

UNIVERSITY OF OKLAHOMA

GRADUATE COLLEGE

IMPROVED LEARNING THROUGH NEURAL COMPONENT SEARCH

A DISSERTATION

SUBMITTED TO THE GRADUATE FACULTY

in partial fulfillment of the requirements for the

Degree of

DOCTOR OF PHILOSOPHY

BY

BRANDON SCOTT MORGAN

Norman, Oklahoma

2024

IMPROVED LEARNING THROUGH NEURAL COMPONENT SEARCH

A DISSERTATION APPROVED FOR THE
SCHOOL OF COMPUTER SCIENCE

BY THE COMMITTEE CONSISTING OF

Dr. Dean Frederick Hougen, Chair

Dr. Charles Nicholson

Dr. Chongle Pan

Dr. Dimitrios Diochnos

Acknowledgements

I first would like to thank my parents, for supporting me through this journey. I would also like to thank the many friends that I've gained along the way during my five year academic career here at the University of Oklahoma. Lastly, I would like to thank my supervisor, Dr. Hougen, for always taking time to discuss my research.

Abstract

Deep learning models contain many different hyper-parameters that need to be tuned prior to training. These hyper-parameters greatly influence the quality of the final model. Historically, most attention and research has been performed on tuning the architecture. However, with the advent of automated machine learning and the success from neural architecture search, automated methods have been successfully applied to other components of neural networks, challenging the very inspiration of the classical methodologies. In this work, automated methods, through the use of evolutionary algorithms, were applied to the *learning components* of a neural network: the loss function, optimizer, learning rate schedule, and output activation function of computer vision models with the goal of finding drop-in replacements for standard components. I expand upon previous research in each of these respective domains through the proposal of new search spaces, surrogate functions, genetic algorithms, and better found components. In the end, multiple loss functions, optimizers, learning rate schedules, and output activation functions, all evolved from scratch, were found to be able to outperform cross-entropy, Adam, one cycle cosine decay, and softmax on the CIFAR datasets.

Contents

Acknowledgements	iv
Abstract	v
List of Figures	viii
List of Tables	xi
1 Introduction	1
2 Related Work	3
2.1 Artificial Neural Networks	3
2.1.1 Foundation	4
2.1.2 Convolutional Neural Networks	5
2.1.3 Neural Architecture Search	6
2.2 Genetic Algorithms	7
2.2.1 Genetic Programming	7
2.2.2 Regularized Evolution	8
2.3 CIFAR	10
3 Motivation	13
4 Proposed Search Space	18
4.1 Overview	18
4.2 Search Space	18
4.3 Particle-Based Genetic Algorithm	20
5 Neural Loss Function Search	23
5.1 Overview	23
5.2 Specific Related Work	24
5.3 Contribution	25
5.4 Shared Methodology	26
5.4.1 Loss Function Initialization	26

5.4.2	Integrity Check	26
5.4.3	Genetic Algorithm	29
5.4.4	Further Implementation Details	31
5.5	NeuroLoss Evolution	33
5.5.1	Methodology	33
5.5.2	Results and Discussion	40
5.5.3	Training Strategies	47
5.5.4	Conclusion	50
5.6	Image-Augmentation Specific Evolution	50
5.7	Specific Related Work	50
5.7.1	Methodology	52
5.7.2	Results	58
5.7.3	Conclusion	64
6	Neural Optimizer Search	67
6.1	Overview	67
6.2	Specific Related Work	68
6.3	Contribution	70
6.4	Methodology	71
6.4.1	Search Space	71
6.4.2	Integrity Check	76
6.4.3	Surrogate Function	78
6.4.4	Early Stopping	79
6.4.5	Particle-Based Genetic Algorithm	80
6.4.6	Optimizer Elimination Protocol	82
6.4.7	Adam Variants	84
6.5	Results	85
6.5.1	Final Optimizers	85
6.5.2	Supplementary Experiments	85
6.6	Transferability Experiments	87
6.7	Results and Discussion	88
6.8	Analysis of the Final Optimizers	90
6.8.1	Observing the Final Learning Rate Schedules and Internal Decay Functions	90
6.8.2	Observing the Final Decay Function	91
6.8.3	Observing the Adaption of QHM	92
6.9	Experimental Implementations and Details	94
6.10	Supplementary PTB Experiments	96
6.11	Conclusions	98

7	Neural Learning Rate Search	105
7.1	Overview	105
7.2	Specific Related Work	106
7.3	Contribution	108
7.4	Methodology	109
7.4.1	Search Space	109
7.4.2	Genetic Algorithm	113
7.4.3	Surrogate Function	116
7.5	Results and Discussion	117
7.5.1	Final Learning Rates	118
7.5.2	Transferability Experiments	119
7.6	Scheduled Decay Functions	121
7.7	Final Learning Rate Equations	122
7.8	Conclusions	123
8	Neural Output Function Search	125
8.1	Overview	125
8.2	Contribution	125
8.3	Methodology	126
8.3.1	Search Space	126
8.3.2	Genetic Algorithm	128
8.3.3	Surrogate Function	128
8.4	Results and Discussion	129
8.4.1	Final Output Activation Functions	129
8.4.2	CIFAR and TinyImageNet	130
8.4.3	Bessel Functions	135
8.5	Conclusions	137
9	Optimizing CIFAR-10	138
10	Conclusions	142
	Bibliography	144
A	Appendix	155
A.1	Neural Loss Function Search	155
A.1.1	Base Losses	155
A.1.2	Cutout Losses	156
A.1.3	Mixup Losses	156
A.1.4	RandAug Losses	158
A.1.5	All Losses	159
A.2	Neural Output Function Search	160

List of Figures

2.1	General workflow of a genetic algorithm.	8
2.2	Two example trees for GP.	9
2.3	Three examples of mutation for GP.	11
2.4	Three examples of mutation for GP.	12
4.1	Example loss function with three active (blue) hidden state nodes, one inactive hidden node (grey), and one root node (white). The final loss equation is given next to the root node in the box. . . .	21
5.1	Overview of the binary phenotypes of cross-entropy, NeuroLoss 1, NeuroLoss 2, and NeuroLoss 3.	28
5.2	Zoomed in view (around $\hat{y} \rightarrow 1$) of the binary phenotypes of cross-entropy, NeuroLoss 1, NeuroLoss 2, and NeuroLoss 3.	29
5.3	Results for the evolutionary algorithm evaluated using the proposed surrogate function.	40
5.4	First Image: original untampered image. Second: base augmentation. Third: cutout augmentation. Fourth: mixup augmentation. Fifth: RandAug augmentation. Sixth: all augmentation	53
5.5	Validation accuracies for <i>base</i> and <i>all</i> techniques for the 1,000 randomly generated loss functions.	54
5.6	Validation accuracies for <i>base</i> and <i>all</i> techniques for the intersection of the best loss functions.	55
5.7	Binary phenotype of the final best loss functions against cross-entropy for normalized loss values.	60
5.8	3D plot of the raw phenotype of the cross-entropy loss.	64
5.9	3D plot of the raw phenotype of the $A2$ loss.	65
5.10	3D plot of the difference in normalized loss values between $A2$ and cross-entropy ($A2 - CE$).	66
6.1	Example optimizer graph with two active (blue) hidden state nodes, two inactive hidden state nodes (grey), and one root node (white). The final weight update equation is given above the root node. Note that this does not include momentum type.	75

6.2	Example decay function applied to the $10^{-4}w$ operand before being applied in the $\ln(x)$ operation. The decay graph contains one active (blue) hidden state node, zero inactive hidden state nodes (grey), and one root node (white). The final decay function equation is given above the root node.	77
6.3	Learning Rate Family 1	91
6.4	Learning Rate Family 2	92
6.5	Outsider Learning Rates	93
6.6	Evolved Internal Decay Functions	94
6.7	Opt4 Internal Decay	95
6.8	Opt7 Weight Update	96
6.9	Opt10 Weight Update	97
6.10	$\text{softsign}(\text{clip}(10^{-5}w, 10^{-5}w - (0.05g^2 + 0.95\hat{s})))$	98
6.11	$\text{softsign}(\frac{10^{-5}w}{\sqrt{1+(10^{-5}w-(0.05g^2+0.95\hat{s}))^2}})$	99
6.12	$\text{softsign}(\frac{10^{-5}w}{\sqrt{1+(10^{-5}w-(0.01g^3+0.99\hat{\lambda}))^2}})$	100
6.13	$\frac{0.3g+0.7\hat{v}}{e^{\hat{v}}}$	101
6.14	$\frac{0.3g+0.7\hat{v}}{e^{10^{-4}w}}$	102
6.15	$\tanh(\text{arcsinh}(0.3g + 0.7\hat{v}))$	103
7.1	(Top Left) Decreasing decay functions. (Top Right) Increasing decay functions. (Bottom Left) Decreasing decay functions with two restarts. (Bottom Right) Example of step wise decay (two steps) for both linear and cosine decay; and, example of the concatenation binary operator ($\text{cat}_{0.5}(x_1, x_2)$) between exponential and cosine decay.	110
7.2	(Top Left) Standard learning rate schedules. (Rest) Best 10 found learning rate schedules for training from scratch.	111
7.3	(Top Left) Standard learning rate schedules. (Rest) Best 10 found learning rate schedules for fine-tuning.	111
7.4	Exact implementation of F_1 learning rate schedule in the proposed computational graph.	114
7.5	Phenotype depiction of S_7 along with two possible mutations, a local and global.	115
7.6	Components of F_2 . Note that all operations are rescaled between 0 and 1.	122
8.1	Exact graph representation for (Left) A_2 and (Right) A_7 . Note, all unused nodes are not shown for brevity.	127
8.2	Training (Left) and test (Right) accuracy for ResNet52v2 on CIFAR-10 for Softmax and $A_{2,3,7}$	132

8.3	Training (Left) and test (Right) accuracy for EfficientNetB0v2 on CIFAR-10 for Softmax and $A_{2,3,7}$.	133
8.4	Training (Left) and test (Right) accuracy for ResNet52v2 on CIFAR-100 for Softmax and $A_{2,3,7}$.	134
8.5	Output of bessel_{i0} and bessel_{i1} .	136
A.1	Binary Phenotype of the final Base losses against cross-entropy for both normalized and raw loss values.	156
A.2	3D plot of the raw phenotype of the $B0$ loss.	157
A.3	3D plot of the raw phenotype of the $B1$ loss.	158
A.4	3D plot of the raw phenotype of the $B2$ loss.	159
A.5	Binary Phenotype of the final Cutout losses against cross-entropy for both normalized and raw loss values.	160
A.6	3D plot of the raw phenotype of the $C0$ loss.	161
A.7	3D plot of the raw phenotype of the $C1$ loss.	162
A.8	3D plot of the raw phenotype of the $C2$ loss.	163
A.9	Binary Phenotype of the final Mixup losses against cross-entropy for both normalized and raw loss values.	164
A.10	3D plot of the raw phenotype of the $M0$ loss.	165
A.11	3D plot of the raw phenotype of the $M1$ loss.	166
A.12	3D plot of the raw phenotype of the $M2$ loss.	167
A.13	Binary Phenotype of the final RandAug losses against cross-entropy for both normalized and raw loss values.	168
A.14	3D plot of the raw phenotype of the $R0$ loss.	169
A.15	3D plot of the raw phenotype of the $R1$ loss.	170
A.16	3D plot of the raw phenotype of the $R2$ loss.	171
A.17	Binary Phenotype of the final All losses against cross-entropy for both normalized and raw loss values.	172
A.18	3D plot of the raw phenotype of the $A0$ loss.	173
A.19	3D plot of the raw phenotype of the $A1$ loss.	174
A.20	Ouput of A_2 over time for EfficientNetB0v2 on CIFAR-10.	175
A.21	Ouput of A_2 over time for EfficientNetB0v2 on CIFAR-100.	176
A.22	Ouput of A_2 over time for ResNet52v2 on CIFAR-100.	177
A.23	Ouput of A_3 over time for EfficientNetB0v2 on CIFAR-10.	178
A.24	Ouput of A_3 over time for EfficientNetB0v2 on CIFAR-100.	179
A.25	Ouput of A_3 over time for ResNet52v2 on CIFAR-100.	180
A.26	Ouput of A_7 over time for EfficientNetB0v2 on CIFAR-10.	181
A.27	Ouput of A_7 over time for EfficientNetB0v2 on CIFAR-100.	182
A.28	Ouput of A_7 over time for ResNet52v2 on CIFAR-100.	183

List of Tables

4.1	The proposed set of unary operations.	20
4.2	The proposed set of binary operations.	20
5.1	Kendall’s Tau Rank Correlation between ConvNet and ResNet9V2 (ResNet), and EffNetV2Small (EffNet). The first row gives the rank correlation for both ConvNet and ResNet9V2 trained for 8,000 steps without RandAug regularization. The remaining rows give the rank correlation for the architectures trained using RandAug regularization at the given image size for 8,000 steps.	35
5.2	Results for Bessel Loss, NeuroLoss1, NeuroLoss4, and cross-entropy using the ConvNet surrogate function. (NL ₁ for NeuroLoss 1, etc.) The mean test accuracy, and standard deviation, across 30 independent runs are reported for each loss function along with the p -value for a one sided statistical t -test for mean accuracy being larger than cross-entropy. The best mean test accuracy is highlighted in bold.	37
5.3	The final three evolved loss functions (NL ₁ for NeuroLoss 1, etc.), along with cross-entropy (CE) for comparison.	41
5.4	Honorable Mentions	42
5.5	The number of training and evaluation images, number of classes, and mean image resolution for the datasets used in this work. . .	42
5.6	Results for DavidNet using resize, cutout, and horizontal flipping, and CutMix for both CIFAR-10 and CIFAR-100. The mean test accuracy and standard deviation across 30 independent runs is reported for each loss function along with the p -value for a Welch’s two-sided statistical t -test for mean difference between the <i>NeuroLoss</i> function and cross-entropy being larger than zero (indicating the function has a larger mean). The best mean test accuracy for each dataset is highlighted in bold.	44

5.7	Results for EffNetV2Small using progressive RandAug regularization, WideResNet (depth=10, width=28) and SEResNeXt (4x64d) using resize, cutout, and horizontal flipping augmentation for both CIFAR-10 and CIFAR-100. The mean test accuracy and standard deviation across 5 independent runs is reported for each loss function. The best mean test accuracy for each combination of dataset and architecture is highlighted in bold.	45
5.8	Results for EffNetV2Small using progressive RandAug regularization when training from scratch for Stanford-CARS196, Oxford-Flowers102, and Caltech101. The mean test accuracy and standard deviation across 5 independent runs is reported for each loss function. The best mean test accuracy for each dataset is highlighted in bold. $CE_{0.10}$ refers to cross-entropy with a label smoothing value of 0.10.	45
5.9	Results for EffNetV2Small using progressive RandAug regularization when fine-tuning for Stanford-CARS196, Oxford-Flowers102, and Caltech101. The mean test accuracy and standard deviation across 5 independent runs is reported for each loss function. The best mean test accuracy for each dataset is highlighted in bold. $CE_{0.10}$ refers to cross-entropy with a label smoothing value of 0.10.	46
5.10	Adapted progressive RandAug regularization training strategy for CIFAR-10 and CIFAR-100.	48
5.11	Original RandAug progressive regularization training strategy for EffNetV2Small for large resolution image datasets.	49
5.12	Kendall Tau's rank correlation between the five different image augmentation techniques for 1000 randomly generated loss functions evaluated on a ResNet9 (6.5M parameters). Top half of the table is the rank correlation for all 1000 loss functions. Bottom half is the rank correlation for best 50.	57
5.13	The final discovered loss functions.	59
5.14	Results for all final loss functions evaluated on the CIFAR datasets using each of the five selected image augmentation techniques. The mean and standard deviation of the test accuracy across 5 independent runs are reported. Red indicates Top 1 performance, while bold indicates Top 5 performance.	62
5.15	Results for all final loss functions evaluated when fine-tuning using each of the five selected image augmentation techniques. The mean and standard deviation of the test accuracy across 5 independent runs are reported. Red indicates Top 1 performance, while bold indicates Top 5 performance.	63

6.1	Optimizer update equations of the form no momentum, momentum, or Nesterov momentum.	71
6.2	The proposed sets of operands and scalar unary operations for the optimizer update equation search space.	72
6.3	The proposed set of scalar unary operations.	73
6.4	The proposed set of binary operations.	73
6.5	The proposed set of state saving unary operations.	74
6.6	The proposed set of reduced unary operations for the decay functions	74
6.7	The proposed set of decay schedule operands for the decay function search space. t denotes the current timestep while T denotes the maximum number of timesteps.	76
6.8	Final 10 optimizers found after evolution. The momentum type, weight update equation, and decay functions used by each optimizer are listed. I refer to each optimizer in the paper by the associated name.	83
6.9	The best five Adam variants found during the first supplementary experiment. Note that all optimizers listed here did not use a momentum type.	86
6.10	The best nine learning rate schedules found from standard evolution and the supplementary experiment on finding learning rate schedules for Adam.	86
6.11	Results for all optimizers and learning rate schedules for CIFAR-10, CIFAR-100, Flowers102 (Flowers), Cars196 (Cars), Caltech101 (Caltech), and TinyImageNet (Tiny). The mean and standard deviation of the test accuracy across 5 independent runs are reported. Red indicates Top 3 performance, while bold indicates Top 8 performance.	88
6.12	Results for all optimizers and learning rate schedules for PTB. The mean and standard deviation of the best validation perplexity is reported (lower is better) across 5 independent runs are reported. Red indicates Top 3 performance, while bold indicates Top 8 performance.	104
7.1	The proposed set of unary and binary operations.	109
7.2	Results for training from scratch (S_i is ours). Four standard learning rate schedules, along with the best found learning rate schedules from Chapter 6 (LR_i) are shown for comparison.	117
7.3	Results for fine-tuning (F_i is ours). Four standard learning rate schedules, along with the best found learning rate schedules from Chapter 6 (LR_i) are shown for comparison.	118

7.4	Rank results for training from scratch (S_i is ours) and fine-tuning (F_i is ours). Four standard learning rate schedules, along with the best found learning rate schedules from Chapter 6 (LR_i) are shown for comparison.	119
7.5	The proposed set of scheduled decay functions. Where r controls the number of restarts, $base$ was set to 0.9999 for training from scratch and 0.99925 for fine-tuning, t is the current time-step, and T is the total number of timesteps.	123
7.6	Genotype equations for the final 10 learning rate schedules for both training from scratch and fine-tuning	124
8.1	The seven final discovered output activation function equations (A_i), along with eight constructed functions (B_i).	130
8.2	Results for ResNet52v2 on the CIFAR and TinyImageNet datasets. The Top 3 best mean test accuracies for each dataset is highlighted in bold.	131
8.3	Results for EfficientNetB0v2 on the CIFAR and TinyImageNet datasets. The Top 3 best mean test accuracies for each dataset is highlighted in bold.	131
9.1	Results for ResNet52v2 on CIFAR-10 for each loss function using A_3	139
9.2	Results for ResNet52v2 on CIFAR-10 for each learning rate schedule using $A_3 + NL_1$	140
9.3	Results for ResNet52v2 on CIFAR-100 for the best combination.	140
9.4	Results for ResNet52v2 on TinyImageNet for the best combination.	141

Chapter 1

Introduction

Artificial neural networks derive from a sub-field of *artificial intelligence* known as *computational intelligence*. Since the advent of artificial neural networks, the search for how to improve performance has long been a source of discussion (Dubrawski, 1997; Yuan and Fine, 1998; Choo, 2000; Chen et al., 1993). Neural networks contain many hyper-parameters that directly influence training quality. Classical heuristic methods, such as grid search and hill-climber, have been applied to tuning these parameters (Alvaro et al., 2009; Battiti and Tecchiolli, 1995; Chalup and Maire, 1999). However, most historically, *genetic algorithms* from the field of *evolutionary computation* have been applied with great success (Leung et al., 2003; Jones, 1993; Montana et al., 1989; Whitley et al., 1990; McInerney and Dhawan, 1993; Schaffer et al., 1992; Branke, 1995). The inspiration for using genetic algorithms for tuning neural networks derive from the very connection of biological principles between the two processes, as both try to model biological phenomena (Signorelli et al., 2016).

Genetic algorithms have long been applied to tuning the architecture of neural networks. The process of searching for the architecture of a neural network has

been referred to as *neural architecture search* (Elsken et al., 2019). However, the searching process for a particular component of a neural network can be abstracted away to a more general form, to what can be referred to as *neural component search*. In this work, I focus on performing neural component search, from biological principles, on the *learning components* of a neural network: loss function, optimizer, learning rate schedule, and output activation function. The learning components refer to components of neural networks that directly affect learning through the update equation for the weights of the model.

The loss function directly controls how to calculate the gradient with respect to the model and input. The optimizer controls how to update the weights of the model from the gradient of the loss. The learning rate schedule controls the magnitude of the update for the optimizer. Lastly, the output activation function controls the final output distribution from which the loss function is applied and obtained. These four important, and pivotal, components can be searched upon for improved performance.

The organization of this dissertation adheres to the following layout. Chapter 2 covers the background knowledge to understand the essentials of this work. Chapter 3 gives the motivation for the thesis behind this work. Chapter 4 covers the proposed search space and genetic algorithm used throughout this work. Chapter 5 introduces *neural loss function search*. Chapter 6 introduces *neural optimizer search*. Chapter 7 introduces *neural learning rate search*. Chapter 8 introduces *neural output activation function search*. Chapter 9 covers the combination of the best found found components from each search applied to the CIFAR datasets. Lastly, Chapter 10 summarizes the research, contributions, and results. All neural component search chapters are listed in chronological order from when they were performed.

Chapter 2

Related Work

The objective of an *artificial neural network* (ANN) is to model biological neural networks found in the brain. It is believed that in doing so, *artificial intelligence* can be created for a specific problem, where artificial intelligence in this regard can be defined as a deep learning model capable learning to perform a task. ANNs have been shown to be extremely successful in capturing non-linearity for regression and classification problems. Historically used alongside ANNs are *genetic algorithms* (GA). The objective of a GA is to model biological evolution as a means to optimize a non-differentiable function. A common non-differentiable objective that needs tuning is the architecture of a neural network, giving rise to *neural architecture search*.

2.1 Artificial Neural Networks

In this section, the basic overview of ANNs will be covered through the discussion of perceptrons and dense models. In addition, the overview of convolutional neural networks and neural architecture search will be discussed as well.

2.1.1 Foundation

ANNs are composed of layers of perceptrons. A perceptron is defined to be the weighted summation of the product between its input and weights, plus a bias term. An activation function is then applied to achieve non-linearity. Historically, the Sigmoid function has been applied for single perceptions. Equation 2.1 gives the output of a single perceptron, where w is the trainable weight, x is the input, b is the bias term, and σ is the Sigmoid function. Equation 2.2 gives the Sigmoid function.

$$a = \sigma(b + \sum_{i=1}^n x_i w_i) \quad (2.1)$$

$$\sigma(x) = \frac{1}{1 + e^{-x}} \quad (2.2)$$

Although perceptrons are powerful non-linear models, they struggle learning harder representations. *Dense neural networks* are ANNs that are composed of multiple layers of perceptrons. Through the combination of many different perceptrons, dense networks are capable of modeling even more non-linearity. Dense ANNs are composed of three main components. An input layer, a number of *hidden layers*, and an output layer. Information is fed to the ANN through the input layer, passed along to the hidden layers to capture non-linearities, and finally sent to the output layer. Equation 2.3 details the output of node j in an arbitrary layer, where A is the selected activation function. For hidden layers, the chosen activation function is usually ReLU (Glorot et al., 2011) or Swish (Ramachandran et al., 2017). Equations 2.4 and 2.5 give the function equations for ReLU and Swish. When one or more dense layers are constructed together,

the model is typically referred to as a *deep learning model*, where deep learning refers to the complexity of the model (i.e. the more layers, the more complexity).

$$a_j = A(b_j + \sum_{i=1}^n x_{ij}w_{ij}) \quad (2.3)$$

$$ReLU(x) = \max(x, 0) \quad (2.4)$$

$$Swish(x) = x\sigma(x) \quad (2.5)$$

2.1.2 Convolutional Neural Networks

For *computer vision* tasks, images are fed into the deep learning model for a specific task. However, dense networks are incapable of modeling the spatial and temporal representations of images. Images are three dimensional objects with a height, width, and RGB channel axis (temporal). This representation has to be flattened into a single vector for a standard dense network. *Convolutional neural networks* (CNNs) were designed to eliminate this problem through the inclusion of the convolution kernel. Like the input image, the convolution kernel is a three dimensional object that is multiplied to a subsection of the input image. The kernel then slides over to next patch of pixels of the image to obtain another product. The exact inter-workings of the convolution kernel are not of importance to this work, as I will only be using state-of-the-art CNN models for computer vision tasks. For more information on CNNs, I will reference my previous work (Morgan, 2022).

Architectures

Many different CNN architectures have been proposed in order to improve performance (Younesi et al., 2024). However, in this work, I mainly focus on four state-of-the-art architectures that are commonly used in computer vision tasks: EfficientNetV2Small (20.5M parameters) and EfficientNetV2B0 (5.3M parameters) (Tan and Le, 2021), WideResNet28-10 (36.5M parameters) (Zagoruyko and Komodakis, 2016), SEResNeXt4x64d (20.2M parameters) (Hu et al., 2018), and ResNet52v2 (15.7M or 23.5M parameters) (He et al., 2016).

2.1.3 Neural Architecture Search

The architecture of a neural network has long been a source of discussion. Automated methods from various fields of discipline have been applied to find the best architecture, commonly referred to as *neural architecture search* (NAS). In fact, over 1000 papers have been published on NAS alone (Elsken et al., 2019). NAS systems are composed of three primary components. First, a *search space* is defined to describe the type of architectures that can be searched. Second, a *search algorithm* is defined to describe how the architectures will be searched upon. Lastly, a *fitness function* (or *surrogate function*) is defined as a means to evaluate candidate architectures. In this work, for each experiment, all three of these components will be discussed in detail. Most important to this work is the NASNet search space (Zoph et al., 2018), a fundamental search space in the field in NAS. The NASNet search space is a computational graph composed of free floating nodes that allows for arbitrary connections and operations. The NASNet search space will serve as the backbone search space in this work from which all adaptations will arise. More on this search space will be discussed.

2.2 Genetic Algorithms

A genetic algorithm (GA) seeks to model the biological process of evolution to optimize a given objective function. Evolution can be described as a process where individuals become more fit to their environment through three main mechanisms: natural selection, differential reproduction (commonly called *crossover*), and mutation. Natural selection occurs when individuals poorly designed for their environment have lower probabilities of reproducing. Differential reproduction after natural selection as those only fit enough to survive are able to reproduce. Lastly, mutation randomly occurs in the genome. GAs leverage these concepts in an automated fashion in order to optimize a given problem. Figure 2.1 gives the workflow overview of a general genetic algorithm. Starting from a random population, the individuals are evaluated using the fitness function, a select few are chosen for reproduction, and a select few are chosen for survival. This process repeats until termination.

2.2.1 Genetic Programming

In this work, the most common type of GA is known as *genetic programming* (GP). GP is composed of a tree-based grammar. A tree-based grammar follows a set of rules for its construction and is composed of internal and leaf nodes. Figure 2.2 gives the construction of two different GP trees. The leaf nodes are the input to the tree (x_1 and x_2) and the internal nodes are nodes that perform an operation. Figure 2.3 shows an example of crossover by swapping the connections between two parent trees (top) to obtain two child trees (bottom). Figure 2.4 shows three examples of mutation on a tree. The top left gives the original tree. The top right shows an example of mutating the operation by changing it. The bottom

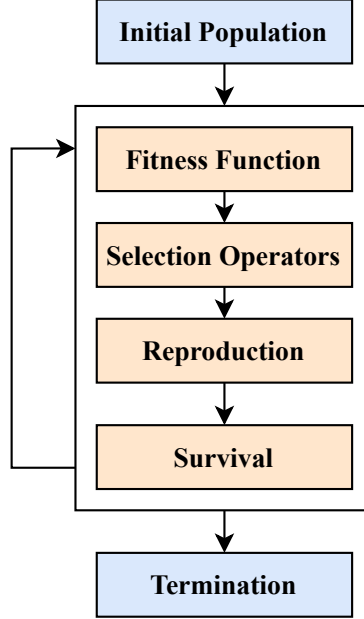


Figure 2.1: General workflow of a genetic algorithm.

left shows an example mutation by inserting an internal node. The bottom right shows an example of mutation by deleting an internal node. Through crossover and mutation, the GA can explore the tree based search space.

2.2.2 Regularized Evolution

In the original NASNet paper, reinforcement learning was applied to a recurrent neural network controller that predicted the architectures. Shortly thereafter its publication, Real et al. (2019) applied a GA to the NASNet search space with

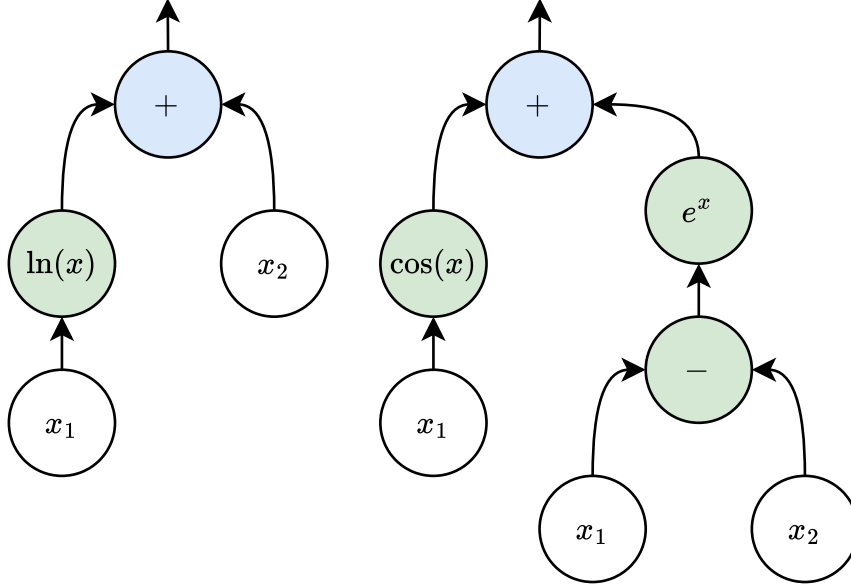


Figure 2.2: Two example trees for GP.

great success. The GA, called *regularized evolution*, was a mutation only, aging, algorithm. The NASNet search space does not easily allow for crossover, due to many connections inside the graph, hence the mutation only portion. The aging component was included as training ANN is stochastic due to the random initialization, batch splitting, and stochastic regularization components. As my proposed search space for neural component search was derived from the NASNet search space, and regularized evolution achieved great success on the search space, this GA was used as the base search algorithm throughout this work. Algorithm 1 gives a brief overview of the algorithm, where Tournament refers to tournament selection, a type of selection operator that chooses the best individual from a random sample of size n .

Algorithm 1 Regularized Evolution

```
1: population = RandomInitialization()
2: validationAcc = TrainModel(population)
3: repeat
4:   winner = Tournament(population, validationAcc, n)
5:   child = Mutation(winner)
6:   valAcc = TrainModel(child)
7:   population = ReplaceOldest(population, child)
8:   validationAcc = Update(validationAcc, valAcc)
9: until Time Expired
```

2.3 CIFAR

In computer vision tasks, it is necessary to have a common task to compare different architectures and methodologies. These common tasks are referred to as *benchmark* tasks. In NAS, the CIFAR-10 and CIFAR-100 datasets are the most common benchmark datasets in computer vision for comparing models. Due to this tradition, and ease of complexity in training models, I chose to compare all of my discovered components on the CIFAR datasets (Krizhevsky and Hinton, 2009). The CIFAR datasets are composed of 32x32 colored images, where CIFAR-10 covers 10 possible classes and CIFAR-100 covers 100 possible classes. However, evaluations were not limited to only the CIFAR datasets. Those that did not are detailed in their respective sections.

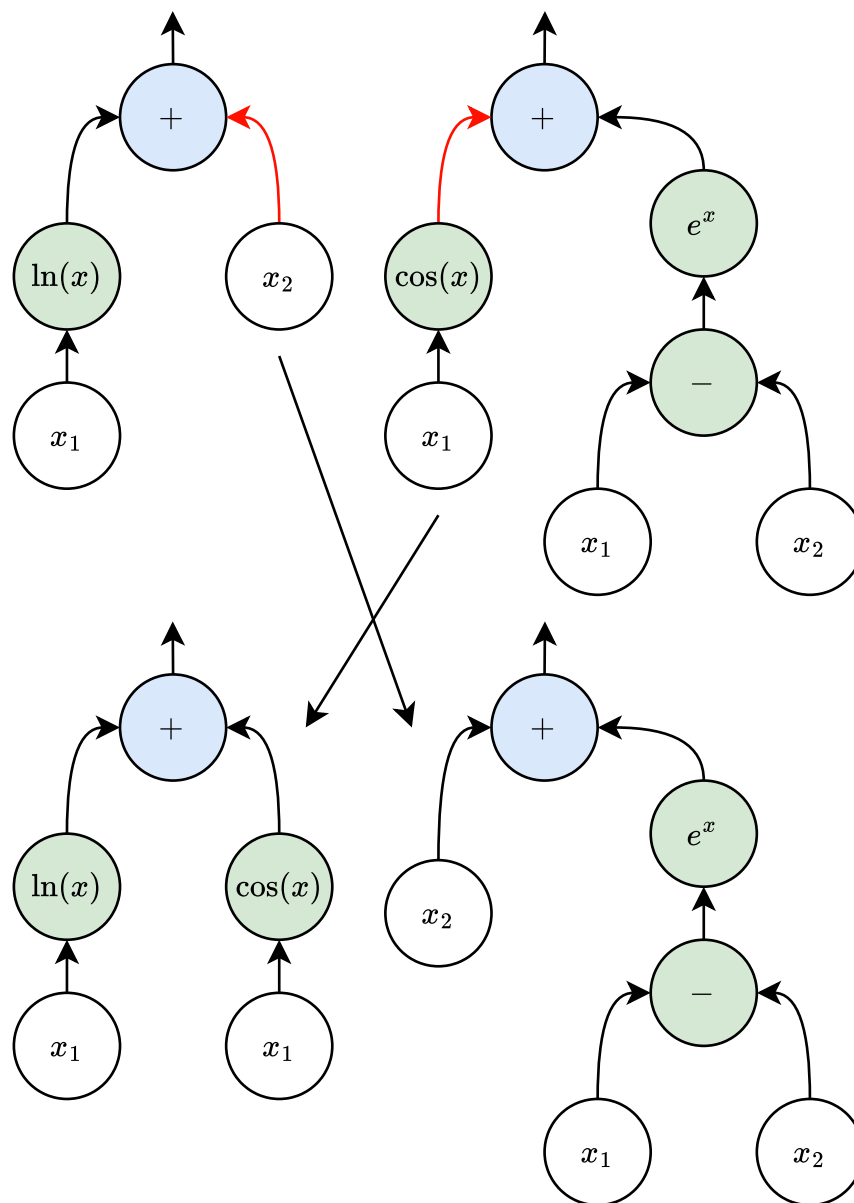


Figure 2.3: Three examples of mutation for GP.

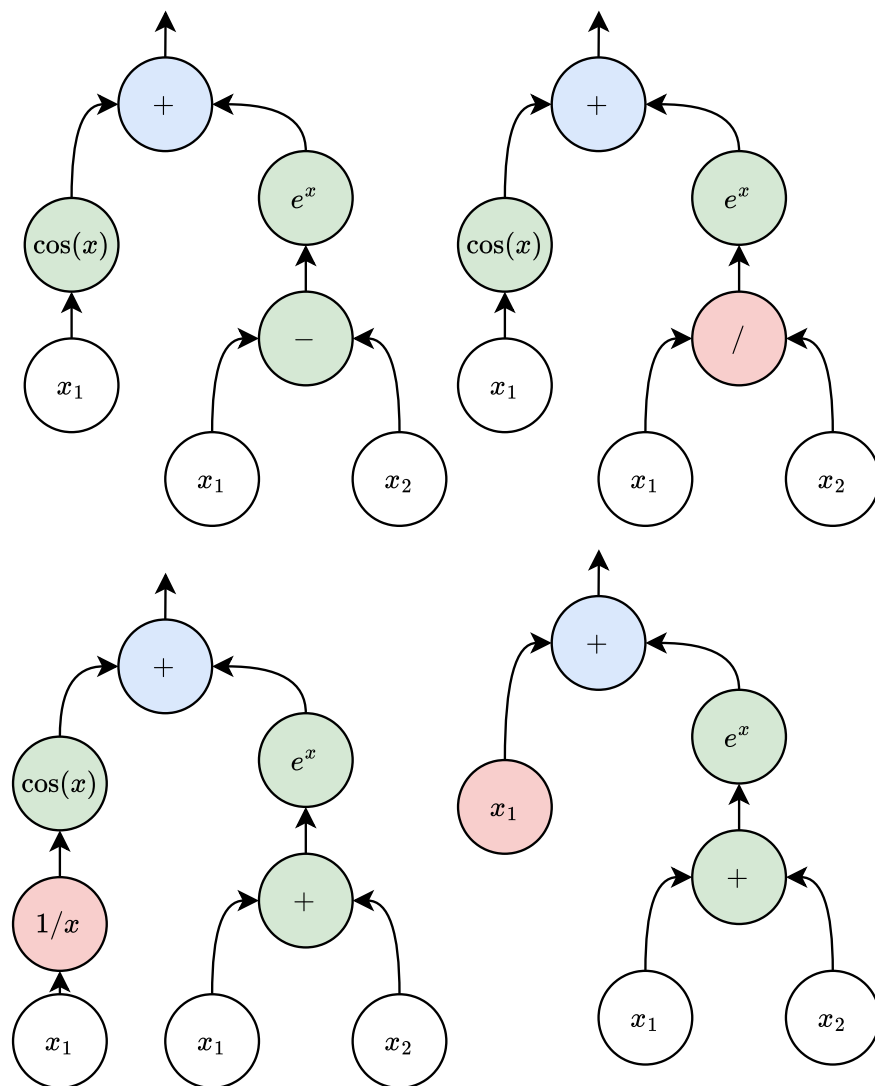


Figure 2.4: Three examples of mutation for GP.

Chapter 3

Motivation

In this section the motivation behind the work is discussed through the lens of *backpropagation*, the learning algorithm of deep learning models.

How do deep learning models learn? Deep learning models learn for a specific task by minimizing an *error function*. An error function, commonly referred to as a *loss function*, is a function that determines how correct or incorrect the outputs of the model are given some input. For classification, deep learning models output a probability distribution for a given set of classes. Historically, the Sigmoid function has been used for modeling binary classification problems as it naturally arises from the formulation of the posterior probability distribution for generalized linear models (Bishop, 2006). However, generalized linear models are not limited to the Sigmoid distribution. For multi-class classification, the Softmax activation also naturally arises due to its multivariate generalization of Sigmoid. Equation 3.1 gives the output of the Softmax activation, where i is the Softmax output for the i^{th} node. In order to create a probability distribution, meaning all probabilities add up to one, the output for each class is scaled the sum of all classes.

$$\text{Softmax}(x_i) = \frac{e^{x_i}}{\sum_j e^{x_j}} \quad (3.1)$$

After applying the Softmax activation to the output of the last layer of the deep learning model, a loss function needs to be applied to measure how correct or incorrect is the models estimation. From information theory, *entropy*, *Kullback-Leibler divergence* (KL), and *cross-entropy* emerged early on as common loss functions to measure the *information*, or randomness, of either one or two probability distributions. The full mathematical relationship between the metrics can be shown below in Equation 3.2, where P and Q are probability distributions, $CE(P, Q)$ is cross-entropy, E is entropy, and KL is KL. In machine learning, P is the distribution of the model's output, and Q is the target distribution. Therefore, KL divergence is the difference between the cross-entropy of the model's output distribution and target distribution, and the randomness of the model's output distribution. In practice, cross-entropy is selected more often than KL, as deep learning models calculate the *loss* over a sampled mini-batch, rather than the population. For the rest of this work, y will always refer to the target output, $\hat{y}$ will refer to the model's prediction. Using this new notation, Equation 3.3 depicts the cross entropy loss ($\mathcal{L}$).

$$\begin{aligned} CE(P, Q) &= KL(P||Q) + E(P) \\ -\sum_i q(x_i) \ln(p(x_i)) &= -\sum_i p(x_i) \ln\left(\frac{p(x_i)}{q(x_i)}\right) - \sum_i p(x_i) \ln(p(x_i)) \end{aligned} \quad (3.2)$$

$$\mathcal{L} = -\sum_i y_i \ln(\hat{y}_i) \quad (3.3)$$

After applying the cross-entropy loss function to the output of the Softmax layer of a deep learning model, a measure of error can be obtained. *Learning* can be defined as the process of minimizing the error of a loss function through the update of the trainable weights from a model. A deep learning model is composed of layers of perceptrons, where each perceptron contains trainable slopes (w_i) and a trainable intercept (b). These weights can be updated in such a way that minimizes the error of the loss function. From calculus, the minimization of a function occurs by moving towards negative of the gradient for the function. For deep learning models, the same process is applied. The simplest algorithm to do so is known as *stochastic gradient descent* (SGD). SGD updates the weights of a model through Equation 3.4, where w^l is the weight matrix of layer l , i is the i^{th} iteration of the algorithm, α_i is the learning rate (controls the step size), and $\frac{\partial \mathcal{L}}{\partial w_l}$ is partial derivative of the loss function with respect to the weight.

$$w_i^l = w_{i-1}^l - \alpha_i \frac{\partial \mathcal{L}}{\partial w_l} \quad (3.4)$$

The partial derivative of the loss function with respect to the weight can be calculated through the chain rule. Assuming w^l is the weight matrix of the last layer of the model, the output of the model can be shown to be $\hat{y} = a^l = \text{Softmax}(z^l) = \text{Softmax}(w^l a^{l-1} + b^l)$, where z^l is an intermediate expression, a^{l-1} is the output of the previous layer. The loss function is then applied. Because the loss function is a product of the output activation function, which is a product of its input, which is the product of the weights, the chain rule of calculus must be applied to obtain the result shown in Equation 3.5.

$$\begin{aligned}\frac{\partial \mathcal{L}}{\partial w_l} &= \frac{\partial \mathcal{L}}{\partial \hat{y}_l} \frac{\partial \hat{y}_l}{\partial w_l} \\ &= \frac{\partial \mathcal{L}}{\partial \hat{y}_l} \frac{\partial \hat{y}_l}{\partial z_l} \frac{\partial z_l}{\partial w_l}\end{aligned}\tag{3.5}$$

The partial derivative of cross-entropy with respect to z_l , is given in Equation 3.6.

$$\frac{\partial \mathcal{L}}{\partial \hat{z}_l} = - \sum_i \frac{y_i}{\hat{y}_i} \frac{\partial \hat{y}_i}{\partial z_l}\tag{3.6}$$

The partial derivative of the Softmax function for node i with respect to its input, z , is given in Equation 3.7, where $\delta_{ij} = 1$ when $i = j$ and $\delta_{ij} = 0$ when $i \neq j$.

$$\frac{\partial \hat{y}_i}{\partial z_l} = \hat{y}_i(\delta_{ij} - \hat{y}_j)\tag{3.7}$$

The partial derivative of z_l with respect to the weights is given in Equation 3.8

$$\frac{\partial z_l}{\partial w_l} = a^{l-1}\tag{3.8}$$

Using the combination of Equations 3.6, 3.7, and 3.8, the partial derivative of the loss function with respect to the weights of the last layer is given in Equation 3.9. The derivative is the difference between the models prediction for the class and the target value multiplied by the output of the previous layer.

$$\frac{\partial \mathcal{L}}{\partial w_l} = (\hat{y}_i - y_i) * a^{l-1}\tag{3.9}$$

Using SGD, the weights of the last layer are updated through Equation 3.10.

$$w_i^l = w_{i-1}^l - \alpha_i(\hat{y}_i - y_i) * a^{l-1} \quad (3.10)$$

As one can see, the update equation is heavily influenced by all four learning components. First, the Softmax activation layer creates a probability distribution for classification. Second, the loss function then measures the amount of error between the probability distribution and the target. Third, the optimizer (SGD in this example) details how the weights are to be updated given the gradient of cross-entropy through the Softmax layer. Fourth, the learning rate schedule controls how far the optimizer should step towards given this information. These four learning components are vital for overall performance. Although these components have been derived from mathematical and statistical principles, the success of AutoML and NAS, and an increase in available computational resources, suggests to *Improve Learning Through Neural Component Search*.

Chapter 4

Proposed Search Space

4.1 Overview

The majority of the experiments performed in this work can be abstracted out as they share similar components. Specifically, the search space and search algorithm. All experiments included in this work use the same search space, a computational graph based off the NASNet search space. All experiments included in this work use either *regularized evolution* or my proposed *particle based mutation only* genetic algorithm.

4.2 Search Space

Although genetic programming is a common way to formulate a search space, tree-based grammars are known to suffer two primary issues: bloating (Luke and Panait, 2006) and redundancy (Neshatian and Zhang, 2009). *Bloating* occurs when the average size of a tree within the population begins to grow uncontrollably. The NASNet search space inherently prevents this as the maximum number

of nodes is prescribed prior to evolution, thus limiting the maximum size of a cell. *Redundancy* occurs in tree-based grammars when parts of the tree contain redundant features, which can lead to excessively large trees. Although redundancy is not inherently a problem, it signifies that a particular expression is needed in multiple places, which tree-based grammars must evolve independently. The NASNet search space inherently solves this problem as hidden state nodes are able to be reused multiple times within the computational graph, thus allowing for redundant features without increasing the number of nodes, thereby speeding up the evolution to acquire such redundant features.

The derivative of the NASNet search space proposed in this work has three components: (1) *input connections*, which are analogous to leaf or terminal nodes for tree-based grammars: y , $\hat{y}$, 1, and -1 , where y is the target output and $\hat{y}$ is the model’s prediction; (2) *hidden state nodes*, which are free-floating nodes able to use any other hidden node or input connection as an argument; and (3) a specialized *output node*, defined to be the final operation before outputting the loss value. All nodes use either a unary or binary function.

The 27 proposed unary operations are listed in Table 4.1, where bessel_{ij} is the Bessel function of degree j ; e denotes exponential; σ is sigmoid activation; softsign is softsign activation; softplus is softplus activation; erf is the Gauss error function; erfc is the complementary error function; $\frac{d}{dx}$ refers to the derivative of the respective function; and $\epsilon = 10^{-7}$ for numerical stability. The seven proposed binary operations are listed in Table 4.2. All operations were chosen from previous related works and available operations in Tensorflow (Abadi et al., 2015). The goal of increasing the number of possible operators is to create an expansive search space, allowing for more expression.

Unless otherwise stated, all search spaces used in this work are based off this

UNARY OPERATIONS	
$-x$	$\ln(x + \epsilon)$
e^x	$\log_{10}(x + \epsilon)$
$\sigma(x)$	$\frac{d}{dx}\sigma(x)$
$\text{softsign}(x)$	$\frac{d}{dx}\text{softsign}(x)$
$\text{softplus}(x)$	$\text{erf}(x)$
$\text{erfc}(x)$	$\sin(x)$
$\sinh(x)$	$\text{arcsinh}(x)$
$\tanh(x)$	$\frac{d}{dx}\tanh(x)$
$\text{arctanh}(x)$	$1/(x + \epsilon)$
$ x $	x^2
$\text{bessel}_{i0}(x)$	$\sqrt{x}$
$\text{bessel}_{i1}(x)$	$\max(x, 0)$
$\text{bessel}_{i1e}(x)$	$\min(x, 0)$
$\text{bessel}_{i0e}(x)$	

Table 4.1: The proposed set of unary operations.

BINARY OPERATIONS	
$x_1 + x_2$	$x_1 * x_2$
$x_1 - x_2$	$x_1/(x_2 + \epsilon)$
$x_1/\sqrt{1 + x_2^2}$	$\max(x_1, x_2)$
$\min(x_1, x_2)$	

Table 4.2: The proposed set of binary operations.

representation. Figure 4.1 gives an example construction of a loss function using the described search space.

4.3 Particle-Based Genetic Algorithm

For reasons that will be discussed later, regularized evolution was incapable of thoroughly exploring different component search spaces. In order to better allow for exploration, while preserving the major components of regularized evolution, I propose a *particle-based genetic algorithm* that allows for massive parallelization.

