Derivation and Analysis of Backpropagation: An Ablation Study on Optimization, Regularization, and Weight Initialization in MLPs

Gregory Okoma

November 2025

Contents

Abstract

Multi-Layer Perceptron (MLP) networks are the simplest of the feedforward networks that are central to many modern AI systems. Despite this, the mathematical details involved in back-propagation and optimization remain elusive. With the availability of high-level machine learning libraries such as TensorFlow and PyTorch, these foundational mechanisms are frequently abstracted away. This project aims to clarify these underlying principles and provide a transparent, first-principles understanding of MLP training. We developed a highly generalizable MLP implemented in NumPy, capable of supporting arbitrary architectures and activation functions, to facilitate experimentation and learning.

1 Introduction

Neural networks make up a significant portion of the modern real-world applications of AI systems, and Multi-Layer Perceptron (MLP) networks are the most foundational of these. Understanding the optimization process of MLPs allows one to gain a more detailed understanding of the strengths and limitations of MLP networks. This will enable further investigation into other architectures, such as Bayesian Neural Networks (BNN), to be pursued with the knowledge of the insights previously gained.

This project seeks to clarify the fundamentals behind the optimization process of MLP networks by: presenting the derivations of the relevant gradient terms that are calculated during back-propagation of an MLP network for the cases where the activation functions are univariate (e.g. ReLU) and multivariate (e.g. Softmax); presenting the optimizers involved in the gradient descent calculations for Stochastic Gradient Descent (SGD), Momentum [8], Root Mean Square Propagation (RMSProp) [9], and Adaptive Moment Estimation (Adam) [5]; presenting the expression for L2-regularization; and presenting a specific regime of weight initialization - He Uniform initialization [3] - which significantly improves learning performance in the MLP network architecture tested when compared to random uniform initialization $w \sim U[-1, 1]$.

We then conduct a comparative analysis of the performance of models trained using each optimizer, with and without He Uniform initialization, with different values of L2-regularization coefficient λ. A summary of the results is provided in the "Results" section and some theoretical explanations and hypotheses are presented in the "Discussion" section.

2 Theoretical Framework

Every neural network is fundamentally a function that takes in a vector-valued input and returns a vector-valued output.

A layer in a neural network is fully connected if each neuron in that layer is connected to each neuron in the previous layer. A neural network in which each layer is fully connected (the input layer is excluded) is a Fully Connected Network (FCN) or, in other words, a Multilayer Perceptron network (MLP). This document explores forward and back-propagation in an MLP network.

2.1 Forward Propagation

Forward propagation in an MLP network from one layer to the next consists of matrix multiplications, matrix additions, and the application of a function g - the activation function - to a matrix, entry-wise or vector-wise. Therefore, forward propagation is essentially basic linear algebra.

Let f denote an MLP network. Then f is just a composition of sub-functions $f^{[l]}$ (corresponding to the layers of f)

$$f(\mathbf{x}) = f^{[L]}(f^{[L-1]}(\cdots(f^{[1]}(\mathbf{x})))) = (f^{[L]} \circ f^{[L-1]} \circ \cdots \circ f^{[1]})(\mathbf{x})$$

where $\mathbf{x}$ is a vector.

To simplify notation, denote the output of layer $l-1$ by the $1 \times p$ vector $\mathbf{a}^{[l-1]}$ and the output of layer l by the $1 \times q$ vector $\mathbf{a}^{[l]}$. Then if the $p \times q$ matrix $\mathbf{W}^{[l]}$ denotes the weight matrix for layer l, and the $1 \times q$ vector $\mathbf{b}^{[l]}$ denotes the bias vector for layer l, then the $1 \times q$ vector $\mathbf{z}^{[l]}$ is given by

$$\mathbf{z}^{[l]} = \mathbf{a}^{[l-1]}\mathbf{W}^{[l]} + \mathbf{b}^{[l]}$$

where $\mathbf{z}^{[l]}$ is known as the "pre-activation" for layer l. The output of layer l is then calculated by applying an activation function $g^{[l]}$:

$$\mathbf{a}^{[l]} = g^{[l]}(\mathbf{z}^{[l]})$$

In the above explanation, the output of layer l was a $1 \times q$ vector for some positive integer q, which corresponds to the output of layer l for one input $\mathbf{x}$ of the MLP network. A single input will have shape $1 \times q_0$. This notation can be generalized to find the output of n inputs to the MLP network by stacking each row vector input $\mathbf{x}$ to form an $n \times q_0$ matrix $\mathbf{X}$. In this case, the output of layer $l-1$ will be the $n \times p$ matrix $\mathbf{A}^{[l]}$ if the same weight matrix $\mathbf{W}^{[l]}$ is used and a new $n \times q$ bias matrix $\mathbf{B}^{[l]}$ is formed by stacking the row vectors $\mathbf{b}^{[l]}$ on top of each other n times. In this case, the pre-activation $\mathbf{Z}^{[l]}$ is given by

$$\mathbf{Z}^{[l]} = \mathbf{A}^{[l-1]}\mathbf{W}^{[l]} + \mathbf{B}^{[l]}$$

where each row of $\mathbf{Z}^{[l]}$ is the pre-activation $\mathbf{z}^{[l]}$ of the corresponding input row $\mathbf{x}$ in $\mathbf{X}$.

From this pre-activation, the output of the layer l can be calculated by applying the activation function $g^{[l]}$ to each row of $\mathbf{Z}^{[l]}$:

$$\mathbf{A}^{[l]} = g(\mathbf{Z}^{[l]}) = \begin{bmatrix} g(\mathbf{z}_1^{[l]}) \\ g(\mathbf{z}_2^{[l]}) \\ \vdots \\ g(\mathbf{z}_n^{[l]}) \end{bmatrix}$$

where $\mathbf{A}^{[l]}$ is an $n \times q$ matrix.

2.2 Cost Function

In order to measure the progress of learning, a function

$$J(\mathbf{Y}_{\text{true}}, \mathbf{Y}_{\text{pred}})$$

is defined which takes as inputs: the $n \times q_L$ matrix of true labels $\mathbf{Y}_{\text{true}}$, and the $n \times q_L$ matrix of outputs of the neural network $\mathbf{Y}_{\text{pred}} = \mathbf{A}^{[L]}$. The goal of learning is to minimize this cost function, which requires the computation of gradients, which is done via back-propagation.

The cost function defined to be an average of loss functions

$$J(\mathbf{Y}_{\text{true}}, \mathbf{A}^{[L]}) = \frac{1}{n}\sum_{i=1}^{n} L(\mathbf{Y}_{i,\text{true}}, \mathbf{A}_i^{[L]})$$

where $\mathbf{Y}_{i,\text{true}}$ is the $1 \times q_L$ row vector corresponding the the correct label for the input row $\mathbf{X}_i$ and $\mathbf{A}_i^{[L]} = \mathbf{a}_i^{[L]}$ is the $1 \times q_L$ row vector corresponding to the neural network's output for the input row $\mathbf{X}_i$.

