4-19-2023

Loss Scaling and Step Size in Deep Learning Optimizatio

Nora Alosily
University of Missouri-St. Louis, namb2@umsystem.edu

Loss Scaling and Step Size in Deep Learning Optimization

Nora Alosily

M.S., Computer Science, University of Missouri-St. Louis, 2017
B.S., Computer Science, Qassim University, Saudi Arabia, 2010

A Dissertation Submitted to The Graduate School
at the University of Missouri-St. Louis
in partial fulfillment of the requirements for the degree
Doctor of Philosophy in Mathematical and Computational Sciences with an
emphasis in Computer Science

May 2023

To my parents, Munirah and Ibrahim.

To my husband, Abdullah.

To my daughters, Sarah and Lubna.

Acknowledgments

I am immensely grateful for the support and guidance of the following individuals, without whom this dissertation would not have been possible. From the bottom of my heart, I extend my sincere appreciation to all who have contributed to my academic and personal growth throughout this journey.

First and foremost, I want to express my gratitude to my supervisor, Professor Sanjiv Bhatia, for his unwavering support and guidance throughout my Ph.D. journey. His computer vision class opened my eyes to high-dimensional data and changed the way I think about data foreve, and he also introduced me to different math topics in an approachable, problem-solving manner. Professor Bhatia not only assisted me in enhancing my research skills but also introduced me to many valuable tools and resources. My dissertation is a result of remote collaboration, which began during the COVID-19 pandemic. I am incredibly grateful to him as a mentor striking the perfect balance between giving me the space to grow and learn independently and providing me with feedback and guidance when needed.

I would also like to express my gratitude to the members of my dissertation committee. Firstly, Professor Badri Adhikari introduced me to deep learning with his hands-on and impactful class, I would not have considered pursuing deep learning optimization otherwise. Secondly, Professor Sharlee Climer's encouragement and creation of an environment of trust, respect, and understanding were indispensable. Lastly, I am honored to have Professor Henry Kang whose invaluable suggestions that helped improve this dissertation.

I would also like to acknowledge the staff and faculty of the Department of Computer Science for providing a stimulating academic environment. Special thanks to Kimberly Stanger for her invaluable support and help, as well as the International Office, especially Lenka Garimella, for making remote collaboration possible during and after the pandemic.

Lastly, I express my heartfelt appreciation to my family and friends, whose love and support sustained me throughout this journey. I am particularly grateful for my husband Abdullah, who was always there to listen, encourage, and offer insightful advice, even during the most challenging times. I am blessed to have him in my life, and I will always cherish his role in my academic success.

Finally, I want to thank all those who contributed, directly or indirectly, to this dissertation. Your support and encouragement were invaluable, and I could not have achieved this accomplishment without you.

Notation and Symbols Guide

This section is devoted for notation convention and followed by list of symbols. It is a useful reference throughout the text. Finally, please zoom in if you find figures or charts are not sufficiently large, because they are either vector images or in high resolution.

Notation

For the sake of simplicity, I follow the notation described below in math description.

Listing

To list a *set* of n elements we write:

$$\{x_1, x_2, \ldots, x_i, \ldots, x_n\} \tag{1}$$

Or alternatively we can write it in a more compact notation:

$$\{x\}_{i=1}^{n} \tag{2}$$

Similarly, the set $\{x_1, x_2, \ldots, x_n\}$ becomes $\{x\}_1^n$. The same goes for vectors and other groups of elements. A vector of n elements is usually written as:

$$[x_1, x_2, \ldots, x_i, \ldots, x_n] \tag{3}$$

Which becomes $[x\overset{n}{\underset{i=1}{]}}$ with the compact notation.

Assignment

Left arrow $\leftarrow$ in math of this text means evaluating the right hand side variables at their current values and then assigning the result to the left hand side. This is to avoid unnecessary complicated notation. For example, instead of writing:

$$x_{t+1} = x_t - g_t \cdot x_t \tag{4}$$

We can simply write:

$$x \leftarrow x - g \cdot x \tag{5}$$

I will also try to avoid super and subscripts unless there is a real need for them, as in unfolding a recurrence. For example, it is helpful to index a recurrent items when unfolding the above:

$$x_1 = x_0 - g_0 \cdot x_0 \tag{6}$$

and

$$x_2 = x_1 - g_1 \cdot x_1 \tag{7}$$

$$x_2 = x_1 - g_1 \cdot x_0 - g_0 \cdot x_0 \tag{8}$$

Symbols

Symbol	Description
L	A layer.
i	A layer counter $\{L_0, \ldots, L_i, \ldots L_n\}$. L_0 is the input layer and L_n is the output layer.
d_i	Dimensions of layer i.
d	Input dimensions, implicit of d_0.
a	An activation or output of a neuron. a_j^i is the output of neuron j at layer i
j	A neuron counter.
A^i	Set of neurons outputs at layer i. $A^i = [a_1, \ldots, a_j, \ldots a_{d_i}]$
W_j^i	Set of weights at unit j and layer i.
k	Weight counter.
$w_{j,k}$	A single weight k at unit j .
N	Number of samples in a dataset.
n	Number of samples in a minibatch, $1 \leq n \leq N$
b	A single bias.
z	An affine function applied on the dot product of the weights and input $z_j = \sum_k^{d_{i-1}} a_k^{i-1} \cdot w_k$.
f	Model function.
$\mathcal{D}$	Dataset $\mathcal{D} = \{(x,y)_1, \ldots, (x,y)_n\} = \{(x,y)_{1:n}\}$.
$\mathcal{X}$	Set of inputs in a dataset. $\mathcal{X} = \{x_1, \ldots, x_n\} = \{x_{1:n}\}$
$\mathcal{Y}$	Set of outputs correspond to the inputs. $\mathcal{Y} = \{y_1, \ldots, y_n\} = \{y_{1:n}\}$
$\mathcal{L}$	Loss value.
θ	Set of network trainable parameters typically weights and biases.
η	Learning rate .
α	scaling factor.
η	Learning rate .
β	First moment hyperparameter.
γ	Second moment hyperparameter.

Abstract

Deep learning training consumes ever-increasing time and resources, and that is due to the complexity of the model, the number of updates taken to reach good results, and both the amount and dimensionality of the data. In this dissertation, we will focus on making the process of training more efficient by focusing on the step size to reduce the number of computations for parameters in each update. We achieved our objective in two new ways: we use loss scaling as a proxy for the learning rate, and we use learnable layer-wise optimizers. Although our work is perhaps not the first to point to the equivalence of loss scaling and learning rate in deep learning optimization, ours is the first to leveraging this relationship towards more efficient training. We did not only use it in simple gradient descent, but also we were able to extend it to other adaptive algorithms. Finally, we use metalearning to shed light on various relevant aspects, including learnable losses and optimizers. In this regard, we developed a novel learnable optimizer and effectively utilized it to acquire an adaptive rescaling factor and learning rate, resulting in a significant reduction in required memory during training.

Contents

i

Acknowledgments **ii**

Notation and Symbols Guide **iv**

1 Introduction **1**

1.1 Neural Network Model 2

1.1.1 Building Blocks 3

1.1.1.1 Classical Unit (Neuron) 4

1.1.1.2 Other Types of Units 5

1.1.2 Neural Network Topology 6

1.1.2.1 Dense Layers 7

1.1.2.2 Convolutional Layers 7

1.1.2.3 Residual Layers 8

1.1.2.4 Attentional Layers 8

1.2 Data 9

1.2.1 Data Types 9

1.2.2 Data Distribution 10

1.3 Learning Algorithm 10

1.3.1 Network Loss 12

1.3.1.1 Loss Functions 12

1.3.1.2 Loss Surface 14

1.4 Dissertation Outline 15

2 Problem Statement **16**

3 Deep Learning Optimization **18**

3.1 Gradient Optimization 20

3.1.1 Gradient Descent 20

3.1.1.1 Variants of Gradient Descent 21

3.1.2 Early Extensions to Gradient Descent 23

3.2 Implementation 25

3.2.1 Tensors 26

3.2.2 Computational Graph 26

3.2.2.1 Autodifferentiation 28

3.3 Future of Deep learning Optimization 28

3.4 Chapter Summary 29

4 Loss Scaling **30**

4.1 Loss Scaling as Step Size 30

4.1.1 Geometrical Interpretation 32

4.1.2 Experiment 33

4.1.2.1 Results and Discussion 34

4.2 Loss Scaling for Numerical Instability 35

4.2.1 Vanishing and Exploding Gradients 35

4.2.1.1 Parameter Initialization and Regularization . . . 37

4.2.1.2 Activation Functions 39

4.2.1.3 Topology 40

4.2.2 Mixed Precision Training 41

4.2.2.1 Loss Scaling in Mixed Precision Training 42

4.3 Chapter Summary 43

5 Loss Scaling for Adaptive Learning Algorithms **44**

5.1 Adaptive Learning Algorithms 45

5.2 Moment Estimation in Optimizers 47

5.2.1 First Moment Estimates 48
5.2.2 Second Moment Estimates 49
5.2.3 Integrating Both Estimates 49
5.2.4 Loss Scaling for Moment Estimates 49
5.2.4.1 Experiment 52
5.2.5 Discussion 53
5.3 Momentum Variables and Step Size 53
5.3.1 Replacing Momentum with Loss 57
5.3.2 Discussion 58
5.4 Chapter Summary 59

6 Step Size in Metalearning **60**
6.1 Metalearning 60
6.1.1 Problem Formulation 61
6.1.2 Metalearning Optimization Algorithm 63
6.2 Learned Representation 64
6.2.1 Learning A Loss 65
6.2.1.1 Meta Loss 66
6.2.1.2 Experiment 67
6.3 Learning Improved Updates 70
6.3.1 Hyperparameter Learning 70
6.3.2 Optimizer Learning 71
6.3.3 Layerwise Update Rule 72
6.3.3.1 Experimental Setup and Results 74
6.3.4 Meta Gradient Rescaling for MPT 76
6.3.4.1 Experimental Setup 76
6.4 Chapter Summary 78

7 Discussion and Future Work **79**
7.1 Summary of Dissertation 79
7.2 Challenges and Limitations 80
7.2.1 Learning Dynamics 80

7.2.2 Meta Optimizer 81
7.2.3 Technical Hurdles 82
7.3 Future Work 82

A Activation Functions **97**

B Adaptive Optimizers **98**

C Code Blocks **101**

List of Figures

1.1 A simple neural network model 4

1.2 Loss surface embedded in three dimensional space. 14

3.1 Computational graph of function $f(x; \theta) = |(\theta_1 d_1 + \theta_2 d_2) - y|$. 19

4.1 Scaling the learning rate and the loss down. 32

4.2 Scaling the learning rate and the loss up. 33

4.3 Execution times for sl vs. lr and scale factor of 0.1. 34

4.4 Execution times for sl vs. lr and scale factor of 10. 34

4.5 Vanishing gradients in shallow and deeper model 37

5.1 Loss surface that is narrow in one dimension. 45

5.2 Execution time of the models using RMSProp and scaled RMSProp. The circle signifies the model size in terms of the number of parameters. 54

5.3 Execution time of the models using Adam and scaled Adam. . . . 55

6.1 Updates of learned optimizer and optimizee. 62

6.2 Loss trajectory. 69

6.3 Performance of our layer-wise learned update. 75

6.4 Performance of our MPT_LURE learned update vs. rescaling. . 77

List of Tables

1.1 Description of neural network units. 6

3.1 List of different order tensors, the value indicates its index if multiple values exist. 27

4.1 Description of the models. 33

5.1 List of change functions in some gradient descent optimizers. . . 46

List of Algorithms

1 Batch Gradient Descent 22
2 MiniBatch Stochastic Gradient Descent 23
3 Online Gradient Descent 24

4 Optimization-Based Meta Learning 64

Chapter 1

Introduction

Most of the optimization problems have been addressed by classical hand-engineered algorithms that provide a machine with detailed step-wise guidance toward the solution. More recently, *machine learning* algorithms are being used to train an automaton on some examples to perform tasks without explicitly stating every step. Generally, the hand-engineered algorithms produce more efficient and precise solutions once the task is well understood whereas the machine learning algorithms tend to approximate; they are suitable for different kinds of tasks. Using machine learning for computationally sensitive tasks would be a waste of resources and may lead to undesirable results, especially tasks that are heavily computational in nature. Other tasks that involve pattern recognition, like recognizing a stop sign, while trivial to humans, have proved to be challenging for conventional algorithms.

We can train a learning algorithm with a number of input examples to achieve an objective and learn certain representation, in what is known as *unsupervised learning* that does not use output examples. Sometimes, however, the objective is to learn a representation that produces a specific output by providing the algorithm with the input examples as well as output examples, in what is known as *supervised learning.* In between these two end of the spectrum lies *semi-supervised learning* where we need to compensate for the missing outputs.

Another method, *reinforcement learning* does not provide explicit outputs but rather some rewards that prods the learning algorithm into the right direction.

We can also categorize machine learning algorithms according to the way they transform their input such as shallow learning and deep learning algorithms. Typical machine learning algorithms learn a task in a single layer, such that the algorithm takes an input, transforms and processes that input, and then produces an output without further processing, which is basically *shallow learning*. Examples of this kind include the support vector machines (SVM), decision trees, and linear and logistic regressors among others. *Deep learning*, on the other hand, is the process of learning a task in multiple layers, such that the model learns internally by transforming the input from one layer to the next. In other words, the output of at least one layer is not to be dispensed to the outside world, but rather injected back into the model to refine the final output. Neural networks, discussed next, are examples of deep learning. Nonetheless, any model that learns in multiple layers is considered a deep learning model [84, 116]. Our main focus in this dissertation is on neural networks, discussed next.

1.1 Neural Network Model

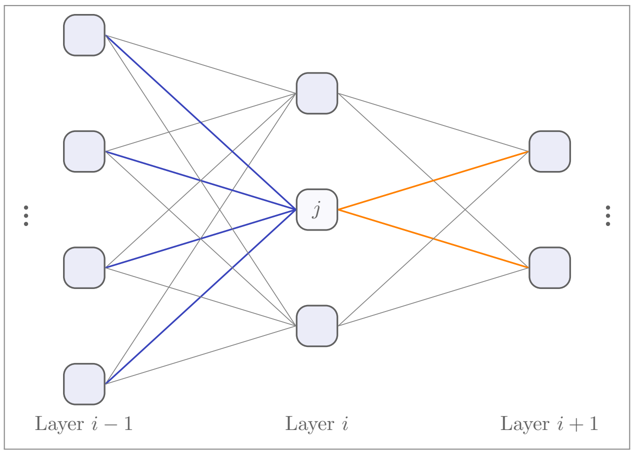
Neural network is a computational model inspired by the biological neural networks in a human brain. It performs a task using a set of interconnected artificial neurons – excitable memory structures – that transmit the right signal for a specific input that is adjusted in a learning process. Each neuron consists of set of adjustable memory units, and neurons are grouped into sets called *layers*. The input layer in the network receives the input data, and the subsequent layers transform the input using some *weights* until the output is generated at the output layer. At the end, a *loss function* compares the output to the desired output to measure the error. This comparison concludes the *forward pass*, and starts the *backward pass* in which the error is used to modify the weights. The two passes are repeated until the network *learns* the weights that minimize the error.

The learning algorithm that minimizes the error is one of three basic elements for a neural network to function properly: the model itself and how it is structured, the data to be transformed by the model, and the learning algorithm that adjusts the model to improve its computational process. We will discuss each element briefly, but the focus of this thesis is on the learning algorithm. First, the model structure or *topology* describes the way the layers of parameters are connected to each other, and how different structures transform the same data in a different manner. The second element is the training data that is provided for the model to learn from and to process in the future. It comes in various distributions; thus one model may transform various types of data differently. Finally, the *learning algorithm* dictates a mechanism to update the parameters. The same learning algorithm may be used to train different neural network topologies. For example, stochastic gradient descent algorithm may be used to train convolutional networks and recurrent networks. Similarly, the same topology may learn using different learning algorithms. We can train the convolutional nets using backpropagation, forward-forward algorithm [50] or genetic algorithms.

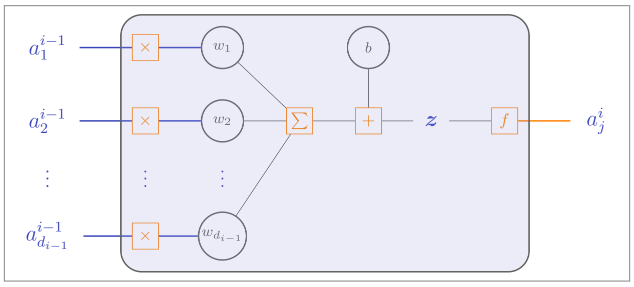
In the remainder of this section, we start with the model itself, followed by a discussion on data in section 1.2, and finally we'll introduce the learning algorithms in more details in sections section 1.3 and section 5.1.

1.1.1 Building Blocks

A neural network may be viewed as a function composition $L_n(\ldots(L_1(L_0)))$, or a mapping $L_0 \mapsto L_1 \mapsto \cdots \mapsto L_n$. Each function may be abstracted as a layer of neurons from the input layer L_0 to the output layer L_n. Neurons are the basic units that perform arithmetic operations; each neuron has its own coefficients and biases known as the *learnable weights* or *parameters* (θ). A neuron receives different inputs from other neurons, applies its parameters on the input, and transmits copies of its output to other neurons. A single layer consists of a number of neurons; this number determines the *width* of a layer. The width of the whole network is the width of its widest layer. The number of layers defines the *depth* of the network.



(a) Dense layers



(b) A simple neuron (j) at layer (i)

Figure 1.1: A simple neural network model

1.1.1.1 Classical Unit (Neuron)

The basic operations that a simple neuron can performs are weight multiplication and bias addition. For a network with n layers $\{i\}_1^n$, each layer can have a variable number of neurons $\{j\}_1^{d_i}$ in which d_i specifies the dimensions or the number of neurons in that layer. Figure 1.1a shows a set of layers $(\ldots, i-1,\ i,\ i+1, \ldots)$ that have $(\ldots, 4, 3, 2, \ldots)$ neurons respectively.

Neuron (j) in Figure 1.1a receives four inputs and thus has four corresponding weights. In the same manner, the neuron transmits copies of its output into two neurons in the following layer. The details of neuron j are shown in Figure 1.1b. Each neuron at layer i receives the same input vector A_{i-1} from previous layer, and transforms it by its weight vector $W_j = [w]_1^{d_{i-1}}$ using the dot product operation:

$$a_j = A_{i-1}^T W_j = \sum_{k=1}^{d_{i-1}} a_k \cdot w_k \tag{1.1}$$

This is optimized in practice by using matrix multiplication to compute outputs of the same layer simultaneously, which is as follows for Figure 1.1a:

$$A_i = A_{i-1}^T \begin{bmatrix} W_1 & W_2 & W_3 \end{bmatrix}_i \tag{1.2}$$

$$\begin{bmatrix} a_1 \\ a_2 \\ a_3 \end{bmatrix}_i = \begin{bmatrix} a_1 & a_2 & a_3 & a_4 \end{bmatrix}_{i-1} \begin{bmatrix} w_{1,1} & w_{2,1} & w_{3,1} \\ w_{1,2} & w_{2,2} & w_{3,2} \\ w_{1,3} & w_{2,3} & w_{3,3} \\ w_{1,4} & w_{2,4} & w_{3,4} \end{bmatrix}_i \tag{1.3}$$

The linear combination in Equation 1.1 is able to produce a hyperplane boundary on the input vector. Adding a bias b to the linear combination makes the boundary more powerful because the hyperplane does not necessarily intersect with the origin. Finally, the expressiveness of the affine function $z = A^T W + b$ is improved by applying an *activation function*.

The activation function may be used to change the shape of the hyperplane, the term activation comes from the original function used – the threshold function where the input is activated or not. Some of the widely used functions are the logistic, the rectifier, the hyperbolic tangent, and the identity functions; please check Appendix A for more details on those functions. If the logistic function is used, a single neuron is viewed as a logistic model.

1.1.1.2 Other Types of Units

For the rest of this dissertation, we will refer to a neuron as a *unit* to generalize to different kinds of cells and to adhere to the terminology of the field. There are different types of units that can be used to compose a layer other than the classical unit. Examples include *bilinear* unit, *lstm* unit, and *gru* unit, which are listed in Table 1.1 below.

Unit	Input	Transformation	Parameters
Linear	x	$y = x^T W + b$	$\theta = \{W,\, b\ \}$
Biinear	x	$y = W_1 x^T W_2 + b$	$\theta = \{W_1,\, W_2,\, b\ \}$
RNN	x, y	$g = x \cdot W_g + b_g$ $h = y \cdot W_h + b_h$ $y \leftarrow tanh(g + h)$	$\theta = \{W_*,\, b_*\ \}$
GRU	x, y	$h = \sigma(W_h \cdot x + W_{h,y} \cdot y + b_h)$ $g = \sigma(W_g \cdot x + W_{g,y} \cdot y + b_g)$ $o = W_{o,y} \cdot y + b_o$ $c = tanh(W_c \cdot + b_{c,x} + h \odot o)$ $y \leftarrow (1 - g) \odot c + g \odot y$	$\theta = \{W_*,\, b_*\ \}$
LSTM	x, y, c	$i = \sigma(x \cdot W_i + y \cdot W_{i,y} + b_i)$ $h = \sigma(x \cdot W_h + y \cdot W_{h,y} + b_h)$ $g = \sigma(x \cdot W_g + y \cdot W_{g,y} + b_g)$ $o = \sigma(x \cdot W_o + y \cdot W_{o,y} + b_o)$ $c \leftarrow h \odot c + i \odot g$ $y \leftarrow o \odot tanh(c)$	$\theta = \{W_*,\, b_*\ \}$

Table 1.1: Description of neural network units.

In the table, different activations can be added to linear layers to produce non-linear transformation. Moreover, existed activation can be changed. Finally, the symbol $\odot$ signifies the Hadamard or element wise product, and $y = f(x)$. Now that have introduced the basic units, we can consider how these building blocks are grouped and connected in a certain way to form various types layers.

