the world can be described as an MDP, however, our agent only receives partial state through observations. Hence as before, there is an MDP model where $S$ is the state space, $A$ is the space of possible actions, $P$ is the state transition probability function, and $R$ is the reward. In addition, there are extra components for a POMDP. $\Omega$ is the set of all possible observations $o \in \Omega$. The observation function $O : S \times A \mapsto p(\Omega)$ maps an action and subsequent state to a probability of receiving an observation.

### 2.2.4 Q-Learning

$Q$-learning [27] is one of the most effective and popular algorithms for learning from delayed rewards in absence of the transition probability and reward function. In $Q$-learning, policies and the value function are represented by a lookup table indexed by state-action pairs. Formally, for each state $s$ and action $a$, we define the $Q$ value

under policy $\pi$ to be:

$$Q^{\pi}(s,a) = R(s,a) + \gamma \sum_{s' \in S} P_{ss'}(a) V^{\pi}(s') \tag{2.3}$$

which reflects the expected discounted rewards from following policy $\pi$.

We can define the optimal value function as the value function when following the optimal policy. It follows an important identity known as the Bellman optimality equation given below.

$$\begin{aligned} Q^*(s,a) &= \mathbb{E}\left\{ r + \gamma \underset{a'}{\max} Q^*(s',a') \mid s,a \right\} \\ &= \sum_{s'} P^a_{ss'} \left[ R^a_{ss'} + \gamma \underset{a'}{\max} Q^*(s',a') \right] \end{aligned} \tag{2.4}$$

The Bellman optimality equation describes the optimal value at time $t$ if we know the optimal value at time $t+1$. The premise behind Q-learning is to estimate the Q function via an iterative update known as value iteration. This effectively propagates the Q-values backwards in time, and over many updates it is guaranteed to converge to the optimal value. Correspondingly, the state-action value update rule is given by

$$Q_{k+1}(s,a) = Q_k(s,a) + \alpha \delta_k, \tag{2.5}$$

where $\delta_k$ is temporal difference error defined (for $Q$-learning) as

$$\delta_k = r_k + \gamma \max_{a' \in A(s')} Q_k(s', a') - Q_k(s,a), \tag{2.6}$$

and $\alpha$ the learning rate.

It is impractical to store these updates in a table form when addressing a large state and/or action space. Due to the curse of dimensionality, the table size grows exponentially with each added dimension. Moreover, tables do not offer any form of generalization since an update to one table cell will not affect other cells. Function approximation techniques offer a practical solution when large spaces are considered

and can effectively replace the tabular form for value function estimation of the value function ($Q_k(s,a)$) [26]. Q-learning based update equations can be derived for a generic function approximation technique with parameters at iteration $k$ denoted by $\theta_k$. Training is achieved by gradient descent methods by defining the loss function $L(\theta_k)$ in a way that minimizes error between the network outputs and the temporal difference error, such as

$$L(\theta_k) = \mathbb{E}\left[\left(y_k - Q\left(s,a;\theta_k\right)\right)^2\right]. \tag{2.7}$$

The above loss function takes the expectation over the distribution of states and actions that are followed (known as the behavior distribution), where $y_k$ is defined as

$$y_k = \mathbb{E}\left[r + \gamma \max_{a'} Q\left(s',a';\theta_{k-1}\right)\right]. \tag{2.8}$$

Taking the derivative of this loss function with respect to the parameters produces the following gradient expression

$$\nabla_{\theta_k} J(\theta_k) = \mathbb{E}\left[\left(r + \gamma \max_{a'} Q\left(s',a';\theta_{k-1}\right) - Q\left(s,a;\theta_k\right)\right)\nabla_{\theta_k} Q\left(s,a;\theta_k\right)\right]. \tag{2.9}$$

If stochastic gradient descent is used, the expectation can be removed from the above equation. With a small enough learning rate $\alpha$, the solution will converge to the expectation, given the following update rule

$$\theta_{k+1} = \theta_k + \alpha\left(r + \gamma \max_{a'} Q_{\theta_{k-1}}\left(s',a'\right) - Q_{\theta_k}\left(s,a\right)\right)\nabla_{\theta_k} Q\left(s,a;\theta_k\right) \tag{2.10}$$

$$= \theta_k + \alpha\delta_k \nabla_{\theta_k} Q\left(s,a;\theta_k\right). \tag{2.11}$$

Note that $\delta_k$ in equation 2.11 matches the definition in equation 2.6

The preceding definitions assume any generic gradient based function approximation technique with parameter vector $\theta$, however it should be noted that a standard feedforward neural network can be used as the function approximation technique, in

which case the parameters $\theta$ would instead be the individual weights of the neural network. As far as implementing this with a neural network, the $\delta_k$ value can simply be backpropagated as an error term to compute the weight updates according to 2.11.

It should be noted that Q-learning is an off-policy method, meaning that it approximates the value of the greedy policy $\pi(a) = \max_a Q(s, a)$ while following an exploratory policy. One popular technique of exploration is $\epsilon$-greedy, which chooses the greedy action with probability $1 - \epsilon$ and selects a random action otherwise. An exploratory policy is important to ensure adequate coverage of the state/action space.

### 2.2.5 SARSA(0)

SARSA is a popular on-policy learning technique that provides an alternative to $Q$-learning. SARSA uses an update similar to Q-learning, with a key difference that the $\delta_k$ is instead given by

$$\delta_k = r_k + \gamma Q_k(s^{'}, a^{'}) - Q_k(s, a). \tag{2.12}$$

Unlike $Q$-learning, SARSA is an on-policy method, meaning that it approximates the value of the current policy being followed instead of learning the value of the optimal policy. The current policy being followed generally consists of greedy actions in addition to some exploratory actions chosen via $\epsilon$-greedy strategies. Choosing greedy actions that have the highest value leads to improvements in the current policy, and as the policy improves, the estimate of the $Q$ value will converge to the optimal value function and the policy will converge to the optimal policy.

#### Eligibility Traces

A useful extension of SARSA is a mechanism known as eligibility traces which allows for faster convergence. The main premise behind eligibility traces is that since the temporal difference updates are applied backwards in time (notice equation 2.5 applies the update from a future time step to a previous time step), a performance boost in learning can be achieved if credit is assigned more than one time step into the

past. Credit assignment into the past can be accomplished by maintaining a history of visited states and applying the current credit assignment update on a decaying subset of past states. Eligibility traces can only be used with on-policy methods since a history of state visitations can only be maintained for the policy that was followed. A parameter $\lambda$ is used to control eligibility, and SARSA with eligibility traces is generally referred to as SARSA($\lambda$).

For gradient descent training techniques, the parameter updates for a popular version of eligibility traces called gradient-descent SARSA is defined as follows

$$\theta_{k+1} = \theta_k + \alpha\delta_k e_k \tag{2.13}$$

$$e_k = \lambda\gamma e_{k-1} + \nabla_{\theta_k} Q\left(s, a; \theta_k\right) \tag{2.14}$$

with $\delta_k$ given in equation 2.12. The update strategy for gradient-descent SARSA essentially maintains a decaying history in $e_k$ of the most recent weight gradients $\nabla_{\theta_k} Q\left(s, a; \theta_k\right)$, and applies the credit assignment to this history of previous gradients, which effectively applies the credit assignment multiple time steps into the past. Note that here, we assume that an update is performed on every time step, hence both the update iteration and the time step are indexed by $k$.

When considering implementing this technique, one can backpropagate the constant 1.0 through the network. The computed gradient update for the weight matrices will correspond to $\nabla_{\theta_k} Q\left(s, a; \theta_k\right)$. From these weight matrices, one can maintain $e_k$ by iteratively applying equation 2.14 on each time step. The gradient update can be computed according to equation 2.13.

## 2.3 Catastrophic Forgetting Overview

Catastrophic forgetting is a well-studied topic in machine learning. This section serves to provide an overview of catastrophic forgetting. First, catastrophic forgetting is defined. Next, a brief overview is provided of existing approaches to mitigating

forgetting including existing neural network architectures and training techniques that have been developed over the years specifically aimed at non-stationary datasets. In addition, related work is covered which treats the issue of catastrophic forgetting in the context of reinforcement learning.

### 2.3.1 Defining Catastrophic Forgetting

Catastrophic forgetting is a term that has been used for decades by researchers to describe a phenomenon that has been observed while training neural networks. Despite the fact that it has been recognized and well studied, a survey of existing literature reveals that catastrophic forgetting seems to have multiple definitions. A common scenario in which forgetting occurs involves a network trained to perform task A then task B. When the network is evaluated on task A after learning task B, its ability to perform task A sharply degrades. [17].

An alternative definition for forgetting is derived from the storage capacity of the network. A neural network is viewed as storing information in the form of mapping input examples to output examples. Under this assumption, forgetting occurs when the network is presented with novel input examples which it must store. Catastrophic forgetting refers to the observed phenomenon where prior information appears to be suddenly erased as a consequence of new information being stored [10]. Another approach to defining catastrophic forgetting is in terms of sequential learning environments where information to be learned arrives over time. Neural networks perform poorly at such tasks due to catastrophic forgetting, where learning to represent newly arrived information disrupts prior representation [28].

The different perspectives for describing forgetting appear to arrive at the same paradigm. Supervised learning implicitly assumes that samples are drawn in an i.i.d. manner from a stationary distribution. Catastrophic forgetting emerges any time this assumption is violated. Weight updates to minimize error for the current samples may

not necessarily minimize error for previously presented samples, and the observed behavior is that it appears to result in a very substantial increase in error.

Despite the fact that catastrophic forgetting has been studied for decades, no precise or consistent definition of the phenomenon could be located. In the remainder of this section, such a definition will be provided. The reader should keep in mind that there may be other ways to defining the problem.

In order to define forgetting, some concept of time should be considered. Otherwise, it is meaningless to refer to prior information or prior training. Time may simply be used to index the current training iteration. However, in a nonstationary setting time is used to index the data that is presently available since the assumption is that data available at time $t$ may not necessarily be the same as data available at time $t + 1$.

To train a model over time, it must be updated such that the parameters are changing over time. Suppose a model with parameters $\theta_t$ is trained at time step $t$. Let $X_t$ denote the data samples and corresponding targets we have available for training at time $t$. Typically, in the stationary setting we have some loss function $L(\theta, X)$ to indicate the error for data $X$ given by some model with parameters $\theta$. Since our model is changing over time we must define a loss function that specifies at what point in time the loss is being measured.

If we are training on a dataset at a given time step $t$, the only loss function that can practically be utilized is one that measures the loss of the data that we have available at time step $t$. Such a loss function can be expressed as $e_t = L(\theta_t, X_t)$. Suppose training is performed until time step $T$ where $T > t$. Since the model is trained online, the parameters are updated at each time step to reduce the loss function at that time step. All losses over time can be represented as a vector which can be defined as $e = (e_0, e_1, e_{...}, e_T)$. Under typical online training, the overall objective function would be to reduce the magnitude of $e$, or to state it more clearly, the following objective function is minimized through training: $J = \|e\|^2$, representing

the squared Euclidean norm of $e$. At each time step $t$, the parameters $\theta_t$ are updated to reduce error for the current samples $X_t$, and not necessarily for past samples.

Forgetting occurs when minimizing the above given objective function does not lead to an optimal set of model parameters at some later time step $T$ where $T < t$. That is, if another loss function is defined to indicate the error at a previous time step given our most recent model parameters $\theta_T$, such a loss function can be expressed as $e'_t = L(\theta_T, X_t)$. The end objective of training is that the most recent model will reduce error for all previous examples. The vector of losses over the past, given the most recent model parameters can be defined as $e' = (e'_0, e'_1, e'_{...}, e'_T)$. The overarching goal of training is to produce a final set of model parameters that reduce error for examples that were previously seen, which can be expressed as the following overall objective function: $J' = \|e'\|^2$. The objective function that is to be minimized through training is $J'$, however minimizing $J'$ assumes our algorithm has access to all previously available training data. In reality, only $J$ can be minimized, and the hope is that through minimizing $J$, $J'$ is also minimized. In a strictly stationary setting in which samples are drawn in an i.i.d. manner, minimizing $J$ will lead to final model parameters in which $J'$ is also minimized. However, due to forgetting effects, in a nonstationary training environment, minimizing $J$ may not necessarily lead to good model parameters. This is the essence of catastrophic forgetting. A model that suffers less from forgetting is one in which minimizing $J$ will also minimize $J'$.

The following definition of forgetting can now be provided. *Forgetting* is the effect where updates to model weights to reduce error at time $T > t$ increases error at time $t$. Forgetting has been observed to occur in a *catastrophic* manner such that the increase in error happens to a significant degree. The model itself may be more than capable of generalizing across past training data with minimal loss if the past data were to be shuffled and presented in a random i.i.d order, however should the data be presented in a nonstationary manner, catastrophic forgetting occurs.

Figure 2.5 illustrates the described forgetting behavior. The solid line labeled $L(\theta_t, X_t)$ indicates a typical training curve for a neural network. In practice, such a

curve may be noisy and may not monotonically decay as shown in the figure. This is particularly true if the data is nonstationary. Suppose training is performed on nonstationary data until some later time $T$ and then the error is measured for the past training data using the latest model given at time $T$. The dashed line labeled $L(\theta_T, X_t)$ indicates such an error should forgetting occur. If the dashed line ever dominates the solid line, as indicated at time step $\tau$, then forgetting has occurred.

One way to induce a nonstationary input sequence is to switch datasets at a particular time step of training. In chapter 3, a mathematical analysis will be provided for catastrophic forgetting with linear networks in the case of training on two datasets labeled P1 and P2. To perform this analysis, it will be assumed that dataset P1 is switched with dataset P2. Under this assumption, several inequalities will be derived which provide bounds on how much the P1 error may increase when training on P2.

In chapter 4, several experiments will be performed which induce nonstationary patterns by switching datasets during training. Loss is measured for the previous dataset while training on the new one. A time series prediction test will also be investigated involving a pendulum model. In the case of the latter, the loss function for previous data will be directly plotted using the latest model parameters allowing explicit measurement of the forgetting phenomenon.

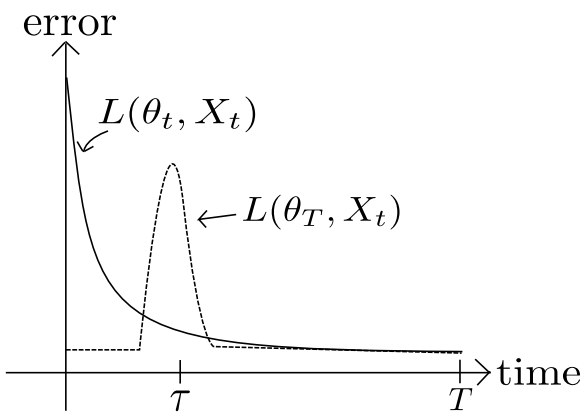



**Figure 2.5:** Illustration of Catastrophic Forgetting

### 2.3.2 Existing Network Architectures

There have been numerous techniques proposed in the literature to address the issue of catastrophic forgetting in neural networks. Prior to the most recent renaissance of neural networks, one can find numerous older ideas and techniques dating back to the 1990s [1]. Catastrophic forgetting became less popular as a research topic during the 2000s; however, more recently with the growing popularity of deep learning architectures, interest in solving catastrophic forgetting has re-emerged.

#### Activation Sharpening

One theory, postulates that catastrophic forgetting is caused by overlapping representations. In canonical neural networks, almost all nodes contribute to every stored pattern. Under this reasoning, an early approach to mitigating catastrophic forgetting was known as activation sharpening [11]. Activation sharpening is a technique that attempts to reduce the distribution of representations in the network by strengthening activations of the largest active subset of hidden layer neurons, while weakening the others. This is achieved by applying an update rule to strengthen a subset of the activations, resulting in a more sparse representation [29].

Activation sharpening works on the hidden layer by selecting the k nodes with the highest activation for "sharpening". If we have a sigmoid activation function that only outputs values in the range $0 < A < 1$, where $A$ is the activation, then the $k$ nodes to be sharpened are updated according to

$$A_{new} = A_{old} + \alpha(1 - A_{old}), \tag{2.15}$$

and the other nodes are updated according to

$$A_{new} = A_{old} - \alpha A_{old}, \tag{2.16}$$

where $A_{old}$ is the original activation of the neuron. The difference in activation is fed back as standard backpropagation error.

### Radial Basis Networks

One technique that works in simpler (i.e. lower dimensional) problem domains is to use radial basis function (RBF) networks [30]. RBF networks are defined as neural networks consisting of one hidden layer where each neuron $j$ in the hidden layer is given a centroid vector (denoted $\vec{c_j}$) and the activation is a nonlinear function that is inversely proportional to the distance from the centroid to the input. For example, if Euclidean distance is used, then the distance is computed as

$$d_j = \|\vec{c_j} - \vec{x}\|^2, \tag{2.17}$$

and a commonly used activation function takes the Gaussian form

$$y_j = e^{-\frac{d_j}{2\sigma^2}}, \tag{2.18}$$

where $y_j$ is the hidden layer activation for neuron $j$ in this case. Typically, centroid locations are chosen at the beginning of training either randomly or via k-means. To feedforward to the output layer, the hidden activations are multiplied by a weight matrix that is trained via standard backpropagation.

RBF networks have local activation functions (zero over most of the input space), and do not suffer as much from catastrophic forgetting. Unfortunately, they have a few disadvantages. The main issue with RBF networks is that one must know where to place the centroids before training can begin. In a truly non-stationary setting, sample data that correctly covers the input space will not be available at the beginning of training. The other issue is that these are not deep networks; RBF networks have one layer and as a result they have trouble generalizing, especially in

high dimensional spaces. They suffer from the curse of dimensionality in that many basis functions are needed to cover higher dimensional spaces [2].

### Fixed Expansion Layer

Another technique that is based on overlapping representations being the root cause, is the Fixed Expansion Layer (FEL) [15]. This scheme works by adding an additional layer of fixed weights known as the FEL layer. When feedforward is performed, a sparse coding technique is used to consistently select only a few neurons in the FEL layer, which has the effect of helping eliminate overlapping representations.

### Maxout Networks And Local Winner-Take-All Networks

One notable technique that has been found to perform well involves types of networks known as local winner-take-all [16] and maxout [31] networks. These techniques add redundant weights to the network whereby only a subset of the neurons are active for every feedforward and backpropagation pass.

