Formalized Generalization Bounds for Perceptron-Like Algorithms

A thesis presented to

the faculty of

the Russ College of Engineering and Technology of Ohio University

In partial fulfillment

of the requirements for the degree

Master of Science

Robin J. Kelby

August 2020

This thesis titled

Formalized Generalization Bounds for Perceptron-Like Algorithms

by

ROBIN J. KELBY

has been approved for

the School of Electrical Engineering and Computer Science

and the Russ College of Engineering and Technology by

Gordon Stewart

Assistant Professor of Electrical Engineering and Technology

Mei Wei

Dean, Russ College of Engineering and Technology

Abstract

KELBY, ROBIN J., M.S., August 2020, Computer Science

Formalized Generalization Bounds for Perceptron-Like Algorithms (24 pp.)

Director of Thesis: Gordon Stewart

Insert your abstract here

Dedication

Dedicated to my Nathan.

Your patience, video games, and good cooking kept me going.

Acknowledgments

Acknowledge later

Table of Contents

List of Figures

Figure Page

1 Introduction

The field of machine learning research has advanced very quickly in the past decade. Machine learning describes the class of computer programs that automatically learn from experience, often employed for classification, recognition, and clustering tasks. One of the classic problems in machine learning is handwritten digit recognition to classify written numbers automatically. Computers have historically struggled to interpret handwritten information because handwriting can vary drastically between writers. While humans can be taught to read as well as learn to read on their own, handwriting recognition can be challenging for computers to accomplish. Several datasets have been created specifically for the problem of handwriting analysis for numerical digits. For example, the MNIST dataset [**?**] is one of the primary datasets for computers to learn how to classify handwritten digits into the numbers 0-9. This dataset allows researchers to compare the performance of multiple models, trained and tested on the same data, but using different machine learning algorithms. Some systems have achieved a near-perfect performance on the MNIST dataset for the problem of handwritten digit classification, and this technology is valuable for processing documents, such as ZIP codes on letters sent through the U.S. Postal Service.

Increasingly, machine learning has been heavily integrated into our daily lives. As described by the authors of "Social media big data analytics", social media companies such as Facebook, Twitter, and YouTube learn from our digital data in order to serve individuals with targeted information and often advertising [**?**]. Retailers track customer purchases to learn about individuals' habits and entice them with specific offers and coupons. The pages we visit, profiles we create, and products we buy are used to predict our future actions and monetize our attention. This kind of task would be almost impossible for a human to complete, due to the vast amounts of data involved per person or account. In addition to social media, retailers, and advertisers, machine learning

techniques are also being employed in critical systems, such as healthcare and infrastructure, where failure can lead to the loss of time, money, and lives. Research to evaluate the use and oversight of machine learning algorithms [**?**] has shown that there are few existing safety principles and regulations for critical systems that rely on machine learning components. Machine learning drives more than websites and commerce; its algorithms are also responsible for the well-being and safety of people around the world, and regulation has largely not caught up with machine learning advances.

The development of machine learning tends to be experimentally driven in most applications. New or finely tuned configurations for internal components can lead to increased accuracy and efficiency or decreased training time compared to other algorithms for a specific problem or dataset. Such refinements can have enormous impacts for researchers studying machine learning problems and algorithms. The process of machine learning differs from algorithm to algorithm, but for most methods, machine learning algorithms learn models from training data to encode the program's knowledge. Models consist of learned parameters, which represent different kinds of data depending on the encoding of the model, and hyperparameters, a small number of variables directly specified by the programmer. Learned models are able to take a new piece of data as input and produce a result or judgment from that data. In the case of handwritten digits, the input to the model is the handwritten digit, and the output is the classification of that digit as a number from 0-9.

Machine learning algorithms can produce models that have millions of learned parameters, and small changes to model training, configuration, or hyperparameters can have enormous impacts on performance. Because of the complexity of the models produced by many machine learning algorithms, most new papers published in the field describe results found through experimentation, as opposed to examining the underlying theory responsible for these advances. Additional research in understanding the theory

behind machine learning may help to understand why some techniques are better suited for some problems than others, as well as potential avenues for exploration.