1.1.2 Neural Network Topology

A neural network is composed of one or more *layers* such that the neurons in a layer operate on the input and produce outputs. In practice, a layer is composed of the same type of units, but different types of connections between layers allow the input to be transformed differently. The flow of transformation can be summarized in two basic types: the *recursive neural networks* that allow the flow to go back to the same layer or to some previous layers, and the *feedforward networks* that allow the flow to proceed in just one direction. In both types of flow, there are different kinds of topologies, or ways to connect the layers. Different types of layers may coexist in the same architecture, and examples are described below.

1.1.2.1 Dense Layers

One of the basic types is the dense topology, which means fully connecting one layer to another. The other layer could be a previous, the same, or next layer in the network. This type of network usually operates on vector input, thus the input needs to be converted into a vector, or flattened, beforehand. One of the first examples in a feedforward network was implemented in 1986 to predict people in family trees [95]. This architecture is prevalent to this day and usually coexists with other types in a single network. The layer can be expressed mathematically as a function multiplication with the result of input layer:

$$Dense \quad g(f(x)) = g \cdot f \tag{1.4}$$

The fact that dense layers throw away some of the embedded information, such as spatial interdependence, motivated the development of convolutional neural networks.

1.1.2.2 Convolutional Layers

Convolutional layers are yet another widely-used topology but for aligned information like pixels in images, and they are the basic element in *convolutional neural networks* (CNN). The convolutional layers consist of trainable filters, or simply a set of shared weights repeatedly aligned and applied on the input. Such a network was used to recognize handwritten digits in 1989 [64]. However, the pattern recognition community shifted towards better performing and more efficient methods that are tailored to the task in hand such as Support Vector Machines (SVM). Since 2010, the convolutional networks, among other neural structures, have regained interest due to their outstanding performance on tasks that have proved to be challenging [23, 47, 62, 83]. The layer can be expressed as a cross correlation with the result of input layer:

$$Convolution \quad g(f(x)) = g * f \tag{1.5}$$

1.1.2.3 Residual Layers

Another interesting type of topology is *residual layers*, in which the output of one layer is added to another layer while possibly skipping some layers in between. Although simple, this type of topology is powerful in transforming the input, and its simple way to mitigate the gradients problems utilizing addition instead of multiplication which injects the gradients differently. It was used for image classification in 2015 [46]. Since then, the architecture has become ubiquitous as the backbone of many different architectures. The layer can be expressed as an addition between two layers:

$$Residual \quad g(f(x)) = f + g \tag{1.6}$$

1.1.2.4 Attentional Layers

More recently, advances in various field have been achieved with the use of attention mechanism, and generally, it aims at selecting the relevant signal for further processing [98]. An attention block or layer is usually encapsulated in a transformer module that also contains some dense or other types of layers, and it first appears in transformers [112], and later help Generative Pretrained Transformer (GPT) models, among others to have superior performance [59, 65, 73]. We partially adapt the mathematical notation introduced by [45]:

$$\textit{Attention} \quad g(f(x)) = g(f(x), h(x)) \tag{1.7}$$

There are other types of units and layers, so please refer to [40, 102] for a complete overview of neural networks. In the remainder of this section, I introduce data briefly and then I will discuss the learning aspects of neural networks, which is the concern of this dissertation.

1.2 Data

Data is the input and the center of interest to a deep learning model, and a group of data examples forms a *dataset*. The instances of data in a dataset are also called *samples*, *observations*, *records*, and *data points*. For instance, the Iris flower dataset [30] is a widely used dataset since 1936. It contains 50 samples from three species of Iris giving a total of 150 data points; each data point has 5 attributes which are {Iris species, sepal length, sepal width, petal length, and petal width}. The attributes in a dataset are also called *dimensions*, *variables*, and *features*. The number of data points vary from few points like in Iris dataset to almost infinite as done in training big models such as GPT, which are trained on massive amount of text data culled from billions of documents and internet pages.

1.2.1 Data Types

Data appears in different forms like text, images, and videos. Nonetheless, they are eventually represented with discrete numbers. After all, the difference is in terms of their dimensions, value range, and value granularity. The factors are subject to preprocessing choice, but they affect the performance tremendously. Generally, the values are scaled and shifted to maintain a balanced flow while being transformed by the network.

1.2.2 Data Distribution

If the data-generating function (distribution) is known, it is easy to characterize the data generated by the distribution. However, data distribution is the underlying problem that machine learning algorithms endeavor to solve. By observing instances of data, the algorithm can capture some aspects of the true distribution to solve for unseen data, and I quote:

> "*Machine learning is about capturing aspects of the unknown distribution from which the observed data are sampled.*" [2]

Therefore, data provided while training should reflect the distribution of data in a real application, or at least data presented at test time. Moreover, providing sufficient amount of balanced data reduces the gap in the samples from the distribution. Thus, correct sampling is important to avoid bias toward part of the data. *Oversampling* and *undersampling* techniques may help, in which oversampling methods aim at populating more samples in regions that have sparse data while undersampling entails choosing data points for removal in parts that are densely sampled.

1.3 Learning Algorithm

Learning algorithm is an algorithm that improves the weights to perform a task after observing relevant data. Many algorithms meet the criteria, but the dominant and widely used algorithm to train neural networks is backpropagation [95]. Although it has limitations, the algorithm is relatively efficient and is able to converge to satisfactory results.

Generally, the process of learning is performed in two steps: the forward pass and the backward pass. The *forward pass* is fairly simple where the parameters

θ in a neural network layer are applied to the input A_0, producing some output. The output is judged by a function called the *loss function* that compares the network output $\hat{y}$ to the desired output y. The amount of deviation between the two outputs called the loss or error, with the goal to minimize the error; the optimal loss is always zero. The goal of the learning process is to find parameters θ that minimize the loss. Once the loss is known, the parameters are adjusted by the learning algorithm in the *backward pass* to minimize the error.

One way to minimize the error is by gradually adjusting the weights to predict better outcome in the next iteration, normally achieved by the *gradient descent* algorithm. Gradient descent iteratively provides information on the descent direction of a function using first-order derivatives until it reaches a local minima. Since the network is a composition of nested functions and each function may contain thousands or millions of weights, it is important to compute the derivatives in the big chain rule efficiently. This is achieved by the *backpropagation* algorithm that exploits the redundant computations at each layer to efficiently update the weights [96, 102].

Backpropagation updates the parameters by abstracting the network as a function and tries to minimize the error with respect to the network parameters using the chain rule [115]. The method uses *automatic differentiation* which is a dynamic programming algorithm that finds derivative of a given function by memorizing differentiation steps in order to avoid redundant computations. This property is especially useful in neural networks that require a lot of such computations. Derivatives needed at a layer are identical but differ from the preceding layer. There are two modes to accumulate the derivatives: one is *forward accumulation* which finds the derivative of every function with respect to a given input, and the other is *backward accumulation* that finds the derivatives of an output with respect to every function. In the backward pass, a neural network uses the backward accumulation to find the derivative of the loss with respect to each parameter. The loss function is described in the next subsection, and gradient descent optimization is discussed in more depth in chapter 3.

1.3.1 Network Loss

It is important to discuss the loss function $f(\hat{y}, y)$ and the loss value $\mathcal{L}$ produced by it, before introducing the gradient descent algorithm that starts with the loss. In supervised learning, the loss function takes two inputs: the desired network output y and an actual network output $\hat{y}$, and produces a loss value that measures the error of the network. The larger the loss value, the more change is needed to adjust the parameters. The choice of an appropriate loss function is critical because of its impact on the whole network. Below I will discuss some of the common loss functions.

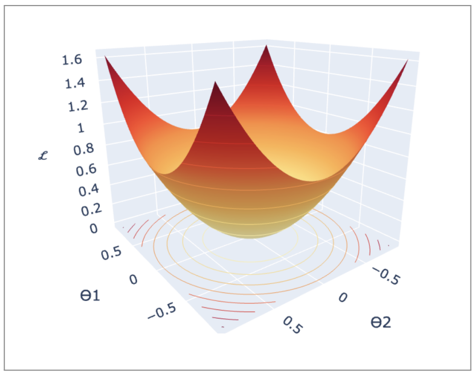
1.3.1.1 Loss Functions

Different types of outputs require different ways to measure the error. One way is to use the binary accuracy where the function outputs true if y and $\hat{y}$ match and false otherwise. Although it provides a straightforward measure of accuracy, there are two downsides to such a solution. First, some tasks require approximate solutions such as networks that generate inexact output like images. The generated output does not need to exactly match the desired output. Moreover, such a measure does not give extra information on the quality of accuracy. We may get the same accuracy from two models, one that closely approximates that desired output and the other that is very different. Such information is useful, because it provides a quantity for future improvement. The loss function needs to provide a specific value to characterize the extent of deviation, and the optimizer uses that value to appropriately adjust the network parameters. We can categorize the loss functions into two categories according to the model task: discriminative loss functions and generative loss functions.

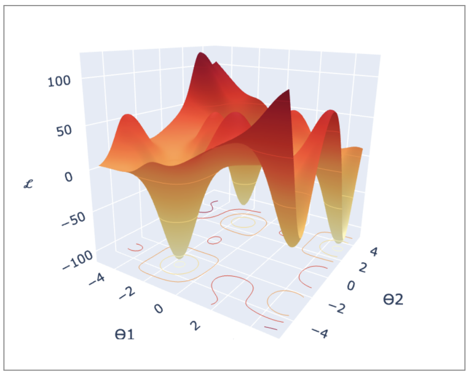
In a discriminative learning in which the goal is to learn a distribution boundary, the network learns to produce an output given some input, as happen in classification and regression. In classification, the networks is expected to produce a probability of each class $P(y|x)$, and the model learns to increase the probability of the correct class. A special case is a binary classification where there is only a single class and it can be on or off. The vector of predicted outputs is usually computed by measuring the membership probability of the output to the assigned class $\hat{y} = \{\hat{y}_1 = P(\hat{y}_1|x),\ \hat{y}_2 = P(\hat{y}_2|x),\ \ldots,\ \hat{y}_c = P(\hat{y}_c|x)$. The cross entropy function is used to measure the loss $-\sum_{k=1}^{c} y_k \log(\hat{y}_k)$. Other loss functions include hinge loss, focal loss, and relative entropy [4, 34, 58, 69]. In regression, on the other hand, the network is expected to produce a continuous-valued possibly multidimensional, output. The model learns to minimize the residual, the discrepancy between its output and the correct output. There are plenty of choices, such as mean absolute error $L1 = |y - \hat{y}|$, mean square error $L2 = \sqrt{(y - \hat{y})^2}$, and Huber loss [55]. The two most used loss functions are $L1$ and $L2$.

In generative learning, in which the goal is to learn a probability distribution and generate outputs given some input and random seeds, loss functions can get fairly complex. For example, in generative adversarial networks, where the model is trained to generate deceptive data such as a picture of people, the model concurrently uses a generative and a discriminative network [41]. The training proceeds by providing the discriminative network with two inputs: one is real and the other produced by the generative network. The discriminative network then tries to guess which one is real and which is not. The generative network then learns how to optimize its performance based on the discriminative feedback. Such a system uses two loss functions, one for each network, in more advanced models like GPTs (Generative Pretrained Transformer) the number of loss functions increase,

Plotting the generated loss as a value of the network parameters function produces a hypersurface called the *loss surface*. Unless the number of network parameters is small, it is impossible to visualize the exact loss surface, especially as the number of parameters can easily reach millions in modern architectures. Nonetheless, visualizing the loss surface can be an invaluable resource to understand the changes as the network grows in width and depth.



(a) Quadratic function



(b) Bird function [57]

Figure 1.2: Loss surface embedded in three dimensional space.

1.3.1.2 Loss Surface

We can define the loss surface as a hypersurface of the network parameters θ embedded in an ambient space of both the parameters and the loss value. In simpler terms, it is the surface connecting the values from the loss function into the parameter space. Each loss point on the loss surface is an averaged loss given input data with a fixed set of parameters. Two examples of loss surface are shown in Figure 1.2. Loss surface of smooth and convex quadratic function $f(\theta) = \theta_1^2 + \theta_2^2$ is shown on the left panel, and the right panel shows a non-convex surface of a function called bird function $f(\theta) = (\theta_1 - \theta_2)^2 + e^{1-sin(\theta_1)^2} \cdot cos(\theta_2) + e^{1-cos(\theta_2)^2} \cdot sin(\theta_1)$

The knowledge of the loss surface of a given network and dataset obviates the need for training. In such a case, finding best parameters would be a problem of finding the minimum value of the surface, a way simpler problem. The purpose of training is to find the loss and then tune the parameters according to that loss. Thus, the loss surface is uncharted unless the parameters are computed in a brute force manner, so when referring to the loss surface it is actually a sampling of the surface.

Only under strict assumptions, the loss surface can be convex where the function is monotonically decreasing everywhere except for one location – the global minima [15], as in Figure 1.2a. Otherwise, the loss is *non-convex* implying that there are multiple local minima, as in Figure 1.2b. At a local minima, the value of the loss surface is less than its value at neighboring points, but there are multiple locations with such a dip in a non-convex surface. It makes optimization harder because any movement out of the local minima increases the loss, and there is no indication on where to go. Sometimes, however, finding a local minima is adequate and methods such as gradient descent can be used.

1.4 Dissertation Outline

Now that we have review the preliminaries for deep learning, the rest of this dissertation is structured as follows:

- In chapter 2, I will introduce the problem statement.
- In chapter 3, we will introduce deep learning optimization and we establish a generation notation for optimization. We also introduce and review some necessary concepts that are used in deep learning optimization such as computational graph and autodifferentiation.
- In chapter 4, we propose scaling the loss in plain gradient descent. Then we present the empirical results that support the proposition. We point out mixed precision training as a method that is closely related but does not fully utilize scaling the loss and we provide the recommendation to use loss scaling to the fullest extent.
- In chapter 5, we provide a literature review on adaptive learning algorithms, and we continue to propose to replace some the hyperparamters with loss scaling. Then we show the results that show improvement that can be gained by the application. We also show that the variables in common optimizers cannot be modified as a function of the loss.
- In chapter 6, we introduce and provide a literature review on metalearning and we use it for various loss scaling and step size schemes.

Chapter 2

Problem Statement

While backpropagation provides an efficient solution for training neural networks, the training itself requires significant amount of time and resources, even with the most advanced hardware. Recent developments achieved by large models require days, and even weeks worth of training time, and naturally, more resources [80]. We address the problem of efficiency in training the neural net by focusing on the step size.

In deep learning optimization, the learning rate is the main determinant of the step size. It is the scale of the future change. Once the gradients are computed from the loss, learning rate is applied to scale each parameter update. Historically, models are small and have only a few parameters. Adding a few multiplications to update these parameters does not have a big impact. However, recent advances in deep learning have led to the creation of models with billions of parameters, and now, each computation step to update these parameters counts.

There is one factor in these parameter updates that is rarely looked at, which is loss scaling. Loss scaling and learning rate are theoretically equivalent in gradient descent. However, unlike learning rate that is applied on each parameter, loss scaling is applied only once to the loss. The two equivalent expressions mathematically diverge in application.

Loss scaling is certainly less intuitive and it may look deceptively useless, but it can be extremely advantageous. In fact, loss scaling has been used only recently in the context of mixed precision training and has never connected explicitly to the step size. Thus, combining loss scaling and learning rate results in redundant application of the step size once in the form of the loss, and at another time in the form of the learning rate.

My hypothesis is that I can speed up the computation by utilizing loss scaling instead of the learning rate, without any significant impact on deep learning optimization.

Throughout this dissertation, I will introduce loss scaling as yet another factor that determines the step size. I will present various ways to utilize the loss scaling to save redundant computations. I will provide proofs on the process to properly scale the loss in various optimizers with experimental results. Finally, we show the importance of the loss and its scale in different meta learning algorithms. I'll evaluate the performance of my hypothesis by comparing regular optimizers to counterparts that use gradients from loss scaling and apply our derived updates, while fixing all other variables. We will measure the time enhancement while making sure that the accuracy is not degraded. In metalearning experiments, we will closely examine the loss trajectory and benchmark our hypothesis against other well-known methods. Finally, we will present findings to draw our final conclusion.

Chapter 3

Deep Learning Optimization

The process of optimization entails improving a set of variables to minimize a cost or maximize a reward. The two objectives can be solved by the same solver with trivial conversion; thus addressing just the minimization problems with learnable weights θ will suffice:

$$\arg\min_{\theta} \quad f(\theta) \tag{3.1}$$

A function f represents the neural network model composition g, ending with the loss function, parameterized by learnable θ, and receives inputs x at different levels, including the expected output at the loss function:

$$f(x;\theta) \tag{3.2}$$

I show the example of such a function in Figure 3.1 in which $x = \{d_1, d_2, y\}$ and $\theta = \{\theta_1, \theta_2\}$. Typically, input is not just one example but rather a dataset that it is split for training and testing $\mathcal{D} = \mathcal{D}^{\text{tr}} \cup \mathcal{D}^{\text{ts}}$, and optimally $\mathcal{D}^{\text{tr}} \cap \mathcal{D}^{\text{ts}} = \emptyset$. The model at training time is therefore:

$$f(\mathcal{D}^{\text{tr}};\theta) = f(y^{tr}; g(x^{tr};\theta)) \tag{3.3}$$

There is a wide range of optimization algorithms that find θ^*, the values of θ that minimize f. Subsequently, the model is tested using the minimized parameters:

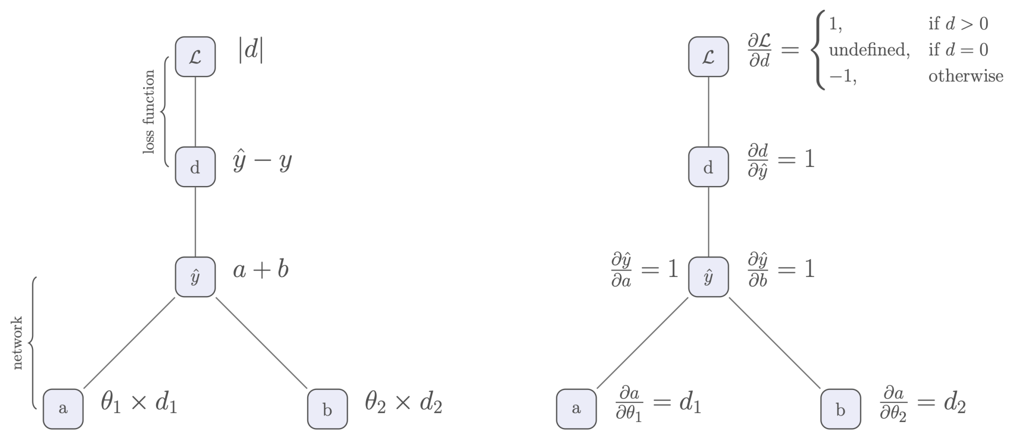
$$f(\mathcal{D}^{\text{ts}};\theta^*) = f(y^{ts}; g(x^{ts};\theta^*)) \tag{3.4}$$

Finally, the model is used as:

$$g(x^{\text{real data}};\theta^*) \tag{3.5}$$

An iterative optimization procedure will update the learnable parameters at each time step according to a function u of the previous parameters and possibly some other parameters, and parameterized by the optimizer's parameters ϕ :

$$\theta \leftarrow u(\theta, \cdot\ ; \phi) \tag{3.6}$$



$\mathcal{L}$
$|d|$
loss function
d
$\hat{y} - y$
$\hat{y}$
$a + b$
network
a
$\theta_1 \times d_1$
b
$\theta_2 \times d_2$
$\frac{\partial \mathcal{L}}{\partial d} = \begin{cases} 1, & \text{if } d > 0 \\ \text{undefined}, & \text{if } d = 0 \\ -1, & \text{otherwise} \end{cases}$
$\frac{\partial d}{\partial \hat{y}} = 1$
$\frac{\partial \hat{y}}{\partial a} = 1$
$\frac{\partial \hat{y}}{\partial b} = 1$
$\frac{\partial a}{\partial \theta_1} = d_1$
$\frac{\partial a}{\partial \theta_2} = d_2$

(a) Computational graph. **(b)** Partial derivatives at each node.

Figure 3.1: Computational graph of function $f(x;\theta) = |(\theta_1 d_1 + \theta_2 d_2) - y|$.

3.1 Gradient Optimization

Gradient optimization is an iterative process that uses gradients to guide the minimization or the maximization of a function. The maximization is known as *gradient ascent* while the minimization is known as *gradient descent*. In both cases, the update function from Equation 3.6 is a function that receives the model parameters θ at a certain step, their gradients from the loss g, and is parameterized by the optimizer parameters ϕ:

$$\theta \leftarrow u(\theta, g; \phi) \tag{3.7}$$

The minimization in gradient descent is defined by moving the parameters in the opposite direction of the change function Δ:

$$u(\theta, g; \phi) = \theta - \Delta(g; \phi) \tag{3.8}$$

The above equation summarizes gradient descent algorithms. Various algorithms process the gradients g and apply their parameters ϕ differently. We discuss gradient descent in more detail below and then we move to other adaptive learning algorithms in section 5.1. But you can quickly review examples of different change functions presented in Table 5.1.

3.1.1 Gradient Descent

Gradient descent has been known to optimize functions toward local minima by moving the function parameters in the direction of the steepest descent. This is achieved using first order derivatives, the gradients g, of the loss with respect to each of the function parameters.

$$g_i = \frac{\partial \mathcal{L}}{\partial \theta_i} \tag{3.9}$$

The change function in gradient descent is just the gradient parameters being updated to minimize the loss or gradient descent as:

$$\begin{aligned} \Delta &= g \\ \theta &\leftarrow \theta - g \end{aligned} \tag{3.10}$$

However, this update is rarely used; the gradient magnitude is altered by a variable called the *learning rate* η. Learning rate determines the size of a movement in the direction of the steepest descent. The modification of gradient descent is given by:

$$\begin{aligned} \Delta &= \eta \cdot g \\ \theta &\leftarrow \theta - \eta \cdot g \end{aligned} \tag{3.11}$$

Before continuing the discussion on the learning rate, we need to briefly describe the computation of the exact gradients from a number of presented examples.

