On the Classification of Intermediate Range Missiles During Launch

by

Jordan Eckert

A thesis submitted to the Graduate Faculty of
Auburn University
in partial fulfillment of the
requirements for the Degree of
Master of Science

Auburn, Alabama
May 2nd, 2020

Keywords: Statistical Learning, Deep Learning, Neural Network, Support Vector Machine, Optimization, Missile Telemetry

Abstract

This thesis explores the comparisons and contrasts of different statistical method approaches to a missile system classification which is sufficiently rapid to be suitable for use during engagement. It is meant to extend the work published by Eckert et al. [9] by using simulated noise from common radar equations and running the dataset through an α-β filter. Additionally, this work expands on their work with the inclusion of another supervised learning technique, the support vector machine. The ultimate goal of this series of work is the rapid quantitative and statistically defensible descriptions of unknown missiles using the simulated telemetry data scheme during flight. The primary two supervised statistical learning methods explored in this thesis are Artificial Neural Networks (ANN) and Support Vector Machines (SVM) . Of these two different statistical learning techniques, SVM classifiers were found to be the optimal classification technique in the context of this framework; SVMs were able to correctly identify 100% of the testing dataset and guarantee a global optima.

Acknowledgments

The author would like to thank first his family for their years of unwavering support and myriad of encouragements throughout this academic journey. The author is thankful for Dr. Mark Carpenter as his guidance, support, and assistance throughout the duration of the project has been immeasurable. Additionally, the author is thankful to Dr. Van Wyk for his support and clear teaching of the optimization methods discussed within the paper. A special thanks to Dr. Roy Hartfield and Noel Cervantes for their efforts in simulating the telemetry data and being a great resource for any of the aerospace aspects of the paper. Lastly, a special thanks to the many members of Auburn's Department of Math and Statistics who spent time offering words of encouragement, edits, and bouncing ideas to better this work, especially Isabel Harris.

Table of Contents

List of Figures

List of Tables

List of Abbreviations

$(b_t/2)$	Tail Exposed Semi-Span (in)
C_{rt}	Tail Root Chord (in)
D_t	Diameter of Missile Throat (in)
L_n	Missile Nose Length (in)
R_b	Radius of Missile (in)
R_n	Radius of Missile Nose (in)
A	Antenna Effective Area
ANN	Artifical Neural Network
BFGS	Broyden-Fletcher-Goldfarb-Shanno
D	Diameter of Missile Body (in)
DoF	Degrees of Freedom
f	star-perforated grain fillet radius (in)
G	Gain Transmitted
L	Missile Length (in)
L-BFGS	Limited Memory Broyden-Fletcher-Goldfarb-Shanno
MSIC	Missile and Space Intelligence Center
N	Number of Star Points

nm	Nautical Mile(s)
PR	Power Received
PT	Power Transmitted
R	Range of Detection
RCS	Radar Cross Section
ri	Initial Grain Propellant Radius (in)
rp	Grain Propellant Radius (in)
SVM	Support Vector Machines
TE	Trailing Edge
VV	Vertical Polarizations
VV	Vertical Signal Sent, Vertical Signal Returned

Chapter 1

Introduction

1.1 Missile Geometry

Classifying missile trajectories can be categorized as a sub-goal of the object detection schema - a common theme among machine learning researchers. Object detection uses an object's set of motions to predict their next steps given a different set of conditions. The application of object detection in aerospace for the prediction of missile trajectories is a rapidly expanding field.

In this area of research, prior studies have proven that a missile can be classified during ascent phase [1] and classification of ballistic missiles can be achieved through the use of neural networks [2]. This paper attempts to improve on the literature by incorporating a full 6 degrees of freedom (DoF) simulation for a missile geometry more consistent with "truth data" as it includes noise from radar detection. Since ballistic missiles are usually neutralized during mid-phase and terminal phases, the timing focal point for our detection scheme will be during the launch phase. Thus, being able to classify the missiles in the shortest amount of time after detection to allow for deployment of proper neutralization techniques during later phases. In addition, this paper will serve as a way to compare and contrast different optimizing methods for neural networks and introduce support vector machines for missile classification.

The paper uses a selection of twelve different ballistic intermediate range solid motor-powered missile classes with a maximum range of approximately 350 nm . For each of these missile class variations, approximately 1,500 6-DoF fly-outs were developed using a 7^{th} - 8^{th}

order Runge-Kutta integrator for the equations of motion as formulate in Etkin [4]. The solid motor burn out follows the technical approach outline in Hartfield, et. al [5]. The aerodynamic loads are calculated using an in-house fast predictor tool similar to MISSILE-DATCOM. For the purposes of this paper, the missiles considered included a right circular perforated solid motor star grain as seen in Figure 1.1 with a profile geometry shape and structure similar to Figure 1.2.

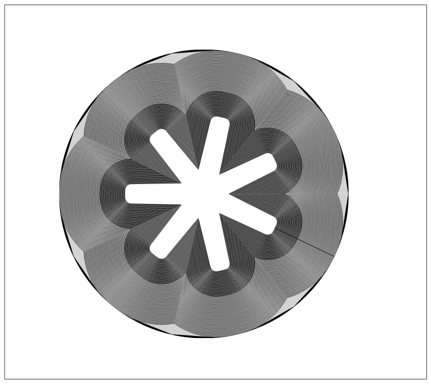

Figure 1.1: Cross Section of Star Grain with 7 Points

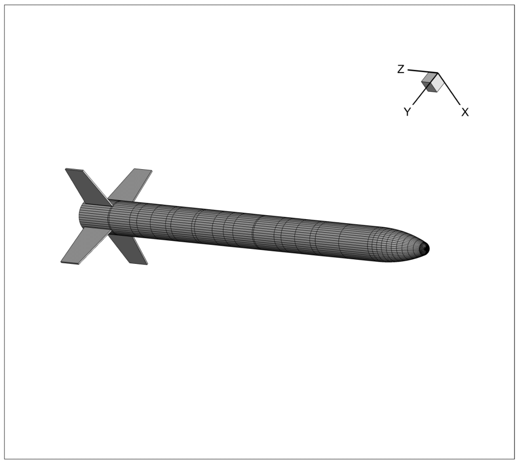

Figure 1.2: Nominal Missile Geometry Example

The geometric data for the nominal missile is summarized in Table 1. The combinations of values in bold are what make up the 12 cases generated for this paper. This configuration includes tail fins for stability during atmospheric flight, but no canards. Variations included increasing and decreasing the body diameter by 6.7% and changing fineness ration and throat diameter by similar amounts. The case of each missile is steel, and the solid propellant is PVC/AP/AL. The 7 pointed star grain is regressive initially for a boost-sustain profile.

PARAMETERS	**AVERAGE**
Nose Radius Ratio, R_n/R_b	.4
Nose Length Ratio L_n/D	2
$(rp+f)/R_b$	0.557
Star Inner Ratio, ri/rp	0.15667
Number of Star Points	7.2
Fillet Radius Ratio, f/rp	0.084
Epsilon, $(\pi * \epsilon)/N$	0.88138
Star Point Half Angle	10.03149°
Throat Diameter Ratio, D_t/D	**0.2943**, **0.30902**
Fineness Ratio, L/D	**14.33367**, **15.76704**
Missile Body Diameter (D)	**0.7**, **0.75**, **0.8**
Tail Semi-Span Ratio, $(b_t/2)/D$	1.4
Tail Root Chord Ratio, C_{rt}/D	1.132
Tail Taper Ratio	0.66
Tail LE Sweep Angle	10°
Tail Axial, TE/D	0.992
Fuel Type	PVC/AP/AL

Table 1.1: Key Parameters for the Nominal Missile Class

1.2 Noise Addition

The simulation will be taken as "truth" data; to this data, noise will be added to the signal to mimic a real-life missile firing. Afterward, an $\alpha - \beta$ filter will be applied to smooth the noisy data. Again, to mimic a real-life scenario.

The noise is added by simulating the power received by the antenna after a signal is sent and returned. To calculate this power, MSIC has provided radar polarization data from live fires which creates a view of the radar cross-section. The equation for the radar cross-section follows:

$$RCS = \sqrt{VV(REAL)^2 + VV(IMAGINARY)^2}\ \mathrm{ft}^2. \tag{1.1}$$

From the radar cross section, the power received comes from rearranging the original radar equation found in Skolnik [3]:

$$PR = \frac{PT * G * RCS * A}{(4\pi)^2 R}. \tag{1.2}$$

Noise for the polar coordinates (R, θ, ϕ) can then be generated by multiplying the power received by some Gaussian noise which ultimately is selected in a range that helps control error. The $\alpha - \beta$ filter is responsible for the conversion of these polar coordinates to Cartesian coordinates. The simulated noise matched MSIC's a priori knowledge of real-life fire scenarios.

1.3 Data Description

Provided are the simulated fly-outs using Table 1 results and noise within each class in this section. A total of 18,000 flights were simulated - 1,500 per each of the 12 classes. Figures 1.3 - 1.5 show a graphical representation for a randomly selected sample of 50 fly-outs truncated to the beginning of the launch phase, which is around 40 seconds of flight time. These graphics represent the 12 cases' range over time, their altitudes over time, and finally altitudes versus their ranges during the 40 seconds, respectively. Results from all 18,000 cases for their full duration of flight are displayed in Appendix A. Figures 1.3-1.5 are a good indications of how separable the data is. If the 12 different geometries show flight patterns that are too separated, the problem is not realistic and becomes uninteresting.

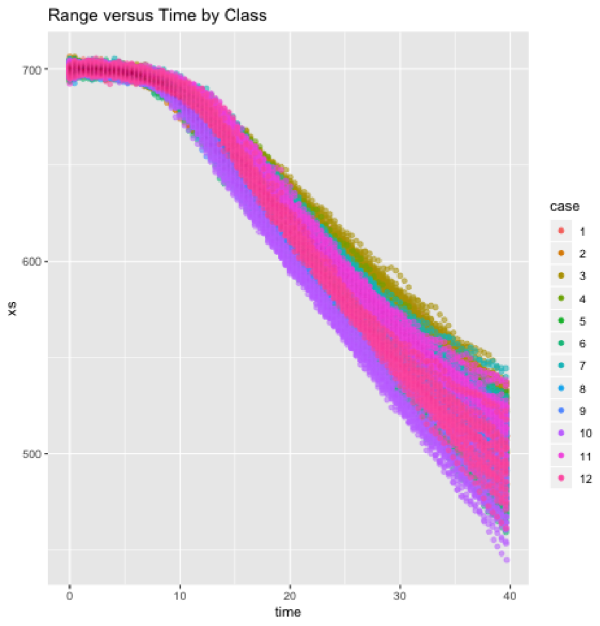

Figure 1.3: 40 Second Interval of Range versus Time Profiles for 50 Randomly Selected Missiles

Looking at the 40-second interval profile from Figures 1.3 - 1.5 we see that there is hardly any separability at this beginning stages. In fact, many of the classes now almost perfectly overlap. Thus, the simulation mimics what we would expect to find using live fire data, and creates a unique, interesting problem.

It's worth noting that most of the classes begin separation during the later stages of flight; this concern will not play a factor during modeling as we are only focused on the beginning seconds of the missile's flight trajectory. The hope moving forward, therefore, is that the statistical learning techniques will be able to determine one pattern's variance from other patterns and make accurate classifications of this noisy data. It is important to note that the visualizations presented in the Appendix figures dictate the telemetry of the rockets from the simulated launch to landing. One of the explicit goals of the paper was to achieve classification as quickly as possible after radar detection for neutralization.

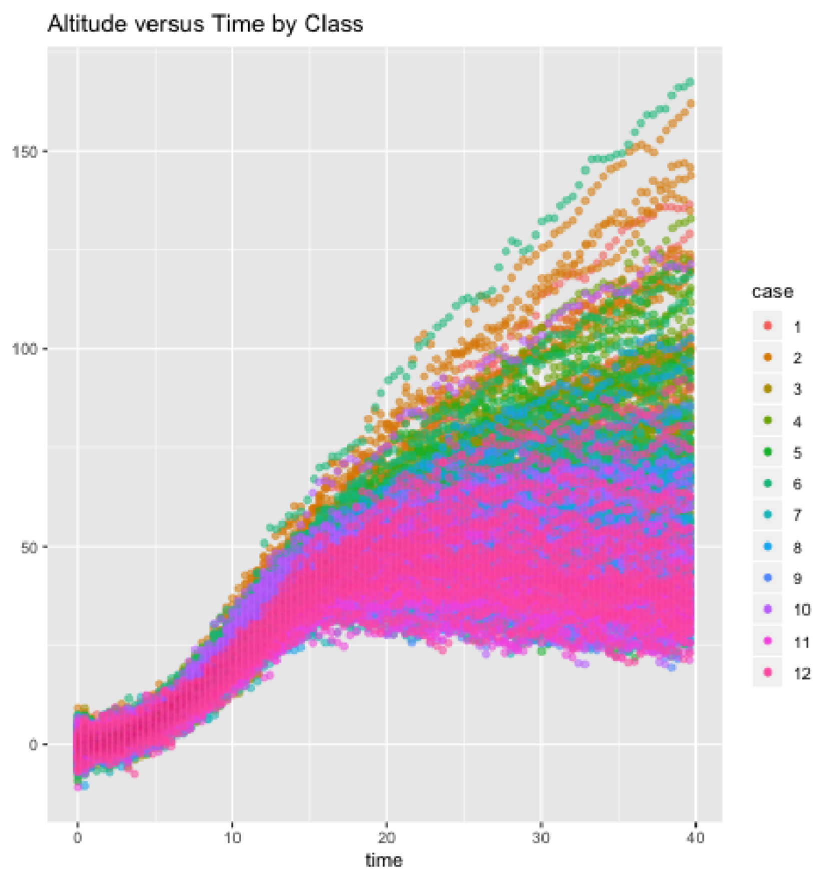

Figure 1.4: 40 Second Interval of Altitude versus Time Profiles for 50 Randomly Selected Missiles

Ergo, the Appendix figures encompass more data than is expected to be used in the modeling. After verifying that the simulation is a success, the next step is processing the shape of the dataframe before modeling. Many statistical learning methods process the entire data vector at once [6], manipulating the vector can help us achieve a better classification accuracy. Currently, the data is a vector containing the following: an ordinal, categorical ID for a missile's case, a point in 3-dimensional space where the rocket is at detection time, and the detection time itself. The vector at time t, would look like: $[Case\#,\ (X_t, Y_t, Z_t),\ t]$. In an effort to increase the input basis (the benefits of doing so are further explored in the next chapter), the data is vectorized so that the new input vector has the same case ID as before, but now includes points X_t, Y_t, and Z_t that have accumulated up to time, t.