Some common examples of loss functions are:

the Mean Squared Error (MSE) loss

$$L(\mathbf{Y}_{i,\text{true}}, \mathbf{A}_i^{[L]}) = \frac{1}{2}(y_{i1} - a_{i1})^2 + \frac{1}{2}(y_{i2} - a_{i2})^2 + \cdots + \frac{1}{2}(y_{iq_L} - a_{iq_L})^2;$$

the Binary Cross Entropy (BCE) loss

$$L(\mathbf{Y}_{i,\text{true}}, \mathbf{A}_i^{[L]}) = \sum_{j=1}^{q_L}[y_{ij}\log(a_{ij}) + (1 - y_{ij})\log(1 - a_{ij})];$$

and the Categorical Cross Entropy (CCE) loss

$$L(\mathbf{Y}_{i,\text{true}}, \mathbf{A}_i^{[L]}) = \sum_{j=1}^{q_L}[y_{ij}\log(a_{ij})].$$

The BCE loss tends to be used in binary classification tasks, e.g. determining whether a handwritten digit is a 0 or a 1; or determining whether or not a T shirt appears in an image. In these cases, the vector of true labels y_i is populated with 0s (False) and 1s (True). The CCE loss tends to be used in categorization tasks, e.g. determining whether the animal in an image is a cat, dog,or mouse (assuming it's definitely one of them); or predicting the next token (from 50000 candidate tokens) in a sequence of tokenized words to ultimately output a string forming a plausible sentence - which is what the large language models, like ChatGPT, do. The vector of true labels y_i in these tasks has one 1 at some index j and 0s elsewhere, as expected.

Notice that the true labels are fixed, and the goal is to minimize the cost as a proxy for getting the best prediction, meaning the real goal is to vary the output $\mathbf{A}^{[L]}$ such that the cost becomes as low as possible. Therefore, the true labels are treated as constants and the derivative of the cost function J with respect to the variables a are required.

The derivatives of the cost w.r.t output neuron j are as follows:

MSE:

$$\frac{\partial L}{\partial a_{ij}} = y_{ij} - a_{ij};$$

BCE:

$$\frac{\partial L}{\partial a_{ij}} = \frac{y_{ij}}{a_{ij}} - \frac{1 - y_{ij}}{1 - a_{ij}};$$

CCE:

$$\frac{\partial L}{\partial a_{ij}} = \frac{y_{ij}}{a_{ij}}.$$

Now the derivative of the cost w.r.t. each activation neuron is clear if any one of these loss functions is chosen. This will be useful later.

[Note: In practice, J is chosen to be a differentiable function so that the derivatives of the cost with respect to any given parameter θ can be computed via the chain rule. One justification of using a differentiable cost function J is that in the event that J was not differentiable, one would need to compute the cost at a value θ and at $\theta \pm \delta$ to select the direction (if any) that reduces the value of the cost. This is computationally expensive, hence computing the derivatives is an attractive alternative.]

2.3 Back-Propagation

Back-propagation is the method by which the gradients are calculated. In an MLP network, the gradients for the neurons $\mathbf{a}^{[L]}$ and pre-activations $\mathbf{z}^{[L]}$ in the final layer are calculated first, followed by the gradients for the neurons $\mathbf{a}^{[L-1]}$ and pre-activations $\mathbf{z}^{[L-1]}$ of the preceding layer, and so on until the gradients for layer 1 are calculated. From these values, gradients for the weights and biases can be calculated, which is the main goal.

2.3.1 Univariate Case

In order to calculate the derivative of the cost w.r.t. any given weight or bias, the derivative of the cost J w.r.t. certain neurons is necessary in order to use the chain rule successfully. Here it will be shown that if you know the derivative of J w.r.t. every neuron in layer l, then you know the derivative of J w.r.t. each neuron in layer $l-1$. Combining this with the base-case assumption - that the derivative of J w.r.t. each neuron in the output layer is known (which will be true if a differentiable cost function J is selected) - yields a proof by induction of the exact formula of the derivative:

$$\frac{\partial J}{\partial a_{ij}^{[l-1]}} = \sum_{t=1}^{q} \frac{\partial J}{\partial a_{it}^{[l]}} g^{[l]}(z_{it}^{[l-1]}) w_{jt}.$$

In all of the following proofs, i is fixed, w_{cd} is the c,d-entry of the weight matrix $\mathbf{W}^{[l]}$ for layer l, and b_d is the d-entry for the bias row vector $\mathbf{b}^{[l]}$ for layer l.

Result 2.1. *Derivative of the cost function J w.r.t. a neuron a.*

Proof. Consider $a_{ij}^{[l-1]}$, i.e. fix j, and consider $a_{it}^{[l]}$, i.e. fix t. Then we have the equation

$$a_{it}^{[l]} = g^{[l]}\left(\sum_{k=1}^{p} a_{ik} w_{kt} + b_k\right)$$

which implies

$$\frac{\partial a_{it}^{[l]}}{\partial a_{ij}^{[l-1]}} = g^{[l]\prime}\left(\sum_{k=1}^{p} a_{ik} w_{kt} + b_k\right) w_{jt}.$$

This equation says that nudging $a_{ij}^{[l-1]}$ by δ will nudge $a_{it}^{[l]}$ by roughly $\frac{\partial a_{it}^{[l]}}{\partial a_{ij}^{[l-1]}}\delta$, which will nudge the cost by roughly $\frac{\partial J}{\partial a_{it}^{[l]}} \frac{\partial a_{it}^{[l]}}{\partial a_{ij}^{[l-1]}}\delta$. But t was arbitrary, so the sum of these changes over all t will give the total change, thus,

$$\frac{\partial J}{\partial a_{ij}^{[l-1]}} = \sum_{t=1}^{q} \frac{\partial J}{\partial a_{it}^{[l]}} g^{[l]\prime}(z_{it}^{[l-1]}) w_{jt}.$$

□

It will also be useful to know the derivative $\frac{\partial J}{\partial z}$ of the cost J with respect to the pre-activation values z. This will also be shown by induction.

Result 2.2. *Derivative of the cost function J w.r.t. a pre-activation z.*

Proof. By the induction hypothesis, $\frac{\partial J}{\partial z_{ik}^{[l]}}$ for each $k \in \{1, \cdots, q_l\}$. Fixing $j \in \{1, \cdots, q_{l-1}\}$, the effect on the cost J caused by shifting $z_{ij}^{[l-1]}$ by δ (just through the pre-activation $z_{ik}^{[l]}$) is

$$\frac{\partial J}{\partial z_{ik}^{[l]}} \frac{\partial z_{ik}^{[l]}}{\partial z_{ij}^{[l-1]}},$$

thus, the total derivative is

$$\frac{\partial J}{\partial z_{ij}^{[l-1]}} = \sum_{s=1}^{q_l} \frac{\partial J}{\partial z_{is}^{[l]}} \frac{\partial z_{is}^{[l]}}{\partial z_{ij}^{[l-1]}}.$$

The only unknowns in the above equation are the $\frac{\partial z_{is}^{[l]}}{\partial z_{ij}^{[l-1]}}$. Considering the equation for $z_{ik}^{[l]}$ shows

$$z_{ik}^{[l]} = \sum_{t=1}^{q_{l-1}} [g^{[l-1]}(z_{it}^{[l-1]}) w_{tk}] + b_k,$$

therefore,

$$\frac{\partial z_{ik}^{[l]}}{\partial z_{ij}^{[l-1]}} = g^{[l-1]\prime}(z_{ij}^{[l-1]}) \cdot w_{jk}.$$

The total derivative is then

$$\frac{\partial J}{\partial z_{ij}^{[l-1]}} = \sum_{s=1}^{q_l} \frac{\partial J}{\partial z_{is}^{[l]}} g^{[l-1]\prime}(z_{ij}^{[l-1]}) w_{js},$$

which can be simplified to

$$\frac{\partial J}{\partial z_{ij}^{[l-1]}} = g^{[l-1]\prime}(z_{ij}^{[l-1]}) \sum_{s=1}^{q_l} \frac{\partial J}{\partial z_{is}^{[l]}} w_{js}.$$

□

The first derivative will be used in further calculations in the back-propagation algorithm, but the second is included for completeness. The derivatives of real interest are those of the cost J w.r.t. a weight w_{rs} or a bias b_s in layer l. These are used to update the parameters in the neural network via gradient descent.