In the case of regular feedforward neural networks, the output of all $n$ neurons within a layer can be computed as $\vec{y} = f(W\vec{x})$, where $\vec{x}$ is the input vector, $W$ denotes the weight matrix and $f$ is a nonlinear activation function. Both local winner-take-all and maxout networks group neurons within each layer such that there are $k$ neurons per group. Moreover, both select the neuron within the group that yields the largest output and activate that single neuron and deactivate the rest. That is where the similarity between the two schemes ends. In the case of local winner-take-all, deactivated neurons within a group are forced to output value of 0. In the case of maxout networks, a group contains a single winning output which is assigned to it. The key difference here is in the number of outputs a layer hosts. A local winner-take-all layer with $n$ neurons will have $n$ outputs. However, a maxout layer with $k$ neurons per group and $n$ total neurons will have $n/k$ outputs in total.

Another way of looking at the differences between maxout and local winner-take-all is in terms of the sparsity induced. Local winner-take-all provides a sparse representation in the feedforward pass, since only a single neuron within a group has a non-zero output. Maxout does not provide a sparse representation because the layer has only a single output per group, and all groups may provide non-zero outputs. Maxout does however have a sparse gradient update because only the neuron that contributed to the group output will be updated. For a more in depth description of maxout and local winner-take-all, the reader is referred to [31] and [16], respectively, with a related discussion available in [17].

### Dropout

Dropout [32] is a training technique that was not originally developed for catastrophic forgetting. It was instead introduced as an improved regularization technique to be used in training. Dropout has commonly been used with rectified linear activations, although [17] reported that it helped with catastrophic forgetting when applied to maxout networks.

Dropout is a regularization technique aimed at reducing over fitting. The scheme works by randomly (usually with a probability 0.5 [33]) setting neuron outputs to zero during training, which effectively yields a random selection of neurons to form smaller (sub)network models out of the larger network being trained. An alternative interpretation of dropout is training $2^n$ models that share weights, where $n$ denotes the number of neurons in a given layer.

During the inference phase, the full network is used, which can be viewed as averaging all of the smaller network models. To guarantee a statistical average, the neuron output magnitudes are all multiplied by the probability that they were not set to zero during the training phase (again, usually 0.5).

### Rehearsal Methods

An alternative approach is based on the assumption that catastrophic forgetting should be mitigated by retraining the network on previously observed data, which leads to two similar classes of solutions: rehearsal methods, and pseudo-rehearsal methods. Rehearsal methods are a class of techniques that involve storing previous training data in a memory buffer, and retraining the network on elements stored in this buffer along with any new information that is to be learned [10]. The size of this buffer, strategy for storing prior samples, and how they are presented, leads to a variety solutions that are all similar in that they store and rehearse the training data.

A recent success in the area of deep reinforcement learning is partially attributed to what is essentially a rehearsal method. In [8], a deep network was able to learn to play Atari video games. A key element to the success of this work was in a technique known as "experience replay," which essentially is a buffer of stored network updates that are applied to the network in random order. While rehearsal methods in general seem to work, they have a major drawback in that they require storing of extensive history. Essentially, they rely on creating an explicit memory structure to train the network on prior data, rendering the approach difficult to scale as more memory and training time are required in larger-scale problems.

One way to enhance rehearsal methods, as well as avoid some of the issues associated with them, is through a class of solutions known as pseudo-rehearsal methods [12]. These methods are similar to rehearsal; however, instead of explicitly storing prior information, they work by feeding randomly generated pseudo-patterns into the network and recording the output. These pseudo-patterns with the saved output targets are again fed into the network along with the new data to be trained. While this technique has been shown to be notably effective, it still requires an extra training step.

### Dual Network Models

Another research thrust is based on observations in neuroscience and human memory in particular. The hippocampi region is involved in new memory formation. The evidence for this is partly based on the observation that damage to the hippocampi causes patients to be unable to form new memories. It seems that new memories begin in the hippocampi and are moved to the neocortex [34]. The fact that there are two separate regions, one for memory formation and one for memory storage, inspires a set of techniques known as dual-network models [13]. Under these models, a "learning" network is used to learn new representations, while a "storage" network is used to store the information. Information is transferred from the learning network to the storage network by creating pseudo-patterns of random inputs and recording the corresponding outputs from the learning network. The input/output pairs that come from the learning network are used as inputs and targets on the storage network. Note that this is a simplification of the technique; please see the referenced papers for more details.

### Ensemble of Learners

Under certain conditions, an ensemble of learners can contribute another layer of robustness to mitigating catastrophic forgetting [15]. The main idea is that due to variations within each network, some learners may forget particular regions of the input space faster than others. By combining all networks into an ensemble, the best out of all of them may be achieved. Furthermore, networks with different hyperparameters naturally tend to learn and forget at different rates. Some networks will be very fast to learn new representations, but will also be fast to forget samples that are not being frequently presented. Other networks will require new information to be presented many times before it can be learned, but they will also be slow to forget prior information.

### 2.3.3 Catastrophic Forgetting in Control Problems

The field of reinforcement learning has also encountered catastrophic forgetting, although it wasn't as well recognized as such at first. In the early 1990s, a computer program known as TD-Gammon was developed that combined reinforcement learning with a neural network based value function and had a very notable success when applied to the game of backgammon [35]. TD-Gammon was able to successfully learn moves that outperformed human experts. While that success was very notable and popularized combining neural networks and reinforcement learning, it was short lived. Neural networks seemed poorly suited to solving many other reinforcement learning test cases. A notable work was published in [3] which demonstrated neural networks diverging and failing to perform well on several test cases. Researchers began to realize that neural networks appeared to be unable to perform well when combined with reinforcement learning, at least for many problem cases.

#### Notable Work that Recognized the Phenomenon

Although researchers have noted instabilities when training neural networks as value functions estimators, it wasn't clear that this may have been due to catastrophic forgetting. The work in [36] demonstrated that temporal difference learning with function approximation techniques in general can diverge if some weights are shared across states. Subsequent research abandoned nonlinear function approximation techniques and instead utilized linear function approximation techniques which were better understood and could have stronger convergence guarantees.

Some researchers have recognized that when using nonlinear function approximators, catastrophic forgetting could be involved in the observed instabilities [37]. The work in [6] attempts to mitigate the forgetting problem in a reinforcement learning setting by utilizing radial basis function networks.

The work in [38] also illustrates how forgetting can occur as it attempts to train a learner to follow the trajectory of a dataset generated by an expert player for a

popular video game. In order for the agent to successfully learn to follow a good trajectory, it was necessary for the dataset to include enough examples of recovery from failure states. It wasn't enough that the agent train on the trajectory that was followed by an optimal policy; the training data had to include examples of recovering from failures. These failure examples had to be presented to the network often enough to prevent it from unlearning, which is notable because an optimal policy will never visit failure states often enough to produce samples that can prevent that policy from being unlearned.

### Recent Successes in Reinforcement Learning

A more recent successes in reinforcement learning was noted earlier in section 2.3.2. The work in [8] trained a network dubbed a "deep Q-network" (DQN) which was able to learn to play multiple Atari games. A key element to this success was in the use of a replay buffer, or also called "experience replay." Note that the terms "replay buffer" and "experience replay" may be used interchangeably to refer to the use of a memory buffer in the context of reinforcement learning.

The replay buffer contains samples of states, actions and rewards. The neural network value function approximation technique was trained by sampling randomly from this table and applying the $Q$-learning update rule to produce a set of updates. Results are also provided in [8] demonstrating that a linear function approximation technique performs poorly by comparison, illustrating that neural networks can outperform other techniques. Utilizing a replay buffer is not a new idea [39] [37], however, it was demonstrated to greatly improve stability in DQN.

### Biological Insight

It may possible to gain insights and inspiration by observing biological systems and how they handle catastrophic forgetting, since biological systems appear to be able to effectively mitigate it somewhat. In neuroscience, the problem is known as the

stability vs. plasticity dilemma, or essentially: how do the weights within a biological neural network maintain enough stability to be able to store knowledge for a long period of time, while being plastic enough to learn new information? [40] [41]

Some experiments have demonstrated catastrophic forgetting in biological systems [42], however it seems to only occur in specific situations. In general, artificial systems suffer in a more pronounced manner. Unfortunately, learning mechanisms governing biological systems are not yet well understood, and it is still not possible to probe neurons in biological systems with enough detail to discover exactly what techniques they employ. However, there are interesting connections between what has been observed in biological systems and several techniques that have been found to work in artificial systems.

It has been observed by probing brains of many different types of animals that a type of memory replay appears to occur during sleep [43] [44] [45] [46]. It has been hypothesized that this replay may be connected to learning reward driven behavior [47]. It seems most, if not all biological systems with brains require sleep, yet the reasons for this requirement is still unknown. In evolutionary terms, sleep would have a negative cost associated with it since it requires a creature to be incapacitated for often a lengthy period of time. The prevailing hypothesis is that sleep is involved in memory formation in the brain via some form of offline consolidation process that is related to the observed replay mechanism [48]. It is notable that a replay mechanism appears to be involved in the brain during sleep, especially since there are similarities to the use of a replay buffer which greatly improved stability in the deep $Q$-network.

# Chapter 3

# A Neuron Clustering Approach

The main approach proposed in this dissertation, dubbed "cluster-select" [49] [50] [51], essentially modifies the standard feedforward network model by proposing an unsupervised learning component. This chapter begins by providing motivation for cluster-select followed by theoretical grounding for what causes catastrophic forgetting and how cluster-select can mitigate the effect. Implementation considerations for cluster-select are discussed in detail.

## 3.1 Motivation

The approach presented here draws some inspiration from several current techniques in addition to some new ideas. One motivation that has already been discussed is that catastrophic forgetting is caused by overlapping global representations. The major implication is that any weight update to a neural network to minimize error in one region of the input space may affect completely unrelated regions that are distant from the region being updated. Perhaps this is why pseudo-rehearsal works so well, by saving prior input/output pairs from random locations in the input space, and feeding them through again later, the training algorithm is forcing the network to only allow local updates without affecting distant unrelated regions. Neural networks with sigmoid based activation functions in particular, have a property where unrelated

regions of the input space are affected by weight update phases. Since the sigmoid activation function is nonzero over most of the input space, changes to the input weights to reduce the error for one input will cause the sigmoid based activation to change its output for a wide range of inputs.

Another motivation for this technique centers on the idea of partitioning the weights to create redundant (or unused) capacity within the network. Most network models employ all of the neurons and all of the weights during the feedforward and weight update phases, meaning that any single weight update iteration can potentially modify the entire network. Alternatively, a network model that partitions the neurons, or weights, such that only a subset are used at any specific time tends to more effectively mitigate catastrophic forgetting. Techniques such as local winner-take-all, maxout, and dropout all perform network partitioning. Consider the brain for an example of a system that must also have redundant capacity. If instead every single neuron in the brain was active at all times and contributing to every single new piece of information that was being learned, one could imagine that catastrophic forgetting would occur. By partitioning the input space such that the entire network is not used on every feedforward pass, a feedforward network is provided with extra hidden capacity.

In neuroscience, a balance emerges between stability and plasticity: stability being a network's ability to remain rigid enough to encode long term information, and plasticity denoting a network's ability to be malleable enough to learn new information. Some motivation comes from the stability vs. plasticity dilemma in that we seek to provide artificial neural networks a sense of stabilizing on input examples they have been trained on. In chapter 4, we experiment with a per-neuron learning rate which we decay. A decaying learning rate applied to each neuron provides individual neurons with a critical period in which they must learn, and it seems to greatly reduce the catastrophic forgetting effect in a recurrent setting. While care must be taken not to make unsubstantiated comparisons to neuroscience, it is worthwhile to note that there appears to be a critical period in many animals

and in humans during which certain changes are only allowed to occur in the brain, and particular learning tasks such as language acquisition can only occur effectively during these periods [52]. This critical learning period could perhaps be one way the brain manages to balance the stability vs. plasticity dilemma.

## 3.2 Analysis

Deep linear networks are created by generating multiple layers that are linear (i.e. have a linear activation function), thus performing multiple linear transformations. In practice, a deep linear network may not be any more powerful than a single linear transformation, since a series of linear transformations can be expressed instead as a single linear transformation. However, Saxe et al. [53] advocates building a mathematical theory for deep neural networks by examining the case of deep linear networks. Such linear networks are demonstrated by [53] to have non-linear training dynamics. Since the linear case is easier to understand, it is useful to analyze the case for insights that may also apply to the nonlinear case. In this section, an analysis of catastrophic forgetting is provided for the linear case.

Suppose there is a linear network with a single hidden layer, and weight matrices $W^{(1)}$ for input to hidden weights and $W^{(2)}$ for hidden to output weights. There are two datasets considered, $P1$ and $P2$. Network output error for dataset $P1$ is defined as

$$e_u = \sum_{u \in P1} \|y^u - W^{(2)} W^{(1)} x^u\|. \tag{3.1}$$

Alternatively, if we apply dataset $P1$ to a matrix $X^u$ where each column is a different sample, the network output error can be equivalently defined as

$$e_u = \|Y^u - W^{(2)} W^{(1)} X^u\|_F, \tag{3.2}$$

where $\|\|_F$ denotes Frobenius norm. Minimizing error using gradient descent yields the following update rule

$$\nabla W_u^{(1)} = \lambda W^{(2)\mathrm{T}}(Y^u - W^{(2)}W^{(1)}X^u)X^{u\mathrm{T}} \tag{3.3}$$

$$\nabla W_u^{(2)} = \lambda(Y^u - W^{(2)}W^{(1)}X^u)X^{u\mathrm{T}}W^{(1)T}, \tag{3.4}$$

where $\lambda$ is a small positive step size. The error and update equations for $P2$ are the same as 3.2, 3.3, and 3.4 except $u \in P1$ is replaced by $v \in P2$. The $T$ superscript is used to denote transpose, such that $X^{uT}$, $W^{(2)T}$, and $W^{(1)T}$ are the input and weight matrices transposed.

Assume the network has been trained on $P1$ long enough such that $\nabla W_u^{(1)} \approx 0$, $\nabla W_u^{(2)} \approx 0$, and $e_u \approx \|\epsilon^u\|_F$. That is, the error has reached a minimum $\epsilon^u$ and the weight updates have become small. Now, suppose training is performed on $P2$ for one step and the weights are updated according to $\nabla W_v^{(1)}$ and $\nabla W_v^{(2)}$, yielding the following $P1$ error:

$$\begin{aligned} e_u &= \|Y^u - (W^{(2)} + \nabla W_v^{(2)})(W^{(1)} + \nabla W_v^{(1)})X^u\|_F \\ &= \|Y^u - W^{(2)}W^{(1)}X^u - W^{(2)}\nabla W_v^{(1)}X^u - \nabla W_v^{(2)}W^{(1)}X^u - \nabla W_v^{(2)}\nabla W_v^{(1)}X^u\|_F \\ &= \|-\epsilon^u + W^{(2)}\nabla W_v^{(1)}X^u + \nabla W_v^{(2)}W^{(1)}X^u + \nabla W_v^{(2)}\nabla W_v^{(1)}X^u\|_F \end{aligned} \tag{3.5}$$

Using properties of norms, yields the following upper bound on 3.5

$$\begin{aligned} &\|-\epsilon^u + W^{(2)}\nabla W_v^{(1)}X^u + \nabla W_v^{(2)}W^{(1)}X^u + \nabla W_v^{(2)}\nabla W_v^{(1)}X^u\|_F \leq \\ &\|\epsilon^u\|_F + \|W^{(2)}\nabla W_v^{(1)}X^u\| + \|\nabla W_v^{(2)}W^{(1)}X^u\| + \|\nabla W_v^{(2)}\nabla W_v^{(1)}X^u\|. \end{aligned}$$

Notice that $\epsilon^u$ is the error that we had before, and the other terms describe how the $P1$ error changed after performing a single update for $P2$. Next, we consider each term separately. Expanding terms and using the sub-multiplicative property of the Frobenius norm (i.e. $\|AB\|_F = \|A\|_F\|B\|_F$) to rearrange terms provides the

following:

$$\begin{aligned}\|W^{(2)}\nabla W_v^{(1)}X^u\|_F &= \|W^{(2)}W^{(2)\mathrm{T}}\epsilon^v X^{v\mathrm{T}}X^u\|_F \\ &\leq \|W^{(2)}W^{(2)\mathrm{T}}\|_F\|\epsilon^v\|_F\|X^{v\mathrm{T}}X^u\|_F \\ &= \lambda\|\epsilon^v\|_F\|W^{(2)}W^{(2)\mathrm{T}}\|_F\underline{\|X^{v\mathrm{T}}X^u\|_F} \end{aligned} \tag{3.6}$$

$$\begin{aligned}\|\nabla W_v^{(2)}W^{(1)}X^u\|_F &\leq \|\epsilon^v\|_F\|X^{vT}W^{(1)T}W^{(1)}X^u\|_F \\ &= \lambda\|\epsilon^v\|_F\underline{\|(W^{(1)}X^v)^{\mathrm{T}}W^{(1)}X^u\|_F} \end{aligned} \tag{3.7}$$

$$\begin{aligned}\|\nabla W_v^{(2)}\nabla W_v^{(1)}x^u\|_F &= \lambda\epsilon^v X^{vT}W^{(1)T}\lambda W^{(2)\mathrm{T}}\epsilon^v X^{v\mathrm{T}}X^u \\ &\leq \lambda^2\|\epsilon^v\|_F^2\|(W^{(2)}W^{(1)}X^v)^{\mathrm{T}}\|_F\underline{\|X^{v\mathrm{T}}X^u\|_F} \end{aligned} \tag{3.8}$$

These inequalities indicate interesting theoretical implications as to what causes forgetting. First, note that all of the terms depend on dataset $P2$ error defined as $\epsilon^v$, meaning that if sufficiently lowering $P1$ error also made $P2$ error low, then $P1$ error will not be affected as severely by training on $P2$.

The most important theoretical implication is that the dot products between the samples from the two datasets cause forgetting. The term $X^{v\mathrm{T}}X^u$ produces a matrix whose entries indicate the dot product between samples from datasets $P1$ and $P2$. That is $[X^{v\mathrm{T}}X^u]_{ij} = x_i^v \cdot x_j^u$. The dot products between the two datasets and dot products between the hidden layer representations are involved. Moreover, 3.6 and 3.8 depend on the Frobenius norm of the dot product matrix between the $P1$ and $P2$ datasets defined as $\|X^{v\mathrm{T}}X^u\|_F$. The other term, 3.7, depends on the Frobenius norm of the matrix of dot products of the hidden layer representations between the two datasets defined as $\|(W^{(1)}X^v)^{\mathrm{T}}W^{(1)}X^u\|_F$ (these terms are underlined in the above equations). The implication of this dependency is that if we can make the

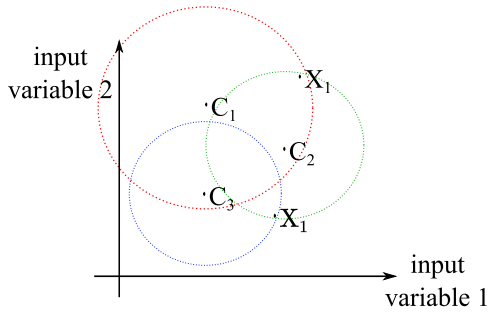
representations more orthogonal in the sense that their dot products are close to zero, then catastrophic forgetting should be reduced.