Finding errors in machine learning algorithms or models can be very difficult. With thousands or millions of parameters learned by the computer, not specified by the programmer, algorithms can easily get stuck in small, local solutions instead of finding the optimal solution. For example, gradient descent is an algorithm tasked with finding the lowest, or global, minimum of a multi-dimensional hillside with many peaks and valleys. Through many iterations, gradient descent travels downward along the gradient until a place is reached where descent is no longer possible. If the algorithm cannot find a deeper valley, this depth is returned as the overall solution. However, gradient descent can fail to find the global solution when the hyperparameters are not tuned correctly by the programmer or deeper valleys take too long to find. Techniques have been developed to mitigate the limitations of gradient descent, such as momentum, but the programmer usually has to experiment with multiple techniques to achieve peak performance. Additionally, few machine learning algorithms have theoretical properties that can be verified, such as a theorem that a learning algorithm will always terminate or find the global solution. Research into verifying machine learning to produce models with optimal behavior is limited due to these difficulties.

One way to increase our knowledge in the theory of machine learning is to verify the correctness of machine learning algorithms through mathematical proofs. Formal verification often entails machine-checked proofs of correctness, where software is built or translated into a proof assistant, such as Coq. Proof assistants allow for the integration of proofs with software specifications and implementations. Mathematical proofs in Coq are guaranteed to be as valid as the proof assistant itself, and because these proofs are portable programs, access to the proofs can allow others to verify proofs as well. Because implementations are written in the same environment as their proofs, the proofs directly

correspond to the implementation verified. The Coq environment also provides libraries containing both implementations of data structures and proofs to aid in the development of verified systems.

Researchers have used the Coq proof assistant to verify many different software systems and prove correctness properties. The CompCert compiler for the C language [**?**] is the first verified compiler, proving that the behavior of a C program compiled with CompCert will not be changed in the transformation of compilation. Verified compilers ensure that the executable program produced by the compiler does not contain errors produced in compilation. For safety-critical applications, executables created by a verified compiler are more secure than executables created by unverified compilers. Another verified system written in Coq is Verdi [**?**], a framework for specifying and implementing distributed systems with tolerance for node faults. In a network of computers, connections can be dropped, packets lost or sent out of order, and nodes can fail or restart. Verdi allows the programmer to specify the fault conditions their distributed system should be resilient against, and the Verdi system itself mechanizes much of the proof process and code extraction for deployment in real-world networks. Distributed system software written with Verdi has been verified to handle faults and errors that may occur. Finally, Coq has also been used to implement microkernels, which are the basis for operating systems. The CertiKOS project [**?**] has developed several microkernels with security properties and proofs of correctness, including mC2, a verified concurrent microkernel. Operating systems allocate memory and computer resources and must defend against malicious processes. As demonstrated by these research projects, the Coq proof assistant can be extended for a diverse range of verified systems.

In this thesis, I will describe my additions to the verification framework MLCert. Building on the Perceptron implementation and existing proofs in MLCert, I present a verified implementation of the Kernel Perceptron algorithm, as well as two variants on the

Kernel Perceptron algorithm: a Budget Kernel Perceptron and a Description Kernel Perceptron. Background information for this thesis is provided in Chapter 2, with an introduction to the Perceptron and Kernel Perceptron algorithms, a more extended discussion of the challenges and tactics of machine learning verification, and the specifications for Budget Kernel Perceptrons and Description Kernel Perceptrons. Chapter 3 describes the methodology for implementing these algorithms in Coq. The proofs for these implementations and their performance results are detailed in Chapter 4. Finally, future work and conclusions are discussed in Chapter 5.

2 Background

This chapter aims to provide necessary background information in order to understand the remainder of this thesis. Sections 2.1 and 2.2 describe the Perceptron algorithm and its descendant, the Kernel Perceptron algorithm. Next, the challenges and methods of formal verification of machine learning are discussed in sections 2.3 and 2.4. Finally, modifications of the Kernel Perceptron algorithm, such as Budget Kernel Perceptrons in section 2.5 and Description Kernel Perceptrons in section 2.6, are detailed as improvements for the Kernel Perceptron.

