2-23-2021

An Angle-based Stochastic Gradient Descent Method for Machine Learning: Principle and Application

Chongya Song
Florida International University, ysong024@fiu.edu

FLORIDA INTERNATIONAL UNIVERSITY

Miami, Florida

AN ANGLE-BASED STOCHASTIC GRADIENT DESCENT METHOD FOR MACHINE LEARNING: PRINCIPLE AND APPLICATION

A dissertation submitted in partial fulfillment of

the requirements for the degree of

DOCTOR OF PHILOSOPHY

in

ELECTRICAL AND COMPUTER ENGINEERING

by

Chongya Song

2021

To: Dean John L. Volakis
College of Engineering and Computing

This dissertation, written by Chongya Song, and entitled An Angle-based Stochastic Gradient Descent Method for Machine Learning: Principle and Application, having been approved in respect to style and intellectual content, is referred to you for judgment.

We have read this dissertation and recommend that it be approved.

_____
Jean Andrian

_____
Nezih Pala

_____
Deng Pan

_____
Alexander Pons

_____
Kang Yen, Major Professor

Date of Defense: February 23, 2021

The dissertation of Chongya Song is approved.

_____
Dean John L. Volakis
College of Engineering and Computing

_____
Andrés G. Gil
Vice President for Research and Economic Development
and Dean of the University Graduate School

Florida International University, 2021

DEDICATION

I dedicate this dissertation to my parents. Without their patience, understanding, support, and most of all love, the completion of this work would not have been possible.

ACKNOWLEDGMENTS

I wish to thank the members of my committee for their support and patience. Also, I would like to thank my co-major professor Dr. Kang Yen in improving my critical thinking and writing skills. Finally, I would like to particularly appreciate my co-major professor Dr. Alexander Pons who guided me into my current research field. All my published papers and issued/filed patents are under his gentle but firm directions.

ABSTRACT OF THE DISSERTATION

AN ANGLE-BASED STOCHASTIC GRADIENT DESCENT METHOD

FOR MACHINE LEARNING: PRINCIPLE AND APPLICATION

by

Chongya Song

Florida International University, 2021

Miami, Florida

Professor Kang Yen, Major Professor

In deep learning, optimization algorithms are employed to expedite the resolution to accurate models through the calibrations of the current gradient and the associated learning rate. A major shortcoming of these existing methods is the manner in which the calibration terms are computed, only utilizing the previous gradients during their computations. Because the gradient is a time-sensitive variable computed at a specific moment in time, it is possible that older gradients can introduce significant deviation into the calibration terms. Although most algorithms alleviate this situation by combining the exponential moving average of the previous gradients, we found that this method is not very effective in practice, as it still causes undesirable accumulated impact on the gradients. Another shortcoming is that these existing algorithms lack the ability to incorporate the cost variance during the computation of the new gradient. Therefore, employing the same strategy in reducing the cost under all circumstances is inherently inaccurate. In addition, we identified that some advanced algorithms employ measurements that are confiscatory, resulting in erratic new gradients in practice. With respect to evaluation, we determined that a high error rate is more likely to result from

the weak ability of translating the reduction in the cost to the error rate, a circumstance that has not been addressed in the research to improve the accuracies of new gradients.

In this dissertation, we propose an algorithm that employs the angle between consecutive gradients as a new metric to resolve all the aforementioned problems. The new and nine existing algorithms are implemented into a neural network and a logistic regression classifier for evaluation. The results show that the new method can improve the ability of cost/error rate reduction by 9.40%/11.11% on MNIST dataset and 41.63%/29.58% on NSL-KDD dataset. Also, the aforementioned translating ability of the new method outperforms other optimizers by 33.06%. One of the main contributions of our work is verifying the feasibility and effectiveness of using the angle between consecutive gradients as a reliable metric in generating accurate new gradients. Angle-based measurements could be incorporated into existing algorithms to enhance the cost reduction and translating abilities.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF SYMBOLS

C	cost function
y_{true}	labels
y_{pred}	predictions
W	weights
w	weight
B	biases
b	bias
G	gradient
η	learning rate
$PARAM$	model parameters
PG	previous gradient
CG	current gradient
NG	new gradient
G_{proj}	projected gradient
θ	angle between PG and CG
a	the normalized θ
V_{PG}	flattened PG
V_{CG}	flattened CG
w_{pg}	the weight of PG
w_{cg}	the weight of CG
s_g	the sum of $\lvert w_{pg} \rvert$ and w_{cg} the slope of $F_{pg}(a,s_g)$
s_η	the slope and the minimum of $F_\eta(a,s_\eta)$

$F_{pg}(a,s_g)$	the function to compute w_{pg}
$F_{\eta}(a,s_{\eta})$	the function to calculate η
DC_{all}	largest differences of all costs
DE_{all}	largest differences of all error rates
S_{cost}	score of cost reduction
S_{error}	score of error rate reduction
O	*big O notation*

ABBREVIATIONS AND ACRONYMS

Adadelta	Adaptive Delta
Adagrad	Adaptive Gradient
Adam	Adaptive Moment Estimation
Adamax	Adam With L_∞ Norm
Aggmo	Aggregated Momentum
AG-SGD	Angle-Based SGD
Amsgrad	Adam With Max Gradient
ANN	Artificial Neural Network
BGD	Batch Gradient Descent
BP	Backward Propagation
CDBN	Convolutional Deep Belief Network
CNN	Convolutional Neural Network
CS	Convergence Section
CUDA	Compute Unified Device Architecture
DBN	Deep Belief Network
DNN	Deep Neural Network
EMA	Exponential Moving Average
FP	Forward Propagation
IGT	Implicit Gradient Transport
Mini-BGD	Mini-Batch Gradient Descent
ML	Machine Learning
MSE	Mean Squared Error

Nadam	Adam with NAG
NAG	Nesterov Accelerated Gradient
OPG	Original Previous Gradient
OT	Order Test
PCA	Principle Component Analysis
Qhadam	Quasi-Hyperbolic Adam
RBM	Restricted Boltzmann Machine
Rmsprop	Root Mean Squared Backpropagation
RPG	Reversed Previous Gradient
SGD	Stochastic Gradient Descent
SMA	Simple Moving Average
TR	Translation Rate
Vanilla	Original BP

CHAPTER 1

INTRODUCTION

1.1 Background

In the field of machine learning (ML), there is an increasing number of applications (e.g., computer vision, speech recognition and natural language processing) that are solved using artificial neural networks (ANNs) [1.1]. The convoluted architecture enables an ANN to approximate functions resulting from any data pattern. This advantage largely improves the generalization ability upon regular ML algorithms, especially on large and complex datasets [1.2]. The accuracy of an ANN is determined by, but not limited to, six factors:

1) Model Architecture

The architecture of an ANN can be classified into three distinct types [1.3]: (1) feed-forward, (2) recurrent, and (3) symmetrically connected. Type 1 is composed of one input, one output and one or more hidden layers. The connections among layers are unidirectional from the input to the output layers. Type 2 differs from type 1 in allowing reversed connections (i.e., bidirectional connections). An ANN belongs to type 3 if the two weights associated with each bidirectional connection have the same value. Each type of ANNs has one or more variants and each variant has its unique advantages on specific aspects, such as accuracy, efficiency, and training method. For example, convolutional neural networks (CNNs) are a variant of type 1 ANNs that employ pooling layers. This special layer can extract valuable characteristics from images and make CNNs perform better than other variants on

object detection tasks [1.4]. Deep belief networks (DBNs) can achieve unsupervised learning by separately training each of its restricted Boltzmann machines (RBM) [1.5] in a bottom to top fashion, using the hidden layer as an input layer for the next RBM [1.6]. Also, multiple types of ANNs can be combined to form a hybrid model. For instance, the pooling layer used by CNNs can be combined with DBNs to form convolutional deep belief networks (CDBNs) [1.7].

2) Model Optimization

Each ANN needs to be optimized regardless of its type, but the optimization approach applied to one type of ANN may not be applicable to other types. For example, the wake sleep algorithm [1.8] that is used to fine-tune a DBN is not applicable on ANNs of other types. Furthermore, the optimization methods can be classified into two types based on their usages: (1) unified algorithm (e.g., Momentum [1.9] and Adam [1.10]) and (2) independent technique (e.g., learning rate warm restarts [1.11]). A type 2 method can be applied to a type 1 method for alleviating the shortcomings of the latter. For instance, if a learning rate warm restarts scheduler is applied to Adam, then Adam is capable of periodically resetting its learning rate, resulting in a different cost reduction trajectory and a lower cost. Another key difference between the two types is the number of improved variables: (1) unified algorithm usually improves multiple variables based on specific theories; and (2) independent technique only controls one specific key variable in a precise manner.

3) Method of Gradient Descent

There are three types of gradient descents: (1) batch, (2) mini-batch, and (3) stochastic. Because these methods are different in the number of samples that are used in computing the gradient, each one has its unique advantages in terms of accuracy and efficiency. A comparison among the three methods is provided in Chapter 2 before the review of existing optimization algorithms.

4) Cost Function

Cost functions are used to quantify the output deviations of ANNs during optimization. The quantified values are used in calculating the new gradients, so they indirectly determine the result of optimization. Common cost functions include, but are not limited to, Mean Squared Error (MSE) [1.12], Cross-Entropy [1.13], Huber [1.14], and Cosine-Similarity [1.15]. Because these functions are based on various mathematical theories, they result in different values for the same output and impact the accuracy of models. For instance, when MSE is employed to quantify the cost of an ANN that uses Sigmoid [1.16] as the activation function, a known problem called "learning rate slowdown" (i.e., the weights and the biases of the model stop changing) [1.17] would occur. However, if we replace MSE by Cross-Entropy, the aforementioned problem can be avoided.

5) Parameter Tuning

Different configurations of the employed optimization algorithms or techniques result in different accuracies. The parameters can be adjusted manually or automatically. A researcher may find the best configuration through performing numerous trials based on feedback (e.g., the cost variance), but this approach is only

applicable to algorithms with a limited number of parameters. Whereas, if there are many tunable parameters, a dedicated algorithm (with fewer parameters) is usually used to reduce the parameter searching space and find the best configuration.

6) Data Pattern

The goal of finding the best configurations on the aforementioned five aspects is to maximize their parameter matching with the data pattern, so the changes in data pattern usually indicate that all existing configurations need to be re-determined. As a result, the data pattern is the most important factor that impacts the accuracy.

1.2 Research Scope

A more generalized ANN can be applied as a regular model to solve problems in various fields (e.g. object detection and anomaly detection). The new optimization method proposed in this dissertation is applied on the most widely adopted type 1 ANNs that employs the traditional perceptron and backward propagation for learning data pattern. As a result, several datasets in different fields are employed to evaluate the proposed method through its applications in different ML algorithms, cost functions, and parameters. Type 1 ANNs is available on all mainstream deep learning libraries (e.g., Tensorflow [1.18], Keras [1.19], Caffe [1.20], PyTorch [1.21]), so the new algorithm can be easily implemented into these libraries by adding only a few lines of codes. Moreover, the proposed method is designed based on stochastic gradient descent, as it can accelerate the training process without compromising the accuracy.

1.3 Problem Statements and Contributions

The biggest disadvantage of existing algorithms is that they only use the previous gradients (i.e. the generated or old gradients) in the computations of new gradients (refer to Chapter 2 for details). Due to the fact that the gradient is a time-sensitive variable that is computed based on the cost at a specific moment, all measurements suffer from the deviations that are introduced from the previous gradients. Due to the lack of reliable metrics (i.e., only the previous gradient is in use), the number of effective measurements is limited. As a result, new algorithms are often created by combining multiple existing measurements. Algorithms created in this manner may generate erratic gradients as the incorporated measurements may conflict with respect to their principles. Another significant shortcoming is that each existing algorithm employs fixed measurements during the entire optimization process (i.e., unchanged with respect to time-sensitive variables: the parameters of the model and the output cost), so they cannot generate the optimal gradients for all specific moments in time. Referring to improvements in the algorithms, researchers mainly misattribute any decrease in accuracy to the loss in cost reduction, leading them to ill-modify their measurements.

Our contribution to the entire research community is proposing/verifying that the angle between consecutive gradients as/is an effective new metric for model optimization. In addition to the new angle-based measurements, more effective measurements can be achieved in using the two metrics together, that is, the previous gradient and the angle. With the increasing number of measurements, the creation of new optimization algorithms becomes easier. Most importantly, measurements that rely on different metrics do not result in internal conflicts, therefore generating more accurate gradients. We

determined that the source and cause of the losses in accuracy are due to the weak ability of translating the reduction in the cost to the error rate. By analyzing the experiment data, we found that the angle-based measurements can significantly improve the aforementioned translating ability compared with existing gradient-based measurements. With respect to the proposed algorithm, the adopted angle-based measurements can be introduced into existing optimization algorithms (e.g., AdaDelta [1.22]), enabling them with the benefits of the cost awareness enhancements. In addition, more variants can be easily created by following the four criteria presented in Chapter 6, which may achieve better results on problems in certain fields. The contributions of this dissertation are classified into three groups: (1) measurements, (2) evaluation, and (3) implementation, which are briefly presented as follows:

1) **Measurements**

   a) **A New Metric: the angle between consecutive gradients (Chapter 2 for deviations of gradients; Chapter 3 for the new metric)**

   All existing optimization algorithms cited in this dissertation only use the current and previous gradients as metrics in generating all measurements, such as using the accumulation of gradients to calculate the learning rate. However, the information provided by gradients is not only limited, but is also deviated (refer to Chapter 2). Therefore, the angle between consecutive gradients is proposed as a new metric to provide more information for generating more accurate new gradients.

b) A New Method of Accumulating Gradients (Chapter 2 for an analysis of the two existing methods; Chapter 3 for the new method)

Existing algorithms accumulate the gradients in two imperfect ways: (1) utilizing the exponential moving average or (2) using the simple moving average. A new accumulating method is proposed to possess all the advantages and avoids all the disadvantages of the two methods.

c) Calibrating The Previous Gradients (Chapter 2 for deviations of previous gradients; Chapter 3 for the proposed measurement)

All existing methods utilize the previous gradients to calibrate the current gradient in computing the new gradient. However, there are deviations on the previous gradients, which can negatively impact the accuracy of the new gradient. No method addresses this problem until the proposed method.

d) Angle-based Learning Rate (Chapter 2 for explanations of problems; Chapter 3 for the proposed measurement)

The learning rates determined by existing algorithms are either static (i.e., set it manually) or inversely vary with the magnitude of the accumulated gradients. The former is inflexible in adjusting the new gradient, and the latter suffers from deviations of the accumulated gradients. To resolve these problems, the learning rate of the proposed method is determined using the angle between consecutive gradients.

e) **Cost Awareness Ability (Chapter 3 for the motivation and proposed measurement)**

All cited algorithms cannot infer the cost variance, so they cannot take effective measures to reduce the cost when the cost is increased. However, the proposed method is able to accurately infer the cost variance and take effective actions to immediately reduce the cost in the next iteration.

f) **Decoupled Measurements (Chapter 2 for explanations of problems; Chapter 4 for the demonstration of decoupled measurements)**

Advanced algorithms (e.g., AdaMax [1.10]) improve the gradient and the learning rate by incorporating the measurements from multiple simple algorithms (i.e., one measurement is to improve gradient and the other one is to improve the learning rate). However, the incorporated measurements may conflict with each other in their functionalities, resulting in erratic new gradients. The measurements adopted in the proposed algorithm are well-decoupled in their functionalities.

2) **Evaluation: Translation Rate (Chapter 5)**

Existing algorithms are typically evaluated by reductions in the (1) cost and (2) error rate. Because ML models reduce the error rate through minimizing the cost, losses in the error rate reduction are misattributed to the losses in cost reduction. However, we found that the former may not be caused by the latter, but is attributed to a low translation rate from the latter to the former. As part of the evaluation of the proposed techniques, a new method that quantifies the capability of translating reductions in the cost to the error rate is presented. The quantified results not only

reveal which algorithms have stronger translation abilities, but also demonstrate that the angle-based measurements are better than existing gradient-based measurements in improving the translation ability.

3) Implementation (Chapter 6 for details)

To comprehensively compare the difference in efficiency among various optimization algorithms, each model script has two implementations. One implementation runs on a CPU and the other one utilizes the power of a GPU. To maximize the utilizations of the available computing powers on the two kinds of processors, a technique of matrix-based multiplication (refer to Chapter 6 for details) is employed in each model script, which can only be found in serious deep learning libraries. Also, the model scripts are able to record the variations of 50 different metrics that are associated with the gradient descent. Subsequently, the information can be output to a file with the designated format and visualized using the plotting script for more in-depth analyses.

1.4 Research Approach and Outline

Figure 1.1 shows the outline of this dissertation. In Chapter 2, we first determine deficiencies of 9 existing optimization algorithms and common challenges in model optimization. Then, possible improvements are proposed to address the major problems on the aforementioned two aspects. In addition, we found deviations on the previous gradients through the analysis of stochastic gradient descent, which motivates us to alleviate the deviations and realize the proposed improvements using the angle between consecutive gradients. As a result, a new optimization algorithm composed of 2 functions

and 6 parameters is proposed in Chapter 3. Next, in-depth interpretations in terms of convergence principle and decoupled measurements are presented in Chapter 4. Then, in Chapter 5, the proposed and other 9 existing algorithms (analyzed in Chapter 2) are implemented into two ML models to compare their accuracies and efficiencies under various conditions (i.e., different datasets, cost functions and batch sizes). The translation rates of all algorithms are calculated based on the data of the aforementioned experiments (i.e., the cost and error rate reductions). In Chapter 6, the criteria of defining new variants of the proposed method and a case study of a non-linear variant (results in a faster convergence) are presented. This chapter also verifies the functionalities of the decoupled measurements and the reliability of the new metric. Because the new metric can be calculated in multiple methods, an in-depth analysis of each method is given before the verification of reliability. Although the computation of the new metric is a part of the principle, we placed it before the verification for a coherent demonstration. At the end of this chapter, the intended application of the proposed method can be found. In addition, we used the experiment data to show that the proposed method does not make the cost trapping into the saddle points in practice. In Chapter 7, two possible improvements that are applied on (1) the computation of the angle and (2) the employment of angle-based learning rate schedulers are presented before the significance of the work.