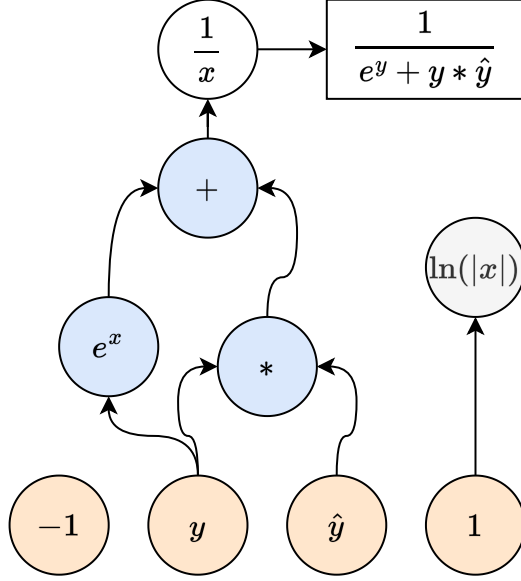


Figure 4.1: Example loss function with three active (blue) hidden state nodes, one inactive hidden node (grey), and one root node (white). The final loss equation is given next to the root node in the box.

These components are (1) mutation only, (2) aging, and (3) parallelism.

In this algorithm, n independent particles run for t time-steps, where at each time-step k random mutations are performed. At each timestep, each particle is mutated k times (mutation only) and the best mutated child is selected as the next position for the particle (aging). Each particle acts independently so that the available search space around each particle can be quickly explored and exhausted. The proposed GA allows for embarrassing parallelization (parallelism) at two levels. First, because each particle is independent from the other, as there is no competition, each particle can be parallelized. Second, because each particle performs k independent mutations at each timestep, each mutated child can be parallelized. Therefore, the only bottleneck in our proposed GA is t , the total number of timesteps. If $n*k$ GPUs are available, the running time of the algorithm

is $O(t * c)$ where c is the mean cost to train a neural network. Pseudocode for the proposed particle based GA is given in Algorithm 2.

Algorithm 2 Particle-Based Genetic Algorithm

```

1: Input:  $n$  (Number of Particles),  $k$  (Number of mutations),  $t$  (Number of
   timesteps)
2:  $initialParticles = \text{Initialization}()$ 
3:  $validationAccuracies = \text{FitnessFunction}(initialParticles)$ 
4:  $particles = \text{BestParticles}(initialParticles, validationAccuracies, n)$ 
5: for  $particle$  in  $particles$  do
6:   for  $timestep$  in  $t$  do
7:      $mutations = []$ 
8:     for  $j$  in  $k$  do
9:        $mutations.append(\text{Mutation}(particle))$ 
10:    end for
11:     $mutationValidationAccuracies = \text{FitnessFunction}(mutations)$ 
12:     $particle = \text{BestParticle}(mutations, mutationValidationAccuracies)$ 
13:  end for
14: end for

```

Chapter 5

Neural Loss Function Search

5.1 Overview

Convolutional neural networks (CNNs) trained for classification problems on balanced datasets are typically evaluated using accuracy. However, CNNs learn in these scenarios by minimizing cross-entropy, a loss metric from information theory that measures how similar two probability distributions are concerning a random variable. Minimizing cross-entropy reduces the sum of entropies across each discrete class between the actual distribution and the prediction. By minimizing this distributional difference, it is believed that the model's output will highly correlate with the evaluation metric, accuracy. In practice, this typically holds true; however, a perfect linear mathematical relationship between the two has not been shown. As a result, other possible loss functions could exist that might yield better accuracy than cross-entropy when minimizing. Searching for other candidate loss functions can be referred to as *neural loss function search* (NLFS).

5.2 Specific Related Work

NLFS is a new but rapidly growing domain (Chebotar et al., 2019; Gonzalez and Miikkulainen, 2020, 2021; Liu et al., 2021; Gao et al., 2021, 2022; Gu et al., 2022; Li et al., 2022; Rayman et al., 2022), that originated from the success of neural architecture search (NAS) (Signorelli et al., 2016; Zoph and Le, 2016; Zoph et al., 2018; Real et al., 2019; Elsken et al., 2019) and AutoML (He et al., 2021). NLFS has been explored for various tasks using two primary representations: computational graphs (Gonzalez and Miikkulainen, 2020; Liu et al., 2021; Gu et al., 2022; Li et al., 2022; Rayman et al., 2022) and Taylor polynomials (Gonzalez and Miikkulainen, 2021; Leng et al., 2022). Here, I focus on computational-graph-based representations.

Gonzalez and Miikkulainen (2020) evolved a loss function for classification and regression using genetic programming (GP); the function’s coefficients were then fine-tuned using covariance-matrix adaptation evolutionary strategy (CMA-ES). Liu et al. (2021) evolved a loss function specifically for object detection. Rayman et al. (2022) found a loss function for classification problems using both GP and a gradient-based local search algorithm. Instead of using CMA-ES to evolve the coefficients of the loss function, they performed meta-learning on the coefficients through gradient descent by comparing with cross-entropy. Li et al. (2022) evolved a general loss function for generic tasks in semantic segmentation, object detection, and instance segmentation using GP. Gu et al. (2022) evolved margin-based softmax loss functions for the specific task of person re-identification using GP.

5.3 Contribution

In this chapter, I perform two experiments of NLFS for image classification. In the first, I conduct NLFS for image classification, rather than object detection (Liu et al., 2021), image segmentation (Li et al., 2022), or person re-identification (Gu et al., 2022). I expand on previous image classification NLFS work (Gonzalez and Miikkulainen, 2020; Rayman et al., 2022) by (1) greatly increasing the allowed operations, (2) using a computational-graph representation rather than grammar-based GP, (3) ensuring the surrogate function transfers effectively to large-scale CNNs with state-of-the-art image regularization techniques, (4) not biasing the loss functions toward cross-entropy through meta-learning, and (5) discovering a set of loss functions, termed *NeuroLoss* functions, that were extremely competitive to cross-entropy across multiple datasets and architectures. In the second, I conduct NLFS for specific image-augmentation techniques using the same procedure discussed in the first experiment. I expand upon previous image classification NLFS work by (1) revealing that different loss functions can perform well on certain image augmentation techniques, while performing poorly on others, (2) exploiting this disparity by performing an evolutionary search on five types of image augmentation techniques in the hopes of finding image augmentation specific loss functions, and (3) discovering a new loss function, termed the *inverse besse*l *logarithm* loss, that performed well across all image augmentation techniques and was extremely competitive with cross-entropy.

5.4 Shared Methodology

Both NLFS endeavors, the first being referred to as the NeuroLoss evolution and the second being referred to as the image-augmentation specific evolution, share many of the same components. For brevity, in this section, the shared methodology for the NeuroLoss evolution and image-augmentation specific evolution are explained. Namely, the proposed loss function initialization, proposed loss function integrity check, genetic algorithm and software implementation details.

5.4.1 Loss Function Initialization

Each loss function was based on the search space detailed in Chapter 4. Each computational graph contains four hidden state nodes and one output node. Each node is initialized by sampling its operation uniformly from the set of all 34 operations. Each connection for each hidden state node is sampled uniformly from all input connections and hidden state nodes. The connections for the root node are determined such that at least one connection is from a hidden state node to prevent the output from being a simple binary or unary operation on the input.

5.4.2 Integrity Check

With the inclusion of many new operations in the proposed search space, and the proposal of free-floating hidden state nodes, the total search space contains a lower bound of 10^{10} possible combinations. Although the search space allows for the exploration of many potential good loss functions, it also allows for many degenerate loss functions. To prevent degenerate loss functions from wasting valuable evaluation time, an integrity check was implemented, similar to that by Liu et al. (2021). If a loss function failed an integrity check, that function was

re-initialized using the procedure described in Section 5.4.1. The integrity check contains four components.

First, the computational graph containing the hidden state nodes for each loss function is checked for cycles. If a cycle is found, the loss function is rejected. In addition, if y or $\hat{y}$ is not included at least once in the graph, it is also rejected.

Second, if the first integrity test is passed, the phenotype of the loss function is acquired for a simple binary case. The *phenotype of the loss function* refers to the function value of the loss for given prediction and target arrays, specifically when $y = 1$ in this case. The phenotype of cross-entropy, normalized between 0 and 1, using this evaluation is plotted in Figure 5.1. As one can see, the loss value for cross-entropy decreases drastically as $\hat{y} \rightarrow 1$, signifying the global minimum, the place where the input-target pairs match. A loss function for a binary case should contain only one global minimum, as there should exist only one solution that minimizes the loss function for all input. Cross-entropy exemplifies this as it is a monotonically decreasing function, as one can see by the figure. To eliminate multi-modal loss landscapes, all loss functions containing more than one global optimum for a binary case are rejected. As a result, the remaining loss functions are either monotonic or roughly parabolic. If the loss function is roughly parabolic, it is rejected if its global optimum is within 1% of the $\hat{y}$ value 0.5 to prevent global minimum values near 0.5, the threshold for accuracy.

Third, the integrity check flips the sign of the loss function if it is either monotonically increasing or an upside-down parabola. This was performed to prevent *inverse learning*—learning an output of 0 for a target output of 1, and an output of 1 for a target output of 0. This simple adjustment inherently speeds up evolution by not requiring loss functions to evolve the proper signage for the root node.

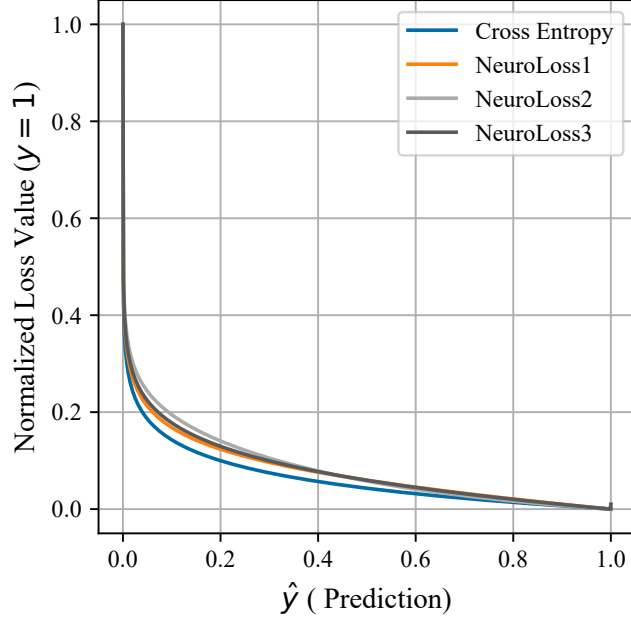


Figure 5.1: Overview of the binary phenotypes of cross-entropy, NeuroLoss 1, NeuroLoss 2, and NeuroLoss 3.

Fourth, the phenotypes of all existing loss functions within the population, and the current loss function, are normalized between 0 and 1. All normalized phenotypes from the population are looped over and the the Euclidean norm of the difference between each normalized phenotype and the current normalized loss function are calculated. If a single normalized difference was below a threshold of 0.01 the loss function was rejected for being too similar to another existing loss function. This was performed to allow for ample exploration of non-explored regions of the search space without simply scaling already existing loss functions. However, repeat loss functions were allowed during evolution as long as the copy died off from within the population, a benefit to the aging mechanism.

After passing all four components of the integrity check, each loss function is evaluated using the proposed surrogate function. The final algorithm for the

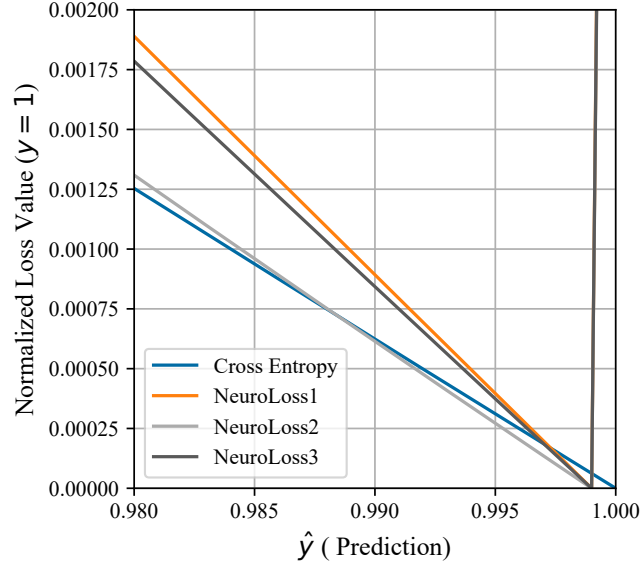


Figure 5.2: Zoomed in view (around $\hat{y} \rightarrow 1$) of the binary phenotypes of cross-entropy, NeuroLoss 1, NeuroLoss 2, and NeuroLoss 3.

integrity check is detailed in Algorithm 3.

5.4.3 Genetic Algorithm

A population-based genetic algorithm (GA) is used to explore the proposed search space. Unlike grammar-based search spaces, my proposed NASNet derivative does not allow for an easy implementation of crossover (recombination), as the hidden state nodes can occur in any order and can connect to any other node. As a result, a mutation-only GA is used. Given the success of regularized evolution on the NASNet search space, I use regularized evolution for my search space. *Regularized evolution* is a mutation-only aging GA that selects a parent through tournament selection from the population. This parent is mutated to obtain a child loss that is evaluated and added back to the population by replacing the oldest. Because training neural networks is highly stochastic, and in order to

Algorithm 3 Integrity Check

```
1: Input: Candidate Loss  $loss$ , Population  $population$ 
2: if ContainsCycle( $loss$ ) == True then
3:   Return False
4: end if
5: if ContainsYandYHat( $loss$ ) == False then
6:   Return False
7: end if
8: if ContainsOneGlobalMinOrMax( $loss$ ) == False then
9:   Return False
10: end if
11: CheckSign( $loss$ )
12: for  $individual$  in  $population$  do
13:   if Norm( $individual - loss$ ) < 0.01 then
14:     Return False
15:   end if
16: end for
17: Return True
```

prevent pre-convergence to local minima and allow for ample exploration, aging is incorporated for better exploration of the search space as each individual has a maximum life span within the algorithm. This is inherently performed in the GA as the new child replaces the oldest.

Mutation was performed by uniformly sampling either the root node or one of the hidden state nodes, then changing either its operation or connection. After mutation, an integrity check (Section 5.4.2) is performed to ensure a successful mutation. If the integrity check fails, mutation is performed again. If 2500 mutations fail in a row, the mutation does not take place, and another iteration of the algorithm is performed using the procedure just discussed.

Mutation was performed by selecting either the root node or one of the hidden state nodes uniformly randomly. Once a node is chosen, either the operation or connection is changed. With 70% probability, the operation is changed either

from unary to unary, or binary to binary, depending on the original operation. With 15% probability, the operation was swapped from unary to binary, or binary to unary, depending upon the original operation. If the original operation was unary, the extra connection obtained from swapping to binary was randomly selected from the set of input and hidden state nodes. If the original operation was binary, then either the left or right connection was eliminated with 50% probability. If the operation was not changed or swapped, the remaining 15% probability went to changing the connection of the node. If the operation was unary, then the connection was changed uniformly random from the set of other possible connections. If the operation was binary, then with 20% probability the argument connections were swapped; else, either were changed uniformly randomly.

The child loss was then trained using the surrogate function. Despite the integrity check performed when creating the loss functions, losses can still not learn during training. To prevent wasting computation, within the surrogate function an early stopping mechanism was incorporated to stop the learning process for degenerate losses. If this occurred, another child loss was obtained through mutation. If seven mutated child loss functions in a row failed integrity checks, the mutation procedure was considered to be failed, and another iteration of the algorithm was performed. The entire genetic algorithm used in this work is detailed in Algorithm 1.

5.4.4 Further Implementation Details

All architectures, training strategies, and loss functions were implemented in Tensorflow (Abadi et al., 2015). The code base for the EffNetV2Small model and

Algorithm 4 Genetic Algorithm

```
1: initialPopulation = LossInitialization(1000)
2: validationAcc = ConvNet(initialPopulation)
3: population = Best100(initialPopulation, validationAcc)
4: repeat
5:   winner = Tournament(population, validationAcc, 20)
6:   maxRedo = 7
7:   while maxRedo > 0 do
8:     maxMutation = 2500
9:     while maxMutation > 0 do
10:      child = Mutation(winner)
11:      if child.CheckIntegrity() then
12:        break inner while loop
13:      else
14:        maxMutation -= 1
15:      end if
16:    end while
17:    valAcc = TrainEffNetV2(child)
18:    if valAcc == None then
19:      maxRedo -= 1
20:    else
21:      break while loop
22:    end if
23:  end while
24:  if maxRedo == 0 then
25:    Pass
26:  else
27:    population = ReplaceOldest(population, child)
28:    validationAcc = Update(validationAcc, valAcc)
29:  end if
30: until Time Expired
```

RandAug regularization strategy were derived from the repository (Li, 2022). The code base for WideResNet and SEResNeXt were derived from the repository (Li, 2017). For computational resources, I had access to a single NVIDIA A100 GPU during evolution.

5.5 NeuroLoss Evolution

In this section, the methodology, results, and discussion are covered for the NeuroLoss evolution.

5.5.1 Methodology

In this section, the methodology behind the proposed surrogate function, initial population size de-couplization, loss function elimination protocol, and software implementation details are explained.

Surrogate Function

It is simple to find specific components of neural networks that perform better than standard versions on a particular dataset or on smaller networks. However, it can be very difficult to find components of neural networks that perform better than standard choices across multiple datasets, on models containing millions of parameters, or when massive amounts of regularization are present during training. To address this issue, I chose to compare the final evolved loss functions on EfficientNetV2Small (EffNetV2Small) (Tan and Le, 2021). EffNetV2Small is a state-of-the-art CNN model containing 20.3 million parameters, that was able to achieve a Top-1% validation accuracy of 83.9% on ImageNet ILSVRC2012 (Deng et al., 2009). Along with the EffNetV2Small model, I use the same training strategy used by Tan and Le (2021): progressive RandAug regularization (Cubuk et al., 2020). *RandAug* is an image augmentation strategy that randomly augments an image by applying n augmentations from a set of possible augmentations, with each augmentation strength capped at a given maximum strength. In *progressive RandAug regularization*, the training strategy is split into four phases, where each

hyper-parameter of augmentation is increased throughout training along with image resolution size. I believe that training loss functions using this state-of-the-art model and state-of-the-art regularization technique will accurately compare the final proposed loss functions against cross-entropy for large-scale CNNs.

Because it is prohibitively expensive to conduct NLFS directly using the full EffNetV2Small training schedule, I use a reduced *surrogate function* during the evolutionary process, as is standard for learning transferable components (Bello et al., 2017b; Bingham et al., 2020; Gonzalez and Miikkulainen, 2020; Liu et al., 2020). An issue that can plague learning transferable structures for neural networks is the correlation between the full model and the surrogate model. If there is no correlation between the smaller surrogate model and the full version, then there is no logical reason to use the surrogate. Although it is simple in concept, it can be very difficult to find an accurate surrogate function for learning transferable structures for a particular component. This problem will be further exemplified shortly with my initial findings. Because I care about the relational ranking of loss functions between the surrogate and the full model, Kendall’s Tau rank correlation (Kendall, 1938) was used instead of Pearson’s Correlation.

As a cheap proxy dataset, all models were trained on the CIFAR-10 dataset (Krizhevsky and Hinton, 2009). For promoting generalization, 5,000 of the training images from CIFAR-10 were used as validation. The best validation accuracy was used as the fitness value during evolution, and for calculating rank correlation.

Chosen Surrogate Function

In the search for a suitable surrogate function, I looked at three possible models: (1) a three-layer ConvNet, containing 14,000 parameters; (2) a ResNet9 version two (ResNet9V2) (He et al., 2016) with nine layers, containing 30,000 parameters;

and (3) EffNetV2Small.

To generate potential loss functions, I performed two preliminary evolutionary runs using the ConvNet and ResNet9 architectures as surrogate functions, both trained on the native dataset image resolution size of 32 for CIFAR-10 without RandAug regularization for 8,000 steps. The best 30 loss functions from each run were then trained using the full EffNetV2Small progressive RandAug training schedule for 64,000 steps. Their final Kendall’s Tau rank correlation is shown in the first row of Table 5.1. Both performed poorly in transferring their final best loss functions to a large-scale model, both achieving a rank correlation of 0.1849, highlighting the issue of finding transferable surrogate functions.

IMAGE SIZE	CONVNET	RESNET	EFFNET
-	0.1849	0.1849	-
32	0.2352	0.1199	0.0344
48	0.1150	0.2649	0.1448
64	0.2768	0.1933	0.3149
80	0.1887	0.3498	0.2643
96	0.2926	0.1218	0.5080
112	0.2873	0.2715	0.3420

Table 5.1: Kendall’s Tau Rank Correlation between ConvNet and ResNet9V2 (ResNet), and EffNetV2Small (EffNet). The first row gives the rank correlation for both ConvNet and ResNet9V2 trained for 8,000 steps without RandAug regularization. The remaining rows give the rank correlation for the architectures trained using RandAug regularization at the given image size for 8,000 steps.

To find a surrogate function with better rank correlation, I took the final 30 loss functions from the ResNet9 evolutionary run (using their associated full scale EffNetV2Small progressive RandAug training schedule validation accuracy), and trained each one on the ConvNet, ResNet9, and EffNetV2Small architectures again, except with progressive RandAug regularization at different image resolution sizes

for 8,000 steps. Table 5.1 shows the correlation between each image resolution size and model trained using progressive RandAug regularization and the full scale EffNetV2Small progressive RandAug training schedule. As one can see, the correlations seem almost random, as no apparent relationship can be derived from either the image resolution or architecture. I believe this exemplifies the difficulty of finding a successfully transferable surrogate function. However, a configuration with an acceptable correlation appeared: EffNetV2Small trained using 96×96 image resolution, with a rank correlation of 0.5080. Although EffNetV2Small is more expensive to train than ResNetV2 and ConvNet, I decided to ensure a transferable surrogate function, and selected it as the surrogate function.

Initial Evolutionary Run

In an initial evolutionary run using the ConvNet architecture as the surrogate function, the best performing loss function was $\frac{1}{n} \sum_i (\min(\text{bessel}_{i0e}(\hat{y}_i), y_i) + \text{bessel}_{i0e}(\hat{y}_i))$, which I call the *Bessel Loss*. As a comparison, Table 5.2 shows the mean validation accuracy on the ConvNet surrogate function after 30 independent runs for *Bessel Loss*, cross-entropy, and the final three *NeuroLoss* functions. As one can see, *Bessel Loss* outperformed all *NeuroLoss* functions, and cross-entropy with a higher mean validation accuracy. In addition, a two-sided statistical Welch’s *t*-test of the mean difference between *Bessel Loss* and the other loss functions being larger than zero (indicating *Bessel Loss* has a larger mean) was performed. The *Bessel Loss* was shown to have a statistically larger mean *NeuroLoss2* and cross-entropy. However, when training *Bessel Loss* on EffNetV2Small using the full training schedule of progressive RandAug regularization on CIFAR-10, the loss struggles to reach a mean test accuracy of 94.51%, compared to *NeuroLoss1*’s 95.95%, *NeuroLoss2*’s 95.91%, and *NeuroLoss3*’s 95.82% test accuracy (From

Table 5.7). This observation reveals that possibly using smaller architectures, with little to no regularization, as a surrogate function for NLFS may not be beneficial as their performance may not correlate well to large-scale CNNs with regularization.

CONVNET SURROGATE FUNCTION		
LOSS	VALIDATION ACCURACY	<i>p</i> -VALUE
BESSELLOSS	58.839 ±0.572	-
NL ₁	58.711±0.455	<i>p</i> = 0.171
NL ₂	58.579±0.612	<i>p</i> = 0.047
NL ₃	58.660±0.557	<i>p</i> = 0.112
CE	58.474±0.569	<i>p</i> = 0.008

Table 5.2: Results for Bessel Loss, NeuroLoss1, NeuroLoss4, and cross-entropy using the ConvNet surrogate function. (NL₁ for NeuroLoss 1, etc.) The mean test accuracy, and standard deviation, across 30 independent runs are reported for each loss function along with the *p*-value for a one sided statistical *t*-test for mean accuracy being larger than cross-entropy. The best mean test accuracy is highlighted in bold.

Early Stopping

To avoid wasting time in training degenerate loss functions, an early stopping method was implemented. Before training on EffNetV2Small 96×96 resolution, a ConvNet was trained on the images, without regularization, for 800 steps. If the training accuracy was below a hand-tuned threshold of 37%, the loss function was rejected and mutated again. If the training accuracy was above this threshold, the loss was trained on the surrogate function, where a secondary early stopping method was implemented. This secondary early stopping method was different from the first. If the training accuracy had not reached a hand-tuned threshold of 25% after 3,000 steps, training was stopped; however, the best validation accuracy was returned as the fitness score to the genetic algorithm.

Initial Population

In an initial shortened evolution I noted that the majority of loss functions for the initial population of the algorithm were degenerate in that the first early stopping mechanism implemented within the surrogate function stopped training due to poor learning. Allowing for degenerate models within the initial population caused three primary issues. First, it created weak competition within the population, as working loss functions would have less competition for mutation. Second, it allowed for early pre-convergence, as only the smaller set of working functions were chosen for mutation. Third, for tournaments containing only degenerate losses, the child loss of the degenerate parent would inherently have higher probability of being degenerate as well. I believe this allowance will degrade evolution quality in the long run. As a possible solution, I decided to decouple the initial population size from the population size. In doing so, I increased the initial population size to allow for more diversity and to allow for more working loss functions. The best from the initial population were then taken to form the first generation, ensuring working initial loss functions.

Loss Function elimination Protocol

Due to the disparity of correlation between the surrogate function and full EffNetV2Small training, I decided to spend the majority of the computational resources on training for transferability rather than searching for loss functions, as the integrity check and early stopping mechanisms eliminated most degenerate solutions. To do so, I implemented a loss function elimination protocol that progressively eliminated candidate loss functions, after evolution, until a final set of winners remained by progressively training them on the different phases of

RandAug regularization for EffNetV2Small. The best 150 loss functions discovered over the entire course of evolution were trained on EffNetV2Small for phase 1 (16,000 steps) of RandAug progressive regularization. The best 50 from that set were then trained again, from scratch, for both phase 1 and 2 (32,000 steps) of RandAug regularization. The best 25 from that set were then trained again until phase 3 (48,000 steps) of RandAug regularization. Finally, the best 10 were taken and trained for the full RandAug progressive regularization training strategy (64,000 steps). Note, all loss functions in the proposed loss function elimination protocol were compared using the validation accuracy for CIFAR-10.

Implementation

For the GA, an initial population size of 1000 was used to allow for ample initial diversity and working loss functions. The best 100 were then taken to form the first generation. A tournament size of 20 was used for selection. The GA was allowed to run for a time frame of 100 hours on an NVIDIA A100 GPU. Each individual within the population was trained using the proposed surrogate function, EffNetV2Small 96 $\times$ 96 resolution for 8,000 steps of Phase 1 of RandAug progressive regularization. Due to computational constraints, the initial population was not trained using this surrogate function, but was instead trained using the ConvNet architecture without RandAug regularization. I did not find this problematic as the purpose of the enlarged initial population was to find workable loss functions in a short amount of time.

5.5.2 Results and Discussion

Evolution

After 100 hours of evolution on the CIFAR-10 dataset, a total of 2,000 iterations had completed, 29% of which did not fail the surrogate function. The best, mean, and median validation scores over time are shown in Figure 5.3. In addition, the max, mean, and min cross-entropy scores across 15 independent training sessions for the surrogate function are plotted for comparison. The algorithm was able to evolve 23 loss functions that were able to surpass the best cross-entropy accuracy on the surrogate function, while 119 loss functions were able to surpass the mean cross-entropy accuracy.

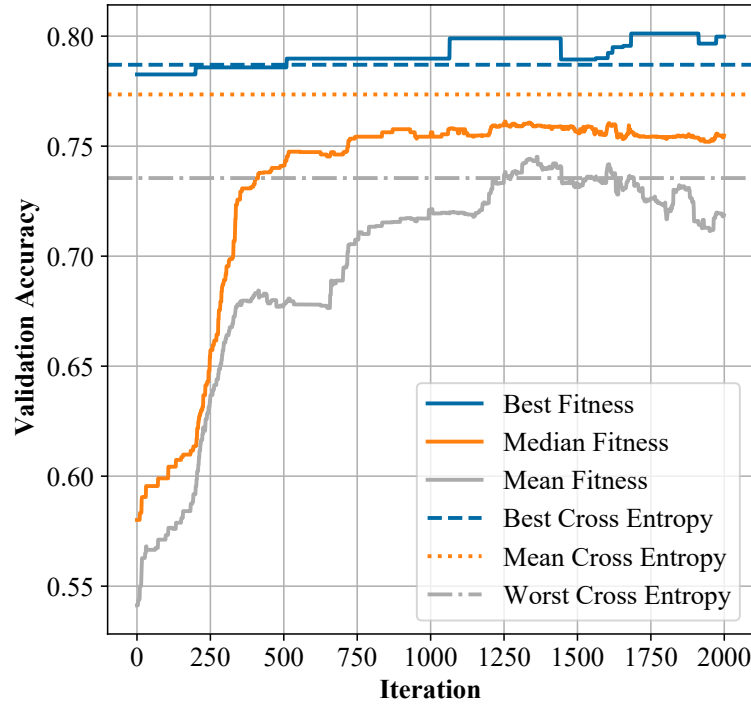


Figure 5.3: Results for the evolutionary algorithm evaluated using the proposed surrogate function.

LOSS EQUATION	
NL ₁	$-\frac{1}{n} \sum_i \left(\ln \left(\left \frac{\hat{y}_i}{\sqrt{1+(\hat{y}_i/(y_i+\epsilon))^2}} \right + \epsilon \right) + \hat{y}_i \frac{\hat{y}_i}{\sqrt{1+(\hat{y}_i/(y_i+\epsilon))^2}} \right)$
NL ₂	$-\frac{1}{n} \sum_i \left(\log_{10} \left(\left \frac{\hat{y}_i}{\sqrt{1+(\hat{y}_i/(y_i+\epsilon))^2}} \right + \epsilon \right) + \sin \left(\frac{\hat{y}_i}{\sqrt{1+(\hat{y}_i/(y_i+\epsilon))^2}} \right) \right)$
NL ₃	$-\frac{1}{n} \sum_i \left(\ln \left(\left \frac{\hat{y}_i}{\sqrt{1+(\hat{y}_i/(y_i+\epsilon))^2}} \right + \epsilon \right) + \min(y_i, \hat{y}_i/(y_i + \epsilon)) \right)$
CE	$-\frac{1}{n} \sum_i y_i \ln(\hat{y}_i + \epsilon)$

Table 5.3: The final three evolved loss functions (NL₁ for NeuroLoss 1, etc.), along with cross-entropy (CE) for comparison.

Best Losses

The final three remaining loss functions, which I call the *NeuroLoss* functions, after running the loss function elimination protocol are listed in Table 5.3. As one can see, the final three functions are very similar as all of them follow the form $\log(y, \hat{y}) + f(y, \hat{y})$, where $\log(y, \hat{y})$ was either $\ln$ or $\log_{10}$ and $f(y, \hat{y})$ was some function. In addition, one can also see the repeated structure of $\hat{y}/(y + \epsilon)$ that is present twice in almost all functions, revealing the power of re-usable hidden state nodes in the proposed search space. It seems that the final loss functions heavily relied on scaling the prediction $\hat{y}$ by some value, mainly $\sqrt{1 + (\hat{y}/(y + \epsilon))^2}$ inside the logarithmic operation. Honorable mentions after running the proposed loss function elimination protocol are found in Table 5.4.

Evaluation

To evaluate the generalizability of the final three loss functions, they were transferred across multiple data sets, architectures, image augmentation techniques, and learning rate schedules and optimizers. Six primary datasets were used in this work, CIFAR-10, CIFAR-100, Stanford-CARS196, Oxford-Flowers102, Caltech101,

LOSS EQUATION
$ \begin{aligned} & -\frac{1}{n} \sum_i \left(\ln \left(\left \frac{\hat{y}_i}{\sqrt{1+(y_i)^2}} \right + \epsilon \right) \right) / \left(\sqrt{1 + (\max(y_i, (\hat{y}_i/(y_i + \epsilon)))^2)} \right) \\ & -\frac{1}{n} \sum_i \max \left(\log \left(\left \frac{\hat{y}_i}{\sqrt{1+(\hat{y}_i/(y_i + \epsilon))^2}} \right + \epsilon \right), \operatorname{arcsinh} \left(\frac{\hat{y}_i}{\sqrt{1+(\hat{y}_i/(y_i + \epsilon))^2}} \right) \right) \\ & -\frac{1}{n} \sum_i \left(\log \left(\left \frac{\hat{y}_i}{\sqrt{1+(\hat{y}_i/(y_i + \epsilon))^2}} \right + \epsilon \right) + \left(\frac{\hat{y}_i}{\sqrt{1+(\hat{y}_i/(y_i + \epsilon))^2}} \right)^2 \right) \\ & -\frac{1}{n} \sum_i \left(\ln \left(\left \frac{\hat{y}_i}{\sqrt{1+(\hat{y}_i/(y_i + \epsilon))^2}} \right + \epsilon \right) + \sinh \left(\frac{\hat{y}_i}{\sqrt{1+(\hat{y}_i/(y_i + \epsilon))^2}} \right) \right) \\ & -\frac{1}{n} \sum_i \left(\ln \left(\left \frac{\hat{y}_i}{\sqrt{1+(\hat{y}_i/(y_i + \epsilon))^2}} \right + \epsilon \right) + \operatorname{erf} \left(\hat{y}_i/(y_i + \epsilon) \right) \right) \\ & -\frac{1}{n} \sum_i \left(\log \left(\left \frac{\hat{y}_i}{\sqrt{1+(\hat{y}_i/(y_i + \epsilon))^2}} \right + \epsilon \right) + \tanh \left(\hat{y}_i/(y_i + \epsilon) \right) \right) \\ & -\frac{1}{n} \sum_i \frac{\ln \left(y_i * \hat{y}_i + \epsilon \right)}{\sqrt{1+(\hat{y}_i/(y_i + \epsilon))^2}} \end{aligned} $

Table 5.4: Honorable Mentions

and ImageNet. The meta-information for each of these datasets are available in Table 5.5.

DATASET	TRAIN	TEST	CLASSES	SIZE
CIFAR10	50,000	10,000	10	32×32
CIFAR100	50,000	10,000	100	32×32
CARS196	8,144	8,041	196	482×700
FLOWERS102	2,040	6,149	102	546×624
CALTECH101	3,060	6,084	102	386×468
IMAGENET	1.28M	50,000	1000	469×387

Table 5.5: The number of training and evaluation images, number of classes, and mean image resolution for the datasets used in this work.

Other than EffNetV2Small, the final loss functions were trained on three other state-of-the art model architectures to assess architecture transferability: WideResNet (depth=28, width=10, 36 million parameters) (Zagoruyko and Komodakis, 2016), SEResNeXt (4×64d, 20 million parameters) (Hu et al., 2018), and DavidNet (custom ResNet9, 6 million parameters, Appendix 5.5.3). Three

different image augmentation techniques were used: progressive RandAug regularization for EffNetV2Small; simple cutout with random cropping and horizontal flipping for WideResNet, SEResNeXt, and DavidNet; and CutMix (Yun et al., 2019) for DavidNet. CutMix was used as it mixes label distribution, testing for whether or not the final loss functions could handle target values not equal to 0 or 1. To assess dataset transferability, the final loss functions were trained using the previous architectures and image augmentation techniques on both CIFAR-10 and CIFAR-100. Lastly, the final loss functions were transferred to three large image resolution benchmark dataset: Cars196, Flowers102, and Caltech101. Using EffNetV2Small, the final loss functions were trained both from scratch and fine-tuning (using the official published ImageNet weights), on the large image resolution datasets. Details on the training strategies used for each architecture can be found in Section 5.5.3.

Results

Results for DavidNet on CIFAR are given in Table 5.6. Results for EffNetV2Small, WideResNet, and SEResNeXt on CIFAR are given in Table 5.7. Results for EffNetV2Small on Cars196, Flowers102, and Caltech101 from scratch are given in Table 5.8, while the results for fine-tuning are given in Table 5.9. As seen in Table 5.6, multiple loss functions achieved significant p -values for having a larger mean test accuracy than cross-entropy across both datasets and image augmentation techniques. However, only one loss, NeuroLoss 1, obtained a significant value for all datasets and augmentation techniques, revealing that it can outperform and increase generalization better than cross-entropy in a computationally stringent environment for DavidNet.

From Tables 5.6 and 5.7, only once was the mean test accuracy of cross-entropy

CIFAR-10 (CUTOUT)			CIFAR-10 (CUTMIX)		
LOSS	TEST ACCURACY	p -VALUE	TEST ACCURACY	p -VALUE	
NL ₁	94.042 ±0.126	$p < 0.001$	91.621 ±0.184	$p < 0.001$	
NL ₂	93.787±0.138	$p = 0.298$	91.409±0.178	$p = 0.581$	
NL ₃	94.006±0.134	$p < 0.001$	91.601±0.169	$p < 0.001$	
CE	93.761±0.134	-	91.419±0.197	-	

CIFAR-100 (CUTOUT)			CIFAR-100 (CUTMIX)		
LOSS	TEST ACCURACY	p -VALUE	TEST ACCURACY	p -VALUE	
NL ₁	74.959 ±0.212	$p = 0.003$	70.877±0.270	$p < 0.001$	
NL ₂	74.814±0.254	$p = 0.421$	70.736±0.244	$p < 0.001$	
NL ₃	74.834±0.263	$p = 0.302$	71.176 ±0.288	$p < 0.001$	
CE	74.802±0.208	-	69.534±0.228	-	

Table 5.6: Results for DavidNet using resize, cutout, and horizontal flipping, and CutMix for both CIFAR-10 and CIFAR-100. The mean test accuracy and standard deviation across 30 independent runs is reported for each loss function along with the p -value for a Welch’s two-sided statistical t -test for mean difference between the *NeuroLoss* function and cross-entropy being larger than zero (indicating the function has a larger mean). The best mean test accuracy for each dataset is highlighted in bold.

better than the best loss function (SEResNeXt for CIFAR-100). Two loss functions dominated the results, NeuroLoss 1 and NeuroLoss 3. In terms of mean test value, NeuroLoss 1 and NeuroLoss 3 surpassed cross-entropy across all datasets and architectures (except NeuroLoss 3 for EffNetV2Small for CIFAR-10, and both losses for SEResNeXt for CIFAR-100). However, NeuroLoss 1 was able to surpass NeuroLoss 3 for all architectures trained upon CIFAR-10, but NeuroLoss 3 was able to surpass Loss 1 and cross-entropy for all architectures trained on CIFAR-100 (except DavidNet using cutout and SEResNeXt).

From Tables Table 5.8, both NeuroLoss 1 or NeuroLoss 3 outperformed cross-entropy, even with a label smoothing constant of $\alpha = 0.10$, across all datasets. However, when changing from training from scratch to fine-tuning using cross-

CIFAR-10			
LOSS	EFFNETV2S	WIDERESNET	SERESNEXT
BESSEL	94.510 $\pm$ 0.115		
NL ₁	95.952 $\pm$ 0.106	95.860 $\pm$ 0.142	95.830 $\pm$ 0.123
NL ₂	95.906 $\pm$ 0.050	95.836 $\pm$ 0.052	95.604 $\pm$ 0.032
NL ₃	95.816 $\pm$ 0.060	95.764 $\pm$ 0.050	95.788 $\pm$ 0.078
CE	95.826 $\pm$ 0.118	95.670 $\pm$ 0.156	95.222 $\pm$ 0.063

CIFAR-100			
LOSS	EFFNETV2S	WIDERESNET	SERESNEXT
NL ₁	78.172 $\pm$ 0.249	79.848 $\pm$ 0.061	79.964 $\pm$ 0.167
NL ₂	77.926 $\pm$ 0.189	79.628 $\pm$ 0.242	80.152 $\pm$ 0.263
NL ₃	78.230 $\pm$ 0.154	80.038 $\pm$ 0.271	79.804 $\pm$ 0.263
CE	77.130 $\pm$ 0.228	79.640 $\pm$ 0.213	80.526 $\pm$ 0.228

Table 5.7: Results for EffNetV2Small using progressive RandAug regularization, WideResNet (depth=10, width=28) and SEResNeXt (4x64d) using resize, cutout, and horizontal flipping augmentation for both CIFAR-10 and CIFAR-100. The mean test accuracy and standard deviation across 5 independent runs is reported for each loss function. The best mean test accuracy for each combination of dataset and architecture is highlighted in bold.

EFFNETV2SMALL (FROM SCRATCH)			
LOSS	CARS196	FLOWERS102	CALTECH101
NL ₁	85.457 $\pm$ 0.244	88.584 $\pm$ 0.508	85.148 $\pm$ 0.210
NL ₂	85.698 $\pm$ 0.361	87.380 $\pm$ 0.461	84.234 $\pm$ 0.341
NL ₃	86.439 $\pm$ 0.418	88.408 $\pm$ 0.440	84.822 $\pm$ 0.285
CE	82.729 $\pm$ 0.285	88.154 $\pm$ 0.299	84.116 $\pm$ 0.119
CE _{0.10}	84.674 $\pm$ 0.254	87.819 $\pm$ 0.228	84.499 $\pm$ 0.332

Table 5.8: Results for EffNetV2Small using progressive RandAug regularization when training from scratch for Stanford-CARS196, Oxford-Flowers102, and Caltech101. The mean test accuracy and standard deviation across 5 independent runs is reported for each loss function. The best mean test accuracy for each dataset is highlighted in bold. CE_{0.10} refers to cross-entropy with a label smoothing value of 0.10.

entropy biased initial weights, Table 5.9 reports that NeuroLoss 2 outperforms all other NeuroLoss functions and cross-entropy, both with and without label

EFFNETV2SMALL (FINE-TUNE)			
Loss	CARS196	FLOWERS102	CALTECH101
NL ₁	91.635±0.109	97.313±0.095	90.556±0.279
NL ₂	91.660±0.053	97.421±0.139	90.565±0.428
NL ₃	91.635±0.091	97.287±0.138	90.302±0.295
CE	91.725±0.233	97.190±0.188	90.217±0.250
CE _{0.10}	91.496±0.122	97.320±0.277	90.434±0.141

Table 5.9: Results for EffNetV2Small using progressive RandAug regularization when fine-tuning for Stanford-CARS196, Oxford-Flowers102, and Caltech101. The mean test accuracy and standard deviation across 5 independent runs is reported for each loss function. The best mean test accuracy for each dataset is highlighted in bold. CE_{0.10} refers to cross-entropy with a label smoothing value of 0.10.

smoothing, on two of the three datasets. With NeuroLoss 2 trailing the other two NeuroLoss functions when training from scratch on Flowers102 and Caltech101, but outperforming both when fine-tuning could indicate that some loss functions might be better inclined to fine-tuning than others.

What Makes *NeuroLoss* Successful?

Figures 5.1 and 5.2 depict the phenotypes of all NeuroLoss functions and cross-entropy in a binary example. From both of these figures, there are two noticeable differences. First, all NeuroLoss functions have a less steep slope compared to cross-entropy, resulting in higher normalized loss values. From Figure 5.1, NeuroLoss 2 has the highest normalized loss values compared to all other functions when $\hat{y} < 0.4$; however, Figure 5.2 indicates that NeuroLoss 2 dips below the other NeuroLoss functions and even cross-entropy as $\hat{y} \rightarrow 1$. NeuroLoss 1 and NeuroLoss 2 seem to hug each other for both small and large $\hat{y}$ values. Perhaps this more gentle slope could encourage more smoothness in the loss landscape than cross-entropy. However, further investigation and study must be conducted

to verify that hypothesis. Second, the global minimum for all NeuroLoss functions do not occur at $\hat{y} = 1$, as with cross-entropy, but instead at $\hat{y} = 0.99998$. This small level of implicit regularization can be replicated in cross-entropy with an extremely small label smoothing constant of $\alpha = 0.00003$. Although the global minimum can be shifted in cross-entropy via label smoothing to match that of the NeuroLoss functions, the slope cannot be changed to equal them. The exact reason for the *NeuroLoss* functions success is unknown, but two possible reasons arise. The implicit learning of an extremely small label smoothing constant and a higher normalized loss values than cross-entropy over the majority of $\hat{y}$ values.

5.5.3 Training Strategies

Table 5.10 shows the adapted progressive RandAug regularization training strategy for CIFAR-10 and CIFAR-100 for EffNetV2Small. The surrogate function was trained for only 8,000 steps of phase 1. The proposed loss function elimination protocol trained on each phase incrementally, where each phase contained 16,000 steps. EffNetV2Small was always trained using the Adam optimizer (Kingma and Ba, 2014) with a maximum learning rate of 0.001 and a batch size of 64. The first 6,400 steps were used to warm up the learning rate linearly from 0 to the maximum, and then the learning rate was decayed using cosine annealing back down to 0 during the rest of the training cycle. This training strategy was only used for the surrogate function and loss function elimination protocol.

When comparing the final loss functions after evolution and the elimination protocol, the training strategy was adjusted for EffNetV2Small to allow for a longer training session. Each phase was increased to 24,000 steps, allowing for 96,000 total steps over the entire course of training. The same optimizer and

PHASE	1	2	3	4
IMAGE SIZE	128	160	192	224
DROPOUT	0.10	0.20	0.30	0.40
RANDAUG	5	8	12	15

Table 5.10: Adapted progressive RandAug regularization training strategy for CIFAR-10 and CIFAR-100.

learning rate strategy were used as before, except the learning rate was held at its maximum of 0.001 for 12,400 steps before being decayed.

For WideResNet and SEResNeXt, SGD with Nesterov momentum of 0.9 was used with a learning rate scheduler that warmed up to 0.1 after 1,000 steps, and then was cosine decayed back down to 0. This same training strategy was used by Zagoruyko and Komodakis (2016) for the WideResNet architecture, except they decayed their learning rate with a piece-wise function rather than cosine. With a batch size of 128, a total of 96,186 steps were performed. For image augmentation, the images were resized to 40×40 , randomly cropped back to 32×32 , cutout was applied with an 8×8 mask, and then the images were randomly flipped horizontally.

For DavidNet, two different image augmentation techniques were applied. For both, SGD with Nesterov momentum of 0.9 was used with a learning rate scheduler that warmed up to 0.4 after 1,176 steps, and then was linearly decayed. For the first configuration using the random crop, cutout, and horizontal flipping, a weight decay value of 0.512 was also applied to the gradients. This training strategy was used by the original DavidNet proposal for the DAWNBench competition (Coleman et al., 2019), an end-to-end deep learning benchmark for training time, cost, and accuracy for CIFAR-10 and ImageNet. With a batch size of 512, 98 steps were performed per epoch, for 30 epochs, totalling to a small budget of

2,940 steps. As a result, the loss functions are evaluated for whether or not they can increase generalization within a very computationally stringent budget. For the second image augmentation technique, CutMix, the exact same parameters and setup were used for DavidNet, except the weight decay was reduced to 0.0256, as CutMix is a much harsher augmentation technique than simple cutout.

When training EffNetV2Small on Stanford-CARS196, Oxford-Flowers102, and Caltech101 from scratch, a few minor changes were in-placed. The same optimizer, learning rate, schedule, and total steps were used as before; however, the progressive RandAug regularization strategy was reverted back to its original form (Tan and Le, 2021) for the much larger image resolution datasets. These changes are reflected in Table 5.11. Due to the increased image resolution, the batch sizes for EffNetV2Small were decreased to 32.

When fine-tuning, the initial weights were set to the official ImageNet2012 weights, and the loss functions were trained for only 20 epochs, with 500 steps per epoch, totalling to only 10,000 steps. The images were set to 224x224, dropout to 0.5, RandAug magnitude to 15, batch size to 64, ptimizer to Adam, and learning rate schedule that warmed up for one epoch to 0.001 before being decayed back down to zero after being held for three epochs.

PHASE	1	2	3	4
IMAGE SIZE	128	186	244	300
DROPOUT	0.10	0.17	0.24	0.30
RANDAUG	5	8	12	15

Table 5.11: Original RandAug progressive regularization training strategy for EffNetV2Small for large resolution image datasets.

5.5.4 Conclusion

In this section, I expanded on previous research in NLFS by proposing a new search space containing more unary and binary operations, along with a new computational graph representation that eliminated some of the downsides associated with GP. In addition, I stressed the importance of ensuring transferable surrogate functions to ensure good rank correlation. After running the evolutionary algorithm and loss function elimination protocol, three loss functions highly competitive with cross-entropy were discovered through pure evolution. The best loss functions, termed *NeuroLoss 1*, *NeuroLoss 2*, and *NeuroLoss 3*, were able to surpass cross-entropy across multiple architectures, datasets, and image augmentation techniques. The proposed search space, integrity check, and elimination protocol can be used for future NFLS endeavors, such as loss functions for image anomaly detection, imbalanced image datasets, or traditional based machine learning algorithms, such as logistic regression and decision trees. My code can be found here: <https://github.com/OUStudent/NeuralLossFunctionEvolution>.

