

# Implementation and Empirical Analysis of Modifications to the Stochastic Gradient Descent Optimizer

Kenan Karavoussanos

November 21, 2018

**Abstract** The choice of Optimizer for an Artificial Neural Networks requires multiple considerations, including length of training time, reliance on initial training constants and ability to converge to a global minimum rather than a local minimum. This report introduces and evaluates improvements to the Stochastic Gradient Descent Optimizer. These improved Optimizers are then compared in the context of an Artificial Neural Network classifier on the MNIST dataset of handwritten digits. This comparison is not universally applicable due to the stochastic nature of different datasets and problems. The results contained within this report are specifically applicable to Multiclass classification problems.

## Contents

<b>1</b>	<b>Definitions and Acronyms</b>	<b>2</b>
1.1	Glossary . . . . .	3
1.2	Acronyms . . . . .	4
<b>2</b>	<b>Introduction</b>	<b>4</b>
<b>3</b>	<b>Methodology</b>	<b>5</b>
3.1	Methodology-I . . . . .	5
3.1.1	Detailed Concepts . . . . .	5
3.1.2	Dataset . . . . .	9
3.1.3	Architecture . . . . .	9
3.1.4	Performance Metrics . . . . .	9
3.1.5	Learning Process Algorithms . . . . .	10
3.1.6	Learning rate schedules . . . . .	12
3.2	Methodology-II . . . . .	13
3.2.1	Adaptive Learning Rate Optimizers . . . . .	13
3.2.2	Contrast with previous work . . . . .	14

<b>4</b>	<b>Analytic Results to be used</b>	<b>14</b>
4.1	ADAGRAD and RMSProp . . . . .	14
4.1.1	Derivation . . . . .	14
4.1.2	Analysis . . . . .	15
4.2	Adam . . . . .	15
4.2.1	Derivation . . . . .	15
4.2.2	Analysis . . . . .	15
<b>5</b>	<b>Experimental Setup and Computational Model</b>	<b>16</b>
5.1	Experimental Setup . . . . .	16
5.2	Computational Environment . . . . .	16
5.2.1	Hardware Specification . . . . .	16
5.3	Computational Model . . . . .	16
5.3.1	Software and Versions . . . . .	16
5.4	Data Sources . . . . .	17
5.5	Software Engineering practices and Public Accessibility . . . . .	17
<b>6</b>	<b>Results</b>	<b>17</b>
6.1	Figures . . . . .	18
6.2	Training Accuracy versus Iteration . . . . .	18
6.3	Training Loss versus Iteration . . . . .	19
6.4	Test Accuracy versus Iteration . . . . .	20
6.4.1	Table . . . . .	20
6.4.2	Plot . . . . .	21
6.5	Test Loss versus Iteration . . . . .	22
6.5.1	Table . . . . .	22
6.5.2	Plot . . . . .	23
6.6	Analysis . . . . .	23
6.6.1	SGD . . . . .	23
<b>7</b>	<b>Summary</b>	<b>23</b>
<b>8</b>	<b>References</b>	<b>24</b>
<b>9</b>	<b>Appendices</b>	<b>25</b>
9.1	Literature Survey . . . . .	25
9.2	Tables . . . . .	27
9.2.1	Iteration versus Accuracy . . . . .	27

# 1 Definitions and Acronyms

The following section contains definitions and elaborations on important, domain-specific, terms and acronyms. This section serves as an aid and reference to the reader of this report.

## 1.1 Glossary

- **Artificial Neural Network:** A massively distributed and parallelized processor made up of small computational units called Neurons that acquires knowledge from its environment through a learning process and stores this knowledge as synaptic weights between neurons [4].
- **Multi-Layer Feed-Forward Network:** An Artificial Neural Network where the neurons are arranged in distinct layers. Where input flows unidirectionally from the input layers to the output layer with no feedback to previous layers. Also referred to as a Multi-Layer Perceptron Network.
- **Loss Function:** A function that is used to evaluate the error in an Artificial Neural Network.
- **Optimizer:** A function that attempts to minimize the Loss Function i.e the error of an Artificial Neural Network by changing the values of the weights in the network.
- **Gradient Descent:** An Optimizer that uses the gradient of the Loss Function when minimizing.
- **Learning/Training Process:** The process of reading inputs, calculating the error and minimizing said error in an Artificial Neural Network.
- **Learning Rate:** The size of the updates an Optimizer makes to the weights of a Network during the Training Process.
- **Generalization:** The ability of a Neural Network to perform its learning task i.e classification on data it did not see during the learning process.
- **Overfitting:** The state of a Network where it fits to the training data too well resulting in poor generalization.
- **Epoch:** A single pass over the set of training inputs in the Training Process.
- **Hyperparameter:** A parameter of an Artificial Neural Network set before the start of the Training Process.
- **Regularization:** A hyperparameter that introduces a penalty for relying too heavily on a specific Neuron in the Network. This prevents the network from overfitting to the training data.
- **Dropout:** The act of temporarily removing a specific Neuron in a Neural Network during the Training Process to prevent the Network from relying too heavily on said Neuron. This also refers to a hyperparameter that determines the probability of Dropout occurring for a specific neuron.
- **Accuracy:** The proportion of correctly classified inputs made by an Artificial Neural Network to the total number of tested inputs.
- **MNIST Dataset:** A dataset of 28x28 pixel handwritten digits from 0 to 9.

## 1.2 Acronyms

- ANN: Artificial Neural Network
- MLP: Multi-Layer Perception Network
- SGD: Stochastic Gradient Descent
- STC: Search-then-Converge
- CLR: Cyclical Learning Rate
- RMS: Root Mean Squared

## 2 Introduction

The purpose of an Optimizer in the context of Artificial Neural Networks is to minimize the Loss Function describing the classification error of the ANN. Stochastic Gradient Descent forms the base for many cutting edge Optimizers used in Artificial Neural Networks. Certain strategies are employed in these optimizers to overcome the limitations SGD. This paper seeks to provide an overview of the theory behind these strategies and compare the Accuracy and Loss of the network once it has gone through the Learning Process.

The specific limitations of SGD are as follows:

1. SGD has slower performance when the gradient of the loss function in one dimension is drastically different from the gradient in other dimensions.
2. Stochastic Gradient Descent has a constant learning rate for the entire training epoch which can cause the ANN to overshoot minima and potentially fail to converge.
3. The learning rate is common across all parameters. This is suboptimal as we may want to perform different sized updates to different parameters at certain points.
4. The affinity to stay at a local minimum instead of searching for a better and or global minimum.
5. Stochastic Gradient Descent performs poorly around saddle points in the Loss Function.

