9-2021

Characterizing Convolutional Neural Network Early-Learning and Accelerating Non-Adaptive, First-Order Methods with Localized Lagrangian Restricted Memory Level Bundling

Benjamin O. Morris

Characterizing Convolutional Neural Network Early Learning and Accelerating Non-Adaptive, First-Order Methods with Localized Lagrangian Restricted Memory Level Bundling

DISSERTATION

Benjamin O. Morris

AFIT-ENS-DS-21-S-049

DEPARTMENT OF THE AIR FORCE
AIR UNIVERSITY

AIR FORCE INSTITUTE OF TECHNOLOGY

Wright-Patterson Air Force Base, Ohio

AFIT-ENS-DS-21-S-049

CHARACTERIZING CONVOLUTIONAL NEURAL NETWORK EARLY LEARNING AND ACCELERATING NON-ADAPTIVE, FIRST-ORDER METHODS WITH LOCALIZED LAGRANGIAN RESTRICTED MEMORY LEVEL BUNDLING

DISSERTATION

Presented to the Faculty

Graduate School of Engineering and Management

Air Force Institute of Technology

Air University

Air Education and Training Command

in Partial Fulfillment of the Requirements for the

Degree of Doctor of Philosophy

Benjamin O. Morris

AFIT-ENS-DS-21-S-049

CHARACTERIZING CONVOLUTIONAL NEURAL NETWORK EARLY LEARNING AND ACCELERATING NON-ADAPTIVE, FIRST-ORDER METHODS WITH LOCALIZED LAGRANGIAN RESTRICTED MEMORY LEVEL BUNDLING

DISSERTATION

Benjamin O. Morris,

Committee Membership:

Jeffery D. Weir, PhD
Chair

Bruce A. Cox, PhD
Member

Aihua Wood, PhD
Member

Adedeji B. Badiru, PhD
Dean, Graduate School of Engineering and Management

AFIT-ENS-DS-21-S-049

Abstract

This dissertation studies the underlying optimization problem encountered during the early-learning stages of convolutional neural networks and introduces a training algorithm competitive with existing state-of-the-art methods.

A Design of Experiments method is introduced to systematically measure empirical second-order Lipschitz upper bound and region size estimates for local regions of convolutional neural network loss surfaces experienced during the early-learning stages. This method demonstrates that architecture choices can significantly impact the local loss surfaces traversed during training.

A Design of Experiments method is used to study the effects convolutional neural network architecture hyperparameters have on different optimization routines' abilities to effectively train and find solutions that generalize well during early learning, demonstrating a relationship between routine selection and network architecture.

A method to accelerate the early learning of non-adaptive, first-order optimization routines is developed. The method decomposes the neural network training problem into a series of unconstrained optimization problems within localized trailing Euclidean trust regions and allows non-adaptive methods to exhibit training results which are competitive with adaptive methods.

Dedication

I would like to thank my girlfriend for her kindness and encouragement throughout this time and my friends and family for their continued support.

This dissertation is dedicated to my late mother.

Acknowledgements

I would like to express my sincere appreciation to my research advisor, Dr. Jeffery Weir, for his guidance and mentorship throughout this process, as well as my research committee members, Dr. Bruce Cox and Dr. Aihua Wood, for their consistent support and insight.

Table of Contents

Page

Abstract iv

List of Figures xi

List of Tables xiv

I. Introduction 1

1.1 Artificial Neural Networks and Deep Learning 1
1.1.1 Common Architecture Components of Convolutional Neural Networks 2
1.2 Training Convolutional Neural Networks 4
1.3 Problems with Existing Convolutional Neural Network Architecture Design and Training Methods 5
1.3.1 Architecture Hyperparameter Challenges 6
1.3.2 Current State of First-Order Methods 7
1.3.3 Unknown Loss Surface Characteristics 7
1.4 Research Contributions 8

II. Literature Review 10

2.1 Convexity and Lipschitzness of the Gradient 10
2.2 Neural Network Loss Surface Characterization 11
2.3 Inexact Oracles and Training Convolutional Neural Networks 13
2.4 Design of Experiments and Its Applications 14
2.5 BatchNorm, Dropout, and Activation Functions 14
2.6 Tradeoff Between Width and Depth 15
2.7 Decision Variable Initialization 15
2.8 Regularization During Neural Network Training 16
2.9 Summary 16

III. Contribution I: Characterizing Convolutional Neural Network Early-Learning Loss Surfaces with Quality Engineering and Stochastic Walks 22

3.1 Introduction 22
3.2 Background 22
3.2.1 Lipschitz Gradients and Their Relationships with Training Neural Networks 22
3.2.2 Neural Network Early Learning 24
3.2.3 Design of Experiments and Its Applications 25
3.2.4 Exploring Neural Network Loss Surfaces 27

Page

3.2.5 Contributions . . . 28
3.3 Methodology . . . 29
3.3.1 Experimental Factor Considerations . . . 30
3.3.2 Experimental Settings . . . 32
3.3.3 Network Training . . . 33
3.3.4 Network Walk Generation . . . 34
3.3.5 Defining Local Regions and Identifying Walk Break Points . . . 36
3.3.6 Response Variables and Factor Effects . . . 36
3.3.7 Interpretation of Results . . . 37
3.4 Results . . . 37
3.4.1 Overview . . . 37
3.4.2 Multivariate Linear Regression . . . 41
3.4.3 Cross Listing Significant Terms . . . 43
3.5 Discussion . . . 46
3.5.1 Overall Observations . . . 46
3.5.2 Models of the Empirical Lipschitz Upper Bounds on the Local Gradients . . . 47
3.5.3 Models of the Number of Steps to Escape the Local Region . . . 49
3.5.4 Cross Listings of Factor Effects . . . 50
3.5.5 Limitations and Next Steps . . . 51
3.6 Conclusion . . . 53

IV. Contribution II: Identifying Convolutional Neural Network Architecture Hyperparameters that Significantly Impact Early Learning of First-Order Optimization Routines with Quality Engineering . . . 54

4.1 Introduction . . . 54
4.2 Background . . . 55
4.2.1 Underlying Optimization Problem . . . 55
4.2.2 Optimization Routine Choice . . . 55
4.2.3 Design of Experiments and Early Learning Background . . . 57
4.2.4 Contributions . . . 57
4.3 Methodology . . . 59
4.3.1 Overview . . . 59
4.3.2 Experimental Factor Considerations . . . 60
4.3.3 Experimental Design . . . 62
4.3.4 Optimization Routine Considerations . . . 63
4.3.5 Data Set Considerations . . . 64
4.3.6 Response Variables . . . 64
4.3.7 Experimental Runs . . . 65

Page

4.4 Results . . . 65
4.4.1 Divergent Instances . . . 66
4.4.2 95% Pairwise Student's T-Confidence Intervals Across All Non-Divergent Instances . . . 68
4.4.3 Pairwise Two-Tailed 95% Student's T-Test Comparisons . . . 72
4.4.4 Multivariate Linear Regression . . . 75
4.4.5 Meta-Learning Models for Train and Test Losses . . . 81
4.5 Discussion . . . 83
4.5.1 Overall Observations . . . 83
4.5.2 Limitations and Next Steps . . . 85
4.6 Conclusion . . . 86

V. Contribution III: Accelerating Non-Adaptive, First-Order Methods with Lagrangian Duality and Localized Euclidean Trust Regions . . . 88

5.1 Introduction . . . 88
5.2 Background . . . 89
5.2.1 Training Convolutional Neural Networks . . . 89
5.2.2 Optimization Routines Used to Train Neural Networks . . . 90
5.2.3 Early Learning Training Heuristics . . . 92
5.2.4 Optimization with Bundle-Level Methods . . . 92
5.2.5 Contributions . . . 94
5.3 Methodology . . . 95
5.3.1 Overview . . . 95
5.3.2 Oracle Call and Non-Adaptive Update . . . 96
5.3.3 Localized Trailing, Hypercube Trust Region . . . 97
5.3.4 Defining the Level Set . . . 98
5.3.5 Auxiliary Problem . . . 99
5.3.6 Secondary Update . . . 100
5.3.7 Accelerating Stochastic Gradient Descent Family Routines with Lagrangian Trust-Falls (LTF) . . . 101
5.4 Results . . . 102
5.4.1 Non-Convex Benchmark Function Experiments . . . 102
5.4.2 Logistic Regression . . . 106
5.4.3 Convolutional Neural Network Experiments . . . 109
5.5 Discussion . . . 114
5.5.1 Overall Observations . . . 114
5.5.2 Next Steps . . . 114
5.6 Conclusion . . . 115

Page

VI. Summary and Conclusions 116

VII. Appendix 118

7.1 Experiment Setup 118
7.2 Second-Order Linear Regression Models of Lipschitz Upper Bound Constants and Number of Steps Estimates 120
7.3 95%Pairwise Student's T-Tests between Alternative Optimizers for Final Training Loss 125
7.4 95%Pairwise Student's T-Tests between Alternative Optimizers for Total Test Loss 130
7.5 CIFAR-10 Full Train Loss Models 134
7.6 FashionMNIST Full Train Loss Models 141
7.7 CIFAR-10 Full Test Loss Models 148
7.8 FashionMNIST Full Test Loss Models 155

Bibliography 162

List of Figures

Figure		Page
1	Common convolutional neural network architecture components.	4
2	Training process of convolutional neural networks.	5
3	Distribution of factor settings exhibited in the 78 divergent instances.	38
4	Three-dimensional visualization of a Brownian walk for one experimental setup across local region on a convolutional neural network loss surface.	39
5	Five Brownian walks across the local region experienced during the early-learning stages.	40
6	Five Steepest ascent walks across the local region experienced during early-learning stages.	41
7	Hypothesized trajectory during early learning with and without BatchNorm present.	49
8	Factors associated with divergent instances.	67
9	95% student t-confidence intervals of final training loss between alternative optimization routines.	70
10	95% student t-confidence intervals of total test loss between alternative optimization routines.	71
11	Adam pairwise, two-tailed 95% student's t-tests of final training loss between alternative optimization routines.	73
12	Adam pairwise, two-tailed 95% student's t-tests of total test loss between alternative optimization routines.	74
13	Meta-learning model framework.	82
14	Optimization trajectories of 25 iterations on the Himmelblau function.	104
15	Optimization trajectories of 500 iterations on the Rosenbrock function.	106

Figure Page

16 Early iterations of logistic regression training results on the MNIST data set. . . . 107

17 Early iterations of logistic regression test results on the MNIST data set. . . . 108

18 CNN early training results on FashionMNIST data set. . . . 110

19 CNN early test results on FashionMNIST data set. . . . 111

20 CNN early training results on the CIFAR-10 data set. . . . 112

21 CNN early test results on the CIFAR-10 data set. . . . 113

22 SGD-Nesterov pairwise, two-tailed 95% student's t-tests of final training loss between alternative optimization routines. . . . 125

23 SGD-Momentum pairwise, two-tailed 95% student's t-tests of final training loss between alternative optimization routines. . . . 126

24 SGD-Vanilla pairwise, two-tailed 95% student's t-tests of final training loss between alternative optimization routines. . . . 127

25 RMSProp pairwise, two-tailed 95% student's t-tests of final training loss between alternative optimization routines. . . . 128

26 AdaGrad pairwise, two-tailed 95% student's t-tests of final training loss between alternative optimization routines. . . . 129

27 SGD-Nesterov pairwise, two-tailed 95% student's t-tests of total test loss between alternative optimization routines. . . . 130

28 SGD-Momentum pairwise, two-tailed 95% student's t-tests of total test loss between alternative optimization routines. . . . 131

29 SGD-Vanilla pairwise, two-tailed 95% student's t-tests of total test loss between alternative optimization routines. . . . 132

Figure Page

30 RMSProp pairwise, two-tailed 95% student's t-tests of total test loss between alternative optimization routines. 133

31 AdaGrad pairwise, two-tailed 95% student's t-tests of total test loss between alternative optimization routines. 134

List of Tables

Table		Page
1	Reference summary table	17
2	Two-way interaction alias structure.	32
3	Experimental factor settings.	33
4	Third-order multivariate linear regression model summaries.	42
5	Full second-order multivariate linear regression model summaries.	42
6	Cross listing of significant terms across both second-order Lipschitz bound models at $\alpha = 0.01$. Factor effects displayed are those at the high, +1, setting; for interactions, this occurs when both factors are set to their high, +1, settings or when both factors are set to their low, -1, settings.	44
7	Cross listing of significant terms across both second-order number of steps to escape the local region models at $\alpha = 0.01$. Factor effects displayed are those at the high, +1, setting; for interactions, this occurs when both factors are set to their high, +1, settings or when both factors are set to their low, -1, settings.	45
8	Cross listing of significant terms found across all second-order regression models with a significance level of $\alpha = 0.01$. Factor effects displayed are those at the high, +1, setting; for interactions, this occurs when both factors are set to their high, +1, settings or when both factors are set to their low, -1, settings.	46
9	Two-way interaction alias structure.	62
10	Experimental factor settings.	63
11	Default hyperparameter settings for each optimization routine.	64
12	Number of divergent cases where each optimization routine that exhibited divergence, by data set.	68

Table Page

13 CIFAR-10 final training loss regression summaries. 75

14 FashionMNIST final training loss regression summaries. 76

15 CIFAR-10 total test loss regression summaries. 76

16 FashionMNIST total test loss regression summaries. 77

17 CIFAR-10 final training loss significant terms cross-listed across all second-order regression models at a significance level of $\alpha = 0.01$. 79

18 FashionMNIST final training loss significant terms cross-listed across all second-order regression models at a significance level of $\alpha = 0.01$. 79

19 CIFAR-10 total test loss significant terms cross-listed across all second-order regression models at a significance level of $\alpha = 0.01$. 80

20 FashionMNIST total test loss significant terms cross-listed across all second-order regression models at a significance level of $\alpha = 0.01$. 80

21 Cross-listing of significant factor effects which affect curvature measurements in the directions studied and affect final training losses across FashionMNIST and CIFAR-10. 81

22 Meta-learning model hit rates by data set, loss type combination used to predict optimization routine(s) achieving minimum mean loss values. Hit Rate, A corresponds to a meta-learning model's ability to predict the optimization routine that achieves the single lowest empirical mean loss value. Hit Rate, B corresponds to a meta-learning model's ability to predict an optimization routine that achieves one of the top two lowest empirical mean loss values. 83

23 Optimization routine settings used for routines to traverse the Himmelblau function. 103

24 Optimization routine settings used for routines to traverse the Rosenbrock function. 105

Table Page

25 Optimization routine settings used for routines to train MNIST logistic regression. 106

26 Optimization routine settings used for routines to train CNNs on FashionMNIST and CIFAR-10. 109

28 Gaussian path second-order linear regression model of Lipschitz Upper Bound Constants. 121

29 Gaussian path second-order linear regression model of number of steps needed to escape local region. 122

30 Steepest ascent path reduced second-order linear regression model of Lipschitz Upper Bound Constants. 123

31 Steepest ascent path reduced second-order linear regression model of number of steps. 124

32 Stochastic Gradient Descent with Nesterov's Accelerated Gradient Full Second-Order Train Loss Model 135

33 Stochastic Gradient Descent with Momentum Full Second-Order Train Loss Model 136

34 Stochastic Gradient Descent Full Second-Order Train Loss Model 137

35 RMSProp Full Second-Order Train Loss Model 138

36 Adam Full Second-Order Train Loss Model 139

37 AdaGrad Full Second-Order Train Loss Model 140

38 Stochastic Gradient Descent with Nesterov's Accelerated Gradient Full Second-Order Train Loss Model 142

39 Stochastic Gradient Descent with Momentum Full Second-Order Train Loss Model 143

40 Stochastic Gradient Descent Full Second-Order Train Loss Model 144

41 RMSProp Full Second-Order Train Loss Model 145

Table Page

42 Adam Full Second-Order Train Loss Model 146

43 AdaGrad Full Second-Order Train Loss Model 147

44 Stochastic Gradient Descent with Nesterov's Accelerated Gradient Full Second-Order Test Loss Model 149

45 Stochastic Gradient Descent with Momentum Full Second-Order Test Loss Model 150

46 Stochastic Gradient Descent Full Second-Order Test Loss Model 151

47 RMSProp Full Second-Order Test Loss Model 152

48 Adam Full Second-Order Test Loss Model 153

49 AdaGrad Full Second-Order Test Loss Model 154

50 Stochastic Gradient Descent with Nesterov's Accelerated Gradient Full Second-Order Test Loss Model 156

51 Stochastic Gradient Descent with Momentum Full Second-Order Test Loss Model 157

52 Stochastic Gradient Descent Full Second-Order Test Loss Model 158

53 RMSProp Full Second-Order Test Loss Model 159

54 Adam Full Second-Order Test Loss Model 160

55 AdaGrad Full Second-Order Test Loss Model 161

CHARACTERIZING CONVOLUTIONAL NEURAL NETWORK EARLY LEARNING AND ACCELERATING NON-ADAPTIVE, FIRST-ORDER METHODS WITH LOCALIZED LAGRANGIAN RESTRICTED MEMORY LEVEL BUNDLING

I. Introduction

1.1 Artificial Neural Networks and Deep Learning

Artificial neural networks are loosely modeled after connections between nerve cells known as neurons in an organism's nervous system [3]. In a biological setting, neurons communicate with each other through chemical transmissions across very small distances known as synapses that separate individual neurons [11]. Once information is passed from the delivering neuron (the presynaptic neuron) to the receiving neuron (the postsynaptic neuron), the postsynaptic neuron will then exhibit chemical changes that can then be translated into further communication to subsequent neurons [11]. Two unique features that neurons have to aid in this chemical communication are dendrites and axons; dendrites allow a neuron to receive chemical information from other neurons while axons allow a neuron to pass information to other neurons [11].

Inspired by this biological process, McCulloch and Pitts [64] are largely credited with first introducing the idea of artificial neural networks in a 1943 paper which described how neurons might be mathematically described. Their original paper later led to the development of what has been often considered the first artificial neural network, the perceptron introduced by Rosenblatt [79]; the perceptron laid

the groundwork for modern-day neural networks. In modern-day artificial neural networks, the ideas of dendrites and axons from biological settings are represented as weighted arcs; synapses are represented as separations between layers of artificial neurons; and the neurons themselves are computational processing elements that create signals, known as activations that are provided as information inputs to later layers in the network [3]. Deep learning networks were later introduced with the idea of hidden layers, layers that provide additional modeling capability and exist between the input layer and the output layer [85]. Convolutional neural networks (CNNs) were introduced by Fukushima [28], [85]. CNNs are now widely used in image processing applications, producing in specific cases, "superhuman" classification results and have made great strides in a variety of disciplines such as medical imaging classification and computer vision tasks [85], [36], [5].

1.1.1 Common Architecture Components of Convolutional Neural Networks

The problem of building and tuning CNNs requires hyperparameter selection related to both the network architecture and the optimization routine using to train the model. Together, these complex decisions dictate how well a specific CNN can perform both during, both, the training and validation stages. Below, is a brief description of some of the common components used in CNNs architectures [2]. Figure 1 illustrates several of these common components.

- Convolutional layers are used to extract image-specific features through the use of kernels.
  - Each kernel is two-dimensional and is much smaller than the spatial dimension of the input layer. Several kernels can exist in a single convo-

lutional layer, each building feature maps. This makes the layer itself a three-dimensional element of the network.

  - Kernels in a convolutional layer slide across the input layer based on the size of its stride, extracting local information.

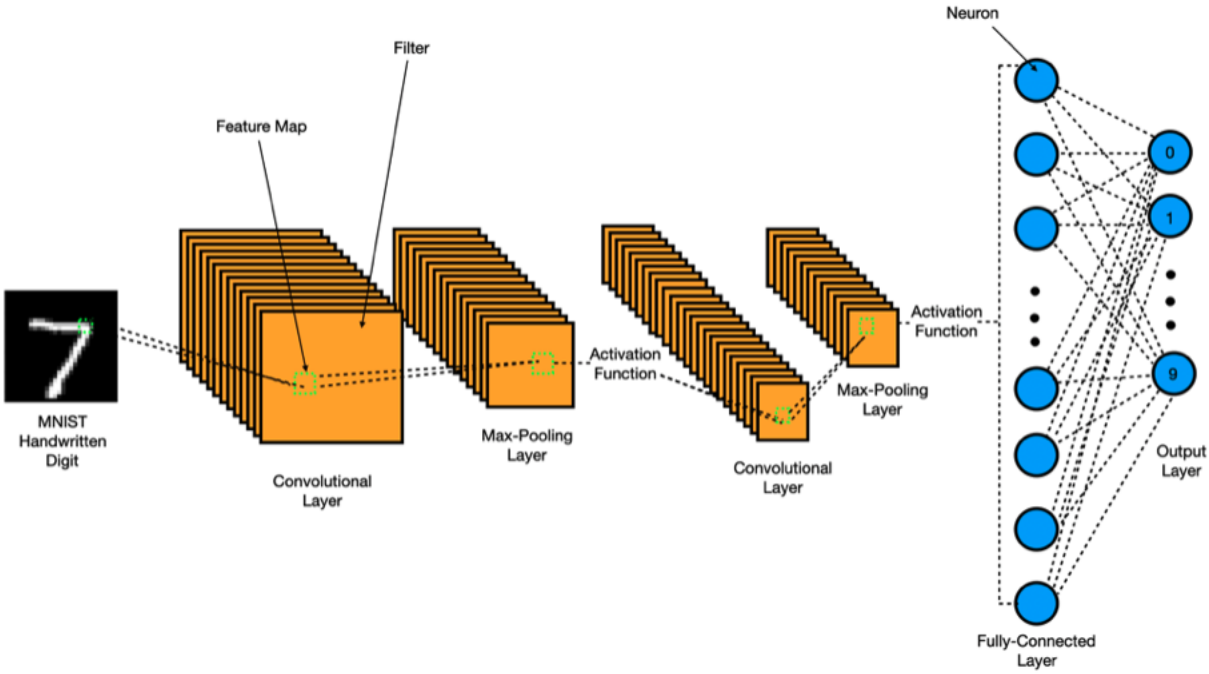
- Fully-connected layers flatten image information that has been passed through previous layers. These layers often appear towards the end of the network's layering, and they are used to prepare for a classification decision to be made.

- Pooling layers often follow a convolutional layer and act as a data reduction tool. These layers create small clusters of a user-defined size that help downsize the amount of information passed along to later layers in ways that aim to only consider the most salient observations from the input layer. The use of pooling layers can help with overfitting and reducing the computational burden exhibited in large-scale networks.

- Activation functions can be used to introduce non-linearity into the network. Allowing for non-linearity not only helps the network compensate for very hard relationships found in the image data but also mimics natural thresholds observed in human brain neurons. Activation functions control the level of influence a neuron has on a specific classification.

- Loss functions are found at the end of a neural network's architecture. These functions penalize the network for poor classifications, impacting how gradient information is passed via backpropagation through the network to update the network's decision variables at each iteration of training.

Figure 1. Common convolutional neural network architecture components.



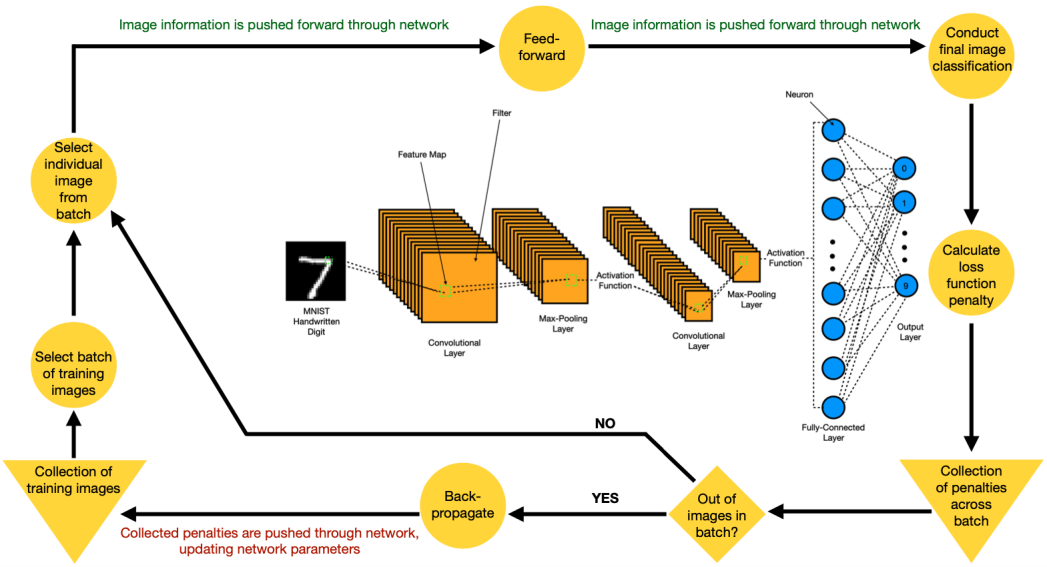
1.2 Training Convolutional Neural Networks

CNNs use backpropagation, an algorithm first introduced by Rumelhart et al. [80], to make updates to its decision variables during training. The method uses a computational graph to represent connections made between different neurons found in subsequent layers of the graph. CNNs that are often intended to be used as models which classify image labels. These networks traditionally begin with a convolutional layer used to translate a raw image into a series of neurons. Subsequent layers take this initial interpretation and transform the information further through layer-specific transformations, eventually outputting a final prediction for classification that is then used as an input for the network's loss function. The loss function penalizes the network for poor classifications, pushing this penalty back through the network using

backpropagation, a numerical differentiation method, to make updates to the decision variables of the network for better future classifications. This process repeats itself until all training images have been used to train the network.

Batch training is a common means of pushing training images through the network more quickly; when batch training is employed, an entire set of images is pushed through the network before an update to the decision variables occurs. Instead, a summary of the penalties incurred from the loss function over the entire batch is used to make updates to the decisions variables of the network. Figure 2 illustrates the training process.

Figure 2. Training process of convolutional neural networks.



1.3 Problems with Existing Convolutional Neural Network Architecture Design and Training Methods

There are a variety of considerations that must be made both during the design and training phases when building a CNN model. The training problem associated with these networks is an optimization problem, with the number of variables equal to the number of trainable weight and bias variables associated across the entire network.

This problem can easily become one of thousands or even millions of variables, and each of these variables must be optimized efficiently in order to develop a model that performs well. Two of the biggest considerations affecting the overall training process are the specifications related to how the neural network is set up (the network's architecture components and hyperparameters) as well as the optimization routine used train the network.

1.3.1 Architecture Hyperparameter Challenges

There are many different network architecture components that can impact a CNN's success to accurately provide classification information. These components are generally selected by the model-builder and should be chosen in a way that best meets the needs of the input data as well as the needs of the classification problem at hand. Decisions related to the number of layers, types of layers, and sizes of each layer impact the size of the model's trainable decision variables. Activation function choices impact the ways in which the data is interpreted through successive layers of the model, and the loss function dictates the error that is communicated through the network using the backpropagation algorithm. These and several other decisions impact the underlying loss surface of the neural network. Often-times complex models are required for real-world classification problems; the added complexity of these models introduces challenging problems for optimization routines to solve. In fact, even in small-scale instances, the problem of finding a global minima on the loss surface is considered to be an NP-Hard problem [12]. Some of the reasons for the difficulty in training CNNs include a non-convex loss surface, computational tractability due to sheer size of large-scale neural networks, and lack of knowledge related to the characteristics exhibited by the underlying loss surface. Many of the routines developed to date must be computationally cheap and must work with limited information

provided by a noisy first-order oracle.

1.3.2 Current State of First-Order Methods

CNNs are often trained via first-order optimization routines. These routines are popular due to their computational cheapness and ability to still operate in the large-scale optimization environment that many neural network instances exhibit. At each iteration of training, noisy, first-order oracle information is produced when using mini-batch sampling methods of training. Gradient information is passed through the network via backpropogation, and updates are made to the decision variables using the selected optimization routine. Non-adaptive methods of training use global hyperparameters which must be adjusted manually during training; adaptive methods make online adjustments to hyperparameters over the course of training [91].

Many adaptive, first-order algorithms have become very popular due to their ability to frequently reduce training loss more quickly than non-adaptive methods [91]. These methods make updates to the decision variables based on information that has been learned as training progresses [91]. Several of these algorithms have seen great successes; however, critics claim that these routines can overfit training data. In other words, they tune the decision variables of the model to fit the training data too closely [91]. A variety of different methods can be employed to help with the problem of overfitting; however, every method has its limitations.

1.3.3 Unknown Loss Surface Characteristics

In general, the loss surface of CNNs are assumed to be represented by a FO oracle that yields function and first-order derivative evaluations at any given point on the loss surface. Unfortunately, due to this representation, most optimization routines are forced to be largely "black-box" in that they are only allowed to operate with a very

limited amount of information. Many researchers have worked to try to characterize portions of the loss surface as well as how different model choices impact the loss surface's character; however, much is still unknown resulting in sometimes inconsistent behavior from the solutions found during training using different methods.

1.4 Research Contributions

This dissertation studies the early learning stages of CNNs. The first contribution focuses on extending methods of empirically exploring the local loss surface of CNNs experienced during the early-learning stages. The goal of this contribution is to identify CNN architecture hyperparameters and combinations of hyperparameters that significantly impact local estimates of curvature and local region size estimates through the use of a Design of Experiments (DoE) methodology. The results show that the selection of a variety of different hyperparameters plays a significant role in affecting the underlying loss surfaces traversed during the early stages of training CNNs.

The second contribution identifies ways in which different first-order optimization routines that are popularly used to train CNNs are affected by the architecture hyperparameters studied in the first contribution. This contribution draws a relationship between CNN architecture hyperparameters and optimization routine selection. Results show that the different optimization routines studied can be affected in both similar and different ways by several common architecture hyperparameters, emphasizing the need to select an optimization routine that works best with the architecture components selected for the network design of an instance.

Finally, the third contribution focuses on developing a method for improving the performance of CNN non-adaptive optimization routines' training capabilities in the early-learning stages. By adapting a bundling method which relies on a Lagrangian

dual expression to the non-convex loss surfaces of CNNs, this contribution not only yields a training routine which can accelerate early learning of non-adaptive methods, but it also demonstrates that updates more similar to convex optimization can be made within localized regions of CNN loss surfaces.

Together, these three contributions seek to improve convolutional neural network training during the early-learning stages by both providing new information as to how the training problem behaves during the early stages of training from the perspectives of local loss surface characteristics and optimization routine success while also introducing a new training method to accelerate non-adaptive updates.

II. Literature Review

2.1 Convexity and Lipschitzness of the Gradient

Bazaraa et al. [7] define convexity for a function, $f : \theta \to \mathbb{R}$ where $\theta \in \mathbb{R}^n$, as:

Definition 1 (Convexity)

$$f(\lambda\theta_y + (1-\lambda)\theta_x) \leq \lambda f(\theta_y) + (1-\lambda) f(\theta_x) \tag{1}$$

for each $\theta_y, \theta_x \in \theta$ *and for each* $\lambda \in (0,1)$

For twice-differentiable functions the Hessian, the matrix of second-order partial derivatives, can be used to help characterize critical points.

Definition 2 (Critical Point) *Critical points are defined as points at which the gradient of the function (first-order derivative) is equal to zero, i.e. when* $\nabla f(\theta) = 0$

In general, neural network loss surfaces are highly non-convex; it is hard to characterize a critical point that is found with just first-order information [17]. If the loss function, f, is twice differentiable, then at some point, $\theta = \theta_0 + \epsilon$, f can be approximated using its second-order Taylor series expansion as follows [7]:

$$f(\theta) = f(\theta_0) + \nabla f(\theta_0)^T \epsilon + \frac{1}{2}\epsilon^T H(\theta_0)\epsilon \tag{2}$$

Several Newton-variant routines have been applied to optimizing neural networks using this second-order information. A primary drawback of many of these routines is that the Hessian can be computationally expensive to calculate in large-scale instances. Even if it is computationally feasible to find the Hessian, routines can still have significant issues [7], [25]. These issues and others have led to the development of

alternative optimization routines; however, the Hessian has been utilized as a means of characterizing levels of convexity exhibited for portions of the neural network loss surface.

Lipschitzness of a loss surface's gradient, sometimes referred to as β-Smoothness, describes the rate of change that the loss surface gradient exhibits within the considered domain. In instances where an upper bound, β, exists on this rate of change, the gradient is known to not change dramatically within the domain [6], [16]:

Definition 3 (β-Smoothness) *A continuously differentiable function, f, is β-Smooth if the gradient ∇f is β-Lipschitz, namely:*

$$\|\nabla f(\theta_y) - \nabla f(\theta_x)\| \leq \beta \|\theta_y - \theta_x\| \tag{3}$$

When f is twice-differentiable, β-Smoothness is equivalent to obtaining an upper bound, β, on the largest eigenvalues of the Hessian of f at any points within the considered domain [16]. In other words:

$$\nabla^2 f(\theta) \preccurlyeq \beta I \ \ \forall \theta \in \Theta \tag{4}$$

2.2 Neural Network Loss Surface Characterization

Many modern optimization methods used to train large-scale instances require a compact, convex set and a Lipschitz gradient in addition to a convex loss surface [43], [9]. Characteristics of CNN loss surfaces can be hard to guarantee; however, studying different properties of these loss surfaces is an active area of research.

Milne [65] show that with ReLu activation functions and a regularized cost function, with certain conditions met, piecewise strong convexity can be found over a specific set that can in some cases include all minima. Using the idea of intrin-

sic dimension introduced by Li, Ding and Sun [57], Fort and Scherlis [25] use low-dimensional hyperplanes to explore the curvature surrounding portions of the neural network loss surface, identifying favorable, hollow regions within the architectures analyzed that exhibit large amounts of positive curvature they deemed the Goldilocks Zone. Ghorbani et al. [29] measure local curvature by estimating the full spectrum of the Hessian. Sagun et al. [82] study the effects that both data and architecture have on the eigenvalues of the loss surface's Hessian. Similarly, Pascanu et al. [72] study the density of eigenvalues of the Hessian at different critical points found through experimentation, comparing the fraction of negative eigenvalues of the Hessian with the training loss observed at critical points. This work built off of work conducted by Bray and Dean [15], who analyze the number of critical points found over Gaussian fields, rank-ordering critical points based on the number of negative eigenvalues found in their respective Hessian matrices. Li, Xu, Taylor, Studer and Goldstein [59] utilize a ratio of the minimum and maximum eigenvalues across portions of the loss surfaces studied in order to identify the level of convexity exhibited in regions surrounding local minima.

Safran and Shamir [81] show that even though neural network loss surfaces are highly non-convex, there can still exist monotonically decreasing paths from initialization points to local minima and that over-parameterizing neural networks could yield easier training, even though larger numbers of variables are introduced. Choromanska et al. [18] empirically show that for large-sized networks, it is very probable that the different minima achieved are very similar in terms of their performance and that reaching bad minima decreases in likelihood as the architecture grows in size.

Due to the very large dimensions exhibited in many neural network decision spaces, it is not feasible to explore these loss surfaces holistically. Instead, several researchers have sought to identify characteristics through empirical findings in local regions.

Goodfellow et al. [32] study trajectories of stochastic gradient descent, showing there exists monotonically decreasing, straight-line trajectories between initialization points and locally optimal solutions. Im et al. [41] utilize barycentric and bilinear interpolation to develop two-dimensional surfaces which are subsequently used to analyze the relationships between the initial and final decision variables found by different optimization methods. Smith and Topin [86] and Li, Xu, Taylor, Studer and Goldstein [59] study different minima found during training.

2.3 Inexact Oracles and Training Convolutional Neural Networks

First-order methods by definition utilize information related to a loss function's gradient, $\nabla f(\theta)$, during the process of optimization. The gradient of a function points an optimization routine in favorable directions. Stochastic Gradient Descent [13], Momentum [75], Nesterov's Accelerated Gradient [68], AdaGrad [23], RMSProp [89], and Adam [48] are some of the most popular methods for training convolutional neural networks. Each of these methods utilizes first-order oracle information to make updates. Stochastic Gradient Descent, Momentum, and Nesterov's Accelerated Gradient are all considered non-adaptive methods whereas AdaGrad, RMSProp, and Adam are considered adaptive methods [91]. Non-adaptive methods use global hyperparameters throughout optimization whereas adaptive methods allow for online adjustments to be made to hyperparameters with the intent of fitting the local loss surface's geometry more closely than non-adaptive methods. Although results often show that adaptive methods can yield very good results to reduce training loss, they have been criticized at having a tendency to overfit the training data [91].

2.4 Design of Experiments and Its Applications

Design of Experiments (DOE) has a rich history dating back to the early 20^{th} century [67]. The process of applying DOE to an area of research is intended to provide a methodology of collecting necessary data that allows for objective statistical conclusions to be made after the experimentation phase has finished. Experimental runs are designed to ensure that specific statistical properties are maintained for later analysis needs. Often, a regression model or response surface is a desired output from conducting the experiments so that the model-builders can predict the response for a future combination of the factors. Two-level, full-factorial designs are a broad field of designs that are popularly employed across different disciplines today; these designs often appear as 2^k designs in the literature Montgomery [67].

2.5 BatchNorm, Dropout, and Activation Functions

Ioffe and Szegedy [42] show that batch normalization layers (colloquially referred to as BatchNorm layers) improve training by reducing covariate shifts. Santurkar et al. [83] show that BatchNorm layers improve training by smoothing the loss surface's gradient. Ghorbani et al. [29] further show that BatchNorm layers yield a smoothing effect on neural network loss surfaces.

Srivastava et al. [87] propose dropout layers to manage overfitting by reducing the number of trainable decision variables in any single layer of the network during an iteration of training. These layers stochastically remove neurons throughout the training process.

Nwankpa et al. [69] provide a summary of many of the popularly used activation functions used today, including sigmoid, Tanh, softmax, and ReLu as well as some of its variants. Combinations of different types of activations in conjunction with the size of the layer are important considerations when designing a neural network as

they can dramatically impact the loss surface [25], [47], [29].

2.6 Tradeoff Between Width and Depth

Width and depth of a neural network are two primary considerations when building an architecture. There is an increasing amount of research showing relationship sbetween width and ease of training. Some theoretical results regarding architectures have pointed towards extremely wide networks being easier to train; Li et al. [58], Oymak and Soltanolkotabi [71], and Yu and Chen [96] show the benefits of network width on the quality of solutions found on the loss surface under certain conditions .

An often competing architectural component to width, depth has also been shown to be an important consideration in designing a network. Safran and Shamir [81] show that deeper networks can approximate functions better than shallower networks, even if the shallower networks are wider. Liang and Srikant [61] show that for a combination of ReLu activation functions and binary step units, deeper networks are preferred for function estimation. Dinh et al. [21] show deep learning architectures with sharp minima are still often capable of generalizing well. In contrast Li, Xu, Taylor, Studer and Goldstein [59] show that network depth increases chaotic behavior over the loss surface for ResNet architectures.

2.7 Decision Variable Initialization

Initialization methods are well-studied due to the large impacts that these methods can have on the final solutions achieved by optimization routines during training [72], [53], [88]. Due to the complex nature of neural network loss surfaces, the initialization point can have dramatic effects on the success of training. Two of the most widely used methods today are Xavier Initialization [30] and He Initialization [37]. Mishkin and Matas propose Layer-Sequential Unit-Variance (LSUV) Initialization [66]. Sussillo

and Abbott [88] introduce Random Walk Initialization to aid in the control of the vanishing gradient problem.

2.8 Regularization During Neural Network Training

The study of regularizing CNN decision variables during training is an active area of research. Weight decay methods impose constraint-like characteristics on neural network decision variables through the use of penalty components incorporated into the loss function [52], [63], [97]. Weight restrictions have shown to yield good results, with some evidence supporting that small decision variables might be better than larger ones as they ensure the network's capabilities are not dictated by a small number of dominant decision variables [87]. Pokutta et al. [74] show that using Conditional Gradient Descent, also known as the Frank-Wolfe Algorithm, with compact, convex constraint sets can yield state-of-the-art results when training CNNs.

2.9 Summary

The references addressed in this section in conjunction with additional references introduced in the following chapters are summarized in Table 1. These references are categorized by broad themes studied throughout the research presented in this dissertation.

Table 1. Reference summary table

Author(s)	Year	Neural Network Hyperparameters	Optimization Algorithms and Theory	Loss Surface Characterization	Design of Experiments	Early-Learning
Achille et al.	2019					x
Agarwal et al.	2016		x			
Aggarwal	2018	x	x	x		
Akiba et al.	2017		x			
Alom et al.	2019	x	x			
Apostol	1974		x	x		
Bazaraa et al.	2006		x	x		
Ben-Tal and Nemirovski	2005		x			
Ben-Tal and Nemirovski	2019		x	x		
Bengio et al.	1994	x	x			
Bianchi et al.	2008	Biological Theory				
Blum and Rivest	1992	x	x			
Bottou	1998		x	x		
Bottou et al.	2018		x			
Bray and Dean	2006		x	x		

Table 1. Reference summary table

Bubeck	2015		x	x		
Carmon et al.	2018		x			
Choromanska et al.	2015			x		
Cook	1977				x	
Dauphin et al.	2014		x	x		
Dinh et al.	2017	x				
Do and Artières	2012		x			
Duchi et al.	2011		x			
Durakovic	2017				x	
Fort and Scherlis	2019			x		
Frankle and Carbin	2019					x
Frankle et al.	2020					x
Fukushima	1980	x				
Ghorbani et al.	2019			x		
Glorot and Bengio	2010	x				
Golatkar et al.	2019					x
Goodfellow et al.	2015			x		
Gron	2017	x	x	x		
Hardt et al.	2016		x			
He et al.	2020			x		
He et al.	2015*a*	x				
He et al.	2015*b*	x				
He et al.	2021	x				
Hinton et al.	2012		x			
Hochreiter and Schmidhuber	1997			x		

Table 1. Reference summary table

Im et al.	2019		x	x		
Ioffe and Szegedy	2015	x				
Juditsky and Nemirovski	2011		x			
Karpathy	2017		x			
Keskar et al.	2017	x		x		
Keskar and Socher	2017		x			
Khan et al.	2020	x				
Kingma and Ba	2015		x			
Kiwiel	1995		x			
Kiwiel	2010		x			
Krizhevsky et al.	n.d.	Benchmark Data Set				
Krizhevsky et al.	2012	x				
Kumar	2017	x				
LeCun et al.	2000	Survey				
LeCun and Cortes	2010	Benchmark Data Set				
Lee et al.	2018		x	x		
Li, Xu, Taylor, Studer and Goldstein	2018	x				
Li et al.	2021	x		x		
Li, Ding and Sun	2018			x		
Liang et al.	2020		x			
Liang and Srikant	2017	x				
Liang et al.	2018			x		
Loshchilov and Hutter	2018		x			
McCulloch and Pitts	1943	x				
Milne	2019		x	x		

Table 1. Reference summary table

Mishkin and Matas	2016	x				
Montgomery	2005				x	
Nesterov	1983		x			
Nwankpa et al.	2018	x				
Oliveira and Sagastizabal	2014		x			
Oymak and Soltanolkotabi	2020	x	x			
Pascanu et al.	2013	x	x			
Paszke et al.	2019	PyTorch Neural Network Framework				
Pokutta et al.	2020		x			
Polyak	1964		x			
Pontes et al.	2016	x			x	
Ranzato et al.	2007	x				
Riesenhuber and Poggio	1999	Biological Theory				
Rosenblatt	1958	x				
Rumelhart et al.	1986		x			
Safran and Shamir	2016	x		x		
Sagun et al.	2018	x		x		
Santurkar et al.	n.d.	x		x		
Scherer et al.	2010	x				
Schmidhuber	2014	Survey				
Smith and Topin	2017			x		
Srivastava et al.	2014	x				
Sussillo and Abbott	2014	x				
Tieleman and Hinton	2012		x			
Tsai et al.	2006				x	

Table 1. Reference summary table

Wilson et al.	2017		x			
Wolpert and Macready	1997		x			
Wu et al.	2016		x			
Xiao et al.	2017	Benchmark Data Set				
Yang and Lee	1999	x			x	
Yu and Chen	1995	x				
Zhang, Wang, Xu and Grosse	2019		x			
Zhang, Chen, Yao, Ge and Dong	2019	x			x	

III. Contribution I: Characterizing Convolutional Neural Network Early-Learning Loss Surfaces with Quality Engineering and Stochastic Walks

3.1 Introduction

The underlying optimization problem associated with training convolutional neural networks is known to be challenging due to the high-dimension, non-convex loss surfaces often exhibited in real-world applications. Currently, most state-of-the-art optimization routines employed to train neural networks rely on their ability to exploit very limited amounts of information during training to make meaningful progress. In this chapter, we extend the use of Design of Experiments (DOE) to empirically show that measures of both curvature and size estimates over local regions of the underlying loss surface are significantly impacted in directions that are not along the original trajectory taken during training. We identify architecture hyperparameters that impact these characteristics, providing insight both for future network design and optimization routine development.

3.2 Background

3.2.1 Lipschitz Gradients and Their Relationships with Training Neural Networks

Neural network loss surfaces exhibit expansive flat regions and sharp local minima [83], [57]. Second-order Lipschitzness, also known as β-Smoothness and found in Definition 4, is a Lipschitz bound of the gradient required to make convergence guarantees for many modern optimization routines. Loss functions which have Lipschitz gradients are those with gradients that change less rapidly. For first-order optimization routines, this property can be extremely useful - a loss function gradient that is

Lipschitz allows an optimization routine to make larger updates with less uncertainty. This can ultimately reduce the risk of a routine quickly entering flat regions that pose a difficult problem to escape; in other instances, it can help the routine avoid regions which would lead it to exhibit divergent or otherwise unstable behavior [83].

In general, lower β constants correspond to function gradients that change less rapidly [16].

Definition 4 (β-Smoothness) *A continuously differentiable function, $f(\theta)$, is β-Smooth if the gradient $\nabla f(\theta)$ is β-Lipschitz, namely:*

$$\|\nabla f(\theta_y) - \nabla f(\theta_x)\|_2 \leq \beta \|\theta_y - \theta_x\|_2 \tag{5}$$

Where θ_x and θ_y are two sets of decision variables found on the local loss surface.

For instances where $f(\theta)$ is twice-differentiable and convex, β-Smoothness is equivalent to obtaining an upper bound, β, on the largest eigenvalue of the Hessian of $f(\theta)$ at any point within its domain [16].

Santurkar et al. [83] study the effects BatchNorm layers can have on the β-Smoothness of the loss surface, proving that neural network loss surfaces without BatchNorm are less smooth than those with BatchNorm. Keskar et al. [45] study the effects that batch sizes have on the solutions found during training, showing that smaller batch sizes find flatter minima which generalize better. These minima have smaller measured eigenvalues relative to the minima found with larger batches. Hochreiter and Schmidhuber [40] define flat minima as those which change minimally within large neighborhoods surrounding the local optimal solution whereas steep minima are those which change rapidly, requiring much greater precision to be found accurately [45]. Intuitively, large, flat regions can be advantageous relative to smaller, sharp regions for an optimization routine to traverse as even with small

perturbations away from a flat optimal region, solutions found can still yield good results [21]. In contrast, small perturbations from a sharp optimal region can yield very different, sub-optimal solutions as the gradient changes much more rapidly in the surrounding area [45].

Li, Ding and Sun [57] utilize a ratio of the minimum and maximum eigenvalues across portions of the loss surfaces they study in order to identify the levels of convexity present in regions surrounding local minima. Sagun et al. [82] use curvature measurements to evaluate the effects of both batch size and network size on regions found at convergence. Fort and Scherlis [25] find high curvature regions deemed Goldilocks zones using Xavier [30] and He [37] initialization methods.

3.2.2 Neural Network Early Learning

Studying early learning of CNNs is an emerging field focused on the identification of behavioral attributes exhibited during the beginning stages of training. Achille et al. [1] draw similarities between the critical periods experienced by a variety of different life forms during early developmental stages and the beginning stages of learning observed in artificial neural networks. Frankle and Carbin [26] introduce the Lottery Ticket Hypothesis, empirically showing that subnetworks appear over the course of training a CNN which are much smaller than the full network; these subnetworks can be pruned and trained to achieve competitive or better solution quality when compared with the original network. In a second work, Frankle et al. [27] study and characterize the very first ten epochs of learning using five standard convolutional neural networks using the CIFAR-10 data set [51], identifying 3 subphases experienced over the course of these first few epochs. Golatkar et al. [31] show that the timing of regularization applied to a neural network can have dramatic effects on its ability to learn in the long-term.

3.2.3 Design of Experiments and Its Applications

Design of Experiments (DOE) is a field of study which focuses on the strategic development of experiments in ways to maximize the amount of information produced during the experiment while simultaneously minimizing the total number of experimental runs required to glean desired insights. An experimental run in this setting consists of a combination of settings for each of the experimental factors included in the study; replicates of a single experimental run are repeated experiments using the same unique combination of settings to account for system variability. Although originally intended for use in agricultural applications and later for manufacturing, DOE is now used in many areas of study as a powerful tool to identify the effects that independent variables have on specific system response variables [67], [24].

DOE is a systematic, statistically rigorous method to build experiments used to identify the effects that measurable factor level changes have on a desired system response variable. An experimental factor is an element of the system hypothesized to impact a system response variable of interest. In DOE, measurable effects from factors comprised of single system elements are referred to as main effects; higher-order effects are those which come from combinations of individual factors. Combinations of factors, referred to as interactions, identify more complex relationships between factors and the effects on the system response variable that these interactions can yield. Oftentimes, higher-order interactions are assumed to not significantly impact the system and are not directly studied. In practice, higher-order effects on the system are assumed to be small and are aliased - their factor effects cannot be statistically distinguished between each other or lower-order effects.