3.1.1.1 Variants of Gradient Descent

The number of examples presented at each update (n) changes the behavior of learning because the exact gradient computation is subject to that number:

$$\begin{aligned} g &= \frac{1}{n}\sum_{i=1}^{n} \nabla_\theta f(x_i;\theta) \\ &= \frac{1}{n}\sum_{i=1}^{n} \nabla_\theta \mathcal{L}_i \\ &= \frac{1}{n}\nabla_\theta \sum_{i=1}^{n} \mathcal{L}_i \end{aligned} \tag{3.12}$$

In *batch gradient descent*, n equals the size of the whole training dataset, while it represent one example in *stochastic gradient descent.* In between comes *minibatch gradient descent*; all are presented next with some details.

Batch Gradient Descent Algorithm The standard notion of gradient descent, sometimes called batch gradient descent, implies that the dataset is fixed. The procedure starts by considering entire data at once, then their gradients are averaged, and finally the parameters are updated. The algorithm achieves incremental improvement by repeating the procedure iteratively on the same dataset in passes called *epochs*. It is presented with details in (algorithm 1).

However, with the aim at generalization, we need large amount of data, and it is impractical to process all data before making a single update. Moreover, using all data at once For this reason, the batch gradient descent algorithm has been replaced with an extension that deals with data in partial batches, the stochastic gradient descent algorithm.

Algorithm 1: Batch Gradient Descent

```
Input : Model, $f(\theta)$
        Dataset, $\mathcal{D} = \{(x, y)\}_1^N$
        Stop, stopping criterion
1 while not stop do
2     $\mathcal{L} = \text{forward-pass}(f, \mathcal{D})$
3     $\nabla\theta = \text{compute-gradients}(f, \mathcal{L})$
4     $g \leftarrow \frac{\nabla\theta}{N}$
5     $\theta \leftarrow \theta - \eta g$
6 end
```

Stochastic Gradient Descent Algorithm SGD (algorithm 2) refers to a variation of gradient descent where the parameter update is performed based on a random subset of data, with the assumption that this subset is well representative of the whole dataset distribution.

Online Gradient Descent Algorithm Sometimes, training is performed in an active environment where data continuously evolves either by addition of more data or by modification of existing data. Examples of such environment include the text over the internet and real world space to train the robots. The size and changing nature of the dataset do not allow for looking back at the same examples. As long as the evolving data are good estimate, the change in data can be quite advantageous because it reduces overfitting.

Algorithm 2: MiniBatch Stochastic Gradient Descent

```
Input : Model, $f(\theta)$
        Dataset, $\mathcal{D} = \{(x, y)\}_1^N$
        Stop, stopping criterion
        Batch size, $n$
1  batches ← $\lfloor \frac{N}{n} \rfloor$
2  while not stop do
3      $\mathcal{D}$ ← shuffle()
4      foreach batch i in batches do
5          $\mathcal{D}_{mini}$ ← sample_batch($\mathcal{D}, i, n$)
6          $\mathcal{L}$ ← forward-pass($f, D_{mini}$)
7          $\nabla\theta$ ← compute-gradients($f, \mathcal{L}$)
8          $g \leftarrow \frac{\nabla\theta}{n}$
9          $\theta \leftarrow \theta - \eta g$
10     end
11 end
```

3.1.2 Early Extensions to Gradient Descent

Taking big steps results in learning faster but it may also result in overshooting. On the other hand, taking small steps converges slower and is more likely to get stuck in some relatively poor local minima. We quantify the movement towards optimal loss surface by a sign and a magnitude. The sign signifies the direction of the movement and the magnitude specifies the amount of movement to be applied to the parameter. Some learning algorithms avoid the use of the magnitude completely and only focus on the direction of minimization. An early example of such learning is Rprop where the modification of the parameters depends on the history of the gradient signs [93] as follows, c_s is between zero and one, and c_b is

Algorithm 3: Online Gradient Descent

```
Input : Model, $f(\theta)$
        Dataset, $\mathcal{D} = \{(x, y)\}_1^\infty$
        Stop, stopping criterion
        Batch size, $n$
1 while not stop do
2     $\mathcal{D}_{mini} \leftarrow$ sample_batch($\mathcal{D}, n$)
3     $\mathcal{L} \leftarrow$ forward-pass($f, D_{mini}$)
4     $\nabla\theta \leftarrow$ compute-gradients($f, \mathcal{L}$)
5     $g \leftarrow \frac{\nabla\theta}{n}$
6     $\theta \leftarrow \theta - \eta g$
7 end
```

more than one $0 < c_s < 1 < c_b$:

$$\begin{aligned} \Delta &= \text{sign(s)} \cdot v \\ s &= g * h \\ v &\leftarrow \begin{cases} v * c_b & \text{if } s < 0 \\ v * c_s & \text{if } s > 0 \\ v & \text{otherwise} \end{cases} \\ h &= \begin{cases} g & \text{if } s > 0 \\ 0 & \text{otherwise} \end{cases} \end{aligned} \tag{3.13}$$

The use of gradients history, both sign and magnitude, can be tracked back to momentum. *Momentum* is a way to minimize the error by considering not only the current gradient values but also the history of previous gradients. The history acts as a velocity of a ball in a curvature that captures the dimensions where the motion (gradients) swings. The minimization will be larger in dimensions where consecutive gradients point to the same direction and smaller in dimensions where gradients oscillate from one direction to another [88]:

$$\begin{aligned} \Delta &= \eta \cdot s \\ s &\leftarrow \beta \cdot s + g \end{aligned} \tag{3.14}$$

A slight modification can improve the performance by a lot which is exploited in the case of Nesterov Accelerated Gradient (NAG) [81, 110]. The change uses the current momentum position as a base for gradient movement, which results in more regular movement of the parameters:

$$\begin{aligned} \Delta &= \eta \cdot v \\ v &= g + \beta \cdot s \\ s &\leftarrow \beta \cdot s + g \end{aligned} \tag{3.15}$$

I introduce the basic gradient descent algorithm, and the use of the first and second moments is discussed in more details in section 5.1. In the next section, I present how deep learning is implemented in practice, with a focus on relevant concepts.

3.2 Implementation

The success of deep learning is attributed to four reasons. First is the availability of extremely large datasets. Problems such as the ones encountered in complex image recognition require lots of data, and such data was not available until the 2000s. Second is the development and use of hardware that support Single Instruction Multiple Data (SIMD) operations. For deep models, big memory and fast processors have been necessary to accommodate the large number and speed of computation as well as the data needed. The use of GPU has helped achieve a milestone in image recognition in 2012 [62]. Since then, there have been many advances in hardware that support vector, matrix, and tensor operations. Third is the advances made in training algorithms. Different activation functions, regularization methods, and optimizers have helped in deep learning. Finally, well-tested libraries help accelerate the research in deep learning, especially that the field requires various computations whose implementation is error-prone and time consuming.

There are many open source libraries for deep learning, and the two most notable are TensorFlow by Google and Pytorch by Meta [1, 86, 87]. The implementation in these two among others is characterized by the use of multidimensional arrays, the *tensors* in mathematics, and derivatives computation or *autodifferentiation*.

3.2.1 Tensors

A tensor is a multidimensional array that represents a multidimensional object of numerical values. It is a generalization of scalars, vectors and matrices to higher dimensions. While we can list The *order* of a tensor is its rank which represents the number of dimensions of the tensors, each dimension is accessed by a single index. For example, a scalar is a tensor of order zero, a vector is is a tensor of order one, a matrix is a tensor of order two and so on. Each single order or rank has n Higher-order tensors have more three or more dimensions and there is no formal way to list them numerically. We show examples of tensors in the Table 3.1 below. We adapt the listing for higher-order tensors form NumPy library, we also adapt its row vector representation.

3.2.2 Computational Graph

Graphs are a powerful tool to present otherwise complex concepts, and they are ubiquitous in mathematics in general and in computer science in particular. They have been utilized for various concepts such as circuit design, automata theory, and here the graph is used for mathematical operations. The computational graph in deep learning is a representation of operations and data flow in a function implemented explicitly or implicitly by deep learning libraries to manage the computations of partial derivatives. A node is created for each operation, and edges indicate how the data flow; the function composition. Figure 3.1 show the

Name	Order	Example	Shape	Indices
Scalar	0	1 3.14 $\sqrt{-1}$	$()$ $()$ $()$ No shape	$\theta = 1$ $\theta = 3.14$ $\theta = \sqrt{-1}$ No indices
Vector	1	$[1]$ $[1, 2]$ $[1, 2, 3]$	(1) (2) (3)	$\theta_1 = 3$ $\theta_2 = 2$ $\theta_3 = 3$
Matrix	2	$\big[[1]\big]$ $\begin{bmatrix} [11 & 12 & 13] \\ [21 & 22 & 23] \end{bmatrix}$	$(1, 1)$ $(2, 3)$	$\theta_{1,1} = 1$ $\theta_{2,3} = 23$
3d Tensor	3	$\big[[1]\big]$ $\begin{bmatrix} \begin{bmatrix} [111 & 112 & 113 & 114] \\ [121 & 122 & 123 & 124] \\ [131 & 132 & 133 & 134] \end{bmatrix} \\ \begin{bmatrix} [211 & 212 & 213 & 214] \\ [221 & 222 & 223 & 224] \\ [231 & 232 & 233 & 234] \end{bmatrix} \end{bmatrix}$	$(1, 1, 1)$ $(2, 3, 4)$	$\theta_{1,1,1} = 1$ $\theta_{2,3,4} = 234$
nd Tensor	n		$(d_1, d_2, \ldots, d_n)$	$\theta_{i_1,i_2,\ldots,i_n} = x$

Table 3.1: List of different order tensors, the value indicates its index if multiple values exist.

graph of this example.

3.2.2.1 Autodifferentiation

Unlike numerical differentiation used in optimization for approximation the differentiation of a function, autodifferentiation provides an exact and faster solution by automatically differentiate each inner function in the chain rule of functions [44]. The differentiation or the (accumulated) derivatives of a function is calculated using the partial derivatives in the chain rule while propagating the input. There are two modes for the process, one is the *forward mode* and the second is the *backward mode.*

Forward Mode Forward mode autodifferentiation finds the differentiation of a function by iteratively computing the derivatives from inputs to each intermediate node in the computational graph up to the output. That required only a single pass with the output. It is helpful and more efficient than backward mode when dealing with vector functions since the number of paths when propagating backward is more than its is in scalar functions.

Backward Mode On the other hand, the backward mode finds the differentiation by first applying the forward pass and then using the output to compute the differentiaion from the output back to the input. Although counterintuitive, it is faster than the forward mode with scalar functions in which paths decrease at the end of the computational graph, which is usually the case of most neural networks.

3.3 Future of Deep learning Optimization

In this chapter, we introduced the traditional approach for deep learning optimization, that are based on the first order derivatives of multivariate function, the gradients. Nonetheless, gradient descent can extend to higher order derivatives, like the second order derivatives or the Hessian, which provides information on function concavity. The computations of Hessian is expensive but approximation like Quasi-Newton methods and L-BFGS in particular can help [36, 49, 70], and they have been used for optimization of various problems successfully [104]. Other approaches include metalearning and perhaps rethinking the whole backpropagation algorithm [50]. Until these approaches reache maturity, gradient descent and backpropagation will remain in use.

3.4 Chapter Summary

In this chapter, we introduced deep learning optimization, and we present some related concept. We also have established the notation for deep learning optimization that will be used consistently in the following chapters, and will be extended in chapter 6. We will present other gradient-based algorithms in chapter 5 with more details. Next, we use the notation to present loss scaling in gradient descent.

Chapter 4

Loss Scaling

Step size and learning rate are often used interchangeably, they essentially refer to the same thing, the factor that controls the amount of the future change during optimization. Learning rate is often used in the context of deep learning whereas step size is used in the context of machine learning overall. We will use the learning rate to refer only to the step size used in gradient descent optimization to update each parameter once gradients are calculated. We will keep the step size to generally refer to factor that control the future update, which include loss scaling.

In this chapter, we'll discuss different modalities of loss scaling and their effect on learning. In the next section, we'll discuss the effect of step size on loss scaling. This will be followed by the effect of loss scaling on gradient operators and mixed precision training.

4.1 Loss Scaling as Step Size

In gradient descent, loss scaling and learning rate are equivalent. Calculating the gradients form the loss and then applying the learning rate on the gradients is equivalent to scaling the loss and then computing the gradients. Historically, however, linear and neural network models have just a few parameters, and it is more natural to apply the step size on the gradients, because those are the future updates. More complex models require more parameters which increase the model capacity to capture certain representation during training, and recent models can reach billions of parameters easily. Choosing between applying the learning rate to each individual parameter or applying the learning rate on the loss only is now more obvious.

Proposition 4.1. *In gradient descent, parameters update from scaling loss by α is equivalent to applying the learning rate with that value.*

Proof. Parameter update is given in Equation 3.10:

$$\theta \leftarrow \theta - \Delta \tag{4.1}$$

Using the learning rate dictates the change as Equation 3.11:

$$\Delta = \eta \cdot g \tag{4.2}$$

Gradient g is given by Equation 3.12:

$$g = \frac{1}{n} \nabla_\theta \sum_i^n \mathcal{L}_i$$

Now, we define new gradient $\hat{g}$ computed from the loss scaled by $\alpha = \eta$:

$$\begin{aligned} \hat{g} &= \frac{1}{n} \nabla_\theta \sum_i^n \alpha \mathcal{L}_i \\ &= \alpha \frac{1}{n} \nabla_\theta \sum_i^n \mathcal{L}_i \\ &= \alpha \, g \end{aligned} \tag{4.3}$$

We use the plain update from Equation 3.7 that uses only gradients and does not use a learning rate:

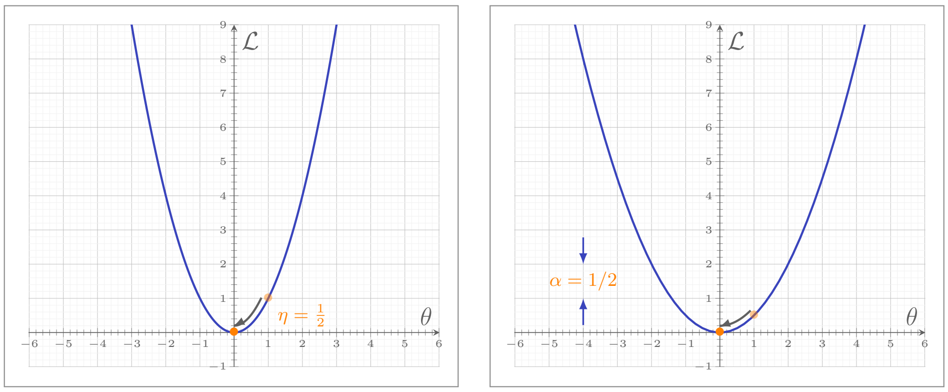
$$\Delta = \hat{g} \tag{4.4}$$

Since $\alpha = \eta$, this update is equivalent to Equation 3.11. □

4.1.1 Geometrical Interpretation

Both scaling the loss and learning rate achieve the same step size but in different ways. In this section, we provide a geometrical interpretation of loss scaling versus learning rate. The geometrical interpretation aid in the interpretation and visualization of the concept. Figure 4.1 shows a single dimensional quadratic function $f = \theta^2$, with $\theta = 1$, and minimum as 0. Gradient is computed as $\nabla_\theta f = 2 \cdot \theta = 2$. However, it results in a large amount of movement as $\theta - g = 1 - 2 = -1$. This large movement just moves the parameters to the other side.

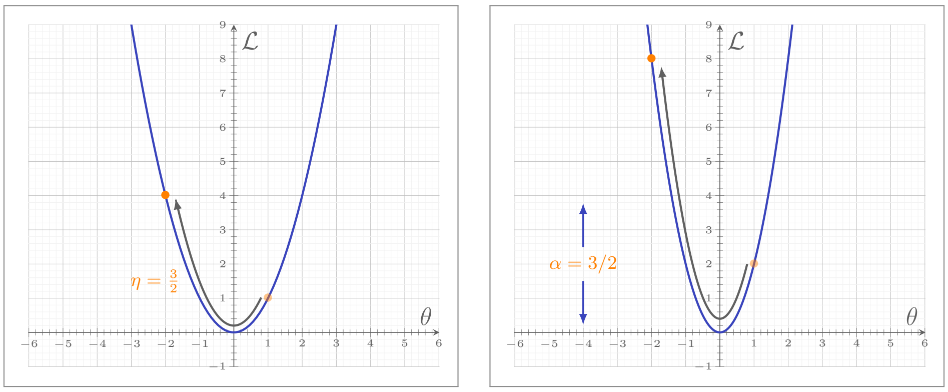
In case of scaling the loss down, $0 \leq \alpha \leq 1$; the loss surface is flattened by that factor, which results in smaller gradients. In other words, the function is scaled by that factor. The quadratic function in Figure 4.1b becomes $\alpha \cdot f = \frac{1}{2}\,\theta^2$ and the gradients now are $g = \frac{1}{2}\,2\,\theta = 1$ and $\Delta = 1 - 1 = 0$. Scaling the loss down can help with the exploding gradients, which is discussed in more details in section 4.2.



(a) Learning rate = $\frac{1}{2}$ **(b)** Scaling the loss by $\frac{1}{2}$

Figure 4.1: Scaling the learning rate and the loss down.

When scaling the loss up, $\alpha \geq 1$ the loss surface stretched by that factor, which results in larger gradients. The quadratic function in Figure 4.2b becomes $\alpha \cdot f = \frac{3}{2}\,\theta^2$ and the gradients now are $3\,\theta = \theta = 3$ and $\Delta = 1 - 3 = -2$. Large amount of scaling is not optimal for the quadratic function, but it is for other functions, and it can help with the vanishing gradients as explained in section 4.2.



(a) Learning rate $= \frac{3}{2}$ **(b)** Scaling the loss by $\frac{3}{2}$

Figure 4.2: Scaling the learning rate and the loss up.

In the next subsection, I present an experiment that shows time enhancement when using loss scaling instead of the learning rate.

4.1.2 Experiment

For the experiments, I created four models: A, B, C, and D. The models are described below and are created with different number of parameters to show the effect of parameters on the execution time. Each model is trained with different batch sizes (32 and 64), different number of epochs (1 and 10), the two methods (learning rate LR and loss scaling SC), and different scaling and learning factor for each method (0.1 and 10).

Table 4.1: Description of the models.

	A	B	C	D
Type	CNN	CNN	Dense	Dense
Parameters	$8,954$	$91,082$	$12,730$	$109,386$
Description	convolution 320 convolution 4624 dense 4010	convolution 1280 convolution 73792 dense 16010	dense 12560 dense 170	dense 100480 dense 8256 dense 650

4.1.2.1 Results and Discussion

I have presented the execution time for each model and each scale factor in Figure 4.3. Each experiment is run for 10 epochs and averaged; the time axis shows the time per epoch. Individual epochs, except for the first one[1], have similar timing.

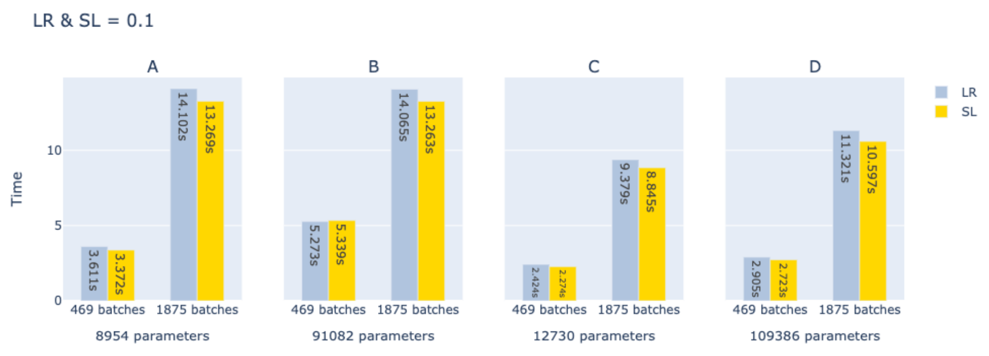


Figure 4.3: Execution times for sl vs. lr and scale factor of 0.1.

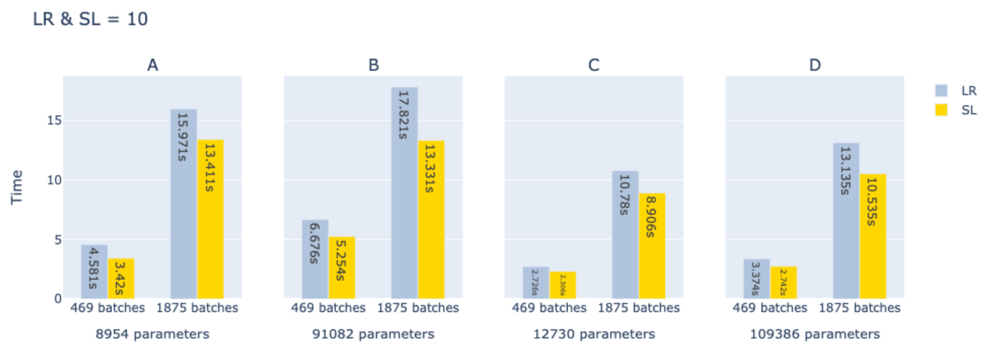


Figure 4.4: Execution times for sl vs. lr and scale factor of 10.

[1]First batches and epochs interleave with other processes unrelated to gradients operations, like data caching, kernel launching and more. For more information, please check this link.

As the experiments show, execution time improves when avoiding the use of learning rate and using loss scaling instead. Experiments also show that the time is sensitive to the number of batches, which represent the number of updates. More updates imply time improvement at each update, and that can be beneficial especially that it has been recommended to use small batches (and subsequently more updates) for better generalization [6, 74].

Now that we showed empirical evidence that loss scaling is beneficial in practice, we move to the possibility of using the loss to help with numerical instability. In particular, to help with vanishing and exploding gradients. We first introduce the problem, and then we move to discuss various ways that help alleviate the problem. Finally, we introduce loss scaling as a tool that helps with vanishing gradients, especially in mixed precision training.