Due to the large amount of computation and calculation required to train an ANN, it is infeasible to trace the training algorithm for more than a few training examples. As such the Loss and Accuracy are used to evaluate the performance of the ANN and its Optimizer. Slow optimizers become prohibitive to training for large datasets so time to convergence has been chosen as a comparison metric. This time to converge shall be measured by comparing the Loss and Accuracy with respect to the number of training examples seen during the learning process. The level of generalization of the network is the most important broad metric in the evaluation of a classifier, thus the final accuracy and loss over the test set i.e Test Accuracy and Test Loss have also been chosen as metrics.

**Significance of problem** The current body of research shows the benefits of each new optimizer and how it overcomes specific limitations. However, the trade-offs are rarely specifically studied and as such this paper serves as a means to correct the selection bias in the body of research.

For specific or niche problem domains and datasets, comparing each optimizer allows for the reader to select an appropriate Optimizer to fit within various real world constraints.

The evaluation of the each optimizer on the standard MNIST Dataset allows for empirical validation of previous theoretical study on Gradient Descent based Optimizers along our metrics.

**Application of the problem solution to other domains** Due to the generality provided by evaluation on a the standard MNIST Dataset, the problem solution is applicable across many domains. However, this generality also makes the comparison less accurate on niche datasets.

**Organisation of the rest of the proposal** The paper consists of the research methodology which describes basic concepts of the work, the experimental setup including basic pseudocoded algorithms and newly developed vectorizations of certain optimizers. Followed by a presentation and analysis of experimental results. Then the Experimental/Computational Setup describing the hardware and software specification of the experimental environment. The data sources and software engineering principles to be used in the future work are also described.

## 3 Methodology

This section will provide a detailed explanation of the concepts being used as well as the experimental process to be followed in this work. This will be split into two sections:

- Methodology-I
- Methodology-II

### 3.1 Methodology-I

This section provides an overview of the concepts involved in the preliminary work, a detailed explanation of each concept, the equations to be used throughout the preliminary work and the pseudo-code of the learning process with the SGD optimizer.

#### 3.1.1 Detailed Concepts

**Artificial Neural Network** An Artificial Neural Network(ANN) is defined as a massively distributed and parallelized processor made up of small computational units called Neurons that acquires knowledge from its environment

through a learning process and stores this knowledge as synaptic weights between neurons [4].

A Neuron takes external data or data from other neurons as input. Each input is multiplied by its respective synaptic weight and then all of these are added up and passed through a function called an activation function to produce a single value called the activation. This activation acts as input to the next layers of Neurons until the Network produces an output.

Neurons are arranged in layers:

- An input layer
- An output layer
- Zero or more hidden layers. So called as the inputs and outputs to and from the neurons in these layers are typically unobserved.

In a feed-forward network, output is passed exclusively to the next layer of neurons. The strength of the connection between two neurons is determined by their synaptic weight. Each neuron in a layer is connected to every neuron in the previous layer and every neuron in the next layer. These values of all the synaptic weights can easily be stored in matrix form called a Weight Matrix. Each layer of a network has its own respective weight matrix.

The learning process is typically modeled by the backpropagation algorithm combined with an update rule. The backpropagation algorithm is responsible for assigning blame for error in the network's output to the neurons of the network. Once this blame has been assigned, the update rule i.e The Optimizer updates the synaptic weights of each neuron in an attempt to minimize the error function.

**Activation Functions** Activations functions take in the weighted sum of inputs to a neuron and produce the output of the Neuron. These activation functions are typically non-linear.

The hidden layers of the Network used in experimentation made use of the ReLU activation function and the output layer makes use of the Softmax activation function.

The ReLU activation function is given by the following equation:

$$g(z) = \begin{cases} z & z > 0 \\ 0 & elsewhere \end{cases} \quad (1)$$

The Softmax activation function is given by the following equation:

$$\sigma(z)_j = \frac{e^{z_j}}{\sum_{k=1}^K e^{z_k}} \quad \text{for } j = 1, \dots, K \quad (2)$$

Where  $K$  is the number of classes in the dataset.

**Loss Function** A function that describes the amount of error in an ANN. This is the function which is minimized by the Optimizer during the Training Process.

The loss function used in the Network used for this experimentation is the categorical cross-entropy function which is given by:

$$L(\hat{\mathbf{y}}, \mathbf{y}) = -\frac{1}{N} \sum_{i=1}^N [y_i \log \hat{y}_i + (1 - y_i) \log (1 - \hat{y}_i)] \quad (3)$$

Where  $\hat{\mathbf{y}}$  is the set of  $N$  guessed class labels and  $\mathbf{y}$  is the set of  $N$  actual class labels.

**Gradient Descent** Gradient Descent is classified as a first-order iterative optimization algorithm. It works by exploiting the principle that given a multivariable function  $J(\theta)$ , that is differentiable at a point  $\underline{a}$ , then  $J(\theta)$  decreases fastest along the direction of the negative gradient of  $J(\theta)$  at  $\underline{a}$  i.e  $-\nabla J(\underline{a})$

Given a weight matrix  $\underline{\Theta}$ , the gradient descent update rule for the weight  $\theta_{ji}$  is given by:

$$\theta_{ji} \leftarrow \theta_{ji} + \Delta\theta_{ji} \quad (4)$$

where the weight update,  $\Delta\theta_{ji}$ , is given by:

$$\Delta\theta_{ji} := -\alpha \frac{\partial J(\theta)}{\partial \theta_{ji}} \quad (5)$$

$\alpha$  is the learning rate and  $\frac{\partial J(\theta)}{\partial \theta_{ji}}$  is the gradient of the error function with respect to  $\theta_{ji}$ .  $\frac{\partial J(\theta)}{\partial \theta_{ji}}$  is calculated using the backpropagation algorithm [10].

**Learning Rate** The Learning Rate of an ANN can be defined as the rate at which parameters change in a network. That is, given a new update on  $\theta_{ji}$ , the proportion of  $-\frac{\partial J(\theta)}{\partial \theta_{ji}}$  that is added to the current  $\theta_{ji}$  is given by,  $\alpha$ , our learning rate.