5.6 Image-Augmentation Specific Evolution

In this section, the methodology, results, and discussion are covered for the image-augmentation specific evolution.

5.7 Specific Related Work

Machine and deep learning models are heavily prone to *over-fitting*, a phenomenon that occurs when the error of the training dataset is small, but the error on a validation or test dataset is large. For image processing, image data augmentations

are commonly applied to add *regularization* by artificially increasing the dataset size. Simple data augmentations include vertically or horizontally shifting, flipping, or translating an image. From the advent of preventing co-adaptation of features within neural networks through *dropout* (Hinton et al., 2012), *cutout* (DeVries and Taylor, 2017) was proposed to drop random cutouts of pixels from the input image before being fed into the model. By dropping out random patches of pixels from the input image, the model must learn to classify an object using the remaining data. Although it was originally applied for regression problems, *mixup* (Zhang et al., 2017) has been successfully applied to classification. Mixup creates artificial images and labels by taking a random linear interpolation between two images. As a result, the model’s output distribution becomes less bimodal, and more uniform, which un-intuitively increases generalization (Müller et al., 2019). Besides the simple data augmentations mentioned earlier, there also exist many others, such as shearing, inversion, equalize, solarize, posterize, contrast, sharpness, brightness, and color. Many works have been published in order to discover the best set of *policies*, combinations of image augmentations, for specific datasets (Cubuk et al., 2018; Ho et al., 2019). However, *random augment* (RandAug) (Cubuk et al., 2020) has been regarded as the only policy capable of generalizing across datasets with extremely little need of fine-tuning or excess training (Tan and Le, 2021). Given n and m , RandAug performs n random augmentations from a list of possible augmentations, performing each one at a strength of m . Simple augmentations, cutout, mixup, and RandAug are all state-of-the-art augmentation techniques that are currently be used in practice for different situations.

5.7.1 Methodology

Image Augmentation Correlation

Because training each candidate loss function on a full scale model is extremely computationally expensive, *surrogate* functions can be used as proxies. It is typical to use smaller models, smaller datasets, and shortened training schedules when constructed surrogate functions (Zoph et al., 2018; Bingham et al., 2020; Bingham and Miikkulainen, 2022). I revealed the disparity of correlation between loss functions trained on smaller models and those trained on full scale models. Now, I investigate this observation further by taking a look at image augmentation techniques.

In order to assess how well loss functions transferred across image augmentation techniques, I constructed five different image augmentations. The first, referred to as *base*, served as the simplest form of image augmentation. In this augmentation, the images were zero-padded before a random crop equal to the training image size was obtained. The resulting image was then randomly horizontally flipped. The rest of the four image augmentations were built using *base* as the foundation. The second, referred to as *cutout*, applied the cutout operation, with a fixed patch size, to the image. The third, referred to as *mixup*, applied the mixup operation by creating artificial images through random linear combinations of images and their labels. The fourth, referred to as *RandAug*, applied the random augmentation strategy to randomly augment the images from a list of possible augmentations. Lastly, the fifth, referred to as *all*, combined all previous augmentations together in order to maximize regularization. Figure 5.4 shows an example of each augmentation on an image from the Oxford-Flowers dataset. As one can see, *base* only slightly alters the image, while *all* completely

distorts it. Despite the massive distortion of the *all* augmentation, the results reveal that the *all* augmentation technique consistently outperforms the other four.



Figure 5.4: First Image: original untampered image. Second: base augmentation. Third: cutout augmentation. Fourth: mixup augmentation. Fifth: RandAug augmentation. Sixth: all augmentation

As with other neural component search methodologies, all searching experiments were performed on the CIFAR-10 dataset. For encouraging generalization, 5,000 of the 50,000 training images were set aside as validation. The validation accuracy is reported in all preliminary results. Using the search space, 1,000 random loss functions were generated and evaluated on the CIFAR-10 dataset. I did not have the computational resources to evaluate all loss functions on a state-of-the-art large-scale model; however, this does not mean I used a small model either. I chose to use a custom ResNet9v2 containing 6.5 million parameters as it was quick to train, while being relatively large (millions of parameters). The model was trained with the Adam optimizer (Kingma and Ba, 2014), batch size of 128, and one cycle cosine decay learning rate schedule (Smith, 2018a) that warmed up the learning rate from zero to 0.001 after 1,000 steps, before being decayed back down to zero. The model was trained for 40 epochs, 500 steps per epoch, totalling to 20,000 steps.

Figure 5.5 shows the scatter-plot of the validation accuracies for all 1,000

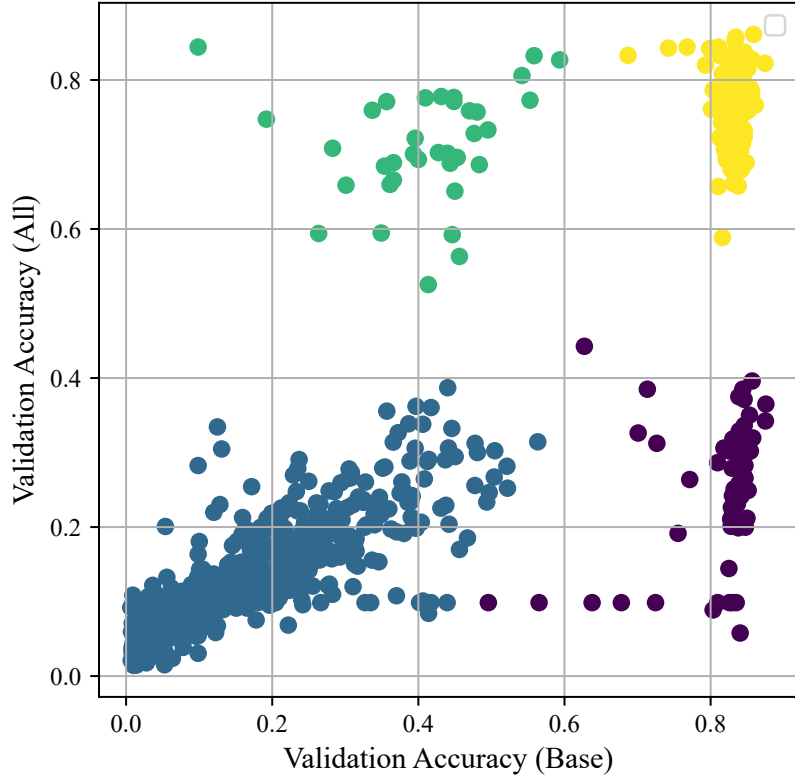


Figure 5.5: Validation accuracies for *base* and *all* techniques for the 1,000 randomly generated loss functions.

loss functions evaluated on *base* and *all* image augmentations. As one can see, there appear to be four clusters of loss functions after running agglomerative clustering with average linkage (colors denote different clusters). The first contain losses that do not perform well on either *base* (< 0.60) nor *all* (< 0.50). The second contain losses that perform well on *base* (> 0.80) but not *all* (< 0.50). The third contain losses that perform decently well on *all* (> 0.60) but not well on *base* (< 0.60). The last contain losses that perform well on *base* (> 0.75) and decently well on *all* (> 0.60). Because I care about preserving the rank in-between loss functions on different image augmentation techniques, I rely upon

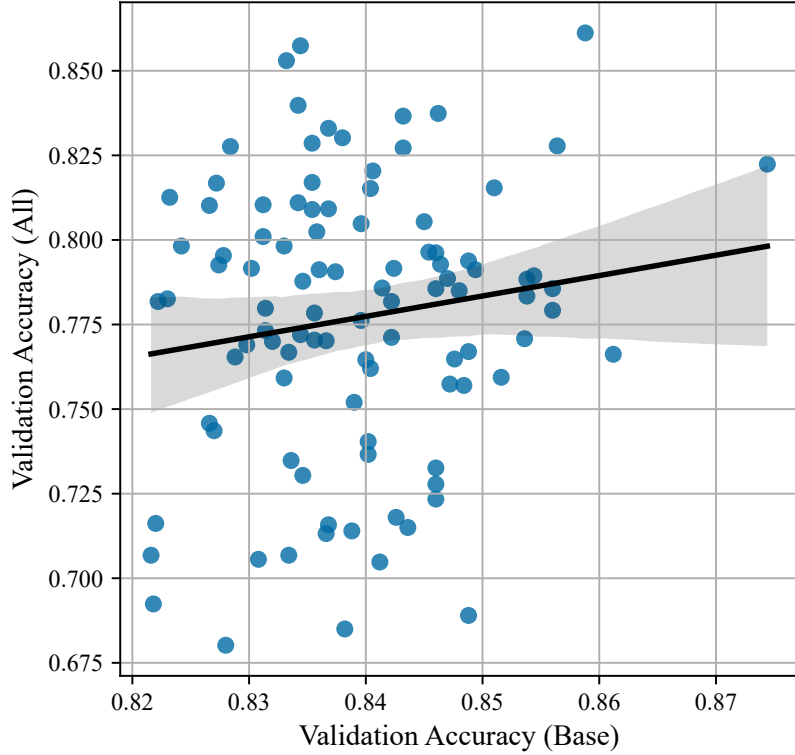


Figure 5.6: Validation accuracies for *base* and *all* techniques for the intersection of the best loss functions.

Kendall's Tau rank correlation (Kendall, 1938). The top half of Table 5.12 gives the Tau rank correlation between all image augmentation techniques across all 1,000 randomly generated loss functions. As one can see, the rank correlations do not appear to poor, with the majority above 0.70, signifying moderate to good rank preservation. However, Figure 5.5 shows that the majority of loss functions generated are degenerate, meaning the loss functions perform poorly on base (< 0.80), all (< 0.60), or both. During evolution, these loss functions would get filtered out due to the early stopping mechanism of the surrogate function. When the intersection of best 50 loss functions from *base* and *all* are joined together, the rank disparity becomes much more clearer. Figure 5.6 zooms in on

the intersection of the best 50 loss functions from base and all. As one can see, the correlation seems almost uniform. The bottom half of Table 5.12 gives the Tau rank correlation between all image augmentation techniques across the intersection of the best 50 loss functions of each respective pair. These rank correlations are much lower, with all but one plummeting below 0.50, indicating weak to moderate preservation of rank. Most noticeably, the rank correlations of *base* between all other image augmentation techniques, except *RandAug*, are extremely poor (< 0.10). This disparity reveals that training loss functions with extremely little to no image augmentation (*base*) have no guarantee to transfer well to other forms of image augmentation, even when using the same model, dataset, and training strategy. Perhaps these results could explain the observation I first noted in NeuroLoss, where their smaller models ($< 0.3\text{M}$ parameters), trained without any image augmentation, transferred poorly to EfficientNetV2Small, trained with *RandAug*. Although, I cannot know this for certain without testing the effects of transferring loss functions across model size, this observation is important for all future NLFS endeavours, as it signifies that training loss functions with very little image augmentation may be predisposing them to lack the ability of effectively transferring to other augmentation techniques.

Evolutionary Runs

From the results discussed in Section 5.7.1, I decided to capitalize on the disparity of rank correlation between image augmentation techniques by performing regularized evolution on each of the five proposed augmentation techniques in the hopes of finding image augmentation specific loss functions. Five evolutionary runs were performed, one for each technique. The same setup as the NeuroLoss evolution was used. The best 100 loss functions from the initial 1,000 randomly

ALL LOSSES	CUTOUT	MIXUP	RANDAUG	ALL
BASE	0.773	0.711	0.743	0.711
CUTOUT	-	0.704	0.743	0.699
MIXUP		-	0.692	0.736
RANDAUG			-	0.677
BEST 50 LOSSES				
BASE	0.079	0.056	0.212	0.042
CUTOUT	-	0.309	0.590	0.357
MIXUP		-	0.442	0.462
RANDAUG			-	0.482

Table 5.12: Kendall Tau’s rank correlation between the five different image augmentation techniques for 1000 randomly generated loss functions evaluated on a ResNet9 (6.5M parameters). Top half of the table is the rank correlation for all 1000 loss functions. Bottom half is the rank correlation for best 50.

generated loss functions formed the first generation for each respective technique. With a tournament size of 20, the genetic algorithm was allowed to run for 2,000 iterations.

Elimination Protocol

In order to properly evaluate the final loss functions, I chose to compare the final functions on WideResNet-28-10 (36M parameters), a large scale state-of-the-art model. To ensure transferability across model sizes, I used a similar loss function elimination protocol to the one in NeuroLoss. The best 100 loss functions discovered over the entire evolutionary search for each run were retrained on the same surrogate function as before, except the training session was extended to 192 epochs instead of 40, totalling to 96,000 steps. The best 24 from that were taken and trained on WideResNet-28-10 for each respective technique. The best 12 from that were taken and trained again. The best 6, compared by averaging their two previous runs on WideResNet, from that were taken and trained again. Finally, the

best three, compared by averaging their previous three runs on WideResNet, were chosen and presented. When training on WideResNet-28-10, the SGD optimizer was used, along with Nesterov’s momentum of 0.9, and a one cycle cosine decay learning rate schedule that warmed up the learning rate from zero to 0.1 after 1,000 steps, before being decayed back down to zero. The model was trained for 300 epochs, 391 steps per epoch, totalling to 117,300 steps.

5.7.2 Results

Final Losses

During the elimination protocol, the best 24 loss functions from the 100 trained on ResNet9v2 for 96,000 steps were trained on WideResNet for 117,300 steps. Using their final best validation accuracies for both models, I was able to construct the rank correlations to assess how well the loss functions would transfer to a larger model. The results for *base*, *cutout*, *mixup*, *RandAug*, and *all* were 0.243, 0.034, 0.112, 0.307, and 0.100. As one can see, these are extremely poor, which confirms my original findings in NeuroLoss of poor correlation between smaller models and large CNNs. Despite this lack of correlation, the majority of the final found loss functions still performed well compared to CE.

The final three loss functions for each augmentation technique, *base* ($B0, B1, B2$), *cutout* ($C0, C1, C2$), *mixup* ($M0, M1, M2$), *RandAug* ($R0, R1, R2$), and *all* ($A0, A1, A2$), are shown in Table 5.13. From the table, a few observations can be made. First, all *base*, *cutout*, and *RandAug* functions used the expression $\hat{y}/(y + \epsilon)$. Second, all three of the *all* functions were of the form $f(\log(\hat{y})) * y$, where it was either $\log(x)$ or $\ln(x)$, and $f(x)$ was dependent upon the function. Lastly, many of the loss functions, across all five of the techniques, used some form

Name	Loss Equation
B0	$-\frac{1}{n} \sum_i \sigma(\text{softplus}(\hat{y}_i/(y_i + \epsilon)))$
B1	$-\frac{1}{n} \sum_i \frac{d}{dx} \sigma(\text{bessel_i0e}(\hat{y}_i/(y_i + \epsilon)))$
B2	$-\frac{1}{n} \sum_i \text{bessel_i1e}(\text{erf}(\hat{y}_i/(y_i + \epsilon)))$
C0	$-\frac{1}{n} \sum_i \ln(\arctan(\hat{y}_i/(y_i + \epsilon)))$
C1	$-\frac{1}{n} \sum_i \ln(\text{softplus}(\hat{y}_i/(y_i + \epsilon)))$
C2	$-\frac{1}{n} \sum_i \sigma(\log(\hat{y}_i/(y_i + \epsilon)))$
M0	$-\frac{1}{n} \sum_i \text{bessel_i1}(\arctan(\log(\hat{y}_i))) * y_i$
M1	$\frac{1}{n} \sum_i \ln(1 + \exp(\hat{y}_i - y_i))$
M2	$\frac{1}{n} \sum_i \text{arcsinh}(\text{bessel_i0}(y_i - \hat{y}_i))$
R0	$\frac{1}{n} \sum_i \frac{d}{dx} \tanh(\text{bessel_i0e}(\hat{y}_i/y_i + \epsilon)) + \text{bessel_i0e}(\hat{y}_i/y_i)$
R1	$\frac{1}{n} \sum_i \log(\hat{y}_i/y_i + \epsilon) / \sqrt{1 + (\log(\hat{y}_i/y_i + \epsilon) + y_i)^2}$
R2	$\frac{1}{n} \sum_i \log(\hat{y}_i/y_i + \epsilon) ^{ \text{bessel_i0}(y_i) }$
A0	$-\frac{1}{n} \sum_i \ln(\frac{d}{dx} \text{softsign}(\log(\hat{y}_i))) * y_i$
A1	$-\frac{1}{n} \sum_i \ln(\text{bessel_i0e}(\log(\hat{y}_i))) * y_i$
A2	$\frac{1}{n} \sum_i y_i / (\text{bessel_i0e}(\ln(\hat{y}_i + \epsilon)))$
CE	$-\frac{1}{n} \sum_i y_i \ln(\hat{y}_i + \epsilon)$

Table 5.13: The final discovered loss functions.

of the bessel function. The binary phenotypes (when $y = 1$) of the normalized loss values for a hand selected number of the final loss functions, based upon the results in Section 5.7.2, can be seen in Figure 5.7. As one can see, many of the final loss functions differ in their phenotype compared to CE, either in terms of learning a shifted global minimum or a much steeper or less steep slope.

Evaluation

To assess the quality of the final loss functions, all were trained on WideResNet-28-10 using the training strategy discussed in Section 5.7.1 on the CIFAR-10 and CIFAR-100 datasets. The results for the CIFAR datasets can be seen in Table 5.14. For comparison, cross-entropy (CE), both with and without label

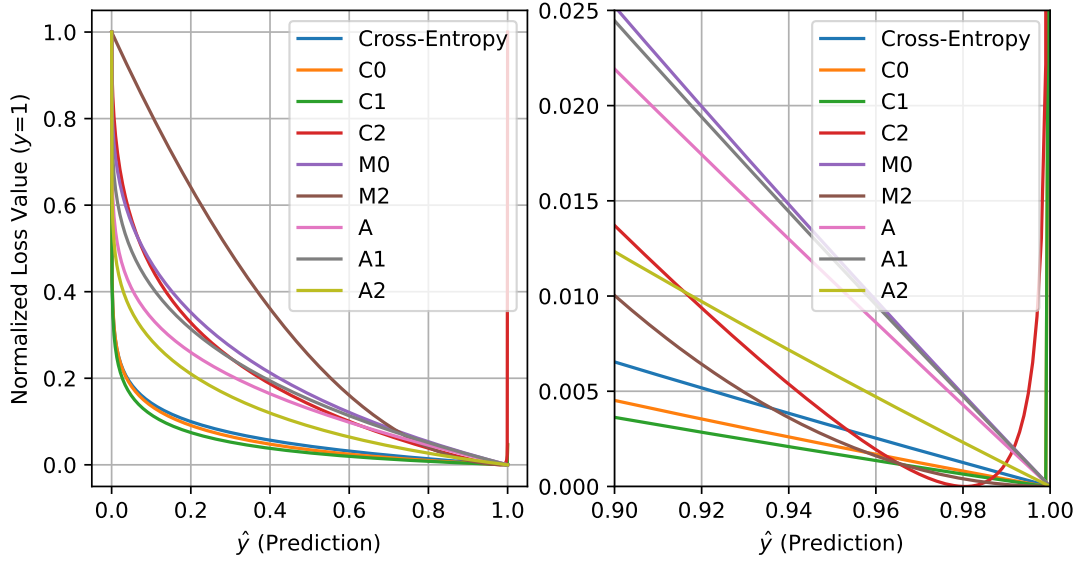


Figure 5.7: Binary phenotype of the final best loss functions against cross-entropy for normalized loss values.

smoothing ($\text{CE}_{0.10}$, $\alpha = 0.10$) were trained. From the table, a few observations arise. First, for CIFAR-10, not all loss functions evolved for each specific image augmentation technique outperformed CE for that technique. For example, all of the final *RandAug* loss functions performed poorly compared to CE on *RandAug*. On the contrary, some loss functions performed well on techniques that they were not evolved upon. For example, functions from the *all* group achieved Top 1 on *base* and *mixup*. Second, when transferring to CIFAR-100, all of the final functions for *base* and *RandAug*, and *M1*, collapse, performing extremely poorly. This observation reinforces the idea that loss functions evolved on a particular dataset, model, and training strategy may not perform well in other environments. Third, only twice did CE or $\text{CE}_{0.10}$ achieve Top 1 across all augmentation techniques on CIFAR-10 and CIFAR-100, indicating great success for the final loss functions.

To further assess the performance of the final loss functions on different datasets

and architectures, the best overall performing functions from CIFAR-10 and CIFAR-100 were selected be fine-tuned on CARS-196 using EfficientNetV2Small and their official ImageNet2012 (Deng et al., 2009) published weights. The best overall functions were chosen to be all *cutout* and *all* functions, and *M0* and *M2*. When fine-tuning on EfficientNetV2Small, the SGD optimizer was used, along with Nesterov’s momentum of 0.9, batch size of 64, and a one cycle cosine decay learning rate schedule that warmed up the learning rate from zero to 0.1 after 1,000 steps, before being decayed back down to zero. The models were trained for 20 epochs, 500 steps per epoch, totalling to 10,000 steps. The results are shown in Table 5.15. Once again, a few loss functions struggle to generalize, *C0*, *C2*, and *M2*. However, the *A2* loss function performs relatively well compared to other discovered loss functions across all techniques. In fact, even combining *A2* with a label smoothing value of $\alpha = 0.10$, $A2_{0.10}$, achieves Top 1 for all techniques on CARS-196. The generalization power of *A2* was further explored by being fine-tuned on Oxford-Flowers-102 and Caltech-101, as shown in Table 5.15. For these other two datasets, the best loss flipped between $A2_{0.10}$ and $CE_{0.10}$.

Inverse Bessel Logarithm Loss

Due to the success of *A2* in terms of generalization across the CIFAR and fine-tuning datasets, it was deemed as the final best loss function discovered in this work, which I called the *inverse bessel logarithm loss*, due to its equation phenotype. The binary phenotypes of loss functions limit how well the loss landscape can be investigated for multiple dimensions. In order to better assess the differences between CE and the *inverse bessel logarithm*, a three-dimensional plot can be constructed where the *x*-axis takes on *y* values, the *y*-axis takes on $\hat{y}$ values, and the *z*-axis takes on the loss $\mathcal{L}(y, \hat{y})$. Figure 5.8 shows the non-normalized three-

LOSS	BASE	CUTOUT	MIXUP	RANDAug	ALL
CIFAR-10					
CE	94.116±0.238	95.400±0.107	96.616 ±0.600	95.012±0.261	96.258 ±0.227
CE _{0.10}	94.080±0.245	95.326±0.240	96.598±0.576	94.864±0.199	96.368 ±0.244
B0	94.018±0.371	95.222±0.106	94.396±0.358	94.510±0.153	91.040±0.182
B1	94.272±0.175	95.324±0.066	95.212±0.397	94.628±0.226	92.400±0.110
B2	94.392±0.226	95.544±0.090	95.552±0.531	94.902±0.116	93.954±0.138
C0	94.446 ±0.202	95.600 ±0.154	95.934±0.441	95.022±0.169	95.076±0.148
C1	94.238±0.344	95.496±0.048	95.936±0.634	95.090 ±0.276	95.012±0.174
C2	94.454 ±0.212	95.710 ±0.141	95.030±0.482	95.164 ±0.143	93.284±0.181
M0	94.432 ±0.286	95.482±0.188	96.174±0.358	94.740±0.240	96.044±0.125
M1	94.446 ±0.141	95.528 ±0.097	96.162±0.435	95.092 ±0.202	95.268±0.080
M2	94.364±0.322	95.510 ±0.065	96.644 ±0.447	95.068 ±0.233	95.448±0.072
R0	94.250±0.267	95.434±0.165	93.136±0.404	94.660±0.227	89.364±0.253
R1	94.106±0.222	95.514 ±0.125	95.546±0.572	94.876±0.210	94.204±0.046
R2	93.704±0.344	95.172±0.184	95.486±0.468	93.858±0.263	92.550±0.123
A0	94.258±0.140	95.488±0.147	96.654 ±0.507	94.758±0.270	96.152 ±0.287
A1	94.566 ±0.178	95.390±0.130	96.086±0.081	94.866±0.249	96.110 ±0.185
A2	94.304±0.231	95.374±0.149	96.246 ±0.440	95.142 ±0.224	96.076±0.239
A2 _{0.10}	93.946±0.276	95.270±0.100	96.644 ±0.519	94.898±0.243	96.418 ±0.19
CIFAR-100					
CE	75.678±0.452	77.540 ±0.472	80.054 ±1.734	77.354 ±0.448	81.166 ±1.203
CE _{0.10}	75.916 ±0.391	76.692±0.703	80.488 ±1.644	76.522 ±0.871	80.748 ±1.666
B0	59.652±1.505	41.734±0.602	45.960±0.833	31.988±0.425	31.988±0.425
B1	21.852±0.893	20.504±1.966	31.602±1.071	11.940±1.665	16.392±0.448
B2	54.822±2.624	22.892±2.199	78.464±1.162	22.892±2.199	49.318±1.225
C0	74.494±1.012	76.098±0.632	79.658 ±2.038	75.654±0.661	79.412±0.640
C1	75.748 ±0.600	76.410±0.656	79.498±2.454	75.934±0.844	79.288±0.787
C2	76.694 ±0.419	78.866 ±0.544	78.914±2.127	76.926 ±0.614	72.340±0.545
M0	75.670±0.495	77.290 ±0.626	79.670 ±1.849	75.850±0.524	79.394±0.907
M1	62.826±0.822	55.574±0.565	49.208±0.984	40.786±0.653	28.504±0.712
M2	76.430 ±0.585	77.278 ±0.533	74.998±1.387	61.574±1.980	40.036±0.251
R0	31.658±0.676	26.438±0.455	25.345±0.881	15.038±0.706	10.280±0.451
R1	17.732±1.834	30.760±1.598	46.834±1.729	15.120±0.117	12.291±0.374
R2	73.854±0.666	60.856±0.859	51.300±1.668	25.148±2.357	15.964±0.383
A0	75.552±0.606	76.798±0.550	78.810±1.971	75.272±0.253	80.320 ±1.475
A1	75.278±0.589	76.630±0.727	78.394±1.946	76.652 ±0.228	81.058 ±0.814
A2	76.034 ±0.771	77.486 ±0.460	80.252 ±2.101	77.020 ±0.729	79.378±0.427
A2 _{0.10}	75.714±0.609	76.954±0.482	79.114±1.853	76.516±0.688	81.728 ±0.809

Table 5.14: Results for all final loss functions evaluated on the CIFAR datasets using each of the five selected image augmentation techniques. The mean and standard deviation of the test accuracy across 5 independent runs are reported. Red indicates Top 1 performance, while bold indicates Top 5 performance.

dimensional representation of CE, while Figure 5.9 shows the non-normalized three-dimensional representation of *inverse bessell logarithm*. Visually, one can see

LOSS	BASE	CUTOUT	MIXUP	RANDAug	ALL
CARS-196					
CE	88.740±0.195	89.506±0.124	88.390±0.189	91.914 ±0.077	91.747±0.076
CE _{0.10}	89.628 ±0.140	89.643 ±0.084	88.700±0.101	91.909 ±0.110	91.765 ±0.159
c0	53.212±34.32	81.129±1.577	85.646±0.185	83.592±1.707	90.902±0.098
c1	89.026 ±0.893	89.583 ±0.112	85.654±0.368	90.593±2.959	90.969±0.203
c2	77.766±0.532	69.161±0.493	61.137±0.782	46.126±1.119	31.501±0.425
m0	84.932±0.245	88.394±0.150	88.394±0.150	90.728±0.213	90.899±0.085
m2	72.459±1.016	36.426±1.270	36.426±1.270	58.866±1.864	11.800±1.122
A0	86.758±0.214	88.638±0.186	88.638 ±0.186	91.063±0.152	91.466±0.123
A1	87.136±0.283	88.019±0.165	88.780 ±0.116	91.071±0.056	91.404±0.135
A2	88.820±0.190	89.255±0.196	88.720±0.157	91.899±0.077	91.775 ±0.108
A2 _{0.10}	89.810 ±0.117	89.909 ±0.259	88.999 ±0.122	91.989 ±0.095	91.780 ±0.172
OXFORD-FLOWERS-102					
CE	89.403±0.338	89.985±0.477	87.985±0.426	93.283±0.205	92.181±0.171
CE _{0.10}	92.613 ±0.110	93.710 ±0.415	89.816 ±0.422	94.786±0.156	92.753±0.211
A2	89.605±0.228	89.855±0.663	88.769±0.331	93.644±0.333	92.340±0.273
A2 _{0.10}	91.778±0.293	92.961±0.483	89.631±0.270	94.819 ±0.167	93.020 ±0.276
CALTECH-101					
CE	90.621±0.629	91.318±0.411	90.053±0.203	90.595±0.308	91.085±0.137
CE _{0.10}	93.261 ±0.241	92.932 ±0.451	90.756±0.076	91.437 ±0.235	91.381±0.158
A2	90.510±0.578	91.032±0.233	90.746±0.220	90.756±0.174	91.561±0.232
A2 _{0.10}	93.041±0.234	92.794±0.397	90.986 ±0.394	91.252±0.276	91.423 ±0.319

Table 5.15: Results for all final loss functions evaluated when fine-tuning using each of the five selected image augmentation techniques. The mean and standard deviation of the test accuracy across 5 independent runs are reported. Red indicates Top 1 performance, while bold indicates Top 5 performance.

that *inverse besse* *l*ogarithm has a slight upward curve when $\hat{y} \sim 0$ and $\hat{y} \sim 1$. To better see this difference, Figure 5.10 shows the normalized three-dimensional difference between *inverse besse* *l*ogarithm and CE. It can be seen that the largest differences occur right before y and $\hat{y}$ are the most different (when $\hat{y} \rightarrow 0$ and $y \rightarrow 1$, or $\hat{y} \rightarrow 1$ and $y \rightarrow 0$). When $\hat{y} \sim 0$ and $y \sim 1$, the differences are extremely small. However, when $\hat{y}$ is slightly increased to 0.02, the difference reaches its max value of 0.188. As a result, it can be stated that *inverse besse* *l*ogarithm has larger normalized loss values than CE when the model is almost at its most incorrect state. This typically occurs only during the very early stages of training, when the model is at its most incorrect state. Perhaps these larger normalized

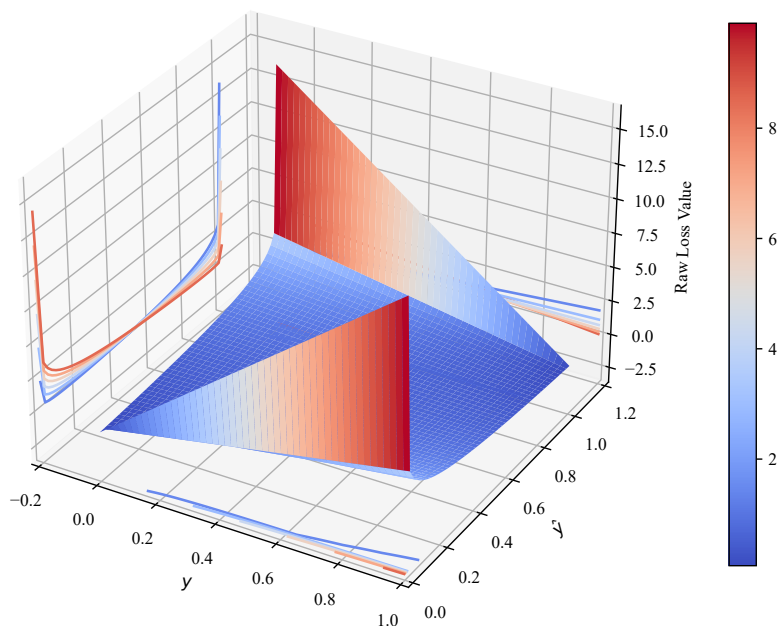


Figure 5.8: 3D plot of the raw phenotype of the cross-entropy loss.

loss values allow the model to expedite maneuvering to a better loss landscape early on during training, as larger loss values directly increase the learning rate. Chapter A gives the binary and three-dimensional phenotypes for the rest of the discovered loss functions.

5.7.3 Conclusion

In this section, I further explored the results of recent work in NLFS that revealed a disparity of rank correlation of loss functions between smaller surrogate functions and large scale models with regularization. Specifically, I revealed another disparity that exists, the lack of rank correlation of loss functions between different image augmentation techniques. I showed that the best per-

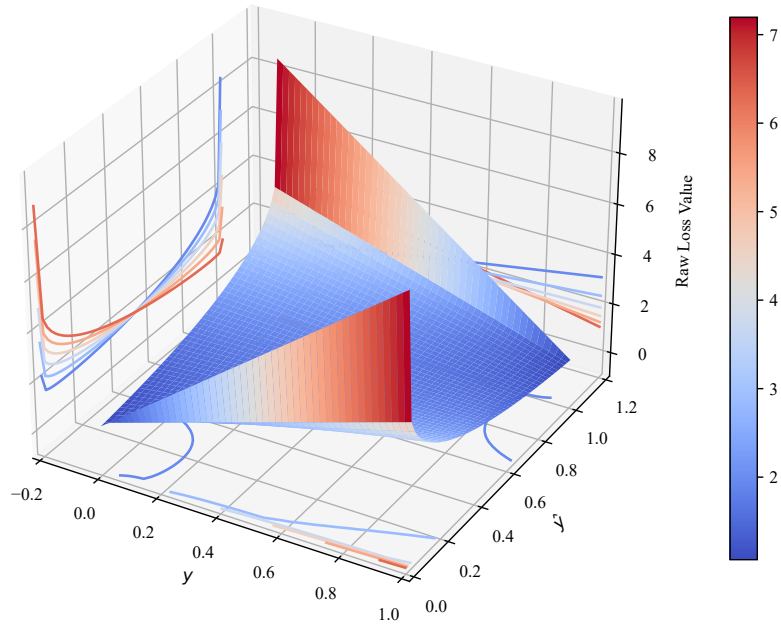


Figure 5.9: 3D plot of the raw phenotype of the $A2$ loss.

forming loss functions, from a set of randomly generated loss functions, across five different image augmentation techniques showed extremely poor rank correlation. I capitalized on this disparity by evolving loss functions on each respective augmentation technique in the hopes to find augmentation specific losses. In the end, I found a set of loss functions that outperformed cross-entropy across all image augmentation techniques, except one, on the CIFAR datasets. In addition, I found one loss function in particular ($A2$), which I call the *inverse besse logarithm*, that performed extremely well when fine-tuning on three large scale image resolution datasets. My code can be found here: <https://github.com/OUStudent/EvolvingLossFunctionsImageAugmentation>.

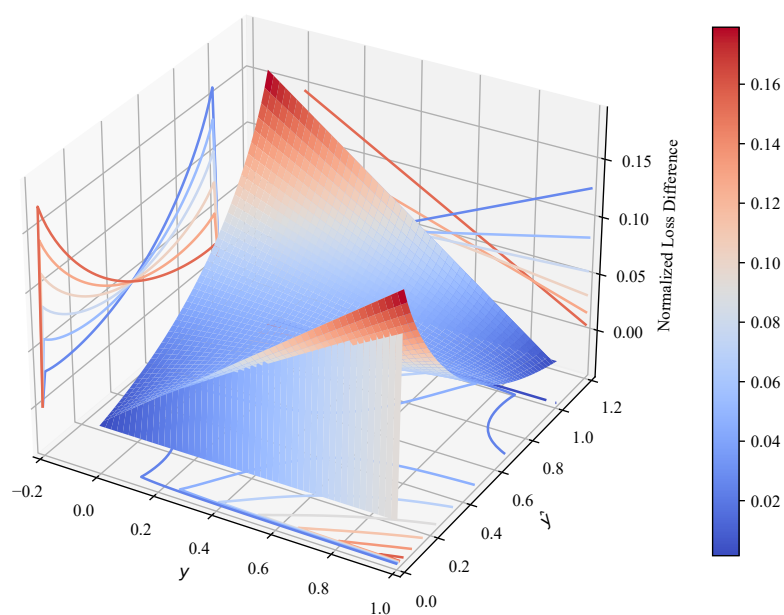


Figure 5.10: 3D plot of the difference in normalized loss values between $A2$ and cross-entropy ($A2 - CE$).

Chapter 6

Neural Optimizer Search

6.1 Overview

Deep learning optimizers are built for solving optimization problems, where the goal is to find a set of parameters that optimizes a loss function in an efficient amount of time. The optimization landscapes of deep neural networks are vast and complex terrains with steep cliffs, saddle points, plateaus, and valleys (Goodfellow et al., 2016). Being able to efficiently and intelligently maneuver across these landscapes is vital in order to achieve better performance and evaluation. The simplest way to update the weights of a network is through batch Stochastic Gradient Descent (SGD). With the goal of expediting convergence, adaptive methods have been created to efficiently scale the learning rate per parameter, such as RMSProp, AdaGrad (Duchi et al., 2011), and Adam (Kingma and Ba, 2014). Since the success of these adaptive methods, many researchers have explored other possible optimizers that could increase convergence and performance (Dozat, 2016; Zaheer et al., 2018; Ma and Yarats, 2018; Lucas et al., 2018; Chen et al., 2022a). Coupled with the optimizer is the learning rate schedule, which also

directly influences the quality of training (Xu et al., 2019; Gotmare et al., 2018).

Since the success of AutoML (He et al., 2021) and neural architecture search (NAS) (Elsken et al., 2019), automated methods have been applied to finding complete deep learning optimizers in the hopes of finding an optimizer that could outperform SGD or Adam as a drop-in replacement (Bello et al., 2017a; Carvalho et al., 2022; Chen et al., 2022b). Searching for candidate optimizer functions can be referred to as *neural optimizer search* (NOS). However, previous work in NOS has become outdated in the sense that their proposed search spaces were created with few operations and argument types. I utilize current deep learning optimizer research to update these deficiencies.

6.2 Specific Related Work

NOS is a relatively new and sparsely explored subdomain of AutoML and NAS (Bello et al., 2017a; Carvalho et al., 2022; Chen et al., 2022b). Chen et al. (2022b) evolved deep learning optimizers for image classification by utilizing a program-based search space. Carvalho et al. (2022) evolved deep learning optimizers using a grammar-based representation with very few operations and arguments. Finally, Bello et al. (2017a) learned deep learning optimizers by maximizing the reward of an recurrent neural network (RNN) controller through reinforcement learning. the work heavily stems from the success of Bello et al. (2017a), which utilized a grammar-based representation containing unary and binary operations. Not only did Bello et al. (2017a) learn the weight update equation for optimizers, but also allowed the controller to decay operands from a set of four possible decay schedules. I greatly expand on this in the proposed search space. Previous work in NOS utilized search spaces containing very simple and primitive operations. Although

Bello et al. (2017a) used a much larger search space than others (Carvalho et al., 2022; Chen et al., 2022b), and simultaneously allowed to decay operands, the search space has become outdated with regards to current deep learning optimizer research.

Since the emergence of SGD, Momentum, Nesterov Momentum (Nesterov, 1983), RMSProp, AdaGrad (Duchi et al., 2011), and Adam (Kingma and Ba, 2014), many other deep learning optimizers have been created (Loshchilov and Hutter, 2019; Dozat, 2016; Zaheer et al., 2018; Phuong and Phong, 2019; Zhang and Mitliagkas, 2017; Jain et al., 2017; Zhuang et al., 2020; Zeiler, 2012; Shazeer and Stern, 2018; Tong et al., 2019; Ginsburg et al., 2019; Ma and Yarats, 2018; Lucas et al., 2018; Zhao et al., 2020; Chen et al., 2022a). The formulation and innovation of these new deep learning optimizers can be used for the advancement of a newer and more powerful search space for NOS. Now I list the most recent deep learning optimizers that have influenced the proposed search space the most. AdaBelief (Zhuang et al., 2020) extends Adam by changing the update rule for the exponential moving average of the squared gradients by replacing the squared gradients term with the squared difference between the gradient and its momentum. AMSGrad (Phuong and Phong, 2019) extends Adam by updating the momentum term to be the maximum between the previous momentum term and the current time-step calculation. SADAM (Tong et al., 2019) extends Adam by incorporating a softplus activation around the square root of the exponential moving average of the squared gradients to calibrate the adaptive learning rate. Demon (Chen et al., 2022a) extends Adam by utilizing a decaying momentum schedule to β for improved convergence. QHM (Ma and Yarats, 2018) replaces the gradient term in standard SGD with a linear combination between the gradient and the momentum term to introduce quasi-hyperbolic momentum. AggMo (Lucas

et al., 2018) extends momentum SGD by taking the average between multiple momentum terms, each with their coefficient, to break oscillations through passive damping. I leverage these concepts by incorporating them in the creation of the new dual-joint search space.

Instead of using a grammar-based representation for the optimizers, I use the search space discussed in Chapter 4. Although reinforcement learning applied to RNN controllers has been successfully used in previous AutoML deep learning problems (Bello et al., 2017a; Zoph et al., 2018), I choose to use its antithesis of evolutionary algorithms to search the search space, which also has had great success in similar scenarios (Real et al., 2019; Chen et al., 2022b; Liu et al., 2020, 2021).

6.3 Contribution

I perform NOS by evolving optimizers on a small ConvNet evaluated on the CIFAR-10 (Krizhevsky and Hinton, 2009) dataset, transferring the final optimizers to CIFAR-100 and TinyImageNet (Le and Yang, 2015), and also fine-tuning EfficientNetV2Small (EffNetV2Small) (Tan and Le, 2021) using their published ImageNet1K weights on Flowers102 (Nilsback and Zisserman, 2008), Cars196 (Krause et al., 2013), and Caltech101 (Fei-Fei et al., 2004). My summarized contributions are: (1) I propose a new dual-joint search space for NOS, allowing for greater exploration of possible optimizers, and the simultaneous exploration of internal decay functions and learning rate schedules; (2) I propose a simple integrity check able to eliminate degenerate optimizers from wasting valuable computational time; (3) I propose a problem-specific, mutation-only genetic algorithm able to be massively parallelized; and (4) I discover and present a set

NO MOMENTUM	MOMENTUM	NESTEROV
	$z_{i+1} = \beta_i * z_i - \alpha * U$	$z_{i+1} = \beta_i * z_i - \alpha * U$
$w_{i+1} = w_i - \alpha * U$	$w_{i+1} = w_i + z_{i+1}$	$w_{i+1} = w_i + \beta_i * z_{i+1} - \alpha * U$

Table 6.1: Optimizer update equations of the form no momentum, momentum, or Nesterov momentum.

of new deep learning optimizers, learning rate schedules, and Adam variants that are capable of surpassing Adam as drop-in replacements across multiple image recognition tasks.

6.4 Methodology

6.4.1 Search Space

The proposed search space is composed of two integral parts, one for the weight update equation and another for decay functions (which includes learning rate schedules). I also expand upon Bello et al. (2017a) by allowing for the inclusion of momentum-type weight updates. Each optimizer took the form of either no momentum, momentum, or Nesterov momentum, listed in Table 6.1, where i is the i^{th} time-step, α is the learning rate, U is the weight update equation that is searched for by the evolutionary algorithm applied to the proposed search space, β_i is the momentum coefficient (cycled between 0.85 and 0.95 during training), and z_i is an intermediate state saved variable.

Optimizer

The 20 proposed operands are listed in Table 6.2, where $\hat{v}$, $\hat{s}$, $\hat{\lambda}$ are the running exponential moving averages of g , g^2 , and g^3 , with $\beta_1 = 0.9$, $\beta_2 = 0.99$, and

$\beta_3 = 0.999$. Unlike Bello et al. (2017a), I do not bias the optimizers towards Adam or RMSProp by incorporating them as operands. The 23 proposed scalar unary operations are listed in Table 6.3, where $\text{drop}(x, p)$ drops the entries of x with probability p , and $\text{norm}(x)$ scales the tensor x by the L2 norm. the 10 proposed binary operations are listed in Table 6.4, where $\text{clip}(x_1, \pm|x_2|)$ clips x_1 to the range $-|x_2|$ to $|x_2|$. In addition to simple scalar type unary operators, I also propose unary operators that perform state-holding operations. Specifically, each state-holding unary operator maintains a state variable z_i that is updated at each time-step. The 3 proposed state saving unary operators are shown in Table 6.5.

OPERANDS	
g	g^2
g^3	$\hat{v}$
$\hat{s}$	$\hat{\lambda}$
$\text{sign}(g)$	$\text{sign}(\hat{v})$
1	2
$10^{-6}w$	$10^{-5}w$
$10^{-4}w$	$10^{-3}w$
$0.3g + 0.7\hat{v}$	$0.05g^2 + 0.95\hat{s}$
$0.01g^3 + 0.99\hat{\lambda}$	
$\frac{1}{3} \sum_j^3 \beta^j v^j - g$	for $\beta^j = [0, 0.9, 0.999]$
$\frac{1}{3} \sum_j^3 \beta^j s^j - g^2$	for $\beta^j = [0, 0.99, 0.999]$
$\frac{1}{3} \sum_j^3 \beta^j \lambda^j - g^3$	for $\beta^j = [0, 0.999, 0.9999]$

Table 6.2: The proposed sets of operands and scalar unary operations for the optimizer update equation search space.

Each computational graph is initialized with four hidden state nodes and one output node. Each node is initialized by sampling its operation from the set of all binary and unary operations. Each connection for each hidden state node was sampled from all input connections and hidden state nodes. The momentum type

SCALAR UNARY OPERATIONS	
x	$-x$
$\ln(x + \epsilon)$	$\sqrt{ x }$
e^x	$ x $
$\text{sigmoid}(x)$	$\frac{d}{dx}\text{sigmoid}(x)$
$\text{softsign}(x)$	$\frac{d}{dx}\text{softsign}(x)$
$\text{softplus}(x)$	$\text{erf}(x)$
$\tanh(x)$	$\text{arctanh}(x)$
$\text{bessel}_{i1e}(x)$	$\text{arcsinh}(x)$
$\max(x, 0)$	$\min(x, 0)$
$\text{drop}(x, 0.5)$	$\text{drop}(x, 0.3)$
$\text{drop}(x, 0.1)$	$\text{norm}(x)$
$\text{erfc}(x)$	

Table 6.3: The proposed set of scalar unary operations.

BINARY OPERATIONS	
$x_1 + x_2$	$x_1 * x_2$
$x_1 - x_2$	$x_1 / (x_2 + \epsilon)$
$x_1 / \sqrt{1 + x_2^2}$	$\max(x_1, x_2)$
$\min(x_1, x_2)$	$0.95x_1 + 0.05x_2$
$\text{clip}(x_1, \pm x_2)$	$ x_1 ^{x_2}$

Table 6.4: The proposed set of binary operations.

of the weight update was sampled uniformly from the three choices mentioned in Section 6.4.1. Lastly, each connection within the graph had a probability of generating a decay function. Figure 6.1 shows an example optimizer’s weight update equation after random initialization.

Decay Function

A decay function is a scheduled function with the ability to decay operands or hidden state nodes within the optimizer graph. Unlike Bello et al. (2017a), I do not apply a simple decay function to operands and outgoing connections, but

STATE SAVING UNARY OPERATIONS
$z_{i+1} = 0.95x_i + 0.05z_i$
$z_{i+1} = x_i - z_i$
$z_{i+1} = \max(x_i, z_i)$

Table 6.5: The proposed set of state saving unary operations.

allow for the creation of decay functions through a separate search space. This subsequent search space is applied to each argument connection between hidden state nodes in the weight update graph. Much like the proposed search space for the weight update, the search space for decay functions is represented as a computational graph with free-floating nodes, along with the same properties as before. The operands in this search space refer to decay schedules. The binary operations stay the same; however, I limit the unary operations to operations with an upper limit of 1 for values between 0 and 1. This was performed to prevent scaling input argument values to greater than their original value. The updated choices of unary operators for the decay functions are shown in Table 6.6.