4.2 Loss Scaling for Numerical Instability

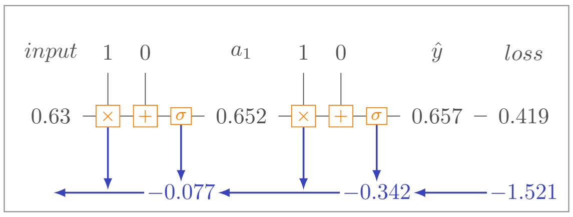
Loss scaling is used to prevent numerical underflow or overflow during training. It is used in particular to help with vanishing and exploding gradients. The problems happen because the hardware is limited to represent numbers within a certain range. There may be a need to go beyond the specified range which arises in deeper models as shown in Figure 4.5, and leads to numerical instability and poor performance. In this section, we present the problem of vanishing and exploding gradients as well as different techniques used to deal with the problem. Finally, we present mixed precision training in subsection 4.2.2 as a technique used to accelerate training and employ loss scaling to avoid vanishing gradients in particular.

4.2.1 Vanishing and Exploding Gradients

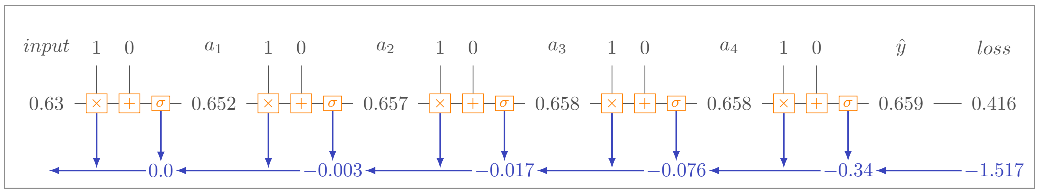
Theoretically, a deep network should be able to learn a representation that is at least as good as its shallower subnet where the last k layers are removed. This can be achieved by transforming the representation of the last k layers by identity function, in which $f(g(x)) = g(x)$. The idea is at the core of the residual networks [47]. Deeper networks therefore, if not at least as expressive as the shallow networks, are actually more expressive. However, the experimental results show otherwise. The performance of deeper networks is worse than their shallower versions and they are harder to train. This phenomenon can be explained by multiple reasons but the main reason behind the difficulty of training is the concept of vanishing and exploding gradients. Increased depth requires consecutive multiplications of gradients during backpropagation. The consecutive multiplications result in exponential growth or decay to the point of no longer being supportable by the assigned datatypes. Before proceeding with the gradients problems, it is necessary to take a closer look at the underlying problems of arithmetic overflow and underflow.

An arithmetic overflow happens when the result of an operation exceeds the largest values that can be represented in the assigned datatype. It is detectable by an overflow flag and may show up as ∞ or `NaN` (Not A Number). If we have a 2-bits type, for instance, with four unsigned number ranges from 0 to 3, then any subtraction from 0 or addition to 3 will result in an overflow. On the other hand, an arithmetic underflow happens when the result of a floating point operation is less than 1 and less than the value that can be represented by the floating part, which shows up as a zero. A number in a datatype that has only a single digit before the floating point and two digits after the floating point as in 7.95 will vanish by any three consecutive divisions by 10 (or multiplications by 0.1) that sets it to 0.0. Both issues, especially the underflow issue, are more complicated than presented here and are presented in detail in [48].

Having discussed both underflow and overflow, we can move to the gradients problems. The *vanishing gradients* problem happens in deep networks when small numbers are consecutively multiplied to extract the gradients but result in zero gradients that are unable to update the parameters anymore. Likewise, the *exploding gradients* problem happens for the same reason but when too big or too small numbers are multiplied and result in overflowed gradients that make erroneous updates. Both of these problems have been fundamental issues in



(a) Shallow Model



(b) Deeper Model

Figure 4.5: Vanishing gradients in shallow and deeper model

deep learning since the 1990s, and the reason behind developing various methods and architectures to rectify them can explain the success achieved by a method [16, 40, 51, 85]. The problems are dealt with by modifying the parameters, the activation functions, or the topology, three types of solutions that we present next.

4.2.1.1 Parameter Initialization and Regularization

Parameters initialization specifies the initial condition which determines whether a model will converge to a good minimum or not. Two initialization schemes may result in the same loss value, but one may be harder to train than the other. Consider a trivial function $f(x, \theta) = x \times \theta_1 \times \theta_2$ and an input $x = \{1\}$. Two isomorphic sets of parameters will generate the same value $f(x; \theta_1 = 1, \theta_2 = 2^{-1}) = \frac{1}{2}$ and $f(x; \theta_1 = 10^{10}, \theta_2 = 20^{-10}) = \frac{1}{2}$. The first is clearly easier to move to an optimal solution of, let's assume, 0. A poor initialization can also reduce the capacity of a model by moving units in the same layer to be identical, or even worse, to initialize them to be identical. Units in the same layer with the same values, or *symmetrical units*, can never learn differently [14]. Another problem is initializing the units with values that saturate the activation functions to their limits, preventing a model from learning. I'll discuss here how different initialization and regularization schemes are used to address the gradients problem.

If we want to prevent gradients from exploding or vanishing, we need to keep the squared singular values[2] of the Jacobian between layers near 1. Larger singular values indicate possible gradient explosions while smaller values indicate possible gradients vanishing [35, 110]. Different initialization schemes provide various ways to keep the Jacobian singular values near 1. Gaussian random initialization can preserve the norm as each layer means that the singular values are close to, but not exactly, 1 [35]. Sussillo and Abbott proposed random walk initialization that aims at producing random matrices where the norm is preserved by random walk correction [109]. Using orthogonal matrices for initialization is proposed as well for the same purpose, especially because they ensure that the Jacobian exact singular values of 1 for linear networks [100]. There are several methods that aim at choosing the right initial conditions in order to keep forward and backward flow under control. Once the parameters are initialized, we can further regularize their values to control the gradients flow by using normalization.

The process of *normalization* adjusts a set of values to a choice of their statistics while keeping their relative relationship. Input normalization is applied in deep learning to help a model converge faster and help to regularize the forward pass. However, it has limited effect on the backward pass since the input layer does not affect any further layers downstream. Batch normalization was introduced to achieve similar effect but on the intermediate nodes [56]. The method suggests that instead of normalizing a layer's inputs with respect to each other, each input can be normalized independently to the minibatch statistics. This helps with the gradients problems as it controls the parameters' growth. Weight normalization does not modify the inputs of a node but rather its weights in a similar way [97]. Other normalization techniques exist; please refer to [54] for an overview. One

[2]The absolute eigenvalues of the squared Jacobian matrix

factor that is closely related to initialization is the choice of activation function.

4.2.1.2 Activation Functions

The choice of activation functions is critical since they influence the network in two steps. First in the forward pass, they transform a unit's inputs into an output affecting neurons in the subsequent layers. Later in the backward pass, their derivatives are multiplied by the upcoming gradients stream which affects the flow going to all preceding layers. Most activation functions relate to the problem of vanishing gradients more than the exploding problem, especially the sigmoid activation functions.

Sigmoid functions are functions that have an S shape, implying that they are monotonically increasing and bound to some finite value. Examples of such functions are the logistic and hyperbolic tangent functions, see Appendix A. Since they are bound to finite values, the functions have been used to estimate probabilities since the 1950s [24, 114]. Despite their ability to transform the input which gives a neuron more expressive power, such transformation binds the result to a small range. A logistic function (σ), for example, squashes input values between zero and one which means that the chance the parameters are going to be saturated to these values is high. Its small derivative $\sigma(1-\sigma)$ with maximum value at $\frac{1}{4}$ exposes the network to the gradients problems.

One solution to the gradients problem is to use the identity function that keeps the input as is. The identity function does not change the gradients flow with derivative of 1 but it reduces the node expressiveness as being a linear unit and throws further burden on initialization. Another alternative is to use a linear rectifier function $ReLU$ that is bounded by zero on one side and not bounded by one on the other [79]. The function has contributed to the success of AlexNet and some of the following architectures in increasing the network depth. While it can keep the original input undisturbed when positive, with gradients of 1, it can be quite problematic with negative inputs as its derivative is zero. *Leaky* $ReLU$ [72], and its parametric generalization [46] keeps derivatives the same when input is positive and also provides more information on the negative inputs with a fractional derivatives.

Another function that is similar to the rectifier but has some nice properties is *softplus* [28]. The function is not bound on the positive side and is differentiable everywhere with the logistic function as its first order derivative. More activation functions that provide different transformations exist [10, 61], but choosing the activation function is tightly coupled with the appropriate initialization. Both initialization and activation mitigate the gradients issues by changing the gradients stream; topology can do the same by changing how the stream flow.

4.2.1.3 Topology

Changing the network topology, or even connections inside neurons, helps with the gradients problems. One of the earliest attempts to achieve this was in *long short term memory* (LSTM) to learn long time dependencies that traditional recurrent neural networks (RNNs) failed to capture [16, 52]. In a classical RNNs neuron, there are at least two weight vectors, one for the current input and one for the output from a previous time step. LSTM adds at least three copies of those with various connections, but most importantly, it adds an internal state connection that goes from one time step to another without passing through any activation function and is not assigned direct weights. The independence allows the gradients to have a channel to propagate backwards with minimal alteration even if other gradients vanish. More successful architectures have similar mechanisms that account for their success and ability to have deeper networks.

One of the successful architectures that allow constructing very deep feedforward networks is ResNet [47]. The rational is that a deeper model should learn what an identical shallower model can learn and more, which does not hold in practice. He, et al, introduced residual connections that allow the information to freely propagate from one layer to the next, in a way similar to that of LSTM [47]. In fact, an investigation shows the resemblance between the RNNs and residual nets [68]. Other architectures that incorporate some gating mechanisms that help with the flow of the input in the forward propagation and updates in the backward propagation include highway networks, VGG networks among others [105, 108]

All the solutions mentioned above are advantageous for problems in existing gradients. However, we can draw a simple yet effective solution from a hardware viewpoint, particularly from the area of mixed precision training. One of the solutions that helps with gradients problems and can be used as a proxy for some of the hyperparameters in learning algorithms is the loss scaling.

4.2.2 Mixed Precision Training

The capacity of a model is proportional to its size [82]. However, larger models consume more time in training and at inference time. There are efforts to lower the time and memory consumption by reducing the precision of the parameters during training, as done in mixed precising training or or using model compression and knowledge distillation after a model has been trained [42, 67, 77]. Mixed precision training is a way to use lower precision format when possible to accelerate training and higher precision format in critical computations to maintain good performance.

Training is typically conducted using the same precision format, such as single precision floating points format, where the numbers occupy 32 bits of memory (FP32). There are other types of precision that are less frequently used. Double precision format numbers occupy 64 bits (FP64) whereas half precision occupies 16 bits (FP16). Higher precision formats are better in performance, but there is a tradeoff between precision on one hand, and memory and time on the other. We can accelerate the training by using mixed precision numbers such that lower precision numbers are used for precision-insensitive calculations and single precision is kept for sensitive calculations [25, 77].

Smaller formats lose some of the accuracy and several techniques have been used to mitigate that effect. One is to keep a copy of the model in the original larger format but implement the forward pass and the gradient calculations in a copy of smaller formats. Once computed, the lower precision gradients update the original copy. Also, small precision formats should be used whenever beneficial. For instance, final reductions and normalization should be computed in larger formats while each tensor reading and writing from memory can be performed in smaller formats. The most interesting technique of all is the *loss scaling*.

4.2.2.1 Loss Scaling in Mixed Precision Training

Loss scaling has been introduced in [77] to mitigate the problems in mixed integer training which is, in its core, identical to Proposition 4.1. The idea is to scale the loss by a factor α to a value that moves the gradients into representable values without affecting the parameters update by scaling the gradients back using the same scaling factor.

$$\hat{g} = \alpha \cdot \frac{\partial \mathcal{L}}{\partial \theta} \tag{4.5}$$

Finally, the gradients are scaled back to match the original gradients using full precision.

$$g = \frac{\hat{g}}{\alpha} \tag{4.6}$$

The factor of scaling in the work is fixed during the entire training which is called *fixed loss scaling*. Since the gradients are scaled back to the original loss, there is no effect of the loss scaling on the final updates. An enhancement is achieved when we have *dynamic loss scaling* where the scaling factor is changed during training [63, 75]. Zhao, et al, proposed a more interesting approach, which introduces the concept of *adaptive loss scaling* [117]. The concept is to adapt different loss scaling factors not only in time for the training, but also for different layers in the network. Scaling the loss back and using learning rate is redundant and computation can be saved by scaling the loss in a way that accounts for the step size. Finally, we present a metalearning method to learn layerwise weights for layerwise step size and can be used for adaptive and learned rescaling factor.

Experiment I introduce a method to rescale the gradients in subsection 6.3.4. In the method, I use a model to rescale gradients differently at each layer, and at the same time I avoid using the learning rate when updating the parameters. The method is not introduced here because it is based on Metalearning which is presented in chapter 6.

4.3 Chapter Summary

In this chapter, we begin with showing how loss scaling and learning rate are equivalent as specifiers of the step size. Besides providing a theoretical proof, we present a geometrical and visual tool for interpretation, supported by our empirical results. Following that, we move to vanishing and exploding gradients, two problems that loss scaling can help with. We review some literature that dealt with the problem successfully. In the end, we present mixed precision training as a technique that leverages loss scaling to prevent gradients from exploding.

Chapter 5

Loss Scaling for Adaptive Learning Algorithms

Gradient descent, described earlier, depends entirely on the learning rate to adjust the gradients in each update. On the other hand, adaptive learning algorithms use information from past gradients to produce adaptive step sizes. In most cases, the information comes from estimates of gradient moments, such as momentum and Nesterov's momentum that use some estimates of first moments [81]. Nonetheless, adaptive optimizers use non-learnable parameters ϕ, called *hyperparameters* to control the effect of the information. Those parameters must be specified ahead of training and may be tuned during training. Different hyperparameter values are required for topology, activations being used, initialization, and the task at hand. It is challenging to decide on the right values in an optimal manner.

Information from past gradients offers a perspective on dimensions that change differently depending on movement. A loss surface embedded in three dimensions is shown in Figure 5.1. The parameters change differently; movements in θ_1 change the loss faster, whereas the same movement along the other dimension changes the loss at a slower rate. Therefore, a history or moving estimates may regulate the learning and help reach a local minimum faster.

In this chapter, we will examine some widely used adaptive algorithms and the

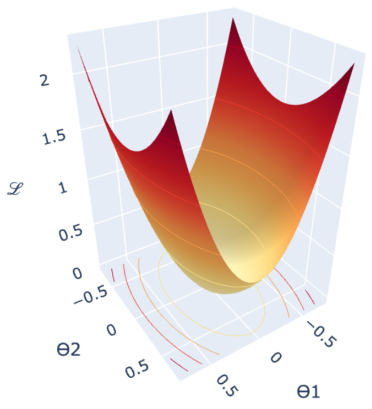


Figure 5.1: Loss surface that is narrow in one dimension.

different approaches they use to update the parameters. Then we cover the general use of the first and second moments in adaptive optimizers, and the possibility of replacing the hyperparameters and variables with scaling the loss. In the end, we present the results and the finding of empirically replacing those hyperparameters with loss scaling. Please see Appendix B for a detailed description of the update rules.

5.1 Adaptive Learning Algorithms

Adaptive Gradient Algorithm, Adagrad for short, adapts the learning rate such that the more frequently updated parameters receive smaller updates. That is done simply by normalizing the gradients to their history of magnitude, the square root of the sum of the squared gradients [27]. The parameters are similarly updated by RMSProp or the Root Mean Square Propagation, but it normalizes the learning rate by a weighted sum of the current and previous magnitudes of gradient [111]. That lets recently updated parameters receive smaller updates. RMSProp inspired later algorithms; some of the most significant algorithms are Adam and Adadelta.

Adam normalizes the learning rate just as RMSProp but updates the parameters

Table 5.1: List of change functions in some gradient descent optimizers.

Optimizer	Update Function $u(\theta, g, \phi)$	Change Function $\Delta(g, \phi)$	Initialization ϕ
SGD	$u = \theta - \Delta$	$\boldsymbol{\Delta = \eta \cdot g}$	$\eta = 0.01$
Momentum	$u = \theta - \Delta$	$\boldsymbol{\Delta = \eta \cdot s}$	$\eta = 0.01$
		$s = \beta \cdot s + g$	$s = 0$
			$\beta = 0.9$
RMSProp	$u = \theta - \Delta$	$\boldsymbol{\Delta = \eta \cdot \frac{g}{\sqrt{r}}}$	$\eta = 0.001$
		$r \leftarrow \gamma\, r + \overline{\gamma}\, g \odot g$	$r = 0$
			$\gamma = 0.9$
Adam	$u = \theta - \Delta$	$\boldsymbol{\Delta = \eta \cdot \frac{s}{\sqrt{r}}}$	$\eta = 0.001$
		$s \leftarrow \beta\, s + \overline{\beta}\, g$	$s = 0$
		$r \leftarrow \gamma\, r + \overline{\gamma}\, g \odot g$	$r = 0$
			$\beta = 0.9$
			$\gamma = 0.99$

differently. It updates parameters according to weighted sum of current and previous gradients [60]. Adadelta normalizes the weight like RMSProp, but does not use an explicit learning rate. It scales the step size by weighted history of the normalized magnitudes of gradients. It is possible to modify adaptive algorithms to incorporate Nesterov's approach to compute updates, as achieved in Nadam [26]. The aforementioned algorithms are among the most common optimizers because of their performance and ease of use, as they are already implemented in libraries like TensorFlow and PyTorch [1, 87]. There is an abundance of other optimizers, and one review identified more than a hundred gradient-based optimization algorithms [103]. I have presented a list of change functions in some gradient descent optimizers in Table 5.1.

One of the challenges in this type of learning algorithms is the hyperparameters tuning or the selection the right values for the hyperparameters at certain steps.

There have been efforts to provide insight into their performance [101, 106]; however, a mathematical or structured solution is still lacking. For example, there is no explicit formula between the learning rate and the depth of the network. Another problem is the fact that the hyperparameters are *typically* fixed for all layers. That is, the learning rate is fixed for all gradients regardless of their position in the network, and regardless of the activations being used.

This chapter extends using the loss from replacing only the learning rate to replace some hyperparameters in adaptive learning algorithms. In particular, I present the effect of using loss scaling in two widely used optimizers, Adam [60] and RMSProp [111]. The two optimizers normalize the gradients, or a history of the gradients, with the magnitude history of the gradients as we see next. Finally, I show that the loss can not be used to replace the momentum and other optimizer variables.

5.2 Moment Estimation in Optimizers

Generally, optimizers in deep learning follow a similar approach; some use estimates of first moments, others use estimates of second moments, and some combine both. First-moment optimizers, like classical momentum, keep the information about gradients' history (moving average) in each update. Second-moment optimizers, like AdaDelta and Adam, keep additional information about the history of magnitude of the gradients (moving variance) and normalize the gradients to its square root. We can say that the update rule can be roughly generalized to have moving average s as well as moving variance r:

$$s \leftarrow \beta \; s \;\; + \overline{\beta} \; g \tag{5.1}$$

$$r \leftarrow \gamma \; r \;\; + \overline{\gamma} \; g \odot g \tag{5.2}$$

where the symbol $\odot$ signifies the Hadamard or element wise product, and $g^{\odot 2} = g \odot g$. If the gradients are not zero, the change is given by:

$$\Delta = \eta \frac{s}{\sqrt{r}} \tag{5.3}$$

With β and γ as the hyperparameters having values in the range $[0, 1]$ and the bar indicates the complement, $(\beta = 0.9 \Rightarrow \overline{\beta} = 0.1)$. The formulation is not exact but can be easily extended to match each optimizer's algorithm. The reason we choose this general notion instead of specific update rules is that it provides an umbrella under which we can address the same problem in different optimizers easily, while we are still specific when specifying the exact values. For example, the typical values for classical momentum, Adam, and RMSprop are $(\beta > 0, \gamma = 1, \text{and } r = 1)$, $(\beta = 0.9, \gamma = 0.999)$, and $(\beta = 0, \gamma = 0.9)$ respectively.

5.2.1 First Moment Estimates

There are four hyperparameters in Equation 5.1 and Equation 5.2: β, γ and their complements. Unfolding the recurrence of s_i in Equation 5.1 gives the classical momentum:

$$s_1 = \overline{\beta} g_1 \tag{5.4a}$$

$$s_2 = \beta \overline{\beta} g_1 + \overline{\beta} g_2 \tag{5.4b}$$

$$s_3 = \beta^2 \overline{\beta} g_1 + \beta \overline{\beta} g_2 + \overline{\beta} g_3 \tag{5.4c}$$

This will let the exact first moment estimate at iteration i to be

$$s_i = \sum_{k=1}^{i} \beta^{i-k} \overline{\beta} \cdot g_k \tag{5.5}$$

$$s_i = \overline{\beta} \sum_{k=1}^{i} \beta^{i-k} \cdot g_k \tag{5.6}$$

5.2.2 Second Moment Estimates

Similarly, the recurrence in Equation 5.2 will unfold:

$$r_1 = \overline{\gamma} g_1^{\odot 2} \tag{5.7a}$$

$$r_2 = \gamma \overline{\gamma} g_1^{\odot 2} + \overline{\gamma} g_2^{\odot 2} \tag{5.7b}$$

$$r_3 = \gamma^2 \overline{\gamma} g_1^{\odot 2} + \gamma \overline{\gamma} g_2^{\odot 2} + \overline{\gamma} g_3^{\odot 2} \tag{5.7c}$$

This will let the exact second moment estimate at iteration i be:

$$r_i = \sum_{k=1}^{i} \overline{\gamma} \gamma^{i-k} \cdot g_k^{\odot 2} \tag{5.8}$$

$$r_i = \overline{\gamma} \sum_{k=1}^{i} \gamma^{i-k} \cdot g_k^{\odot 2} \tag{5.9}$$

5.2.3 Integrating Both Estimates

Substituting Equation 5.6 and Equation 5.9 in Equation 5.3:

$$\Delta_i \leftarrow \eta \, \frac{\overline{\beta}}{\sqrt{\overline{\gamma}}} \quad \frac{\sum_{k=1}^{i} \beta^{i-k} \cdot g_k}{\sqrt{\sum_{k=1}^{i} \gamma^{i-k} \cdot g_k^{\odot 2}}} \tag{5.10}$$

5.2.4 Loss Scaling for Moment Estimates

If gradient computations were not expensive in term of processing time, we could scale the loss differently. That is, we could provide a scaling factor $\eta\overline{\beta}$ for the first moment and another scaling factor $\overline{\gamma}$ for the second moment. However, since gradients are calculated only once, then a single scaling factor is needed.