The value of the Learning Rate is crucial to the convergence of the network. Too low of a learning rate and the time to convergence becomes exceedingly high. Too high of a learning rate and the network may fail to converge due to continually stepping over the minimum. The Learning Rate can be global to all weights in the network or local to individual or groups of weights. Additionally, the learning rate can be static for the entire learning process or can be adaptive throughout the learning process.

**Momentum** The Momentum Method is an acceleration method [10] used to overcome sudden changes in gradient, that are common around minima, and reduce the time to convergence by setting the next weight update equal to a linear combination of:

1.  $\frac{\partial J(\theta)}{\partial \theta_{ji}}$  the gradient of the error function with respect to  $\theta_{ji}$
2.  $\Delta\theta_{ji}$  the previous weight update.

$$\Delta\theta_{ji_{t+1}} := -\alpha \frac{\partial J(\theta)}{\partial \theta_{ji}} + \beta \Delta\theta_{ji} \quad (6)$$

Where  $\beta < 1$  is the momentum, or exponential decay, constant

Doing this improves the ability of the optimizer to avoid local minima by allowing it to follow the overall trend in gradient rather than the instantaneous gradient. This also speeds up convergence as once the trend in the gradient is found, the optimizer accelerates along this trend towards the minimum.



### 3.1.2 Dataset

The Dataset the ANN will be trained on is the MNIST training set. It is a dataset of 60000 28x28 pixel images of handwritten decimal digits from 0 to 9. In total there are 10 distinct classes of images. The classes are evenly distributed thus the dataset is considered balanced.

The Dataset the ANN will be tested on is the MNIST test set. It is similar to the above training set however there are only 10000 images.

### 3.1.3 Architecture

**Input Layer** The Architecture of our ANN will be a Multilayer Feed Forward Network with 2 hidden layers. It has been proven that a Neural Network with two hidden layers can approximate any bounded continuous function to arbitrary accuracy [5]. The data takes the form of 28x28 pixel images. With each pixel being an individual input we have,  $N_i$ , the number of inputs equal to 784.

**Hidden Layers** Due to the fact that all Optimizers will be run on the same network architecture the number of neurons in each hidden layer will be determined beforehand rather than via cross-validation, this prevents the architecture from being tailored to any one optimizer as well as controlling a variable in the experiment.

Panchal and Panchal [8] suggest a simple approach to selecting the number of neurons in a network by providing some rules of thumb. Specifically,

1.  $N_h \in [N_o, N_i]$
2.  $N_h < 2 \times N_i$
3.  $N_h = \frac{2}{3}N_i + N_o$

Following these guidelines as well as the results in [1] the following architecture has been proposed:

$$N_{h1} = N_{h2} = 300 \tag{7}$$

so

$$N_h = N_{h1} + N_{h2} = 600 \tag{8}$$

### 3.1.4 Performance Metrics

This section describes the manner in which the various Optimizers are compared and evaluated.

During training on a given Optimizer, the Training Accuracy and Loss were sampled after every 100 training examples. The Test Accuracy and Loss were sampled every 6000 examples. These four metrics were all averaged over 10 training process instances.

The training metrics serve as a temporal plot of the network learning speed while the test metrics serve as a temporal plot of the generalization of the network.

Stochastic Gradient Descent serves as our control for comparison for the other Optimizers.

### 3.1.5 Learning Process Algorithms

This section provides the pseudocoded algorithms for the learning process with the SGD optimizer, the SGD optimizer with momentum. The pseudocoded algorithms for the adaptive learning rate methods can be found in Methodology-II.

**Learning Process With SGD** The algorithm below shows the full learning process involving the backpropagation algorithm and the SGD optimizer.

---

```

Input: Training Symbols  $\{(x_0, y_0), \dots, (x_n, y_n)\}$ 
Output: Weight Matrices  $\Theta$ 
1 Initialize all weights in all weight matrices  $\Theta^{(l)}$  to small non-zero
  gaussian random numbers.
2 Initialize all update matrices  $\Delta\theta^{(l)}$  to zero matrices.
3 foreach  $(\underline{x}_i, y_i)$  in Symbols do
4   Set  $\underline{a}^{(1)} = \underline{x}_i$ ;
5   foreach Layer  $l \in \{1, \dots, L-1\}$  do
6     Set  $\underline{z}^{(l+1)} = \Theta^{(l)} \underline{a}^{(l)}$ 
7     Set  $\underline{a}^{(l+1)} = g(\underline{z}^{(l+1)})$ 
8   end
9   Set  $\underline{\delta}^{(L)} = \underline{a}^{(L)} - y_i$ 
10  foreach Layer  $l \in \{L-1, \dots, 2\}$  do
11    Set  $\underline{\delta}^{(l)} = (\Theta^{(l)}) \underline{\delta}^{(l+1)} \times g'(\underline{z}^{(l)})$ 
12  end
13  foreach Layer  $l \in \{1, \dots, L-1\}$  do
14    Set  $\Delta\theta^{(l)} = -\alpha[\underline{\delta}^{(l+1)} \underline{a}^{(l)T}]$ 
15    Set  $\Theta^{(l)} = \Theta^{(l)} + \Delta\theta^{(l)}$ 
16  end
17 end
18 return  $\Theta$ 

```

---

**SGD with Momentum** The algorithm shall be modified by adding a proportion,  $\beta$ , of the previous update matrix i.e  $\Delta\theta_i^{(l)}$  to the latest update matrix.