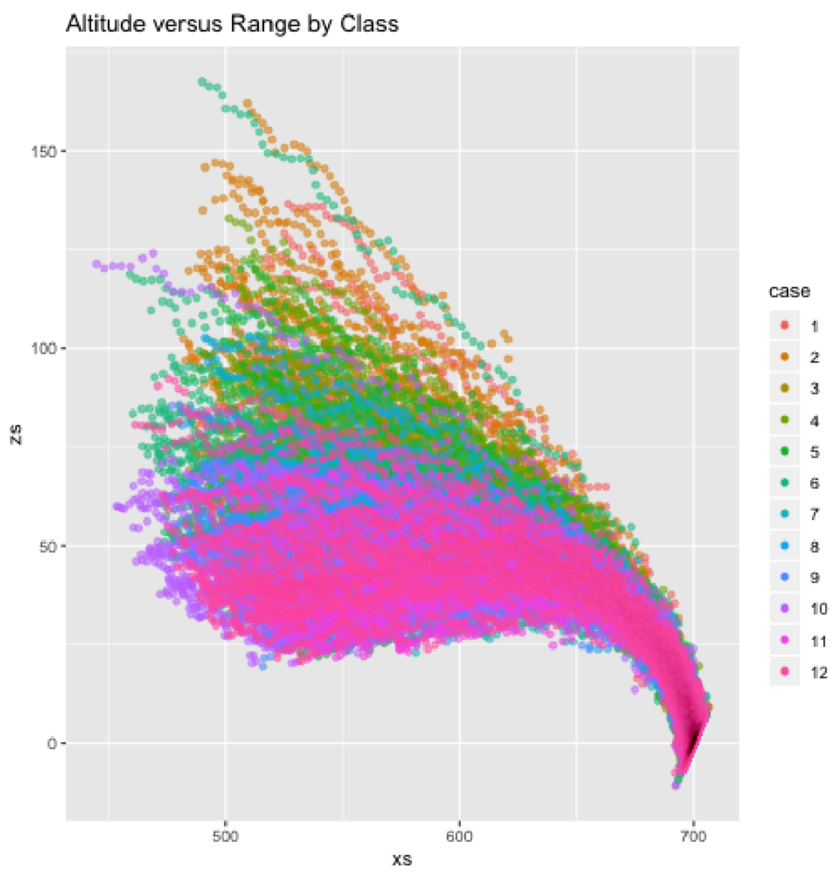

Figure 1.5: 40 Second Interval of Altitude versus Range Profiles for 50 Randomly Selected Missiles

Thus, for any time such that $t > 1$, the input vector becomes $[Case\#,\ X_1, \ldots, X_t,\ Y_1, \ldots, Y_t,\ Z_1, \ldots, Z_t]$. It is important to note here that X_1, Y_1, and Z_1 do not represent the position at launch but rather after 10 seconds of flight time. In reality, radar detection will be variable, which is why such an extreme amount of time was truncated. Extremely conservative estimates will assume that it takes 10 seconds or less for radar detection. This accounts for any delayed detection time by the radar and keeps the vectors' lengths fixed for analysis.

Time was not measured in even intervals. After processing the data, 30 seconds of flight data had a time value of $t = 50$. In this way, the simulated radar detection times match the real life radar detection by not having an consistent, evenly measured interval. The corresponding input vector for a single 30 seconds of flight data is: $[Case\#,\ X_1, \ldots, X_{50},\ Y_1, \ldots, Y_{50},$

$Z_1, \ldots, Z_{50}]$. There exists a balance between accuracy and time of detection, as the time of detection increases, so does the accuracy but it also cuts into potential time for neutralization. 30 seconds of data seems to be a "sweet spot" in terms of this trade off, and was used in the initial analysis of the problem [9]. The process of vectorizing the dataframe to the new form was done using a macro in popular statistics software, SAS.

The data processing steps defined above do achieve the goal of increasing the input basis possibilities, but have the drawback of losing the time element the previous input vector possessed. The original pre-vectorized flight telemetry can be modeled as a function of time, i.e. say the flight telemetry before processing can be described by function $f(t)$. Then the following equation for velocity can be trivially shown to be:

$$v(t) = f'(t). \tag{1.3}$$

Since the first 10 seconds are being truncated and we are only using a maximum of 30 seconds of flight data the function $v(t)$ is defined and exists for for all points in this interval being modeled. The input basis can be expanded, with marginally little effort, to include these velocities in the X, Y, and Z directions: $[V_{1x}, \ldots, V_{tx}, V_{1y}, \ldots, V_{ty}, V_{1z}, \ldots, V_{tz}]$ Since velocity is a function of time, the inclusion of the velocities to the input vector allows the models to "recapture" some of the time element lost by the vectorizing process and as an added benefit further increases the size of the input basis.

Initially, there was a discussion to also add acceleration since the second derivative of the initial telemetry flight data could also be defined in our window of modeling. However, under consultation it became clear that the acceleration data would be too chaotic and noisy to capture when using radar in the field. Since the goal of this paper is to develop these methodologies for use during engagement, and acceleration would not be available during such a time, it was left out in the modeling process. This speaks to the limitations of using radar.

Chapter 2

Neural Networks

2.1 Introduction to Neural Networks

Every neural network is composed of three elements - an input layer, atleast one hidden layer, and an output layer. The central idea of a neural network is to use a nonlinear function on inputs to model a response [7]. The underlying goal is to find an equation based on input data that can offer an accurate prediction on new, realistic data.

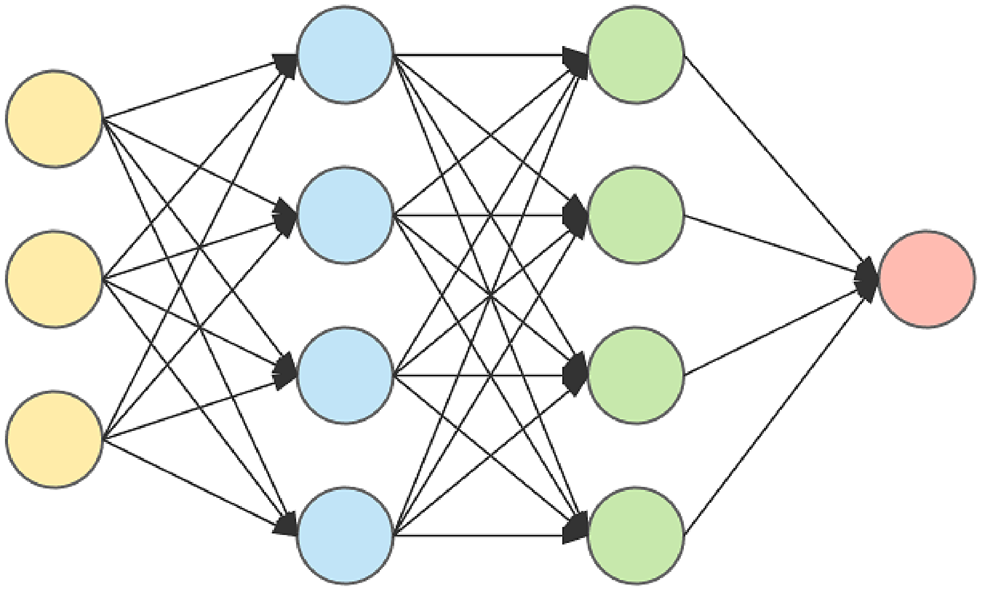

Figure 2.1: Graphical Representation of an Example Neural Network

Suppose we have a p-component vector of inputs $X = [x_1 ... x_p]'$. A large class of models can be formed by nonlinear functions using these combinations as inputs to make up a mathematical basis. As a direct result, models have the flexibility to be generalized for any function in $\mathcal{R}^p$ [7]. Equation (2.1) shows how the dot products of x_1 and x_2 can be expanded to a 2^{nd} order degree polynomial very simply [11]:

$$(x_1 \cdot x_2) = \frac{[(x_1 + x_2)^2 - (x_1 - x_2)^2]}{4} \tag{2.1}$$

Each layer is composed of units called neurons or nodes (in Figure 1, a node is represented by one of the colored circles). Each neuron outputs a real number which is then passed along to the next layer. Passing it from layer to layer forms a repeated application of a simple, chosen nonlinear function called the "activation function" (denoted as $\sigma(x)$) [11]. As the number of hidden layers increases, the network is termed as having more "depth." Hence the colloquial term, "deep learning". Choosing the correct activation function is an important aspect of the accuracy of the network since incorrect choices can lead to poor testing results. Increases in the depth and complexity of the network cause activation at each layer to become computationally more expensive.

The application of the activation function on the weighted combination of the previous layers' values needs to be done in component-wise fashion; so that for every $x \epsilon \mathcal{R}^m$, $\sigma : \mathcal{R}^m \rightarrow \mathcal{R}^m$ the following always holds:

$$(\sigma(x))_i = \sigma(x_i) \tag{2.2}$$

where m is the size of the basis represented by $m = 1, 2, ..., M$ p-vectors of inputs [8].

The next step in the network involves the previous weighted combinations of inputs (α_i) and a bias term (α_0) being fed into the current layer's activation function. The results from each of the previous layer's activations are gathered and then used in a "two-stage" regression. Once the initial basis goes through all the hidden layers a final transformation is completed by the output function to get the results of a single forward pass [8]. Mathematically, this process is represented by Equations (2.3)-(2.5):

$$Z_m = \sigma(\alpha_{0m} + \alpha_m X); \;\; m = 1, ..., M \tag{2.3}$$

$$T_k = \beta_{0k} + \beta_k^T Z; \;\; k = 1, ..., K \tag{2.4}$$

$$g_k(T) = Y_k; \;\; k = 1, ..., K \tag{2.5}$$

where $Z = (Z_1, ..., Z_m)$, $T = (T_1, ..., T_k)$, and the subscript k are dependent on the network modeling a regression or classification problem. For regression, $K = 1$ is used in the output layer to get a single numerical value; otherwise, it would denote the k^{th} class of a K-sized classification problem; m has been defined earlier [7]. For the purpose of our classification networks $K = 12$.

Colloquially, these "hidden layers" get their name from the fact that each Z_m is indirectly observed [7]. Ultimately, Z_m acts the same as the basis expansion discussed earlier in the section of original inputs X. Another way to increase the networks' complexities is to add neurons in each layer because additional neurons expand the potential basis that is fed to the next layer.

Mathematically, if we collect a vector of all the previous real numbers produced by the neurons in one layer (denote this vector as V) then we can show that the vector of inputs for the next layer has the form:

$$Z_m = \sigma(\alpha_{0m} + \sum \alpha_m V) \Rightarrow Z = \sigma(b + WV) \tag{2.6}$$

where W is a matrix representing the weights ($\sum \alpha_m$) and introduced biases (b) for the current hidden layer. The dimensions of W are $q \times p$ where q is the number of neurons in the current layer and p is the number of neurons from previous layer. The bias vector is the size of the number of neurons on the current layer that are adding their own unique biases [8].

Assume, as an example, there is a neural network with two hidden layers with the size of the layers being 2 and 3, respectively. The first iteration of the activation function's application occurs on the original X_i's from the input layer [7]:

$$Z_1 = \sigma(b^{[1]} + W^{[1]}X). \tag{2.7}$$

The output is then fed into layer two, which then has an output of the form [7]:

$$\begin{aligned} &Z_2 = \sigma(b^{[2]} + W^{[2]}Z_1) \Rightarrow \\ &\sigma(b^{[2]} + W^{[2]}\sigma(b^{[1]} + W^{[1]}X)) \,\epsilon\, \mathcal{R}^2. \end{aligned} \tag{2.8}$$

Afterwards, the 3 neurons in the next layer receive the output from each of the second layer neurons, Z_2. The output of Z_2 is in $\mathcal{R}^2$ meaning $W^{[3]}$ is a matrix of size $(3, 2)$ and $W^{[3]} \,\epsilon\, \mathcal{R}^3$ with a corresponding bias vector that is also $b^{[3]} \,\epsilon\, \mathcal{R}^3$. The output from this second hidden layer would be iterative:

$$\begin{aligned} &Z_3 = \sigma(b^{[3]} + W^{[3]}Z_2) \Rightarrow \\ &\sigma(b^{[3]} + W^{[3]}\sigma(b^{[2]} + W^{[2]}\sigma(b^{[1]} + W^{[1]}X)) \,\epsilon\, \mathcal{R}^3. \end{aligned} \tag{2.9}$$

These outputs become the inputs into a regression for T_k to find the corresponding β_k weights [8].

The output function, $g_k(T)$, is the final transformation of the previous functions' outputs. For a regression problem, this is just the identity function, $g_k(T) = T_k$. At the time of this writing, classification problems commonly use the softmax function [7]:

$$g_k(T) = \frac{e^{T_k}}{\sum_{l=1}^{K} e^{T_l}}. \tag{2.10}$$

The function acts the same way a multinomial logit link would, creating a probability for each of the K classes that in turn the final linear combination classifies in a specific k^{th} class. Ultimately, $g_k(T)$ is a mapping from the dimension of the input space to the output space where $\mathcal{R}^M \to \mathcal{R}^K$. This "forward pass" creates a preliminary prediction for the network.

To recap so far, neural networks have the following primary characteristics [10]:

1. At any stage during the forward pass of a neural network, there exists an activation function value for each neuron.

2. Each neuron is connected to each other neuron and these connections are what determine the input for the next neuron. Each of these connections is weighted.

3. At a neuron, an activation function is applied to all the incoming inputs to generate a new input for neurons in subsequent hidden layers or the output layer.

At this point the machine has a preliminary prediction; it is important to assess how "well" it did modeling. The network evaluates where mistakes were made and uses a corrected weight matrix W on subsequent passes, learning from its past mistakes - which is where the nomenclature "statistical learning" is derived from. This leads to the last primary characteristic:

4. A learning algorithm is used to adjust the weights between neurons when given an input–output pairing.

The learning process is referred to as "backpropagation." Any backpropagation algorithm updates the weights one layer at a time going from right to left. Given enough time any neural network will completely learn the patterns from all the data fed into it, overfitting the model to the specified data. Typically modeling is broken into two steps to avoid such overfitting - a training step and a testing step. The data is split such that a portion will be used specifically for training and another portion will be used only for testing. "Epochs" are an arbitrarily defined hyperparameter representing a time to cutoff training. The epoch hyperparameter is often chosen before training, but it can be set to stop during training after certain criteria are met. An epoch is generally seen as "one pass over the entire dataset". The network will formulate a model during the training step, and afterward, all the weights and thresholds will be optimized to reduce the error defined by the difference between the output guessed during network training and the correct output provided [13]. The model is then tested on the fresh holdout data as a way to measure potential overfitting that might have occurred.

If we think of all the unknown weight parameters as a vector θ then we can define a risk function, $R(\theta)$. The risk function is dependent on whether the model is a regression or classification model. Objectively, the goal is to minimize $R(\theta)$ as accurately and efficiently as possible. The next few sections show effective ways of minimizing $R(\theta)$. The network learns tasks from examples over the span of many epochs. As each epoch finishes, the after-most weights are updated, which, in turn, recursively updates the previous weights. The optimization will continue to learn as the number of epochs increases, trying to converge to a global minima of the objective risk function.