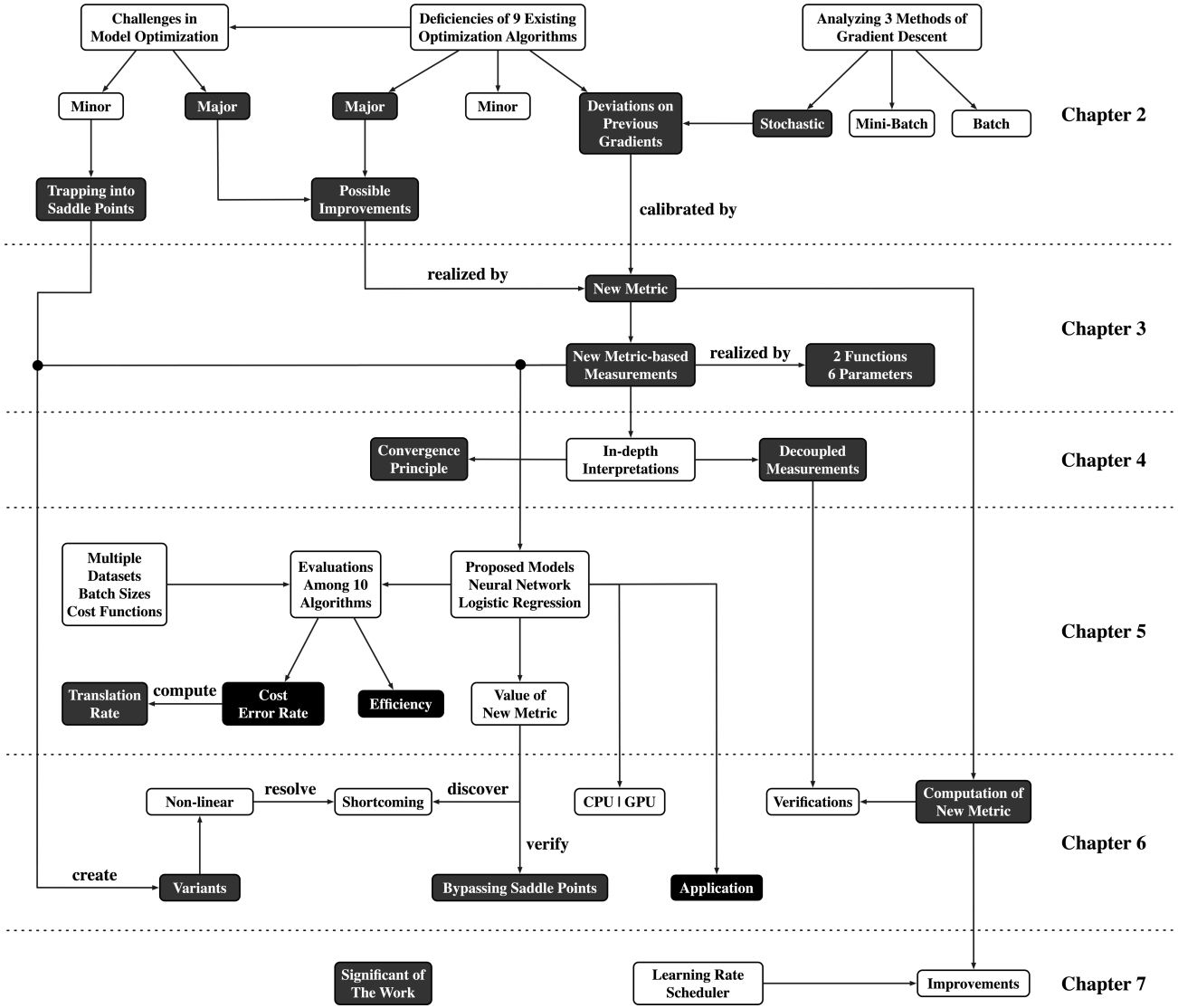


Figure 1.1 – Outline of dissertation (black blocks are important content)

CHAPTER 2

LITERATURE REVIEW

2.1 Model Optimization

2.1.1 Overview

Model optimization intends to maximize the accuracy through searching the best configuration that delivers the lowest output cost. To measure the improvement, the cost is quantified by a cost function $C(y_{pred}, y_{true})$ such as MSE. Due to the fact that predictions y_{pred} made by an $ANN(W, B)$ are determined by weights (W) and biases (B) [2.1], model optimization is a math problem to find the best parameters to minimize $C(ANN(W, B), y_{true})$ (abbreviated as C). Furthermore, the gradient (abbreviated as G) of a function is a vector that points to the direction of steepest slope, the cost reduction can be accomplished by repeatedly applying gradient descent (GD) that is composed of the following three procedures [2.2]:

1) Computing $G(\partial C/\partial w, \partial C/\partial b)$ ($w \in W$ and $b \in B$) of C
2) Multiplying a learning rate η to G to adjust its magnitude.
3) Updating the model parameters (abbreviated as *PARAM,* i.e., *W or B*) by subtracting ηG or adding its reverse $-\eta G$ (i.e., ∇G).

Obtaining G of C needs to pre-compute the partial derivative of each weight $\partial C/\partial w$ and bias $\partial C/\partial b$, so early GD method is only applicable to regular ML models with simple architectures, such as logistic regression (i.e., an ANN without hidden layer). More complex models benefit from GD until backward propagation (BP) is proposed in [2.3]. The new method enables us to simultaneously compute all partial derivatives using only

one forward propagation (FP) which is then followed by one BP. With respect to implementation, matrix-based multiplication [2.4] that is realized by state-of-the-art deep learning libraries makes the training of a deep neural network (DNN) feasible in practice. In recent years, the capability of parallel computation on advanced GPUs is utilized to create various large-scale DNNs.

2.1.2 Variants of Gradient Descent

GD has three variants that are different in the number of samples used to compute each *G*. Because a more accurate *G* requires more samples during its calculation, there exist a trade-off between *G* accuracy and the interval to perform a *PARAM* update. Consequently, one of the main motivations to create these variants is to reduce the aforementioned interval. Otherwise, the task of training a DNN on a large dataset is still almost unachievable even though both BP and matrix-based multiplication are employed [2.1].

2.1.2.1 Batch Gradient Descent

Because Batch Gradient Descent (BGD) computes each *G* from a full training dataset, its *G* is more accurate than the other two variants but incurs the longest updating interval. Also, BGD is guaranteed to converge to one of the local minima on non-convex surfaces and the global minimum on convex surfaces. However, BGD is intractable for datasets that cannot fit in memory and not applicable for training online ML models that receive new data in real-time. Moreover, BGD will incur redundant computation on

samples that are similar in data pattern (i.e., the values of features are almost identical among these samples).

2.1.2.2 Mini-Batch Gradient Descent

Mini-Batch Gradient Descent (Mini-BGD) improves the updating frequency of *PARAM* upon BGD by computing each *G* from a subset of the training data. Due to the large reduction in batch size, the computation of each *G* can be accelerated by the matrix-based multiplication in practice. However, *G* of Mini-BGD is less accurate than that of BGD because the data pattern of a subset is somewhat deviated from the full dataset, rendering a more erratic cost convergence trajectory (this problem can be effectively alleviated by setting larger batch sizes). Another noticeable problem is the sequence of mini-batches remains unchanged during the entire training process. Although a few researchers establish some meaningful sequences to improve the accuracy (i.e., Curriculum Learning) [2.5, 2.6], the ML models would overfit on the fixed sequences in most cases.

2.1.2.3 Stochastic Gradient Descent

Stochastic Gradient Descent (SGD) is originally updating *PARAM* based on *G* of one sample and shuffling all samples before every epoch. This method avoids the redundant computation of BGD and minimizes *PARAM* updating interval; however, its *G* has a larger deviation than the other two variants, leading to the most unstable optimization process. Therefore, SGD is often combined with Mini-BGD to compute *G*s from mini-batches that contain stochastic samples. Furthermore, ML models will not

overfit on specific sample sequences, resulting in a better generalization capability. Although SGD cannot guarantee to converge to an exact local minimum on non-convex surfaces due to the variance of G, its oscillated convergence trajectory could make the cost jumping to a lower local minimum, especially when η warm restarts method (i.e., aggressively decay η and reset it by certain epochs) [1.11] is employed. In addition, SGD would generate almost identical convergence trajectory as BGD when η annealing method (i.e., decay η by a certain number of epochs) [2.7–2.10] is used, making the cost almost certainly converge to one of the local minima on non-convex surfaces and the global minimum on convex surface.

2.1.3 Challenges of Employing Stochastic Gradient Descent

According to the analysis of the three GD variants, SGD can not only accelerate the cost convergence using small batch sizes (instead of using a full training dataset), but also prevent the model from overfitting on specific sample sequences by reshuffling the samples in each batch before each epoch. As a result, SGD is adopted by the majority of ML tasks in various fields and all experiments in this dissertation. However, SGD is imperfect in the following aspects.

2.1.3.1 Inaccurate Gradient

As mentioned, the deviation of G is caused by the inevitable pattern variance between the full training dataset and its mini-batch. Although increasing the batch size can reduce the pattern variance, this method will incur a longer *PARAM* updating interval and slow down the cost convergence. Because the magnitude of G can be calibrated by η,

the deviation primarily affecting the direction of G. In terms of computation, the acceleration realized by the matrix-based multiplication will be gradually compromised with the increasing of batch size. Existing remedies can be classified into three types: (1) multiplying dedicated ηs to elements of each G for calibrating the magnitude and direction of G; (2) combining multiple recent Gs to form a more accurate G; and (3) combining remedies in (1) and (2).

2.1.3.2 Uncertain Learning Rate

The learning rate η correlates to many factors, so there is no consensus on the η adjustment strategy. From a global view, η should be gradually decreased when SGD proceeds to alleviate overshooting and obtain a better minimum (i.e., η annealing) [2.7-2.10]. Also, η could be periodically reset to the initial value when it is lower than a specific threshold (i.e., η warm restarts) [1.11]. From a local perspective, each element of G is a partial derivative that is determined by the slope with respect to the corresponding parameter of C, so a smaller magnitude may indicate a shorter distance to the minimum in the corresponding dimension. Therefore, some η schedulers utilize this indication by increasing/decreasing η with G increases/decreases, so that the cost can converge faster when it is far from the minimum and approach a lower minimum in the final stage. However, other η schedulers that are designed for sparse data adjust η in the opposite way, which also deliver state-of-the-art results in some ML tasks [2.11, 2.12].

It is reasonable to apply a certain η scheduler on a specific problem, but the improvement is often (far) below the expectation, as the actual cost surface is much more complex than the assumed scenario that is used to design the adopted η scheduler.

Furthermore, almost all η schedulers have to preset their parameters before executing SGD, so the improvement in cost reduction could be understood as a better matching between the parameters and the data pattern. This means η still remains highly-uncertain on new tasks because the experiences (i.e., the successful configurations) on the previous tasks could not be valuable references. Consequently, if η is too small, the convergence will be very slow. Whereas, if η is too large, the cost will be oscillated within a large range or even translate to a non-convergence failure.

2.1.3.3 Weak Learning Ability on Sparse Data

A sparse dataset contains some sparse features that are composed of a few possible values (i.e., values are mostly identical). As a result, the slopes of cost surfaces formed by these sparse features are very small. It means that Gs of the sparse features are distinctly smaller than that of the regular features, and the associated W and B will not be adequately updating during the entire training phase. Although Gs could be balanced by multiplying different ηs, this method has a limited application scenario (refer to subsection 2.4.2) and can introduce side effects that compromise the improvements in other aspects (e.g., slow convergence).

2.1.3.4 Others

In addition to the challenges regarding G, η and sparse data, minor problems remain unresolved:

2.1.3.4.1 Trapping into Saddle Points

A saddle point refers to a minimum on a plateau of the cost surface, which is surrounded by surfaces with decreasing slopes toward it [2.13]. It is evident that identifying and escaping from a saddle point is a difficult problem that needs to be solved because it is surrounded by surfaces with decreasing slopes, and this characteristic is the same as the global minimum. Because η warm restarts [1.11] may reach multiple cost minima, we may infer a point as a saddle point if it is higher than the obtained minimum by a large magnitude. However, this method will fail when all minima obtained are saddle points.

2.1.3.4.2 Qualities of Minima

The quality of a minimum cost refers to the generalization ability of the trained ML model. Given a study in [2.14], the number of local minima increases exponentially with the number of *PARAM* (i.e., the complexity of the model). As a result, determining the qualities (i.e., generalization abilities) of all minima through testing the trained models on testing dataset is infeasible in practice. In addition, a complex DNN using an advanced architecture can achieve a very low minimum via memorizing all training samples, compromising the reliability of the evaluation on the testing dataset [2.15]. As a result, some researchers have claimed that the sharpness of the surface surrounding the minimum could be used to infer the quality. More specifically, they have concluded that a sharper minimum has a higher generalization error rate [2.16]. An intuitive explanation is that a sharper minimum indicates a more irregularity training cost surface, so *PARAM* that is determined by this accidental cost convergence would not perform well on the

testing cost surface. However, other researchers' work suggest that the sharpness may not be an accurate indicator because they have identified that some well-generalized local minima can be surrounded by surfaces with arbitrary sharpness [2.17]. Consequently, the evaluation on testing dataset can be substituted, if we can find a reliable metric to quantify the qualities of minima obtained from the training dataset in time. This means that more and better generalization results can be obtained within the same period.

2.2 Introduction of Existing Optimization Algorithms

2.2.1 Overview

The optimization algorithms or optimizers are designed to improve accuracy through resolving the problems associated with SGD (include but not limited to the aforementioned ones). Because SGD updates W and B by subtracting ηG, existing methods resolve the problems by improving G, η or both. More concretely, each variable will be calibrated by one or more terms that are dedicated to reduce its deviation based on a certain theory. In this sense, η is not only a target variable that needs to be improved, but also a calibration term for G.

During SGD calculation, numerous Gs will be sequentially generated, constituting a G chain. As a result, the previous G (PG) associates with the current G (CG), as PG determines the current cost (i.e., the current location on the cost surface) that is used to compute CG. Furthermore, the new G (NG) computed by ηCG also correlates with PG, but with a smaller extent. Therefore, almost all mainstream optimizers utilize PG to improve G, η or both, so that the missing information could be compensated by PG. This indicates that the generated NG may benefit from PG in terms of magnitude, direction or

both. Finally, existing optimizers update *PARAM* by subtracting the calibrated *NG*, as shown by Equation 2.1.

$$PARAM_{t+1} = PARAM_t - NG \tag{2.1}$$

Due to the fact that all optimizers are solely differing in their determinations of *NG*s, Equation 2.1 is omitted in the introduction of 9 existing optimizers in the following subsections. In addition to Vanilla SGD, the other 8 optimizers are introduced based on variables (i.e., G, η, or both) they intend to improve .

2.2.2 Original SGD: Vanilla

Vanilla [2.1] refers to the original SGD that realizes BP without improvement. It computes *NG* by ηCG, as defined in Equation 2.2. It is evident that η is the only adjustable parameter with a fixed value, so *NG* positively varies with *CG* that is determined by the slope of the cost surface. Because a higher slope causes a larger *CG* and indicates a longer distance to the minimum, a larger *NG* reduces the number of steps to approach the minimum. On the other hand, a smaller *NG* resulting from a lower slope will alleviate overshooting and drive the cost to more rapidly approach the minimum. In terms of the direction, *CG* will accurately point in the direction that causes the largest cost reduction, unless the batch size is too small (refer to subsection 2.1.2.1 for the correlation between accuracy and batch size).

$$NG = \eta CG \tag{2.2}$$

2.2.3 Improving Gradient: Momentum

Momentum [1.9] is the earliest and most widely used improvement to *G*. As shown by Equations 2.3 and 2.4, Momentum replaces *CG* in Vanilla by a new variable M_t. Each M_t is obtained from combining the previous M_{t-1} and *CG*. Because M_0 is initialized to 0, M_{t-1} accumulates all *PG*s to time *t-1*, and M_t is the accumulation of all generated *G*s to time *t*. Two coefficients β and $1-\beta$ are weights applied to *PG*s (i.e., M_{t-1}) and *CG*. β is set to 0.9 by default, so each *PG* is gradually reduced by 0.1 when generating a new *NG*. Furthermore, a more recent *G* has a larger influence on *NG* and a higher β will enhance this tendency. As a result, *NG* is generated from the exponential moving average (EMA) of all *G*s, which endows the movement of cost convergence with an inertia-like property, gradually increasing/decreasing the magnitude on a decreasing/increasing slope. In summary, Momentum is an optimizer that not only utilizes *PG*s to make improvement, but also heavily relies on *PG*s.

$$NG = \eta M_t \tag{2.3}$$

$$M_t = \beta M_{t-1} + (1-\beta)CG \tag{2.4}$$

Momentum adopts EMA to prevent the cost from being guided into a ravine that is formed by a surface with a much larger slope in one dimension than the others [2.18], as shown in Figure 2.1. More specifically, when the cost is close to a ravine, *CG* may guide the cost into the ravine. Then, one or more of the following steps will be spend on escaping from the ravine. Due to the fact that there may exist numerous ravines on the way to the minimum, the additional steps that are spent to move away from these ravines would largely delay the cost convergence. However, if *PG*s are used, they will calibrate

the direction of *CG* and generate a *NG* that does not point to the ravine. As a result, *NG* would guide the cost to cross the ravine directly [2.19].