Unary Operations	
x	$\text{sigmoid}(x)$
$\frac{d}{dx}\text{sigmoid}(x)$	$\text{erf}(x)$
$\text{erfc}(x)$	$\tanh(x)$
$\arctan(x)$	$\text{bessel}_{i1e}(x)$
x^2	$\sqrt{x}$
$\text{softsign}(x)$	$\frac{d}{dx}\text{softisgn}(x)$
$\frac{d}{dx}\tanh(x)$	

Table 6.6: The proposed set of reduced unary operations for the decay functions

Unlike Bello et al. (2017a), I do not limit the decay schedules to decay only, but also include their increase as well. the proposed schedules are the following: linear decay (ld), linear increase (li), linear decrease with restart (ldr), linear increase

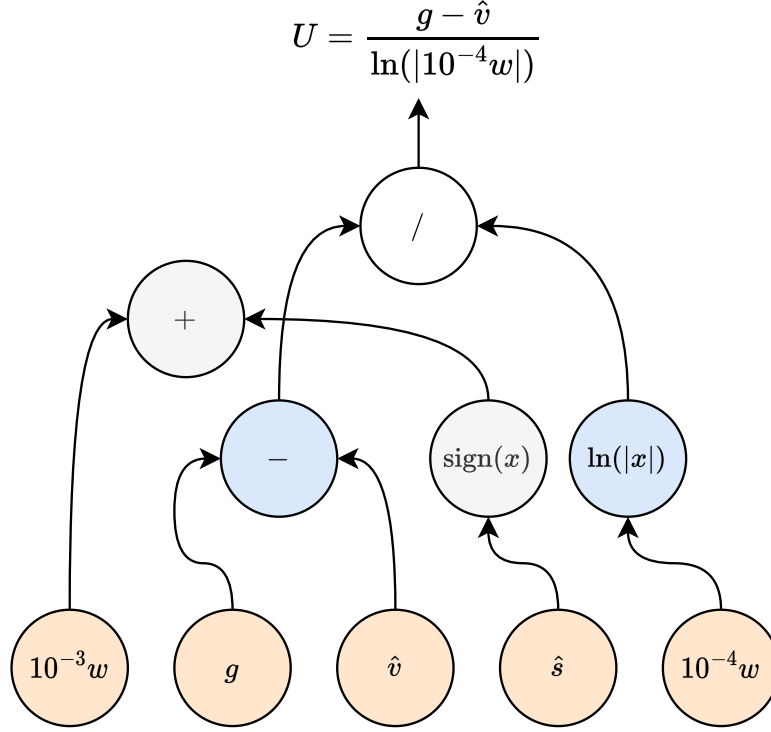


Figure 6.1: Example optimizer graph with two active (blue) hidden state nodes, two inactive hidden state nodes (grey), and one root node (white). The final weight update equation is given above the root node. Note that this does not include momentum type.

with restart (*lir*), cosine decay (*cd*), cosine increase (*ci*), cosine decrease with restart (*cdr*), cosine increase with restart (*cir*), cyclic cosine decay (*ccd*), cyclic cosine increase (*cci*), exponential decay (*ed*), exponential increase (*ei*), demon decay (*dd*) (Chen et al., 2022a), and demon increase (*di*). In total, there are 14 different possible schedules to be used as operands within the decay search space graph, shown in Table 6.7.

Lastly, the decay functions can be used to create unique learning rate schedules. Unless the decay function is applied inside a non-distributive function, such as $\tanh(x)$, the decay function can be factored out alongside the learning rate. As

Decay Schedules	
<i>ld</i>	$1 - t/T$
<i>li</i>	t/T
<i>ldr</i>	$1 - \text{mod}(2t, T)/T$
<i>lir</i>	$\text{mod}(2t, T)/T$
<i>cd</i>	$0.5(1 + \cos(t\pi/T))$
<i>ci</i>	$0.5(1 - \cos(t\pi/T))$
<i>cdr</i>	$0.5(1 + \cos(\pi \text{ mod}(2t, T)/T))$
<i>cir</i>	$0.5(1 - \cos(\pi \text{ mod}(2t, T)/T))$
<i>ccd</i>	$0.5(1 + \cos(2t\pi/T))$
<i>cci</i>	$0.5(1 - \cos(2t\pi/T))$
<i>ed</i>	$0.01^{(t/T)}$
<i>ei</i>	$1 - 0.01^{t/T}$
<i>dd</i>	$\frac{(0.95*(1-t/T))}{(0.05+0.95*(1-t/T))}$
<i>di</i>	$0.95 - \frac{0.95*(1-t/T)}{(0.05+0.95*(1-t/T))}$

Table 6.7: The proposed set of decay schedule operands for the decay function search space. t denotes the current timestep while T denotes the maximum number of timesteps.

a result, the proposed complete search space can optimize not only the weight update equation, but also the learning rate schedule and internal decay functions for non-distributive functions.

Each computational graph for the decay functions is initialized with one hidden state node and one output node. This was performed to simplify the decay schedules. Each node in the decay function graph was initialized similar to the weight update equation graph. Figure 6.2 shows an example decay function being applied to a connection from within the example optimizer graph from Figure 6.1.

6.4.2 Integrity Check

All together, the proposed search for both the weight update equation and decay functions contain an enormous number of possible combinations, creating an

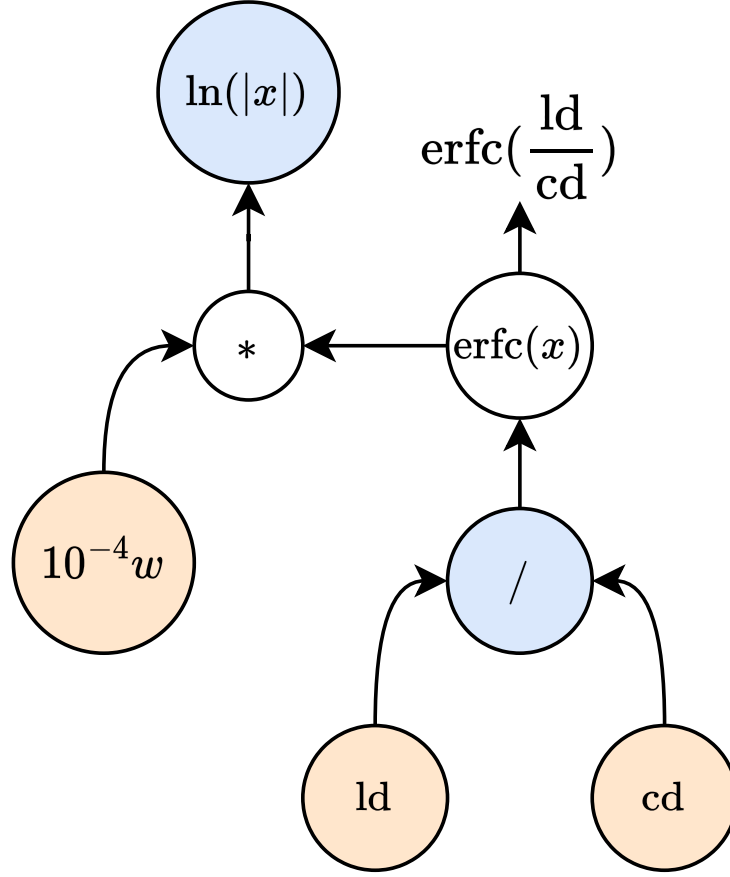


Figure 6.2: Example decay function applied to the $10^{-4}w$ operand before being applied in the $\ln(|x|)$ operation. The decay graph contains one active (blue) hidden state node, zero inactive hidden state nodes (grey), and one root node (white). The final decay function equation is given above the root node.

extremely sparse search space for workable optimizers. To prevent degenerate optimizers from wasting valuable evaluation time, an integrity check was implemented from the motivation of Liu et al. (2021) in their evolution of loss functions. The integrity check needs to be accurate in detecting degeneracy while also being simple to compute. the proposed integrity check was based on the shifted sphere optimization problem, where the goal is to minimize $f(x) = \sum_i^n (x_i - \beta_i)^2$ for n variables, where β_i is a shift constant for the i^{th} variable.

Each optimizer was tested at minimizing the optimization problem with different learning rates. Each optimizer was given the same initial point and was run for a set number of iterations. If none of the final objective function values for each learning rate was below a hand-designed threshold, it was rejected as it was unable to achieve moderate convergence on a simple problem. The chosen initial point, associated shift constants, number of variables, number of iterations, and threshold were hand designed based on the success from current deep learning optimizers.

Upon initialization, if a connection was sampled to contain a decay function, the associated decay function underwent a simpler integrity check: the decay function was rejected if it scaled the component to less than 0 or more than 1. The decay function looped initialization until it found a graph that satisfied this integrity check.

6.4.3 Surrogate Function

With the goal of learning optimizers that perform well on large-scale deep learning models, evaluating each candidate optimizer on a large-scale model can be extremely computationally expensive. Surrogate functions are cheap proxy models used to estimate the performance of deep learning components at large-scale (Liu et al., 2020; Bingham et al., 2020; Bello et al., 2017a). From the success of Bello et al. (2017a), I also choose to utilize a small ConvNet as the surrogate function. The ConvNet contains three convolution layers, followed by batch normalization (Ioffe and Szegedy, 2015) and Swish activation (Ramachandran et al., 2017). Except for the first convolution layer, each subsequent convolution layer maintained a stride of 2, and doubled the previous number of filters/channels. The starting

number of filters/channels for the first convolution layer was referred to as the *base*.

6.4.4 Early Stopping

Even though an integrity check was implemented to prevent degenerate optimizers from wasting valuable evaluation time, some optimizers may not perform well for deep learning tasks. To prevent these optimizers from wasting valuable time, two early stopping mechanisms were implemented. First, each optimizer was trained on the ConvNet with *base* 48 (totaling to 0.212M parameters) for 800 steps at each of the following learning rates: 10, 1, 0.1, 0.01, 0.001, 0.0001, and 0.00001. In addition, a one cycle cosine decay learning rate schedule with a linear warmup (Smith, 2018a) was applied to every optimizer. As a result, all optimizers were learning to change the given learning rate schedule using their decay functions during evolution. If none of the final training accuracies from each learning rate session yielded an accuracy above a hand-designed threshold of 25%, the optimizer was rejected. Second, if the optimizer passed the first early stopping mechanism, the optimizer was trained again on the ConvNet, except for 8,000 steps using the best found learning rate. After 1,000 steps, if the training accuracy ever fell below a hand-designed threshold of 40%, training was stopped and the best validation accuracy was returned as the fitness value. If the optimizer passed both early stopping criteria, the best validation accuracy was used as the optimizers final fitness value. For the validation dataset, 5,000 of the 50,000 training images for CIFAR-10 were set aside as validation.

6.4.5 Particle-Based Genetic Algorithm

Real et al. (2019) applied *regularized evolution*, a mutation only, population based genetic algorithm (GA), to the NASNet search space with great success, but I found it non-optimal for the problem. In some preliminary runs, *regularized evolution* converged extremely fast. I believe that this occurrence was due to the extremely sparse search space, paired with the integrity check, creating an extremely sharp local minima for each optimizer. As a result, only a few non-degenerate solutions exist nearby each optimizer, which are quickly exhausted within a few iterations of the algorithm. To address this problem, I propose a problem-specific, particle-based GA with the same components as *regularized evolution*, (1) mutation only, (2) aging, and (3) parallelism (as described in Chapter 4). The full pseudocode for the proposed particle based GA is given in Algorithm 5.

Because the search space is extremely sparse, a poor initial point would be likely to generate poor mutations. As a possible solution, I use an enlarged initial population, as I performed in Chapter 5. For each run, I generate $10n$ randomly initialized optimizers and evaluate them on a ConvNet with $base=32$. The best n were then taken to be the initial points for each run. I ran the particle-based GA three times, each with different configurations. With the goal of having each run evaluate approximately 1,800 optimizers over the course of evolution, I ran the following three configurations: (1) $n = 50, k = 6, t = 6$; (2) $n = 50, k = 5, t = 7$; and (3) $n = 50, k = 7, t = 5$. Although the proposed algorithm allows for massive parallelization, it was performed sequentially as I only had access to one NVIDIA 4090 GPU during evolution. Each run took approximately six days to complete.

Mutation was performed by selecting an active node randomly, and then

Algorithm 5 Particle-Based Genetic Algorithm

```
1: Input:  $n$  (Number of Particles),  $k$  (Number of mutations),  $t$  (Number of
   timesteps)
2:  $initialParticles = \text{OptimizerInitialization}(100 \times n)$ 
3:  $validationAccuracies = \text{ConvNet}(initialParticles, \text{base} = 32)$ 
4:  $particles = \text{BestParticles}(initialParticles,$ 
    $validationAccuracies, n)$ 
5: for  $particle$  in  $particles$  do
6:   for  $timestep$  in  $t$  do
7:      $mutations = []$ 
8:     for  $j$  in  $k$  do
9:       repeat
10:         $child = \text{Mutation}(particle)$ 
11:        until  $child.\text{IntegrityCheck}()$ 
12:         $mutations.append(child)$ 
13:      end for
14:       $mutationValidationAccuracies = \text{ConvNet}(mutations,$ 
         $\text{base} = 48)$ 
15:       $particle = \text{BestParticle}(mutations,$ 
         $mutationValidationAccuracies)$ 
16:    end for
17: end for
```

performing one of six possible mutations. First, the operation was mutated by randomly sampling the list of remaining available operations. Second, the argument connection was mutated by randomly sampling the list of remaining available arguments and hidden state nodes. Third, a unary operation was changed to a binary operation, or visa-versa. For unary to binary, the extra connection was sampled randomly from the list of possible connections (either argument connections or other hidden state nodes); for binary to unary, one of the connections was randomly dropped. Fourth, which was only available for binary operations, the argument connections were swapped. Fifth, the momentum type of the weight update was sampled from the list of remaining types. Lastly, the decay schedule for one of the connections of an active node was mutated. If a

decay schedule was not present for the given connection at time of mutation, a randomly initialized decay schedule was assigned. If a decay schedule was present, it was either deleted or mutated. If mutated, the computational graph of the decay schedule was mutated using the operations one through four.

Each mutation was passed through the integrity check, from Section 6.4.2, to ensure non-degeneracy. If the integrity check failed, mutation was looped until a child mutation was found that passed the integrity check. All mutated child optimizers were then evaluated using the surrogate function described in 6.4.3. If the optimizer failed the first early stopping mechanism, mutation was performed again until a child mutation was found that passed the first early stopping mechanism.

6.4.6 Optimizer Elimination Protocol

I previously showed the disparity of correlation between selected surrogate functions and larger models for loss functions. Therefore, I devised an optimizer elimination protocol that progressively eliminated optimizers based upon their performance on increasingly larger ConvNets, because I am unsure how well the optimizers trained on a small ConvNet would correlate to larger models. For each run of the particle-based GA, I took the best 50 optimizers and trained them on a ConvNet with *base* 64 (0.375M parameters) for 16,000 steps. The best 24 were then taken and trained on a ConvNet with *base* 96 (0.839M parameters) for the same number of steps. The best 12 were then taken and trained with *base* 128 (1.487M parameters). Unlike the surrogate function used during evolution, the optimizer elimination protocol ran each optimizer three times for each ConvNet size to get an average of the validation accuracy on the CIFAR-10 dataset to use

Name	Momentum	Weight Update Equation	Decay Functions
Opt1	None	$(0.3g + 0.7\hat{v}) + \text{softsign}(\text{clip}(10^{-5}w, 10^{-5}w - (0.05g^2 + 0.95\hat{s})))$	None
Opt2	None	$(0.3g + 0.7\hat{v}) + \text{softsign}\left(\frac{10^{-5}w}{\sqrt{1+(10^{-5}w-(0.05g^2+0.95\hat{s}))^2}}\right)$	None
Opt3	None	$(0.3g + 0.7\hat{v}) + \text{softsign}\left(\frac{10^{-5}w}{\sqrt{1+(10^{-5}w-(0.01g^3+0.99\hat{\lambda}))^2}}\right)$	None
Opt4	None	$\frac{t_i^1(0.3g+0.7\hat{v})}{t_i^2\text{clip}(2,t_i^3e^{\hat{v}})}$	$t_i^1 = \text{erfc}(\text{erfc}(ci))$ $t_i^2 = \frac{d}{dx}\tanh(cir)$ $t_i^3 = \arctan(dd)$
Opt5	None	$\frac{t_i^1(0.3g+0.7\hat{v})}{t_i^2\text{clip}(2,e^{\hat{v}})}$	$t_i^1 = \text{erfc}(\text{erfc}(ci))$ $t_i^2 = \frac{d}{dx}\tanh(cir)$
Opt6	None	$\frac{t_i^1(0.3g+0.7\hat{v})}{t_i^2 t_i^3e^{10^{-4}w} }$	$t_i^1 = \text{erfc}(\text{erfc}(ci))$ $t_i^2 = \frac{d}{dx}\tanh(cir)$ $t_i^3 = \frac{d}{dx}\tanh(ci)$
Opt7	None	$\tanh(t_i^1\text{arcsinh}(0.3g + 0.7\hat{v}))$	$t_i^1 = \max(cci, lir)$
Opt8	None	$\text{arcsinh}(t_i^1\text{arcsinh}(0.3g + 0.7\hat{v}))$	$t_i^1 = \max(cci, lir)$
Opt9	Nesterov	$t_i^1ge^{\arctan(0.05g^2+0.95\hat{s})}$	$t_i^1 = \text{erfc}(ed)$
Opt10	Nesterov	$\text{bessel}_{i1e}(\text{bessel}_{i1e}(t_i^1g))$	$t_i^1 = dd * li$

Table 6.8: Final 10 optimizers found after evolution. The momentum type, weight update equation, and decay functions used by each optimizer are listed. I refer to each optimizer in the paper by the associated name.

for comparison.

Finally, after performing optimizer elimination, the best six from each run were taken and trained on EfficientNetV2Small (20.3M parameters) using their proposed progressive RandAug regularization (Cubuk et al., 2020) for 64,000 steps. This was performed to ensure the best optimizers performed well on large-scale deep learning models. From the final 18 optimizers (best six from each of the three runs), the best six were hand selected based on their validation accuracy

and uniqueness. To ensure that the search space for each final optimizer had been completely exhausted, the final six optimizers from the proposed optimizer rejection protocol were taken to be used as initial particles in the proposed GA. Each particle ran for $k = 12$, $t = 3$, and with $base = 64$. In addition, each child mutation was trained on the ConvNet three times to obtain a mean of the validation accuracy. This was performed to reduce the noise of training stochastic neural networks. The final best 24 optimizers from the six runs were evaluated on EfficientNetV2Small. The best 10 optimizers from all optimizers evaluated on EfficientNetV2Small were then taken to be the final reported optimizers.

6.4.7 Adam Variants

I performed two supplementary experiments with the goal of (1) obtaining variants of the Adam optimizer and (2) variants of learning rate schedules for Adam. For the first experiment, I hand programmed the Adam optimizer into the search space and ran the proposed GA using Adam as the initial particle. It ran using $k = 16$, $t = 3$, and $base$ 48. I allowed all components to be mutated. For the second experiment, I used the Adam optimizer again as the initial particle to the proposed GA, except this time only the decay schedules were allowed to be mutated for each connection, not the weight update equation. It ran using $k = 12$, $t = 4$, and $base$ 48.

6.5 Results

6.5.1 Final Optimizers

The final 10 optimizers discovered after optimizer elimination are listed in Table 6.8. There appear to be three families of optimizers present in the final results, with two outsiders. All optimizers, except the two outsiders (Opt9 and Opt10), used the quasi-hyperbolic momentum (Ma and Yarats, 2018) update equation $(0.3g + 0.7\hat{v})$, showcasing the importance of a linear combination between the gradient and its exponential moving average. The first family of optimizers (Opt1, Opt2, and Opt3) took the form $(0.3g + 0.7\hat{v}) + \text{softsign}(x)$, where x was dependent upon the optimizer. This family heavily relied upon a weight decay of $10^{-5}w$ inside the softsign operation. The second family of optimizers (Opt4, Opt5, and Opt6), took the form $(t_i^t(0.3g + 0.7\hat{v})) / (t_i^2x)$ where x was dependent upon the optimizer. This family heavily relied upon decay functions, with the majority directly affecting the learning rate schedule. The last family of optimizers (Opt7 and Opt8) are exactly the same, except for Opt8 trading the outside $\tanh(x)$ function of Opt7 for $\text{arcsinh}(x)$. Lastly, the two outsiders (Opt9 and Opt10) seem to be descendants from two non-included families of optimizers; however, they are the only two final optimizers to incorporate a momentum type. Seven of the final ten optimizers contain decay functions, with four of them directly effecting the learning rate schedule.

6.5.2 Supplementary Experiments

The final best five optimizers found from the first supplementary experiment for finding variants of Adam are shown in Table 6.9. As one can see, the evolution

swapped the division operator between $\hat{v}$ and $\hat{s}$ in favor of the clip operator, as well as replacing the square root operator for $\hat{s}$ with some other function. From the second supplementary experiment on finding learning rate schedules for Adam, I took the best schedules found, along with the best learning rate schedules found during the standard evolution, to report in Table 6.10. Note that each learning rate schedule present is multiplied by lr_{cd} , the cosine learning rate decay schedule with linear warmup mentioned in Section 6.4.4. See Appendix 6.8 for the plots of the discovered learning rate schedules and internal decay functions.

Adam Variants	
A	$\hat{v}/\sqrt{\hat{s}}$
A1	$\text{clip}(\hat{v}, \sqrt{\hat{s}})$
A2	$\text{clip}(\hat{v}, \ln(\hat{s}))$
A3	$\text{clip}(\hat{v}, \sqrt{ \ln(\hat{s}) })$
A4	$\text{clip}(\hat{v}, \text{sigmoid}(\hat{s}))$
A5	$\text{norm}(\text{clip}(\hat{v}, \sqrt{\hat{s}}))$

Table 6.9: The best five Adam variants found during the first supplementary experiment. Note that all optimizers listed here did not use a momentum type.

Learning Rate Schedules			
LR1	$\frac{\text{erfc}(\text{erfc}(ci))}{\frac{d}{dx} \tanh(cir)}$	LR6	$\frac{\arctan(li) * \text{erfc}(cci)}{\sqrt{\frac{d}{dx} \text{softsign}(cci)}}$
LR2	$\frac{\text{erfc}(\text{erfc}(ci))}{\frac{d}{dx} \tanh(cir) * \frac{d}{dx} \tanh(ci)}$	LR7	$\frac{1}{\sqrt{\frac{d}{dx} \text{softsign}(lir)}}$
LR3	$\frac{\arctan(li)}{\frac{d}{dx} \tanh(lri) * \sqrt{\frac{d}{dx} \text{softsign}(di)}}$	LR8	$\frac{1}{\sqrt{\text{softsign}(\arctan(ei))}}$
LR4	$\frac{\text{sigmoid}(li)^2}{\text{sigmoid}(2 * \text{softsign}(ld))}$	LR9	$\tanh(\max(cci, lri))$
LR5	$\frac{1}{\sqrt{\text{erf}(ed)}}$		

Table 6.10: The best nine learning rate schedules found from standard evolution and the supplementary experiment on finding learning rate schedules for Adam.

6.6 Transferability Experiments

To assess generalization, all found optimizers, Adam variants, and learning rate schedules (trained on Adam) were transferred to various image classification tasks. Each were trained on CIFAR-10 and CIFAR-100 from scratch using EffNetV2Small with progressive RandAug regularization strategy. Each were also trained on TinyImageNet (Tiny) using a custom ResNet9 (6.5M parameters). For further study on fine-tuning scenarios, each were trained for fine-tuning EffNetV2Small using its official ImageNet1K weights on Flowers102, Cars196, and Caltech101. For comparison, the results for Adam, RMSProp, SGD, and Nesterov momentum are recorded as well for each experiment. In addition, as a baseline comparison to Bello et al. (2017a), I have included their best two discovered optimizers, PowerSign-ld (linear decay) and AddSign-ld (linear decay). Lastly, because many of the discovered optimizers heavily utilize the quasi-hyperbolic momentum term, I also include the original QHM (Ma and Yarats, 2018) optimizer for comparison. To assess how much the decay functions influence the quality of the found optimizers containing them, each were re-run without them. Opt4 was retrained twice, once without using t_i^1 and t_i^2 (Opt4₁); and a second time without using any t_i (Opt4₂). Opt6, Opt7, Opt8, Opt9, and Opt10 were all retrained without using any t_i (Opt6₁, Opt7₁, Opt8₁, Opt9₁, Opt10₁). See Section 6.9 for exact implementation and training details for each experiment. The results for all image recognition experiments are recorded in Table 6.11. In addition to image recognition, a supplementary experiment was performed where the final optimizers were evaluated on language modeling for the PTB (Marcus et al., 1993) dataset. Those results are discussed in Section 6.10.

6.7 Results and Discussion

OPTIMIZER	CIFAR-10	CIFAR-100	FLOWERS	CARS	CALTECH	TINY
ADAM	95.69±0.15	77.24±0.33	97.36±0.14	91.30±0.16	91.41±0.11	47.43±0.82
RMSProp	95.55±0.05	77.62±0.23	97.44±0.21	90.68±0.16	91.53±0.34	46.93±0.38
SGD	95.04±0.19	77.34±0.30	97.48±0.07	91.39 ±0.21	92.76 ±0.44	44.60±0.27
NESTEROV	95.21±0.22	77.78±0.24	97.73 ±0.18	91.29±0.17	91.52±0.28	48.38 ±0.44
POWERSign	95.51±0.25	78.01±0.26	97.61 ±0.18	90.39±0.24	92.23 ±0.19	47.84±0.27
ADDSign	95.27±0.14	78.15±0.15	97.52±0.17	90.52±0.16	91.87±0.25	48.15 ±0.18
QHM	95.00±0.10	77.45±0.18	97.66 ±0.19	91.31 ±0.16	92.10±0.25	47.49±0.20
OPT1	95.58±0.19	78.55±0.18	97.45±0.09	91.47 ±0.28	92.22 ±0.10	47.46±0.41
OPT2	95.48±0.20	78.87±0.31	97.36±0.11	91.54 ±0.09	92.16±0.29	47.56±0.40
OPT3	95.47±0.27	78.95 ±0.29	97.76 ±0.12	91.54 ±0.20	92.15±0.29	47.83±0.73
OPT4	95.95 ±0.06	79.31 ±0.15	97.24±0.12	89.36±0.36	92.23 ±0.25	47.96±0.81
- OPT4 ₁	95.21±0.17	77.45±0.27	97.35±0.09	91.40 ±0.21	92.06±0.10	45.88±0.56
- OPT4 ₂	95.48±0.12	77.42±0.37	97.45±0.09	90.33±0.22	92.10±0.20	45.13±0.92
OPT5	96.02 ±0.19	79.05 ±0.37	97.23±0.21	89.08±0.10	92.18±0.23	47.80±1.02
OPT6	96.16 ±0.16	79.46 ±0.36	97.27±0.17	89.06±0.23	91.71±0.32	48.51 ±0.42
- OPT6 ₁	94.98±0.15	77.55±0.12	97.51±0.15	88.87±0.17	92.17±0.45	48.20 ±0.53
OPT7	95.82±0.07	79.44 ±0.21	97.25±0.12	89.32±0.19	91.65±0.15	47.92±0.54
- OPT7 ₁	94.99±0.09	77.45±0.26	97.37±0.11	88.94±0.21	92.18±0.28	48.23 ±0.50
OPT8	95.74±0.09	79.23 ±0.29	97.26±0.15	89.46±0.17	91.98±0.25	47.96±0.19
- OPT8 ₁	95.02±0.20	77.47±0.41	97.59 ±0.09	89.07±0.10	92.08±0.22	47.91±0.36
OPT9	95.85±0.19	79.06 ±0.49	97.54±0.15	90.57±0.21	91.08±0.35	46.64±1.64
- OPT9 ₁	95.64±0.18	77.51±0.23	97.51±0.13	90.92±0.07	91.42±0.32	47.44±0.66
OPT10	96.04 ±0.13	79.80 ±0.27	97.11±0.16	89.20±0.19	91.97±0.29	47.60±0.41
- OPT10 ₁	95.51±0.18	78.12±0.11	97.32±0.13	90.58±0.10	91.99±0.16	48.82 ±0.48
LR1	95.99 ±0.15	77.64±0.20	97.31±0.14	90.53±0.10	91.36±0.49	43.23±0.29
LR2	96.13 ±0.10	78.10±0.17	97.18±0.24	90.25±0.30	91.21±0.43	43.31±0.19
LR3	96.23 ±0.07	78.48±0.07	97.29±0.23	90.56±0.04	91.20±0.34	41.89±0.27
LR4	95.57±0.06	77.15±0.15	97.53±0.24	90.45±0.26	90.90±0.24	44.80±0.32
LR5	95.97 ±0.13	78.55±0.45	96.72±0.20	91.47±0.09	88.99±0.19	47.04±0.65
LR6	94.57±0.11	74.42±0.29	97.41±0.16	89.63±0.36	91.76±0.16	39.81±0.76
LR7	95.63±0.12	77.31±0.40	97.23±0.20	91.79±0.21	89.93±0.14	46.84±0.38
LR8	95.28±0.15	76.64±0.23	97.21±0.08	91.74±0.11	89.47±0.35	48.20 ±0.32
LR9	95.76±0.05	77.69±0.36	97.61 ±0.20	90.61±0.17	91.39±0.35	43.27±0.45
A1	95.26±0.16	77.50±0.22	97.66 ±0.13	91.36 ±0.15	92.20 ±0.28	47.92±0.45
A2	95.17±0.16	77.60±0.35	97.19±0.18	91.22±0.20	92.21 ±0.12	48.12±0.40
A3	95.04±0.23	77.04±0.29	97.50±0.09	91.29±0.06	92.20 ±0.16	47.65±0.22
A4	95.04±0.19	77.29±0.27	97.50±0.18	91.30±0.13	92.21 ±0.13	48.15±0.43
A5	94.95±0.38	77.55±0.23	97.65 ±0.21	91.60 ±0.14	91.92±0.18	48.40 ±0.50

Table 6.11: Results for all optimizers and learning rate schedules for CIFAR-10, CIFAR-100, Flowers102 (Flowers), Cars196 (Cars), Caltech101 (Caltech), and TinyImageNet (Tiny). The mean and standard deviation of the test accuracy across 5 independent runs are reported. Red indicates Top 3 performance, while bold indicates Top 8 performance.

From Table 6.11, one can see that many optimizers, learning rate schedules, and Adam variants were able to outperform Adam and the other standard deep

learning optimizers across the image recognition datasets. The results can be broken down into two sections: those trained from scratch (CIFAR and Tiny) and those fine-tuned (Flowers, Cars, and Caltech). When training from scratch on CIFAR and Tiny, Opt4, Opt5, Opt6, Opt7, Opt8, and Opt10 performed relatively well compared to the baseline deep learning optimizers. However, Opt6 elevated itself above the rest by always being in the Top 3. Opt6₁ is the quasi-hyperbolic momentum update equation $(0.3g + 0.7\hat{v})$ scaled by exponential of the weight decay $\exp(10^{-4}w)$. For small weight values, the exponential weight decay approaches one, making Opt6 equivalent to standard quasi-hyperbolic momentum. However, Opt6 simultaneously learned decay functions. These decay functions can be factored out along side the learning rate, giving rise to LR2. The success of Opt6 is heavily dependent upon the inherently learned LR2 schedule, as Opt6₁ always under-performed Opt6 when training from scratch. Opt10₁ always out-performed Nesterov’s momentum when training from scratch. Appendix 6.8 discusses the effect of the scaling on the gradients for Nesterov. I empirically noticed that Opt10₁ liked large learning rates around 10. I believe that the double scaling of the gradients clips larger gradient values, allowing for larger learning rates to scale gradients near zero to have more of an effect, which empirically seems beneficial. When fine-tuning using transfer learning, there appears to be a flip in performance between the optimizers, as Opt1, Opt3, A1, and A5 achieved Top 3 atleast once across Flowers, Cars, and Caltech. Opt1, Opt2, and Opt3 can be seen as extensions of QHM as they are detailed by QHM+ x , where x , was dependent upon the optimizer. Although the results for Opt1-Opt3 are similar to QHM for fine-tuning, the results from CIFAR reveal that the additional terms are beneficiary as Opt1-Opt3 all outperformed QHM. Appendix 6.8 discusses the effect of each learned term to the weight update equation. For the Adam variants,

the majority of their success occurs when fine-tuning, as all achieved Top 8 once across the three datasets. Not regarding CIFAR-10, A1 and A5 outperformed Adam across all other datasets, giving empirical evidence to the swapping of the division operator with the clip, along with adding a normalization operator for A5.

6.8 Analysis of the Final Optimizers

In this section, more investigation is performed into the analysis of the final discovered decay functions, learning rate schedules, and adaptations of the QHM optimizer.

6.8.1 Observing the Final Learning Rate Schedules and Internal Decay Functions

The base learning rate schedule applied to all experiments was one cycle cosine decay with linear warmup. The learning rate is linearly warmed up to its peak, being held for a number of iterations, before being decayed using cosine decay back down to zero. All evolved optimizers learn to adapt this schedule through the use of distributive decay functions. For visualization, Figure 6.3, Figure 6.4, and Figure 6.5 plot the discovered learning rates over time normalized between 0 and 1 to allow for comparison. The actual scaled differ quite drastically. In each, the one cycle cosine decay with linear warm-up base schedule is plotted in dashed lines for comparison. From the figure, there appear to be two families of learning rates (Figure 6.3 and Figure 6.4), along with two outsider learning rates (Figure 6.5). Figure 6.6 plots the three internal decay functions used by the final

optimizers, Opt4, Opt7/8, and Op10.

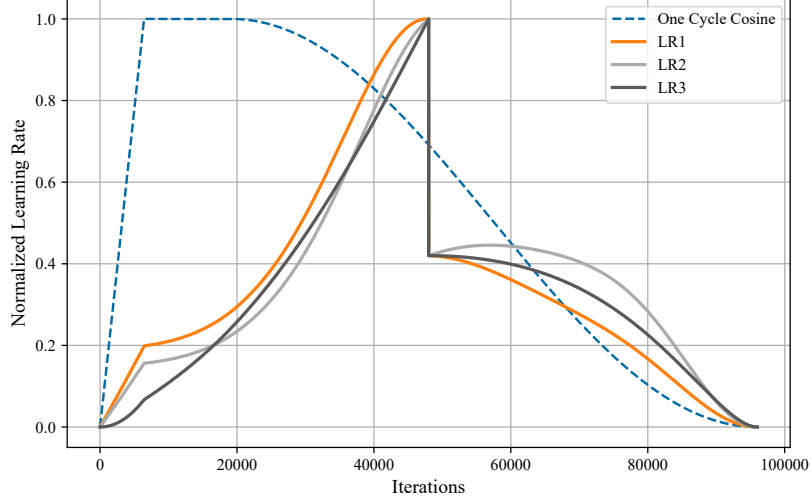


Figure 6.3: Learning Rate Family 1

6.8.2 Observing the Final Decay Function

To assess how the three internal decay functions affect the weight update over time, in conjunction with the respective learned learning rate, eight time slices were plotted for the output distribution of the weight update. A one cycle learning rate totalling to 96,000 steps, with the first 6,400 being used for warm up and then being held for the next 12,800 before being cosine decayed, was used for obtaining the time slices. The eight time slices were recorded, at iterations 1K, 10K, 25K, 45K, 65K, 75K, 85K, and 93 or 95K. For example, for the internal decay function t_i^3 in Opt4, given an input of $x = e^{\hat{v}}$ ranging from 0 to 20, the output of $\text{clip}(2, t_i^3 x)$ times the learned learning rate of t_i^1/t_i^2 is plotted in Figure 6.7. As one can see, this conjunction warms up to clipping 2 before being decayed over time, eventually clipping 2 to zero. Although it may seem that this learned internal decay function

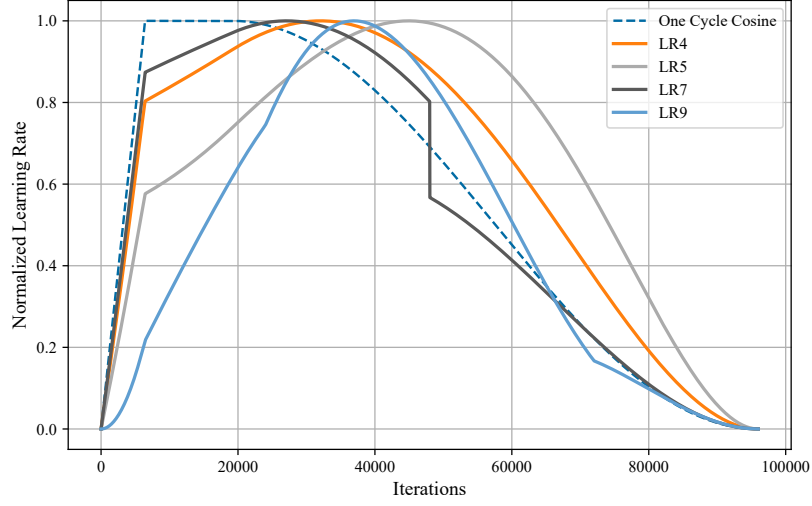


Figure 6.4: Learning Rate Family 2

inside the clip operator with conjunction of the learned learning rate decreases the weight update value, it actually increases the weight update as it is used as the divisor of $(0.3g + 0.7\hat{v})$, where smaller values yield larger quotients. Unlike Figure 6.7, Figure 6.8 and 6.9 give the raw weight update values for optimizers 7 and 10. Figures 6.8 and 6.9 are extremely similar as they perform symmetric squashing functions on their respective inputs, greatly limiting the magnitude of the output. Scaling of the output is achieved by changing the maximum learning rate.

6.8.3 Observing the Adaption of QHM

Optimizers Opt1, Opt2 and Opt3 can be seen as extensions of QHM, $QHM + \text{softsign}(x)$, where x is dependent upon the optimizer. In an aid to assess how each $\text{softsign}(x)$ effects the weight update equation for their various variable inputs, Figure 6.10, 6.11, 6.12 gives the 3D plots for Opt1, Opt2, and Opt3. For

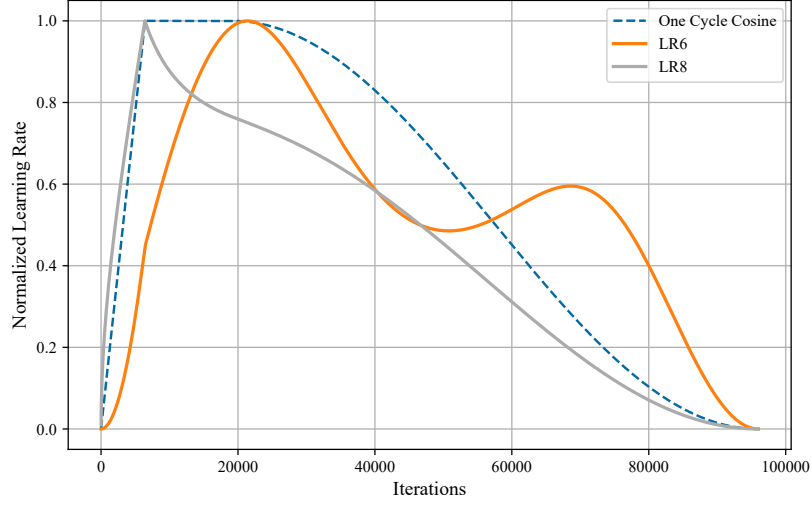


Figure 6.5: Outsider Learning Rates

example, the two primary inputs to $\text{softsign}(x)$ for Opt3 are the weights w and the exponential moving average of the cubed gradients $0.01g^3 + 0.99\hat{\lambda}$. With these serving as the x and y axis, the output of

$$\text{softsign}\left(\frac{10^{-5}w}{\sqrt{1 + (10^{-5}w - (0.01g^3 + 0.99\hat{\lambda}))^2}}\right)$$

from Opt3 can be seen in Figure 6.12. As one can see, the term only influences the weight update equation when the exponential moving average of the cubed gradients is centered around zero, where the sign and magnitude of w directly control the sign and magnitude of the update. Because the weight update equations for Opt4/5, Opt6, and Opt7 can be easily represented by two variable inputs, their 3D weight update equation with respect to their variable inputs is plotted in Figure 6.13, 6.14, and 6.15.

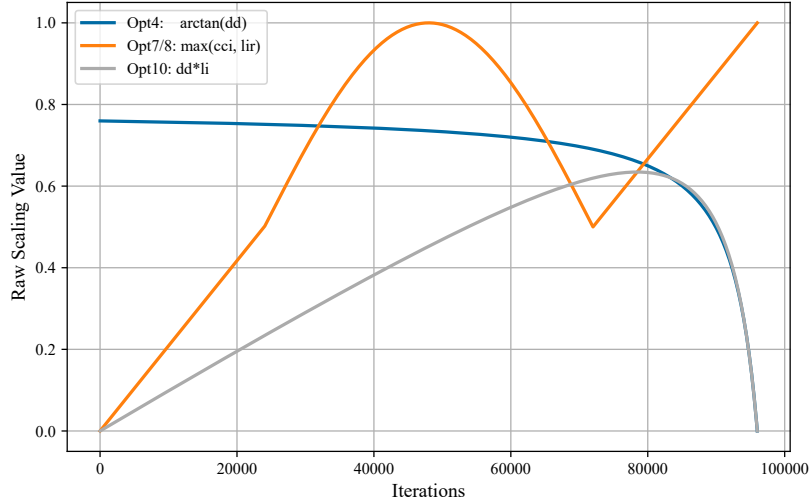


Figure 6.6: Evolved Internal Decay Functions

6.9 Experimental Implementations and Details

Six primary datasets were used in this work, CIFAR-10, CIFAR-100, Stanford-CARS196, Oxford-Flowers102, Caltech101, and Tiny. For each experiment, all optimizers and Adam-based learning rate schedules performed a learning rate test by selecting the best learning rate, from 10, 1, 0.1, 0.01, 0.001, 0.0001, and 0.00001, after a 1/3 of the training epochs (for Adam-based learning rate schedules, the learning rate test was performed for 2/3 of the training epochs). For CIFAR-10 and CIFAR-100, all optimizers were trained for 96,000 steps with a batch size of 64 on EffNetV2Small. Progressive RandAug regularization was used for image augmentation. For TinyImageNet, all optimizers were trained for 300,000 steps with a batch size of 256 on ResNet9 (6.5M parameters). For image augmentation, the images were resized using bicubic interpolation to 72x72, before a random 64x64 crop was applied, followed by random translation and horizontal flipping. For CIFAR-10, CIFAR-100, and TinyImageNet, the learning

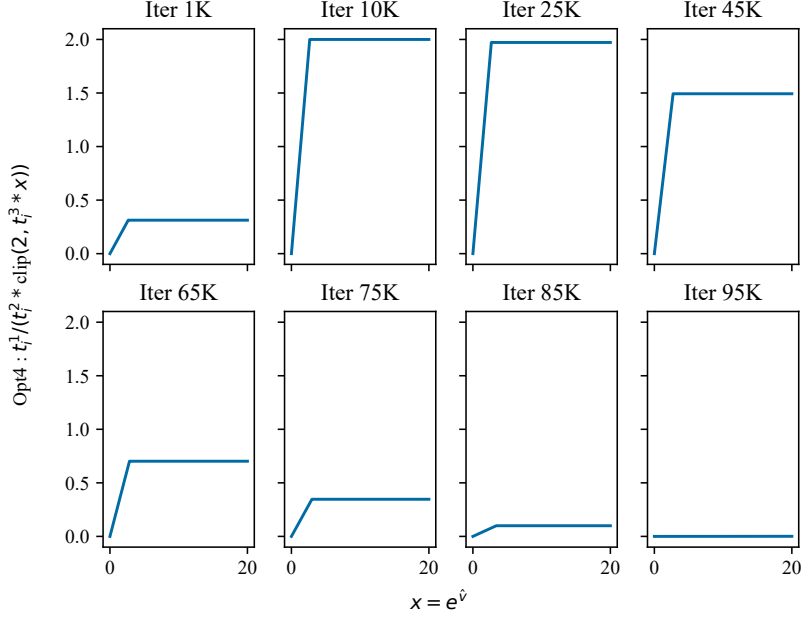


Figure 6.7: Opt4 Internal Decay

rate was linearly warmed up for 6,400 steps, being held for 12,800 steps at its maximum value, before being decayed with cosine annealing. For fine-tuning, the published ImageNet1K weights for EffNetV2Small were used as the initial weights before being fine-tuned for 10,000 steps on Flowers102, Cars196, and Caltech101 with a batch size of 64. For regularization, dropout of 0.5 was placed before the softmax layer, while RandAug regularization of magnitude 15 was used for image augmentation. For fine-tuning, all images were resized to 224x224. Due to the limited number of weight updates, the learning rate test was altered to fine-tune the learning rate. Specifically, the best found learning rate using the procedure described earlier was tested after being halved and multiplied by five. For example, a best found learning rate of 0.01 is tested at 0.05 and 0.005, where the final learning rate is chosen from this final set. All code was implemented in Tensorflow (Abadi et al., 2015), although the PTB experiments were performed

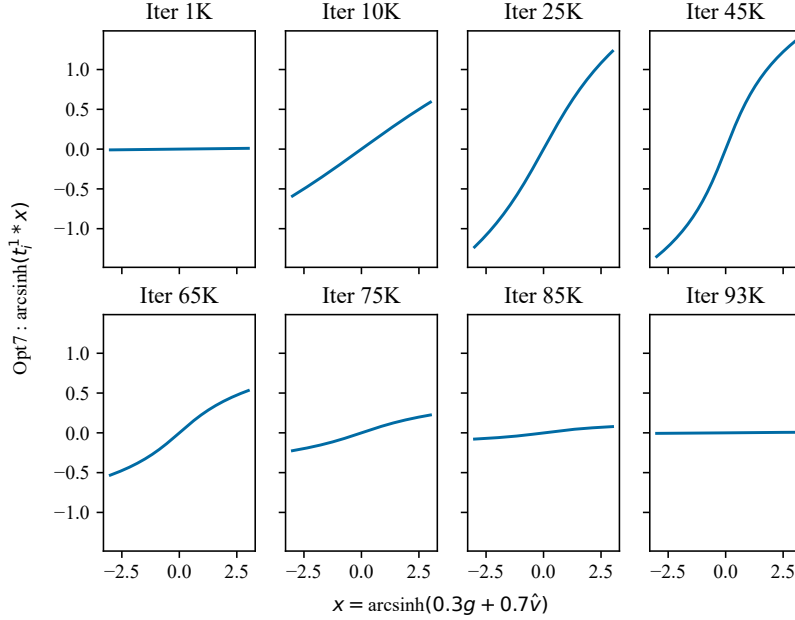


Figure 6.8: Opt7 Weight Update

in Pytorch (Paszke et al., 2019).

6.10 Supplementary PTB Experiments

To assess generalization across deep learning domains, the final optimizers and learning rate schedules were transferred to language modeling on the PTB dataset. A three layer AWD-LSTM (Merity et al., 2017) with 1150 hidden units (24M parameters), softmax weight tying, variational dropout, embedding dropout, gradient norm clipping, truncated back propagation, and L_2 weight decay, was trained for 99,600 steps. The best validation perplexity for each optimizer is reported, where lower is better. The results are recorded in Table 6.12. From Table 6.12, three key points can be discussed. First, the best optimizer is arguably tied between Nesterov’s momentum and Opt10₁, Opt10 without a decay function.

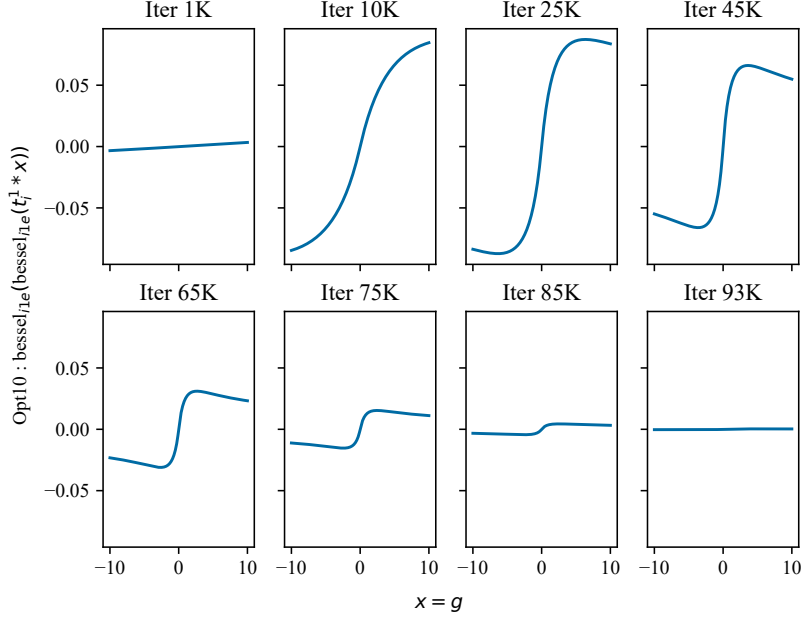


Figure 6.9: Opt10 Weight Update

Opt10₁ can be seen as an extension of Nesterov’s momentum with weight gradient scaling. In addition, Opt10₁ greatly outperformed Opt10, possibly showcasing how the learned decay function is specifically designed for image recognition as Opt10 outperformed Opt10₁ on CIFAR. Second, all the discovered Adam variants outperform Adam for language modeling. Due to the clipping nature of the Adam variants, the absolute magnitude of the exponential moving average of the gradients is inherently reduced during the weight update, which prevents a moving average of exploding gradients from affecting the quality of the model, a common problem with LSTMs (Zhang et al., 2019). However, LR5, LR6, and LR7 were able to push standard Adam past all the Adam variants without the need of clipping. Third, despite Opt1, Opt2, and Opt3 being extensions of QHM, all outperformed QHM, indicating the importance the respective additional terms.