2.1 The Perceptron Algorithm

The Perceptron algorithm was initially published in 1957 by Frank Rosenblatt. Highly influential in the early growth and development of the field of artificial intelligence, the Perceptron [**?**] provided one of the first methods for computers to iteratively learn to classify data into discrete categories. In order to classify n-dimensional data, the Perceptron learns a weight vector with n parameters as well as a bias term. Both the weight vector and bias consist of positive integers greater than or equal to zero which encode a linear hyperplane separating two or more categories in n-dimensional space.

Figure 2.1: Perceptron Pseudocode

```
Definition Perceptron (w:Params) (epochs:nat) (training_set:list (Label * Data)) :
for i in epochs:
    for j in size(training_set):
        (example, true_label) = training_set[j]
        predict = Predict(example, w)
        if predict != true_label:
            w = Update(w, training_set[j]).
```

The most basic Perceptron algorithm has the following steps. Before training, each parameter in the weight vector w is initialized to zero. The algorithm consists of two nested loops, as shown by the pseudocode in Figure 2.1. For this algorithm, we require the weight vector, the number of epochs, and the training set as input. The training set consists of labeled training examples, where the label is either 0 or 1. The outer loop uses the number of epochs to control the number of iterations over the entire training set. The inner loop executes for every training example in the training set and has two main steps. First, the n-dimensional data inside the training example and the weight vector are used to calculate the Perceptron's predicted label for this example, without using the training example's true label. The calculation for Perceptron prediction is shown in pseudocode in Figure 2.2 takes as input the weight vector and a single training example to produce a predicted label for the given example.

Figure 2.2: Perceptron Prediction Pseudocode

```
Definition Predict (example:Data) (w:Params):
    (bias, weight) = w
    bias + dot_product(weight, example).
```

The true label and the calculated predicted label are then compared. If both labels are the same, the Perceptron correctly classified this training example. However, if the predicted label is different, the weight vector is updated using the example to improve classification over time. This update is the second step of the inner loop.

The Perceptron algorithm is powerful despite its simplicity. However, there are limitations to the Perceptron's classification. The Perceptron cannot classify data that is not linearly separable with 100% accuracy, such as points classified by the exclusive-OR function, a binary operator that returns TRUE when its two inputs are the opposite of each

other. Despite the simplicity of exclusive-OR, the Perceptron cannot produce a model, or linear hyperplane, such that all the points classified by exclusive-OR as TRUE are also classified by the Perceptron as TRUE, and all the points classified by exclusive-OR as FALSE are also classified by the Perceptron as FALSE. The Perceptron can achieve at best 75% accuracy for the exclusive-OR function. This restriction on the Perceptron in part caused the first AI Winter, a severe decline in artificial intelligence research, due to unreasonable expectations for the Perceptron in fields where data is not linearly separable.

While the Perceptron is limited to classification of linearly separable data, the Perceptron Convergence Theorem states that the Perceptron is guaranteed to converge to a solution on linearly separable data. This property of the Perceptron algorithm was first proven on paper by Papert in 1961 [**?**] and Block in 1962 [**?**]. However, this proof was not verified by machine until 2017 through the work of Murphy, Gray, and Stewart [**?**] in the Coq proof assistant.

2.2 The Kernel Perceptron

The Kernel Perceptron improved on the Perceptron algorithm with the introduction of the kernel trick by Aizerman, Braverman, and Rozoner [**?**]. Using kernel functions, the classification of the Perceptron can be expanded to include non-linearly separable data. There are four main modifications for the Kernel Perceptron: prediction, kernel functions, parameter space, and weight vector update. Prediction for the Kernel Perceptron uses kernel functions to produce non-linear hyperplanes instead of linear hyperplanes. Because of kernalization, the prediction function changes so that in addition to the weight vector w and the current training example, the training set and training labels are required as well. The bias term is no longer necessary.