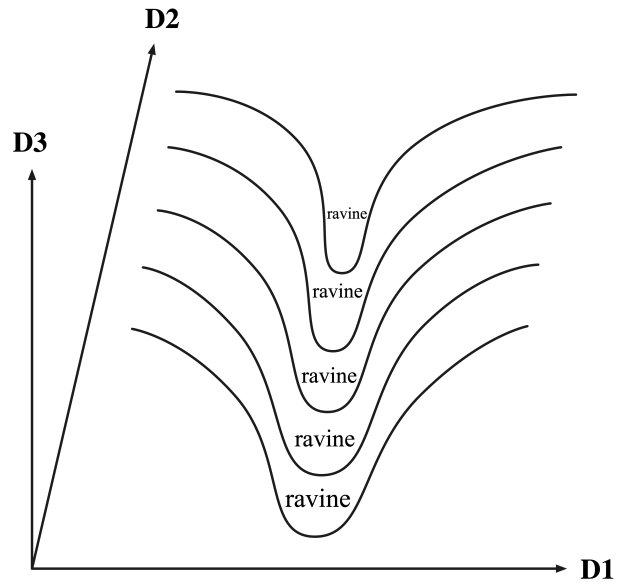


Figure 2.1 – A ravine on a 3-dementional cost surface
(the slope in D3 is much larger than the slopes in D1 and D2)

Employing the accumulated *PG*s also brings some disadvantages. For example, if *PG*s guide the cost to a place that is close to a hill, then *CG* will point to a direction to bypass the hill (for reducing the cost). However, the resulted *NG* will still point to the hill and increase the cost because it is dominated by *PG*s (i.e., inertia-like property). In the worst case, the cost may climb on and cross the hill (e.g., a high cost surface), arriving to another route that cannot reach the global minimum. Although Momentum is not a perfect optimizer, it validates the advantages of employing EMA and motivates most optimizers created afterward to adopt EMA in generating their measurements.

2.2.4 Improving Learning Rate

2.2.4.1 AdaGrad

AdaGrad (Adaptive Gradient) [2.20] is an optimizer that adapts the parameter η, as defined in Equations 2.5 and 2.6. Its goal is to improve the learning ability on sparse data [2.21, 2.22]. As we mentioned, the cost surface in the dimension formed by a sparse feature would be very flat (i.e., the slope is very small), which results in a small G. Therefore, the cost reduction with respect to this dimension will be inadequate. If there are multiple sparse features in the dataset, the convergence and the overall cost reduction (i.e., take all dimensions into consideration) will be substantially slowed down and compromised, respectively. Due to the fact that the sparsity of a feature inversely correlates with the magnitude of the corresponding G, AdaGrad improves the learning ability on a sparse feature by increasing the corresponding η when detecting a small G. This objective is realized by dividing the root of a new variable V_t which is the accumulated element-based squares of all generated Gs to time t. Then, the elements of the new learning rate term $\eta / \sqrt{(V_t + \varepsilon)}$ will be inversely varying with the corresponding elements of V_t. Finally, a G resulting from a sparse feature will be increased, no longer impeding the cost convergence. Another advantage of AdaGrad is that the learning rate term $\eta / \sqrt{(V_t + \varepsilon)}$ is automatically adjusted according to V_t instead of maintaining a fixed value. With respect to computation, the elements of V_t can be allocated in the diagonal of a matrix, so that the computation of NG is accelerated using libraries that implement optimized matrix-vector production [2.23].

$$NG = (\eta / \sqrt{(V_t + \varepsilon)})CG \tag{2.5}$$

$$V_t = V_{t-1} + CG^2 \tag{2.6}$$

There are two disadvantages with respect to η and the convergence speed. V_t is accumulating Gs (i.e., PGs or V_{t-1} and CG) in a simple moving average (SMA) method, and it becomes larger with more mini-batches that are processed. As a result, the learning rate term will gradually approach 0 (i.e., saturated) as more mini-batches are processed, making AdaGrad unsuitable when applied on big data. Moreover, AdaGrad is actually functioning as a "gradient balancer" that not only increases Gs of sparse features, but also decreases Gs of regular features. However, it is not always a reasonable choice to improve the accuracy by carrying out the former with the cost of the latter. Due to the fact that the slope of the cost surface formed by a sparse feature is very small, the maximum cost reduction with respect to this feature is very limited. This indicates that the overall cost will still be very close to the optimal minimum, even if the cost in the aforementioned dimension (i.e., formed by the sparse feature) is far from optimal. Consequently, if AdaGrad is employed on a dataset with a few sparse features, it would not significantly improve the overall accuracy. Instead, it would slow down the cost convergence due to the reductions on Gs of the regular features.

2.2.4.2 RMSprop

RMSprop (Root Mean Squared Backpropagation) [2.24] improves AdaGrad by replacing SMA by EMA in computing V_t, as shown in Equations 2.7 and 2.8. Due to the adoption of EMA, the magnitude of V_t has a specific upper bound under each setting. For instance, if the magnitude of each G is 1 unit and the angles between all consecutive Gs are 0°, V_t will infinitely approach, but never reach 1 under the default configuration (i.e.,

β = 0.9). Therefore, the deficiency of the learning rate reduction that is caused by the infinite increasing of V_t can be completely avoided.

$$NG = (\eta / \sqrt{(V_t + \varepsilon)})CG \quad 2.7$$

$$V_t = \beta V_{t-1} + (1-\beta)CG^2 \quad 2.8$$

However, the adoption of EMA will also weaken the learning ability on sparse features. For example, if the magnitude of a G of a sparse feature is 0.1 unit and all other settings remains the same as the previous example, the upper bound of the corresponding V_t is 0.1. Because V_t is inversely related to η, the calibrated η of this sparse feature (i.e., η / 0.1 = 10η) will be stronger than a regular feature (i.e., η / 1 = η) by 9η during the entire training stage. Whereas, if SMA is adopted, V_t will increase when more mini-batches are processed. For instance, the difference in the aforementioned two calibrated ηs will be increased to 90η on the 100th mini-batch and 900η on the 1000th mini-batch, respectively. These results show that RMSprop will only increase the learning ability on sparse features by a certain fixed magnitude instead of repeatedly enhancing it, as done in AdaGrad. Furthermore, if the difference in sparsity between sparse and regular features is large, the increased magnitude in the learning ability determined in RMSprop may be inadequate. As a result, RMSprop should be applied on larger datasets due to its non-diminishing learning ability, and AdaGrad could achieve better results on smaller datasets because of its stronger (i.e., increasing) learning ability on sparse features. In this sense, RMSprop is not an improvement, but a variant of AdaGrad.

2.2.4.3 AdaDelta

AdaDelta (Adaptive Delta) [1.22] is an improvement based on RMSprop, which is defined in Equations 2.9–2.11. It provides another method to resolve the aforementioned diminished learning ability of AdaGrad. To counteract the increasing V_t on the denominator, the learning rate η on the numerator is replaced by a new variable D (i.e., Delta) which refers to the difference in *PARAM* before and after each update. Because both D_{t-1} and V_t are initialized to 0 and employ EMA in accumulating the corresponding Gs, their values are comparable during the entire training phase. It indicates the entire learning rate $\sqrt{(D_{t-1} + \varepsilon)} / \sqrt{(V_t + \varepsilon)}$ will not approach 0, but oscillate around 1 (i.e., a stable learning rate). The adoption of EMA also makes both D and V dominated by their corresponding values in recent mini-batches, so the computation of the learning rate is confined to a fixed window size.

$$NG_t = (\sqrt{(D_{t-1} + \varepsilon)} / \sqrt{(V_t + \varepsilon)})CG \tag{2.9}$$

$$D_{t-1} = \beta D_{t-2} + (1-\beta)NG_{t-1} \tag{2.10}$$

$$V_t = \beta V_{t-1} + (1-\beta)CG^2 \tag{2.11}$$

When applying AdaDelta on a dataset, both D_{t-1} and V_t will be increased/decreased on a regular/sparse feature (caused by the slope of a surface that is formed by a feature, refer to subsection 2.2.4.1). The ratios of D_{t-1} and V_t (i.e., the learning rate term) on the two types of features are comparable in magnitude, indicating that the learning ability on a sparse feature is not enhanced compared with a regular feature. In this sense, AdaDelta is actually a "gradient equalizer" that applies consistent learning ability on all kinds of features. Although AdaDelta is proposed to resolve the diminished learning term of AdaGrad, it should be considered as a special Vanilla (i.e., a

fixed η) with a slightly fluctuated η, and nothing similar to AdaGrad and RMSprop in terms of practical behaviors.

2.2.5 Improving Gradient and Learning Rate

2.2.5.1 Adam

Adam (Adaptive Moment Estimation) [1.10] combines Momentum with RMSprop to obtain all advantages of both optimizers. Referring to Equation 2.12, Adam replaces CG by a new variable M_t as Momentum, and divides η by the square of another new variable V_t as done in RMSprop. However, these two new variables M_t and V_t are respectively different from the original definitions (used in Momentum and RMSprop) in dividing two calibration terms $1-\beta_1^t$ and $1-\beta_2^t$, as shown in Equations 2.13–2.16. As a result, Adam computes NG from the calibrated M_t and V_t.

$$NG = (\eta / \sqrt{V_t} + \varepsilon)M_t \tag{2.12}$$

$$M_t = m_t / (1-\beta_1^t) \tag{2.13}$$

$$V_t = v_t / (1-\beta_2^t) \tag{2.14}$$

$$m_t = \beta_1 m_{t-1} + (1-\beta_1)CG \tag{2.15}$$

$$v_t = \beta_2 v_{t-1} + (1-\beta_2)CG^2 \tag{2.16}$$

The advantages of employing M_t and V_t are explained when we introduce Momentum and RMSprop. The aforementioned two calibration terms are to counteract the deviations caused by employing EMA in computing m_t and v_t. A rigorous deduction of the exact deviations can be found in [2.25], and an intuitive explanation is presented as follows. Due to the adoption of EMA, both m_t and v_t are always dominated by their recent values. Because m_0 and v_0 are initialized to 0, EMA will establish the early m_ts and v_ts

bias to 0, which are distinctly smaller than the later ones. If we calibrate m_t and v_t by Equations 2.13 and 2.14, a value generated earlier will be increased by a larger magnitude with the corresponding denominator approaching 0 (note: the superscripts of β_1 and β_2 refer to the power in math). As a result, the bias on each m_t and v_t can be precisely calibrated by a dedicated value, which in turn generates a more accurate *NG*.

Although the biases caused by EMA persist longer if β_1 and β_2 are larger, it rapidly decays when more mini-batches are processed. More specifically, assume that the magnitude of each *G* is 1 unit and the angles between all consecutive *G*s are 0˚ when $\beta =$ 0.9. Then, all $\beta^t s$ since 175th mini-batch (i.e., $\beta^{175} = 0.9^{175} = 9.8274 \times 10^{-9}$) are smaller than the threshold $\varepsilon = 1 \times 10^{-8}$ that is designed for preventing the denominator becomes 0 during the computation (note: changing the assumed conditions such as the angle will not change the reduction rate of β^t). This means that the two calibration terms $(1-\beta_1^t)$ and $(1-\beta_2^t)$ will infinitely approach 1, and the associated calibrations to m_0 and v_0 will disappear after the 175th mini-batch. Due to the fact that the number of mini-batches in an ML task would reach levels of 10^5, 10^6, 10^7 or even a larger number, the benefits from calibrating the biases during the first 175 mini-batches could be negligible. Consequently, Adam could be considered as a simple combination of Momentum and RMSprop.

2.2.5.2 AdaMax

AdaMax (Max refers to l_∞ norm) [1.10] is a variant of Adam, which is defined in Equations 2.17–2.20. To resolve the bias of V_t caused by EMA, AdaMax performs the accumulation of *G*s in l_∞ norm instead of l_2 norm (i.e., $\sqrt{CG^2}$ in Adam). The reason for choosing the l_∞ norm is that it presents a very high numerical stability as the l_2 norm in

vector quantification. Due to the adoption of l_∞ norm, V_t is always determined by the larger value between the accumulated *PG*s (i.e., $\beta_2 v_{t-1}$) and *CG* (a detailed mathematical deduction can be found in [1.10]). As a result, the term (i.e., $\beta_2 v_{t-1}$ or *CG*) that biases to 0 will never be selected as V_t.

$$NG = (\eta / V_t) M_t \tag{2.17}$$

$$M_t = m_t / (1-\beta_1^t) \tag{2.18}$$

$$V_t = \beta_2^\infty v_{t-1} + (1-\beta_2^\infty) CG^\infty = \max(\beta_2 v_{t-1}, CG) \tag{2.19}$$

$$m_t = \beta_1 m_{t-1} + (1-\beta_1) CG \tag{2.20}$$

There are many shortcomings that can be found from the definition of AdaMax. For instance, AdaMax does not employ the l_∞ norm to remove the bias on M_t or m_t. One of the reasonable explanations is that the reliability of a *G* is not only determined by its magnitude, but also determined by its direction. Therefore, the magnitude cannot be used to judge the accuracy of m_t. However, the same justification can be used to refute the employment of l_∞ norm in computing V_t. Due to the fact that the accuracy of the learning rate η / V_t is also not correlated with magnitudes of *G*s, selecting a larger value for V_t will not result in accurate *NG*s. In addition, if $\beta_2 v_{t-1}$ is larger and selected as V_t, the reliable *CG* (computed based on the most-updated parameters of the model) will be abandoned, rendering an inaccurate *NG*. Furthermore, always generating V_t from the larger term/gradient(s) (i.e., $\beta_2 v_{t-1}$ or *CG*) will lead to a smaller learning rate. Compared with the rapid disappeared bias on V_t (bias disappears after 175th mini-batch, refer to subsection 2.2.5.1), the side-effect (i.e., slower convergence) from the countermeasure of removing the bias will persist during the entire training stage. Therefore, AdaMax might not be a successful variant of Adam.

2.2.5.3 Nadam

Nadam (Nesterov-accelerated Adaptive Moment Estimation) [2.26] makes improvement by incorporating NAG (Nesterov Accelerated Gradient) into Adam, which is defined in Equations 2.21–2.25. NAG is an independent method that enables any optimizer with the prescient ability to improve the accuracy of *NG* [2.27]. More specifically, Nadam computes *NG* from $M_t(m_t)$ and uses it to update the current $PARAM_t$ to a new state called $PARAM_{proj}$. However, $PARAM_{proj}$ is not the $PARAM_{t+1}$ that we want to obtain, but rather to compute a projected G_{proj} which contains the information of the cost surface one step ahead. As a result, m_t will be calibrated by G_{proj} to generate a more accurate M_{proj} and *NG*.

$$NG = (\eta / \sqrt{V_t} + \varepsilon)M_{proj} \tag{2.21}$$

$$M_{proj} = (\beta_1 m_t + (1-\beta_1)G_{proj}) / (1-\beta_1^t) \tag{2.22}$$

$$V_t = v_t / (1-\beta_2^t) \tag{2.23}$$

$$m_t = \beta_1 m_{t-1} + (1-\beta_1)CG \tag{2.24}$$

$$v_t = \beta_2 v_{t-1} + (1-\beta_2)CG^2 \tag{2.25}$$

Nadam benefits the cost reduction in two scenarios. If m_t leads the cost to a surface with a higher cost, G_{proj} would point in the direction of leaving the surface. Therefore, G_{proj} will counteract m_t and generate a M_{proj} that bypasses the high-cost surface. However, this advantage can be obtained only when β_1 is closing 0. More concretely, when β_1 is closing 1 (e.g., $\beta_1 = 0.9$ under the default setting), m_t (i.e., the accumulated *G*s) is dominated by recent *G*s that drive the cost to the high-cost surface. In this case, the magnitude of G_{proj} is not adequate to counteract m_t and reduce the cost to a lower value.

In addition, when m_t and G_{proj} roughly align with each other and point in the direction of a lower cost, M_{proj} will be a G that begins with the head of m_t and ends up with the tail of G_{proj}. Nadam can reduce the cost to a lower value in only one step (i.e., achieved by M_{proj}) instead of two steps (i.e., realized by m_t and G_{proj}) as Adam is employed. In terms of computation, the two steps of Adam need to execute 2 BPs which corresponds to 4 FPs. Whereas, Nadam needs to spend one additional FP to compute G_{proj} in addition to spend 1 BP (i.e., 2 FPs) in calculating m_t (i.e., 3 FPs in total). Because 1 step of Nadam corresponds to 2 steps of Adam (i.e., m_t and G_{proj} can be achieved by M_{proj}), the computation cost per step for Nadam and Adam are 1.5 FPs (i.e., 3 FPs / 2 steps = 1.5 FPs / step) and 2 FPs (i.e., 2 FPs / 1 step), respectively. Therefore, Nadam is more efficient than Adam by (2 – 1.5) / 2 = 25% in cost reduction. However, the advantage in cost reduction is obtained with a lower model testing frequency, which in turn renders a lower possibility of capturing lower costs. More concretely, Adam updates $PARAM_t$ every 1 BP = 2 FPs, but Nadam increase this interval to 1 BP + 1 FP = 2 FPs + 1FP = 3 FPs. Meaning that Nadam is more likely to miss a lower cost than Adam by (3 – 2) / 3 $\approx$ 33% within the same period. It is worth to mention that there is no evidence to show that the reliability of M_{proj} is higher than m_t (i.e., the accumulated Gs), as G_{proj} in M_{proj} (i.e., the projected G based on m_t) is only a one-step-ahead G of m_t, and there is nothing special in accuracy with regard to G_{proj}. Due to the aforementioned advantages in computation and disadvantage of missing lower costs, Nadam might not be an improved variant of Adam.