There appears to be a balance in forgetting, between the dot products of the two datasets and how similar their error surfaces are. To minimize forgetting, the error surface between $P1$ and $P2$ should be related in the sense that minimizing $\epsilon^u$ also minimizes $\epsilon^v$. If this is not the case for some samples, suggesting that $P2$ error is not minimized by minimizing $P1$, then those samples need to be uncorrelated in the sense that their dot product is small. In addition to the samples themselves being orthogonal, their hidden representations must also be orthogonal. If one of the layers yields a representation with a large dot product, that layer will cause the entire network to experience catastrophic forgetting.

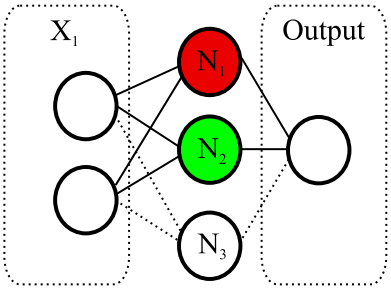
Cluster-select attempts to make the hidden representation less distributed, and more localized as well as reduce the dot products between the datasets. As described previously, this is accomplished by forcing activations to zero if they are distant in the input space. In terms of orthogonality between samples, forcing some activations to zero will reduce the magnitude of the dot product. If the sample is distant enough in the input space that it has an entirely different set of nonzero activations, then the corresponding dot products between the activations will be zero.

## 3.3 The Cluster-Select Approach

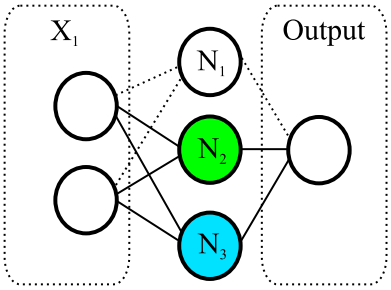
The general framework of cluster-select involves assigning each neuron a cluster in addition to its regular weights. When an input is observed, $k$ out of $n$ neurons which have the nearest centroids are selected. A sub-network is built out of the $k$ neurons such that only those neurons are used in the feedforward process. This selection process has the effect of partitioning the input space such that different regions are assigned to different neurons. Overlap is minimized via the inherent segmentation of the representations. Moreover, the approach supports redundancy in the network, such that not all neurons are active at the same time.




(a) Illustration of three centroids ($C_1$, $C_2$, and $C_3$) and two sample points ($X_1$ and $X_2$)




(b) Neurons $N_1$ and $N_2$ are selected when $X_1$ is the input.




(c) Neurons $N_2$ and $N_3$ are selected when $X_2$ is the input.

**Figure 3.1:** Illustration of the cluster-select process

Figure 1 provides a coarse illustration of the proposed technique. Assuming a network that takes two-dimensional input data, with two samples labeled $X_1$ and $X_2$ and three hidden neurons in networks labeled $N_1$ $N_2$, and $N_3$. These neurons each contain centroids labeled $C_1$, $C_2$ and $C_3$. In Figure 3.1(a) the two sample points are shown along with 3 centroids plotted as a function of the input space. If two out of three nearest centroids are selected, then Figure 3.1(b) depicts what occurs when $X_1$ is propagated through the network. In this case, since centroids $C_1$ and $C_2$ are nearest to $X_1$, they are selected and a sub-network consisting of two hidden neurons ($N_1$ and $N_2$) is invoked. However, should $X_2$ be propagated through the network, centroids $C_2$ and $C_3$ are selected given that they are nearest to $X_2$ and a sub-network consisting of $N_2$ and $N_3$ is used, as illustrated in Figure 3.1(c)

Suppose neuron $j$ has centroid vector $\vec{c_j}$ in addition to weight vector $\vec{w_j}$. During the feedforward phase, a distance between the layer's input $\vec{x}$ and each neuron's

centroid vector $\vec{c_j}$ is computed. We consider a vector of distances $\vec{d}$ where each neuron's distance $d_j$ is computed as

$$d_j = ||\vec{x} - \vec{c_j}||^2 \tag{3.9}$$

The squared Euclidean distance is utilized as the distance measure. Neuron selection is achieved by selecting the elements of $\vec{d}$ that correspond to the $k$ smallest $d_j$ distance values. These neurons are allowed to have a nonzero activation which is computed using the standard formulation for feedforward networks, $a_j = f(\vec{x} \cdot \vec{w})$ where $f(\cdot)$ is the activation function used. The neurons that are not the $k$ nearest have their activation values forced to zero. During the backpropagation phase, the gradient is only propagated through neurons with non-zero activation.

### 3.3.1 Feedforward Implementation Details

In the case of regular multilayer perceptrons, each neuron is associated with a weight vector. In general, all of the weight vectors of the neurons are combined (as column vectors) to yield a weight matrix $W$. More specifically, the elements of W are $w_{ji}$, where $i$ is an index into the previous layer and $j$ is an index into the current layer. $W^{(1)}$ refers to the weights of the first hidden layer, $W^{(2)}$ refers to the weights of the next layer. (To simplify this discussion only networks with one hidden layer will be addressed, since it is straightforward to generalize to multiple layers.)

During each feedforward one typically passes a group of samples through each layer as a mini-batch. If the matrix of samples for a mini-batch is defined as $X$, where each sample is a column vector in this matrix, then the output of the hidden layer can be computed by performing $Y \leftarrow f(W^{(1)}X)$, where $f(\cdot)$ is the hidden activation function. This will produce $Y$ which will be a matrix of column vectors in which each vector represents the hidden layer activations for a given sample. The final network output can be computed as $Z \leftarrow g(W^{(2)}X)$, where $g$ is the output activation function, if it exists.

As an example for how to mask out neurons, dropout is applied in a feedforward pass using the above formulation. To formulate dropout, an $R$ matrix can be defined consisting of random values that take on 0 or 1 with probability 0.5. To apply feedforward with dropout, this mask matrix can be used to set activations to zero by performing $Y \leftarrow f(R \circ (W^{(1)}X))$ where $\circ$ denotes the Hadamard product representing element-wise multiplication.

Cluster-select assigns a centroid to each neuron. One way to formulate this is to define a centroid matrix $\mathcal{C}$. This matrix will have the same dimensions as the weight matrix $W$, where every neuron is allocated a centroid as a column vector in this matrix. This matrix will be used to select neurons whose centroids are nearest to the sample point. The first step in performing this is to compute the distances. The matrix $Y^d$ is employed to indicate the distance of each neurons to each sample point. $Y^d$ is defined as $\mathbb{R}^{h \times k}$, where $h$ is the number of hidden neurons and $k$ is the number of samples, essentially it is the same size as $Y$. It is similar to the neuron output matrix $Y$, with the exception that it stores distances instead of neuron outputs. To understand how to obtain the elements of $Y^d$ it is useful to understand how to compute the elements of $Y$.

$$Y \leftarrow f(W^{(1)}X) \tag{3.10}$$

$$y_{jl} \leftarrow f(w_{j1}x_{1l} + w_{j2}x_{2l} + \cdots + w_{jn}x_{nl}) \tag{3.11}$$

This is simply the definition of matrix multiplication. Note the order in which the elements in the matrix are indexed. Since $Y^d$ has the same dimensions as $Y$, the centroid distances will need to be computed by indexing elements in the same order. In this case, Euclidean distances will be computed instead of a simple multiplication. The elements of $Y^d$ are obtained by finding centroid distances such that each element of $Y^d$ produces the following equation:

$$y^d_{jl} \leftarrow (c_{j1} - x_{1l})^2 + (c_{j2} - x_{2l})^2 + \cdots + (c_{jn} - x_{nl})^2 \tag{3.12}$$

Expanding terms gives:

$$y_{jl}^d \leftarrow c_{j1}^2 - 2c_{j1}x_{1l} + x_{1l}^2 + c_{j2}^2 - 2c_{j2}x_{2l} + x_{2l}^2 + \dots \\ + c_{jn}^2 - 2c_{jn}x_{nl} + x_{nl}^2 \tag{3.13}$$

Terms can be rearranged to get:

$$y_{jl}^d \leftarrow c_{j1}^2 + c_{j2}^2 + \cdots + c_{jn}^2 \tag{3.14}$$

$$-2[c_{j1}x_{1l} + c_{j2}x_{2l} + \cdots + c_{jn}x_{nl}] \tag{3.15}$$

$$+x_{1l}^2 + x_{2l}^2 + \cdots + x_{nl}^2 \tag{3.16}$$

Each of the terms 3.14, 3.15 and 3.16 can be computed. Namely, 3.14 is a column vector that consists of $\sum_i c_{ji}^2$, 3.15 is $2\mathcal{C}X$, and 3.16 is a row vector that consists of $\sum_i x_{il}^2$. Most scientific computing environments, including Numpy [54] and Matlab [55], can compute each term in an optimized manner, allowing for a very efficient implementation of this scheme.

Once $Y^d$ has been computed, it is used to deactivate neurons that are at a great enough distance. That is, if a neuron's distance is beyond some threshold for a particular sample, its activation is set to 0. That neuron is not updated in the backpropagation phase since it was unused. Selective feedforward and weight updating is achieved by generating a masking matrix from $Y^d$ defined as $M$ consisting of the binary values 0 and 1. $M$ has the same dimensions as both $W$ and $\mathcal{C}$ and is used to deactivate neurons. If $y_{jl}^d$ exceeds the threshold, then that neuron's centroid is at a great enough distance such that the corresponding $m_{jl}^d$ element in the mask matrix is set to 0.

$$m_{jl} \leftarrow \begin{cases} 1 & \text{if } y_{jl}^d < y_l^{d(k)} \\ 0 & \text{else} \end{cases} \tag{3.17}$$

Selecting $k$ nearest neurons for each sample requires performing a column-wise sort operation on $Y^d$ and taking the $k$'th row and using it as a threshold. In the notation above $y_l^{d,(k)}$ represents the $k$'th order statistic ($k$'th smallest value) on column $l$'th of $Y^d$. During feedforward the $M$ matrix is used to mask out neurons whose centroids are sufficiently distant from the samples by performing $Y \leftarrow f(M \circ (W^{(1)}X))$. This is similar to how the mask matrix $R$ was utilized earlier in dropout. The matrix $M$ is also used in the backpropagation phase to ensure that weights are not updated for neurons that were not selected.

Note that the process outlined above was written in the context of training mini-batches (i.e. groups of samples). It can easily also be applied to the case where a single sample is propagated through the network whereby the $X$ matrix of inputs is a single column vector. Several other matrices also reduce to vector form, which does not affect applicability. A good implementation should apply to both cases of mini-batches or single vectors of inputs.

While the details presented thus far are the same for the case of feedforward and recurrent networks, there are nonetheless differences in how the centroids are placed, and the training regime used in each. These differences depends on the training scenario. The following chapters consider applying cluster-select to different scenarios. The next chapter considers classification and regression tasks, and chapter 5 applies cluster-select in the more difficult reinforcement learning setting.

### 3.3.2 Covariance Estimation

Cluster-select can be extended to utilize Mahalanobis distance instead of Euclidean distance to associate inputs with neurons. The use of Mahalanobis distance allows the centroids to take covariance into consideration when covering the input space.

Covariance estimation is a well-studied problem [56, 57]. In environments where data arrives sequentially, the covariance matrix is required to be updated sequentially [58]. Some models simplify covariance estimation by limiting to a diagonal matrix

[59, 60]. Moreover, when the number of observations $n$ is comparable to the number of variables $p$ the covariance estimation problem becomes more challenging. In such scenarios, the sample covariance matrix is not well-conditioned nor is it necessarily invertible (despite the fact that those two properties are required for most applications). When $n \leq p$, the inversion cannot be computed at all [61, Sec. 2.2]. In such case, a desirable estimator would outperform the sample covariance matrix, both in finite samples and asymptotically.

The next chapter contains two experiments that utilize a sequential covariance estimator. For details of the covariance estimator which we used, the reader is referred to [62].

# Chapter 4

# Mitigating Catastrophic Forgetting in Classifier and Regression Problems

This chapter explores applying cluster-select to several feedforward and recurrent training scenarios. We begin by modifying the most common mini-batch feedforward training algorithm to incorporate cluster-select. Next we describe how to apply cluster-select in training a recurrent dataset. Finally, catastrophic forgetting experiments are performed on multiple classification and regression datasets. It should be noted that many of the test cases in this chapter have been artificially designed such that they are non-stationary. The next chapter will tackle the more difficult case of reinforcement learning in which the problem itself is non-stationary.

## 4.1 Feedforward with Cluster-Select

The previous chapter described how neurons are selected during the feedforward and backpropagation phases. What has not yet been discussed is how the centroids are placed or moved during training. In this section, we describe how the feedforward

variation of cluster select modifies the standard training algorithm [2] for training a neural network classifier via mini-batch updates. In particular, we describe details such as when and how to place centroids. Figure 4.1 provides a technical description of our training algorithm. Note that the recurrent variation differs from this and will be described in section 4.2.1.

Centroids should ideally be placed in a way that minimizes overlap between different regimes in the input space. One approach to address overlap is to have a network which detects regime changes and places new centroids to cover novel inputs. Network error is one good indicator of novel inputs. If the network has sufficiently learned one regime, then a novel presentation from a different regime should produce a notable error on the network's output/s.

The network detects the error by maintaining a moving average over recent output errors. Let $e_l$ denote the mean squared error in the current mini-batch for all samples belonging to class label $l$, then a moving average error $\epsilon_l$ can be updated according to

$$\epsilon_l^{new} \leftarrow (1-\alpha)e_l + \alpha\epsilon_l, \tag{4.1}$$

where alpha is some constant close to but less than 1.0.

The neural network should be viewed as a black box that can reconfigure itself to process non-stationary data and internally allocate resources as needed. As a consequence of this, the network must internally detect the change in regime. A moving average error provides the network with a data-driven method to internally detect regime changes.

Eligibility is a concept borrowed from reinforcement learning [63], which provides a way to track recent usage of neurons. Once placement of a new centroid is triggered, the neurons that have the lowest eligibility will have their centroids overwritten by the new ones. During each feedforward pass, should a neuron be selected, its eligibility is additively increased by 1. All neurons have their eligibility decayed with time by a

constant factor regardless of whether they are selected. Eligibility's main purpose is to keep track of which neurons are not being frequently used.

Centroids are placed simply by taking samples belonging to the current mini-batch and overwriting the centroid location of neurons that have the lowest eligibility. Once the new centroids are placed, the moving average error $\epsilon_l$ is reset to $e_l$ to prevent the $\epsilon_l > ke_l$ criteria for placing more centroids from triggering again immediately.

During training centroids are also moved closer to samples for which they were the winners (i.e. closest). If a neuron was selected by its input, then its centroid is moved by some small constant to the location of the sample that selected it.

$$\vec{c}_j^{new} \leftarrow \vec{c} + \beta(\vec{x} - \vec{c}) \tag{4.2}$$

This works to slowly shift the centroids toward samples that selected them. $\beta$ is a small constant close to 0 and is included as a hyperparameter.

When training begins, the centroids are initialized to a value that is distant from all sample points (all elements are set to $-10$). The moving average error estimate is initialized such that it will trigger placement of new centroids on the very first epoch.

In section 4.3.2, we also explore adding an ensemble of learners. As previously discussed, an ensemble can aid in mitigating catastrophic forgetting due to variations within each network within the ensemble. To promote variations within each network, we explore lowering the learning rate for some of the networks in the ensemble after detecting the regime change. This simple technique greatly improved results in the tests with the ensemble of learners. Lowering the learning rate may allow some networks to become slow learners, but more stable at retaining prior representations. The networks that are faster learners will be more plastic in the sense that they learn the new representations. The ensemble of learners as a whole will utilize both types of networks to have both desirable properties of stability and plasticity.

**Algorithm 2:** Cluster-Select Training

Initialize all centroids to some distant location.
Initialize $\epsilon_l \; \forall \, l$ to a very small value.
**for** *each mini-batch of training* **do** *training loop*
    feedforward mini-batch using centroid selection.
    Compute error.
    **for** *each class label* **do**
        Compute class label error as $e_l$.
        **if** $\epsilon_l > te_l$ **then**
            Choose centroids from samples that caused error increase. Place the new centroids on least eligible neurons.
            $\epsilon_l \leftarrow e_l$ (reset moving average error to prevent re-triggering)
        **end**
        $\epsilon_l \leftarrow (1 - \alpha)e_l + \alpha\epsilon_l$
    **end**
    $\vec{c}_j^{new} \leftarrow \vec{c} + \beta(\vec{x} - \vec{c})$
    Backpropagate error and update weights.
**end**

**Figure 4.1:** Training algorithm used for mini-batch training with cluster-select

## 4.2 Recurrent Network

Online non-stationary tasks are the essence of where catastrophic forgetting has perhaps the most impact. In this section, we provide motivation for designing an online task and why it is an important test case, then the details for training a recurrent neural network (RNN) with cluster-select follow.

### 4.2.1 Online Non-stationary Task

Online non-stationary training scenarios are common in cases such as reinforcement learning which will be covered in the next chapter. In this chapter, we choose to evaluate a simple time series dataset that exhibits an abrupt change in its behavior. The goal of this work is to design a simple test case that mimics some of the difficulties encountered in real world situations, particularly a non-stationary situation in which the data is received online.

In a true online scenario, a RNN can only be trained on data as it is being received. While there are methods to present data at a different time, namely, rehearsal techniques, which store data in a buffer and replay it to the RNN later. Such techniques can be considered an alternative method of mitigating catastrophic forgetting, and as mentioned previously, they suffer from scalability issues in that they require explicit storage for past training data. Instead, we choose to rely on the RNN architecture to mitigate catastrophic forgetting as we feed the data in an online fashion.

While the dataset used here has been pre-generated such that all of the samples are available offline, we choose to simulate an online environment by restricting training such that samples from both regimes are presented to the network only once. Although the time series prediction problem is by itself fairly simple to learn, this online training setting causes catastrophic forgetting to occur with standard network models [28].

### 4.2.2 Recurrent Training Details

For the recurrent variation of cluster-select, an Elman [22] network is used to perform time series prediction. The input consists of the state of the system at the current time step combined with the hidden network output at the previous time step. The network should be able to learn to infer the future state of the system by using its internal feedback from the previous time step. Unlike the training technique used for a feedforward network, mini-batches can no longer be used in an online training setting, unless some form of rehearsal method is used to save past examples.