During the experimental design phase, experimental factors are identified and factor levels are chosen for each factor to ensure statistical properties are maintained throughout experimentation, allowing for later statistical analyses which provide un-

biased results. Factor levels may not necessarily be realistic in practice, but they are used to explore the decision space surrounding each factor and the relationships between these decisions and the desired system response variable [67]. Factors are studied during the experiment in unique combinations referred to as experimental setups; the individual factor settings within each experimental setup are dictated by the experimental design selected by the experimenter.

Two-level factorial designs are a broad field of designs popularly employed across different disciplines today; these designs are often referred to as 2^k designs or 2^{k-n} designs for full-factorial and fractional-factorial designs, respectively [67]. Two-level, full-factorial designs consist of k factors, each with two levels. Each factor in a 2^k design has predefined high and low levels; these levels are often coded as +1 and -1, respectively. Factor levels are chosen to produce meaningful differences between response values found as the factor settings are varied [67].

Full-factorial designs are comprised of sets of experimental runs which encompass all combinations of factor levels, resulting in a total of 2^k total experimental runs for a single replicate of each experimental setup when each of the k factors has binary factor levels. Generally, multiple replicates of each experimental setup are desired to model response variability and build confidence in the experimental results [67]. Full-factorial designs are often viewed as inefficient when experiment resources are limited due to the sheer number of experimental runs required for mid-to-large-sized experiments; many experimenters opt to instead utilize more efficient designs such as fractional-factorial designs [67].

Fractional-factorial designs reduce the number of required runs by assuming that the effects of higher-order interactions are negligible and thus do not need to be studied. By making this assumption, higher-order interactions between different experimental factors are aliased with each other [67]. Although the amount of information

available to be studied is reduced in these designs, the number of required experimental runs to study the factor effects of lower-order factors can be dramatically reduced [67].

Fractional-factorial designs provide an orthogonal framework, which when used with multivariate linear regression, can provide unbiased estimates of factor effects [67]. Through the use of multivariate linear regression, underlying variation in the observed system response variable is explained using the experimental factors as the model's independent variables. Resultant models with high explanatory power are those with factors that can significantly influence the response variable with accuracy.

DOE has not been heavily used within the realm of neural network research to date. Tsai et al. [90] use the Taguchi method as part of a hybrid genetic algorithm to simultaneously find an optimal neural network structure and identify optimal hyperparameter settings. Yang and Lee [95] develop an experimental design to find an optimal neural network and an optimal learning rate to map velocity signals from an accelerometer to a piezoelectric sensor signal. More recently, Zhang, Chen, Yao, Ge and Dong [98] introduce a method of hyperparameter tuning for neural networks through the use of orthogonal designs. Pontes et al. [76] use DOE to tune a two-layer neural network as part of an algorithm for surface roughness prediction.

3.2.4 Exploring Neural Network Loss Surfaces

During training, a CNN undergoes adjustments to its underlying decision variables; these adjustments achieve different values along the loss surface. The goal during training is to identify a set of decision variable choices that yield favorable, low-loss regions of the loss surface. Unfortunately, due to the very large dimension of many real-world neural network decision spaces, it is not feasible to completely explore loss surfaces in their entirety. Instead, researchers have developed methods

for exploring specific portions of the loss surface.

Goodfellow et al. [32] use linear interpolation to explore trajectories from initialization to convergence when training neural networks with stochastic gradient descent, showing that there exist monotonically decreasing paths from initialization to a final solution. Im et al. [41] utilize barycentric and bilinear interpolation to analyze the trajectories between the initial and final decision variable values as well as the quality of minima found by different optimization methods. Smith and Topin [86] adopt the idea of interpolation to compare different minima found during training as well as the local peaks separating them. Li, Ding and Sun [57] utilize interpolation and a filter-wise normalization method for residual networks (ResNets) to measure loss function curvature. Santurkar et al. [83] use interpolated Lipschitz bound estimates in their empirical results reinforcing their findings regarding BatchNorm's smoothing effects. Sagun et al. [82] study the effects that overparameterization and batch size can have on solution quality, finding paths along the same level sets of the loss surface which connect solutions found using varying hyperparameters.

3.2.5 Contributions

Our work extends the use of DOE for studying CNNs and expands on current efforts to characterize the early stages of learning. Unlike previous works which explore the loss surface, we focus on statistically identifying the effects architecture hyperparameters have on the local regions found during early learning, measuring empirical Lipschitz upper-bounds on gradients and estimates of the number of steps required to escape local regions. The following contributions are made in this chapter:

- We generate 1,152 neural network instances and provide results that explain significant amounts of underlying of Lipschitz bounds on the local gradients measured and local region size variation experienced within these local regions.

- We empirically show that hyperparameters and higher-order combinations of hyperparameters significantly affect loss surfaces and identify factor effects which show strong, statistically significant influence on curvature and local region size.
- We provide insight through our findings that can aid in future network and optimization routine development.

3.3 Methodology

A subset of factors that current research has noted as influential to the overall learning process of CNNs are first identified. A resolution IV, fractional-factorial experimental design is selected to dictate individual experimental runs. Resolution IV, fractional-factorial designs are ideal for characterizing systems as all first-order factors are not aliased with each other or second-order interactions; additionally, most second-order interactions are not aliased with each other [67]. Choosing a resolution IV design significantly reduces the number of required experimental runs while still ensuring statistical inference of factor effects. For this experiment, there are a total of nine factors, introduced in Section 3.3.1. Using our resolution IV, fractional-factorial design choice with nine factors results in 64 total experimental runs required to generate a single replicate, and 18 replicates are used; in total this equates to 1,152 total experimental runs.

For each experimental run, a CNN is first trained to a point which attempts to stop short of learning stages past what is defined as early learning. Then, from this point, walks are generated, consisting of successive perturbations away from the point on the loss surface found after training completed. Five of these walks are generated that follow layer-wise perturbations drawn from a standard Gaussian distribution, designated as Brownian walks, and five of these walks follow the steepest gradient directions, designated as steepest ascent walks. Each walk stops generating steps once

it leaves the local region considered during training, or it has exceeded the amount of data it is provided to evaluate the loss surface after each successive step. In this study, the local region considered during training is defined as the portion of the loss surface explored at or below the level set of the loss surface found from decision variables initialization at the beginning of training.

Together, both types of walks generate response values to empirically measure β-constants which serve as empirical Lipschitz bounds on the estimated gradient and the estimated number of steps required to escape the local region considered during training. The resulting observations from these walks are summarized and subsequently analyzed. In total, the 1,152 experimental runs result in 5,760 walks of each type generated.[1]

3.3.1 Experimental Factor Considerations

Current research has identified several hyperparameters which are either known to or suspected to impact CNN training and generalization. This study selects nine of these factors for inclusion in the experimental design to study the effects these hyperparameters and their combinations have on the underlying curvature of local loss surfaces found in convolutional neural networks.

BatchNorm layers introduced into deep learning architectures by Ioffe and Szegedy [42] are shown to improve the loss surface's β-smoothness by Santurkar et al. [83]. Santurkar et al. [83] also empirically show smoothing along gradient directions as well as the random directions that are studied. Ghorbani et al. [29] shows a similar smoothing effect in their work.

Dropout layers (Dropout) [87], can be implemented following different layer types; these layers are popularly used to manage overfitting. Dropout techniques reduce the

[1]All experimental runs are conducted using an HP Z8 G4 Workstation with Intel(R) Xeon(R) Silver 4210 CPU @ 2.20GHz, 128 GB RAM, and PyTorch [73].

number of parameters capable of learning during each batch experienced while training a network with stochastic methods [87]. This method can improve generalization while also breaking local relationships known as feature co-adaptation that can arise between parameters Aggarwal [3].

Max pooling layers (Max Pooling) [78] provides a means of down-sampling features usually from convolutional layers [77], [84]. These layers can improve generalization while simultaneously reducing computational load through their ability to choose the most salient local features within small regions of their input layers.

Network width is a common hyperparameter related to the number of neurons which exist in each layer of the network. Li, Ding and Sun [57] extend the work of Yu and Chen [96], showing that if there are more neurons than samples in the last layer of a network, the deep network's loss surface has no set-wise strict local minima. Shallow, wide networks are successful in some applications [71].

An often competing architecture component to network width, network depth defines the number of layers which exist in the network. Safran and Shamir [81] show that deeper networks can approximate functions better than shallower networks, even if the shallower networks are wider; this finding is also supported by Liang and Srikant [61]. Dinh et al. [21] show that deep learning architectures can still generalize well even in sharp minimas; however, Li, Xu, Taylor, Studer and Goldstein [59] show that network depth increases chaotic elements of the loss surface in ResNet architectures and can make them much harder to train.

Different activation functions have been shown to impact loss surface behavior [25], [35], [62]. Rectifier linear units (ReLU) and Leaky ReLu activation functions are both widely used and well-studied [47], [33], [3]. Both of these activation functions introduce nonlinear relationships into a network, improving the potential expressiveness of the system [3]. Both ReLu and Leaky ReLu activation functions seek to

address issues related to the vanishing and exploding gradient problems which can occur during training [33], [3].

Initialization of network decision variables has proven to be a very important consideration when designing a CNN [83]. For instance, deeper networks can have exponentially more difficulty learning due to the compounding effect small initial decision variables can have on later layers in the network [10], [72], [53], [88]. Two of the most widely used methods today are Xavier Initialization and He Initialization, the former is proposed by Glorot and Bengio [30] and the latter is proposed by He et al. [37].

3.3.2 Experimental Settings

From the factors described Section 3.3.1, we identify nine experimental factors. A resolution IV, fractional-factorial experimental design is employed to dictate the individual experimental runs using these nine factors. This design is ideal for factor screening as it does not alias any main effects with second-order effects, and only some second-order effects are aliased with each other. The second-order alias structure is provided in Table 2. The selected resolution IV design requires 64 total experimental runs to generate a single replicate; a total of 18 replicates are conducted.

Table 2. Two-way interaction alias structure.

Two-Way Interaction	Alias
Convolutional Width * Fully-Connected Width	Max Pooling * Initialization
Convolutional Width * Max Pooling	Fully-Connected Width * Initialization
Convolutional Width * Initialization	Fully-Connected Width * Max Pooling

For each experimental run, a neural network is developed using the factor settings associated with the run and is subsequently trained using a batch size of 64 images.

Table 3. Experimental factor settings.

Factor	High Setting (+1)	Low Setting (-1)
Activation	ReLu used for all Convolutional and FC layers	Leaky ReLu used for all Convolutional and FC layers
Max Pooling	Used after Convolutional layers, before activation	Not used at all
Initialization Method	Xavier Uniform initialization	He Uniform initialization
Dropout Layers	Used after Convolutional and FC layers	Not used at all
BatchNorm Layers	Used after Convolutional and FC layers	Not used at all
Convolutional Width	Wide layers	Narrow layers
Fully-Connected Width	Wide layers	Narrow layers
Convolutional Depth	Ten layers	One layer
Fully-Connected Depth	Ten layers	One layer

The goal of these settings is not to necessarily develop state-of-the-art or even remotely successful network setups but rather to create instances that emphasize the traits the individual factors and combinations of factors impart onto the learning problem. We acknowledge that many of the settings might not be realistic in practice. Each individual experimental setup's factor settings can be found in Section 7.1.

3.3.3 Network Training

All experimental runs are trained using Stochastic Gradient Descent with a Momentum value of 0.9, a learning rate of 0.01, and batch sizes of 64 data points using the MNIST data set [55]. Every network uses a Cross-Entropy loss function and a final output layer of 10 neurons with Soft-Max activation; this final layer is standard across all experimental runs and is not counted as one of the Fully-Connected Layers for the Fully-Connected Depth factor considered in Table 3.

To stop learning from progressing past the early-learning stages, a stopping criterion are used. The criterion allow for up to six complete training sets to be exposed to a network instance (a maximum exposure of 360,000 total data points) if a raw training loss threshold of 0.2 is not achieved. We choose six complete epochs instead of the ten used to define the early-learning stages in Frankle et al. [27] due to the ease-of-training often exhibited on the MNIST data set. In the event that the raw training loss threshold of 0.2 is achieved, the network is allowed to finish its current

epoch before moving onto the walk generation stage.

In total, 1,152 network instances are used to develop the results, equating to 5,760 total walks of each type.

3.3.4 Network Walk Generation

After training is completed for a single experimental run, Brownian walks and steepest ascent walks are generated starting from the final decision variable set found from the training stage. A step along either walk type is defined as a perturbation of the decision variables from the previous step that is considered. These successive steps effectively create walks throughout the local loss surface. Each step of a walk uses a batch size of 64 data points and at most a total training set size of 60,000 data points, to take a maximum of 938 steps, providing information to interpolate 937 individual Lipschitz upper bound values.

After each step along a walk, both the loss value and gradient of the loss surface are evaluated. The evaluated gradient values and corresponding decision variables are used as inputs into Equation 4 to find empirical Lipschitz bounds on the local gradient.

Due to the sheer size of the dimension that most neural network loss surfaces exhibit, it is not computationally feasible to build walks that explore all or even most of the directions that could be taken. Instead, the Brownian walks are used to provide a series of unbiased samples that randomly visit faces of the local loss surface, further and further away from the decision variables during the training stage. In a single experimental run, five independent Brownian walks are generated. The method for generating a single Brownian walk can be found in Algorithm 1.

Algorithm 1 Layer-wise Brownian motion process walk procedure.

Let θ^* be the decision variable values found at the end of training.
Initialize variable values, $\theta_0 \leftarrow \theta^*$.
Initialize fixed step size, η. **while** $Walk\ Loss \leq Initialization\ Loss$ **do**
  for *Layer j of* θ_i *and Step* $\leq Max\ Step$ **do**
    Store dimensions of layer j.
    Generate direction vector $d'_j \sim \mathcal{N}(0, 1)$ of appropriate dimension.
    Normalize the direction vector $d_j \leftarrow \frac{d_j}{\|d_j\|}$
    Take a Brownian step $\theta_{i+1,j} \leftarrow \theta_{i,j} + \eta d_j$.
  end
end

In addition to the Brownian walk generation process, a second set of five walks are generated for each experimental run in the steepest directions of the local loss surface beginning from the decision variable set found at the end of the training stage. These walks are deemed walks of steepest ascent as they follow standard gradient steps in the direction of the gradient. The goal of these walks is to identify portions of the local loss surface that are steepest; these walks are intended to identify areas corresponding to high areas of the local loss surface. The method for defining a single steepest ascent walk can be found in Algorithm 2.

Algorithm 2 Layer-wise steepest ascent walk procedure.

Let θ^* be the decision variable values found at the end of training.
Initialize variable values, $\theta_0 \leftarrow \theta^*$.
Initialize fixed step size, η.
while $Walk\ Loss \leq Initialization\ Loss\ and\ Step \leq Max\ Step$ **do**
  for *Layer j of* θ_i **do**
    Take a steepest ascent step $\theta_{i+1,j} \leftarrow \theta_{i,j} + \eta\hat{\nabla} f(\theta_{i,j})$.
  end
end

3.3.5 Defining Local Regions and Identifying Walk Break Points

The walk generation stage has the ability to quickly leave a local region around the converged decision variables and enter into high-loss portions of the loss surface that are not likely to be encountered during training. A simple break-point rule is used to estimate when the local region has been left while taking steps along a walk. After each step of a walk, the loss value corresponding to the step is compared with the initialization loss value found at the beginning of the training stage. If the loss value at this step exceeds the initialization loss, the walk immediately stops generating steps.

3.3.6 Response Variables and Factor Effects

After each walk has been generated for each experimental run, both the maximum observed Lipschitz value and the length (number of steps taken) of the walk are recorded as response values to be studied. We choose to take the maximum Lipschitz value observed along each local walk to provide an empirical upper bound on the Lipschitz bound of the gradient in this region; this estimates the maximum amount of the gradient change in the local region. The number of steps taken during a walk type before leaving a local training region is interpreted as complimentary measures of a local region's size. In instances where more steps are required to leave the local region, the local region size is interpreted to be larger. With two types of walks generated, there are a total of four responses associated with each experimental run:

- Maximum Observed Brownian Lipschitz Upper Bound Values
- Number of Steps Before Leaving Local Region along Brownian Walk
- Maximum Observed Steepest Ascent Lipschitz Upper Bound Values
- Number of Steps Before Leaving Local Region along Steepest Ascent Walk

3.3.7 Interpretation of Results

In DOE, measured factor effects found through multivariate linear regression consist of coefficients representing the mean effect each studied factor has on the response variable of interest [67]. In this study, coefficient signs of factor effects found through multivariate linear regression correspond directly to the directional impact that the effect has on the modeled response variable. For example, a statistically significant first-order factor with an effect coefficient whose sign is positive indicates that the first-order factor positively affects the modeled response variable when it is set to its high setting and negatively affects the modeled response variable when it is set to its low setting. Second-order factors whose settings are both fixed to their high settings or are both set to their low settings will illicit the same effect on the response variable.

In the context of our study, by identifying statistically significant factors we find neural network architecture components and combinations of components which have strong effects on empirical Lipschitz bounds on the local gradients, β, as well as the number of steps taken before escaping the local region along either walk type. Observing the respective coefficient signs of each of these statistically significant factors provides empirical evidence of how a specific architecture hyperparameters influences underlying CNN loss surfaces.

3.4 Results

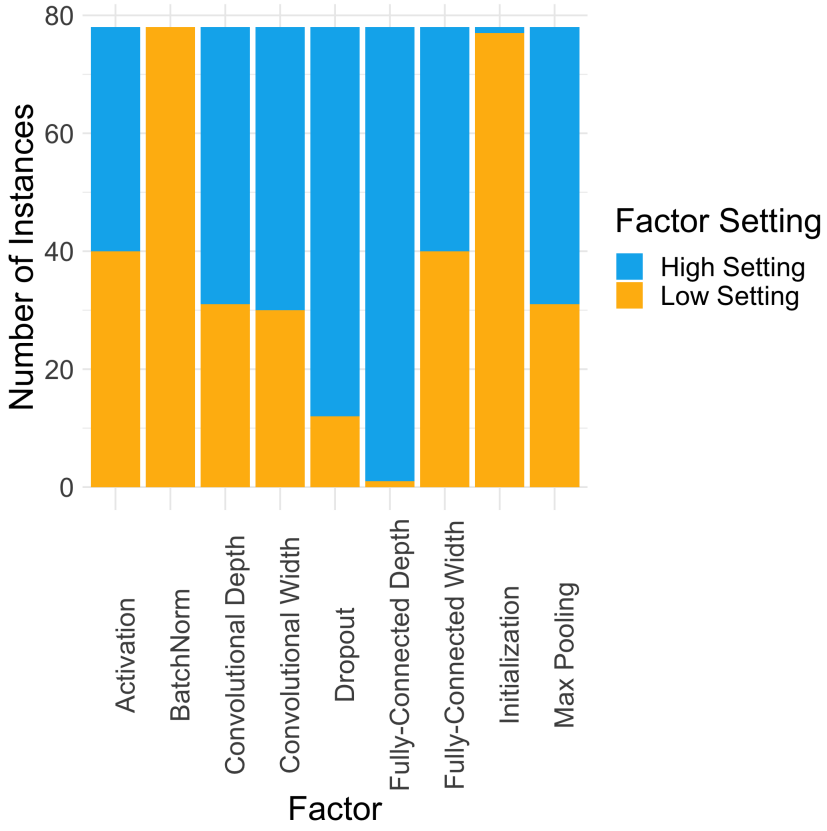
3.4.1 Overview

After generating local walks across all experimental setups and replicates, we analyze the empirical Lipschitz bounds on the local gradient and the size of the local region through multivariate linear regression models. Statistically significant factor effects are identified and cross-listed across models to provide insights into the ways

in which local regions encountered during early-learning are affected by architecture hyperparameters.

In total 78 replicates exhibited divergent behavior during training. Figure 3 illustrates the factor setting distributions for these divergent instances. Divergent instances do not utilize BatchNorm; additionally, almost all of these instances utilize He initialization as well as the deep fully-connected layer setting.

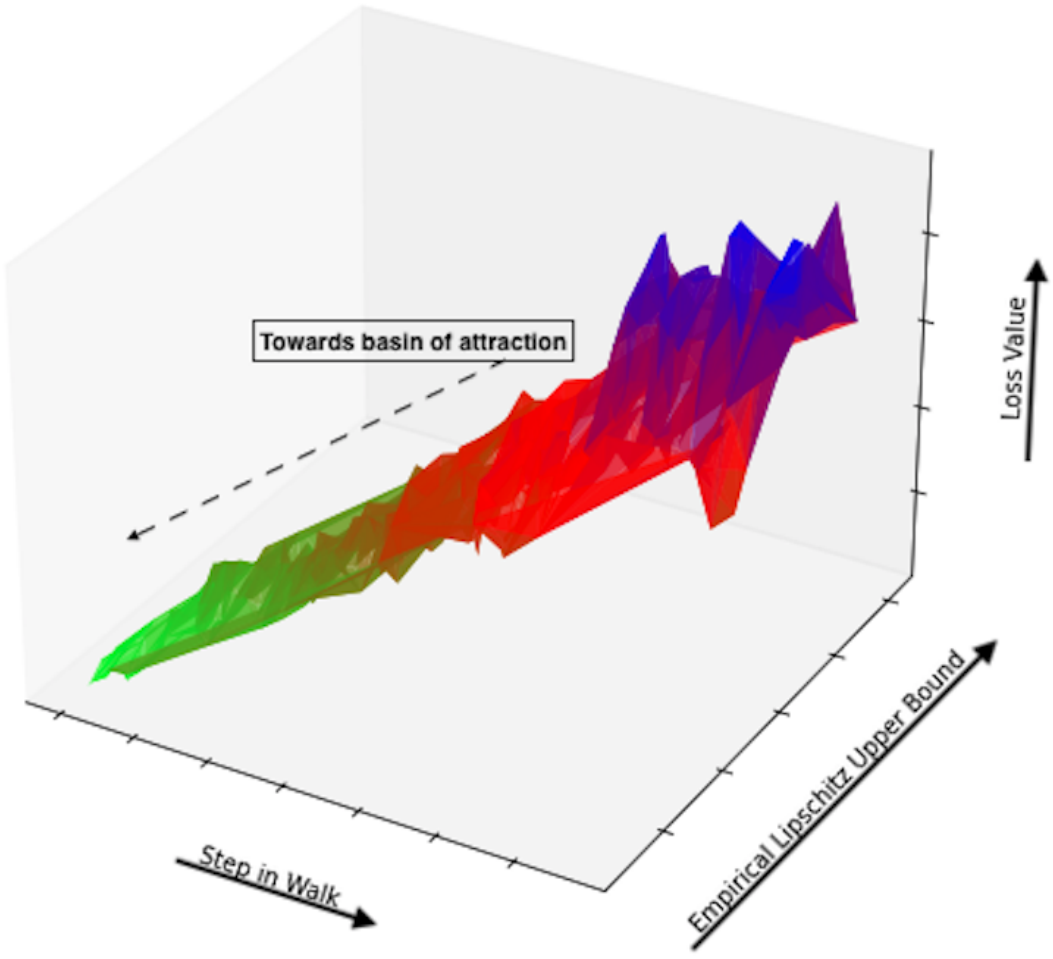
Figure 3. Distribution of factor settings exhibited in the 78 divergent instances.



To illustrate the walk generation process, the following visualizations illustrate measured response variables across local regions of CNN loss surface instances. Figure 4 illustrates movement along a single Brownian walk, showing that the further away from the point of origination a walk progresses, both the second-order Lipschitz estimates and loss values become increasingly large. Additionally, the loss surface's

local maxima and minima grow in magnitude, indicating regions which could pose more difficult problems for optimization routines as they traverse the local loss surface.

Figure 4. Three-dimensional visualization of a Brownian walk for one experimental setup across local region on a convolutional neural network loss surface.



The paths generated for different experimental setups in Figures 5 and 6 show that the loss surface's curvature and size can differ greatly when different hyperparameters are used to build network instances.

Figure 5. Five Brownian walks across the local region experienced during the early-learning stages.

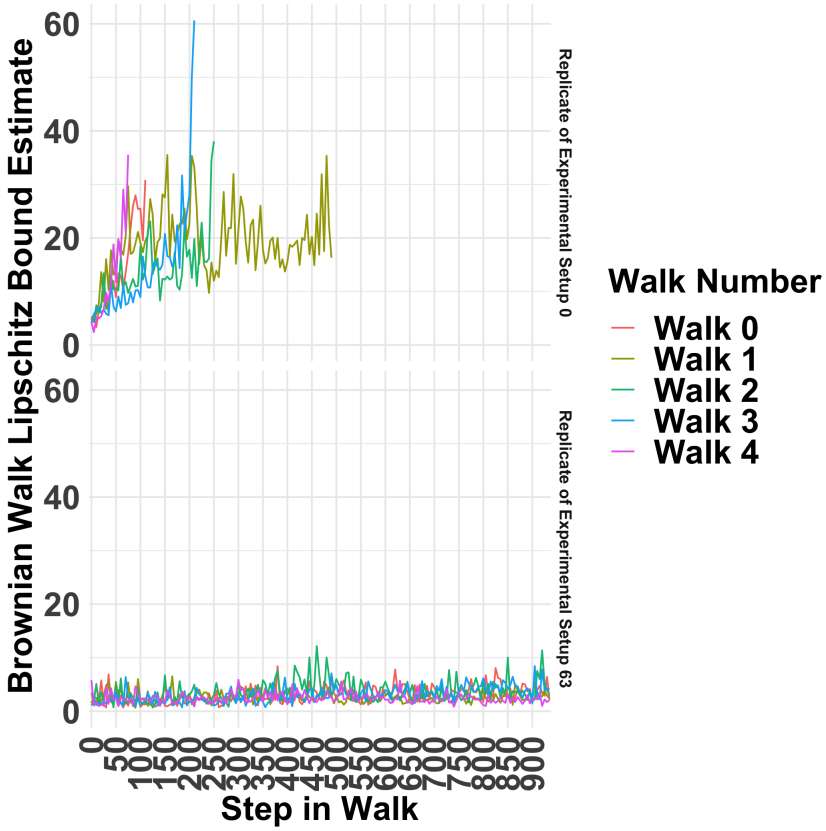
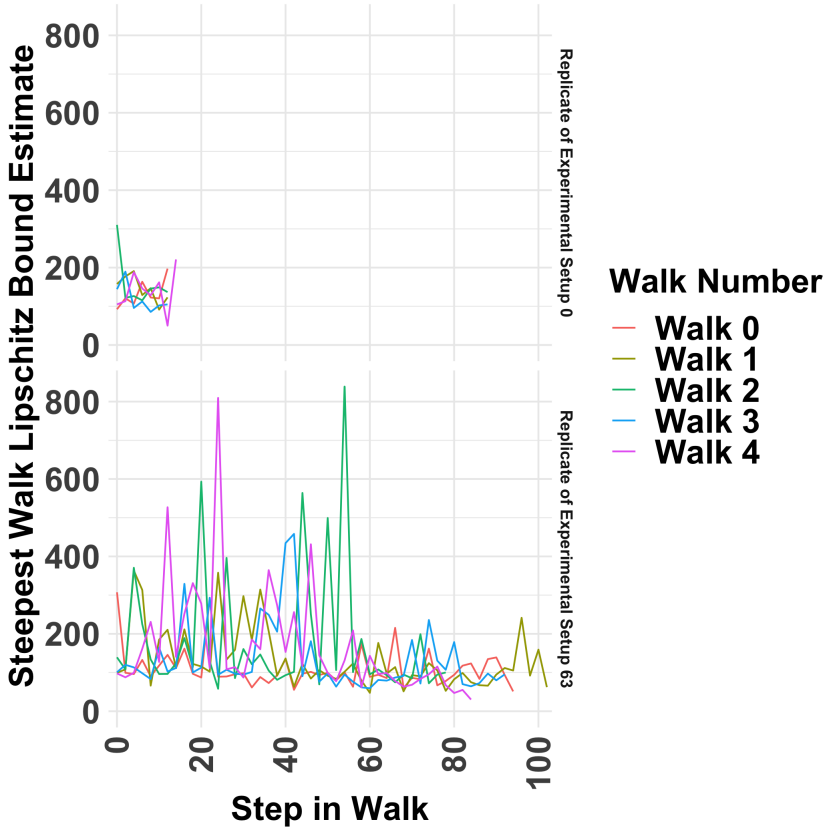


Figure 6. Five Steepest ascent walks across the local region experienced during early-learning stages.



3.4.2 Multivariate Linear Regression

Multivariate linear regression models are developed for each response variable. All models yield statistical significance, suggesting that the measured response variables studied are truly impacted by the decisions related to the convolutional neural networks' architectures. We set $\alpha = 0.01$ as the level of significance to study model significance and factor effects; in many cases, a majority of the factors considered show statistical significance at this level. By restricting $\alpha = 0.01$, the statistical findings identify highly significant factors. Data points are flagged as outlier observations and are removed using Cook's Distance [19]. Additionally, levels of multicollinearity between all factors in each model is measured using variance inflation factors (VIFs);

across all models, VIF values did not exceed 1.25, suggesting that there is minimal multicollinearity exhibited in these models.

First, third-order multivariate linear regression models are developed for each response variable. Although the experimental design that this study utilizes is only a resolution IV design, meaning that there is aliasing with most third-order factors, we choose to first study these higher-order models to provide a point of comparison for the level of impact higher-order factors have on explanatory power. Due to the alias structure, most of the third order factors cannot be distinguished from each other.

Table 4. Third-order multivariate linear regression model summaries.

Response Variable	Transform Utilized	Adjusted R^2	Model p-value
Brownian Lipschitz Value	Box-Cox ($\lambda = -0.050$)	0.855	$2.2e^{-16}$
Brownian Number of Steps	Box-Cox ($\lambda = 0.500$)	0.690	$2.2e^{-16}$
Steepest Ascent Lipschitz Value	Box-Cox ($\lambda = -0.250$)	0.530	$2.2e^{-16}$
Steepest Ascent Number of Steps	Box-Cox ($\lambda = 0.100$)	0.753	$2.2e^{-16}$

Second, models that only utilize first-order and second-order interaction factors are developed. Table 5 summarizes the full second-order multivariate regression models. In total, there are nine main factor effects studied, corresponding to each of the factors identified for this study and 33 second-order effects. We note that although there are truly 36 second-order effects, three second-order effects are not considered due to the alias structure provided in Table 2. Full results of the regression models can be found in Section 7.2.

Table 5. Full second-order multivariate linear regression model summaries.

Response Variable	Transform	Adjusted R^2	Significant 1^{st}-Order Factors	Significant 2^{nd}-Order Factors
Brownian Lipschitz Value	Box-Cox ($\lambda = -0.050$)	0.797	8 of 9	28 of 33
Brownian Number of Steps	Box-Cox ($\lambda = 0.500$)	0.559	5 of 9	25 of 33
Steepest Ascent Lipschitz Value	Box-Cox ($\lambda = -0.250$)	0.404	9 of 9	25 of 33
Steepest Ascent Number of Steps	Box-Cox ($\lambda = 0.100$)	0.619	8 of 9	27 of 33

3.4.3 Cross Listing Significant Terms

After developing the second-order regression models, we cross-list statistically significant factors at $\alpha = 0.01$. We conduct cross listings for significant factors in each of the models, separately, first identifying statistically significant factors that share the same signs in their factor effect coefficients. By cross listing statistically significant factors with factor effect coefficients which share the same sign, we identify factors that affect the curvature and size estimates of the local loss regions in similar ways.

We first note that factor effects displayed in the following tables relate to each factor set to its high, +1, setting; if the factor is set to its low, -1, setting the sign of the corresponding factor effect is reversed. Second-order interaction effects displayed are the effects which occur when both factors are set to their high, +1, settings; however, these effects are also true when both factors are set to their low, -1, settings $(-1 * -1 = 1)$. If one of the two factors is set to its high, +1, setting and the other is set to its low, -1, setting, the sign of the corresponding second-order interaction effect displayed is reversed $(-1 * 1 = -1)$. The regression results in Section 7.2 are also displayed in this way.

Tables 6 and 7 summarize the coefficients that are identified as statistically significant and share the same coefficient sign across both second-order models. Table 8 cross lists the factors found to be significant and consistent in each of the independent cross listings for the second-order Lipschitz bound and number of steps models.

Table 6. Cross listing of significant terms across both second-order Lipschitz bound models at $\alpha = 0.01$. Factor effects displayed are those at the high, +1, setting; for interactions, this occurs when both factors are set to their high, +1, settings or when both factors are set to their low, -1, settings.

Factor	Factor Effect
Activation	Increases Upper Bound
BatchNorm	Increases Upper Bound
Convolutional Depth	Decreases Upper Bound
Dropout	Decreases Upper Bound
Fully-Connected Depth	Decreases Upper Bound
Initialization	Decreases Upper Bound
BatchNorm * Dropout	Increases Upper Bound
BatchNorm * Initialization	Increases Upper Bound
BatchNorm * Max Pooling	Decreases Upper Bound
Convolutional Depth * Activation	Decreases Upper Bound
Convolutional Depth * BatchNorm	Decreases Upper Bound
Convolutional Depth * Fully-Connected Depth	Increases Upper Bound
Convolutional Depth * Fully-Connected Width	Increases Upper Bound
Convolutional Width * Activation	Increases Upper Bound
Convolutional Width * Dropout	Decreases Upper Bound
Convolutional Width * Fully-Connected Width	Decreases Upper Bound
Convolutional Width * Max Pooling	Decreases Upper Bound
Dropout * Activation	Increases Upper Bound
Dropout * Initialization	Decreases Upper Bound
Fully-Connected Depth * Dropout	Decreases Upper Bound
Fully-Connected Depth * Max Pooling	Increases Upper Bound

Table 7. Cross listing of significant terms across both second-order number of steps to escape the local region models at $\alpha = 0.01$. Factor effects displayed are those at the high, +1, setting; for interactions, this occurs when both factors are set to their high, +1, settings or when both factors are set to their low, -1, settings.

Factor	Factor Effect
BatchNorm	Decreases Number of Steps
Convolutional Depth	Increases Number of Steps
Dropout	Increases Number of Steps
Initialization	Decreases Number of Steps
Max Pooling	Increases Number of Steps
Activation * Initialization	Increases Number of Steps
BatchNorm * Activation	Decreases Number of Steps
BatchNorm * Initialization	Increases Number of Steps
BatchNorm * Max Pooling	Decreases Number of Steps
Convolutional Depth * Activation	Increases Number of Steps
Convolutional Depth * Dropout	Increases Number of Steps
Convolutional Depth * Fully-Connected Depth	Increases Number of Steps
Convolutional Depth * Fully-Connected Width	Increases Number of Steps
Convolutional Depth * Max Pooling	Increases Number of Steps
Convolutional Width * Activation	Increases Number of Steps
Convolutional Width * BatchNorm	Increases Number of Steps
Convolutional Width * Convolutional Depth	Decreases Number of Steps
Convolutional Width * Dropout	Decreases Number of Steps
Convolutional Width * Fully-Connected Depth	Decreases Number of Steps
Convolutional Width * Initialization	Decreases Number of Steps
Convolutional Width * Max Pooling	Decreases Number of Steps
Dropout * Initialization	Decreases Number of Steps
Fully-Connected Depth * BatchNorm	Increases Number of Steps
Fully-Connected Depth * Dropout	Increases Number of Steps
Fully-Connected Depth * Initialization	Decreases Number of Steps
Fully-Connected Width * Activation	Decreases Number of Steps
Fully-Connected Width * BatchNorm	Decreases Number of Steps
Fully-Connected Width * Dropout	Increases Number of Steps
Max Pooling * Activation	Increases Number of Steps
Max Pooling * Dropout	Increases Number of Steps

Table 8. Cross listing of significant terms found across all second-order regression models with a significance level of $\alpha = 0.01$. Factor effects displayed are those at the high, +1, setting; for interactions, this occurs when both factors are set to their high, +1, settings or when both factors are set to their low, -1, settings.

Factor	Factor Effect on Lipschitz Upper Bound	Factor Effect on Number of Steps
BatchNorm	Increases Upper Bound	Decreases Number of Steps
Convolutional Depth	Decreases Upper Bound	Increases Number of Steps
Dropout	Decreases Upper Bound	Increases Number of Steps
Initialization	Decreases Upper Bound	Decreases Number of Steps
BatchNorm * Initialization	Increases Upper Bound	Increases Number of Steps
BatchNorm * Max Pooling	Decreases Upper Bound	Decreases Number of Steps
Convolutional Depth * Activation	Decreases Upper Bound	Increases Number of Steps
Convolutional Depth * Fully-Connected Depth	Increases Upper Bound	Increases Number of Steps
Convolutional Depth * Fully-Connected Width	Increases Upper Bound	Increases Number of Steps
Convolutional Width * Activation	Increases Upper Bound	Increases Number of Steps
Convolutional Width * Dropout	Decreases Upper Bound	Decreases Number of Steps
Convolutional Width * Max Pooling	Decreases Upper Bound	Decreases Number of Steps
Dropout * Initialization	Decreases Upper Bound	Decreases Number of Steps
Fully-Connected Depth * Dropout	Decreases Upper Bound	Increases Number of Steps

3.5 Discussion

3.5.1 Overall Observations

The hyperparameters that are studied in this chapter significantly impact the loss surface along both walk types. Although several factors affect the loss surfaces' second-order Lipschitz bound and local size measures in different ways along both types of directions, there are also many factors that significantly impact second-order Lipschitz upper bounds and local sizes in consistent ways across the consistent directions considered.

Our multivariate regression models account for substantial amounts of the underlying response variation measured in all cases with third-order models capable of accounting for over 50% of underlying response variation. These results show that within a certain level of error, network-designers can control for both second-order Lipschitz upper bounds and local sizes with hyperparameter choices across local regions of the loss surface encountered during early-learning. Future network design may need to simultaneously consider optimization routine selection to ensure that architecture hyperparameters and combinations of these hyperparameters induce de-

sired loss surface features that can be readily exploited by a routine's specific capabilities.

3.5.2 Models of the Empirical Lipschitz Upper Bounds on the Local Gradients

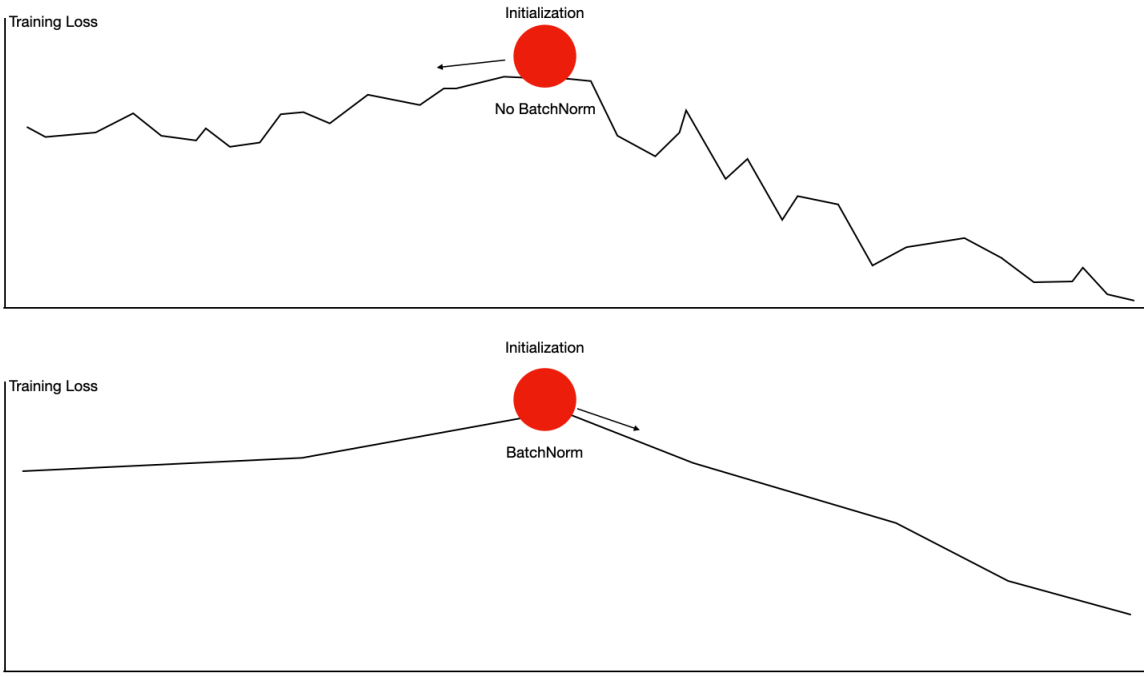
All of the models used to study empirical Lipschitz upper bounds on the local gradients in these regions show statistical significance, indicating that the underlying loss surfaces' bounds on local gradients along the local walks are significantly impacted by the architecture choices made using our experiment. The third-order empirical Lipschitz upper bound models explain large amounts of the underlying variability across both walk types; in both cases over 50% of the total variation is explained by these two models, with the Brownian walk model yielding an Adjusted R^2 of over 85%. Although individual factor effects are not distinguishable due to the alias structure induced on higher-order terms, we find that explanatory power of these models supports recent findings indicating architecture hyperparameters significantly affect the behavior of the local loss surface.

The full second-order model of the empirical Lipschitz upper bounds developed for the Brownian walks continues to explain large amounts of underlying response variation with a very small reduction in the Adjusted R^2 value. On the other hand, the second-order model of the empirical Lipschitz upper bounds model associated with steepest ascent walks explains much less than its third-order counterpart. We hypothesize that this observation indicates that the decisions associated with architecture choices have more complex impacts on the local loss surface's steepest directions and may also be affected in greater ways by factors that are not measured in this study. The incorporation of three-way interactions without aliasing might be required to more readily model this response with a multivariate linear regression model.

Comparisons of the signs of statistically significant terms found in Table 6 across the two second-order Lipschitz upper bound models identifies factor effects that are similar along both walk types. Terms with positive coefficients for models of either second-order Lipschitz upper bound response variable correspond to terms that significantly increase Lipschitz upper bounds along a walk type whereas negative coefficients correspond to terms that significantly decrease Lipschitz upper bounds along a walk type. Several of the main factor effect coefficients as well as second-order interaction effect coefficients match in sign across models of both walk types, suggesting that both in the steepest ascent and Brownian directions, the second-order Lipschitz upper bounds on the gradient are affected by these factors in similar ways.

Of note, BatchNorm showed statistical significance in increasing Lipschitz upper bounds locally along both the steepest ascent and Brownian directions. We believe this finding is complementary to Ghorbani et al. [29] and Santurkar et al. [83]; that is, although it has been shown that BatchNorm has a smoothing effect across loss surfaces, our results indicate that BatchNorm may simultaneously cause a SGD-Momentum to find a region with higher second-order Lipschitz bounds on the local gradient relative to regions that it is drawn to when BatchNorm is not present during early-learning stages. Figure 7, below, illustrates this point.

Figure 7. Hypothesized trajectory during early learning with and without BatchNorm present.



We also found that BatchNorm appears to help prevent divergent behavior - Figure 3 shows that observed divergent cases did not utilize BatchNorm, further emphasizing the smoothing effect found by Santurkar et al. [83].

3.5.3 Models of the Number of Steps to Escape the Local Region

The third-order and second-order regression models used to study the number of steps required to escape the local region for each of the walk types show strong statistical significance. Additionally, over 50% of variation is explained in each of these models, suggesting that the underlying responses are reasonably modeled using the hyperparameters studied in this experiment.

The two types of walks each provide estimates of the local region size along different portions of the loss surfaces that are studied. Steepest ascent walks provide size

estimates along portions of the loss surface with very steep faces; these walks often leave the region more quickly than Brownian walks due to their ability to quickly find high loss regions. Brownian walks randomly explore the local regions of the loss surface. Exploration of the the region size along these walks is not biased towards a specific type of loss surface face.

Together, these responses are used to study the relative sizes of the different local regions explored. Larger local regions require more steps to escape the local region along the different walk types. If a region is both large and flat, it could be favorable for an optimization routine to traverse due to the lower precision requirements in estimating a descent direction during early learning [40]. In the case of our experiments, factors which increase the number of steps are those which lead to larger local regions of the loss surface. Factors which consistently affect size in the same ways across both walk types are those which lead to more consistent sizes along the different directions that are studied. We find similar results to those found in the second-order Lipschitz models - a variety of different architecture hyperparameters can affect size estimates in consistent ways along both path types; however, there are also several hyperparameters that inconsistently affect the local region size along these different directions.

3.5.4 Cross Listings of Factor Effects

Cross listing factors which significantly impact second-order Lipschitz estimates as well as the number of steps needed to escape the local region provides a more complete perspective of how these individual factors and combinations of factors influence the local loss surface. Table 8 summarizes the statistically significant factors found across all second-order linear regression models. Although several factors are included in this table, there are also several which are not included. This observation indicates that

different architecture components might induce competing characteristics onto local loss surfaces; other architecture hyperparameters might have no effect across one of the walk types while having strong effects along the other walk type.

We believe that this observation must be considered when developing an architecture-optimization routine combination. A routine that depends on heightened levels of curvature within the local training region may fall into low curvature regions if care is not placed as to how it traverses the surface; likewise, if a routine requires lower levels of curvature to be stable, it could fall into high curvature regions that could cause it to quickly become divergent.

3.5.5 Limitations and Next Steps

The MNIST data set and the number of factors considered in this study were both selected due to resource constraints. The MNIST data set is often considered an easy data set to train on. A natural next step would be to conduct a series of experiments similar to this experiment on a different, more challenging data set with more factors considered at once.

The fractional-factorial design used in this study reduces the number of experimental runs considered to maximize the amount of information that can be derived from the resources that are available. By using this design, some two-way interaction terms are aliased with each other and almost all three-way interactions are aliased. Especially in the case of modeling curvature along steepest ascent descent directions, it appears that three-way interactions may be necessary to truly model this response using the architecture hyperparameters considered in this study. A follow up study that utilizes a larger-scale experimental design could help mitigate the need to alias terms.

There are three experimental setups that could not be trained across all seeds

and some experimental setups could not be trained on specific seeds; additionally, in some instances, there are stochastic walks of either type that immediately left the local region. This data loss caused the experimental design to lose its orthogonality property; although linear regression models have been shown to be robust to data loss [67], future experimental designs could benefit from mitigating data loss as much as possible.

Walks are generated in this study using a maximum of 938 steps along either walk type; there are several instances in which the full 938 steps are taken without exceeding the initialization loss threshold defined in Section 3.3.4. These walks are stopped prematurely. The choice of a 938 step threshold is used to keep walks from generating steps indefinitely; however, the choice reduces the amount of variability observed along walks that never exceed the initialization threshold. The reduced amount of variability biases the regression models; however, this is an unavoidable consequence and can be viewed as analogous to limitations on instrument measurement constraints found in traditional manufacturing settings [67]. A future experimental design could allow for longer maximum walk generation periods; this could improve response variability.

Many of the steepest ascent walks find an extremely steep direction very quickly, reducing the ability to measure Lipschitz bounds on the gradients before exiting the local region. The reduction in observable steps reduces the level of variation that can be measured between experiments. We believe that this could impact the reduced explanatory power exhibited in the second-order regression model; future work could choose steepest ascent walk step rules which are less aggressive with smaller step sizes, allowing for more observations to be taken. This observation also indicates that the loss surface can very quickly exhibit non-Lipschitz properties outside of the local region.

3.6 Conclusion

In this chapter, we leverage a DOE approach to explore local regions of convolutional neural network loss surfaces. Characterizing local regions of loss surfaces that are not necessarily along the original walk taken during optimization provides further insight into the ways in which we can exploit architecture hyperparameter selection to make the underlying optimization problems associated with training more ideal for traversal by an optimization routine. Identifying subsets of hyperparameters that can improve desired underlying loss surface characteristics for ease-of-optimization such as improving second-order Lipschitz upper bound and local region size estimates; however, from our experiments it is clear that an interesting and challenging problem can arise from network design choices. Individual architecture hyperparameters and combinations of these hyperparameters can change certain portions of the loss surface's curvature to be more desirable for an optimization routine at the expense of potentially hurting other portions of the loss surface. Additionally, our results show that not only individual hyperparameters but also specific combinations of these hyperparameters can have statistically significant impacts on loss surface characteristics. Continuing to develop insights into the ways in which the local loss surface changes based on these architecture hyperparameters and their combinations will ultimately allow current and future optimization routines to train convolutional neural networks more effectively.

IV. Contribution II: Identifying Convolutional Neural Network Architecture Hyperparameters that Significantly Impact Early Learning of First-Order Optimization Routines with Quality Engineering

4.1 Introduction

There are several popular optimization routines used to train deep convolutional neural networks. Due to the high dimension of these training problems, most routines depend solely on first-order information to make updates during the training process. First-order routines employed to train CNNs can be categorized as either non-adaptive or adaptive routines. Non-adaptive, first-order optimization routines generally use a global set of hyperparameters to dictate successive updates and do not adjust their hyperparameters during training whereas adaptive routines can make online adjustments to compensate for the data used in the training problem and local loss surface geometry. Both categories have been shown to yield state-of-the-art training results for CNNs, with adaptive methods being popular choices for many instances due to their ability to reduce training loss quickly. With that being said, there has been recent criticism which indicates that adaptive routines find solutions that generalize more poorly compared to their non-adaptive counterparts. In this chapter, we explore the effects that common architecture decisions have on different routines' abilities to both train successfully and also find solutions that can generalize well.

4.2 Background

4.2.1 Underlying Optimization Problem

The underlying optimization problem associated with training CNNs using batch-training approaches is large-scale, stochastic, and non-convex. The loss surface itself can exhibit expansive flat regions and sharp local minima, challenging obstacles for an optimization routine to traverse during training [83], [45]. Sections 3.2.1 and 3.2.4 provide additional background information related to loss surface characteristics of different optimization problems as well as additional methods that have been applied to characterize local CNN loss surfaces. In our first contribution, we extend recent research seeking to characterize the local loss surfaces of CNNs by showing that local loss surface curvature as well as region size can be significantly affected by a variety of different architecture hyperparameters. The findings of our first contribution as well as several other researchers' contributions indicate that a variety of decisions can affect the underlying optimization problem that first-order optimization routines seek to solve.