2.2.5.4 AMSGrad

AMSGrad (Adam with Max Gradient) [2.28] is a variant that removes the two calibration terms of Adam, defined in Equations 2.26–2.29. Another modification is that V_t is determined by the larger value between the accumulated *PG*s (i.e., V_{t-1}) and all generated *G*s (i.e., v_t). The reason of adopting this approach is that authors found some mini-batches that are more valuable than others, resulting in a larger cost reduction. Because they believe a large cost reduction would cause a large *G*, they intend to reuse the large *G*s to foster the cost convergence. As a result, *NG*s generated using AMSGrad are always composed of the most valuable *G*s.

$$NG = (\eta / \sqrt{V_t} + \varepsilon)M_t \tag{2.26}$$

$$V_t = \max(V_{t-1}, v_t) \tag{2.27}$$

$$M_t = \beta_1 m_{t-1} + (1-\beta_1)CG \tag{2.28}$$

$$v_t = \beta_2 v_{t-1} + (1-\beta_2)CG^2 \tag{2.29}$$

It is evident that the modification in computing V_t has many critical deficiencies. For example, the accuracy of *G* is not determined by the magnitude but by its direction. The accuracy positively correlates with the magnitude only when the direction is pointing towards a lower cost, which cannot be guaranteed in practice. Conversely, even if the magnitude of *G* could be used to quantify the reliability, we can easily find the computation of V_t violates the principle of SGD. For example, if V_{t-1} is larger than v_t for several consecutive mini-batches (it happens especially when a learning rate decay scheduler is used), the outdated *G*s that are accumulated in V_{t-1} are repeatedly selected as V_t. As a result, *NG*s (computed from V_t) that are only generated from *PG*s (i.e., V_{t-1}) are inaccurate because *G* is a time-sensitive variable as the output cost (refer to in Chapter 1).

2.2.6 Others

In recent years, other methods proposed to resolve the shortcomings of the aforementioned optimization algorithms. For example, AdamW (W refers to weight) fixes the weight decay of Adam [2.29]; QHAdam (Quasi-Hyperbolic Adam) computes *NG* by averaging *CG* and m_t in Adam [2.30]; AggMo (Aggregated Momentum) updates $PARAM_t$ by averaging multiple M_ts in Momentum [2.31].

2.3 Classification and Hierarchy of Existing Optimization Algorithms

A summary of existing optimization algorithms can be found in Table 2.1. Although we introduce these optimizers based on the components (i.e., *G*, η or both) which they intend to improve, we classify them into 4 groups according to their actual behaviors in cost reduction. In group 1, there are two optimizers, Vanilla and AdaDelta. Vanilla uses a fixed η to adjust *CG* that is computed from BP. AdaDelta works as a gradient equalizer that sets comparable ηs for all features, so it can be considered a special Vanilla with a slightly fluctuated η. In group 2, Momentum is the only optimizer that employs accumulated *G*s with a fixed η. It is the only optimizer that solely improves *G*, so it is often incorporated into newer optimizers to enhance *G*. In group 3, the two optimizers AdaGrad and RMSprop improve the learning ability on sparse data by improving/decreasing *G*s of sparse/regular features. To resolve the problem of η diminishing on AdaGrad, RMSprop accumulates *G*s in EMA instead of SMA. In group 4, each optimizer is a combination of multiple optimizers. Adam and AdaMax are combinations of Momentum and RMSprop, but they are separately using l_2 and l_∞ norms in accumulating *G*s. Nadam is the most complicated optimizer because it combines an

additional technique NAG onto Adam to obtain the prescience ability. AMSGrad is the only Adam variant that abandons the bias-correction terms, improving the accuracy by reusing the largest *G*s.

Table 2.1 – Summary of existing optimization algorithms

Group	Optimizer	Improving		EMA	$\eta \propto G$	Summary of the Actual Behaviors
		G	η			
1	Vanilla [2.1]					the original SGD with a fixed η
	AdaDelta [1.22]		•	•	•	Vanilla with a slightly fluctuated η
2	Momentum [1.9]	•		•		using *G*s with a fixed η
3	AdaGrad [2.20]		•	SMA	•	balancing *G*s of features
	RMSprop [2.24]		•	•	•	AdaGrad that accumulates *G*s in EMA
4	Adam [1.10]	•	•	•	•	Momentum + RMSprop (l_2 norm)
	AdaMax [1.10]	•	•	•	•	Adam that employs l_∞ norm
	Nadam [2.26]	•	•	•	•	Adam + NAG
	AMSGrad [2.28]	•	•	•	•	Adam that always employs the largest *G*s

A hierarchical chart that presents the development relationship among existing optimizers can be found in Figure 2.2. These optimizers are divided into 4 groups as shown in Table 2.1. They connected through arrow lines that indicate their inherited relationship. Each black block with a white letter refers to the component that the corresponding optimizer intends to improve. When EMA is first incorporated, the key word "EMA" is shown on the corresponding line. For example, Adam is the combination of Momentum and RMSprop, which improves G/η based on the former/latter.

Momentum/RMSprop adopts EMA in improving G/η, and thus Adam inherits EMA in computing the two components.

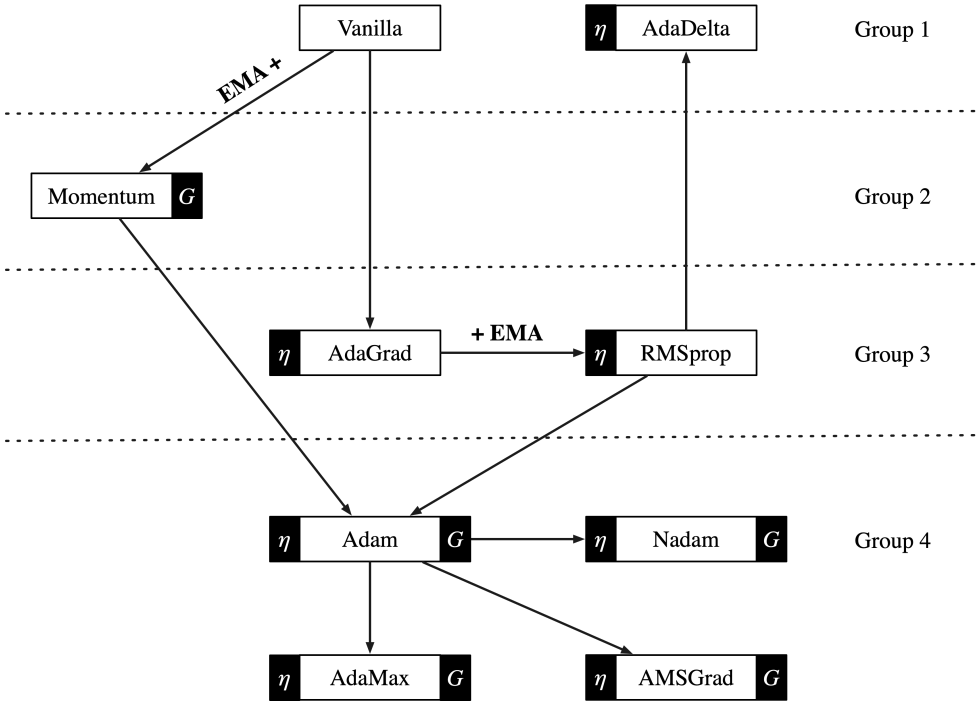


Figure 2.2 – Development relationship of existing optimization algorithms

2.4 Deficiencies on Existing Measurements

Based on Table 2.1 and Figure 2.2, we learn that the early optimizers in groups 1–3 make improvements by incorporating innovative techniques, such as computing a dedicated η for each feature to increase the learning ability on sparse data (i.e., adopted by AdaGrad). By contrast, the newer optimizers in the group 4 are combining multiple existing optimizers and/or techniques with minor enhancements. For example, Adam is unique in quantifying and calibrating the biases on G and η. However, these biases will rapidly disappear after the 175th mini-batch, so Adam does not make an observable improvement in accuracy. An in-depth analysis on the common measurements and the associated deficiencies are given below.

2.4.1 Exponential Moving Average

According to the column under the keyword "EMA" in Table 2.1 and Figure 2.2, all optimizers in addition to AdaGrad adopt EMA when trying to improve G, η or both. Because the earliest optimization algorithm Vanilla solely employs CG, researchers attribute all problems (including, but not limited to high training error rate and slow convergence) to CG. Then, PGs are introduced in the computation of NG to calibrate the deviation of CG via reducing its weight. Due to the fact that an earlier PG would be less reliable in computing NG, EMA is adopted to use the more recently generated Gs to dominate NG and approximates the magnitude of NG to CG (or preventing G/η from becoming infinite large or small as training progresses) [1.9].

However, recent papers show some shortcomings that challenge the adoption of EMA. As we mentioned, the authors of Adam found a bias in employing EMA [2.25]. Also, some researchers have proposed that a lower β (i.e., less weighted PGs) performs better in their respective experiments [2.32, 2.33]. In addition, a recent theory [2.28] suggests that adaptive optimizers that use EMA converge to different and less optimal minima than Vanilla. The authors concluded that generating NG from recent Gs is unreliable, a phenomenon found in (yet not limited to) the fields of object recognition, character-level language modeling, and constituency parsing. These results are combined to showcase multiple issues (including, but not limited to the bias and over-weighted PGs) with EMA, which should be attributed to the high deviation of PGs. Evidence suggests that the deviation of PGs will result in a lower accuracy. For example, multiple experiments in [2.34] have shown that AMSGrad consistently achieves less optimal

results because it will keep using *PG*s by abandoning *CG* when the value of V_{t-1} is larger than that of v_t (refer to Equation 2.27).

Other evidences show that *PG*s should not dominate the computation of *NG*. Although the deviation of an earlier *PG* will be reduced by multiplying more βs when EMA is employed, the magnitude of deviation will also increase as it becomes more outdated. As a result, the actual deviation of each *PG* may not be effectively reduced by EMA. In this sense, the process of accumulating *PG*s in EMA becomes a process of accumulating deviations of *PG*s. Therefore, EMA may not result in an accurate *NG,* especially when β is closing 1 (i.e. more rely on *PG*s).

2.4.2 Inverse Relationship Between Gradient and Learning Rate

As we learn from Figure 2.2 and the column under "$\eta \propto G$" in Table 2.1, any optimizer that is derived from AdaGrad inversely adjusts its η based on *G*. The goal is to improve η for a small *G* that results from a sparse feature, so that the cost in the corresponding dimension can be better reduced. As we mentioned, if this method is applied to a regular dataset with a few sparse features, it would slow down the convergence and not obtain a noticeable cost reduction. Whereas, if the majority of a dataset are sparse features, directly improving ηs of all features (e.g., setting a static large η for all features) is a much easier approach than the aforementioned method. Particularly, this method can completely avoid the mentioned problems with respect to the diminished η and biased V_t. Therefore, the method of inversely adjusting η based on each *G* is only suitable for datasets with a certain number of sparse features (i.e., not too less or many). For such a dataset, if we do not increase ηs of its sparse features, the cost in the corresponding dimensions will be less optimal, in turn weakening the overall cost

reduction. In this case, inversely adjusting η based on each G is a more fine-grained method than improving η of all Gs. It is evident that the improvement will be maximized when the regular and sparse features of this dataset are numerically comparable. However, we need to note that this kind of dataset is rare in practice, limiting the application scope of the measurement.

2.4.3 Non-decoupled Measurements

In recent years, some researchers have pointed out that state-of-the-art results for many tasks (e.g., object recognition in computer vision [2.35] and natural language processing in machine translation [2.36]) have been achieved using simple optimizers (e.g., Momentum). Also, the advanced optimizers like Adam may be susceptible to render a non-convergence failure in some cases [2.37]. By analyzing existing algorithms, we found that measurements of advanced optimizers may conflict in their functionalities, which could be the reason for causing the aforementioned negative results. For example, Adam is a combination of Momentum and RMSprop. As we mentioned, Momentum is designed to increase/decrease G of a cost surface with a high/low-slope. Whereas, RMSprop is trying to balance all Gs by increasing/decreasing G on a flat/steep cost surface. As a result, the two conflicted approaches are combined to generate erratic NGs, translating to an unstable convergence or even a non-convergence failure in practice.

Furthermore, multiple parameters introduced by different methods may become an obstacle during configuration. For instance, each optimizer in group 4 has three hyper-parameters (i.e., η, β_1, β_2), but most optimizers in group 1–3 have only one hyper-parameter (i.e., η). The difficulty of finding the best configuration grows exponentially when a new parameter is introduced, especially when the parameters are non-decoupled

in their functionalities and the ML model architecture is complex. Consequently, some researchers rely on dedicated algorithms [2.38] to optimize parameters, apparently increasing the complexity of a ML task. Some algorithms employ special methods to intelligently reduce the parameter searching space [2.39], and others simply use brute-force methods [2.40] which act as simple iterators that cannot save the configuration time in practice.

In addition, deficiencies of incorporated methods are also imported which would further weaken the accuracy of *NG*. For instance, the shortcoming of slow convergence of AdaGrad will be transferred to Adam via introducing RMSprop. Moreover, if we intuitively understand the generation of accurate *NG*s is a process of searching a perfect parameter matching among all employed measurements, the possibility of achieving reliable *NG*s would be lower when more measurements are introduced.

2.4.4 Strategy of Approaching Lower Minima

It is evident that the surface slope becomes lower when the cost is closer to the minimum, such that the key of approaching lower minima is to reduce the magnitude of *NG* with slope decreases. Due to the fact that Vanilla solely uses the original *CG* as *NG* which positively changes with the slope, it is the only optimizer that can achieve this goal without compromise. This is also one of the main reasons for the highest accuracy in some ML tasks are still achievable by Vanilla [2.35, 2.36]. Whereas, AdaGrad adjusts its *G*s by increasing/decreasing their magnitudes when the slope is low/high (i.e., in the opposite way as Vanilla). In addition, all other optimizers employ EMA in computing η, *G* or both, reducing the dependency (i.e., weight) on *CG*. Employing EMA will compromise the positive correlation between *NG* and the slope, and make the cost

wandering around the minimum by a larger range. As a result, almost all existing optimizers have a deficiency in the strategy of approaching lower minima.

2.5 Summary

The most advanced trend in improving optimization algorithms is not to focus on improving the simple optimizers (e.g., Vanilla), but the complex ones (e.g., Adam) as shown in [2.29–2.31]. According to the presented recent results, the accuracy and stability of an optimizer is generally weakened if more methods are incorporated. Based on our analysis of deficiencies on existing measurements, we believe that "less is more" should be the principle of designing a new optimizer. Although an optimizer with fewer improvements upon Vanilla may limit its application scope (e.g., AdaGrad is best for datasets with a certain number of sparse features), applying a dedicated algorithm to a specific kind of problem is a widely accepted strategy in the field of ML. In addition, recent experiments of language modelling demonstrate that tuning [2.41] and/or regulating parameters [2.42] can produce state-of-the-art results compared to employing more complex models. The results show that improvements result from tuning an optimization algorithm is comparable with or even better than improvements toward the architecture of ML models. Therefore, simple optimizers in groups 1–3 would be better options in terms of making improvements and practical usage.

CHAPTER 3

DESIGN OF ANGLE-BASED STOCHASTIC GRADIENT DESCENT

3.1 Motivation

The creation of angle-based SGD (AG-SGD) is motivated by resolving the shortcomings of existing optimization algorithms, which are presented in Chapter 2. In addition, AG-SGD inherits some existing measurements that benefit its accuracy, such as the incorporation of *PG*s in its *NG* computation. The justifications of adopting and abandoning certain existing measurements are listed in Tables 3.1 and 3.2, respectively. All these justifications are analyzed in Chapter 2, and there are four improvements to the adopted measurements that need to be mentioned, they are:

1) Referring to the third and fourth rows of Table 3.1 and the second-to-last row of Table 3.2, AG-SGD intends to adjust its η based on the progress of convergence. More specifically, the cost convergence can be accelerated by increasing η when the cost is approaching the minimum, and a lower cost can be obtained by decreasing η when the cost is wandering around the minimum. As a result, the strategy of adjusting η will be different from all other η schedulers (i.e., η annealing and η warm restarts) mentioned in Chapter 2.
2) As shown by the first two rows in Table 3.2, both EMA and SMA are abandoned in computing *G*s and ηs due to the listed reasons, so the approach adopted by AG-SGD will be unique at consistently employing the accurate gradients (i.e., resolving the shortcoming of EMA) without suffering from the unlimited gradient increasing problem (i.e., resolving the shortcoming of SMA).

3) Although AG-SGD abandons NAG due to its uncertainty in improvement (refer to the last row of Table 3.2), the idea of prescience ability that comes with NAG is adopted by AG-SGD (refer to second-to-last row of Table 3.1). As a result, AG-SGD will realize an awareness ability to prevent the cost from increasing.
4) As shown by the last row of Table 3.1, AG-SGD also intends to improve both components of G and η, so the incorporated measurements can be decoupled in functionality, otherwise, the generated NG becomes erratic (refer to Chapter 2).

Table 3.1 – Existing measurements adopted by AG-SGD

Adopted Measurement	**Justification**
Improving G	improving the ability of crossing cost ravines
Incorporating PGs	providing more information
Improving η	early/middle stage (the cost is approaching the minimum): faster convergence (increasing η) final stage (the cost is wandering around the minimum): stronger convergence (decreasing η)
Dynamic η	
Prescience/Awareness Ability	preventing the cost from being increased reducing the cost in one step when the cost is increased
Decoupling Measurements	the incorporated measurements should be decoupled in functionality

Table 3.2 – Existing measurements abandoned by AG-SGD

Abandoned Measurement	**Justification**
EMA(Gs) / EMA(ηs)	dominated by outdated/unreliable gradients
SMA(Gs) / SMA(ηs)	unlimited increasing in magnitude
Calibrating η based on Gs	should be based on the progress of convergence
$\eta \propto G$	narrow application scope: only targeting at sparse data
Max (Gs_{t-1}, Gs_t)	critical deficiency in principle
NAG	uncertain improvement

In addition to the four improvements to existing measurements, the most important measurement that has never been adopted by all existing optimizers is the calibration of

outdated *PG* [3.1]. By alleviating the deviation of *PG*, all measurements using *PG* can be enhanced simultaneously, resulting in a significant improvement to the overall accuracy. As a result, the proposed AG-SGD uses a new metric (i.e., the angle between consecutive gradients) to minimize the deviation of *PG*, and the proposed improvements are based on this new metric .