2.2 Optimization Algorithms for Neural Networks

As discussed in the previous section, the neural network functions as a sort of mapping from the inputs to either the classes of a classification or a numerical value for regression.

Simply put, the network "function", $h(*, w)$, acts as a sort of "black-box" machine turning inputs into outputs [11]:

$$x_n \,\epsilon\, \mathcal{R}^{d_0} \rightarrow h(x_n, w) \rightarrow y \,\epsilon\, \mathcal{R}^{d_y}. \tag{2.11}$$

Define an arbitrary network with J layers. Then for each layer $j = 1, \dots, J$ the weights and biases for the j^{th} layer become:

$$w_j \,\epsilon\, \mathcal{R}^{d_j \times d_{j-1}} \tag{2.12}$$

$$b_j \,\epsilon\, \mathcal{R}^{d_j} \tag{2.13}$$

and as we advance through the layers, the number of total parameters that become necessary for estimation during the training step increases.

Each layer takes in the weights and biases of the previous layer as an input, meaning the total number of parameters necessary to be updated is a cumulative sum of the previous layers up to the output layer. For layer $i \,\epsilon\, \{1, \dots, J\}$, the total number of parameters is represented by Equation (2.14):

$$d_i = \sum_{j=1}^{i} (d_j d_{j-1} + d_j). \tag{2.14}$$

The loss function, $l(*)$, gives information for any example input-output pair for the network. If there exists no difference between the guessed and correct output, then the loss is zero. If the difference exists, then measurement error exists as well. With the choice of a loss function we can set up our optimization problem minimizing the average loss across all examples [11], done below:

$$\min_{w \,\epsilon\, \mathcal{R}^d} R(w) = \min_{w \,\epsilon\, \mathcal{R}^d} [\frac{1}{n} \sum_{i=1}^{n} (l(h(x_i, w), y_i)] \tag{2.15}$$

Using the "machine" analogy in Equation (2.11) the goal for optimization of neural networks is to generate a sequence of points $(h(x_i, w), y_i)$ that converge to the minimization of our objective risk function, $R(w)$, as the number of iterations of the neural network, denoted k increases $(k \rightarrow \infty)$ [14]. If the objective function is convex, the optimization will produce a global minima; however, most objective loss functions in the application of neural networks are not strictly convex - if they are convex at all. Optimization can therefore lead to answers that are not the true global minima if the function is not strictly convex. If our objective function $R(w)$ is continually differentiable then the optimization problem is a smooth optimization problem.

2.2.1 Stochastic Gradient Descent

Approximate solutions to the optimization problem have been approached using gradient descent. The gradient of the average loss is computed using chain rule differentiation; however, the computation of the gradient is expensive. If using the method of "steepest descent", we initialize a matrix, w_0, to determine an optimal α_k, and update the weight matrix at the k^{th} iteration using the gradient:

$$w_{k+1} = w_k - \alpha_k \nabla R(w_k) \tag{2.16}$$

where α_k is the "step-size" for the optimization process. If α_k satisfies the Wolfe conditions,

$$\begin{aligned} &||\nabla R(w_k)|| \rightarrow 0;\ k \rightarrow \infty \qquad (2.17)\\ &\Rightarrow w_{k+1} = w_k;\ k \rightarrow \infty \end{aligned}$$

then computing $\nabla R(w_k)$ at each step of optimization the steepest descent converges geometrically to Equation (2.18). As $\rho \to 1$ convergence speed increases [11]:

$$\begin{aligned} &\frac{||w_{k+1} - w^*||}{||w_k - w^*||} \leq \rho \Rightarrow \\ &||w_k - w^*|| \leq c\rho^k \Rightarrow \\ &\mathcal{O}(\rho^k). \end{aligned} \tag{2.18}$$

Proving that optimizing $\nabla R(w)$ using the entire dataframe is exhaustive computationally. Instead of using the whole gradient at once, stochastic gradient descent use n-discrete optimizations by approximating $R_n(w)$ from the n-sums of the averages of the objective function $f(w, \xi_i)$ where ξ_i represents a batch of observations from the truth distribution. In true stochastic gradient descent methods, this ξ_i value would be a single observations, not a batch. Using a "mini-batch" approach creates a compromise between the full gradient and single point approaches in both accuracy and time complexity. Since this random variable only will be used to generate the stochastic direction [11], the following analysis will hold for batch approaches.

Analysis of Stochastic Gradient Methods

The complexity of selected objective functions for neural network tends to create quasi-convex to nonconvex optimization problems. Therefore, this analysis will be based primarily on the use of a general objective function F instead of using one that is strictly convex. The algorithm for the stochastic gradient method follows the iterative process [11]:

Algorithm 1 SGD Algorithm:

```
1: Initialization Choose a starting value for $w_1$
2: for $k = 1, 2, \ldots, K$ do {K represents the number of iterations until convergence}
3:     Generate a realization of the random variable $\xi_k$
4:     Compute the stochastic vector gradient {The gradient will be expressed as $g(w_k, \xi_k)$}
5:     Choose a step-size $\alpha_k > 0$
6:     $w_{k+1} \leftarrow w_k - \alpha_k g(w_k, \xi_k)$
7: end for
```

In order for convergence to happen, two fundamental assumptions must be made. This assumptions will be addressed and derived from the fundamental **Lemma 2.1** and **Lemma 2.2** [11]:

Lemma 2.1 (Lipschitz-continuous objective gradients): The objective function is continuously differentiable and the gradient function ($\nabla F : \mathcal{R}^d \rightarrow \mathcal{R}^d$), is Lipschitz continuous with the Lipschitz constant $L > 0$ such that

$$||\nabla F(w) - \nabla F(\bar{w})||_2 \leq L||w - \bar{w}||_2 \text{ for all } \{w, \bar{w}\} \subset \mathcal{R}^d. \tag{2.19}$$

The assumption in **Lemma 2.1** is that the gradient of F is not an exploding gradient, meaning it will not rapidly change. If the gradient is exploding, we could create an upper bound to the gradient such that it will no longer be exploding. This assumption is crucial for convergence analysis. The second fundamental lemma is:

Lemma 2.2: If the gradient is Lipschitz continuous and does not rapidly change, then the iterates of SGD Algorithm satisfy the inequality for all $k \, \epsilon \, \mathcal{R}^p$

$$E[F(w_{k+1})|\xi_k] - F(w_k) \leq -\alpha_k \nabla F(w_k)^T E[g(w_k,\xi_k)|\xi_k] + \frac{1}{2}\alpha_k^2 L E[||g(w_k,\xi_k)||_2^2|\xi_k]. \quad (2.20)$$

Lemma 2.2 bounds above the expected decrease in the objective function for the k^{th} step. The convergence is guaranteed as long this bound is a deterministic quantity that asymptotically ensures descent in F. To ensure that this bound is deterministic, we have further assumptions covered in the following [11]:

Lemma 2.3 (First and Second Moment Limits): The objective function and SGD Algorithm satisfy:

a.) The sequence of iterates $\{w_k\}$ is contained in an open set over which F is bounded below by a scalar.

b.) there exists a scalar such that $\mu_G \geq \mu > 0$ such that for all $k \,\epsilon\, \mathcal{R}^p$,

$$\nabla F(w_k)^T E_{\xi_k}[g(w_k,\xi_k)] \geq \mu ||\nabla F(w_k)||_2^2 \quad (2.21)$$

and

$$||E_{\xi_k}[g(w_k,\xi_k)]||_2 \leq \mu_G ||\nabla F(w_k)||_2^2. \quad (2.22)$$

c.) Lastly, we will assume there exists scalars $M \geq 0$ and $M_V \geq 0$ such that,

$$V_{\xi_k}[g(w_k,\xi_k)] \leq M + M_V ||\nabla F(w_k)||_2^2 \quad (2.23)$$

Lemma 2.3.b shows that in expectation, the negative of the gradient vector is a direction of sufficient descent for F from w_k. The norm is comparable to the norm of the gradient. By bounding the variance of the gradient in **Lemma 2.3.c**, it will allow the variance to always be nonzero at any point for F where the descent might become stationary.

The combination of the **Lemma 2.1** and **Lemma 2.3** proves that no matter how w_k is iterated, the optimization process is Markovian. Meaning the current iteration does not depending on any past iterates beyond the previous step [11].

Since our step-size hyperparameter will be fixed before training, I will focus primarily on the analysis of using a fixed step-size ($\bar{\alpha}$) for general objective stochastic gradient descent, i.e. $\alpha_k = \bar{\alpha}$ for all k that satisfy:

$$0 < \bar{\alpha} < \frac{\mu}{LM_G}. \tag{2.24}$$

Under the stated assumptions, the expected sum-of-squares and average-squared gradients of F corresponding to the SG iterates satisfy:

$$E[\frac{1}{K}\sum_{k=1}^{K}||\nabla F(w_k)||_2^2] \leq \frac{\bar{\alpha}LM}{\mu} + \frac{2(F(w_1) - F_{inf})}{K\mu\bar{\alpha}}. \tag{2.25}$$

This is easily shown by taking the expectation of the total optimality gap under the new fixed step-size value, $\bar{\alpha}$.

$$E[F(w_{k+1})] - E[F(w_k)] \leq -\frac{1}{2}\mu\bar{\alpha}E[||\nabla F(w_k)||_2^2] + \frac{1}{2}\bar{\alpha}^2 LM. \tag{2.26}$$

After summing both sides of the inequality for $k \,\epsilon\, \{1, \ldots, K\}$ and recalling **Lemma 2.3.a**, we can define F_{inf} as the bounding scalar and the equation becomes:

$$F_{inf} - F(w_1) \leq E[F(w_{k+1})] - F(w_1) \leq -\frac{1}{2}\mu\bar{\alpha}\sum_{k=1}^{K}E[||\nabla F(w_k)||_2^2] + \frac{1}{2}K\bar{\alpha}^2 LM. \tag{2.27}$$

Rearranging and dividing by K, yields the desired results:

$$E[\frac{1}{K}\sum_{k=1}^{K}||\nabla F(w_k)||_2^2] \leq \frac{\bar{\alpha}LM}{\mu} + \frac{2(F(w_1) - F_{inf})}{K\mu\bar{\alpha}} \tag{2.28}$$

therefore proving that the average-squared gradients of F corresponding to the stochastic gradient iterates satisfies the above inequality. When M = 0, then no noise is present and the strictly convex case is recovered [11].

The stochastic gradient method will end up spending more time in regions were the objective has a small gradient as a result of K being in the denominator of Equation (2.25). As $K \to \infty$, the new bounding equation becomes:

$$E[\frac{1}{K}\sum_{k=1}^{K}||\nabla F(w_k)||_2^2] \leq \frac{\bar{\alpha}LM}{\mu}. \tag{2.29}$$

By reducing the fixed step-size, the average norm of the gradients can be bound to be arbitrarily small. However, doing so increases the time complexity [11].

2.2.2 Adam

Adam optimization is an improvement to the stochastic gradient descent optimization method. The foundations of the Adam optimizer comes from its efficient stochastic optimization relying on the first order gradients. However, instead of using just the gradient Adam incorporates adaptive moment estimation from estimates of the gradient to the objective function's first and second moments [17]. In the context of neural network the gradient of the cost function is stochastically optimized; this gradient is a random variable under this optimization scheme.

Adam is an adaptive learning rate method, meaning that the algorithm will produce individual learning rates for the parameters for a bounded step-size. These parameter updates are invariant to the rescaling of the gradient [17].

In the following algorithm g_k^2 represents the element-wise square $g_k \bigodot g_k$. All operations in Algorithm 4 are element-wise, and β_i^t represents β_i raised to the power t.

Algorithm 2 Adam

```
Require: $\alpha$ (step-size)
Require: $\beta_1, \beta_2 \,\epsilon\, [0, 1)$ (Exponential decay rates for moment estimates
Require: $R_n(w)$; $w_0$; $m_0 = 0$ ($1^{st}$ moment vector); $v_0 = 0$ ($2^{nd}$ moment vector); $k = 0$
1: while $w_k$ not converged do
2:     $k \leftarrow k + 1$
3:     $g_k \leftarrow \nabla R_n(w_{k-1})$
4:     $m_k \leftarrow \beta_1 \cdot m_{k-1} + (1 - \beta_1) g_k$
5:     $v_k \leftarrow \beta_2 \cdot v_{k-1} + (1 - \beta_2) g_k^2$
6:     $\hat{m_k} \leftarrow \frac{m_k}{1 - \beta_1^k}$ (Bias-corrected first moment estimate)
7:     $\hat{v_k} \leftarrow \frac{v_k}{1 - \beta_2^k}$ (Bias-corrected second moment estimate)
8:     $w_k \leftarrow w_{k-1} - \alpha \frac{\hat{m}_k}{\sqrt{\hat{v_k}} + \epsilon}$
9: end while
```

Looking at lines, 4 and 5 of the algorithm, we calculate the moving averages m_k and v_k respectively where the β_i hyperparameter control the exponential decay rates of these two moving averages. Ideally, these moving averages will be unbiased estimators for our first and second moments. For m_1, m_2, m_3 the following calculations are true:

$$m_1 = (1 - \beta_1)g_1 \tag{2.30}$$

$$m_2 = \beta_1(1 - \beta_1)g_1 + (1 - \beta_1)g_2 \tag{2.31}$$

$$m_3 = \beta_1^2(1 - \beta_1)g_1 + \beta_1(1 - \beta_1)g_2 + (1 - \beta_1)g_3. \tag{2.32}$$

By following this pattern, there exists a formula for the moving averages. The expectation of m can be found at any value of t:

$$\begin{aligned} E[m_t] &= E[(1 - \beta_1)\Sigma_{i=1}^{k} \beta_1^{k-i} g_i] \\ E[m_t] &= E[g_i](1 - \beta_1)\Sigma\beta_i^{t-i} + \xi \\ E[m_t] &= E[g_i](1 - \beta_1^k) + \xi. \end{aligned} \tag{2.33}$$

Equation (2.33) uses g_i as an approximation of g_t leading to the error term ξ where $\xi = 0$ if the moment is stationary. Optimal tuning of the β_1 hyperparameter keeps this error small, assigning lower weights to past gradients. Through simulation, the algorithm is shown to have $\mathcal{O}(\sqrt{T})$ [17]. Similar calculations can be done for v_t.

This bias tends to have not much effect after a long period of training. But, it should be noted that these estimates are biased towards zero during the beginning due to the initialization. Lines 6 and 7 of the algorithm are added to correct the bias for the first/second moment estimates.

While step-size is a tune-able hyperparameter, Adam's update creates an effective step-size bounded by α that establishes a "*trust region*" around the current parameter [17]. Current gradient estimates beyond this region do not provide sufficient information and are thus ignored in the update. The effective step-size bounded such by α is:

$$\Delta_k = \frac{\alpha \cdot \hat{m_k}}{\sqrt{\hat{v_k}}} \tag{2.34}$$

with the boundary for Δ_k:

$$|\Delta_k| \leq \frac{\alpha \cdot \hat{m_k}}{\sqrt{\hat{v_k}}}. \tag{2.35}$$

In most common scenarios, the effective step-size will not reach the specified tuned hyperparameter as $\frac{\hat{m_k}}{\sqrt{\hat{v_k}}} \approx \pm 1$ since $\frac{E[g]}{\sqrt{E[g^2]}} \leq 1 \Rightarrow |\Delta_k| < \alpha$ [17].