Another difference when applying cluster-select to the time series dataset is that no centroids were placed or moved during training. Instead, centroids were initialized randomly with each centroid selected from a Gaussian distribution with mean and variances similar to that of the data that is trained on. Each centroid provided a fixed region in the input space where a particular neuron would be activated. A per-neuron

learning rate was maintained and decayed for each neuron that was selected. Let $\vec{l}$ be a vector of learning rates for neurons in the hidden layer, where $l_j$ is the learning rate of neuron $j$ in the hidden layer. If neuron j is one of the $k$ out of $n$ neurons that were selected then its learning rate is decayed by $\zeta$

$$l_j^{new} \leftarrow \zeta l_j \tag{4.3}$$

Decaying the learning rate has the effect of causing individual neurons to eventually stop learning, meaning that as the network learns a region of the input space, training slows until the network no longer has capacity to represent new information for that particular region.

In theory, when novel data is encountered by the network, it will belong to a new region of the input space which will select at least some new neurons. Newly selected neurons will have an initially high learning rate, and can thus learn and compensate for the neurons that have stopped learning, without interfering.

The decay rate $\zeta$ must be chosen very carefully for the task. In some sense it can be seen as a parameter that balances stability and plasticity. It should be near, but slightly less than 1.0. If it is too close to 1.0 then the network will effectively have no decay and forget prior representations. If the decay rate is too small then training will diminish too early before sufficient learning has taken place.

All tests had multiple hyperparameters that had to be chosen carefully. A software package known as hyperopt [64] was used to perform automatic hyperparameter optimization. This was advantageous for several reasons. First, it allowed better objectivity in comparing results since having a human manually tweak settings can lead to biases if the human researcher does not invest a sufficient effort into tweaking parameters to find the optimal set for each technique. Secondly, it saves the extra work of having to tweak extra settings which can be time consuming.

## 4.3 Simulation Results and Analysis

In order to explore different aspects of the proposed approach, experiments were performed on multiple datasets under different conditions. There were a variety of tasks performed, including several feedforward classification tasks as well as an autoregression task. First, experiments were ran on a test case with the MNIST handwritten digit dataset. MNIST is a popular dataset used for classification tasks in machine learning [65]. Secondly, a test case was considered with added noise to MNIST. Adding noise allowed cluster-select to perform even better compared to the other techniques. Next, the 20 newsgroups dataset [66] was tested, which another popular dataset used in machine learning. An artificial dataset consisting of random binary patterns was also considered. Experiments were performed with online covariance estimation to utilize Mahalanobis distance with cluster-select using a dimensionality reduced MNIST dataset, and a gas dataset with added noise. To apply cluster-select to a RNN, a test case was constructed using motion from an ideal pendulum to create a non-stationary regression task. This section details all of these experiments.

In order to simulate a dynamic environment, all feedforward tests partitioned the datasets into two parts, $P1$ and $P2$. The general training approach for the nonstationary classification tests where dataset $P1$ was switched with $P2$ was derived from [17] and [16]. Once training on $P1$ was complete, training was switched to dataset $P2$ and performance was measured on both $P1$ and $P2$ while the network continued to learn $P2$. The objective function which hyperopt optimized over was $min(P1_{error} + P2_{error}) \quad \forall \quad epochs$ or the minimum error reached for $P1 + P2$ at some point in training. If this error is 0, it would mean that at some point in training the network was able to learn both $P1$ and $P2$ with no error.

Initially, we had included learning rate as a hyperparameter, however we noticed that results were often difficult to compare since it affects the speed of convergence of the network. We ended up with some networks that had very high learning rates

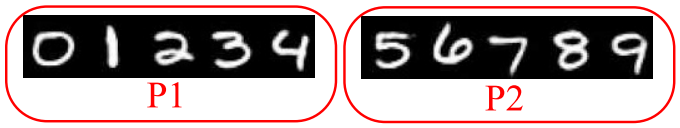



**Figure 4.2:** How MNIST was split into $P1$ and $P2$

and were very unpredictable. Although they often did well to minimize the hyperopt objective function, when plotted, results were sometimes very noisy and converged overall to much worse final solutions. This rendered the results difficult to interpret. It was decided that using a constant learning rate across all simulations made the results much easier to compare. All of the networks for the non-recurrent tasks had 2 hidden layers and an output layer. In the recurrent simple pendulum test case, only a single hidden layer was used in an Elman network.

Finally, an important parameter for the classification tasks was the threshold $t$ at which to wait for error to increase before detecting the $P1$ to $P2$ change. It turns out (from the tests performed on these datasets) that this parameter has a very large range that effectively triggers the change when necessary. The range for this threshold parameter $t$ was measured by estimating how high and how low it could be while still triggering the placement of new centroids at both the beginning of training, and when the $P1$ to $P2$ switch actually occurred. This parameter was simply left at a known good value and was not included in the hyperparameter search. This parameter also depends on the moving average rate, $\alpha$ in the previous section, which was left at 0.95. As a result, these were not actually hyperparameters included in the search. For classification with cluster-select, the output layer had a hyperbolic tangent activation with "-1" being assigned to the target for the incorrect class and "+1" being assigned to the target for the correct class.

### 4.3.1 MNIST Experiment

MNIST [65] is a popular dataset used for classification tasks in machine learning, which consists of 60,000 greyscale images, each 28 by 28 pixels, and is divided into

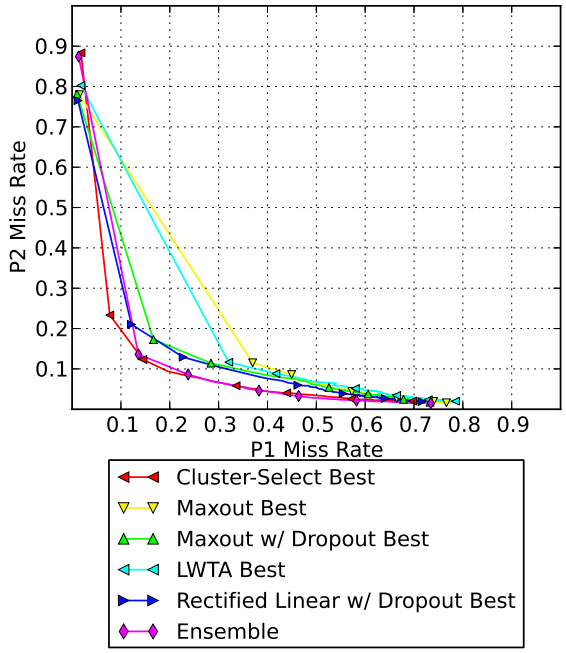



**Figure 4.3:** P1 miss rate vs. P2 miss rate possibilities frontiers for MNIST forgetting task

50,000 training images and 10,000 test images. Image pixel values were normalized to be between 0.0 (solid black) and 1.0 (solid white).

### Test Setup

In order to test catastrophic forgetting, the task was made non-stationary by dividing the dataset into two parts called $P1$ and $P2$ (as illustrated in Figure 4.2). For $P1$, only images of digits 0-4 were used; for $P2$ images of digits 5-9 were used, meaning the network would classify which of the 5 digits the input image belonged to. For $P1$ this would classify as digits 0-4. After training was switched to $P2$ the 5 class labels would switch to digits 5-9 such that digit "0" from $P1$ would change labels to digit "5" from $P2$ and digit "1" would change to digit "6", and so on. The network was first trained on $P1$ until the test error rate did not increase after 100 consecutive epochs. Afterwards training was switched to $P2$. As the network learned the $P2$ training set, both $P1$ and $P2$ test rates were observed.

For this test 40 runs were performed using each technique, (local winner-take-all, local winner-take-all with dropout, maxout, maxout with dropout, cluster-select). Each run was given a unique set of hyperparameters using hyperopt. The objective function hyperopt minimized was $min(P1_{error} + P2_{error}) \quad \forall \quad epochs$ That is, the minimum summation of both $P1$ and $P2$ test error for all of the epochs of training. The set of hyperparameters that produced the best result according to this criteria was considered the winner. The winning run for each technique is the one considered for plotting. The learning rate was set to a fixed value of 0.005 for all runs. The networks all had two hidden layers and an output layer consisting of 5 outputs to predict which of the 5 digits the input image belonged to.

The cluster-select technique required 5 total hyperparameters. There was a hyperparameter for the number of centroids selected on each feedforward pass (what $k$ should be when selecting $k$ out of $n$), the total number of neurons, how many centroids to replace when the $P1$ to $P2$ change was detected, and also the speed to move each centroid ($\beta$ in the previous section). In addition, it wasn't certain if clustering should occur only on (1) the first hidden layer, (2) the second, or (3) both, therefore a discrete hyperparameter was included to select one of the 3 options.

The maxout and local winner-take-all runs had only two hyperparameters. The first being the number of nodes per group, which indicated how many neurons are in each group and was varied from 2 to 128. The other parameter was the number of hidden neurons which was varied from 1024 to 2048. Softmax was used as the output layer since this is commonly used for networks with these types of activations. Results are also included that combine these techniques with dropout (using a dropout rate of 0.5).

For the ensemble of learners, only a single run was performed by taking the 5 best performing networks and further varying their hyperparameters slightly to produce 16 networks. Having different hyperparameters is a way to help diversify the network predictions and behavior relative to each other. This 16 network ensemble was evaluated by producing a linear combination of the outputs of all 16 networks.

When a network detected the $P1$ to $P2$ change, it had a 0.5 probability of also lowering its learning rate. This meant that roughly half of the networks had their learning rates lowered when the change was detected. Learning rate was lowered by 50% when the change was detected.

#### Simulation Results

Figure 4.3 shows the results for this test. As the network forgets $P1$ it learns $P2$ producing a curve if both errors are plotted together. This technique of plotting the impact of catastrophic forgetting was first introduced in [17] and illustrates the error on $P2$ relative to $P1$. The closer the curve gets to the bottom left hand corner, the better the network was able to capture both $P1$ and $P2$ at some point during training. How close the curve gets to the bottom left corner corresponds directly to the loss function used for hyperopt. Cluster-select performs the best on this task. Note that the ensemble here was made of the 5 best cluster-select networks. It seems to give nearly identical performance.

### 4.3.2 MNIST Experiment with Noise

#### Test Setup

This test is similar to the last, except white Gaussian noise with zero mean and variance 2.0 was added to the normalized images, making the task a bit more challenging. In addition, the ensemble was organized a bit differently, and a technique of further diversifying the ensemble of learners was attempted where some of the networks in the ensemble had their learning rate lowered.

An attempt was made to improve ensemble results, hence the ensemble was constructed with 16 networks instead of 5 as in the previous test case. To produce the parameters for these 16 networks, 5 best performing networks were taken and their hyperparameters were further varied slightly to produce 16 networks. Having different hyperparameters is a way to help diversify the network predictions and behavior

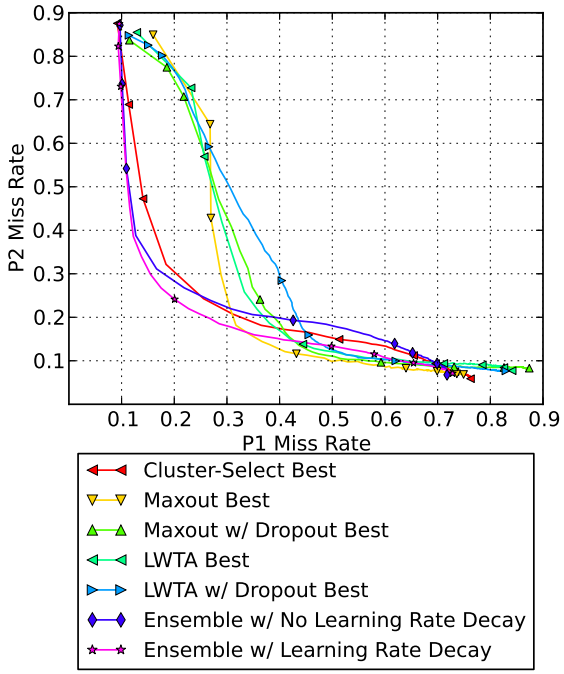



**Figure 4.4:** Results for MNIST Test with noise

relative to each other. This 16 network ensemble was evaluated by producing a linear combination of the outputs of all 16 networks. When a network detected the $P1$ to $P2$ change, it had a 0.5 probability of also lowering its learning rate. This meant that roughly half of the networks had their learning rates lowered when the change was detected. Learning rate was lowered by 50% when the change was detected.

### Simulation Results

The cluster-select technique performs well in this test case, with the ensemble of 16 cluster-select networks performing the best. To determine if lowering the learning rate helped, an ensemble that does not lower the learning rate after detecting the $P1$ to $P2$ change is also shown. This ensemble is otherwise identical to the one that does lower the rate. Comparing both ensembles demonstrates that lowering the learning rate turned out to be the key in gaining extra performance out of the ensemble.

It should be noted that while the ensemble of 16 networks produced the best results, it takes nearly 16 times more CPU resources to run. While parallelizing

an ensemble of networks is possible, the implementation may not be trivial. The matrix operations for feedforward are already parallelized within the network, which means that each network within the ensemble must run on individual machines or CPUs. For the test case presented here, the code was not parallelized this way, thus performing the single run required several days. Because of the computational overhead, we were unable to run many experiments with the hyperparameters for the ensemble. As a result, the hyperparameter of 0.5 probability of lowering the learning rate for a particular network and the hyperparameter of lowering it by 50% may not be optimal, since only a few combinations were attempted. While an ensemble can help significantly as shown here, it may or may not be worth the investment. Regardless, the single cluster-select network still significantly outperformed the other known techniques.

Adding noise to the test also caused further improvements of our technique in relation to the other techniques. Comparing to the previous test without noise, it appears that the noise adversely affects local winner-take-all and maxout networks more so than cluster-select. The ensemble improvements are also more notable when noise is added. At least in this test case, our technique appears to be significantly more effective in noisy environments.

### 4.3.3 20 Newsgroups Experiment

#### Test Setup

The 20 newsgroups dataset [66] is a text classification dataset consisting of 18,837 posts to 20 Usenet newsgroups. The task is to determine which of the 20 newsgroups the post was submitted to, based on the contents of the post. Before this could be fed to a neural network, some technique of feature extraction had to be performed. We chose to use the TF-IDF method to extract 2000 features [67]. This dataset has a problem of fitting only to 2 or 3 features if they are included; so we chose to remove the headers, footers, and quotations block as recommended by the scikit-learn

documentation when dealing with this dataset [68]. Removing these features makes this a much more challenging problem. We decided to reshuffle the training and testing data such that we randomly chose 2048 feature vectors for testing, and 16,789 for training.

This set had 20 total classes, and was divided into two segments, $P1$ and $P2$, each consisting of 10 class labels. When no improvement was observed for 300 epochs, training was switched from $P1$ to $P2$ such that a new set of classes would map to the old labels. Error rates were observed on the $P1$ and $P2$ test sets during the switch from P1 to P2 and plotted in figure 4.5.

It is possible to achieve a better training accuracy by including many more than 2000 features, however this would greatly slow down training, and we wanted this test to measure forgetting, not necessarily to achieve the highest accuracy on this data set. For this test we fixed the learning rate to .025.

#### Simulation Results

Figure 4.5 shows the results. A total of 75 experiments were ran for each type of network with random hyperparameter sampling. All techniques were able to achieve an accuracy of around 30% on $P1$. In the context of learning both P1 and P2, cluster-select was able to perform the best. The ensemble of 5 cluster-select networks performed only slightly better than a single network.

### 4.3.4 Autoassociative Encoder Experiment

#### Test Setup

In order to evaluate this technique on a regression task, an artificial dataset was created containing 200 random binary patterns. Each binary pattern contains 100 random binary inputs and 100 random binary outputs. The task is to train a neural network to associate a given 100 bit input to an unrelated 100 bit output. All networks

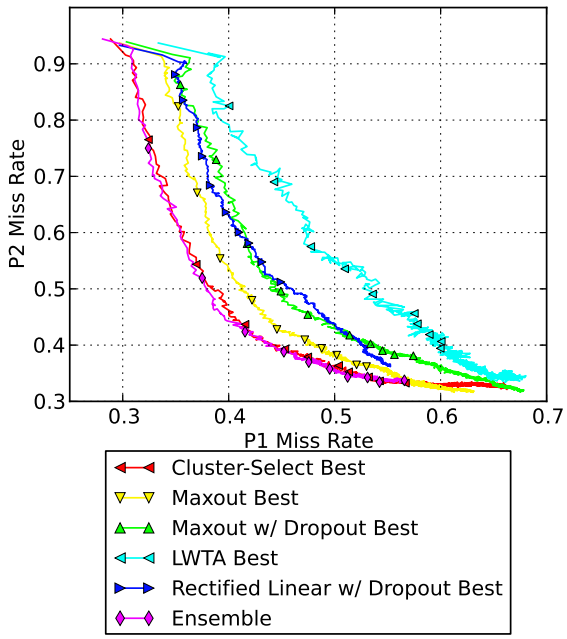



**Figure 4.5:** P1 miss rate vs. P2 miss rate possibilities frontiers for 20 newsgroups dataset forgetting task

in this test had linear outputs since this was a regression problem, and the binary targets were either $-1$ or $+1$.

The dataset was divided into two parts each containing 100 patterns. As before, training was performed on the first dataset $P1$, then when the error had not decreased for 300 epochs, training was switched to the second dataset $P2$. Error rate is measured in the percentage of total bits the network got incorrect. A bit was considered correct if the sign of the network output matched the sign of the bit (i.e. if the network output is less than 0 the bit must be $-1$ to be considered correct, else if the network output is greater than 0 the bit must be 1 to be correct). Randomly guessing would produce an error rate of about 50%, hence the graph scale in figure 4.6 goes to 50%. The dataset was small enough that we did not need to divide the dataset up into mini-batches, instead we trained on the entire batch during each epoch.