Result 2.3. *Derivative of the cost function J w.r.t. a weight w_{rs}.*

Proof. Notice that w_{rs} only shows up in the equations for $a_{is}^{[l]}$ for each i, that is, for each input row $\mathbf{x}_i$:

$$a_{is}^{[l]} = g^{[l]}\left(\sum_{t=1}^{q_{l-1}} a_{it}^{[l-1]} w_{ts} + b_s\right),$$

therefore,

$$\frac{\partial J}{\partial w_{rs}} = \frac{\partial J}{\partial a_{is}^{[l]}} \frac{\partial a_{is}^{[l]}}{\partial w_{rs}}.$$

The derivative $\frac{\partial J}{\partial a_{is}^{[l]}}$ has already been calculated above, so all that's left to do is to determine $\frac{\partial a_{is}^{[l]}}{\partial w_{rs}}$. This is given by

$$\frac{\partial a_{is}^{[l]}}{\partial w_{rs}} = g^{[l]\prime}\left(\sum_{t=1}^{q_{l-1}} a_{it}^{[l-1]} w_{ts} + b_s\right) a_{ir}^{[l-1]}$$

So the total derivative is

$$\frac{\partial J}{\partial w_{rs}} = \frac{\partial J}{\partial a_{is}^{[l]}} g^{[l]\prime}\left(\sum_{t=1}^{q_{l-1}} a_{it}^{[l-1]} w_{ts} + b_s\right) a_{ir}^{[l-1]}.$$

□

Result 2.4. *Derivative of the cost function J w.r.t. a weight b_s.*

Proof. Notice that b_s only shows up in the equations for $a_{is}^{[l]}$ for each i, that is, for each input row $\mathbf{x}_i$:

$$a_{is}^{[l]} = g^{[l]}\left(\sum_{t=1}^{q_{l-1}} a_{it}^{[l-1]} w_{ts} + b_s\right),$$

therefore,

$$\frac{\partial J}{\partial b_s} = \frac{\partial J}{\partial a_{is}^{[l]}} \frac{\partial a_{is}^{[l]}}{\partial b_s}.$$

The derivative $\frac{\partial J}{\partial a_{is}^{[l]}}$ has already been calculated above, so all that's left to do is to determine $\frac{\partial a_{is}^{[l]}}{\partial b_s}$. This is given by

$$\frac{\partial a_{is}^{[l]}}{\partial b_s} = g^{[l]\prime}\left(\sum_{t=1}^{q_{l-1}} a_{it}^{[l-1]} w_{ts} + b_s\right).$$

So the total derivative is

$$\frac{\partial J}{\partial b_s} = \frac{\partial J}{\partial a_{is}^{[l]}} g^{[l]\prime}\left(\sum_{t=1}^{q_{l-1}} a_{it}^{[l-1]} w_{ts} + b_s\right).$$

□

The three derivatives required for the back-propagation algorithm are:

$$\frac{\partial J}{\partial a_{ij}^{[l-1]}} = \sum_{t=1}^{q} \frac{\partial J}{\partial a_{it}^{[l]}} g^{[l]\prime}(z_{it}^{[l-1]}) w_{jt} \tag{1}$$

$$\frac{\partial J}{\partial w_{rs}} = \frac{\partial J}{\partial a_{is}^{[l]}} g^{[l]\prime}\left(\sum_{t=1}^{q_{l-1}} a_{it}^{[l-1]} w_{ts} + b_s\right) a_{ir}^{[l-1]} \tag{2}$$

$$\frac{\partial J}{\partial b_s} = \frac{\partial J}{\partial a_{is}^{[l]}} g^{[l]\prime}\left(\sum_{t=1}^{q_{l-1}} a_{it}^{[l-1]} w_{ts} + b_s\right) \tag{3}$$

2.3.2 Multivariate Case

The above proofs rely on the assumption that the activation functions $g^{[l]}$ and $g^{[l-1]}$ act element-wise on the column vectors $\mathbf{z}_i^{[l]}$ and $\mathbf{z}_i^{[l-1]}$, respectively, i.e. the activation functions are univariate. Some examples of univariate activation functions will be given later. In the cases where the activation functions are multivariate (e.g. a softmax activation function), the derivative calculations will involve a Jacobian matrix for the activation function. Every univariate function can be generalized to a multivariate function, therefore, the back-propagation algorithm will use the multivariate derivatives.

Definition 2.1 (Jacobian Matrix)**.** *If $\boldsymbol{f}\colon \mathbb{R}^n \mapsto \mathbb{R}^m$ is a function, where*

$$\boldsymbol{f}(x) = \left[f_1(x), f_2(x), \cdots, f_m(x)\right],$$

then the Jacobian matrix of f is:

$$D\boldsymbol{f} = \begin{bmatrix} \frac{\partial f_1}{\partial x_1} & \frac{\partial f_1}{\partial x_2} & \cdots & \frac{\partial f_1}{\partial x_n} \\ \frac{\partial f_2}{\partial x_1} & \frac{\partial f_2}{\partial x_2} & \cdots & \frac{\partial f_2}{\partial x_n} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial f_m}{\partial x_1} & \frac{\partial f_m}{\partial x_2} & \cdots & \frac{\partial f_m}{\partial x_n} \end{bmatrix}$$

In other words, the i, j entry of $D\boldsymbol{f}$ is $\frac{\partial f_i}{\partial x_j}$.

In the case of an activation function $\mathbf{g} : \mathbb{R}^q \mapsto \mathbb{R}^q$

$$D\mathbf{g} = \begin{bmatrix} \frac{\partial g_1}{\partial x_1} & \frac{\partial g_1}{\partial x_2} & \cdots & \frac{\partial g_1}{\partial x_q} \\ \frac{\partial g_2}{\partial x_1} & \frac{\partial g_2}{\partial x_2} & \cdots & \frac{\partial g_2}{\partial x_q} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial g_q}{\partial x_1} & \frac{\partial g_q}{\partial x_2} & \cdots & \frac{\partial g_q}{\partial x_q} \end{bmatrix}$$

It can be proven by induction that if the derivative $\frac{\partial J}{\partial a_{ik}^{[l]}}$ is known for each $k \in \{1, \cdots q_l\}$, then $\frac{\partial J}{\partial a_{ij}^{[l-1]}}$ can be found for each $j \in \{1, \cdots, q_{l-1}\}$. First, the derivative of J w.r.t. the pre-activations will be found, assuming the derivative $\frac{\partial J}{\partial a_{ik}^{[l]}}$ is known.