4.2.2 Optimization Routine Choice

The choice of an optimization routine used to train CNNs is restricted to routines which can operate solely with first-order oracle information while also making computationally cheap updates. Optimization routines employed to train CNNs can be broadly classified into non-adaptive routines that utilize a global set of hyperparameters to inform successive updates and adaptive routines that actively adjust to the local loss landscape based on real-time information.

Non-adaptive, first-order optimization routines that have been popularly used to train CNNs include Stochastic Gradient Descent (termed SGD-Vanilla for the remainder of this chapter) [13], Stochastic Gradient Descent with Momentum (SGD-

Momentum) [75], and Stochastic Gradient Descent with Nesterov's Accelerated Gradient (SGD-Nesterov) [68]. All three of these optimization routines utilize a global learning rate to make updates. The latter two routines equip SGD-Vanilla with momentum coefficients which can speed up convergence rates of SGD-Vanilla.

Adaptive optimization routines currently in use include AdaGrad [23], RMSProp [89], and Adam [48]. All of these optimization routines attempt to adjust their hyperparameters as training progresses to fit the geometry of the data set being studied; this adaptive behavior provides more flexibility in how parameterization takes place [91]. AdaGrad and RMSProp both take updates which depend on storage of the square terms of the gradient with RMSProp.

In addition to the original research presenting each routine, we refer to Gron [33] and Wilson et al. [91] for summaries of each of these methods.

Adaptive methods have shown success in quickly reducing training loss in the early stages of learning [46], [91]. This trait combined with their limited tuning requirements has made these methods popular for training CNNs [91]. With that being said, there have been critiques of how well adaptive methods are able to find solutions that generalize well. Wilson et al. [91] show that adaptive approaches can perform much worse than methods that do not employ adaptive techniques. Keskar et al. [45] found that large batch sizes can lead to overfitting of the training data for Adam. Adaptive methods' potential inabilities to consistently find solutions that not only have low training loss quickly but also produce solutions which generalize well poses a potential limiting factor in using them to train a CNN.

In contrast, there is increasing research indicating that non-adaptive methods exhibit favorable properties for traversing CNN loss surfaces. The oscillatory behavior exhibited by non-adaptive methods has been shown to help avoid sub-optimal local minima [14]. Lee et al. [56] show that descent family methods can avoid saddle

points. Hardt et al. [34] show that stochastic gradient descent can achieve solutions with strong generalization capabilities.

These contrasting viewpoints have left researchers unable to definitively choose an optimization routine to train specific CNN instances.

4.2.3 Design of Experiments and Early Learning Background

The periods studied in this work pertain to the early stages of learning for a variety of different network architectures. We refer to Section 3.2.2 for background information related to early learning studies that have been conducted for neural network instances. Additionally, we study architecture hyperparameter decisions using a Design of Experiments methodology and the same hyperparameters and settings studied in our first contribution. We refer to Section 3.2.3 for additional background information related to Design of Experiments methodologies.

4.2.4 Contributions

This chapter extends the use of DOE to study neural network characteristics and identifies neural network architecture hyperparameters which affect performance differences exhibited between different optimization routines during the early-learning stages of training. The goal of this study is to draw relationships between architecture hyperparameter decisions and optimization routine selection.

We build off of our first contribution, which identifies different architecture hyperparameters that can significantly affect loss surface characteristics experienced during the early-learning stages. Acknowledging that we can significantly affect local loss surface geometries which are traversed by different first-order optimization routines, a natural next step is to study the ways in which these architecture hyperparameter decisions can affect loss surface traversal by different first-order methods. Additionally,

recognizing that there are competing viewpoints which exist related to optimization routine selection indicates that these routines may favor different instances' loss surfaces due to inherent traits induced by hyperparameter decisions. Through our research, we aim to draw relationships between CNN architecture hyperparameter decisions and optimization routine selection, demonstrating that certain optimization routines cater to the prevailing architecture decisions of different network instances in a variety of ways.

Our approach utilizes a fractional-factorial, resolution IV Design of Experiments (DOE) methodology to study a total of 8,448 neural network instances trained across two popular benchmark data sets, identifying relationships between neural network architecture hyperparameter choices and the quality of solutions found by training CNN instances using six popular optimization routines.

- We provide an analysis of divergent cases found during experimentation, identifying CNN architecture hyperparameters which might be more likely to lead specific optimization routines into divergent portions of loss surfaces.

- We statistically show that across all instances studied, adaptive methods can provide competitive training results; however, non-adaptive methods can find solutions that are as-good or better at generalizing well, even in the beginning stages of learning.

- We develop multivariate linear regression models that explain upwards of 90% total variation of the underlying responses analyzed. These models identify statistically significant architecture hyperparameters' effects on both training and testing for each of the studied optimization routines, pointing towards the similarities and differences between how these hyperparameters are used by different optimization routines during early-learning phases.

- We utilize the developed regression models to build four separate meta-learning models for each data set, loss type combination studied, using architecture hyperparameters to predict optimization routine performance with high accuracy.

Together, these results provide an empirical, statistically rigorous means of identifying the effects common CNN architecture hyperparameters have on different optimization routines' abilities to train and find solutions that generalize well.

4.3 Methodology

4.3.1 Overview

Our methodology consists of first identifying common CNN hyperparameters used in many existing architectures. After identifying nine of these hyperparameters, a resolution IV, fractional-factorial experimental design consisting of 64 total experimental factor combinations per replicate is selected. The resolution IV, fractional-factorial design is ideal for characterizing system factor effects as all first-order factors are not aliased and only some second-order factor effects are aliased with each other. This design balances experimental runs while still ensuring statistical inference can be made for first and second-order factor effects.

In this study, the identified CNN architecture hyperparameters act as experimental factors whose effects are later studied. Binary factor settings are defined for each factor; these settings are traditionally interpreted as high and low settings in DOE frameworks and are meant to exacerbate the effects each factor has on the system being studied. Six popular optimization routines currently employed to train CNNs are selected to be studied, and two popular benchmark data sets are identified to be used in conjunction with the experimental setup. Finally, each of the six identified optimization routines are individually trained on all 64 experimental runs dictated

by the resolution IV, fractional-factorial experimental design with 11 replicates, for each of the two data sets considered in this study.

As in the previous contribution, an experimental run in this setting consists of a combination of settings for each of the experimental factors included in the study; replicates of a single experimental run are repeated experiments using the same unique combination of settings to account for system variability. With 64 factor combinations considered per replicate and 11 replicates, the resolution IV, fractional-factorial design leads to 704 total experimental runs per optimization routine, data set combination and a total of 8,448 total experimental runs considered in this study.

After all of the experimental runs are completed, a statistical analysis ensues. This analysis begins with exploratory statistical comparisons, followed by multivariate linear regression models and the construction of meta-learning frameworks.

4.3.2 Experimental Factor Considerations

Current research has identified several hyperparameters which are either known to or suspected to impact CNN training and generalization. This study selects nine of these factors for inclusion in the experimental design to study the effects these hyperparameters and their combinations have on the underlying optimization problem associated with training CNNs.

BatchNorm layers introduced into deep learning architectures by Ioffe and Szegedy [42] are shown to improve the loss surface's β-smoothness by Santurkar et al. [83]. Santurkar et al. [83] also empirically show smoothing along gradient directions as well as the random directions that are studied. Ghorbani et al. [29] shows a similar smoothing effect in their work.

Dropout layers (Dropout) [87], can be implemented following different layer types; these layers are popularly used to manage overfitting. Dropout techniques reduce the

number of parameters capable of learning during each batch experienced while training a network with stochastic methods [87]. This method can improve generalization while also breaking local relationships known as feature co-adaptation that can arise between parameters Aggarwal [3].

Max pooling layers (Max Pooling) [78] provides a means of down-sampling features usually from convolutional layers [77], [84]. These layers can improve generalization while simultaneously reducing computational load through their ability to choose the most salient local features within small regions of their input layers.

Network width is a common hyperparameter related to the number of neurons which exist in each layer of the network. Li, Ding and Sun [57] extend the work of Yu and Chen [96], showing that if there are more neurons than samples in the last layer of a network, the deep network's loss surface has no set-wise strict local minima. Shallow, wide networks are successful in some applications [71].

An often competing architecture component to network width, network depth defines the number of layers which exist in the network. Safran and Shamir [81] show that deeper networks can approximate functions better than shallower networks, even if the shallower networks are wider; this finding is also supported by Liang and Srikant [61]. Dinh et al. [21] show that deep learning architectures can still generalize well even in sharp minimas; however, Li, Xu, Taylor, Studer and Goldstein [59] show that network depth increases chaotic elements of the loss surface in ResNet architectures and can make them much harder to train.

Different activation functions have been shown to impact loss surface behavior [25], [35], [62]. Rectifier linear units (ReLU) and Leaky ReLu activation functions are both widely used and well-studied [47], [33], [3]. Both of these activation functions introduce nonlinear relationships into a network, improving the potential expressiveness of the system [3]. Both ReLu and Leaky ReLu activation functions seek to

address issues related to the vanishing and exploding gradient problems which can occur during training [33], [3].

Initialization of network decision variables has proven to be a very important consideration when designing a CNN [83]. For instance, deeper networks can have exponentially more difficulty learning due to the compounding effect small initial decision variables can have on later layers in the network [10], [72], [53], [88]. Two of the most widely used methods today are Xavier Initialization and He Initialization, the former is proposed by Glorot and Bengio [30] and the latter is proposed by He et al. [37].

4.3.3 Experimental Design

Using the architecture hyperparameters described in the previous section, a resolution IV, fractional-factorial experimental design is employed to dictate the individual experimental runs, using the architecture hyperparameters to build experimental factors. This design is ideal for factor screening as it does not alias any main effects with second-order effects, and only some second-order effects are aliased with each other. The second-order alias structure is provided in Table 9. The selected resolution IV design requires 64 total experimental runs to generate a single replicate; a total of 11 replicates are conducted for each optimization routine, data set combination.

Table 9. Two-way interaction alias structure.

Two-Way Interaction	Alias
Convolutional Width * Fully-Connected Width	Max Pooling * Initialization
Convolutional Width * Max Pooling	Fully-Connected Width * Initialization
Convolutional Width * Initialization	Fully-Connected Width * Max Pooling

For each experimental factor combination, a neural network is developed using the factor settings associated with the factor combination and subsequently trained

for 10 total epochs using a standard batch size of 64 images.

After training, the final training losses, measured using cross-entropy loss, total training time elapsed (in seconds), and the test losses, also measured using cross-entropy loss across the data set's test set, are recorded for later analysis.[1]

Table 10. Experimental factor settings.

Factor	High Setting (+1)	Low Setting (-1)
Activation	ReLu used for all Convolutional and FC layers	Leaky ReLu used for all Convolutional and FC layers
Max Pooling	Used after Convolutional layers, before activation	Not used at all
Initialization Method	Xavier Uniform initialization	He Uniform initialization
Dropout Layers	Used after Convolutional and FC layers	Not used at all
BatchNorm Layers	Used after Convolutional and FC layers	Not used at all
Convolutional Width	Wide layers	Narrow layers
Fully-Connected Width	Wide layers	Narrow layers
Convolutional Depth	Ten layers	One layer
Fully-Connected Depth	Ten layers	One layer

The goal with these settings is not to necessarily develop state-of-the-art or even remotely successful network setups but rather to create instances that emphasize the traits the individual factors as well as combinations of these factors impart onto the learning problem to study the ways in which individual training routines traverse the loss surface. As a result, many of the settings, above, might not be realistic in practice.

4.3.4 Optimization Routine Considerations

We consider six of the most current, popular optimization routines used to train CNNs. Three of these routines are non-adaptive: SGD-Vanilla, SGD-Momentum, and SGD-Nesterov; the other three routines are adaptive: AdaGrad, RMSProp, and Adam. Each optimization routine uses its default hyperparameter settings defined by PyTorch [73]. These settings can be found in Table 11, below. Our choice to keep the hyperparameter settings at their default levels simulates an initial guess of settings

[1] All experimental runs were conducted using an HP Z8 G4 Workstation with Intel(R) Xeon(R) Silver 4210 CPU @ 2.20GHz, 128 GB RAM; PyTorch was used to implement the neural network instances [73].

when training a neural network architecture for the first time.

Table 11. Default hyperparameter settings for each optimization routine.

Optimization Routine	Default Learning Rate	Default Additional Hyperparameters
SGD-Vanilla	0.01	Not Applicable
SGD-Momentum	0.01	Momentum = 0.9
SGD-Nesterov	0.01	Momentum = 0.9
AdaGrad	0.01	$\epsilon = 1e-10$
RMSProp	0.01	$\alpha = 0.99$, $\epsilon = 1e-08$
Adam	0.001	$\beta^1 = 0.9$, $\beta^2 = 0.999$, $\epsilon = 1e-08$

4.3.5 Data Set Considerations

Two of the most popular benchmark data sets used to train CNNs are the grey-scale data set, FashionMNIST [94], and the 3-channel color data set, CIFAR-10 [51]. The FashionMNIST data set is comprised of 28x28 images that span 10 classes of clothing articles with 6,000 examples per class and a test set of 10,000 examples. A more challenging data set, CIFAR-10, is comprised of 32x32 images that span 10 classes of common animal and object images with 6,000 examples per class and a test set of 10,000 examples. Both of these data sets are considered in this study, separately.

4.3.6 Response Variables

This study focuses on identifying the effects that common CNN architecture hyperparameters have on different optimization routines' abilities to train CNNs and find solutions that generalize well. The two response variables we consider are final training losses found at the end of training and the total test losses calculated across all test examples exposed to a CNN instance. Final training loss is measured with the final batch found during training, an estimate of the local loss value in the region found by an optimization routine. Total test loss, a measure of a network's ability

to generalize, is found by calculating the sum of individual total cross-entropy loss values found for each image used to evaluate the network after training is completed; this stage of the modeling process is commonly referred to as the testing phase. The cross-entropy loss function is a common loss function used for image classification tasks [3].

4.3.7 Experimental Runs

Each experimental run consists of first building a neural network using runs of specific factor level settings provided in Table 10; levels for each factor combination are dictated by the settings in the fractional-factorial, resolution IV design, found in Section 7.1. The network is trained using each of the optimization routines considered in this study, independently: SGD-Momentum, SGD-Nesterov, SGD-Vanilla, RMSProp, Adam, and AdaGrad. Final training loss and total test loss on a test set are measured and stored. Eleven replicates are conducted for each experimental factor combination. This process is conducted on both the CIFAR-10 and FashionMNIST data sets, resulting in 4,224 experimental runs conducted across the six optimization routines on each of the individual data sets or 8,448 experimental runs in total.

4.4 Results

The following subsections explore the effects that network architecture can have on the training process and generalization capabilities during early-learning by different optimization routines applied to train CNNs. First, divergent instances found during experimentation are analyzed to identify commonalities in architecture hyperparameter settings that might have led to this behavior. Next, comparisons are made between optimization routines' performance levels across all of the instances studied for both, the FashionMNIST and the CIFAR-10, data sets. Optimization routine

performance across is evaluated using final training loss and total test loss.

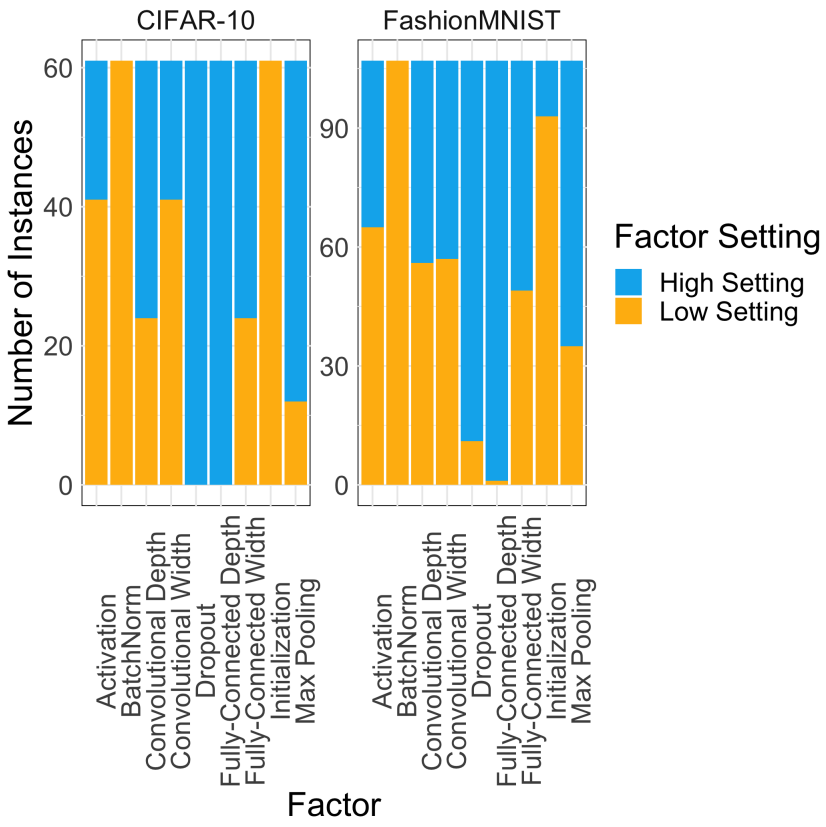
After these comparisons, multivariate linear regression models are fit to the train and test losses observed during experimentation. The regression models are built for each optimization routine, data set combination, separately. Considering all optimization routine, data set combinations, there are in total 24 different individual regression models that are built and studied to identify commonalities and differences in how these optimization routines are affected by architecture hyperparameters in terms of their abilities to find solutions that yield both, low final training losses as well as total total test losses.

Finally, the same multivariate regression models are used to build four separate meta-learning models that can be used to predict the optimization routines which achieve the minimum mean final training losses and total test losses on each of the considered data sets. These models demonstrate that there is a relationship between CNN architecture hyperparameter decisions and optimization routine selection.

4.4.1 Divergent Instances

Before making statistical comparisons between the individual optimization routines considered in this study, we analyze the factors associated with the divergent instances. Figure 8 illustrates the distributions of each factor level for each data set, respectively, across instances that exhibit divergent behavior.

Figure 8. Factors associated with divergent instances.



All observed instances that exhibit true divergence occurred when using non-adaptive methods; additionally, as illustrated in Table 12, almost all of the divergent instances did not utilize BatchNorm and had ten fully-connected layers. This might suggest that BatchNorm's smoothing effect [83] plays an important role in keeping non-adaptive methods from diverging in deep, dense networks. We note that adaptive methods might avoid true divergent behavior through their ability to adjust to local loss surface features, reducing step sizes in volatile regions.

Table 12. Number of divergent cases where each optimization routine that exhibited divergence, by data set.

Data Set	Optimization Routine	Number of Divergent Instances
CIFAR-10	SGD-Momentum	22
CIFAR-10	SGD-Nesterov	27
CIFAR-10	SGD-Vanilla	12
FashionMNIST	SGD-Momentum	45
FashionMNIST	SGD-Nesterov	49
FashionMNIST	SGD-Vanilla	13

In Table 12 it can be seen that SGD-Momentum and SGD-Nesterov exhibited the most divergent cases over both data sets. SGD-Vanilla exhibited divergence but much less than its counterparts, suggesting that making larger steps can lead to poor updates in some cases when compared to the smaller, slower steps of SGD-Vanilla.

4.4.2 95% Pairwise Student's T-Confidence Intervals Across All Non-Divergent Instances

Comparisons using 95% pairwise Student's t-confidence intervals are made between all optimization routines, across all instances that do not diverge or are otherwise removed; if an instance diverges or is removed for at least one optimization routine, then the corresponding seed related to this divergent instance is removed from consideration for all optimization routines. Additional runs are removed from each optimization routine's data set if they are identified as influential points using Cook's Distance [19]. This removal process is done to ensure that comparisons are not biased towards optimization routines that achieved lower train and test losses on non-divergent instances and instances deemed as outlier behavior using Cook's Distance while exhibiting true divergent or otherwise unfavorable behavior on instances in which another optimization routine did not.

Considering the remaining experimental runs, 95% Student's t-confidence intervals

are developed to statistically compare the final training and test losses across all instances between the different optimization routines.

As illustrated in Figure 9, RMSProp's performance is statistically worse than all of the other optimization routines considered in terms of training loss on both the CIFAR-10 and FashionMNIST data sets, exhibiting the highest mean loss values as well as most variability. Adam's performance is statistically better than SGD-Vanilla and AdaGrad on both data sets. SGD-Momentum, SGD-Nesterov, and AdaGrad are not statistically different from each other on either data set. Finally, AdaGrad and SGD-Vanilla are not statistically different on either data set.

Figure 9. 95% student t-confidence intervals of final training loss between alternative optimization routines.

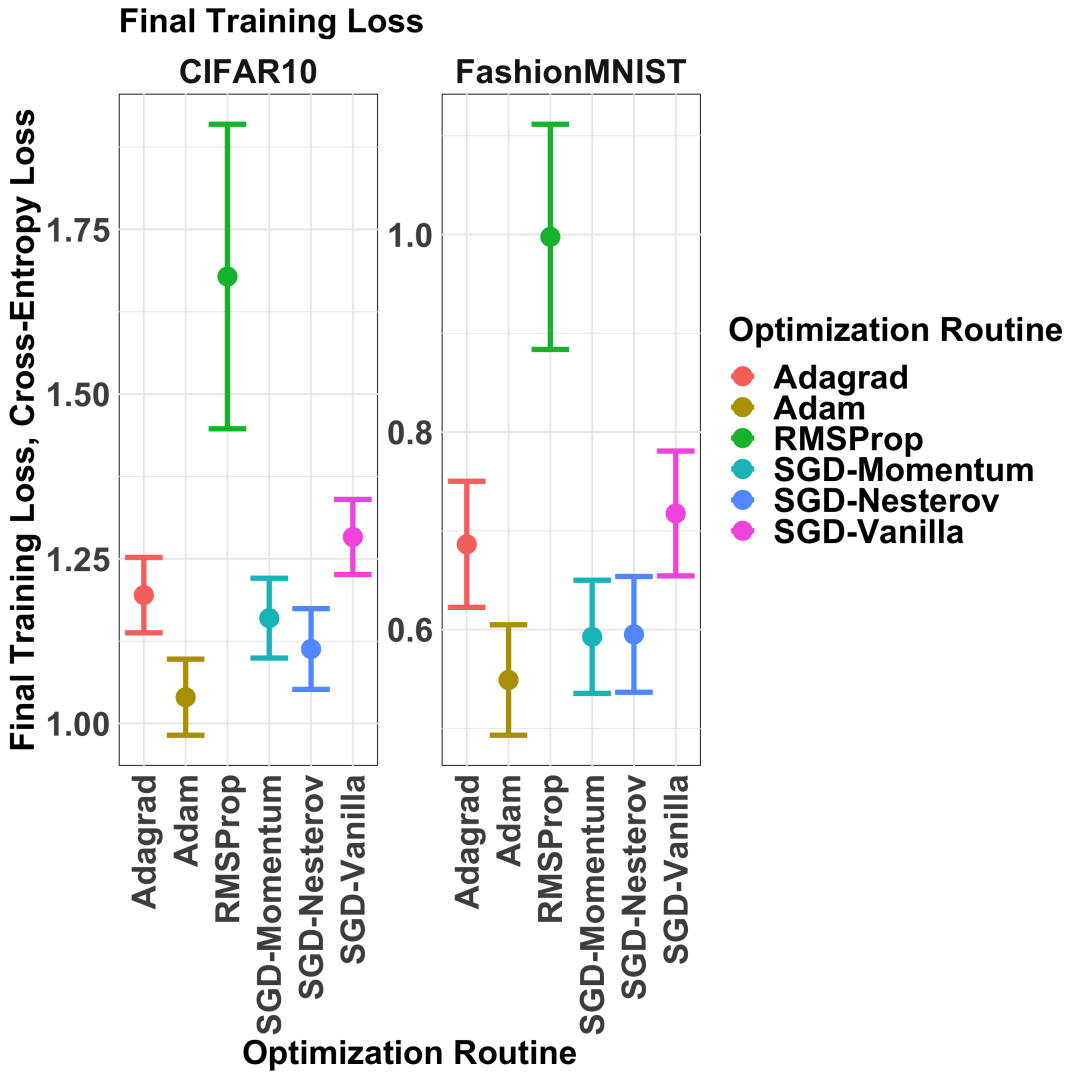
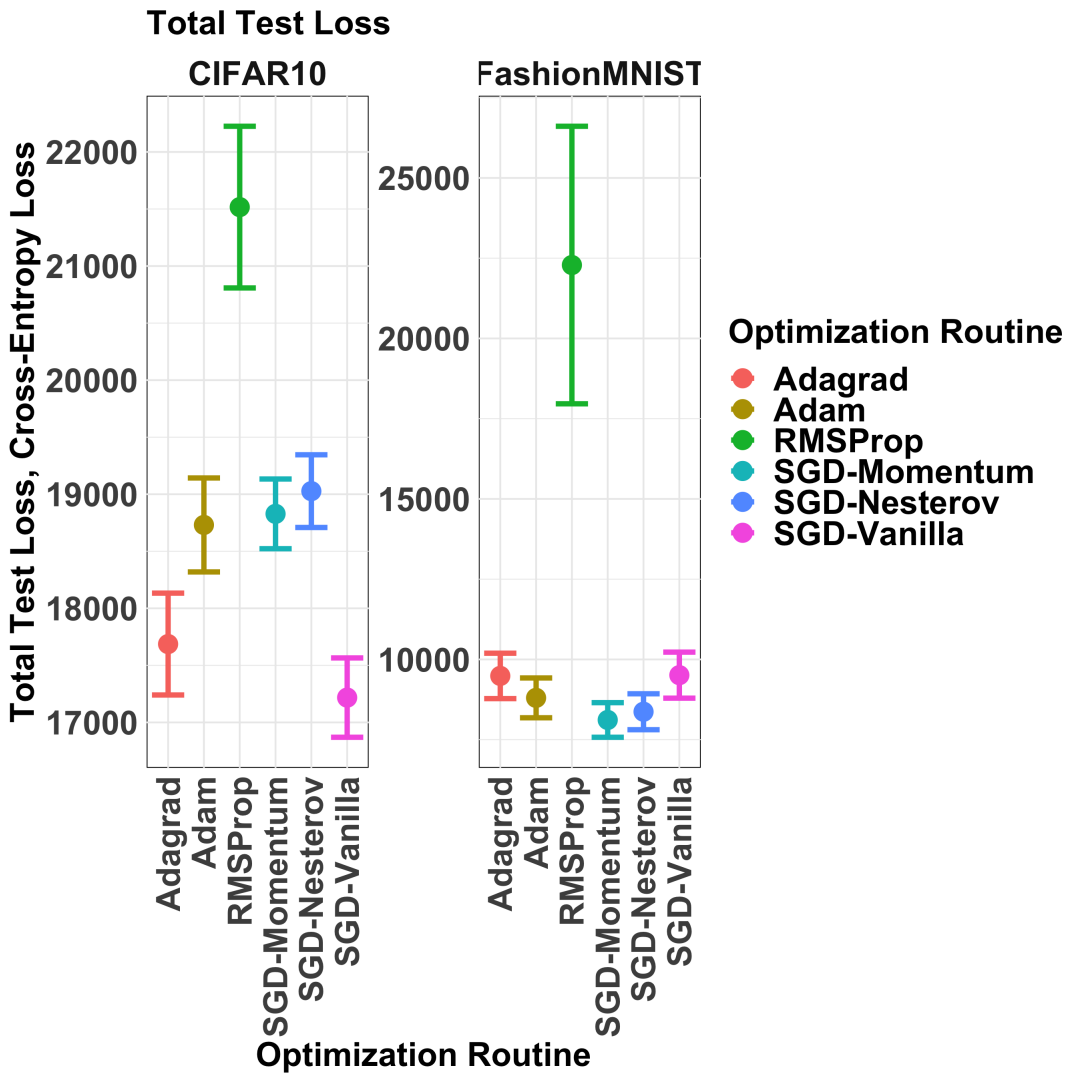


Figure 10 illustrates confidence intervals of each optimization routine's test losses on either data set. RMSProp's performance is statistically worse than all of the other optimization routines considered in terms of training loss on both the CIFAR-10 and FashionMNIST data sets, exhibiting the highest mean loss values as well as most variability. Adam, AdaGrad, SGD-Momentum, and SGD-Nesterov are not statistically different from each other on the FashionMNIST data set. SGD-Vanilla is statistically slightly worse than SGD-Momentum; on the harder CIFAR-10 data set, there is more

separation between these optimization routines. SGD-Vanilla statistically outperforms all other optimization routines with the exception of AdaGrad. Additionally, AdaGrad statistically outperforms Adam, SGD-Nesterov, and SGD-Nesterov on the CIFAR-10 data set and is arguably the second-best optimization routine on this data set. Adam, SGD-Nesterov, and SGD-Momentum are not statistically different on the CIFAR-10 data set.

Figure 10. 95% student t-confidence intervals of total test loss between alternative optimization routines.



These comparisons illustrate a similar finding to the intuition provided by Wil-

son et al. [91] - reducing training loss effectively does not always correspond to a better solution quality on unseen data across these different optimization routines. Additionally, Adam, an adaptive method, showed stronger training results than test results, further suggesting that adaptive methods may not necessarily test as well as they train. Finally, SGD-Vanilla performs considerably better in finding high quality solutions in terms of total test loss during the early-learning periods studied compared to the quality of its solutions relative to other routines on training data.

4.4.3 Pairwise Two-Tailed 95% Student's T-Test Comparisons

To increase the level of fidelity of when optimization routines might statistically differ across the different experimental setups that are considered, two-tailed 95% Student's t-tests are developed to make pairwise comparisons between optimization routines within the different experimental setups. For each experimental setup, there are a total of 66 observations (11 replicates corresponding to each of the six optimization routines). This information is adjusted using the same removal process described in the previous section to ensure pairwise comparisons contain consistent information across all optimization routines.

Results for the Adam routine are found in Figure 12 and Figure 11 for both final training and total test loss comparisons, respectively. Individual charts associated with comparisons for each of the other optimization routines' final training and total test losses can be found in Sections 7.3 and 7.4.

Figure 11 shows that Adam performs statistically worse than RMSProp more frequently on different experimental setups than when compared with the other optimization routines on both, CIFAR-10 as well as FashionMNIST. It can also be seen from Figure 12 that SGD-Vanilla and AdaGrad are the most comparable to Adam on instances trained using CIFAR-10 as well as on FashionMNIST in terms of total

test loss.

Figure 11. Adam pairwise, two-tailed 95% student's t-tests of final training loss between alternative optimization routines.

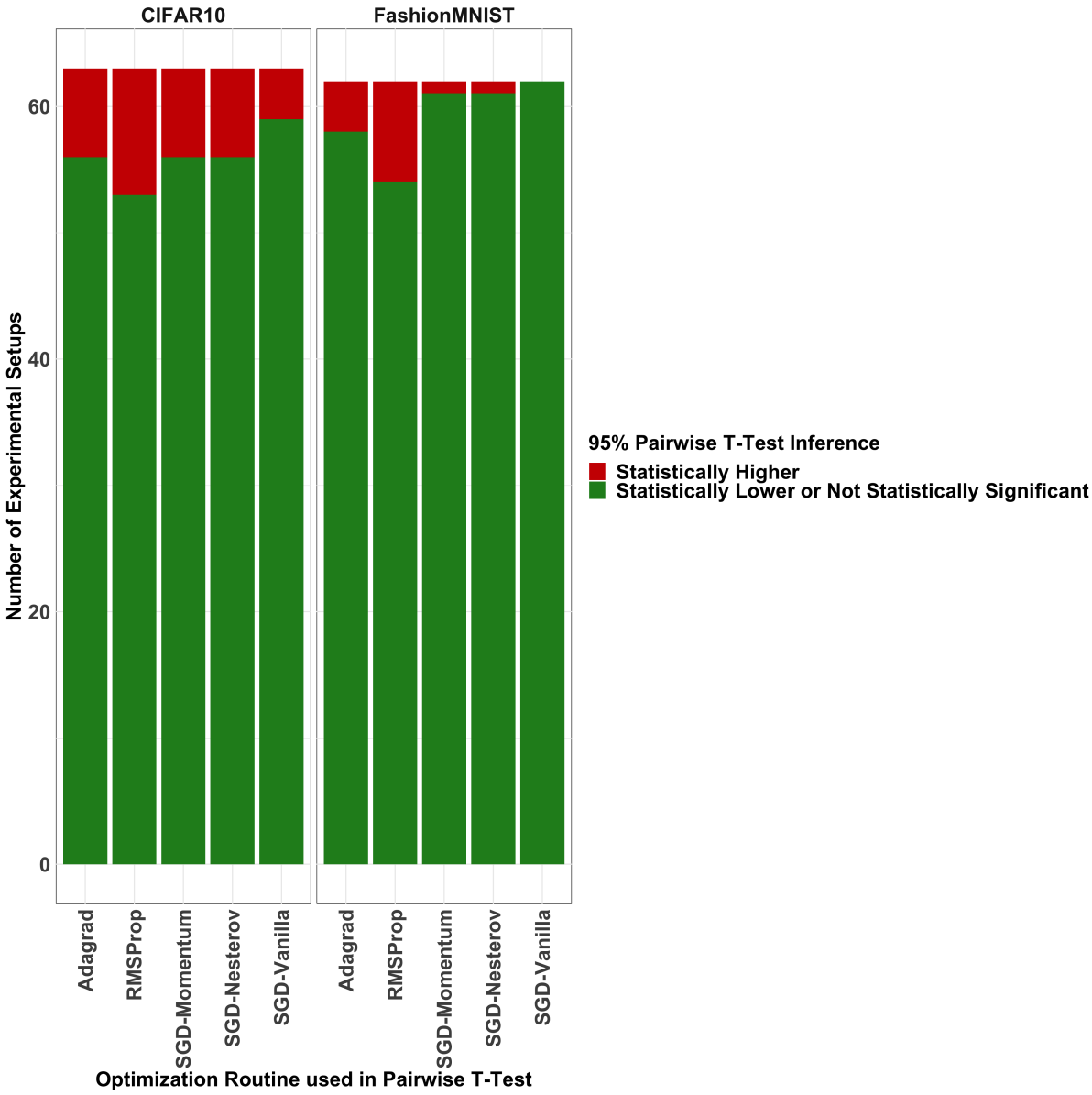
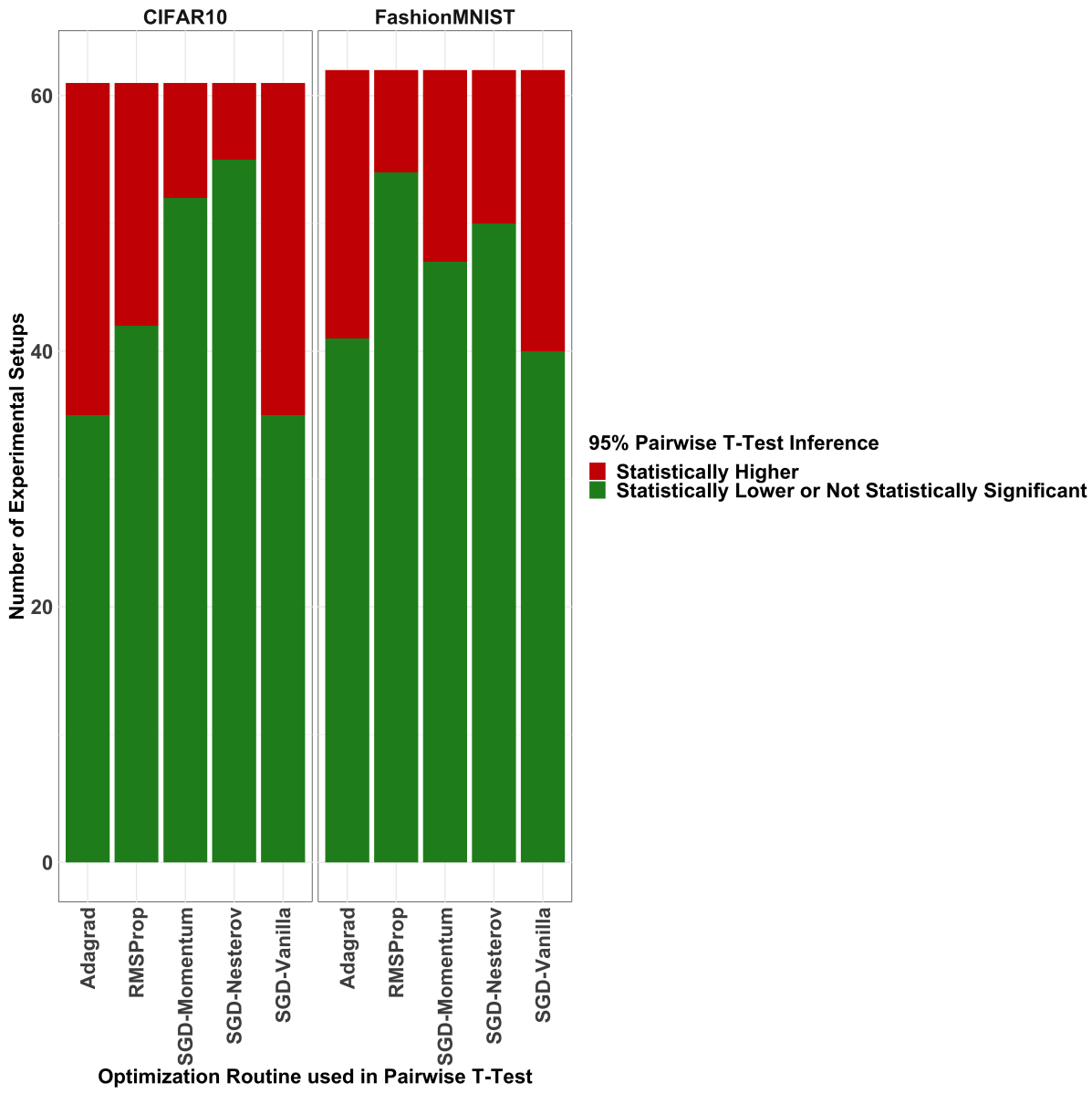


Figure 12. Adam pairwise, two-tailed 95% student's t-tests of total test loss between alternative optimization routines.



In general, these comparisons and the comparisons made in the previous section show that although RMSProp tends to do consistently worse both in terms of final training loss and total test loss relative to the other optimization routines, no optimization routine clearly outperforms all of the others considered on either data set,

always. Furthermore, this section illustrates the same trend illustrated in the previous section with greater fidelity - if an optimization routine is capable of training well, it may not necessarily generalize well. The following section uses multivariate linear regression models to identify statistically significant architecture hyperparameters which affect the final training and total test loss outcomes for each optimization routine.

4.4.4 Multivariate Linear Regression

Multivariate linear regression is used to identify architecture hyperparameters and second-order interactions of hyperparameters that statistically impact each individual optimization routine's ability to train and test well. In total, there are 24 second-order regression models, modeling each the final train losses and total test losses achieved by each optimization routine, separately on both data sets. We use an α significance level of 0.01 to study factor effects.

Tables 13 and 14 summarize the results of models built for final training loss, by data set. Full model information for CIFAR-10 and FashionMNIST models of final training losses can be found in 7.5 and 7.6

Table 13. CIFAR-10 final training loss regression summaries.

Optimization Routine	Adjusted R^2	1^{st}-Order Significant Terms	Total 1^{st}-Order Terms	2^{nd}-Order Significant Terms	Total 2^{nd}-Order Terms
SGD-Nesterov	0.9375	8	9	15	33
SGD-Momentum	0.9378	7	9	15	33
SGD-Vanilla	0.9247	7	9	10	33
RMSProp	0.8682	7	9	26	33
Adam	0.9412	9	9	19	33
AdaGrad	0.9357	9	9	17	33

Table 14. FashionMNIST final training loss regression summaries.

Optimization Routine	Adjusted R^2	1^{st}-Order Significant Terms	Total 1^{st}-Order Terms	2^{nd}-Order Significant Terms	Total 2^{nd}-Order Terms
SGD-Nesterov	0.949	7	9	23	33
SGD-Momentum	0.948	8	9	24	33
SGD-Vanilla	0.956	8	9	14	33
RMSProp	0.882	6	9	16	33
Adam	0.964	8	9	14	33
AdaGrad	0.958	9	9	20	33

Across both sets of final training loss model summaries, it can be seen that a large amount of the underlying response variation can be explained using the architecture hyperparameters used in our study. We also see that a large number of both first and second-order terms show statistical significance, indicating that the underlying training problem is truly being impacted by a variety of architecture hyperparameter decisions as well as their combinations.

The following two multivariate regression model summaries found in Tables 15 and 14 summarize the results of models built for total test loss, by data set. Full model information for CIFAR-10 and FashionMNIST models of total test losses can be found in 7.7 and 7.8.

Table 15. CIFAR-10 total test loss regression summaries.

Optimization Routine	Adjusted R^2	1^{st}-Order Significant Terms	Total 1^{st}-Order Terms	2^{nd}-Order Significant Terms	Total 2^{nd}-Order Terms
SGD-Nesterov	0.813	8	9	14	33
SGD-Momentum	0.810	6	9	13	33
SGD-Vanilla	0.908	9	9	14	33
RMSProp	0.770	7	9	18	33
Adam	0.836	7	9	17	33
AdaGrad	0.874	8	9	15	33

Table 16. FashionMNIST total test loss regression summaries.

Optimization Routine	Adjusted R^2	1^{st}-Order Significant Terms	Total 1^{st}-Order Terms	2^{nd}-Order Significant Terms	Total 2^{nd}-Order Terms
SGD-Nesterov	0.916	8	9	19	33
SGD-Momentum	0.925	8	9	20	33
SGD-Vanilla	0.941	6	9	17	33
RMSProp	0.741	4	9	13	33
Adam	0.919	6	9	15	33
AdaGrad	0.932	6	9	14	33

Again, across both sets of total test loss model summaries, it can be seen that a large amount of the underlying response variation can be explained using the architecture hyperparameters used in our study. We also see that a large number of both first and second-order terms show statistical significance, indicating that an optimization routine's ability to find solutions which achieve good generalization capabilities are truly being impacted by a variety of architecture hyperparameter decisions as well as their combinations.

In summary, across all models, high amounts of the underlying response variability can be explained using linear regression with first and second-order terms[2].

All models have strong overall significance levels, and the explanatory power of the models is spread over a large number of statistically significant terms in every case. The large numbers of statistically significant terms exhibited across the models enforce the findings found in our first contribution, further suggesting the choice of architectural features highly impacts the underlying optimization problem. Additionally, with different terms showing significance for different optimization routines and in some cases with reverse signs of parameter estimates, we find that the ef-

[2]Variance inflation factors (VIFs) are generated for every factor in each model to ensure that there is minimal multicollinearity within these models; there are no VIFs which exceed a value of 1.25, suggesting multicollinearity is very limited in these models.

fects induced by these architecture hyperparameters on the underlying optimization problems might be exploited in different ways by the different optimization routines considered.

To study the similarities and differences between architecture hyperparameters' effects on optimization routines, cross-listings of the significant terms between the linear regression models for each loss type, data set combination are used to compare the significant terms, identifying commonalities and differences that exist. The cross-listing procedure consists of first identifying statistically significant factors across all linear regression models corresponding to a loss type, data set combination and then comparing these factors' coefficient signs. Coefficients corresponding to statistically significant factors which match in sign across all linear regression models correspond to those which affect each optimization routine in the same way for a loss type, data set combination. These commonalities point towards architecture features that when selected can be expected to consistently affect the outcome of the learning process. The results of these cross-listings can be found, below, for CIFAR-10 final training loss, FashionMNIST final training loss, CIFAR-10 total test loss, and FashionMNIST total test loss in Tables 17, 18, 19, and 20, respectively. An $\alpha = 0.01$ is employed for all cross-listings.

Table 17. CIFAR-10 final training loss significant terms cross-listed across all second-order regression models at a significance level of $\alpha = 0.01$.

Factor	Factor Effect
BatchNorm	Decreases Training Loss
Convolutional Depth	Increases Training Loss
Dropout	Increases Training Loss
Fully-Connected Depth	Increases Training Loss
Max Pooling	Increases Training Loss
Convolutional Depth * Fully-Connected Depth	Decreases Training Loss
Convolutional Depth * Max Pooling	Increases Training Loss
Fully-Connected Depth * BatchNorm	Increases Training Loss
Fully-Connected Depth * Dropout	Increases Training Loss

Table 18. FashionMNIST final training loss significant terms cross-listed across all second-order regression models at a significance level of $\alpha = 0.01$.

Factor	Factor Effect
BatchNorm	Decreases Training Loss
Convolutional Depth	Increases Training Loss
Dropout	Increases Training Loss
Fully-Connected Depth	Increases Training Loss
Max Pooling	Increases Training Loss
Convolutional Depth * Fully-Connected Depth	Decreases Training Loss
Convolutional Depth * Max Pooling	Increases Training Loss
Fully-Connected Depth * BatchNorm	Increases Training Loss
Fully-Connected Depth * Dropout	Increases Training Loss
Fully-Connected Depth * Max Pooling	Decreases Training Loss

There are very few factors which consistently affect all optimization routines in consistent, statistically significant ways in terms of finding solutions which minimize training loss well during the early-learning stages. This indicates that the training process exhibited by different optimization routines may be very unique to an optimization routine in terms of its relationship with architecture hyperparameters with just a subset of architecture hyperparameters showing a consistent, significant effect on the learning process of each routine.

Table 19. CIFAR-10 total test loss significant terms cross-listed across all second-order regression models at a significance level of $\alpha = 0.01$.

Factor	Factor Effect
Dropout	Increases Test Loss
Fully-Connected Width	Increases Test Loss
Convolutional Depth * Max Pooling	Increases Test Loss
Fully-Connected Depth * Dropout	Increases Test Loss

Table 20. FashionMNIST total test loss significant terms cross-listed across all second-order regression models at a significance level of $\alpha = 0.01$.

Factor	Factor Effect
Dropout	Increases Test Loss
BatchNorm * Activation	Increases Test Loss
Convolutional Depth * Max Pooling	Increases Test Loss
Fully-Connected Depth * Dropout	Increases Test Loss

There are very few factors which consistently affect all optimization routines in consistent, statistically significant ways in terms of finding solutions which generalize well during the early-learning stages. This further emphasizes the findings from the previous pair of cross listings, indicating the relationship between an optimization routine and the architecture hyperparameters of a neural network can be unique for each of the optimization routines studied.

As a final point of comparison, we cross list the factors which show consistent factor effects on training results across all optimization routines and are also statistically significant with those which consistently affect the underlying loss surface's second-order Lipschitz constants found in the first contribution. Table 21, below, illustrates our findings, showing that there is an inverse relationship between curvature levels and training loss in the early-learning stages across the data sets and instances we study.

Table 21. Cross-listing of significant factor effects which affect curvature measurements in the directions studied and affect final training losses across FashionMNIST and CIFAR-10.

Factor	Effect on Lipschitz Bound	Effect on CIFAR-10 Training Loss	Effect on FashionMNIST Training Loss
BatchNorm	Increases Upper Bound	Decreases Training Loss	Decreases Training Loss
Convolutional Depth	Decreases Upper Bound	Increases Training Loss	Increases Training Loss
Dropout	Decreases Upper Bound	Increases Training Loss	Increases Training Loss
Fully-Connected Depth	Decreases Upper Bound	Increases Training Loss	Increases Training Loss
Convolutional Depth * Fully-Connected Depth	Increases Upper Bound	Decreases Training Loss	Decreases Training Loss
Fully-Connected Depth * Dropout	Decreases Upper Bound	Increases Training Loss	Increases Training Loss

There appears to be an inverse relationship between hyperparameters' affects on curvature in the early learning stages and early training loss estimates; as curvature of the local loss surface is decreased, the amount of observed training loss is increased during this stage of learning. Although not all factors which affect curvature have this relationship with those that appeared to statistically affect final training loss estimates, this subset of factors show strong signs of being highly influential to the early-learning process.

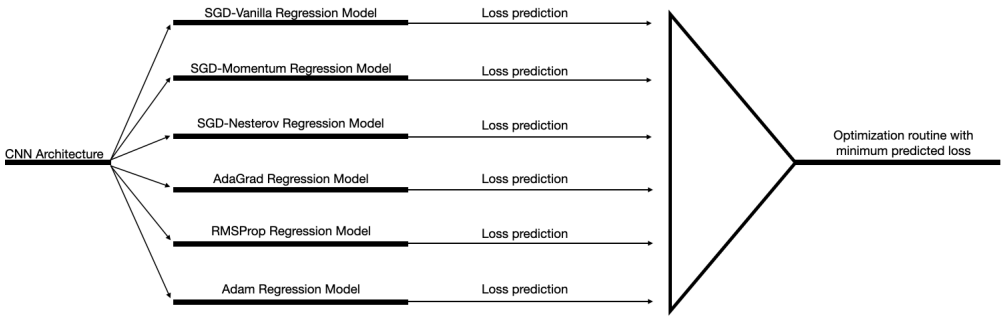
4.4.5 Meta-Learning Models for Train and Test Losses

Meta-learning models for both final training and test losses on CIFAR-10 and FashionMNIST are developed to accurately predict the optimization routines which achieve the minimum mean loss values given specific CNN architecture decisions. By achieving a high level of predictive power, these meta-learning models indicate that we can obtain predictable learning outcomes by matching the choice of an optimization routine with architecture hyperparameters that promote its ability to achieve better performance relative to the other routines that are considered in this study during the early-learning stages. Each of the meta-learning models utilizes the six multivariate linear regression models described in Section 4.4.4 for a loss type, data set combination. In total there are four meta-learning models.