---

**Input:** Training Symbols  $\{(x_0, y_0), \dots, (x_n, y_n)\}$   
**Output:** Weight Matrices  $\Theta$

- 1 Initialize all weights in all weight matrices  $\Theta^{(l)}$  to small non-zero gaussian random numbers.
- 2 Initialize all update matrices  $\Delta\theta_0^{(l)}$  to zero matrices.
- 3 **foreach**  $(x_i, y_i)$  in Symbols **do**
- 4     Set  $\underline{a}^{(1)} = \underline{x}_i$ ;
- 5     **foreach** Layer  $l \in \{1, \dots, L-1\}$  **do**
- 6         Set  $\underline{z}^{(l+1)} = \Theta^{(l)} \underline{a}^{(l)}$
- 7         Set  $\underline{a}^{(l+1)} = g(\underline{z}^{(l+1)})$
- 8     **end**
- 9     Set  $\underline{\delta}^{(L)} = \underline{a}^{(L)} - y_i$
- 10    **foreach** Layer  $l \in \{L-1, \dots, 2\}$  **do**
- 11       Set  $\underline{\delta}^{(l)} = (\Theta^{(l)}) \underline{\delta}^{(l+1)} \times g'(\underline{z}^{(l)})$
- 12    **end**
- 13    **foreach** Layer  $l \in \{1, \dots, L-1\}$  **do**
- 14       Set  $\Delta\theta_{i+1}^{(l)} = -\alpha[\underline{\delta}^{(l+1)} \underline{a}^{(l)T}] + \beta \Delta\theta_i^{(l)}$
- 15       Set  $\Theta^{(l)} = \Theta^{(l)} + \Delta\theta_{i+1}^{(l)}$
- 16    **end**
- 17 **end**
- 18 **return**  $\Theta$

---

### 3.1.6 Learning rate schedules

This section discusses the Search-Then-Converge and Cyclical Learning Rate schedules and provides the governing equations for said schedules.

**Search-Then-Converge** This strategy involves decreasing the learning rate as time passes. The learning rate is given by the following equation:

$$\alpha = \frac{\alpha_0}{1 + \frac{i}{T}} \quad (9)$$

Where  $i$  is the index of the current training symbol,  $\alpha_0$  and  $T$  are selected hyperparameters. For our experimentation,  $\alpha_0 = 0.01$ . To Determine a suitable value for  $T$ , the training process was run with various values of  $T$  the validation accuracy was evaluated at the end of each training process. The  $T$  that produced the largest accuracy was selected. This was 225000

**Cyclical Learning Rates** There are 3 hyperparameters that need selecting for this method:

- Stepsize - The number of iterations taken in a half-cycle of the periodic function.
- $\alpha_{min}$  - The infimum of our periodic function.
- $\alpha_{max}$  - The supremum of our periodic function.

It is recommended that the stepsize ranges from 2 to 10 times the number of data points in your training set. It is noted that there is no significant difference in performance along this range. Thus in our case:

$$stepsize = 2 \times |Trainingset| = 120000 \quad (10)$$

The learning rate boundaries are estimated with a test. The test is to run the algorithm for a number of epochs with linearly increasing learning rates. The learning rate versus validation accuracy graph is plotted.  $\alpha_{min}$  is the learning rate at which the accuracy starts to increase and  $\alpha_{max}$  is the point at which the validation accuracy starts to become jagged or decrease.

The learning rate, using a triangular periodic function, is given by the following equation:

$$\alpha = \alpha_{min} + (\alpha_{max} - \alpha_{min}) \times \text{Max}\{0, (1 - x)\} \quad (11)$$

where

$$x = \left| \frac{iteration}{stepsize} - 2 \times (CycleLength) + 1 \right| \quad (12)$$

and

$$CycleLength = \left\lfloor 1 + \frac{iteration}{2 \times stepsize} \right\rfloor \quad (13)$$

Using the hyperparameter selection method described by Smith [12],  $\alpha_{min} = 0.02$  and  $\alpha_{max} = 0.12$

## 3.2 Methodology-II

### 3.2.1 Adaptive Learning Rate Optimizers

This section proposes various vectorized update rules for the ADAGRAD, RMSProp and Adam Optimizers.

**Adaptive Gradient Method** The vectorized update rule for ADAGRAD is given by:

$$\Delta\theta_{t+1} = -\alpha\mathbf{g}_{t+1} \oslash [\epsilon\mathbf{I} + \mathbf{G}_{t+1}]^{\circ\frac{1}{2}} \quad (14)$$

where

- $\mathbf{g}_t$  is the gradient matrix  $\underline{\delta}^{(l+1)}\underline{a}^{(l)T}$
- $\mathbf{G}_{t+1}$ , termed the normalization matrix is the sum of squares of gradients before time  $t+1$  and is iteratively defined as:

$$\mathbf{G}_{t+1} = \mathbf{G}_t + \mathbf{g}_t^{\circ 2} \quad (15)$$

- $\epsilon\mathbf{I}$  is a smoothing matrix where all elements are a small number  $\epsilon$  to prevent division by zero.
- $\circ$  is the Hadamard power operator

**RMSProp** The main problem with ADAGRAD is that the elements of the normalization matrix are monotonically increasing thus the learning rate is monotonically decreasing at each iteration. The vectorized update rule is identical to that of ADAGRAD except in this case  $\mathbf{G}_t$  is now defined as matrix of the the running average of past squared gradients.

$$\mathbf{G}_{t+1} = \frac{1}{t+1}[t\mathbf{G}_t + \mathbf{g}_t^{\circ 2}] \quad (16)$$

**Adam** Adam combines the benefits of keeping a running average of past squared gradients, the variance, as well as a running average of past gradients, the mean. The first moment term acts as a momentum term as it is a direction preserving exponentially decaying average of previous gradients.

The vectorized update rule for Adam is given by:

$$\Delta\theta_{t+1} = -\alpha\hat{\mathbf{m}}_{t+1} \oslash [\epsilon\mathbf{I} + \hat{\mathbf{v}}_{t+1}^{\circ\frac{1}{2}}] \quad (17)$$

where  $\hat{\mathbf{m}}_{t+1}$  and  $\hat{\mathbf{v}}_{t+1}$  are the bias corrected first and second moments of the gradients. This bias correction term is introduced as  $\mathbf{m}_0$  and  $\mathbf{v}_0$  are zero matrices and thus the moments are biased towards zero.

$\hat{\mathbf{m}}_{t+1}$  is given by the equation:

$$\hat{\mathbf{m}}_{t+1} = \frac{1}{1 - \beta_1^{t+1}} \mathbf{m}_{t+1} \quad (18)$$

$\hat{\mathbf{v}}_{t+1}$  is given by the equation:

$$\hat{\mathbf{v}}_{t+1} = \frac{1}{1 - \beta_2^{t+1}} \mathbf{v}_{t+1} \quad (19)$$

where,

$$\mathbf{m}_{t+1} = \beta_1 \mathbf{m}_t + (1 - \beta_1) \mathbf{g}_{t+1} \quad (20)$$

and,

$$\mathbf{v}_{t+1} = \beta_2 \mathbf{v}_t + (1 - \beta_2) \mathbf{g}_{t+1}^{\circ 2} \quad (21)$$

### 3.2.2 Contrast with previous work

The current work differs from previous work mainly due to the aggregation of various Optimizers that have not been compared in a single source. Additionally, the comparison metrics used in this work extend the comparisons made in previous research.