Proposition 5.1. *Let the change of parameters at time i be:*

$$\hat{\Delta}_i = \eta \frac{\hat{s}_i}{\sqrt{\hat{r}_i}} \tag{5.11}$$

That has the following estimates from scaled gradients:

$$\begin{aligned} \hat{s} &\leftarrow \beta\ \hat{s}\ \ + \alpha\, \hat{g} \\ \hat{r} &\leftarrow \gamma\ \hat{r}\ \ + \hat{g}^{\odot 2} \end{aligned} \tag{5.12}$$

Then, the loss scaling factor that makes $\hat{\Delta}_i$ equivalent to the change in Equation 5.3, $\hat{\Delta}_i = \Delta_i$, is:

$$\alpha = \frac{\eta\overline{\beta}}{\sqrt{\overline{\gamma}}} \tag{5.13}$$

In other words, the proper values to scale the loss and remove two hyperparameters which are $\overline{\beta}$, and $\overline{\gamma}$ is $\alpha = \frac{\eta\overline{\beta}}{\sqrt{\overline{\gamma}}}$, given their new values in Equation 5.12. The proof of the proposition that has been put forth is below.

Proof. We have already established that the change at time i is given by Equation 5.10. If we scale the loss by:

$$\alpha = \frac{\eta\overline{\beta}}{\sqrt{\overline{\gamma}}} \tag{5.14}$$

Then we will have the following update:

$$\Delta_i = \ \frac{\alpha}{\sqrt{\alpha^2}} \quad \frac{\sum_{k=1}^{i} \beta^{i-k} \cdot g_k}{\sqrt{\sum_{k=1}^{i} \gamma^{i-k} \cdot g_k^{\odot 2}}} \tag{5.15}$$

This implies that any scaling factor will be eliminated due to the normalization. Still, there is a window of improvement, especially that deep learning libraries use the complementary software, like TensorFlow and PyTorch. Each parameter derivative and squared derivative are multiplied by their respective complement and finally update the value of each parameter by multiplying with the learning rate. Therefore, adding a single multiplication will improve the performance if we scale the loss appropriately.

First, let's compute a scaled gradient $\hat{g}$ from a scaled loss:

$$\alpha g = \hat{g} = \frac{\partial \hat{\mathcal{L}}}{\partial \theta} = \frac{\partial \alpha \mathcal{L}}{\partial \theta} \tag{5.16}$$

Then, we remove the complement from the scaled moving average $\hat{s}$ and scale the gradient by α:

$$\hat{s}_i \leftarrow \beta \ \hat{s}_{i-1} \ + \alpha \ \hat{g}_i \tag{5.17}$$

$$\hat{s}_i = \alpha \ \sum_{k=1}^{i} \beta^{i-k} \cdot \hat{g}_k \tag{5.18}$$

We remove the complement from the moving variance $\hat{r}_i$:

$$\hat{r}_i \leftarrow \gamma \ \hat{r}_{i-1} \ + \ \hat{g}_i \odot \hat{g}_i \tag{5.19}$$

$$r_i = \sum_{k=1}^{i} \gamma^{i-k} \cdot \hat{g}_k^{\odot 2} \tag{5.20}$$

We need to remove the learning rate from the change since we scale the gradients, so we let the change be:

$$\Delta_i = \frac{\hat{s}_i}{\sqrt{\hat{r}_i}} \tag{5.21}$$

Using scaled moments estimates from Equation 5.18 and Equation 5.20:

$$\Delta_i = \frac{\alpha \ \sum_{k=1}^{i} \beta^{i-k} \cdot \hat{g}_k}{\sqrt{\sum_{k=1}^{i} \gamma^{i-k} \cdot \hat{g}_k^{\odot 2}}} \tag{5.22}$$

Substituting $\hat{g}$ with αg from Equation 5.16 :

$$\Delta_i = \frac{\alpha \ \sum_{k=1}^{i} \beta^{i-k} \cdot \alpha g_k}{\sqrt{\sum_{k=1}^{i} \gamma^{i-k} \cdot \alpha^2 g_k^{\odot 2}}} \tag{5.23}$$

Then, we extract the constants out of the summations:

$$\Delta_i = \alpha \cdot \frac{\sum_{k=1}^{i} \beta^{i-k} \cdot g_k}{\sqrt{\sum_{k=1}^{i} \gamma^{i-k} \cdot g_k^{\odot 2}}} \tag{5.24}$$

Substituting for α from Equation 5.14:

$$\Delta_i = \frac{\eta\overline{\beta}}{\gamma} \cdot \frac{\sum_{k=1}^{i} \beta^{i-k} \cdot g_k}{\sqrt{\sum_{k=1}^{i} \gamma^{i-k} \cdot g_k^{\odot 2}}} \tag{5.25}$$

This is the update that we needed in Equation 5.10, with two hyperparameters multiplications reduced for each parameter. $\square$

5.2.4.1 Experiment

Similar to loss scaling experiment presented in subsection 4.1.2, the experiments for loss scaling for adaptive learning algorithms entail comparing the original optimizer with a modified version that applies the rules enumerated above. The algorithm involves first scaling the loss and then using the modified optimizer. To make a fair comparison, I created two custom optimizers, one for the original method and the other for the modified version. The reason is that TensorFlow has implemented some core acceleration for built-in optimizers like Adam that I can not directly replicate and this is one reason to switch to PyTorch for the experiments. The optimizers are used to train identical models of the same structure and weights with the same data to fix all other variables.

In the current experiments, I use the same models as used in the first experiments described in subsection 4.1.2. The four models: A, B, C, and D were described in Table 4.1. They are created with different number of parameters and different type of connections in which two connection types are dense and the other two are CNN to show their effect on execution time.

I trained each model with both optimizers Adam and RMSProp and their modified versions, with different batch sizes (32, 64, and 128). That gives 48 experiments; 4 experiments corresponding to the four models, 4 experiemnts corresponding to the optimizers and 3 for the batches. I ran each experiment for 20 epochs and computed the average time for batches per epoch. For example, selecting 128 example in each batch of 60000 examples in MNIST yields 469 batches per epoch, and the total time is divided by 20 to show how long it takes to run 469 updates on average. Finally, The hyperparameters values are just the default values, for Adam they are enumerated as $\eta = 0.001$, $\beta = 0.9$, and $\gamma = 0.999$, while for RMSProp they are $\eta = 0.001$ and $\gamma = 0.9$. I'll make my code, details and logs available via Github. The AWS instance type used to produce the experiments is `G4dn` which is one of the G instances that use hardware accelerators. It has 1 GPU, 8 virtual CPUs, 32 GiB of memory, and 16 GiB of GPU memory. The experiments are shown in Figure 5.2 and Figure 5.3.

5.2.5 Discussion

The experiments show that there is improvement in execution time when scaling the loss versus using the original optimizer. Similar to the first experiment, the time is directly proportional to the number of batches and the model size. More batches means more updates and more parameters means more computations in each update. Moreover, there is more improvement in Adam than in RMSProp because the former uses more hyperparameters and thus exploits the benefits of scaling the loss.

5.3 Momentum Variables and Step Size

Momentum has hyperparameters as well as variables that save the moving average. We cannot use a loss variable as a proxy of them as shown below. First, the

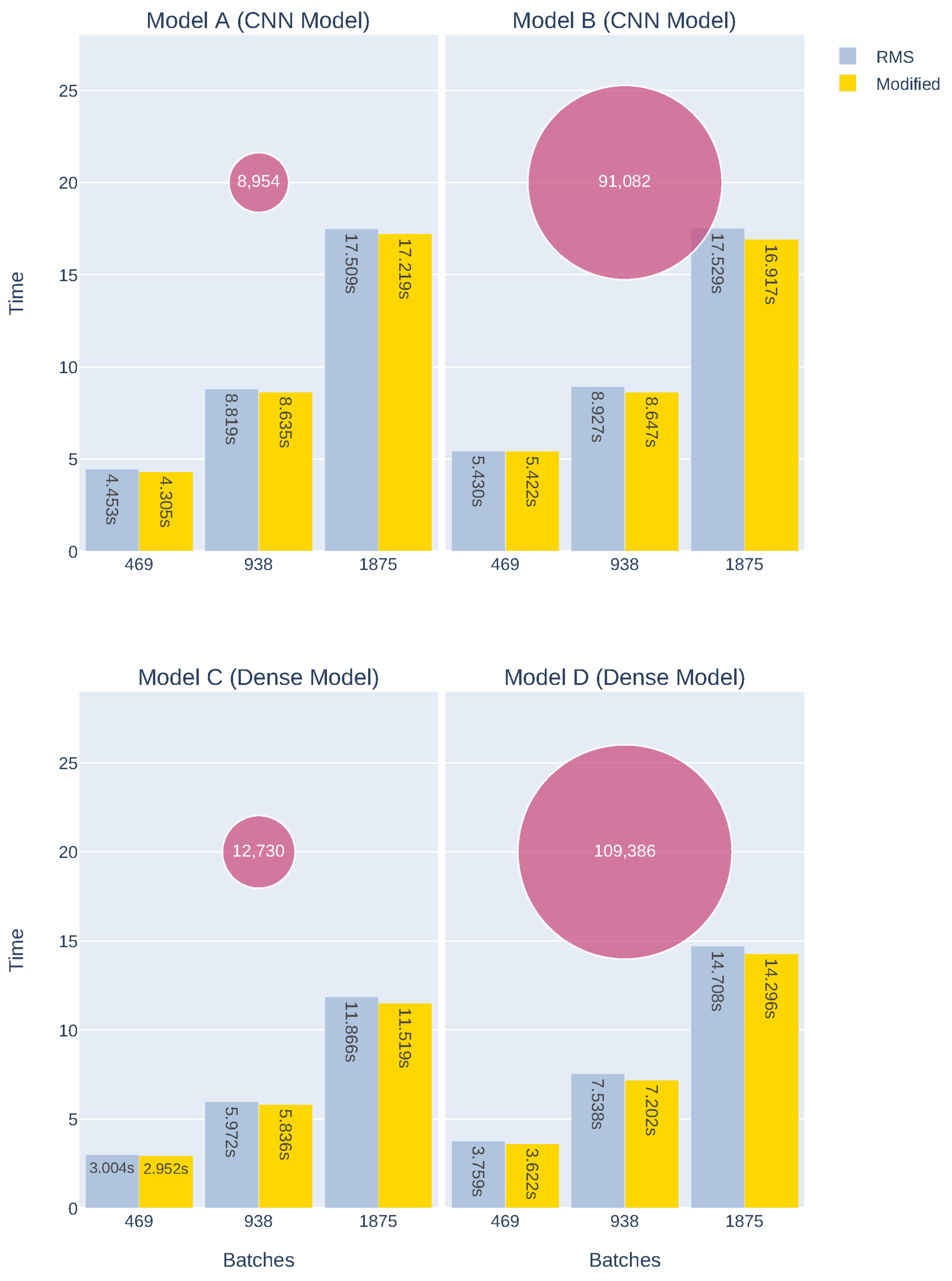


Figure 5.2: Execution time of the models using RMSProp and scaled RMSProp. The circle signifies the model size in terms of the number of parameters.

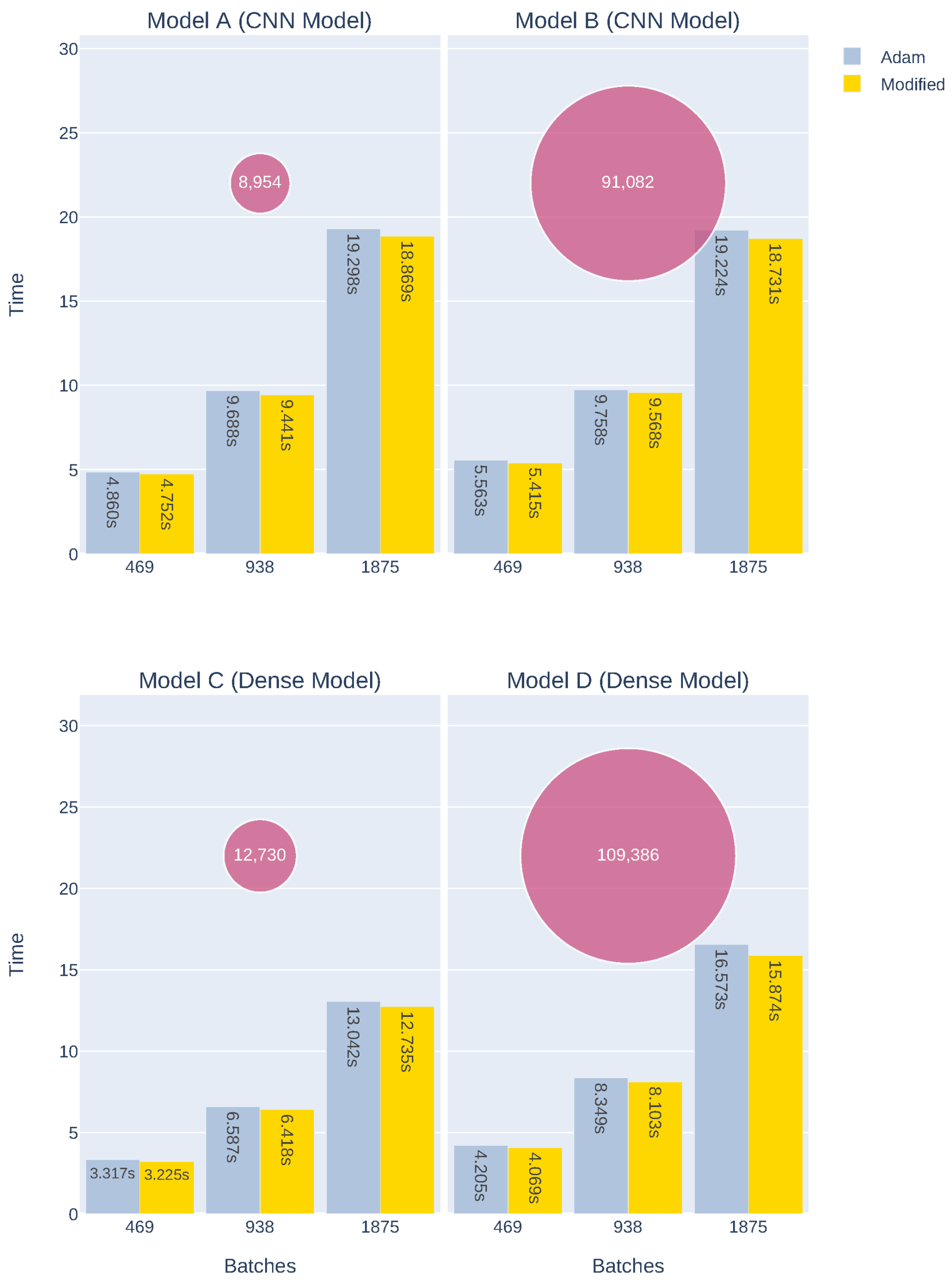


Figure 5.3: Execution time of the models using Adam and scaled Adam.

update rule that gives a momentum to the weights is:

$$\begin{aligned} \Delta &= \eta \cdot s \\ \theta &\leftarrow \theta - \eta \cdot s \end{aligned} \tag{5.26}$$

The velocity s keeps track of the previous weights movements or the gradients according to a momentum coefficient β:

$$s \leftarrow g + \beta \cdot s \tag{5.27}$$

Unfolding the recurrence:

$$s_1 = g_1 \tag{5.28a}$$

$$s_2 = \beta g_1 + g_2 \tag{5.28b}$$

$$s_3 = \beta^2 g_1 + \beta g_2 + g_3 \tag{5.28c}$$

This will let the exact value at iteration i to be:

$$s_i = \sum_{k=1}^{i} \beta^{i-k} \cdot g_k \tag{5.29}$$

$$= \sum_{k=1}^{i} \beta^{i-k} \cdot \frac{\partial \mathcal{L}_k}{\partial \theta_{k-1}} \tag{5.30}$$

To rephrase it, a momentum variable s_i in iteration i is a weighted sum of the partial derivatives of the loss at that iteration with respect to the current parameter. By substituting s_i in Equation 5.26 with Equation 5.30:

$$\theta_i \leftarrow \theta_{i-1} - \eta \sum_{k=1}^{i} \beta^{i-k} \cdot \frac{\partial \mathcal{L}_k}{\partial \theta_{k-1}} \tag{5.31}$$

5.3.1 Replacing Momentum with Loss

Instead of keeping a history of gradients v in addition to computing the gradients in each iteration, we can use an update rule similar to Equation 3.10:

$$\theta_i \leftarrow \theta_{i-1} - \eta g_i \tag{5.32}$$

Except that we keep a history of losses, say a momentum loss ℓ, and the gradients are calculated only once from that loss:

$$g_i = \frac{\partial \ell_i}{\partial \theta_{i-1}} \tag{5.33}$$

The momentum loss variable ℓ is a scalar that keeps a history of both the current loss $\mathcal{L}_i$ and the previous losses ℓ_{i-1}:

$$\ell_i \leftarrow \beta\, \ell_{i-1} + \mathcal{L}_i \tag{5.34}$$

Unfolding the recurrence:

$$\ell_1 = \mathcal{L}_1 \tag{5.35a}$$

$$\ell_2 = \beta\mathcal{L}_1 \qquad + \mathcal{L}_2 \tag{5.35b}$$

$$\ell_3 = \beta^2\mathcal{L}_1 + \beta\mathcal{L}_2 + \mathcal{L}_3 \tag{5.35c}$$

$$\ell_i = \sum_{k=1}^{i} m^{i-k} \cdot \mathcal{L}_k \tag{5.36}$$

Taking the partial derivatives of both sides:

$$\frac{\partial \ell_i}{\partial \theta_{i-1}} = \sum_{k=1}^{i} \beta^{i-k} \cdot \frac{\partial \mathcal{L}_k}{\partial \theta_{i-1}} \tag{5.37}$$

Substituting Equation 5.37 in Equation 5.32:

$$\theta_i \leftarrow \theta_{i-1} - \eta \sum_{k=1}^{i} \beta^{i-k} \cdot \frac{\partial \mathcal{L}_k}{\partial \theta_{i-1}} \tag{5.38}$$

5.3.2 Discussion

The difference between Equation 5.31 and Equation 5.38 is in terms of the index. In Equation 5.31, the index k changes the semantics to compute the gradients with respect to the weights in different iterations. Take for instance the gradients at $i = 3$:

$$\begin{aligned} g_3 &= \sum_{k=1}^{3} m^{3-k} \cdot \frac{\partial \mathcal{L}_k}{\partial \theta_{k-1}} \\ &= m^2 \frac{\partial \mathcal{L}_1}{\partial \theta_0} + m \frac{\partial \mathcal{L}_2}{\partial \theta_1} + \frac{\partial \mathcal{L}_3}{\partial \theta_2} \end{aligned}$$

However, in Equation 5.38, the index i is constant, and the gradients are computed from different losses but with respect to the current weights only. Unfolding the same iteration:

$$\begin{aligned} g_3 = \frac{\partial \ell_3}{\partial \theta_2} &= \sum_{k=1}^{3} m^{3-k} \cdot \frac{\partial \mathcal{L}_k}{\partial \theta_2} \\ &= m^2 \frac{\partial \mathcal{L}_1}{\partial \theta_2} + m \frac{\partial \mathcal{L}_2}{\partial \theta_2} + \frac{\partial \mathcal{L}_3}{\partial \theta_2} \end{aligned}$$

Therefore, we cannot use loss scaling instead of momentum variables. Nevertheless, approximation of those variables using history of the loss is worth an investigation.

5.4 Chapter Summary

In this chapter, we reviewed the adaptive learning algorithms and provided detailed description on their update rules. Then, we established a general notation for update rules in adaptive learning algorithms that use moment estimation. We proved theoretically that given the general notation, some of the hyperparameters can be omitted and replaced by a scaling factor of the loss. Thereafter, we present the empirical evidence that we can make training more efficient when applying the scale factor. Finally, we showed that we cannot replace the variables with some average of the loss. In the next chapter, we'll introduce metalearning and use it to show that learnable loss will learn a proper scaling factor, and we provide layer-wise optimizers that learn the step size to make training more efficient.

Chapter 6

Step Size in Metalearning

We have set the optimizer parameters manually thus far and note that setting those parameters is not trivial. It requires human skill, trial and error, and in many cases, parameter search [17]. Even after finding proper parameter values, these values may be often difficult to interpret and explain, making it challenging to explain why a specific setting was chosen or how it affects the model performance. Nonetheless, two areas help with the problem, which are *hyperparameter optimization* and *meta learning* [18, 53]. They solve similar problems. However, hyperparameter optimization is restricted to producing better hyperparameters, whereas meta-learning is more general.

6.1 Metalearning

Metalearning, or learning to learn, uses past learning episodes or experiences to improve future learning, which is regarded as a more innate way of learning. Still, metalearning is an emerging field and even more so in the automation of certain aspects of machine learning. It faces a lot of challenges such as the lack of a firm mathematical foundation, the requirement of significant computational resources, and the difficulty of interpretation of the discovered settings. Nonetheless, the field provides promising solutions to the automation in different deep learning problem. It helps in various learning applications, such as few-shot learning which is concerned with learning from limited amount of examples for the same label or scenario. heterogeneous task learning in which the same model is trained on tasks that are diverse such different modalities like texts and images. Other applications include optimization, model compression, architecture search, and fast learning.

A simple example of machine learning is to *learn* the learning rate, i.e. $(\phi = \eta)$ [9, 66]. However, meta-learning can extend to the point where optimization is not required at inference time. For example, one can embed and then map an input to previously seen examples or to centroids of classes of observations and subsequently produce an output based on the closest match, something similar to *k-nearest neighbors* and *k-means* as done in the meta learners *matching networks* and *prototypical networks* [107, 113]. While such methods help to quickly solve new problems, the underlying learned representation still requires traditional learning.