Chapter 3

Support Vector Machines

3.1 Introduction to Support Vector Machines

Support vectors have become a defacto “Swiss-army-knife” in statistical learning for solving multidimensional function classification problems, and for good reason. Support vector machines “have proved to be more accurate predictors in classification and regression then neural nets…” [22]. SVMs had a strong theoretical foundation before their construction for application purposes. This is reversed from neural networks, which found their uses in applications only to have the theory behind their workings filled in later. [21]

3.2 Support Vector Classifier

Before I begin in earnest with the discussion of support vector machines, it is necessary to understand support vector classifiers. Discussions about the two-dimensional support vector classifier generalize easily to higher dimensions.

Consider the simple case classifying a series of inputs into one of two separable classes (this will be generalized to many classes later). That is consider $x \,\epsilon\, \mathcal{R}^p$ and $y \,\epsilon\, [-1, 1]$ with no overlap in the y_i. We can define a set of N training observations then such that $(x_1, y_1) \ldots (x_N, y_N)$. The idea behind a support vector classifier is to define a $p - 1$ dimensional hyperplane [20]:

$$\{x : f(x) \equiv x^T\beta + \beta_0 = 0\}. \tag{3.1}$$

The hyperplane divides the p-dimensional input space into two halves. For our feature space, the hyperplane will be a line that separates the two classes $\{1, -1\}$. We can use this to construct our basic classification rule [7], denoted $G(x)$.

$$\text{Class A:}\{x : f(x) = x^T\beta + \beta_0 < 0\} \tag{3.2}$$

$$\text{Class B:}\{x : f(x) = x^T\beta + \beta_0 > 0\} \tag{3.3}$$

$$\Rightarrow G(x) = sign[x^T\beta + \beta_0]. \tag{3.4}$$

If this hyperplane that is able to perfectly separate the two classes exists then it is called a "separating hyperplane."

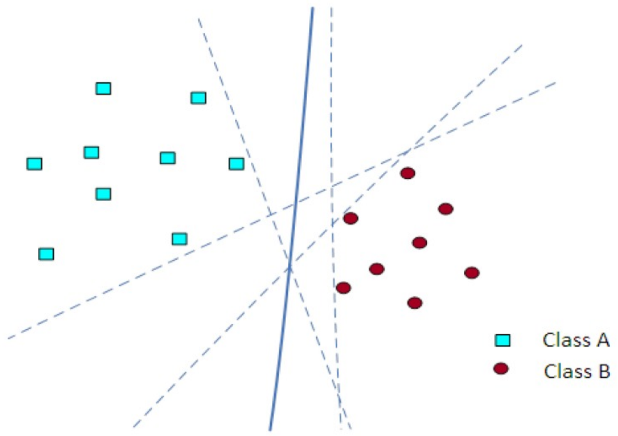

Figure 3.1: Example of Potential Separating Hyperplanes for Simple Case [24]

Thus, for some input vector (x^*), we can plug it into our classification rule $G(x^*)$. That vector belongs to either Group A or B depending on if $G(x^*)$ is positive or negative. Depending on the chosen x^* a large class of potential separating hyperplanes can exist, and therefore can be used as classifiers. See Figure 3.1 for the graphical example of the potential separating hyperplanes of the simple case.

From the geometric definition of a hyperplane, Equation (3.1) gives a distance from some point x to the defined hyperplane $f(x)$. The assumed separability of the classes means that a margin function

$$f(x) = x^T\beta + \beta_0 \text{ exists with } y_i f(x_i) > 0 \;\forall\; i. \tag{3.5}$$

Equation (3.5) will be used for determining the optimal hyperplane. Since the separability of the classes is assumed, the optimization problem is defined:

$$\begin{aligned} &\max_{\beta,\beta_0,||\beta||=1} \text{M} \;\; \text{subject to} \\ &y_i(x_i^T\beta + \beta_0) \geq M, i = 1, \dots, N \end{aligned} \tag{3.6}$$

where M creates a band that is M units away from hyperplane on either side. $2M$ is labeled as the margin [7]. Using $M = \frac{1}{||\beta||}$ where $M > 0$, the optimization is rewritten in terms of β as:

$$\begin{aligned} &\max_{\beta,\beta_0} ||\beta|| \;\; \text{subject to} \\ &y_i f(x_i^T\beta + \beta_0) \geq 1, i = 1, \dots, N. \end{aligned} \tag{3.7}$$

The margin is a measure of distance from each training observation perpendicular to the separating hyperplane. The optimal classifying hyperplane will be the one that has the maximum margin defined by the above optimization problem; that is the hyperplane that has the farthest minimum distance to the training observations [18]. This is a convex problem with a linear inequality constraint.

When the data is perfectly linearly separable, there will always be points that lie on the margin's boundary. There exist vectors from these points to the optimal separating hyperplane that are known as "support vectors" as they are "supporting" the hyperplane.

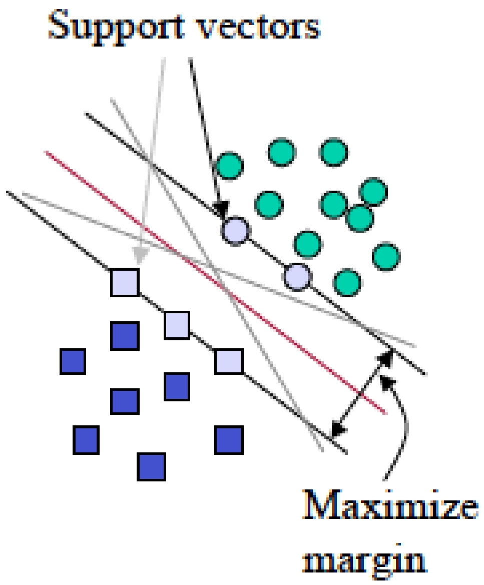

Figure 3.2: Graphical Representation of Support Vectors [19]

Moving these vectors slightly would directly result in a slight change the hyperplane. The name "support vector classifier" is a result of optimizing the support vectors to get the optimal separating hyperplane. By selecting support vectors, support vector classifiers and by extension support vector machines are able to choose their model sizes [23].

Adding new input vectors can create a new optimal separating hyperplane [18]. Looking at vector $f(x^*)$ and seeing if x^* lies far away from our hyperplane a decision can be made in confidence since the magnitude of the testing set vector is far away from our decision boundary. Only the support vectors and observations close to the hyperplane decision boundary will be used in selecting the optimal. However, these new observation changes can be drastic if a new point is added closer to observations from the other class than the observation's own class. For something like k-nearest-neighbors, the decision boundary would just bend, but hyperplanes are usually linear in nature, and would just rotate or shift while maintaining its linearity.

Relaxing the assumption that the classes must be separable allows for overlap between the two classes. To generalize previous results, an additional variable will be introduced. Slack (ξ_i) represents the distance measure for the misclassifications on the wrong side of the margins. A penalty function defined by Vapnik [20]:

$$F_\sigma(\xi_i) = \sum_{i=1}^{N} \xi_i^\sigma \tag{3.8}$$

allows for the modification of the constraints depending on the half-margins. The modification can take two forms, measuring the overlap in actual distance from the margin in Equation (3.9), or measuring the relative distances of overlap in Equation (3.10) (this measure changes with the width of the margin M in). The two optimizations lead to different solutions.

$$y_i(x_i^T\beta + \beta_0) \geq M - \xi_i \tag{3.9}$$

or

$$\begin{gathered} y_i(x_i^T\beta + \beta_0) \geq M(1 - \xi), \\ \forall\, i,\ \xi_i > 0,\ F_\sigma(\xi_i) \leq K \end{gathered} \tag{3.10}$$

where K is some constant. The first problem will not be discussed in this paper. Optimizing the overlap in relative distance instead of actual distance creates the convex problem with linear constraints [7]. All support vector classifiers in this paper will be optimized henceforth under Equation (3.10).

By forcing a bound on $\sum_i(\xi_i)$ the total proportional amount by which the prediction $f(x_i)$ falls on the wrong side of the margin is bounded to K. The new optimization equation under this operation becomes [18]:

$$\min ||\beta|| \text{ subject to } \begin{cases} y_i(x_i^T\beta + \beta_0) \geq 1 - \xi_i \; \forall \, i \\ \xi_i \geq 0, \; \Sigma\xi_i \leq K \end{cases} \tag{3.11}$$

The points that will have the most impact on minimizing $||\beta||$ subject to the restraints are the ones that are not too far within the class boundary. This allows every point to contribute, but not all contributions will be equally significant.

3.2.1 Optimization of Support Vector Classifier

Solving the optimization problem is equivalent to finding the vector β that solves the following functional [20]:

$$\Phi(\beta, \xi) = \min_{\beta, \beta_0} \frac{1}{2}||\beta||^2 + C\sum_{i=1}^{N} \xi_i \text{ subject to} \tag{3.12}$$

$$\xi_i \geq 0, y_i(x_i^T\beta + \beta_0) \geq 1 - \xi_i \; \forall i \tag{3.13}$$

where C will become a cost tuning parameter replacing K. It is typical that C will be set before training based off the expected noise of the dataset. Some machine learning methods, however, will go through and train several SVM models at different C value and then compare their performances [18]. Cross-validation of different support vector classifiers can also be used to find optimal hyperparameter values for C.

Optimizing the convex problem over linear constraints is a Lagrangian problem. The solution is given by the saddle points of the Lagrangian primal function with multipliers α, μ [18, 7, 20].

$$\Phi(\beta, \xi, \alpha, \mu) = \frac{1}{2}||\beta||^2 + C\sum_{i=1}^{N}\xi_i - \sum_{i=1}^{N}\alpha_i[y_i(x^T\beta + \beta_0) - 1 + \xi_i] - \sum_{i=1}^{N}\mu_i\xi_i \tag{3.14}$$

which are minimized for all β, β_0, and ξ_i, while simultaneously maximizing α_i. The primal function is differentiated with respect to the correct parameter and set equal to zero:

$$\beta = \sum_{i=1}^{N}\alpha_i y_i x_i \tag{3.15}$$

$$0 = \sum_{i=1}^{N}\alpha_i y_i \tag{3.16}$$

$$\alpha_i = C - \mu_i, \forall\ i \tag{3.17}$$

with the following constraints: $\alpha_i, \mu_i, \xi_i \geq 0$.

The Wolfe (Lagrangian) dual objective function is the substitution of the optimal values for Equations (3.15)-(3.17) into Equation (3.14) and it gives the lower bound to the objective function we are attempting to optimize. It also has the additional bonus of being easier to solve than the primal function as instead of minimizing over the gradient subject to linear constraints the dual problem takes the maximization over the single dual variable α [18]

$$\max_{\alpha}\ \{L_D\} = \max_{\alpha}\{\sum_{i=1}^{N}\alpha_i - \frac{1}{2}\sum_{i=1}^{N}\sum_{j=1}^{N}\alpha_i\alpha_j y_i y_j x_i^T x_j\} \tag{3.18}$$

subject two constraints, Equations (3.19) and (3.20), as well as the Karush-Kuhn-Tucker conditions (3.21)-(3.23):

$$0 \leq \alpha_i \leq C \tag{3.19}$$

$$\sum_{i=1}^{N} \alpha_i y_i = 0 \tag{3.20}$$

$$\alpha_i[y_i(x_i^T + \beta + \beta_0) - (1 - \xi_i)] = 0 \tag{3.21}$$

$$\mu_i \xi_i = 0 \tag{3.22}$$

$$y_i(x_i^T \beta + \beta_0) - (1 - \xi_i) \geq 0. \tag{3.23}$$

Due to the extent of the characterization of the solution to both the dual and primal optimizations, we can see some of the properties of β.

$$\hat{\beta} = \sum_{i=1}^{N} \hat{\alpha}_i y_i x_i \tag{3.24}$$

Equation (3.24) is the solution, where nonzero coefficients $\hat{\alpha}_i$. $\hat{\beta}$ are represented solely in terms of the support vectors due to the constrains in Equation (3.23) being exactly met in Equation (3.21). For the support vectors that lie on the edge of the margin, and hence $\hat{\xi}_i = 0$, the constraint in Equation (3.18) holds with strict equalities with the remaining slack having $\hat{\alpha}_i = C$.

Any such margin points solve for β_0. For stability of the solution,

$$\bar{\beta}_0 = -\frac{1}{2}\hat{\beta} \cdot [x_i + x_j] \tag{3.25}$$

where x_i and x_j are any support vectors on the margin [18]. Once solutions are derived for $\hat{\beta}$ and $\hat{\beta}_0$, the final hard decision boundary can be written as:

$$\begin{aligned} G\hat{(x)} &= sgn[f\hat{(x)}] \\ &= sgn[x^T\hat{\beta} + \hat{\beta}_0]. \end{aligned} \tag{3.26}$$

3.3 Support Vector Machines

Up to this point we assumed that $y \ \epsilon \ [1, -1]$. Support vector machines act as a function-fitting problem using regularization to map into the dimensions of an enlarged space. This expansion allows for two things: better class separation in higher dimensions improving the overall classification accuracies, and forming nonlinear boundaries in lower dimensions.

The steps are similar to the previously discussed cases. First, the feature space is enlarged using basis expansion, $h_m(x), m = 1, \ldots, M$. All M basis functions are then applied to all the input features at each i^{th} observation [7]. Fitting the support vector classifier using input features so that $\boldsymbol{h(x_i)} = (h_1(x_i), \ldots, h_M(x_i))$ produces an often nonlinear $f(x_i) = \boldsymbol{h(x_i)}^T\beta + \beta_0$. We still arrive at the same classifier $G(x)$ [7].

3.3.1 Computing Support Vector Machines using Kernels

The change in the optimization function and its solution involves inner products of input features. Since we transformed the feature vectors using basis expansion function $\boldsymbol{h(x_i)}$, the Wolfe-Lagrangian dual function becomes

$$L_D = \sum_{i=1}^{N} \alpha_i - \frac{1}{2}\sum_{i=1}^{N}\sum_{j=1}^{N} \alpha_i\alpha_j y_i y_j \langle \boldsymbol{h(x_i)}, \boldsymbol{h(x_j)} \rangle. \tag{3.27}$$

Inner products are able to provide some form of similarity; the geometry of the vectors can be used in the interpretation of these models [19]. A final solution function $f(x)$ can be written in terms of this inner product, which if specified correctly is computed cheaply [7]. Because the actual transformation of $\boldsymbol{h(x)}$ is not used just the inner product of the transformation. A new function that computes the inner product in the transformed space can be defined. This kernel function:

$$K(x_i, x_i^T) = \langle \boldsymbol{h(x_i)}, \boldsymbol{h(x_i)^T} \rangle \tag{3.28}$$

means that specifying the initial basis transformations $\boldsymbol{h(x_i)}$ is unnecessary. K should be a symmetric, positive semi-definite function.

