Learning to Backpropagate

Roy Henha Eyono

School of Computer Science and Applied Mathematics

University of the Witswatersrand

Supervised by: Assoc. Prof. Benjamin Rosman.

Dissertation submitted to the Faculty of Science, University of the Witwatersrand, in fulfillment of the requirements for the degree of

Master of Science

2020

Abstract

The backpropagation algorithm is regarded as the de-facto standard for gradient optimization in artificial neural networks. Since the conception of the method, modifications of the algorithm have been proposed. Adaptive learning rate methods are examples of custom modification to the backpropagation equation as they scale the gradient equation during training. Existing gradient modifications are largely based on theoretical fundamentals of gradient optimization, but few methods optimize for modifications that are based on data. We are motivated by the idea of discovering better custom gradient update equations for gradient optimization.

In this paper, we present our parametrized backpropagation learning framework (*PBLF*) which learns modifications of the backpropagation gradient update equation for stochastic gradient optimization. We achieve this by optimizing parts of the backpropagation equation to produce custom gradient update equations for gradient optimization. We evaluate our custom equations by training our target network on a validation dataset. In our dissertation, we provide empirical analysis and evidence to support *PBLF* as a competitive alternative to standard backpropagation.

In our experiments, we report competitive empirical performances on CIFAR10 with our custom gradient update equations sampled from *PBLF*. Our data-driven method offers promising custom update equations for gradient optimization.

Declaration

I, Roy Henha Eyono, hereby declare the contents of this research proposal to be my own work unless otherwise explicitly referenced. This research proposal is submitted for the degree of Master of Science (Dissertation) at the University of the Witwatersrand, Johannesburg. This work has not been submitted to any other university, nor for any other degree.

Signature

Date

Acknowledgments

This is an amalgamation of the efforts of many. I, alone, cannot take credit for this work. I would firstly like to thank my supervisor, Benjamin Rosman, for his support and for giving me a chance to explore this area of work. I am forever grateful for his continued support in my academic career and personal endeavours. To my close friends and family, Cedric, Genevieve, Tlou, Kwanda, and many others. This is for all those who identify themselves in this manuscript. They have sat with me and helped me when times were difficult. I greatly appreciate all your support. To my parents, this work is an ode to your unparalleled love and support. I am forever indebted to you and the home that you both fought so hard to build. Thank you for being amazing parents, and teaching me values that have guided me during the course of this dissertation and beyond. Thank you Mama and Papa. Scott, Ian and Tinty, I love you all.

Contents

List of Figures

Chapter 1

Introduction

The backpropagation algorithm [Rumelhart et al., 1986] is the de-facto standard for credit assignment in artificial neural networks. Credit assignment [Schmidhuber, 2015], in artificial neural networks, is the task of identifying neurons and weights that are responsible for the observed prediction from input data. In this work, we present a data-driven method that automatically learns custom gradient update equations for optimization in artificial neural networks.

Previous work on gradient credit assignment relies on theoretical expertise when designing custom gradient update modifications, but few optimize for custom modifications from data. Existing work on adaptive gradient learning rates [Kingma and Ba, 2014] [Zeiler, 2012] are prominent examples of custom gradient update equations for gradient optimization. These adaptive methods collect gradient information during optimization and progressively scale the gradient until convergence. The literature on adaptive learning rates suggest that there are empirical advantages in custom gradient equations, as these methods have become a staple in gradient optimization. However, current custom modifications of the gradient update equations are inspired from theory. We are interested in custom gradient update equations that are optimized from a validation dataset.

Backprop Evolution [Alber et al., 2018] is one such solution that addresses the need for custom gradient update equations from validation data. In backprop evolution, the method discovers custom gradient update equations by searching for new propagation rules from a domain-specific language that maximizes generalization performance after a few epochs of training. Backprop evolution explores through a domain-specific language (DSL) composed of operands, unary functions and binary functions to construct a variation of backpropagation.

In this dissertation, we seek to learn custom gradient update equations from a *continuous* search space of gradient update equations instead of a discrete repository

of operands, unary functions and binary functions as proposed in backprop evolution [Alber et al., 2018]. We call our approach, the parametrized backpropagation learning framework (*PBLF*). Our work hopes to build the foundations for learning new gradient update equations from a continuous parametrization of gradient update equations.

Our research question for this dissertation is the following:

- What are the benefits of designing a continuous search space for learning custom gradient update equations?

Understanding the benefits and shortfalls of adopting a continuous search space for learning custom gradient equations allows us to better search for compelling candidate solutions for credit assignment.

Our work coincides with current work in the field of meta-learning. Meta-learning is inspired by the concept of *learning to learn* developed in the field of psychology [Harlow, 1949]. Efforts have been made by researchers to build intelligent agents capable of *learning to learn* [Li and Malik, 2016, Finn et al., 2017, Zoph and Le, 2016, Andrychowicz et al., 2016, Lee et al., 2015]. *PBLF* shares similarities with this existing literature, as the framework, adopts the meta-learning approach for credit assignment in artificial neural networks. Informally, we refer to this as *Learning to Backpropagate*.

The dissertation is organized as follows:

- Chapter 2: We introduce the necessary background. We discuss the backpropagation algorithm and provide some background on gradient optimization for artificial neural networks.

- Chapter 3: We elaborate on the Backprop evolution work and its relevance in this dissertation.

- Chapter 4: We explore related work on credit assignment, particularly the alternatives and variants of the backpropagation algorithm. This chapter will provide context on current trends in credit assignment.

- Chapter 5: We present our parametrized backpropagation learning framework (*PBLF*).

- Chapter 6: We report our results and findings from experiments with our learning framework, *PBLF*.

- Chapter 7: We discuss possible extensions of this research project.
- Appendix A: We present the discrete repository of operands, unary and binary functions from the Backprop evolution method. We also provide preliminary analysis on the learned equations from Backprop evolution.
- Appendix B: We provide comprehensive results for our experiments on CIFAR10.
- Appendix C: We study the effects of perturbing activation function derivatives in the backpropagation equation.

Chapter 2

Background

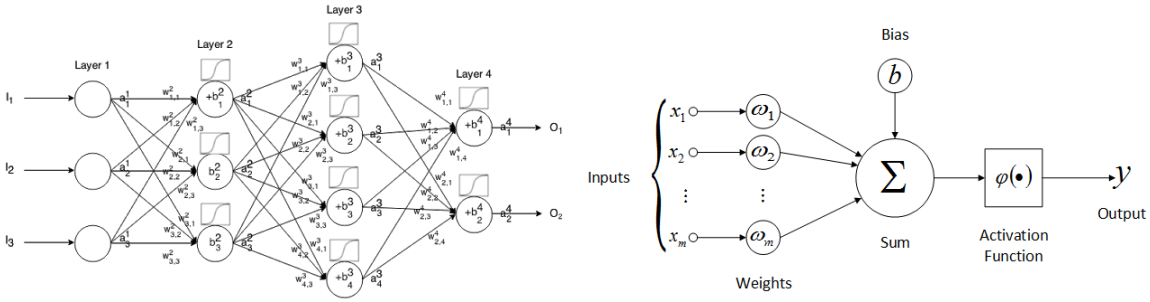
In this chapter, we discuss the fundamentals of the backpropagation algorithm as well as discuss existing gradient optimization algorithms. Section 2.1 provides some explanation on what a neural network is. This is followed by an elaborate introduction to the credit assignment problem and the backpropagation algorithm (Section 2.2 & 2.3). Lastly, Section 2.4 and 2.5 is a discussion on gradient descent and its variants.

2.1 The Artificial Neural Network

An Artificial Neural Network (ANN) [McCulloch and Pitts, 1943] is a mathematical model inspired from biological neural networks in animal brains. This model is intended to be a universal function approximator which learns from data.

The ANN is composed of units called artificial neurons, which are basic mathematical functions such as the tanh or linear function. These artificial neurons are connected by weighted connections. Each connection, like the synapses in a biological brain, transmit a signal from one artificial neuron to another. The overarching goal of a neural network is to receive some input or signal, propagate it through the neural network topology and finally receive a desired output. Figure 2.1 illustrates a neural network in further detail.

The initial motivation for ANN was to solve problems in a similar fashion to the human brain. Over-time, in the hopes of adapting the ANN to solve specific problems, the model has progressively deviated from biology. However, the artificial neural network still closely resembles the biological neural network in architecture, but falls short in the learning mechanism. In the subsequent chapter, we discuss the problem of learning in artificial neural networks also regarded as the credit assignment problem.



(a) Feed-Forward Artificial Neural Network (b) An Artificial Neuron

Figure 2.1: Subfigure (2.1a) [Sornam and Prabhakaran, 2017] is a schematic representation of an artificial neural network which is composed of weighted connections and artificial neurons. In subfigure (2.1b) [Tosun et al., 2016], a schematic visualization of an artificial neuron is illustrated. The firing procedure of an artificial neuron is also depicted in this illustration.

2.2 Credit Assignment

An artificial neural network (NN) consists of many simple, connected processors called neurons, each producing a sequence of real-valued activation's. Input neurons get activated through sensors perceiving an environment, hidden neurons get activated through weighted connections from previously activated neurons, and output neurons trigger actions. Credit assignment [Schmidhuber, 2015] in artificial neural networks is the task of identifying neurons and weights that are responsible for the desired prediction. Credit assignment is a mechanism that determines the role that each neuron contributes towards the outcome of the neural network.

A particular supervised learning credit assignment algorithm called the backpropagation algorithm [Rumelhart et al., 1986] is the standard for credit assignment surpassing previously proposed algorithms for the problem. The backpropagation idea is a method to minimize errors for a cost function through gradient descent. The authors [Rumelhart et al., 1986] had described this idea in their 1986 paper, and we discuss the backpropagation algorithm in further detail.

2.2.1 Backpropagation

Learning in an artificial neural network can be understood as minimizing a cost functions by adapting control parameters (weights). Rumelhart et al. [1986] describe backpropagation as a method to assign partial derivatives to individual weights and subsequently adapt the weights using gradient descent. These partial derivatives are computed with respect to the NN's cost function.

Artificial Neural Networks are parametrized by weights θ . The goal of backpropagation is to compute the partial derivative $\frac{\partial C}{\partial \theta}$ of the cost function C for a neural network $h_\theta(x)$, where x is input. Equation 2.1 is a simple example of a quadratic cost function used commonly in NN's. In Equation 2.1, y is the label, x is a data input and finally n is the number of training example.

$$C = \frac{1}{2n} \sum_x ||y(x) - h_\theta(x)||^2 \tag{2.1}$$

In the backpropagation procedure, each layer l is assigned an error term δ^l. For each layer, the error term δ^l is derived from error terms in subsequent layers $\delta^{l+1\ldots L}$, hence the concept of error backpropagation. The output layer L is the only layer whose error term δ^L has no error dependencies, hence δ^L is the element-wise product of the partial derivative of the cost function C with respect to the activation a_L at the output layer L and the activation function derivative σ' parametrized by the layer preactivation z^L. The error term δ^L is illustrated in Equation 2.2.

$$\delta^L = \nabla_a C \odot \sigma^{'}(z^L) \tag{2.2}$$

Regarding the error term δ^l, this term is derived from matrix multiplying the subsequent layer's term δ^{l+1} with the weight transpose matrix $(\theta^{l+1})^T$ and subsequently multiplying (element-wise) the activation function derivative σ' parametrized by the layer preactivations z^l. This is illustrated in Equation 2.3.

$$\delta^l = (\theta^{l+1})^T \delta^{l+1} \odot \sigma^{'}(z^l) \tag{2.3}$$

Once the layer error terms have been allocated, the partial derivative $\frac{\partial C}{\partial \theta}$ can be computed as illustrated in Equation 2.4. A schematic visualization of the backpropagation procedure is illustrated in Figure 2.2.

$$\frac{\partial C}{\partial \theta^l} = \delta^{l+1} (a^l)^T \tag{2.4}$$

The backpropagation algorithm is solely responsible for computing weight partial derivatives. Beyond assigning partial derivatives to weights to sufficiently learn, the weights of the neural network need to be adapted. Gradient descent is a method used in conjunction with the backpropagation algorithm to perform weight adaptation. We further discuss this method in Section 2.3.

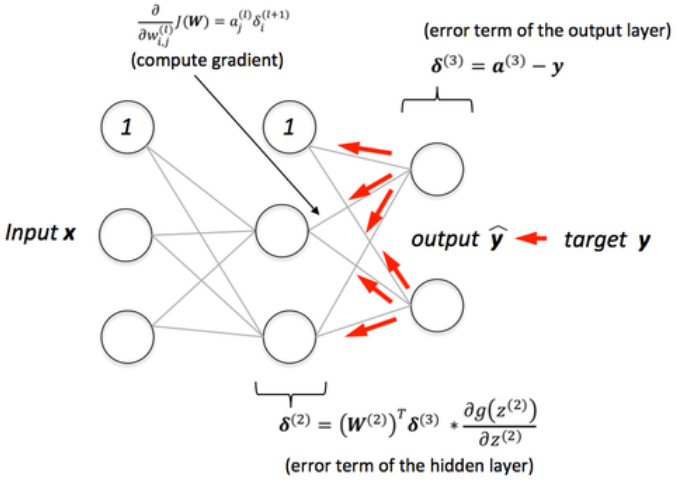


Figure 2.2: Schematic visualization of the backpropagation procedure (Image Source: sebastianraschka.com/faq/docs/visual-backpropagation.html).

2.3 Gradient Descent

Cost or Loss Functions are functions that measure how well a neural network fits a data distribution from a sample of data. This definition applies across the various types of cost functions [Rosasco et al., 2004] defined for either supervised, unsupervised or semi-supervised tasks. Learning in artificial neural networks is understood as minimizing the network's cost function. Gradient descent [**?**] is an algorithm that leverages gradients to iteratively identify local minima in a cost function.