Result 2.5. *Derivative of the cost function J w.r.t. a pre-activation z.*

Proof. Consider $z_{ik}^{[l]}$, i.e. fix k. Then the total derivative of the cost J w.r.t. $z_{ik}^{[l]}$ is the dot product of the derivatives w.r.t. neurons and the k-th column of the jacobian matrix:

$$\frac{\partial J}{\partial z_{ik}^{[l]}} = \begin{bmatrix} \frac{\partial J}{\partial a_{i1}^{[l]}} \\ \frac{\partial J}{\partial a_{i2}^{[l]}} \\ \vdots \\ \frac{\partial J}{\partial a_{iq}^{[l]}} \end{bmatrix} \cdot \begin{bmatrix} \frac{\partial g_1}{\partial z_{ik}^{[l]}} \\ \frac{\partial g_2}{\partial z_{ik}^{[l]}} \\ \vdots \\ \frac{\partial g_q}{\partial z_{ik}^{[l]}} \end{bmatrix} = \sum_{t=1}^{q} \frac{\partial J}{\partial a_{it}^{[l]}} \frac{\partial g_t}{\partial z_{ik}^{[l]}}$$

□

Result 2.6. *Derivative of the cost function J w.r.t. a neuron a.*

Proof. Consider $a_{ij}^{[l-1]}$, i.e. fix j. Then the total derivative of the cost J w.r.t. $a_{ij}^{[l-1]}$ is

$$\frac{\partial J}{\partial a_{ij}^{[l-1]}} = \sum_{s=1}^{q} \frac{\partial J}{\partial z_{is}^{[l]}} \frac{\partial z_{is}^{[l]}}{\partial a_{ij}^{[l-1]}}.$$

But $\frac{\partial J}{\partial z_{is}^{[l]}}$ is known from the previous result and

$$z_{is}^{[l]} = \sum_{t=1}^{q_{l-1}} a_{it}^{[l-1]} w_{ts} + b_s$$

which implies that

$$\frac{\partial z_{is}^{[l]}}{\partial a_{ij}^{[l-1]}} = w_{js}.$$

Therefore, the required derivative is

$$\frac{\partial J}{\partial a_{ij}^{[l-1]}} = \sum_{s=1}^{q} \frac{\partial J}{\partial z_{is}^{[l]}} w_{js}.$$

□

With these two results, the derivatives of the cost J w.r.t. the arbitrary weight w_{jk} and bias b_k of layer l can be found.

Result 2.7. *Derivative of the cost function J w.r.t. a weight w_{jk}.*

Proof. The weight w_{jk} only shows up in the equation for $z_{ik}^{[l]}$ for each i:

$$z_{ik}^{[l]} = \sum_{t=1}^{q_{l-1}} a_{it}^{[l-1]} w_{tk} + b_k,$$

thus,

$$\frac{\partial J}{\partial w_{jk}} = \frac{\partial J}{\partial z_{ik}^{[l]}} \frac{\partial z_{ik}^{[l]}}{\partial w_{jk}} = \frac{\partial J}{\partial z_{ik}^{[l]}} a_{ij}^{[l-1]}.$$

□

Result 2.8. *Derivative of the cost function J w.r.t. a bias b_k.*

Proof. The bias b_k only shows up in the equation for $z_{ik}^{[l]}$ for each i:

$$z_{ik}^{[l]} = \sum_{t=1}^{q_{l-1}} a_{it}^{[l-1]} w_{tk} + b_k,$$

thus,

$$\frac{\partial J}{\partial b_k} = \frac{\partial J}{\partial z_{ik}^{[l]}} \frac{\partial z_{ik}^{[l]}}{\partial b_k} = \frac{\partial J}{\partial z_{ik}^{[l]}}.$$

□

The four required partial derivatives are:

$$\frac{\partial J}{\partial z_{ik}^{[l]}} = \sum_{t=1}^{q} \frac{\partial J}{\partial a_{it}^{[l]}} \frac{\partial g_t}{\partial z_{ik}^{[l]}} \tag{4}$$

$$\frac{\partial J}{\partial a_{ij}^{[l-1]}} = \sum_{s=1}^{q} \frac{\partial J}{\partial z_{is}^{[l]}} w_{js} \tag{5}$$

$$\frac{\partial J}{\partial w_{jk}} = \frac{\partial J}{\partial z_{ik}^{[l]}} a_{ij}^{[l-1]} \tag{6}$$

$$\frac{\partial J}{\partial b_k} = \frac{\partial J}{\partial z_{ik}^{[l]}}. \tag{7}$$

From these, the general back-propagation algorithm will be formed by first computing $\frac{\partial J}{\partial a^{[L]}}$ (which is just the simple derivative of the cost), then computing $\frac{\partial J}{\partial z^{[L]}}$ with (4). Then a loop begins to compute $\frac{\partial J}{\partial a^{[L-1]}}$ using (5), followed by $\frac{\partial J}{\partial z^{[L-1]}}$ using (4) for layer l, then compute $\frac{\partial J}{\partial a^{[L-2]}}$ using (5), and so on until $\frac{\partial J}{\partial z^{[1]}}$ is computed for layer 1, which is the first hidden layer. Once all of these are calculated, (6) and (7) can be used to compute the derivative of any weight and bias in the MLP network when the input row $\mathbf{x}_i$ is considered.

3 Implementation Details

Gradient descent is the method by which parameters are updated in a neural network and involves the computation of the gradients, which is handled by back-propagation. In Stochastic Gradient Descent (SGD) parameter updates have the following form:

$$\theta_t = \theta_{t-1} - \alpha \left(\frac{\partial J}{\partial \theta} \right)_t$$

where θ is a specific weight w or bias b in the neural network; α is the learning rate; and J is the cost function.

3.1 Optimizers

Optimizers are used to speed up the progress of learning by aiming to update the parameters of the neural network in the direction of steepest descent. SGD is the default optimizer. The optimizers explored in this paper are:

- Stochastic Gradient Descent (SGD)
- Momentum [8]
- Root Mean Square Propagation (RMSProp) [9]
- Adaptive Moment Estimation (Adam) [5]

3.1.1 Exponentially Weight Moving Averages (EWMA or EMA)

Momentum [8], RMSProp [9], and Adam [5] require the use of exponentially weighted averages. To understand this, consider the finite sequence

$$(x_t)_{t=0}^{T}.$$

If the scalar β is selected in the open interval $(0, 1)$, then the exponentially weighted average of (x_t) is

$$\begin{aligned} \text{EMA}_T &= (1-\beta)x_T + \beta(1-\beta)x_{T-1} + \beta^2(1-\beta)x_{T-2} + \cdots + \beta^T(1-\beta)x_0 \\ \text{EMA}_T &= (1-\beta)x_T + \beta \text{EMA}_{T-1}. \end{aligned}$$

Notice that the weights $(\beta^{t-1}(1-\beta))_{t=1}^{T+1}$ for a geometric sequence whose sum is $1 - \beta^{T+1} \neq 1$. This implies that the value computed is not a mathematical expectation since the "probabilities" do not sum to 1. This will be useful later.

3.1.2 SGD

Fix a parameter θ. SGD uses just the gradient to make an update to θ at time t

$$\theta_t = \theta_{t-1} - \alpha \cdot \frac{\partial J}{\partial \theta}.$$

3.1.3 Momentum [8]

Fix a parameter θ. The Momentum optimizer [8] uses the term v_t to make an update to θ at time t

$$\theta_t = \theta_{t-1} - \alpha \cdot v_t$$

where $v_0 = 0$ and

$$v_t = \beta \cdot v_{t-1} + (1-\beta) \cdot \frac{\partial J}{\partial \theta}.$$

In order to make the update value a mathematical expectation of previous gradients, v_t is divided by a bias correction term $1 - \beta^t$ giving the update rule

$$\theta_t = \theta_{t-1} - \alpha \cdot \frac{v_t}{1-\beta^t}.$$

The MLP network tested in this paper will use $\beta = 0.9$ for the Momentum optimizer [8].