The process of obtaining a prediction from a single meta-learning model occurs by first providing a vector of -1s and 1s, corresponding to factor settings for each of the

nine factors, to each individual regression model associated with a specific loss type, data set combination. After each regression model independently makes a prediction using the setup vector, the minimum predicted loss value across the six independent regression models is selected. This minimum value and the optimization routine corresponding to the linear regression model which made the prediction are stored for that architecture setup as the meta-learning model's predicted best optimization routine. Figure 13, below, illustrates the general procedure followed by each meta-learning model.

Figure 13. Meta-learning model framework.



A model's hit rate is calculated by counting the number of architecture setups in which the predicted optimization routine achieving the minimum mean loss value matches with the optimization routine which actually empirically achieves the minimum mean loss value (Hit Rate, A) or one of the minimum two mean loss values (Hit Rate, B). Summaries of results to evaluate the efficacy for each of the meta-learning models can be found in Table 22.

Table 22. Meta-learning model hit rates by data set, loss type combination used to predict optimization routine(s) achieving minimum mean loss values. Hit Rate, A corresponds to a meta-learning model's ability to predict the optimization routine that achieves the single lowest empirical mean loss value. Hit Rate, B corresponds to a meta-learning model's ability to predict an optimization routine that achieves one of the top two lowest empirical mean loss values.

Data Set	Loss Type	Hit Rate, A	Hit Rate, B
CIFAR-10	Final Training Loss	0.66	0.88
FashionMNIST	Final Training Loss	0.73	0.89
CIFAR-10	Total Test Loss	0.67	0.94
FashionMNIST	Total Test Loss	0.66	0.80

Across all meta-learning models, each model is able to use a network architecture setup to predict the optimization routine which achieves the minimum mean loss value in over 60% of the architecture setups considered. Additionally, across all meta-learning models, each model is able to use the network architecture to correctly predict one of the two optimization routines which achieve the minimum empirical mean loss values for an architecture setup in at least 80% of the architecture setups considered in this study. These findings show that the relationship between an optimization routine's success on a network with specific architecture hyperparameters and its ability to outperform other popular optimization routines during early learning on the same architecture setup can be modeled accurately.

4.5 Discussion

4.5.1 Overall Observations

Although two of the adaptive methods that are considered in this study, Adam and AdaGrad, train well relative to the other optimization routines, neither of these optimization routines consistently outperform non-adaptive methods in finding solutions which achieve the lowest mean total test loss values, even in the early stages of learning. These findings empirically support the results of Wilson et al. [91] and oth-

ers that have recently critiqued adaptive optimization routines' abilities to generalize well even if they can train effectively, especially in the early stages of learning.

Although adaptive methods are not definitively better; they show strong success on specific experimental setups. Furthermore, non-adaptive methods are also not definitively better across all of the instances that are considered in this study. Instead, our findings indicate that specific network architecture hyperparameters can have large impacts on the early-learning process. A small minority of architecture hyperparameters and combinations of these hyperparameters impact all of the optimization routines in the same way while the majority of these hyperparameters and combinations affect each routine, differently.

Although most factors considered do not affect all optimization routines in the same ways, there are still commonalities between the factors which affect early training and test results. Through the analysis of these architecture hyperparameters' effects on each optimization routine's ability to achieve low loss values with multivariate linear regression, we find high adjusted R^2 values and highly significant regression models. The findings from these regression models show that large amounts of the underlying variation exhibited for both train and test losses can be explained through the choices associated with network architecture hyperparameters. The developed models provide a means of identifying CNN architecture hyperparameters that work well with an identified optimization routine to reduce loss quickly in the early stages of learning.

Finally, the high predictive capability of the meta-learning models reinforces the notion implied through the regression model findings - optimization routine selection should be at least partially dictated by the prevailing architecture choices of a specific CNN instance.

4.5.2 Limitations and Next Steps

This study utilized a two-level, fractional factorial design with replicates. This design choice did not allow for nonlinear relationships to be modeled between the response variable and experimental factors when building the multivariate linear regression models. A few of the residual plots studied during the regression analysis show that there are possibly some signs of curvature in the studied response variables. Addressing potential nonlinearity in future designs with center points for factors that can be numerically represented could lead to more powerful regression models.

Unavoidable consequences of measuring the specific response variables in this study are the inherent limits associated with the range that these values can take based on the data types and specific network architectures considered. The predisposed upper and lower limits inherent to an architecture reduce the power of the regression models by reducing the variability between some of the individual observations. Lower limits for loss values are associated with the theoretical minimums which can be achieved for each architecture setup. Additionally, unless an optimization routine traverses into very poor regions of the loss surface, there is generally also an observable upper limit for loss values; that is, when a routine performs poorly it tends to not dramatically improve or worsen its solution quality.

Divergent instances are an unavoidable element in using this experimental setup and cannot be stopped from occurring without further biasing the results. In the cases of SGD-Momentum and SGD-Nesterov, two full experimental setups are completely removed when building the regression models as all 11 replicates for each of these experimental setups diverge. This amount of data loss causes the associated regression models for each of these routines to lose their orthogonality properties, which could cause some bias of the coefficients in these models. Although regression has been shown to be robust to information loss as described by Montgomery [67], future

designs should consider alternative factor levels to ensure divergence is mitigated as much as possible. It is also important to note that although this data loss is not ideal, it allows for further analysis of factors that lead non-adaptive methods to divergent behavior.

Finally, this study identifies nine popular hyperparameters that are considered in CNN architecture design with two levels assigned for each factor. Future design choices could provide more granularity in understanding the effects of these hyperparameters by choosing experimental designs that allow for additional factor levels. There are also many more CNN architecture hyperparameters that could be considered.

4.6 Conclusion

This chapter focuses on studying the decision space related to optimization routine selection when training CNNs with specific architecture hyperparameters. We first statistically compare final training loss and total test loss results between different optimization routines across two data sets and a variety of different instances generated by a resolution IV, fractional-factorial experimental design with replicates. Our results show that although adaptive methods might train and test well in many instances, there are also a large number of instances in which they perform worse than their non-adaptive counterparts. We then identify architecture hyperparameters and combinations of these hyperparameters that impact train and test losses using multivariate linear regression models, finding strong explanatory power and large amounts of statistical significance in the respective effects of these hyperparameters. The results of these models are cross-referenced with each other to find similarities and differences between the factors that statistically impact learning. Finally, meta-learning models are built using the regression models to accurately predict the best

performing optimization routines of those considered in this study, enforcing findings that architecture can dramatically affect optimization routines' performances in sometimes very different ways. Through these experiments, insights are provided that help to distinguish optimization routine selection given specific CNN architecture hyperparameters. This is an exciting research direction, and we believe that continuing to statistically identify relationships between optimization routine performance and underlying network architectures can dramatically reduce lead times in developing state-of-the-art learning systems.

V. Contribution III: Accelerating Non-Adaptive, First-Order Methods with Lagrangian Duality and Localized Euclidean Trust Regions

5.1 Introduction

The process of training a deep CNN is generally considered to be a stochastic, non-convex, large-scale optimization problem. Due to the nonconvexity of this problem, large-scale, state-of-the-art convex methods cannot be utilized, directly. In practice, the scale of this optimization problem does not warrant the ability to estimate full, second-order information, resulting instead in the use of both adaptive and non-adaptive, first-order optimization routines. Non-adaptive, first-order methods employ global hyperparameters which must be adjusted manually over the course of training; these methods often make progress more slowly than their adaptive counterparts but have been recognized to yield models that often generalize favorably when compared with adaptive methods. Adaptive methods, on the other hand, can make online adjustments to their hyperparameters over the course of learning. These methods are very popular as they can have faster convergence rates in practice compared to non-adaptive methods due to their ability to provide more specialized learning information to decision variables. Although adaptive methods train well, there has been criticism suggesting that they do not generalize well [91].

In this chapter, we equip non-adaptive, first-order routines with an acceleration method inspired by updates made by Non-Euclidean Restricted Memory Level (NERML) bundling, a state-of-the-art, large-scale optimization routine. Our method not only yields a first-order optimization routine that accelerates early learning of non-adaptive, first-order routines to a level competitive with adaptive methods, but it also demonstrates the efficacy of applying updates akin to large-scale convex optimization to inform updates made on the non-convex instances encountered when

training deep learning models.

5.2 Background

5.2.1 Training Convolutional Neural Networks

The underlying optimization problem associated with training CNNs is large-scale, stochastic, and non-convex. A variety of elements can affect training success, ranging from architecture hyperparameters, described in our first contribution and optimization routine selection, described in our second contribution. Wilson et al. [91] show that the generalization gap might be greater for adaptive methods. Keskar et al. [45] show that larger batch sizes can lead to sharper minima which generalize poorly compared to those found with small batch sizes. The timing and type of regularization methods [31] has been shown to affect a neural network's long-term ability to learn. Initialization schemes such as He Initialization [37] and Xavier Initialization [30] have been shown to improve the overall training process.

Additionally, the loss surfaces of these instances can be highly complex. Expansive flat regions, sharp minima, and saddle points can drive an optimization routine into poor regions of the loss surface [20], [18], [45], [83].

Optimization routines used to train CNNs limit the assumptions made about the underlying global problem geometry and rely solely on oracle calls with gradient evaluations that come from backpropogation [80]. Additionally, in most neural network training instances, it is too costly to evaluate the gradient using the full training data set at once, and instead mini-batches are utilized to reduce the computational burden of gradient estimation [39]. These methods generally have slower convergence rates, but they have cheap computational costs per iteration [54].

5.2.2 Optimization Routines Used to Train Neural Networks

Currently, there are several first-order methods employed to train CNNs. These methods can be separated into two broad categories: non-adaptive methods and adaptive methods [91]. Non-adaptive methods utilize global hyperparameters that remain unchanged unless they have been scheduled to do so; if the hyperparameters are scheduled to change over the course of learning, they must be manually adjusted [33]. In contrast to non-adaptive methods, adaptive methods adjust hyperparameters to fit the local geometry in an online fashion. The capability of online adjustments reduces the need to choose optimization routine hyperparameters as carefully and scheduling hyperparameter adjustments can be less burdensome. Adaptive methods can reduce the time spent on tuning while simultaneously still achieving success in reducing training error quickly, especially in the early-learning stages [91], [46]. Due to their often fast training speeds, adaptive optimization routines are very popular, and can perform very well in several different applications, with Adam, proposed by Kingma and Ba [48], often being the default algorithm of choice for many modern-day applications [60], [44].

We briefly describe updates made by each optimization routine; in addition to the papers proposing each method, we refer to Gron [33] and Wilson et al. [91] for summaries of each.

- Stochastic gradient descent is commonly used to train neural networks, making its updates by estimating the gradient and subtracting the gradient multiplied by a learning rate, η, from the current decision variable set [13].

- Momentum [75], utilizes previous gradient information to accelerate updates made by standard gradient descent.

- Nesterov's Accelerated Gradient [68] adjusts Momentum by using a look-ahead

procedure to evaluate an iteration's gradient as if the iteration had made an update in the direction of Momentum.

- AdaGrad [23] is similar to stochastic gradient descent but instead of making standard gradient updates, it adjusts the learning rate for each decision variable by an accumulation of past gradient information, making the learning rate adaptive.

- RMSProp, also an adaptive algorithm [89], adjusts AdaGrad by using exponentially decaying gradient information to emphasize recent gradient information.

- Adam [48], one of the most popular methods for training neural networks, combines the work of Momentum and RMSProp, exponentially decaying momentum as well as accumulated gradient information.

Although adaptive methods have shown strong success in quickly reducing training error, there have been critiques related to their ability to find solutions that generalize well. Wilson et al. [91] showed that modern, adaptive approaches may perform worse than methods that do not employ adaptive techniques [91]. Adaptive methods are argued to be prone to exhibit overfitting behavior, providing sub-par results relative to non-adaptive methods [91]. Keskar et al. [45] found that large batch sizes can lead to overfitting of the training data for Adam.

In contrast to the risk of achieving poor solution quality when using adaptive methods, the oscillatory behavior exhibited by non-adaptive methods has been shown to help avoid sub-optimal local minima [14]. Lee et al. [56] show that descent family methods can avoid saddle points. Hardt et al. [34] show that stochastic gradient descent can achieve solutions with strong generalization capabilities.

5.2.3 Early Learning Training Heuristics

Several studies focusing on characterizing the early learning behavior of CNNs can be found in Section 3.2.2. The often faster convergence speeds obtained through the use of adaptive methods and the potentially stronger generalization capabilities achieved by non-adaptive methods have prompted several researches to build heuristic methods that aim to effectively switch between adaptive and non-adaptive methods. These heuristics aim to achieve the faster training speeds of adaptive methods while maintaining the final solution quality of non-adaptive methods. Wu et al. [93] and Keskar and Socher [46] propose different methods of switching from Adam to stochastic gradient descent over the course of training; both methods have shown to yield competitive early and late-stage training and generalization results. Akiba et al. [4] show that gradually transitioning from RMSProp to stochastic gradient descent during training can yield better solution quality.

5.2.4 Optimization with Bundle-Level Methods

Bundle-level methods are popular convex optimization techniques; we briefly summarize the description of bundle-level methods provided by Oliveira and Sagastizabal [70].

Bundle-level methods utilize first-order oracle information to build first-order Taylor series approximations (Equation 6) at different points along the loss surface over the course of training. The linear approximations are used to build linearized models of the loss surface at each iteration, k. The collection of first-order Taylor series approximations which are built from previous oracle calls over the course of optimization are canonically called a bundle, denoted as the set J.

$$h_j(\theta) = f(\theta^{k-1}) + \langle \nabla f(\theta^{k-1}), \theta - \theta^{k-1} \rangle \tag{6}$$

These linear, approximate models, sometimes referred to as cutting plane models, can be expressed in the form found in Equation 7.

$$\hat{f}^k(\theta) = \max_{\theta \in \Theta^k, j \in J} (h_j(\theta)) \tag{7}$$

Updates made through bundling are informed successive evaluations of the cutting plane models in concert with a level set, defined using a level set parameter, τ. The level set parameter, τ, is used together with global constraints, $\theta \in \Theta$, at each iteration together with the cutting plane model to build a new feasible region to optimize within. The full feasible region considered at an iteration, k, can be described as in Equation 8 [70].

$$\Theta^k := \{\theta \in \Theta : \hat{f}^k(\theta) \leq (1 - \tau) f(\theta^{k-1})\} \tag{8}$$

In convex optimization, the linearized model found at each iteration is used with the feasible regions, Θ^k at each iteration to find a new lower bound of the loss surface, $f^k_{lower}(\theta)$ (Equation 9) [70].

$$f^k_{lower}(\theta) = \min_{\theta \in \Theta^k} \hat{f}_k(\theta) \tag{9}$$

An update can be made by projecting the previous iterate, θ^{k-1}, onto the new level set at iteration k. Through successive oracle calls the cutting plane model and level set are adjusted. Minimizing the cutting plane model within the feasible region, Θ^k creates an iterative means of finding lower bounds which are closer and closer to the optimal solution; projections onto these updated level sets at each successive iterate lead to better solutions.

Full-memory, bundle-level methods carry all first-order linear approximations at successive updates to inform the next update, building an increasingly more complex

approximation of the loss surface as additional oracle calls are made. Carrying large numbers of hyperplanes can become computationally taxing in large-scale instances [9], [22].

To address the high computational costs exhibited in large-scale instances optimized with bundle methods, Kiwiel [49] and Ben-Tal and Nemirovski [8] propose restricted memory-level bundling methods with nearly dimension independent properties. The Non-Euclidean Restricted Memory Level (NERML) bundling method introduced by Ben-Tal and Nemirovski [8] uses specific geometries to make fast, closed-form projections at each iteration. This capability allows the resulting algorithm to exhibit nearly dimension independent properties. Kiwiel [50] introduces bundle methods used for convex loss surfaces with inexact oracles. Oliveira and Sagastizabal [70] further build off of the exact-version of NERML to account for inexact oracle information in convex optimization while still allowing for the bundle-size control which NERML allows. Do and Artières [22] develop a restricted-memory level bundling technique that can be applied to both convex and non-convex regularized loss functions using a Lagrangian representation.

5.2.5 Contributions

To address the issue of slow early learning often exhibited by non-adaptive first-order learning methods utilized to train CNNs, we extend the use of a Lagrangian dual representation for a bundle update, inspired by updates made by Ben-Tal and Nemirovski [8], to accelerate learning within localized trailing Euclidean trust regions. Unlike the aforementioned bundle methods which use a set of update rules; our research focuses instead on extending updates made by non-adaptive first-order methods commonly used to train CNNs at each iteration of training through the use of bundling and fast projections. In addition to being an extension of a non-adaptive

update rather than a full routine, our proposed method can solve non-convex instances as in Do and Artières [22]; however, we do not require the objective function to be regularized. Our method is designed to work with noisy oracles and is competitive with adaptive methods over the early-learning stages we have studied, making its application to CNN tasks favorable. We demonstrate the efficacy of this method on standard non-convex benchmark functions, logistic regression, and CNN training. Through this study we demonstrate:

- We equip non-adaptive methods with Lagrangian Trust-Falls, a first-order extension method which can accelerate non-adaptive training to levels competitive with adaptive methods.
- We show that we can decompose the learning problem of CNNs into a series of large-scale optimization problems.

5.3 Methodology

5.3.1 Overview

Accelerating non-adaptive, first-order methods by using localized Euclidean trust regions and bundle-type updates begins by first making a first-order oracle call and taking a non-adaptive update. The oracle information used to inform the non-adaptive update as well as the position of the decision variable set before the update is taken are used in concert with the decision variable set after the update is taken to make a secondary update informed by solving an auxiliary problem of low dimension. There are six primary steps which comprise an iteration using this routine.

1. Call FO oracle to obtain $f(\theta^{k-1})$ and $\nabla f(\theta^{k-1})$.
2. Take a non-adaptive update using SGD-Vanilla, SGD-Momentum, or SGD-Nesterov to obtain an initial update, $\theta^{k'}$.

3. Impose a localized, trailing hypercube trust region, $\theta \in \Theta^{k'}$, centered at θ^{k-1}.

4. Build a FO Taylor series approximation of the local loss surface using iterate $\theta^{k'}$ and the gradient from the previous oracle call, $\nabla f(\theta^{k-1})$; append this hyperplane, $h_j(\theta)$, to the bundle, J, comprising the current cutting plane model, $\hat{f}^k(\theta)$.

5. Use a level-set parameter, τ, to control the size of a feasible region used in an auxiliary problem of low-dimension to find an optimal learning rate vector, λ_*^k.

6. Make a projected update onto the imposed hypercube geometry using λ_*^k.

The following sections outline the different components of the algorithm; Section 5.3.7 outlines the full optimization routine.

5.3.2 Oracle Call and Non-Adaptive Update

An iteration begins with a first-order oracle call using the decision variable from the previous iteration, θ^{k-1}. This information is used to immediately take a non-adaptive update in the full dimension of the decision space with step-size η to $\theta^{k'}$, an initial update during a single iteration. This update can be taken using vanilla Stochastic Gradient Descent (SGD-Vanilla), Stochastic Gradient Descent with Momentum (SGD-Momentum), or Stochastic Gradient Descent with Nesterov's Accelerated Gradient (SGD-Nesterov) as described in Section 5.2.2.

We adopt a generalized non-adaptive update expression similar to the expression found in Wilson et al. [91]. Equation 10 summarizes the updates of each of these methods using the binary hyperparameters β_1 and β_2. When $\beta_1 = \beta_2 = 0$, the update is equivalent to SGD-Vanilla; if $\beta_1 > 0$ and $\beta_2 = 0$, the update is equivalent to SGD-Momentum; if $\beta_1 = \beta_2 > 0$, the update is equivalent to SGD-Nesterov. If

SGD-Momentum or SGD-Nesterov is selected, then a momentum term, $M > 0$, must be selected.

$$\theta^{k'} \leftarrow \theta^{k-1} - \eta \nabla f(\theta^{k-1} + \beta_2 M(\theta^{k-1} - \theta^{k-2})) + \beta_1 M(\theta^{k-1} - \theta^{k-2}) \tag{10}$$

This update occurs at each iteration, regardless of the outcome of the auxiliary problem solved in a later portion of the iteration, ensuring that progress is always at least made in traditional non-adaptive directions throughout training. The previous iteration's decision variable values, θ^{k-1}, its associated gradient $\nabla f(\theta^{k-1})$, and its loss function value, $f(\theta^{k-1})$ are stored to be used in the remainder of the algorithm.

5.3.3 Localized Trailing, Hypercube Trust Region

Traditional bundle-level methods require global constraints imposed on the decision variables in order to ensure that minimizing the cutting plane models developed throughout optimization are not unbounded. Ben-Tal and Nemirovski [8] consider specific constraint geometries which have closed-form projection solutions when optimizing over the decision space; however, directly applying global constraints is generally not viable in non-convex optimization scenarios.

Instead of applying global constraint sets over the entire decision space, we adapt the hypercube geometry studied in Ben-Tal and Nemirovski [8] to localized regions surrounding the decision variables at an iteration, $k-1$. This strategy is illustrated in Equation 11.

$$\Theta_i^{k'} := \{\theta_i \in \mathbb{R} : \theta_i^{k-1} - \epsilon \leq \theta_i \leq \theta_i^{k-1} + \epsilon\} \tag{11}$$

The closed form projection solution for the hypercube geometry, the clip function, can be found in Equation 12.

$$\underset{\theta_i \in \Theta_i^{k'}}{\mathrm{P}}(\theta_i) := \min(\theta_i^{k-1} + \epsilon, \max(\theta_i, \theta_i^{k-1} - \epsilon)) \tag{12}$$

We call these localized constraint sets, trailing Euclidean trust regions, as they are centered at the previous iterate, θ^{k-1}, and are used to bound forward progress within the local region, reducing the likelihood of greedy, erroneous steps from occurring over the local, non-convex space.

5.3.4 Defining the Level Set

After making the non-adaptive update, finding $\theta^{k'}$, and centering a hypercube trust region, $\Theta^{k'}$, around the previous iteration, θ^{k-1}, a first-order Taylor series approximation of the local loss surface is found using $\theta^{k'}$ and the gradient used to inform the update, $\nabla f(\theta^{k-1})$. By choosing to build the first-order Taylor series approximation using $\theta^{k'}$ instead of θ^{k-1}, we are approximating the loss surface at $\theta^{k'}$ as if the gradient $\nabla f(\theta^{k-1})$ has been evaluated at $\theta^{k'}$. The trailing hypercube trust region, $\Theta^{k'}$ can be viewed as a tolerance of risk in using the previous iterate's gradient to produce an extended update.

The training problem associated with CNNs limits the amount of information related to both individual decision variables as well as the trust-worthiness of the local loss surface. We have found through empirical testing that in these instances half-widths which are a function of the mean layer-wise standard deviation of the weights at initialization, denoted $\bar{\sigma}^{init}$, show success.

At each iteration, the constraint set, $\Theta^{k'}$, is centered at the previous iteration, θ^{k-1}. Centering the constraint set at the previous iteration provides control over the update size of both the non-adaptive update as well as the secondary extension update, described in more detail in Section 5.3.6. This control acts as a means of preventing both the non-adaptive update as well as the secondary extension update from over-

stepping too far from the previous iteration's position. It also simultaneously provides localized bounding which allows for bundle-type optimization problems to be solved in this local region without risk of being unbounded.

5.3.5 Auxiliary Problem

At each iteration, we attempt to project the non-adaptive update at or below a level set controlled by the level set parameter, τ, while remaining within the local hypercube trust region, $\Theta^{k'}$. In practice we use the previous oracle call information to define the level $f^k_{level} = f(\theta^{k-1})$. This yields a bundle of hyperplanes offset by the current level set with each hyperplane, $h_j(\theta)$, taking the form found in Equation 13.

$$h_j(\theta) = f(\theta^{k-1}) + \langle \nabla f(\theta^{k-1}), \theta - \theta^{k-1} \rangle - (1-\tau) f^k_{level} \; \forall j \in J \tag{13}$$

The primal form of this problem's objective function can be found in Equation 14 and constraint set in Equation 15.

$$A(\theta) = \underset{\theta \in \Theta^{k'}}{\operatorname{argmin}} \frac{1}{2} ||\theta - \theta^{k'}||_2^2 \tag{14}$$

$$s.t. \;\; h_j(\theta) \leq 0 \; \forall j \in J \tag{15}$$

The Lagrangian dual form of the auxiliary problem, found in Equation 16, uses dual variables, λ_j, with each element of the vector corresponding to each hyperplane of the current bundle, $h_{j \in J}(\theta)$. The solution to the Lagrangian dual at iteration k yields a vector, λ^k_*. This method can carry multiple approximating hyperplanes, $|J| \geq 1$, to build local estimates of the loss surface over the course of training, providing full control of the problem dimension in Equation 16.

$$L(\lambda) = \underset{\lambda \geq 0}{\operatorname{argmax}} \frac{1}{2} || \underset{\theta \in \Theta^{k'}}{\mathrm{P}} (\theta^{k'} - \langle \lambda, \nabla h_{j \in J}(\theta) \rangle) - \theta^{k'} ||_2^2 + \langle \lambda, h_{j \in J}(\underset{\theta \in \Theta^{k'}}{\mathrm{P}} (\theta^{k'} - \langle \lambda, \nabla h_{j \in J}(\theta) \rangle)) \rangle \tag{16}$$

Ultimately, the extension is controlled by both the size of the trust region, Θ^k, and the level set parameter, τ. Adjusting these two hyperparameters provides control of the extension, mitigating divergent or otherwise erratic behavior while maintaining faster training.

5.3.6 Secondary Update

Each element of the dual's solution vector, λ^k, is used to directly scale the gradients of the first-order Taylor series approximations built from past oracle calls used to create the bundle. The scaled vectors are used to create a composite vector which is then applied as a secondary update shown in Equation 17.

$$\theta^k \leftarrow \underset{\theta \in \Theta^{k'}}{\mathrm{P}} (\theta^{k'} - \langle \lambda_*, \nabla h_{j \in J}(\theta) \rangle) \tag{17}$$

The optimal vector, λ_*, can be viewed as a series of secondary learning rates, with each learning rate adjusting decent with respect to its corresponding first-order Taylor series approximation. When the bundle size is one, the single element of the vector, λ_*, is applied to the same, single gradient vector used to inform the non-adaptive update used to obtain $\theta^{k'}$. The projection procedure, $\mathrm{P}_{\theta \in \Theta}$ is applied after the secondary gradient update takes place to ensure that the extension does not deviate too far from the local region.

5.3.7 Accelerating Stochastic Gradient Descent Family Routines with Lagrangian Trust-Falls (LTF)

The optimization routine in its entirety is illustrated in Algorithm 3. It consists of first making a non-adaptive update and then subsequently extending the update within a Euclidean trust region using oracle information which has been carried in the current bundle, J.

Algorithm 3 Lagrangian Trust-Fall algorithm.

Choose learning rate, η.
Choose $\beta_1 \in \{0, 1\}$ and $\beta_2 \in \{0, 1\}$, where $\beta_2 \leq \beta_1$.
Choose $M \in (0, 1)$ if $\beta_1 > 0$ (and if $\beta_2 > 0$).
Choose half-width parameter, ϵ, to define the half-width of the trailing hypercube trust region $\Theta^{k'}$.
Choose bundle memory parameter, Γ.
Choose level parameter, τ.
Let θ^0 be the decision variable set found at initialization.
for $k \in K$ **do**

Call first-order oracle to obtain $f(\theta^{k-1})$ and $\nabla f(\theta^{k-1})$.
Let $\theta^{k'} \leftarrow \theta^{k-1} - \eta \nabla f(\theta^{k-1} + \beta_2 M(\theta^{k-1} - \theta^{k-2})) + \beta_1 M(\theta^{k-1} - \theta^{k-2})$.
Use θ^{k-1} to build trailing hypercube trust region, $\Theta^{k'}$.
Let $h_j(\theta) = f(\theta^{k-1}) + \langle \nabla f(\theta^{k-1}), \theta - \theta^{k'} \rangle - (1 - \tau) f^k_{level}$.
Append $h_j(\theta)$ to current bundle, J.

if $L(\lambda)$ *yields a solution,* λ^k_*. **then**

Update $\theta^k \leftarrow \mathrm{P}_{\theta \in \Theta^{k'}}(\theta^{k'} - \langle \lambda^k_*, \nabla h_{j \in J}(\theta) \rangle)$.

end

if $|J| \geq \Gamma$ **then**

Remove hyperplane $h_{min(j)}$ from bundle.

end

end

5.4 Results

Experiments are conducted on different problem types to emphasize the impact using non-adaptive stochastic gradient methods with LTF can have on the acceleration of early loss reduction. We choose to compare this method to popular benchmark optimization routines commonly used to train CNNs. The first set of experiments graphically explores the optimization routines' trajectories in low-dimensional space while traversing standard non-convex, deterministic benchmark functions. We then study logistic regression on the MNIST data set using 19 replicates. Finally, we conduct experiments using a CNN trained on both the grey-scale FashionMNIST data set as well as the three-channel color CIFAR-10 data set using 19 replicates, each.[1] Learning rates are found through grid searches; the learning rate for each routine yielding the best results found is reported. To build an intuition of the parameter settings for the hypercube half-width, ϵ, and level set parameter, τ, used in LTF updates, grid searches are carried out to identify settings for these two hyperparameters as well.

5.4.1 Non-Convex Benchmark Function Experiments

Two, popular non-convex benchmark functions are studied to compare the performance of our method with peer optimization routines. The first function that is studied is the two-dimensional Himmelblau function, found in Definition 5.

Definition 5 (Himmelblau function)

$$f(x, y) = (x^2 + y - 11)^2 + (x + y^2 - 7)^2$$

[1] All stochastic experimental runs are conducted using an HP Z8 G4 Workstation with Intel(R) Xeon(R) Silver 4210 CPU @ 2.20GHz, 128 GB RAM, and PyTorch [73].

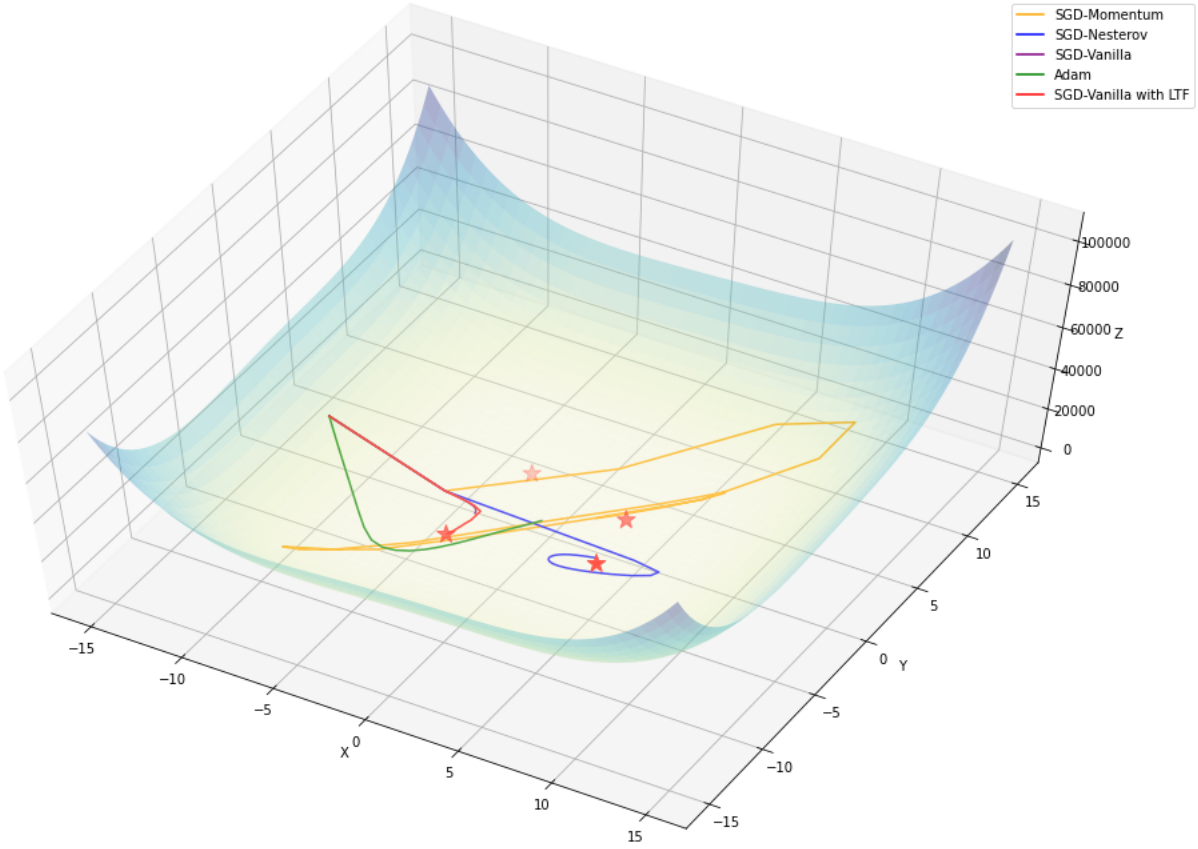
Optimization routine settings used to study trajectories along the Himmelblau function are found in Table 23. For these instances, LTF is equipped to the best-performing SGD-Vanilla setup.

Table 23. Optimization routine settings used for routines to traverse the Himmelblau function.

Optimization Routine	Learning Rate	Additional Hyperparameters
SGD-Nesterov	0.001	Momentum = 0.9
SGD-Momentum	0.001	Momentum = 0.9
SGD-Vanilla	0.001	Not Applicable
SGD-Vanilla with LTF	0.001	$\tau = 0.4$, $\epsilon = 3$, $\Gamma = 1$
Adam	0.100	$\beta^1 = 0.9$, $\beta^2 = 0.999$

The trajectories of each routine along the Himmelblau function for 10 iterations can be found in 14. We see that SGD-Vanilla equipped with LTF is the only routine to find convergence with SGD-Nesterov close behind.

Figure 14. Optimization trajectories of 25 iterations on the Himmelblau function.



The second function that is studied is the two-dimensional Rosenbrock function, found Definition 6.

Definition 6 (Rosenbrock function)

$$f(x, y) = (1 - x)^2 + 100(y - x^2)^2$$

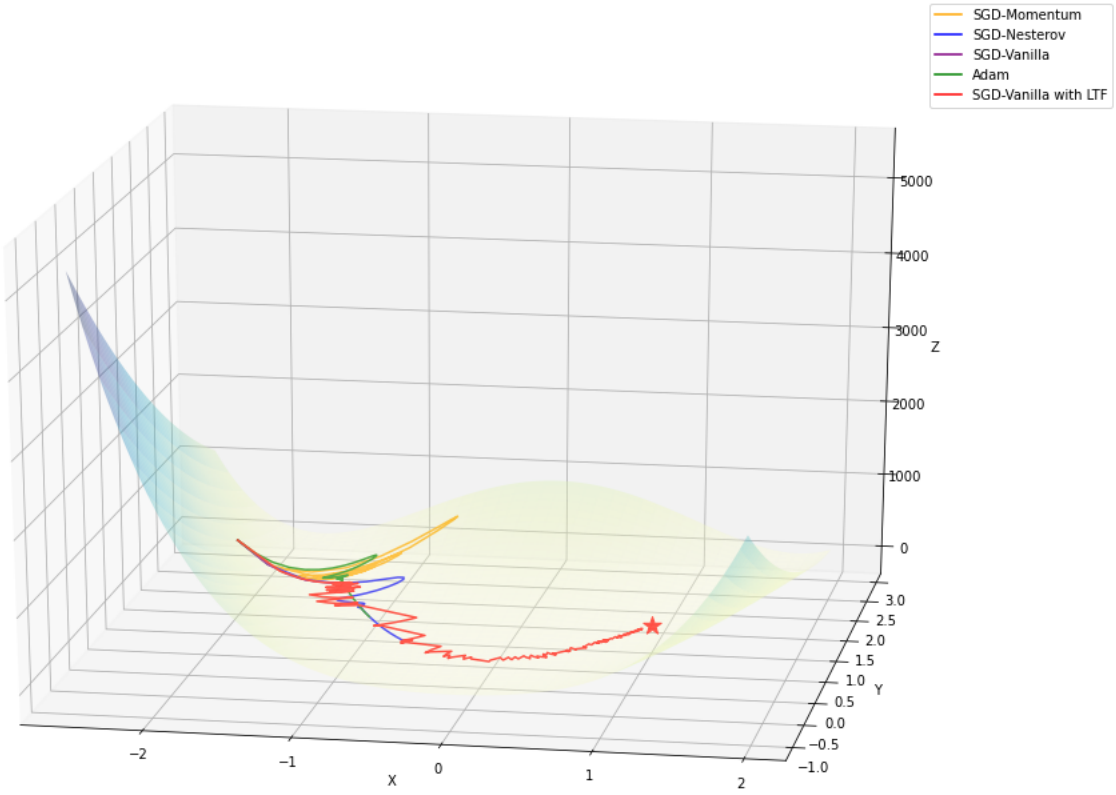
Optimization routine settings used to study trajectories along the Rosenbrock function are found in Table 24. For these instances, LTF is equipped to the best-performing SGD-Vanilla setup.

Table 24. Optimization routine settings used for routines to traverse the Rosenbrock function.

Optimization Routine	Learning Rate	Additional Hyperparameters
SGD-Nesterov	0.0001	Momentum = 0.9
SGD-Momentum	0.0001	Momentum = 0.9
SGD-Vanilla	0.0001	Not Applicable
SGD-Vanilla with LTF	0.0001	$\tau = 0.4$, $\epsilon = 3$, $\Gamma = 1$
Adam	0.1000	$\beta^1 = 0.9$, $\beta^2 = 0.999$

The trajectories of each routine along the Rosenbrock function for 500 iterations can be found in Figure 15. We note that SGD-Vanilla with LTF is significantly faster at reaching near-convergence when compared with all peer routines. SGD-Nesterov is the closest behind SGD-Vanilla equipped with LTF; however, it is markedly slower. We also point out the trajectories of SGD-Vanilla with LTF exhibit a jagged, oscillatory behavior; this behavior illustrates that the trust regions restrict updates from continuing too far, allowing for correction between iterations.

Figure 15. Optimization trajectories of 500 iterations on the Rosenbrock function.



5.4.2 Logistic Regression

Logistic regression instances are trained for two epochs on the MNIST data set. Table 25 shows the hyperparameter settings for each optimization routine. For these instances, LTF is equipped to the best-performing SGD-Nesterov setup.

Table 25. Optimization routine settings used for routines to train MNIST logistic regression.

Optimization Routine	Learning Rate	Additional Hyperparameters
SGD-Nesterov	0.010	Momentum $= 0.9$
SGD-Momentum	0.010	Momentum $= 0.9$
SGD-Nesterov with LTF	0.010	$\tau = 0.2$, $\epsilon = 0.2$, $\Gamma = 1$
AdaGrad	0.010	$\epsilon = 1e - 10$
Adam	0.001	$\beta_1 = 0.9$, $\beta_2 = 0.999$

Early training and test results can be found in Figures 16 and 17.

Figure 16. Early iterations of logistic regression training results on the MNIST data set.

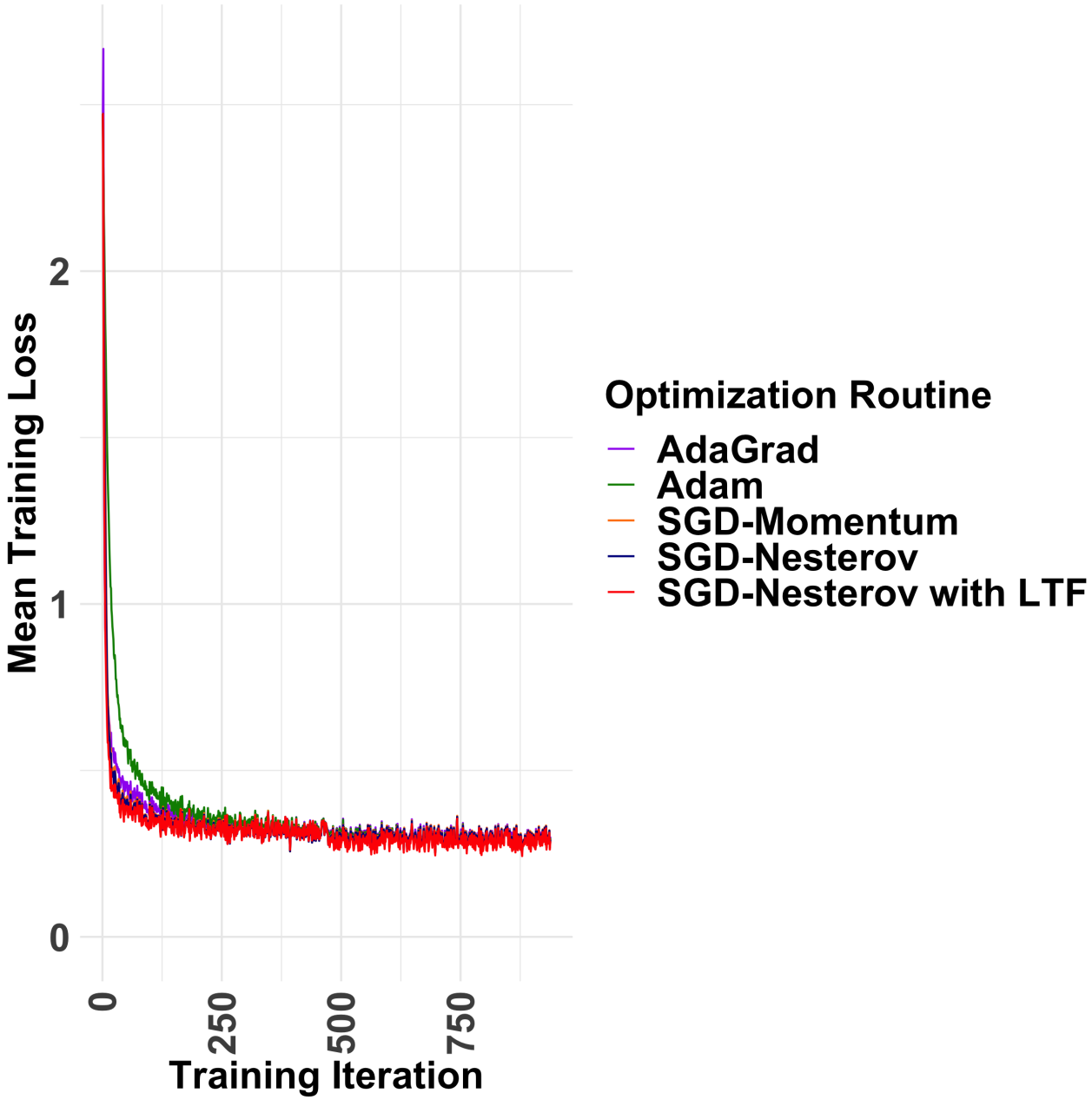
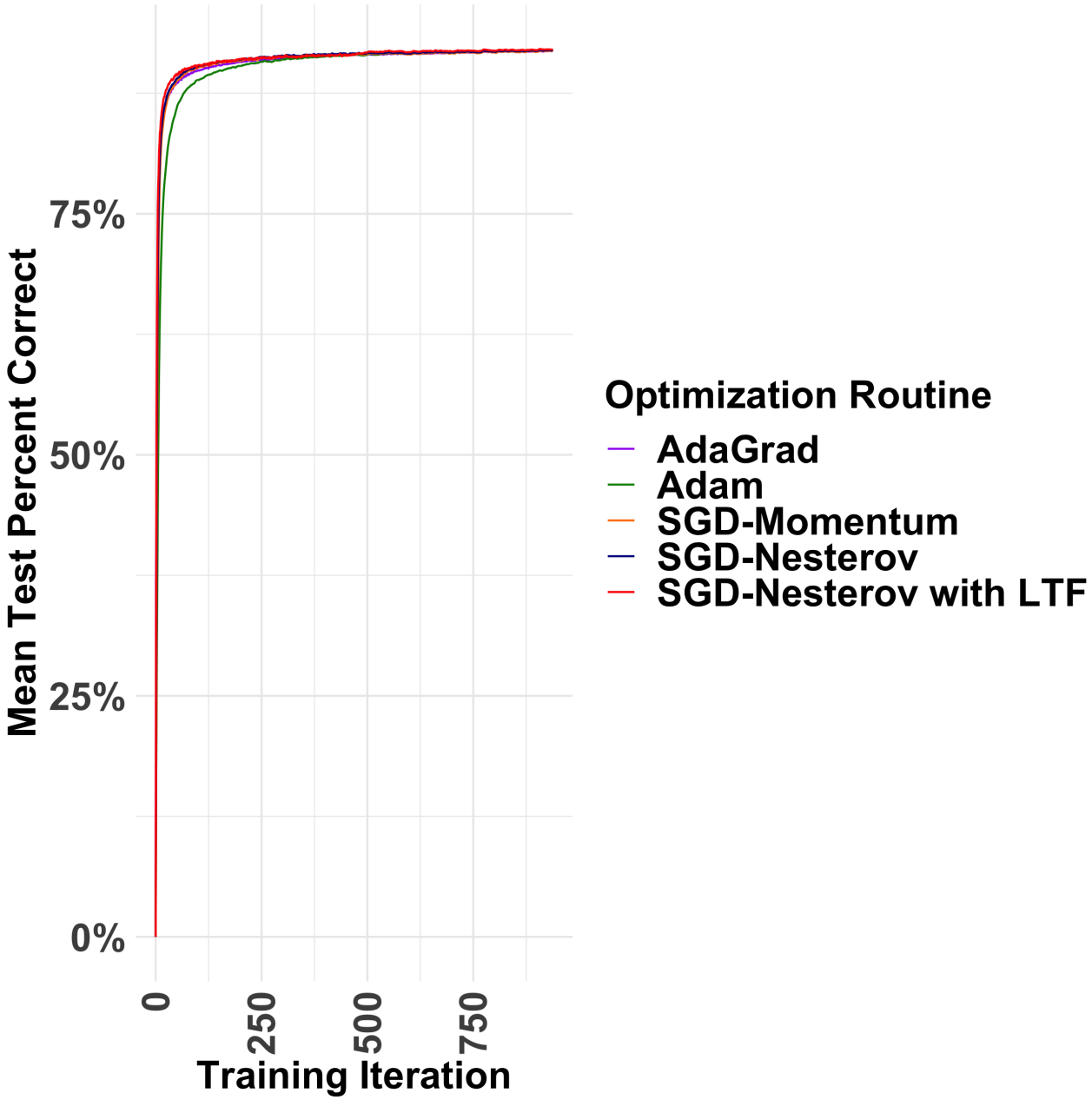


Figure 17. Early iterations of logistic regression test results on the MNIST data set.



The early acceleration exhibited by equipping SGD-Nesterov with LTF allows for training and test loss to be decreased more quickly when compared with any of the other peer routines considered across these instances.

5.4.3 Convolutional Neural Network Experiments

Experiments using a CNN with two convolutional layers, each followed by max-pooling layers, and three fully-connected layers are utilized to study the efficacy of this routine on computer vision learning tasks. The FashionMNIST data set [94], a 28 by 28, one-channel image benchmark data set and the CIFAR-10 data set [51], a popular 32 by 32, three-color channel image benchmark data set are used for these experiments. The CIFAR-10 data set has 50,000 training examples and 10,000 test examples used to measure the quality of the solution on unseen data. The FashionMNIST data set has 60,000 training examples and 10,000 test examples used to measure the quality of the solution on unseen data.

Across these instances, LTF is equipped to the best-performing SGD-Nesterov setup. Table 26 shows the resulting hyperparameter configurations for each optimization routine. Ten epochs are considered to heuristically examine early learning as described by Frankle et al. [27] in their study of training ResNet architectures on CIFAR-10.

Table 26. Optimization routine settings used for routines to train CNNs on FashionMNIST and CIFAR-10.

Optimization Routine	Learning Rate	Additional Hyperparameters
SGD-Nesterov	0.010	Momentum = 0.9
AdaGrad	0.010	$\epsilon = 1e-10$
SGD-Nesterov with LTF	0.010	$\tau = 0.05$, $\epsilon = \frac{\sigma^{init}}{16}$, $\Gamma = 1$
Adam	0.001	$\beta^1 = 0.9$, $\beta^2 = 0.999$

The FashionMNIST results for both, training and test results, are found in Figures 18 and 19, below.

Figure 18. CNN early training results on FashionMNIST data set.