Gradient descent is an optimization algorithm used to minimize a function by iteratively moving in the direction of steepest descent as defined by the negative of the gradient. The step in the direction of the steepest descent can similarly be varied in magnitude, this magnitude is often called the step size or learning rate and is characterized by the value α. The gradient descent algorithm is illustrated in Equation 2.5, where n represents the iteration.

$$\theta_{n+1} = \theta_n - \alpha \nabla C(\theta_n) \tag{2.5}$$

There are two general types of gradient descent, namely **batch gradient descent** and **stochastic gradient descent**. In batch gradient descent, the partial derivative $\frac{\partial}{\partial \theta_n} C(\theta_n)$ is computed one for the entire dataset. Once this partial derivative is computed, a single step is taken in the negative direction of the gradient.

On the other spectrum, stochastic gradient descent (SGD) takes the approach of iteratively computing the gradient $\frac{\partial}{\partial \theta_n} C(\theta_n)$ for small batches of the entire dataset.

Algorithm 1 Backpropagation with Stochastic Gradient Descent

Require: Training Set $\{(x^{(1)}, y^{(1)})...(x^{(m)}, y^{(m)})\}$
Require: Initialize weights θ_{ij}^{l} randomly. [He et al., 2015]
Require: Layer Activations $a^{(l)}$ where l is the layer index.
1: **for** $i = 1$ to m **do**
2: Initialize first layer activation: $a^{(1)} \leftarrow x^{(i)}$
3: Perform forward computation to compute a^{l} for $l = 2, 3, ..., L$
4: Using $y^{(i)}$, compute $\delta^{L} = a^{L} - y^{(i)}$
5: Compute $\delta^{(L-1)}, \delta^{(L-2)}, ..., \delta^{(2)}$
6: $\theta^{l} = \theta^{l}$ - $\alpha \delta^{l+1} (a^{l})^{T}$

SGD, although noiser, can escape local minima easily and it is for this reason that it is preferred over vanilla gradient descent for optimization. Algorithm 1 illustrates how the backpropagation algorithm and stochastic gradient descent are used to train artificial neural networks.

Improvements on SGD have been proposed and adopted particularly due to the difficult of choosing a sufficient learning rate. In Section 2.4, we discuss these variants.

2.4 Variants of Stochastic Gradient Descent

Stochastic Gradient Descent has seen many improvements, many of which are currently used in practice. It's particularly the learning rate in SGD that has seen the most improvement, due to its sensitivity. Setting the learning rate too high can cause divergence and conversely setting the learning rate too low can stagnate convergence. In this section, we explore popular improvements on this gradient optimization technique. A more comprehensive survey of the various techniques has been curated by Ruder [2016].

2.4.1 Momentum

The momentum method [Qian, 1999], commonly known as SGD with momentum, stems from momentum in physics. Essentially, SGD with momentum remembers the update $\Delta\theta$ at each optimization step and determines the next optimization step as a linear combination of the gradient and the previous update. The update for SGD with momentum is described in Equation 2.6.

$$\Delta\theta_n = \beta\Delta\theta_{n-1} - (1 - \beta)\nabla C(\theta_n) \tag{2.6}$$

The term β in Equation 2.6 is the momentum parameter which essentially scales the effect of momentum on the gradient update. The weight update equation is described in Equation 2.7 where α is the learning rate. Momentum keeps the gradient traveling in the same direction, ultimately preventing oscillations. This technique is widely used in current implementations of SGD.

$$\theta_{n+1} = \theta_n - \alpha\Delta\theta_n \tag{2.7}$$

Nestorov momentum or Nesterov Accelerated Gradient [Nesterov, 1983] is a slight modification of the momentum method. Nestorov momentum calculates the gradient for cost $C(\theta_n + \beta\Delta\theta_{n-1})$ instead of cost $C(\theta_n)$. Nestorov momentum is often understood as performing a look-ahead gradient evaluation and then performing a correction [Sutskever et al., 2013]. Nestorov's momentum has gained popularity in use over the vanilla momentum method.

2.4.2 Adam

The Adaptive Moment Estimation (Adam) optimizer [Kingma and Ba, 2014] is another algorithm which adapts the learning rate for each parameter. The algorithm is often considered a combination of RMSProp and momentum as the algorithm stores an exponentially decaying average of past squared gradients v_t like RMSProp and also keeps an exponentially decaying average of past gradients m_t like momentum. Both v_t and m_t are described in equations 2.8 and 2.9 respectively.

$$m_t = \beta_1 m_{t-1} - (1 - \beta_2)\nabla C(\theta_n) \tag{2.8}$$

$$v_t = \beta_2 v_{t-1} - (1 - \beta_2)\nabla C(\theta_n)^2 \tag{2.9}$$

Both m_t and v_t are initialized as vectors of zero's, this results in a biased towards zero especially during the initial time steps. To counteract this effect, a bias-corrected estimate of both m_t and v_t is needed, these estimates are identified as $\hat{m}_t$ and $\hat{v}_t$ respectively. These estimates are described in Equation 2.10.

$$\hat{m}_t = \frac{m_t}{1 - \beta_1}, \hat{v}_t = \frac{v_t}{1 - \beta_2} \tag{2.10}$$

The update rule for the Adam optimizer can be simply understood as SGD with momentum, normalized by a decaying average of past squared gradients. This is illustrated in Equation 2.11.

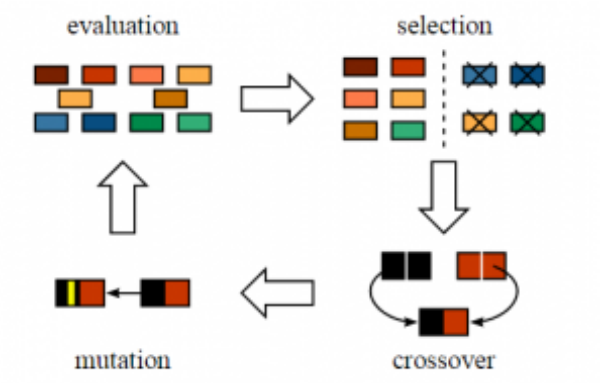


Figure 2.3: [Hu, 2020] Schematic visualization of a genetic algorithm.

$$\theta_{n+1} = \theta_n - \frac{\alpha}{\sqrt{\hat{v}_n} + \epsilon}\hat{m}_t \tag{2.11}$$

In Equation 2.11, ϵ accounts for the case where $\sqrt{v} = 0$, as is the case with RMSProp.

2.5 Neuroevolution

Neuroevolution is an area in artificial intelligence that uses evolutionary algorithms to generate neural networks architectures and learn network parameters. Neuroevolution is a gradient-free credit assignment algorithm, inspired by evolution in nature. Neuroevolution algorithms are often considered an alternative to gradient optimization techniques like backpropagation with SGD.

Neuroevolution has been used extensively for designing neural network architectures [Stanley et al.]. The authors [Stanley et al.] have compiled an extensive review of the various applications of neuroevolution for designing network architectures.

A popular neuroevolution method is the genetic algorithm [Mitchell, 1998], which gains inspiration from the procedure of natural selection in evolution. Genetic algorithms generate solutions (weight parameters, architectures, etc..) through optimization using bio-inspired operators such as mutation, crossover and selection. Figure 2.3 illustrates details on how a genetic algorithm functions.

The genetic algorithm framework for learning hyper-parameters in neural networks is as follows:

1. ***Sample*** a set of parameters and ***evaluate*** each set of parameters. The first batch of samples is commonly referred to as a generation.

2. ***Select*** the best performing parameters.

3. ***Mutate*** and ***crossover*** the best performing parameters to populate the next generation.

4. ***Re-evaluate*** the new generation. **If** unsatisfied with the current generation return to Step 2, **Else**: Goto Step 5.

5. ***Choose*** best performing solution in this generation and **Stop**.

The genetic algorithm framework illustrated in Figure 2.3 closely relates the neuroevolution framework. Neuroevolution can be loosely understood as a genetic algorithm where the solutions are the weights or architecture of a network.

2.6 Bayesian Optimization

Bayesian Optimization (BO) [Močkus, 1975] is a technique for optimizing black-box function especially when the function is noisy and expensive to evaluate. BO addresses the shortfalls of random search by using past evaluations to choose the next query point. BO is an integral component of *PBLF*, hence understanding the algorithm is a necessary prerequisite.

BO consists of two components: a *statistical surrogate model* to model the unknown black box function and an *acquisition function* which trades off between exploration and exploitation to decide the next query point that meets our optimization goal (maximization/minimization).

2.6.1 Statistical Surrogate Model

A surrogate model is a necessary requirement to optimize our black box function f. Below are some constraints of black-box functions that motivate the need for a surrogate model:

- f is expensive to evaluate.
- f is noisy.
- f is continuous and non-convex.
- f is non-differentiable.

A surrogate model of our black-box function f allows for us to circumvent the aforementioned constraints. The Gaussian Process (GP) [Rasmussen, 2004] is the default choice for modeling black-box functions for BO. GP's can construct a prior distribution over functions $p(f|X)$ and compute a posterior distribution $p(f|X, y)$ after having observed some data y. The posterior is used to make predictions f_* given new input X_* which is necessary for our acquisition function. Other lesser popular techniques for black-box modeling include Random Forest [Criminisi et al., 2011] and t-Student Processes [Shah et al., 2014].

$$g(x) \sim GP(\mu(x), k(x, x')) \tag{2.12}$$

Formally, we describe the GP surrogate model $g(x)$ of f by Equation 2.12, where x is the black-box function query point, $\mu(x)$ the mean function and $k(x, x'; \theta)$ the covariance function with respect to the covariance parameters θ. Given data D, the posterior mean $\mu(x; \theta, D)$ and variance $\sigma(x; \theta, D)$ can be computed explicitly, which will be used as arguments towards the acquisition function.

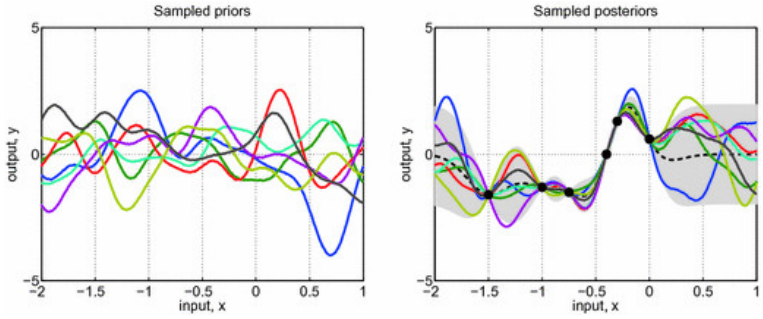


Figure 2.4: [Mateo et al., 2016] Gaussian Process. Left: Functions drawn from the GP prior. Right: Functions drawn from the posterior.

2.6.2 Acquisition Function

An acquisition function $a(x)$ is an inexpensive function that evaluates how desirable it is to evaluate f at point x for the optimization objective. We could select x arbitrarily but this would be wasteful. In BO, the acquisition function is essential in determining the next query point for evaluation from the statistical surrogate model. The next evaluation point will allow for us to update our posterior in our surrogate model. Acquisition functions are designed to trade off exploration of the search space and exploitation of current promising areas.

In this subsection, we discuss three popular acquisition functions: *Probability of Improvement (PI)*, *Expected Improvement (EI)* and *Upper Confidence Bound (UCB)*.

Probability of Improvement (PI) is the first acquisition function designed for BO. In practice, PI gets stuck in local optima and under-explores globally. Although it is not often used in practice, PI serves the basis for the EI acquisition function. Suppose

$$f' = min(\mathbf{f}) \tag{2.13}$$

is the optimal value of the surrogate model $\mathbf{f}$ observed so far. PI evaluates query points most likely to improve on f'. PI achieves this by using the cumulative distribution function at each query point x to determine the probability $P(f(x) \geq f')$. This translates to the following PI acquisition function:

$$a_{PI}(x) = \int_{-\infty}^{f'} \mathcal{N}(f; \mu(x), k(x,x))df \tag{2.14}$$

The point x with the maximum probability of improvement is selected. PI samples query points that most likely improve on f' regardless of the size of the improvement resulting in local optima and under-exploration. Expected Improvement (EI) seeks to circumvent the shortfalls of PI by accounting for the size of the improvement. This translates to a new variation of Equation 2.14:

$$a_{EI}(x) = \int_{-\infty}^{f'} (f' - f)\mathcal{N}(f; \mu(x), k(x,x))df \tag{2.15}$$

where the query point x with the highest EI is selected.

An alternative to the EI and PI acquisition functions is the Upper Confidence Bound (UCB). The UCB acquisition function is a linear combination of the mean and variance of query point x:

$$a_{UCB}(x) = \mu(x) + \beta\sqrt{k(x,x)}, \tag{2.16}$$

where β is a trade-off parameter. In our experiments, we use the UCB acquisition function because it has been shown that the iterative application of this acquisition function will converge to the true global minimum of f [Slivkins, 2019].

2.6.3 Exploration-Exploitation Tradeoff

The exploration-exploitation trade-off is a well studied constraint popularized by the multi-armed bandit problem [Katehakis and Veinott Jr, 1987]. In the multi-armed bandit problem, each machine provides a random reward from a probability

distribution specific to that machine. The objective is to maximize the sum of rewards earned through a sequence of lever pulls. The crucial tradeoff the gambler faces at each trial is between *exploitation* of the machine that has the highest expected payoff and *exploration* to get more information about the expected payoffs of the other machines.

Bayesian Optimization is a favoured solution to the multi-armed bandit problem because the mean $\mu(x)$ and variance $k(x, x)$ terms in BO acquisition functions can be explicitly interpreted as the trade-off between exploitation (evaluating at points with low mean) and exploration (evaluating at points with high uncertainty). BO acquisition functions are designed to tackle this trade-off efficiently in search.