3.1.4 RMSProp [9]

Fix a parameter θ. The RMSProp optimizer [9] uses the term s_t to make an update to θ at time t

$$\theta_t = \theta_{t-1} - \alpha \cdot \frac{\frac{\partial J}{\partial \theta}}{\sqrt{s_t} + \epsilon}$$

where $s_0 = 0$, ϵ is a small constant, and

$$s_t = \beta \cdot s_{t-1} + (1 - \beta) \cdot \left(\frac{\partial J}{\partial \theta}\right)^2.$$

In order to make the square-rooted value a mathematical expectation of previous gradients, s_t is divided by a bias correction term $1 - \beta^t$ giving the update rule

$$\theta_t = \theta_{t-1} - \alpha \cdot \frac{\frac{\partial J}{\partial \theta}}{\sqrt{s_t/(1 - \beta^t)} + \epsilon}$$

The MLP network tested in this paper will use $\beta = 0.999$ for the Momentum optimizer [8].

3.1.5 Adam [5]

Fix a parameter θ. The Adam optimizer [5] combines both Momentum [8] and RMSprop [9] to give the update to θ at time t

$$\theta_t = \theta_{t-1} - \alpha \cdot \frac{v_t}{\sqrt{s_t} + \epsilon}$$

where $v_0 = s_0 = 0$, ϵ is a small constant, and

$$v_t = \beta_1 \cdot v_{t-1} + (1 - \beta_1) \cdot \frac{\partial J}{\partial \theta}, \qquad s_t = \beta_2 \cdot s_{t-1} + (1 - \beta_2) \cdot \left(\frac{\partial J}{\partial \theta}\right)^2.$$

In order to make the numerator of the update term a mathematical expectation of previous gradients, v_t is divided by a bias correction term $1 - \beta_1^t$; and in order to make the square-rooted value a mathematical expectation of previous gradients, s_t is divided by a bias correction term $1 - \beta_2^t$ giving the update rule

$$\theta_t = \theta_{t-1} - \alpha \cdot \frac{v_t/(1 - \beta_1^t)}{\sqrt{s_t/(1 - \beta_2^t)} + \epsilon}$$

The MLP network tested in this paper will use $\beta_1 = 0.9$ and $\beta_2 = 0.999$ for the Adam optimizer [5].

3.2 Parameter Initialization

The choice of parameter initialization has a significant effect on the learning performance of neural networks. Here, two methods of weight initialization will be explored: Uniform Initialization and He Uniform Initialization.

3.2.1 Uniform Initialization

With Uniform Initialization, the weight matrices are initialized to have random entries sampled from the Uniform [-1,1] distribution. The bias row vectors are initialized to zero vectors.

3.2.2 He Uniform Initialization [3]

With He Uniform Initialization, for each l the weight matrix W for layer l is initialized to have random entries sampled from the Uniform $\left[-\sqrt{\frac{6}{p}}, \sqrt{\frac{6}{p}}\right]$ distribution, where p is the number of rows of W which is equal to the number of neurons in layer $l - 1$. The bias row vectors are initialized to zero vectors.

The theoretical justification for the multiplier $\sqrt{\frac{6}{p}}$ is that this is the multiplier required to preserve the variance of the pre-activations from layer l to layer $l + 1$ when ReLU functions are used. This was shown in [3] which used an approach similar to that in [2].

3.3 L2-Regularization

Recall the cost function J has the form

$$J(\mathbf{Y}_{\text{true}}, \mathbf{A}^{[L]}) = \frac{1}{n} \sum_{i=1}^{n} L(\mathbf{Y}_{i,\text{true}}, \mathbf{A}_i^{[L]})$$

where $\mathbf{Y}_{\text{true}}$ is an $n \times q$ matrix whose rows correspond to the true labels; $\mathbf{A}^{[L]}$ is an $n \times q$ matrix whose rows correspond to the neural network's output prediction of the true labels; $\mathbf{Y}_{i,\text{true}}$ is a $1 \times q$ row vector corresponding to the true label for the input $\mathbf{x}_i$; and $\mathbf{A}_i^{[L]}$ is a $1 \times q$ row vector corresponding to the neural network's output prediction for the input $\mathbf{x}_i$.

Regularization is a technique that aims to reduce overfitting in neural networks so that they generalize well to unseen data. It works by adding a regularization term, to the cost function to give the new cost function

$$J(\mathbf{Y}_{\text{true}}, \mathbf{A}^{[L]}) = \frac{1}{n} \sum_{i=1}^{n} L(\mathbf{Y}_{i,\text{true}}, \mathbf{A}_i^{[L]}) + \frac{\lambda}{2n} \sum w^2$$

where λ is a positive scalar and $\sum w^2$ represents the sum of all weights in the neural network. It is this cost function that the MLP network will aim to minimize during gradient descent.

In this case, the regularization term is $\frac{\lambda}{2n} \sum w^2$ which corresponds to L2 regularization. In this paper, the four choices for the L2-regularization [4] coefficient λ explored are $\lambda = 0, 1, 2$, and 3.

(L1 regularization has the regularization term $\frac{\lambda}{n} \sum |w|$ and is also popular, but is not explored in this paper.)

4 Experiments and Results

4.1 Datasets

Three datasets were used to produce the results in this paper: the Canadian Institute For Advanced Research 10-Class (CIFAR-10) image classification dataset [6]; the Modified National Institute of Standards and Technology (MNIST) dataset of handwritten digits [1]; and the Fashion-MNIST dataset [10]. The following shows the amount of data used in the training and cross-validation performance measure:

- CIFAR-10 [6] : 10000 training, 10000 cross-validation
- MNIST Handwritten Digits [1] : 10000 training, 10000 cross-validation
- Fashion-MNIST [10] : 10000 training, 10000 cross-validation

The results from the CIFAR-10 dataset [6] will be the main focus, with the other two datasets used supplementarily.

4.2 Model Architecture

For all three datasets, MLP networks were trained. For each dataset, the MLP networks had 4 identical hidden layers of 128 neurons with Leaky Rectified Linear Unit (LReLU) activation functions with leakage coefficient $\alpha = 0.1$. The output layer had 10 neurons and a softmax activation function.

For both of the MNIST datasets [1] [10], the MLP networks were trained on mini-batches of size 100 for 20 epochs. For the CIFAR-10 dataset [6], the MLP networks were also trained on mini-batches of size 100, but were trained for a total of 40 epochs to give the models more time to converge.

4.3 Results

4.3.1 Optimizer Performance

On all datasets, with $U[-1, 1]$ weight initialization there was no significant learning progress on all optimizer and λ combinations, except for the Adam optimizer [5] with $\lambda = 3$ (Figure 1).

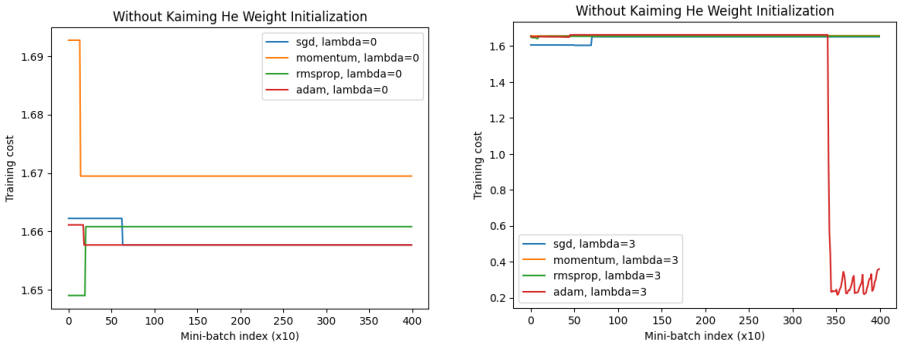
On all datasets with He Uniform weight initialization [3] there was significant learning progress on all optimizer and λ combinations.