We start with a general problem formulation, and then outline a widely used form of metalearning which involves two levels of optimization. Then, we use meta loss to learn a scaled loss, and apply meta learning to learn the proper step size in various metalearning settings.

6.1.1 Problem Formulation

In this section, we extend our discussion on optimization and establish a formulation for metalearning that learns an optimizer in few-shot learning. Without loss of generality, the formulation incorporates learning other types of representation, such as learning initial conditions and learning a loss. In Equation 3.6 we stated that an iterative optimization procedure will update the learnable parameters θ according to a function parameterized by non-learnable ϕ which denotes the optimizer's parameters:

$$\theta \leftarrow u(\theta, .; \phi) \tag{6.1}$$

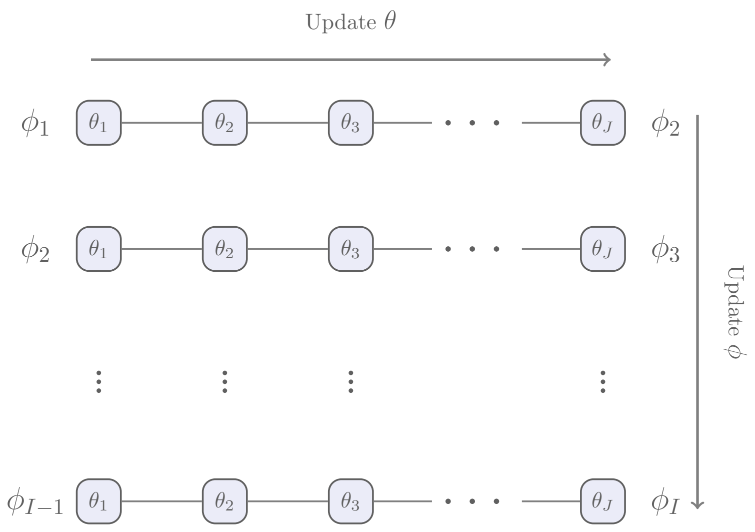


Figure 6.1: Updates of learned optimizer and optimizee.

In hand-engineered optimizers like SGD and adam, ϕ is set manually, but it is can be learned with metalearning.

Metalearning is conducted in two, possibly interleaved, phases: *meta-training phase* in which the parameters ϕ are updated, and *meta-testing phase* in which these parameters are evaluated. During meta-training, the learned optimizer observes the performance of an optimizee parameterized by θ. We call the learned optimizer the *outer learner* (ϕ) and we call the optimizee *inner learner* (θ). The outer learner then receives information from the inner learner and subsequently produces an update. After several inner updates, the outer optimizer is updated by judging the performance of θ. Sequence of updates during meta training are illustrated in Figure 6.1, starting form ϕ_1 that is updated to ϕ_2 after sequence of of inner updates of the optimizee from θ_1 to θ_J, and the outer updates continue until reaching a stopping criterion at ϕ_I. At the time of meta-testing, the inner learner is initialized and trained by the learned optimizer as we described in earlier chapters. This phase is just for testing and reporting the performance of the meta learner.

One important aspect is the way data is split during the two phases. Throughout both phases, data $\mathcal{D} = \{\mathcal{D}\}_1^T$ comes from tasks that follow the same distribution $\mathcal{T} = \{\mathcal{T}_1, \ldots, \mathcal{T}_i, \ldots, \mathcal{T}_T\}$ and $\mathcal{T}_i \sim P(\mathcal{T}_j)$. During meta-training phase, the dataset is split into training and validation $\mathcal{D}_{meta} = \mathcal{D}^{\mathrm{t}}_{meta} \cup \mathcal{D}^{\mathrm{v}}_{meta}$. The training and its evaluation is usually done in two nested loops. The inner loop updates θ using parameters ϕ for J times. We adapt the notation partially from reference [53]:

$$\begin{aligned} \theta^* = &\quad \arg\min_{\theta} f(\mathcal{D}^{\mathrm{t}}_{meta}; \theta) \\ \approx \theta_J = u(f(&\mathcal{D}^{\mathrm{t}}_{meta}; \theta_{J-1} \ldots (u(f(\mathcal{D}^{\mathrm{t}}_{meta}; \theta_0); \phi) \ ; \ \phi) \end{aligned} \tag{6.2}$$

The outer loop starts with an initialized ϕ that is updated after each inner loop using $\mathcal{D}^{\mathrm{v}}_{meta}$ and θ_J for I times, using an optimizer parameterized by Φ:

$$\begin{aligned} \phi^* = &\quad \arg\min_{\phi} u(\mathcal{D}^{\mathrm{v}}_{meta}; \phi) \\ \approx \phi_I = U(u(&\mathcal{D}^{\mathrm{v}}_{meta}; \phi_{I-1} \ldots (U(u(\mathcal{D}^{\mathrm{v}}_{meta}; \phi_0); \Phi) \ ; \ \Phi) \end{aligned} \tag{6.3}$$

Now that we have established a formulation, we present a detailed two-level optimization-based metalearning algorithm that can be extended to metalearning other representation.

6.1.2 Metalearning Optimization Algorithm

A general algorithm for meta leaning is outlined in algorithm 4. The algorithm describes the general meta learning algorithm. For an episode of meta learning on a single task (lines 5-9) please refer to Grefenstette, et al, [43] (page 6) who outlined a meta algorithm that covers a wide range of algorithms that are able to use diverging models and differentiable optimizers. Regular machine learning algorithm is part of the last loop (lines 12-15) which entails training on a single task and then testing once a model is produced. In meta learning, typically the set of evaluation tasks contains just a single task, $\mathcal{T}^{\mathrm{val}} = \{\mathcal{T}_1\}$.

Algorithm 4: Optimization-Based Meta Learning

```
Input  : $\phi$ meta optimizer.
         $\theta$: optimizee (model to be optimized)
         $\mathcal{T}$: set of tasks.
         episodes: learning episodes.

1  $\mathcal{T}^{\text{tr}}$, $\mathcal{T}^{\text{val}} \leftarrow$ split-sample($\mathcal{T}$)
   // meta-training
2  foreach $i$ in $I$ do
3      $\mathcal{D}_i \leftarrow$ get_data($\mathcal{T}_i$)
4      $\mathcal{D}^{\text{t}}_{meta}$, $\mathcal{D}^{\text{v}}_{meta} \leftarrow$ split-sample($\mathcal{D}_{meta}$)
5      foreach $j$ in $J$ do
6          $\theta \leftarrow$ meta-train($\theta, \phi, \mathcal{D}^{\text{t}}_{meta}$)
7      end
8      $\phi \leftarrow$ meta-test($\theta, \phi, \mathcal{D}^{\text{v}}_{meta}$)
9  end
   // meta-testing
10 foreach task $i$ in $\mathcal{T}^{val}$ do
11     $\mathcal{D}_i \leftarrow$ get_data($\mathcal{T}_i$)
12     $\mathcal{D}^{\text{tr}}$, $\mathcal{D}^{\text{ts}} \leftarrow$ split-sample($\mathcal{D}_i$)
13     $\theta \leftarrow$ train($\theta, \phi, \mathcal{D}^{\text{tr}}$)
14     test($\theta, \phi, \mathcal{D}^{\text{ts}}$)
15 end
```

6.2 Learned Representation

Metalearning is utilized to learn representation for different objectives, and the details of metalearning change accordingly. Learned representation ϕ, can vary from initial conditions to learning optimizers that produce update rules. In this section we will highlight some of the common learned representations and we name few methods that use metalearning for that representation.

Learning better initial conditions is common in few shot learning and produce better models for transfer learning. A popular method is model-agnostic meta-learning (MAML) in which the learned representation is the set of parameters that can generalize well to a new task [29]. In MAML, each θ in the inner loop is initialized to current ϕ; the inner update is:

$$\theta \leftarrow \theta - \eta_\theta \nabla_\theta f(\mathcal{D}^{\mathrm{t}}_{meta}; \theta) \tag{6.4}$$

The learned set of parameters in the outer loop in MAML is given by:

$$\phi \leftarrow \phi - \eta_\phi \nabla_\phi f(\mathcal{D}^{\mathrm{v}}_{meta}; \theta) \tag{6.5}$$

Each ϕ update depends on how well these parameters behave starting from their current point ϕ and ending with θ. The method influenced many other meta-learner algorithms that focus on learning better initialization [90].

On the other hand, learning a metric is relatively successful in which a similarity measure is learned to map data points to their nearest match and subsequently make a decision. Matching networks for few shot learning embed the examples during metatraining into a new space and then use an attention mechanism to match never-before-seen example to some previously seen input and subsequently predict its output [113]. Prototypical networks learns prototypes of each class and then maps new example to the correct prototypes [107].

Other common objectives that we are interested in are learning a loss, learning an optimizer and learning better hyperparamters, which we discuss in the following sections. We start with meta loss and we use it to show that learned loss can learn a proper loss scaling that compensates for the lack of learning rate inside the optimizer. We devote the last section for learning better update which include both learning hyperparamters and learning an optimizer. For a detailed discussion on metalearning, please refer to [12, 20, 53].

6.2.1 Learning A Loss

The selection of the right loss function for a task in hand can be sometimes challenging. The challenges include sensitivity to outliers, or to unbalanced data. Moreover, the loss tends to plateau after iterations of updates, which prevents the model from improving, or results in numerical instability. Learning a loss or reward in case of reinforcement learning is an ongoing exploration area in metalearning, especially that they are relatively easy to train and yield promising results.

Learned loss or reward is usually achieved by some form of deep learning optimization [11, 118], but it may be done using other techniques [39]. Task-adaptive loss function (metal) learns a loss function using MAML framework for few-shot learning. The loss function is a two dense layers with ReLu nonlinearity, regularized by other learnable parameters [8]. Extending the work of Bechtle, et al, [11], Raymond, et al, [92] suggested updating the loss function after learning to avoid overfilling the number of iterations used for training. Gao, et al, [31] use similar approach to learn a loss but learning with implicit gradients [71]. For physics-informed neural networks (PINNs) that are used to approximate partial differential equations (PDEs), Psaros, et al, [89] proposed using a learned loss instead of losses usually parameterized across various constraints. In the following section we introduce another few-shot learning learned loss proposed by Bechtle, et al [11].

6.2.1.1 Meta Loss

Bechtle, et al, suggest that instead of using loss functions, a learned loss called *Meta Learning via Learned Loss* (ML^3) may be used. The learned loss is a model g parameterized by θ and operates on two inputs: the produced output of the network $f(x;\theta)$, alongside the desired output y. The meta loss of θ is then:

$$\mathcal{L}_\theta = g(y, f(x;\theta);\phi) \tag{6.6}$$

Then, the model is updated using the gradients from the meta loss:

$$\theta \leftarrow \theta - \eta \nabla_\theta \mathcal{L}_\theta \tag{6.7}$$

The parameters of the loss model are finally updated using the loss generated by a well-defined loss function ($\mathcal{L}_\phi$) that is typically used for the task at hand such as the *mean squared error* for regression or *cross entropy loss* for classification, and that conclude the meta training phase:

$$\phi \leftarrow \phi - \eta_\phi \nabla_\phi \mathcal{L}_\phi \tag{6.8}$$

The inner and outer learning rates are different (η, η_ϕ) and different optimizers can be used. There is no test split during both meta training and meta testing. The reason is that the loss function is only used for training. The meta testing is basically described by Equation 6.6 and Equation 6.7.

We hypothesise that *if loss is learned, then we can omit the learning rate from the optimizer and the meta loss will learn a scaled loss that compensates for the learning rate.* We present our findings that empirically verify our hypothesis, but first we describe the model for meta loss. Unlike the work of Gonzalez and Miikkulainen [39], who noticed that the learned loss functions provide implicit regularization of the learning rate, we provide the mathematical foundation of the phenomena, which is that a loss scaling is a step size, and learned losses should learn a proper scale.

6.2.1.2 Experiment

Task I selected to work on the problem to approximate a sinusoidal wave from its x-axis input. A sinusoidal wave is a periodic function expressed by

$$f(x) = A\,sin(b(x - c)) + d \tag{6.9}$$

A is the amplitude or the height of the sinusoidal function, b affects the period of the wave which is the length of one complete cycle, c is the phase or the horizontal shift, and d is the vertical shift. The problem is important since providing good estimate can help with other periodic problems that exist in physics and finance among other fields.

During meta training, the model is trained on 100 samples in each task from the function $f(x) = \sin(x - \pi)$, ($A = 1$, $b = 1$, $c = \pi$, $d = 0$, and $x \in [-2.0, 2.0]$). During meta testing, the mode is tested on 100 samples in each task from the function $f(x) = A \sin(x - w)$, ($A \sim [0.2, 5.0]$, $b = 1$, $c \sim [-\pi, \pi]$, $d = 0$, and $x \in [-2.0, 2.0]$). There is no testing split in the data because the loss function is only used while training. The meta learner (loss model) is trained with Adam and learning rate of (0.001). The optimizee (optimized model) is trained with SGD and learning rate of (0.001).

Hardware and Software Settings The experiment is implemented using Python and Pytorch library. It is conducted on an Apple M1 chip device with 8 cores and 8GB of memory. The source code is cloned from the original paper source code linked from Github.

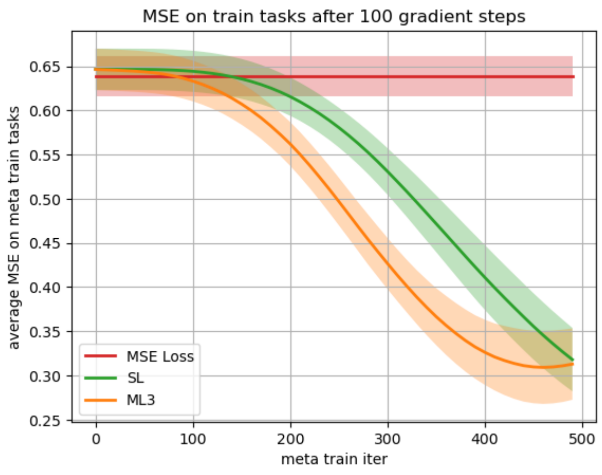
The confusion between the two loss models is avoided by referring to the paper's learned loss with the name they suggest (**ml3**) and my model with **sl**. For this experiment, **sl** should learn a loss that makes up for the step size given that we use an SGD without the learning rate. Therefore, I did not implement a custom optimizer but I just used a learning rate ($\eta = 1$) which is equivalent to not using a learning rate:

$$\theta - 1 \cdot g = \theta - g \tag{6.10}$$

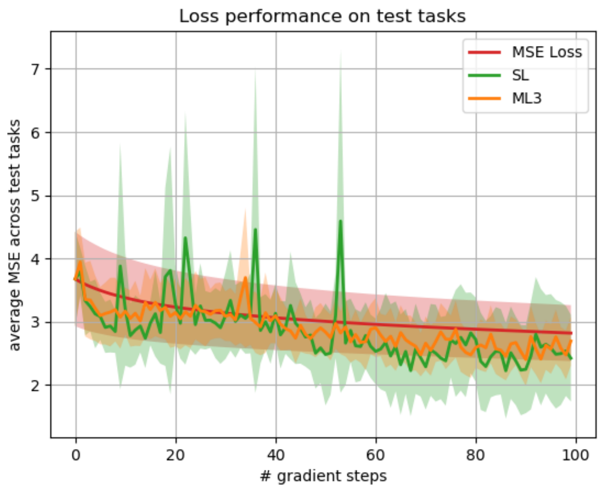
The reason is that this experiment is not to measure the speed or memory performance but rather the learning trajectory; whether the two meta learners will learn similarly or not.

Discussion In the first iteration, **sl** and **ml3** produce the same output as they initially have the same models. However, since **sl** is not used with a learning rate, the effect of the gradients computed from **sl** loss is greater on the parameters. I reduced the gain of **sl** parameters as well as the learning rate of its meta optimizer to control the behavior and to mimic scaling the loss. The result is shown in Figure 6.2.

It should be noted that, if we did not change the gain of the loss model, our approach (**sl**) produces similar loss trajectory to **ml3** but in a shorter time.



(a) Train tasks.



(b) Test tasks.

Figure 6.2: Loss trajectory.

However, the performance on training tasks starts to decrease after that point, something we also noticed happening in **ml3** after 500 episodes of training with the same settings. In the paper they stopped training once reaching that point. We can follow similarly and stop training earlier to achieve similar results.

6.3 Learning Improved Updates

We have introduced deep learning optimization in chapter 3 and chapter 5, and we stated that the non-learnable parameters in an optimizers are called hyperparamters. Notwithstanding the limited setting, hyperparamters refer to any setting in the environment ahead of training. That includes the number of layers, the number of neurons in each layers, activations, and regularization parameters. However, here we are interested in the optimizer's non-learnable parameters such as the learning rate. In this regard, learning optimizers and hyperparamteters are two close representations. We first introduce learning hyperparamters, restricting our discussion to parameters of an optimizer, and then move to learning optimizers.

6.3.1 Hyperparameter Learning

Hyperparameters are typically set by hand, and if not set manually, they may be discovered through random or grid search whose space is defined by each hyperparameter and its possible configuration [17]. Nevertheless, search is resource intensive and its input requires contextual understanding. Metalearning provide an elegant solution in which the learned representation is the proper hyperparamters. Meta learning has been extended to learn an optimizer's non-learnable parameters, which are the category of hyperparameters we are studying, and we can say that efforts in this area revolve around learning better learning rate.

For few-shot learning and as an extension to MAML, [66] introduced METASGD to learn adaptive inner learning rate, η_θ in Equation 6.4 . Baik et. al. [9] has also introduced another approach to lean an inner learning rate beside regularization hyperparameter. The work of Behl, et al [13], extends the inner learning rate to additionally learn the outer learning rate, η_θ and η_ϕ. Transfer learning demands for learning differently at different layers, and METALER introduced layerwise learning rates [22]. The method extends MAML and AUTOLR [94] to learn learning rates for each layer. The efforts offer simple means of learning learning rates, and next we move to another extreme presenting methods trying to learn a whole optimizer.

6.3.2 Optimizer Learning

Many efforts have primarily focused on learning optimization, especially since choosing the right optimizer and tuning its parameters demands experience. The work of Andrychowicz, et al [3] starts with the use of recurrent networks for their similarity with adaptive optimizers. The proposed meta optimizer consists of two layers of LSTM (long short-term memory) cells and a single dense layer, and the network receives the gradients as an input and produces an update. The representation ϕ is updated using the sum of the training losses as the main objective is to learn an optimizer that trains well. For few shot learning, Ravi and Larochelle [91] follows similarly by using a custom recurrent network that receive the gradients along with the loss. The representation is updated using the validation loss following the methodology in few-shot learning. other meta optimizers include use of other types of layers like convolution and attention [32, 33, 78]. Learning an optimizer that produces effective update rules can be quite beneficial, as it can accelerate learning and produce better models. However, learning optimizers have proved to be significantly challenging despite their application, and it remains an active research area [21, 76].

Learning better hyperparameters and better optimizers are related if we limit the former to optimization, but we should draw some distinctions. First, learning optimizers typically involves learning a model that learns to update, while learning optimization hyperparameters learns values that improve the optimizations. Subsequently, learning optimizers inherit all existing challenges of deep learning. Second, the input to hyperparameter learning is the hyperparamters and the output is an optimized version of those hyperparameters, but the input to optimizer learning is more complex. The input for optimizer learning is a function of the optimizee's parameters, alongside other information that may be regarded as relevant.

Having introduced both learning to optimize and learning better hyperparameters, we have devised a layerwise update rule that we introduce next named LURE.

6.3.3 Layerwise Update Rule

Learning a black-box optimization as described in subsection 6.3.2 is renowned for its difficulty [76]. However, the efforts we described in subsection 6.3.1 use primitive ways to learn better hyperparameters. To be specific, all those methods utilize scalars to learn better hyperparameters. I have named our method as LURE for Layerwise Update Rule. LURE is in the middle between learning hyperparameters and learning an optimizer, as it provides a layerwise update mechanism. For each layer, we created simple model consisting of one unit followed by a hyperbolic tangent (tanh) non-linearity. That provides a layerwise meta-learning for an optimizer, but here we draw the connection to hyperparameter learning.

Instead of providing a complex model, each layerwise model contains a single unit that corresponds to the learning rate. We enforce this domain knowledge by initializing the wights to positive uniform distribution scaled by (0.001). That prevents initial dropping of critical information regarding gradients direction. The non-linearity works to regulate the incoming gradients. Finally, the nonlinear transformation is multiplied by -1 to provide an update to the model that simulates gradient descent.

The model, or the optimizee is identical to the one used in the previous experiment, a three dense layer model with rectifier non-linearity (ReLU), Appendix C. I used the Jax library to help in simplifying the implementation greatly which otherwise get fairly complex and inconvenient.

We introduce a novel implementation that consists of a few lines providing general framework for custom layerwise optimizers that can adapt to any model structure. To our knowledge, there are no layerwise metalearners. Moreover, the architecture is usually flattened, resulting in loss of layer order information, i.e. which layer come first and which comes next. The implementation of the optimizer is shown in Codeblock 6.1.