3.2 Principle

3.2.1 New Metric: Angle Between Consecutive Gradients

To calibrate the inaccurate *PG*, we need to quantify its deviation. To achieve this goal, we need to find an accurate *G* as a reference for quantification. In principle, *CG* is calculated based on the most-updated SGD state, so it is more reliable than *PG* in computing *NG* [3.2]. Some researchers have mentioned that the distortions affect *PG* and have recognized the importance of utilizing *CG* in *NG* calculation. They have proposed various methods to enhance *PG*'s compliance with *CG*. For example, implicit gradient transport (IGT) alleviates the "staleness" of *PG* by transforming *PG* into *CG* without explicitly using the Hessian technique to reduce the parameter's variance and bias as it is updated over time [3.3]. In addition, various authors have indicated their results associated with state-of-the-art tasks such as object recognition in computer vision [3.4] and natural language processing in machine translation [3.5], which are relying exclusively on *CG*. In support, a recent study [3.6] suggests adaptive optimizers (utilize *CG* and *PG*) converge to sub-optimal minima compared to the simplistic gradient descent (only use *CG*). This phenomenon can be found in, but not limited to the fields of object recognition, character-level language modeling, and constituency parsing. These results

and the demonstrations in Chapter 2 are combined to show that *CG* is a qualified gradient reference and can be used to quantify the deviation of *PG* by its angle with *PG*.

3.2.2 New Measurement: Calibrating The Deviation of The Previous Gradient

AG-SGD technique generates *NG* by the following four steps: (1) determine the inner angle, θ between *PG* and *CG* (the method of computing θ can be found in Chapter 6); (2) adjust the weights of *PG* and *CG* according to θ, (3) combine the weighted *PG* and *CG*, (4) multiply the learning rate η, accordingly based on the *G* combination. This is possible, as the gradient matches the parameters of a neural network with regard to its data structure (i.e., multi-dimensional matrix), the elements of gradient can be flattened into a vector *V*. As a result, the inner angle between *PG* and *CG* can be computed by the equation below (the detailed explanation of the computation and the associated reliability can be found in Chapter 6).

$$\theta = arccos((V_{PG} \bullet V_{CG}) / (|V_{PG}| \, |V_{CG}|))(180 / \pi) \quad (1)$$

Figure 3.1 shows the deviation between the two *G*s, where the number at the tip of each arrow indicate the θ value, the angle deviation between the associated *PG* and *CG*. For example, PG-54 means that the angle of *PG* relative to that of *CG* is 54°. The figure denotes that a smaller θ renders a closer alignment, indicating a smaller deviation of *PG*. Furthermore, we can divide the range of θ into three subsets [0°, 90°), (90°, 180°], and [90°], in order to follow the corresponding actions:

1) When $\theta < 90°$, *PG* (blue) is roughly aligning with *CG* (black). In this case, *PG* = OPG will be used to compute *NG* without calibration.

2) When $\theta > 90°$, *PG* (red) has a significant deviation from *CG*. Under this circumstance, *PG* has to be calibrated prior to *NG* computation. An easy way is to reverse the direction of *PG*, so that the reversed past *G* (RPG) will be roughly aligning with *CG* as in the previous case. For instance, RPG-18 (yellow) is a reversal in direction of PG-162, which will be used to compute *NG* directly.

3) When $\theta = 90°$, *PG* (green) is perpendicular with *CG*. It indicates that the two *G*s have no correlation (i.e., two vectors have no correlation when they are orthogonal). Therefore, *PG* should be abandoned when we compute *NG*.

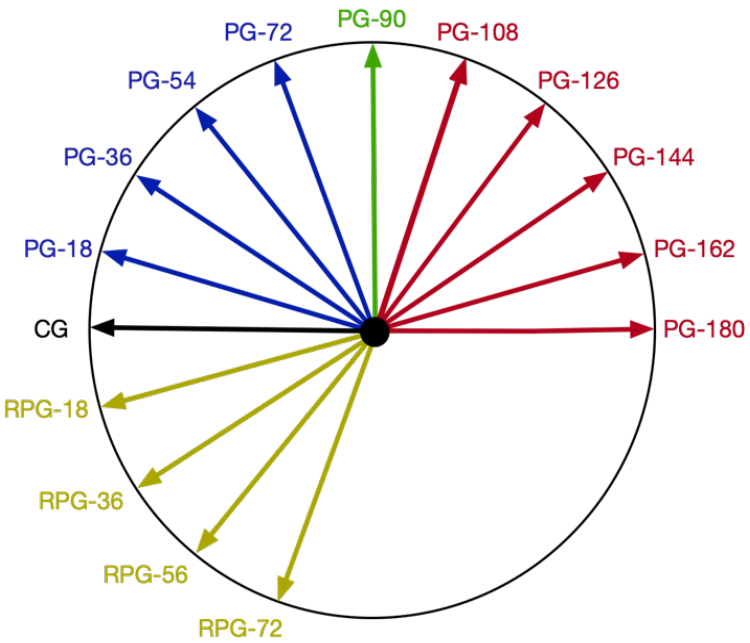


Figure 3.1 – Quantifying the deviation of *PG* using the inner angle between *PG* and *CG* (the numbers at the tip of the arrows are angles; the deviation of *PG* varies with the angle)

To further utilize the above properties, two key points related to the accuracy of *PG* in determining *NG* need to be indicated. (1) As θ approaches 0˚ or 180˚, the OPG or RPG aligns more with *CG* and becomes more valuable in *NG* computation. (2) As θ

approaches 90˚, *PG* will have less contribution to the estimation of *NG* as it correlates less with *CG*.

Due to the fact that an accurate *NG* must result from reliable *G*s, the improvement can be realized by adjusting the weights between *PG* and *CG* according to θ, that is:

1) When θ approaches 0° or 180°, we can increase the ratio w_{pg}/w_{cg} by a larger value because now the OPG or the RPG has better accuracy.
2) When θ approaches 90°, the value of the ratio w_{pg}/w_{cg} will be decreased, i.e., *CG* dominates *NG* computation in this case.

Moreover, it is evident that with a larger θ, the SGD trajectory becomes more chaotic. This occurs as the two *G*s will counteract each other rendering an *NG* with higher uncertainty. To prevent the cost from being misled to a higher value, the learning rate should be decreased when θ approaches 180°.

3.3 Specifications

3.3.1 Parameters Overview

To comprehend the above concepts, we implement our technique using six parameters and two functions as shown in Table 3.3. The parameters w_{pg} and w_{cg} are weights of *PG* and *CG*, respectively; s_g controls the magnitude of the *NG* by limiting these two weights. Equation 2 (i.e., $F_{pg}(a,s_g)$) is used to compute w_{pg}; s_η is the lower bound of the learning rate η that is determined by Equation 3 (i.e., $F_\eta(a,s_\eta)$). The third column "Calculation" lists all necessary parameters required to compute the corresponding parameters and functions. For instance, w_{pg} and s_g are needed to compute w_{cg}.

Table 3.3 – Angle-based Parameters and Functions

Symbol	Explanation	Calculation	Property
a	the normalized angle of θ between PG and CG	PG, CG	dynamic
w_{pg}	the weight of PG	$F_{pg}(a,s_g)$	a-based
w_{cg}	the weight of CG	w_{pg}, s_g	a-based
s_g	the sum of $\lvert w_{pg}\rvert$ and w_{cg} the intercept point on vertical axis the slope of $F_{pg}(a,s_g)$	–	user defined, default value 1.0
$F_{pg}(a,s_g)$	the function to compute w_{pg}	a, s_g	–
s_η	the minimum of $F_\eta(a,s_\eta)$ smaller than the intercept point on vertical axis by 1	–	user defined, default value 1.0
η	the learning rate	$F_\eta(a,s_\eta)$	a-based
$F_\eta(a,s_\eta)$	the function to calculate η	a, s_η	–

$$F_{pg}(a, s_g) = s_g(1 - 2a) \tag{2}$$

$$F_\eta(a, s_\eta) = -s_\eta(a - 2) \tag{3}$$

3.3.2 Work-through

To commence AG-SGD, we initialize the two user-defined parameters s_g and s_η with the value 1.0. These initial values of s_g and s_η correspond to the two straight lines L_0 (red) in Figure 3.2, respectively (note: black lines L_1 and L_2 are definitions that correspond to a higher s_g and s_η). In each epoch, θ between PG and CG is normalized as $a \in [0, 1]$ by dividing its value by 180°. Next, the values of a and s_g are input to $F_{pg}(a,s_g)$ to compute w_{pg}, then w_{cg} is determined using $s_g - \lvert w_{pg}\rvert$. η is computed according to $F_\eta(a,s_\eta)$. Finally, NG is formed as $\eta(w_{pg}PG + w_{cg}CG)$. Here the functions $F_{pg}(a,s_g)$ and $F_\eta(a,s_\eta)$ are defined as $s_g(1-2a)$ and $-s_\eta(a-2)$, respectively. There are four different cases that arise given the various possible measurements of these components:

1) When $a < 0.5$ (i.e., $\theta < 90^{\circ}$), we have $w_{pg} > 0$ and OPG = PG. Since w_{pg} increases as a (or θ) approaches 0 (or 0˚) and w_{cg} inversely varies with w_{pg} (because $w_{cg} = s_g - |w_{pg}|$ and s_g is a fixed value), the OPG will have a greater contribution to NG generation as a (or θ) gets close to 0 (or 0˚).
2) When $a > 0.5$ (i.e., $\theta > 90^{\circ}$), we have $w_{pg} < 0$, which means that the minus sign reverses the direction of PG. Therefore, the RPG aligns more with CG and gradually dominates NG computation as a (or θ) approaches 1 (or 180˚).
3) When $a = 0.5$ (i.e., $\theta = 90^{\circ}$), the contribution of PG in calculating NG will be zero, i.e., NG is solely determined by CG because $w_{pg} = 0$.
4) The value η is monotonically decreasing in the entire range of a (or θ) to mitigate the possibilities of rendering a chaotic SGD trajectory. A larger value of a (or θ) will mislead to a higher cost value.

It is critical to understand that although the computation of NG is dominated by OPG or RPG as θ approaches 0˚ or 180˚ separately, OPG and RPG are closely aligned with CG. In this sense, AG-SGD behaves similar to Vanilla SGD when θ approaches 0˚, 180˚, and 90˚ because NG is gradually dominated by CG under these circumstances.

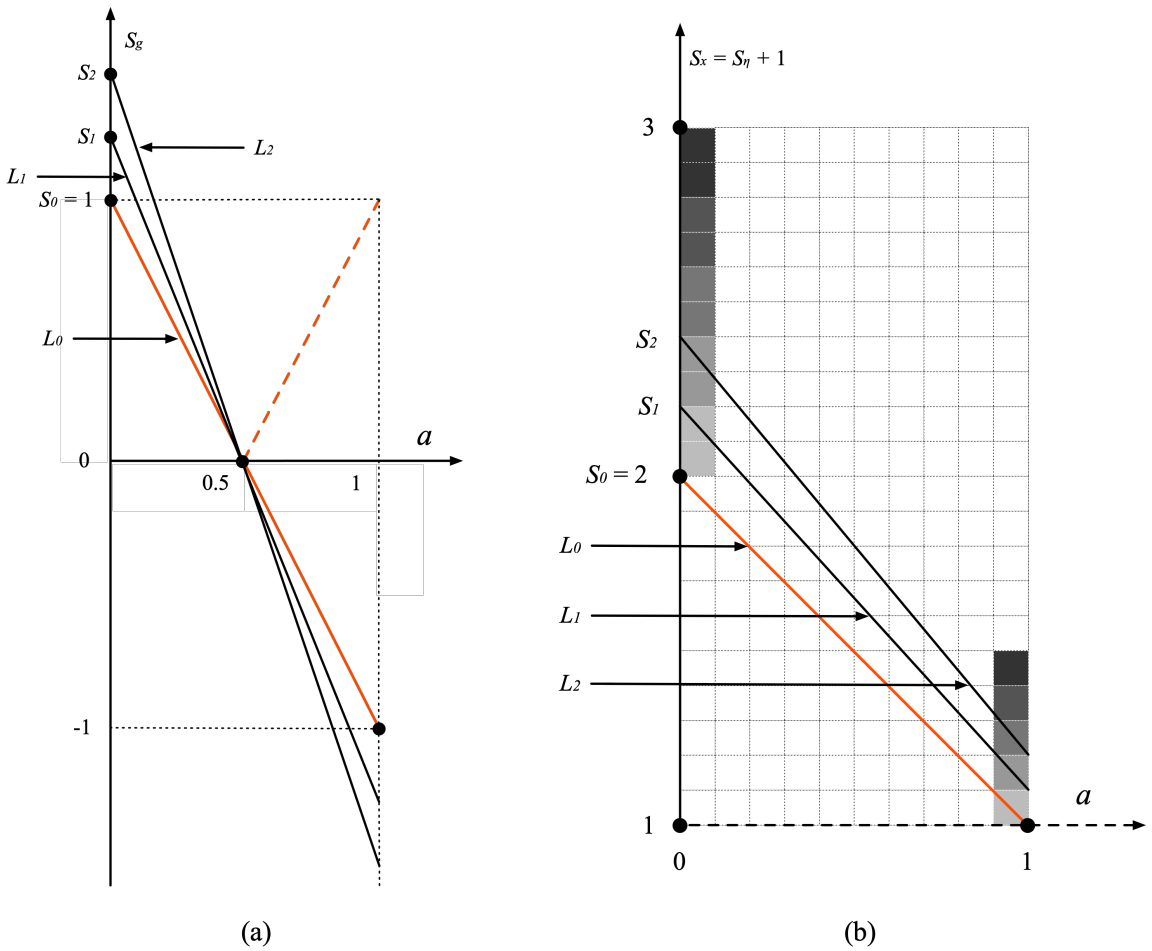


Figure 3.2 – Definitions of the functions $F_{pg}(a,s_g)$ in (a) and $F_\eta(a,s_\eta)$ in (b) (lines L_1 and L_2 are alternative definitions that supportive; the horizontal axis is the normalized angle between PG and CG; two intercept points on vertical axes s_g and s_η are parameters of the proposed method)

3.4 Awareness Ability: One-step Cost Reduction

Due to the deviation of G between the mini-batch and the full dataset, the cost might be misguided to a higher value as SGD proceeds in time. Referring to Figure 3.3, if PG is not reversed, the cost reduction would follow the blue Gs. Each step in the high cost areas is susceptible to being misguided to a higher value by the red Gs, since they are influenced by the aforementioned pattern deviation. The key in resolving this problem is to reduce the cost as soon as possible, as indicated in the figure with the A, B, and C positions. If we reverse PG prior to combining it with CG, the cost reduction would

follow the green *G*s, i.e., points 2, 5, and 7. Consequently, AG-SGD would converge faster and generate less obtuse angles than other optimizers, due to the one-step cost reduction.

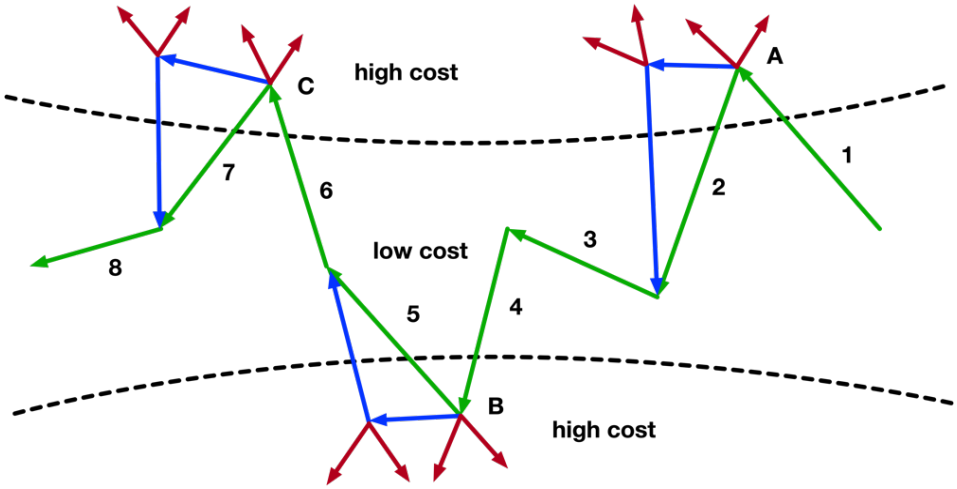


Figure 3.3 – SGD trajectory of AG-SGD: *PG* will be reversed when the cost is increased (green *G*s 2, 5, 7 are resulting from AG-SGD; blue *G*s are resulting from other optimizers, red *G*s will result in higher costs if *PG* is not reversed)

3.5 Pseudocode

The pseudocode of AG-SGD is given below.

AG-SGD

Setting values for s_g and s_η, where $s_g \geq 1$ and $s_\eta \geq 1$
From the second epoch:

1. Recording *PG*
   Computing *CG*
2. Computing θ between *PG* and *CG*
   $\theta = arccos((V_{PG} \bullet V_{CG}) / (|V_{PG}| \, |V_{CG}|))(180 / \pi)$
   Normalizing θ
   $\theta \rightarrow a \in [0,1]$
3. Computing the weights of *G*s
   $w_{pg} \leftarrow F_{pg}(a,s_g) = s_g \, (1 - 2a)$
   $w_{cg} \leftarrow s_g - |w_{pg}|$
4. Computing $\eta \leftarrow F_\eta(a,s_\eta) = -s_\eta \, (a - 2)$
5. Computing $NG \leftarrow \eta \, (w_{pg}PG + w_{cg}CG)$

until the last epoch

3.6 Summary

The newly proposed metric (i.e., a) represents the difference in direction between PG and CG, so the weight of PG should be inversely changed with a when CG is assumed to be accurate (i.e., $PG \propto a$). Also, this chapter presents the realizations of the three (of four) a-based improvements to existing measurements mentioned in the first subsection of this chapter. They are:

1) Abandoning EMA and SMA

After calibrating PG, the weight of CG is also determined. The calibrated weights are more reliable than static weights (i.e., β) adopted by EMA in improving accuracy. Due to the fact that AG-SGD sets the sum of weights to a fixed value (e.g., 1.0 under the default setting), each generated NG has a definite upper bound as EMA under all configurations, instead of infinitely increasing as in SMA.