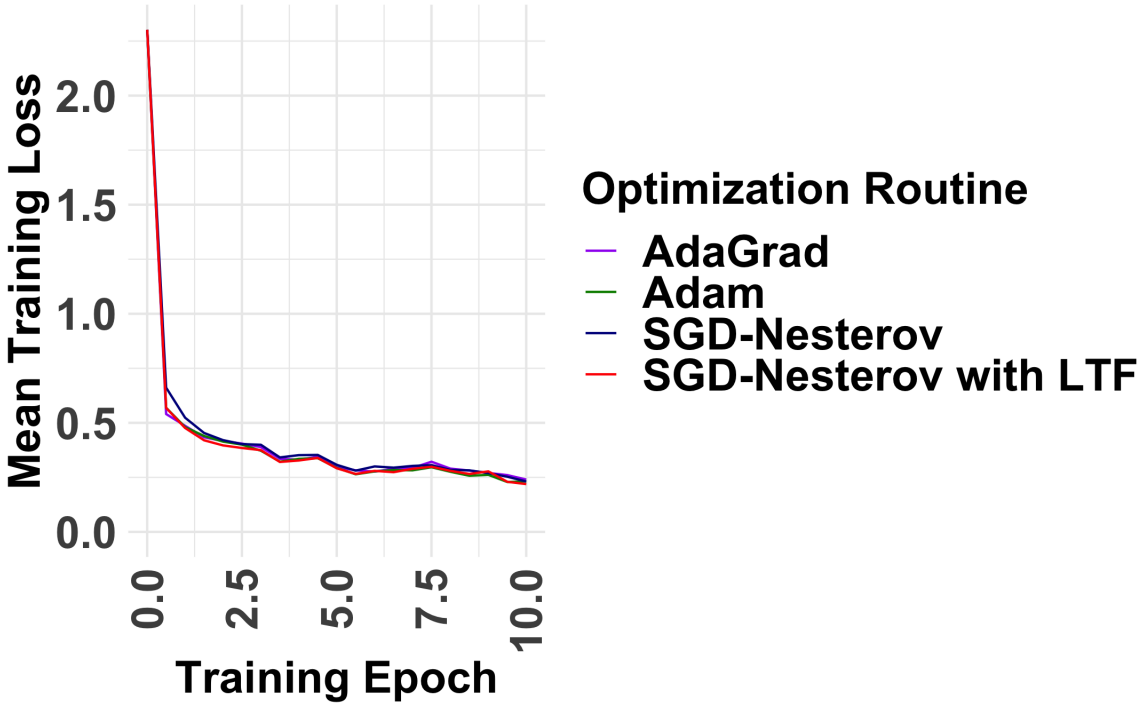
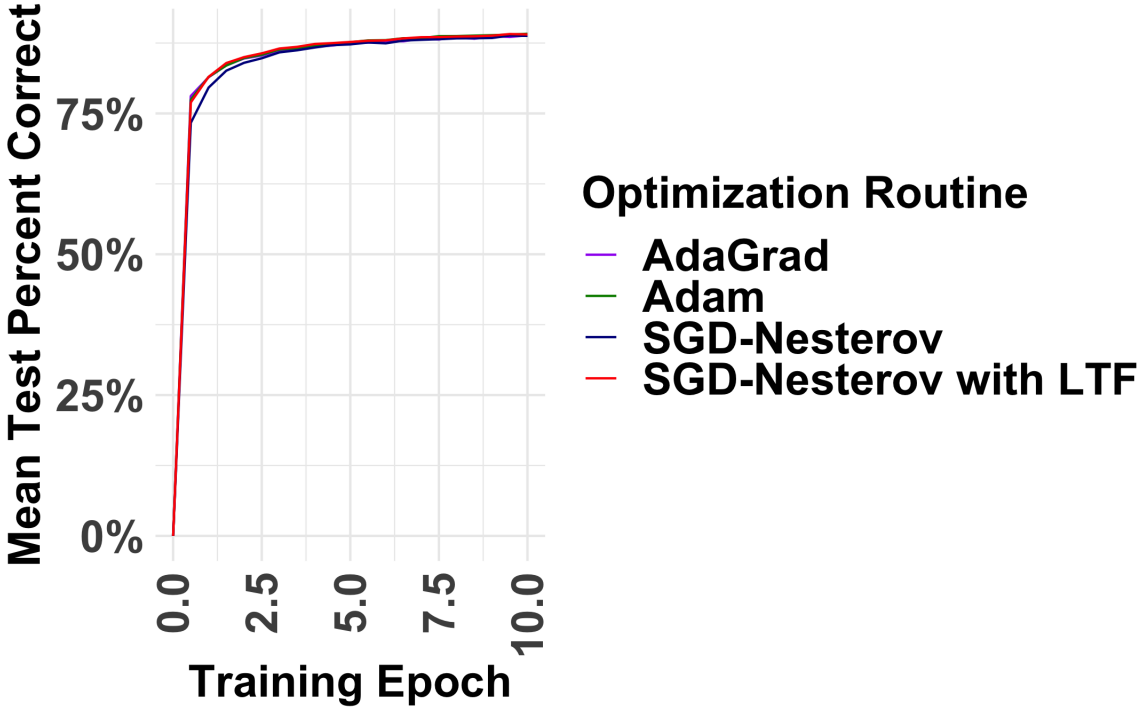


Figure 19. CNN early test results on FashionMNIST data set.



The CIFAR-10 results for both, training and test results, are found in Figures 20 and 21.

Figure 20. CNN early training results on the CIFAR-10 data set.

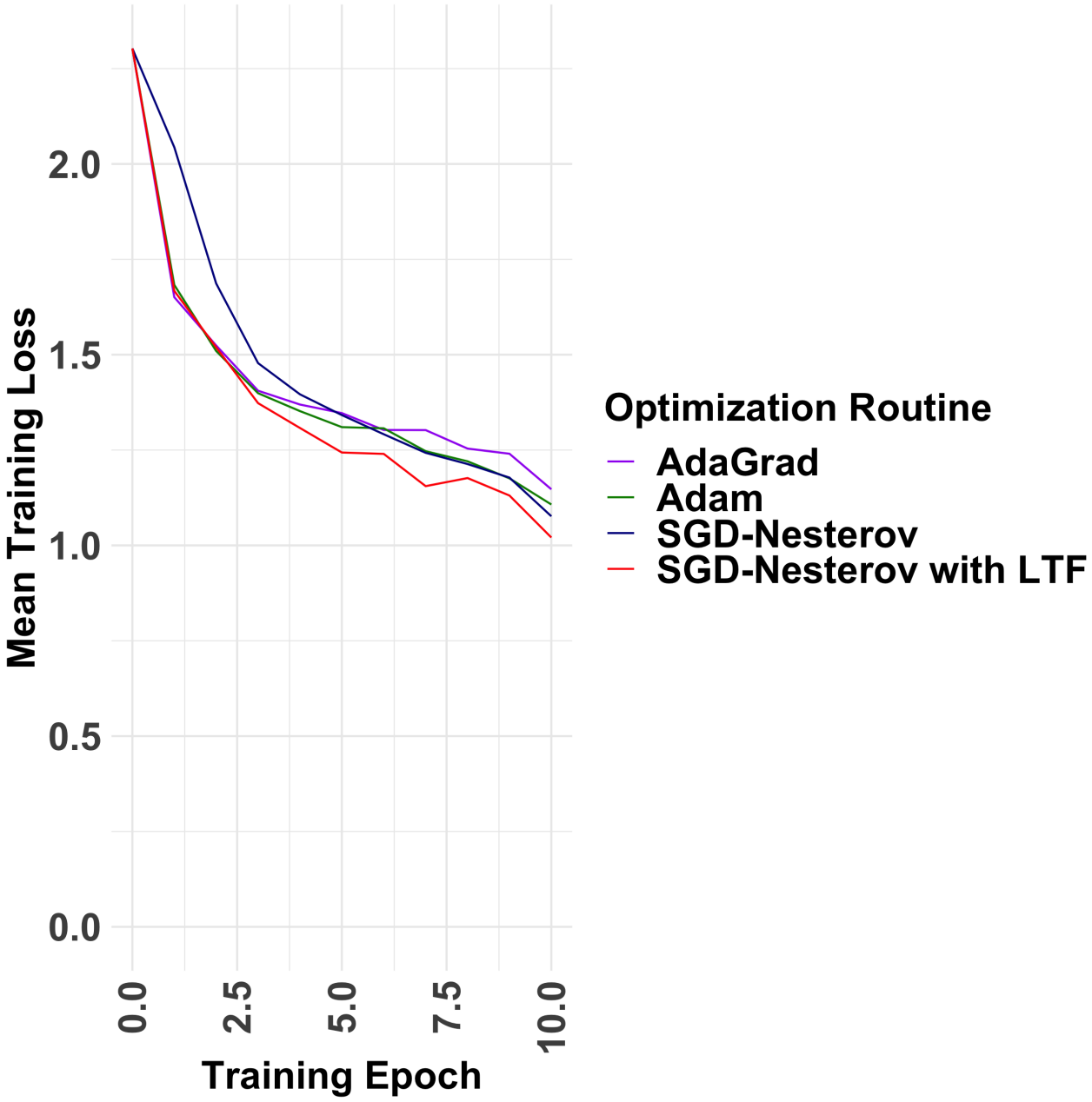
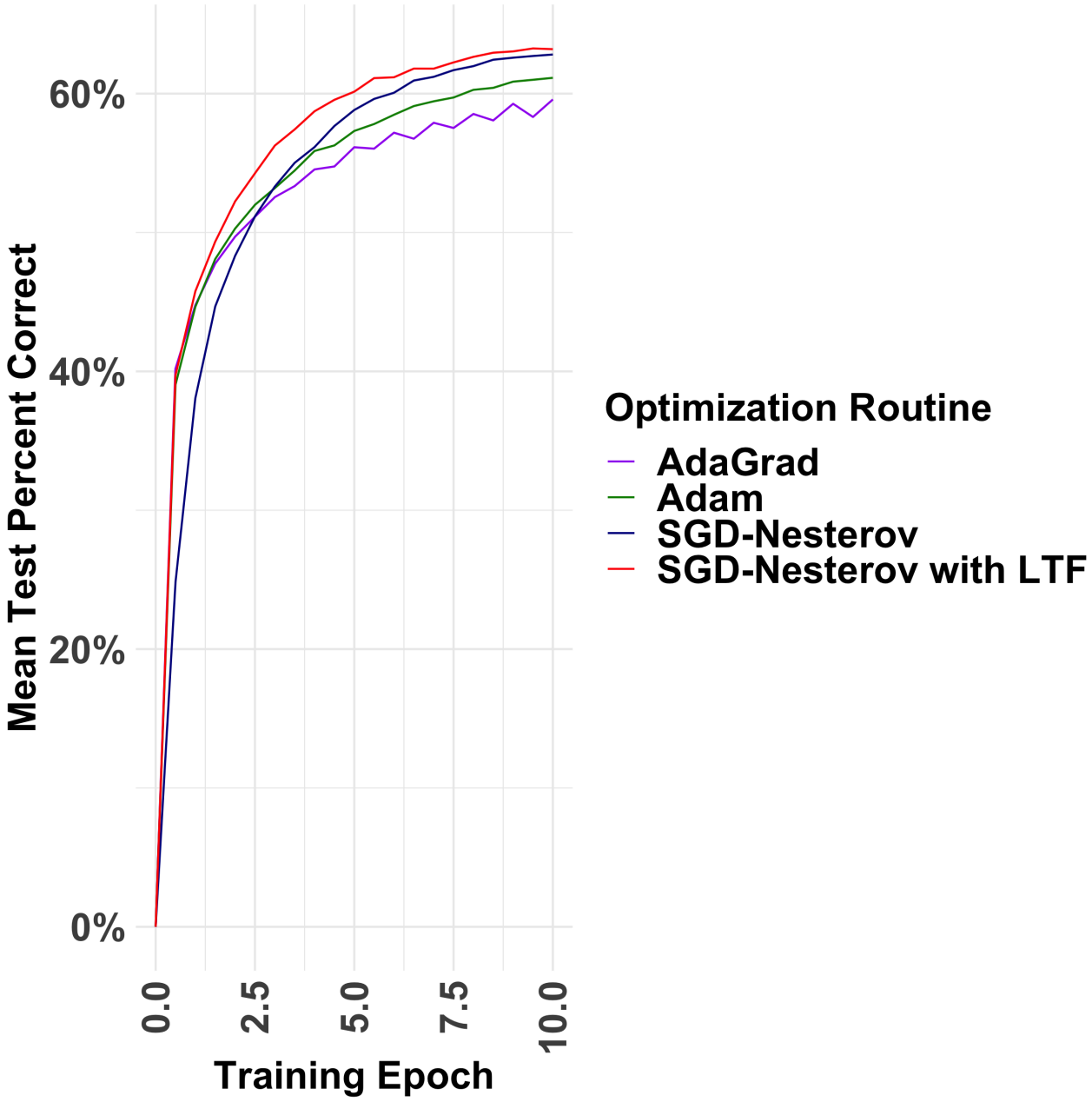


Figure 21. CNN early test results on the CIFAR-10 data set.



Results across both data sets show that SGD-Nesterov equipped with LTF performs competitively with adaptive methods. Additionally, SGD-Nesterov is accelerated in these instances with the most notable improvement exhibited across the

harder CIFAR-10 instances.

5.5 Discussion

5.5.1 Overall Observations

We find that applying LTF to extend non-adaptive updates can consistently accelerate early learning of non-adaptive stochastic gradient methods on both convex and non-convex problems. This method also yields competitive performance when compared with state-of-the-art adaptive methods such as Adam and AdaGrad in reducing early training loss on CNN training instances. The feasible regions built through trailing hypercubes and first-order Taylor series approximations restricted by level sets are used in concert with the auxiliary dual problem in Equation 16 to find optimal secondary learning rate vectors. These secondary learning rates provide adjustments to be made to the original update along the same gradient direction.

The method reduces the need to schedule learning rates during early learning by finding optimal dual variables, λ_*^k. These dual variables act as secondary learning rates. Although two new hyperparameters are introduced, they do not require the same level of precision that a tuned learning rate requires. By solving a localized Lagrangian dual problem at each iteration, this method also demonstrates we can successfully decompose the training problem associated with CNN instances into a series of optimization problems, making updates within local regions of the loss surface more similar to those made in traditional convex optimization applications.

5.5.2 Next Steps

There are a variety of future research directions we believe could be promising to pursue. We have found that using hypercube half-widths that are a function of the mean standard deviation across network layers found during initialization to be

an effective way of building the trailing trust regions for CNN training instances. We believe that additional work needs to focus on heuristically defining these half-widths to allow for more effective updates to be made by LTF. Annealing the trailing hypercubes over the course of learning may allow for more dramatic updates early when gradients are high [27] and larger gains can be made while ensuring the cube is not too large in later stages of training.

All of the results conveyed in this study utilize a single first-order Taylor series approximation to accelerate the decision variables within a trailing hypercube trust region. Carrying more than a single first-order Taylor series approximation from oracle calls made in successive iterations could help to approximate the local loss surface more effectively, making for more informed updates. This could especially be useful in instances where the loss surface exhibits increased levels of smoothness as shown to be the case when BatchNorm is utilized [83].

Finally, we only study trailing hypercube geometries in this chapter. Alternative trailing geometries may also prove to be an effective means of controlling the localized region being optimized.

5.6 Conclusion

This chapter demonstrates the efficacy of applying updates similar to those made in large-scale convex optimization to the problem of training instances with non-convex underlying loss surfaces. The results show that this method is competitive with standard state-of-the-art training procedures when training convolutional neural networks while also proving to be effective relative to peer routines at traversing standard benchmark functions as well as logistic regression.

VI. Summary and Conclusions

The research presented in this dissertation can improve U.S. Air Force Auto Machine Learning (AutoML) capabilities for deep learning applications [38]. Methods of identifying architecture hyperparameters which can affect early learning are developed and show that architecture hyperparameters can be used to inform optimization routine selection. We also demonstrate that faster training convergence rates can be found by equipping non-adaptive optimization routines with bundle-type updates.

Chapter III systematically explores the loss surfaces of CNNs comprised of different hyperparameters using an new approach for loss surface exploration through Design of Experiments. Two types of stochastic walks are used to generate walks across the local loss surface, providing local estimates of second-order Lipschitz constants as well as local region size. The findings from this study show strong indications that both the first and second-order decisions associated with choosing many popular CNN architecture hyperparameters can significantly impact these loss surface characteristics, locally.

Chapter IV studies six different optimizer routines commonly used to train CNNs and draws relationships between optimization routine performance in the early-learning stages with architecture hyperparameter decisions. The results from this study empirically show that solutions which achieve low training error do not always generalize well. Additionally, we find each optimization routine is uniquely affected by several different architecture hyperparameters and their combinations both in terms of training and finding solutions that can generalize well. Finally, we demonstrate that the choice of optimization routine used to train a CNN instance should take into account the architecture hyperparameters of the instance.

Chapter V introduces a method of incorporating bundle-type updates into non-adaptive optimization routines, allowing for acceleration of these routines in the early

learning stages to levels competitive with state-of-the-art adaptive optimization routines currently employed to train CNNs. The resulting method also demonstrates that updates can be made within localized regions of the loss surfaces found in CNN instances using methods more similar to large-scale convex optimization.

Incorporating our findings into systems which possess limited computational capabilities and have limited resources to perform classification tasks can reduce the lead time for both, model selection and model training, and improve the overall performance of the system and ultimately improve mission effectiveness.

VII. Appendix

7.1 Experiment Setup

Experiment Setup No.	Convolutional Width	Convolutional Depth Number of Layers	Fully-Connected Width	Fully-Connected Depth Number of Layers	Dropout	BatchNorm	Max Pooling	Activation	Initialization
1	Narrow	Ten	Narrow	One	No Dropout	No BatchNorm	No Max Pooling	Leaky ReLu	He
2	Narrow	One	Narrow	One	No Dropout	No BatchNorm	Max Pooling	ReLu	Xavier
3	Narrow	One	Narrow	One	No Dropout	BatchNorm	No Max Pooling	ReLu	He
4	Narrow	Ten	Narrow	One	No Dropout	BatchNorm	Max Pooling	Leaky ReLu	Xavier
5	Narrow	One	Narrow	One	Dropout	No BatchNorm	No Max Pooling	ReLu	He
6	Narrow	Ten	Narrow	One	Dropout	No BatchNorm	Max Pooling	Leaky ReLu	Xavier
7	Narrow	Ten	Narrow	One	Dropout	BatchNorm	No Max Pooling	Leaky ReLu	He
8	Narrow	One	Narrow	One	Dropout	BatchNorm	Max Pooling	ReLu	Xavier
9	Narrow	Ten	Wide	One	No Dropout	No BatchNorm	No Max Pooling	ReLu	Xavier
10	Narrow	One	Wide	One	No Dropout	No BatchNorm	Max Pooling	Leaky ReLu	He
11	Narrow	One	Wide	One	No Dropout	BatchNorm	No Max Pooling	Leaky ReLu	Xavier
12	Narrow	Ten	Wide	One	No Dropout	BatchNorm	Max Pooling	ReLu	He
13	Narrow	One	Wide	One	Dropout	No BatchNorm	No Max Pooling	Leaky ReLu	Xavier
14	Narrow	Ten	Wide	One	Dropout	No BatchNorm	Max Pooling	ReLu	He
15	Narrow	Ten	Wide	One	Dropout	BatchNorm	No Max Pooling	ReLu	Xavier
16	Narrow	One	Wide	One	Dropout	BatchNorm	Max Pooling	Leaky ReLu	He
17	Narrow	Ten	Narrow	Ten	No Dropout	No BatchNorm	No Max Pooling	ReLu	He
18	Narrow	One	Narrow	Ten	No Dropout	No BatchNorm	Max Pooling	Leaky ReLu	Xavier
19	Narrow	One	Narrow	Ten	No Dropout	BatchNorm	No Max Pooling	Leaky ReLu	He
20	Narrow	Ten	Narrow	Ten	No Dropout	BatchNorm	Max Pooling	ReLu	Xavier

21	Narrow	One	Narrow	Ten	Dropout	No BatchNorm	No Max Pooling	Leaky ReLu	He
22	Narrow	Ten	Narrow	Ten	Dropout	No BatchNorm	Max Pooling	ReLu	Xavier
23	Narrow	Ten	Narrow	Ten	Dropout	BatchNorm	No Max Pooling	ReLu	He
24	Narrow	One	Narrow	Ten	Dropout	BatchNorm	Max Pooling	Leaky ReLu	Xavier
25	Narrow	Ten	Wide	Ten	No Dropout	No BatchNorm	No Max Pooling	Leaky ReLu	Xavier
26	Narrow	One	Wide	Ten	No Dropout	No BatchNorm	Max Pooling	ReLu	He
27	Narrow	One	Wide	Ten	No Dropout	BatchNorm	No Max Pooling	ReLu	Xavier
28	Narrow	Ten	Wide	Ten	No Dropout	BatchNorm	Max Pooling	Leaky ReLu	He
29	Narrow	One	Wide	Ten	Dropout	No BatchNorm	No Max Pooling	ReLu	Xavier
30	Narrow	Ten	Wide	Ten	Dropout	No BatchNorm	Max Pooling	Leaky ReLu	He
31	Narrow	Ten	Wide	Ten	Dropout	BatchNorm	No Max Pooling	Leaky ReLu	Xavier
32	Narrow	One	Wide	Ten	Dropout	BatchNorm	Max Pooling	ReLu	He
33	Wide	One	Narrow	One	No Dropout	No BatchNorm	No Max Pooling	Leaky ReLu	Xavier
34	Wide	Ten	Narrow	One	No Dropout	No BatchNorm	Max Pooling	ReLu	He
35	Wide	Ten	Narrow	One	No Dropout	BatchNorm	No Max Pooling	ReLu	Xavier
36	Wide	One	Narrow	One	No Dropout	BatchNorm	Max Pooling	Leaky ReLu	He
37	Wide	Ten	Narrow	One	Dropout	No BatchNorm	No Max Pooling	ReLu	Xavier
38	Wide	One	Narrow	One	Dropout	No BatchNorm	Max Pooling	Leaky ReLu	He
39	Wide	One	Narrow	One	Dropout	BatchNorm	No Max Pooling	Leaky ReLu	Xavier
40	Wide	Ten	Narrow	One	Dropout	BatchNorm	Max Pooling	ReLu	He
41	Wide	One	Wide	One	No Dropout	No BatchNorm	No Max Pooling	ReLu	He
42	Wide	Ten	Wide	One	No Dropout	No BatchNorm	Max Pooling	Leaky ReLu	Xavier
43	Wide	Ten	Wide	One	No Dropout	BatchNorm	No Max Pooling	Leaky ReLu	He
44	Wide	One	Wide	One	No Dropout	BatchNorm	Max Pooling	ReLu	Xavier
45	Wide	Ten	Wide	One	Dropout	No BatchNorm	No Max Pooling	Leaky ReLu	He
46	Wide	One	Wide	One	Dropout	No BatchNorm	Max Pooling	ReLu	Xavier
47	Wide	One	Wide	One	Dropout	BatchNorm	No Max Pooling	ReLu	He
48	Wide	Ten	Wide	One	Dropout	BatchNorm	Max Pooling	Leaky ReLu	Xavier
49	Wide	One	Narrow	Ten	No Dropout	No BatchNorm	No Max Pooling	ReLu	Xavier
50	Wide	Ten	Narrow	Ten	No Dropout	No BatchNorm	Max Pooling	Leaky ReLu	He
51	Wide	Ten	Narrow	Ten	No Dropout	BatchNorm	No Max Pooling	Leaky ReLu	Xavier
52	Wide	One	Narrow	Ten	No Dropout	BatchNorm	Max Pooling	ReLu	He
53	Wide	Ten	Narrow	Ten	Dropout	No BatchNorm	No Max Pooling	Leaky ReLu	Xavier
54	Wide	One	Narrow	Ten	Dropout	No BatchNorm	Max Pooling	ReLu	He
55	Wide	One	Narrow	Ten	Dropout	BatchNorm	No Max Pooling	ReLu	Xavier
56	Wide	Ten	Narrow	Ten	Dropout	BatchNorm	Max Pooling	Leaky ReLu	He
57	Wide	One	Wide	Ten	No Dropout	No BatchNorm	No Max Pooling	Leaky ReLu	He
58	Wide	Ten	Wide	Ten	No Dropout	No BatchNorm	Max Pooling	ReLu	Xavier
59	Wide	Ten	Wide	Ten	No Dropout	BatchNorm	No Max Pooling	ReLu	He
60	Wide	One	Wide	Ten	No Dropout	BatchNorm	Max Pooling	Leaky ReLu	Xavier

61	Wide	Ten	Wide	Ten	Dropout	No BatchNorm	No Max Pooling	ReLu	He
62	Wide	One	Wide	Ten	Dropout	No BatchNorm	Max Pooling	Leaky ReLu	Xavier
63	Wide	One	Wide	Ten	Dropout	BatchNorm	No Max Pooling	Leaky ReLu	He
64	Wide	Ten	Wide	Ten	Dropout	BatchNorm	Max Pooling	ReLu	Xavier

We note that layer width settings related to "Narrow" use a baseline width vector for each layer based on the layer type (Fully-Connected or Convolutional) as well as the number of layers used (ten or one). "Wide" layer width settings multiply the baseline width by a factor of two.

7.2 Second-Order Linear Regression Models of Lipschitz Upper Bound Constants and Number of Steps Estimates

Table 28. Gaussian path second-order linear regression model of Lipschitz Upper Bound Constants.

Factor	Estimate	Std. Error	T-Value	P-Value
Intercept	3.2304813	0.0081263	397.5336005	0.0000000
Convolutional Width	-0.0047632	0.0077535	-0.6143195	0.5390316
Convolutional Depth	-0.4038366	0.0078256	-51.6047308	0.0000000
Fully-Connected Width	0.1745591	0.0078707	22.1782103	0.0000000
Fully-Connected Depth	-0.5537944	0.0079515	-69.6469172	0.0000000
BatchNorm	0.2185858	0.0081170	26.9294677	0.0000000
Max Pooling	0.0409303	0.0076164	5.3739819	0.0000001
Dropout	-0.3703548	0.0080568	-45.9680647	0.0000000
Activation	0.1826839	0.0076548	23.8653831	0.0000000
Initialization	-0.1047469	0.0081017	-12.9290129	0.0000000
Convolutional Width * Convolutional Depth	-0.0290980	0.0075979	-3.8297595	0.0001298
Convolutional Width * Fully-Connected Width	-0.0412976	0.0074920	-5.5122673	0.0000000
Convolutional Width * Fully-Connected Depth	-0.0242907	0.0076611	-3.1706696	0.0015299
Convolutional Width * BatchNorm	0.0323019	0.0077414	4.1726099	0.0000306
Convolutional Width * Max Pooling	-0.0830246	0.0077848	-10.6648925	0.0000000
Convolutional Width * Dropout	-0.0476158	0.0077460	-6.1471365	0.0000000
Convolutional Width * Activation	0.0781380	0.0078330	9.9755198	0.0000000
Convolutional Width * Initialization	0.0273004	0.0076343	3.5760384	0.0003521
Convolutional Depth * Fully-Connected Width	0.1093770	0.0080806	13.5358401	0.0000000
Convolutional Depth * Fully-Connected Depth	0.0471513	0.0076422	6.1699000	0.0000000
Convolutional Depth * BatchNorm	-0.3500525	0.0078107	-44.8169318	0.0000000
Convolutional Depth * Max Pooling	0.0397165	0.0077581	5.1193487	0.0000003
Convolutional Depth * Dropout	-0.0264913	0.0077706	-3.4091727	0.0006566
Convolutional Depth * Activation	-0.1260619	0.0074532	-16.9138383	0.0000000
Convolutional Depth * Initialization	-0.0990636	0.0078782	-12.5744546	0.0000000
Fully-Connected Width * Fully-Connected Depth	0.0913472	0.0076099	12.0037936	0.0000000
Fully-Connected Width * BatchNorm	-0.0402175	0.0078550	-5.1199556	0.0000003
Fully-Connected Width * Dropout	0.1765406	0.0078632	22.4514285	0.0000000
Fully-Connected Width * Activation	0.0135621	0.0074970	1.8089933	0.0705108
Fully-Connected Depth * BatchNorm	0.0186668	0.0079423	2.3502892	0.0187967
Fully-Connected Depth * Max Pooling	0.0741039	0.0075118	9.8649551	0.0000000
Fully-Connected Depth * Dropout	-0.3659697	0.0078354	-46.7070428	0.0000000
Fully-Connected Depth * Activation	-0.0027150	0.0076703	-0.3539652	0.7233795
Fully-Connected Depth * Initialization	-0.0320263	0.0078601	-4.0745186	0.0000468
BatchNorm * Max Pooling	-0.0851258	0.0076064	-11.1913004	0.0000000
BatchNorm * Dropout	0.0530104	0.0080629	6.5746274	0.0000000
BatchNorm * Activation	0.1086868	0.0076444	14.2179073	0.0000000
BatchNorm * Initialization	0.1873481	0.0080964	23.1397377	0.0000000
Max Pooling * Dropout	0.0382654	0.0075873	5.0433566	0.0000005
Max Pooling * Activation	-0.0173573	0.0076239	-2.2766845	0.0228463
Dropout * Activation	0.0506252	0.0075712	6.6865773	0.0000000
Dropout * Initialization	-0.0824082	0.0080767	-10.2032230	0.0000000
Activation * Initialization	0.0072170	0.0076708	0.9408487	0.3468269

Table 29. Gaussian path second-order linear regression model of number of steps needed to escape local region.

Factor	Estimate	Std. Error	T-Value	P-Value
Intercept	39.7765172	0.1615181	246.2665515	0.0000000
Convolutional Width	1.7260253	0.1554425	11.1039479	0.0000000
Convolutional Depth	0.1699066	0.1555887	1.0920239	0.2748732
Fully-Connected Width	1.5905402	0.1554480	10.2319757	0.0000000
Fully-Connected Depth	-0.1821339	0.1608704	-1.1321778	0.2576120
BatchNorm	-0.1086924	0.1614341	-0.6732925	0.5007912
Max Pooling	1.1799752	0.1525673	7.7341301	0.0000000
Dropout	5.0913847	0.1598938	31.8422966	0.0000000
Activation	-0.3754069	0.1545684	-2.4287429	0.0151850
Initialization	-6.1477542	0.1604982	-38.3041893	0.0000000
Convolutional Width * Convolutional Depth	-0.3383442	0.1521149	-2.2242676	0.0261733
Convolutional Width * Fully-Connected Width	1.0544166	0.1508852	6.9882054	0.0000000
Convolutional Width * Fully-Connected Depth	-1.2713093	0.1550353	-8.2001307	0.0000000
Convolutional Width * BatchNorm	1.2085976	0.1553143	7.7816243	0.0000000
Convolutional Width * Max Pooling	-0.0783340	0.1545373	-0.5068938	0.6122509
Convolutional Width * Dropout	-2.0877334	0.1555440	-13.4221395	0.0000000
Convolutional Width * Activation	1.8251167	0.1587573	11.4962692	0.0000000
Convolutional Width * Initialization	-1.1015743	0.1536522	-7.1692696	0.0000000
Convolutional Depth * Fully-Connected Width	2.4341493	0.1598047	15.2320213	0.0000000
Convolutional Depth * Fully-Connected Depth	1.2653300	0.1549340	8.1668955	0.0000000
Convolutional Depth * BatchNorm	3.9017994	0.1555040	25.0913152	0.0000000
Convolutional Depth * Max Pooling	1.3852940	0.1556668	8.8990950	0.0000000
Convolutional Depth * Dropout	2.1549304	0.1544370	13.9534610	0.0000000
Convolutional Depth * Activation	2.1723536	0.1507657	14.4088035	0.0000000
Convolutional Depth * Initialization	0.4229430	0.1558122	2.7144400	0.0066607
Fully-Connected Width * Fully-Connected Depth	0.6176435	0.1543732	4.0009756	0.0000640
Fully-Connected Width * BatchNorm	-1.2992140	0.1553290	-8.3642697	0.0000000
Fully-Connected Width * Dropout	0.6838787	0.1557016	4.3922389	0.0000114
Fully-Connected Width * Activation	-0.7751971	0.1516736	-5.1109549	0.0000003
Fully-Connected Depth * BatchNorm	4.5104858	0.1607802	28.0537424	0.0000000
Fully-Connected Depth * Max Pooling	-0.2395958	0.1521157	-1.5750888	0.1152968
Fully-Connected Depth * Dropout	0.3888884	0.1588425	2.4482643	0.0143875
Fully-Connected Depth * Activation	0.1835299	0.1545805	1.1872769	0.2351727
Fully-Connected Depth * Initialization	-0.1349209	0.1593701	-0.8465883	0.3972636
BatchNorm * Max Pooling	-0.9299987	0.1524721	-6.0994680	0.0000000
BatchNorm * Dropout	1.8358742	0.1599791	11.4757138	0.0000000
BatchNorm * Activation	-1.4181775	0.1544915	-9.1796475	0.0000000
BatchNorm * Initialization	2.3603558	0.1604322	14.7124857	0.0000000
Max Pooling * Dropout	0.4789762	0.1520025	3.1511078	0.0016358
Max Pooling * Activation	0.3432235	0.1546758	2.2189859	0.0265307
Dropout * Activation	-0.2626205	0.1532786	-1.7133540	0.0867071
Dropout * Initialization	-1.0597023	0.1597507	-6.6334752	0.0000000
Activation * Initialization	0.4527415	0.1552239	2.9167007	0.0035527

Table 30. Steepest ascent path reduced second-order linear regression model of Lipschitz Upper Bound Constants.

Factor	Estimate	Std. Error	T-Value	P-Value
Intercept	3.1112806	0.0022238	1399.0821688	0.0000000
Convolutional Width	0.0142622	0.0021465	6.6443426	0.0000000
Convolutional Depth	-0.0616343	0.0021333	-28.8910237	0.0000000
Fully-Connected Width	-0.0074969	0.0021383	-3.5059887	0.0004588
Fully-Connected Depth	-0.0104687	0.0022244	-4.7063603	0.0000026
BatchNorm	0.0553174	0.0022238	24.8753530	0.0000000
Max Pooling	-0.0103139	0.0021105	-4.8870011	0.0000011
Dropout	-0.0090405	0.0022046	-4.1007961	0.0000418
Activation	0.0291586	0.0021315	13.6797257	0.0000000
Initialization	-0.0264538	0.0022040	-12.0025605	0.0000000
Convolutional Width * Convolutional Depth	-0.0034118	0.0021048	-1.6209763	0.1050839
Convolutional Width * Fully-Connected Width	-0.0082124	0.0020738	-3.9600204	0.0000760
Convolutional Width * Fully-Connected Depth	0.0004316	0.0021418	0.2015039	0.8403125
Convolutional Width * BatchNorm	-0.0028622	0.0021458	-1.3338158	0.1823232
Convolutional Width * Max Pooling	-0.0056564	0.0021212	-2.6666049	0.0076861
Convolutional Width * Dropout	-0.0141394	0.0021493	-6.5785486	0.0000000
Convolutional Width * Activation	0.0199724	0.0021937	9.1045001	0.0000000
Convolutional Width * Initialization	-0.0005965	0.0021199	-0.2813672	0.7784401
Convolutional Depth * Fully-Connected Width	0.0173664	0.0021954	7.9104716	0.0000000
Convolutional Depth * Fully-Connected Depth	0.0132892	0.0021353	6.2235251	0.0000000
Convolutional Depth * BatchNorm	-0.0187112	0.0021358	-8.7605622	0.0000000
Convolutional Depth * Max Pooling	-0.0310593	0.0021502	-14.4446888	0.0000000
Convolutional Depth * Dropout	-0.0044434	0.0021232	-2.0927697	0.0364185
Convolutional Depth * Activation	-0.0090691	0.0020762	-4.3682011	0.0000128
Convolutional Depth * Initialization	0.0189379	0.0021437	8.8341183	0.0000000
Fully-Connected Width * Fully-Connected Depth	-0.0241000	0.0021391	-11.2664000	0.0000000
Fully-Connected Width * BatchNorm	0.0233464	0.0021399	10.9098014	0.0000000
Fully-Connected Width * Dropout	-0.0091438	0.0021448	-4.2631600	0.0000205
Fully-Connected Width * Activation	0.0090480	0.0021020	4.3044151	0.0000171
Fully-Connected Depth * BatchNorm	0.0217891	0.0022278	9.7804451	0.0000000
Fully-Connected Depth * Max Pooling	0.0124952	0.0021075	5.9289350	0.0000000
Fully-Connected Depth * Dropout	-0.0214180	0.0022048	-9.7144540	0.0000000
Fully-Connected Depth * Activation	0.0098142	0.0021311	4.6051543	0.0000042
Fully-Connected Depth * Initialization	0.0032532	0.0022010	1.4780158	0.1394647
BatchNorm * Max Pooling	-0.0122160	0.0021109	-5.7869633	0.0000000
BatchNorm * Dropout	0.0103406	0.0022049	4.6898384	0.0000028
BatchNorm * Activation	0.0014447	0.0021366	0.6761886	0.4989512
BatchNorm * Initialization	0.0229844	0.0022035	10.4308320	0.0000000
Max Pooling * Dropout	0.0031776	0.0021054	1.5092478	0.1312968
Max Pooling * Activation	0.0196014	0.0021433	9.1456123	0.0000000
Dropout * Activation	0.0094398	0.0021223	4.4478510	0.0000089
Dropout * Initialization	-0.0152985	0.0021954	-6.9683663	0.0000000
Activation * Initialization	0.0078401	0.0021449	3.6553115	0.0002594

Table 31. Steepest ascent path reduced second-order linear regression model of number of steps.

Factor	Estimate	Std. Error	T-Value	P-Value
Intercept	4.5934359	0.0200218	229.4221502	0.0000000
Convolutional Width	-0.2029830	0.0193109	-10.5113426	0.0000000
Convolutional Depth	0.0003434	0.0192116	0.0178730	0.9857408
Fully-Connected Width	-0.2669818	0.0192467	-13.8715464	0.0000000
Fully-Connected Depth	0.4238338	0.0200390	21.1504568	0.0000000
BatchNorm	-0.0517101	0.0200206	-2.5828401	0.0098263
Max Pooling	0.1872224	0.0189954	9.8562094	0.0000000
Dropout	0.9393034	0.0198445	47.3331492	0.0000000
Activation	0.1173245	0.0191998	6.1107136	0.0000000
Initialization	-0.7071105	0.0198459	-35.6300115	0.0000000
Convolutional Width * Convolutional Depth	-0.1860618	0.0189363	-9.8256467	0.0000000
Convolutional Width * Fully-Connected Width	-0.1743684	0.0186858	-9.3315931	0.0000000
Convolutional Width * Fully-Connected Depth	-0.1188961	0.0192900	-6.1636199	0.0000000
Convolutional Width * BatchNorm	0.1268975	0.0193186	6.5686858	0.0000000
Convolutional Width * Max Pooling	-0.0441593	0.0190961	-2.3124773	0.0207905
Convolutional Width * Dropout	-0.4285329	0.0193537	-22.1421945	0.0000000
Convolutional Width * Activation	0.1291864	0.0197489	6.5414566	0.0000000
Convolutional Width * Initialization	-0.0584613	0.0190843	-3.0633173	0.0022002
Convolutional Depth * Fully-Connected Width	0.1943459	0.0197592	9.8357131	0.0000000
Convolutional Depth * Fully-Connected Depth	0.2455323	0.0192315	12.7671895	0.0000000
Convolutional Depth * BatchNorm	-0.1857993	0.0192207	-9.6666381	0.0000000
Convolutional Depth * Max Pooling	0.2362046	0.0193474	12.2086228	0.0000000
Convolutional Depth * Dropout	0.1961242	0.0191053	10.2654342	0.0000000
Convolutional Depth * Activation	0.3207984	0.0186996	17.1553425	0.0000000
Convolutional Depth * Initialization	-0.1022914	0.0192997	-5.3001623	0.0000001
Fully-Connected Width * Fully-Connected Depth	-0.3350792	0.0192453	-17.4109721	0.0000000
Fully-Connected Width * BatchNorm	-0.0154415	0.0192489	-0.8022024	0.4224728
Fully-Connected Width * Dropout	0.0440295	0.0192995	2.2813733	0.0225668
Fully-Connected Width * Activation	-0.0927546	0.0189179	-4.9030147	0.0000010
Fully-Connected Depth * BatchNorm	0.3160023	0.0200655	15.7485381	0.0000000
Fully-Connected Depth * Max Pooling	0.0203014	0.0189740	1.0699605	0.2846870
Fully-Connected Depth * Dropout	0.5285239	0.0198528	26.6221998	0.0000000
Fully-Connected Depth * Activation	-0.1508417	0.0191895	-7.8606561	0.0000000
Fully-Connected Depth * Initialization	-0.0614072	0.0198267	-3.0971967	0.0019641
BatchNorm * Max Pooling	-0.4159499	0.0189970	-21.8955406	0.0000000
BatchNorm * Dropout	-0.0699127	0.0198451	-3.5229200	0.0004305
BatchNorm * Activation	-0.3366118	0.0192280	-17.5063748	0.0000000
BatchNorm * Initialization	0.3510075	0.0198422	17.6899714	0.0000000
Max Pooling * Dropout	0.0847853	0.0189399	4.4765350	0.0000077
Max Pooling * Activation	0.1929597	0.0192894	10.0034178	0.0000000
Dropout * Activation	0.0221980	0.0190974	1.1623560	0.2451445
Dropout * Initialization	-0.2388716	0.0197587	-12.0894660	0.0000000
Activation * Initialization	0.0462538	0.0193217	2.3938866	0.0167063

7.3 95%Pairwise Student's T-Tests between Alternative Optimizers for Final Training Loss

Figure 22. SGD-Nesterov pairwise, two-tailed 95% student's t-tests of final training loss between alternative optimization routines.

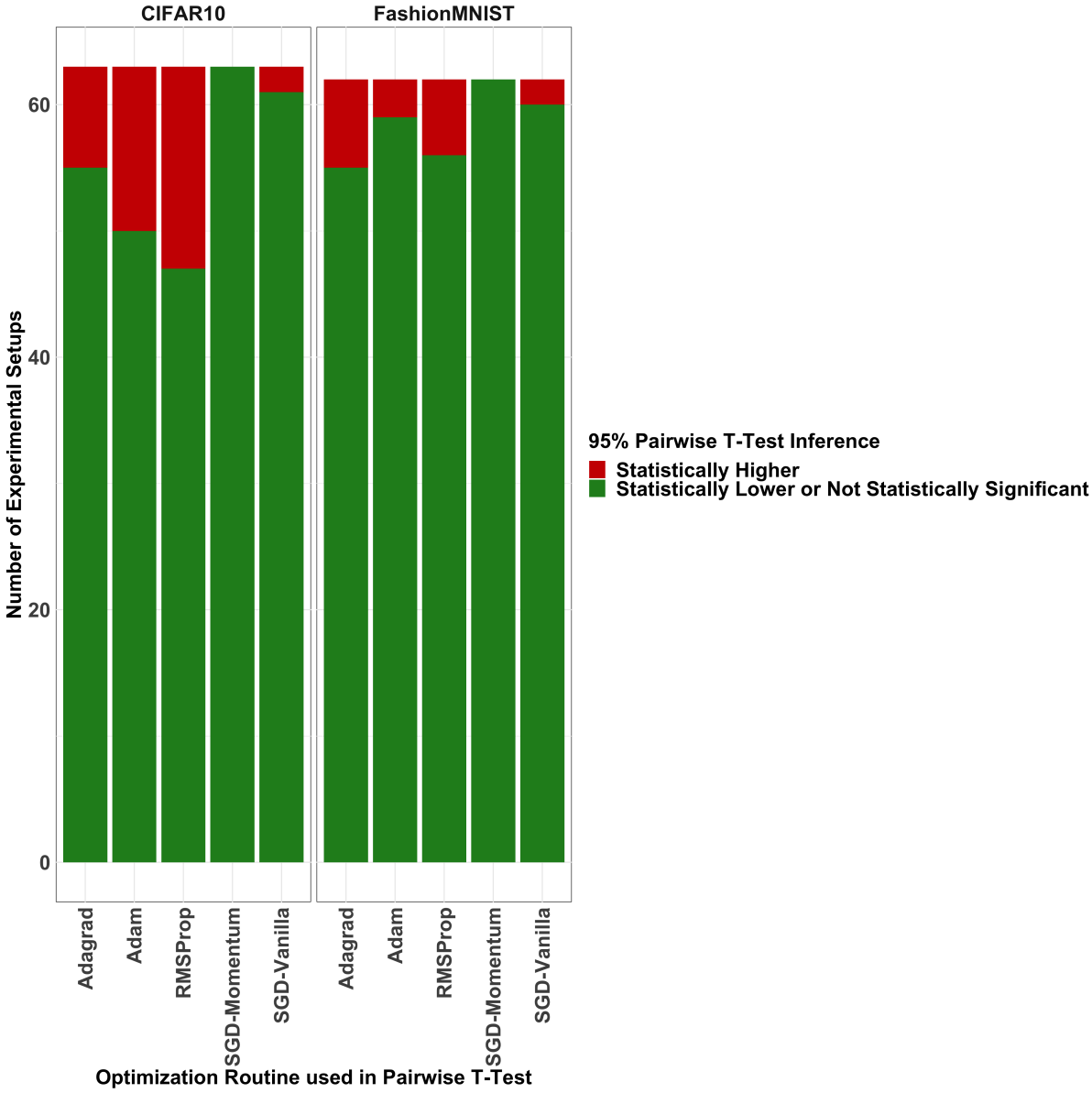


Figure 23. SGD-Momentum pairwise, two-tailed 95% student's t-tests of final training loss between alternative optimization routines.

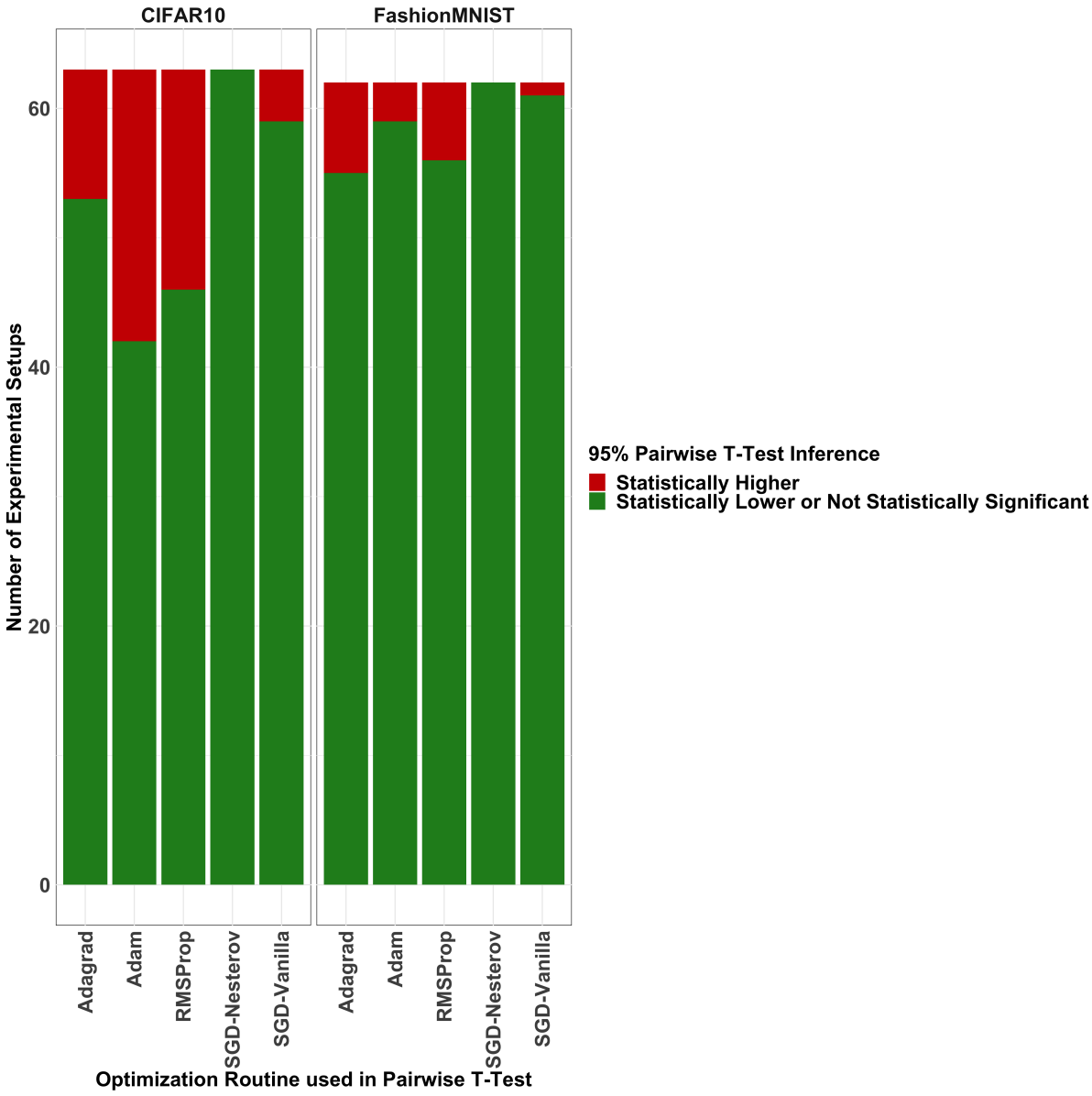


Figure 24. SGD-Vanilla pairwise, two-tailed 95% student's t-tests of final training loss between alternative optimization routines.