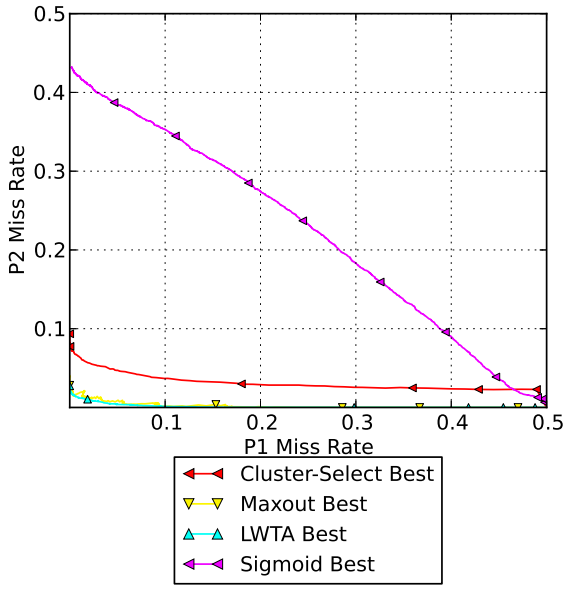



**Figure 4.6:** $P1$ Miss Rate vs. $P2$ Miss Rate Possibilities Frontiers for Autoassociative Encoder Forgetting Task

#### Simulation Results

For this test 100 experiments were performed per network type. As illustrated in Figure 4.6, this test actually shows local winner-take-all and maxout ahead of cluster-select. A network with sigmoid hidden layers was also included for comparison. This test shows that all three techniques (maxout, local winner-take-all, and cluster-select) perform well on this regression test. We believe cluster-select did not achieve the performance it could have because there was still some overlap between the clusters that were selected. Some centroids for $P1$ were likely selected when training on $P2$. Considering that each of the 100 vectors in P2 had to have $k$ centroids selected ($k$ being the number of centroids selected, a hyperparameter), at least some of those inputs likely selected centroids that were allocated for $P1$.

### 4.3.5 Reduced MNIST Experiment

In theory, Mahalanobis distance can provide a richer metric compared to the Euclidean distance since it considers the spread of samples around a centroid. For

the following two test cases, we wanted to evaluate how incorporating covariance estimation into the cluster-select process could help improve performance. In the other experiments, Euclidean distance was used.

Mahalanobis distance requires extra computational power compared to Euclidean distance. However, These computations can be performed in an accelerated manner using parallel hardware such as GPUs. The results here are preliminary and are intended to shed light on whether including the covariance matrix can improve performance. We did not attempt to build a highly efficient parallel implementation of Mahalanobis distance, and as a result we had to limit the number of dimensions and size of the datasets in these test cases.

### Test Setup

For the first test with covariance, the MNIST handwritten digit dataset was used. An autoencoder [69] was constructed consisting of a hidden layer with 50 activations. MNIST was reduced to 50 dimensions using this autoencoder. The dataset was then divided into two subsets. As before, samples that had class labels for digits 0 through 4 were placed in subset $P1$, and samples for which the class was 5 through 9 were placed in subset $P2$. A network was trained on $P1$ to predict which of the 5 classes the sample belonged. Once training on $P1$ was complete (no improvement on test error was observed for 30 epochs), the dataset was switched to $P2$. After switching to $P2$, both $P1$ and $P2$ error were observed for forgetting.

### Simulation Results

Figure 4.7 illustrates a definite improvement of using Mahalanobis distance over Euclidean because it shows that the network was able to capture more of $P1$ without misclassifying $P2$. Forgetting curves for maxout and LWTA networks are also shown. The results depicted in Figure 4.7 demonstrate that cluster-select using Euclidean

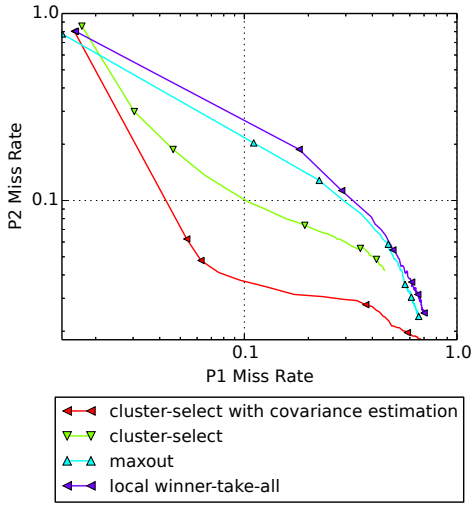



**Figure 4.7:** Cluster-Select MNIST Result

distance outperforms both LWTA and maxout networks. Adding covariance to cluster-select provides a significant boost in performance.

### 4.3.6 Experiment with Gas Sensor Array Dataset

#### Test Setup

For the second catastrophic forgetting test in which covariance was incorporated into the distance calculation, a dataset was utilized that consists of readings from a gas sensor array under dynamic gas mixtures [70]. This dataset has 19 dimensions, including a time dimension which was removed. The task is to determine which of two gas concentrations (Ethylene and CO, or Methane and Ethylene) are present, hence creating a binary classification task. The dataset was first normalized such that each dimension had zero mean and unit variance. Then, to make the dataset more challenging, Gaussian noise was added with variance 1.0.

To test catastrophic forgetting, dataset $P1$ was scrambled such that the input dimensions were randomly rearranged, producing dataset $P2$. Just as before, training is performed on $P1$ until it reaches satisfactory performance (no improvement

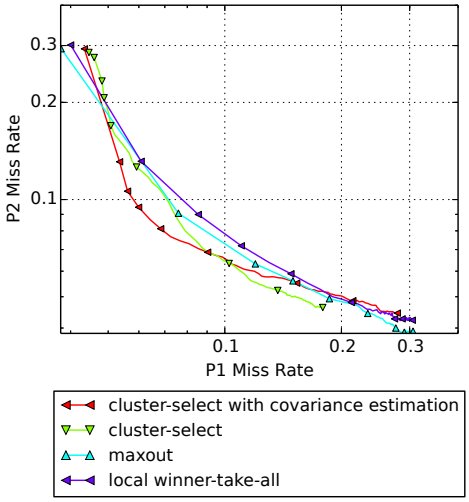



**Figure 4.8:** Cluster-Select Gas Sensor Array Dataset Result

observed on the test set for 30 epochs). Training is then switched to $P2$ and both $P1$ and $P2$ are observed.

#### Simulation Results

The results for the gas dataset follow the same trends as before, cluster-select outperforms both LWTA and maxout. Adding covariance further boosts performance. Cluster-select with covariance estimation is generally closer to the bottom left hand corner of the graph indicating better performance.

### 4.3.7 Pendulum Experiment

For this task a dataset was generated consisting of an ideal 2-d pendulum with no friction. The simple pendulum provides a very straightforward case from physics that is both simple to train, yet can exhibit non-stationary behavior depending on the speed at which the pendulum is swinging. It can be modeled from equation 4.4. This model is commonly found in most introductory physics texts, including [71].

$$\begin{aligned} \frac{d\theta}{dt} &= \omega \\ \frac{d\omega}{dt} &= -\frac{gsin(\theta)}{l} \end{aligned} \tag{4.4}$$

If the pendulum is swinging fast enough it will rotate in a complete circle. However, if the speed is not great enough to counter the effects of gravity, the pendulum will swing back and forth in both directions. Two datasets were created using the above equations where initial conditions were set such that in the first dataset $P1$ the pendulum swings in a complete circle, see Figure 4.9(a), and in dataset $P2$ the pendulum swings back and forth, see Figure 4.9(b). Generating the datasets in this manner allows us to control exactly when, and to what effect the non-stationarity occurs within the dataset, which consequently allows us to precisely observe when, and to what degree, forgetting occurred.

Two separate datasets were generated at velocities that could trigger both behaviors and training was switched from one dataset to the other while measuring performance on both. Dataset $P1$ and $P2$ were each 250 seconds long for a total of 500 seconds. The dataset was generated using scipy's odeint [72] function to integrate the above equations with a sample rate of 30 $samples/sec$. The initial velocity for $P1$ was set to 25.0 $meters/sec$ and to 3.0 $meters/sec$ for $P2$. Gravity was 9.81 $meters/sec^2$ and the pendulum had length 1.0 meters.

### Test Setup

For this problem, the network had to predict $sin(\theta)$ and $-cos(\theta)$ instead of the state variables. These variables may create a better representation for the network instead of predicting $\theta$ and $\omega$ directly, as the angle $\theta$ will contain discontinuities if it is restricted to be on the interval $[0, 2\pi]$.

An Elman [22] network with a single hidden layer was used. The network was shown the input for a single time step in addition to its previous hidden output.

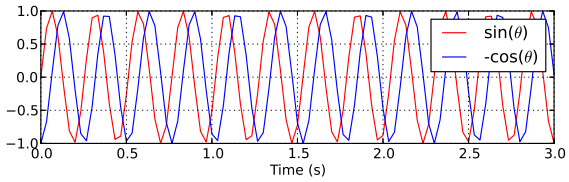
The task was to correctly learn to predict $sin(\theta)$ and $-cos(\theta)$ for the next time step. The network was initialized with a vector of 0s for the hidden state and provided 10 seconds to initialize the hidden state before training began. The dataset was switched to $P2$ at 250.0 seconds, and the hidden layer was again allowed to initialize for 10 seconds before training on $P2$ began at 260.0 seconds.

Training was halted on each time step, and a test was ran over all of $P1$ and $P2$ to measure mean squared prediction errors for $P1$ and $P2$. The error measurement was performed by running the network on $P1$, and measuring the mean squared prediction error for each time step, and taking the mean of that for the whole dataset (hence in Figure 4.10 it is labeled "Mean of MSE"). The network was again ran on $P2$ to generate similar prediction error. The plots in Figure 4.10 show these error measurements for both $P1$ and $P2$.

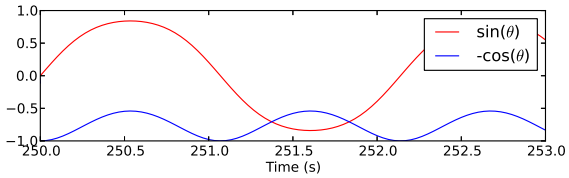
One potential issue is that the previous time step state has only two inputs compared to potentially 256 inputs for the Elman network's previous hidden state. To balance this ratio, the previous time step's state inputs were sometimes duplicated. Otherwise the network may not be able to distinguish those two inputs as being important out of several hundred total inputs. The number of additional duplicate input vectors was set as a hyperparameter for all of the network types.

It should be noted that with this test, there is no training or test dataset. There is only one time series dataset for each regime that is used for both training and testing. Since the pendulum speed isn't dynamic or varying, a test set would have been very similar to the training set and would not have changed the performance. While predicting a pendulum position is an easy problem to solve with a RNN, it appears to be a difficult problem for the network to learn to deduce new pendulum behavior without losing its ability to predict prior behavior, at least when both behavioral regimes of $P1$ and $P2$ were presented only once.

A total of 5 sets of tests were ran. For the first set of tests, the clustering technique was used as described in Section 4.2.1 with learning rate decay. This first set of tests included hyperparameters for the number of hidden neurons, the learning rate,




(a) $P1$ Dataset



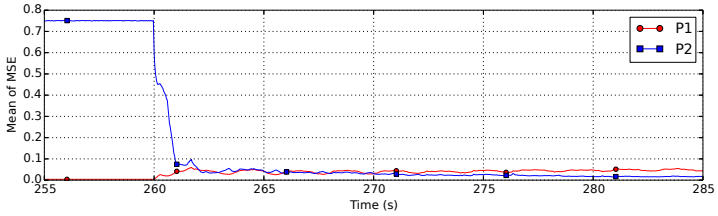

(b) $P2$ Dataset

**Figure 4.9:** Sample of $P1$ and $P2$ datasets. This plot shows $sin(\theta)$ and $-cos(\theta)$ which is the data that is fed to the neural network. The data shown indicates the relative x and y position of the pendulum if it is swinging around the origin. In 4.9(a) the pendulum swings completely around in circles. In 4.9(b) however, the pendulum is swaying back and forth. To see that the pendulum is not swinging around, notice the y position $-cos(\theta)$ does not swing higher than $-0.5$ units.

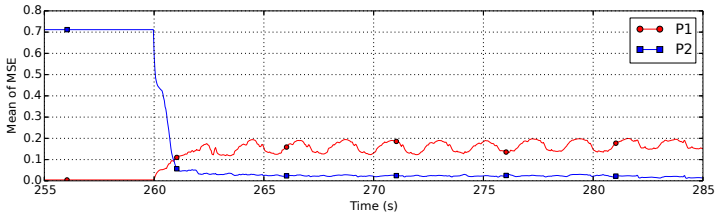
number of centroids selected on each feedforward pass, and learning rate decay rate $\zeta$. The main purpose of the second set of tests was to establish the effectiveness the learning rate decay parameter was having on the results, so the rate $\zeta$ was fixed to 1.0 to effectively disable it. All other hyperparameters were the same as before.

Since it is known that both winner-take-all and maxout networks help with forgetting, a set of tests were ran with hidden layers that had each of these types of networks. The hyperparameters for these tests included the learning rate, the number of hidden nodes, and the number of nodes per group.

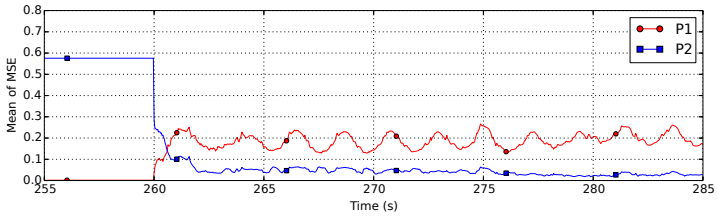
To compare to the baseline, a set of tests is included for a regular neural network with hyperbolic tangent hidden activations. This network had hyperparameters for the number of hidden neurons and the learning rate. Each set of tests were run 100 times using hyperopt with random hyperparameter selection. The best performing runs for each set were selected for the plots in Figure 4.10



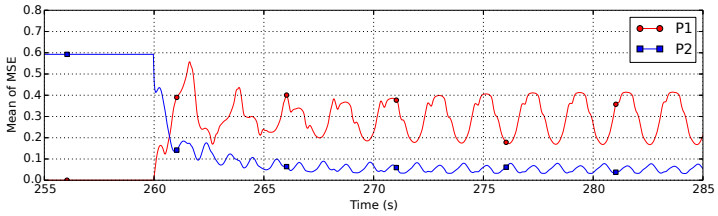

(a) Results for cluster-select with learning rate decay



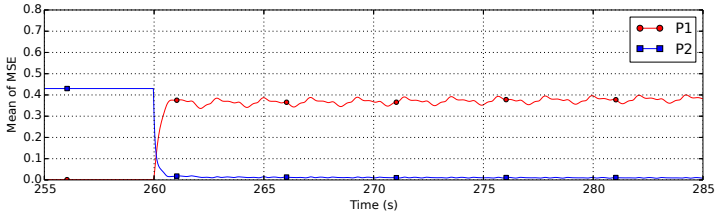

(b) Results for cluster-select with no learning rate decay




(c) Results for local winner-take-all




(d) Results for maxout




(e) Results for regular neural network with tanh activations

**Figure 4.10:** Illustration of $P1$ and $P2$ error rates during training on the recurrent simple pendulum task

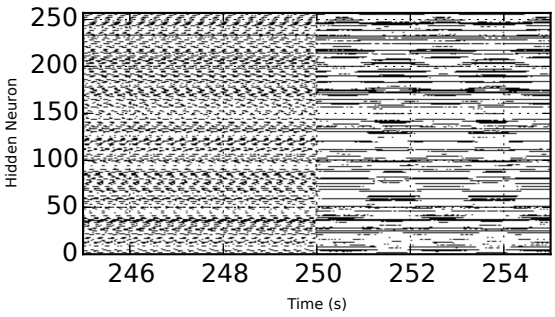



**Figure 4.11:** Selected Neurons During the Regime Change

### Simulation Results

As shown in Figure 4.10, the clustering technique with weight decay performed very well. The pendulum occupies the same physical positions in $P2$ as it does in $P1$, meaning that the regime can't be deduced from the inputs at a specific time step alone. Even still, this network was able to determine (via recurrence) which regime it was in and predict the next time step with little error, after only having been trained on $P1$ and $P2$ once. The other network architectures did not manage to successfully retain both $P1$ and $P2$.

The learning rate decay $\zeta$ was found to be a very sensitive hyperparameter. Setting it too low caused the learning rate to decay rapidly and the network failed to learn anything. Setting it too high (very close to 1.0) effectively disabled the decay and caused the network to forget $P1$ when learning $P2$. It could be argued that this parameter is dataset dependent. Had the network been trained on a different task, it would need to be set to a different value. In theory, this parameter may reflect the sensitivity and delicate balance that stability vs. plasticity has to be tuned for in the real world. In order to retain prior representations, a learning system has to fixate on those representations after being exposed for a certain amount of time. The precise amount of exposure needed depends on the task. The $\zeta$ parameter determines the exposure time the network needs before it permanently retains a particular regime.

Figure 4.11 shows a binary plot of which neurons were selected during the regime change at 250 seconds. This plot was generated from the data for the winning case with learning rate decay. The black lines mean that a particular neuron was selected during a particular time (64 neurons were selected out of 256 total). This depiction shows that there is a substantial overlap between the centroids selected for each dataset, meaning that centroids selected for $P1$ are again being selected for $P2$ which will cause the network to lose some of the $P1$ representation. However, the learning rate decay parameter $\zeta$ accounts for this by ensuring the neurons that were selected for $P1$ have their learning rates decayed such that their representations can no longer be destroyed when training on $P2$. Some new neurons are also being selected which have not had their learning rate decayed, and are thus able to learn $P2$ without interfering with the network's representation of $P1$.

# Chapter 5

# Mitigating Catastrophic Forgetting in Reinforcement Learning Environments

Reinforcement learning is a challenging scenario in which catastrophic forgetting naturally occurs. When an agent explores the environment, it does so through a sequence of observations it acquires. Such a sequence of observations will likely be temporally correlated in that consecutive observations will be similar. If a neural network based value function is updated on each time step, then temporal correlations in the training data will likely trigger catastrophic forgetting.

Even if an agent can overcome the difficulties due to consecutive observations being correlated and can reach a near optimal policy, it may not be able to maintain this policy. An optimal policy, by definition, will only follow the optimal trajectory in the state space, avoiding regions that lead to failure. In many problems, the states which follow the optimal trajectory only represent a very small subset of the total state space. As mentioned previously in chapter 2, the work in [38] demonstrates that training data for an agent must include observations made from non-optimal states in order for the agent to maintain an optimal trajectory. Essentially, due to

catastrophic forgetting, if the agent stops observing non-optimal states, then it will lose the representations it had previously learned for such failure states, and may start visiting those states again, which will cause an optimal policy to degrade to a non-optimal one.

Catastrophic forgetting may manifest as poor or degraded performance in learning a policy. It may also manifest as an "unlearning" effect whereby an agent seems to learn an effective policy but quickly unlearns or regresses to a much poorer policy. Without some strategy to help with catastrophic forgetting, degraded performance and unlearning has been noted in [6] [5] [4] [8].