The two kernels tested for our application of support vector machines were:

$$\text{Radial Kernel: } K(x, x^T) = \exp(-\gamma||x - x^T||^2) \tag{3.29}$$

$$\text{Linear Kernel: } K(x, x^T) = (1 + \langle x, x^T \rangle). \tag{3.30}$$

It is worth noting, an input space can be mapped into a feature space that mimics the hidden layers in ANN models, resulting in a nonlinear classifier for cases where the separating hyperplane might not exist [21]. The kernel function for neural network approximations is:

$$\text{Neural Networks: } K(x, x^T) = \tanh(\kappa_1 \langle x, x^T \rangle + \kappa_2). \tag{3.31}$$

Solving the prior optimization problem under the new conditions, we have the solution function $f\hat{(x)}$:

$$f\hat{(x)} = \sum_{i=1}^{N} \hat{\alpha}_i y_i K(x_i, x^T) + \hat{\beta}_0. \tag{3.32}$$

In enlarged dimension spaces, the separation of the classes is often clearer. Large values of the cost parameter C will discourage any positive slack, overfitting a wiggly boundary in the original dimension space [7]. Having smaller C values smooths the boundary decision as $||\beta||$ is decreased. Again, cross-validation methods exist for choosing C to achieve a good testing error.

The decision boundary can also account for any bias within the kernel. So for a non-separable, higher dimension problem, the decision function becomes simply:

$$G\hat{(}x) = sgn[\sum_{i=1}^{N} \hat{\alpha}_i K(x_i, x_i^T)]. \tag{3.33}$$

Chapter 4

Results

4.1 Neural Network Results

The results presented are a comparison to the methods used in Eckert et al. [9]. The confusion matrices and accuracies are reported under their corresponding datasets in side-by-side comparisons. Afterwards, support vector machines were fit to the dataset that was simulated using radar equation noise

Learning for all statistical learning methods was done on a 75/25% split where 75% of the data is used to train the model and 25% is used for testing. By using this split we can train the model on a set of data and then present it with "new" data to evaluate it. This method is utilized as a way to avoid overfitting to the original dataset. An additional note, the paper for Eckert et al. does not include results from their runs using SGD optimizers; however, the optimal combination of layers and units was still found to be a 4 layer, 120 unit per layer network, even though it was not reported.

4.1.1 Stochastic Gradient Descent

For the dataset which was simulated with Gaussian noise that was applied to Cartesian coordinates for the radar noise, the optimal unit and layer combination for SGD optimizers in Keras was found to be the 4 layer, 120 units combination. A small change was made to the optimization algorithm discussed in Chapter 2; the ξ_i random variable no longer represents a single point, but a collection of a few points called a "batch." This mini-batch optimization is a compromise of the speed of SGD and the accuracy of using the entire gradient. All the results discussed in Chapter 2 still generalize to mini-batch stochastic gradient optimization [11]. The chosen network ended up being a complex, deep network because of the deficits of using regular SGD algorithms on the dataframe. Despite this complexity, the result was still underwhelming. On the original dataset with simulated Gaussian noise, the SGD optimizer was only ever was able to score an accuracy around 67.07%. The confusion matrix for the chosen SGD optimized neural network is listed in Table 4.1:

Cases	1	2	3	4	5	6	7	8	9	10	11	12
1	331	21	1	6	57	0	0	0	0	0	0	0
2	12	263	0	0	5	37	0	0	0	0	0	0
3	1	0	285	26	13	0	54	0	15	0	1	0
4	3	0	6	217	38	7	2	36	20	1	0	0
5	41	7	8	30	152	21	0	0	17	2	0	0
6	1	54	0	22	9	253	0	2	8	53	0	0
7	0	0	58	8	0	0	249	16	9	1	46	0
8	0	0	0	24	0	0	5	211	8	14	5	24
9	0	0	8	60	78	28	10	73	274	46	0	2
10	0	0	1	0	0	37	0	3	8	272	0	0
11	0	0	0	1	0	0	55	14	0	0	311	17
12	0	0	0	0	0	0	0	44	5	0	21	316

Table 4.1: Confusion Matrix of SGD Optimizer for Dataset with Gaussian-Distributed Noise

The SGD optimization algorithm is unable to overcome local minima when optimizing the objective function. In the basic SGD framework there is not a mechanism to help the optimizer out of situations when it gets stuck in local minima. Under the Gaussian noise framework, it is likely that the objective function is only locally convex.

The SGD optimizer on the dataset with noise simulated by the radar equation framework faired much worse than it's counterpart as we can see by the confusion matrix in Table 4.2. Under the same hyperparameters as before, the SGD optimizer model was only able to achieve a categorical accuracy of approximately 35.87%. This severe decrease in accuracy can be explained by the fact that the noise from the radar equation has comparatively greater noise. As a direct result, this dataset is harder to classify. The added noise is changes the shape of the objective function as well.

Cases	1	2	3	4	5	6	7	8	9	10	11	12
1	99	61	4	7	26	26	0	0	4	5	0	1
2	151	235	1	6	47	117	0	0	1	9	0	0
3	8	1	89	30	2	0	24	14	3	1	7	1
4	12	4	63	153	22	4	3	22	17	12	0	0
5	36	19	4	24	52	28	0	1	8	19	0	0
6	6	3	0	1	3	0	0	0	1	0	0	0
7	0	0	72	10	1	0	105	56	1	0	50	43
8	0	0	98	74	7	0	54	181	63	5	6	41
9	3	1	11	38	28	6	3	34	145	38	0	7
10	54	58	3	25	176	205	1	6	128	272	1	1
11	0	0	5	1	0	0	95	13	0	0	255	102
12	0	0	11	1	0	0	93	66	8	0	66	176

Table 4.2: Confusion Matrix of SGD Optimizer for Dataset with Noise Simulated by the Radar Equation

4.1.2 Adam

Eckert et al. [9] under their specific data framework, a 4 layer neural network with 50 units per layer would produce the best results. Their optimal model was able to produce a 91.24% categorical accuracy. The confusion matrix for their model is provided in Table 4.3.

Looking at the same neural network structures and hyperparameters under the dataset with noise simulated by the radar equation, the Adam optimizing neural network had a categorical accuracy of 88.64%, for which the confusion matrix is recorded in Table 4.4. Again, this decrease in categorical accuracies is to expected as a result of the added complexities that the radar equation noise framework brings to the dataset. Marginal accuracy increases could probably be made if the network was expanded to have more depth, and/or had different tuning of hyperparameters.

Cases	1	2	3	4	5	6	7	8	9	10	11	12
1	360	4	4	4	19	0	0	0	0	0	0	0
2	17	374	0	2	2	23	0	0	0	0	0	0
3	0	0	349	4	3	0	21	0	0	0	0	0
4	2	2	7	355	5	1	2	38	0	0	0	0
5	8	0	3	0	336	1	2	1	16	0	0	0
6	0	15	0	0	3	321	0	2	1	14	0	0
7	0	0	11	1	2	0	350	5	1	0	10	0
8	0	0	0	5	0	1	8	309	2	2	3	21
9	0	0	0	0	11	0	1	1	322	1	0	2
10	0	0	0	0	0	10	0	0	12	352	0	0
11	0	0	0	0	0	0	13	1	1	0	336	3
12	0	0	0	0	0	0	0	9	3	1	8	361

Table 4.3: Confusion Matrix of Adam Optimizer for Dataset with Gaussian-Distributed Noise

Cases	1	2	3	4	5	6	7	8	9	10	11	12
1	380	8	6	12	6	1	0	0	0	0	0	0
2	14	360	0	0	28	10	0	0	0	0	0	0
3	1	0	372	5	0	0	9	0	0	0	0	0
4	11	0	5	292	19	0	10	3	0	0	0	0
5	3	20	0	21	299	4	0	7	7	0	0	0
6	1	4	1	0	1	320	0	0	25	9	0	0
7	0	0	8	23	2	0	315	2	0	0	9	0
8	0	0	0	5	8	0	2	285	15	0	13	6
9	0	1	0	0	3	9	0	24	282	2	0	15
10	0	0	0	1	0	28	0	0	7	370	0	4
11	0	0	0	0	0	0	16	25	2	0	350	7
12	0	0	0	0	0	0	0	9	9	1	4	364

Table 4.4: Confusion Matrix of Adam Optimizer for Dataset with Noise Simulated by the Radar Equation

4.2 Support Vector Machine Results

4.2.1 Linear Kernel

Support vector machines using the linear kernel equation were fit in R to the dataset with radar equation noise to compete against the two SGD-variant neural networks. 10-fold cross-validation techniques were applied. In this technique, the data was separated into 10 equal sized segments. The first segment would be removed and training and testing would be done across the other 9 segments with the results recorded. The model would then add back the removed segment and repeat this process by removing the second segment, and so on. This allows for a robust estimate of the accuracies, and an optimal cost hyperparameter value can be selected from these results. Figure 4.1 Shows the accuracies of different 10 different cost values $C \, \epsilon \, [0, \frac{1}{4}, \frac{1}{2}, 1, 2, 4, 16, 32, 64, 128]$:

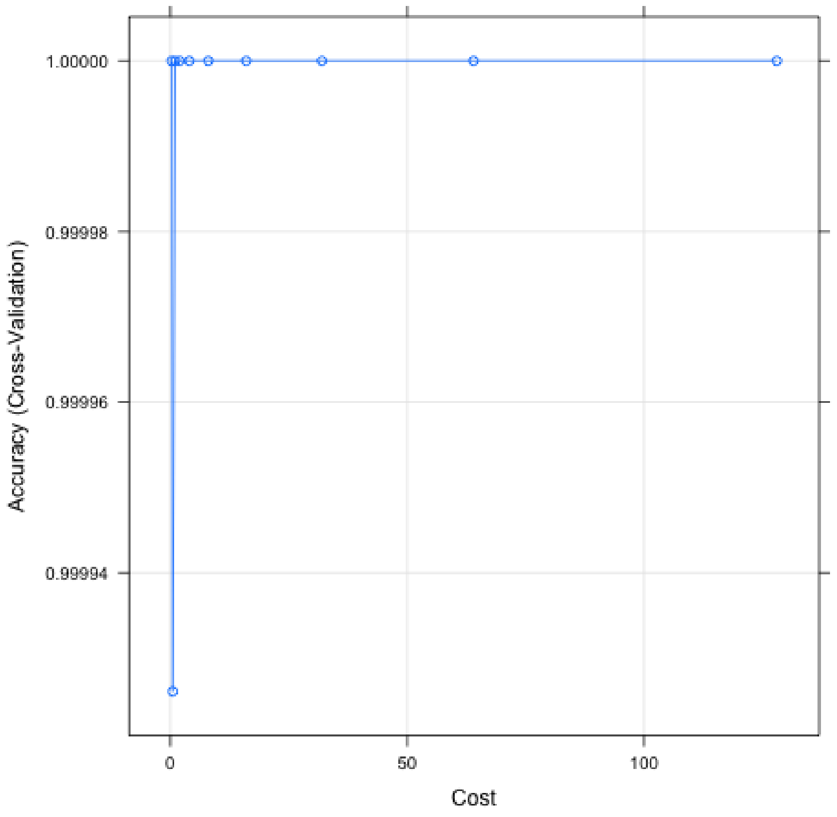

Figure 4.1: Linear Kernel Cost Cross-Validation

All of the cross-validated cost functions' values recorded values at 100% accuracy with one exception. When $C = 1$, the cost value's cross-validation accuracy was below 100%, but still above 99.99%. Since almost all of the potential cost parameters had an equal performance, the selection of an "optimal" cost comes down to preference. A confusion matrix of the results using a support vector machine with an arbitrarily chosen C value of $C = 10$ is shown below.

Cases	1	2	3	4	5	6	7	8	9	10	11	12
0	394	0	0	0	0	0	0	0	0	0	0	0
1	0	372	0	0	0	0	0	0	0	0	0	0
2	0	0	373	0	0	0	0	0	0	0	0	0
3	0	0	0	348	0	0	0	0	0	0	0	0
4	0	0	0	0	374	0	0	0	0	0	0	0
5	0	0	0	0	0	403	0	0	0	0	0	0
6	0	0	0	0	0	0	378	0	0	0	0	0
7	0	0	0	0	0	0	0	368	0	0	0	0
8	0	0	0	0	0	0	0	0	389	0	0	0
9	0	0	0	0	0	0	0	0	0	381	0	0
10	0	0	0	0	0	0	0	0	0	0	350	0
11	0	0	0	0	0	0	0	0	0	0	0	370

Table 4.5: Confusion Matrix of Predicted versus Actual Values for Linear Kernel

4.2.2 Radial Kernel

Support vector machines using radial kernel were fit to the dataset. Again the same 10-fold cross-validation techniques were applied as in the linear kernel. The selected cost values were the same as those tested in the linear kernel, $C \epsilon [0, \frac{1}{4}, \frac{1}{2}, 1, 2, 4, 16, 32, 64, 128]$. Typically, linear kernels do well with linearly separable data whereas the radial kernel works well on nonparametric data. Upon completion of initialization, the cross-validation accuracies were recorded in Figure 4.2.

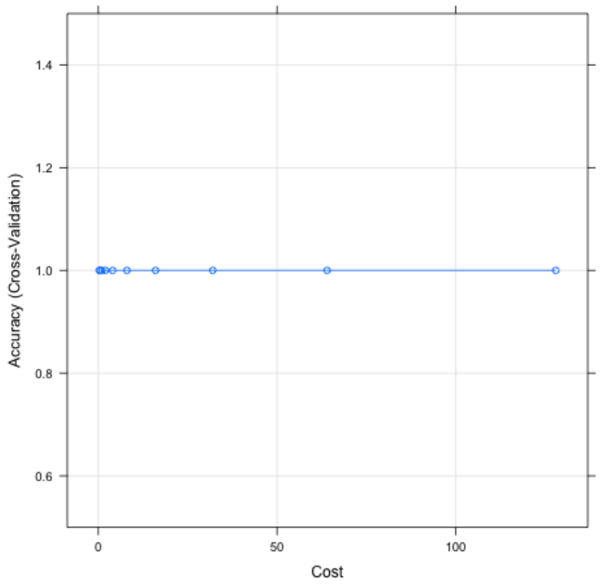

Figure 4.2: Radial Kernel Cost Cross-Validation

Again, the cross-validation provided excellent results that are robust. All cost function hyperparameter values scored 100% accuracies. The radial kernel, regardless of cost hyperparameter value, 100% correctly identified the testing observations for every 10-fold cross validation. Because the chosen C value has no bearings on the model's accuracy, the same $C = 10$ value for the hyperparameter was chosen based on preference, not performance. Table 4.6 is the confusion matrix for this support vector machine.