Bayesian Optimization is similarly a favoured solution for *PBLF* because of its simplicity and its ability to trade-off exploration and exploitation in search, an important trait towards learning unique backpropagation equations.

2.7 Summary

In this chapter, we discussed the necessary background on backpropagation and credit assignment in artificial neural networks, including the various gradient optimization techniques used in practice. Knowledge of the backpropagation algorithm and gradient optimization is beneficial towards understanding how our learning framework in *PBLF* (Chapter 4) learns unique backpropagation equations for gradient optimization.

In the subsequent chapter, we discuss alternatives and variants to the backpropagation algorithm in the form of related work. In the related work chapter, we discuss research that has followed since the conception of the backpropagation algorithm. The related work will help to provide context on alternative approaches to credit assignment in artificial neural networks.

Chapter 3

Backprop Evolution

Beyond PBLF, two solutions that automatically learn custom gradient update equations are Target propagation [Lee et al., 2014] and Backprop evolution [Alber et al., 2018]. An understanding of Target propagation, in addition to a more comprehensive understanding of Backprop evolution, will provide the needed context for our learning framework. In this chapter, we delve into these two related methods.

Target Propagation: The main idea of target propagation is to associate, with each neuron activation, a local target value. This target value serves as a replacement for the global loss gradient. To illustrate this concept, let us construct a simple neural network:

$$h_i = \sigma(W_i^T h_{i-1}), \quad i = 1, ..., L \tag{3.1}$$

where h_i is the state of the i^{th} hidden layer, W_i is the weight matrix at i^{th} hidden layer, and σ is the hidden activation function. Not that $h_0 = x$ which is the input data and L represents the index for the last layer of the network.

Target propagation learns a target value $\hat{h}_i$ for each i^{th} hidden layer, and defines a local layer target loss for example, the MSE Loss (L):

$$L_i(\hat{h}_i, h_i) = ||\hat{h}_i - h_i(x; W_i^T)||_2^2 \tag{3.2}$$

With this local layer loss function, W_i can be updated locally within its layer via stochastic gradient descent, where $\hat{h}_i$ is considered as a constant with respect to W_i:

$$W_i^{(t+1)} = W_i^t - \eta_i \frac{\partial L_i(\hat{h}_i, h_i)}{\partial W_i} \tag{3.3}$$

,η_i is a layer-specific learning rate.

Despite its ability to learn custom gradient update values, the critical limitation with Target Propagation is its inability to scale to complex datasets [Bartunov et al.,

2018]. The authors [Bartunov et al., 2018] assess the scalability of biologically motivated deep-learning algorithms and show that Target Propagation and Feedback Alignment perform significantly worse than standard backpropagation on the CIFAR dataset. We discuss Feedback Alignment in our related work. However, this is a limitation that was addressed by Backprop evolution, as the method reports competitive results on complex datasets like CIFAR10.

Backprop Evolution: The method discovers new variations of the backpropagation equations by learning new propagation rules that optimize the generalization performance after a few epochs of training. The authors find several update equations that report similarly to standard backpropagation at convergence.

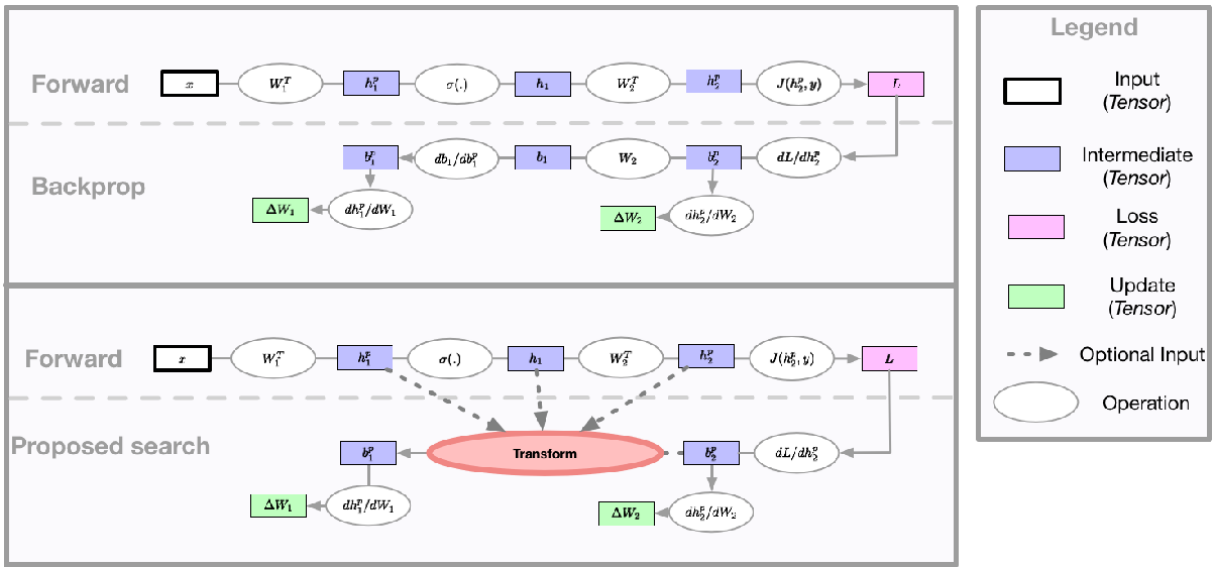


Figure 3.1: [Alber et al., 2018] Backprop evolution: the main contribution is to use evolutionary methods to find a custom computational graph for gradient optimization by learning parts of the backpropagation computational graph.

[Alber et al., 2018] define a simple neural network as follows:

$$h_i^p = W_i^T h_{i-1}, \qquad h_i = \sigma(h_i^p) \tag{3.4}$$

where $h_0 = x$ is the input to the network, i indexes the layer and W_i is the weight matrix of the i-th layer. The authors similarly define the partial derivative functions b_i and b_i^p as modules in the backpropagation equations:

$$b_L = \frac{\partial J(f(x), y)}{\partial h_L}, \qquad b_{i+1}^p = b_{i+1}\frac{\partial h_{i+1}}{\partial h_{i+1}^p}, \qquad b_i = b_{i+1}^p W_{i+1}^T \tag{3.5}$$

Note that $b_i^p = \frac{\partial J(f(x),y)}{\partial h_i^p}$. Backprop evolution leverages this notation to break up the backpropagation equation into modular parts. For convenience, we will interchangeably address these partial derivatives functions as modules in the backpropagation equation. Weight updates are computed as follows: $\Delta W_i = b_i^p \frac{\partial h_i^p}{\partial W_i}$. These backpropagation functions are depicted in the computational graph illustrated in Figure 3.1. In this work, the authors modify the backpropagation computational graph (Figure 3.1) and use a search method to find better formulas for b_i^p.

Backprop evolution searches through a domain-specific language (DSL) composed of operands, unary functions and binary functions to learn b_i^p. The authors learn unique DSL equations for b_i^p by searching through the space of DSL equations using an evolutionary algorithm. We provide an exhaustive list of the DSL search space in Appendix A.1. The fitness of an update equation solution, in the context of evolution, is measured on a validation test set.

The search was conducted on Wide ResNets with 16 layers and a width multiplier of 2 (WRN 16-2) with SGD (with and without momentum) initially for 20 epochs and then subsequently on 100 epochs to test generalization on longer training times. The search was also conducted on WRN 28-10 networks for 20 epochs to test generalization to larger models since WRN 28-10 has 10x more parameters than WRN 16-2 networks. In our experiments, we also make use of WRN 28-10 to evaluate our framework.

Experimental results showed that for specific scenarios, Backprop evolution learned gradient update equations that provide competitive empirical performance at convergence. In Appendix A.2, we offer some analysis for the custom gradient update equations sampled by Backprop evolution. We demonstrate that custom gradient update equations are analogous to learning an implicit learning rate. The analysis is still preliminary but could potentially help us build more robust learning frameworks.

In the next chapter, we part discuss related algorithms and methods that address shortfalls with the standard backpropagation algorithm.

Chapter 4

Related Work

In this chapter, we provide context on the backpropagation algorithm. We discuss alternatives and variants of the algorithm since its inception. This chapter may also serve as background as some of these techniques will be used in experiments conducted in this dissertation. In Section 3.1, 3.2 & 3.3, we present alternatives to the backpropagation algorithm, where we discuss their benefits and shortfalls. Finally, in Section 3.4, we present the various variants of the backpropagation algorithm.

4.1 Decoupled Neural Interfaces: Synthetic Gradients

In backprop, gradient updates in a layer can only occur once all the subsequent modules of the network have been executed, and gradients from subsequent layers have been backpropagated to it. This phenomena is commonly known as update locking as illustrated in Figure 4.1.

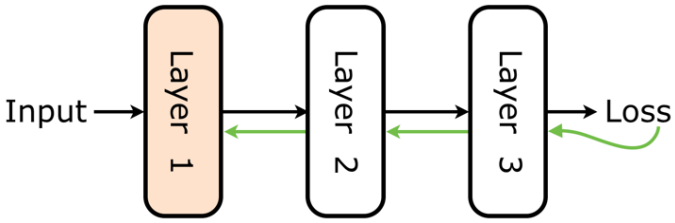
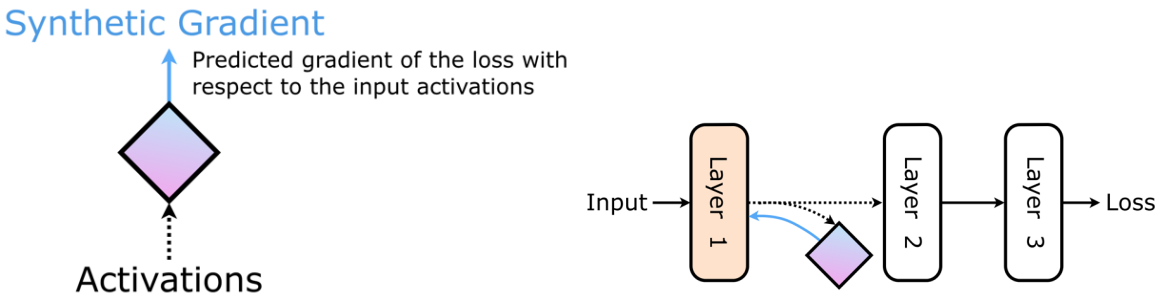


Figure 4.1: [Jaderberg et al., 2016] A simple illustration of update locking in neural networks. Layer 1 is reliant backpropagated error signals from Layer 3 to perform an update.

[Jaderberg et al., 2016] propose the mechanism of synthetic gradients to decouple update locking by providing asynchronous gradient updates to the layers of a network.



(a) (b)

Figure 4.2: [Jaderberg et al., 2016] The synthetic gradient module (Figure 4.2a), often just a single layer neural net, provides layer-wise gradient predictions with respect to the networks cost function as illustrated in Figure 4.2b.

The synthetic gradient model predicts the gradient of the loss of the network with respect to the layer activation's at each layer in a neural network. This model allows for asynchronous gradient updates at each layer, relaxing the update locking constraint found in standard backpropagation. Figure 4.2 illustrates this model quite neatly.

Although updates are performed asynchronously, the synthetic gradient module still requires the true gradient to be backpropagated as an error signal to the synthetic module to improve on its predictions. This is performed over many training iterations. However the backprop algorithm remains slightly more better empirically than the proposed synthetic gradient model. Nevertheless, the approach remains a novel solution to the problem of update locking found in standard backpropagation.

4.2 Feedback Alignment

Feedback alignment [Lillicrap et al., 2016] is a variant of the backpropagation algorithm that uses a randomly initialized matrix instead of the weight transpose matrix in the backpropagation equation to obtain gradient values for weights in the network. This algorithm is widely considered more biologically plausible to the standard backpropagation procedure.

The motivation for its biological plausibility is that standard backpropagation stores information of the synaptic strength of all weights at each weight update to compute its gradient, it is believed that the brain can't afford to transport(or backpropagate) that amount of information across its dendrites, this is known as the weight transport problem. Hence, randomly initialized matrices for backpropagation is more

biologically plausible as there is no need to transport weight information across dendrites at every weight update. Figure 4.3 provides an illustration comparing feedback alignment against backpropagation.

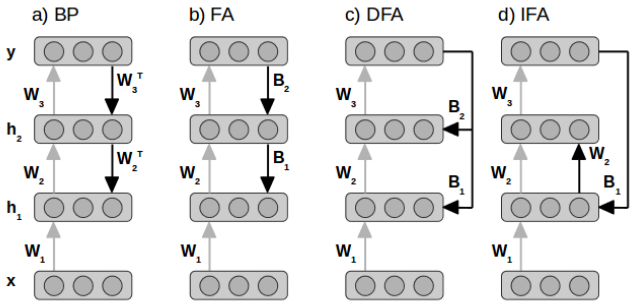


Figure 4.3: [Nøkland, 2016] Illustration comparing backpropagation (BA) against feedback alignment (FA) and its variants, Direct Feedback Alignment (DFA) and Indirect Feedback Alignment (IFA).

Feedback Alignment (FA) and its variants, Direct Feedback Alignment (DFA) and Indirect Feedback Alignment (IFA), demonstrate the flexibility of the backpropagation algorithm. Let B_k represent a randomly generated fixed random matrix B at the k_{th} layer. As illustrated in Figure 4.3, FA uses a random B matrix in place of the weight transpose matrix W^T. With DFA, each layer does not depends on gradients from subsequent layers but solely the last layer with a random matrix B_k. In contrast, IFA computes the gradient signal of an early layer using the last layer, then gradients are forwardly propagated with the weight matrix W instead of the weight transpose W^T. These algorithms dispel previously held assumptions about the backpropagation algorithm and propose variants of the algorithm that provide competitive results on datasets like MNIST and CIFAR10. Although these methods fail to scale to CIFAR100 and ImageNet [Bartunov et al., 2018], they stretch our imagination on what's possible and reignite the previously lost connection with biology.