## 4 Analytic Results to be used

This Section shall provide derivations for the newly vectorized update rules for the ADAGRAD, RMSPROP and Adam Optimizers. It will be shown that given the  $M \times N$  update matrix, each element,  $\theta_{i,j}$ , in the matrix is the single parameter update rule for that optimizer.

### 4.1 ADAGRAD and RMSProp

#### 4.1.1 Derivation

Given the vectorized update rule:

$$\begin{aligned} \Delta \theta &= -\alpha \mathbf{g} \oslash [\epsilon \mathbf{I} + \mathbf{G}]^{\circ \frac{1}{2}} \\ \Leftrightarrow \Delta \theta &= -\alpha \begin{bmatrix} g_{11} & \dots & g_{1n} \\ \vdots & \ddots & \vdots \\ g_{m1} & \dots & g_{mn} \end{bmatrix} \oslash \left( \begin{bmatrix} \epsilon & \dots & \epsilon \\ \vdots & \ddots & \vdots \\ \epsilon & \dots & \epsilon \end{bmatrix} + \begin{bmatrix} G_{11} & \dots & G_{1n} \\ \vdots & \ddots & \vdots \\ G_{m1} & \dots & G_{mn} \end{bmatrix} \right)^{\circ \frac{1}{2}} \\ \Leftrightarrow \Delta \theta &= -\alpha \begin{bmatrix} g_{11} & \dots & g_{1n} \\ \vdots & \ddots & \vdots \\ g_{m1} & \dots & g_{mn} \end{bmatrix} \oslash \begin{bmatrix} \sqrt{\epsilon + G_{11}} & \dots & \sqrt{\epsilon + G_{1n}} \\ \vdots & \ddots & \vdots \\ \sqrt{\epsilon + G_{m1}} & \dots & \sqrt{\epsilon + G_{mn}} \end{bmatrix} \end{aligned}$$

$$\iff \Delta\theta = \begin{bmatrix} -\alpha \frac{g_{11}}{\sqrt{\epsilon + G_{11}}} & \dots & -\alpha \frac{g_{1n}}{\sqrt{\epsilon + G_{1n}}} \\ \vdots & \ddots & \vdots \\ -\alpha \frac{g_{m1}}{\sqrt{\epsilon + G_{m1}}} & \dots & -\alpha \frac{g_{mn}}{\sqrt{\epsilon + G_{mn}}} \end{bmatrix}$$

$$\text{Thus } \Delta\theta_{ij} = -\alpha \frac{g_{ij}}{\sqrt{\epsilon + G_{ij}}}$$

#### 4.1.2 Analysis

The benefit of using this vectorized equation is that all operations are element-wise algebraic operations. As such, the equation is simple to understand for the reader and programmatic implementation is also simple. The presence of an interchangeable normalization matrix,  $\mathbf{G}_{t+1}$ , allows various other normalizations (other than Sum of Squared Gradients and the RMS of Gradients) to be easily implemented without having to derive a new vectorized equation.

### 4.2 Adam

#### 4.2.1 Derivation

Given the vectorized update rule:

$$\Delta\theta = -\alpha \hat{\mathbf{m}} \oslash [\epsilon \mathbf{I} + \hat{\mathbf{v}}^{\circ \frac{1}{2}}]$$

$$\iff \Delta\theta = -\alpha \begin{bmatrix} \hat{m}_{11} & \dots & \hat{m}_{1n} \\ \vdots & \ddots & \vdots \\ \hat{m}_{m1} & \dots & \hat{m}_{mn} \end{bmatrix} \oslash \left( \begin{bmatrix} \epsilon & \dots & \epsilon \\ \vdots & \ddots & \vdots \\ \epsilon & \dots & \epsilon \end{bmatrix} + \begin{bmatrix} \hat{v}_{11} & \dots & \hat{v}_{1n} \\ \vdots & \ddots & \vdots \\ \hat{v}_{m1} & \dots & \hat{v}_{mn} \end{bmatrix}^{\circ \frac{1}{2}} \right)$$

$$\iff \Delta\theta = -\alpha \begin{bmatrix} \hat{m}_{11} & \dots & \hat{m}_{1n} \\ \vdots & \ddots & \vdots \\ \hat{m}_{m1} & \dots & \hat{m}_{mn} \end{bmatrix} \oslash \begin{bmatrix} \epsilon + \sqrt{\hat{v}_{11}} & \dots & \epsilon + \sqrt{\hat{v}_{1n}} \\ \vdots & \ddots & \vdots \\ \epsilon + \sqrt{\hat{v}_{m1}} & \dots & \epsilon + \sqrt{\hat{v}_{mn}} \end{bmatrix}$$

$$\iff \Delta\theta = \begin{bmatrix} -\alpha \frac{\hat{m}_{11}}{\epsilon + \sqrt{\hat{v}_{11}}} & \dots & -\alpha \frac{\hat{m}_{1n}}{\epsilon + \sqrt{\hat{v}_{1n}}} \\ \vdots & \ddots & \vdots \\ -\alpha \frac{\hat{m}_{m1}}{\epsilon + \sqrt{\hat{v}_{m1}}} & \dots & -\alpha \frac{\hat{m}_{mn}}{\epsilon + \sqrt{\hat{v}_{mn}}} \end{bmatrix}$$

$$\text{Thus } \Delta\theta_{ij} = -\alpha \frac{\hat{m}_{ij}}{\epsilon + \sqrt{\hat{v}_{ij}}}$$

#### 4.2.2 Analysis

Given the fact that  $\epsilon < 1$  and that it is no longer square rooted. It can be deduced that the magnitude of smoothing used in this method is less than that of the other Adaptive Gradient Methods in this work. This is possibly due to the fact that, there is already a bias correction to prevent the  $\hat{v}_{ij}$  term from being zero. So having the usual magnitude of smoothing may further bias the term away from zero, which is undesirable.

## 5 Experimental Setup and Computational Model

This section will describe various components of the experimental setup such that the experiment can be repeated and the validity of claims made in future components of this work can be tested.

### 5.1 Experimental Setup

The Keras Neural Network API running on a Tensorflow Backend was used during experimentation.