In this chapter, several reinforcement learning test cases involving Markov decision processes (MDPs) and partially observable MDPs are explored. First we examine an MDP test case which models a cart-pole system where all of the variables are known. Next, we investigate POMDP test cases, where the first test case models a cart pole system with only partial state information. Finally, we explore the pong video game in the arcade learning environment [18]. The arcade learning environment test case examines using cluster-select with standard on-policy methods and compares it to the off-policy deep Q-network (DQN) algorithm with the replay buffer. The cluster-select network is demonstrated to improve stability in a similar manner to using a replay buffer, and allows for on-policy training which isn't possible with a replay buffer.

## 5.1 Forgetting in MDPs

Fully observable Markov Decision Processes (MDPs) have the property that the full state of the environment is available to the agent. This section explores catastrophic forgetting in MDPs by examining a classic reinforcement learning problem involving a cart-pole system.

**Table 5.1:** Constants for Cart-Pole Test

| constant | description | value |
|---|---|---|
| $g$ | gravity $(m/sec^2)$ | 9.81 |
| $l$ | pole length(m) | 0.5 |
| $m_p$ | pole mass (kg) | 0.01 |
| $m_c$ | cart mass(kg) | 1.0 |
| $F$ | horizontal cart force(N) | 10, 0, or -10 depending on action |

### 5.1.1 Cart-Pole Experiment

For our study, we considered the classic cart-pole reinforcement learning problem with no friction. The problem involves a simulated cart on a horizontal track with a pole attached to it. The action space has been discretized such that a total of 3 actions involve applying a left force, right force, or no force. This essentially produces bang-bang [73] controls. An episode consists of the cart fixed with the pole initialized with a small random angle and velocity. An episode proceeds until one of the state variables either grows too large (within reasonable bounds) or 1000 steps elapses.

The differential equations governing the dynamics of this problem are given in equations 5.1 and 5.2 below. These equations include several physical constants, such as mass of the cart and poll as well as the pole length. The values that were selected for these constants are given in table 5.1. The derivations for the cart-pole equations can be found in [74].

$$\ddot{\theta} = \frac{gsin\theta + cos\theta\left(\frac{-F+m_p l\dot{\theta}^2 sin\theta}{m_c+m_p}\right)}{l\left(\frac{4}{3} - \frac{m_p cos^2\theta}{m_c+m_p}\right)} \tag{5.1}$$

$$\ddot{x} = \frac{F + m_p l\left(\dot{\theta}^2 sin\theta - \ddot{\theta}cos\theta\right)}{m_c + m_p} \tag{5.2}$$

Bounding the variables was necessary, since a system having no friction would mean that these state variables could grow unbounded, potentially introducing numerical stability issues. A negative reward was assigned if the episode ended prematurely due to one of the state variables exceeding its predetermined bounds.

The goal of the task is to balance the pole upright by applying the horizontal forces to the cart, hence a small positive reward was applied for every frame that the pole was in an upright position.

Each step of the system was simulated using the Runge-Kutta method of numerically solving the differential equations that govern the system. Each step in the simulation consisted of roughly 20 milliseconds of simulated time, such that 50 steps are equivalent to one second. These tests all used the $SARSA(0)$ learning algorithm with a neural network based value function approximator. One temporal difference update was performed on the network for every step (no batch updates, or replay buffers were used). The centroids for the cluster-select network were initialized randomly and were not moved for this test.

### Test Setup

To test each method, a random search was performed over the hyperparameters with hyperopt [64]. Hyperparameters generally included: the learning rate, a small decay constant for the learning rate to decay, the number of hidden neurons, the gamma constant for temporal difference learning, the amount of reward to provide the agent for balancing the pole relative to the amount of negative reward for going out of bounds, the initial $\epsilon$ chosen for $\epsilon$-greedy exploration, and the amount to decay $\epsilon$. For cluster-select, there was an additional hyperparameter denoting the number of neurons to select for a feedforward pass.

Each activation function was examined separately, where reasonable selections for the hyperparameters were provided for the random search. Upon performing approximately 200 runs for each activation function with a given set of hyperparameters, those that produced the best results were selected. Note that in the plots, performance was measured as a function of the total number of steps that the agent was able to balance the pole and collect reward for an episode.

### Results

Figure 5.1 provides results for a simple case with tabular value representation. This particular result had its hyperparameters hand-tuned (i.e. no random searches of hyperparameters were performed), and it is provided as a simple baseline performance level. The value function was maintained in a table of 80,000 states where the entry in the table was obtained as a function of the 4 state variables. Quantization was performed over the state variables such that the cart position had 10 bins; the cart velocity, pole angle, and pole angular velocity all had 20 bins. The binning was performed over the valid ranges of these state variables. The number of bins for each state variable was a hyperparameter that was hand-tuned. Figure 5.1 also shows a fit to an exponential curve of the form $f(x) = a - b\exp(-cx)$ where $a, b, c$ are constants pertaining to the fitted curve.

Figure 5.3 illustrates the results for a neural network with linear rectified activation functions. This activation function produced some agents with the best performance. Unfortunately, the good performance was unstable, and would often regress as illustrated in this figure. These agents would learn to balance the pole well, then suddenly regress to terrible performance. We hypothesize that this sudden regression is caused by catastrophic forgetting in the hidden layers. Essentially, after the agent begins to learn to balance the pole well, it is unable to maintain this policy since the network is no longer being trained on the failure states. Eventually it drops the pole, and 'unlearns' the previous captured representation. A plot of performance for networks with sigmoid and hyperbolic tangent activations is also provided in Figure 5.4 and Figure 5.5, however these networks did not reach adequate performance. It is unclear why these particular activation functions failed to deliver a proficient policy. It is possible that they simply required more training time, or that a good set of hyperparameters was never found. It is also possible that these activation functions are a poor match for this particular problem.

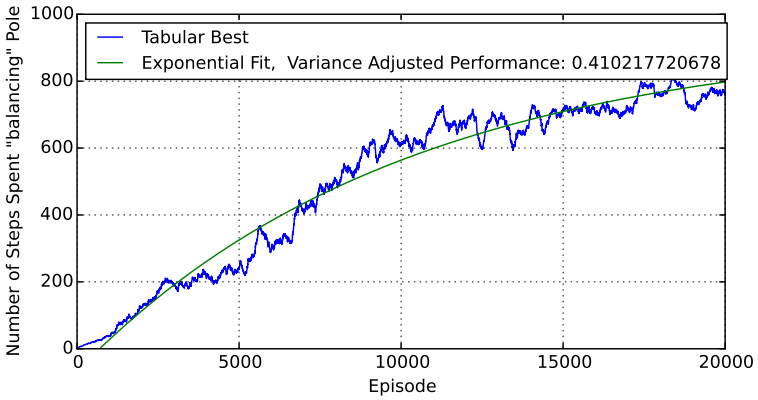



**Figure 5.1:** Result for a Tabular $Q_{s,a}$ Estimator

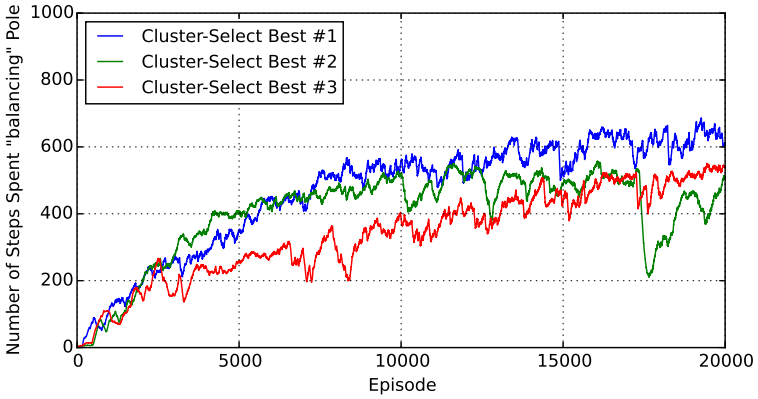



**Figure 5.2:** Result for Cluster-Select Neural Net $Q_{s,a}$ Estimator

On the other hand, the cluster-select technique generally had a much smoother learning curve. In particular, the learning profile does not exhibit sudden dips (regressions) in performance, as Figure 5.2 clearly illustrates. In addition, Table 5.2 provides an objective measure of performance expressed as the log of the variance-adjusted performance. To compute the latter we first fit the performance curve to an exponential function, as depicted in Figure 5.1. Next, we measure the mean squared deviation of the original learning curve from the fitted function. Finally, we define the variance-adjusted performance as the mean of the squared values of the original learning curve relative to the mean squared deviation from the fitted function. This metric favors a learner that is both stable in its learning profile as well as reaches a high performance level.

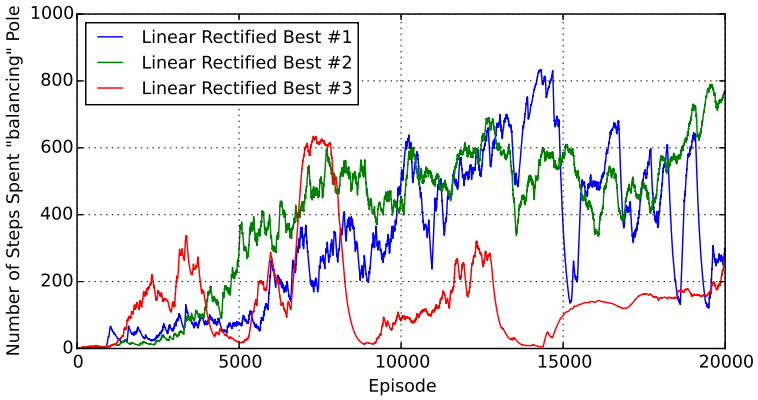



**Figure 5.3:** Result for a Neural Net $Q_{s,a}$ Estimator with Linear Rectified Activations

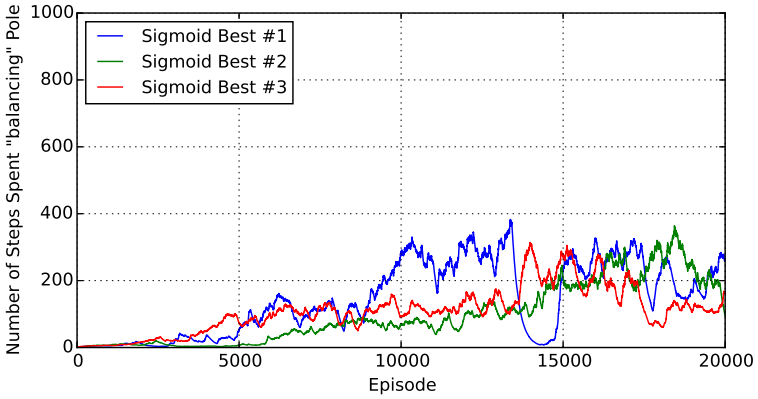



**Figure 5.4:** Result for a Neural Net $Q_{s,a}$ Estimator with Sigmoid Activations

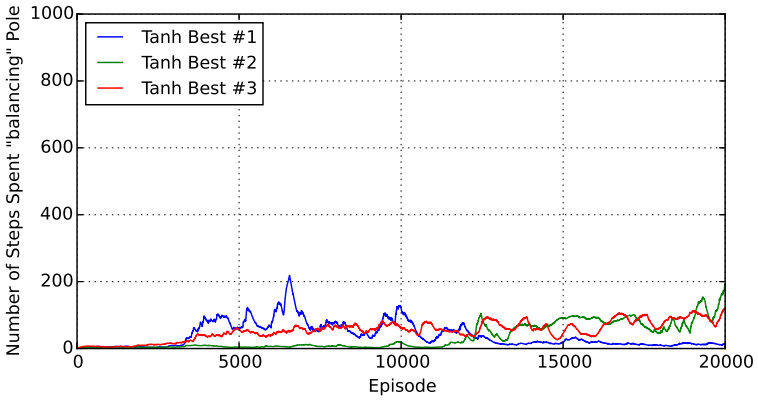



**Figure 5.5:** Result for a Neural Net $Q_{s,a}$ Estimator with Hyperbolic Tangent Activations

**Table 5.2:** Summary of Results

| | Tabular | Tanh | Sigmoid | Linear Rectified | Cluster-Select |
|---|---|---|---|---|---|
| Performance (Mean Squared Sum) | 310311 | 3674 | 17682 | 202616 | 243417 |
| Deviation from Exponential Fit | 1944 | 225.4 | 1083 | 7618 | 801.6 |
| Log Variance Adjusted Performance | 5.073 | 2.791 | 2.793 | 3.281 | **5.716** |

## 5.2 POMDP Environment

POMDPs represent a more challenging environment in which the agent is not provided with the full state of the environment. Instead, information is available to the agent in the form of observations which only contain a partial representation of the full state. This section investigates two partially observable experiments. The first of which is a cart-pole experiment which is similar to the MDP cart-pole test case previously covered, except that a state variable has been removed to create a POMDP. The second experiment involves the pong video game in the arcade learning environment.

### 5.2.1 Partially Observable Cart-Pole Experiment

For the partially observable cart-pole simulation, an Elman network with 2 hidden layers was constructed such that the output from the first hidden layer is fed back as input for the next time step. The system has 4 state variables: pole angle, pole angular velocity, cart position, and cart velocity. To make the cart-pole simulation partially observable, we chose to omit one of the 4 state variables.

The cart pole simulation was similar to the fully observable case described earlier, however there were several notable improvements to the training methodology. An eligibility trace update rule was performed, as described in chapter 2. Moreover, the centroids for cluster-select were moved using an adaptive technique described next.

The general problem with moving centroids is that it directly impacts which neurons are associated with different regions of the input space, thus changing the representation and distorting previous training. It has been observed that moving centroids during training results in poor performance, perhaps because of the changing representation. Unfortunately, online clustering techniques require that the centroids be updated and moved closer to the data clusters.

The real issue is that centroids can not be moved during training since it invalidates any prior training. Nevertheless, it may be important to perform online clustering so the centroids cover the data regions more naturally, instead of being fixed to their initial random locations. To allow the centroids to be moved closer to the data clusters without severely affecting performance, a per-neuron $\xi$ term was provided to balance training the neural network and moving the centroids. The $\xi$ term was initialized at 1 for all neurons. If the neuron $j$ belonged to the set of selected neurons, its $\xi_j$ term was decayed by a constant factor $\xi_j = \alpha\xi_j$ where $\alpha$ is the decay rate. During updates, the centroids were moved closer to the input which selected them

$$\vec{c}_j^{new} = \vec{c}_j + \xi_j(\vec{c}_j - \vec{x}), \tag{5.3}$$

where $\vec{x}$ is the input that selected the centroid, and $\vec{c}_j$ is the centroid for neuron $j$. The learning rate for the input weights to neuron $j$ was multiplied by $1 - \xi_j$, meaning that when a neuron is first selected, no training occurs on that neuron, but instead its corresponding centroid is moved closer to the data that selected it. As $\xi_j$ decays, the neuron begins to learn, and the centroid slows down and eventually stops moving. The $\xi$ term, in general allows the network to initialize the centroids before training commences.

### Test Setup and Results

The hyperparameters were similar for the POMDP cart-pole test as for the fully observable cart-pole experiment. The number of neurons in the first and second

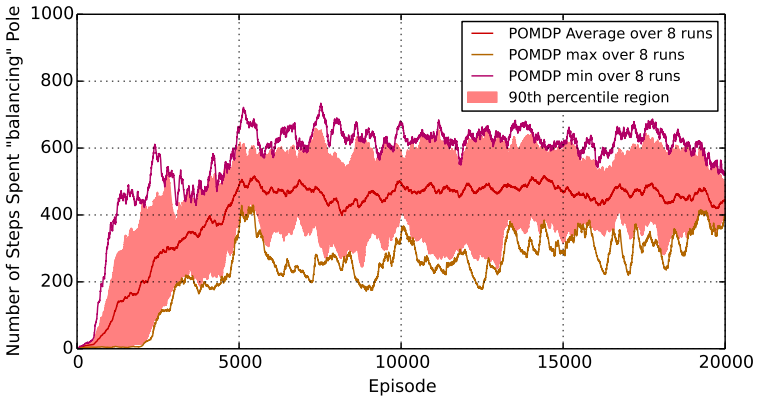



**Figure 5.6:** Result for Cluster-Select Neural Net $Q_{s,a}$ Estimator on a POMDP Cart-pole Test Case

hidden layers were both hyperparameters along with the standard reinforcement learning hyperparameters as before. In addition, a $\xi$ decay hyperparameter was also included. For the POMDP cart-pole problem, the hyperparameter search proved difficult. Approximately 3,000 runs were performed before finding a single run that performed relatively well.

The good performance of this single run turned out to be reproducible. That is, upon rerunning with the same hyperparameters but different initialization conditions, good performance was observed. Figure 5.6 is a plot of performing this run 8 different times, each with different initialization conditions. The top average score achieved was 730 steps spent balancing the pole. This result is comparable overall to the fully observable MDP result.

### 5.2.2 Arcade Learning Environment Experiment

The arcade learning environment [18] has recently become an extremely popular benchmark for reinforcement learning. It allows one to run experiments which are aimed at training agents to play many different Atari 2600 video games. A recent success has been published by Mnih et al. [8] which uses a deep convolutional network,

dubbed the deep Q-network (DQN), that learned to play multiple Atari 2600 video games.

In the experiment performed here, we attempt to reproduce some of the DQN work, but on a smaller scale. The experiments are not performed using screen images of the game, but instead are conducted using an explicit partial state representation that is extracted from the game's memory. Even with this simpler partial representation, the experiment appears to suffer from instabilities which are likely caused by catastrophic forgetting. We attempt to reproduce the DQN training algorithm, including the replay buffer, and freezing the value function. The replay-buffer was compared with a cluster-select network and we demonstrated in this case that cluster-select outperforms using a replay buffer in both convergence speed and stability. In running other experiments, we noted that at least some mitigating seems necessary. Naively using a regular feedforward network with no strategy for mitigating catastrophic forgetting failed to yield a proficient agent.

### Reproducing Deep Q-Network Mitigation Techniques

The work by Mnih et al. [8], introduces the DQN algorithm and includes several techniques aimed at mitigating forgetting which we attempt to reproduce. Figure 5.7 illustrates the proposed algorithm which is nearly identical to the DQN training algorithm. Many of the modifications made to $Q$-learning by DQN were considered including the use of a replay buffer. In addition, we generate targets from a value function which is frozen every $C$ steps.

The replay buffer contains sets of state/action transitions and associated rewards incurred. Every time a transition occurs, its corresponding state, action, and associated reward is inserted into the replay buffer. A circular data structure is utilized, such that when it is full, the oldest entries are overwritten by any new transitions. The $Q$ value estimator network is updated by randomly sampling mini-batches of transitions from the replay buffer and applying gradient descent with the $Q$-learning update rule.