4.3 Summary

In this chapter, we discussed the various alternatives and variants of the backpropagation algorithm. We've similarly spoken on the known alternatives, namely neuroevolution. Despite the alternatives, backprop remains the de-facto standard for credit assignment in artificial neural networks. The variants, largely brain-inspired,

provide interesting solutions to the credit assignment problem. They dispel previously held assumptions about backprop and improve on the algorithms shortfalls (i.e. update locking, life-long learning, memory). This chapter aimed to provide context for the backpropagation algorithm as a solution for credit assignment in artificial neural networks.

In this dissertation, we are particularly interested in automatically learning custom update equations for stochastic gradient optimization. In the following chapter, we present our framework, *PBLF*, to demonstrate a feasible solution to learning custom update equations that compete empirically against standard backpropagation.

Chapter 5

Methodology

We propose our framework, the parametrized BP learning framework (PBLF), to construct custom gradient update equations for stochastic gradient optimization (SGD). We achieve this by learning parts of the BP equation for stochastic gradient optimization, particularly the linear activation function derivative. We construct a continuous search space of gradient update equations for optimization and use Bayesian optimization to efficiently sample custom update equations that optimize on the predefined validation dataset.

In this chapter, we describe our learning framework and investigate the differences in our method against Backprop evolution. We conclude this chapter by discussing the potential benefits of custom gradient optimization for artificial neural networks

5.1 Parametrized Backpropagation Learning Framework

In this section, we describe our framework, which learns custom gradient update equations for stochastic gradient optimization. Our framework searches parts (modules) of the backpropagation equation to produce custom gradient values that optimize the generalization performance of a neural network.

Instead of learning the module $\frac{\partial J(\theta)}{\partial h_i^p}$ in Backprop evolution, as described in Chapter 3, we learn the linear activation function derivative $\sigma'_\phi(h_i^l)$ in the backpropagation equation. σ' represents the linear activation function derivative, l represents the layer, i represents the neuron index and h represented the layer pre-activation. We parametrize $\sigma'_\phi(h_i^l)$ by the parameter ϕ to induce variation in the function. We chose the linear activation function derivative since all neural network architectures, deterministic and stochastic, implicitly have linear activation functions at each layer l. To

illustrate this, let us define h_i as the state of the i^{th} hidden layer:

$$h_i = \sigma(W_i^T h_{i-1}), \quad i = 1, ..., L \tag{5.1}$$

W_i is the weight matrix at i^{th} hidden layer, and σ is any arbitrary hidden activation function. We define the linear activation function as $\sigma_{linear}(x) = x$. Hence by definition,

$$\sigma_{linear}[\sigma(W_i^T h_{i-1})] = \sigma(W_i^T h_{i-1}), \quad i = 1, ..., L \tag{5.2}$$

Equation 5.2 shows that implicitly each layer has a linear activation. Learning the linear activation function derivative is a general solution for modifying parts of the backpropagation equation. Similarly, $\sigma'_{linear}(x) = 1$ is a simple derivative. In our experiments, we similarly optimize for which parts of the backpropagation equation to learn.

To learn $\sigma'_\phi(h_i^l)$, we search ϕ with Bayesian Optimization [Močkus, 1975] until the generalization performance of the model is optimized. Optimizing for ϕ is analogous to optimizing the gradient update equation, hence for convenience we denote the gradient equation by $\tilde{g}_\phi(\theta_t) = J'(\theta_t; \phi)$.

PBLF is similar in implementation to the synthetic gradient method (defined in Section 4.1) in that both methods have secondary neural network models that perform backpropagation for the target network, both methods refer to their secondary networks as meta-network. The difference in both methods is that *PBLF* learns custom gradient values while the synthetic gradient method seeks to approximate the standard gradient.

We chose Bayesian Optimization, described in Section 2.7, because of its recent successes [Chen et al., 2018a][Snoek et al., 2012] and empirical benefits in low-dimensional spaces for expensive black-box evaluations [Wang et al., 2013]. Following convention, we use Gaussian process (GP) [Rasmussen, 2004] priors over functions as our probabilistic prior model and Expected Improvement (EI) [Močkus, 1975] as our acquisition function for exploration.

With *PBLF*, we construct a continuous search space of gradient update equation governed by the parameter ϕ in $\tilde{g}_\phi(\theta_t)$, hence we are not limited to Bayesian optimization. Given that we've parametrized the backpropagation equation $\tilde{g}_\phi(\theta_t)$, we can leverage other search algorithms. In our experiments, we run experiments with random search to compare against Bayesian optimization.

5.1.1 Details

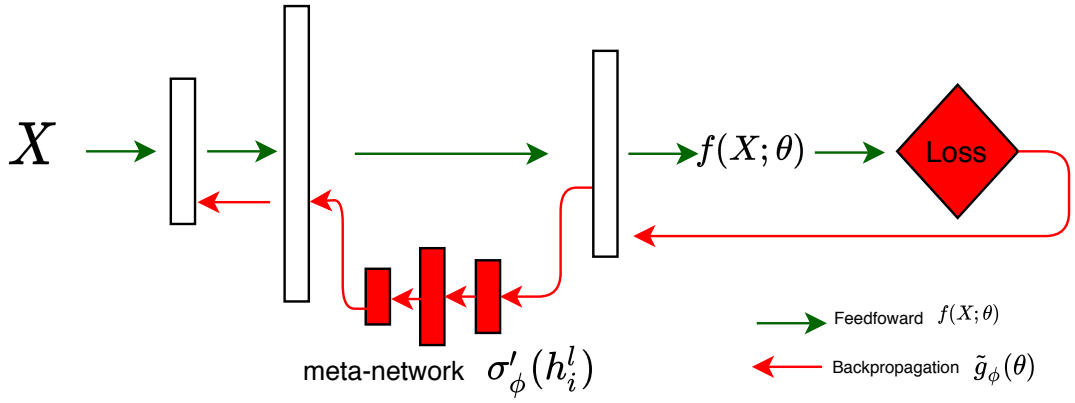


Figure 5.1: We learn $\sigma'(h_i^l;\omega)$ with Bayesian optimization. $\sigma'(h_i^l;\omega)$ is a meta-network parametrized by weight ω. The parameter h_i^l is the $l-$layer pre-activation at the ith neuron. $f(X;\theta)$ represents the target network parametrized by weight θ and input X. In our experiments, $f(X;\theta)$ is the WideResNet architecture.

PBLF consists of **three components**:

- **A target network** $f(X;\theta)$: which is the subject of the experiment, this could be a convolution neural network. The weights θ are learned with the backpropagation equation $\tilde{g}_\phi(\theta_t)$.
- **The meta-network** $\sigma'_\phi(h_i^l)$: a neural network with weights ϕ, which explicitly denotes the linear activation derivative and implicitly represents our parametrized backpropagation equation $\tilde{g}_\phi(\theta_t)$. The parameter ϕ is optimized to produce custom gradient update equations.
- **The optimization component**: which learns $\tilde{g}_\phi(\theta_t)$ for the target network $f(X;\theta)$. The evaluation function for the optimization component is the accuracy of $f(X;\theta)$ on a held-out validation set.

Together, these components optimize for a custom gradient update equation $\tilde{g}_\phi(\theta_t)$ that maximizes the generalization performance of the target network $f(X;\theta)$. We demonstrate this in Algorithm 2. Similarly, Figure 5.1 illustrates a parametrized backpropagation equation $\tilde{g}_\phi(\theta_t)$ for a target network $f(X;\theta)$.

We acknowledge that we can learn custom gradient update equations by learning an end to end gradient update equation. We followed the framework of Target propagation (Chapter 3) and Feedback Alignment (Section 4.2), which both make

modifications to parts of the backpropagation equation. However, to approximate learning an end to end update equation, in our method, we learn the appropriate number of meta-networks and their optimal positions in the backpropagation equation. This allows us to learn the optimal parts of the backpropagation equation for modification.

Algorithm 2 Parametrized Backpropagation Learning Framework (PBLF)

Require: Learning rate α
Require: Tasks $\mathcal{T}$ and $\mathcal{T}'$
randomly initialize ϕ and θ
for $iteration = 1, 2, \ldots$ **do**
    re-initialize θ
    for $epochs = 1, 2, \ldots, N$ **do**
        Evaluate $\tilde{g}_\phi(\theta) = J'(\theta; \phi)$ with respect to task $\mathcal{T}$.
        Perform gradient descent: $\theta' \leftarrow \theta - \alpha \tilde{g}_\phi(\theta_t)$
    end for
    Compute accuracy $\mathcal{A}$ for $f(X; \theta)$ on task $\mathcal{T}'$.
    Update ϕ such that $\mathcal{A}$ is maximized.
end for
    return ϕ =0

In Appendix C, we study the effects of parametrizing an activation function derivative in the backpropagation equation. We perform this study by parametrizing the ReLU derivative and perturbing its parameters in training for a simple binary classification task. We illustrate the gradient trajectories across each perturbation of the ReLU derivative to study the effects in gradient optimization. We found that perturbations to the activation function derivative is similar to perturbing the learning rate of a neural network in training.

5.2 Backprop Evolution vs PBLF

Backprop Evolution [Alber et al., 2018] and *PBLF* both learn custom gradient update equations for gradient optimization. Although backprop evolution leverage evolutionary algorithms instead of Bayesian optimization, both techniques ultimately learn modular parts of the backpropagation equation.

The key differentiating factor for *PBLF* is the continuous parametrization of the backpropagation equation. The choice for a continuous search space parametrized by an artificial neural network presents a more general approach to optimizing for

custom gradient update equations. In Backprop evolution, the search space for gradient update equations is a repository of operands and operations. We provide an exhaustive list of the repository in Section 3.2. Although Backprop evolution presents an expertly designed discrete search space, there are a few limitations with a discrete search space of update equations:

- A discrete repository has a discrete number of states while continuous spaces have an uncountably infinite number of states.
- Difficult to compute derivatives with a discrete search space. Hence, no gradient optimization (refer to Section 2.4).
- Difficult to construct a universal function approximation from a discrete search space of operands and operations.

These are limitations that *PBLF* aims to circumvent with its continuous search space of gradient update equations. Search space design is an essential component of the optimization process, and investing in search space design can result in an efficient optimization procedure.

Regularization is a technique that makes slight modifications to a learning algorithm such that the neural network model generalizes better. Adam (described in Section 2.5.2) is an example of a regularization technique. The algorithm adaptively scales the gradient in gradient optimization. *PBLF* and Backprop evolution provide us with the opportunity to make data-driven modifications to the backpropagation equation to improve generalization in a neural network model. In the following chapter, we evaluate our framework on an image recognition dataset and critically evaluate our findings.

Chapter 6

Experiment

In this chapter, we evaluate *PBLF* on the CIFAR10 dataset to demonstrate how our method performs against standard BP. In our results, we show that our method learns custom gradient update equations that report Top-1 accuracies of 95.8% on CIFAR10. We produce these results from only 100 Bayesian optimization samples from our continuous search space, which is 10x less than backprop evolution.

Throughout our experiments, we use the CIFAR-10 dataset. The choice for the CIFAR-10 dataset is motivated by the expensive computational cost of learning new gradient update equations, which is also acknowledged by backprop evolution [Alber et al., 2018]. Figure 5.1 illustrates our learning framework, *PBLF*, previously described in Section 4.3.

We show that, despite our 4-dimensional continuous search space of gradient update equations, we can apply methods like random search to sample custom gradient update equations for gradient optimization.

6.1 Detail

For this experiment, our target network of choice is the Wide Residual Network (WRN) Architecture [Zagoruyko and Komodakis, 2016]. We chose WRN with 16 layers and a width multiplier of 2 (WRN 16-2), as this is the same architecture employed by backprop evolution [Alber et al., 2018] for CIFAR10.

To recap from our methodology in Section 5.1, we parameterize the linear activation function derivatives in the backpropagation equation which serves as our search space of gradient update equations. Our choice for the linear derivative is motivated by the implicit abundance of linear activation functions at each layer of a neural network, which we show in Equation 5.2. Learning for linear activation function derivatives presents a general solution for learning custom gradient update equations

Parameter	Range	Description
$w_1 \in \mathbb{R}$	$\|w_1\| < 1$	Weight in first layer of the meta-network.
$b_1 \in \mathbb{R}$	$\|b_1\| < 1$	Bias in first layer of the meta-network.
$w_2 \in \mathbb{R}$	$\|w_2\| < 1$	Weight in second layer of the meta-network.
$b_2 \in \mathbb{R}$	$\|b_2\| < 1$	Bias in second layer of the meta-network.
mode	4-bit binary	Number of meta-networks and their positions.

Table 6.1: Meta-network parameter search space for CIFAR10 experimental setup

across a variety of neural network architectures. We parameterize the linear derivative with a neural network which we call a meta-network, which is different from the Domain Specific Language (DSL) as previously defined in Section 4.3.

Due to our parametrization, learning the weights of the meta-network is analogous to learning a new gradient update equation. For this experiment, we employ a simple single-input single-output (SISO) neural network with one hidden layer of width 1 with bias. Our choice for the SISO meta-network is that activation function derivatives are uni-variate functions in practice. Our SISO meta-network was a practical choice for our experimental setup for its simple search space and empirically we observed that there numerous gradient update solutions within the search as illustrated in Figure 6.4. We constraint the weights our meta-network by $|w| < 1$ following the theoretical bounds established in Xavier's neural network initialization [Glorot and Bengio, 2010].