The default SGD, Adagrad, RMSProp and Adam Optimizers available in Keras were used. Additionally, custom CLR and STC Optimizers were implemented. To implement these, custom Keras callbacks were created to set the correct learning rates at every training batch. An additional callback was created to do the sampling of the performance metrics during each Optimizer's training process.

The training process consisted of a single epoch, and each training process was repeated ten times to average the results of each Optimizer.

Once all training processes were completed and performance metrics averaged, the performance metrics were plotted and the test metrics were tabulated.

### 5.2 Computational Environment

#### 5.2.1 Hardware Specification

- CPU: Intel(R) Core(TM) i5-3570k CPU @ 3.4-3.8 GHz
- GPU: AMD Radeon HD 5700 Series
- RAM: 4GB

### 5.3 Computational Model

- Artificial Neural Network

#### 5.3.1 Software and Versions

- Operating System: Windows 10
- Python v3.6.0
- Keras v2.2.4
- Tensorflow v1.11.0
- numpy v1.14.3



## 5.4 Data Sources

The data is available in the Keras package and can be retrieved by importing *keras.datasets.mnist* and calling *keras.datasets.mnist.load\_data()*. The training and test images consist of 28x28 pixel black and white handwritten letters and digits.

## 5.5 Software Engineering practices and Public Accessibility

The code will follow Object Oriented Design with a focus on modularity to aid in debugging and review of the code by the evaluator of this work.

The full source code shall be available at the [Github repository](#) for this work. This repository is publicly accessible however changes to the repository are restricted to the author(s) of this work.

# 6 Results

This section presents and analyses the results of experimentation.

## 6.1 Figures

### 6.2 Training Accuracy versus Iteration

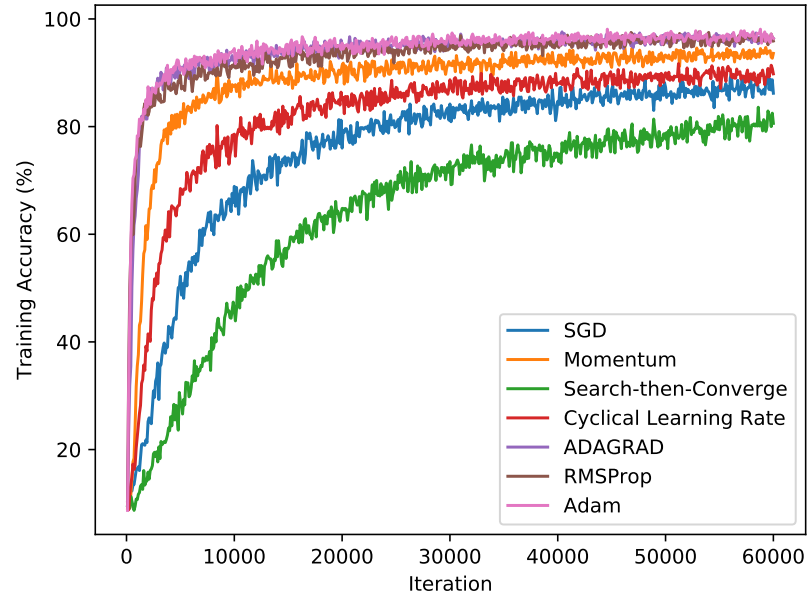


Figure 1: Training Accuracy versus Iteration

### 6.3 Training Loss versus Iteration

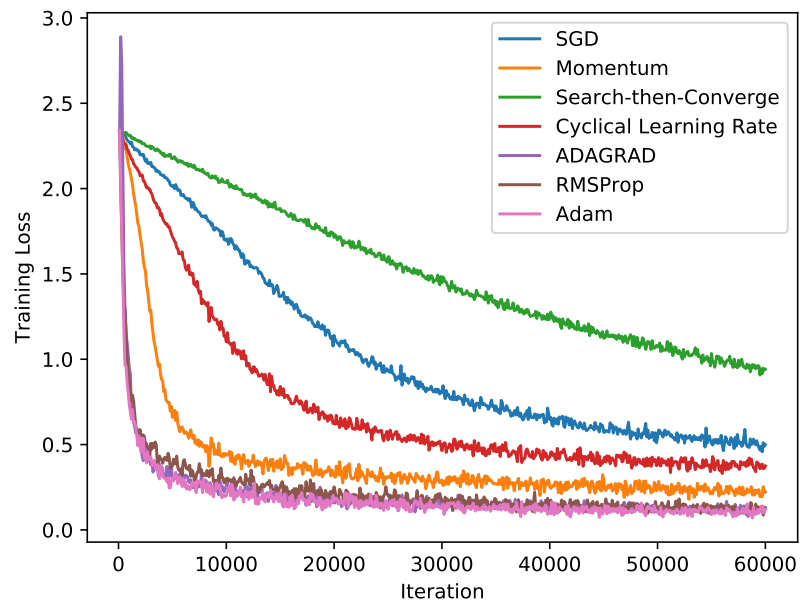


Figure 2: Training Loss versus Iteration

## 6.4 Test Accuracy versus Iteration

### 6.4.1 Table

Optimizer	Iteration									
	6000	12000	18000	24000	30000	36000	42000	48000	54000	60000
SGD	58.3	72.8	78.3	81.8	83.9	85.3	86.2	87.0	87.6	88.1
Momentum	84.7	89.2	90.1	91.4	92.0	92.5	92.8	93.0	93.6	93.8
Search-then-Converge	33.4	53.2	64.4	70.6	74.0	76.7	78.7	80.1	81.2	82.2
Cyclical Learning Rate	71.6	80.9	84.7	86.7	87.8	88.6	89.2	89.7	90.0	90.4
Adagrad	91.4	93.7	94.4	95.2	95.6	95.9	96.1	96.4	96.6	96.6
RMSProp	88.8	92.4	93.5	94.7	94.9	95.5	95.9	96.1	96.4	96.5
Adam	92.1	94.1	94.8	95.5	96.0	96.4	96.6	96.6	96.7	96.9