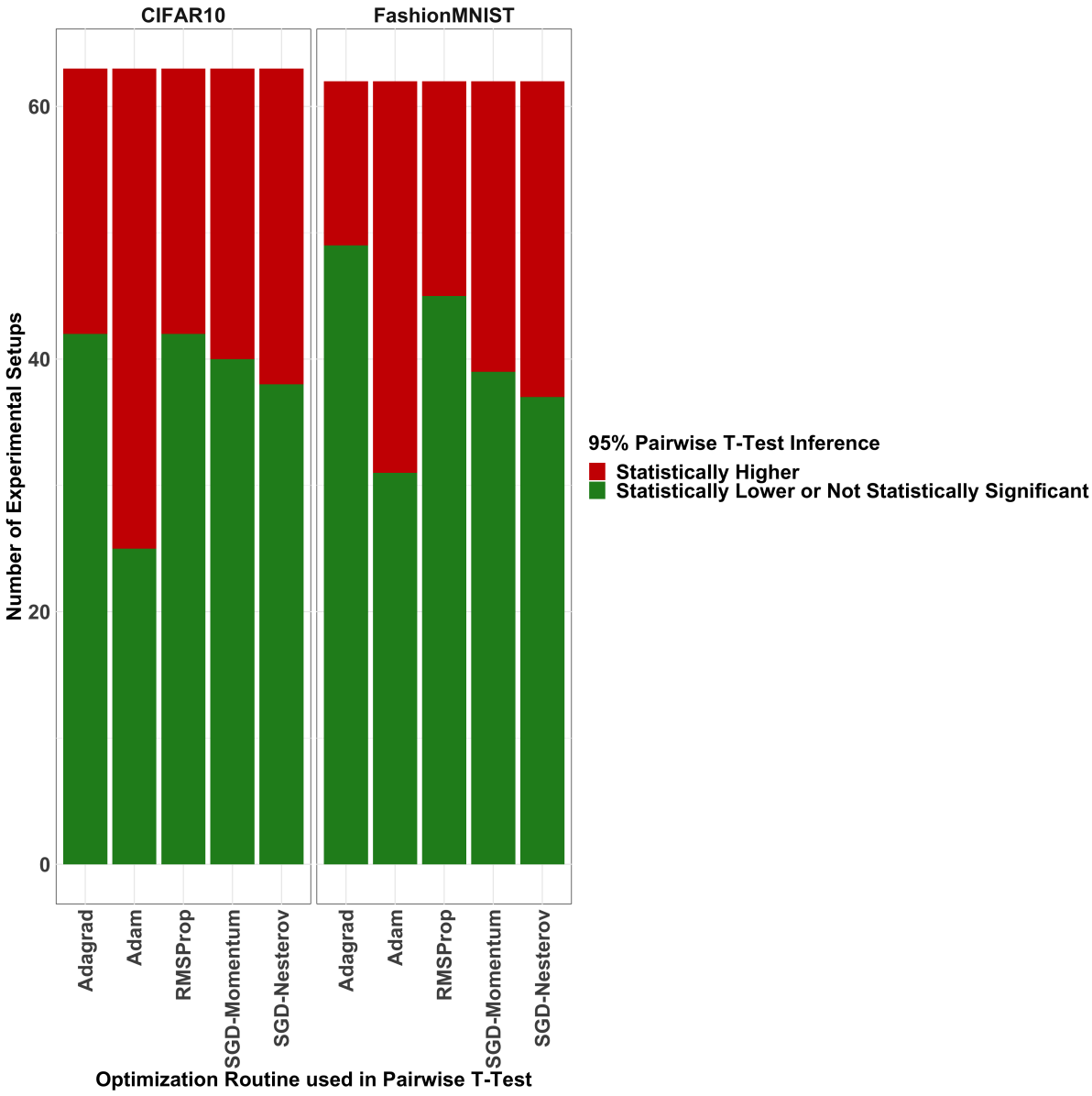


Figure 25. RMSProp pairwise, two-tailed 95% student's t-tests of final training loss between alternative optimization routines.

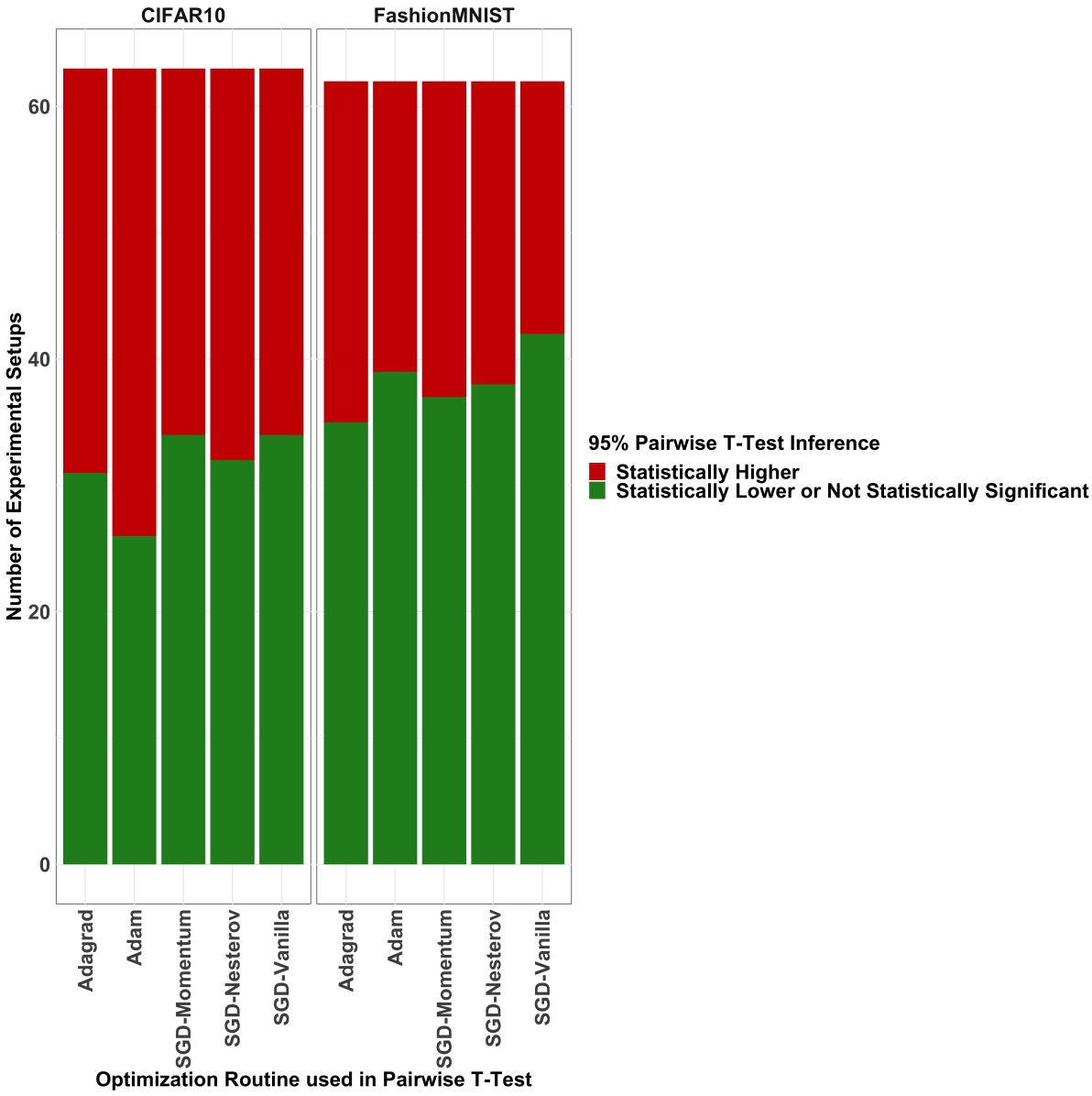
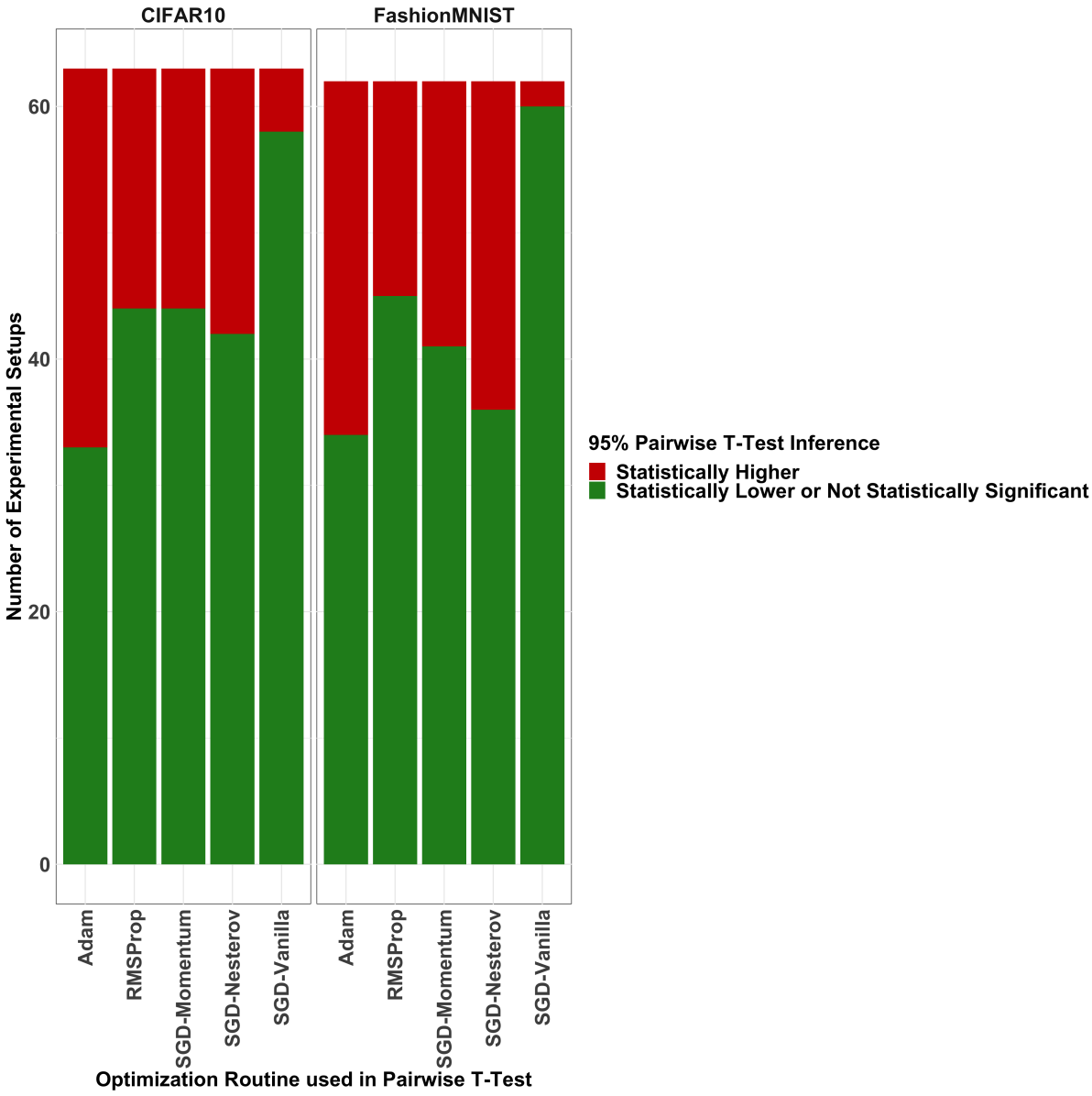


Figure 26. AdaGrad pairwise, two-tailed 95% student's t-tests of final training loss between alternative optimization routines.



7.4 95%Pairwise Student's T-Tests between Alternative Optimizers for Total Test Loss

Figure 27. SGD-Nesterov pairwise, two-tailed 95% student's t-tests of total test loss between alternative optimization routines.

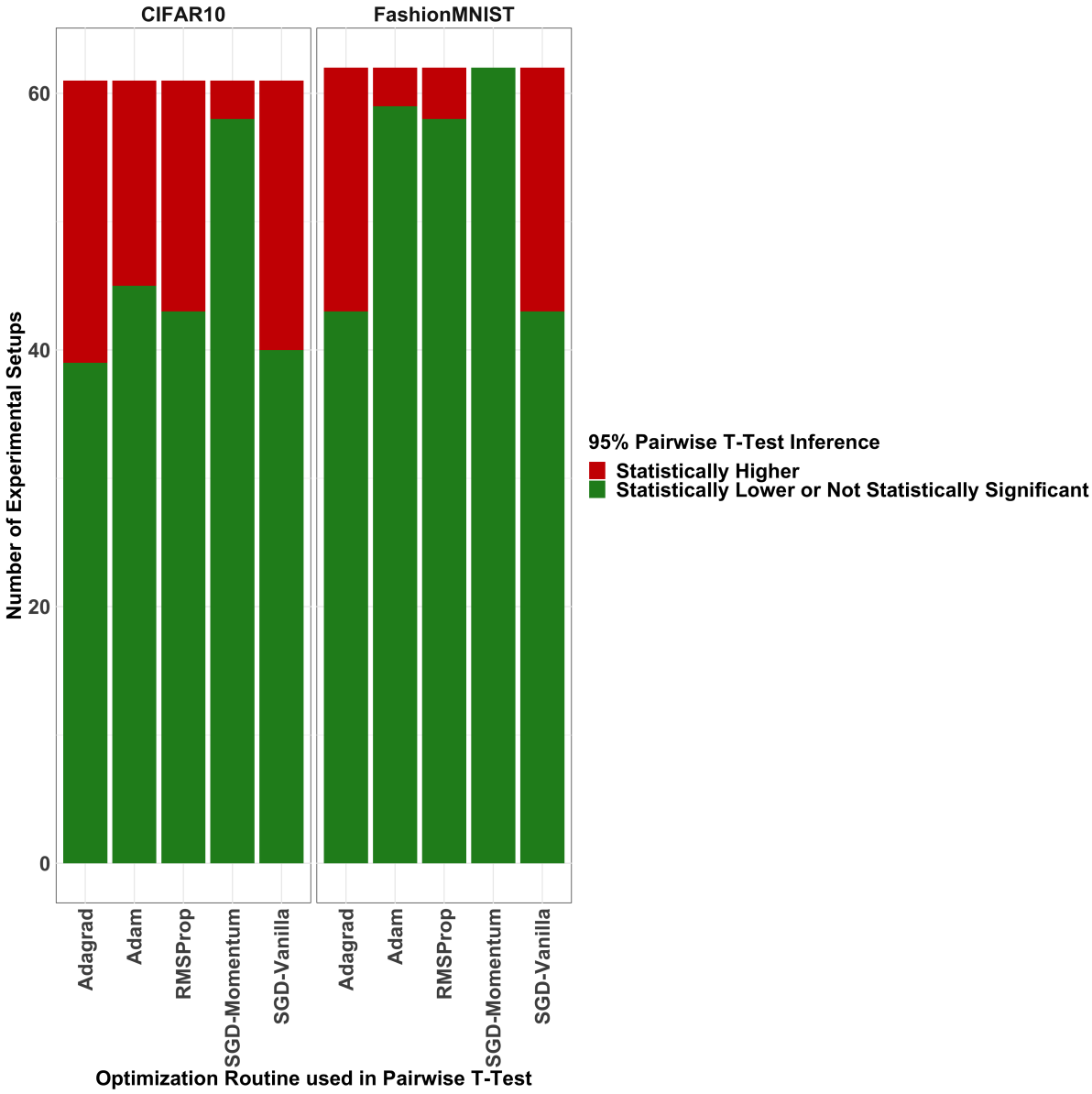


Figure 28. SGD-Momentum pairwise, two-tailed 95% student's t-tests of total test loss between alternative optimization routines.

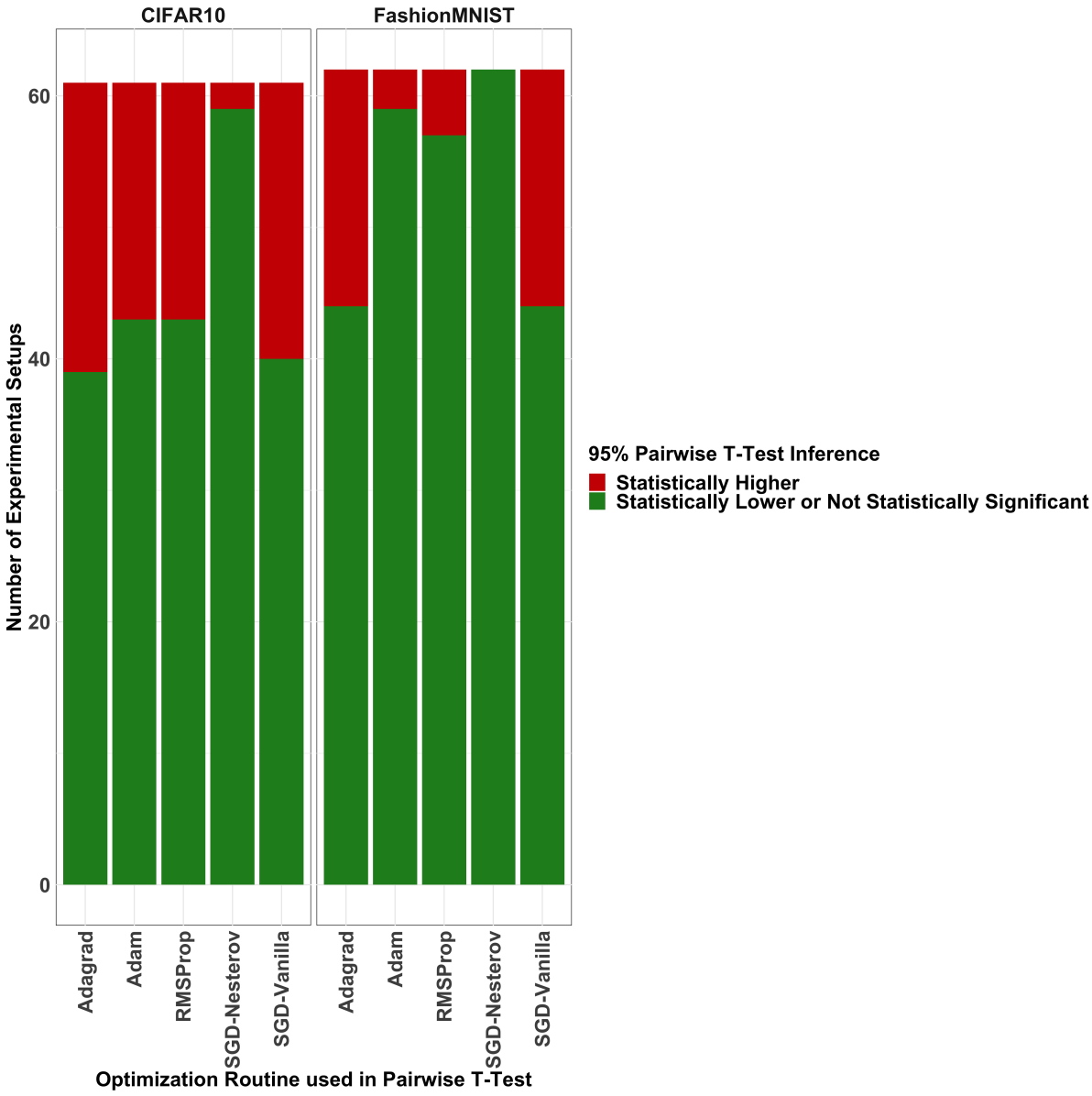


Figure 29. SGD-Vanilla pairwise, two-tailed 95% student's t-tests of total test loss between alternative optimization routines.

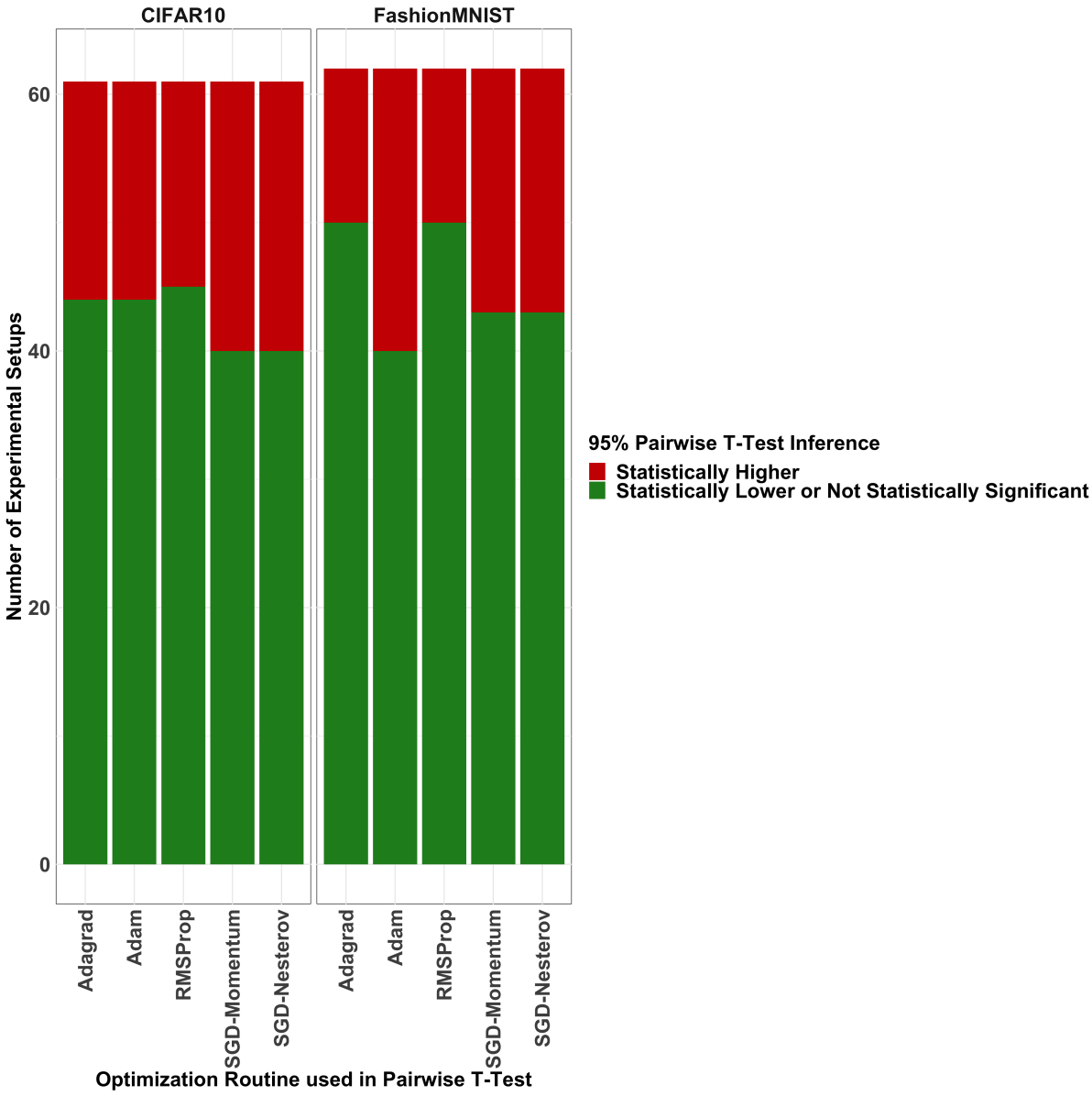


Figure 30. RMSProp pairwise, two-tailed 95% student's t-tests of total test loss between alternative optimization routines.

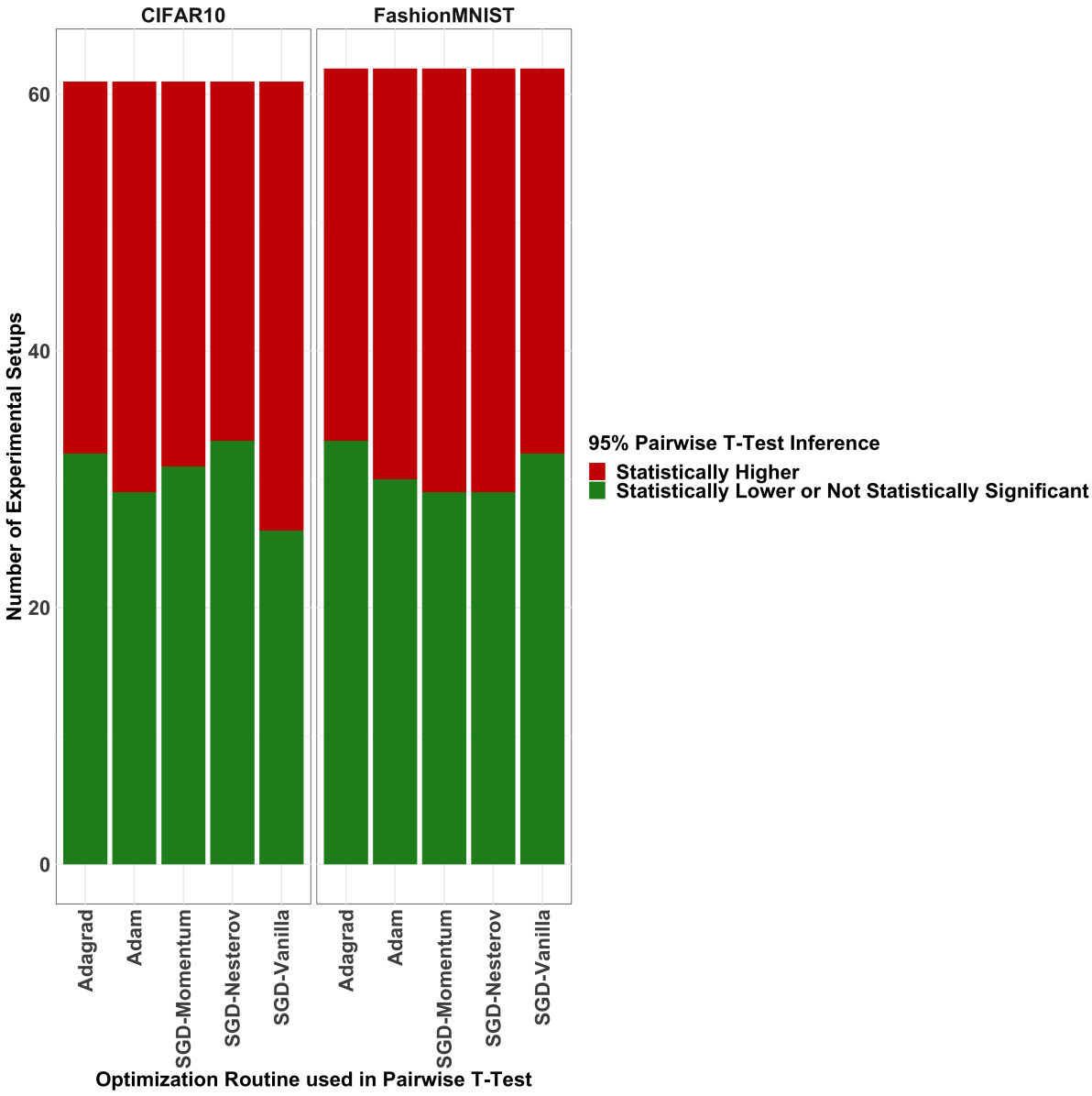
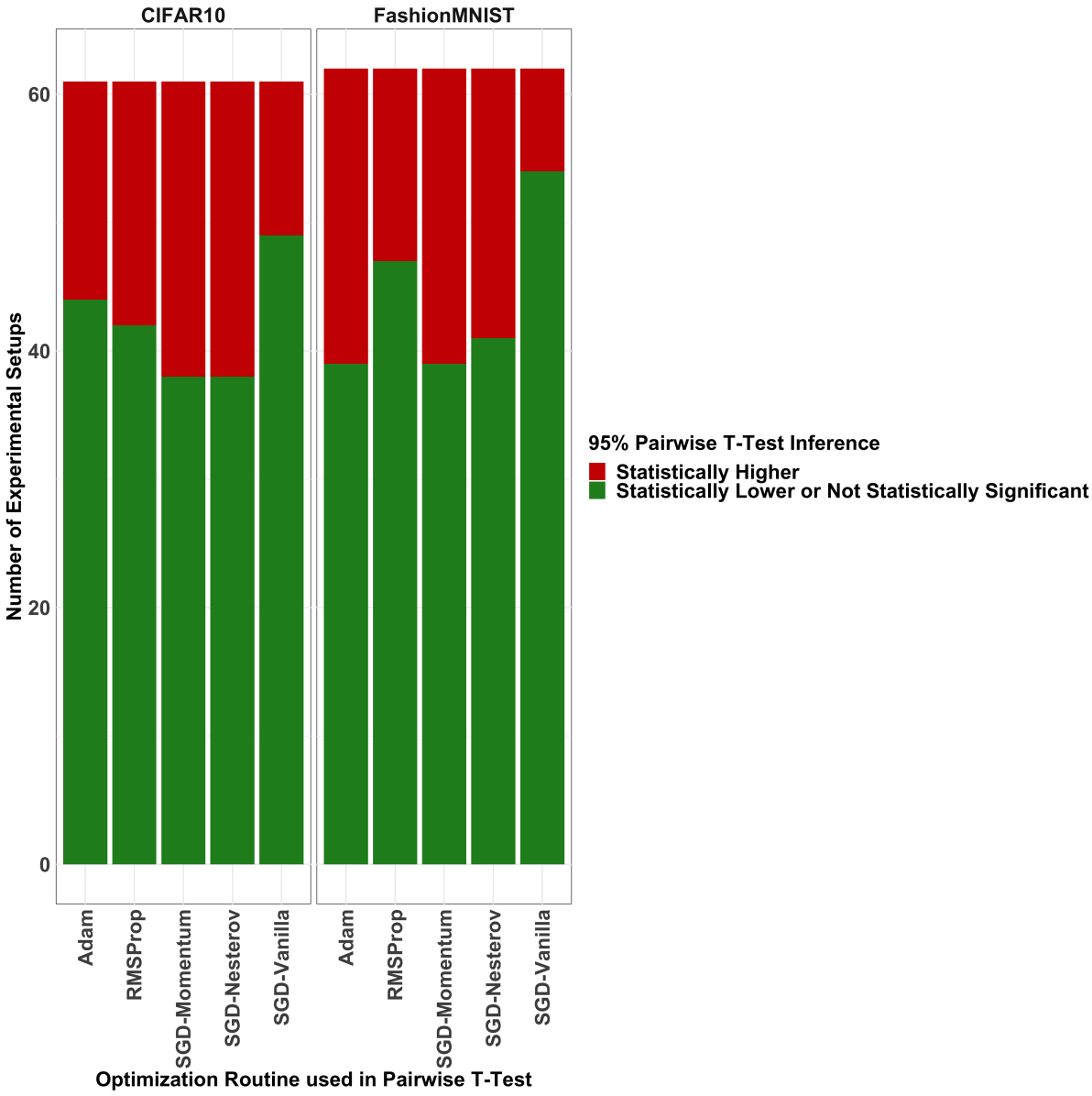


Figure 31. AdaGrad pairwise, two-tailed 95% student's t-tests of total test loss between alternative optimization routines.



7.5 CIFAR-10 Full Train Loss Models

Table 32. Stochastic Gradient Descent with Nesterov's Accelerated Gradient Full Second-Order Train Loss Model

Factor	Estimate	Std. Error	T-Value	P-Value
Intercept	-0.0171360	0.0085050	-2.0148163	0.0443523
Convolutional Width	-0.0991012	0.0084862	-11.6778839	0.0000000
Convolutional Depth	0.4411341	0.0084829	52.0028086	0.0000000
Fully-Connected Width	-0.0592860	0.0084833	-6.9885473	0.0000000
Fully-Connected Depth	0.5374881	0.0085191	63.0919536	0.0000000
BatchNorm	-0.1417289	0.0085077	-16.6588352	0.0000000
Max Pooling	0.0892242	0.0084391	10.5726872	0.0000000
Dropout	0.2470902	0.0085240	28.9874891	0.0000000
Activation	-0.0028403	0.0085097	-0.3337645	0.7386694
Initialization	-0.0257512	0.0085253	-3.0205743	0.0026261
Convolutional Width * Convolutional Depth	-0.0326712	0.0084416	-3.8702465	0.0001202
Convolutional Width * Fully-Connected Width	-0.0336666	0.0084423	-3.9878337	0.0000746
Convolutional Width * Fully-Connected Depth	0.0163372	0.0085082	1.9201791	0.0552908
Convolutional Width * BatchNorm	0.0078633	0.0084886	0.9263374	0.3546288
Convolutional Width * Max Pooling	0.0128241	0.0085017	1.5084168	0.1319540
Convolutional Width * Dropout	0.0445279	0.0084917	5.2436964	0.0000002
Convolutional Width * Activation	0.0095231	0.0085159	1.1182700	0.2638820
Convolutional Width * Initialization	0.0140811	0.0085002	1.6565645	0.0981102
Convolutional Depth * Fully-Connected Width	0.0605555	0.0085071	7.1182009	0.0000000
Convolutional Depth * Fully-Connected Depth	-0.2058503	0.0084979	-24.2237430	0.0000000
Convolutional Depth * BatchNorm	-0.0114149	0.0084864	-1.3450851	0.1790864
Convolutional Depth * Max Pooling	0.0885686	0.0084982	10.4219885	0.0000000
Convolutional Depth * Dropout	-0.0166566	0.0084965	-1.9603997	0.0503936
Convolutional Depth * Activation	-0.0120127	0.0084555	-1.4206854	0.1559077
Convolutional Depth * Initialization	-0.0113748	0.0085019	-1.3379036	0.1814152
Fully-Connected Width * Fully-Connected Depth	0.0250614	0.0084985	2.9489157	0.0033082
Fully-Connected Width * BatchNorm	-0.0160023	0.0084870	-1.8855169	0.0598244
Fully-Connected Width * Dropout	0.0036389	0.0084970	0.4282577	0.6686113
Fully-Connected Width * Activation	-0.0052578	0.0084573	-0.6216846	0.5343763
Fully-Connected Depth * BatchNorm	0.1301365	0.0085138	15.2853102	0.0000000
Fully-Connected Depth * Max Pooling	-0.0627070	0.0084301	-7.4384793	0.0000000
Fully-Connected Depth * Dropout	0.2292417	0.0085089	26.9415396	0.0000000
Fully-Connected Depth * Activation	-0.0335694	0.0084857	-3.9560006	0.0000850
Fully-Connected Depth * Initialization	-0.0061578	0.0085091	-0.7236686	0.4695404
BatchNorm * Max Pooling	0.0086014	0.0084348	1.0197530	0.3082408
BatchNorm * Dropout	-0.0505156	0.0085281	-5.9234515	0.0000000
BatchNorm * Activation	0.0170156	0.0085046	2.0007471	0.0458524
BatchNorm * Initialization	0.0232802	0.0085210	2.7320764	0.0064718
Max Pooling * Dropout	0.0046179	0.0084421	0.5470115	0.5845664
Max Pooling * Activation	-0.0045395	0.0084801	-0.5353048	0.5926298
Dropout * Activation	-0.0080519	0.0084865	-0.9487875	0.3430960
Dropout * Initialization	-0.0410535	0.0085097	-4.8243266	0.0000018
Activation * Initialization	-0.0455346	0.0084836	-5.3673398	0.0000001

Table 33. Stochastic Gradient Descent with Momentum Full Second-Order Train Loss Model

Factor	Estimate	Std. Error	T-Value	P-Value
Intercept	0.0371158	0.0082845	4.4801457	0.0000089
Convolutional Width	-0.0947885	0.0082897	-11.4344409	0.0000000
Convolutional Depth	0.4464144	0.0082649	54.0132412	0.0000000
Fully-Connected Width	-0.0615731	0.0082707	-7.4447022	0.0000000
Fully-Connected Depth	0.5226155	0.0082937	63.0137885	0.0000000
BatchNorm	-0.1519643	0.0082845	-18.3431919	0.0000000
Max Pooling	0.0710249	0.0082517	8.6073164	0.0000000
Dropout	0.2195280	0.0082940	26.4682169	0.0000000
Activation	0.0019935	0.0082948	0.2403321	0.8101508
Initialization	-0.0145458	0.0083003	-1.7524382	0.0801834
Convolutional Width * Convolutional Depth	-0.0339806	0.0082549	-4.1164123	0.0000436
Convolutional Width * Fully-Connected Width	-0.0290640	0.0082504	-3.5227238	0.0004579
Convolutional Width * Fully-Connected Depth	0.0282745	0.0082883	3.4113635	0.0006876
Convolutional Width * BatchNorm	-0.0025162	0.0082897	-0.3035287	0.7615868
Convolutional Width * Max Pooling	0.0125210	0.0082727	1.5135268	0.1306458
Convolutional Width * Dropout	0.0424455	0.0082909	5.1195345	0.0000004
Convolutional Width * Activation	-0.0064685	0.0082888	-0.7803863	0.4354557
Convolutional Width * Initialization	0.0150646	0.0082825	1.8188514	0.0694072
Convolutional Depth * Fully-Connected Width	0.0622936	0.0082926	7.5119146	0.0000000
Convolutional Depth * Fully-Connected Depth	-0.2297714	0.0082721	-27.7766526	0.0000000
Convolutional Depth * BatchNorm	-0.0072599	0.0082649	-0.8783980	0.3800616
Convolutional Depth * Max Pooling	0.0733724	0.0082909	8.8497646	0.0000000
Convolutional Depth * Dropout	-0.0067103	0.0082727	-0.8111362	0.4175928
Convolutional Depth * Activation	-0.0088773	0.0082595	-1.0747950	0.2828767
Convolutional Depth * Initialization	-0.0032087	0.0082773	-0.3876565	0.6984008
Fully-Connected Width * Fully-Connected Depth	0.0287857	0.0082690	3.4811719	0.0005336
Fully-Connected Width * BatchNorm	-0.0179393	0.0082707	-2.1690143	0.0304537
Fully-Connected Width * Dropout	-0.0042213	0.0082773	-0.5099853	0.6102398
Fully-Connected Width * Activation	-0.0016206	0.0082532	-0.1963578	0.8443933
Fully-Connected Depth * BatchNorm	0.1339611	0.0082937	16.1522159	0.0000000
Fully-Connected Depth * Max Pooling	-0.0482071	0.0082441	-5.8474682	0.0000000
Fully-Connected Depth * Dropout	0.2106787	0.0082959	25.3956322	0.0000000
Fully-Connected Depth * Activation	-0.0274219	0.0082835	-3.3104089	0.0009846
Fully-Connected Depth * Initialization	-0.0192386	0.0082897	-2.3207885	0.0206156
BatchNorm * Max Pooling	0.0029852	0.0082517	0.3617708	0.7176442
BatchNorm * Dropout	-0.0365713	0.0082940	-4.4093588	0.0000122
BatchNorm * Activation	0.0069199	0.0082948	0.8342468	0.4044571
BatchNorm * Initialization	0.0180958	0.0083003	2.1801272	0.0296163
Max Pooling * Dropout	0.0088676	0.0082549	1.0742199	0.2831341
Max Pooling * Activation	-0.0042239	0.0082686	-0.5108347	0.6096452
Dropout * Activation	-0.0084905	0.0082859	-1.0246928	0.3059002
Dropout * Initialization	-0.0372147	0.0082926	-4.4876722	0.0000086
Activation * Initialization	-0.0394123	0.0082812	-4.7592596	0.0000024

Table 34. Stochastic Gradient Descent Full Second-Order Train Loss Model

Factor	Estimate	Std. Error	T-Value	P-Value
Intercept	0.3279035	0.0076511	42.8570059	0.0000000
Convolutional Width	-0.0813387	0.0076575	-10.6220481	0.0000000
Convolutional Depth	0.3571268	0.0076521	46.6701722	0.0000000
Fully-Connected Width	-0.0489545	0.0076575	-6.3929903	0.0000000
Fully-Connected Depth	0.4473652	0.0076566	58.4290827	0.0000000
BatchNorm	-0.0783727	0.0076511	-10.2433183	0.0000000
Max Pooling	0.0331224	0.0076511	4.3291005	0.0000173
Dropout	0.2382066	0.0076459	31.1546528	0.0000000
Activation	0.0013952	0.0076575	0.1822000	0.8554829
Initialization	-0.0088266	0.0076511	-1.1536434	0.2490724
Convolutional Width * Convolutional Depth	-0.0240345	0.0076566	-3.1390729	0.0017719
Convolutional Width * Fully-Connected Width	-0.0143959	0.0076511	-1.8815406	0.0603471
Convolutional Width * Fully-Connected Depth	0.0296683	0.0076521	3.8771282	0.0001165
Convolutional Width * BatchNorm	-0.0148200	0.0076575	-1.9353433	0.0533836
Convolutional Width * Max Pooling	-0.0157949	0.0076470	-2.0654932	0.0392740
Convolutional Width * Dropout	0.0182959	0.0076521	2.3909537	0.0170895
Convolutional Width * Activation	0.0327229	0.0076511	4.2768888	0.0000218
Convolutional Width * Initialization	0.0120795	0.0076470	1.5796312	0.1146801
Convolutional Depth * Fully-Connected Width	0.0660041	0.0076566	8.6206049	0.0000000
Convolutional Depth * Fully-Connected Depth	-0.1591411	0.0076575	-20.7822900	0.0000000
Convolutional Depth * BatchNorm	-0.0094092	0.0076521	-1.2296208	0.2192861
Convolutional Depth * Max Pooling	0.0643664	0.0076521	8.4115544	0.0000000
Convolutional Depth * Dropout	0.0179172	0.0076470	2.3430269	0.0194300
Convolutional Depth * Activation	-0.0016171	0.0076566	-0.2112040	0.8327946
Convolutional Depth * Initialization	0.0040565	0.0076521	0.5301132	0.5962153
Fully-Connected Width * Fully-Connected Depth	-0.0110265	0.0076521	-1.4409632	0.1500789
Fully-Connected Width * BatchNorm	-0.0030243	0.0076575	-0.3949431	0.6930151
Fully-Connected Width * Dropout	0.0101660	0.0076521	1.3285176	0.1844756
Fully-Connected Width * Activation	0.0117118	0.0076511	1.5307271	0.1263256
Fully-Connected Depth * BatchNorm	0.0843831	0.0076566	11.0210370	0.0000000
Fully-Connected Depth * Max Pooling	-0.0062816	0.0076459	-0.8215637	0.4116279
Fully-Connected Depth * Dropout	0.2332299	0.0076511	30.4831619	0.0000000
Fully-Connected Depth * Activation	-0.0112418	0.0076521	-1.4691069	0.1422898
Fully-Connected Depth * Initialization	0.0076128	0.0076459	0.9956663	0.3197845
BatchNorm * Max Pooling	0.0056420	0.0076511	0.7374079	0.4611417
BatchNorm * Dropout	-0.0046256	0.0076459	-0.6049728	0.5454092
BatchNorm * Activation	0.0066903	0.0076575	0.8736825	0.3826154
BatchNorm * Initialization	-0.0194868	0.0076511	-2.5469222	0.0110982
Max Pooling * Dropout	-0.0102008	0.0076566	-1.3323008	0.1832304
Max Pooling * Activation	-0.0048687	0.0076470	-0.6366853	0.5245550
Dropout * Activation	-0.0050402	0.0076521	-0.6586632	0.5103464
Dropout * Initialization	-0.0391095	0.0076566	-5.1079851	0.0000004
Activation * Initialization	-0.0250677	0.0076470	-3.2780982	0.0011010

Table 35. RMSProp Full Second-Order Train Loss Model

Factor	Estimate	Std. Error	T-Value	P-Value
Intercept	0.1965682	0.0127307	15.4405368	0.0000000
Convolutional Width	0.0168004	0.0127337	1.3193582	0.1875234
Convolutional Depth	0.4095908	0.0127393	32.1517939	0.0000000
Fully-Connected Width	0.0449750	0.0127187	3.5361297	0.0004355
Fully-Connected Depth	0.2691709	0.0127198	21.1615787	0.0000000
BatchNorm	-0.5646123	0.0127399	-44.3183244	0.0000000
Max Pooling	0.0934736	0.0127075	7.3557610	0.0000000
Dropout	0.1568240	0.0127122	12.3364769	0.0000000
Activation	-0.0560400	0.0127282	-4.4028336	0.0000125
Initialization	0.0057840	0.0126966	0.4555555	0.6488649
Convolutional Width * Convolutional Depth	0.0519001	0.0127334	4.0758936	0.0000516
Convolutional Width * Fully-Connected Width	0.0430618	0.0127253	3.3839592	0.0007583
Convolutional Width * Fully-Connected Depth	-0.0466855	0.0127236	-3.6692100	0.0002637
Convolutional Width * BatchNorm	-0.1222548	0.0127341	-9.6005551	0.0000000
Convolutional Width * Max Pooling	0.0292045	0.0127129	2.2972363	0.0219287
Convolutional Width * Dropout	0.0727943	0.0127086	5.7279542	0.0000000
Convolutional Width * Activation	-0.0529859	0.0127355	-4.1604815	0.0000361
Convolutional Width * Initialization	0.0388512	0.0126965	3.0599929	0.0023064
Convolutional Depth * Fully-Connected Width	0.0474348	0.0127253	3.7276002	0.0002105
Convolutional Depth * Fully-Connected Depth	-0.2522699	0.0127298	-19.8172115	0.0000000
Convolutional Depth * BatchNorm	0.0980889	0.0127315	7.7044475	0.0000000
Convolutional Depth * Max Pooling	0.0755034	0.0127085	5.9411752	0.0000000
Convolutional Depth * Dropout	-0.0965187	0.0127028	-7.5982432	0.0000000
Convolutional Depth * Activation	-0.0686279	0.0127386	-5.3874063	0.0000001
Convolutional Depth * Initialization	0.0137465	0.0126961	1.0827379	0.2793345
Fully-Connected Width * Fully-Connected Depth	-0.0936563	0.0127321	-7.3559254	0.0000000
Fully-Connected Width * BatchNorm	-0.0436433	0.0127250	-3.4297361	0.0006432
Fully-Connected Width * Dropout	0.0340050	0.0126964	2.6783300	0.0075899
Fully-Connected Width * Activation	-0.0596663	0.0127234	-4.6894951	0.0000034
Fully-Connected Depth * BatchNorm	0.2060170	0.0127303	16.1831385	0.0000000
Fully-Connected Depth * Max Pooling	-0.0542859	0.0126973	-4.2753745	0.0000220
Fully-Connected Depth * Dropout	0.0934539	0.0127038	7.3563553	0.0000000
Fully-Connected Depth * Activation	0.0400512	0.0127198	3.1487332	0.0017165
Fully-Connected Depth * Initialization	-0.0342712	0.0127078	-2.6968581	0.0071848
BatchNorm * Max Pooling	0.0296831	0.0127107	2.3352862	0.0198380
BatchNorm * Dropout	0.1032988	0.0127023	8.1322851	0.0000000
BatchNorm * Activation	0.0559434	0.0127385	4.3916689	0.0000132
BatchNorm * Initialization	-0.0193188	0.0126993	-1.5212447	0.1286947
Max Pooling * Dropout	-0.0500993	0.0127346	-3.9341010	0.0000927
Max Pooling * Activation	-0.0140225	0.0127103	-1.1032358	0.2703414
Dropout * Activation	0.0241845	0.0127056	1.9034491	0.0574324
Dropout * Initialization	-0.0380785	0.0127209	-2.9933903	0.0028658
Activation * Initialization	-0.0032837	0.0126995	-0.2585725	0.7960487

Table 36. Adam Full Second-Order Train Loss Model

Factor	Estimate	Std. Error	T-Value	P-Value
Intercept	-0.0901304	0.0075078	-12.0049584	0.0000000
Convolutional Width	-0.0895842	0.0075053	-11.9361937	0.0000000
Convolutional Depth	0.3173289	0.0075078	42.2662950	0.0000000
Fully-Connected Width	-0.0797820	0.0075073	-10.6272927	0.0000000
Fully-Connected Depth	0.5002248	0.0075073	66.6320013	0.0000000
BatchNorm	-0.1450845	0.0075078	-19.3246067	0.0000000
Max Pooling	0.1394934	0.0075060	18.5843234	0.0000000
Dropout	0.2378058	0.0075079	31.6740016	0.0000000
Activation	0.0444608	0.0075062	5.9232154	0.0000000
Initialization	-0.0359629	0.0075070	-4.7905824	0.0000021
Convolutional Width * Convolutional Depth	-0.0284852	0.0075062	-3.7948942	0.0001615
Convolutional Width * Fully-Connected Width	0.0092916	0.0075042	1.2381895	0.2160921
Convolutional Width * Fully-Connected Depth	0.0421042	0.0075042	5.6107400	0.0000000
Convolutional Width * BatchNorm	-0.0036215	0.0075053	-0.4825321	0.6295902
Convolutional Width * Max Pooling	0.0028375	0.0075078	0.3779450	0.7055946
Convolutional Width * Dropout	0.0494497	0.0075056	6.5884146	0.0000000
Convolutional Width * Activation	0.0190233	0.0075078	2.5337903	0.0115170
Convolutional Width * Initialization	0.0425708	0.0075047	5.6725811	0.0000000
Convolutional Depth * Fully-Connected Width	0.0177223	0.0075068	2.3608140	0.0185287
Convolutional Depth * Fully-Connected Depth	-0.1361845	0.0075068	-18.1413955	0.0000000
Convolutional Depth * BatchNorm	0.0422635	0.0075078	5.6292426	0.0000000
Convolutional Depth * Max Pooling	0.1052735	0.0075056	14.0260851	0.0000000
Convolutional Depth * Dropout	-0.0282967	0.0075078	-3.7689912	0.0001788
Convolutional Depth * Activation	-0.0226117	0.0075053	-3.0127771	0.0026892
Convolutional Depth * Initialization	0.0290491	0.0075071	3.8695666	0.0001200
Fully-Connected Width * Fully-Connected Depth	-0.0034819	0.0075078	-0.4637777	0.6429620
Fully-Connected Width * BatchNorm	0.0044531	0.0075073	0.5931649	0.5532768
Fully-Connected Width * Dropout	-0.0362456	0.0075071	-4.8282106	0.0000017
Fully-Connected Width * Activation	0.0285011	0.0075044	3.7979405	0.0001596
Fully-Connected Depth * BatchNorm	0.1432577	0.0075073	19.0825119	0.0000000
Fully-Connected Depth * Max Pooling	-0.0913504	0.0075047	-12.1724929	0.0000000
Fully-Connected Depth * Dropout	0.2630284	0.0075071	35.0374879	0.0000000
Fully-Connected Depth * Activation	-0.0382339	0.0075044	-5.0948867	0.0000005
Fully-Connected Depth * Initialization	-0.0044427	0.0075078	-0.5917449	0.5542269
BatchNorm * Max Pooling	-0.0009158	0.0075060	-0.1220052	0.9029326
BatchNorm * Dropout	-0.0003655	0.0075079	-0.0486834	0.9611866
BatchNorm * Activation	-0.0534153	0.0075062	-7.1161623	0.0000000
BatchNorm * Initialization	0.0025502	0.0075070	0.3397052	0.7341881
Max Pooling * Dropout	-0.0317074	0.0075062	-4.2241576	0.0000274
Max Pooling * Activation	-0.0089519	0.0075079	-1.1923241	0.2335686
Dropout * Activation	0.0100591	0.0075060	1.3401443	0.1806660
Dropout * Initialization	-0.0100497	0.0075068	-1.3387358	0.1811240
Activation * Initialization	-0.0636175	0.0075039	-8.4779638	0.0000000