In addition to learning the weights of the meta-network, we also learn the optimal number of meta-network modules and their optimal positions for our custom gradient update equation. We refer to this as the mode of the meta-network and represent this quantity as an n bit binary value, where n refers to the maximum number of meta-networks we want to learn in the gradient update equation. As a requirement, we predefined the n layers of potential interest in our search space. In Table 6.1, we provide a list of the search space parameters for our SISO meta-network.

We leverage Bayesian Optimization (BO) to learn custom linear derivatives. Our evaluation function for optimization is the validation accuracy of the WRN 16-2.

We split our dataset into three parts: training, validation and test set. For our training and validation set: We apply identical Gaussian normalization transformations to both training and validation datasets, the Gaussian parameters are as follows: $\mu = [0.4914, 0.4822, 0.4465]$, $\sigma = [0.2023, 0.1994, 0.2010]$. To distinguish between the training and validation set, we use a Random Crop and a Random Horizontal Flip to the training set and perform a random split of the entire dataset and allocate each set to the training and validation dataset respectively. For our test set: We make use of

the whole dataset and apply a different Gaussian normalization transformation from that of the training and validation set, the test dataset Gaussian parameters are as follows: $\mu = [0.485, 0.456, 0.406]$, $\sigma = [0.229, 0.224, 0.225]$. Our Top-1 accuracy stems from the test set, and our Top-1 validation stems from the validation dataset.

In our setup, we ran BO for a 100 iterations with 20 random initial samples of gradient update solutions. We employed the Gaussian Process as our statistical surrogate model of choice and an Upper Confidence Bound (UCB) acquisition function with trade-off parameter $\beta = 0.1$. To supplement the candidate solutions proposed by the acquisition function, we randomly sampled from the search space concurrently to improve the Gaussian process fit. In our results, we use BO and Random Search (RS) for *PBLF* to sample custom gradient update equations. For RS, we randomly sample 100 gradient update solutions as a comparison against BO for *PBLF*.

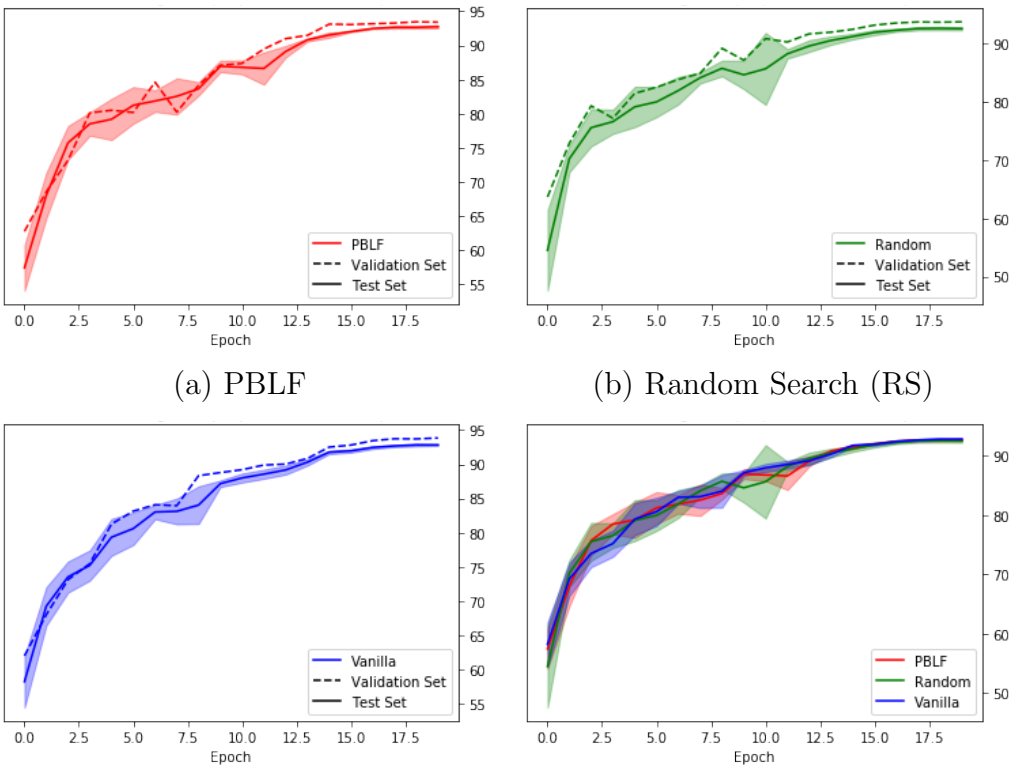
6.2 Results

For 20 epochs of CIFAR10, we performed three experiments: PBLF with BO (PBLF-BO), PBLF with Random Search (PBLF-RS) and Standard BP. These experiments will report how well PBLF fares empirically against Standard BP.

For PBLF-BO, we run 100 iterations of BO with 20 random initial samples of custom gradient update solutions. In PBLF-RS, we randomly sampled 100 gradient update solutions from our search space. Our results are averaged over five runs of each method. For 20 epochs, we reported negligible differences across the custom gradient update equations sampled by each method. We reported the following figures: Standard BP reported the highest Top-1 accuracy of $92.86 \pm 0.03\%$, followed by PBLF-BO with $92.77 \pm 0.04\%$, resulting in an average difference of 0.09% between the two approaches. Of the three methods, PBLF-RS reported the weakest result with a Top-1 accuracy of $92.59 \pm 0.13\%$. Figure 6.1 reports the Top-1 accuracy curves of these methods for 20 epochs with their respective validation curves.

For 20 epochs of training, we observe little empirical advantages across the approaches. The three methods range between Top-1 accuracies of 92.59% and 92.86%. Already, we begin to observe competitive Top-1 accuracies with custom gradient update equations at earlier epochs. Table 6.2 shows the meta-network parameters learned by PBLF-BO and PBLF-RS for 20 epochs.

To evaluate PBLF-RS and PBLF-BO for longer training times, we trained our target network for 100 epochs and compared it against Standard BP. As reported previously with 20 epochs of training, we observed slight differences in empirical



(a) PBLF (b) Random Search (RS)

(c) Standard BP (d) PBLF vs RS vs BP

Figure 6.1: (a) Top-1 Accuracy Curve of best-performing gradient update equation learned by PBLF for **20 epochs** of CIFAR10. PBLF is compared against RS and BP, as illustrated in figures (b)(c)(d). Figures (a)(b)(c) show the accuracy and validation curves of PBLF, RS and BP, respectively. In Figure (d), we compare these accuracy plots and notice negligible differences in final accuracy between standard BP and the custom gradient update equations. In Table 6.2, we report the final accuracy of this experiment. Standard BP reports the highest Top-1 accuracy of 92.86%, an increase of 0.09% from that of PBLF, closely followed by RS with a Top-1 accuracy of 92.59%.

performance across the three approaches for 100 epochs. The Top-1 accuracies for the three methods ranged between 95.00% and 95.47%.

With longer training times of 100 epochs, standard BP reported the highest Top-1 accuracy of $95.47 \pm 0.03\%$. This was followed by PBLF-RS with $95.28 \pm 0.03\%$ and PBLF-BO with $95.00 \pm 0.52\%$ with a custom gradient update equation. Figure 6.2 and Table 6.2 illustrate the result of this experiment. In Appendix B, we provide comprehensive results for each trial of our experiments, Table B.1 shows the Top-1 accuracy for each experimental trial across the three methods on CIFAR10 with their respective averages. We also show the corresponding validation accuracy in Table B.2.

In these experiments, we observed that PBLF (BO and RS) had sampled custom

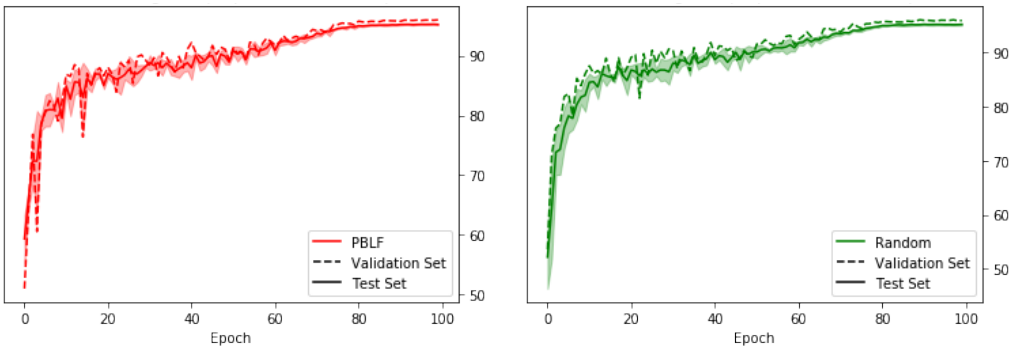
Table 6.2: Resulting gradient updates parameters learned by PBLF and Random Search (RS). We show their respective Top-1 Accuracy's for 20 and 100 epochs. W_k and B_k correspond to the weight and bias of the meta-network at layer k. The mode corresponds to the number of meta-networks in the gradient update equation and their respective positions. Although the Standard BP equation outperforms PBLF and RS by a slight margin, we find that both approaches learn very different gradient update equations.

	W_1	W_2	B_1	B_2	Mode	Top-1
100 EPOCHS						
PBLF-BO	0.0365	0	0.5816	0.375	0010	95±0.52
PBLF-RS	0.2282	0.2304	-0.765	0.8942	1110	95.28±0.03
Standard	0	0	0	1	1111	**95.47±0.03**
20 EPOCHS						
PBLF-BO	0.0002	0	0.4306	0	1000	92.77±0.04
PBLF-RS	0.824	0.2906	0.3945	0.8482	1100	92.59±0.13
Standard	0	0	0	1	1111	**92.86±0.03**

gradient update equations within our search space that are different from the standard BP gradient update equation, as shown in Table 6.2. In Table 6.2, we report the weights and biases of the meta-network and the mode learned by BO and RS for 20 and 100 epochs of training. The learned meta-network parameters shown in Table 6.2 suggest that our search space of gradient update equations has several competitive solutions.

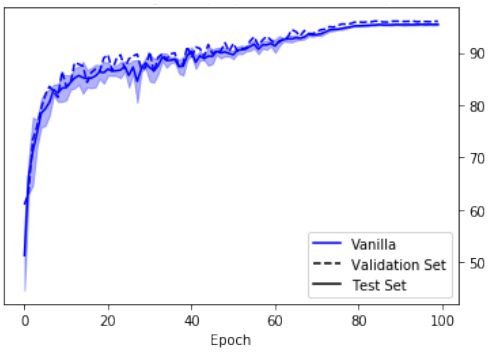
To measure the robustness of our learned equations, we trained the custom gradient update equations in Table 6.2 until convergence (600 epochs) and compared its performances against standard BP. Although, standard BP reported the highest Top-1 Accuracy of 96.23%, we found that our custom gradient update equations could train for till convergence. PBLF-RS produced an accuracy of 95.97%, and PBLF-BO reported 95.93%. Interestingly, the differences in Top-1 accuracy between the three methods fall within a per cent of each other. These results suggest that it is not necessary to train the target network until convergence to find interesting custom gradient update equations. In Figure 6.4, we show the Top-1 accuracy curves for this convergence experiment on CIFAR10.

Beyond the best performing parameters reported in Table 6.2, we found that both PBLF-RS and PBLF-BO had sampled several other gradient update solutions. In Figure 6.3, we plot the samples found by PBLF-RS and PBLF-BO for 20 and 100 epochs with their respective empirical performances. From this Figure, we notice that

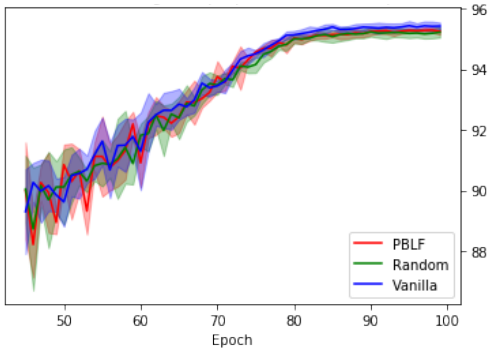


(a) PBLF

(b) Random Search(RS)



(c) Standard BP



(d) PBLF vs RS vs BP

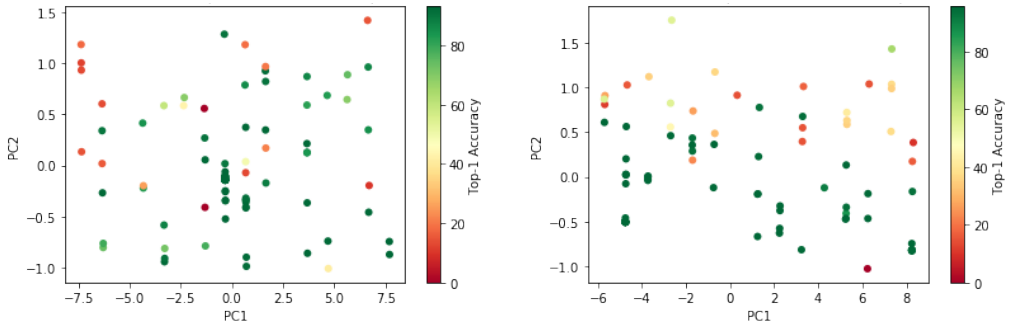
Figure 6.2: (a) Top-1 Accuracy Curve of best performing gradient update equation learned by PBLF for **100 epochs** of CIFAR10. PBLF is compared against RS and BP as illustrated in figures (b)(c)(d). Figures (a)(b)(c) show the accuracy and validation curves of PBLF, RS and BP respectively. In Figure (d), we compare amongst these accuracy plots, and notice negligible differences in final accuracy. In Table 6.2, we report the final accuracy of this experiment. Standard BP reports the highest Top-1 accuracy of 95.47%, followed by RS with 95.28% and PBLF with 95%.

there are several useful custom gradient update equations, suggesting that our search space has several local maxima.