Figure 3: Test Accuracy versus Iteration Matrix

### 6.4.2 Plot

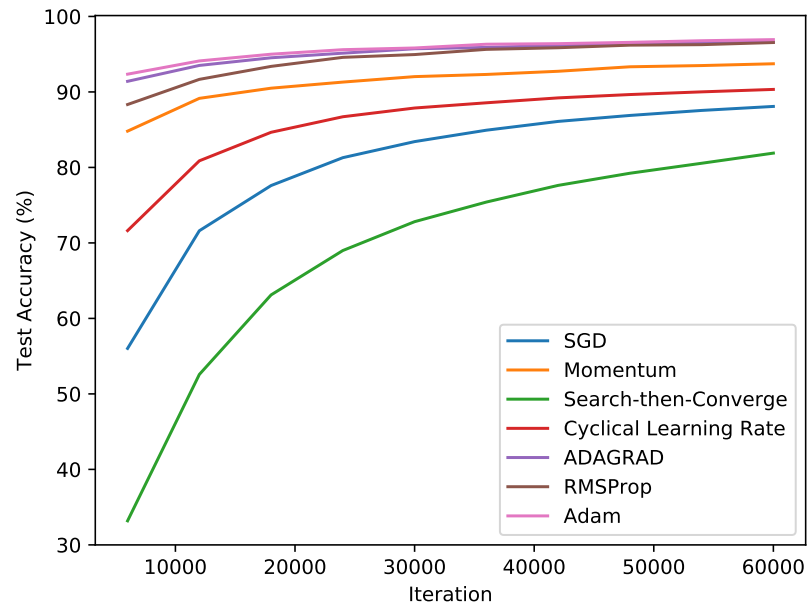


Figure 4: Test Accuracy versus Iteration

## 6.5 Test Loss versus Iteration

### 6.5.1 Table

Optimizer	Iteration									
	6000	12000	18000	24000	30000	36000	42000	48000	54000	60000
SGD	1.94	1.55	1.18	0.92	0.77	0.66	0.59	0.53	0.50	0.47
Momentum	0.55	0.38	0.34	0.30	0.28	0.26	0.25	0.24	0.22	0.21
Search-then-Converge	2.13	1.94	1.75	1.57	1.41	1.27	1.15	1.05	0.96	0.89
Cyclical Learning Rate	1.60	0.96	0.68	0.55	0.48	0.44	0.41	0.38	0.37	0.35
Adagrad	0.28	0.21	0.18	0.15	0.14	0.13	0.13	0.12	0.11	0.11
RMSProp	0.37	0.25	0.21	0.17	0.16	0.14	0.14	0.12	0.11	0.11
Adam	0.26	0.19	0.16	0.14	0.13	0.12	0.11	0.11	0.10	0.09

Figure 5: Test Loss versus Iteration Matrix

### 6.5.2 Plot

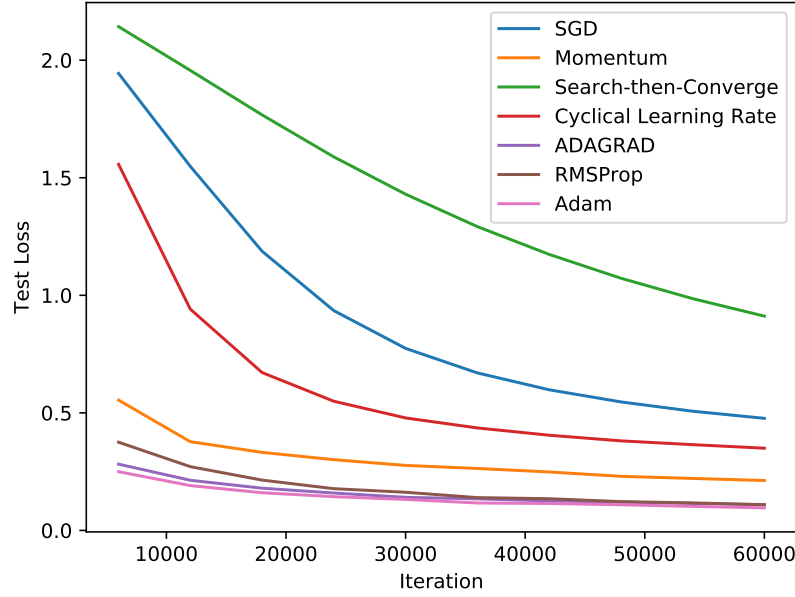


Figure 6: Test Loss versus Iteration

## 6.6 Analysis

### 6.6.1 SGD

As the control for our comparison between the optimizers

## 7 Summary

Choosing an optimizer for ones problem domain and constraints takes a lot of time seeing as there is no single source of comparison for popular optimizers. This work and future work is intended to wholly and fairly compare the time and generalization performance of various popular gradient based optimizers. This shall provide the reader with an initial source of information to improve the ability of the reader to make an informed choice of optimizer. The results of this comparison are applicable across many problem domains due to the level of generality maintained during experimentation.

## 8 References

### References

- [1] Dan Claudiu Ciresan, Ueli Meier, Luca Maria Gambardella, and Jürgen Schmidhuber. Deep big simple neural nets excel on handwritten digit recognition. *CoRR*, abs/1003.0358, 2010.
- [2] Christian Darken and John E Moody. Note on learning rate schedules for stochastic optimization. In *Advances in neural information processing systems*, pages 832–838, 1991.
- [3] John Duchi, Elad Hazan, and Yoram Singer. Adaptive subgradient methods for online learning and stochastic optimization. *Journal of Machine Learning Research*, 12(Jul):2121–2159, 2011.
- [4] S. Haykin. Neural networks: A comprehensive foundation. *find just now*, 1:24–24, 1999.
- [5] Kurt Hornik, Maxwell B. Stinchcombe, and Halbert White. Multilayer feedforward networks are universal approximators. *Neural Networks*, 2: 359–366, 1989.
- [6] Diederik P Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.
- [7] Yann LeCun, Léon Bottou, Yoshua Bengio, and Patrick Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, 1998.
- [8] G Panchal and Mahesh Panchal. Review on methods of selecting number of hidden nodes in artificial neural network. *International Journal of Computer Science and Mobile Computing*, 3(11):455–464, 2014.
- [9] Sebastian Ruder. An overview of gradient descent optimization algorithms. *arXiv preprint arXiv:1609.04747*, 2016.
- [10] David E Rumelhart, Geoffrey E Hinton, and Ronald J Williams. Learning representations by back-propagating errors. *nature*, 323(6088):533–536, 1986.
- [11] Patrice Y. Simard, David Steinkraus, and John C. Platt. Best practices for convolutional neural networks applied to visual document analysis. In *ICDAR*, 2003.
- [12] Leslie N Smith. Cyclical learning rates for training neural networks. In *Applications of Computer Vision (WACV), 2017 IEEE Winter Conference on*, pages 464–472. IEEE, 2017.