On the CIFAR-10 dataset [6], models trained with SGD consistently had the lowest training and cross-validation cost with the smallest fluctuations as well as the highest training and cross-validation accuracies for each value of λ tested. Models trained with the Momentum optimizer [8] followed, which was followed by the models trained with RMSProp [9] and Adam [5] closely tied in both cost and accuracy metrics during the training process (Figure 2). The final costs and accuracies for models trained on the CIFAR-10 dataset [6] for $\lambda = 0$ are summarized in Table 1. We observed large periodic spikes in the cost and accuracy graphs for models trained with RMSProp [9].

Optimizer	Training Cost	CV Cost	Training Accuracy	CV Accuracy
SGD	0.2498	0.2857	0.38	0.3369
Momentum [8]	0.3138	0.3379	0.3038	0.2739
RMSProp [9]	0.3787	0.4177	0.3051	0.2675
Adam [5]	0.4089	0.4392	0.2987	0.272

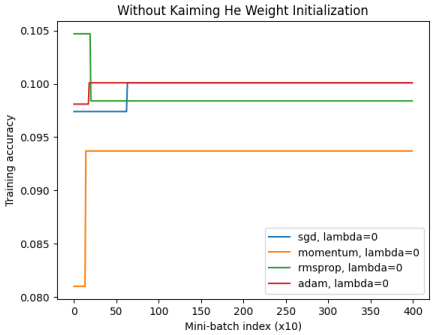
Table 1: Final training cost, cross-validation cost, training accuracy, and cross-validation accuracy for the He-initialized [3] MLPs trained with each optimizer with $\lambda = 0$ on the CIFAR-10 dataset [6].

On the MNIST digits dataset [1], the main difference observed in models trained with different optimizers was the initial convergence rate (Figure 3). The models with the highest convergence rates initially were those trained using the RMSProp [9] and Adam [5] optimizers; followed by the models trained with Momentum [8]; and finally the models trained using SGD. Although training was terminated at epoch 20, the slope of the accuracy graphs remained positive and the slope of the cost graphs remained negative up to epoch

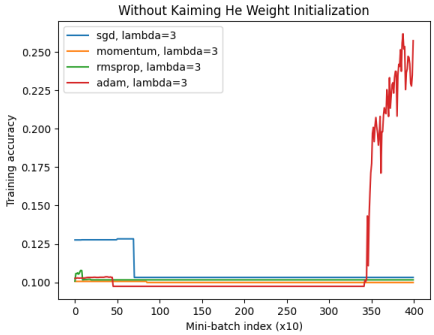


(a) Training cost with $\lambda = 0$

(b) Training cost with $\lambda = 3$



(c) Training accuracy with $\lambda = 0$



(d) Training accuracy with $\lambda = 3$

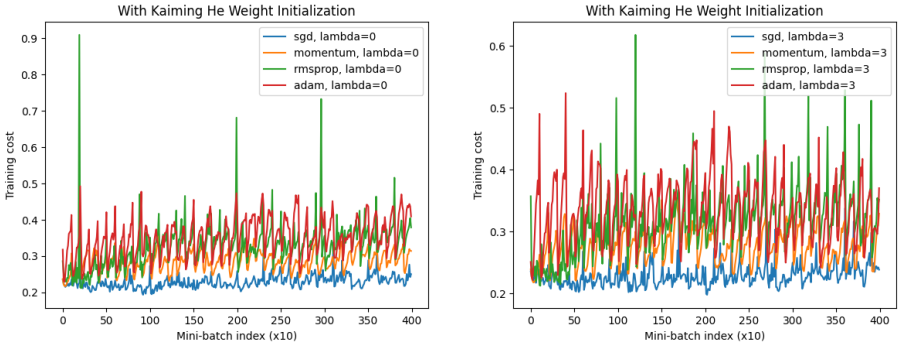
Figure 1: CIFAR-10 [6]. Training cost ((a) and (b)) and training accuracy ((c) and (d)) for each of the four optimizers tested without He Uniform weight initialization [3]. Accuracy is measured on a scale from 0 to 1. We see that the cost and accuracy were effectively constant with the L2-regularization [4] coefficient $\lambda = 0$ and $\lambda = 3$ (which was also true for $\lambda = 1$ and 2), but, for $\lambda = 3$, the cost and accuracy began to move significantly in the direction of progress at around epoch 33.

20, suggesting that training had not yet reached convergence. From roughly epoch 5 onward, the training accuracies for models trained were practically equivalent across optimizers when accounting for fluctuations. Models trained with RMSProp [9] again exhibited large periodic spikes in the cost and accuracy graphs. The final costs and accuracies for models trained on the MNIST digits dataset [1] for $\lambda = 1$ are summarized in Table 2.

Optimizer	Training Cost	CV Cost	Training Accuracy	CV Accuracy
SGD	0.0371	0.0482	0.8996	0.8711
Momentum [8]	0.0376	0.0521	0.9099	0.8799
RMSProp [9]	0.0283	0.0378	0.9317	0.9142
Adam [5]	0.0435	0.0633	0.9145	0.881

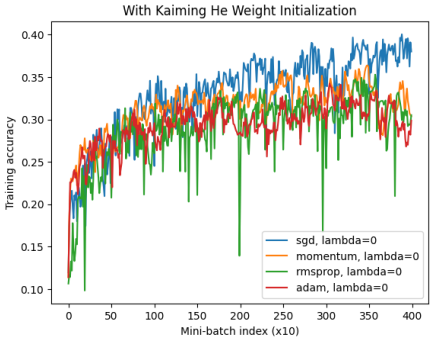
Table 2: Final training cost, cross-validation cost, training accuracy, and cross-validation accuracy for the He-initialized [3] MLPs trained with each optimizer with $\lambda = 0$ on the MNIST handwritten digits dataset [1].

On the Fashion-MNIST dataset [10], the training cost and accuracy exhibited a pattern similar to that of the models trained on the MNIST digits dataset [1]: for most choices of λ, the models with the highest initial rates of convergence (Figure 4) were those trained using RMSProp [9] and Adam [5]; followed by those trained with Momentum [8]; and finally the models trained with SGD. The cost and accuracy graphs for RMSProp [9] again exhibited large periodic spikes (Figure 4). Table 3 summarizes the final costs and accuracies for the models trained on the Fashion-MNIST dataset [10] with $\lambda = 0$.