2) Inversely adjust η based on a ($\eta \propto a$)

Due to the fact that a larger a indicates the closeness of the cost to a minimum, η is inversely adjusted with a to both accelerate the cost reduction during the early/middle stage of convergence (i.e., the cost is approaching the minimum) and enhance the cost convergence during the final stage (i.e., the cost is wandering around the minimum).

3) Prescience Ability (one-step cost reduction)

With respect to the cost reduction, an obtuse angle means the cost might be guided to a higher value and a backward movement is needed. AG-SGD realizes this goal by reversing the direction (i.e., sign) of PG when $\theta > 90^\circ$. Most importantly, even if there is a misidentification to the cost variance (i.e., the cost is not increased when $\theta > 90^\circ$), the

reversed *PG* is also (roughly) aligning with the reliable *CG*, combining to form an accurate *NG*.

CHAPTER 4

IN-DEPTH INTERPRETATION OF ANGLE-BASED STOCHASTIC GRADIENT DESCENT

4.1 Introduction

This chapter reveals the convergence principle behind AG-SGD by demonstrating the *NG* variance that is determined by the two parameters s_g and s_η. Then, the independent functionalities realized by parameters and the associated tuning strategies in various cases are presented. Also, the actual behavior of AG-SGD will be analyzed in a practical context. Finally, all improvements realized by AG-SGD are summarized.

4.2 Variance Pattern of New Gradient

The *NG* magnitude is determined by combining the two functions $F_{pg}(a,s_g)$ and $F_\eta(a,s_\eta)$. Figure 4.1 shows the variance pattern between θ and the corresponding *NG* magnitude when *CG* and *PG* are 1.0 unit in magnitude. For example, the *NG* magnitude will be larger/smaller than *CG* and *PG* when the angle is in [0˚, 120˚)/(120˚, 180˚) and will not be changed when the angle is 120˚ or 180˚ under the default setting (the green line). Furthermore, the *NG* roughly decreases as the angle increases, which stabilizes SGD's trajectory. Since w_{pg} is close to 0 as θ approaches 90˚, the most reliable *CG* gradually dominates *NG* computation, producing a peak at $\theta = 90$˚. When θ is close to 0˚ or 180˚, *CG* becomes better aligned with OPG or RPG, rendering a larger w_{pg} and forms a larger *NG*. The two troughs are achieved when OPG and RPG have larger difference from *CG* in determining an accurate *NG*, so these two *G*s are involved in the *NG*

computation with greater comparable weights. Consequently, the smaller *NG*s are attributed to the weight assignment between w_{pg} and w_{cg}.

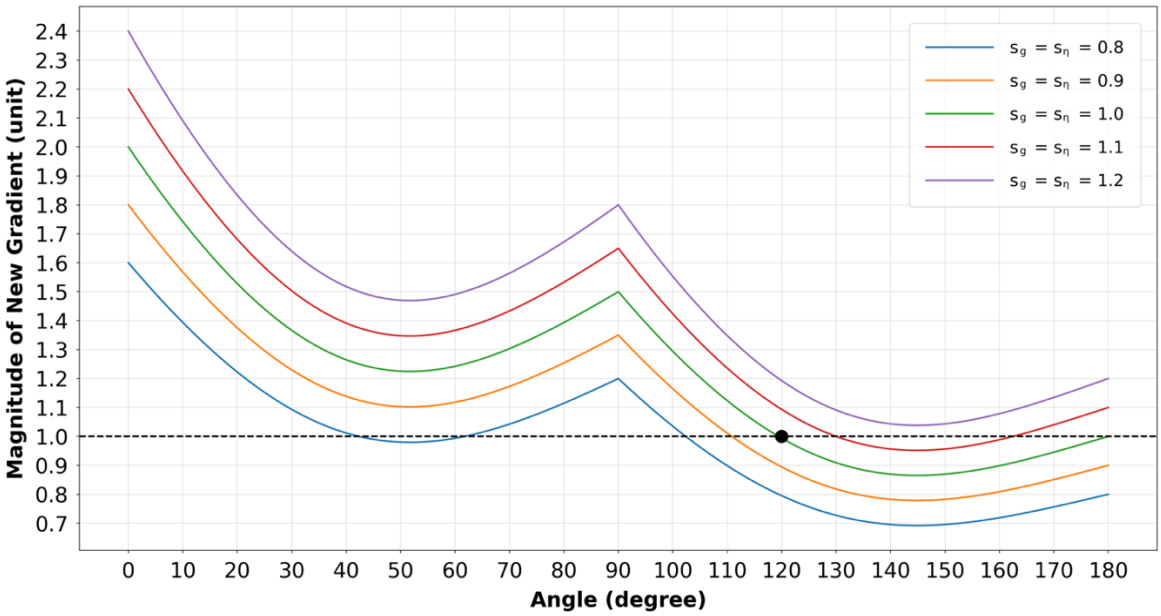


Figure 4.1 – Magnitude variance of *NG* with the changes of s_g and s_η
(the black dot indicates NG magnitude which is reduced when the angle is larger than 120˚ under the default setting)

4.3 Convergence Guarantee

In the early and middle stages of SGD, the cost is repeatedly reduced until it approaches to one of the minima (phase 1). Once the cost is oscillating within a small range and cannot be reduced further, it means that the result has converged (phase 2). One of the distinct differences between these two phases is that the averaged θ between consecutive *G*s of the phase 2 is larger than that of phase 1. As shown in Figure 4.2, the main reason that causes this phenomenon is that more obtuse angles are generated with the cost and repeatedly overshoot/wander around the minimum during the final converging stage. In this case, if we reduce the step length every time an obtuse angle is generated, the step length would gradually approach 0 and result in a lower converged cost.

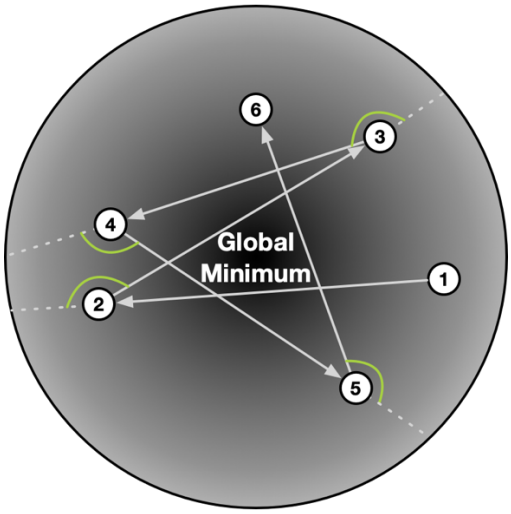


Figure 4.2 – Cost variance trajectories in the final SGD stage
(a darker color indicates a lower cost and more obtuse angles would be generated during the cost wandering around the minimum)

This convergence principle is utilized and employed by AG-SGD, as shown in Figure 4.1. Under its default configuration (i.e., $s_g = s_\eta = 1.0$), *NG* magnitude is smaller than 1.0 unit when θ is in (120˚, 180˚), so the step length will be reduced when θ is in this section. A section (e.g., (120˚, 180˚)) that causes a reduction in *NG* magnitude and fosters a cost convergence is called Convergence Section (CS). If this default CS is still narrow and renders a non-convergence failure, we could enlarge the CS by decreasing s_g or s_η to achieve a stronger convergence.

4.4 Decoupled Parameters: Gradient Weight and Learning Rate

Figures 4.3 and 4.4 show the variance pattern between the angle and the corresponding *NG* magnitude when the two parameters (i.e., s_η and s_g) are varying in [0.8, 1.2]. These two parameters are mutually complementary in their functionalities, where their differences are:

1) s_g is mainly used to change the *NG* magnitude when θ approaches 0˚ or 180˚, as shown in Figure 4.3. Adjusting s_g will primarily modify the upper bound of the CS and the associated *NG* strength reduction.

2) s_η is primarily used to adjust the *NG* magnitude when θ is close to 90˚, as shown in Figure 4.4. Changing s_η will mainly adjust the lower bound of the CS and the related *NG* strength reduction.

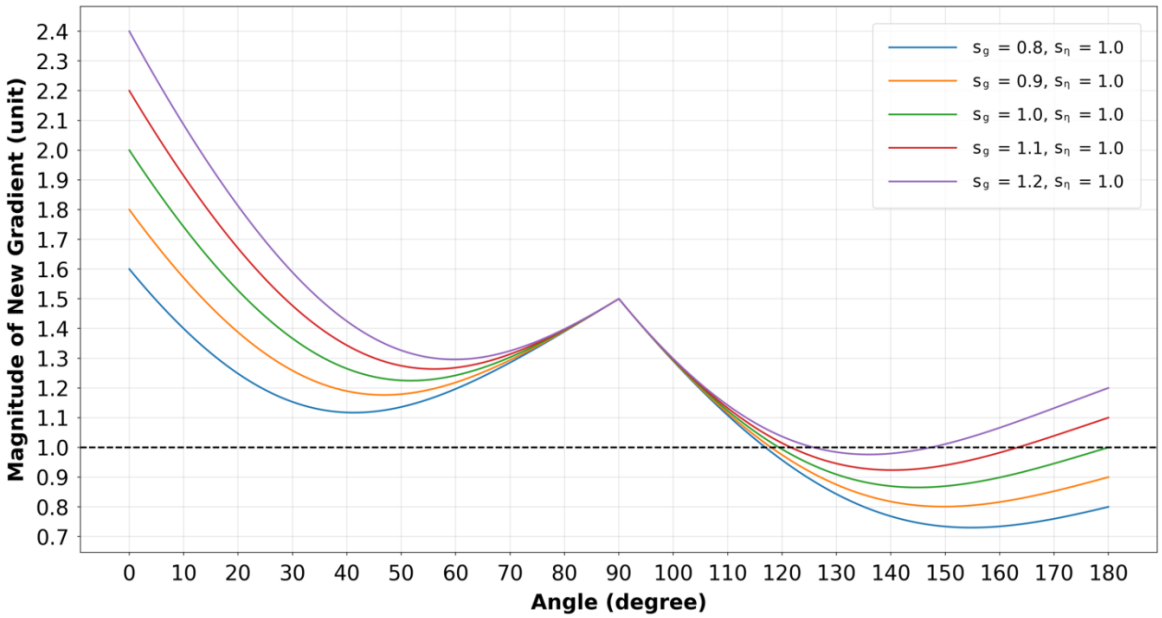


Figure 4.3 – Magnitude variance of *NG* with the changes of s_g (affecting the NG magnitude when the angle is close to 90˚)

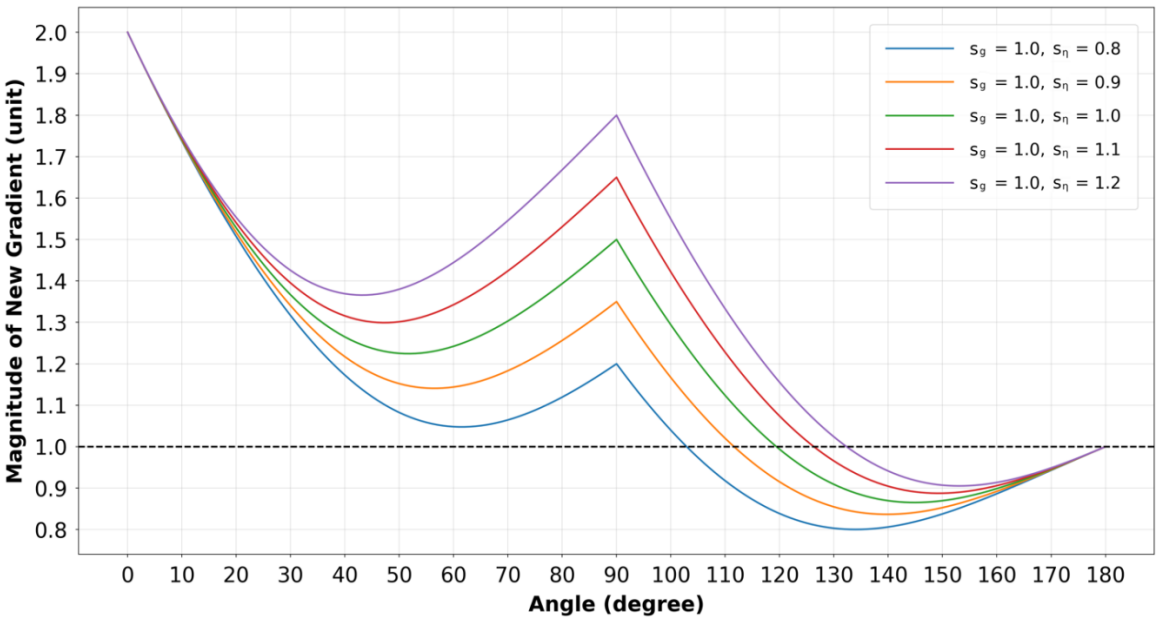


Figure 4.4 – Magnitude variance of *NG* with the changes of s_η
(affecting the NG magnitude when the angle is close to 0˚ and 180˚)

4.5 Configuring Strategy

It is evident that adjusting these two parameters together will obtain all the aforementioned changes at once. However, if the s_g and s_η are changed by the same magnitude, they should be adjusted within the recommended configuration section (0.820, 1.156), due to:

1) When $s_g = s_\eta \leq 0.820$, the average of *NG* for the entire section of θ is less than 0.9996, which is averaged from the 181 *NG*s of all integral angles in [0˚, 180˚]. In this case, SGD will not be accelerated, violating the goal of employing optimizers.
2) When $s_g = s_\eta \geq 1.156$, the CS is an empty set. It means that a non-convergence failure is inevitable, as *NG* will not be reduced in the entire section of θ, which is verified by the experiment in Chapter 6.

It needs to be mentioned that this recommended configuration section applies to datasets in any problem field, as they are solely determined by the characteristic of AG-SGD. Since the average of this section is 0.988 (i.e., (0.820 + 1.156) / 2), the default values of s_g and s_η are set to 1.0 for achieving an accelerated SGD that mitigates the non-convergence failure. In addition, the averaged *NG* magnitude is 1.219 under the default configuration, so it is a conservative setting that mitigates overshooting, but it adversely impacts the SGD speed. Therefore, increasing the default values appropriately may speed-up SGD without missing the optimal result in practice. Moreover, $F_\eta(a,s_\eta)$ is also designed with the consideration of mitigating the non-convergence failure. Referring to Figure 3.2 (b), η will be increased by a smaller magnitude with a approaching 1. For example, if we increase η (a = 0) by 2 units (i.e., from s_1 to s_2), η (a = 1) is increased by 1 unit. It is important to mention that the variance pattern of the *NG* magnitude will not be changed when both s_g and s_η change with the same magnitude, as shown in Figure 4.1. This property is advantageous and utilized during the early and middle tuning stages to foster faster configuration searching. For example, if we are not satisfied with the lowest achieved cost, we should decrease the step lengths in the entire section of θ by reducing both s_g and s_η. To make a fine adjustment, we need to separately change the values of s_g and s_η. For instance, if the cost is still randomly oscillating in a large range after many epochs, it indicates the cost is approaching a minimum with a large step length. Since more obtuse angles are generated during this process, we should first try to decrease s_g to reduce the step lengths when θ is close to 180˚. Due to the fact that decreasing s_g will also reduce the step lengths when θ is close to 0˚, a slower overall convergence speed is the

side-effect of this manipulation. Consequently, there is a trade-off between the convergence speed and overshooting [4.1].

4.6 Summary

Referring to Figure 4.5, AG-SGD is different from all analyzed optimizers (i.e., solely using *PG* as the metric) in its adoption of the new metric a (or θ). With respect to practical behavior, AG-SGD can be considered as a special Vanilla that employs *PG* only when it is reliable. Also, AG-SGD is unique in the following two aspects (under the default setting):

1) Possessing a definite range for an effective parameter tuning (i.e., s_g and $s_\eta \in (0.820, 1.156)$)
2) Possessing a definite condition for the cost convergence (i.e., $\theta \in (120^\circ, 180^\circ)$)

These two advantages are combined to give users a direct feedback to the cost movements that result from their adjustments to parameters, reducing parameter searching space and efforts on finding the optimal configuration. All improvements that are realized by AG-SGD and the associated explanations are summarized in Table 4.1.