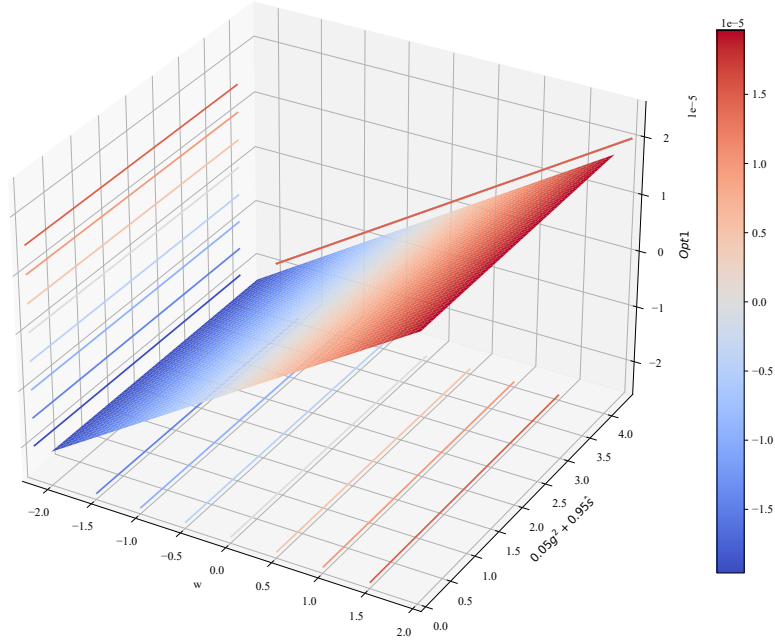


Figure 6.10: $\text{softsign}(\text{clip}(10^{-5}w, 10^{-5}w - (0.05g^2 + 0.95\hat{s})))$

6.11 Conclusions

In this chapter, I expanded on previous research in NOS by proposing a new search space containing the most up-to-date research from deep learning optimizers. This new search space allows for the simultaneous optimization of the weight update equation, internal decay functions, and adaptation of the learning rate schedule. I searched the space using the proposed particle-based GA that is able to be massively parallelized. I save computational resources by incorporating an integrity check to get rid of degenerate optimizers with the initialization and mutation operator in order to make intelligent mutational choices. In addition, I perform two supplementary experiments to obtain Adam variants and new learning rate schedules for Adam. After transferring the final optimizers, I found five

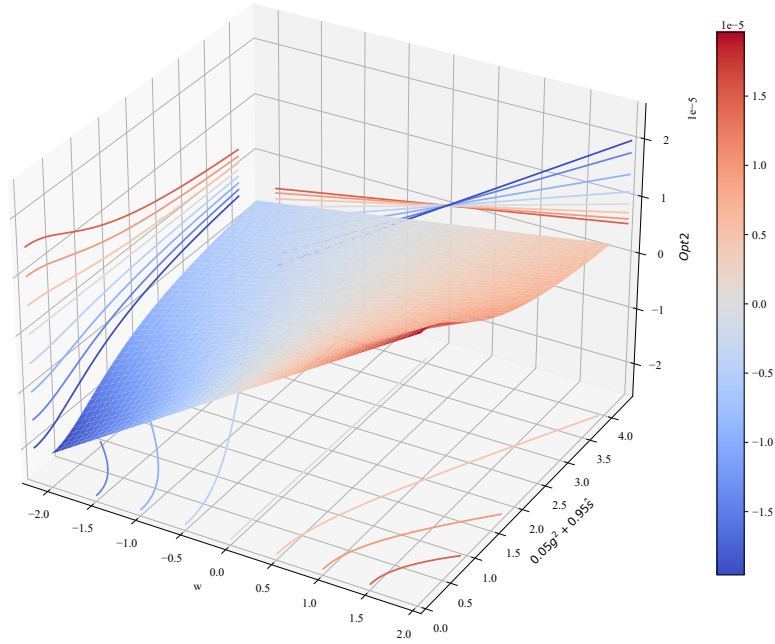


Figure 6.11: $\text{softsign}\left(\frac{10^{-5}w}{\sqrt{1+(10^{-5}w-(0.05g^2+0.95\hat{s}))^2}}\right)$

new optimizers and two Adam variants, that consistently outperformed standard deep learning optimizers when training from scratch and fine-tuning on image classification tasks: Opt1, Opt3, Opt4, Opt6, Opt10, A1, and A5. My code can be found here: <https://github.com/OUStudent/NeuralOptimizerSearch>.

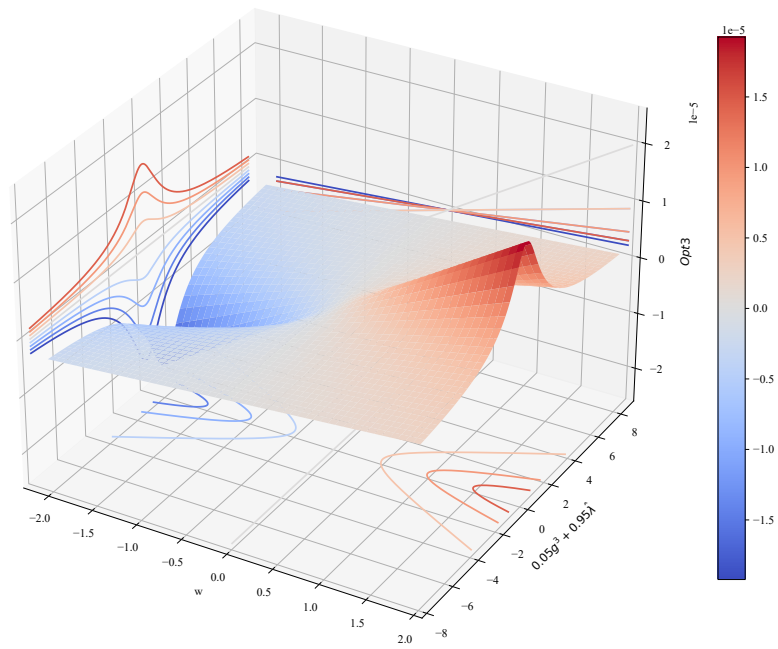


Figure 6.12: $\text{softsign}\left(\frac{10^{-5}w}{\sqrt{1+(10^{-5}w-(0.01g^3+0.99\hat{\lambda}))^2}}\right)$

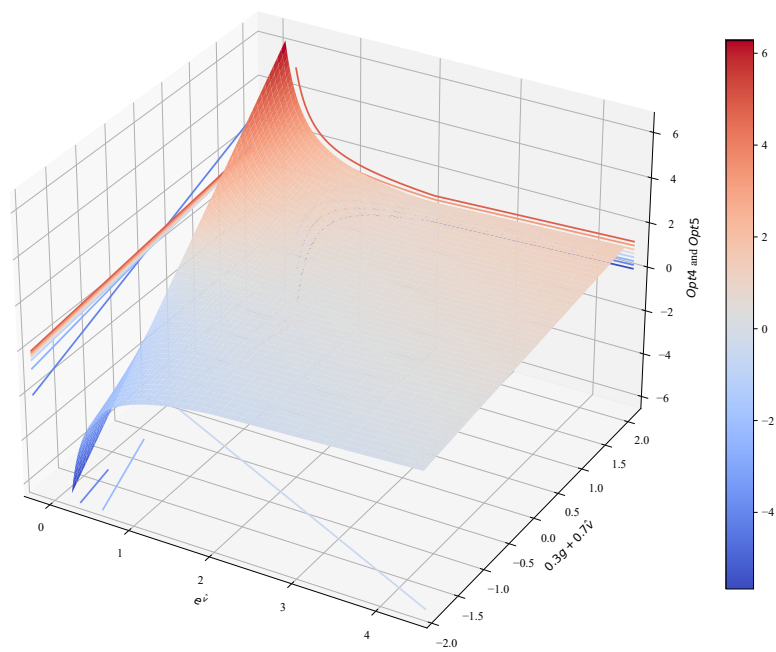


Figure 6.13: $\frac{0.3g+0.7\hat{v}}{e^{\hat{v}}}$

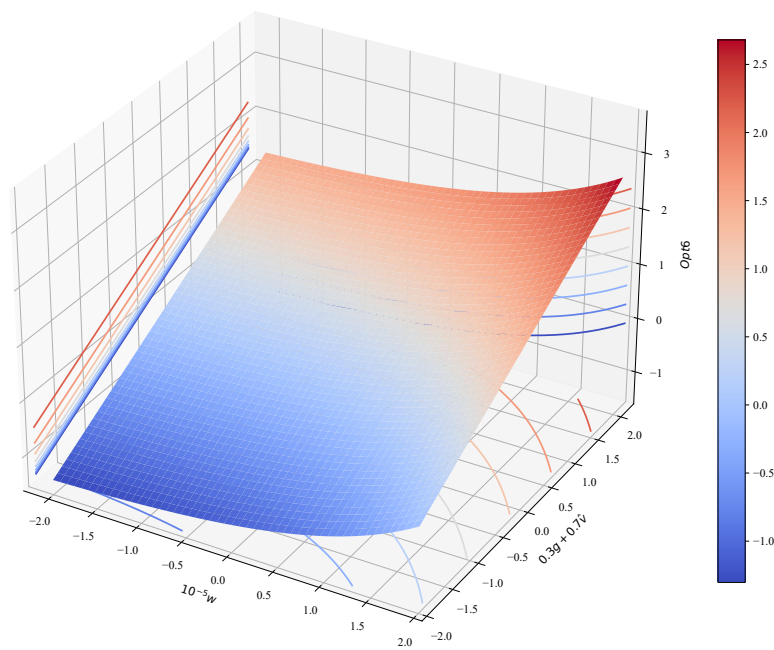


Figure 6.14: $\frac{0.3g+0.7\hat{v}}{e^{10^{-4}w}}$

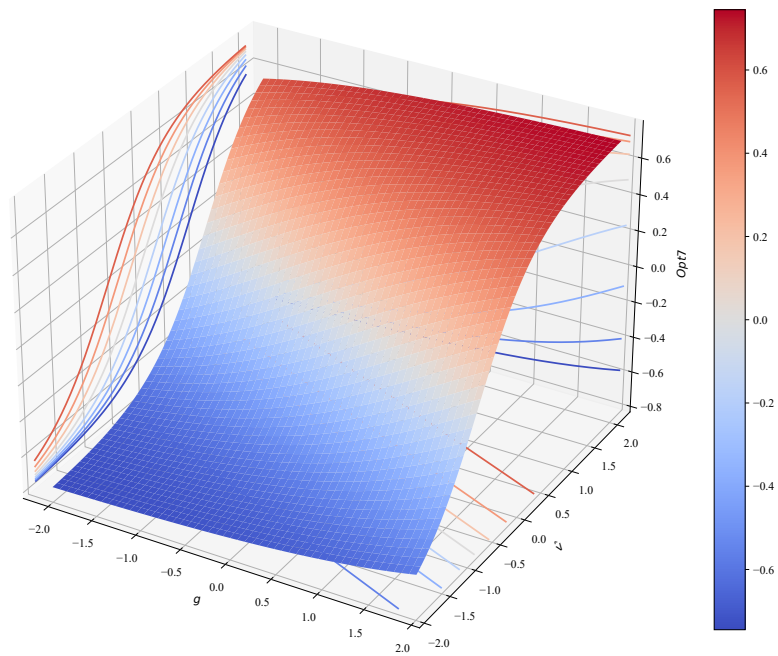


Figure 6.15: $\tanh(\operatorname{arcsinh}(0.3g + 0.7\hat{v}))$

OPTIMIZER	PTB
ADAM	68.56±0.13
RMSProp	67.47 ±0.20
SGD	68.07±0.33
NESTEROV	65.33 ±0.22
POWERSIGN	71.64±0.20
ADDSIGN	70.72±0.17
QHM	69.22±0.15
OPT1	67.74 ±0.10
OPT2	67.62 ±0.18
OPT3	67.62 ±0.15
OPT4	69.81±0.21
- OPT4 ₁	67.45 ±0.12
- OPT4 ₂	68.99±0.10
OPT5	71.56±0.15
OPT6	70.27±0.13
- OPT6 ₁	69.25±0.22
OPT7	70.47±0.14
- OPT7 ₁	69.29±0.24
OPT8	72.51±0.17
- OPT8 ₁	69.11±0.19
OPT9	68.73±0.19
- OPT9 ₁	70.69±0.23
OPT10	71.99±0.15
- OPT10 ₁	65.34 ±0.15
LR1	73.19±0.26
LR2	72.14±0.31
LR3	71.82±0.33
LR4	73.31±0.32
LR5	67.87±0.07
LR6	67.84 ±0.11
LR7	68.01±0.15
LR8	73.20±0.12
LR9	72.57±0.20
A1	68.26±0.20
A2	68.22±0.12
A3	68.28±0.12
A4	68.22±0.14
A5	68.41±0.21

Table 6.12: Results for all optimizers and learning rate schedules for PTB. The mean and standard deviation of the best validation perplexity is reported (lower is better) across 5 independent runs are reported. Red indicates Top 3 performance, while bold indicates Top 8 performance.

Chapter 7

Neural Learning Rate Search

7.1 Overview

The learning rate for a deep learning optimizer controls the magnitude of the step size to take into the direction of the gradient. Too small of a magnitude and the model either takes much longer to converge, or never reaches the same solution space, as a larger learning rate. Too large of a magnitude and the weights of the model could diverge by either jumping out of an optimal solution space or lead to infinity weights.

To combat both of these issues, learning rate schedules have been devised to change the learning rate throughout the training strategy. Typical learning rate schedules *warm-up* the learning rate from zero to its maximum, being held for a number of epochs, before finally decaying back down to zero. This allows for the early parts of learning to take larger steps, while decaying the step size for better convergence towards the end of training.

Fine-tuning is a popular technique in transfer learning where a larger model trained on a harder problem is transferred to a smaller problem (Kornblith et al.,

2019). Typically, these models are not trained for a full training schedule on the smaller problem, but a reduced number of epochs in order to save computational resources. Being able to achieve sufficient performance in a limited number of epochs can be partially responsible to the learning rate schedule.

7.2 Specific Related Work

The learning rate of a deep learning optimizer has long been known as a key factor in the final quality of a deep learning model (Bottou, 1998; Lecun et al., 1998). Success in terms of improved model quality from introducing a learning rate schedule can be seen in early works, such as LeNet (Lecun et al., 1998). These early learning rate schedules were piece-wise functions that often times halved the learning rate at fixed epoch locations (He et al., 2016; Zagoruyko and Komodakis, 2016). In doing so, the lower learning rate towards the end of the training schedule would allow for better convergence to the final solution. Much research has gone into creating new deep learning optimizers or SGD variants to reduce the need of learning rate tuning (Kingma and Ba, 2014; Zeiler, 2012; Tieleman et al., 2012; Schaul et al., 2013; Martens, 2010). Most notably, the Adam optimizer (Kingma and Ba, 2014) was proposed as an adaptive deep learning optimizer with the ability to adaptively change the learning rate for each individual parameter of the model up to the maximum set learning rate as a means to increase performance and reduce the necessity of learning rate tuning. However, performance can still be increased with scheduled learning rates.

Despite intuitional belief that large initial learning rates would improve performance, success has been found in *warming up* learning rates. The learning rate starts from zero and is *warmed up* to its maximum value before being decayed

(Vaswani et al., 2017; Li et al., 2019). Learning rate warm up schemes were introduced as heuristics for adaptive deep learning optimizers, such as Adam and RMSProp, to reduce the variance of their batch statistics (Liu et al., 2019).

Smith has contributed much to the conversation on learning rate schedules (Smith, 2017, 2018b; Smith and Topin, 2019). Smith (2017) showed that the learning rate can be cycled between a minimum and maximum bound during training, either with or without a piece-wise or exponentially decreasing value, for faster convergence. Smith and Topin (2019) also showed faster convergence can be achieved using only one cycle for cyclic learning rates schedules. In this one cycle cyclic learning rate schedule, the learning rate exhibits curriculum learning until it reaches its peak, before being decayed, exhibiting simulated annealing (Smith and Topin, 2019).

From the inspiration of restart techniques in gradient free optimization, Loshchilov and Hutter (2016) proposed a simple warm restart technique for SGD. In this proposal, the learning rate is decreased using cosine annealing multiple times throughout training (i.e. restarts).

Transfer learning is a popular technique in computer vision where a model trained on a larger and harder dataset is transferred over for fine-tuning on a smaller and easier dataset for improved performance (Kornblith et al., 2019; Tan and Le, 2019, 2021). It is standard to take the final ImageNet trained model and fine-tune on these smaller datasets. The primary objective of fine-tuning in these scenarios is to achieve acceptable performance with a limited computational budget, not requiring a full training strategy from scratch.

From the success neural architecture search (Zoph et al., 2018), other components of neural networks have been explored. In the context of this work, both Bello et al. (2017a) and I searched for deep learning optimizers along with internal

decay functions. These internal decay functions can be factored out alongside with the learning rate, giving rise to new learning rate schedules. In this work, I expand upon both Bello et al. (2017a) and my previous work on optimizers by solely focusing on searching for learning rate schedules. Although the learning rate schedules found by me, previously, were shown to be superior to one cycle cosine for the Adam optimizer, the scaling of the final found learning rates were very different, with many learning to either increase or decrease the maximum learning rate. This inherently biases learning rate schedules to particular datasets and architectures. In the hopes of finding maximum learning rate agnostic learning rates schedules, I eliminate this bias in the work for accurate comparison by always re-scaling all learning rates between zero and the maximum set learning rate.

7.3 Contribution

My previous work on neural optimizer search was not primarily focused on finding learning rate schedules. In this work, I expand upon my previous work by solely focusing on *neural learning rate search*, not only when training from scratch, but also fine-tuning. I perform a mutation-only, particle-based genetic algorithm for massive parallelization to search the proposed search space. For schedules when training from scratch (96,000 weight updates), I search for candidate learning rate schedules on CIFAR-10 using ResNet18v2. The best were then transferred to ResNet52v2 and WideResNet (He et al., 2016; Zagoruyko and Komodakis, 2016) for CIFAR-10 and CIFAR-100 (Krizhevsky and Hinton, 2009), and EfficientNetV2B0 (Tan and Le, 2021) for TinyImageNet (Le and Yang, 2015). For schedules when fine-tuning (10,000 weight updates), I search for candidate learning rate schedules using an aggregated fitness on Cars, Caltech, and Flowers using EfficientNetV2B0.

The best were then transferred to Cars, Caltech, Flowers, Dogs, and Pets (Krause et al., 2013; Fei-Fei et al., 2004; Nilsback and Zisserman, 2008; Khosla et al., 2011; Parkhi et al., 2012) using EfficientNetV2Small.

In the end, I found multiple learning rate schedules across both environments that outperformed standard, and previously found, learning rate schedules in terms of aggregated fitness, and mean and median rank across the tasks.

7.4 Methodology

7.4.1 Search Space

My previously designed search space for learning rate schedules in Chapter 6 had its limitations. Namely, each learning rate schedule was created by relying on the joint multiplication (or division) of decay schedules (needing to jointly optimize multiple graphs simultaneously); limited computational graph sizes for each decay schedule (only two node computational graphs); and, limited number of decay schedule operands (included only basic schedules). I expand upon these limitations in the work.

Unary Operations			Binary Operations	
step _{inc2}	$\sigma(x)$	$\frac{d}{dx}\sigma(x)$	$x_1 + x_2$	$x_1 * x_2$
step _{inc3}	softsign(x)	$\frac{d}{dx}\text{softsign}(x)$	$x_1 - x_2$	$x_1 / (x_2 + \epsilon)$
step _{inc4}	erf(x)	erfc(x)	$x_1 / \sqrt{1 + x_2^2}$	$\max(x_1, x_2)$
step _{inc5}	tanh(x)	arctanh(x)	$\min(x_1, x_2)$	cat _{0.05} (x_1, x_2)
step _{dec2}	bessel _{i1e} (x)	$\sqrt{x}$	cat _{0.25} (x_1, x_2)	cat _{0.50} (x_1, x_2)
step _{inc3}	x	x^2	cat _{0.75} (x_1, x_2)	cat _{0.95} (x_1, x_2)
step _{dec4}	step _{dec5}	reverse(x)		

Table 7.1: The proposed set of unary and binary operations.

For the adaptation of the my previous search space, a computational graph is used to represent a candidate learning rate. Each computational graph consists

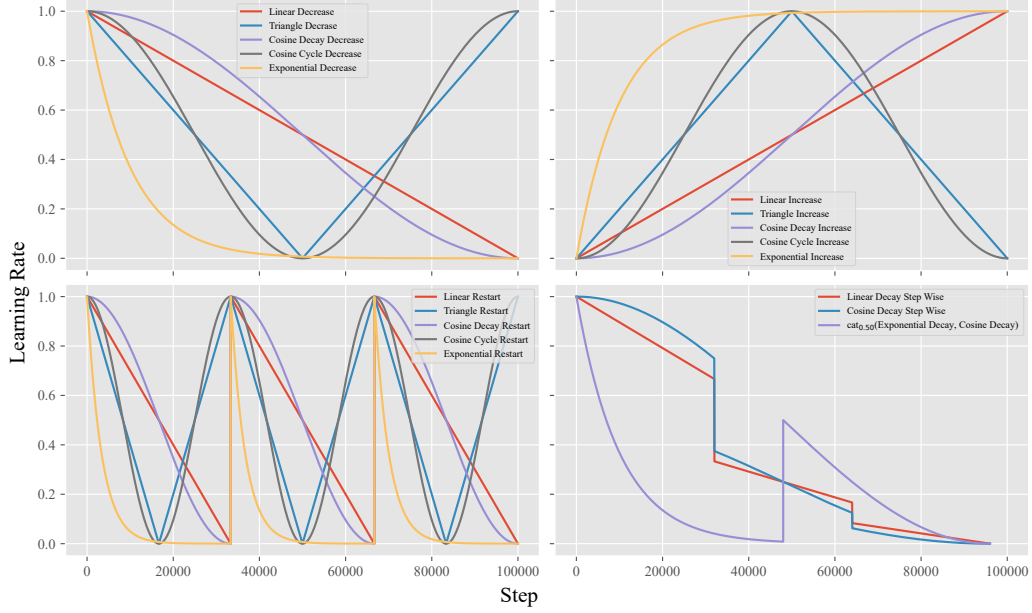


Figure 7.1: (Top Left) Decreasing decay functions. (Top Right) Increasing decay functions. (Bottom Left) Decreasing decay functions with two restarts. (Bottom Right) Example of step wise decay (two steps) for both linear and cosine decay; and, example of the concatenation binary operator ($cat_{0.5}(x_1, x_2)$) between exponential and cosine decay.

of three integral parts. (1) *Base learning rate schedules*, which perform a basic learning rate schedule that can be used as an input connections for later in the graph. (2) Free-floating nodes (four in total), termed *hidden state nodes*, that accept an input connection (or another hidden state node) as an argument, perform an operation (either unary or binary), and output its result to be used by other hidden state nodes. (3) A final hidden state node, termed the *output node*, which is used as the final operation before the learning rate schedule is produced. The final output node is a multiplication between an output from a selected hidden state node and a *scheduled decay function*. Because the output node only selects the output from one hidden state node, there remains a probability that some of the nodes in the graph become unused. This mechanism allows for the

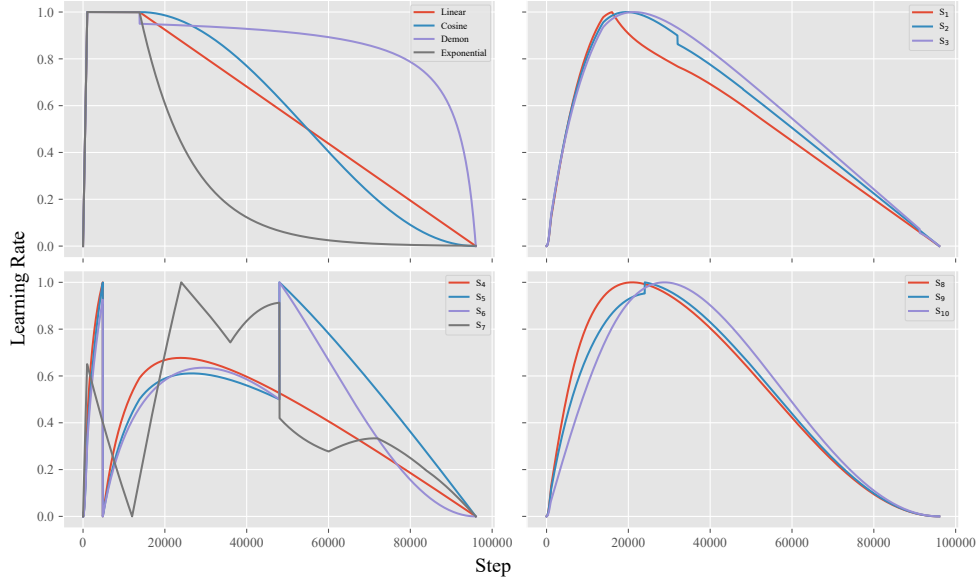


Figure 7.2: (Top Left) Standard learning rate schedules. (Rest) Best 10 found learning rate schedules for training from scratch.

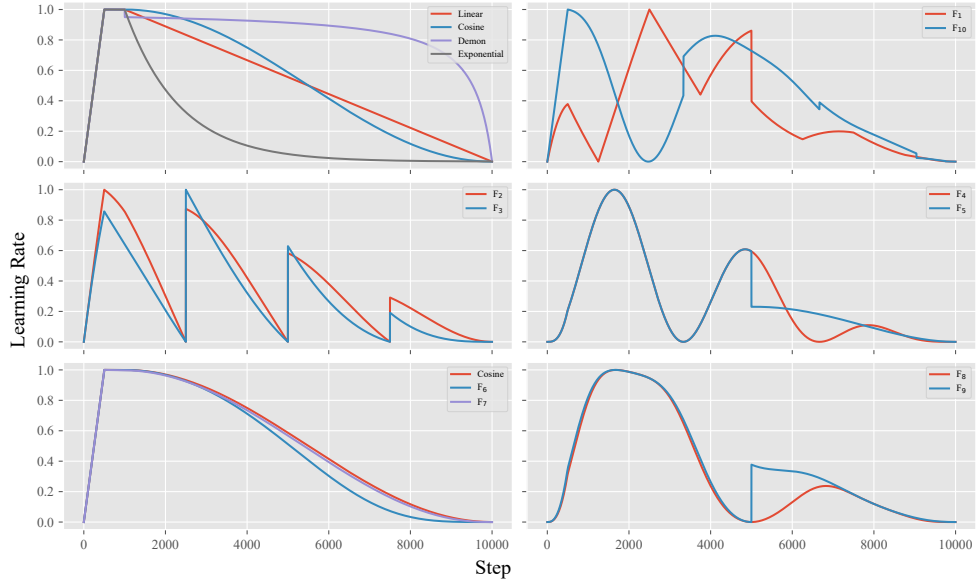


Figure 7.3: (Top Left) Standard learning rate schedules. (Rest) Best 10 found learning rate schedules for fine-tuning.

learning rate schedule to grow or shrink in complexity through the utilization of more or lesser nodes. An input connection to a node can either be the output

from another node or base learning rate schedule.

There exist five base *decay functions*: linear, cosine, exponential, triangle, and demon (Chen et al., 2022a). The *base learning rate schedules* are an adaption of a *decay function* through the following two choices. First, the decay function is either increasing or decreasing. Second, the function either has 0, 1, 2, or 3 restarts. With these options, numerous base learning rate schedules can be formed prior to even being used inside the graph. The final *scheduled decay function* is a standard learning rate schedule for a particular decay function. Meaning, it includes a typical warmup, hold, and decay period. Figure 7.2 (Top Left) depicts the four possible types of scheduled decay functions. For each candidate learning rate schedule, one of these scheduled decay functions was chosen to be multiplied by the input connection of the output node. More information can be found in Section 7.6.

Each hidden state node performs either a unary or binary operation. In the proposed search space, there exist 22 possible unary and 13 possible binary operations. The 22 proposed unary operations are listed in Table 7.1, where bessel_{i1e} is the Bessel function of degree 1; σ is sigmoid activation; softsign is softsign activation; softplus is softplus activation; erf is the Gauss error function; erfc is the complementary error function; $\frac{d}{dx}$ refers to the derivative of the respective function; $\text{step}_{\text{inc}_i}$ refers to a step wise increase with i steps; $\text{step}_{\text{dec}_i}$ refers to a step wise decrease with i steps; and reverse simply reverses (horizontally flips) the learning rate. The unary operations were adapted from my previous work, including only operations whose output are between zero and one, while also introducing three more types of operands: $\text{step}_{\text{inc}_i}$, $\text{step}_{\text{dec}_i}$, and reverse . The 13 proposed binary operations are listed Table 7.1, where $\text{cat}_i(x_1, x_2)$ is a concatenation of the first $i\%$ of x_1 and the last $(1 - i)\%$ of x_2 . Figure 7.1 depicts both decreasing and

increasing decay functions, and the restart, step wise decay (halves at every step), and concatenation operator.

Figure 7.4 depicts the representation for one of the final found fine-tuned learning rate schedules (F_1) in the search space. The two bottom rectangles refer to the *base learning rate schedules* that are used as operands to the *hidden state nodes* (blue nodes). The output node is shown in white. The gray rectangle refers to the final *scheduled decay function* that is multiplied to the argument of the output node to produce the final candidate learning rate schedule. Note, unused nodes in the graph are not depicted for brevity.

7.4.2 Genetic Algorithm

To search the proposed search space, I rely upon evolutionary computation due to its success in similar scenarios. Although *regularized evolution* has shown great success in searching for components of neural networks, especially on the original NASNet search space (Real et al., 2019), I decided to use a mutation-only, particle-based genetic algorithm due its simplicity and success in the preliminary runs. The simplicity of a mutation only algorithm allows for incremental local steps towards a local optimal solution.

Mutation in the proposed computational graph was performed by either randomly sampling from the remaining *scheduled decay functions* for the output node, or mutating one of the hidden state nodes (including output node). If a node was to be mutated, it was performed by either randomly sampling operations (either unary or binary), randomly sampling input connections from other nodes or *base learning rate schedules*, or changing an already existing *base learning rate schedule*. If a *base learning rate schedules* was to be mutated, it was performed by

$$lr = cd_{scheduled} * [cd_inc_1_r + step_{inc_3}(tri_inc_3_r)]$$

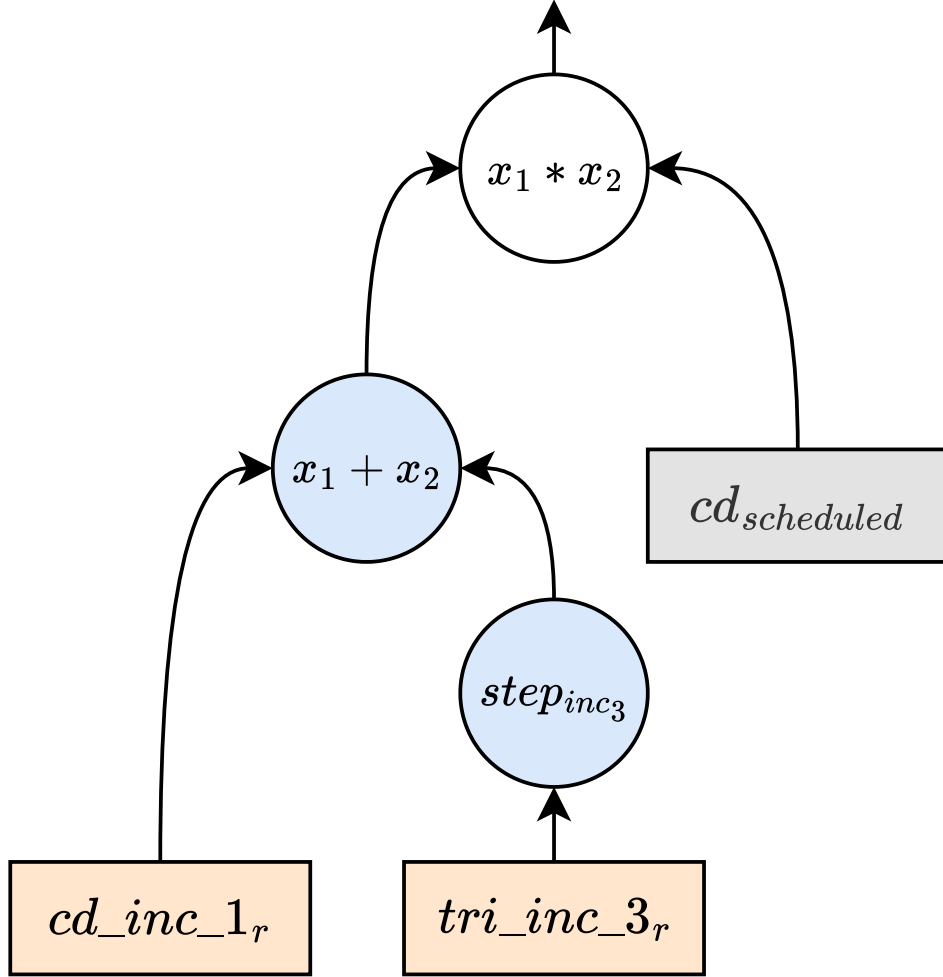


Figure 7.4: Exact implementation of F_1 learning rate schedule in the proposed computational graph.

either swapping from increasing to decreasing (or visa versa), randomly sampling from the remaining *decay functions*, or randomly sampling number of restarts.

Although the proposed search space allows for more diverse learning rate schedules, I found that mutation proved difficult for *phenotype* (actual learning rate schedule) exploration following standard mutation techniques in current NASNet derivative search spaces. Although the *genotype* (computational graph

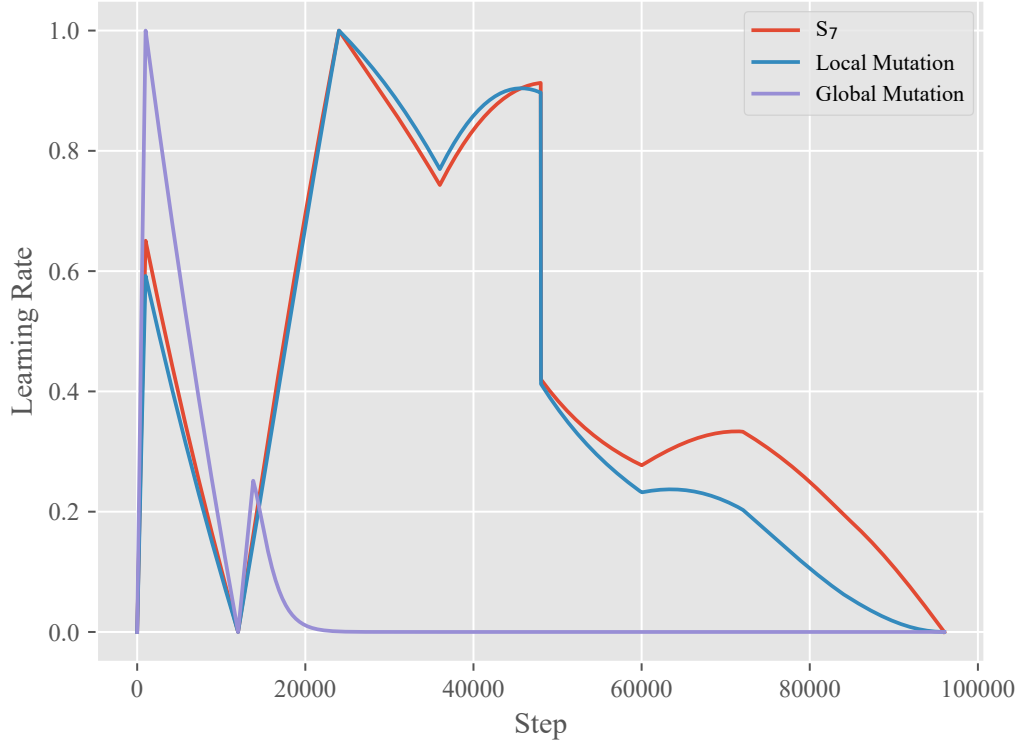


Figure 7.5: Phenotype depiction of S_7 along with two possible mutations, a local and global.

representation) of a schedule could be easily mutated through the procedure described above, the resulting phenotype can be extremely different. Figure 7.5 shows an example of S_7 along with two possible mutations. The first, called *local mutation*, simply swaps the scheduled cosine decay of S_7 with scheduled linear decay, allowing for a very similar, but different, learning rate schedule. The second on the other hand, called *global mutation*, yields a much different learning rate schedule than S_7 , but only simply swaps the scheduled cosine decay with scheduled exponential decay. One simple switch of the scheduled decay inside the genotype of the learning rate equation and the phenotype of the learning rate schedule becomes drastically different. I believe this is sub-optimal behavior as mutations should be performed in localized fashions, equating to incremental steps towards a

local optima. To handle this problem, I propose a phenotypical mutation system that only accepted mutations whose phenotype change was within a certain range of difference from the original. Specifically, I mutate each particle a total of 1,000 times, choosing only the closest k (using Euclidean distance - ignoring repeats) for evaluation.

7.4.3 Surrogate Function

Unlike neural architecture search, automated methods or heuristics cannot be applied to estimating the performance of a learning rate schedule due to its direct impact of training during all phases of training. As a result, all candidate learning rate schedules must be fully evaluated. Following tradition, all candidate learning rate schedules when training from scratch were evaluated on CIFAR-10 before being transferred to other datasets. For fine-tuning, no recognized dataset or protocol currently exists. I propose to evaluate all candidate learning rate schedules, when fine-tuning, on Cars, Caltech, and Flowers by aggregating their results in the hopes of finding dataset independent learning rate schedules. This was performed as preliminary results showed learning rates schedules became highly sensitive to specific datasets when fine-tuning.

To ensure ample exploration of the search space, I sampled 400 learning rate schedules, selecting the best $n = 8$ to be used as the initial particles. Each particle was ran with $k = 12$ mutations for $t = 3$ time-steps. When searching for training from scratch, all candidate learning rate schedules were trained using an ResNet18v2 (5.5M parameters), RandAug (Cubuk et al., 2020) image augmentation ($n = 2$, $m = 10$), Stochastic Depth (Huang et al., 2016) of 0.10, dropout of 0.5, Adam optimizer, maximum learning rate of 1e-3, and batch

size of 128 for 96,000 weight updates. The Adam optimizer was used due its inherent in-variance for tuning a maximum learning rate, unlike for SGD. When searching for fine-tuning, all candidate learning rate schedules were trained using EfficientNetV2B0, dropout of 0.5, RandAug image augmentation ($n = 2$, $m = 10$), SGD optimizer with momentum of 0.9, and batch size of 128 for 10,000 weight updates. The maximum learning rate for SGD was chosen by running each schedule on three possible maximum learning rates (0.1, 0.05, and 0.01) for each dataset, selecting the best for each.

7.5 Results and Discussion

	RESNET52v2		WIDERESNET		EFFNETV2B0
RATE	CIFAR10	CIFAR100	CIFAR10	CIFAR100	TINY
LINEAR	95.962±0.075	78.782±0.446	96.580±0.112	80.288±0.297	55.590±0.455
COSINE	95.798±0.150	78.833±0.322	96.630±0.103	80.458±0.267	55.154±0.594
DEMON	95.948±0.132	78.558±0.198	96.633±0.052	80.190±0.334	56.036 ±0.305
EXPON.	95.142±0.158	76.272±0.250	96.008±0.062	78.842±0.182	53.056±0.305
S ₁	96.027±0.153	79.540±0.263	96.805 ±0.084	80.820±0.321	55.658±0.353
S ₂	96.170±0.172	79.570±0.177	96.792 ±0.152	81.135 ±0.347	55.998 ±0.283
S ₃	96.215±0.081	79.820±0.185	96.730±0.099	80.867±0.305	56.146 ±0.355
S ₄	96.265 ±0.076	79.775±0.215	96.710±0.064	80.602±0.212	55.102±0.446
S ₅	96.258±0.068	79.942±0.101	96.788±0.099	80.530±0.215	55.582±0.466
S ₆	96.267 ±0.076	79.908±0.156	96.683±0.130	80.920 ±0.202	55.050±0.474
S ₇	96.185±0.108	79.978 ±0.315	96.752±0.146	80.683±0.312	55.178±0.170
S ₈	95.995±0.153	79.567±0.268	96.775±0.126	80.962 ±0.170	56.014 ±0.304
S ₉	96.135±0.099	79.780±0.376	96.810 ±0.087	80.870±0.321	55.696±0.393
S ₁₀	96.228±0.132	80.167 ±0.221	96.797 ±0.050	81.050 ±0.151	55.766±0.400
LR ₀	96.293 ±0.116	80.598 ±0.211	96.803 ±0.123	80.877±0.251	54.920±0.367
LR ₁	96.347 ±0.080	80.458 ±0.285	96.737±0.118	80.898±0.285	54.876±0.327
LR ₂	96.360 ±0.161	80.520 ±0.378	96.755±0.108	80.863±0.191	54.974±0.402
LR ₄	96.123±0.117	79.348±0.237	96.725±0.150	80.818±0.112	50.000±0.314
LR ₇	96.010±0.080	79.185±0.370	96.337±0.071	79.137±0.147	52.506±0.354
LR ₈	96.128±0.096	79.930±0.217	96.718±0.058	81.015 ±0.157	55.926 ±0.154

Table 7.2: Results for training from scratch (S_i is ours). Four standard learning rate schedules, along with the best found learning rate schedules from Chapter 6 (LR_i) are shown for comparison.

EFFICIENTNETV2SMALL

RATE	CARS	CALTECH	FLOWERS	DOGS	PETS
LINEAR	90.837±0.186	92.465 ±0.237	97.863 ±0.229	88.972±0.228	92.859±0.139
COSINE	90.730±0.260	92.426±0.141	97.814 ±0.188	88.932±0.149	92.679±0.313
DEMON	89.780±0.215	92.212±0.181	97.632±0.199	89.037±0.186	92.854±0.117
EXPON.	91.332 ±0.115	92.344±0.378	97.844 ±0.101	89.373±0.177	93.034±0.146
F ₁	90.658±0.054	92.202±0.200	97.814 ±0.119	90.347 ±0.175	93.099 ±0.270
F ₂	90.832±0.126	92.558 ±0.200	97.866 ±0.171	89.580±0.328	93.055 ±0.056
F ₃	90.936±0.190	92.535 ±0.155	97.756±0.181	90.014 ±0.190	93.039±0.085
F ₄	90.889±0.361	91.798±0.474	97.726±0.124	89.559±0.073	92.532±0.261
F ₅	90.917±0.157	92.465±0.138	97.723±0.080	89.466±0.165	92.963±0.089
F ₆	90.991 ±0.123	92.255±0.201	97.746±0.132	88.832±0.070	92.799±0.184
F ₇	90.889±0.118	92.600 ±0.162	97.749±0.091	88.865±0.217	92.788±0.136
F ₈	91.103 ±0.246	92.594 ±0.195	97.642±0.134	89.145±0.086	92.837±0.262
F ₉	90.969 ±0.128	92.390±0.157	97.779±0.135	88.937±0.141	92.952±0.278
F ₁₀	90.805±0.058	92.364±0.114	97.710±0.173	89.828±0.173	93.104 ±0.134
LR ₀	90.061±0.103	92.419±0.183	97.681±0.114	90.464 ±0.167	92.701±0.355
LR ₁	89.882±0.257	92.170±0.481	97.795±0.184	90.562 ±0.126	93.306 ±0.197
LR ₂	90.086±0.215	92.360±0.201	97.707±0.313	90.207 ±0.172	93.061 ±0.249
LR ₄	90.412±0.242	92.308±0.291	97.580±0.182	89.713±0.164	93.006±0.120
LR ₇	91.043 ±0.113	92.032±0.252	97.795±0.083	89.170±0.146	92.979±0.118
LR ₈	90.320±0.246	92.304±0.231	97.739±0.092	89.897±0.151	93.017±0.092

Table 7.3: Results for fine-tuning (F_i is ours). Four standard learning rate schedules, along with the best found learning rate schedules from Chapter 6 (LR_i) are shown for comparison.

7.5.1 Final Learning Rates

The best 10 found learning rate schedules when training from scratch (referred to as S_i) and fine-tuning (referred to as F_i) are reported in Figures 7.2 and 7.3. The exact learning rate equations can be found in Section 7.7. When training from scratch, there are two major observations that can be made. First, $S_{1,2,3,8,9,10}$ greatly delay the learning rate warm-up period, peaking around 20% into training. Second, $S_{4,5,6,7}$ exhibit extremely unique behavior with very sharp inclines and declines. When fine-tuning, two major observations can be made. First, $F_{1,2,3,4,5,8,9,10}$ exhibit many full or partial restarts, possibly indicating that learning rates with restarts can be beneficial for fine-tuning. Second, F_1 is extremely similar to S_7 , indicating that this type of learning rate can perform extremely well in both environments.

FROM SCRATCH S_i				FINE-TUNE F_i		
RATE	AVG. AC.	MN. R.	MD. R.	AVG. AC.	MN. R.	MD. R.
LINEAR	81.440	15.8	17.0	92.599	9.20	10.0
COSINE	81.375	16.2	17.0	92.516	12.2	13.0
DEMON	81.473	14.6	18.0	92.303	16.8	16.0
EXPON.	79.864	19.6	20.0	92.785	7.00	7.00
F/S ₁	81.770	9.80	11.0	92.824	8.40	5.00
F/S ₂	81.933	6.40	5.00	92.778	5.80	5.00
F/S ₃	81.956	7.60	9.00	92.778	5.80	5.00
F/S ₄	81.691	11.4	13.0	92.501	14.4	13.0
F/S ₅	81.820	8.60	6.00	92.707	9.80	11.0
F/S ₆	81.766	9.20	8.0	92.525	13.2	15.0
F/S ₇	81.755	9.40	9.00	92.578	11.0	10.0
F/S ₈	81.863	8.60	7.00	92.664	10.2	14.0
F/S ₉	81.858	7.40	8.00	92.605	10.2	9.00
F/S ₁₀	82.002	4.60	4.00	92.762	9.20	10.0
LR ₀	81.898	6.00	3.00	92.665	12.6	17.0
LR ₁	81.863	7.60	6.00	92.743	9.20	7.00
LR ₂	81.894	7.20	8.00	92.684	10.4	11.0
LR ₄	80.603	14.4	13.0	92.604	13.0	13.0
LR ₇	80.635	17.6	19.0	92.604	10.2	10.0
LR ₈	81.943	8.00	7.00	92.655	11.2	12.0

Table 7.4: Rank results for training from scratch (S_i is ours) and fine-tuning (F_i is ours). Four standard learning rate schedules, along with the best found learning rate schedules from Chapter 6 (LR_i) are shown for comparison.

7.5.2 Transferability Experiments

To assess generalization for training from scratch, all learning rate schedules were evaluated on CIFAR-10 and CIFAR-100 using ResNet52v2 (15.7M Params) and WideResNet (36.5M Params), dropout of 0.5, Stochastic Depth of 0.2, RandAug image augmentation ($n = 2$, $m = 10$), Adam optimizer, maximum learning rate of 1e-3, and batch size of 128 for 96,000 steps. In addition, all learning rate schedules when training from scratch were also evaluated in TinyImageNet using EfficientNetV2B0 (5.3M Params), dropout of 0.5, RandAug image augmentation ($n = 2$, $m = 10$), Adam optimizer, maximum learning rate of 1e-3, and batch size of 256 for 95,795 steps.

To assess generalization for fine-tuning, all learning rate schedules were evaluated on Cars, Caltech, Flowers, Dogs, and Pets using EfficientNetV2Small (20.6M

Params), dropout of 0.5, RandAug image augmentation ($n = 2$, $m = 15$), SGD optimizer with momentum of 0.9, and batch size of 128 for 10,000 weight updates. The maximum learning rate was chosen by running each schedule on five possible maximum learning rates (0.1, 0.075, 0.05, 0.025, and 0.01) for each dataset, selecting the best for each.