Cases	1	2	3	4	5	6	7	8	9	10	11	12
0	394	0	0	0	0	0	0	0	0	0	0	0
1	0	372	0	0	0	0	0	0	0	0	0	0
2	0	0	373	0	0	0	0	0	0	0	0	0
3	0	0	0	348	0	0	0	0	0	0	0	0
4	0	0	0	0	374	0	0	0	0	0	0	0
5	0	0	0	0	0	403	0	0	0	0	0	0
6	0	0	0	0	0	0	378	0	0	0	0	0
7	0	0	0	0	0	0	0	368	0	0	0	0
8	0	0	0	0	0	0	0	0	389	0	0	0
9	0	0	0	0	0	0	0	0	0	381	0	0
10	0	0	0	0	0	0	0	0	0	0	350	0
11	0	0	0	0	0	0	0	0	0	0	0	370

Table 4.6: Confusion Matrix of Predicted versus Actual Values for Radial Kernel SVM

4.3 Discussion of Results

Of these results, the best performance was done by the radial kernel support vector machine, which is able to 100% correctly classify the training set. These discussed results go on to highlight a major problem with neural networks. All neural networks are susceptible to becoming trapped in local minima. SVMs offer a better option for the classification of intermediate range missiles during launch over neural networks, regardless of any of the different optimizers discussed and used in this paper, because they guarantee a global minima.

Ultimately, the decision to choose radial kernel SVMs as the best option is highlighted by these factors:

- The radial kernel support vector machine guaranteed 100% accuracy on the testing set regardless of the kernel and cross-validated hyperparameter cost value. Adam optimized neural networks were only able to achieve ∼89% accuracy.
- Support vector machines' optimization does not get stuck in local minima. Since all neural network optimizers are susceptible to local minima, all neural networks required

multiple iterations to guarantee results that would be more than random guessing. This was hard coded into a for loop and added substantial computational time and power outside of the necessary $\mathcal{O}(\cdot)$ as a result for each of the SGD-variant optimizers. On a side note, there are other SGD-variant optimizers - like RMSprop and Adagrad - that have built in momentum tools so that they might overcome local minima. However, using these different optimizers still does not guarantee a global convergence.

- Computationally, support vector machines were quicker, more efficient, and cost less to run than neural networks. The required optimal layer and unit combinations for SGD and Adam were quite large. On average, a single iteration of a neural network (on either optimizer) was slower than a single iteration of the support vector machine. This effect naturally compounds as the density of the neural networks grows.

Chapter 5

Conclusion

5.1 Conclusion

This paper serves as an expansion to Eckert et al. [9] wherein they address the classification of intermediate range missiles during launch using radar telemetry. This paper looks at classifying more realistic data that had simulated noise from radar detection through the radar equation and filtering it using the common $\alpha - \beta$ filer. This paper applies the same neural network optimizers used in Eckert et al. and further adds two different support vector machines, one with linear and one with a radial kernel, to compare the categorical classification accuracies of both statistical learning methods. All data was trained on a 75% split with 25% being used for testing the models.

Comparisons of the neural networks with the same optimization methods and complexities were fit for the two dataset frameworks. The networks for both optimizers had lower accuracies on the dataset simulated with radar equation noise compared to the Gaussian noisy one. Most likely, this is a byproduct of the data simulated using the radar equation noise producing more realistic, and therefore harder-to-classify noise when compared to the Gaussian noise.

Cross-validation of two different kernel support vector machines compared against the neural networks showed that the radial kernel support vector machine was able to outperform the neural networks and achieve a perfect classification accuracy on all the test cases. While the linear kernel did not achieve a perfect classification accuracy across all cost hyperparameter values (99.994% at $C = 1$), it still performed better compared to the neural networks and remains a viable option for the classification of intermediate range missiles during launch. Overall, support vector machines are a better option for this classification.

The support vector machines always achieved a unique, global optimum that guaranteed high accuracies. They also did not require repeated initializations by user intervention to achieve these results and SVMs were able to run computationally faster than their neural network counterparts.

5.2 Future Research

The biggest extension of this work will be finding a way to overcome the limitations of radar. Acceleration could not be used in the input vector because of such limitations. Additionally, the simulated telemetries operate under the assumption of a fixed radar hitting the rocket cleanly and returning a signal back each time. Additional noise, such as false positives, could be added to test the model's capability of finding the difference between these 12 telemetry cases and some flying object, like an aircraft.

Even further, using satellite images to make the classifications under a convolutional neural network could introduce a more realistic approach to the data simulations. Comparisons to support vector machines and such networks could also be done.

Bibliography

[1] Park, S., Jeong, J., Ryoo, C.K., and Choi, K. *Detection and Classification of a Ballistic Missile in Ascent Phase.* IEEE 11th Internation Conference on Control, Automation and Systems, Incheon, Korea, Oct. 2011

[2] Singh, U. Padmanabhan, V., and Agarwal, A., *Dynamic Classification of Ballistic Missiles Using Neural Networks and Hidden Markov Models.* Applied Soft Computing, Vol. 19, pp. 280-289, March 2014

[3] Skolnik, M. I., *Radar Handbook*, 3rd Edition, McGraw-Hill, 2008.

[4] Etkin, B. Dover Publications, Inc. *Dynamics of Atmospheric Flight* Mineloa, New York 2000.

[5] Hartfield, R., Jenkins, R., Burkhalter, J., and Foster, W., *Analytical Methods for Predicting Grain Regression in Tactical Solid-Rocket Motors*, Journal of Spacecraft and Rockets, Vol 41. No. 4, July-August 2004, pg. 689-693

[6] Eglen, Stephen J. *A quick guide to teaching R programming to computational biology students.* PLoS computational biology vol. 5,8 (2009): e1000482. doi:10.1371/journal.pcbi.1000482

[7] Friedman, J., Tibshirani, R., and Hastie, T. (2001) *Elements of Statistical Learning* Springer, 2nd. ed. pg. 389-440

[8] Higham, Catherine F., and Desmond J. Higham. *Deep Learning: An Introduction for Applied Mathematicians.* SIAM Review, vol. 61, no. 3, 2019, pp. 860–891., doi:10.1137/18m1165748.

[9] Eckert, J., Carpenter, M., Hartfield, R. Cervantes, N. *Classification of Intermediate Range Missiles During Launch* AIAA Scitech 2020 Forum, AIAA, 2020, Orlando, FL

[10] Leke, Collins Achepsah. *Deep Learning and Missing Data in Engineering Systems.* Springer, 2019

[11] Bottou, L, et al. *Optimization Methods for Large-Scale Machine Learning*, "SIAM Review." SIAM, 2018, pp. 223–311.

[12] Liu, Danqing. *A Practical Guide to ReLU.* Medium, Medium, 30 Nov. 2017, https://medium.com/@danqing/a-practical-guide-to-relu-b83ca804f1f7.

[13] Oeding, Luke, and Hamza Jaffali. *Learning Algebraic Models of Quantum Entanglement.* ArXiv, 8AD, https://arxiv.org/abs/1908.10247.

[14] Skajaa, A., *Limited Memory BFGS for Nonsmooth Optimization*, Courant Institute of Mathematical Science, New York, 2010

[15] Nocedal, J., Wright, S. *Numerical Optimization*, Numerical Optimization, Springer, 2006, pp. 136–180.

[16] Liu, Dong C., and Jorge Nocedal. *On the Limited Memory BFGS Method for Large Scale Optimization.* Mathematical Programming, vol. 45, no. 1-3, 1989, pp. 503–528., doi:10.1007/bf01589116.

[17] Kingma, D., Ba, J. *Adam: A method for Stochastic Optimization*, arXiv, ICLR, 2015

[18] Gunn, S. *"Support Vector Machines for Classification and Regression"* University of Southampton Faculty of Engineering, Science and Mathematics. May 10th, 1998

[19] Berwick, R., *An Idiot's Guide to Support Vector Machines*, Massachusetts Institute of Technology, Laboratory for Decision Information Systems

[20] C. Cortes and V. Vapnik. *Support vector networks.* Machine Learning, 20:273 – 297,1995.

[21] Kecman, V., *Learning and Soft Computing Support Vector Machines, Neural Networks and Fuzzy Logic Models*, Learning and Soft Computing Support Vector Machines, Neural Networks and Fuzzy Logic Models, MIT Press, 2015, p. 139.

[22] Breiman, L., *Statistical Modeling: The Two Cultures (with Comments and a Rejoinder by the Author)*, Statistical Science, vol. 16, no. 3, 2001, pp. 199–231., doi:10.1214/ss/1009213726.

[23] Rychetsky, M. et al. *Accelerated training of support vector machines*, IJCNN'99., International Joint Conference on Neural Networks., Proceedings (Cat. No.99CH36339) 2 , 1999, 998-1003 vol.2.

[24] *Meat and Fish Freshness Inspection System Based on Odor Sensing - Scientific Figure on ResearchGate.* Available from: https://www.researchgate.net/figure/Optimal-separating-hyperplane_fig4_233828536 [accessed 2 Feb, 2020]

[25] Hlavac, M., *stargazer: Well-Formatted Regression and Summary Statistics Tables.* 2018, R package version 5.2.2., https://CRAN.R-project.org/package=stargazer