Our results are especially interesting, considering that we sample custom gradient update equations from a simple 4-dimensional continuous search space and report similar Top-1 accuracies to standard BP. Despite stark differences across the update equations for each method, these custom gradients can also be trained till convergence.

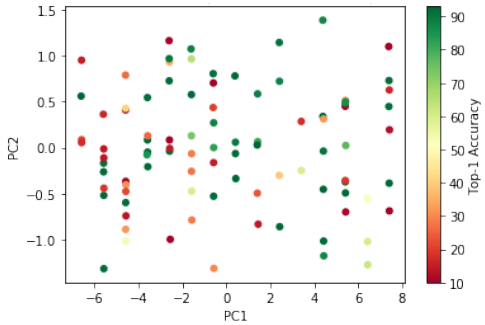
6.3 Discussion

In our results, we showed that our framework learns competitive gradient update equations from only 100 epochs of training. In this section, we critically evaluate our

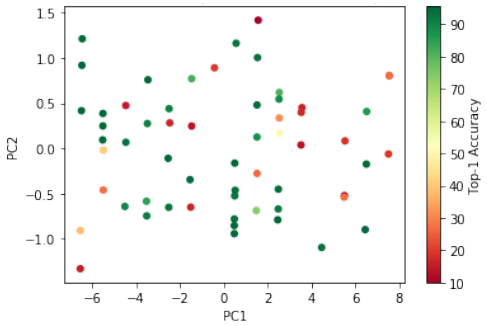


(a) PBLF-BO (20 epochs)

(b) PBLF-BO (100 epochs)



(c) PBLF-RS (20 epochs)



(d) PBLF-RS (100 epochs)

Figure 6.3: Scatter plot of gradient update solutions sampled from PBLF-BO and PBLF-RS for 20 and 100 epochs. We reduced the dimensionality of gradient update solution to 2 dimensions (PC1 and PC2). The colour scale represents the Top-1 accuracy of a gradient update solution. We observe that there are several custom gradient update equations that report Top-1 accuracies north of 80%.

study.

6.3.1 Shortfalls

The learning objective defined in our framework aims to optimize Top-1 validation accuracy for a predefined target network on a single dataset. We maximize the Top-1 validation accuracy as a proxy to maximize the Top-1 test accuracy to avoid overfitting. As a consequence, we found that some learned gradient update equations were particularly sensitive to hyperparameter changes, for example we observed that changes to the batch size would result in exploding gradients. This observation highlights the lack of generalization in our learning objective. However, in contrast, we found that the best performing gradient update equations found on PBLF for 100 epochs could train for longer times. In Figure 6.4, we show how these update equations performed for 600 epochs on CIFAR10. Overall, generalization to hyperparameter changes is a shortfall of our method. These observations suggest that

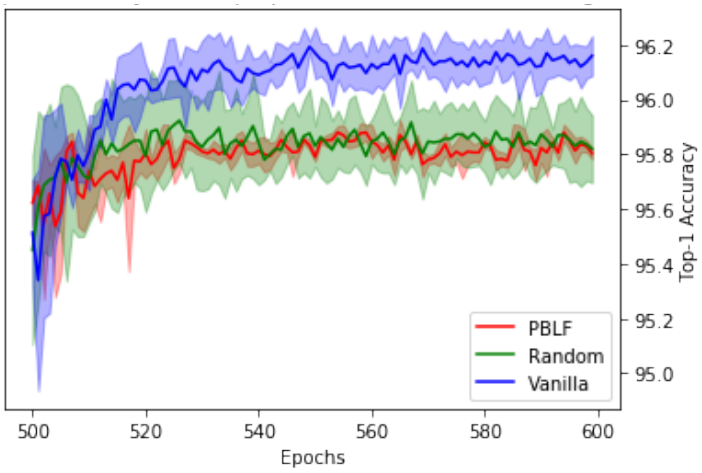


Figure 6.4: The optimal custom gradient update equations from Table 6.2 trained till convergence (600 epochs). Standard BP reports the highest Top-1 Accuracy of 96.23%. PBLF-RS reports an accuracy of 95.97%, and PBLF-BO reports 95.93%. The custom gradient update equations in this experiment are optimized for 100 epochs. We notice that despite the budget of 100 epochs, our custom equations can be trained until convergence. PBLF-RS, PBLF-BO and Standard BP fall within a per cent of each other at convergence.

our learning objective requires some revision to ensure that we learn custom gradient update equations that generalize to a set of optimization and neural network hyperparameters.

In our experiments, we witnessed wall clock training times amounting to several GPU days. In particular, the PBLF-BO experiment for 100 epochs ran for 14 GPU days on the Nvidia GeForce RTX 2080 Ti. We could mitigate these long training times with low-fidelity evaluations of potential gradient update solutions. Backprop evolution [Alber et al., 2018] also report long training times spanning over 2-3 days across 500 CPU workers. We would need to address the computational burden of running these experiments.

6.3.2 Strengths and Opportunities

In our work, we have constructed a continuous search space of gradient update equations; in doing so, this offers several strengths and opportunities. Synthetic gradients [Jaderberg et al., 2016] propose a similar search space and methodology. However, their objective is to reproduce the original gradient equation to solve the update locking problem, and we discuss this in Chapter 4.1 in our related work. In this work,

we seek to learn custom gradient update equations that maximize Top-1 validation accuracy, a similar objective shared with backprop evolution [Alber et al., 2018]. Backprop evolution leverages a discrete search space of gradient update equations and evolutionary search to find custom gradient update equations.

With a 4-dimensional continuous search space, our method was robust enough for us to use random search to find custom gradient update equations that compete empirically with the standard gradient update equations. We were similarly able to reach convergence at 600 epochs despite only having to optimize for 100 epochs of training. The continuous search space presented the opportunity for us to build a Gaussian process surrogate model of gradient update equations, to allow for a more sample efficient search with Bayesian optimization. We learned custom gradient update equations from 100 Bayesian optimization samples, which is 10x less than backprop evolution.

In Figure 6.3, we show a scatter plot of all the samples from BO and RS for 20 and 100 epochs. From these surface plots, we observe that the search space has several local maxima. The convexity of the search space could account for the reason why our framework could not reproduce the standard gradient equation, due to the numerous local maxima.

Beyond our continuous reparametrization, our method produces reproducible gradient update solutions. It is only necessary to have the saved meta-network architecture and the layer configuration to apply the custom gradient equation, the approach does not rely on an array of operands, unary and binary functions as is the case with backprop evolution.

The continuous parametrization of the gradient update equation presents the opportunity for learning custom gradient update equations with gradient-based optimization. Gradient-based optimization for custom gradient equations is a similar proposition to ADAM. The ADAM optimizer is an adaptive learning rate procedure that regularizes the learning rate post-backpropagation. Our work learns custom gradient values prior to the optimization step. The similarity lies in the fact that both approaches regularize the gradient step except that we apply a transformation before the gradient step. We believe that exploring this avenue of research could open up interesting questions on custom gradients for gradient optimization for artificial neural networks.

Chapter 7

Conclusion

In this work, we presented a method which learns custom gradient update equations for gradient optimization. Our method, *PBLF*, sampled several custom gradient update equations that reported similar performances to standard backpropagation at convergence. We discovered competitive gradient update equations from only 100 Bayesian optimization samples, a significant improvement in sample efficiency from Backprop evolution. With Bayesian optimization, our continuous search space made it easier to build a surrogate model of our search space to optimize for sample efficiency.

Overall, data-driven custom update equations offer a promising alternative for gradient optimization in artificial neural networks. We encourage future work in this area of research.

7.1 Future Work

Some future directions in this area of research:

7.1.1 Single Phase Training

Training across two phases, SGD and Evolutionary/Bayesian optimization, is an added computational overhead as discussed in Section 6.3.1. Research invested towards simultaneously training a target network and learning a custom gradient update equation would greatly improve the computational cost of training. It would be also interesting to explore the possibility of using SGD to learn custom update equations with our *PBLF* method. SGD could also greatly improve the performances of our custom gradient update equations.

Appendices

Appendix A

Backprop Evolution

A.1 Search Space

We present an exhaustive list of the domain specific language constructed in Backprop evolution for searching custom gradient update equations.

- Operands:

  - W_i, sgn(W_i) (weight matrix of the current layer and sign of it)

  - R_i, S_i (Gaussian and Bernoulli random matrices, same shape as W_i),

  - R_{Li} (Gaussian random matrix mapping from b_L^p to b_i^p),

  - h_i^p, h_i, h_{i+1}^p (hidden activations of the forward propagation),

  - b_L^p, b_{i+1}^p (backward propagated values),

  - $b_{i+1}^p \frac{\partial h_{i+1}^p}{\partial h_i}, b_{i+1}^p \frac{\partial h_{i+1}^p}{\partial h_i^p}$ (backward propagated values according to gradient backward propagation),

  - $b_{i+1}^p \cdot R_i, (b_{i+1}^p \cdot R_i) \odot \frac{\partial h_i}{\partial h_i^p}$ (backward propagated values according to feedback alignment),

  - $b_L^p \cdot R_{Li}, (b_L^p \cdot R_{Li}) \odot \frac{\partial h_i}{\partial h_i^p}$ (backward propagated values according to direct feedback alignment).

- Unary functions $u(x)$:

  - x (identity), x^t (transpose), $1/x$, $|x|$, $-x$ (negation), $\mathbb{1}_{x>0}$ (1 if x greater than 0), x^+ (ReLU), sgn(x) (sign), $\sqrt{|x|}$, sgn(x)x^2, x^3, ax, $x + b$

- x + gnoise(g), $x\odot$ (1 + gnoise(g)) (add or multiply with Gaussian noise of scale $g \in (0.01, 0.1, 0.5, 1.0)$),

- $\text{drop}_d(x)$ (dropout with drop probability $d \in (0.01, 0.1, 0.3)$), $\text{clip}_c(x)$ (clip values in range $[-c, c]$ and $c \in (0.01, 0.1, 0.5, 1.0)$),

- $x/||\cdot||_0^{elem}$, $x/||\cdot||_1^{elem}$, $x/||\cdot||_2^{elem}$, $x/||\cdot||_{-inf}^{elem}$, $x/||\cdot||_{inf}^{elem}$ (normalizing term by vector norm on flattened matrix),

- $x/||\cdot||_0^{col}$, $x/||\cdot||_1^{col}$, $x/||\cdot||_2^{col}$, $x/||\cdot||_{-inf}^{col}$, $x/||\cdot||_{inf}^{col}$, $x/||\cdot||_0^{row}$, $x/||\cdot||_1^{row}$, $x/||\cdot||_2^{row}$, $x/||\cdot||_{-inf}^{row}$, $x/||\cdot||_{inf}^{row}$, (normalizing term by vector norm along columns or rows of matrix),

- $x/||\cdot||_{fro}$, $x/||\cdot||_1$, $x/||\cdot||_{-inf}$, $x/||\cdot||_{inf}$, (normalizing term by matrix norm),

- $(x - \hat{m}(x))/\sqrt{\hat{s}(x^2)})$ (normalizing with running averages with factor $r = 0.9$).

- Binary functions $f(x, y)$:

  - $x + y$, $x - y$, $x \odot y$, x/y (element-wise addition, subtraction, multiplication, division),

  - $x \cdot y$ (matrix multiplication),

  - x (keep left),

  - $\min(x, y)$, $\max(x, y)$ (minimum and maximum of x and y).

A.2 Further Analysis

The observations reported in Backprop evolution [Alber et al., 2018] suggest that there are custom backpropagation equations that scale to complex datasets. Yet, we lack analysis of why these custom equations yield competitive empirical results. In this section, we demonstrate that modular changes to the backpropagation equation are analogous to scaling the gradient loss. This section follows up on the results reported by Backprop evolution.

In our analysis, we maintain the same notation used in Backprop evolution for the sake of consistency.

In Backprop evolution, the gradient update ($\Delta W_i = b_i^p \frac{\partial h_i^p}{\partial W_i}$) is defined as follows:

$$W_i^{n+1} = W_i^n + \alpha \Delta W_i^n \tag{A.1}$$

Where n is the training iteration and α is the learning rate. Backprop evolution uses evolutionary methods to replace the module b_i^p in the backpropagation equation. For the sake of clarity, we will use the module r_i^p to represent the searched replacement for module b_i^p.

Mathematically, replacing b_i^p with r_i^p translates to,

$$\Delta W_i^{new} = \Delta W_i \frac{r_i^p}{b_i^p}, \quad b_i^p \neq 0 \tag{A.2}$$

where we express the new update equation ΔW_i^{new} as a function of the old update equation ΔW_i. We write this explicity as follows:

$$\Delta W_i^{new} = b_i^p \frac{\partial h_i^p}{\partial W_i} \frac{r_i^p}{b_i^p}, \quad b_i^p \neq 0 \tag{A.3}$$

Within the context of SGD, this translates to:

$$W_i^{n+1} = W_i^n + \alpha \Delta W_i^{new} = W_i^n + \alpha \Delta W_i \frac{r_i^p}{b_i^p}, \quad b_i^p \neq 0 \tag{A.4}$$

Hence, we can treat $\beta = \alpha \frac{r_i^p}{b_i^p}$ as the new learning rate. This suggests that replacing b_i^p with r_i^p is equivalent to scaling the gradient.

This analysis holds true only for the case where $b_i^p \neq 0$, which is predominately the case. When $b_i^p = 0 \implies \frac{r_i^p}{b_i^p}$ is undefined, hence Equation A.3 now translates to,

$$\Delta W_i^{new} = r_i^p \frac{\partial h_i^p}{\partial W_i} \neq \Delta W_i \frac{r_i^p}{b_i^p}, \quad b_i^p = 0 \tag{A.5}$$

We cannot express the new update equation ΔW_i^{new} as a function of the old update equation ΔW_i, hence this can no longer be interpreted as gradient scaling. We regard this case as gradient noise which is often considered beneficial in artificial neural networks [Neelakantan et al., 2015].