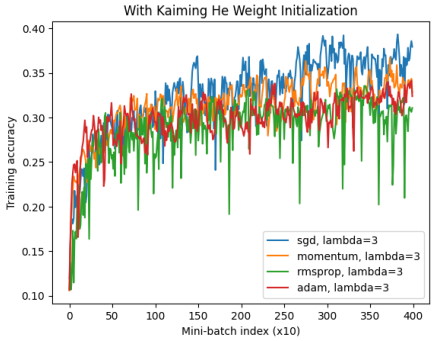


(a) Training cost with $\lambda = 0$

(b) Training cost with $\lambda = 3$



(c) Training accuracy with $\lambda = 0$



(d) Training accuracy with $\lambda = 3$

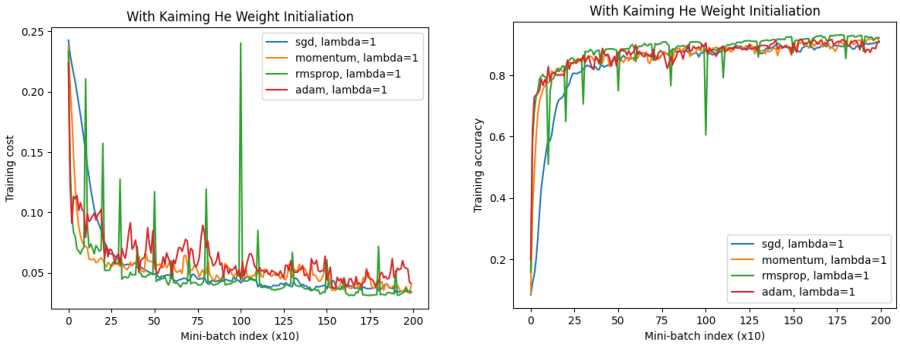
Figure 2: CIFAR-10 [6]. Training cost ((a) and (b)) and training accuracy ((c) and (d)) for each of the four optimizers tested with He Uniform weight initialization [3]. Accuracy is measured on a scale from 0 to 1. The training was very noisy for all optimizers, but was notably the lowest for SGD, which corresponded with higher long term progress in the training accuracy for SGD compared to the other optimizers.

Optimizer	Training Cost	CV Cost	Training Accuracy	CV Accuracy
SGD	0.0717	0.0778	0.7876	0.7705
Momentum [8]	0.0748	0.0859	0.8173	0.795
RMSProp [9]	0.0818	0.0960	0.8155	0.7911
Adam [5]	0.0913	0.1050	0.8118	0.7918

Table 3: Final training cost, cross-validation cost, training accuracy, and cross-validation accuracy for the He-initialized [3] MLPs trained with each optimizer with $\lambda = 0$ on the Fashion-MNIST dataset [10].

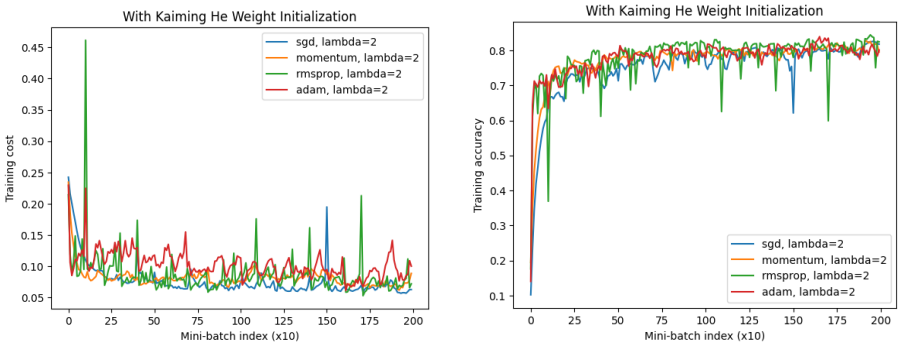
4.3.2 L2-Regularization [4] Performance

No significant differences were visible in the generalization performance when varying the L2-regularization [4] coefficients λ for any of the models trained. Figure 5 shows the generalization gap for the CIFAR-10 models for $\lambda = 1$ and $\lambda = 3$ for each optimizer tested.



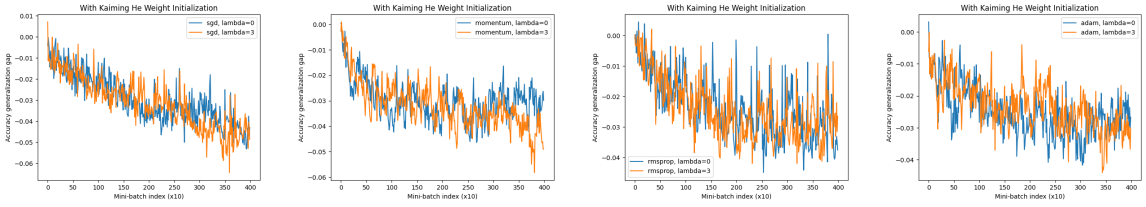
(a) Training cost with $\lambda = 1$ (b) Training accuracy with $\lambda = 1$

Figure 3: MNIST digits [1] (a) training cost and (b) training accuracy for each of the four optimizers tested with He Uniform weight initialization [3] and L2-regularization [4] coefficient $\lambda = 1$. Accuracy is measured on a scale from 0 to 1.



(a) Training cost with $\lambda = 2$ (b) Training accuracy with $\lambda = 2$

Figure 4: Fashion-MNIST [10] (a) training cost and (b) training accuracy for each of the four optimizers tested with He Uniform weight initialization [3] and L2-regularization [4] coefficient $\lambda = 2$. Accuracy is measured on a scale from 0 to 1.



(a) SGD (b) Momentum [8] (c) RMSProp [9] (d) Adam [5]

Figure 5: CIFAR-10 [6]. Accuracy generalization gap for each optimizer for L2-regularization [4] coefficients $\lambda = 0$ and $\lambda = 3$. No significant difference in generalization performance can be seen by increasing λ for any optimizer choice.

However, as mentioned earlier, the model trained on the CIFAR-10 dataset with the Adam optimizer and $\lambda = 3$ without He Uniform weight initialization began to make significant learning progress at roughly epoch 34 of training as shown in Figure 1.

5 Discussion

In the results section, it was seen that initializing weights uniformly at random in the interval $[-1, 1]$ yielded little if any learning progress, while He Uniform weight initialization [3] yielded significant learning progress in all the models for all optimizers tested and choices of L2-regularization [4] coefficient λ tested. However there are still questions about the convergence behavior caused by the optimizers and some choices of λ.

To answer these questions it will be useful to recall the update formulas for the optimizers:

- SGD
$$\theta_t = \theta_{t-1} - \alpha \cdot \frac{\partial J}{\partial \theta}.$$
- Momentum [8]
$$\theta_t = \theta_{t-1} - \alpha \cdot \frac{v_t}{1 - \beta^t}.$$
- RMSProp [9]
$$\theta_t = \theta_{t-1} - \alpha \cdot \frac{\frac{\partial J}{\partial \theta}}{\sqrt{s_t} + \epsilon}$$
- Adam [5]
$$\theta_t = \theta_{t-1} - \alpha \cdot \frac{v_t}{\sqrt{s_t} + \epsilon}$$

where v_t is an exponentially weighted sum of the previous gradients and s_t is an exponentially weighted sum of the squares of the previous gradients (the bias correction terms have been omitted for simplicity).

5.1 Optimizer performance

It was clear that SGD outperformed the other optimizers in the models as shown in Figure 2.

A possible explanation for this is that SGD beat Momentum [8] since Momentum [8] is an average, meaning it reacts to persistent changes in the overall gradient direction in the optimization landscape more slowly than SGD. This may prevent Momentum [8] from making the sharp turns necessary that would lead to the steepest descent, while SGD would be able to perform the sharp turns.

As for RMSProp [9] and Adam [5], notice the denominator of the update step is $\sqrt{s_t}+\epsilon$. In regions where the optimization landscape is consistently flat (has a small gradient), s_t will be close to zero, meaning that the update for both RMSProp [9] and Adam [5] will be larger on that basis. The lack of spikes in the cost and accuracy graphs for Adam [5] can be attributed to the fact that v_t will also necessarily be small whenever s_t was small, thus the numerator of the update for Adam [5] will also be small when the denominator is small. The same cannot be said for RMSProp [9] since there could be a smaller subregion with a larger gradient within the overall flat region of the optimization landscape. If the parameter vector θ ever touched this subregion, it would lead to an unreasonably large update step that could overshoot the local minimum and possibly land in a region on the opposite side of the local minimum with a steep enough gradient in the opposite direction such that the next update step lands the parameter vector θ back on the flat region. This steep gradient would also mean that a large value was added to s_t meaning RMSProp [9] could continue to take a few more steps towards the local minimum without being subjected to another event where s_t was too small.