The results for training from scratch and fine-tuning can be found in Tables 7.2 and 7.3. For comparison, the results from four standardly used learning rate schedules are shown (Linear, Cosine, Demon, and Exponential). In addition, the best found learning rate schedules found in my previous work were also evaluated for comparison (referred to as LR_i , as denoted in original chapter). The mean and standard deviation across 5 independent runs are recorded for each experiment. Bold indicates Top 5 performance for that particular experiment. As one can see from Table 7.2, only once did a standard learning rate achieve Top 5 performance on a task. To assess overall generalization across datasets and architectures, Table 7.4 records the average mean test accuracy, and mean and median rank for both training from scratch and fine-tuning. Rank refers to sorted order in performance for a particular experiment (lower is better). As one can see, S_{10} achieved the highest average mean test accuracy and lowest mean rank. However, LR_0 did achieve the lowest median rank. The S_{10} learning rate schedule has four main characteristics. First, it greatly slows down reaching the maximum warm-up period until 29% into the schedule; second, the warmup period is not linear, but exhibits Gaussian like structure; third, it eliminates holding the learning rate when the maximum learning rate is reached; and lastly, it exhibits a cosine like decay. These characteristics could reveal letting the learning rate take longer to warmup using a non-linear function can improve performance.

From Table 7.3 for fine-tuning, the results are not as dominate than when

training from scratch. Multiple standard learning rate schedules achieved Top 5 performance across multiple datasets. In fact, Table 7.4 shows that the exponential decay schedule achieves Top 5 performance for fine-tuning for average mean test accuracy, and mean and median rank. Table 7.4 also reveals that F_1 achieved the highest average mean test accuracy across the datasets while being tied for first in terms of median rank. Both F_2 and F_3 tie for third place for average mean test accuracy, and first for mean and median rank. Both F_2 and F_3 are extremely similar, learning four restarts with decreasing peaks. This could possibly indicate that learning rates with restarts may perform well when fine-tuning. Arguably, F_1 is the most drastic looking learning rate, with multiple inclines and declines during the early phases of training before smoothing out for later stages. Despite these unorthodox mechanisms, F_1 surprisingly tied for best median rank and achieves first for highest average mean test accuracy. A sibling to F_1 , S_7 , did not achieve top performance when training from scratch, but outperformed all standard learning rate schedules in terms of better average mean test accuracy, and mean and median rank.

7.6 Scheduled Decay Functions

All *scheduled decay functions* when training from scratch warmed up the learning rate from 0 to 0.001 over the course of 1,000 steps, being held at its maximum for 12,800 steps, and was then decayed back down for the rest of the 82,200 steps (totalling to 96,000 steps). All *scheduled decay functions* when fine-tuning warmed up the learning rate from 0 to its respective maximum over the course of 500 steps, being held at its maximum for 500 steps, and was then decayed back down for the rest of the 9,000 steps (totalling to 10,000 steps). Figure 7.6 depicts the

components of the F_2 learning rate and how they are combined to form the final learning rate schedule. Note, the results from all operations in the computational graph for all learning rate schedules are rescaled between 0 and 1. Only the final learning rate schedule is multiplied by the maximum learning rate to obtain the end result.

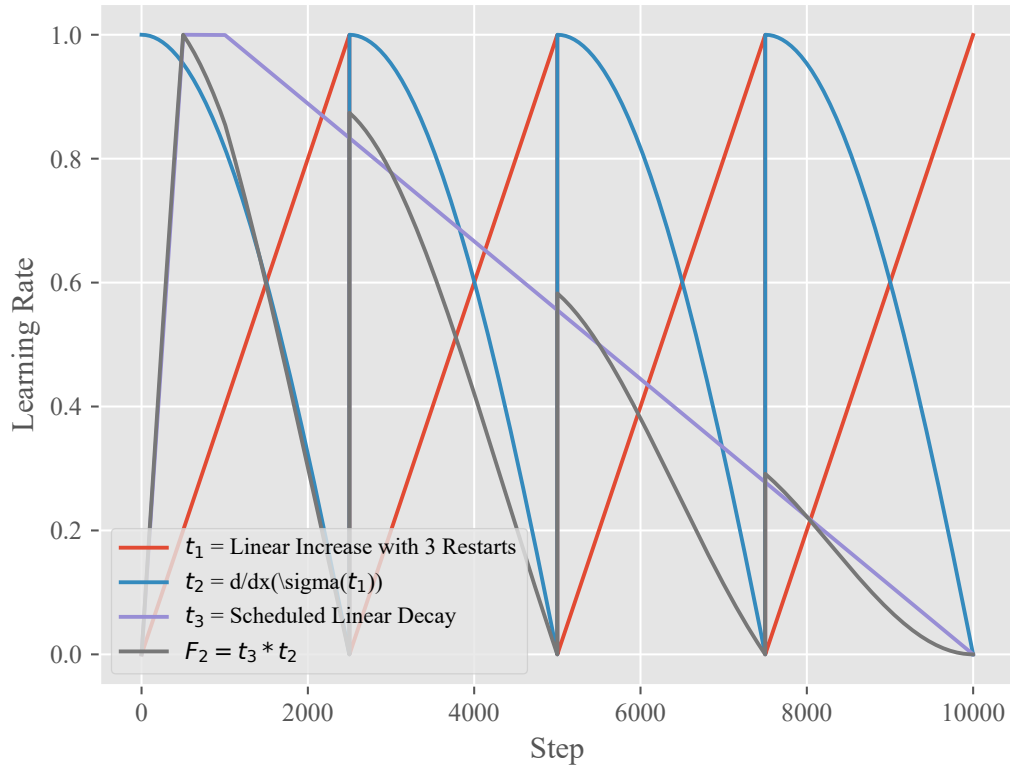


Figure 7.6: Components of F_2 . Note that all operations are rescaled between 0 and 1.

7.7 Final Learning Rate Equations

The final 10 learning rate equations for both training from scratch and fine-tuning are given Table 7.6. The x_y_z template formatting refers to scheduled decay function with a x decay function, being either increasing ($y = inc$) or decreasing

($y = dec$), along with z number of restarts. For example, the $ld_{scheduled}$ for S_1 means that the scheduled decay function was selected to be linear decay. The $ed_{inc_0_r}$ refers to a base learning rate schedule argument that exponentially increases with 0 restarts.

Decay Schedules	
ld_{dec}	$1 - \text{mod}(r * t, T)/T$
ld_{inc}	$\text{mod}(r * t, T)/T$
cd_{dec}	$0.5(1 + \cos(\pi \text{mod}(r * t, T)/T))$
cd_{inc}	$1 - 0.5(1 + \cos(\pi \text{mod}(r * t, T)/T))$
cc_{dec}	$0.5(1 + \cos(2t\pi/T))$
cc_{inc}	$1 - 0.5(1 + \cos(2t\pi/T))$
ed_{dec}	$base^{\text{mod}(r*t, T)}$
ed_{inc}	$(1 - base)^{\text{mod}(r*t, T)}$
dd_{dec}	$\frac{\text{mod}(0.95*(1-r*t/T))}{(0.05+0.95*(1-t/T))}$
dd_{inc}	$0.95 - \frac{\text{mod}(0.95*(1-r*t/T))}{(0.05+0.95*(1-t/T))}$

Table 7.5: The proposed set of scheduled decay functions. Where r controls the number of restarts, $base$ was set to 0.9999 for training from scratch and 0.99925 for fine-tuning, t is the current time-step, and T is the total number of timesteps.

7.8 Conclusions

In this chapter, I expanded upon previous research in automated methods for discovering learning rates by specifically focusing on *neural learning rate search* for both training from scratch and fine-tuning. I proposed a new search space that allowed for many diverse and novel learning rate schedules. After searching the search space using a mutation-only, particle-based genetic algorithm, 10 learning rate schedules for both environments were reported. In the end, S_{10} and $F_{1,2,3}$ performed the best when training from scratch or fine-tuning, in terms

NAME	EQUATION
S_1	$ld_{scheduled} * [(ed_inc_0_r) \arctan(tri_dec_2_r)]$
S_2	$ld_{scheduled} * [(ed_inc_0_r) \arctan(step_inc_2(tri_dec_2_r))]$
S_3	$ld_{scheduled} * [cat_{0.50}(ed_inc_0_r, \arctan(tri_dec_2_r))]$
S_4	$ld_{scheduled} * [reverse(\arctan(dd_dec_1_r))]$
S_5	$ld_{scheduled} * [reverse(\arctan(dd_dec_2_r))]$
S_6	$cd_{scheduled} * [reverse(\arctan(dd_dec_2_r))]$
S_7	$ld_{scheduled} * [cd_inc_1_r + step_inc_4(tri_inc_3_r)]$
S_8	$cd_{scheduled} * [\frac{d}{dx} \tanh(ed_dec_0_r)]$
S_9	$cd_{scheduled} * [step_inc_4(ed_dec_0_r)]$
S_{10}	$cd_{scheduled} * [\frac{d}{dx} \text{softsign}(ed_dec_0_r)]$
F_1	$cd_{scheduled} * [cd_inc_1_r + step_inc_2(tri_inc_3_r)]$
F_2	$ld_{scheduled} * [\frac{d}{dx} \sigma(ld_inc_3_r)]$
F_3	$cd_{scheduled} * [reverse(ld_inc_3_r)]$
F_4	$cd_{scheduled} * [mix_{0.5}(cc_inc_2_r, cc_inc_0_r)]$
F_5	$cd_{scheduled} * [mix_{0.5}(cc_inc_2_r, (ed_inc_0_r)^2)]$
F_6	$cd_{scheduled} * [\frac{d}{dx} \sigma(\frac{d}{dx} \text{softplus}(cd_dec_0_r))]$
F_7	$cd_{scheduled} * [\text{erfc}(\frac{d}{dx} \text{softsign}(dd_dec_0_r))]$
F_8	$cd_{scheduled} * [\frac{d}{dx} \tanh(\sigma(cd_inc_0_r^{(tri_dec_1_r)}))]$
F_9	$cd_{scheduled} * [\frac{d}{dx} \tanh(step_dec_2(cd_inc_0_r^{(tri_dec_1_r)}))]$
F_{10}	$cd_{scheduled} * [\frac{d}{dx} \sigma(dd_dec_2_r) + step_inc_3(cc_inc_1_r^2)]$

Table 7.6: Genotype equations for the final 10 learning rate schedules for both training from scratch and fine-tuning .

of either yielding the best average mean test accuracy, or mean and median rank across the environment respective datasets and architectures. My code can be found at the following repository: <https://github.com/OUStudent/EvolvingLearningRateSchedules/>.

Chapter 8

Neural Output Function Search

8.1 Overview

Recently, success from neural loss function search (Liu et al. (2021); Gu et al. (2022); Gonzalez and Miikkulainen (2020), including my previous works), the search for finding a replacement of cross-entropy for classification, has yielded very promising results. From the success of automated machine learning being applied to many different facets of neural networks (Real et al., 2019; Bello et al., 2017a; Ramachandran et al., 2017), I propose to search for a parametric output activation function layer that could simply be used as a drop-in replacement for Softmax.

8.2 Contribution

In this work I use a search space designed for parametric output activation functions, based off the NASNet search space for architectures (Zoph et al., 2018). The search space was then explored using my mutation-only, particle-based genetic

algorithm. The candidate parametric output activation functions were evaluated on the CIFAR-10 dataset (Krizhevsky and Hinton, 2009) using a ResNet52v2 (23.5M parameters) (He et al., 2016) for a limited number of epochs. The best were then evaluated using ResNet52v2 and EfficientNetV2B0 (5.3M parameters) (Tan and Le, 2021) on CIFAR-10, CIFAR-100, and TinyImageNet (Le and Yang, 2015). From the success of NLFS and NAFS, I perform and propose the first ever, to the knowledge, *neural output activation function search* (NOAFS), with the goal of finding a drop-in replacement for the Softmax activation in deep learning for image classification.

8.3 Methodology

8.3.1 Search Space

In this work, each output activation function was represented by an activation function. In order to create a probability distribution for all output activation functions, each output per class was normalized by the sum of the outputs for all other class nodes for the activation. This can be seen in Equation 8.1, where x_i is the output of the i^{th} node, and A is the output activation function.

$$Output(x_i) = \frac{A(x_i)}{\sum_j A(x_j)} \quad (8.1)$$

The same search space discussed in Chapter 4 was used in this work, adapted for activation functions. Each activation was constructed by a computational graph containing three integral parts. First, each hidden state node receives an input, either from other hidden state nodes or the input to the layer (x). Second, each hidden state node performs either a unary or binary operation. Lastly, a

root node is used to denote the final operation before the activation values are returned. All nodes not connected to the root node, either directly or indirectly, are unused and not used in the final computation. This allows for the dynamic increase or decrease in complexity of the activation.

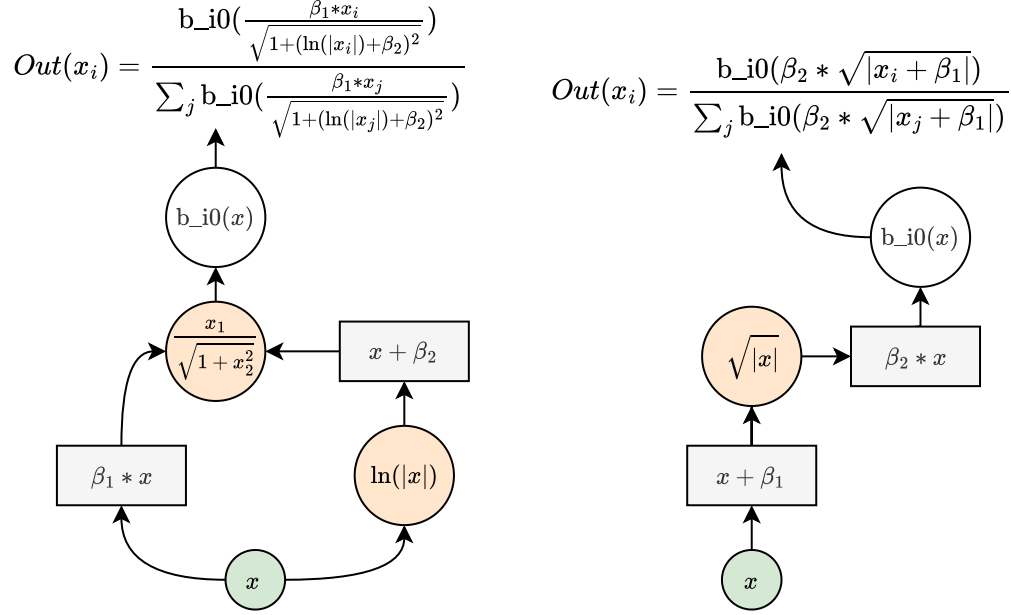


Figure 8.1: Exact graph representation for (Left) A_2 and (Right) A_7 . Note, all unused nodes are not shown for brevity.

To allow for parametric activation functions, trainable coefficients were randomly assigned, upon initialization, to the connection paths between hidden state nodes to either adjust the slope ($\beta * x_{input}$) or intercept ($x_{input} + \beta$). Parametric coefficients for activation functions are usually applied at three different levels: neuron, channel, and layer. Assuming an input, x , with shape $(32, 32, 3)$, then neuron level parametric variables would have shape $(32, 32, 3)$ as well; channel level parametric variables would have shape $(3,)$, being applied only along the channel axis; and layer level parametric variables would have shape $(1,)$. In order

to keep the output activation shape the same for all neurons, and limit the number of added parameters, the trainable coefficients were only applied at the layer level. All slope coefficients were initialized to one during training, while all intercept coefficients were initialized to zero. Figure 8.1 shows the computational graph representation for two of the final discovered activation functions, A_2 and A_7 using the described search space.

8.3.2 Genetic Algorithm

From my previous success, I chose to use my mutation-only, particle-based genetic algorithm to search over the search space. The pseudocode for the algorithm can be found in Algorithm 1. After randomly sampling 400 random candidate output activation functions, the best 24 were taken to be used as initial particles. Each particle ran for $t = 3$ timesteps, performing $k = 8$ mutations at each timestep.

Mutation on the computation graph was performed by randomly selected a node, either a root or hidden state node, and mutation either its connection, operation, or trainable coefficient. A connection was mutated by randomly sampling from the list of remaining possible connections. An operation was mutated by randomly sampling from the list of remaining possible operations. A trainable coefficient was mutated by either randomly adding a slope or intercept, if it was not present, or randomly removing a slope or intercept, if it was present.

8.3.3 Surrogate Function

Like previous works in searching for components of neural networks for image classification, I evaluate all candidate output activation functions on the CIFAR-10 dataset for a limited number of epochs. All initial random candidate output activa-

tion functions were trained on a simple three layer ConvNet. This was performed to save computation. When running the particle-based genetic algorithm, each particle was trained using a ResNet52v2 (23.5M parameters), RandAug (Cubuk et al., 2020) image augmentation ($n = 2$, $m = 10$), Stochastic Depth (Huang et al., 2016) of 0.20, dropout of 0.5 before the final dense layer, Adam (Kingma and Ba, 2014) optimizer, maximum learning rate of 1e-3, and a batch size of 128 for 41 epochs (totaling to 16,031 steps). A one cycle cosine decay (Smith, 2018a) learning rate schedule was also used for training, using three epochs to warm-up the learning rate.

8.4 Results and Discussion

8.4.1 Final Output Activation Functions

After running the particle based genetic algorithm, the final seven found output activation functions are shown in the top half of Table 8.1, where *Soft* refers to Softmax, *Soft_p* refers to parametric Softmax ($\exp(\beta_1 * x + \beta_2)$), and A_i refers to the i^{th} discovered output activation. From the table, four major observations can be made. First, A_1 and A_3 can be seen as mutations of Softmax and Sigmoid. Second, A_{1-3} , use the common expression, $\beta_1 * x / \sqrt{1 + (\ln(|x| + \beta_2))^2}$. Third, A_{4-7} all use the base function of $\text{bessel}_{i0}(\sqrt{|x|})$, while only changing the number and placement of the parametric parameters. Lastly, all discovered output activation functions included trainable parameters.

Due to the majority of the discovered output activation functions being based around the $\text{bessel}_{i0}(\sqrt{|x|})$ function, five output activation (B_1 through B_5) functions were constructed to study their effects, also shown in Table 8.1. Lastly,

Reference	Output Activation
A_1	$\sigma(\frac{\beta_1 * x}{\sqrt{1+(\ln(x)+\beta_2)^2}} + \beta_3)$
A_2	$\text{bessel}_{i0}(\frac{\beta_1 * x}{\sqrt{1+(\ln(x)+\beta_2)^2}})$
A_3	$\exp(\frac{\beta_1 * x}{\sqrt{1+(\ln(x)+\beta_2)^2}})$
A_4	$\text{bessel}_{i0}(\sqrt{ \beta_1 * x } + \beta_2)$
A_5	$\text{bessel}_{i0}(\beta_1 * \sqrt{ x } + \beta_2)$
A_6	$\text{bessel}_{i0}(\beta_1 * \sqrt{ \beta_2 * x + \beta_3 } + \beta_4)$
A_7	$\text{bessel}_{i0}(\beta_1 * \sqrt{ x + \beta_2 })$
B_1	$ \beta_1 * x + \beta_2 $
B_2	$\sqrt{ \beta_1 * x + \beta_2 }$
B_3	$\text{bessel}_{i0}(\beta_1 * x + \beta_2)$
B_4	$\text{bessel}_{i0}(x)$
B_5	$\text{bessel}_{i0}(\sqrt{ x })$
B_6	$\text{bessel}_{i0}(\frac{x}{\sqrt{1+(\ln(x)+\beta_1)^2}})$
B_7	$\text{bessel}_{i0}(\frac{\beta_1 * x}{\sqrt{1+(\ln(x))^2}})$
B_8	$\text{bessel}_{i0}(\frac{x}{\sqrt{1+(\ln(x))^2}})$

Table 8.1: The seven final discovered output activation function equations (A_i), along with eight constructed functions (B_i).

three other activation functions (B_6 through B_8) were constructed to study the effects of the different parametric variables for A_2 .

8.4.2 CIFAR and TinyImageNet

To assess generalization, all discovered output activation functions were transferred to CIFAR-10, CIFAR-100, and TinyImageNet using a ResNet52v2 and EfficientNetB0v2 architectures. The training for ResNet52v2 across all datasets was the same as the surrogate function, except for 246 epochs. For EfficientNetB0v2, the training schedule was the same as ResNet52v2, except Stochastic Depth was not used. For all experiments, each output activation function was evaluated across 10 independent runs. Table 8.2 and Table 8.3 shows the mean, standard

ResNet52v2						
CIFAR-10			CIFAR-100		TINYIMAGENET	
OUTPUT	MEAN $\pm$ STD.	MEDIAN	MEAN $\pm$ STD	MEDIAN	MEAN $\pm$ STD	MEDIAN
<i>Soft</i>	95.978 $\pm$ 0.099	95.945	78.825 $\pm$ 0.419	78.820	64.059 $\pm$ 0.310	64.030
<i>Soft_p</i>	33.778 $\pm$ 32.32	24.600	16.952 $\pm$ 31.90	01.000	00.500 $\pm$ 0.000	00.500
<i>A</i> ₁	96.561 $\pm$ 0.162	96.540	49.815 $\pm$ 37.99	80.555	00.735 $\pm$ 0.118	00.785
<i>A</i> ₂	96.655 $\pm$ 0.123	96.665	81.518 $\pm$ 0.162	81.490	64.549 $\pm$ 0.504	64.585
<i>A</i> ₃	96.693 $\pm$ 0.071	96.710	81.589 $\pm$ 0.228	81.590	58.289 $\pm$ 19.81	65.990
<i>A</i> ₄	96.587 $\pm$ 0.147	96.580	81.465 $\pm$ 0.199	81.400	00.500 $\pm$ 0.000	00.500
<i>A</i> ₅	96.579 $\pm$ 0.087	96.580	81.225 $\pm$ 0.274	81.190	66.425 $\pm$ 0.370	66.525
<i>A</i> ₆	96.543 $\pm$ 0.099	96.535	81.187 $\pm$ 0.215	81.225	65.736 $\pm$ 1.495	66.220
<i>A</i> ₇	96.650 $\pm$ 0.103	96.625	81.247 $\pm$ 0.259	81.235	66.501 $\pm$ 0.205	66.530
<i>B</i> ₁	18.446 $\pm$ 1.903	03.280	17.904 $\pm$ 0.608	03.280	00.786 $\pm$ 0.162	00.720
<i>B</i> ₂	20.340 $\pm$ 1.372	19.780	02.136 $\pm$ 0.335	02.060	00.754 $\pm$ 0.082	00.690
<i>B</i> ₃	96.502 $\pm$ 0.146	96.440	80.715 $\pm$ 0.428	80.655	63.048 $\pm$ 0.393	63.040
<i>B</i> ₄	76.514 $\pm$ 27.07	88.040	47.082 $\pm$ 11.57	45.660	21.886 $\pm$ 2.793	23.510
<i>B</i> ₅	93.156 $\pm$ 0.172	93.205	81.496 $\pm$ 0.271	81.390	66.608 $\pm$ 0.190	66.610
<i>B</i> ₆	96.679 $\pm$ 0.129	96.665	81.453 $\pm$ 0.204	81.520	64.463 $\pm$ 0.386	64.460
<i>B</i> ₇	96.484 $\pm$ 0.103	96.475	79.927 $\pm$ 1.823	80.880	63.568 $\pm$ 0.362	63.625
<i>B</i> ₈	91.894 $\pm$ 1.997	92.075	76.267 $\pm$ 5.960	79.075	60.838 $\pm$ 3.720	62.635

Table 8.2: Results for ResNet52v2 on the CIFAR and TinyImageNet datasets. The Top 3 best mean test accuracies for each dataset is highlighted in bold.

EfficientNetB0v2						
CIFAR-10			CIFAR-100		TINYIMAGENET	
OUTPUT	MEAN $\pm$ STD.	MEDIAN	MEAN $\pm$ STD	MEDIAN	MEAN $\pm$ STD	MEDIAN
<i>Soft</i>	84.869 $\pm$ 2.697	83.775	58.875 $\pm$ 1.710	59.045	55.020 $\pm$ 0.618	55.065
<i>Soft_p</i>	10.000 $\pm$ 0.000	10.000	62.514 $\pm$ 0.784	62.240	55.450 $\pm$ 0.551	55.730
<i>A</i> ₁	88.540 $\pm$ 0.913	88.510	62.897 $\pm$ 0.543	62.840	53.411 $\pm$ 0.607	53.590
<i>A</i> ₂	90.286 $\pm$ 0.375	90.230	63.064 $\pm$ 0.610	63.195	55.393 $\pm$ 0.440	55.375
<i>A</i> ₃	90.181 $\pm$ 0.188	90.180	62.889 $\pm$ 0.725	62.815	34.332 $\pm$ 20.50	35.145
<i>A</i> ₄	88.452 $\pm$ 1.830	88.730	60.100 $\pm$ 3.083	60.875	00.500 $\pm$ 0.000	00.500
<i>A</i> ₅	89.525 $\pm$ 0.730	89.895	60.806 $\pm$ 2.239	61.370	55.174 $\pm$ 0.459	55.085
<i>A</i> ₆	84.370 $\pm$ 9.424	89.055	53.639 $\pm$ 18.46	61.605	55.365 $\pm$ 0.575	55.470
<i>A</i> ₇	87.292 $\pm$ 4.337	89.280	62.624 $\pm$ 1.005	62.600	55.706 $\pm$ 0.294	55.730
<i>B</i> ₁	15.698 $\pm$ 1.568	15.330	01.977 $\pm$ 0.168	02.105	00.654 $\pm$ 0.045	00.680
<i>B</i> ₂	16.144 $\pm$ 1.039	16.000	01.884 $\pm$ 0.119	01.920	00.660 $\pm$ 0.075	00.630
<i>B</i> ₃	10.000 $\pm$ 0.000	10.000	62.785 $\pm$ 0.502	62.775	55.465 $\pm$ 0.411	55.370
<i>B</i> ₄	10.000 $\pm$ 0.000	10.000	59.604 $\pm$ 1.797	60.190	54.836 $\pm$ 0.343	54.730
<i>B</i> ₅	87.579 $\pm$ 3.165	88.935	63.020 $\pm$ 0.725	62.900	55.593 $\pm$ 0.553	55.625
<i>B</i> ₆	90.014 $\pm$ 0.289	90.030	63.102 $\pm$ 0.490	63.070	55.687 $\pm$ 0.457	55.675
<i>B</i> ₇	90.122 $\pm$ 0.388	90.215	62.423 $\pm$ 0.811	62.345	55.489 $\pm$ 0.370	55.585
<i>B</i> ₈	89.759 $\pm$ 0.295	89.720	62.912 $\pm$ 0.534	62.705	55.530 $\pm$ 0.224	55.575

Table 8.3: Results for EfficientNetB0v2 on the CIFAR and TinyImageNet datasets. The Top 3 best mean test accuracies for each dataset is highlighted in bold.

deviation, and median values for all output activation functions for ResNet52v2 and EfficientNetB0v2 across CIFAR and TinyImageNet. Bold indicates Top 3

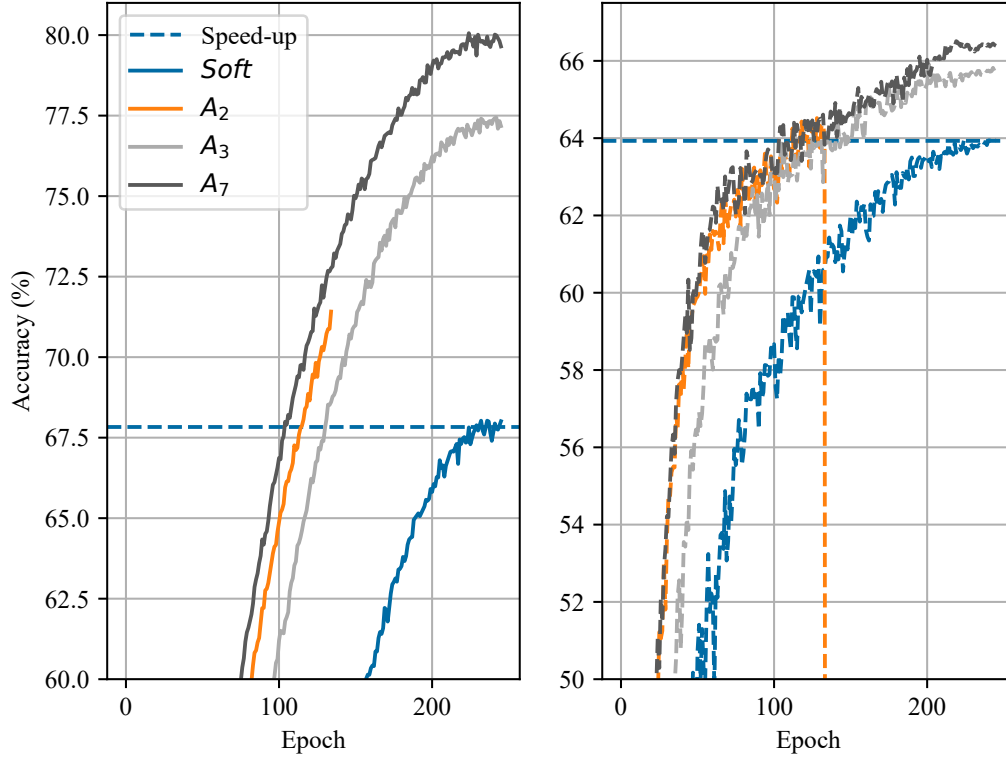


Figure 8.2: Training (Left) and test (Right) accuracy for ResNet52v2 on CIFAR-10 for Softmax and $A_{2,3,7}$.

performance for that particular column.

From Table 8.2, six major observations can be made. First, never once did either Softmax nor parametric Softmax achieve Top 3 performance. Second, parametric Softmax performed extremely poorly, revealing the results are not due to parametric parameters alone. Third, despite A_3 being a derivative of Softmax, its results were much more stable than parametric Softmax while also achieving Top 3 performance across two of the datasets for best mean and median values. This indicates that scaling the input x by $\beta_1 * x / \sqrt{1 + (\ln(|x| + \text{beta}_2))^2}$ before being fed into exp can lead to more stable results. Fourth, top performance was consistently achieved across multiple datasets for A_2 , A_3 , and A_7 , two of which

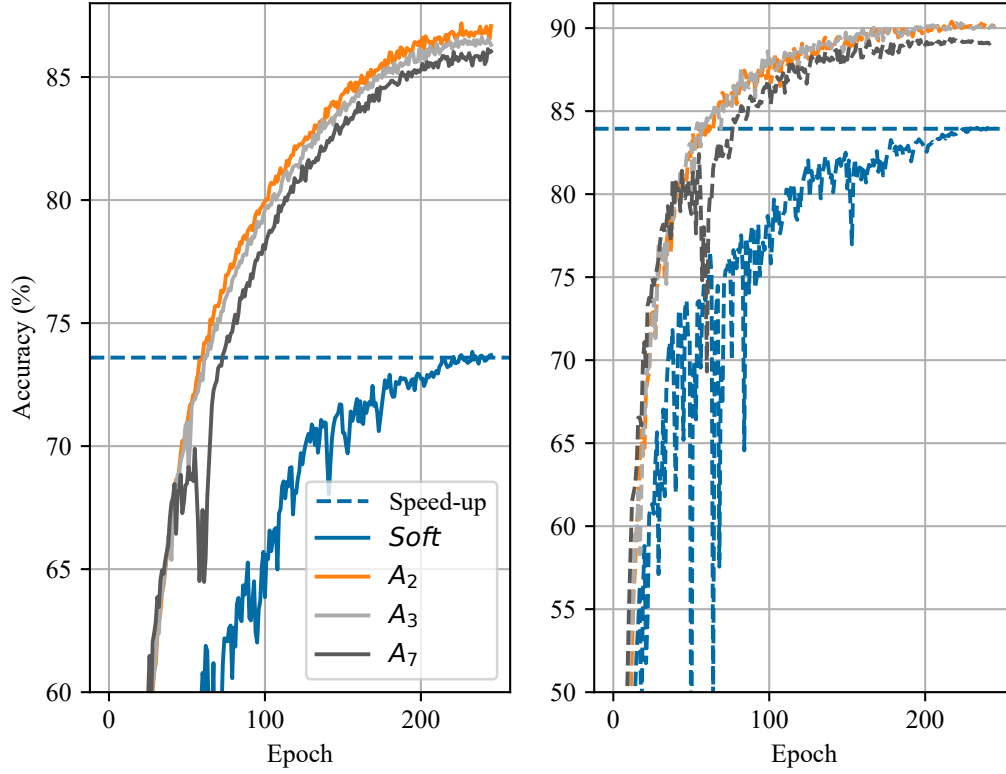


Figure 8.3: Training (Left) and test (Right) accuracy for EfficientNetB0v2 on CIFAR-10 for Softmax and $A_{2,3,7}$.

use the bessel_{i0} function, and two of which use the $\beta_1 * x / \sqrt{1 + (\ln(|x| + \text{beta}_2))^2}$ operator. Fifth, both B_1 and B_2 fail to learn, indicating that the trainable absolute value and square root of the absolute values are not enough to contribute to the success of A_{4-7} . In addition, B_4 , being only the bessel_{i0} function, also fails to learn, indicating that it is not enough to learn by its own. However, with the inclusion of parametric values, B_3 does improve learning for the CIFAR datasets. Despite this deficiency, B_5 matches and exceeds performance on CIFAR-100 and TinyImageNet compared to A_7 . This indicates that the bessel_{i0} function alone, or with parametric values, does not contribute to its success, nor does the absolute

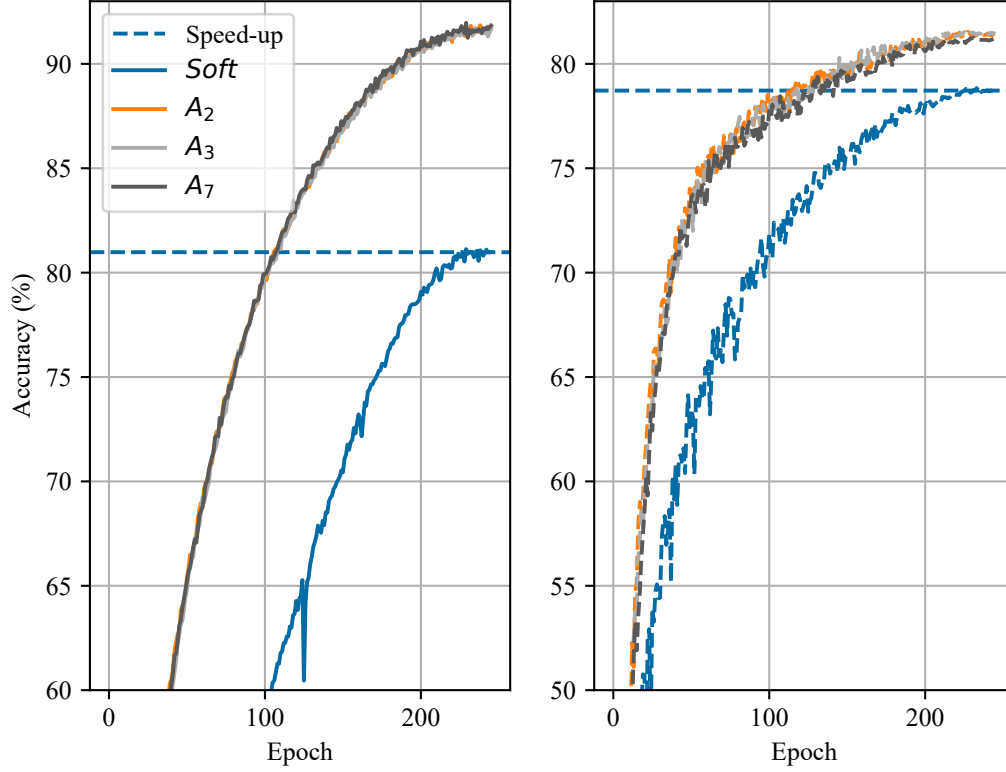


Figure 8.4: Training (Left) and test (Right) accuracy for ResNet52v2 on CIFAR-100 for Softmax and $A_{2,3,7}$.

or square root values, but their unique combination. Lastly, B_8 under-performs Softmax across all datasets, while both B_6 and B_7 outperform on, except for B_7 on TinyImageNet. This indicates that the placement of either parametric value in the expression $\beta_1 * x / \sqrt{1 + (\ln(|x| + \text{beta}_2)^2)}$ is vital for its success, where $\beta_1 * x$ appears to be the most important.

From Table 8.3, four major observations can be made. First, despite poor performance of the parametric Softmax on ResNet52v2, the results are much better on EfficientNetB0v2, as Soft_p outperformed Softmax only on CIFAR-100 and TinyImageNet. However, Soft_p failed to learn on CIFAR-10. Second, A_2

and A_3 perform extremely well on CIFAR-10 and CIFAR-100, but A_3 struggles on TinyImageNet for both architectures while A_2 does not. It appears the bessel_{i0} function solves this issue as a replacement of the exponential. Third, the non-parametric B_5 performs well again across all datasets, taking two Top 3 performances. Lastly, B_8 once again under-performs both B_6 and B_7 , while both B_6 and B_7 outperform Softmax, indicating that the inclusion for one of the parameters is vital for performance.

Lastly, Figure 8.2 and 8.4 show the training and test accuracies for the median runs of $A_{2,3,7}$ and Softmax on CIFAR-10 and CIFAR-100 using the ResNet52v2 architectures. As one can see, the speedup, the time it takes for $A_{2,3,7}$ to reach Softmax, is almost half. Figure 8.3 also shows the training and test accuracies for the median runs of $A_{2,3,7}$ and Softmax on CIFAR-10 using the EfficientNetB0v2 architecture. As one can see, the speedup for $A_{2,3,7}$ to reach Softmax is actually less than half. As A_2 reaches the test accuracy of Softmax by epoch 55 out of 246, a 77.6% speedup.

Due to the parametric nature of A_2 , A_3 , and A_7 , the exact output distribution for each node can be plotted per epoch. This is shown in Chapter A.

8.4.3 Bessel Functions

Although a few discovered output activation functions did not include a Bessel function (A_1 and A_3), they did not outperform Softmax across all datasets and architectures. In fact, only $A_{2,5,7}$ and B_6 were able to outperform Softmax across all datasets and architectures. Although A_2 and B_6 , and A_5 and A_7 , have similar components, they both share the same operation, bessel_{i0} . The bessel_{i0} is the modified Bessel function of the first kind with degree zero. Figure 8.5 shows the

output of the bessel_{i0} function and its derivative (which is defined to be bessel_{i1} - the modified Bessel function of the first kind with degree one). Equation 8.2 gives the exact calculation of the modified Bessel function of the first kind, where n denotes the degree. The function is symmetric, elevated above the x-axis, and exhibits parabolic shape. Despite the similarity to a parabola, the derivative is much different, as $\frac{d}{dx}x^2 = 2x$ is a straight line.

$$I_n(x) = \frac{1}{\pi} \int_0^\pi e^{x \cos \theta} \cos(n\theta) d\theta \quad (8.2)$$

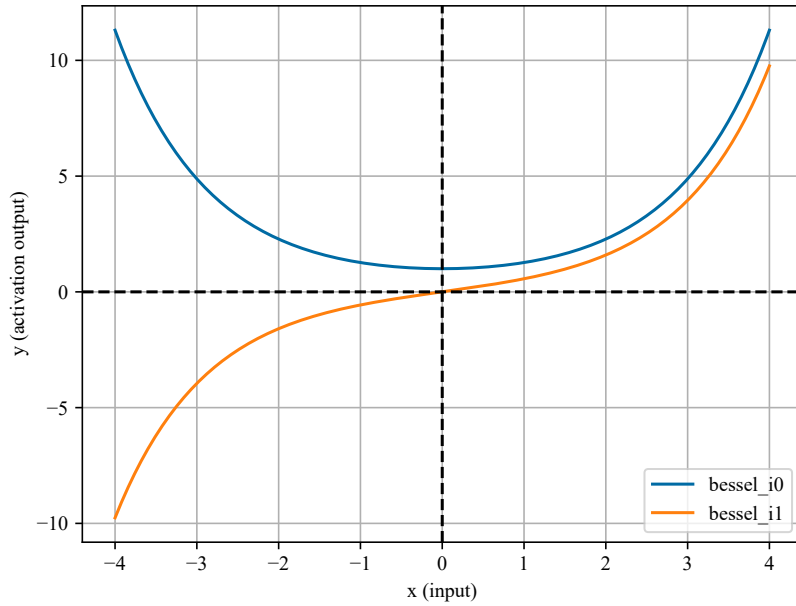


Figure 8.5: Output of bessel_{i0} and bessel_{i1} .

Named after Friedrich Wilhelm Bessel, during his study of planetary motion, the Bessel functions were first discovered as solutions to differential equations from Kepler's equations of planetary motion (Britannica). These functions also appear as solutions for differential equations in other fields of physics and mathematics

(Robert, 1990). The Bessel function is the solution to the second-order ordinary differential equation given in Equation 8.3, where the solutions are the Bessel function of the first and second kind.

$$x^2 \frac{d^2 y}{dx^2} + x \frac{dy}{dx} + (x^2 - n^2)y = 0 \quad (8.3)$$

The modified Bessel function is the solution to the second-order ordinary differential equation given in Equation 8.4, where the solutions are a combination between the Bessel and modified Bessel functions of the first and second kinds.

$$x^2 \frac{d^2 y}{dx^2} + x \frac{dy}{dx} - (x^2 + n^2)y = 0 \quad (8.4)$$

8.5 Conclusions

In this chapter, I introduced a new domain of neural component search, through the combination of neural loss function search and neural activation function search, to introduce *neural output activation function search*. After searching the search space using a particle-based genetic algorithm, multiple output activation functions were found that greatly improved performance over Softmax. Namely, $A_{2,5,7}$ and B_6 were able to outperform Softmax across all datasets and architectures. Each of them contained the the modified Bessel function of the first kind with degree zero, bessel_0 . My code can be found here: <https://github.com/OUStudent/NeuralOutputActivationFunctionSearch>.

Chapter 9

Optimizing CIFAR-10

In Chapters 5, 6, 7, and 8, neural components were discovered that outperformed their respective counterparts. However, each of these components were evaluated in a plus one manner. Meaning, never were they combined together to observe any improvement in performance. In this section, the objective is to maximize the test accuracy on the CIFAR-10 dataset using the same ResNet52v2 training procedure discussed in Chapter 8. To do so, the best output activation function was chosen. Next, using that chosen output activation function, the best loss function was chosen. Next, using that chosen output activation and loss function, the best optimizer was chosen. Lastly, using that best combination of output activation, loss function, and optimizer, the best learning rate was chosen. This final combination contains purely evolved components.

From Table 8.2, it can be seen that the A_3 output activation function achieved the best mean test result on CIFAR-10 for ResNet52v2. Therefore, it was selected as the output activation function. Table 9.1 lists the results across 10 independent runs of ResNet52v2 on CIFAR-10 for a group hand-selected loss functions discovered in Chapter 5 using the A_3 output activation function. As one can

see, NL_1 outperformed all other loss functions while also outperforming A_3 with cross-entropy. Next, the optimizers were tested. Results for all optimizers under performed A_3 . It appears that the success of A_3 is heavily tied to the Adam optimizer, as it was used during its evolution. Lastly, the learning rate schedules were tested. Using the A_3 output activation function, NL_1 loss function, and Adam optimizer, the results across 10 independent runs of ResNet52v2 on CIFAR-10 for a group of hand-selected learning rate schedules discovered in Chapter 6 and 7 are reported in Table 9.2. As one can see, the LR_0 learning rate outperformed all other schedules.

OUTPUT	MEAN $\pm$ STD.	MEDIAN
<i>Soft</i>	95.978 $\pm$ 0.099	95.945
A_3	96.693 $\pm$ 0.071	96.71
NL_1	96.765 $\pm$ 0.096	96.770
NL_2	96.601 $\pm$ 0.154	96.595
NL_3	96.587 $\pm$ 0.105	96.585
C_0	96.652 $\pm$ 0.125	96.630
C_1	96.493 $\pm$ 0.110	96.480
C_2	96.664 $\pm$ 0.119	96.680
M_0	96.520 $\pm$ 0.161	96.495
M_2	66.676 $\pm$ 36.36	96.140
A_1	96.609 $\pm$ 0.135	96.605
A_3	96.630 $\pm$ 0.127	96.605

Table 9.1: Results for ResNet52v2 on CIFAR-10 for each loss function using A_3 .

In total, the A_3 output activation function with NL_1 loss, Adam optimizer, and LR_0 learning rate schedule was able to improve performance for the standard components on CIFAR-10 for ResNet52v2 from a median test accuracy of 95.945% to 96.880%. To check for dataset transferability, this final best combination of components were evaluated using the same architecture and procedure, except on CIFAR-100. The results across 10 independent runs are listed in Table 9.3. As one

OUTPUT	MEAN $\pm$ STD.	MEDIAN
<i>Soft</i>	95.978 $\pm$ 0.099	95.945
<i>NL</i> ₁	96.765 $\pm$ 0.096	96.770
<i>S</i> ₂	96.804 $\pm$ 0.131	96.800
<i>S</i> ₄	96.687 $\pm$ 0.115	96.655
<i>S</i> ₆	96.731 $\pm$ 0.098	96.715
<i>S</i> ₈	96.806 $\pm$ 0.101	96.835
<i>S</i> ₉	96.774 $\pm$ 0.115	96.780
<i>S</i> ₁₀	96.741 $\pm$ 0.096	96.695
<i>LR</i> ₀	96.840 $\pm$ 0.126	96.880
<i>LR</i> ₁	96.746 $\pm$ 0.123	96.715
<i>LR</i> ₂	96.705 $\pm$ 0.097	96.710

Table 9.2: Results for ResNet52v2 on CIFAR-10 for each learning rate schedule using $A_3 + NL_1$.

can see, the best combination greatly outperformed Softmax and A_3 , improving performance from a mean test accuracy of 78.820% to 81.960%, indicating that the best combination did not over-fit to CIFAR-10. In addition, Table 9.4 shows the results when transferring the best combination to TinyImageNet. As one can see, the best combination greatly outperformed Softmax and A_3 , improving performance from a mean test accuracy of 64.030% to 67.730%.

OUTPUT	MEAN $\pm$ STD.	MEDIAN
<i>Soft</i>	78.825 $\pm$ 0.419	78.820
<i>A</i> ₃	81.589 $\pm$ 0.228	81.59
$A_3 + NL_1 + LR_0$	81.977 $\pm$ 0.237	81.960

Table 9.3: Results for ResNet52v2 on CIFAR-100 for the best combination.

OUTPUT	MEAN $\pm$ STD.	MEDIAN
<i>Soft</i>	64.059 $\pm$ 0.310	64.030
A_2	64.549 $\pm$ 0.504	64.585
A_3	58.289 $\pm$ 19.81	65.990
$A_3 + NL_1 + LR_0$	67.679 $\pm$ 0.250	67.730

Table 9.4: Results for ResNet52v2 on TinyImageNet for the best combination.

Chapter 10

Conclusions

Deep learning models learn through direct influence of the loss function, output activation function, optimizer, and learning rate schedule. These four major components of the learning procedure can be referred to as the learning components of a network. From the success of neural architecture search, applications of automated machine learning, and increases in computational resources for training deep learning models, the learning components of neural networks have been explored. Specifically, *neural loss function search* was performed to find a drop-in replacement of cross-entropy for classification; *neural optimizer search* was performed to find a drop-in replacement for the Adam optimizer; *neural learning rate search* was performed to find a drop-in replacement learning rate schedule for one cycle cosine decay; and finally, *neural output activation function search* was performed to find a drop-in replacement activation function for Softmax. After performing each search individually, the best components were chosen to optimize results on the CIFAR-10 dataset using the ResNet52v2 architecture. Performance from baseline components was able to be improved from 95.945% to 96.880%, from 78.820% to 81.960% on CIFAR-100, and from 64.030% to

67.730% on TinyImageNet. In the end, the learning was improved through *neural component search*.

Bibliography

- Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dandelion Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. TensorFlow: Large-scale machine learning on heterogeneous systems, 2015. URL <https://www.tensorflow.org/>. Software available from tensorflow.org.
- Alvaro, Jorge, and Jose R. Dorronsoro. Finding optimal model parameters by deterministic and annealed focused grid search. *Neurocomputing*, 72(13): 2824–2832, 2009.
- R. Battiti and G. Tecchiolli. Training neural nets with the reactive tabu search. *IEEE Transactions on Neural Networks*, 6(5):1185–1200, 1995. doi: 10.1109/72.410361.
- Irwan Bello, Barret Zoph, Vijay Vasudevan, and Quoc V Le. Neural optimizer search with reinforcement learning. In *International Conference on Machine Learning*, pages 459–468. PMLR, 2017a.
- Irwan Bello, Barret Zoph, Vijay Vasudevan, and Quoc V. Le. Neural optimizer search with reinforcement learning. In Doina Precup and Yee Whye Teh, editors, *Proceedings of the 34th International Conference on Machine Learning*, volume 70 of *Proceedings of Machine Learning Research*, pages 459–468. PMLR, 06–11 Aug 2017b. URL <https://proceedings.mlr.press/v70/bello17a.html>.
- Garrett Bingham and Risto Miikkulainen. Discovering parametric activation functions. *Neural Networks*, 148:48–65, 2022.