Our analysis suggests that when changing modular parts of the backpropagation equation in the backprop evolution framework, the gradient replacement r_{i+1}^p is being backpropagated to earlier layers of the network, as described here:

$$b_i = r_{i+1}^p W_{i+1}^T \tag{A.6}$$

Backpropagating r_{i+1}^p can hurt training if there is no consideration for earlier layers. In Table (?), we report the gradient update equations that Backprop evolution has learned that consistently perform well across all setups. We found that

the resulting equations where merely normalizing the standard gradient computation. Coincidently, GradNorm [Chen et al., 2018b] also demonstrate empirically that gradient normalization greatly improves training and avoids overfitting in artificial neural networks. Although preliminary, this is consistent with our claim that Backprop evolution learns an appropriate scaling method for gradient computation.

We hope that this preliminary analysis will encourage future work that seeks to interpret custom gradient update equations that arise in search.

Appendix B

Comprehensive Results

We report the comprehensive results for our PBLF-RS and PBLF-BO experiments described in Table 6.2 in Section 6.2. In this chapter, we report the Top-1 accuracies across all five experimental trials. Table B.1 shows the Top-1 Accuracy for each experiment and Table B.2 reports respective the Top-1 Validation Accuracy.

100 EPOCHS						Top-1 Avg.
PBLF-BO	95.42	95.38	95.29	95.17	93.72	95 ± 0.52
PBLF-RS	95.22	95.26	95.55	95.08	95.27	95.28 ± 0.03
Standard	95.38	95.3	95.48	95.46	95.73	**95.47±0.03**
20 EPOCHS						
PBLF-BO	92.78	92.89	92.94	92.45	92.78	92.77±0.04
PBLF-RS	92.87	92.74	92.65	92.71	91.97	92.59±0.13
Standard	92.98	92.76	93.08	92.83	92.63	**92.86±0.03**

Table B.1: Top-1 Accuracy of PBLF, Random Search (RS) and Standard BP for CIFAR10 across multiple training jobs.

100 EPOCHS						Top-1 Val Avg.
PBLF-BO	96.119	96.139	96.099	96.019	94.819	95.84±0.33
PBLF-RS	95.979	95.839	95.92	96.059	96.159	96.00±0.02
Standard	96.059	96.159	96.279	96.059	95.86	**96.08±0.02**
20 EPOCHS						
PBLF-BO	93.559	93.76	93.559	93.439	93.48	93.56±0.02
PBLF-RS	93.99	93.99	94.06	93.69	93.439	**93.83±0.07**
Standard	93.89	93.44	93.839	93.839	93.399	93.68±0.06

Table B.2: Top-1 Validation Accuracy of PBLF, Random Search (RS) and Standard BP for CIFAR10 across multiple training jobs.

Appendix C

PBLF Gradient Trajectories

In our experimental results, in Section 6.2, we show how PBLF (parametrized backpropagation learning framework) learns custom gradient update equations for gradient optimization. To achieve this, we learn the activation function derivative in the backpropagation equation. To study the effects of our PBLF method, in this chapter, we perturb the ReLU activation function derivative in the backpropagation equation for a simple binary classification task and compare gradient trajectories across each perturbation.

Across the various perturbations, we observed accelerated and decelerated gradient trajectories on non-adaptive gradient optimization algorithms like SGD. However, with adaptive gradient optimization algorithms, like Adam, we reported little changes to the gradient trajectory as we perturbed the activation function derivative. These findings suggest a positive correlation between the learning rate of a network and perturbations to the activation function derivative in the backpropagation equation.

C.1 Experimental Details

For this task, we construct a target neural network with ReLU activation and no bias that takes as input (x_1, x_2) and then outputs a softmax probability predicting a target probability distribution. There are two target probability distributions: $\mathcal{N}(\mu_1, I_2)$ and $\mathcal{N}(\mu_2, I_2)$, such that $\mu_1 = [1, 1]$ and $\mu_2 = [-1, -1]$. We define this as the binary classification task.

The ReLU activation function or the Rectified Linear Unit is defined as follows:

$$f(x) = max(0, x) \tag{C.1}$$

In this experiment, to induce various backpropagation equations, we perturb the ReLU derivative $f'(x)$ with a hyperparameter β to produce different custom backpropagation equations. Note that x represents the layer pre-activations as is necessary when computing activation function partial derivatives. Hence, $f'(x) := f'(x;\beta)$ is defined as:

$$f'(x;\beta) = \begin{cases} 0 & x \leq 0 \\ \beta & x > 0 \end{cases} \tag{C.2}$$

$f'(x;\beta)$ is our meta-network architecture for this binary classification task. For convenience, we define $f'(x;\beta)$ by the constant β. For example, $\beta = 2$ would be equivalent to:

$$f'(x;\beta = 2) = \begin{cases} 0 & x \leq 0 \\ 2 & x > 0 \end{cases} \tag{C.3}$$

Hence, $\beta = 1$ would represent the vanilla ReLU derivative.

C.2 Gradient Trajectories

In this experiment, we trained a three-hidden layer neural network to perform a simple binary classification task across different $f'(x;\beta)$ functions with different β parameters. In the three-hidden layer network, the first hidden layer had one neuron while the other hidden layers had two neurons, all with no bias. Across these experiments, we experimented with different optimization techniques to further investigate the gradient trajectories of these networks. The loss function used in this particular experiment is the binary cross-entropy loss. Note, we learn the two weights of the target network: $w_1^{(2)}$ and $w_2^{(2)}$ from the second$^{(2)}$ hidden layer of the network. The other weights of the target network are frozen.

Initially, we trained our network independently on both $f'(x;\beta = 1)$ and $f'(x;\beta = 5)$ perturbations on the second hidden layers' activation function derivative. After 100 epochs of batch gradient descent (with momentum), we noticed that the $f'(x;\beta = 5)$ perturbation appeared to have an accelerated trajectory as compared to the $f'(x;\beta = 1)$ perturbation across numerous trials. Overall the $f'(x;\beta = 5)$ perturbation had faster convergence. In Figure C.1, we show a few trials from our experiments for gradient trajectories $\beta = 1$ and $\beta = 5$.

To further evaluate the effects of β on the gradient trajectories, we subsequently compared the gradient trajectories of $\beta = 1$ and $\beta = 0.5$. Across numerous trials with gradient descent (with momentum), we observed the opposite effect as observed

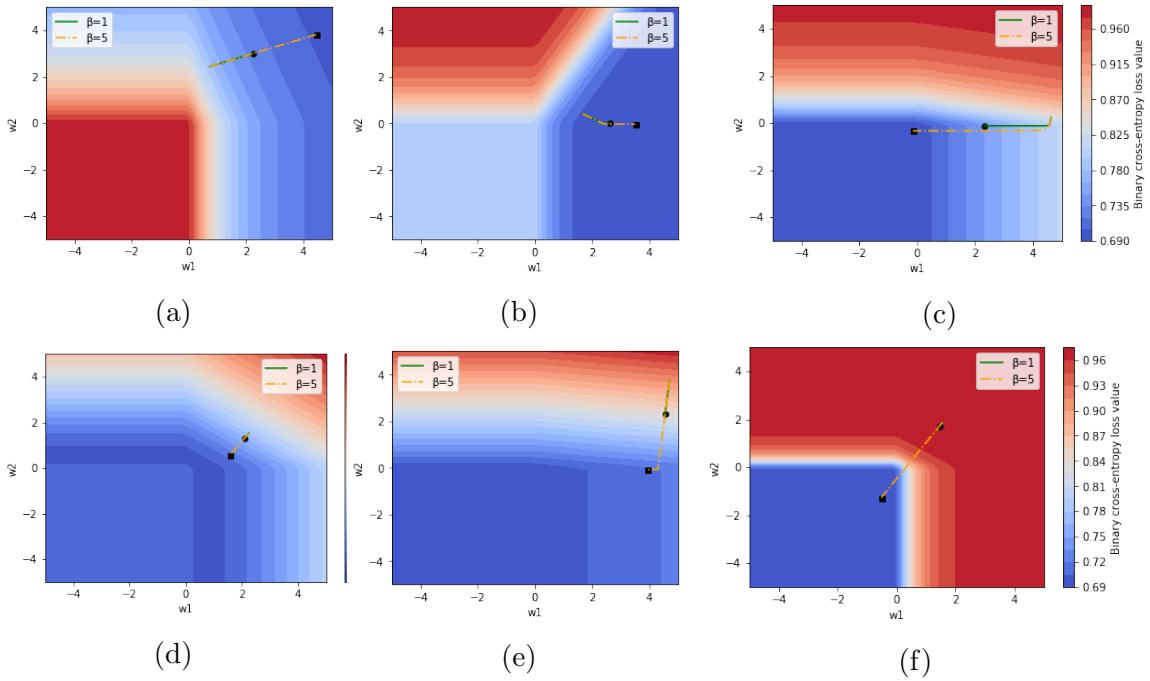


Figure C.1: Gradient trajectories of $\beta = 1$ and $\beta = 5$. Note that the black square and circle in the plots are the endpoints of the trajectories. As illustrated in the plots, $\beta = 5$ appears to have an accelerated trajectory as compared to $\beta = 1$ (vanilla ReLU derivative). The optimizer used in this experiment is GD with a learning rate of 0.5 and a momentum of 0.5. Initially, both trajectories travel the same path where eventually $\beta = 1$ comes to a halt. Figure C.3c is an exception as both trajectories take on slightly different paths. These plots suggest a correlation between β and the learning rate of the network.

in the trial comparing functions $\beta = 1$ and $\beta = 5$. We had observed that $\beta = 0.5$ had slower convergence as compared to $\beta = 1$. In Figure C.2, we show a few trials from our experiments for gradient trajectories $\beta = 1$ and $\beta = 0.5$.

This comparison indicates that varying the hyperparameter β offers either an accelerated or decelerated gradient trajectory. As depicted in Figure C.2, we observe that across our gradient perturbations, they all traverse in the same gradient direction. Still, during gradient optimization, the gradient with the smallest β gradually decelerates, suggesting that changes across β are indicative of the acceleration of the gradient trajectory.

At this point, these observations on the binary classification task compliment our claim, where we mentioned that learning parts of the backpropagation procedure are analogous to finding the optimal learning rate for a network.

To evaluate the β parameter on an adaptive learning rate, we ran an experiment on the Adam [Kingma and Ba, 2014] optimizer instead of batch gradient descent.

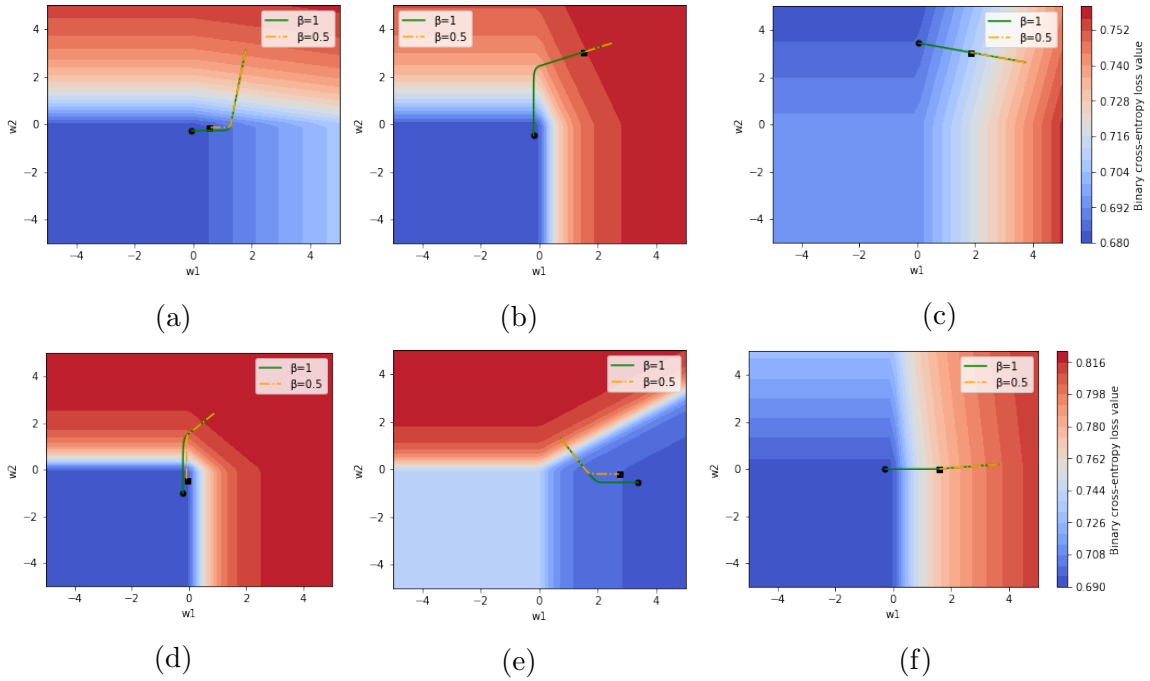


Figure C.2: Gradient trajectories of $\beta = 1$ and $\beta = 0.5$. As illustrated in the plots, $\beta = 1$ appears to have an accelerated trajectory as compared to $\beta = 0.5$. In this experiment, we tested a smaller β to see if it slows down learning as compared to $\beta = 1$ (vanilla ReLU derivative). The optimizer used in this experiment is SGD with a learning rate of 1.5 and momentum of 0.5. These plots further suggest a correlation between β and the learning rate of the network. Both Figure C.1 and this Figure demonstrate that changes to the hyperparameter β results in changes to the learning rate of the weights w_1 and w_2.