Table 37. AdaGrad Full Second-Order Train Loss Model

Factor	Estimate	Std. Error	T-Value	P-Value
Intercept	0.1768821	0.0071851	24.6179843	0.0000000
Convolutional Width	-0.1061402	0.0071804	-14.7819562	0.0000000
Convolutional Depth	0.2572037	0.0071851	35.7969391	0.0000000
Fully-Connected Width	-0.0899450	0.0071841	-12.5199934	0.0000000
Fully-Connected Depth	0.4719600	0.0071749	65.7797304	0.0000000
BatchNorm	-0.0928635	0.0071851	-12.9245000	0.0000000
Max Pooling	0.1233973	0.0071804	17.1853227	0.0000000
Dropout	0.2033225	0.0071851	28.2978994	0.0000000
Activation	0.0307575	0.0071781	4.2848790	0.0000211
Initialization	-0.0381496	0.0071841	-5.3102698	0.0000002
Convolutional Width * Convolutional Depth	-0.0650672	0.0071804	-9.0617881	0.0000000
Convolutional Width * Fully-Connected Width	-0.0121984	0.0071784	-1.6993195	0.0897375
Convolutional Width * Fully-Connected Depth	0.0611319	0.0071779	8.5166421	0.0000000
Convolutional Width * BatchNorm	0.0039802	0.0071804	0.5543199	0.5795506
Convolutional Width * Max Pooling	0.0019911	0.0071851	0.2771197	0.7817763
Convolutional Width * Dropout	0.0334283	0.0071804	4.6554957	0.0000039
Convolutional Width * Activation	0.0309534	0.0071761	4.3133954	0.0000186
Convolutional Width * Initialization	0.0414612	0.0071784	5.7758353	0.0000000
Convolutional Depth * Fully-Connected Width	0.0344245	0.0071841	4.7917487	0.0000021
Convolutional Depth * Fully-Connected Depth	-0.1110035	0.0071749	-15.4711916	0.0000000
Convolutional Depth * BatchNorm	0.0769410	0.0071851	10.7084495	0.0000000
Convolutional Depth * Max Pooling	0.1194751	0.0071804	16.6390760	0.0000000
Convolutional Depth * Dropout	-0.0094654	0.0071851	-1.3173679	0.1881795
Convolutional Depth * Activation	-0.0070174	0.0071781	-0.9776015	0.3286351
Convolutional Depth * Initialization	0.0369642	0.0071841	5.1452764	0.0000004
Fully-Connected Width * Fully-Connected Depth	-0.0033454	0.0071761	-0.4661834	0.6412404
Fully-Connected Width * BatchNorm	-0.0064102	0.0071841	-0.8922768	0.3725748
Fully-Connected Width * Dropout	-0.0097920	0.0071841	-1.3630086	0.1733518
Fully-Connected Width * Activation	0.0491171	0.0071779	6.8427960	0.0000000
Fully-Connected Depth * BatchNorm	0.1198561	0.0071749	16.7050232	0.0000000
Fully-Connected Depth * Max Pooling	-0.0735417	0.0071779	-10.2455199	0.0000000
Fully-Connected Depth * Dropout	0.2403532	0.0071749	33.4993890	0.0000000
Fully-Connected Depth * Activation	-0.0317042	0.0071784	-4.4166189	0.0000117
Fully-Connected Depth * Initialization	-0.0006799	0.0071761	-0.0947444	0.9245471
BatchNorm * Max Pooling	0.0106357	0.0071804	1.4812191	0.1390328
BatchNorm * Dropout	0.0142417	0.0071851	1.9821211	0.0478864
BatchNorm * Activation	-0.0336258	0.0071781	-4.6844658	0.0000034
BatchNorm * Initialization	-0.0065533	0.0071841	-0.9121894	0.3620072
Max Pooling * Dropout	-0.0034929	0.0071804	-0.4864496	0.6268124
Max Pooling * Activation	-0.0017501	0.0071761	-0.2438728	0.8074063
Dropout * Activation	0.0122718	0.0071781	1.7096114	0.0878150
Dropout * Initialization	-0.0087755	0.0071841	-1.2215130	0.2223347
Activation * Initialization	-0.0735618	0.0071779	-10.2483183	0.0000000

7.6 FashionMNIST Full Train Loss Models

Table 38. Stochastic Gradient Descent with Nesterov's Accelerated Gradient Full Second-Order Train Loss Model

Factor	Estimate	Std. Error	T-Value	P-Value
Intercept	-0.6250900	0.0082986	-75.3247234	0.0000000
Convolutional Width	-0.0762917	0.0081580	-9.3517310	0.0000000
Convolutional Depth	0.2907228	0.0081795	35.5427241	0.0000000
Fully-Connected Width	-0.0185891	0.0081148	-2.2907685	0.0223250
Fully-Connected Depth	0.5528653	0.0083006	66.6053051	0.0000000
BatchNorm	-0.1217901	0.0082986	-14.6759747	0.0000000
Max Pooling	0.0992531	0.0081114	12.2361888	0.0000000
Dropout	0.4414588	0.0082181	53.7177525	0.0000000
Activation	-0.0020564	0.0081759	-0.2515223	0.8014970
Initialization	-0.0563056	0.0082753	-6.8040171	0.0000000
Convolutional Width * Convolutional Depth	-0.0419976	0.0080497	-5.2172657	0.0000003
Convolutional Width * Fully-Connected Width	-0.0323310	0.0080251	-4.0287506	0.0000633
Convolutional Width * Fully-Connected Depth	0.0177486	0.0080991	2.1914199	0.0288077
Convolutional Width * BatchNorm	0.0382966	0.0081580	4.6943447	0.0000033
Convolutional Width * Max Pooling	-0.0415331	0.0081310	-5.1079769	0.0000004
Convolutional Width * Dropout	-0.0332125	0.0082116	-4.0446007	0.0000593
Convolutional Width * Activation	-0.0071071	0.0081937	-0.8673851	0.3860801
Convolutional Width * Initialization	0.0389740	0.0080694	4.8298650	0.0000017
Convolutional Depth * Fully-Connected Width	0.0434588	0.0082067	5.2955595	0.0000002
Convolutional Depth * Fully-Connected Depth	-0.0971151	0.0081346	-11.9385774	0.0000000
Convolutional Depth * BatchNorm	-0.0626408	0.0081795	-7.6582340	0.0000000
Convolutional Depth * Max Pooling	0.1264184	0.0082116	15.3951866	0.0000000
Convolutional Depth * Dropout	0.0629239	0.0081310	7.7387376	0.0000000
Convolutional Depth * Activation	-0.0020477	0.0080356	-0.2548301	0.7989422
Convolutional Depth * Initialization	-0.0479276	0.0081652	-5.8697137	0.0000000
Fully-Connected Width * Fully-Connected Depth	-0.0102986	0.0081317	-1.2664729	0.2058386
Fully-Connected Width * BatchNorm	-0.0312611	0.0081148	-3.8523613	0.0001297
Fully-Connected Width * Dropout	0.0082416	0.0081652	1.0093519	0.3132155
Fully-Connected Width * Activation	0.0216648	0.0080170	2.7023454	0.0070814
Fully-Connected Depth * BatchNorm	0.0433046	0.0083006	5.2170378	0.0000003
Fully-Connected Depth * Max Pooling	-0.0549781	0.0080727	-6.8103719	0.0000000
Fully-Connected Depth * Dropout	0.4169704	0.0081933	50.8914012	0.0000000
Fully-Connected Depth * Activation	-0.0356806	0.0081202	-4.3940320	0.0000132
Fully-Connected Depth * Initialization	0.0001976	0.0082512	0.0239495	0.9809009
BatchNorm * Max Pooling	-0.0186451	0.0081114	-2.2986136	0.0218721
BatchNorm * Dropout	-0.1161266	0.0082181	-14.1305591	0.0000000
BatchNorm * Activation	-0.0001648	0.0081759	-0.0201604	0.9839222
BatchNorm * Initialization	0.0512809	0.0082753	6.1968349	0.0000000
Max Pooling * Dropout	0.0263873	0.0080497	3.2780305	0.0011062
Max Pooling * Activation	-0.0098160	0.0081713	-1.2012811	0.2301193
Dropout * Activation	-0.0035323	0.0080900	-0.4366244	0.6625419
Dropout * Initialization	-0.0535261	0.0082067	-6.5222765	0.0000000
Activation * Initialization	-0.0382528	0.0081403	-4.6992068	0.0000032

Table 39. Stochastic Gradient Descent with Momentum Full Second-Order Train Loss Model

Factor	Estimate	Std. Error	T-Value	P-Value
Intercept	-0.6156716	0.0081973	-75.1070655	0.0000000
Convolutional Width	-0.0649579	0.0080515	-8.0678529	0.0000000
Convolutional Depth	0.3016770	0.0080790	37.3408300	0.0000000
Fully-Connected Width	-0.0258813	0.0080421	-3.2182254	0.0013588
Fully-Connected Depth	0.5535468	0.0081982	67.5206141	0.0000000
BatchNorm	-0.1384683	0.0082070	-16.8720056	0.0000000
Max Pooling	0.1053167	0.0080152	13.1397005	0.0000000
Dropout	0.4331965	0.0081326	53.2663937	0.0000000
Activation	-0.0006312	0.0080707	-0.0782067	0.9376895
Initialization	-0.0557170	0.0081522	-6.8346018	0.0000000
Convolutional Width * Convolutional Depth	-0.0368016	0.0079746	-4.6148569	0.0000048
Convolutional Width * Fully-Connected Width	-0.0359507	0.0079634	-4.5144699	0.0000076
Convolutional Width * Fully-Connected Depth	0.0176094	0.0080425	2.1895449	0.0289385
Convolutional Width * BatchNorm	0.0330024	0.0080511	4.0991460	0.0000471
Convolutional Width * Max Pooling	-0.0431000	0.0080249	-5.3707704	0.0000001
Convolutional Width * Dropout	-0.0384658	0.0080830	-4.7588384	0.0000024
Convolutional Width * Activation	-0.0040155	0.0081063	-0.4953526	0.6205311
Convolutional Width * Initialization	0.0388391	0.0080069	4.8507090	0.0000016
Convolutional Depth * Fully-Connected Width	0.0462739	0.0081058	5.7087404	0.0000000
Convolutional Depth * Fully-Connected Depth	-0.0900609	0.0080719	-11.1572884	0.0000000
Convolutional Depth * BatchNorm	-0.0588453	0.0080790	-7.2837560	0.0000000
Convolutional Depth * Max Pooling	0.1114820	0.0080832	13.7918171	0.0000000
Convolutional Depth * Dropout	0.0669038	0.0080255	8.3364037	0.0000000
Convolutional Depth * Activation	0.0028994	0.0079667	0.3639448	0.7160264
Convolutional Depth * Initialization	-0.0394519	0.0080743	-4.8861149	0.0000013
Fully-Connected Width * Fully-Connected Depth	-0.0117966	0.0080493	-1.4655394	0.1432933
Fully-Connected Width * BatchNorm	-0.0317646	0.0080415	-3.9500614	0.0000873
Fully-Connected Width * Dropout	0.0141367	0.0080727	1.7511651	0.0804242
Fully-Connected Width * Activation	0.0251583	0.0079718	3.1559182	0.0016794
Fully-Connected Depth * BatchNorm	0.0397833	0.0082076	4.8471225	0.0000016
Fully-Connected Depth * Max Pooling	-0.0536727	0.0080128	-6.6983995	0.0000000
Fully-Connected Depth * Dropout	0.4033352	0.0081320	49.5987877	0.0000000
Fully-Connected Depth * Activation	-0.0472930	0.0080607	-5.8670850	0.0000000
Fully-Connected Depth * Initialization	-0.0151387	0.0081528	-1.8568678	0.0638158
BatchNorm * Max Pooling	-0.0223392	0.0080206	-2.7852341	0.0055164
BatchNorm * Dropout	-0.0997711	0.0081224	-12.2834195	0.0000000
BatchNorm * Activation	0.0061204	0.0080692	0.7584912	0.4484524
BatchNorm * Initialization	0.0500641	0.0081430	6.1481126	0.0000000
Max Pooling * Dropout	0.0228669	0.0079796	2.8656675	0.0043059
Max Pooling * Activation	-0.0207768	0.0080819	-2.5707988	0.0103844
Dropout * Activation	-0.0000816	0.0080170	-0.0101748	0.9918852
Dropout * Initialization	-0.0522850	0.0080978	-6.4567050	0.0000000
Activation * Initialization	-0.0355833	0.0080745	-4.4068956	0.0000124

Table 40. Stochastic Gradient Descent Full Second-Order Train Loss Model

Factor	Estimate	Std. Error	T-Value	P-Value
Intercept	-0.4707278	0.0069880	-67.3622440	0.0000000
Convolutional Width	-0.0476407	0.0069847	-6.8206982	0.0000000
Convolutional Depth	0.2410546	0.0069840	34.5152424	0.0000000
Fully-Connected Width	-0.0485713	0.0069840	-6.9546525	0.0000000
Fully-Connected Depth	0.5253125	0.0069651	75.4205756	0.0000000
BatchNorm	-0.0656380	0.0069885	-9.3922833	0.0000000
Max Pooling	0.0404799	0.0069792	5.8000453	0.0000000
Dropout	0.4276456	0.0069885	61.1927203	0.0000000
Activation	0.0002261	0.0069630	0.0324769	0.9741018
Initialization	-0.0258601	0.0069880	-3.7006374	0.0002335
Convolutional Width * Convolutional Depth	-0.0229500	0.0069792	-3.2883219	0.0010627
Convolutional Width * Fully-Connected Width	-0.0136425	0.0069792	-1.9547288	0.0510474
Convolutional Width * Fully-Connected Depth	0.0300061	0.0069630	4.3093872	0.0000189
Convolutional Width * BatchNorm	-0.0158710	0.0069850	-2.2721680	0.0234052
Convolutional Width * Max Pooling	-0.0203174	0.0069840	-2.9091335	0.0037493
Convolutional Width * Dropout	-0.0078974	0.0069850	-1.1306249	0.2586339
Convolutional Width * Activation	0.0201006	0.0069651	2.8859059	0.0040336
Convolutional Width * Initialization	0.0064041	0.0069847	0.9168677	0.3595552
Convolutional Depth * Fully-Connected Width	0.0375593	0.0069880	5.3748255	0.0000001
Convolutional Depth * Fully-Connected Depth	-0.1201896	0.0069610	-17.2660538	0.0000000
Convolutional Depth * BatchNorm	0.0235026	0.0069823	3.3660346	0.0008079
Convolutional Depth * Max Pooling	0.0664725	0.0069847	9.5168268	0.0000000
Convolutional Depth * Dropout	0.0640169	0.0069823	9.1684569	0.0000000
Convolutional Depth * Activation	0.0024684	0.0069579	0.3547533	0.7228906
Convolutional Depth * Initialization	0.0091662	0.0069840	1.3124625	0.1898317
Fully-Connected Width * Fully-Connected Depth	-0.0114658	0.0069610	-1.6471381	0.1000175
Fully-Connected Width * BatchNorm	-0.0055756	0.0069823	-0.7985390	0.4248521
Fully-Connected Width * Dropout	-0.0174388	0.0069823	-2.4975787	0.0127529
Fully-Connected Width * Activation	0.0048740	0.0069579	0.7004915	0.4838734
Fully-Connected Depth * BatchNorm	0.0757427	0.0069665	10.8724620	0.0000000
Fully-Connected Depth * Max Pooling	-0.0189341	0.0069579	-2.7212175	0.0066800
Fully-Connected Depth * Dropout	0.4260911	0.0069665	61.1631105	0.0000000
Fully-Connected Depth * Activation	-0.0072068	0.0069847	-1.0317995	0.3025534
Fully-Connected Depth * Initialization	-0.0034686	0.0069651	-0.4979917	0.6186599
BatchNorm * Max Pooling	0.0098395	0.0069774	1.4101909	0.1589661
BatchNorm * Dropout	-0.0005485	0.0069880	-0.0784947	0.9374589
BatchNorm * Activation	0.0056734	0.0069641	0.8146657	0.4155649
BatchNorm * Initialization	0.0054332	0.0069885	0.7774539	0.4371765
Max Pooling * Dropout	-0.0064440	0.0069774	-0.9235456	0.3560688
Max Pooling * Activation	0.0075312	0.0069610	1.0819073	0.2796987
Dropout * Activation	-0.0033488	0.0069641	-0.4808713	0.6307713
Dropout * Initialization	-0.0314709	0.0069885	-4.5032354	0.0000079
Activation * Initialization	-0.0285119	0.0069630	-4.0948008	0.0000476

Table 41. RMSProp Full Second-Order Train Loss Model

Factor	Estimate	Std. Error	T-Value	P-Value
Intercept	-0.6427019	0.0189388	-33.9356671	0.0000000
Convolutional Width	0.0123268	0.0189581	0.6502148	0.5157955
Convolutional Depth	0.5922416	0.0190126	31.1499181	0.0000000
Fully-Connected Width	-0.0071722	0.0189797	-0.3778852	0.7056458
Fully-Connected Depth	0.6687804	0.0189348	35.3202061	0.0000000
BatchNorm	-0.7749157	0.0190252	-40.7310973	0.0000000
Max Pooling	0.1685069	0.0189292	8.9019359	0.0000000
Dropout	0.2435785	0.0189141	12.8781662	0.0000000
Activation	0.0080455	0.0189171	0.4253033	0.6707639
Initialization	0.0567122	0.0189140	2.9984316	0.0028230
Convolutional Width * Convolutional Depth	-0.0057136	0.0189849	-0.3009534	0.7635514
Convolutional Width * Fully-Connected Width	-0.0406695	0.0189994	-2.1405649	0.0327005
Convolutional Width * Fully-Connected Depth	-0.0615506	0.0189588	-3.2465405	0.0012314
Convolutional Width * BatchNorm	-0.0563713	0.0189586	-2.9733941	0.0030602
Convolutional Width * Max Pooling	0.0425232	0.0189018	2.2496891	0.0248204
Convolutional Width * Dropout	0.0446709	0.0189022	2.3632630	0.0184239
Convolutional Width * Activation	-0.0813342	0.0189519	-4.2916093	0.0000206
Convolutional Width * Initialization	0.0764317	0.0188532	4.0540474	0.0000568
Convolutional Depth * Fully-Connected Width	0.0356105	0.0189527	1.8789176	0.0607269
Convolutional Depth * Fully-Connected Depth	-0.3918385	0.0190203	-20.6010478	0.0000000
Convolutional Depth * BatchNorm	-0.0696808	0.0189318	-3.6806175	0.0002530
Convolutional Depth * Max Pooling	0.0543520	0.0189202	2.8726926	0.0042098
Convolutional Depth * Dropout	-0.1095021	0.0188761	-5.8011079	0.0000000
Convolutional Depth * Activation	-0.0062483	0.0190139	-0.3286177	0.7425561
Convolutional Depth * Initialization	-0.0233022	0.0189235	-1.2313895	0.2186464
Fully-Connected Width * Fully-Connected Depth	0.0263346	0.0189672	1.3884275	0.1655079
Fully-Connected Width * BatchNorm	-0.0531417	0.0189421	-2.8054795	0.0051826
Fully-Connected Width * Dropout	-0.1324596	0.0189068	-7.0059199	0.0000000
Fully-Connected Width * Activation	-0.0421942	0.0189690	-2.2243714	0.0264845
Fully-Connected Depth * BatchNorm	0.1742889	0.0190285	9.1593700	0.0000000
Fully-Connected Depth * Max Pooling	-0.1308975	0.0189141	-6.9206222	0.0000000
Fully-Connected Depth * Dropout	0.3154055	0.0189076	16.6813965	0.0000000
Fully-Connected Depth * Activation	0.0377854	0.0189192	1.9971960	0.0462426
Fully-Connected Depth * Initialization	-0.0533017	0.0189155	-2.8178811	0.0049890
BatchNorm * Max Pooling	0.0788902	0.0189067	4.1726000	0.0000344
BatchNorm * Dropout	0.1823879	0.0188634	9.6689041	0.0000000
BatchNorm * Activation	-0.0410278	0.0189865	-2.1608886	0.0310881
BatchNorm * Initialization	0.0382559	0.0189006	2.0240598	0.0433944
Max Pooling * Dropout	-0.0049865	0.0189488	-0.2631585	0.7925163
Max Pooling * Activation	0.0256642	0.0189148	1.3568365	0.1753294
Dropout * Activation	-0.0076841	0.0188639	-0.4073422	0.6838979
Dropout * Initialization	-0.0155448	0.0189598	-0.8198811	0.4126005
Activation * Initialization	-0.0203469	0.0189240	-1.0751887	0.2827105

Table 42. Adam Full Second-Order Train Loss Model

Factor	Estimate	Std. Error	T-Value	P-Value
Intercept	-0.7084736	0.0062780	-112.8498065	0.0000000
Convolutional Width	-0.0497953	0.0062776	-7.9322291	0.0000000
Convolutional Depth	0.2035809	0.0062784	32.4257148	0.0000000
Fully-Connected Width	-0.0737695	0.0062773	-11.7517224	0.0000000
Fully-Connected Depth	0.5548355	0.0062779	88.3793948	0.0000000
BatchNorm	-0.0420673	0.0062780	-6.7007196	0.0000000
Max Pooling	0.1101507	0.0062784	17.5444469	0.0000000
Dropout	0.4145948	0.0062776	66.0436663	0.0000000
Activation	0.0095516	0.0062784	1.5213546	0.1286535
Initialization	-0.0369217	0.0062776	-5.8815040	0.0000000
Convolutional Width * Convolutional Depth	-0.0315129	0.0062779	-5.0196754	0.0000007
Convolutional Width * Fully-Connected Width	-0.0123830	0.0062779	-1.9724874	0.0489742
Convolutional Width * Fully-Connected Depth	0.0110954	0.0062773	1.7675260	0.0776057
Convolutional Width * BatchNorm	0.0023653	0.0062776	0.3767861	0.7064546
Convolutional Width * Max Pooling	-0.0060643	0.0062779	-0.9659865	0.3344075
Convolutional Width * Dropout	0.0055780	0.0062780	0.8884908	0.3746030
Convolutional Width * Activation	0.0046464	0.0062779	0.7401161	0.4594947
Convolutional Width * Initialization	0.0084213	0.0062771	1.3415967	0.1801917
Convolutional Depth * Fully-Connected Width	0.0148162	0.0062771	2.3603537	0.0185497
Convolutional Depth * Fully-Connected Depth	-0.0700008	0.0062776	-11.1509106	0.0000000
Convolutional Depth * BatchNorm	-0.0162048	0.0062784	-2.5810555	0.0100662
Convolutional Depth * Max Pooling	0.0913492	0.0062780	14.5506362	0.0000000
Convolutional Depth * Dropout	0.0097676	0.0062779	1.5558688	0.1202223
Convolutional Depth * Activation	-0.0079681	0.0062780	-1.2692073	0.2048180
Convolutional Depth * Initialization	0.0355576	0.0062779	5.6639440	0.0000000
Fully-Connected Width * Fully-Connected Depth	-0.0305270	0.0062776	-4.8628515	0.0000015
Fully-Connected Width * BatchNorm	0.0210603	0.0062773	3.3549777	0.0008395
Fully-Connected Width * Dropout	-0.0569070	0.0062779	-9.0646874	0.0000000
Fully-Connected Width * Activation	0.0095083	0.0062771	1.5147551	0.1303168
Fully-Connected Depth * BatchNorm	0.0332986	0.0062779	5.3041097	0.0000002
Fully-Connected Depth * Max Pooling	-0.0533208	0.0062776	-8.4938397	0.0000000
Fully-Connected Depth * Dropout	0.4229237	0.0062773	67.3731081	0.0000000
Fully-Connected Depth * Activation	-0.0154831	0.0062776	-2.4664043	0.0139027
Fully-Connected Depth * Initialization	-0.0028735	0.0062784	-0.4576891	0.6473276
BatchNorm * Max Pooling	-0.0051481	0.0062784	-0.8199767	0.4125277
BatchNorm * Dropout	-0.0499719	0.0062776	-7.9603687	0.0000000
BatchNorm * Activation	-0.0057916	0.0062784	-0.9224661	0.3566251
BatchNorm * Initialization	0.0167759	0.0062776	2.6723434	0.0077198
Max Pooling * Dropout	0.0118227	0.0062779	1.8832323	0.0601117
Max Pooling * Activation	0.0058603	0.0062780	0.9334698	0.3509214
Dropout * Activation	0.0022160	0.0062779	0.3529866	0.7242120
Dropout * Initialization	-0.0265923	0.0062771	-4.2363981	0.0000260
Activation * Initialization	-0.0508309	0.0062779	-8.0968292	0.0000000

Table 43. AdaGrad Full Second-Order Train Loss Model

Factor	Estimate	Std. Error	T-Value	P-Value
Intercept	-0.5497942	0.0071598	-76.7888707	0.0000000
Convolutional Width	-0.0791752	0.0071598	-11.0582667	0.0000000
Convolutional Depth	0.2066233	0.0071598	28.8587475	0.0000000
Fully-Connected Width	-0.0590886	0.0071598	-8.2528157	0.0000000
Fully-Connected Depth	0.5668066	0.0071598	79.1649653	0.0000000
BatchNorm	-0.0521025	0.0071598	-7.2770784	0.0000000
Max Pooling	0.1113864	0.0071591	15.5587831	0.0000000
Dropout	0.4274597	0.0071591	59.7088260	0.0000000
Activation	0.0189606	0.0071598	2.6481901	0.0082867
Initialization	-0.0487285	0.0071591	-6.8065446	0.0000000
Convolutional Width * Convolutional Depth	-0.0607438	0.0071598	-8.4839867	0.0000000
Convolutional Width * Fully-Connected Width	-0.0145625	0.0071598	-2.0339210	0.0423608
Convolutional Width * Fully-Connected Depth	0.0497409	0.0071598	6.9472266	0.0000000
Convolutional Width * BatchNorm	-0.0056984	0.0071598	-0.7958915	0.4263826
Convolutional Width * Max Pooling	0.0061515	0.0071591	0.8592550	0.3905132
Convolutional Width * Dropout	0.0085939	0.0071591	1.2004190	0.2304093
Convolutional Width * Activation	0.0191286	0.0071598	2.6716574	0.0077348
Convolutional Width * Initialization	0.0289838	0.0071591	4.0485484	0.0000577
Convolutional Depth * Fully-Connected Width	0.0421728	0.0071598	5.8902099	0.0000000
Convolutional Depth * Fully-Connected Depth	-0.1038624	0.0071598	-14.5062887	0.0000000
Convolutional Depth * BatchNorm	0.0634143	0.0071598	8.8569706	0.0000000
Convolutional Depth * Max Pooling	0.1152674	0.0071591	16.1008855	0.0000000
Convolutional Depth * Dropout	0.0354406	0.0071591	4.9504501	0.0000009
Convolutional Depth * Activation	-0.0032567	0.0071598	-0.4548513	0.6493665
Convolutional Depth * Initialization	0.0301091	0.0071591	4.2057298	0.0000296
Fully-Connected Width * Fully-Connected Depth	-0.0006657	0.0071598	-0.0929806	0.9259474
Fully-Connected Width * BatchNorm	0.0004764	0.0071598	0.0665403	0.9469680
Fully-Connected Width * Dropout	-0.0245346	0.0071591	-3.4270654	0.0006482
Fully-Connected Width * Activation	0.0310282	0.0071598	4.3336564	0.0000170
Fully-Connected Depth * BatchNorm	0.0842867	0.0071598	11.7721836	0.0000000
Fully-Connected Depth * Max Pooling	-0.0519507	0.0071591	-7.2566325	0.0000000
Fully-Connected Depth * Dropout	0.4380071	0.0071591	61.1821237	0.0000000
Fully-Connected Depth * Activation	-0.0265624	0.0071598	-3.7099326	0.0002249
Fully-Connected Depth * Initialization	-0.0007268	0.0071591	-0.1015185	0.9191699
BatchNorm * Max Pooling	0.0127827	0.0071591	1.7855258	0.0746372
BatchNorm * Dropout	0.0208172	0.0071591	2.9078125	0.0037625
BatchNorm * Activation	-0.0171938	0.0071598	-2.4014246	0.0166082
BatchNorm * Initialization	0.0214139	0.0071591	2.9911562	0.0028833
Max Pooling * Dropout	-0.0097836	0.0071598	-1.3664595	0.1722622
Max Pooling * Activation	-0.0006383	0.0071591	-0.0891561	0.9289850
Dropout * Activation	0.0070072	0.0071591	0.9787813	0.3280483
Dropout * Initialization	-0.0289810	0.0071598	-4.0477361	0.0000579
Activation * Initialization	-0.0574079	0.0071591	-8.0189021	0.0000000

7.7 CIFAR-10 Full Test Loss Models

Table 44. Stochastic Gradient Descent with Nesterov's Accelerated Gradient Full Second-Order Test Loss Model

Factor	Estimate	Std. Error	T-Value	P-Value
Intercept	3.6575193	0.0003385	10804.8302504	0.0000000
Convolutional Width	-0.0025941	0.0003382	-7.6693981	0.0000000
Convolutional Depth	-0.0009299	0.0003382	-2.7496140	0.0061397
Fully-Connected Width	0.0024485	0.0003375	7.2544953	0.0000000
Fully-Connected Depth	-0.0018713	0.0003386	-5.5264934	0.0000000
BatchNorm	-0.0031662	0.0003385	-9.3534304	0.0000000
Max Pooling	-0.0017151	0.0003350	-5.1201605	0.0000004
Dropout	0.0140543	0.0003391	41.4411380	0.0000000
Activation	-0.0019099	0.0003388	-5.6379068	0.0000000
Initialization	-0.0000837	0.0003390	-0.2469900	0.8049973
Convolutional Width * Convolutional Depth	-0.0010881	0.0003355	-3.2430091	0.0012459
Convolutional Width * Fully-Connected Width	-0.0002864	0.0003352	-0.8542236	0.3933095
Convolutional Width * Fully-Connected Depth	-0.0010117	0.0003379	-2.9944547	0.0028584
Convolutional Width * BatchNorm	-0.0003649	0.0003382	-1.0787311	0.2811251
Convolutional Width * Max Pooling	-0.0018003	0.0003386	-5.3174877	0.0000001
Convolutional Width * Dropout	-0.0003338	0.0003384	-0.9862607	0.3243880
Convolutional Width * Activation	-0.0004298	0.0003387	-1.2691431	0.2048639
Convolutional Width * Initialization	-0.0006819	0.0003377	-2.0196273	0.0438497
Convolutional Depth * Fully-Connected Width	0.0000802	0.0003385	0.2369811	0.8127493
Convolutional Depth * Fully-Connected Depth	0.0059650	0.0003378	17.6587697	0.0000000
Convolutional Depth * BatchNorm	0.0007506	0.0003382	2.2195669	0.0268075
Convolutional Depth * Max Pooling	0.0031785	0.0003384	9.3922172	0.0000000
Convolutional Depth * Dropout	0.0040170	0.0003386	11.8645300	0.0000000
Convolutional Depth * Activation	-0.0010876	0.0003358	-3.2387182	0.0012645
Convolutional Depth * Initialization	-0.0019752	0.0003378	-5.8480189	0.0000000
Fully-Connected Width * Fully-Connected Depth	0.0001962	0.0003385	0.5796146	0.5623841
Fully-Connected Width * BatchNorm	-0.0018147	0.0003375	-5.3767093	0.0000001
Fully-Connected Width * Dropout	0.0005473	0.0003378	1.6205325	0.1056239
Fully-Connected Width * Activation	-0.0011230	0.0003355	-3.3472703	0.0008653
Fully-Connected Depth * BatchNorm	-0.0000174	0.0003386	-0.0514898	0.9589517
Fully-Connected Depth * Max Pooling	0.0014851	0.0003347	4.4364704	0.0000108
Fully-Connected Depth * Dropout	0.0052133	0.0003386	15.3949183	0.0000000
Fully-Connected Depth * Activation	-0.0007473	0.0003374	-2.2150276	0.0271191
Fully-Connected Depth * Initialization	0.0004836	0.0003388	1.4275766	0.1539148
BatchNorm * Max Pooling	-0.0000565	0.0003350	-0.1686218	0.8661489
BatchNorm * Dropout	0.0003173	0.0003391	0.9355593	0.3498626
BatchNorm * Activation	0.0013962	0.0003388	4.1215855	0.0000427
BatchNorm * Initialization	-0.0004506	0.0003390	-1.3293975	0.1842034
Max Pooling * Dropout	0.0014247	0.0003355	4.2462978	0.0000250
Max Pooling * Activation	-0.0000434	0.0003381	-0.1284394	0.8978427
Dropout * Activation	-0.0000484	0.0003381	-0.1432301	0.8861548
Dropout * Initialization	0.0007586	0.0003385	2.2411539	0.0253675
Activation * Initialization	0.0006474	0.0003374	1.9189537	0.0554468

Table 45. Stochastic Gradient Descent with Momentum Full Second-Order Test Loss Model

Factor	Estimate	Std. Error	T-Value	P-Value
Intercept	3.6569438	0.0003193	11452.6908611	0.0000000
Convolutional Width	-0.0020813	0.0003191	-6.5220044	0.0000000
Convolutional Depth	-0.0003105	0.0003188	-0.9741003	0.3303824
Fully-Connected Width	0.0020962	0.0003181	6.5890549	0.0000000
Fully-Connected Depth	-0.0007829	0.0003187	-2.4567174	0.0142912
BatchNorm	-0.0029490	0.0003193	-9.2356338	0.0000000
Max Pooling	-0.0011929	0.0003177	-3.7548342	0.0001896
Dropout	0.0139872	0.0003200	43.7051918	0.0000000
Activation	-0.0011164	0.0003201	-3.4877707	0.0005211
Initialization	-0.0001654	0.0003194	-0.5178737	0.6047291
Convolutional Width * Convolutional Depth	-0.0010946	0.0003177	-3.4449958	0.0006092
Convolutional Width * Fully-Connected Width	-0.0006478	0.0003170	-2.0431704	0.0414531
Convolutional Width * Fully-Connected Depth	-0.0008789	0.0003191	-2.7540842	0.0060563
Convolutional Width * BatchNorm	-0.0004751	0.0003191	-1.4887477	0.1370566
Convolutional Width * Max Pooling	-0.0008161	0.0003195	-2.5545510	0.0108676
Convolutional Width * Dropout	-0.0001883	0.0003193	-0.5898281	0.5555184
Convolutional Width * Activation	-0.0002061	0.0003193	-0.6454225	0.5188895
Convolutional Width * Initialization	-0.0004917	0.0003184	-1.5445607	0.1229571
Convolutional Depth * Fully-Connected Width	0.0003238	0.0003189	1.0152276	0.3103889
Convolutional Depth * Fully-Connected Depth	0.0056151	0.0003177	17.6730912	0.0000000
Convolutional Depth * BatchNorm	0.0008764	0.0003188	2.7491795	0.0061466
Convolutional Depth * Max Pooling	0.0028091	0.0003193	8.7980069	0.0000000
Convolutional Depth * Dropout	0.0023239	0.0003195	7.2743476	0.0000000
Convolutional Depth * Activation	-0.0009846	0.0003185	-3.0917021	0.0020784
Convolutional Depth * Initialization	-0.0017639	0.0003185	-5.5373037	0.0000000
Fully-Connected Width * Fully-Connected Depth	-0.0004672	0.0003186	-1.4664174	0.1430359
Fully-Connected Width * BatchNorm	-0.0016069	0.0003181	-5.0509303	0.0000006
Fully-Connected Width * Dropout	0.0004510	0.0003185	1.4159075	0.1572991
Fully-Connected Width * Activation	-0.0011844	0.0003177	-3.7276337	0.0002108
Fully-Connected Depth * BatchNorm	-0.0004761	0.0003187	-1.4941332	0.1356439
Fully-Connected Depth * Max Pooling	0.0013845	0.0003173	4.3639791	0.0000149
Fully-Connected Depth * Dropout	0.0055115	0.0003185	17.3031004	0.0000000
Fully-Connected Depth * Activation	-0.0005819	0.0003181	-1.8291641	0.0678498
Fully-Connected Depth * Initialization	0.0000212	0.0003192	0.0665562	0.9469562
BatchNorm * Max Pooling	-0.0001798	0.0003177	-0.5660892	0.5715357
BatchNorm * Dropout	0.0001220	0.0003200	0.3811813	0.7031979
BatchNorm * Activation	0.0005110	0.0003201	1.5964715	0.1108874
BatchNorm * Initialization	-0.0002282	0.0003194	-0.7146743	0.4750763
Max Pooling * Dropout	0.0016007	0.0003177	5.0378436	0.0000006
Max Pooling * Activation	0.0003718	0.0003185	1.1674493	0.2434728
Dropout * Activation	0.0000486	0.0003193	0.1522741	0.8790197
Dropout * Initialization	0.0005701	0.0003189	1.7874131	0.0743536
Activation * Initialization	-0.0000165	0.0003187	-0.0516424	0.9588301

Table 46. Stochastic Gradient Descent Full Second-Order Test Loss Model

Factor	Estimate	Std. Error	T-Value	P-Value
Intercept	1.9846263	2.21e-05	89871.9675875	0.0000000
Convolutional Width	-0.0001908	2.21e-05	-8.6389096	0.0000000
Convolutional Depth	0.0008426	2.21e-05	38.1495215	0.0000000
Fully-Connected Width	0.0000890	2.21e-05	4.0307634	0.0000622
Fully-Connected Depth	0.0008705	2.21e-05	39.4237999	0.0000000
BatchNorm	-0.0001200	2.21e-05	-5.4272448	0.0000001
Max Pooling	0.0001390	2.21e-05	6.2919652	0.0000000
Dropout	0.0011943	2.21e-05	54.0838966	0.0000000
Activation	-0.0000810	2.21e-05	-3.6623144	0.0002705
Initialization	-0.0000667	2.21e-05	-3.0207205	0.0026217
Convolutional Width * Convolutional Depth	-0.0000987	2.21e-05	-4.4696683	0.0000093
Convolutional Width * Fully-Connected Width	0.0000151	2.21e-05	0.6835780	0.4944875
Convolutional Width * Fully-Connected Depth	-0.0000080	2.21e-05	-0.3607643	0.7183940
Convolutional Width * BatchNorm	0.0000015	2.21e-05	0.0686170	0.9453158
Convolutional Width * Max Pooling	0.0000772	2.21e-05	3.4907358	0.0005146
Convolutional Width * Dropout	0.0000504	2.21e-05	2.2834132	0.0227309
Convolutional Width * Activation	0.0000474	2.21e-05	2.1466269	0.0321959
Convolutional Width * Initialization	0.0000028	2.21e-05	0.1281307	0.8980855
Convolutional Depth * Fully-Connected Width	0.0000556	2.21e-05	2.5193575	0.0119978
Convolutional Depth * Fully-Connected Depth	-0.0003559	2.21e-05	-16.1163498	0.0000000
Convolutional Depth * BatchNorm	-0.0001962	2.21e-05	-8.8712990	0.0000000
Convolutional Depth * Max Pooling	0.0003916	2.21e-05	17.7233943	0.0000000
Convolutional Depth * Dropout	0.0000228	2.21e-05	1.0301121	0.3033441
Convolutional Depth * Activation	-0.0000588	2.21e-05	-2.6589441	0.0080334
Convolutional Depth * Initialization	-0.0000607	2.21e-05	-2.7470724	0.0061812
Fully-Connected Width * Fully-Connected Depth	0.0000220	2.21e-05	0.9937778	0.3207043
Fully-Connected Width * BatchNorm	-0.0000730	2.21e-05	-3.3038215	0.0010066
Fully-Connected Width * Dropout	0.0000036	2.21e-05	0.1635349	0.8701486
Fully-Connected Width * Activation	-0.0000378	2.21e-05	-1.7120169	0.0873750
Fully-Connected Depth * BatchNorm	-0.0000768	2.21e-05	-3.4813275	0.0005327
Fully-Connected Depth * Max Pooling	-0.0000960	2.21e-05	-4.3477959	0.0000160
Fully-Connected Depth * Dropout	0.0002958	2.21e-05	13.3950268	0.0000000
Fully-Connected Depth * Activation	-0.0000117	2.21e-05	-0.5326468	0.5944618
Fully-Connected Depth * Initialization	-0.0000151	2.21e-05	-0.6850448	0.4935621
BatchNorm * Max Pooling	-0.0000202	2.21e-05	-0.9128218	0.3616779
BatchNorm * Dropout	-0.0000043	2.21e-05	-0.1958656	0.8447772
BatchNorm * Activation	0.0000430	2.21e-05	1.9456658	0.0521296
BatchNorm * Initialization	-0.0000218	2.21e-05	-0.9851844	0.3249035
Max Pooling * Dropout	0.0001639	2.21e-05	7.4221133	0.0000000
Max Pooling * Activation	0.0000441	2.21e-05	1.9967786	0.0462697
Dropout * Activation	-0.0000572	2.21e-05	-2.5869036	0.0099024
Dropout * Initialization	-0.0000365	2.21e-05	-1.6530108	0.0988160
Activation * Initialization	-0.0001290	2.21e-05	-5.8420775	0.0000000

Table 47. RMSProp Full Second-Order Test Loss Model

Factor	Estimate	Std. Error	T-Value	P-Value
Intercept	1.9860877	3.48e-05	57126.9403500	0.0000000
Convolutional Width	-0.0000570	3.48e-05	-1.6375105	0.1020224
Convolutional Depth	-0.0001596	3.48e-05	-4.5917085	0.0000053
Fully-Connected Width	0.0002589	3.45e-05	7.5027587	0.0000000
Fully-Connected Depth	-0.0005437	3.45e-05	-15.7479252	0.0000000
BatchNorm	-0.0007113	3.47e-05	-20.4785156	0.0000000
Max Pooling	0.0000114	3.45e-05	0.3297416	0.7417047
Dropout	0.0004398	3.45e-05	12.7591167	0.0000000
Activation	-0.0001105	3.48e-05	-3.1749521	0.0015716
Initialization	-0.0001241	3.42e-05	-3.6240603	0.0003134
Convolutional Width * Convolutional Depth	0.0000072	3.48e-05	0.2070930	0.8360040
Convolutional Width * Fully-Connected Width	0.0000927	3.45e-05	2.6852387	0.0074383
Convolutional Width * Fully-Connected Depth	-0.0000281	3.45e-05	-0.8145571	0.4156332
Convolutional Width * BatchNorm	-0.0003509	3.47e-05	-10.1047270	0.0000000
Convolutional Width * Max Pooling	0.0002373	3.45e-05	6.8771381	0.0000000
Convolutional Width * Dropout	0.0001287	3.45e-05	3.7324395	0.0002068
Convolutional Width * Activation	-0.0001276	3.48e-05	-3.6709769	0.0002621
Convolutional Width * Initialization	-0.0000532	3.42e-05	-1.5538846	0.1207133
Convolutional Depth * Fully-Connected Width	-0.0000441	3.45e-05	-1.2784342	0.2015664
Convolutional Depth * Fully-Connected Depth	0.0003134	3.45e-05	9.0802450	0.0000000
Convolutional Depth * BatchNorm	-0.0007610	3.47e-05	-21.9108769	0.0000000
Convolutional Depth * Max Pooling	0.0001887	3.45e-05	5.4693204	0.0000001
Convolutional Depth * Dropout	0.0000513	3.45e-05	1.4868770	0.1375468
Convolutional Depth * Activation	-0.0002772	3.48e-05	-7.9709166	0.0000000
Convolutional Depth * Initialization	-0.0001075	3.42e-05	-3.1405127	0.0017654
Fully-Connected Width * Fully-Connected Depth	-0.0003088	3.48e-05	-8.8814978	0.0000000
Fully-Connected Width * BatchNorm	-0.0000038	3.45e-05	-0.1104363	0.9120985
Fully-Connected Width * Dropout	-0.0000063	3.42e-05	-0.1848086	0.8534385
Fully-Connected Width * Activation	-0.0000636	3.45e-05	-1.8432058	0.0657678
Fully-Connected Depth * BatchNorm	-0.0005262	3.45e-05	-15.2627209	0.0000000
Fully-Connected Depth * Max Pooling	0.0001963	3.42e-05	5.7358329	0.0000000
Fully-Connected Depth * Dropout	0.0001607	3.42e-05	4.6974609	0.0000032
Fully-Connected Depth * Activation	0.0000388	3.45e-05	1.1245677	0.2611999
Fully-Connected Depth * Initialization	0.0000336	3.45e-05	0.9730889	0.3308820
BatchNorm * Max Pooling	0.0000138	3.45e-05	0.4016256	0.6880956
BatchNorm * Dropout	0.0003787	3.45e-05	10.9774225	0.0000000
BatchNorm * Activation	0.0000347	3.47e-05	0.9985866	0.3183778
BatchNorm * Initialization	0.0001201	3.42e-05	3.5096985	0.0004806
Max Pooling * Dropout	-0.0002465	3.47e-05	-7.1030789	0.0000000
Max Pooling * Activation	-0.0000088	3.45e-05	-0.2564767	0.7976663
Dropout * Activation	0.0000209	3.45e-05	0.6056463	0.5449674
Dropout * Initialization	0.0000736	3.45e-05	2.1357519	0.0330839
Activation * Initialization	0.0001826	3.42e-05	5.3335700	0.0000001

Table 48. Adam Full Second-Order Test Loss Model

Factor	Estimate	Std. Error	T-Value	P-Value
Intercept	1.9850577	3.07e-05	64573.8062166	0.0000000
Convolutional Width	-0.0002054	3.07e-05	-6.6829840	0.0000000
Convolutional Depth	0.0002167	3.07e-05	7.0476150	0.0000000
Fully-Connected Width	0.0001113	3.07e-05	3.6209343	0.0003164
Fully-Connected Depth	-0.0002159	3.07e-05	-7.0213226	0.0000000
BatchNorm	-0.0000231	3.07e-05	-0.7524896	0.4520287
Max Pooling	-0.0003210	3.07e-05	-10.4423243	0.0000000
Dropout	0.0015227	3.07e-05	49.5221343	0.0000000
Activation	-0.0001089	3.07e-05	-3.5447736	0.0004211
Initialization	-0.0000704	3.07e-05	-2.2906914	0.0223000
Convolutional Width * Convolutional Depth	-0.0001915	3.07e-05	-6.2319928	0.0000000
Convolutional Width * Fully-Connected Width	0.0000250	3.07e-05	0.8124231	0.4168459
Convolutional Width * Fully-Connected Depth	0.0000639	3.07e-05	2.0774261	0.0381537
Convolutional Width * BatchNorm	-0.0000992	3.07e-05	-3.2271962	0.0013128
Convolutional Width * Max Pooling	0.0000321	3.07e-05	1.0433680	0.2971652
Convolutional Width * Dropout	-0.0000736	3.07e-05	-2.3950735	0.0168986
Convolutional Width * Activation	-0.0000469	3.07e-05	-1.5256886	0.1275731
Convolutional Width * Initialization	-0.0001012	3.07e-05	-3.2941082	0.0010408
Convolutional Depth * Fully-Connected Width	-0.0000052	3.07e-05	-0.1676700	0.8668950
Convolutional Depth * Fully-Connected Depth	-0.0001011	3.07e-05	-3.2883773	0.0010619
Convolutional Depth * BatchNorm	-0.0000546	3.07e-05	-1.7746164	0.0764285
Convolutional Depth * Max Pooling	0.0001986	3.07e-05	6.4611664	0.0000000
Convolutional Depth * Dropout	0.0004304	3.07e-05	13.9972637	0.0000000
Convolutional Depth * Activation	0.0000516	3.07e-05	1.6775549	0.0939140
Convolutional Depth * Initialization	-0.0002309	3.07e-05	-7.5114049	0.0000000
Fully-Connected Width * Fully-Connected Depth	-0.0000568	3.07e-05	-1.8486631	0.0649597
Fully-Connected Width * BatchNorm	-0.0000227	3.07e-05	-0.7371571	0.4612924
Fully-Connected Width * Dropout	0.0000852	3.07e-05	2.7719909	0.0057307
Fully-Connected Width * Activation	0.0000293	3.07e-05	0.9533994	0.3407416
Fully-Connected Depth * BatchNorm	-0.0003227	3.07e-05	-10.4940656	0.0000000
Fully-Connected Depth * Max Pooling	0.0002785	3.07e-05	9.0611674	0.0000000
Fully-Connected Depth * Dropout	0.0003194	3.07e-05	10.3907535	0.0000000
Fully-Connected Depth * Activation	0.0001542	3.07e-05	5.0181160	0.0000007
Fully-Connected Depth * Initialization	0.0000641	3.07e-05	2.0857134	0.0373934
BatchNorm * Max Pooling	0.0000313	3.07e-05	1.0198093	0.3081977
BatchNorm * Dropout	-0.0001194	3.07e-05	-3.8828383	0.0001138
BatchNorm * Activation	0.0001680	3.07e-05	5.4651215	0.0000001
BatchNorm * Initialization	0.0000047	3.07e-05	0.1542277	0.8774780
Max Pooling * Dropout	0.0001883	3.07e-05	6.1284996	0.0000000
Max Pooling * Activation	0.0001031	3.07e-05	3.3540432	0.0008426
Dropout * Activation	-0.0000007	3.07e-05	-0.0231551	0.9815336
Dropout * Initialization	0.0000702	3.07e-05	2.2842835	0.0226759
Activation * Initialization	0.0001035	3.07e-05	3.3692511	0.0007982