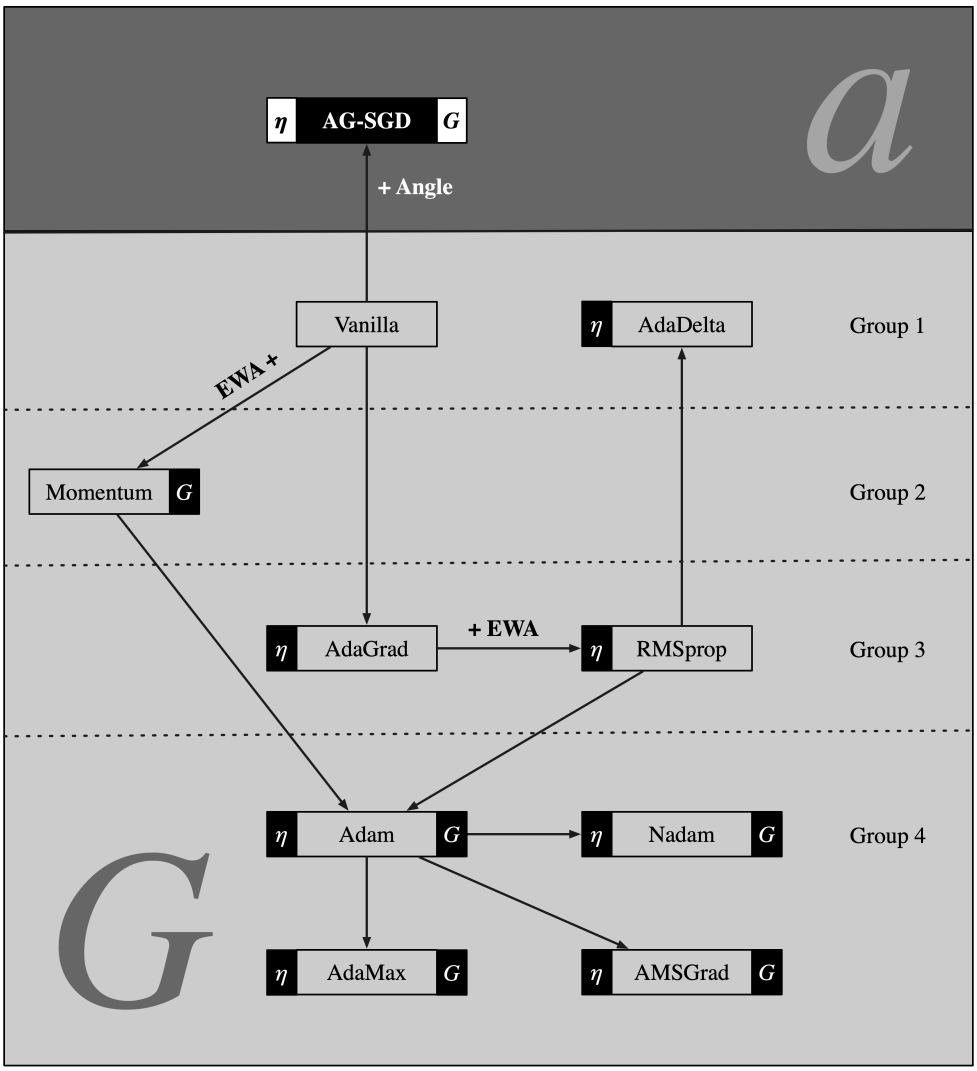


Figure 4.5 – Differences between AG-SGD and other optimizers

Table 4.1 – Improvements realized by AG-SGD

Improvement	**Explanation**
New Metric *a*	the angle between *PG* and *CG*
Calibrating *PG*s (|*PG*s| ∝ *a*)	*a* positively correlates with the deviation on *PG*
***a*-based *G*s**	weights of *PG* and *CG* are determined by *a*
***a*-based *η* (∝ *a*)**	early/middle stage: acceleration (larger η) final stage: stronger convergence (smaller η)
One-step Cost Reduction	reducing the cost by one *NG* after it is increased
Decoupled Parameters	s_g: control *NG* when *a* approaches 0 or 1 s_η: control *NG* when *a* approaches 0.5

CHAPTER 5

EVALUATIONS OF ACCURACY AND EFFENCIENCY

5.1 Introduction

This chapter compares (1) the cost reduction, (2) the ability of translating the reduction in cost to error rate, and (3) time complexities among different optimizers. Group-based results are also calculated for an in-depth analysis. To avoid any bias, in comparing the proposed technique with other optimizers, two experiments are conducted with the following characteristics: (1) machine learning algorithms, (2) cost functions, (3) batch sizes, and (4) datasets associated with different fields. In the first experiment, AG-SGD is implemented with a fully-connected vanilla neural network and evaluated on the handwritten digits dataset MNIST [5.1]. The second experiment employs a logistic regression classifier to evaluate AG-SGD on a network-based intrusion detection dataset NSL-KDD [5.2]. In terms of comparison, 10 different SGD optimizers: (1) Vanilla SGD, (2) Momentum, (3) RMSprop, (4) Adam, (5) Nadam, (6) AdaMax, (7) AdaDelta, (8) AdaGrad, (9) AMSGrad, and (10) AG-SGD are evaluated under the same conditions in both experiments.

5.2 Neural Network on Digital Recognition

5.2.1 Scheme

The fully-connected vanilla neural network has 4 layers. Each layer employs Sigmoid [1.16] as the activation function. The numbers of neurons from the input to the output layers are 784, 256, 112, and 10, respectively. The output cost is quantified using

the MSE [1.12]. For the usage of the dataset MNIST, we employ the pre-split 60,000 samples for training and 10,000 samples for testing. To evaluate AG-SGD under the intended application scenario (refer to Chapter 6), the batch size and number of epochs are set to 8 and 50 (note: we have found that larger values for these quantities do not reduce the testing costs further for all compared optimizers). As a result, there are 375,000 gradients (i.e., 60,000 / 8 × 50) to be generated during SGD to minimize the output cost.

A good optimization result could occur accidentally, when the data are well-matched among the different techniques and they could generate skewed positive cost reduction for the proposed method [5.3]. To avoid this possibility, the classifiers evaluated will not employ any additional technique such as dropout [5.4], weight decay [5.5], learning rate decay [5.6]. Therefore, the difference in the cost reduction should be attributed solely to the adopted optimizer. In addition, instead of evaluating the best result, the best 5 results will be selected to represent the performance of each optimizer. As a reference, the error rate of a fully-connected vanilla neural network on the dataset MNIST is about 2% based on the Kaggle leaderboard scores [5.7].

5.2.2 Results

Figures 5.1 and 5.2 show the epoch-based average of the 5 minimal costs and the associated error rates from the above-mentioned SGD optimizers, respectively. Each value in the figures is computed by averaging the corresponding 5 values on the same epoch. The 10 optimizers are rated into 4 levels based on the results. In level-1, Adam and Nadam are outperformed by all the other optimizers with relative significant

magnitudes in cost and error reductions. Also, they are unstable as shown with oscillated curves. Particularly, Nadam is considered the worst optimizer in performance due to the increasing cost beyond epoch no.40, which translates to a non-convergence failure. In level-2, AdaMax, AMSGrad, and RMSprop perform better in the cost/error rate reduction and stability than the optimizers in the previous level. In addition, AdaMax converges faster than the rest of the optimizers in the same group during the first 14 epochs. In level-3, the Vanilla, Momentum, AdaDelta, and AdaGrad are more stable than the aforementioned optimizers because their curves are fluctuating within smaller ranges. These optimizers achieve the lowest costs and error rates compared with all of the optimizers analyzed, which aligns with the results in [5.3]. In level-4, AG-SGD outperforms all of the other optimizers in terms of the cost and error rate reductions. Although its converging speed is slower than that of AdaMax in the first 14 epochs, AG-SGD greatly increases its speed between epochs 14 to 19 and obtains a cost as low as the best cost achieved by the others on epoch no.18. This advantage in cost is maintained until the last epoch, which occupies 66% of the training time (i.e., (50 – 18 + 1) / 50). It can be expected that the error rate could be further reduced along with the elongation of the training time. Whereas, if the cost of AG-SGD is oscillating as the other optimizers, it would only end-up with a local optimal error rate.

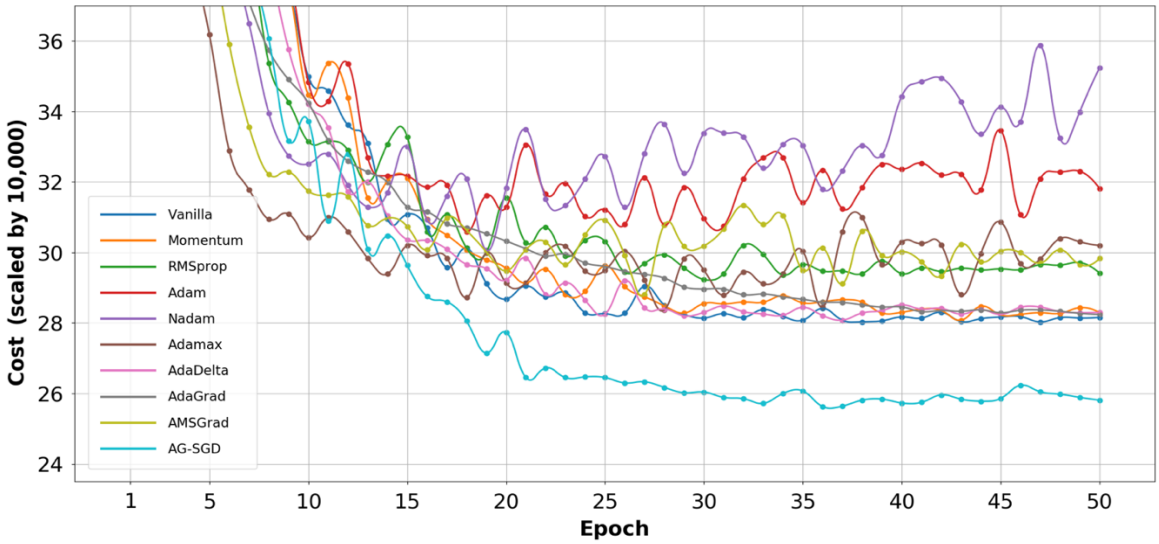


Figure 5.1 – Epoch-based average of the 5 minimal costs
(zoom-in, each dot represents the cost of specific optimizer on the corresponding epoch)

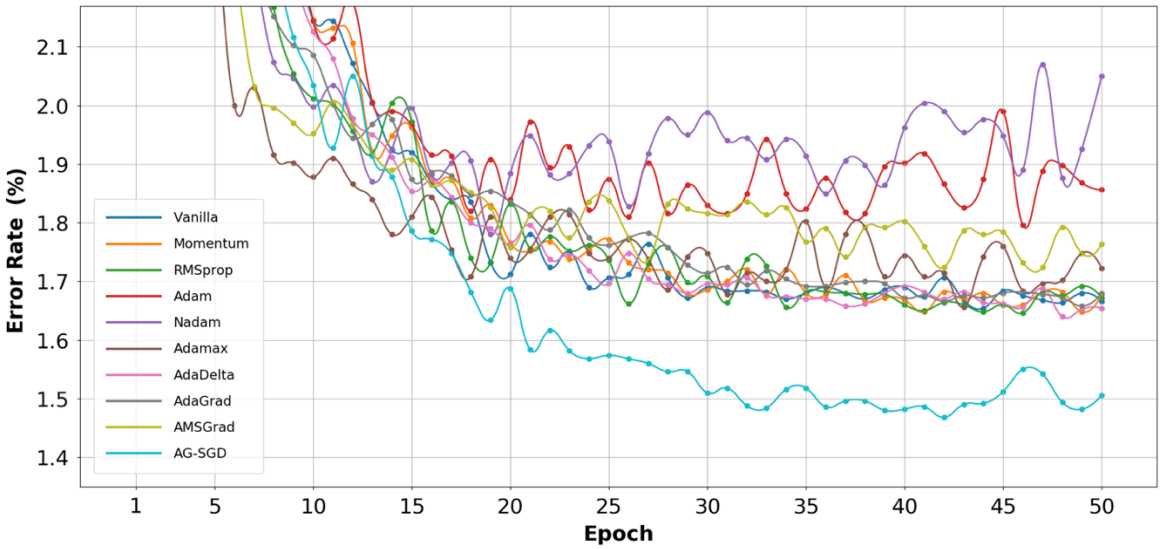


Figure 5.2 – Epoch-based average of the 5 minimal error rates
(zoom-in, each dot represents the error rate of specific optimizer on the corresponding epoch)

The above observations can be verified by the distributions of the best 5 results in Figures 5.3 and 5.4, in which the data dots are the outliers of the corresponding results. The optimizers are rated into 3 levels based on the existence of outliers and the range of values (i.e., the length of boxes). In level-1, there are 6 optimizers, Vanilla, Momentum, RMSprop, Adam, AdaGrad, and AMSGrad, and all have at least one outlier in both

figures. Although the ranges of the non-outlaid values are more concentrated, it means that these optimizers would generate local minimal results in most cases. Employing them in practice would miss the optimal result by a relative higher probability. In level-2, we have 2 optimizers, Nadam and AdaDelta, due to their scattered results. Although their results do not have an outlier, the larger value ranges would make their results relative harder to predict compared with the optimizers in level-1. Since the optimizers in the first two levels have distinct characteristics in their resulting distributions, it is difficult to judge which level would perform better in practice. In level-3, AG-SGD and AdaMax both have no outliers, and are more concentrated than the other two levels. The distinct difference between these two optimizers is that the former could generate a lower cost and error rate than the latter.

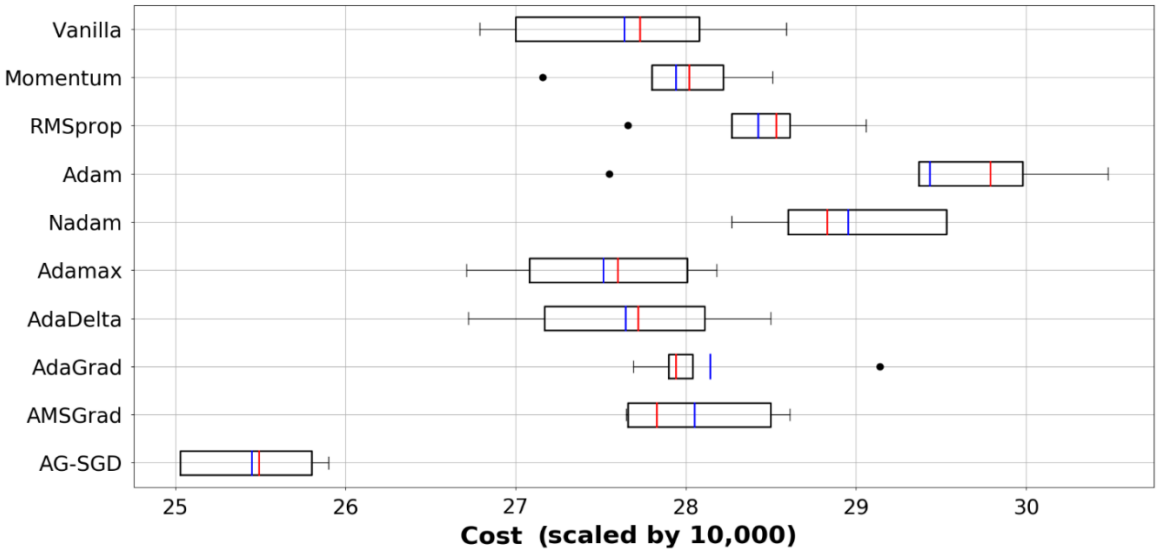


Figure 5.3 – Distributions of the 5 minimal costs (MNIST, black dots are outliers)

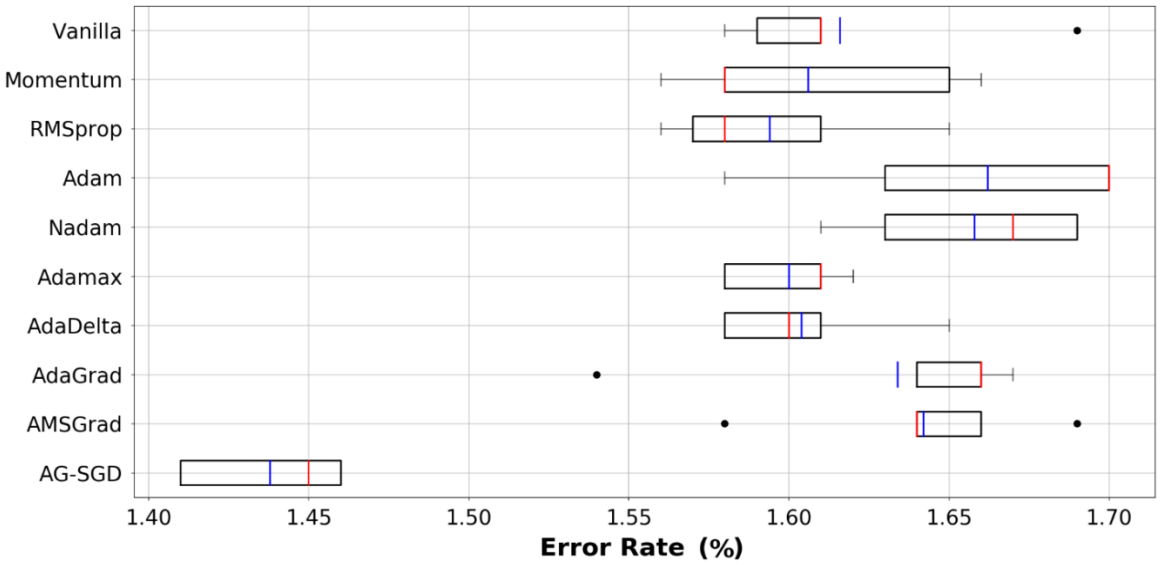


Figure 5.4 – Distributions of the 5 minimal error rates (MNIST, black dots are outliers)

Tables 5.1 and 5.2 show the comparisons of the 5 minimal costs and the corresponding error rates from the 10 SGD optimizers. The data show that AG-SGD is the most accurate optimizer with the best stability. If we quantify the advantages by averaging the minimal costs (i.e., 27.36) and the variances (i.e., 0.4564) from all the other optimizers, AG-SGD has a better performance by 8.52% = (27.36 – 25.03) / 27.36 in the cost reduction and 62.80% = (0.4564 – 0.1698) / 0.4564 in stability.