Upon conducting comparisons between $\beta = 1$ and $\beta = 0, 5, 100$ on the Adam optimizer across numerous trials, we observed that the gradient trajectories were almost negligible in performance. Even in the extreme case ($\beta = 1$ and $\beta = 100$), all gradient trajectories followed ended up in similar local minima.

As defined in Section 2.5.4, Adam optimizer is an optimizer with an adaptive learning rate. Given our previous observations with SGD, we hypothesis that Adam normalizes the accelerated effects of meta-network perturbations as a consequence of its adaptive procedure. Our observations on Adam suggest that perturbing the activation function derivatives in the backpropagation equation has little to no effect on the gradient trajectory when an adaptive optimizer is applied. In Figure C.2, we show a few trials from our experiments on Adam.

Figure C.4 provides quantitative evidence demonstrating the correlation between β and the velocity of the gradient trajectories for the binary classification task.

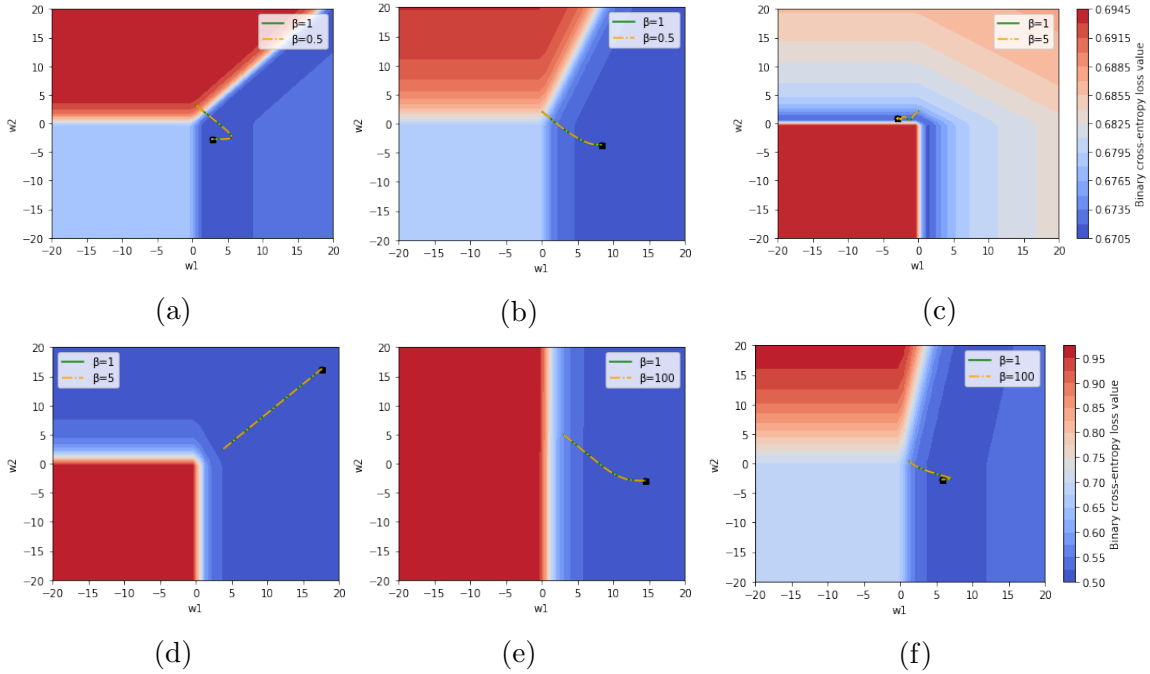


Figure C.3: As opposed to Figure C.1 and Figure C.2, we used the Adam optimizer instead of SGD with momentum. We compared $\beta = 1$ (vanilla ReLU derivative) against $\beta = 0.5, 5, 100$, yet the networks had achieved the exact same gradient trajectories. Adam is reputably known as an adaptive learning rate optimizer. Adam had reported little changes to the gradient trajectory in comparison to the SGD experiment. This experiment supports our intuition that β acts like a learning rate for weights w_1 and w_2.

In the SGD experiment, we observed accelerated and decelerated learning trajectories across various β parameters. Although, with Adam, we noticed negligible differences across multiple β values. These results suggest that with non-adaptive gradient optimizers, meta-network perturbations behave similarly to the learning rate of the target neural network.

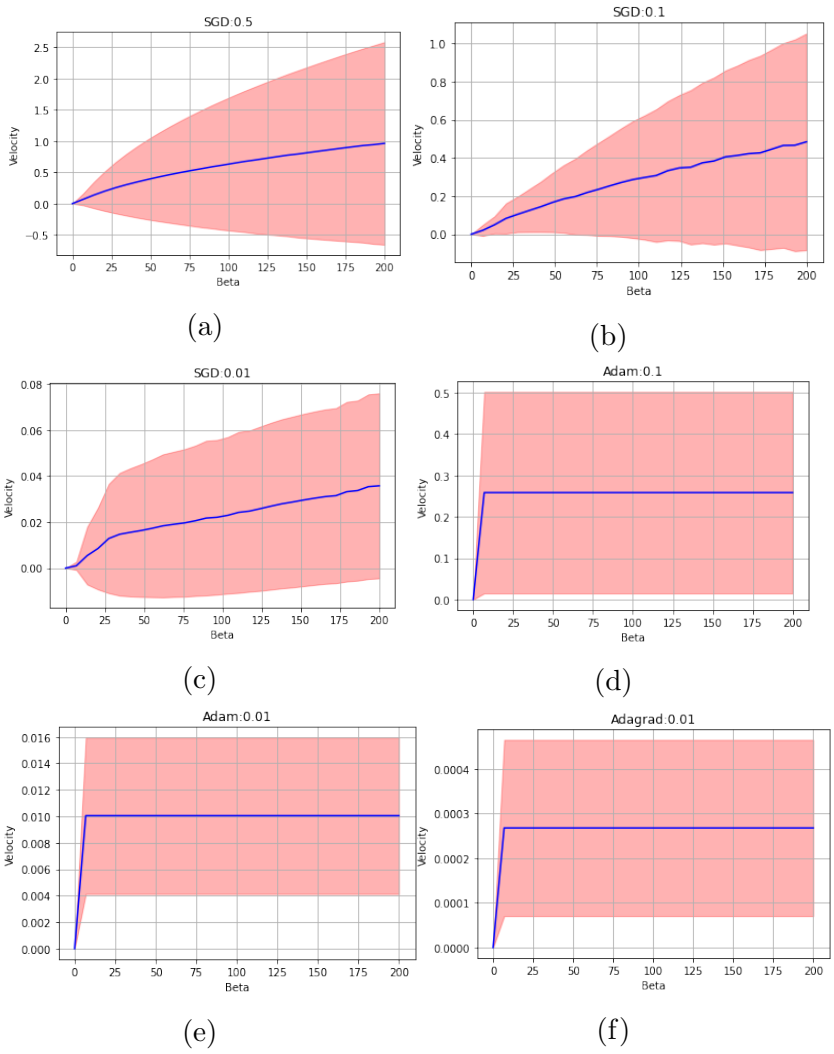


Figure C.4: Quantitative results are demonstrating the velocity of the gradient trajectory per β parameter. We compare the gradient trajectory across SGD, Adam and Adagrad. In each plot, we indicate the predefined learning rate for each optimizer, i.e. *Adam: 0.5*. For SGD, plots (C.4a), (C.4b) & (C.4c) suggest a positive correlation between β and the velocity of the gradient trajectory. More precisely, an increase in β results in an increase in velocity of the gradient trajectory. For the adaptive optimizers, Adam & Adagrad, we observe constant velocity across the various β values except when $\beta = 0$ which is simply dropout. We averaged these results over ten runs.

Bibliography

M. Alber, I. Bello, B. Zoph, P.-J. Kindermans, P. Ramachandran, and Q. Le. Backprop evolution. *arXiv preprint arXiv:1808.02822*, 2018.

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.

S. Bartunov, A. Santoro, B. Richards, L. Marris, G. E. Hinton, and T. Lillicrap. Assessing the scalability of biologically-motivated deep learning algorithms and architectures. In *Advances in Neural Information Processing Systems*, pages 9390–9400, 2018.

Y. Chen, A. Huang, Z. Wang, I. Antonoglou, J. Schrittwieser, D. Silver, and N. de Freitas. Bayesian optimization in alphago. *arXiv preprint arXiv:1812.06855*, 2018a.

Z. Chen, V. Badrinarayanan, C.-Y. Lee, and A. Rabinovich. Gradnorm: Gradient normalization for adaptive loss balancing in deep multitask networks. In *International Conference on Machine Learning*, pages 794–803, 2018b.

A. Criminisi, J. Shotton, and E. Konukoglu. Decision forests for classification. *Regression, Density Estimation, Manifold Learning and Semi-Supervised Learning*, 2011.

C. Finn, P. Abbeel, and S. Levine. Model-agnostic meta-learning for fast adaptation of deep networks. In *Proceedings of the 34th International Conference on Machine Learning-Volume 70*, pages 1126–1135. JMLR. org, 2017.

X. Glorot and Y. Bengio. Understanding the difficulty of training deep feedforward neural networks. In *Proceedings of the thirteenth international conference on artificial intelligence and statistics*, pages 249–256, 2010.

H. F. Harlow. The formation of learning sets. *Psychological review*, 56(1):51, 1949.

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.

Z. Hu. Statistical optimization of supply chain financial credit based on deep learning and fuzzy algorithm. *Journal of Intelligent & Fuzzy Systems*, (Preprint):1–12, 2020.

M. Jaderberg, W. M. Czarnecki, S. Osindero, O. Vinyals, A. Graves, D. Silver, and K. Kavukcuoglu. Decoupled neural interfaces using synthetic gradients. *arXiv preprint arXiv:1608.05343*, 2016.

M. N. Katehakis and A. F. Veinott Jr. The multi-armed bandit problem: decomposition and computation. *Mathematics of Operations Research*, 12(2):262–268, 1987.

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

D.-H. Lee, S. Zhang, A. Biard, and Y. Bengio. Target propagation. arxiv preprint. *arXiv preprint arXiv:1412.7525*, 2014.

D.-H. Lee, S. Zhang, A. Fischer, and Y. Bengio. Difference target propagation. In *Joint european conference on machine learning and knowledge discovery in databases*, pages 498–515. Springer, 2015.

K. Li and J. Malik. Learning to optimize. *arXiv preprint arXiv:1606.01885*, 2016.

T. P. Lillicrap, D. Cownden, D. B. Tweed, and C. J. Akerman. Random synaptic feedback weights support error backpropagation for deep learning. *Nature communications*, 7:13276, 2016.

F. Mateo, J. Muoz, V. Laparra, J. Verrelst, and G. Camps-Valls. Learning structures in earth observation data with gaussian processes. volume 9785, pages 78–94, 08 2016. ISBN 978-3-319-44411-6. doi: 10.1007/978-3-319-44412-3_6.

W. S. McCulloch and W. Pitts. A logical calculus of the ideas immanent in nervous activity. *The bulletin of mathematical biophysics*, 5(4):115–133, 1943.

M. Mitchell. *An introduction to genetic algorithms*. 1998.

J. Močkus. On bayesian methods for seeking the extremum. In *Optimization Techniques IFIP Technical Conference*, pages 400–404. Springer, 1975.

A. Neelakantan, L. Vilnis, Q. V. Le, I. Sutskever, L. Kaiser, K. Kurach, and J. Martens. Adding gradient noise improves learning for very deep networks. *arXiv preprint arXiv:1511.06807*, 2015.

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.

A. Nøkland. Direct feedback alignment provides learning in deep neural networks. In *Advances in neural information processing systems*, pages 1037–1045, 2016.

N. Qian. On the momentum term in gradient descent learning algorithms. *Neural networks*, 12(1):145–151, 1999.

C. E. Rasmussen. Gaussian processes in machine learning. In *Advanced lectures on machine learning*, pages 63–71. Springer, 2004.

L. Rosasco, E. D. Vito, A. Caponnetto, M. Piana, and A. Verri. Are loss functions all the same? *Neural Computation*, 16(5):1063–1076, 2004.

S. Ruder. An overview of gradient descent optimization algorithms. *arXiv preprint arXiv:1609.04747*, 2016.

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

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

A. Shah, A. Wilson, and Z. Ghahramani. Student-t processes as alternatives to gaussian processes. In *Artificial intelligence and statistics*, pages 877–885, 2014.

A. Slivkins. Introduction to multi-armed bandits. *arXiv preprint arXiv:1904.07272*, 2019.

J. Snoek, H. Larochelle, and R. P. Adams. Practical bayesian optimization of machine learning algorithms. In *Advances in neural information processing systems*, pages 2951–2959, 2012.

M. Sornam and M. Prabhakaran. A new linear adaptive swarm intelligence approach using back propagation neural network for dental caries classification. In *2017 IEEE International Conference on Power, Control, Signals and Instrumentation Engineering (ICPCSI)*, pages 2698–2703. IEEE, 2017.

K. O. Stanley, J. Clune, J. Lehman, and R. Miikkulainen. Designing neural networks through neuroevolution.

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.

E. Tosun, K. Aydin, and M. Bilgili. Comparison of linear regression and artificial neural network model of a diesel engine fueled with biodiesel-alcohol mixtures. *Alexandria Engineering Journal*, 55(4):3081–3089, 2016.

Z. Wang, M. Zoghi, F. Hutter, D. Matheson, N. De Freitas, et al. Bayesian optimization in high dimensions via random embeddings. In *IJCAI*, pages 1778–1784, 2013.

S. Zagoruyko and N. Komodakis. Wide residual networks. *arXiv preprint arXiv:1605.07146*, 2016.

M. D. Zeiler. Adadelta: an adaptive learning rate method. *arXiv preprint arXiv:1212.5701*, 2012.

B. Zoph and Q. V. Le. Neural architecture search with reinforcement learning. *arXiv preprint arXiv:1611.01578*, 2016.