Appendices

Appendix A

Exploratory Graphics

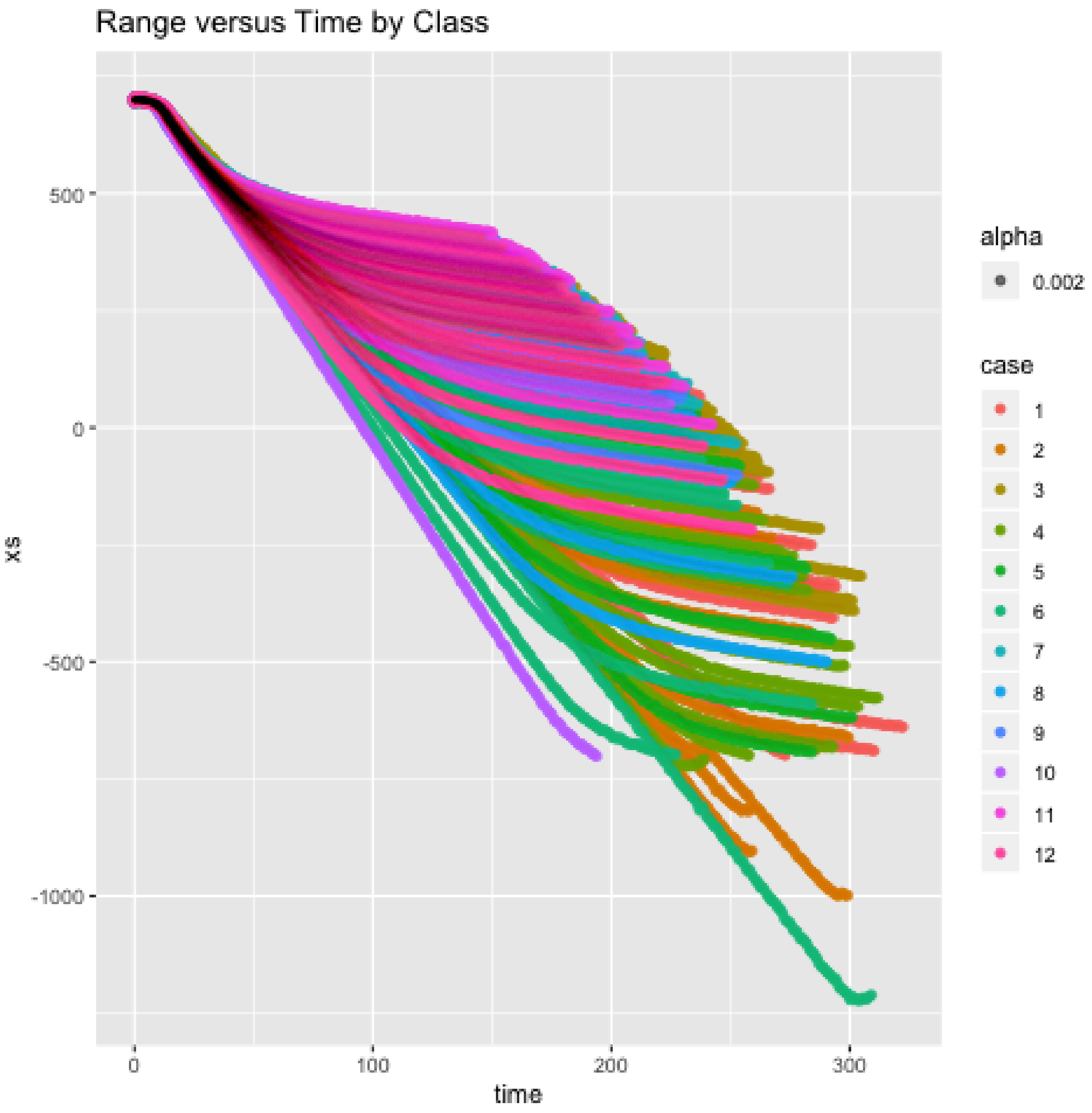

Figure A.1: Range versus Time Profiles for 50 Randomly Selected Missiles

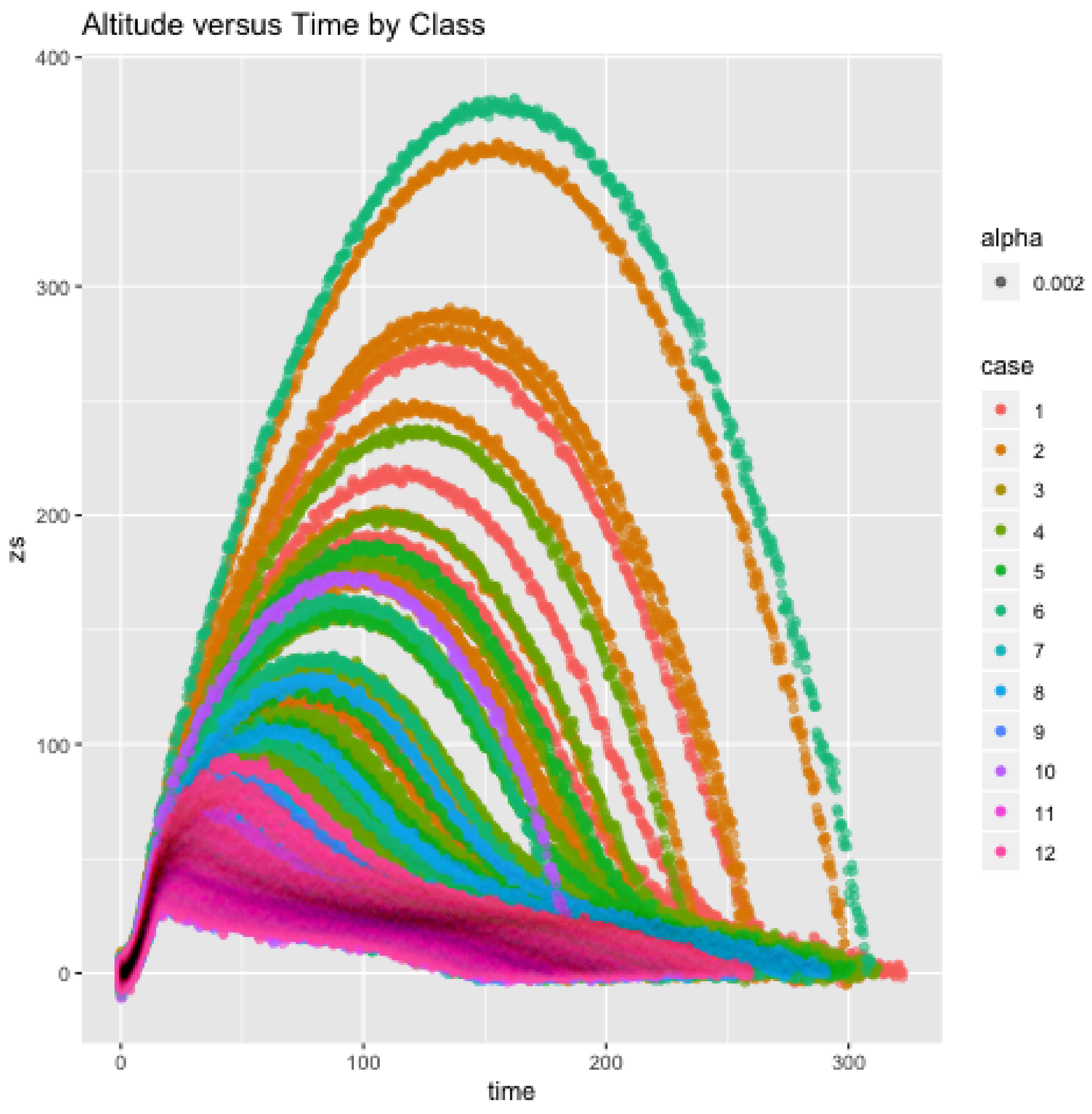

Figure A.2: Altitude versus Time Profiles for 50 Randomly Selected Missiles

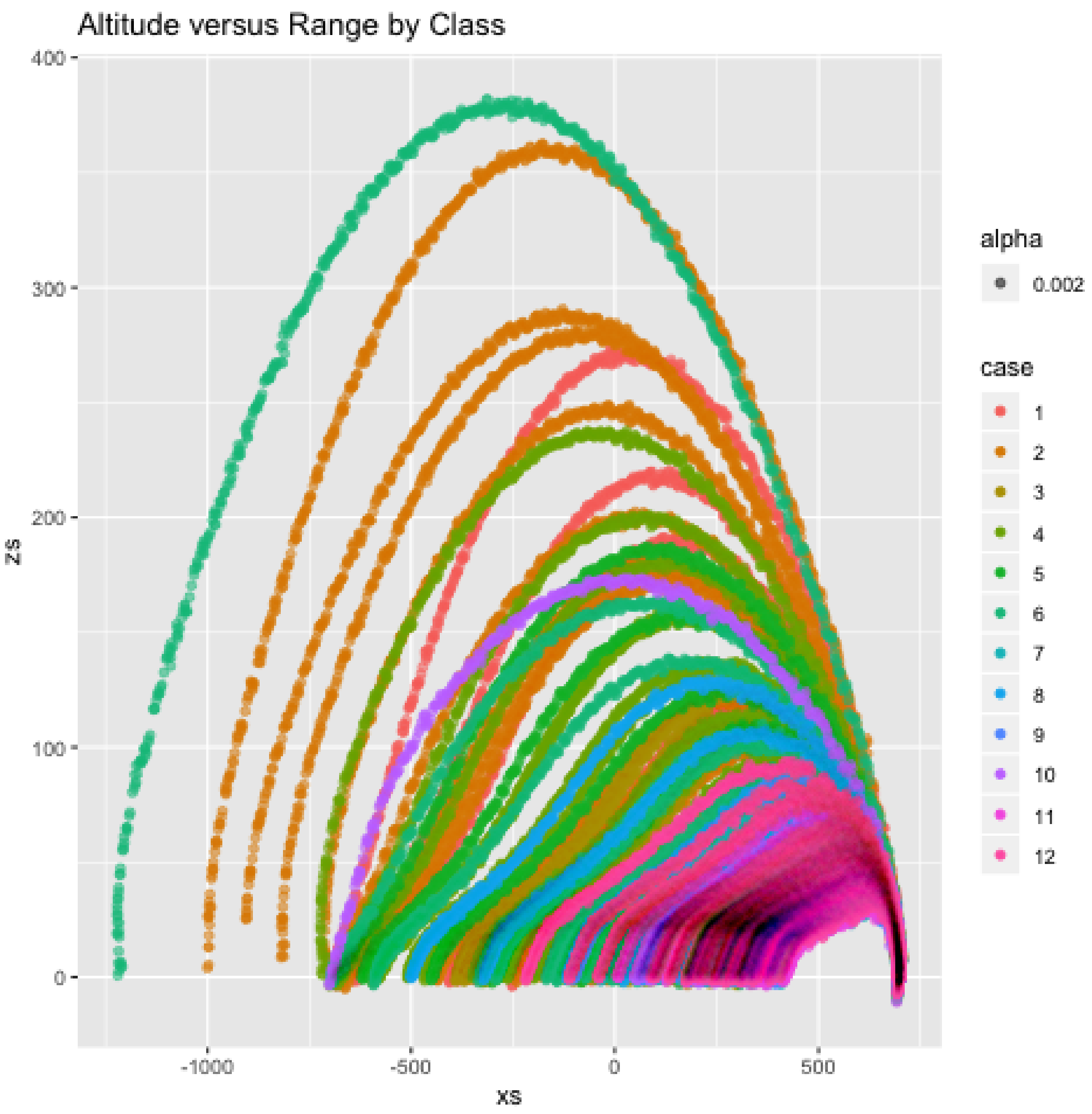

Figure A.3: Altitude versus Range Profiles for 50 Randomly Selected Missiles

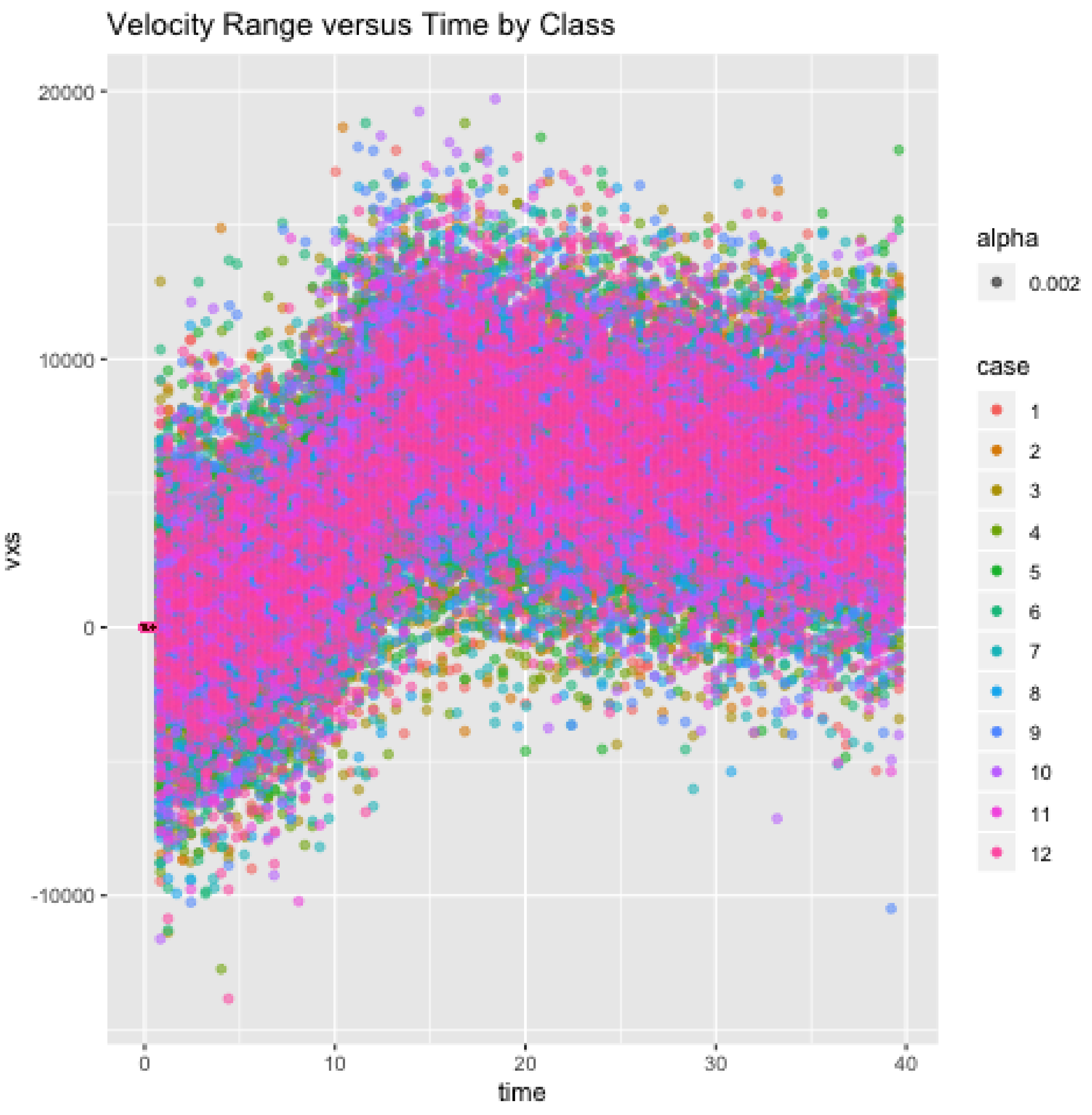

Figure A.4: 20 Second Interval of Range versus Time Velocity Profiles for 50 Randomly Selected Missiles

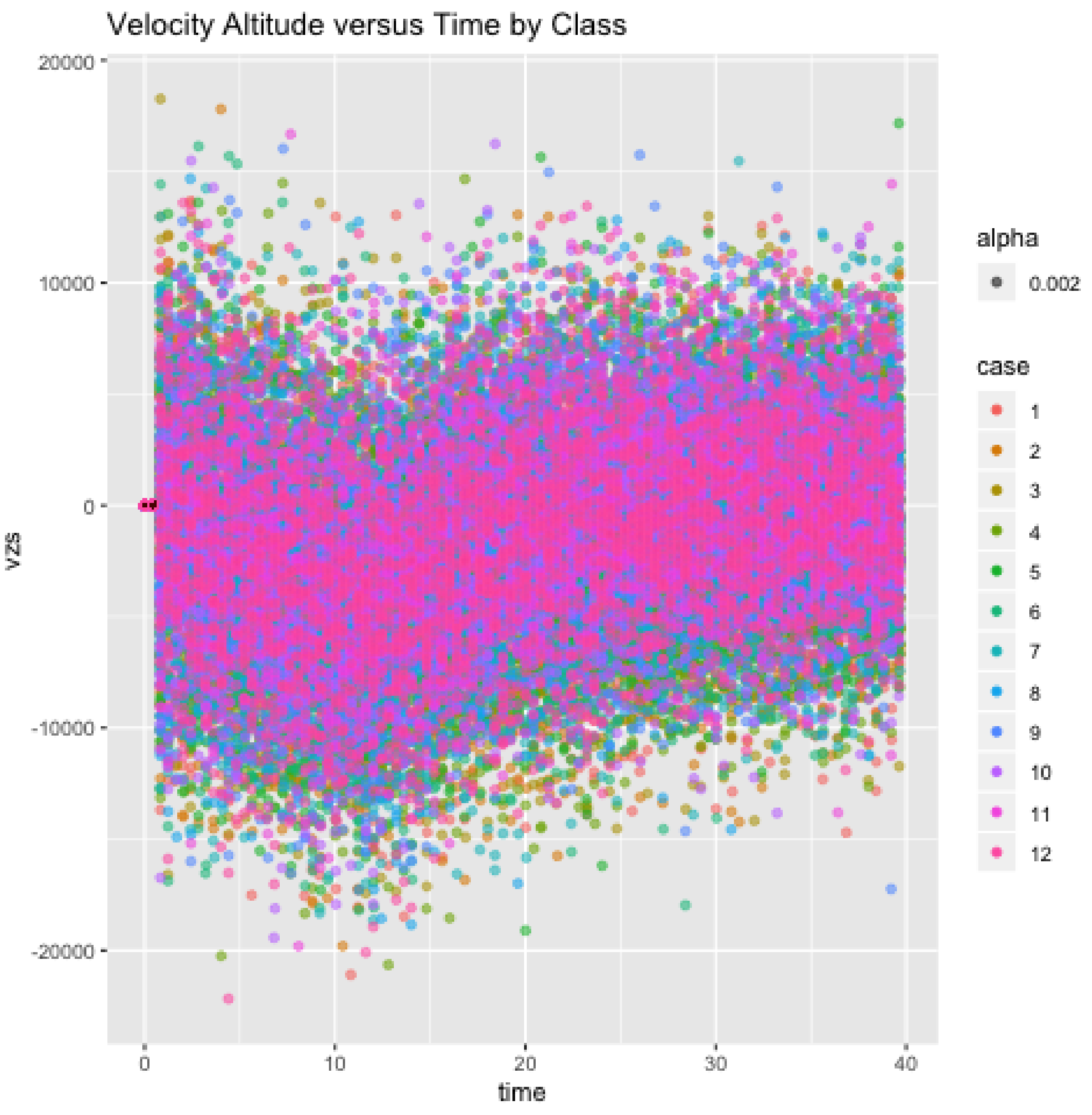

Figure A.5: 20 Second Interval of Altitude versus Time Velocity Profiles for 50 Randomly Selected Missiles