Table 49. AdaGrad Full Second-Order Test Loss Model

Factor	Estimate	Std. Error	T-Value	P-Value
Intercept	1.9846414	3.09e-05	64209.6777683	0.0000000
Convolutional Width	-0.0001669	3.09e-05	-5.3988700	0.0000001
Convolutional Depth	0.0011243	3.09e-05	36.3760919	0.0000000
Fully-Connected Width	0.0000947	3.09e-05	3.0651366	0.0022653
Fully-Connected Depth	0.0005978	3.09e-05	19.3400533	0.0000000
BatchNorm	0.0001540	3.09e-05	4.9823682	0.0000008
Max Pooling	-0.0001516	3.09e-05	-4.9038889	0.0000012
Dropout	0.0013969	3.09e-05	45.1917266	0.0000000
Activation	-0.0000598	3.09e-05	-1.9343819	0.0534972
Initialization	-0.0001218	3.09e-05	-3.9397612	0.0000903
Convolutional Width * Convolutional Depth	0.0000340	3.09e-05	1.0991539	0.2721054
Convolutional Width * Fully-Connected Width	0.0000572	3.09e-05	1.8514411	0.0645567
Convolutional Width * Fully-Connected Depth	0.0000379	3.09e-05	1.2263328	0.2205148
Convolutional Width * BatchNorm	-0.0000903	3.09e-05	-2.9202466	0.0036180
Convolutional Width * Max Pooling	-0.0000528	3.09e-05	-1.7073695	0.0882280
Convolutional Width * Dropout	0.0001215	3.09e-05	3.9304024	0.0000938
Convolutional Width * Activation	-0.0000931	3.09e-05	-3.0120303	0.0026952
Convolutional Width * Initialization	-0.0001446	3.09e-05	-4.6798104	0.0000035
Convolutional Depth * Fully-Connected Width	0.0001031	3.09e-05	3.3364227	0.0008967
Convolutional Depth * Fully-Connected Depth	-0.0007680	3.09e-05	-24.8459094	0.0000000
Convolutional Depth * BatchNorm	-0.0000554	3.09e-05	-1.7916500	0.0736513
Convolutional Depth * Max Pooling	0.0001387	3.09e-05	4.4881608	0.0000085
Convolutional Depth * Dropout	0.0002524	3.09e-05	8.1653639	0.0000000
Convolutional Depth * Activation	-0.0000421	3.09e-05	-1.3615092	0.1738217
Convolutional Depth * Initialization	-0.0000719	3.09e-05	-2.3259293	0.0203271
Fully-Connected Width * Fully-Connected Depth	-0.0001351	3.09e-05	-4.3720697	0.0000143
Fully-Connected Width * BatchNorm	-0.0000242	3.09e-05	-0.7831196	0.4338404
Fully-Connected Width * Dropout	0.0000276	3.09e-05	0.8935713	0.3718800
Fully-Connected Width * Activation	-0.0000951	3.09e-05	-3.0779802	0.0021713
Fully-Connected Depth * BatchNorm	0.0000468	3.09e-05	1.5149208	0.1302756
Fully-Connected Depth * Max Pooling	0.0001408	3.09e-05	4.5548841	0.0000063
Fully-Connected Depth * Dropout	0.0002101	3.09e-05	6.7966774	0.0000000
Fully-Connected Depth * Activation	0.0001096	3.09e-05	3.5475319	0.0004167
Fully-Connected Depth * Initialization	0.0000334	3.09e-05	1.0813413	0.2799439
BatchNorm * Max Pooling	0.0000462	3.09e-05	1.4949376	0.1354131
BatchNorm * Dropout	-0.0002646	3.09e-05	-8.5612936	0.0000000
BatchNorm * Activation	0.0000348	3.09e-05	1.1264896	0.2603715
BatchNorm * Initialization	0.0000246	3.09e-05	0.7942546	0.4273353
Max Pooling * Dropout	0.0000371	3.09e-05	1.2001907	0.2304999
Max Pooling * Activation	0.0000516	3.09e-05	1.6691808	0.0955601
Dropout * Activation	-0.0001312	3.09e-05	-4.2472190	0.0000248
Dropout * Initialization	-0.0000064	3.09e-05	-0.2070695	0.8360200
Activation * Initialization	0.0000355	3.09e-05	1.1496954	0.2506896

7.8 FashionMNIST Full Test Loss Models

Table 50. Stochastic Gradient Descent with Nesterov's Accelerated Gradient Full Second-Order Test Loss Model

Factor	Estimate	Std. Error	T-Value	P-Value
Intercept	1.9743858	9.21e-05	21448.4204503	0.0000000
Convolutional Width	-0.0006468	9.02e-05	-7.1698213	0.0000000
Convolutional Depth	0.0013973	9.04e-05	15.4534280	0.0000000
Fully-Connected Width	0.0002988	9.01e-05	3.3175475	0.0009634
Fully-Connected Depth	0.0026210	9.21e-05	28.4674976	0.0000000
BatchNorm	-0.0004337	9.17e-05	-4.7267067	0.0000029
Max Pooling	0.0008212	8.99e-05	9.1376283	0.0000000
Dropout	0.0063144	9.08e-05	69.5212828	0.0000000
Activation	-0.0001271	9.05e-05	-1.4048904	0.1605749
Initialization	-0.0002591	9.17e-05	-2.8252517	0.0048826
Convolutional Width * Convolutional Depth	-0.0004228	8.91e-05	-4.7444034	0.0000026
Convolutional Width * Fully-Connected Width	-0.0001644	8.90e-05	-1.8458527	0.0654093
Convolutional Width * Fully-Connected Depth	0.0000183	9.02e-05	0.2029219	0.8392653
Convolutional Width * BatchNorm	0.0002065	9.00e-05	2.2935120	0.0221657
Convolutional Width * Max Pooling	-0.0002815	8.98e-05	-3.1342474	0.0018077
Convolutional Width * Dropout	-0.0005264	9.06e-05	-5.8097338	0.0000000
Convolutional Width * Activation	-0.0000289	9.10e-05	-0.3179473	0.7506363
Convolutional Width * Initialization	0.0001470	8.98e-05	1.6374139	0.1020718
Convolutional Depth * Fully-Connected Width	-0.0000842	9.09e-05	-0.9264074	0.3546093
Convolutional Depth * Fully-Connected Depth	0.0001994	9.06e-05	2.2010241	0.0281170
Convolutional Depth * BatchNorm	-0.0002938	9.03e-05	-3.2542882	0.0012011
Convolutional Depth * Max Pooling	0.0015457	9.09e-05	17.0129933	0.0000000
Convolutional Depth * Dropout	0.0007431	8.97e-05	8.2852337	0.0000000
Convolutional Depth * Activation	-0.0002370	8.92e-05	-2.6584041	0.0080620
Convolutional Depth * Initialization	-0.0005979	9.07e-05	-6.5938396	0.0000000
Fully-Connected Width * Fully-Connected Depth	0.0000447	9.02e-05	0.4953426	0.6205409
Fully-Connected Width * BatchNorm	-0.0002427	8.99e-05	-2.7009299	0.0071112
Fully-Connected Width * Dropout	0.0002837	9.04e-05	3.1375958	0.0017875
Fully-Connected Width * Activation	0.0001606	8.91e-05	1.8028751	0.0719128
Fully-Connected Depth * BatchNorm	-0.0004740	9.19e-05	-5.1561035	0.0000003
Fully-Connected Depth * Max Pooling	-0.0001326	8.97e-05	-1.4781307	0.1399011
Fully-Connected Depth * Dropout	0.0026308	9.08e-05	28.9741262	0.0000000
Fully-Connected Depth * Activation	-0.0003730	9.04e-05	-4.1269086	0.0000420
Fully-Connected Depth * Initialization	-0.0000635	9.17e-05	-0.6928478	0.4886750
BatchNorm * Max Pooling	-0.0004979	8.96e-05	-5.5557604	0.0000000
BatchNorm * Dropout	-0.0007652	9.11e-05	-8.4000435	0.0000000
BatchNorm * Activation	0.0004250	9.04e-05	4.7023426	0.0000032
BatchNorm * Initialization	0.0003355	9.16e-05	3.6629796	0.0002715
Max Pooling * Dropout	0.0003926	8.90e-05	4.4137704	0.0000121
Max Pooling * Activation	-0.0004112	9.06e-05	-4.5364729	0.0000069
Dropout * Activation	-0.0002274	8.95e-05	-2.5412107	0.0112992
Dropout * Initialization	0.0001351	9.08e-05	1.4869678	0.1375521
Activation * Initialization	-0.0000012	9.06e-05	-0.0137047	0.9890701

Table 51. Stochastic Gradient Descent with Momentum Full Second-Order Test Loss Model

Factor	Estimate	Std. Error	T-Value	P-Value
Intercept	1.9742058	8.55e-05	23089.7372631	0.0000000
Convolutional Width	-0.0006756	8.39e-05	-8.0479203	0.0000000
Convolutional Depth	0.0015094	8.43e-05	17.9142571	0.0000000
Fully-Connected Width	0.0002727	8.39e-05	3.2482009	0.0012258
Fully-Connected Depth	0.0027712	8.56e-05	32.3892635	0.0000000
BatchNorm	-0.0003646	8.55e-05	-4.2656143	0.0000231
Max Pooling	0.0008458	8.36e-05	10.1127112	0.0000000
Dropout	0.0061912	8.47e-05	73.1201250	0.0000000
Activation	-0.0000894	8.42e-05	-1.0619788	0.2886697
Initialization	-0.0003039	8.49e-05	-3.5788012	0.0003729
Convolutional Width * Convolutional Depth	-0.0003862	8.32e-05	-4.6409582	0.0000043
Convolutional Width * Fully-Connected Width	-0.0001856	8.31e-05	-2.2334871	0.0258822
Convolutional Width * Fully-Connected Depth	0.0000589	8.39e-05	0.7012975	0.4833875
Convolutional Width * BatchNorm	0.0001614	8.38e-05	1.9265801	0.0545008
Convolutional Width * Max Pooling	-0.0002467	8.37e-05	-2.9474668	0.0033276
Convolutional Width * Dropout	-0.0005320	8.42e-05	-6.3219398	0.0000000
Convolutional Width * Activation	-0.0000983	8.45e-05	-1.1634537	0.2451049
Convolutional Width * Initialization	0.0000615	8.36e-05	0.7357015	0.4621980
Convolutional Depth * Fully-Connected Width	-0.0001808	8.45e-05	-2.1401416	0.0327432
Convolutional Depth * Fully-Connected Depth	0.0002215	8.43e-05	2.6274530	0.0088212
Convolutional Depth * BatchNorm	-0.0004353	8.41e-05	-5.1757269	0.0000003
Convolutional Depth * Max Pooling	0.0015134	8.43e-05	17.9594972	0.0000000
Convolutional Depth * Dropout	0.0006027	8.35e-05	7.2163089	0.0000000
Convolutional Depth * Activation	-0.0002488	8.31e-05	-2.9927322	0.0028778
Convolutional Depth * Initialization	-0.0006231	8.43e-05	-7.3961401	0.0000000
Fully-Connected Width * Fully-Connected Depth	0.0000409	8.39e-05	0.4868917	0.6265117
Fully-Connected Width * BatchNorm	-0.0003099	8.38e-05	-3.6995435	0.0002357
Fully-Connected Width * Dropout	0.0002693	8.41e-05	3.2005378	0.0014437
Fully-Connected Width * Activation	0.0001985	8.32e-05	2.3855826	0.0173593
Fully-Connected Depth * BatchNorm	-0.0005052	8.54e-05	-5.9141188	0.0000000
Fully-Connected Depth * Max Pooling	-0.0001497	8.37e-05	-1.7891438	0.0740926
Fully-Connected Depth * Dropout	0.0026378	8.47e-05	31.1458046	0.0000000
Fully-Connected Depth * Activation	-0.0003769	8.41e-05	-4.4788230	0.0000090
Fully-Connected Depth * Initialization	-0.0001068	8.50e-05	-1.2564667	0.2094325
BatchNorm * Max Pooling	-0.0004213	8.35e-05	-5.0463492	0.0000006
BatchNorm * Dropout	-0.0007233	8.47e-05	-8.5379306	0.0000000
BatchNorm * Activation	0.0004527	8.39e-05	5.3937300	0.0000001
BatchNorm * Initialization	0.0002929	8.49e-05	3.4491110	0.0006016
Max Pooling * Dropout	0.0002978	8.30e-05	3.5861366	0.0003628
Max Pooling * Activation	-0.0004016	8.43e-05	-4.7625621	0.0000024
Dropout * Activation	-0.0002095	8.35e-05	-2.5103941	0.0123199
Dropout * Initialization	0.0001015	8.44e-05	1.2020046	0.2298328
Activation * Initialization	-0.0000033	8.43e-05	-0.0391485	0.9687849

Table 52. Stochastic Gradient Descent Full Second-Order Test Loss Model

Factor	Estimate	Std. Error	T-Value	P-Value
Intercept	1.9743380	8.35e-05	23644.9255561	0.0000000
Convolutional Width	-0.0005282	8.35e-05	-6.3242504	0.0000000
Convolutional Depth	0.0028717	8.34e-05	34.4382380	0.0000000
Fully-Connected Width	0.0001679	8.35e-05	2.0118826	0.0446490
Fully-Connected Depth	0.0039748	8.35e-05	47.5867910	0.0000000
BatchNorm	-0.0003507	8.36e-05	-4.1948194	0.0000311
Max Pooling	0.0012878	8.34e-05	15.4476509	0.0000000
Dropout	0.0063305	8.36e-05	75.7223537	0.0000000
Activation	-0.0001667	8.35e-05	-1.9969808	0.0462483
Initialization	-0.0001895	8.36e-05	-2.2671277	0.0237136
Convolutional Width * Convolutional Depth	-0.0005644	8.34e-05	-6.7691782	0.0000000
Convolutional Width * Fully-Connected Width	-0.0000701	8.34e-05	-0.8413049	0.4004899
Convolutional Width * Fully-Connected Depth	-0.0001775	8.36e-05	-2.1243826	0.0340191
Convolutional Width * BatchNorm	0.0004046	8.35e-05	4.8460923	0.0000016
Convolutional Width * Max Pooling	-0.0002286	8.34e-05	-2.7414105	0.0062877
Convolutional Width * Dropout	0.0000017	8.35e-05	0.0200107	0.9840410
Convolutional Width * Activation	0.0003321	8.36e-05	3.9739473	0.0000787
Convolutional Width * Initialization	0.0001203	8.35e-05	1.4409088	0.1500972
Convolutional Depth * Fully-Connected Width	0.0001895	8.36e-05	2.2663653	0.0237605
Convolutional Depth * Fully-Connected Depth	-0.0013072	8.34e-05	-15.6704961	0.0000000
Convolutional Depth * BatchNorm	-0.0003304	8.35e-05	-3.9575382	0.0000842
Convolutional Depth * Max Pooling	0.0016926	8.35e-05	20.2696822	0.0000000
Convolutional Depth * Dropout	0.0005857	8.35e-05	7.0161547	0.0000000
Convolutional Depth * Activation	-0.0002616	8.33e-05	-3.1396788	0.0017688
Convolutional Depth * Initialization	-0.0002449	8.35e-05	-2.9335423	0.0034705
Fully-Connected Width * Fully-Connected Depth	0.0001031	8.34e-05	1.2357575	0.2170000
Fully-Connected Width * BatchNorm	-0.0001122	8.34e-05	-1.3451133	0.1790629
Fully-Connected Width * Dropout	0.0001745	8.34e-05	2.0925843	0.0367777
Fully-Connected Width * Activation	-0.0000122	8.34e-05	-0.1462902	0.8837382
Fully-Connected Depth * BatchNorm	-0.0001290	8.36e-05	-1.5443782	0.1229887
Fully-Connected Depth * Max Pooling	-0.0006156	8.34e-05	-7.3814429	0.0000000
Fully-Connected Depth * Dropout	0.0028332	8.36e-05	33.9025523	0.0000000
Fully-Connected Depth * Activation	-0.0001593	8.35e-05	-1.9079590	0.0568416
Fully-Connected Depth * Initialization	-0.0001126	8.36e-05	-1.3472301	0.1783807
BatchNorm * Max Pooling	-0.0000862	8.33e-05	-1.0343690	0.3013526
BatchNorm * Dropout	-0.0002419	8.35e-05	-2.8968743	0.0038972
BatchNorm * Activation	0.0002914	8.36e-05	3.4880590	0.0005197
BatchNorm * Initialization	0.0000478	8.35e-05	0.5728514	0.5669455
Max Pooling * Dropout	0.0005403	8.34e-05	6.4809112	0.0000000
Max Pooling * Activation	-0.0000516	8.35e-05	-0.6180121	0.5367862
Dropout * Activation	-0.0001745	8.35e-05	-2.0888828	0.0371109
Dropout * Initialization	-0.0003858	8.35e-05	-4.6205592	0.0000046
Activation * Initialization	-0.0007309	8.35e-05	-8.7488466	0.0000000

Table 53. RMSProp Full Second-Order Test Loss Model

Factor	Estimate	Std. Error	T-Value	P-Value
Intercept	1.9811249	0.0001434	13817.8002059	0.0000000
Convolutional Width	-0.0002356	0.0001435	-1.6421219	0.1010622
Convolutional Depth	-0.0001819	0.0001433	-1.2697357	0.2046461
Fully-Connected Width	0.0007821	0.0001433	5.4583094	0.0000001
Fully-Connected Depth	0.0003266	0.0001432	2.2807844	0.0228932
BatchNorm	-0.0026398	0.0001433	-18.4195718	0.0000000
Max Pooling	0.0001366	0.0001433	0.9529898	0.3409596
Dropout	0.0023867	0.0001433	16.6540829	0.0000000
Activation	0.0019829	0.0001433	13.8385971	0.0000000
Initialization	0.0001461	0.0001434	1.0186377	0.3087648
Convolutional Width * Convolutional Depth	-0.0002054	0.0001433	-1.4336001	0.1521809
Convolutional Width * Fully-Connected Width	0.0002237	0.0001433	1.5614637	0.1189150
Convolutional Width * Fully-Connected Depth	-0.0003656	0.0001433	-2.5515860	0.0109577
Convolutional Width * BatchNorm	-0.0006101	0.0001432	-4.2598678	0.0000236
Convolutional Width * Max Pooling	0.0011223	0.0001432	7.8365163	0.0000000
Convolutional Width * Dropout	0.0001552	0.0001433	1.0830413	0.2792031
Convolutional Width * Activation	-0.0001925	0.0001432	-1.3441467	0.1793832
Convolutional Width * Initialization	-0.0000520	0.0001433	-0.3628053	0.7168716
Convolutional Depth * Fully-Connected Width	-0.0001900	0.0001434	-1.3248294	0.1857065
Convolutional Depth * Fully-Connected Depth	0.0003174	0.0001433	2.2146705	0.0271386
Convolutional Depth * BatchNorm	-0.0031879	0.0001435	-22.2130100	0.0000000
Convolutional Depth * Max Pooling	0.0005899	0.0001435	4.1117957	0.0000444
Convolutional Depth * Dropout	0.0001063	0.0001435	0.7403534	0.4593605
Convolutional Depth * Activation	-0.0016903	0.0001434	-11.7908690	0.0000000
Convolutional Depth * Initialization	-0.0000687	0.0001433	-0.4799069	0.6314596
Fully-Connected Width * Fully-Connected Depth	-0.0005290	0.0001434	-3.6901979	0.0002434
Fully-Connected Width * BatchNorm	-0.0002623	0.0001434	-1.8296554	0.0677724
Fully-Connected Width * Dropout	-0.0004585	0.0001434	-3.1969447	0.0014582
Fully-Connected Width * Activation	-0.0000643	0.0001434	-0.4486524	0.6538362
Fully-Connected Depth * BatchNorm	-0.0008585	0.0001434	-5.9879599	0.0000000
Fully-Connected Depth * Max Pooling	0.0000228	0.0001434	0.1593153	0.8734713
Fully-Connected Depth * Dropout	0.0010820	0.0001434	7.5464806	0.0000000
Fully-Connected Depth * Activation	-0.0003346	0.0001434	-2.3333269	0.0199437
Fully-Connected Depth * Initialization	-0.0003437	0.0001432	-2.3995668	0.0167030
BatchNorm * Max Pooling	0.0003638	0.0001434	2.5373211	0.0114097
BatchNorm * Dropout	0.0013798	0.0001433	9.6290668	0.0000000
BatchNorm * Activation	0.0013510	0.0001434	9.4180586	0.0000000
BatchNorm * Initialization	0.0001613	0.0001433	1.1255831	0.2607693
Max Pooling * Dropout	-0.0004673	0.0001433	-3.2608507	0.0011705
Max Pooling * Activation	0.0000524	0.0001434	0.3653071	0.7150044
Dropout * Activation	-0.0008872	0.0001435	-6.1836964	0.0000000
Dropout * Initialization	0.0000511	0.0001432	0.3566016	0.7215091
Activation * Initialization	0.0000240	0.0001433	0.1676495	0.8669127

Table 54. Adam Full Second-Order Test Loss Model

Factor	Estimate	Std. Error	T-Value	P-Value
Intercept	1.9746158	8.66e-05	22790.2005935	0.0000000
Convolutional Width	-0.0005329	8.67e-05	-6.1480107	0.0000000
Convolutional Depth	0.0006419	8.67e-05	7.4068467	0.0000000
Fully-Connected Width	0.0001173	8.67e-05	1.3530813	0.1765001
Fully-Connected Depth	0.0019644	8.66e-05	22.6773825	0.0000000
BatchNorm	-0.0001198	8.67e-05	-1.3819795	0.1674527
Max Pooling	0.0002679	8.67e-05	3.0904502	0.0020839
Dropout	0.0064685	8.67e-05	74.6361845	0.0000000
Activation	0.0004040	8.66e-05	4.6629174	0.0000038
Initialization	-0.0001660	8.67e-05	-1.9153234	0.0558898
Convolutional Width * Convolutional Depth	-0.0006637	8.67e-05	-7.6571341	0.0000000
Convolutional Width * Fully-Connected Width	0.0000662	8.66e-05	0.7640597	0.4451089
Convolutional Width * Fully-Connected Depth	0.0003050	8.67e-05	3.5192184	0.0004630
Convolutional Width * BatchNorm	-0.0000445	8.67e-05	-0.5138734	0.6075152
Convolutional Width * Max Pooling	-0.0001011	8.67e-05	-1.1664009	0.2438802
Convolutional Width * Dropout	-0.0001954	8.67e-05	-2.2540885	0.0245228
Convolutional Width * Activation	0.0000329	8.67e-05	0.3791041	0.7047344
Convolutional Width * Initialization	-0.0000507	8.66e-05	-0.5858268	0.5581952
Convolutional Depth * Fully-Connected Width	-0.0002251	8.67e-05	-2.5977316	0.0095967
Convolutional Depth * Fully-Connected Depth	-0.0002440	8.67e-05	-2.8154196	0.0050190
Convolutional Depth * BatchNorm	-0.0002869	8.66e-05	-3.3107077	0.0009821
Convolutional Depth * Max Pooling	0.0010003	8.67e-05	11.5390536	0.0000000
Convolutional Depth * Dropout	0.0008265	8.66e-05	9.5394217	0.0000000
Convolutional Depth * Activation	-0.0002761	8.67e-05	-3.1859645	0.0015118
Convolutional Depth * Initialization	-0.0004189	8.67e-05	-4.8338565	0.0000017
Fully-Connected Width * Fully-Connected Depth	-0.0002204	8.67e-05	-2.5427059	0.0112305
Fully-Connected Width * BatchNorm	-0.0001054	8.67e-05	-1.2156926	0.2245434
Fully-Connected Width * Dropout	0.0000792	8.67e-05	0.9134241	0.3613582
Fully-Connected Width * Activation	0.0003640	8.67e-05	4.2004893	0.0000304
Fully-Connected Depth * BatchNorm	-0.0002627	8.67e-05	-3.0319727	0.0025263
Fully-Connected Depth * Max Pooling	0.0001633	8.67e-05	1.8840991	0.0599976
Fully-Connected Depth * Dropout	0.0027708	8.67e-05	31.9769325	0.0000000
Fully-Connected Depth * Activation	-0.0000314	8.66e-05	-0.3619031	0.7175420
Fully-Connected Depth * Initialization	0.0001085	8.67e-05	1.2514256	0.2112294
BatchNorm * Max Pooling	-0.0000449	8.67e-05	-0.5178546	0.6047359
BatchNorm * Dropout	-0.0008291	8.66e-05	-9.5689183	0.0000000
BatchNorm * Activation	0.0004258	8.67e-05	4.9131376	0.0000011
BatchNorm * Initialization	0.0000096	8.67e-05	0.1106541	0.9119248
Max Pooling * Dropout	0.0003197	8.67e-05	3.6877191	0.0002452
Max Pooling * Activation	0.0001898	8.67e-05	2.1902887	0.0288578
Dropout * Activation	-0.0000506	8.67e-05	-0.5833098	0.5598871
Dropout * Initialization	0.0000532	8.67e-05	0.6135922	0.5396995
Activation * Initialization	-0.0001556	8.67e-05	-1.7951358	0.0730965

Table 55. AdaGrad Full Second-Order Test Loss Model

Factor	Estimate	Std. Error	T-Value	P-Value
Intercept	1.9741957	9.02e-05	21895.1540721	0.0000000
Convolutional Width	-0.0001156	9.02e-05	-1.2814876	0.2004759
Convolutional Depth	0.0030451	9.02e-05	33.7711565	0.0000000
Fully-Connected Width	0.0001773	9.02e-05	1.9662581	0.0496907
Fully-Connected Depth	0.0037180	9.02e-05	41.2347166	0.0000000
BatchNorm	0.0005381	9.02e-05	5.9674897	0.0000000
Max Pooling	0.0009665	9.02e-05	10.7174386	0.0000000
Dropout	0.0063297	9.02e-05	70.2008719	0.0000000
Activation	0.0001776	9.02e-05	1.9693519	0.0493337
Initialization	-0.0004314	9.02e-05	-4.7839303	0.0000021
Convolutional Width * Convolutional Depth	-0.0001697	9.02e-05	-1.8820452	0.0602730
Convolutional Width * Fully-Connected Width	0.0002018	9.02e-05	2.2381276	0.0255480
Convolutional Width * Fully-Connected Depth	-0.0000651	9.02e-05	-0.7216900	0.4707426
Convolutional Width * BatchNorm	0.0000773	9.02e-05	0.8571387	0.3916817
Convolutional Width * Max Pooling	-0.0003853	9.02e-05	-4.2733120	0.0000221
Convolutional Width * Dropout	0.0000842	9.02e-05	0.9338038	0.3507492
Convolutional Width * Activation	0.0000560	9.02e-05	0.6212960	0.5346210
Convolutional Width * Initialization	-0.0001813	9.02e-05	-2.0106451	0.0447723
Convolutional Depth * Fully-Connected Width	0.0001165	9.02e-05	1.2915829	0.1969572
Convolutional Depth * Fully-Connected Depth	-0.0017358	9.02e-05	-19.2502381	0.0000000
Convolutional Depth * BatchNorm	-0.0002071	9.02e-05	-2.2972787	0.0219174
Convolutional Depth * Max Pooling	0.0012856	9.02e-05	14.2566274	0.0000000
Convolutional Depth * Dropout	0.0004720	9.02e-05	5.2343973	0.0000002
Convolutional Depth * Activation	-0.0002153	9.02e-05	-2.3878860	0.0172280
Convolutional Depth * Initialization	-0.0002587	9.02e-05	-2.8688953	0.0042518
Fully-Connected Width * Fully-Connected Depth	-0.0003216	9.02e-05	-3.5662043	0.0003886
Fully-Connected Width * BatchNorm	0.0001411	9.02e-05	1.5642814	0.1182345
Fully-Connected Width * Dropout	-0.0000024	9.02e-05	-0.0265302	0.9788425
Fully-Connected Width * Activation	-0.0000923	9.02e-05	-1.0232421	0.3065712
Fully-Connected Depth * BatchNorm	0.0001719	9.02e-05	1.9061322	0.0570705
Fully-Connected Depth * Max Pooling	-0.0004861	9.02e-05	-5.3902268	0.0000001
Fully-Connected Depth * Dropout	0.0024433	9.02e-05	27.0972826	0.0000000
Fully-Connected Depth * Activation	0.0001441	9.02e-05	1.5986020	0.1103913
Fully-Connected Depth * Initialization	0.0000415	9.02e-05	0.4599011	0.6457398
BatchNorm * Max Pooling	0.0001622	9.02e-05	1.7983400	0.0725835
BatchNorm * Dropout	-0.0007254	9.02e-05	-8.0448257	0.0000000
BatchNorm * Activation	0.0002340	9.02e-05	2.5950835	0.0096686
BatchNorm * Initialization	0.0002466	9.02e-05	2.7341646	0.0064229
Max Pooling * Dropout	0.0001514	9.02e-05	1.6790363	0.0936219
Max Pooling * Activation	0.0002694	9.02e-05	2.9879457	0.0029137
Dropout * Activation	-0.0002766	9.02e-05	-3.0681497	0.0022428
Dropout * Initialization	-0.0000747	9.02e-05	-0.8289499	0.4074347
Activation * Initialization	-0.0005094	9.02e-05	-5.6491847	0.0000000

Bibliography

1. Achille, A., Rovere, M. and Soatto, S. [2019], 'Critical learning periods in deep neural networks'.

2. Agarwal, N., Allen-Zhu, Z., Bullins, B., Hazan, E. and Ma, T. [2016], 'Finding approximate local minima faster than gradient descent', *49th Annual ACM SIGACT Symposium* .

3. Aggarwal, C. C. [2018], *Neural Networks and Deep Learning*, Springer International Publishing AG.

4. Akiba, T., Suzuki, S. and Fukuda, K. [2017], 'Extremely large minibatch sgd: Training resnet-50 on imagenet in 15 minutes'.

5. Alom, M. Z., Taha, T. M., Yakopcic, C., Westberg, S., Sidike, P., Nasrin, M. S., Hasan, M., Essen, B. C. V., Awwal, A. A. S. and Asari, V. K. [2019], 'A state-of-the-art survey on deep learning theory and architectures', *Electronics* .

6. Apostol, T. M. [1974], *Mathematical Analysis, Second Edition*, Addison-Wesley Publishing Company, Inc.

7. Bazaraa, M. S., Sherali, H. D. and Shetty, C. M. [2006], *Nonlinear Programming: Theory and Algorithms*, John Wiley and Sons, Inc.

8. Ben-Tal, A. and Nemirovski, A. [2005], 'Non-euclidean restricted memory level method for large-scale convex optimization', *Math. Program., Ser. A* pp. 407–456.

9. Ben-Tal, A. and Nemirovski, A. [2019], 'Lectures on modern convex optimization'.

10. Bengio, Y., Simard, P. and Frasconi, P. [1994], 'Learning long-term dependencies with gradient descent is difficult', *IEEE Transactions on Neural Networks* **5**(2), 157–166.

11. Bianchi, L., Dorigo, M., Gambardella, L. M. and Gutjahr, W. J. [2008], 'Communication networks in the brain', *Alcohol Res Health* .

12. Blum, A. and Rivest, R. L. [1992], 'Training a 3-node neural network is np-complete', *Neural Networks* .

13. Bottou, L. [1998], On-line learning and stochastic approximations, *in* 'In On-line Learning in Neural Networks', Cambridge University Press, pp. 9–42.

14. Bottou, L., Curtis, F. E. and Nocedal, J. [2018], 'Optimization methods for large-scale machine learning', *arXiv e-prints* .

15. Bray, A. J. and Dean, D. S. [2006], 'Statistics of critical points of gaussian fields on large-dimensional spaces', *Physics Review Letters* .

16. Bubeck, S. [2015], *Convex Optimization: Algorithms and Complexity*, Foundations and Trends in Machine Learning.

17. Carmon, Y., Duchi, J. C., Hinder, O. and Sidford, A. [2018], 'Accelerated methods for non-convex optimization', *AISTATS* .

18. Choromanska, A., Henaff, M., Mathieu, M., erard, G., Arous, B. and LeCun, Y. [2015], 'The loss surfaces of multilayer networks', *AISTATS* .

19. Cook, R. D. [1977], 'Detection of influential observation in linear regression', *Technometrics* **19**(1), 15–18.
URL: *http://www.jstor.org/stable/1268249*

20. Dauphin, Y. N., Pascanu, R., Gulcehre, C., Chon, K., Ganguli, S. and Bengio, Y. [2014], 'Identifying and attacking the saddle point problem in high-dimensional non-convex optimization', *Conference on Neural Information Processing Systems* .

21. Dinh, L., Pascanu, R., Bengio, S. and Bengio, Y. [2017], 'Sharp minima can generalize for deep nets', *Proceedings of the 34th International Conference on Machine Learning* .

22. Do, T. M. T. and Artières, T. [2012], 'Regularized bundle methods for convex and non-convex risks', *Journal of Machine Learning Research* **13**(114), 3539–3583.
URL: *http://jmlr.org/papers/v13/do12a.html*

23. Duchi, J., Hazan, E. and Singer, Y. [2011], 'Adaptive subgradient methods for online learning and stochastic optimization', *Journal of Machine Learning Research* **12**, 2121–2159.

24. Durakovic, B. [2017], 'Design of experiments application, concepts, examples: State of the art', *Periodicals of Engineering and Natural Sciences* .

25. Fort, S. and Scherlis, A. [2019], 'The goldilocks zone: Towards better understanding of neural network loss landscapes', *Proceedings of the AAAI Conference on Artificial Intelligence* **33**, 3574–3581.

26. Frankle, J. and Carbin, M. [2019], 'The lottery ticket hypothesis: Finding sparse, trainable neural networks'.

27. Frankle, J., Schwab, D. J. and Morcos, A. S. [2020], 'The early phase of neural network training'.

28. Fukushima, K. [1980], 'Neocognitron: A self-organizing neural network model for a mechanism of pattern recognition unaffected by shift in position', *Biological Cybernetics* .

29. Ghorbani, B., Krishnan, S. and Xiao, Y. [2019], 'An investigation into neural net optimization via hessian eigenvalue density', *Proceedings of the 36th International Conference on Machine Learning* .

30. Glorot, X. and Bengio, Y. [2010], 'Understanding the difficulty of training deep feedforward neural networks', *Proceedings of the 13th International Conference on Artificial Intelligence and Statistics (AISTATS) 2010* .

31. Golatkar, A., Achille, A. and Soatto, S. [2019], 'Time matters in regularizing deep networks: Weight decay and data augmentation affect early learning dynamics, matter little near convergence'.

32. Goodfellow, I. J., Vinyals, O. and Saxe, A. M. [2015], 'Qualitatively characterizing neural network optimization problems', *International Conference on Learning Representations* .

33. Gron, A. [2017], *Hands-On Machine Learning with Scikit-Learn and TensorFlow: Concepts, Tools, and Techniques to Build Intelligent Systems*, 1st edn, O'Reilly Media, Inc.

34. Hardt, M., Recht, B. and Singer, Y. [2016], 'Train faster, generalize better: Stability of stochastic gradient descent'.

35. He, F., Wang, B. and Tao, D. [2020], 'Piecewise linear activations substantially shape the loss surfaces of neural networks', *CoRR* **abs/2003.12236**.
URL: *https://arxiv.org/abs/2003.12236*

36. He, K., Zhang, X., Ren, S. and Sun, J. [2015*a*], 'Deep residual learning for image recognition', *arXiv e-prints* .

37. He, K., Zhang, X., Ren, S. and Sun, J. [2015*b*], 'Delving deep into rectifiers: Surpassing human-level performance on imagenet classification', *IEEE International Conference on Computer Vision (ICCV 2015)* **1502**.

38. He, X., Zhao, K. and Chu, X. [2021], 'Automl: A survey of the state-of-the-art', *Knowledge-Based Systems* **212**, 106622.
URL: *http://dx.doi.org/10.1016/j.knosys.2020.106622*

39. Hinton, G., Srivastava, N. and Swersky, K. [2012], 'Neural networks for machine learning lecture 6a overview of mini-batch gradient descent', *Coursera* .

40. Hochreiter, S. and Schmidhuber, J. [1997], 'Flat Minima', *Neural Computation* **9**(1), 1–42.
URL: *https://doi.org/10.1162/neco.1997.9.1.1*

41. Im, D. J., Tao, M. and Branson, K. [2019], 'An empirical analysis of the optimization of deep network loss surfaces', *AAAI* .

42. Ioffe, S. and Szegedy, C. [2015], 'Batch normalization: Accelerating deep network training by reducing internal covariate shift', *Proceedings of the 32nd International Conference on Machine Learning* .

43. Juditsky, A. and Nemirovski, A. [2011], 'First-order methods for nonsmooth convex large-scale optimization, i: General purpose methods'.

44. Karpathy, A. [2017], 'A peek at trends in machine learning'.
URL: *https://medium.com/@karpathy/a-peek-at-trends-in-machine-learning-ab8a1085a106*

45. Keskar, N., Mudigere, D., Nocedal, J., Smelyanskiy, M. and Tang, P. T. P. [2017], 'On large-batch training for deep learning: Generalization gap and sharp minima', *ArXiv* **abs/1609.04836**.

46. Keskar, N. S. and Socher, R. [2017], 'Improving generalization performance by switching from adam to sgd'.

47. Khan, A., Sohail, A., Zahoora, U. and Saeed, A. [2020], 'A survey of the recent architectures of deep convolutional neural networks', *Artificial Intelligence Review* **53**.

48. Kingma, D. P. and Ba, J. [2015], 'Adam: A method for stochastic optimization', *CoRR* **abs/1412.6980**.

49. Kiwiel, K. [1995], 'Proximal level bundle method for convex nondifferentable optimization, saddle point problems and variational inequalities', *Mathematical Programming Series B* .

50. Kiwiel, K. [2010], 'Bundle methods for convex minimization with partially inexact oracles', *Computational Optimization and Applications* .

51. Krizhevsky, A., Nair, V. and Hinton, G. [n.d.], 'Cifar-10 (canadian institute for advanced research)'.
URL: *http://www.cs.toronto.edu/ kriz/cifar.html*

52. Krizhevsky, A., Sutskever, I. and Hinton, G. E. [2012], 'Imagenet classification with deep convolutional neural networks', *Advances in Neural Information Processing Systems,* .

53. Kumar, S. K. [2017], 'On weight initialization in deep neural networks', *CoRR* **abs/1704.08863**.
URL: *http://arxiv.org/abs/1704.08863*

54. LeCun, Y., Bottou, L., Orr, G. B. and Muller, K.-R. [2000], *Neural Networks: Tricks of the Trade*, Springer, Berlin, Heidelberg.

55. LeCun, Y. and Cortes, C. [2010], 'MNIST handwritten digit database'.
URL: *http://yann.lecun.com/exdb/mnist/*

56. Lee, J. D., Panageas, I., Piliouras, G., Simchowitz, M., Jordan, M. I. and Recht, B. [2018], 'First-order methods almost always avoid saddle points', *arXiv e-prints* .

57. Li, D., Ding, T. and Sun, R. [2018], 'Over-parameterized deep neural networks have no strict local minima for any continuous activations', *CoRR* **abs/1812.11039**.
URL: *http://arxiv.org/abs/1812.11039*

58. Li, D., Ding, T. and Sun, R. [2021], 'On the benefit of width for neural networks: Disappearance of bad basins'.

59. Li, H., Xu, Z., Taylor, G., Studer, C. and Goldstein, T. [2018], 'Visualizing the loss landscape of neural nets', *32nd Conference on Neural Information Processing Systems (NeurIPS 2018)* .

60. Liang, D., Ma, F. and Li, W. [2020], 'New gradient-weighted adaptive gradient methods with dynamic constraints', *IEEE Access* **8**, 110929–110942.

61. Liang, S. and Srikant, R. [2017], 'Why deep neural networks for function approximation?', *International Conference on Learning Representations* .

62. Liang, S., Sun, R., Li, Y. and Srikant, R. [2018], 'Understanding the loss surface of neural networks for binary classification', **80**, 2835–2843.
URL: *http://proceedings.mlr.press/v80/liang18a.html*

63. Loshchilov, I. and Hutter, F. [2018], 'Fixing weight decay regularization in adam', *arXiv e-prints* .

64. McCulloch, W. S. and Pitts, W. H. [1943], 'A logical calculus of the ideas of immanent in nervous activity', *Bulletin of Mathematical Biophysics* pp. 115–133.

65. Milne, T. [2019], 'Piecewise strong convexity of neural networks', *Conference on Neural Information Processing Systems* pp. 1–11.

66. Mishkin, D. and Matas, J. [2016], 'All you need is a good init', *International Conference on Learning Representations* .

67. Montgomery, D. C. [2005], *Design and Analysis of Experiments, 6th Edition*, John Wiley and Sons, Inc.

68. Nesterov, Y. [1983], 'A method of solving a convex programming problem with convergence rate o(1/k2)', *Soviet Mathematics Doklady* .

69. Nwankpa, C. E., Ijomah, W., Gachagan, A. and Marshall, S. [2018], 'Activation functions: Comparison of trends in practice and research for deep learning', *arXiv e-prints* .

70. Oliveira, W. and Sagastizabal, C. [2014], 'Level bundle methods for oracles with on-demand accuracy', *Optimization Methods and Software* .

71. Oymak, S. and Soltanolkotabi, M. [2020], 'Toward moderate overparameterization: Global convergence guarantees for training shallow neural networks', *IEEE Journal on Selected Areas in Information Theory* **PP**, 1–1.

72. Pascanu, R., Mikolov, T. and Bengio, Y. [2013], 'On the difficulty of training recurrent neural networks', *Proceedings of the 30th International Conference on Machine Learning* .

73. Paszke, A., Gross, S., Massa, F., Lerer, A., Bradbury, J., Chanan, G., Killeen, T., Lin, Z., Gimelshein, N., Antiga, L., Desmaison, A., Kopf, A., Yang, E., DeVito, Z., Raison, M., Tejani, A., Chilamkurthy, S., Steiner, B., Fang, L., Bai, J. and Chintala, S. [2019], Pytorch: An imperative style, high-performance deep learning library, *in* H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox and R. Garnett, eds, 'Advances in Neural Information Processing Systems 32', Curran Associates, Inc., pp. 8024–8035.
URL: *http://papers.neurips.cc/paper/9015-pytorch-an-imperative-style-high-performance-deep-learning-library.pdf*

74. Pokutta, S., Spiegel, C. and Zimmer, M. [2020], 'Deep neural network training with frank–wolfe', *arXiv e-prints* .

75. Polyak, B. [1964], 'Some methods of speeding up the convergence of iteration methods', *USSR Computational Mathematics and Mathematical Physics* .

76. Pontes, F., Amorim, G., Balestrassi, P., Paiva, A. and Ferreira, J. [2016], 'Design of experiments and focused grid search for neural network parameter optimization', *Neurocomputing* .

77. Ranzato, M., Huang, F. J., Boureau, Y.-L. and LeCun, Y. [2007], 'Unsupervised learning of invariant feature hierarchies with applications to object recognition', pp. 1–8.

78. Riesenhuber, M. and Poggio, T. [1999], 'Hierarchical models of object recognition in cortex', *Nature Neuroscience* **2**, 1019–25.

79. Rosenblatt, F. [1958], 'The perceptron: A probabilistic model for information storage and organization in the brain', *Psychological Review* .

80. Rumelhart, D., Hinton, G. and Williams, R. [1986], 'Learning representations by back-propagating errors'.

81. Safran, I. and Shamir, O. [2016], 'On the quality of the initial basin in over-specified neural networks', *Proceedings of the 33rd International Conference on Machine Learning* .

82. Sagun, L., Evci, U., Guney, V. U., Dauphin, Y. and Bottou, L. [2018], 'Empirical analysis of the hessian of over-parametrized neural networks'.

83. Santurkar, S., Tsipras, D., Ilyas, A. and Madry, A. [n.d.], 'How does batch normalization help optimization?', *32nd Conference on Neural Information Processing Systems (NIPS 2018)* .

84. Scherer, D., Müller, A. and Behnke, S. [2010], 'Evaluation of pooling operations in convolutional architectures for object recognition', *20th International Conference on Artificial Neural Networks (ICANN)* .

85. Schmidhuber, J. [2014], 'Deep learning in neural networks: An overview', *arXiv e-prints* .

86. Smith, L. N. and Topin, N. [2017], 'Exploring loss function topology with cyclical learning rates', *CoRR* **abs/1702.04283**.
URL: *http://arxiv.org/abs/1702.04283*

87. Srivastava, N., Hinton, G., Krizhevsky, A., Sutskever, I. and Salakhutdinov, R. [2014], 'Dropout: A simple way to prevent neural networks from overfitting', *Journal of Machine Learning Research* **15**(56), 1929–1958.
URL: *http://jmlr.org/papers/v15/srivastava14a.html*

88. Sussillo, D. and Abbott, L. [2014], 'Random walks: Training very deep nonlinear feed-forward networks with smart initialization', *CoRR* **abs/1412.6558**.
URL: *http://arxiv.org/abs/1412.6558*

89. Tieleman, T. and Hinton, G. [2012], 'Lecture 6.5 - rmsprop, coursera: Neural networks for machine learning', *Technical report* .

90. Tsai, J.-T., Chou, J.-H. and Liu, T.-K. [2006], 'Tuning the structure and parameters of a neural network by using hybrid taguchi-genetic algorithm', *IEEE transactions on neural networks / a publication of the IEEE Neural Networks Council* **17**, 69–80.

91. Wilson, A. C., Roelofs, R., Stern, M., Srebro, N. and Recht, B. [2017], The marginal value of adaptive gradient methods in machine learning, *in* 'NeurIPS'.

92. Wolpert, D. and Macready, W. [1997], 'No free lunch theorems for optimization', *IEEE Transactions on Evolutionary Computation* **1**(1), 67–82.

93. Wu, Y., Schuster, M., Chen, Z., Le, Q. V., Norouzi, M., Macherey, W., Krikun, M., Cao, Y., Gao, Q., Macherey, K., Klingner, J., Shah, A., Johnson, M., Liu, X.,

Łukasz Kaiser, Gouws, S., Kato, Y., Kudo, T., Kazawa, H., Stevens, K., Kurian, G., Patil, N., Wang, W., Young, C., Smith, J., Riesa, J., Rudnick, A., Vinyals, O., Corrado, G., Hughes, M. and Dean, J. [2016], 'Google's neural machine translation system: Bridging the gap between human and machine translation'.

94. Xiao, H., Rasul, K. and Vollgraf, R. [2017], 'Fashion-mnist: a novel image dataset for benchmarking machine learning algorithms', *ArXiv* **abs/1708.07747**.

95. Yang, S. M. and Lee, G. S. [1999], 'Neural network design by using taguchi method', *Journal of Dynamic Systems, Measurement, and Control* .

96. Yu, X.-H. and Chen, G.-A. [1995], 'On the local minima free condition of back-propagation learning', *IEEE Transactions on Neural Networks* **6**(5), 1300–1303.

97. Zhang, G., Wang, C., Xu, B. and Grosse, R. [2019], 'Three mechanisms of weight decay regularization', *arXiv e-prints* .

98. Zhang, X., Chen, X., Yao, L., Ge, C. and Dong, M. [2019], 'Deep neural network hyperparameter optimization with orthogonal array tuning', *CoRR* **abs/1907.13359**.
URL: *http://arxiv.org/abs/1907.13359*

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. REPORT DATE *(DD-MM-YYYY)*	**2. REPORT TYPE**	**3. DATES COVERED** *(From - To)*
16-09-2021	**Dissertation**	**Oct 2018 – Aug 2021**

4. TITLE AND SUBTITLE

Characterizing Convolutional Neural Network Early Learning and Accelerating Non-Adaptive, First-Order Methods with Localized Lagrangian Restricted Memory Level Bundling

5a. CONTRACT NUMBER

5b. GRANT NUMBER

5c. PROGRAM ELEMENT NUMBER

6. AUTHOR(S)

Morris, Benjamin O., NH-03

5d. PROJECT NUMBER

5e. TASK NUMBER

5f. WORK UNIT NUMBER

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)

Air Force Institute of Technology
Graduate School of Engineering and Management (AFIT/EN)
2950 Hobson Way
WPAFB, OH 45433-7765

8. PERFORMING ORGANIZATION REPORT NUMBER

AFIT-ENS-DS-21-S-049

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)

Headquarters Air Force Materiel Command A9A
ATTN: Julia Phillips, PhD
4225 Logistics Ave
WPAFB, OH 45433
(937) 503-2406, julia.phillips.4@us.af.mil

10. SPONSOR/MONITOR'S ACRONYM(S)

HQ AFMC A5/8/9 A9A

11. SPONSOR/MONITOR'S REPORT NUMBER(S)

12. DISTRIBUTION/AVAILABILITY STATEMENT

Distribution Statement A: Approved For Public Release; Distribution Unlimited

13. SUPPLEMENTARY NOTES

This material is declared a work of the U.S. Government and is not subject to copyright protection in the United States.

14. ABSTRACT **This dissertation studies the underlying optimization problem encountered during the early-learning stages of convolutional neural networks and introduces a training algorithm competitive with existing state-of-the-art methods. A Design of Experiments method is introduced to systematically measure empirical second-order Lipschitz upper bound and region size estimates for local regions of convolutional neural network loss surfaces experienced during the early-learning stages. This method demonstrates that architecture choices can significantly impact the local loss surfaces traversed during training. A Design of Experiments method is used to study the effects convolutional neural network architecture hyper parameters have on different optimization routines' abilities to effectively train and find solutions that generalize well during early learning, demonstrating a relationship between routine selection and network architecture. A method to accelerate the early learning of non-adaptive, first-order optimization routines is developed. The method decomposes the neural network training problem into a series of unconstrained optimization problems within localized trailing Euclidean trust regions and allows non-adaptive methods to exhibit training results which are competitive with adaptive methods.**

15. SUBJECT TERMS

Artificial Intelligence; Image Classification; Machine Learning; Neural Networks

16. SECURITY CLASSIFICATION OF:			**17. LIMITATION OF ABSTRACT**	**18. NUMBER OF PAGES**	**19a. NAME OF RESPONSIBLE PERSON**
a. REPORT	**b. ABSTRACT**	**c. THIS PAGE**			**Jeffery Weir, PhD, AFIT/ENS**
U	**U**	**U**	**UU**	**188**	**19b. TELEPHONE NUMBER** *(Include area code)* *(937) 255-3636 EXT 4523*