- Garrett Bingham, William Macke, and Risto Miikkulainen. Evolutionary optimization of deep learning activation functions. GECCO '20, pages 289–296, New York, NY, USA, 2020. Association for Computing Machinery. ISBN 9781450371285. doi: 10.1145/3377930.3389841. URL <https://doi.org/10.1145/3377930.3389841>.
- Christopher M. Bishop. *Pattern Recognition and Machine Learning (Information Science and Statistics)*. Springer-Verlag, Berlin, Heidelberg, 2006. ISBN 0387310738.
- Léon Bottou. Online algorithms and stochastic approximations. *Online learning in neural networks*, 1998.
- Jürgen Branke. Evolutionary algorithms for neural network design and training. 1995.
- The Editors of Encyclopaedia Britannica. Bessel function.
- Pedro Carvalho, Nuno Lourenço, and Penousal Machado. Evolving adaptive neural network optimizers for image classification. In Eric Medvet, Gisele Pappa, and Bing Xue, editors, *Genetic Programming*, pages 3–18, Cham, 2022. Springer International Publishing.
- S. Chalup and F. Maire. A study on hill climbing algorithms for neural network training. In *Proceedings of the 1999 Congress on Evolutionary Computation-CEC99 (Cat. No. 99TH8406)*, volume 3, pages 2014–2021 Vol. 3, 1999. doi: 10.1109/CEC.1999.785522.
- Yevgen Chebotar, Artem Molchanov, Sarah Bechtle, Ludovic Righetti, Franziska Meier, and Gaurav S. Sukhatme. Meta learning via learned loss. In *International Conference on Pattern Recognition (ICPR)*, pages 4161–4168, 2019.
- D. Chen, C.L. Giles, G.Z. Sun, H.H. Chen, Y.C. Lee, and M.W. Goudreau. Constructive learning of recurrent neural networks. In *IEEE International Conference on Neural Networks*, pages 1196–1201 vol.3, 1993. doi: 10.1109/ICNN.1993.298727.
- John Chen, Cameron Wolfe, Zhao Li, and Anastasios Kyrillidis. Demon: Improved neural network training with momentum decay. In *ICASSP 2022 - 2022 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pages 3958–3962, 2022a. doi: 10.1109/ICASSP43922.2022.9746839.
- Xiangning Chen, Chen Liang, Da Huang, Esteban Real, Yao Liu, Kaiyuan Wang, Cho-Jui Hsieh, Yifeng Lu, and Quoc V Le. Evolved optimizer for vision. In *First Conference on Automated Machine Learning (Late-Breaking Workshop)*, 2022b.

- Kiam Choo. Learning hyperparameters for neural network models using hamiltonian dynamics, 2000.
- Cody Coleman, Daniel Kang, Deepak Narayanan, Luigi Nardi, Tian Zhao, Jian Zhang, Peter Bailis, Kunle Olukotun, Chris Ré, and Matei Zaharia. Analysis of DAWNBench, a time-to-accuracy machine learning performance benchmark. *SIGOPS Oper. Syst. Rev.*, 53(1):14–25, July 2019. ISSN 0163-5980. doi: 10.1145/3352020.3352024. URL <https://doi.org/10.1145/3352020.3352024>.
- Ekin D Cubuk, Barret Zoph, Dandelion Mane, Vijay Vasudevan, and Quoc V Le. Autoaugment: Learning augmentation policies from data. *arXiv preprint arXiv:1805.09501*, 2018.
- Ekin D Cubuk, Barret Zoph, Jonathon Shlens, and Quoc V Le. Randaugment: Practical automated data augmentation with a reduced search space. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition Workshops*, pages 702–703, 2020.
- Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Li Fei-Fei. Imagenet: A large-scale hierarchical image database. In *2009 IEEE Conference on Computer Vision and Pattern Recognition*, pages 248–255, 2009. doi: 10.1109/CVPR.2009.5206848.
- Terrance DeVries and Graham W Taylor. Improved regularization of convolutional neural networks with cutout. *arXiv preprint arXiv:1708.04552*, 2017.
- T. Dozat. Incorporating nesterov momentum into adam. *ICLR Workshops*, 2016.
- A. Dubrawski. Tuning neural networks with stochastic optimization. In *Proceedings of the 1997 IEEE/RSJ International Conference on Intelligent Robot and Systems. Innovative Robotics for Real-World Applications. IROS '97*, volume 2, pages 614–620 vol.2, 1997. doi: 10.1109/IROS.1997.655075.
- John Duchi, Elad Hazan, and Yoram Singer. Adaptive subgradient methods for online learning and stochastic optimization. *Journal of Machine Learning Research*, 2011.
- Thomas Elsken, Jan Hendrik Metzen, and Frank Hutter. Neural architecture search: A survey. *The Journal of Machine Learning Research*, 20(1):1997–2017, 2019.
- Li Fei-Fei, Rob Fergus, and Pietro Perona. Learning generative visual models from few training examples: An incremental Bayesian approach tested on 101 object categories. *Computer Vision and Pattern Recognition Workshop*, 2004.

- Boyan Gao, Henry Gouk, and Timothy M. Hospedales. Searching for robustness: Loss learning for noisy classification tasks. In *Proceedings of the IEEE/CVF International Conference on Computer Vision (ICCV)*, pages 6670–6679, October 2021.
- Boyan Gao, Henry Gouk, Yongxin Yang, and Timothy Hospedales. Loss function learning for domain generalization by implicit gradient. In Kamalika Chaudhuri, Stefanie Jegelka, Le Song, Csaba Szepesvari, Gang Niu, and Sivan Sabato, editors, *Proceedings of the 39th International Conference on Machine Learning*, volume 162 of *Proceedings of Machine Learning Research*, pages 7002–7016. PMLR, 17–23 Jul 2022. URL <https://proceedings.mlr.press/v162/gao22b.html>.
- Boris Ginsburg, Patrice Castonguay, Oleksii Hrinchuk, Oleksii Kuchaiev, Vitaly Lavrukhin, Ryan Leary, Jason Li, Huyen Nguyen, and Jonathan M. Cohen. Stochastic gradient methods with layer-wise adaptive moments for training of deep networks. *CoRR*, abs/1905.11286, 2019. URL <http://arxiv.org/abs/1905.11286>.
- Xavier Glorot, Antoine Bordes, and Yoshua Bengio. Deep sparse rectifier neural networks. In Geoffrey Gordon, David Dunson, and Miroslav Dudík, editors, *Proceedings of the Fourteenth International Conference on Artificial Intelligence and Statistics*, volume 15 of *Proceedings of Machine Learning Research*, pages 315–323, Fort Lauderdale, FL, USA, 11–13 Apr 2011. PMLR.
- Santiago Gonzalez and Risto Miikkulainen. Improved training speed, accuracy, and data utilization through loss function optimization. In *2020 IEEE Congress on Evolutionary Computation (CEC)*, pages 1–8, 2020. doi: 10.1109/CEC48606.2020.9185777.
- Santiago Gonzalez and Risto Miikkulainen. Optimizing loss functions through multi-variate Taylor polynomial parameterization. In *Proceedings of the Genetic and Evolutionary Computation Conference, GECCO ’21*, pages 305–313, New York, NY, USA, 2021. Association for Computing Machinery. ISBN 9781450383509. doi: 10.1145/3449639.3459277. URL <https://doi.org/10.1145/3449639.3459277>.
- Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, 2016.
- Akhilesh Gotmare, Nitish Shirish Keskar, Caiming Xiong, and Richard Socher. A closer look at deep learning heuristics: Learning rate restarts, warmup and distillation. *arXiv preprint arXiv:1810.13243*, 2018.
- Hongyang Gu, Jianmin Li, Guangyuan Fu, Chifong Wong, Xinghao Chen, and Jun Zhu. AutoLoss-GMS: Searching generalized margin-based softmax loss function

- for person re-identification. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 4744–4753, June 2022.
- Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Identity mappings in deep residual networks. In *European Conference on Computer Vision*, pages 630–645. Springer, 2016.
- Xin He, Kaiyong Zhao, and Xiaowen Chu. AutoML: A survey of the state-of-the-art. *Knowledge-Based Systems*, 212:106622, 2021.
- Geoffrey E Hinton, Nitish Srivastava, Alex Krizhevsky, Ilya Sutskever, and Ruslan R Salakhutdinov. Improving neural networks by preventing co-adaptation of feature detectors. *arXiv preprint arXiv:1207.0580*, 2012.
- Daniel Ho, Eric Liang, Xi Chen, Ion Stoica, and Pieter Abbeel. Population based augmentation: Efficient learning of augmentation policy schedules. In *International conference on machine learning*, pages 2731–2741. PMLR, 2019.
- Jie Hu, Li Shen, and Gang Sun. Squeeze-and-excitation networks. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 7132–7141, 2018.
- Gao Huang, Yu Sun, Zhuang Liu, Daniel Sedra, and Kilian Q Weinberger. Deep networks with stochastic depth. In *Computer Vision—ECCV 2016: 14th European Conference, Amsterdam, The Netherlands, October 11–14, 2016, Proceedings, Part IV 14*, pages 646–661. Springer, 2016.
- Sergey Ioffe and Christian Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. In Francis Bach and David Blei, editors, *Proceedings of the 32nd International Conference on Machine Learning*, volume 37 of *Proceedings of Machine Learning Research*, pages 448–456, Lille, France, 07–09 Jul 2015. PMLR.
- Prateek Jain, Sham M. Kakade, Rahul Kidambi, Praneeth Netrapalli, and Aaron Sidford. Accelerating stochastic gradient descent for least squares regression, 2017. URL <https://arxiv.org/abs/1704.08227>.
- Antonia J Jones. Genetic algorithms and their applications to the design of neural networks. *Neural Computing and Applications*, 1(1):32–45, 1993.
- M. G. Kendall. A new measure of rank correlation. *Biometrika*, 30(1/2):81–93, 1938. ISSN 00063444. URL <http://www.jstor.org/stable/2332226>.

- Aditya Khosla, Nityananda Jayadevaprakash, Bangpeng Yao, and Li Fei-Fei. Novel dataset for fine-grained image categorization. In *First Workshop on Fine-Grained Visual Categorization, IEEE Conference on Computer Vision and Pattern Recognition*, Colorado Springs, CO, June 2011.
- Diederik P Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.
- Simon Kornblith, Jonathon Shlens, and Quoc V Le. Do better imagenet models transfer better? In *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*, pages 2661–2671, 2019.
- Jonathan Krause, Michael Stark, Jia Deng, and Li Fei-Fei. 3d object representations for fine-grained categorization. In *4th International IEEE Workshop on 3D Representation and Recognition (3dRR-13)*, Sydney, Australia, 2013.
- Alex Krizhevsky and Geoffrey Hinton. Learning multiple layers of features from tiny images. Technical report, University of Toronto, Toronto, Ontario, 2009.
- Ya Le and Xuan S. Yang. Tiny imagenet visual recognition challenge. 2015.
- Y. Lecun, L. Bottou, Y. Bengio, and P. Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, 1998. doi: 10.1109/5.726791.
- Zhaoqi Leng, Mingxing Tan, Chenxi Liu, Ekin Dogus Cubuk, Xiaojie Shi, Shuyang Cheng, and Dragomir Anguelov. Polyloss: A polynomial expansion perspective of classification loss functions. *arXiv preprint*, 2022. doi: 10.48550/ARXIV.2204.12511. URL <https://arxiv.org/abs/2204.12511>.
- F.H.F. Leung, H.K. Lam, S.H. Ling, and P.K.S. Tam. Tuning of the structure and parameters of a neural network using an improved genetic algorithm. *IEEE Transactions on Neural Networks*, 14(1):79–88, 2003. doi: 10.1109/TNN.2002.804317.
- Hao Li, Tianwen Fu, Jifeng Dai, Hongsheng Li, Gao Huang, and Xizhou Zhu. Autoloss-zero: Searching loss functions from scratch for generic tasks. In *2022 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 999–1008, 2022. doi: 10.1109/CVPR52688.2022.00108.
- Wei Li. cifar-10-cnn: Play deep learning with CIFAR datasets. <https://github.com/BIGBALLON/cifar-10-cnn>, 2017.
- Wei Li. Keras EfficientNetV2. https://github.com/leondgarse/keras_efficientnet_v2, 2022.

- Yuanzhi Li, Colin Wei, and Tengyu Ma. Towards explaining the regularization effect of initial large learning rate in training neural networks. *Advances in neural information processing systems*, 32, 2019.
- Hanxiao Liu, Andy Brock, Karen Simonyan, and Quoc Le. Evolving normalization-activation layers. *Advances in Neural Information Processing Systems*, 33: 13539–13550, 2020.
- Liyuan Liu, Haoming Jiang, Pengcheng He, Weizhu Chen, Xiaodong Liu, Jianfeng Gao, and Jiawei Han. On the variance of the adaptive learning rate and beyond. *arXiv preprint arXiv:1908.03265*, 2019.
- Peidong Liu, Gengwei Zhang, Bochao Wang, Hang Xu, Xiaodan Liang, Yong Jiang, and Zhenguo Li. Loss function discovery for object detection via convergence-simulation driven search. *arXiv preprint arXiv:2102.04700*, 2021.
- Ilya Loshchilov and Frank Hutter. Sgdr: Stochastic gradient descent with warm restarts. *arXiv preprint arXiv:1608.03983*, 2016.
- Ilya Loshchilov and Frank Hutter. Decoupled weight decay regularization. *ICLR*, 2019.
- James Lucas, Richard S. Zemel, and Roger B. Grosse. Aggregated momentum: Stability through passive damping. *CoRR*, abs/1804.00325, 2018. URL <http://arxiv.org/abs/1804.00325>.
- Sean Luke and Liviu Panait. A comparison of bloat control methods for genetic programming. *Evolutionary Computation*, 14(3):309–344, 2006.
- Jerry Ma and Denis Yarats. Quasi-hyperbolic momentum and adam for deep learning. *CoRR*, abs/1810.06801, 2018. URL <http://arxiv.org/abs/1810.06801>.
- Mitchell P. Marcus, Beatrice Santorini, and Mary Ann Marcinkiewicz. Building a large annotated corpus of English: The Penn Treebank. *Computational Linguistics*, 19(2):313–330, 1993. URL <https://aclanthology.org/J93-2004>.
- James Martens. Deep learning via hessian-free optimization. In *Proceedings of the 27th International Conference on International Conference on Machine Learning, ICML’10*, pages 735–742, Madison, WI, USA, 2010. Omnipress. ISBN 9781605589077.
- Michael McInerney and Atam P Dhawan. Use of genetic algorithms with back-propagation in training of feedforward neural networks. In *IEEE international conference on neural networks*, pages 203–208. IEEE, 1993.

- Stephen Merity, Nitish Shirish Keskar, and Richard Socher. Regularizing and optimizing lstm language models. *arXiv preprint arXiv:1708.02182*, 2017.
- David J Montana, Lawrence Davis, et al. Training feedforward neural networks using genetic algorithms. In *IJCAI*, volume 89, pages 762–767, 1989.
- Brandon Morgan. Efficient neural architecture search using genetic algorithm, 2022.
- Rafael Müller, Simon Kornblith, and Geoffrey E Hinton. When does label smoothing help? *Advances in neural information processing systems*, 32, 2019.
- Kourosh Neshatian and Mengjie Zhang. Unsupervised elimination of redundant features using genetic programming. In Ann Nicholson and Xiaodong Li, editors, *AI 2009: Advances in Artificial Intelligence*, pages 432–442, Berlin, Heidelberg, 2009. Springer Berlin Heidelberg. ISBN 978-3-642-10439-8.
- YU. E. Nesterov. A method of solving a convex programming problem with convergence rate $o(1/k^2)$. 1983.
- M-E. Nilsback and A. Zisserman. Automated flower classification over a large number of classes. In *Proceedings of the Indian Conference on Computer Vision, Graphics and Image Processing*, Dec 2008.
- O. M. Parkhi, A. Vedaldi, A. Zisserman, and C. V. Jawahar. Cats and dogs. In *IEEE Conference on Computer Vision and Pattern Recognition*, 2012.
- Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. Pytorch: An imperative style, high-performance deep learning library. In *Advances in Neural Information Processing Systems 32*, pages 8024–8035. Curran Associates, Inc., 2019.
- Tran Thi Phuong and Le Trieu Phong. On the convergence proof of amsgrad and a new version. *CoRR*, abs/1904.03590, 2019. URL <http://arxiv.org/abs/1904.03590>.
- Prajit Ramachandran, Barret Zoph, and Quoc V Le. Searching for activation functions. *arXiv preprint arXiv:1710.05941*, 2017.
- Christian Rayman, Qi Chen, Bing Xue, and Mengjie Zhang. Learning symbolic model-agnostic loss functions via meta-learning. *arXiv preprint arXiv:2209.08907*, 2022.

- Esteban Real, Alok Aggarwal, Yanping Huang, and Quoc V Le. Regularized evolution for image classifier architecture search. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 33, pages 4780–4789, 2019.
- Christian Robert. Modified bessel functions and their applications in probability and statistics. *Statistics and Probability Letters*, 9(2):155–161, 1990. ISSN 0167-7152. doi: [https://doi.org/10.1016/0167-7152\(92\)90011-S](https://doi.org/10.1016/0167-7152(92)90011-S). URL <https://www.sciencedirect.com/science/article/pii/016771529290011S>.
- J David Schaffer, Darrell Whitley, and Larry J Eshelman. Combinations of genetic algorithms and neural networks: A survey of the state of the art. In *[Proceedings] COGANN-92: International Workshop on Combinations of Genetic Algorithms and Neural Networks*, pages 1–37. IEEE, 1992.
- Tom Schaul, Sixin Zhang, and Yann LeCun. No more pesky learning rates. In Sanjoy Dasgupta and David McAllester, editors, *Proceedings of the 30th International Conference on Machine Learning*, volume 28 of *Proceedings of Machine Learning Research*, pages 343–351, Atlanta, Georgia, USA, 17–19 Jun 2013. PMLR. URL <https://proceedings.mlr.press/v28/schaul13.html>.
- Noam Shazeer and Mitchell Stern. Adafactor: Adaptive learning rates with sublinear memory cost. *CoRR*, abs/1804.04235, 2018. URL <http://arxiv.org/abs/1804.04235>.
- M. Signorelli, V. Vinciotti, and E. C. Wit. Neat: an efficient network enrichment analysis test. *BMC Bioinformatics*, 17:352, 2016. URL <https://doi.org/10.1186/s12859-016-1203-6>.
- Leslie N Smith. Cyclical learning rates for training neural networks. In *2017 IEEE winter conference on applications of computer vision (WACV)*, pages 464–472. IEEE, 2017.
- Leslie N Smith. A disciplined approach to neural network hyper-parameters: Part 1–learning rate, batch size, momentum, and weight decay. *arXiv preprint arXiv:1803.09820*, 2018a.
- Leslie N Smith. A disciplined approach to neural network hyper-parameters: Part 1–learning rate, batch size, momentum, and weight decay. *arXiv preprint arXiv:1803.09820*, 2018b.
- Leslie N Smith and Nicholay Topin. Super-convergence: Very fast training of neural networks using large learning rates. In *Artificial intelligence and machine learning for multi-domain operations applications*, volume 11006, pages 369–386. SPIE, 2019.

- Mingxing Tan and Quoc Le. Efficientnet: Rethinking model scaling for convolutional neural networks. In *International conference on machine learning*, pages 6105–6114. PMLR, 2019.
- Mingxing Tan and Quoc Le. Efficientnetv2: Smaller models and faster training. In *International Conference on Machine Learning*, pages 10096–10106. PMLR, 2021.
- Tijmen Tieleman, Geoffrey Hinton, et al. Lecture 6.5-rmsprop: Divide the gradient by a running average of its recent magnitude. *COURSERA: Neural networks for machine learning*, 4(2):26–31, 2012.
- Qianqian Tong, Guannan Liang, and Jinbo Bi. Calibrating the learning rate for adaptive gradient methods to improve generalization performance. *CoRR*, abs/1908.00700, 2019. URL <http://arxiv.org/abs/1908.00700>.
- Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Lukasz Kaiser, and Illia Polosukhin. Attention is all you need. *Advances in neural information processing systems*, 30, 2017.
- Darrell Whitley, Timothy Starkweather, and Christopher Bogart. Genetic algorithms and neural networks: Optimizing connections and connectivity. *Parallel computing*, 14(3):347–361, 1990.
- Zhen Xu, Andrew M Dai, Jonas Kemp, and Luke Metz. Learning an adaptive learning rate schedule. *arXiv preprint arXiv:1909.09712*, 2019.
- Abolfazl Younesi, Mohsen Ansari, MohammadAmin Fazli, Alireza Ejlali, Muhammad Shafique, and Jörg Henkel. A comprehensive survey of convolutions in deep learning: Applications, challenges, and future trends. *arXiv preprint arXiv:2402.15490*, 2024.
- Jen-Lun Yuan and T.L. Fine. Neural-network design for small training sets of high dimension. *IEEE Transactions on Neural Networks*, 9(2):266–280, 1998. doi: 10.1109/72.661122.
- Sangdoo Yun, Dongyoon Han, Seong Joon Oh, Sanghyuk Chun, Junsuk Choe, and Youngjoon Yoo. Cutmix: Regularization strategy to train strong classifiers with localizable features. In *Proceedings of the IEEE/CVF International Conference on Computer Vision*, pages 6023–6032, 2019.
- Sergey Zagoruyko and Nikos Komodakis. Wide residual networks. *arXiv preprint arXiv:1605.07146*, 2016.

- Manzil Zaheer, Sashank Reddi, Devendra Sachan, Satyen Kale, and Sanjiv Kumar. Adaptive methods for nonconvex optimization. In S. Bengio, H. Wallach, H. Larochelle, K. Grauman, N. Cesa-Bianchi, and R. Garnett, editors, *Advances in Neural Information Processing Systems*, volume 31. Curran Associates, Inc., 2018. URL <https://proceedings.neurips.cc/paper/2018/file/90365351ccc7437a1309dc64e4db32a3-Paper.pdf>.
- Matthew D. Zeiler. ADADELTA: an adaptive learning rate method. *CoRR*, abs/1212.5701, 2012. URL <http://arxiv.org/abs/1212.5701>.
- Hongyi Zhang, Moustapha Cisse, Yann N Dauphin, and David Lopez-Paz. mixup: Beyond empirical risk minimization. *arXiv preprint arXiv:1710.09412*, 2017.
- Jian Zhang and Ioannis Mitliagkas. Yellowfin and the art of momentum tuning, 2017. URL <https://arxiv.org/abs/1706.03471>.
- Jingzhao Zhang, Tianxing He, Suvrit Sra, and Ali Jadbabaie. Why gradient clipping accelerates training: A theoretical justification for adaptivity. *arXiv preprint arXiv:1905.11881*, 2019.
- Shen-Yi Zhao, Yin-Peng Xie, and Wu-Jun Li. Stochastic normalized gradient descent with momentum for large batch training, 2020. URL <https://arxiv.org/abs/2007.13985>.
- Juntang Zhuang, Tommy Tang, Yifan Ding, Sekhar Tatikonda, Nicha C. Dvornek, Xenophon Papademetris, and James S. Duncan. Adabelief optimizer: Adapting stepsizes by the belief in observed gradients. *CoRR*, abs/2010.07468, 2020. URL <https://arxiv.org/abs/2010.07468>.
- Barret Zoph and Quoc V. Le. Neural architecture search with reinforcement learning. *arXiv preprint*, abs/1611.01578, 2016. doi: 10.48550/ARXIV.1611.01578. URL <https://arxiv.org/abs/1611.01578>.
- Barret Zoph, Vijay Vasudevan, Jonathon Shlens, and Quoc V. Le. Learning transferable architectures for scalable image recognition. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, June 2018.

Appendix A

Appendix

A.1 Neural Loss Function Search

In this appendix, the binary and three-dimensional phenotypes of the discovered loss functions from Chapter 5 (Image-augmentation specific evolution) will be covered.

A.1.1 Base Losses

The normalized and raw binary phenotypes of the final base losses are shown in Figure A.1. The non-normalized three-dimensional representations for $B0$, $B1$, and $B2$ are shown in Figures A.2, A.3, and A.4. As one can see, the base functions are extremely different from CE. Specifically, their slope values are much less steeper than CE. Their three-dimensional representations look like saddles. Perhaps these saddle representations pose problems at higher dimensions, more than one class, as all base functions collapse when transferring to CIFAR-100.

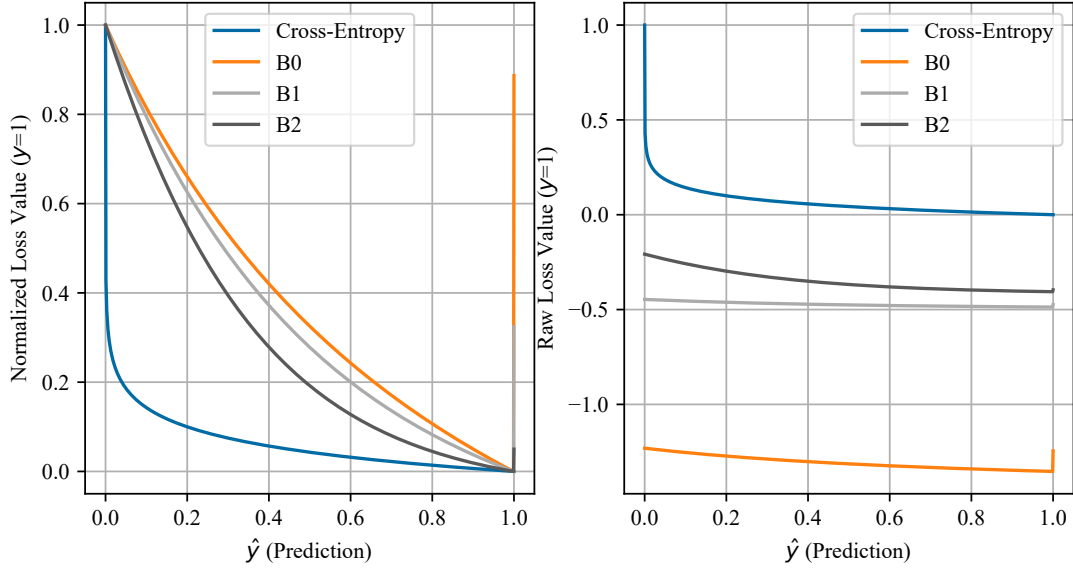


Figure A.1: Binary Phenotype of the final Base losses against cross-entropy for both normalized and raw loss values.

A.1.2 Cutout Losses

The normalized and raw binary phenotypes of the final cutout losses are shown in Figure A.5. The non-normalized three-dimensional representations for $C0$, $C1$, and $C2$ are shown in Figures A.6, A.7, and A.8. As one can see from the binary phenotypes, the final cutout functions learn a little hook around $\hat{y} \sim 1$, a small label smoothing value. Unlike the base losses, the cutout losses are not so similar in their three-dimensional representations, despite having similar binary phenotypes. For example, $C0$ and $C1$ are similar, but are very different from $C2$.

A.1.3 Mixup Losses

The normalized and raw binary phenotypes of the final mixup losses are shown in Figure A.9. The non-normalized three-dimensional representations for $M0$,

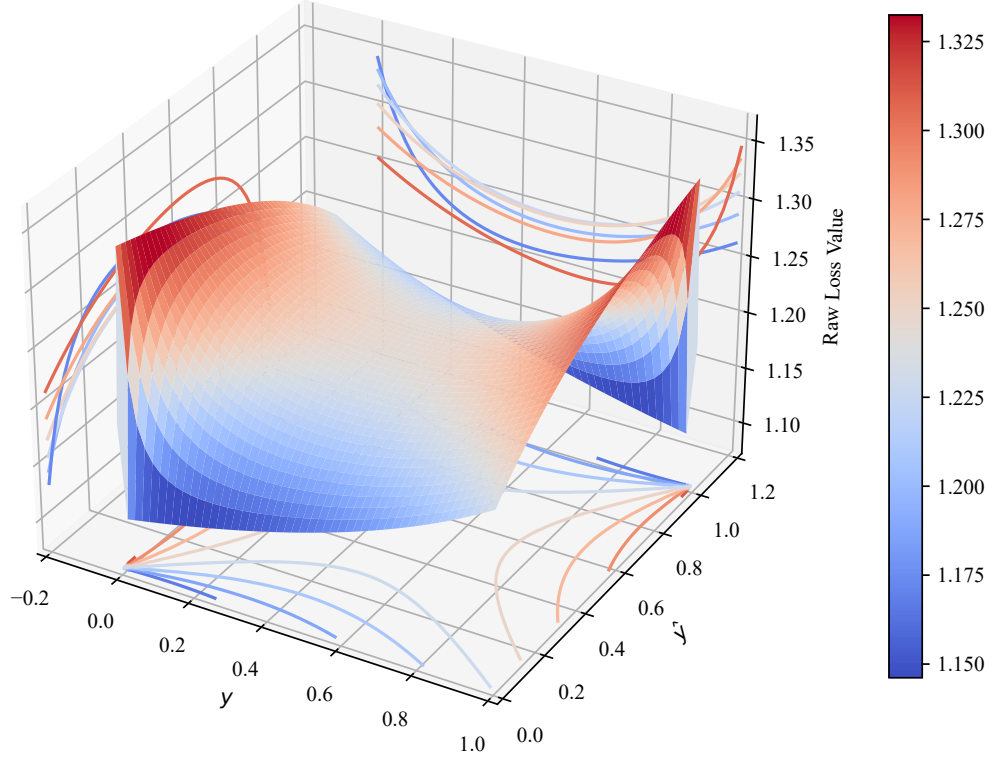


Figure A.2: 3D plot of the raw phenotype of the $B0$ loss.

$M1$, and $M2$ are shown in Figures A.10, A.11, and A.12. As one can see from the binary phenotypes, the final mixup loss functions learn less steeper slopes than CE. In addition, $M2$ and $M1$ appear to have the same normalized values, indicating that they have the exact same shape, but different scaling. From the three-dimensional representations, $M0$ is different from $M1$ and $M2$, better resembling CE, except with more exaggerated curves when y and $\hat{y}$ are most different, resembling A2.

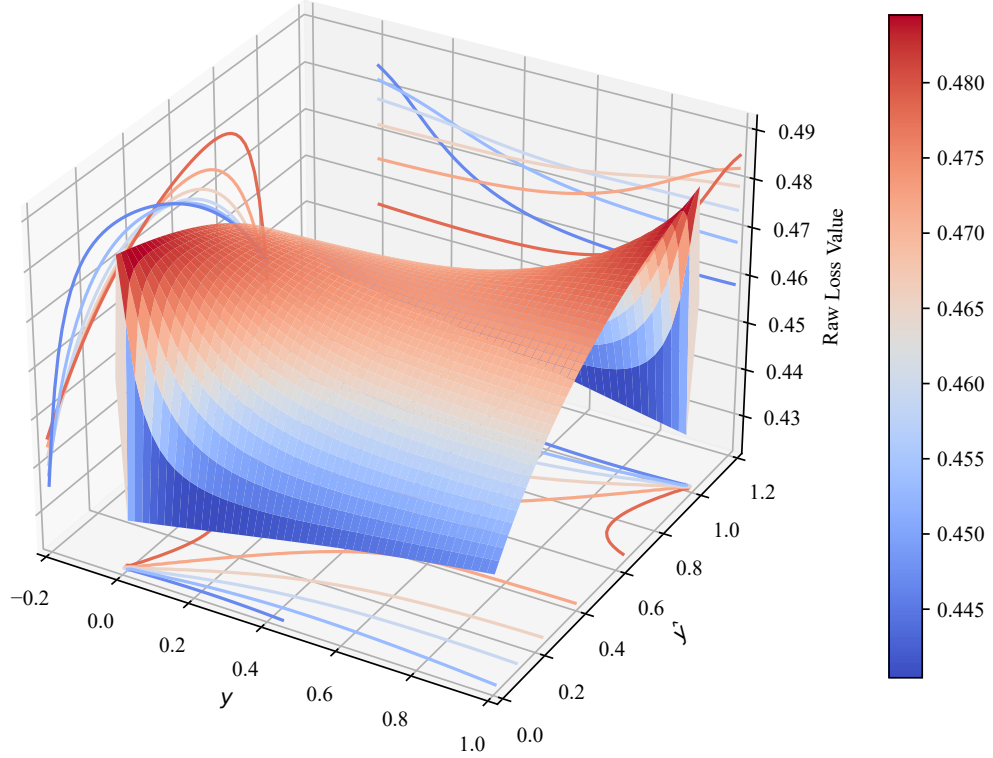


Figure A.3: 3D plot of the raw phenotype of the $B1$ loss.

A.1.4 RandAug Losses

The normalized and raw binary phenotypes of the final mixup losses are shown in Figure A.13. The non-normalized three-dimensional representations for $R0$, $R1$, and $R2$ are shown in Figures A.14, A.15, and A.16. As one can see from the binary phenotypes, the final RandAug functions host a variety of differences within themselves. Mainly, $R0$ is the only loss function discovered where its maximum loss value does not occur when $\hat{y} = 0$ for $y = 1$, but actually when $\hat{y} = 1$. Meaning, $R0$ gives the greatest penalty to when the model's output directly matches the target. It seems that this level of regularization is too much for deep

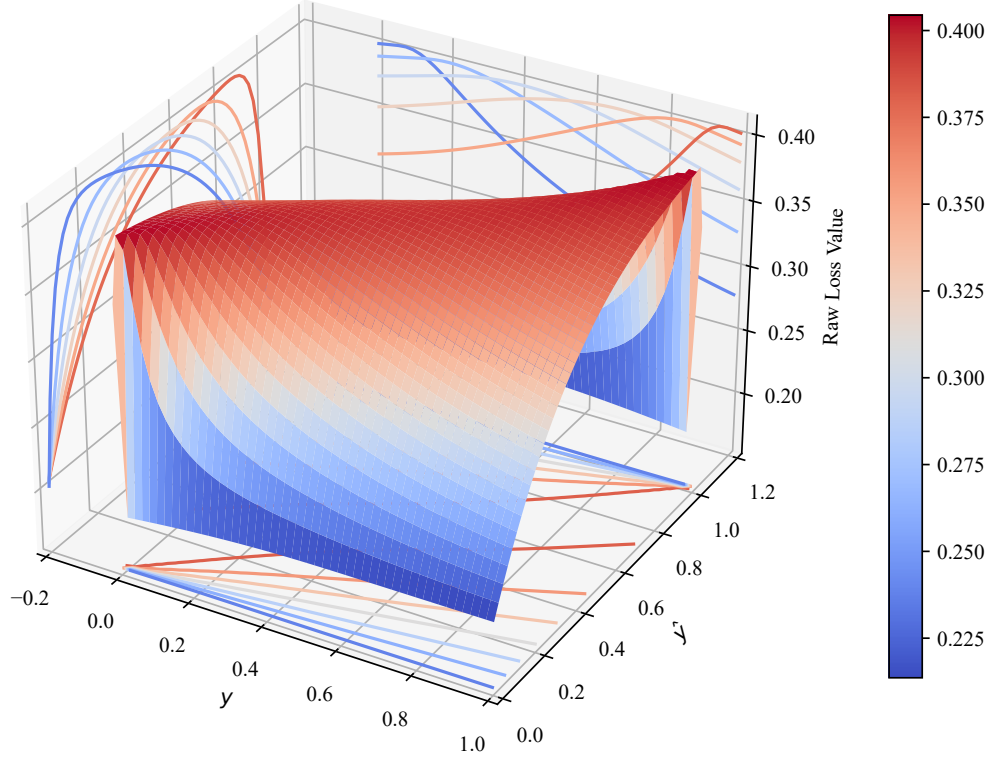


Figure A.4: 3D plot of the raw phenotype of the $B2$ loss.

learning models as $R0$ performs extremely poorly on CIFAR-100.

A.1.5 All Losses

The normalized and raw binary phenotypes of the final mixup losses are shown in Figure A.17. The non-normalized three-dimensional representations for $A0$, $A1$, and $A2$ are shown in Figures A.18 and A.19. As one can see from the binary and three-dimensional phenotypes, the three final all functions are extremely similar in shape. The main difference is $A2$ has larger raw loss values. Empirically, these results seem to support the larger raw loss values as $A2$ consistently outperformed

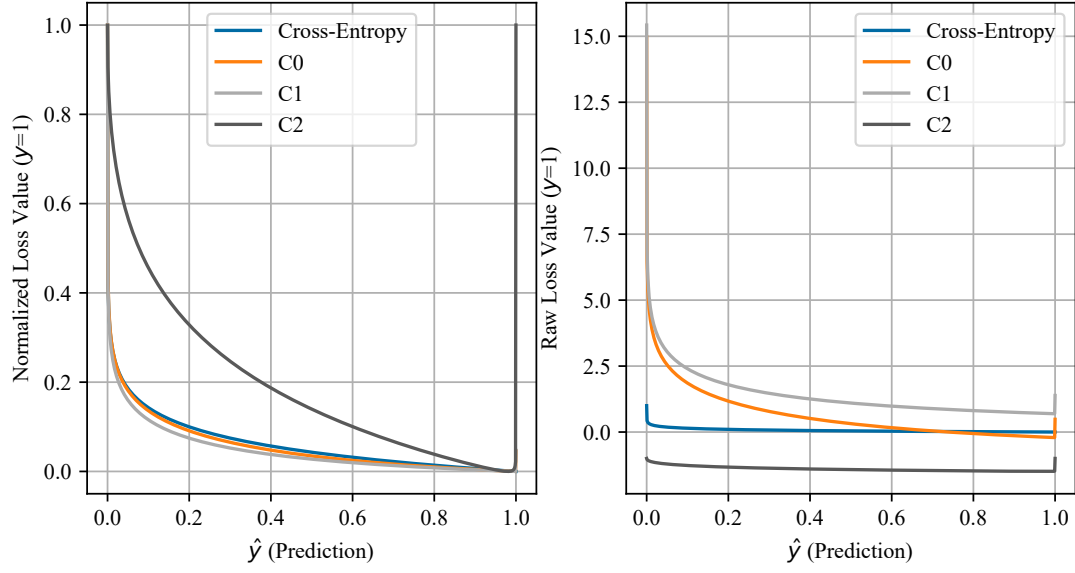


Figure A.5: Binary Phenotype of the final Cutout losses against cross-entropy for both normalized and raw loss values.

both A0 and A1.

A.2 Neural Output Function Search

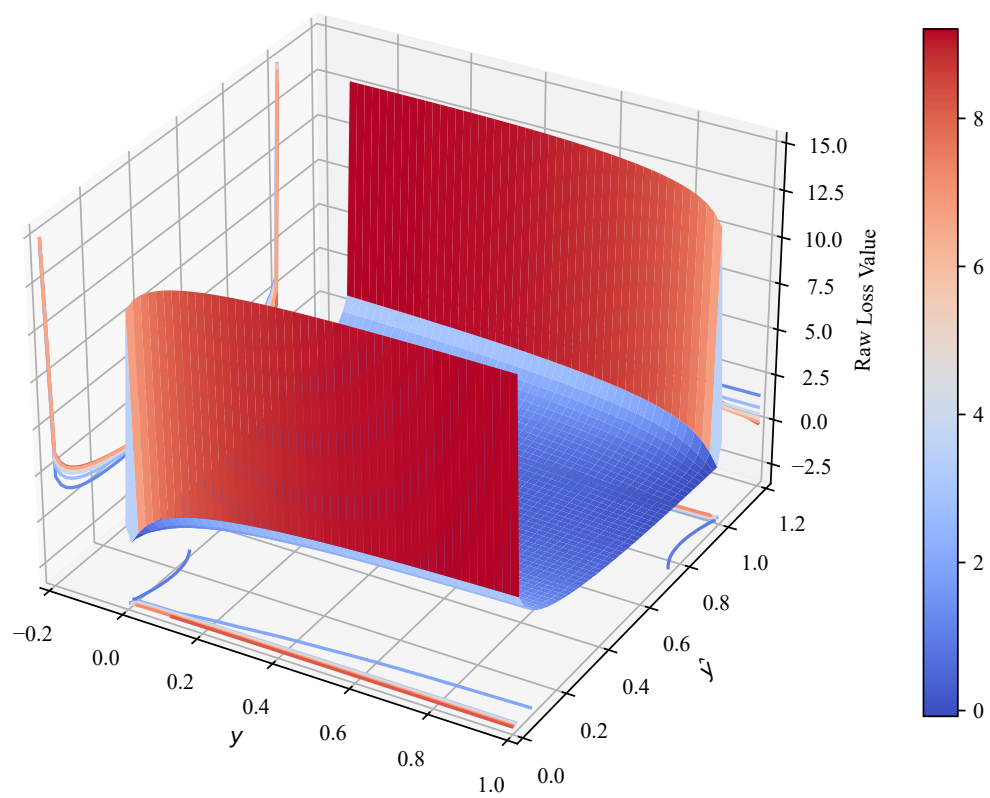


Figure A.6: 3D plot of the raw phenotype of the $C0$ loss.

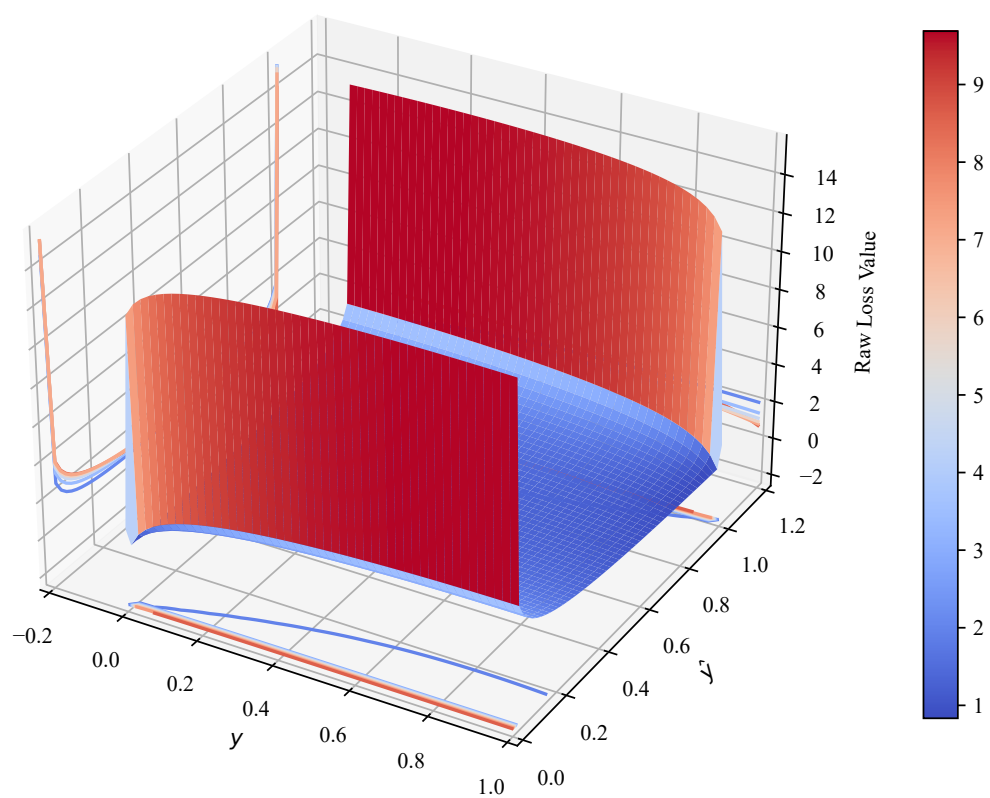


Figure A.7: 3D plot of the raw phenotype of the $C1$ loss.

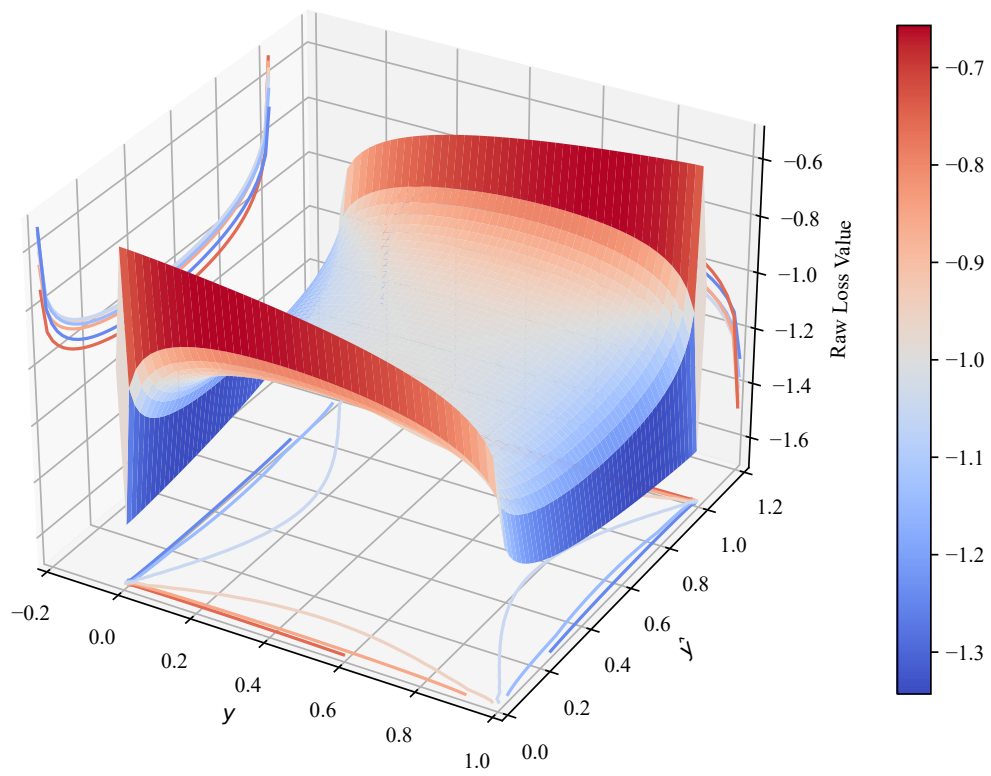


Figure A.8: 3D plot of the raw phenotype of the $C2$ loss.

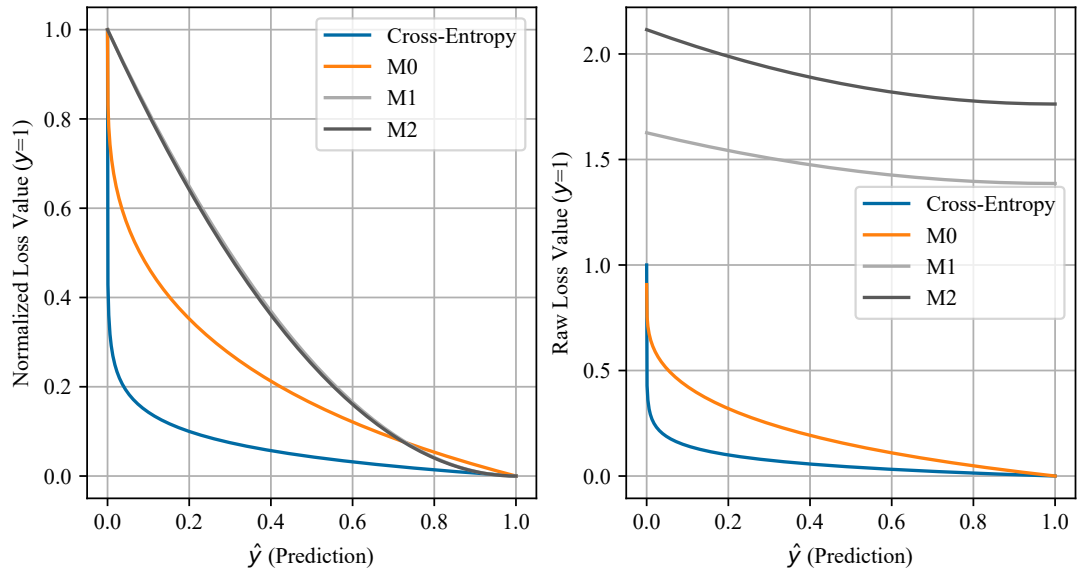


Figure A.9: Binary Phenotype of the final Mixup losses against cross-entropy for both normalized and raw loss values.