## 9 Appendices

### 9.1 Literature Survey

The following literature review seeks to critically evaluate the current evaluations of various prolific optimizers used in Artificial Neural Networks. In particular, the benefits of each optimizer in comparison to others as well as the discussed trade-offs of the optimizer. The time to convergence as well as level of generalization will be the comparison metrics discussed.

Kingma and Ba [6] provides an unbiased empirical evaluation of the Adam Optimizer with respect to other optimizers. Adam can be described as a combination of RMSProp, Momentum and a bias-correction term. Multiple statistics are used for the comparison and shows that Adam is faster to converge than all the other tested optimizers, however it is admitted that Adam tends to over-fit to the training data so regularization and dropout strategies were used to improve generalization. For a Multi-layer Neural Network with dropout stochastic regularization, the training cost versus iteration number is reported for multiple Optimizers. However, the generalization of these optimizers, along our chosen metrics is not reported.

Ruder [9] provides an overview of various gradient based optimizers and their specific limitations. However the overview is wholly theoretical and does not report any empirical statistics in its analysis. Provided derivations of each optimizer have been used in the current work for the development of the algorithms used in each A visualization of some of the discussed optimizers is provided but this does not specifically pertain to the domain of our problem as the visualized objective functions are the Beale Function and a hyperparaboloid. It is important to note the poor performance around saddle points of the optimizers with static learning rates.

Darken and Moody [2] discusses learning rate schedules as a solution to a static learning rate as well as introduces the Search-Then-Converge (STC) Method. A major limitation to learning rate schedules discussed is that the learning rate decreases too quickly so convergence is not guaranteed. STC attempts to solve this problem by slowing down this reduction in learning rate and then increasing the reduction once a certain time threshold has been reached. This gives the optimizer time to *search* for a local minimum and then *converge*, hopefully, once it is found. The paper describes that this method is *guaranteed* to converge, however there is no proven guarantee that the optimizer will find a local minimum within the "search" period. This bias warrants that further investigation be reported in the current work to validate the claims of this paper.

Duchi et al. [3] proposes the Adaptive Subgradient Method ,or ADAGRAD. The performance of this method is gauged along two metrics. The online error as

well as the test set performance of the resultant classifier after one learning epoch. Various other adaptive gradient methods are tested for comparison and it is shown that ADAGRAD is the best optimizer to use when the training data is sparse. However, for small datasets, ADAGRAD is beat by the AROW algorithm. It is also shown that, on MNIST classification, the error rate for ADAGRAD is 0.4% greater than that of the Passive-Aggressive Algorithm. The version of ADAGRAD tested in this paper involves the use of Regularised Dual Averaging and thus differs from the ADAGRAD version to be tested in the current work.

Smith [12] proposes the use of cyclical learning rates as opposed to adaptive learning rates. Cyclical learning rates (CLR) are shown to have less computational cost than other adaptive methods. The accuracy at 25000 and 70000 training iterations is used as a comparison metric for the different optimizers tested. The method involves sampling a periodic function which is bounded by a maximum and minimum learning rate. This prevents the learning rate from monotonically decreasing as in ADAGRAD or other learning rate scheduling techniques. This improves the performance of the optimizer around saddle points. Selecting a minimum and maximum learning rate as well as the cycle length of the periodic function is covered by a selection of tests contained within the paper.

An interesting result in the paper shows that when using CLR with a gradient acceleration method, The Nesterov Method, it took only 25000 training iterations to achieve accuracy similar to the regular Nesterov Method after 70000 iterations. Similarly, RMSProp with CLR achieves the same improvement. However, when CLR is used in conjunction with the Adam Optimizer, the performance is worse than the regular Adam Optimizer. This suggests some conflict with CLR and either the first-moment term, the bias correction or both in the Adam Optimizer.

Simard et al. [11] describes the best practices for Convolutional Neural Networks. In particular using elastic deformation to increase the number of training examples is suggested as a standard for CNN training. It is shown that elastic deformation produces better generalization of the network when compared to affine deformation which in turn is better than no deformation. These methods may potentially be used if the network overfits the training data. This is a computationally expensive method to prevent overfitting when compared to regularization or dropout methods. However, the distortion can be precomputed and reused for various experiments which could potentially save computation time across all experiments in this work. Although this article describes best practices for CNN's this best practice applies to our domain as the dataset is the same.

LeCun et al. [7] compares various types of classifiers on the MNIST database. A MLP with one hidden layer achieved a 4.7% test error with 300 hidden units and 4.5% for a network with 1000 hidden units. This marginal improvement for a large increase in hidden units suggests that the network with 1000 units was overfit i.e poorly generalized, thus the increase in units past a certain threshold between 300 and 1000 increases test error. Alternatively, the increase in number of hidden units could follow a law of diminishing returns and thus while adding units does not increase test error, doing so may not strictly decrease test error. Irrespective of which case holds, it can be seen that arbitrarily adding hidden units does not significantly improve test error and thus other guidelines such as computational time should be considered when deciding on network architecture. A similar pattern holds for a network with two hidden layers.

Ciresan et al. [1] evaluate the effect of the number of layers and number of neurons per layer in a Multilayer Perceptron(MLP) Artificial Neural Network when classifying Handwritten Digits. Training examples were distorted using elastic and affine deformations. This distortion was used to artificially increase the number of training examples by distorting all training images in each training epoch. GPU acceleration was used to reduce the training time for each network. The performance of each architecture was measured primarily via the minimum test error achieved at each training epoch. While trialling various architectures the best minimum test error was achieved by a network with 5 hidden layers with 2500,2000,1500,1000,500,10-architecture at 0.32% error. However the training time required to achieve this result was 114.5 hours. A two layer MLP with 1000,500,10-architecture achieved 0.44% error on the test set while only taking 23.4 hours of training time.

## 9.2 Tables

### 9.2.1 Iteration versus Accuracy

Optimizer	Iteration Number									
	10000	20000	30000	40000	50000	60000	70000	80000	90000	100000
SGD	30	50	70	80	85	90	92	95	96	99
Momentum	40	60	80	85	88	90	99	99	99	99

Table 1: Mock Table of test accuracy of an Optimizer versus Iteration Number