In the pseudocode KernelPredict function shown in Figure 2.3, K represents an arbitrary kernel function. Kernel functions form a class of functions that take two

Figure 2.3: Kernel Perceptron Prediction Pseudocode

```
Definition KernelPredict (example:Data) (w:KernelParams)
        (training_set:list (Label * Data) (K:Kernel):
    for i in size(training_set):
        (label, data) = training_set[i]
        sum += w[i] * label * K(example, data)
    return sum.
```

examples as input and produce a single value. By using non-linear kernel functions, the Kernel Perceptron can classify data that is not linearly separable. For example, the Kernel Perceptron can classify the exclusive-OR function with 100% accuracy using a quadratic kernel. By using kernel functions in prediction, the parameters used by the Kernel Perceptron have different cardinality compared to the parameters of the Perceptron. The Kernel Perceptron requires one parameter per training example for its classification, regardless of the dimensionality of the data. Therefore, the size of the weight vector is dependent on the size of the training set.

Finally, the weight vector update for the Kernel Perceptron is somewhat different from that of the Perceptron. When a training example is misclassified by the Kernel Perceptron, its parameter is incremented and the rest of the weight vector is unchanged. The full Kernel Perceptron algorithm is shown in Figure 2.4.

The Kernel Perceptron improves upon the Perceptron, but the Kernel Perceptron has its own limitations. The size of the parameter space for the Kernel Perceptron limits its usefulness in applications where memory is at a premium, as the size of the weight vector is dependent on the number of training examples, not the dimensionality of the training data. Also, the Kernel Perceptron, due to the use of kernel functions, is not guaranteed to

Figure 2.4: Kernel Perceptron Pseudocode

```
Definition KernelPerceptron (w:KernelParams) (epochs:nat)
        (training_set:list (Label * Data)) (K:Kernel):
for i in epochs:
    for j in size(training_set):
        (example, true_label) = training_set[j]
        predict = KernelPredict(example, w, training_set, K)
        if predict != true_label:
            w = Update(w, j).
```

converge to a solution or terminate, unlike the Perceptron algorithm. This means that the Perceptron Convergence Theorem cannot be used to prove the correctness of an implementation of the Kernel Perceptron.

2.3 Approaches to Machine Learning Verification

Verifying machine learning algorithms is a difficult problem in software engineering. Machine learning algorithms can produce thousands or millions of parameters in their models, which interact to classify data. The learning process for machine learning models can be tedious for humans to trace, and the model parameters generated during training are often not human-interpretable for manual verification of correctness. The authors of [**?**] describe how machine learning researchers do not agree on a standard definition of what human interpretability is or how models should be able to be interpreted by humans. Interpretability varies between algorithms and tends to be more difficult for neural algorithms, including the Perceptron family of algorithms. Some formal verification in the field of machine learning has been performed, as shown by [**?**], but many algorithms have

not been verified correct. Even for implementations with paper proofs of correctness, few have been proven correct by machine.

2.4 MLCert Framework

To facilitate the verification of machine learning algorithms, Bagnall and Stewart developed MLCert [**?**], an open-source tool built in the Coq proof assistant. MLCert employs generalization error to prove correctness for machine learning algorithms. Generalization error, as described by Levin, Tishby, and Solla [**?**], is an important indicator for the robustness of a machine learning model; algorithms with low generalization error can generalize from the training examples used in training to correctly classify unseen examples from the same domain of data in testing. Instead of trying to verify the model directly, MLCert verifies the generalization bounds for machine learning implementations built in its framework. Bounds on the generalization error indicate that an algorithm has bounds on mistakes made during testing, and the size of the parameter space contributes heavily to the tightness of the generalization bounds. Verified generalization bounds guarantee worst-case performance for a model. Previous work in the MLCert framework [**?**] has resulted in an implementation of the Perceptron algorithm with proofs to verify its generalization bounds. However, to the best of our knowledge, no one has implemented the Kernel Perceptron in Coq or formally proven its correctness of generalization bounds using machine-checked proofs.

The parameter space for the Kernel Perceptron is dependent on the number of training examples. This means that, as compared to the Perceptron algorithm, the Kernel Perceptron has very loose generalization bounds due to the increased size of the parameter space. The tightness of the generalization bounds matters because tighter bounds provide a stronger guarantee for performance. To tighten the generalization bounds of the Kernel Perceptron, one approach is to limit the number of parameters.