First, the outer optimizer receives the gradients in form of PyTree, and the gradient tree is flattened into leaves and its original structure in line 4, with each leaf corresponding to a layer of the optimizee. To prevent an optimizer parameter growth with the number of parameters in the optimizee, Andrychowicz, et al [3] use coordinate wise parameter sharing, which is implemented in [91] by simply adding extra small dimensions to the input tensors. We follow a similar method but for each layer (line 7). Then for each layer, we created a model of one unit mimicking the learning rate (line 8), followed by a non-linearity (line 9). The hyperbolic tangent non-linearity maintains the original sign, a behavior desirable in processing gradients. Finally, we restructured the learned update back that it can be assigned directly to the optimizee's parameters.

```
1  class Opt(nn.Module):
2    @nn.compact
3    def __call__(self, tree):
4      leaves, structure = tree_flatten(tree)
5      result =[]
6      for i, leaf in enumerate(leaves):
7        leaf = leaf[..., None]
8        leaf = nn.Dense(1, kernel_init=uniform(0.001), use_bias=False,
      name=f'layer{i}')(leaf)
9        leaf = nn.tanh(leaf)
10       leaf = leaf * -1.0
11       result.append(leaf[..., 0])
12     t = tree_unflatten(structure, result)
13     return t
```

Codeblock 6.1: Dynamic optimizer for layerwise learning rate using Jax

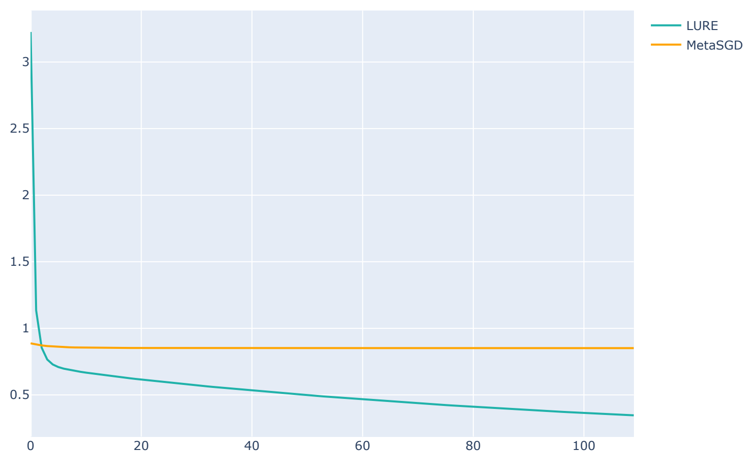
6.3.3.1 Experimental Setup and Results

Task Similar to the previous experiment, I used the method to approximate a sinusoidal wave. We use the same settings and training procedures in both LURE and METASGD. During meta training, the model is trained on 100 samples in each task from the function $f(x) = sin(x - \pi)$, ($A \sim [0.5, 2.0], b = 1, c = \pi, d = 0$, and $x \in [-5.0, 5.0]$). During meta testing, the model is tested on 100 samples in each task from the function $f(x) = A\,sin(x-w)$, ($A \sim [0.5, 2.0], b = 1, c \sim [-\pi, \pi], d = 0$, and $x \in [-5.0, 5.0]$). We trained the optimizer with 10000 episodes, and we reinitialized the optimizee after t=110 steps. During meta-testing, we trained the optimizee for t steps, and then we evaluate its performance at the last time step.

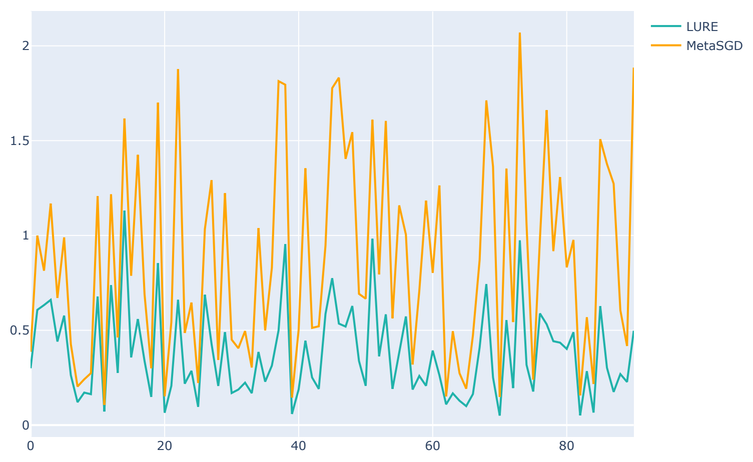
Hardware and Software Settings The experiment is implemented with Jax library. Jax is an open-source Python library providing an autodifferentiation mechanism for NumPy tensors, and it provides its own data structure called `PyTree` that facilitates the building of a computational graph [19]. Similar to the previous experiment, it is conducted on Apple M1 chip device, with 8 cores, and 8GB of memory.

Discussion The results shown in Figure 6.3 compare the performance of our method to MetaSGD [66], and the comparison is at meta-test time. The smooth lines in Figure 6.3a show the averaged optimization trajectory over episodes of learning, line 13 in algorithm 4. Our method shows a significant training improvement with a fast convergence. We also show the result of testing during meta-testing phase in Figure 6.3b, and the jagged line is due to the lack of averaging at different steps. Each point represents testing the optimizee trained by our proposed optimizer, corresponding to line 14 in algorithm 4.

Our proposed layerwise optimizer provides a general skeleton for any layer-wise optimizer. In particular, the two lines 8-9 in Codeblock 6.1 can be replaced by a more complex optimization model. In the following section, we use a more complex model using our proposed layer-wise optimizer to rescale the gradients after being scaled up in mixed precision training (MPT). In the following section,



(a) Average training loss during meta test time.



(b) Test loss during meta test time.

Figure 6.3: Performance of our layer-wise learned update.

we use a more complex model using in our proposed layer-wise optimizer to rescale the gradients after being scaled up in MPT.

6.3.4 Meta Gradient Rescaling for MPT

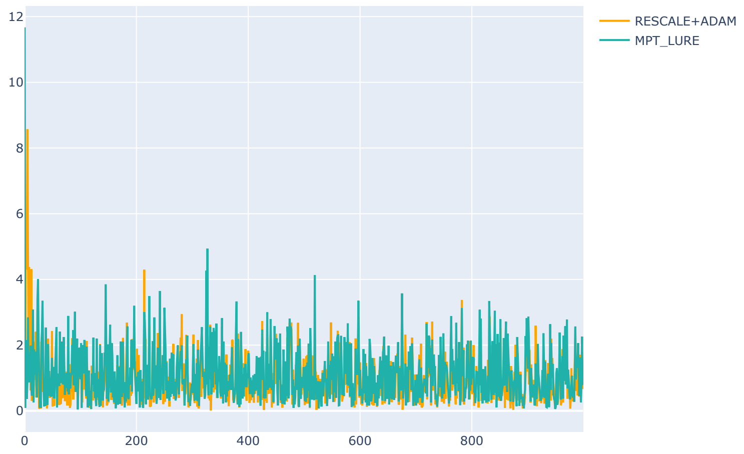
Although loss scaling is only present in mixed precision training research (subsection 4.2.2), it has not been connected explicitly to the step size. That results in redundant application of the learning rate. In this section, I use the same concept of layerwise update to rescale the gradients and produce an update for mixed precision training. We name the optimizer (MPT_LURE)

6.3.4.1 Experimental Setup

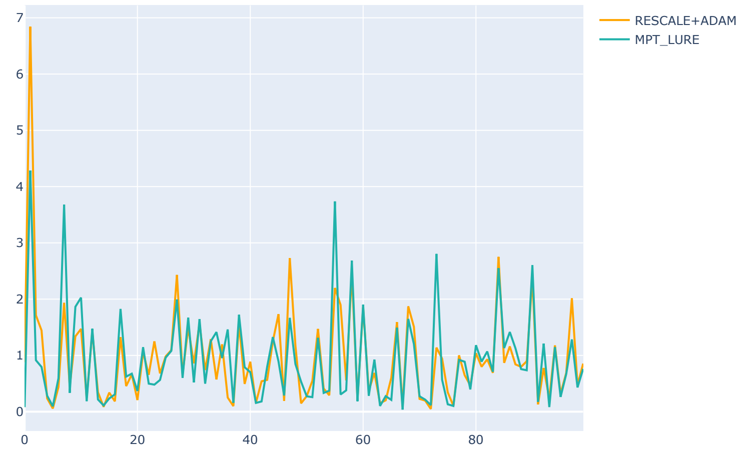
In this experiment, I first meta-trained the model that rescales the gradients from scaled loss. Once meta-training is done, I used the learned model as an optimizer that receives the scaled gradients and produces updates. I compare our optimizer to rescaling the loss and then using Adam optimizer [60].

Hardware and Software Settings I use the same settings as the previous experiment. However, in this experiment, I make use of the JMP library. This library provides utilities for MPT in Jax across platforms [7]. The MPT policy for this experiment is to use full precision format for parameters, and half precision format for forward and backward computations. The sine wave setting is exactly like the previous experiment, but we trained our model for only 6000 episodes. We use Adagrad to train our model with a learning rate of 0.001, we have not conducted meta-testing whilst training. We have introduced the concept of layerwise optimizer and we showed example in this section, so for the model used in this experiment, please refer to Codeblock C.2 in Appendix C.

Discussion The result of the experiment is shown in Figure 6.4, and we report regular training, or meta-testing after meta training is finished. Our proposed layerwise optimizer with minimal number of parameters, which are only two per layer, one is for learnable rescaling, and one for learnable learning



(a) Regular training loss.



(b) Test loss during meta test time.

Figure 6.4: Performance of our MPT_LURE learned update vs. rescaling.

rate. The results of our model that consists of only (12) learnable parameters show its competition to a more complicated process done in MPT, which is rescaling the gradient first and then using the rescaled gradients in an optimizer, which is in our case Adam optimizer. It is worth mentioning that Adam is continuously keeping (3522) parameters in the memory for the same model aside from its hyperparameters. For each one of the parameters in optimizee $\theta = \{(40, 40),\ (1600, 40),\ (40, 1)\}$, Adam maintains two copies, one for the first moment and one for the second moment as in Equation 5.3. In conclusion, our model show similar results with significantly less parameters and preprocessing.

6.4 Chapter Summary

Metalearning offers exciting possibilities for breakthroughs that can address difficult problems such as machine learning optimization. We started with developing a mathematical framework for metalearning, and outlined a general algorithm commonly used for metalearning. Then I provided examples and literature review of learned representations. I focused on three learned types of representation which are loss learning, hyperparameter learning, and optimizer learning and discussed each in some details. I introduced learned loss and used an example to show that the learned loss can learn a scaled loss given that we omit the learning rate from the optimizer while training. Second, I presented learned hyperparameters and learned optimizers and introduced the method that offers a means to create layerwise optimizer that transforms its input and learns improved updates. Finally, I tuned the method for mixed precision training to learn dynamic rescaling and updates.

Chapter 7

Discussion and Future Work

In this chapter, I will summarize the contributions of my dissertation and then, highlight the limitations and challenges that were encountered during the course of this dissertation. I will also present the opportunities for future research and improvement in the work.

7.1 Summary of Dissertation

Throughout this dissertation, we focus on making deep learning training more efficient by exploiting the fact that loss scaling works as a step size. We showed their equivalence in simple gradient descent and how to use loss scaling to adapt the step size in adaptive learning algorithms. We showed that by exploiting the presented relationship, we can save valuable resources during training. From this viewpoint, we showed that learnable losses adjust and learn the proper step size as well. Finally, and based on the relationship, we designed a learnable optimizer that can learn a proper rescaling loss factor and step size that make training more efficient.

7.2 Challenges and Limitations

I had planned to conduct experiments that have not been presented here and I will summarize them in two areas: learning dynamics and meta optimizers.

7.2.1 Learning Dynamics

Before embarking on the journey of this research, I proposed to scale the loss based on the architecture of the network including its depth, number of neurons in each layer, and activations. After starting the research, I found that the problem is more complex than it looks.

The problem involves the exploration of modes of evolution of complex systems over time. It is a widely recognized obstacle in other fields such as complex system analysis and has not been solved yet due to the many changing variables that should be taken into consideration. I spent precious amount of time attempting to solve this, but I stopped trying to solve the problem once I realized the extent of its complexity. The scope of the problem is beyond this research, but I will name a number of approaches trying to understand learning dynamics in deep learning.

Most of the efforts to solve the problem of learning dynamics focus primarily on exploring the limits of the deep learning system. Goldt, et al, attempted to reduce the number of layers [37] while Saxe, et al, tried to remove non-linear activations [99, 100]. Overparameterization also provides a tool to understand the learning dynamics [5, 38]. Also, drawing connection to similar fields that already try to understand systems helps [85]. Formalizing and framing learning dynamics in deep learning is unsolved, but attempts to understand it yields useful applications, like better initialization, activations, and topology.

7.2.2 Meta Optimizer

Our first intention was to develop extension to black-box optimizers such as done in the work of Andrychowicz, et al [3] and Ravi and Larochelle [91]. These optimizers have great potential but are rarely applied in practice. They have high level of complexity and it is not trivial to control their training, among other reasons. One of the reasons for them being notoriously difficult to manage is that they are black-box learners that inherit deep learning issues. When trying to implement those optimizers, we noticed a phenomena of that the learned optimizer is highly task specific, they somehow remember the examples they are trained to optimize long episodes of learning. This phenomenon is something similar to overfitting, but different in that it does not result in a decrease in the test performance. We assume it is related to the problem of overparameterization, which can be advantageous, but the problem is that the learning horizon in meta optimizers plateaus after some time, while the simple gradient descent optimizers like SGD continue to learn. The problem is interesting, and calls for more investigation.

Our proposed layerwise optimizer could be viewed as a black-box adaptation. However, meta optimizers described above do not apply domain knowledge. We provide a means of learning (black box) while keeping minimalistic number of parameters that we assume they learn specific representation, like learning the learning rate. In other words, we are not throwing many parameters and hoping that they learn improved update, that is way harder problem to learn and manage. Moreover, we control the initialization to force learning specific representation, like learning rate and rescaling. We provide random but controlled initialization, and learn as done in black box learning, but we know what each parameter is expected to learn. It is very safe gamble. That is why our meta learner is in between black-box optimizers and hyperparamter learning. Nonetheless, we believe that building more complex optimizer will help greatly, but we will continue our research in this area with optimizers that impose more knowledge derived from existed hand-engineered adaptive optimizers. We plan to continue exploring the problem that learned optimizers are task specific which can provide better understanding to deep learning in general.

7.2.3 Technical Hurdles

The obstacles that I have encountered while conducting this research and devising the experiments also spanned some technical aspects. First, I found that there was no plain gradient descent optimizer implemented in TensorFlow. All optimizers apply the learning rate, and using one is pointless. I tried to edit and contribute to TensorFlow open source library, but that part was specifically at the core of TensorFlow library for being essential. However, all essential code becomes a legacy and trying to fix that will take unnecessarily long time. Instead, I built a plain gradient descent optimizer as well as a regular optimizer and compared the results. I followed the same approach in comparing modified Adam and RMSProb to their regular counter parts.

Finally, PyTorch and TensorFlow provide reliable and fast implementation for deep learning. However, it was challenging to use them when implementing metalearning algorithms. Their computational graph is embedded inside the optimizer for fast execution. However, modifying and working with stationary graphs is cumbersome. It results in broken gradients and unnecessarily long code. Therefore, after some research with trial and error, I eventually settled with Jax which provides dynamic tools to implement advanced deep learning concepts.

7.3 Future Work

There are a lot of exciting future opportunities for exploration in metalearning. One in particular is the generalization of adaptive optimizers. We have provided a metalearning framework to learn layerwise updates that mimics simple gradient descent. However, we can extend it using the domain knowledge to construct a generalization of adaptive learning methods. That can simply be done by following the techniques advanced by Andrychowicz, et al [3] and Ravi and Larochelle [91] in using hidden states that retain update history, but we will differ in three aspects. First, we will provide updates that treat each layer differently. Second, we will reduce the complexity of the model greatly by using less parameters. Finally, and this point is related to the previous point, we will employ our knowledge in adaptive learning methods to learn simpler, yet powerful optimizers.

Last but not the least, this dissertation unfolds the effect of loss scaling and its relationship to the step size. Providing our theoretical analysis and empirical findings, it has become clear that loss scaling should play a bigger role in benefiting future optimization. In the end, the importance of loss scaling stems from the importance of loss itself, and although optimization efforts focus on gradients once the loss is found, loss can be better utilized for an ultimate improved learning.

Bibliography

[1] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard, et al. Tensorflow: A system for large-scale machine learning. In *12th {USENIX} symposium on operating systems design and implementation ({OSDI} 16)*, pages 265–283, 2016. URL.

[2] G. Alain and Y. Bengio. What regularized auto-encoders learn from the data-generating distribution. *The Journal of Machine Learning Research*, 15(1):3563–3593, 2014. URL.

[3] M. Andrychowicz, M. Denil, S. Gomez, M. W. Hoffman, D. Pfau, T. Schaul, B. Shillingford, and N. De Freitas. Learning to learn by gradient descent by gradient descent. In *Advances in neural information processing systems*, pages 3981–3989, 2016. URL.

[4] H. Araki. Relative entropy for states of von neumann algebras ii. *Publications of the Research Institute for Mathematical Sciences*, 13(1):173–192, 1977. URL.

[5] S. Arora, S. S. Du, W. Hu, Z. Li, R. R. Salakhutdinov, and R. Wang. On exact computation with an infinitely wide neural net. *Advances in neural information processing systems*, 32, 2019.

[6] À. R. Atrio and A. Popescu-Belis. Small batch sizes improve training of low-resource neural mt. *arXiv preprint arXiv:2203.10579*, 2022. URL.

[7] I. Babuschkin, K. Baumli, A. Bell, S. Bhupatiraju, J. Bruce, P. Buchlovsky,

D. Budden, T. Cai, A. Clark, I. Danihelka, A. Dedieu, C. Fantacci, J. Godwin, C. Jones, R. Hemsley, T. Hennigan, M. Hessel, S. Hou, S. Kapturowski, T. Keck, I. Kemaev, M. King, M. Kunesch, L. Martens, H. Merzic, V. Mikulik, T. Norman, G. Papamakarios, J. Quan, R. Ring, F. Ruiz, A. Sanchez, R. Schneider, E. Sezener, S. Spencer, S. Srinivasan, W. Stokowiec, L. Wang, G. Zhou, and F. Viola. The DeepMind JAX Ecosystem, 2020. URL.

[8] S. Baik, J. Choi, H. Kim, D. Cho, J. Min, and K. M. Lee. Meta-learning with task-adaptive loss function for few-shot learning. In *Proceedings of the IEEE/CVF International Conference on Computer Vision*, pages 9465–9474, 2021. URL.

[9] S. Baik, M. Choi, J. Choi, H. Kim, and K. M. Lee. Meta-learning with adaptive hyperparameters. *Advances in neural information processing systems*, 33:20755–20765, 2020.

[10] J. T. Barron. Continuously differentiable exponential linear units. *arXiv preprint arXiv:1704.07483*, 2017. URL.

[11] S. Bechtle, A. Molchanov, Y. Chebotar, E. Grefenstette, L. Righetti, G. Sukhatme, and F. Meier. Meta learning via learned loss. In *2020 25th International Conference on Pattern Recognition (ICPR)*, pages 4161–4168. IEEE, 2021. URL.

[12] J. Beck, R. Vuorio, E. Z. Liu, Z. Xiong, L. Zintgraf, C. Finn, and S. Whiteson. A survey of meta-reinforcement learning. *arXiv preprint arXiv:2301.08028*, 2023.

[13] H. S. Behl, A. G. Baydin, and P. H. Torr. Alpha maml: Adaptive model-agnostic meta-learning. *arXiv preprint arXiv:1905.07435*, 2019. URL.

[14] Y. Bengio. Practical recommendations for gradient-based training of deep architectures. In *Neural networks: Tricks of the trade*, pages 437–478. Springer, 2012. URL.

[15] Y. Bengio, N. L. Roux, P. Vincent, O. Delalleau, and P. Marcotte. Convex neural networks. In *Advances in neural information processing systems*, pages 123–130, 2006. URL.

[16] Y. Bengio, P. Simard, and P. Frasconi. Learning long-term dependencies with gradient descent is difficult. *IEEE transactions on neural networks*, 5(2):157–166, 1994. URL.

[17] J. Bergstra and Y. Bengio. Random search for hyper-parameter optimization. *Journal of machine learning research*, 13(2), 2012.

[18] B. Bischl, M. Binder, M. Lang, T. Pielok, J. Richter, S. Coors, J. Thomas, T. Ullmann, M. Becker, A.-L. Boulesteix, et al. Hyperparameter optimization: Foundations, algorithms, best practices, and open challenges. *Wiley Interdisciplinary Reviews: Data Mining and Knowledge Discovery*, page e1484, 2021.

[19] J. Bradbury, R. Frostig, P. Hawkins, M. J. Johnson, C. Leary, D. Maclaurin, G. Necula, A. Paszke, J. VanderPlas, S. Wanderman-Milne, and Q. Zhang. JAX: composable transformations of Python+NumPy programs, 2018.

[20] C. Chen, X. Chen, C. Ma, Z. Liu, and X. Liu. Gradient-based bi-level optimization for deep learning: A survey. *arXiv preprint arXiv:2207.11719*, 2022.

[21] T. Chen, X. Chen, W. Chen, H. Heaton, J. Liu, Z. Wang, and W. Yin. Learning to optimize: A primer and a benchmark, 2021.

[22] Y. Chen, J. Li, H. Jiang, L. Liu, and C. Ding. Metalr: Layer-wise learning rate based on meta-learning for adaptively fine-tuning medical pre-trained models. *arXiv preprint arXiv:2206.01408*, 2022. URL.

[23] D. C. Ciresan, U. Meier, J. Masci, L. M. Gambardella, and J. Schmidhuber. Flexible, high performance convolutional neural networks for image classification. In *Twenty-second international joint conference on artificial intelligence*, 2011. URL.

[24] D. R. Cox. The regression analysis of binary sequences. *Journal of the Royal Statistical Society: Series B (Methodological)*, 20(2):215–232, 1958. URL.

[25] D. Das, N. Mellempudi, D. Mudigere, D. Kalamkar, S. Avancha, K. Banerjee, S. Sridharan, K. Vaidyanathan, B. Kaul, E. Georganas, et al. Mixed precision training of convolutional neural networks using integer operations. *arXiv preprint arXiv:1802.00930*, 2018. URL.

[26] T. Dozat. Incorporating nesterov momentum into adam. In *Proceedings of 4th International Conference on Learning Representations*, 2016. URL.

[27] J. Duchi, E. Hazan, and Y. Singer. Adaptive subgradient methods for online learning and stochastic optimization. *Journal of machine learning research*, 12(7), 2011. URL.

[28] C. Dugas, Y. Bengio, F. Bélisle, C. Nadeau, and R. Garcia. Incorporating second-order functional knowledge for better option pricing. *Advances in neural information processing systems*, 13:472–478, 2000. URL.

[29] C. Finn, P. Abbeel, and S. Levine. Model-agnostic meta-learning for fast adaptation of deep networks. In *International conference on machine learning*, pages 1126–1135. PMLR, 2017. URL.