**Algorithm 3:** Q-learning with experience replay

Initialize replay memory $D$ to capacity $N$
Initialize action-value function $Q$ with random weights $\theta$
Initialize target action-value function $\hat{Q}$ with random weights $\theta^{-}$
**for** *episode = 1,M* **do**
  **for** $t = 1,T$ **do**
    With probability $\epsilon$ select a random action $a_t$
    Otherwise select $a_t = argmax_a Q(o_t, a; \theta)$
    Execute action $a_t$ in emulator and observe reward $r_t$ and observation $o_{t+1}$
    Store transition $(o_t, a_t, r_t, o_{t+1})$ in $D$
    Sample random mini-batch of transitions $(o_j, a_j, r_j, o_{j+1})$ from $D$
    $$y_j \leftarrow \begin{cases} r_j & \text{if episode terminates at next step} \\ r_j + \gamma max_{a'} \hat{Q}(o_{j+1}, a'; \theta^{-}))^2 & \text{otherwise} \end{cases}$$
    Perform a gradient descent step on $(y_j - Q(O_j, a_j; \theta))^2$ with respect to the network parameters $\theta$
    Every $C$ steps reset $\hat{Q} = Q$
  **end**
**end**

**Figure 5.7:** Modified algorithm from deep Q-learning with experience replay.

Similar to [8], a separate target network was provided for generating targets for the value function. Every $C$ steps, the weights for the current $Q$-network are copied to create the $\hat{Q}$-network. The $\hat{Q}$-network generates targets that are used for updating the weights to the $Q$-network. Using a frozen network to generate targets for value updates was empirically shown in [8] to greatly improve results. Since changing the value of one state will likely have an immediate effect on other states, a feedback effect could occur where changes to the value function could have an immediate effect on the policy, leading to oscillations or other instabilities. Generating targets from a separate $\hat{Q}$-network can help prevent feedback effects, however it has the expense of potentially slowing training time significantly.

Gradient clipping was implemented, in which updates to the loss function were bounded to the range $(-1, 1)$. Clipping the gradient prevents large changes from occurring to the network parameters and is helpful in stabilizing training.

### Comparison to Cluster-Select

For the cluster-select experiment, the centroids were moved by decaying a $\xi$ parameter which balanced between learning and moving centroids as described in section 5.2.1. The standard $SARSA(\lambda)$ algorithm was employed without the replay buffer. The use of $SARSA$ means that our technique is on-policy and does not suffer from the stability issues of off-policy techniques such as $Q$-learning.

A basic assumption in supervised learning is that the samples are drawn in a stationary manner. If this assumption is violated, then oscillations in the learning tend to take place. Generally, a replay buffer can be viewed as a heuristic technique for imposing stationarity, since the statistics change slowly when sampling randomly from the past history. While a replay buffer works for mitigating catastrophic forgetting, cluster-select can be considered an alternative technique. In this work, we wish to compare both approaches and show that in at least one test case, cluster-select offers several advantages over a replay buffer.

A replay buffer attempts to mitigate forgetting by storing past history into a large data structure and replaying from it. One issue is that this approach does not scale, since as the problem size grows the buffer must be made larger. Moreover, It was noted earlier in this chapter that an agent must be trained on observations from non-optimal states in order to maintain an optimal policy. Suppose the agent reaches an optimal policy through a replay buffer, then only optimal state-actions will be stored in the buffer. At some point, the optimal states stored in the replay buffer will overwrite any non-optimal ones, and the neural network may still fail to maintain its optimal policy due to catastrophic forgetting.

One disadvantage of using a replay buffer is that it only works best with off-policy techniques. Off-policy learning means that the value function of the greedy policy is learned while following a non-greedy policy. On-policy learning attempts to approximate the value of the current policy being followed which may include non-greedy or exploratory actions. If a replay buffer is combined with on-policy learning,

then a transition which is stored in the buffer may reflect an action which would be taken by an older policy, instead of the current policy. If the value function is updated with values from transitions which would not be taken by the current policy, then by definition the agent is not learning the current policy. In practice, updating on such actions may degrade the current policy or prevent it from being improved.

A replay buffer can not be combined with eligibility traces, which is a consequence of being limited to off-policy learning. Eligibility traces (covered previously in chapter 2) can significantly speed up training by allowing credit to be assigned more than one time step into the past. Eligibility traces only work with on-policy techniques because the credit can only be assigned into the past for actions that were followed, since the resulting state trajectory of actions that were not taken is unknown during training.

In addition to the increased memory requirements, a replay buffer can also impose significant CPU requirements due to the nature of mini-batch updating. When training online with $SARSA$, updates are performed during every step as samples are received. By contrast, the DQN algorithm performs a whole mini-batch update for every step.

In the DQN paper [8], generating $Q$-learning targets from a separate frozen value function was demonstrated to help improve stability and to obtain better results. In our case, we are using $SARSA$ instead of $Q$-learning, which is more stable in general. We did not see the need to generate targets from a separate frozen value function when performing on-policy learning.

Even with all of these disadvantages, experience replay was demonstrated to greatly aid in solving a very difficult problem. We would like to show that cluster-select, which is an alternative network architecture aimed at mitigating catastrophic forgetting, can also perform well at the same tasks. Since replay buffers are not being utilized, cluster-select training can be performed using on-policy techniques which can be combined with eligibility traces to greatly speed up learning.

### Test Setup

This test was performed using the arcade learning environment [18] on the Pong video game. A full episode of pong was played until either the player or the opponent scored 21 points. The score reflected in the plots is the opponent score subtracted from the player score providing for a range of $-21$ to 21. Similar to the methodology in the DQN paper, a step consists of 4 emulated frames in which an action consists of a button press that is held for the 4 frames of emulation.

The state that was presented to the agent consists of 5 values pulled from the game's memory, including the y position of the player paddle, the x and y position of the ball, and the horizontal and vertical velocity of the ball. This captures much of the dynamics of the game, however there is still hidden state involving the opponent position.

As in all other experiments performed thus far, hyperopt [64] was used to perform a hyperparameter search over many of the parameters. We attempted to be as fair as possible in comparing cluster-select to the DQN results. For both approaches we included reinforcement learning hyperparameters, including $\gamma$ and $\epsilon$ (for $\epsilon$-greedy search), learning rate, and a decay parameter for the learning rate. The lower limit for the learning rate was also a hyperparameter. A reward multiplier hyperparameter was included to scale the rewards returned by the arcade learning environment.

For the DQN algorithm, we used a network with a single hidden layer of rectified linear units. The number of hidden units was a hyperparameter which was varied from 64 to 2048. The replay buffer was set to a fixed size of 1,000,000 steps since that was used in the DQN paper. For the cluster-select experiment, we had hyperparameters for the number of hidden units, number of centroids to be selected, and the decay rate for $\xi$.

### Results

Since the hyperparameter space was very large, many runs were performed using a random hyperparameter search over reasonable values. Approximately 3,000 runs were performed for both cluster-select with $SARSA$ and for a standard feedforward neural network with the DQN algorithm. The actual raw training curves for both techniques were found to be difficult to interpret. That is, under both cluster-select and with the DQN algorithm, some of the runs exhibited many different behaviors, including regressions and instabilities. Perhaps due to poor hyperparameter initialization, most of the runs failed to converge to a good solution at all.

The top performers in the hyperparameter search for both the DQN algorithm and for cluster-select were isolated and compared, particularly in terms of stability and convergence speed. For convergence speed, the number of episodes it took to reach a particular score was measured. For stability, the measure is the number of consecutive episodes a particular run was able to be at or above a particular score. If a run suffers from catastrophic forgetting, or regresses for other reasons, it will drop in performance. Stability is measured in the sense that the agent is able to maintain a particular score without dropping in performance. Both plots were created from the top 8 performers for both techniques, and the shaded regions illustrate the 90'th percentile region.

Figure 5.8 is a measure of converge speed for both $SARSA$ with cluster-select and a standard feedforward neural network with the DQN algorithm. It makes sense that the DQN algorithm would take longer to converge since it freezes the value function and only issues an update every 10,000 episodes. By contrast, cluster-select with $SARSA$ results in an update to the value function at every step. Moreover, cluster-select with $SARSA$ employs eligibility traces to speed up convergence. The x axis in this plot reflects the number of episodes elapsed prior to exceeding a particular score given on the y axis. This plot does not include regressions in performance,

which a plot of the actual training curve would include. It only illustrates the time it took before the technique reached a performance level. As illustrated in the figure, both cluster-select with $SARSA$, and the DQN algorithm with the replay buffer were able to reach an average score of about 20.5, however cluster-select exhibited faster convergence to higher scores.

Figure 5.9 is a measure of stability. Since regressions or instabilities cause the training curve to degrade, we chose to plot the number of consecutive episodes during which the agent maintained a particular score. In this plot, the x axis reflects the longest consecutive sequence of episodes normalized to 20,000 episodes in which the agent exceeded a score given on the y axis. For this example, cluster-select exhibits a more stable profile in that it maintains higher scores for longer durations. To give an example, suppose a particular run reaches an average score of 20 at episode $15,000$ but then drops below 20 at episode $16,000$ and never returns to a score of 20. Then the maximum consecutive number of episodes for a score of 20 would be $1,000$ episodes, or 0.05 when normalized.

An attempt was also made to train a regular neural network using $SARSA$ and regular $Q$-learning, which failed to converge even after a extensive hyperparameter search. No results are shown for this setting since these experiments failed, emphasizing the need for some technique for mitigating forgetting. While a score of approximately 20.5 points was achieved using both techniques, it has been demonstrated that using a cluster-select neural network based value function estimator with the on-policy $SARSA$ algorithm outperformed training a regular neural network with the DQN algorithm in terms of convergence speed and stability.

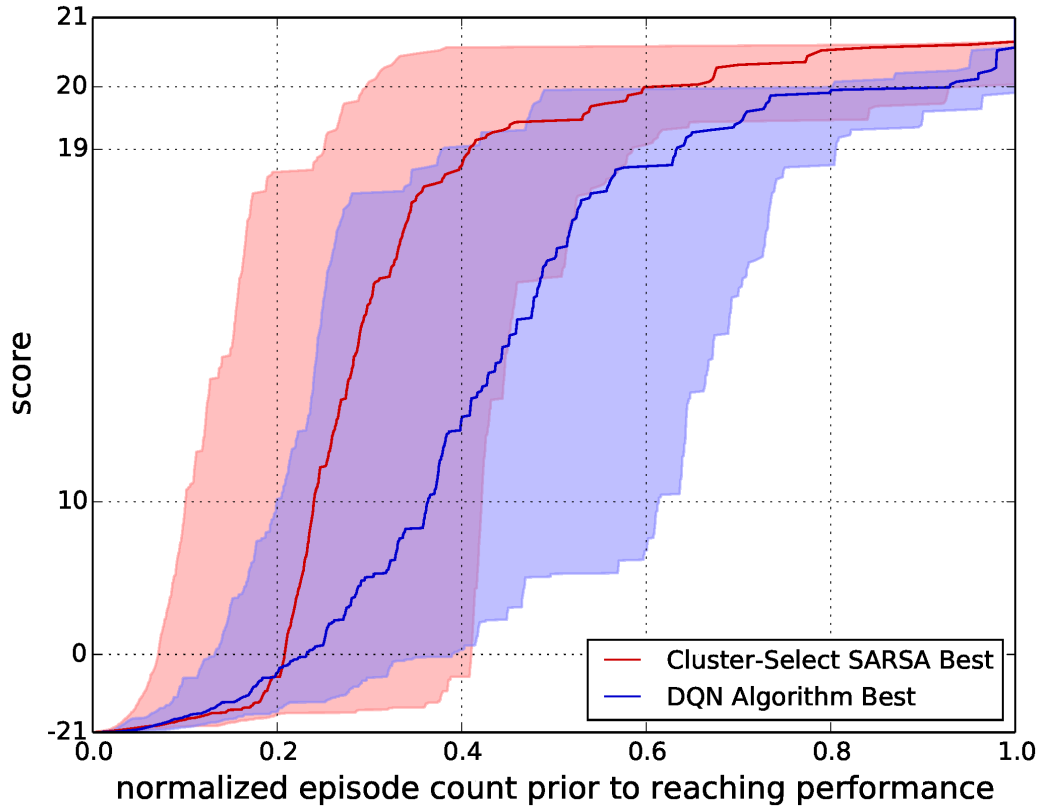



**Figure 5.8:** Plot of Number of Episodes Before Achieving a Score (Convergence Speed)

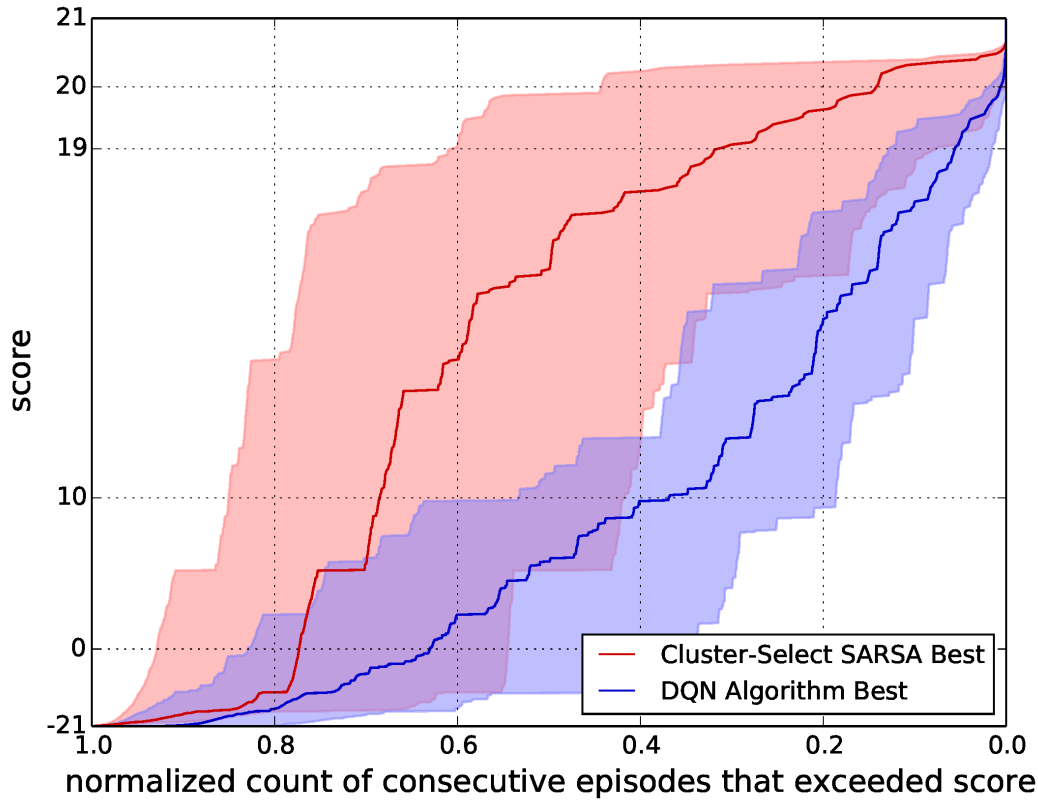



**Figure 5.9:** Number of Consecutive Episodes which were Greater Than a Particular Score (Stability)

# Chapter 6

# Conclusions

## 6.1 Summary of Contributions

The work presented here explored catastrophic forgetting in neural networks in the context of supervised and reinforcement learning. One major contribution was to develop a mathematical model of catastrophic forgetting which revealed it is primarily caused by overlapping global representations, where changes to the weights associated with one region of the input space can negatively affect distant regions. The overlap was shown to correspond to the magnitude of the dot product between samples in addition to the dot products between their hidden layer representations. Motivated to remove this overlap, a technique of partitioning neurons by clustering (dubbed cluster-select) was introduced.

Cluster-select was first tested in classification and regression settings. The classification problem was made non-stationary by switching the class labels during training. Multiple test cases on several classification datasets were considered which demonstrated cluster-select yielded improved performance. Cluster-select was further extended by incorporating covariance estimation to utilize Mahalanobis distance when computing the distance from centroids to sample points resulting in additional

improvement. The regression problem involved a simple pendulum which is non-stationary due to the varying speeds at which the pendulum swings. Cluster-select was demonstrated to outperform other state-of-the-art techniques on these classification and regression tasks.

Next, the work shifted to reinforcement learning test cases. Reinforcement learning is much more realistic in terms of where catastrophic forgetting may actively be precluding neural networks from being useful in solving difficult problems. Tests were first performed on a MDP environment with a cart-pole system, demonstrating that cluster-select produced the most stable training curve compared to other neural network architectures. Tests were also performed on a POMDP cart-pole problem illustrating performance comparable to the fully observable case. Finally, the arcade learning environment test case was developed, in which the DQN algorithm was reproduced along with its required replay buffer. Cluster-select was demonstrated to outperform the DQN algorithm in terms of both convergence speed and stability.

## 6.2 Future Work

Several directions exist in which this work can be extended:

- We hypothesize that clustering should not be performed over the input space, instead the output space should be used to guide clustering. If a function has a more complex structure in certain regions, then more neurons need to be allocated in those regions to provide more resources in learning additional complexity. If instead, the function is relatively smooth for particular regions, then only a few neurons should be needed in those regions. The output space must be used to determine the appropriate locations for the centroids

- Perhaps one way to utilize the output space is to allow the error gradient to guide where centroids are allocated. In particular, the magnitude of the error gradient provides information for what the neural network has learned in contrast to what

it hasn't learned. We were able to use the error magnitude in the classification tests to guide allocation of centroids for novel inputs, however it wasn't as straightforward to accomplish this in the regression and in the reinforcement learning test cases. More work needs to be performed to better understand how centroid allocation can correspond to estimation errors.

- In reinforcement learning, more investigation is needed to discover better techniques of tuning the hyperparameter values. Some of our test cases required 3,000 runs (using random search) before finding an optimal set of hyperparameters that gave adequate performance. To improve reinforcement learning such that it is more practical for real world applications, better techniques need to be developed that aren't as sensitive to hyperparameters.

- We performed several preliminary tests which utilized Mahalanobis distance with covariance estimation providing for improvements in our results. This work could probably be taken even further. We found the Mahalanobis distance to be slow computationally for high dimensional datasets when calculated on CPU architectures. Computing distances for multiple neurons scales naturally to parallel architectures such as GPU architectures. This work could be moved to GPU architectures for a performance boost, which would enable it to be used on more challenging problems.