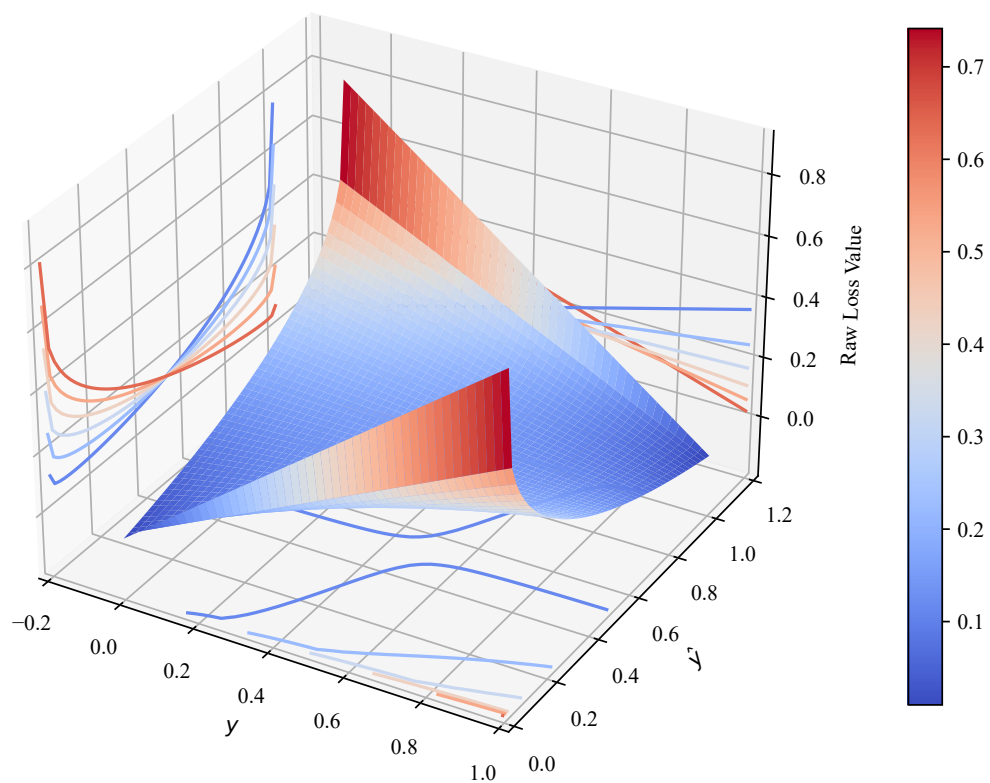


Figure A.10: 3D plot of the raw phenotype of the $M0$ loss.

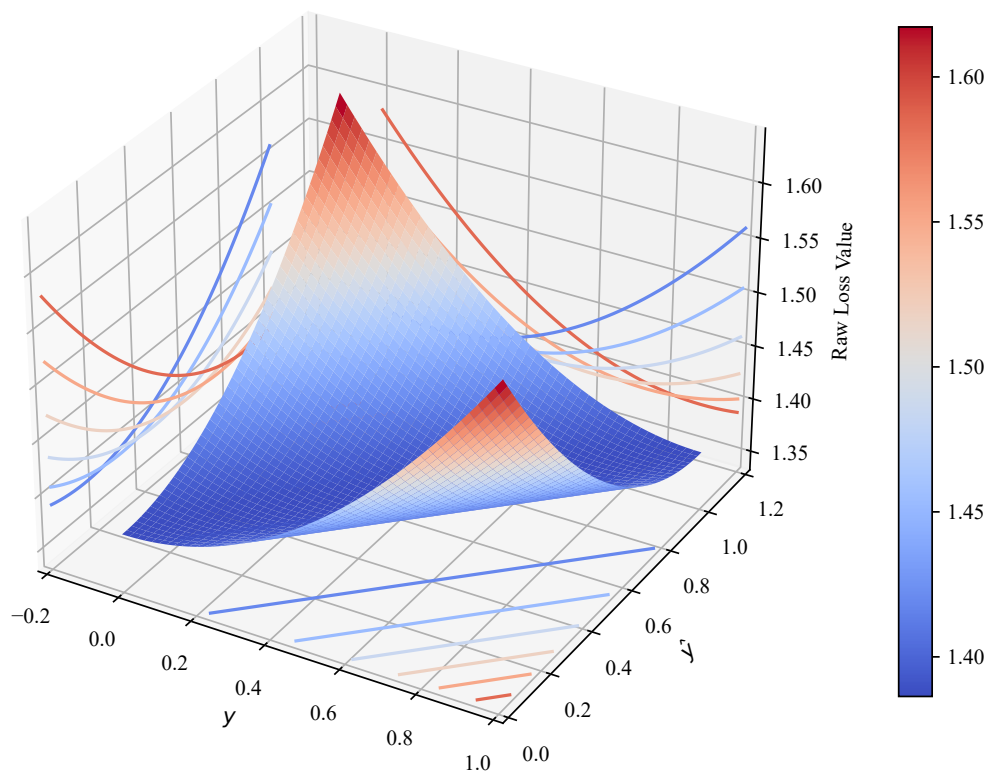


Figure A.11: 3D plot of the raw phenotype of the $M1$ loss.

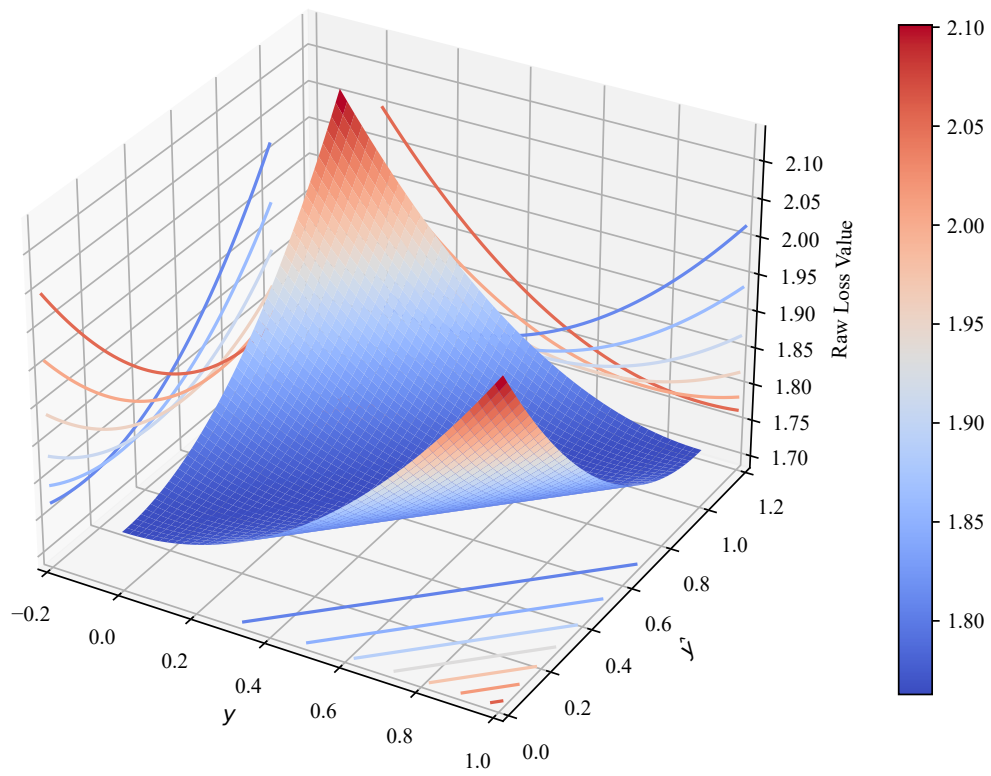


Figure A.12: 3D plot of the raw phenotype of the $M2$ loss.

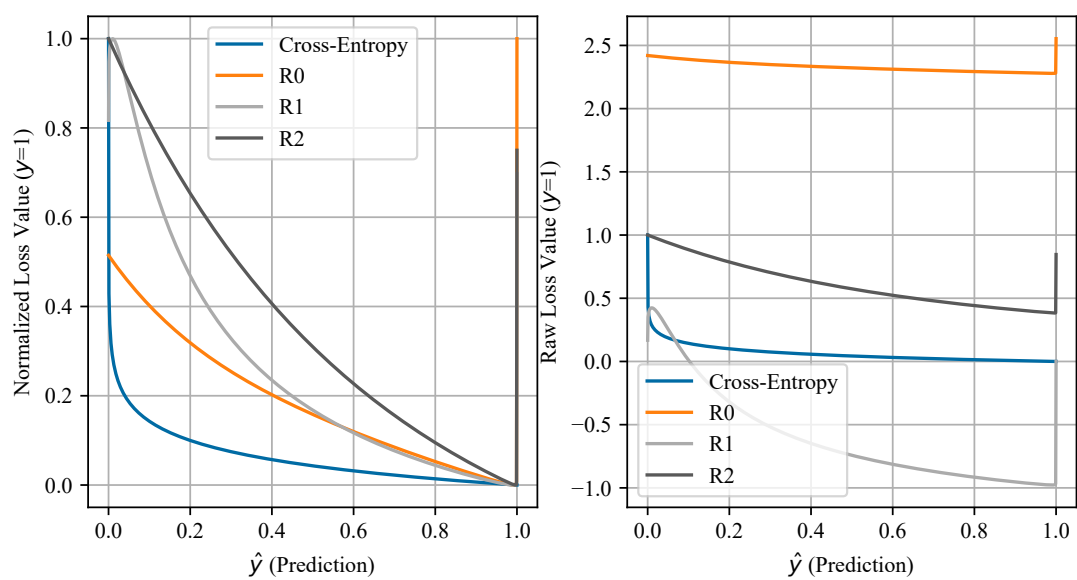


Figure A.13: Binary Phenotype of the final RandAug losses against cross-entropy for both normalized and raw loss values.

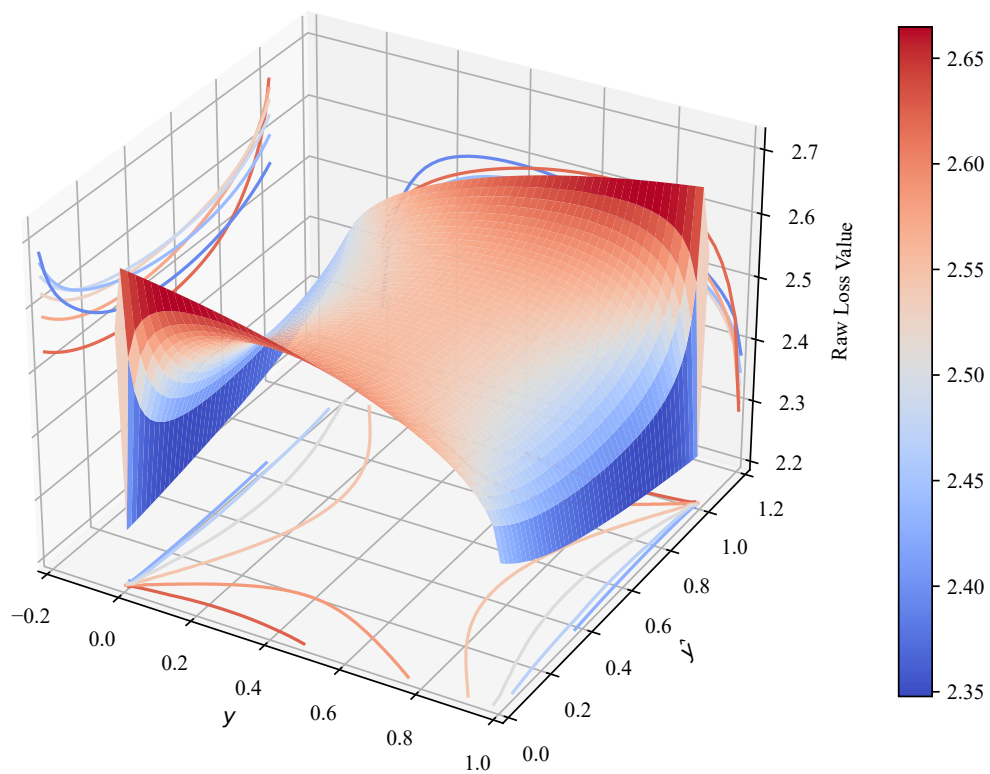


Figure A.14: 3D plot of the raw phenotype of the R_0 loss.

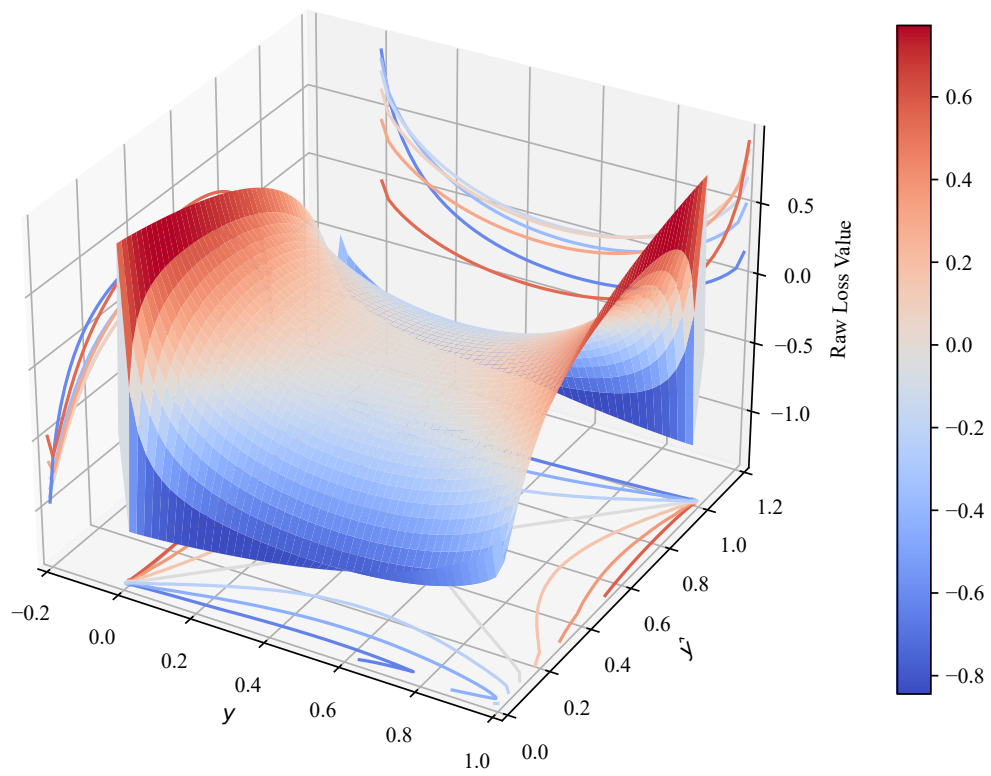


Figure A.15: 3D plot of the raw phenotype of the $R1$ loss.

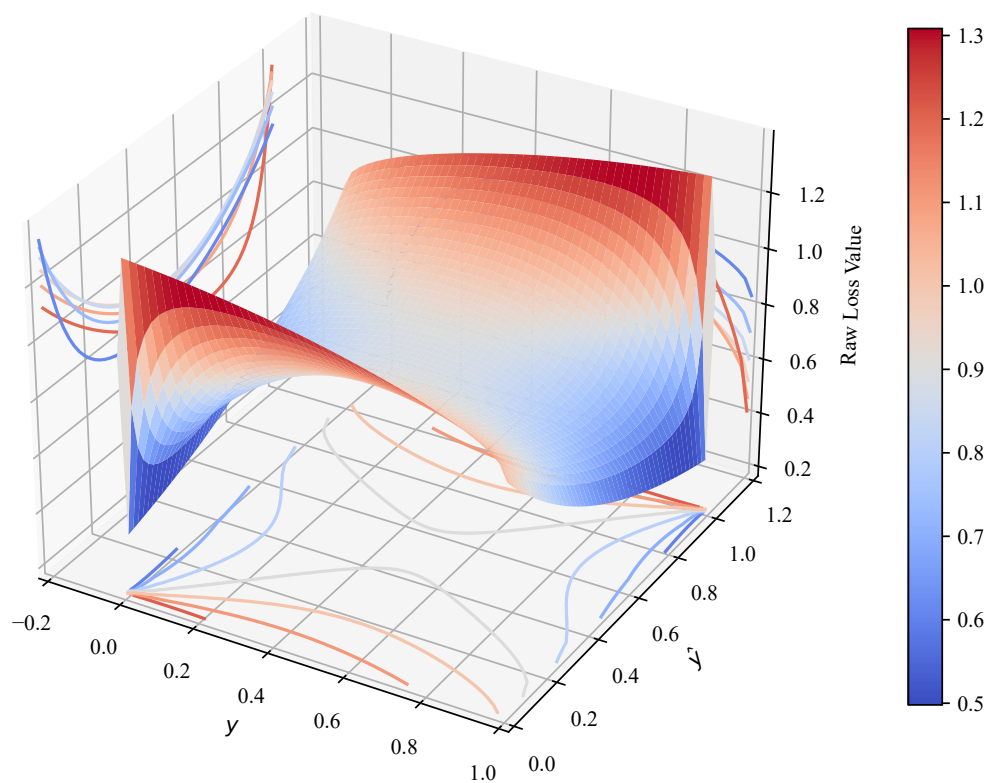


Figure A.16: 3D plot of the raw phenotype of the $R2$ loss.

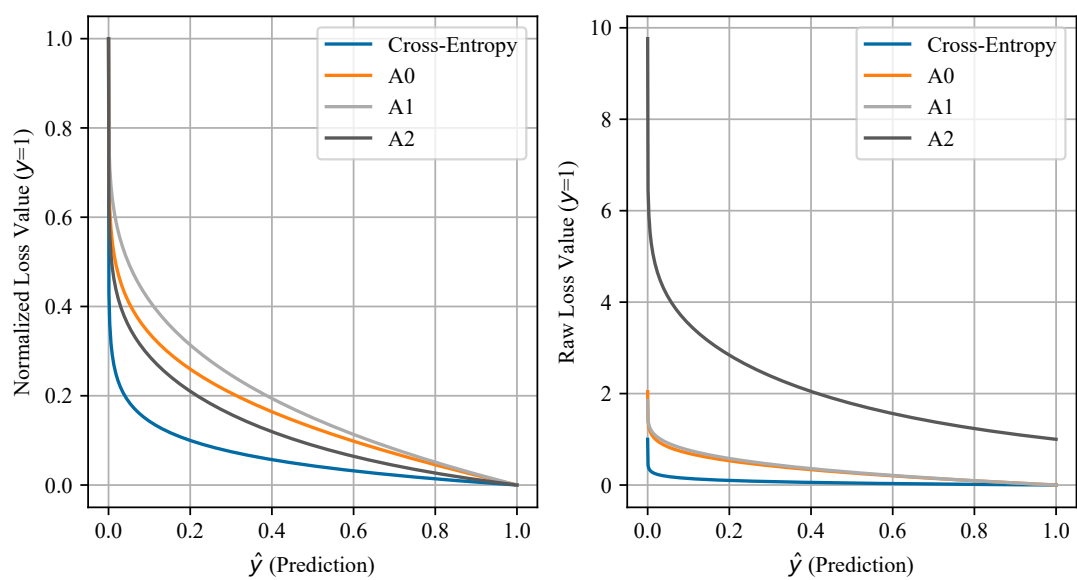


Figure A.17: Binary Phenotype of the final All losses against cross-entropy for both normalized and raw loss values.

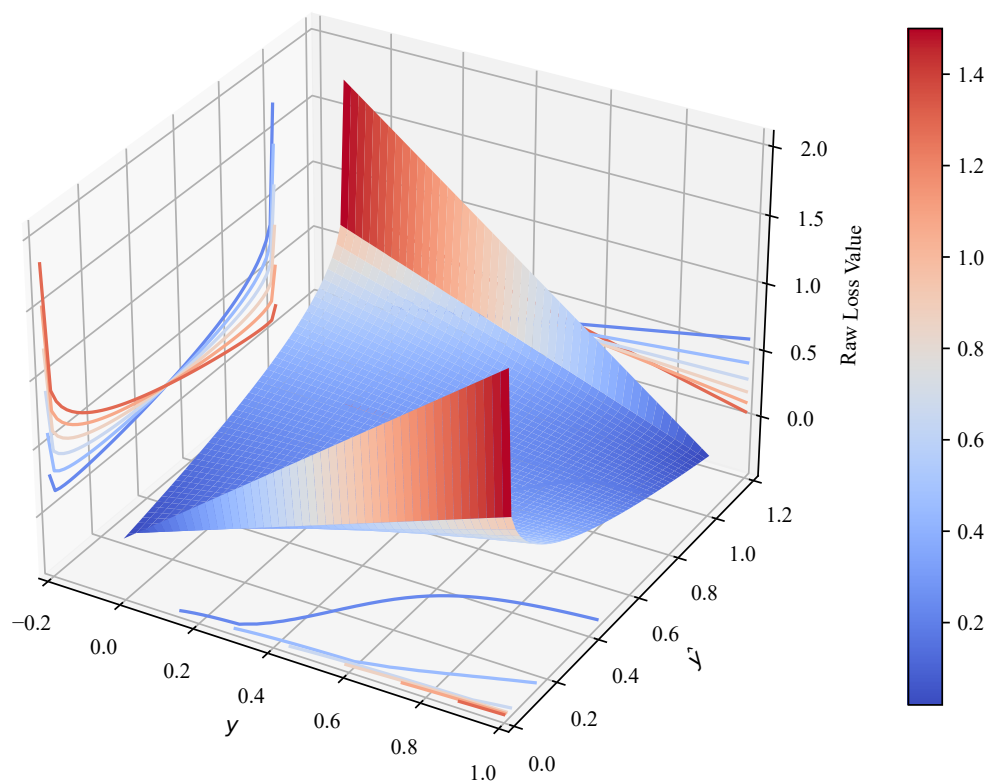


Figure A.18: 3D plot of the raw phenotype of the A0 loss.

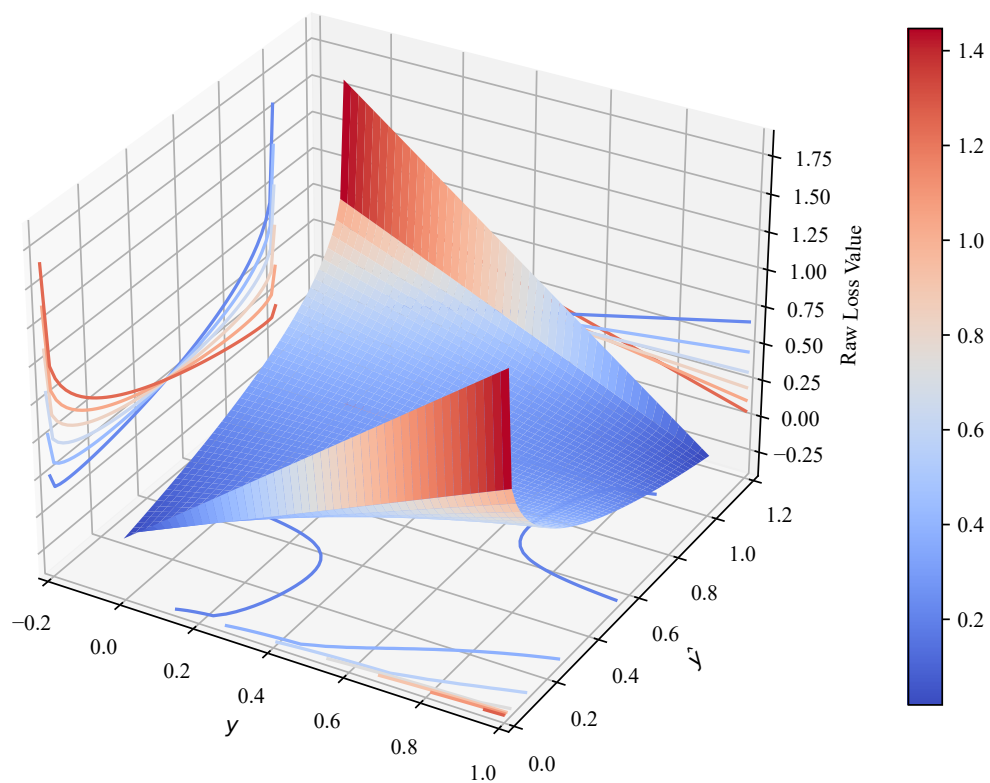


Figure A.19: 3D plot of the raw phenotype of the A1 loss.

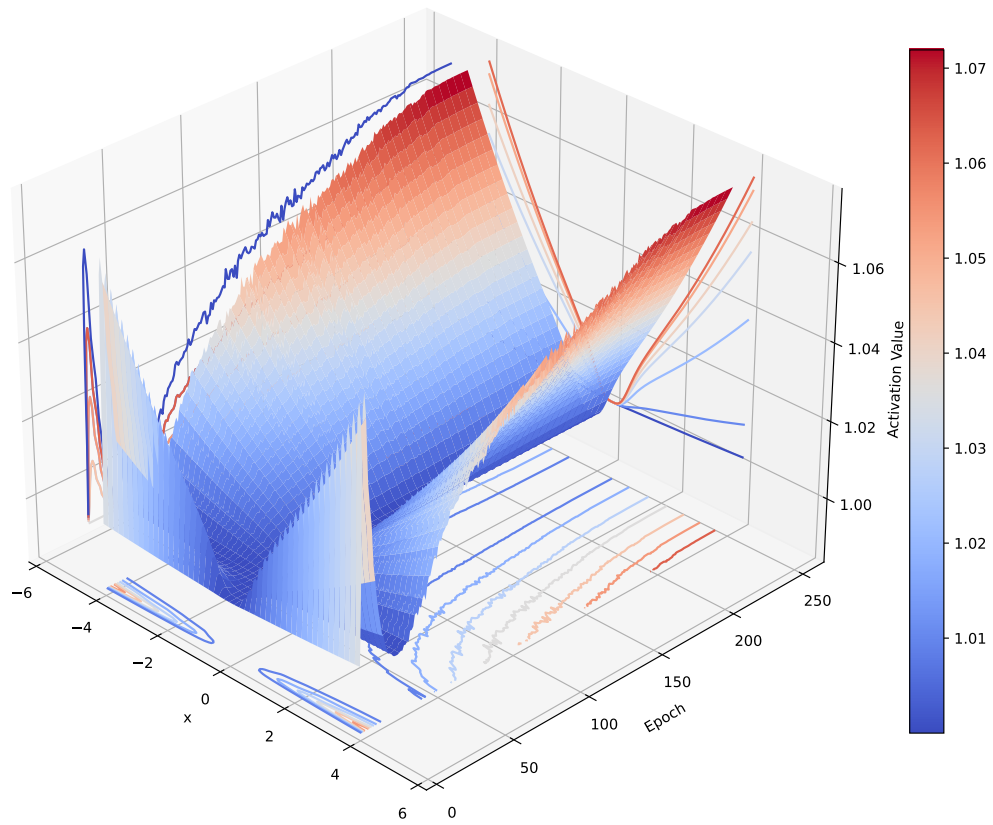


Figure A.20: Output of A_2 over time for EfficientNetB0v2 on CIFAR-10.

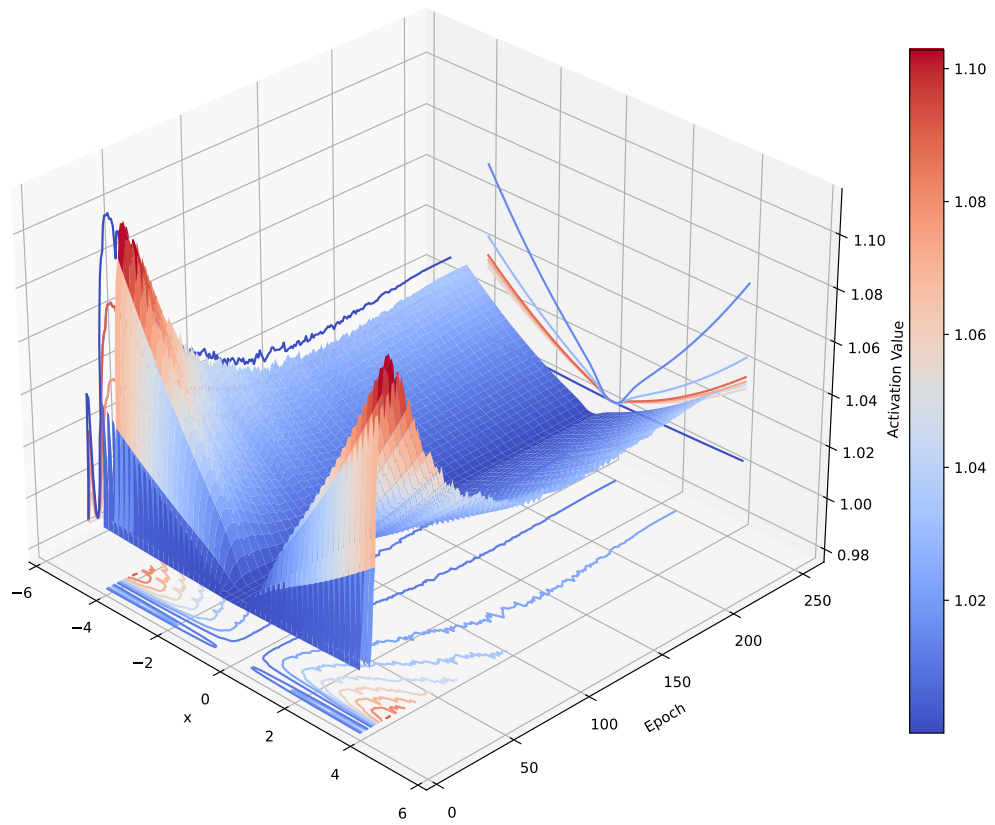


Figure A.21: Output of A_2 over time for EfficientNetB0v2 on CIFAR-100.

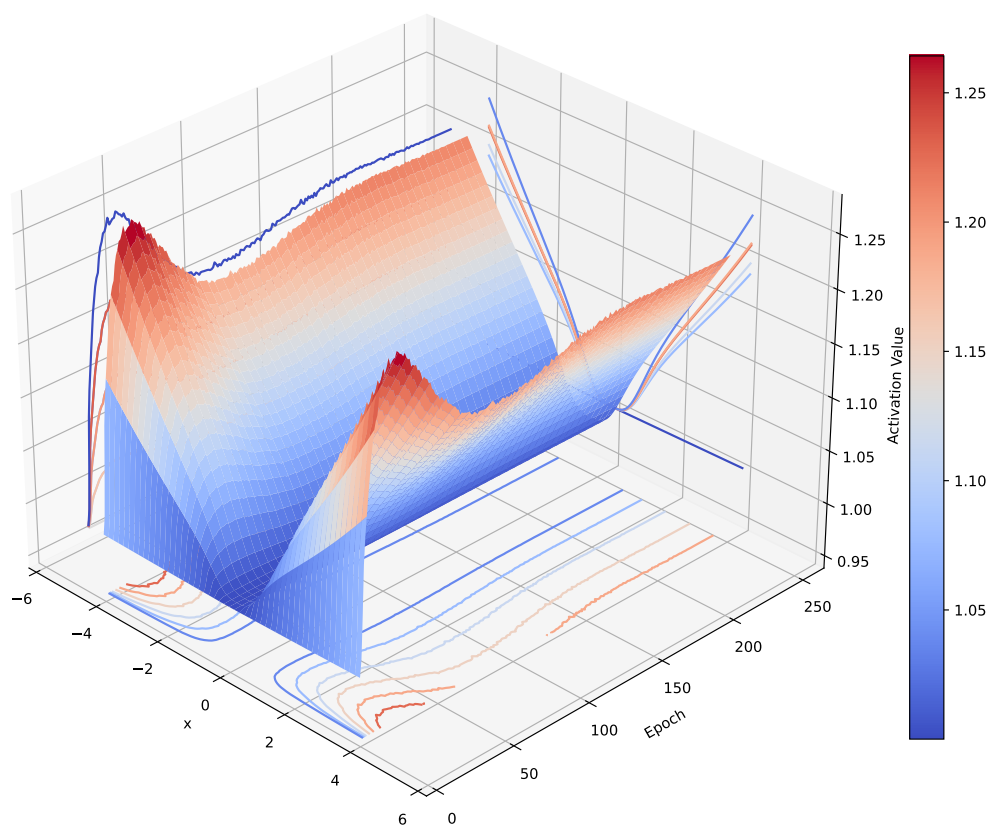


Figure A.22: Ouput of A_2 over time for ResNet52v2 on CIFAR-100.

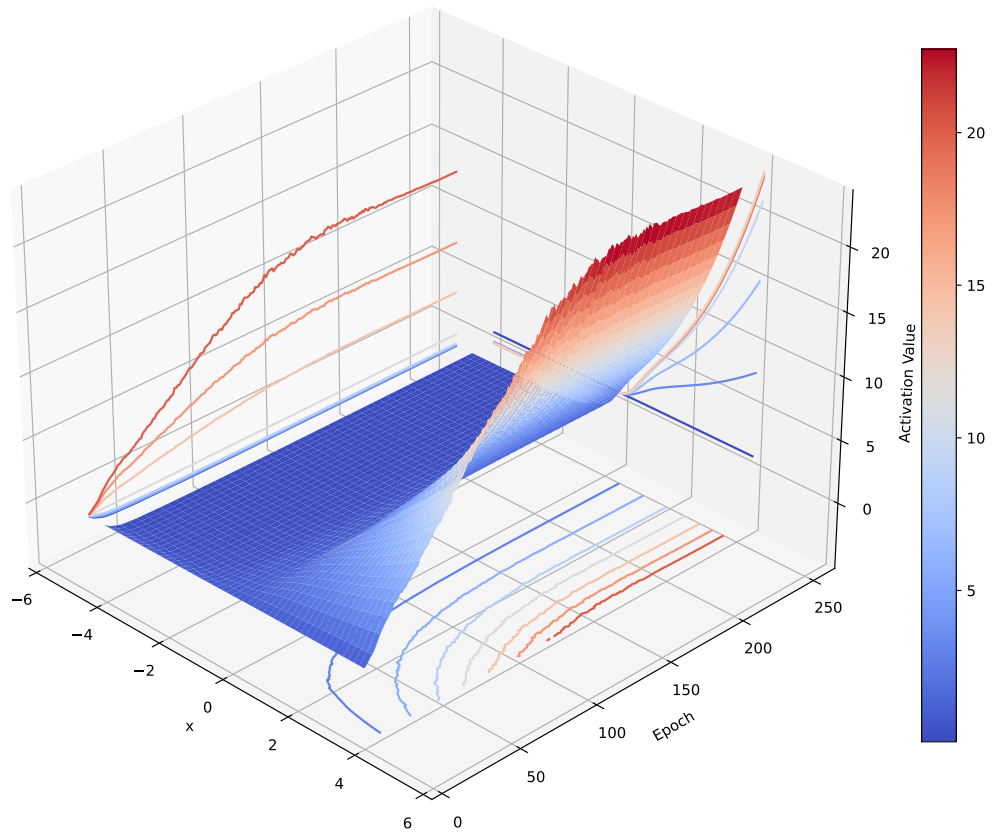


Figure A.23: Output of A_3 over time for EfficientNetB0v2 on CIFAR-10.

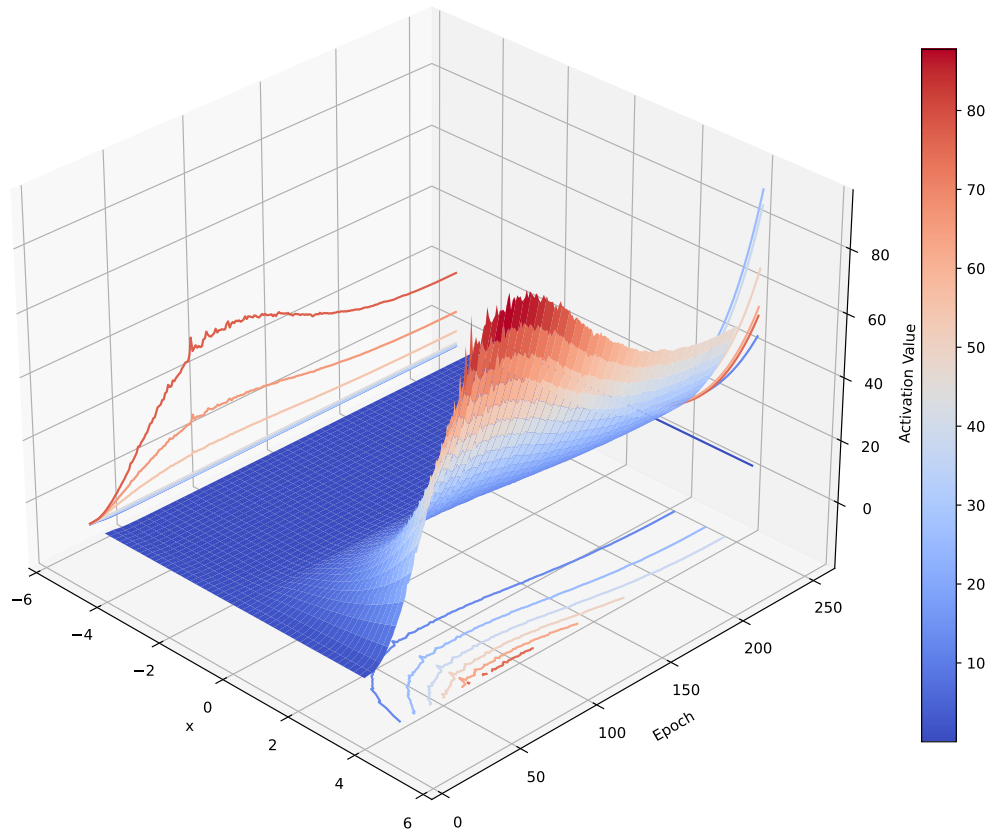


Figure A.24: Output of A_3 over time for EfficientNetB0v2 on CIFAR-100.

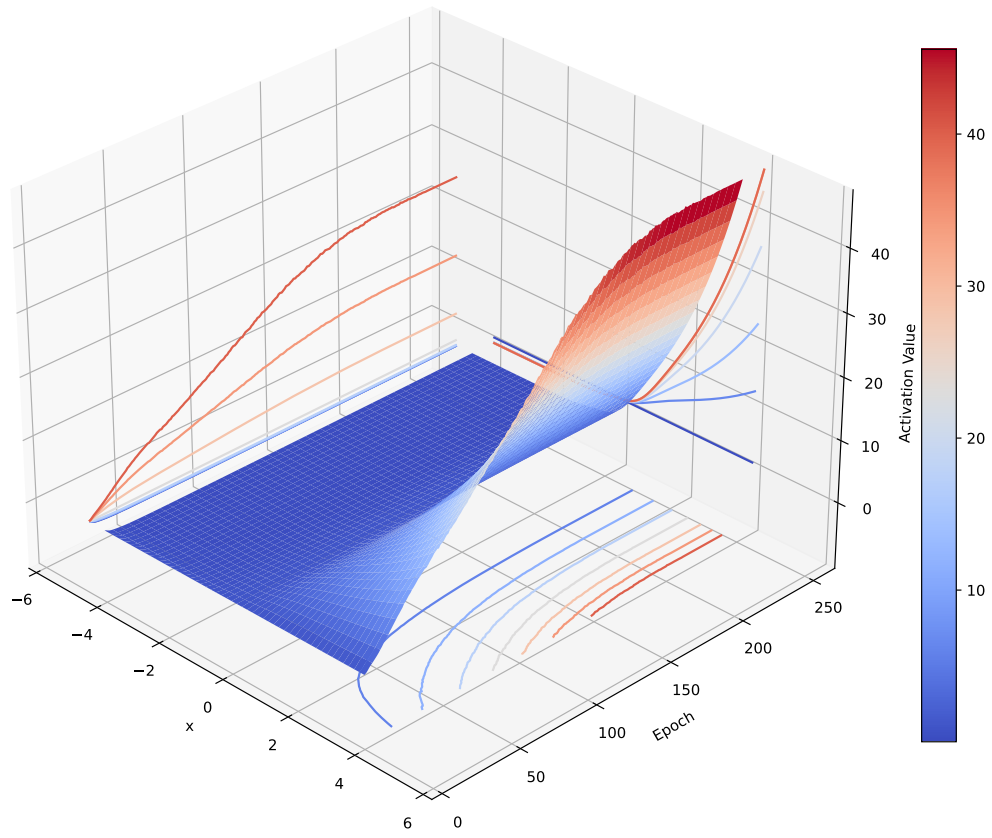


Figure A.25: Ouput of A_3 over time for ResNet52v2 on CIFAR-100.

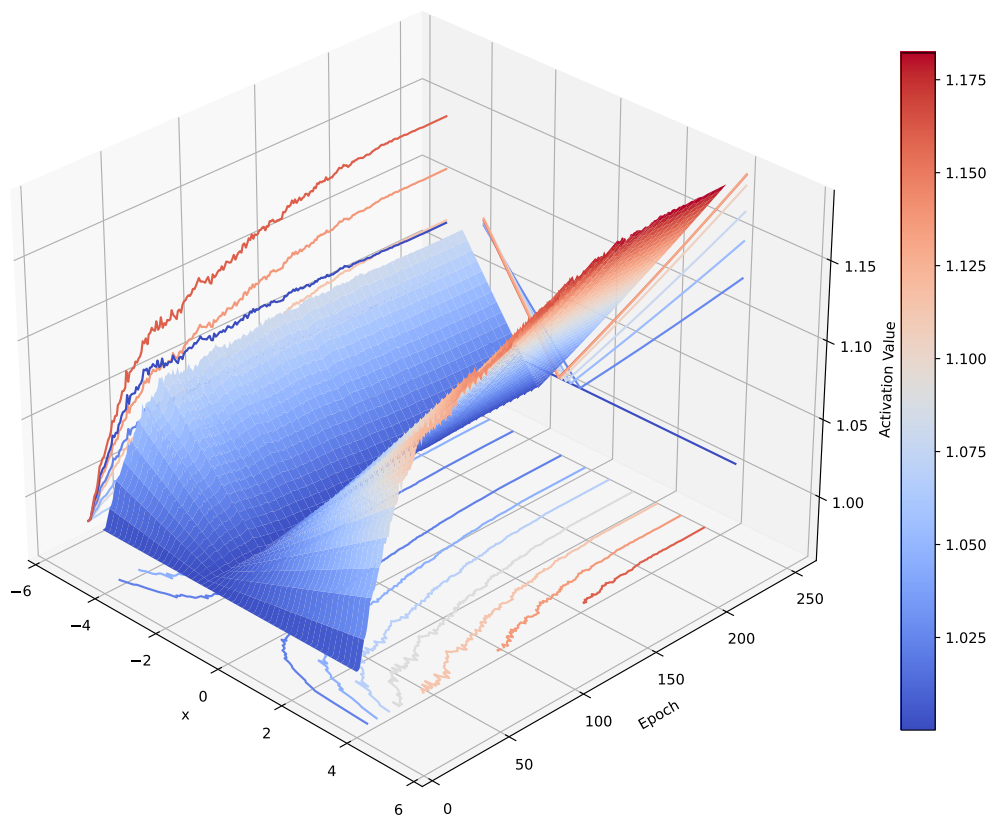


Figure A.26: Output of A_7 over time for EfficientNetB0v2 on CIFAR-10.

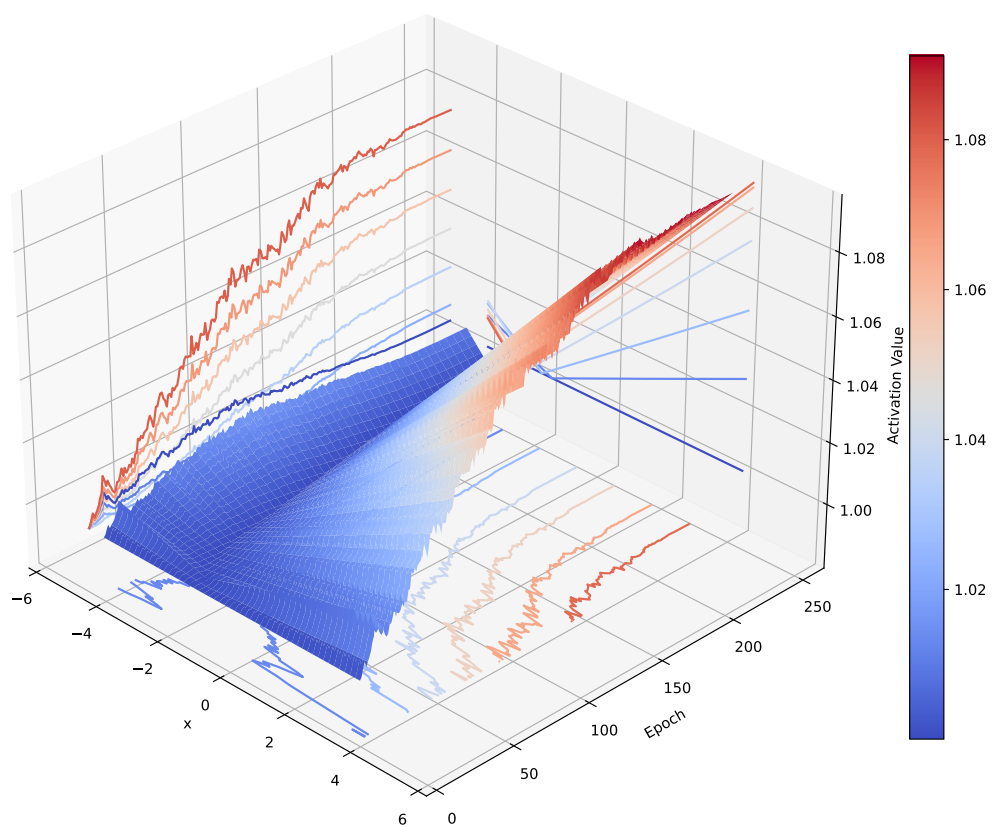


Figure A.27: Output of A_7 over time for EfficientNetB0v2 on CIFAR-100.

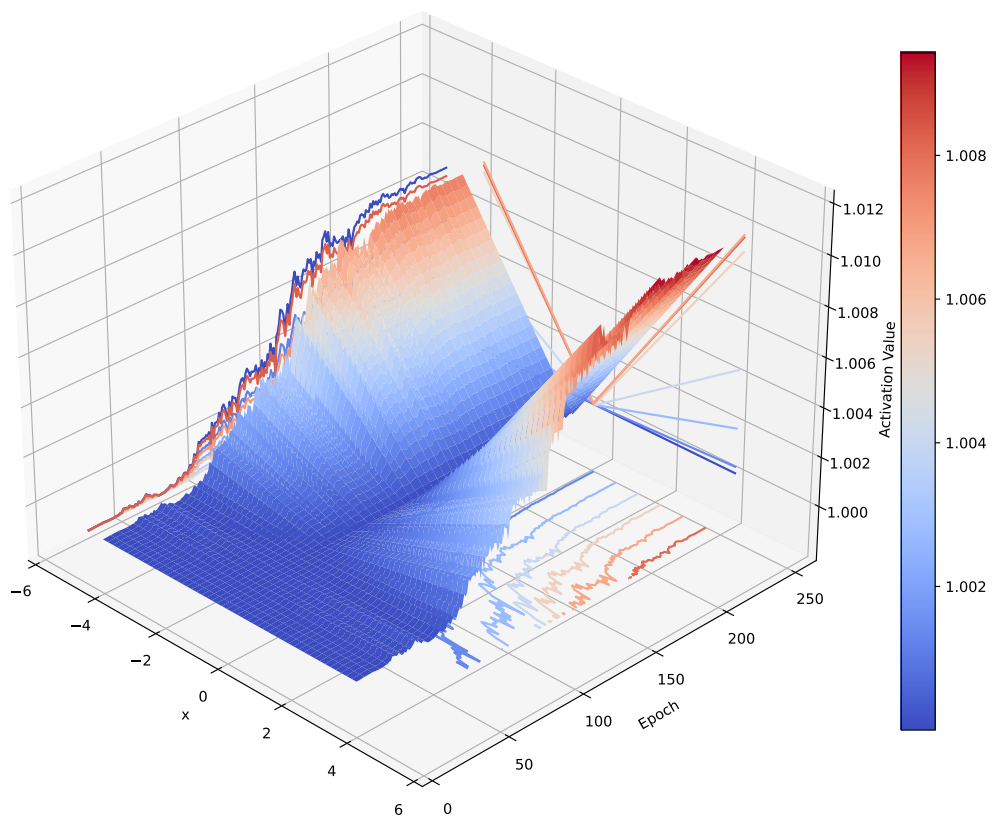


Figure A.28: Output of A_7 over time for ResNet52v2 on CIFAR-100.