[30] R. A. Fisher. The use of multiple measurements in taxonomic problems. *Annals of Eugenics*, 7(2):179–188, 1936.

[31] B. Gao, H. Gouk, Y. Yang, and T. Hospedales. Loss function learning for domain generalization by implicit gradient. In *International Conference on Machine Learning*, pages 7002–7016. PMLR, 2022. URL.

[32] M. Garnelo, D. Rosenbaum, C. Maddison, T. Ramalho, D. Saxton, M. Shanahan, Y. W. Teh, D. Rezende, and S. A. Eslami. Conditional neural processes. In *International conference on machine learning*, pages 1704–1713. PMLR, 2018.

[33] E. Gärtner, L. Metz, M. Andriluka, C. D. Freeman, and C. Sminchisescu. Transformer-based learned optimization. *arXiv preprint arXiv:2212.01055*, 2022. URL.

[34] C. Gentile and M. K. Warmuth. Linear hinge loss and average margin. *Advances in neural information processing systems*, 11, 1998. URL.

[35] X. Glorot and Y. Bengio. Understanding the difficulty of training deep feed-forward neural networks. In *Proceedings of the thirteenth international conference on artificial intelligence and statistics*, pages 249–256, 2010. URL.

[36] D. Goldfarb, Y. Ren, and A. Bahamou. Practical quasi-newton methods for training deep neural networks. *Advances in Neural Information Processing Systems*, 33:2386–2396, 2020. URL.

[37] S. Goldt, M. Advani, A. M. Saxe, F. Krzakala, and L. Zdeborová. Dynamics of stochastic gradient descent for two-layer neural networks in the teacher-student setup. *Advances in neural information processing systems*, 32, 2019.

[38] S. Goldt, M. S. Advani, A. M. Saxe, F. Krzakala, and L. Zdeborová. Generalisation dynamics of online learning in over-parameterised neural networks. *arXiv preprint arXiv:1901.09085*, 2019.

[39] S. Gonzalez and R. Miikkulainen. Optimizing loss functions through multivariate taylor polynomial parameterization. In *Proceedings of the Genetic and Evolutionary Computation Conference*, pages 305–313, 2021. URL.

[40] I. Goodfellow, Y. Bengio, and A. Courville. *Deep Learning*. MIT Press, 2016. URL.

[41] I. Goodfellow, J. Pouget-Abadie, M. Mirza, B. Xu, D. Warde-Farley, S. Ozair, A. Courville, and Y. Bengio. Generative adversarial nets. In *Advances in neural information processing systems*, pages 2672–2680, 2014. URL.

[42] J. Gou, B. Yu, S. J. Maybank, and D. Tao. Knowledge distillation: A survey. *International Journal of Computer Vision*, 129:1789–1819, 2021.

[43] E. Grefenstette, B. Amos, D. Yarats, P. M. Htut, A. Molchanov, F. Meier, D. Kiela, K. Cho, and S. Chintala. Generalized inner loop meta-learning. *arXiv preprint arXiv:1910.01727*, 2019. URL.

[44] A. Griewank et al. On automatic differentiation. *Mathematical Programming: recent developments and applications*, 6(6):83–107, 1989. URL.

[45] M.-H. Guo, T.-X. Xu, J.-J. Liu, Z.-N. Liu, P.-T. Jiang, T.-J. Mu, S.-H. Zhang, R. R. Martin, M.-M. Cheng, and S.-M. Hu. Attention mechanisms in computer vision: A survey. *Computational Visual Media*, 8(3):331–368, 2022. URL.

[46] K. He, X. Zhang, S. Ren, and J. Sun. Delving deep into rectifiers: Surpassing human-level performance on imagenet classification. In *Proceedings of the IEEE international conference on computer vision*, pages 1026–1034, 2015. URL.

[47] K. He, X. Zhang, S. Ren, and J. Sun. Deep residual learning for image recognition. *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, Jun 2016. URL.

[48] J. L. Hennessy and D. A. Patterson. *Computer architecture: a quantitative approach*. Elsevier, 2017. URL.

[49] P. Hennig and M. Kiefel. Quasi-newton methods: A new direction. *The Journal of Machine Learning Research*, 14(1):843–865, 2013. URL.

[50] G. Hinton. The forward-forward algorithm: Some preliminary investigations. *arXiv preprint arXiv:2212.13345*, 2022.

[51] S. Hochreiter, Y. Bengio, P. Frasconi, J. Schmidhuber, et al. Gradient flow in recurrent nets: the difficulty of learning long-term dependencies, 2001. URL.

[52] S. Hochreiter and J. Schmidhuber. Long short-term memory. *Neural computation*, 9(8):1735–1780, 1997. URL.

[53] T. Hospedales, A. Antoniou, P. Micaelli, and A. Storkey. Meta-learning in neural networks: A survey. *IEEE transactions on pattern analysis and machine intelligence*, 44(9):5149–5169, 2021. URL.

[54] L. Huang, J. Qin, Y. Zhou, F. Zhu, L. Liu, and L. Shao. Normalization techniques in training dnns: Methodology, analysis and application. *arXiv preprint arXiv:2009.12836*, 2020. URL.

[55] P. J. Huber. Robust estimation of a location parameter. *Breakthroughs in statistics: Methodology and distribution*, pages 492–518, 1992.

[56] S. Ioffe and C. Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. *arXiv preprint arXiv:1502.03167*, 2015. URL.

[57] M. Jamil and X.-S. Yang. A literature survey of benchmark functions for global optimisation problems. *International Journal of Mathematical Modelling and Numerical Optimisation*, 4(2):150–194, 2013. URL.

[58] K. Janocha and W. M. Czarnecki. On loss functions for deep neural networks in classification. *arXiv preprint arXiv:1702.05659*, 2017. URL.

[59] J. Jumper, R. Evans, A. Pritzel, T. Green, M. Figurnov, O. Ronneberger, K. Tunyasuvunakool, R. Bates, A. Žídek, A. Potapenko, et al. Highly accurate protein structure prediction with alphafold. *Nature*, 596(7873):583–589, 2021. URL.

[60] D. P. Kingma and J. Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014. URL.

[61] G. Klambauer, T. Unterthiner, A. Mayr, and S. Hochreiter. Self-normalizing neural networks. *Advances in neural information processing systems*, 30, 2017. URL.

[62] A. Krizhevsky, I. Sutskever, and G. E. Hinton. Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems*, pages 1097–1105, 2012. URL.

[63] O. Kuchaiev, B. Ginsburg, I. Gitman, V. Lavrukhin, J. Li, H. Nguyen, C. Case, and P. Micikevicius. Mixed-precision training for nlp and speech recognition with openseq2seq. *arXiv preprint arXiv:1805.10387*, 2018. URL.

[64] Y. LeCun, B. Boser, J. S. Denker, D. Henderson, R. E. Howard, W. Hubbard, and L. D. Jackel. Backpropagation applied to handwritten zip code recognition. *Neural computation*, 1(4):541–551, 1989. URL.

[65] Z. Li, C.-Z. Lu, J. Qin, C.-L. Guo, and M.-M. Cheng. Towards an end-to-end framework for flow-guided video inpainting. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 17562–17571, 2022. URL.

[66] Z. Li, F. Zhou, F. Chen, and H. Li. Meta-sgd: Learning to learn quickly for few-shot learning. *arXiv preprint arXiv:1707.09835*, 2017.

[67] T. Liang, J. Glossner, L. Wang, S. Shi, and X. Zhang. Pruning and quantization for deep neural network acceleration: A survey. *Neurocomputing*, 461:370–403, 2021. URL.

[68] Q. Liao and T. Poggio. Bridging the gaps between residual learning, recurrent neural networks and visual cortex. *arXiv preprint arXiv:1604.03640*, 2016. URL.

[69] T.-Y. Lin, P. Goyal, R. Girshick, K. He, and P. Dollár. Focal loss for dense object detection. In *Proceedings of the IEEE international conference on computer vision*, pages 2980–2988, 2017. URL.

[70] D. C. Liu and J. Nocedal. On the limited memory bfgs method for large scale optimization. *Mathematical programming*, 45(1-3):503–528, 1989.

[71] J. Lorraine, P. Vicol, and D. Duvenaud. Optimizing millions of hyperparameters by implicit differentiation. In *International Conference on Artificial Intelligence and Statistics*, pages 1540–1552. PMLR, 2020. URL.

[72] A. L. Maas, A. Y. Hannun, and A. Y. Ng. Rectifier nonlinearities improve neural network acoustic models. In *Proc. icml*, volume 30, page 3, 2013. URL.

[73] A. Madani, B. Krause, E. R. Greene, S. Subramanian, B. P. Mohr, J. M. Holton, J. L. Olmos Jr, C. Xiong, Z. Z. Sun, R. Socher, et al. Large language models generate functional protein sequences across diverse families. *Nature Biotechnology*, pages 1–8, 2023. URL.

[74] D. Masters and C. Luschi. Revisiting small batch training for deep neural networks. *arXiv preprint arXiv:1804.07612*, 2018.

[75] N. Mellempudi, S. Srinivasan, D. Das, and B. Kaul. Mixed precision training with 8-bit floating point. *arXiv preprint arXiv:1905.12334*, 2019. URL.

[76] L. Metz, N. Maheswaranathan, J. Nixon, D. Freeman, and J. Sohl-Dickstein. Understanding and correcting pathologies in the training of learned optimizers. In *International Conference on Machine Learning*, pages 4556–4565. PMLR, 2019. URL.

[77] P. Micikevicius, S. Narang, J. Alben, G. Diamos, E. Elsen, D. Garcia, B. Ginsburg, M. Houston, O. Kuchaiev, G. Venkatesh, et al. Mixed precision training. *arXiv preprint arXiv:1710.03740*, 2017. URL.

[78] N. Mishra, M. Rohaninejad, X. Chen, and P. Abbeel. A simple neural attentive meta-learner. *arXiv preprint arXiv:1707.03141*, 2017. URL.

[79] V. Nair and G. E. Hinton. Rectified linear units improve restricted boltzmann machines. In *ICML*, 2010. URL.

[80] D. Narayanan, M. Shoeybi, J. Casper, P. LeGresley, M. Patwary, V. Korthikanti, D. Vainbrand, P. Kashinkunti, J. Bernauer, B. Catanzaro, et al. Efficient large-scale language model training on gpu clusters using megatron-lm. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–15, 2021. URL.

[81] Y. E. Nesterov. A method for solving the convex programming problem with convergence rate o (1/kˆ 2). In *Dokl. akad. nauk Sssr*, volume 269, pages 543–547, 1983.

[82] B. Neyshabur, S. Bhojanapalli, D. McAllester, and N. Srebro. Exploring generalization in deep learning. *Advances in neural information processing systems*, 30, 2017. URL.

[83] J. Ngiam, Z. Chen, D. Chia, P. W. Koh, Q. V. Le, and A. Y. Ng. Tiled convolutional neural networks. In *Advances in neural information processing systems*, pages 1279–1287, 2010. URL.

[84] O. Okwuashi and C. E. Ndehedehe. Deep support vector machine for hyperspectral image classification. *Pattern Recognition*, page 107298, 2020. URL.

[85] R. Pascanu, T. Mikolov, and Y. Bengio. On the difficulty of training recurrent neural networks. In *International conference on machine learning*, pages 1310–1318, 2013. URL.

[86] A. Paszke, S. Gross, S. Chintala, G. Chanan, E. Yang, Z. DeVito, Z. Lin, A. Desmaison, L. Antiga, and A. Lerer. Automatic differentiation in pytorch, 2017. URL.

[87] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, et al. Pytorch: An imperative style, high-performance deep learning library. In *Advances in neural information processing systems*, pages 8026–8037, 2019. URL.

[88] B. T. Polyak. Some methods of speeding up the convergence of iteration methods. *USSR Computational Mathematics and Mathematical Physics*, 4(5):1–17, 1964. URL.

[89] A. F. Psaros, K. Kawaguchi, and G. E. Karniadakis. Meta-learning PINN loss functions. *Journal of Computational Physics*, 458:111121, 2022. URL.

[90] A. Rajeswaran, C. Finn, S. M. Kakade, and S. Levine. Meta-learning with implicit gradients. *Advances in neural information processing systems*, 32, 2019. URL.

[91] S. Ravi and H. Larochelle. Optimization as a model for few-shot learning. In *International conference on learning representations*, 2017. URL.

[92] C. Raymond, Q. Chen, B. Xue, and M. Zhang. Online loss function learning. *arXiv preprint arXiv:2301.13247*, 2023. URL.

[93] M. Riedmiller and H. Braun. Rprop-a fast adaptive learning algorithm. In *Proc. of ISCIS VII), Universitat*. Citeseer, 1992. URL.

[94] Y. Ro and J. Y. Choi. Autolr: Layer-wise pruning and auto-tuning of learning rates in fine-tuning of deep networks. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 35:3, pages 2486–2494, 2021. URL.

[95] D. E. Rumelhart, G. E. Hinton, and R. J. Williams. Learning representations by back-propagating errors. *nature*, 323(6088):533–536, 1986. URL.

[96] D. E. Rumelhart, G. E. Hinton, R. J. Williams, et al. Learning representations by back-propagating errors. *Cognitive modeling*, 5(3):1, 1988. URL.

[97] T. Salimans and D. P. Kingma. Weight normalization: A simple reparameterization to accelerate training of deep neural networks. *Advances in neural information processing systems*, 29:901–909, 2016. URL.

[98] A. Santana and E. Colombini. Neural attention models in deep learning: Survey and taxonomy. *arXiv preprint arXiv:2112.05909*, 2021. URL.

[99] A. M. Saxe, J. L. McClelland, and S. Ganguli. Dynamics of learning in deep linear neural networks. In *NIPS Workshop on Deep Learning*, 2013.

[100] A. M. Saxe, J. L. McClelland, and S. Ganguli. Exact solutions to the non-linear dynamics of learning in deep linear neural networks. *arXiv preprint arXiv:1312.6120*, 2013. URL.

[101] T. Schaul, S. Zhang, and Y. LeCun. No more pesky learning rates. In *International conference on machine learning*, pages 343–351. PMLR, 2013.

[102] J. Schmidhuber. Deep learning in neural networks: An overview. *Neural networks*, 61:85–117, 2015. URL.

[103] R. M. Schmidt, F. Schneider, and P. Hennig. Descending through a crowded valley–benchmarking deep learning optimizers. *arXiv preprint arXiv:2007.01547*, 2020. URL.

[104] A. W. Senior, R. Evans, J. Jumper, J. Kirkpatrick, L. Sifre, T. Green, C. Qin, A. Žídek, A. W. Nelson, A. Bridgland, et al. Improved protein structure prediction using potentials from deep learning. *Nature*, 577(7792):706–710, 2020.

[105] K. Simonyan and A. Zisserman. Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556*, 2014. URL.

[106] L. 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*, 2018. URL.

[107] J. Snell, K. Swersky, and R. Zemel. Prototypical networks for few-shot learning. *Advances in neural information processing systems*, 30, 2017. URL.

[108] R. K. Srivastava, K. Greff, and J. Schmidhuber. Highway networks. *arXiv preprint arXiv:1505.00387*, 2015.

[109] D. Sussillo and L. Abbott. Random walk initialization for training very deep feedforward networks. *arXiv preprint arXiv:1412.6558*, 2014. URL.

[110] I. Sutskever, J. Martens, G. Dahl, and G. Hinton. On the importance of initialization and momentum in deep learning. In *International conference on machine learning*, pages 1139–1147, 2013. URL.

[111] T. Tieleman and G. Hinton. 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. URL.

[112] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Ł. Kaiser, and I. Polosukhin. Attention is all you need. *Advances in neural information processing systems*, 30, 2017.

[113] O. Vinyals, C. Blundell, T. Lillicrap, D. Wierstra, et al. Matching networks for one shot learning. *Advances in neural information processing systems*, 29, 2016.

[114] S. H. Walker and D. B. Duncan. Estimation of the probability of an event as a function of several independent variables. *Biometrika*, 54(1-2):167–179, 1967. URL.

[115] P. Werbos. Paul j. *Paul John. Beyond regression: new tools for prediction and analysis in the behavioral sciences*, 1, 1974. URL.

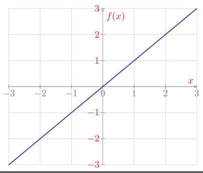
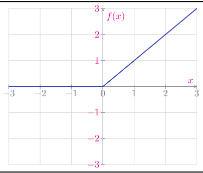
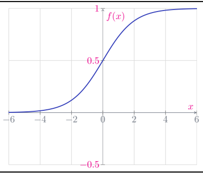
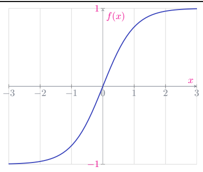
[116] M. Wiering, M. Schutten, A. Millea, A. Meijster, and L. Schomaker. Deep support vector machines for regression problems. In *International workshop on advances in regularization, optimization, kernel methods, and support vector machines*, pages 53–54, 2013.

[117] R. Zhao, B. Vogel, and T. Ahmed. Adaptive loss scaling for mixed precision training. *arXiv preprint arXiv:1910.12385*, 2019. URL.

[118] H. Zou, T. Ren, D. Yan, H. Su, and J. Zhu. Reward shaping via meta-learning. *arXiv preprint arXiv:1901.09330*, 2019. URL.

Appendix A

Activation Functions

Function	Description	Range	Graph
Identity	$f(x) = x$	$(-\infty, \infty)$	
Rectifier linear (ReLu)	$f(x) = max(0, x)$	$[0, \infty)$	
Logistic (sigmoid)	$f(x) = \frac{1}{1+\exp^{-x}}$	$(0, 1)$	
Hyperbolic tangent (tanh)	$f(x) = \frac{\exp^{x} - \exp^{-x}}{\exp^{x} + \exp^{-x}}$	$(-1, 1)$	

List of widely used activation functions with single dimensional input

Appendix B

Adaptive Optimizers

Optimizer	Hyper. Param.	Update Rule	
Plain SGD	η	**while not** *stop* **do**	
	g	$g = \frac{\nabla \theta}{m}$	*Compute gradient estimate*
		$\theta \leftarrow \theta - \eta g$	*Apply estimate*
		end	
Momentum	η, α	**while not** *stop* **do**	
	g, v	$g = \frac{\nabla \theta}{m}$	*Compute gradient estimate*
		$v \leftarrow \alpha v - \eta g$	*Compute velocity update*
		$\theta \leftarrow \theta + v$	*Apply velocity update*
		end	
Nesterov	η, α	**while not** *stop* **do**	
	$g, v, \hat{\theta}$	$\hat{\theta} \leftarrow \theta + \alpha v$	*Apply update on a copy*
		$g = \frac{\nabla \hat{\theta}}{m}$	*Compute gradient estimate at copy*
		$v \leftarrow \alpha v - \eta g$	*Compute velocity update*
		$\theta \leftarrow \theta + v$	*Apply velocity update*
		end	

AdaGrad	η, α g, r	**while not** *stop* **do** $g = \frac{\nabla\theta}{m}$ *Compute gradient estimate* $r \leftarrow r + g \odot g$ *Track magnitude history* $g \leftarrow \frac{1}{\sqrt{r}} g$ *Normalize to history* $\theta \leftarrow \theta - \eta g$ *Apply normalized gradients* **end**
RMSProp	η, α, ρ g, r	**while not** *stop* **do** $g = \frac{\nabla\theta}{m}$ *Compute gradient estimate* $r \leftarrow \rho r + (1-\rho) g \odot g$ *Weighted magnitude history* $g \leftarrow \frac{1}{\sqrt{r}} g$ *Normalize to history* $\theta \leftarrow \theta - \eta g$ *Apply normalized gradients* **end**
Adam	η, α, ρ g, r, s	**while not** *stop* **do** $t \leftarrow t + 1$ $g = \frac{\nabla\theta}{m}$ *Compute gradient estimate* $r \leftarrow \rho r + (1-\rho) g \odot g$ *Weighted magnitude history* $s \leftarrow \tau s + (1-\tau) \odot g$ *Weighted gradient history* $\hat{r} = \frac{r}{1-\rho^t}, \hat{s} = \frac{s}{1-\tau^t}$ *Correct histories* $g \leftarrow \frac{\hat{s}}{\sqrt{\hat{r}}}$ *Normalize gradients to magnitudes history* $\theta \leftarrow \theta - \eta g$ *Apply normalized histories* **end**

		1 **while not** *stop* **do**
		2 $g = \frac{\nabla\theta}{m}$ *Compute gradient estimate*
		3 $r \leftarrow \rho r + (1-\rho) g \odot g$ *Weighted magnitude history*
AdaDelta	η, α g, r, s	4 $g \leftarrow \frac{\sqrt{s}}{\sqrt{r}} \odot g$ *Normalize gradients by magnitude and scale it by history of normalized magnitude*
		5 $s \leftarrow \rho s + (1-\rho) g \odot g$ *Weighted normalized magnitude*
		6 $\theta \leftarrow \theta + g$ *Apply update*
		7 **end**

List of the update rules of widely used optimizers, steps are partially adapted from [40]

Appendix C

Code Blocks

```python
class Model(nn.Module):

  @nn.compact
  def __call__(self, x):
    x = nn.Dense(40, name="layer1")(x)
    x = nn.relu(x)
    x = nn.Dense(40, name="layer2")(x)
    x = nn.relu(x)
    x = nn.Dense(1,  name="layer3")(x)
    return x
```

Codeblock C.1: Model used in metalearning for regression tasks implemented in Jax

```python
class OptMPT(nn.Module):
  @nn.compact
  def __call__(self, tree):
    leaves, structure = tree_flatten(tree)
    result =[]
    for i, leaf in enumerate(leaves):
      leaf = leaf[..., None]
      leaf = nn.Dense(1, kernel_init= nn.initializers.uniform(scale=
    rescale), use_bias=False,name=f’layer_a{i}’)(leaf)
```

```
      leaf = nn.Dense(1, kernel_init=init2, use_bias=False, name=f'layer_b{i}')(leaf)
      leaf = nn.tanh(leaf)
      leaf = leaf * -1.0
      result.append(leaf[..., 0])
    t = tree_unflatten(structure, result)
    return t
```

Codeblock C.2: Dynamic optimizer for layerwise rescaling and learning rate using Jax