- An analysis of forgetting was performed in chapter 3 in which we attempted to characterize the upper bound on forgetting when training across two datasets. The linear case was analyzed since linear networks have been observed to follow similar training dynamics to nonlinear networks. Future work should analyze the nonlinear case to explore how forgetting emerges in such scenarios. In particular, it would be interesting to look at sparse activation functions such as rectified linear, maxout, or cluster-select networks to analyze how forcing activations to zero impacts the forgetting phenomenon.

## 6.3 Concluding Remarks

In terms of broader impact, we hope to bring attention to the problem of catastrophic forgetting. If catastrophic forgetting could be solved in an optimal manner, neural networks will become the leading approach in solving reinforcement learning as well as other challenging control problems. More research needs to be performed to discover better techniques of mitigating catastrophic forgetting, and at the very least, we hope this work will bring attention to the problem and encourage researchers to find more solutions. If neural networks can be improved such that they don't suffer from forgetting, and learn in a more stable manner from non-stationary data, then they will become a very powerful component in driving future technologies.

## 6.4 Publications

The following publications resulted from the work in this dissertation:

- Ben Goodrich and Itamar Arel. Neuron clustering for balancing stability and plasticity in recurrent neural networks. *Systems, Man, and Cybernetics, Part B: Cybernetics, IEEE Transactions on*, submitted for publication, review pending.

- T. Lancewicki, Ben Goodrich, and Itamar Arel. Sequential covariance-matrix estimation with application to mitigating catastrophic forgetting. In *Machine Learning and Applications and Workshops (ICMLA), 2015 14th International Conference on*, page to appear. IEEE, 2015.

- Ben Goodrich and Itamar Arel. Mitigating catastrophic forgetting in temporal difference learning with function approximation. In *Proceedings of the 2nd Multidisciplinary Conference on Reinforcement Learning and Decision Making*, 2015.

- Ben Goodrich and Itamar Arel. Neuron clustering for mitigating catastrophic forgetting in feedforward neural networks. In *Computational Intelligence in Dynamic and Uncertain Environments (CIDUE), 2014 IEEE Symposium on*, pages 62–68. IEEE, 2014.

- Ben Goodrich and Itamar Arel. Unsupervised neuron selection for mitigating catastrophic forgetting in neural networks. In *Circuits and Systems (MWSCAS), 2014 IEEE 57th International Midwest Symposium on*, pages 997–1000. IEEE, 2014.

- Ben Goodrich and Itamar Arel. Reinforcement learning based visual attention with application to face detection. In *Computer Vision and Pattern Recognition Workshops (CVPRW), 2012 IEEE Computer Society Conference on*, pages 19–24. IEEE, 2012.

- Ben Goodrich and Itamar Arel. Consolidated actor critic reinforcement learning model applied to face detection. In *Proceedings of the 50th Annual Southeast Regional Conference*, pages 379–380. ACM, 2012.

- Nicole Pennington, Ben Goodrich, and Itamar Arel. Contrasting infant perception data with a reinforcement learning visual search model. In *BICA*, pages 282–287, 2011.

# Bibliography

[1] O.-M. Moe-Helgesen and H. Stranden, "Catastophic forgetting in neural networks," *Dept. Comput. & Information Sci., Norwegian Univ. Science & Technology (NTNU), Trondheim, Norway, Tech. Rep*, 2005. 1, 22

[2] Y. LeCun, L. Bottou, G. Orr, and K. Muller, "Efficient backprop," in *Neural Networks: Tricks of the trade*, G. Orr and M. K., Eds. Springer, 1998. 1, 4, 5, 8, 24, 44

[3] J. A. Boyan and A. W. Moore, "Generalization in reinforcement learning: Safely approximating the value function," in *Advances in Neural Information Processing Systems 7*. MIT Press, 1995, pp. 369–376. 2, 28

[4] J. R. N. Forbes, "Reinforcement learning for autonomous vehicles," Ph.D. dissertation, UNIVERSITY of CALIFORNIA, 2002. 2, 69

[5] S. Weaver, L. Baird, and M. Polycarpou, "Preventing unlearning during online training of feedforward networks," in *Intelligent Control (ISIC), 1998. Held jointly with IEEE International Symposium on Computational Intelligence in Robotics and Automation (CIRA), Intelligent Systems and Semiotics (ISAS), Proceedings*, 1998, pp. 359–364. 2, 69

[6] V. U. Cetina, "Multilayer perceptrons with radial basis functions as value functions in reinforcement learning." in *ESANN*, 2008, pp. 161–166. 2, 28, 69

[7] V. Uc-Cetina, "A novel reinforcement learning architecture for continuous state and action spaces," *Advances in Artificial Intelligence*, vol. 2013, p. 7, 2013. 2

[8] V. Mnih, K. Kavukcuoglu, D. Silver, A. A. Rusu, J. Veness, M. G. Bellemare, A. Graves, M. Riedmiller, A. K. Fidjeland, G. Ostrovski *et al.*, "Human-level

control through deep reinforcement learning," *Nature*, vol. 518, no. 7540, pp. 529–533, 2015. 2, 3, 26, 29, 69, 77, 78, 79, 81

[9] Y. N. Dauphin and Y. Bengio, "Big neural networks waste capacity," *arXiv preprint arXiv:1301.3583*, 2013. 2

[10] R. Ratcliff, "Connectionist models of recognition memory: constraints imposed by learning and forgetting functions." *Psychological review*, vol. 97, no. 2, p. 285, 1990. 2, 18, 26

[11] R. M. French, "Using semi-distributed representations to overcome catastrophic forgetting in connectionist networks," in *In Proceedings of the 13th Annual Cognitive Science Society Conference*. Erlbaum, 1991, pp. 173–178. 2, 22

[12] A. Robins, "Catastrophic forgetting, rehearsal and pseudorehearsal," *Connection Science*, vol. 7, no. 2, pp. 123–146, 1995. 2, 26

[13] R. M. French, "Pseudo-recurrent connectionist networks: An approach to the'sensitivity-stability'dilemma," *Connection Science*, vol. 9, no. 4, pp. 353–380, 1997. 2, 27

[14] B. Ans and S. Rousset, "Avoiding catastrophic forgetting by coupling two reverberating neural networks," *Comptes Rendus de l'Académie des Sciences-Series III-Sciences de la Vie*, vol. 320, no. 12, pp. 989–997, 1997. 2

[15] R. Coop, A. Mishtal, and I. Arel, "Ensemble learning in fixed expansion layer networks for mitigating catastrophic forgetting," *IEEE transactions on neural networks and learning systems*, 2013. 2, 24, 27

[16] R. K. Srivastava, J. Masci, S. Kazerounian, F. Gomez, and J. Schmidhuber, "Compete to compute," in *Advances in Neural Information Processing Systems*, 2013, pp. 2310–2318. 2, 24, 25, 49

[17] I. J. Goodfellow, M. Mirza, X. Da, A. Courville, and Y. Bengio, "An empirical investigation of catastrophic forgeting in gradient-based neural networks," *arXiv preprint arXiv:1312.6211*, 2013. 2, 18, 25, 49, 53

[18] 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, 06 2013. 3, 69, 77, 82

[19] S. Haykin, "Neural networks: a comprehensive foundation 2nd edition," *Upper Saddle River, NJ, the US: Prentice Hall*, 1999. 4, 7

[20] R. O. Duda, P. E. Hart, and D. G. Stork, *Pattern classification*. John Wiley & Sons,, 1999. 4

[21] F. Rosenblatt, "The perceptron: a probabilistic model for information storage and organization in the brain." *Psychological review*, vol. 65, no. 6, p. 386, 1958. 5

[22] J. L. Elman, "Finding structure in time," *Cognitive science*, vol. 14, no. 2, pp. 179–211, 1990. 10, 47, 62

[23] P. J. Werbos, "Backpropagation through time: what it does and how to do it," *Proceedings of the IEEE*, vol. 78, no. 10, pp. 1550–1560, 1990. 10

[24] Y. Bengio, P. Simard, and P. Frasconi, "Learning long-term dependencies with gradient descent is difficult," *Neural Networks, IEEE Transactions on*, vol. 5, no. 2, pp. 157–166, 1994. 10

[25] R. Pascanu, T. Mikolov, and Y. Bengio, "On the difficulty of training recurrent neural networks," *arXiv preprint arXiv:1211.5063*, 2012. 10

[26] R. S. Sutton and A. G. Barto, *Introduction to reinforcement learning*. MIT Press, 1998. 11, 15

[27] C. J. Watkins and P. Dayan, "Q-learning," *Machine learning*, vol. 8, no. 3-4, pp. 279–292, 1992. 13

[28] A. Robins, "Sequential learning in neural networks: A review and a discussion of pseudorehearsal based methods," *Intelligent Data Analysis*, vol. 8, no. 3, pp. 301–322, 2004. 18, 47

[29] R. French, "Dynamically constraining connectionist networks to produce distributed, orthogonal representations to reduce catastrophic interference," 1994. 22

[30] D. S. Broomhead and D. Lowe, "Radial basis functions, multi-variable functional interpolation and adaptive networks," DTIC Document, Tech. Rep., 1988. 23

[31] I. J. Goodfellow, D. Warde-Farley, M. Mirza, A. Courville, and Y. Bengio, "Maxout networks," *arXiv preprint arXiv:1302.4389*, 2013. 24, 25

[32] G. E. Hinton, N. Srivastava, A. Krizhevsky, I. Sutskever, and R. R. Salakhutdinov, "Improving neural networks by preventing co-adaptation of feature detectors," *arXiv preprint arXiv:1207.0580*, 2012. 25

[33] N. Srivastava, "Improving neural networks with dropout," Ph.D. dissertation, University of Toronto, 2013. 25

[34] J. L. McClelland, B. L. McNaughton, and R. C. O'Reilly, "Why there are complementary learning systems in the hippocampus and neocortex: insights from the successes and failures of connectionist models of learning and memory." *Psychological review*, vol. 102, no. 3, p. 419, 1995. 27

[35] G. Tesauro, "Temporal difference learning and td-gammon," *Communications of the ACM*, vol. 38, no. 3, pp. 58–68, 1995. 28

[36] J. N. Tsitsiklis and B. Van Roy, "An analysis of temporal-difference learning with function approximation," *Automatic Control, IEEE Transactions on*, vol. 42, no. 5, pp. 674–690, 1997. 28

[37] C. Gaskett, D. Wettergreen, and A. Zelinsky, "Q-learning in continuous state and action spaces," in *Australian Joint Conference on Artificial Intelligence*. Springer, 1999, pp. 417–428. 28, 29

[38] S. Ross and D. Bagnell, "Efficient reductions for imitation learning," in *International Conference on Artificial Intelligence and Statistics*, 2010, pp. 661–668. 28, 68

[39] M. Riedmiller, "Neural fitted q iteration–first experiences with a data efficient neural reinforcement learning method," in *Machine Learning: ECML 2005*. Springer, 2005, pp. 317–328. 29

[40] S. Grossberg, "How does a brain build a cognitive code?" in *Studies of Mind and Brain*. Springer, 1982, pp. 1–52. 30

[41] W. C. Abraham and A. Robins, "Memory retention–the synaptic stability versus plasticity dilemma," *Trends in neurosciences*, vol. 28, no. 2, pp. 73–78, 2005. 30

[42] R. M. French and A. Ferrara, "Modeling time perception in rats: Evidence for catastrophic interference in animal learning," in *Proceedings of the 21st Annual Conference of the Cognitive Science Conference*. Citeseer, 1999, pp. 173–178. 30

[43] L. Buhry, A. H. Azizi, and S. Cheng, "Reactivation, replay, and preplay: how it might all fit together," *Neural plasticity*, vol. 2011, 2011. 30

[44] M. S. Nokia, M. Penttonen, and J. Wikgren, "Hippocampal ripple-contingent training accelerates trace eyeblink conditioning and retards extinction in rabbits," *The Journal of Neuroscience*, vol. 30, no. 34, pp. 11 486–11 492, 2010. 30

[45] A. S. Dave and D. Margoliash, "Song replay during sleep and computational rules for sensorimotor vocal learning," *Science*, vol. 290, no. 5492, pp. 812–816, 2000. 30

[46] W. E. Skaggs, B. L. McNaughton, M. Permenter, M. Archibeque, J. Vogt, D. G. Amaral, and C. A. Barnes, "Eeg sharp waves and sparse ensemble unit activity in the macaque hippocampus," *Journal of neurophysiology*, vol. 98, no. 2, pp. 898–910, 2007. 30

[47] H. F. Ólafsdóttir, C. Barry, A. B. Saleem, D. Hassabis, and H. J. Spiers, "Hippocampal place cells construct reward related sequences through unexplored space," *Elife*, vol. 4, p. e06063, 2015. 30

[48] D. Margoliash and M. F. Schmidt, "Sleep, off-line processing, and vocal learning," *Brain and language*, vol. 115, no. 1, pp. 45–58, 2010. 30

[49] B. Goodrich and I. Arel, "Unsupervised neuron selection for mitigating catastrophic forgetting in neural networks," in *Circuits and Systems (MWSCAS), 2014 IEEE 57th International Midwest Symposium on*. IEEE, 2014, pp. 997–1000. 31

[50] ——, "Neuron clustering for mitigating catastrophic forgetting in feedforward neural networks," in *Computational Intelligence in Dynamic and Uncertain Environments (CIDUE), 2014 IEEE Symposium on*. IEEE, 2014, pp. 62–68. 31

[51] ——, "Mitigating catastrophic forgetting in temporal difference learning with function approximation," in *Proceedings of the 2nd Multidisciplinary Conference on Reinforcement Learning and Decision Making*, 2015. 31

[52] E. Knudsen *et al.*, "Sensitive periods in the development of the brain and behavior," *Cognitive Neuroscience, Journal of*, vol. 16, no. 8, pp. 1412–1425, 2004. 33

[53] A. M. Saxe, J. L. McClelland, and S. Ganguli, "Exact solutions to the nonlinear dynamics of learning in deep linear neural networks," *arXiv preprint arXiv:1312.6120*, 2013. 33

[54] S. Van Der Walt, S. C. Colbert, and G. Varoquaux, "The numpy array: a structure for efficient numerical computation," *Computing in Science & Engineering*, vol. 13, no. 2, pp. 22–30, 2011. 40

[55] MATLAB, *version 8.4 (R2014B)*. Natick, Massachusetts: The MathWorks Inc., 2014. 40

[56] P. J. Bickel and E. Levina, "Regularized estimation of large covariance matrices," *The Annals of Statistics*, vol. 36, no. 1, pp. pp. 199–227, 2008. 41

[57] A. Rohde and A. B. Tsybakov, "Estimation of high-dimensional low-rank matrices," *Ann. Statist.*, vol. 39, no. 2, pp. 887–930, 04 2011. 41

[58] G. S. Babu and S. Suresh, "Meta-cognitive neural network for classification problems in a sequential learning framework," *Neurocomputing*, vol. 81, pp. 86 – 96, 2012. 41

[59] E. Manitsas, R. Singh, B. Pal, and G. Strbac, "Distribution system state estimation using an artificial neural network approach for pseudo measurement modeling," *IEEE Transactions on Power Systems*, vol. 27, no. 4, pp. 1888–1896, Nov 2012. 42

[60] S. Young, J. Lu, J. Holleman, and I. Arel, "On the impact of approximate computation in an analog destin architecture," *IEEE Transactions on Neural Networks and Learning Systems*, vol. 25, no. 5, pp. 934–946, May 2014. 42

[61] E. Theodorou, J. Buchli, and S. Schaal, "A generalized path integral control approach to reinforcement learning," *J. Mach. Learn. Res.*, vol. 11, pp. 3137–3181, Dec. 2010. 42

[62] T. Lancewicki, B. Goodrich, and I. Arel, "Sequential covariance-matrix estimation with application to mitigating catastrophic forgetting," in *Machine Learning and Applications and Workshops (ICMLA), 2015 14th International Conference on*. IEEE, 2015, p. to appear. 42

[63] R. S. Sutton and A. G. Barto, *Introduction to Reinforcement Learning*, 1st ed. Cambridge, MA, USA: MIT Press, 1998. 44

[64] J. Bergstra, D. Yamins, and D. Cox, "Making a science of model search: Hyperparameter optimization in hundreds of dimensions for vision architectures," in *Proceedings of The 30th International Conference on Machine Learning*, 2013, pp. 115–123. 48, 71, 82

[65] Y. LeCun and C. Cortes, "The mnist database of handwritten digits," 1998. 49, 50

[66] J. Rennie. 20 newsgroups dataset. [Online]. Available: http://people.csail.mit.edu/jrennie/20Newsgroups/ 49, 55

[67] J. Ramos, "Using tf-idf to determine word relevance in document queries," in *In Proceedings of the First Instructional Conference on Machine Learning.*, 2013. 55

[68] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay, "Scikit-learn: Machine learning in Python," *Journal of Machine Learning Research*, vol. 12, pp. 2825–2830, 2011. 56

[69] G. E. Hinton and R. R. Salakhutdinov, "Reducing the dimensionality of data with neural networks," *Science*, vol. 313, no. 5786, pp. 504–507, 2006. 59

[70] J. Fonollosa, S. Sheik, R. Huerta, and S. Marco, "Reservoir computing compensates slow response of chemosensor arrays exposed to fast varying gas

concentrations in continuous monitoring," *Sensors and Actuators B: Chemical*, vol. 215, pp. 618–629, 2015. 60

[71] P. M. Fishbane, S. G. Gasiorowicz, and S. T. Thornton, *Physics*. Prentice-Hall, 2005. 61

[72] E. Jones, T. Oliphant, P. Peterson *et al.*, "SciPy: Open source scientific tools for Python," 2001–, [Online; accessed 2014-10-18]. [Online]. Available: http://www.scipy.org/ 62

[73] R. Bellman, I. Glicksberg, and O. Gross, "On the'bang-bang'control problem," DTIC Document, Tech. Rep., 1955. 70

[74] R. V. Florian, "Correct equations for the dynamics of the cart-pole system," *Center for Cognitive and Neural Studies (Coneural), Romania*, 2007. 70

# Vita

Benjamin Frederick Goodrich was born in Knoxville, TN on Febuary 26th 1985. He was home schooled growing up, and took a strong personal interest in programming computers from a young age. In May 2006 he received an Associates of Applied Science degree in Computer Science and Information Technology from Pellissippi State Technical Community College. Then, he attended The University of Tennessee where he earned his Bachelor of Science degree in Computer Engineering in May 2011. He has worked internships at B&W Y-12, and at Broadcom in San Jose, CA. In 2011 he began his studies as a graduate student at the University of Tennessee where he focused on machine learning and reinforcement learning. In December 2015, he earned his PhD in computer engineering also from the University of Tennesssee.