5.2 L2-Regularization [4] without He Uniform [3]

In Figure 1, plots (b) and (d) illustrate a case where weights were initialized uniformly at random in the interval $[-1, 1]$ and significant learning began after around 33 epochs or 3300 mini-batches. This phenomenon may be explained by considering the L2-regularized cost function

$$J(\mathbf{Y}_{\text{true}}, \mathbf{A}^{[L]}) = \frac{1}{n} \sum_{i=1}^{n} L(\mathbf{Y}_{i,\text{true}}, \mathbf{A}_i^{[L]}) + \frac{\lambda}{2n} \sum w^2$$

where $\sum w^2$ is the sum of all weights in the network. Differentiating this w.r.t. a single weight w in the network gives

$$\frac{\partial J}{\partial w} = \frac{1}{n}\sum_{i=1}^{n}\frac{\partial L_i}{\partial w} + \frac{\lambda}{n}w.$$

This shows that even when θ is in a region where the optimization landscape is flat ($\frac{1}{n}\sum_{i=1}^{n}\frac{\partial L_i}{\partial w} \approx 0$), there will be a "force" ($\frac{\lambda}{n}w$ for each weight) pushing each weight w back towards the origin in the parameter update. This means that any weights initialized outside the He Uniform [3] range $[-\sqrt{\frac{6}{p}}, \sqrt{\frac{6}{p}}]$ could eventually move into this range if the optimization landscape along the path taken by θ remains relatively flat, converting the learning problem to one similar to the case of He Uniform weight initialization [3]. The "force" described previously would increase with larger values of λ, making this event more likely.

5.3 L2-Regularization [4] on Generalization Performance

L2-regularization [4] in neural networks is designed to incentivize neural networks to find solutions with small weights. If, without L2-regularization, the local minimum found at convergence had small weights, then adding L2-regularization [4] would not have had much of an effect on finding small weights. In other words, if there is no risk of overfitting, L2-regularization [4] would yield no significant generalization benefits.

The generalization performance was unaffected by the addition of L2-regularization [4] in this study as seen in Figure 5. Learning progress also seemed to be unhindered by the different choices of λ selected.

6 Conclusion

In this project, we saw that He Uniform [3] weight initialization $U[-\sqrt{6/p}, \sqrt{6/p}]$ is almost always better than uniform random weight initialization $U[-1, 1]$ - on every dataset tested, the performance of models trained with He Uniform weight initialization [3] produced significant learning progress. This is in contrast to the performance of the models trained with $U[-1, 1]$ weight initialization where all but one model made practically no learning progress.

All optimizers allowed MLP models with appropriately initialized weights to be trained effectively to varying degrees: for the models trained on the CIFAR-10 dataset [6], SGD was found to be the most effective for long term training progress when weights were initialized via He Uniform weight initialization [3] while remaining competitive in short term training progress. Momentum [8] followed SGD, and RMSProp [9] and Adam [5] followed Momentum. There may be evidence that Adam is the ideal optimizer for the optimization landscape presented by the CIFAR-10 dataset [6], but this conclusion has not been tested here.

All optimizers allowed MLP models with appropriately initialized weights to be effectively trained to virtually the same degree on both the MNIST digits and the Fashion-MNIST datasets, with the main point of difference being the initial convergence rate: for all values of λ tested with He Uniform weight initialization, RMSProp and Adam showed superior initial rates of convergence when compared to Momentum, which in turn had superior initial convergence rates to SGD generally. By epoch 3, differences in learning progress between optimizers were mostly gone and all optimizers seemed to be continuing to converge past epoch 20.

6.1 Limitations

The models trained with each optimizer were trained with a specific learning rate corresponding to that optimizer, meaning it is possible that any given learning rate selected was above or below the optimal learning rate, therefore, this may have interfered with the comparisons of the optimizers. If a viable solution to this is found, once the optimal learning rate for each optimizer is identified one can train a sufficiently large number of model for a fixed architecture to obtain mean performance statistics such as the mean training cost at each mini-batch, but this would be computationally expensive.

This study did not investigate the sizes of the gradients back-propagated through the network, meaning the effects of He Uniform initialization [3] were not investigated as thoroughly as possible. For a more thorough analysis, the gradients can be checked in the case where weights were i.i.d. $w \sim U[-1, 1]$; and in the case where weights were i.i.d. $w \sim U\left(-\sqrt{\frac{6}{p}}, \sqrt{\frac{6}{p}}\right)$, i.e. He Uniform weight initialization [3]. This requires the history of the some or all of the gradients calculated during the training process to be stored, which requires a large amount of storage.

This study only trained the MNIST [1] [10] models for 20 epochs and the CIFAR-10 [6] models for 40 epochs. This means that the training for some models was likely terminated before convergence. In this case, there may have been more useful insights obtained had models been allowed to train for longer, say 100 epochs each.

The datasets chosen did not overfit significantly enough for the effects of regularization to be observed clearly, therefore it may be useful to repeat this study on a dataset that an MLP is more prone to overfitting. Randomly generated data may be sufficient to test this.

Only a subset of the full datasets was used to train the models: the first 10000 of the official training sets for each of CIFAR-10 [6], MNIST digits [1], and Fashion-MNIST [10] were used as training data for the models, and the next 10000 images of the official training sets were used as cross-validation data. This means that the analysis may not be comparable to other studies, given that optimizing over a whole dataset may provide different results.

6.2 Future Work

The next step in this project is to implement and analyse the performance of a BNN using Variational Inference/Copula Variational Inference to approximate the distribution of the parameters in the BNN and ultimately make the resulting predictive integral tractable.

Another way this project can be extended is to explore the effects of second-order optimization methods along with a comparison to the first-order methods explored here. This would potentially lead to more efficient convergence [7].

References

[1] Li Deng. The mnist database of handwritten digit images for machine learning research [best of the web]. *IEEE Signal Processing Magazine*, 29(6):141–142, 2012.

[2] Xavier Glorot and Yoshua 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. JMLR Workshop and Conference Proceedings, 2010.

[3] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian 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.

[4] Arthur E Hoerl and Robert W Kennard. Ridge regression: Biased estimation for nonorthogonal problems. *Technometrics*, 12(1):55–67, 1970.

[5] Diederik P Kingma. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

[6] Alex Krizhevsky, Geoffrey Hinton, et al. Learning multiple layers of features from tiny images.(2009), 2009.

[7] Dong C Liu and Jorge Nocedal. On the limited memory bfgs method for large scale optimization. *Mathematical programming*, 45(1):503–528, 1989.

[8] David E Rumelhart, Geoffrey E Hinton, and Ronald J Williams. Learning representations by back-propagating errors. *nature*, 323(6088):533–536, 1986.

[9] T. Tieleman and G. Hinton. Lecture 6.5 – rmsprop. COURSERA: Neural Networks for Machine Learning, 2012.

[10] Han Xiao, Kashif Rasul, and Roland Vollgraf. Fashion-mnist: a novel image dataset for benchmarking machine learning algorithms. *arXiv preprint arXiv:1708.07747*, 2017.