2.5 Budget Kernel Perceptron Algorithms

Budget Kernel Perceptrons are a family of algorithms which modify the Kernel Perceptron to limit the size of the parameters for the model while minimizing the impact on the accuracy of the model. Budget Kernel Perceptrons are often employed in areas where computer memory or resources are at a premium, and their modifications are customized for the requirements of their field. One strategy for Budget Kernel Perceptrons is to keep a set number of training examples for classification called support vectors, which specific rules for updating this set over time to maintain its size as the classification boundary changes. For the base Kernel Perceptron algorithm described in Section 2.2, every training example is a support vector. An example of a budget update rule is described in the article "Tracking the best hyperplane with a simple budget Perceptron", where the authors describe an update procedure where one support vector is selected at random for each replacement [**?**]. Another update rule is to always select the oldest support vector for replacement, as this support vector may no longer be necessary for correct classification.

Other strategies minimize the impact of removed support vectors through more creative means. Dekel, Shalev-Shwartz, and Singer present the Forgetron, where each support vector is "forgotten" over time by decreasing its impact on the model, which means that there is always an oldest support vector to be removed with the least influence on the model [**?**]. The authors of [**?**] compute the distance of each support vector from the classification hyperplane and remove the support vector with the greatest distance. Finally, the Projectron and Projectron++ algorithms described by [**?**] store both a support set and a projection onto the support set to reduce the overall size of the model. All these methods balance model size with increased classification error compared to the base Kernel Perceptron.

Of these studies, only Cramer, Kandola, and Singer [**?**] provide paper proofs of their Budget Kernel Perceptron's generalization bounds. The nature and function of Budget Kernel Perceptrons complements our research in proving generalization error for machine learning algorithms. By implementing a Budget Kernel Perceptron, the bounds on the size of the parameter space can improve the bounds on generalization error compared to the base Kernel Perceptron algorithm.

2.6 Description Kernel Perceptrons

In contrast to Budget Kernel Perceptrons, another method of encoding the Kernel Perceptron parameters involves description-length bounds. During training, the Kernel Perceptron will make the set number of mistakes, bounded by some value L. Using L, the number of support vectors is less than or equal to the number of mistakes, which will always be less than or equal to the size of the training set. This method requires a record of every misclassification made during training. Only training examples that were misclassified are included in the set of support vectors and used to calculate the hyperplane.

Description-length bounds describe the process of training, as each misclassification is recorded in order to produce and change the hyperplane. Instead of fixing the size of the support vectors on an arbitrary value, description-length bounds alow for every misclassification to be represented, while excluding training examples that are never misclassified and therefore not essential for the model. The generalization error for a Kernel Perceptron using description-length bounds is dependent on the number of misclassifications, which provides a bound on the size of the parameter space.

2.7 Chapter Summary

This chapter summarizes the background of this thesis, discussing the Perceptron and Kernel Perceptron algorithms, as well as variants of the Kernel Perceptron algorithm with

improved generalization bounds. Chapter 3 will next describe my extensions to the MLCert framework to implement three Kernel Perceptron algorithms: the base Kernel Perceptron algorithm, a Budget Kernel Perceptron, and a Description Kernel Perceptron, with generalization proofs for each implementation written in Coq.

3 METHODS

Blurb about what this chapter is about

3.1 Structure of Perceptron Implementations in MLCert

3.2 Kernel Perceptron Coq Implementation

3.3 Budget Kernel Perceptron Coq Implementation

3.4 Description Kernel Perceptron Coq Implementation

3.5 Chapter Summary

4 Proofs and Experimental Results

Blurb about what this chapter is about

4.1 Kernel Perceptron Generalization Proofs

4.2 Kernel Perceptron Haskell Extraction

4.3 Budget Kernel Perceptron Generalization Proofs

4.4 Budget Kernel Perceptron Haskell Extraction

4.5 Description Kernel Perceptron Generalization Proofs

4.6 Description Kernel Perceptron Haskell Extraction

4.7 Chapter Summary

5 CONCLUSIONS