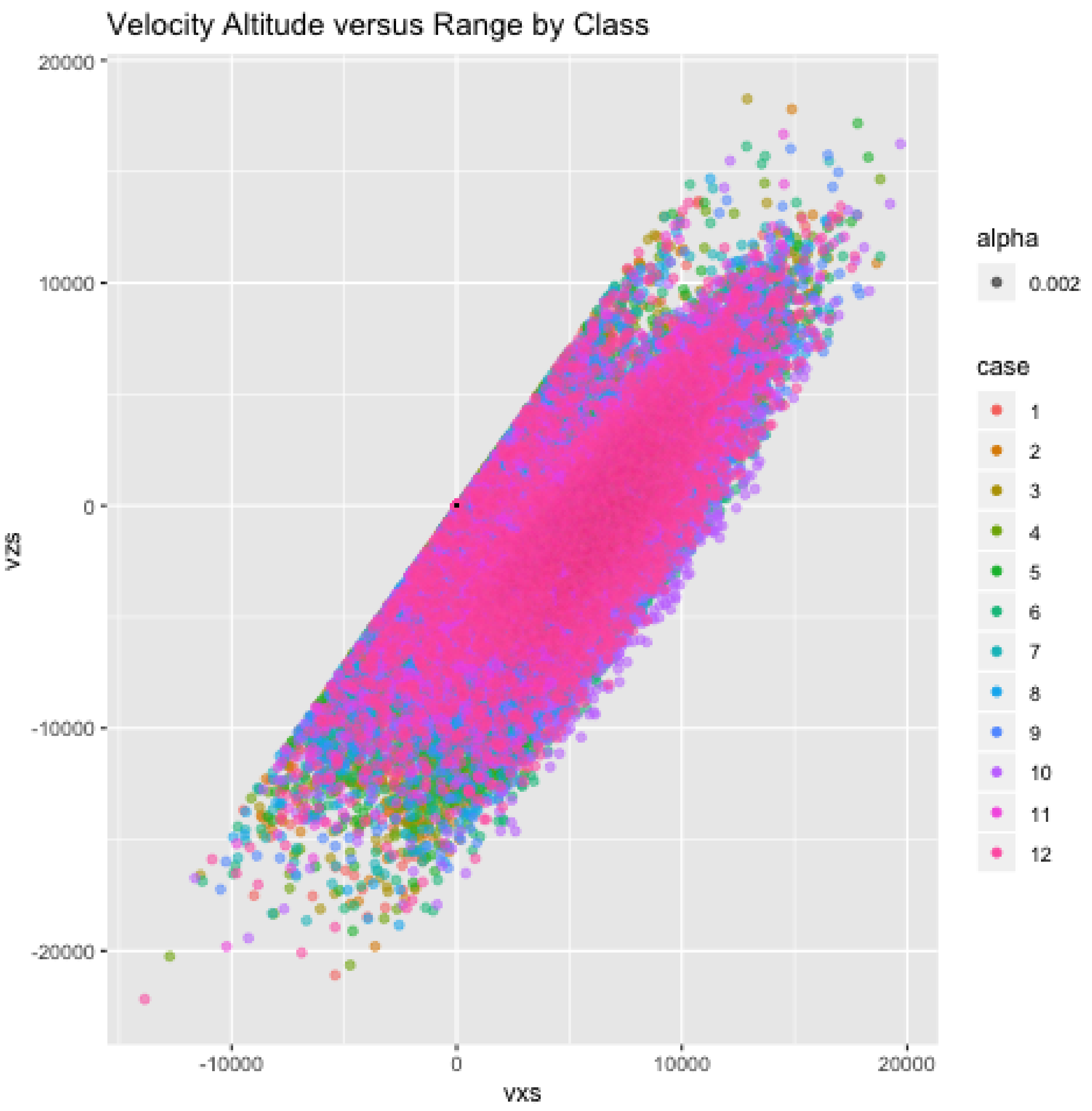

Figure A.6: 20 Second Interval of Altitude versus Range Velocity Profiles for 50 Randomly Selected Missiles

Appendix B

Proof of **Lemma 2.2**

Proof. By **Lemma 2.1**, the iterates generated by the SGD Algorithm satisfy [11]:

$$\begin{aligned} F(w_{k+1}) - F(w_k) &\leq \nabla F(w_k)^T(w_{k+1} - w_k) + \frac{1}{2}L||w_{k+1} - w_k||_2^2 && \text{(B.1)} \\ &\leq -\alpha_k \nabla F(w_K)^T g(w_k, \xi_k) + \frac{1}{2}\alpha_k^2 L||g(w_k, \xi_k)||_2^2. && \text{(B.2)} \end{aligned}$$

After taking the expectations of both sides to the inequalities conditioned on ξ_k, it will be shown that only w_{k+1} depends on ξ_k.

□

Appendix C

Proof of **Lemma 2.4**

Proof. By **Lemma 2.2** and Equation (2.21), it follows that:

$$E[F(w_{k+1})|\xi_k] - F(w_k) \leq -\alpha_k \nabla F(w_k)^T E[g(w_k, \xi_k)|\xi_k] + \frac{1}{2}\alpha_k^2 L E_{xi_k}[||g(w_k, \xi_k)||_2^2] \tag{C.1}$$

$$\leq -\mu\alpha_k ||\nabla F(w_k)||_2^2 + \frac{1}{2}\alpha_k^2 L E_{\xi_k}[||g(w_k, \xi_k)||_2^2], \tag{C.2}$$

which is Equation Equation (2.24). Assumption 4.3 gives:

$$E_{\xi_k}[||g(w_k, \xi_k)||_2^2] \leq M + M_G ||\nabla F(w_k)|| \text{ with } M_G := M_V + \mu_G^2 \geq \mu \geq 0$$

which yields Equation (2.25) □

Appendix D
Using SAS and the L-BFGS Optimization

The primary results for this thesis were computed in statistical program R. For robustness measures, these results were compared against the results from statistical software SAS. This proprietary software differs from R in several aspects. The first aspect is the choice in the optimizer. Instead of using traditional stochastic gradient methods, SAS uses quasi-Newton optimization. The second aspect is in training. SAS has built-in criteria to stop a current training session if the validation accuracy has not increased in some set number of consecutive epochs. The algorithm then picks another point on the objective function to start and begins optimizing again and this iteration will not be used in reporting accuracies. SAS always, as a result, report a better accuracy when compared to a single iteration of the Keras code in R as it will avoid running entirely into local minima.

There are two reasons I decided that these result's inclusion in the main text is unnecessary. The first is, these neural networks were run by a different software than the other two, and as such cannot be fairly compared. The second is, that this was still a neural network and while the method was able to achieve better accuracy than traditional SGD and the Adam variant, it was worse than SVMs and did not guarantee the global optima.

D.1 Newtonian Versus Quasi-Newton Methods

The discussion of the L-BFGS optimization algorithm is strengthened when discussed in the context of Newton versus quasi-Newton optimization methods. Therefore, a brief introduction of the two method types will be explained before a discussion of the inner mechanisms of the L-BFGS algorithm.

D.1.1 Newtonian Methods

Under generic gradient descent methods of optimization, the direction of movement (defined as the vector $-\nabla R_n(w)$) and the step length (defined as the scalar α_k) provide the next iterate through an updating function after both are calculated respectively. While SGD optimization follows iterative updating, it is important to note that the algorithm computes the direction of movement before the step-size. There is a flexibility of SGD to extend to Newton methods that reside in the fact that computing the search direction first is a requirement for the steps of Newton's methods.

In Newton's methods, the area around w_k is assumed to be quadratic and can be well approximated by a quadratic in terms of the search distance, which is calculated first by finding the minimum of the quadratic [14]. For ease of notation the search direction will be referred to as $D_k = -\nabla R_n(w_k)$:

$$R_n(w_k + D_k) \approx R_n(w_k) + (D_k)^T \nabla R_n(w_k) + \frac{1}{2}(D_k)^T \nabla^2 R_n(w_k) D_k \tag{D.1}$$

and by setting Equation (B.1) equal to 0 and solving for D_k the minimizer is obtained:

$$D_k = -[\nabla^2 R_n(w_k)]^{-1} \nabla R_n(w_k). \tag{D.2}$$

Under the assumption that the Hessian ($\nabla^2 R_n(w_k)$) is positive definite, Equation (B.2) is how Newton optimizers "step" when going through gradient descent. For any objective function, once the iterates come sufficiently close to the minima, it is easy to show that the convergence is quadratic [15]. The rate of convergence for Newtonian methods is $\mathcal{O}(p^3)$ meaning that the rate of convergence is quite costly when the number of variables is large.

D.1.2 Quasi-Newton Methods

A major drawback to Newton methods is the requirement that the Hessian is calculated at every iteration. The standard replacement then is to define [14]:

$$D_k = -C_k \nabla R_n(w_k) \tag{D.3}$$

where C_k is updated by a formula involving the inverse of the true Hessian known as the "quasi-Newton" updating formula. We will assume that C_k is a symmetric, positive definite matrix where $(-C_k \nabla R_n(w_k))^T \nabla R_n(w_k) < 0$. Equation (B.2) and Equation (B.3) have equality when $C_k = (\nabla^2 R_n(w_k))^{-1}$.

If we allow $B_k := C_k^{-1}$, $S_k = w_{k+1} - w_k$, and $Y_k = \nabla R_n(w_{k+1}) - \nabla R_n(w_k)$, then imposing the well known secant equation to update the matrix:

$$B_{k+1}(\alpha_k D_k) = \nabla R_n(w_{k+1}) - \nabla R_n(w_k) \tag{D.4}$$

we get the matrix norm that creates the the best update:

$$C_{k+1} = \arg \min_{C} ||C - C_k||. \tag{D.5}$$

That is, the closest C_{k+1} matrix to C_k among all potential symmetric, positive definite matrices satisfying the secant equation will be the update [14]. Each matrix norm choice creates a different update formula, meaning the algorithm only needs to remember the previous iterates update, not the entire Hessian.

D.2 BFGS Algorithm

Since the L-BFGS algorithm is a natural extension to the BFGS algorithm, it's worth taking some time to discuss the foundations of the BFGS algorithm. The BFGS algorithm works by using:

$$C_{k+1} = (I - \rho_k S_k (Y_k)^T) C_k (I - \rho_k Y_k (S_k)^T + \rho_k S_k (S_k)^T \qquad \text{(D.6)}$$

as the update formula where $\rho = ((Y_k)^T S_k)^{-1}$.

Algorithm 3 BFGS Algorithm:

```
1: Initialize: $w_0$, convergence tolerance: $\delta$, $C_0$
2: $k \leftarrow 0$
3: while $||\nabla R_n(w_k)|| \geq \delta$ do
4: $D_k \leftarrow -C_k \nabla R_n(w_k)$
5: $\alpha_k \leftarrow \Gamma(w_k, \nabla R_n(w_k))$
6: $w_{k+1} = w_k - \alpha_k D_k$
7: $S_k = w_{k+1} - w_k$, $Y_k = \nabla R_n(w_{k+1}) - \nabla R_n(w_k)$
8: Calculate $C_{k+1}$ by means of Equation (2.36)
9: $k \leftarrow k + 1$
10: end while
Output: $w_k$, $R_n(w_k)$, $\nabla R_n(w_k)$
```

where the gamma function represents a line search algorithm that satisfies the Wolfe Conditions. These conditions are for some $0 < \beta' < \frac{1}{2}$ and $\beta' < \beta < 1$ [16]:

$$R_n(w_k - \alpha_k \nabla R_n(w_k)) \leq \nabla R_n(w_k) + \beta' \alpha_K (\nabla R_n(w_k))^T D_k \qquad \text{(D.7)}$$

and

$$\nabla R_n(w_k + \alpha_k D_k)^T D_k \geq \beta (\nabla R_n(w_k))^T D_k. \qquad \text{(D.8)}$$

The algorithm has a strong self-correcting property that makes it desirable for neural network optimization depending on selecting the right line search method. The success of the method, then, depends on the approximation of C_k to the inverse of the true Hessian and achieves superlinear convergence when close to the minimizer.

While BFGS algorithms are among the most popular updating formula for C_{k+1} it still has a computational cost of $\mathcal{O}(p^2)$ (plus any gradient/function calculation costs) as $C_k(I - \rho_k Y_k (S_k)^T) = C_k - \rho_k (C_k Y_k)(S_k)^T$ [14]. The computation cost of this quasi-Newtonian method is lower because it's a simple matrix-vector multiplication, not solving a linear system. The cost can be quite expensive, nonetheless, when a lot of variables are used in the input and the multiplication becomes large.

D.3 L-BFGS Algorithm

The L-BFGS algorithm was developed for a less computationally extensive method for a large input basis. The algorithm is quasi-Newton, but instead of storing the entire C_k matrix as is the case in the BFGS, the algorithm only stores information of the most recent iterates that represent the approximations implicitly [15]. As the iterations progresses, the current information of the Hessian becomes less and less dependent on the earlier prior iterates. The curvature information from only the most recent iterations is used in the L-BFGS instead of the dense $k \times k$ matrix approximations store by BFGS which creates an almost linear rate of convergence.

The difference in the L-BFGS algorithm to the BFGS algorithm is the storing of C_k is done using a modified C_k implicitly for m previous iterations. After a new iterate is computed, the oldest vector pair of iterates is replaced by the newly calculated pair. A double loop in the algorithm makes these updates.

Begin by choosing an initial Hessian approximate C_k that can vary from iteration to iteration. Repeated application of Equation (2.36) leads to an update for C_k by calculating $C_k \nabla R_n(w_k)$ [15].

Algorithm 4 L-BFGS Algorithm:

```
1: $q \leftarrow \zeta_k \nabla R_n(w_k)$    where    $\zeta_k = ((s_{k-1})^T Y_{k-1})((Y_{k-1})^T Y_{k-1})^{-1}$
2: for $i = k-1, k-2, \ldots, k-m$ do
3:     $\alpha_i \leftarrow \rho_i (s_i)^T q$
4:     $q \leftarrow q - \alpha_i Y_i$
5: end for
6: for $i = k-m, k-m+1, \ldots, k-1$ do
7:     $\beta \leftarrow \rho_i (Y_i)^T r$
8:     $r \leftarrow r + s_i(\alpha_i - \beta)$
9: end for
Output: $D_k = -r$
```

The past m iterations implicitly do the operations required by the inverse Hessian for computing the next search direction [14]. Note the line search is set first to the unit step length and is only changed if it does not satisfy the Wolfe conditions. The updating is done in $4mp$ multiplications bringing computational costs down to $\mathcal{O}(mp)$.

There are very few cases where $m > p$ is required to converge. When the objective function is locally Lipschitz continuous, then the probability that that an optimization algorithm that is initialized randomly will encounter a point where the objective function is not differentiable equals zero by Rademacher's Theorem. The L-BFGS is such an algorithm, meaning that it's application on nonsmooth optimization is just as effective as in the smooth case. [14, 16].

One potential downside to the L-BFGS is the inability to account for the nonconvex case. The properties of global optimization for L-BFGS on nonconvex problems has not been fully researched - however there exists a way around the problem. By picking a sufficiently large amount of different initial starting points on the function and then running the optimization algorithm you can compare the locations of the minima. If on a particular iteration of the algorithm it seems to not be converging to the priori minima of the loss function, then the iteration can be terminated and restart from a different location. The probability that one of these starting points will contain the true minima approaches 1 as the number of points selected approach infinity.

D.4 Results of L-BFGS

Full results containing the confusion matrix are not shown, as this was just to verify that the accuracies received during testing for the neural networks in R were robust. The selected unit and layer pairwise combination under the Gaussian noise dataset in Eckert et al. [9] was a 2 layer, 20 unit network. It is unsurprising that the required network depth was drastically cut compared to the results in the main text as the L-BFGS optimizer is better than both variants of SGD discussed and is biased under SAS's framework. The accuracy under this method reported 98% categorical accuracy.

Under the radar equation dataset, again a 2 layer, 20 unit neural network was tested. The accuracy was decreased, following the pattern seen in the main text, but not nearly as drastically. Instead, the accuracy ended up being around 95%. While these results are certainly more impressive than the other two SGD variants, it still fails short of the SVM results and does not change the conclusions drawn in the main text.