Table 5.1 – Comparison of the 5 minimal costs (MNIST)

GP	Optimizer	The 5 Minimal Costs scaled by 10,000 listed from the highest to the lowest					Each			Group			Overall		
							Avg	Std	Var	Avg	Std	Var	Avg	Std	Var
1	Vanilla	28.59	28.08	27.73	27.00	26.79	27.64	0.7477	0.5591	**27.64**	0.7304	0.5338	**28.09**	**0.6193**	**0.4091**
	AdaDelta	28.50	28.11	27.72	27.17	26.72	27.64	0.7131	0.5085						
2	Momentum	28.51	28.22	28.02	27.80	27.16	27.94	0.5093	0.2594	**27.94**	0.5093	0.2594			
3	AdaGrad	29.14	28.04	27.94	27.90	27.69	28.14	0.5723	0.3275	**28.29**	0.5433	0.2960			
	RMSprop	29.06	28.61	28.53	28.27	27.66	28.43	0.5142	0.2644						
4	Adam	30.48	29.98	29.79	29.37	27.55	29.43	1.1261	1.2681	**28.49**	0.6942	0.5471			
	AdaMax	28.18	28.01	27.60	27.08	26.71	27.52	0.6188	0.3829						
	Nadam	29.53	29.53	28.83	28.60	28.27	28.95	0.5639	0.3180						
	AMSGrad	28.61	28.50	27.83	27.66	27.65	28.05	0.4681	0.2192						
–	**AG-SGD**	**25.90**	**25.80**	**25.49**	**25.03**	**25.03**	**25.45**	**0.4121**	**0.1698**	**25.45**	**0.4121**	**0.1698**	**25.45**	**0.4121**	**0.1698**

Table 5.2 – Comparison of the 5 minimal error rates (MNIST)

GP	Optimizer	The 5 Minimal Error Rates percentages listed from the highest to the lowest					Each			Group			Overall		
							Avg	Std	Var	Avg	Std	Var	Avg	Std	Var
1	Vanilla	1.69	1.61	1.61	1.59	1.58	1.62	0.0434	0.0019	**1.61**	0.0361	0.0014	**1.62**	**0.0411**	**0.0018**
	AdaDelta	1.65	1.61	1.60	1.58	1.58	1.60	0.0288	0.0008						
2	Momentum	1.66	1.65	1.58	1.58	1.56	1.61	0.0456	0.0021	**1.61**	0.0456	0.0021			
3	AdaGrad	1.67	1.66	1.66	1.64	1.54	1.63	0.0537	0.0029	**1.61**	0.0451	0.0021			
	RMSprop	1.65	1.61	1.58	1.57	1.56	1.59	0.0365	0.0013						
4	Adam	1.70	1.70	1.70	1.63	1.58	1.66	0.0550	0.0030	**1.64**	0.037	0.0016			
	AdaMax	1.62	1.61	1.61	1.58	1.58	1.60	0.0187	0.0004						
	Nadam	1.69	1.69	1.67	1.63	1.61	1.66	0.0363	0.0013						
	AMSGrad	1.69	1.66	1.64	1.64	1.58	1.64	0.0402	0.0016						
–	**AG-SGD**	**1.46**	**1.46**	**1.45**	**1.41**	**1.41**	**1.44**	**0.0259**	**0.0007**	**1.44**	**0.0259**	**0.0007**	**1.44**	**0.0259**	**0.0007**

5.3 Logistic Regression on Network-based Intrusion Detection

There are 41 features and 2 classes (i.e., normal and anomaly) in the dataset NSL-KDD. We select all 25192 instances in the file "KDDTrain+_20Percent" as the training samples and all 22544 instances in the file "KDDTest+" as the testing samples. To employ the logistic regression classifier on the data, we apply the Principle Component Analysis (PCA) [5.8] to convert all data to numeric. As a result, PCA generates 85 features in the training and the testing sets. With respect to the configuration of the logistic regression classifier, there are 4 differences compared with the fully-connected vanilla neural network. These are: (1) the numbers of input and output nodes are reduced to 85 and 2, respectively; (2) the cost function is changed to Cross-Entropy [1.13] which is more suitable for quantifying the output cost of binary classification problems than the MSE; (3) the number of epochs is limited to 30 due to the reduction in samples and complexity of classifier; (4) the batch size is set to 1 to better reveal the difference in obtaining the optimal results among the 10 optimizers. Although a lower batch size will have a higher probability of making the SGD trajectory more chaotic and mislead the output cost to a higher value, a robust optimizer should result in a good result even though its batch size is 1. As a result, there are 755,760 gradients (i.e., (25,192 / 1) × 30) that will be generated to minimize the output cost in each experiment. For comparison, the lowest testing error rates achieved by the three neural networks [5.9–5.11] are 20.7%, 19.87%, and 16.69%, respectively. The results of this experiment are shown in Figures 5.5 and 5.6, and Tables 5.3 and 5.4.

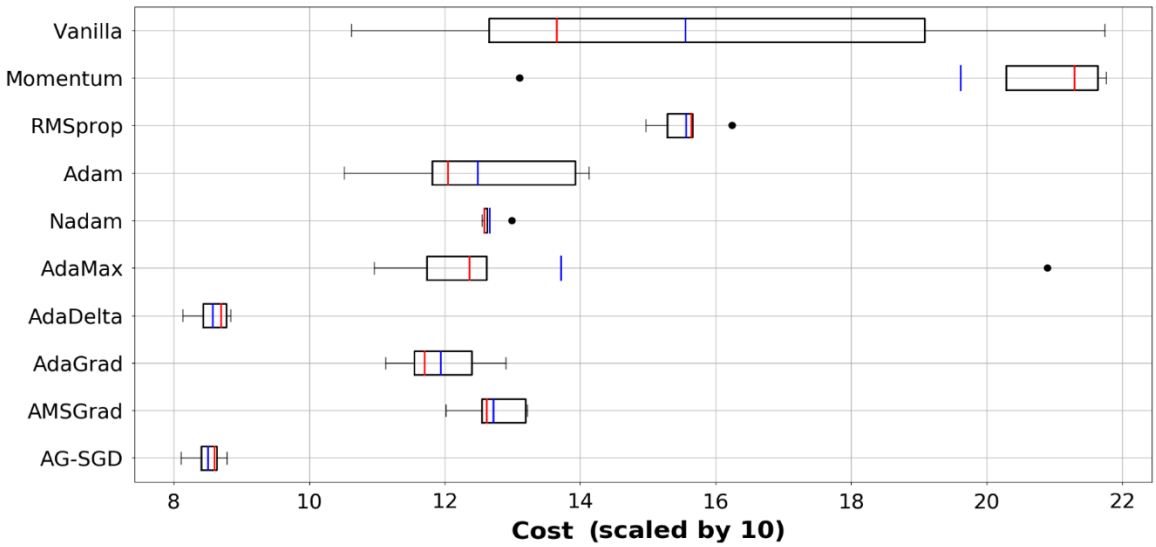


Figure 5.5 – Distributions of the 5 minimal costs
(NSL-KDD, black dots are outliers)

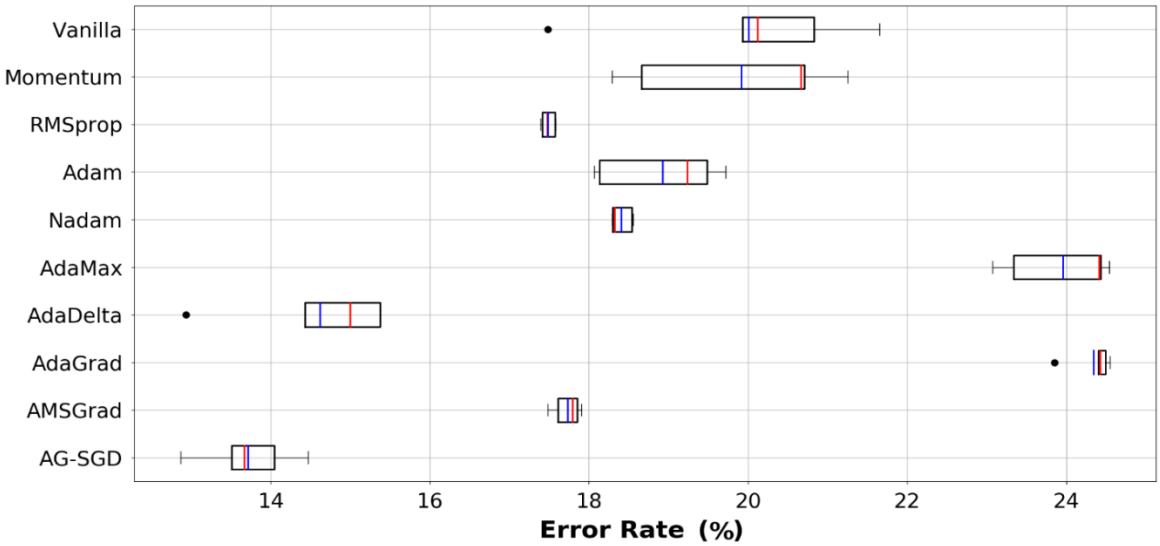


Figure 5.6 – Distributions of the 5 minimal error rates
(NSL-KDD, black dots are outliers)

Table 5.3 – Comparison of the 5 minimal costs (NSL-KDD)

GP	Optimizer	The 5 Minimal Costs scaled by 10 listed from the highest to the lowest					Each			Group			Overall		
							Avg	Std	Var	Avg	Std	Var	Avg	Std	Var
1	Vanilla	21.74	19.09	13.65	12.65	10.62	15.55	4.6675	21.7855	**12.06**	2.4803	10.9357	**14.58**	**2.0815**	**7.4158**
	AdaDelta	8.84	8.77	8.70	8.43	8.13	8.57	0.2930	0.0858						
2	Momentum	21.76	21.64	21.29	20.28	13.11	19.62	3.6856	13.5835	**19.62**	3.6856	13.5835			
3	AdaGrad	12.90	12.40	11.70	11.55	11.13	11.93	0.7066	0.4992	**13.75**	0.5907	0.3624			
	RMSprop	16.24	15.66	15.64	15.29	14.96	15.56	0.4748	0.2255						
4	Adam	14.13	13.92	12.05	11.82	10.51	12.49	1.5250	2.3255	**12.90**	1.5693	4.7816			
	AdaMax	20.89	12.62	12.36	11.73	10.96	13.71	4.0637	16.5134						
	Nadam	12.99	12.62	12.58	12.58	12.55	12.67	0.1849	0.0342						
	AMSGrad	13.22	13.19	12.62	12.55	12.02	12.72	0.5034	0.2534						
–	**AG-SGD**	**8.78**	**8.64**	**8.60**	**8.40**	**8.11**	**8.51**	**0.2601**	**0.0677**	**8.51**	**0.2601**	**0.0677**	**8.51**	**0.2601**	**0.0677**

Table 5.4 – Comparison of the 5 minimal error rates (NSL-KDD)

GP	Optimizer	The 5 Minimal Error Rates percentages listed from the highest to the lowest					Each			Group			Overall		
							Avg	Std	Var	Avg	Std	Var	Avg	Std	Var
1	Vanilla	21.66	20.83	20.12	19.93	17.48	20.00	1.5664	2.4535	**17.31**	1.2912	1.7428	**19.47**	**0.8160**	**0.9676**
	AdaDelta	15.37	15.37	15.00	14.43	12.94	14.62	1.0159	1.0320						
2	Momentum	21.25	20.71	20.67	18.66	18.29	19.92	1.3416	1.8000	**19.92**	1.3416	1.8000			
3	AdaGrad	24.55	24.49	24.43	24.40	23.85	24.34	0.2821	0.0796	**20.91**	0.1852	0.0437			
	RMSprop	17.58	17.57	17.47	17.41	17.39	17.48	0.0882	0.0078						
4	Adam	19.72	19.48	19.23	18.13	18.07	18.93	0.7740	0.5990	**19.76**	0.4460	0.2838			
	AdaMax	24.54	24.44	24.41	23.34	23.07	23.96	0.6975	0.4865						
	Nadam	18.56	18.54	18.32	18.30	18.29	18.40	0.1357	0.0184						
	AMSGrad	17.90	17.85	17.80	17.61	17.48	17.73	0.1768	0.0313						
–	**AG-SGD**	**14.46**	**14.05**	**13.67**	**13.51**	**12.87**	**13.71**	**0.5969**	**0.3563**	**13.71**	**0.5969**	**0.3563**	**13.71**	**0.5969**	**0.3563**

5.4 Translation Rate

The reduction in cost may not be translated to a reduction in error rate, especially when the former is obtained from non-critical output neurons. Assume an ANN with 10 output neurons is employed to identify a number (i.e., 0~9) and the prediction is determined by the output neuron with the highest activation value. If the correct number is 0 but the activation of the first output neuron (for predicting 0) is lower than the second one (for predicting 1), reducing the overall cost from the output neurons 3 to 10 will not change the wrong classification. Whereas, the accuracy could be improved only when the cost reduction is achieved by increasing/decreasing the activation of the first/second output neuron. Therefore, there is a difference in the ability of translating the reduction in cost to the error rate among optimizers. Because there is no method to quantify this difference, we name it as Translation Rate (TR) and calculate it by the following steps:

1) Obtaining multiple costs/error rates and calculating their averages (e.g., 5 or more) of each compared optimizer.
2) Computing the largest differences of all costs (DC_{all}) and error rates (DE_{all}) of all optimizers.
3) With respect to each optimizer, calculating the difference (DC_{each}/DE_{each}) between its averaged cost/error rate and the highest cost/error rate of all optimizers.
4) Scoring the abilities in the cost (S_{cost}) and the error rate (S_{error}) reductions of each optimizer by using DC_{each} / DC_{all} and DE_{each} / DE_{all}, respectively.
5) Determining TR of an optimizer by S_{error} / S_{cost}.

For instance, DC_{all} and DE_{all} in the second experiment (NSL-KDD) are 13.65 (i.e., 21.76 – 8.11) and 11.68 (i.e., 24.55 – 12.87), respectively. With respect to Vanilla, its DC_{each} and DE_{each} are 6.21 (i.e., 21.76 – 15.55) and 4.55 (i.e., 24.55 – 20.00), respectively. Then, the corresponding S_{cost} and S_{error} of Vanilla are 0.4549 (i.e., 6.21 / 13.65) and 0.3896 (i.e., 4.55 / 11.68). Finally, TR of Vanilla is 0.8563 (i.e., 0.3896 / 0.4549). Due to the fact that the four variables (i.e., DC_{all}, DE_{all}, DC_{each} and DC_{each}) used to compute TR are associated with other evaluated optimizers, TR of each optimizer is not an absolute value but a relative one for comparing with others. As a result, if TRs of two optimizers are 1.0 and 1.2, the latter is stronger than the former by 20% (i.e., (1.2 – 1.0) / 1.0 = 0.2 / 1.0) in its ability of translating the cost reduction to the error rate on specific datasets.

Tables 5.5 and 5.6 list TRs achieved on datasets MNIST and NSL-KDD for all optimizers. The optimizers with lower TRs indicate that their losses in accuracy may not be caused by an inefficacy of reducing costs, but its incapacity of translating the advantages in costs to the error rates. Because the cost is quantified by a specific cost function, a low TR might be caused by a mis-matching between the employed optimizer and the cost function. In this sense, an optimizer with a lower TR means that it has less options in selecting cost functions. This indicates that the optimizer has a narrower applicability in practice, as each cost function provides its own unique advantages on specific problems. For example, we could infer that Momentum may not match well with the Sigmoid function according to its low TR = 0.6659 in Table 5.5. Whereas, Momentum may be extremely-well suited for working with the Cross-Entropy based on its high TR = 2.5285 in Table 5.6. These results indicate that changing the cost function from the Sigmoid to Cross-Entropy may improve the accuracy on MNIST when

Momentum is employed. Referring to column “Each” under “Diff” in Table 5.7, AG-SGD and Adam present much better stability in TR than all other optimizers. However, two TRs of Adam are not comparable with AG-SGD, so the latter is the best optimizer in TR from an overall perspective.

Table 5.5 – Translation rates (MNIST, Sigmoid)

GP	Optimizer	DC_{all}	DC_{each}	S_{cost}	DE_{all}	DE_{each}	S_{error}	TR		
								Each	Group	Overall
1	Vanilla	5.45	2.84	0.5211	0.29	0.08	0.2759	**0.5294**	**0.5956**	**0.6503**
	AdaDelta		2.84	0.5211		0.10	0.3448	**0.6617**		
2	Momentum		2.54	0.4661		0.09	0.3103	**0.6659**	**0.6659**	
3	AdaGrad		2.34	0.4661		0.07	0.2414	**0.5179**	**0.7632**	
	RMSprop		2.05	0.3761		0.11	0.3793	**1.0084**		
4	Adam		1.05	0.1927		0.04	0.1379	**0.7159**	**0.5765**	
	AdaMax		2.96	0.5431		0.10	0.3448	**0.6349**		
	Nadam		1.53	0.2807		0.04	0.1379	**0.4913**		
	AMSGrad		2.43	0.4459		0.06	0.2069	**0.4640**		
–	**AG-SGD**		5.03	0.9229		0.26	0.8966	**0.9714**	**0.9714**	**0.9714**

Table 5.6 – Translation rates (NSL-KDD, Cross-Entropy)

GP	Optimizer	DC_{all}	DC_{each}	S_{cost}	DE_{all}	DE_{each}	S_{error}	TR		
								Each	Group	Overall
1	Vanilla	13.65	6.21	0.4549	11.68	4.55	0.3896	**0.8563**	**0.8680**	**1.1842**
	AdaDelta		13.19	0.9663		9.93	0.8502	**0.8798**		
2	Momentum		2.14	0.1568		4.63	0.3964	**2.5285**	**2.5285**	
3	AdaGrad		9.83	0.1568		0.21	0.0180	**0.1147**	**0.7237**	
	RMSprop		6.2	0.4542		7.07	0.6053	**1.3327**		

4	Adam		9.27	0.6791		5.62	0.4812	**0.7085**	**0.6166**	
	AdaMax		8.05	0.5897		0.59	0.0505	**0.0857**		
	Nadam		9.09	0.6659		6.15	0.5265	**0.7907**		
	AMSGrad		9.04	0.6623		6.82	0.5839	**0.8817**		
–	**AG-SGD**		13.25	0.9707		10.84	0.9281	**0.9561**	**0.9561**	**0.9561**

Table 5.7 – Translation rates in varied categories

GP	**Optimizer**	**TR**							
		MNIST	**NSL-KDD**	**Diff**			**Avg**		
				Each	**Group**	**Overall**	**Each**	**Group**	**Overall**
1	Vanilla	0.5294	0.8563	0.3269	0.2725		0.6928	0.7318	
	AdaDelta	0.6617	0.8798	0.2181			0.7708		
2	Momentum	0.6659	2.5285	1.8626	1.8626		1.5972	1.5972	
3	AdaGrad	0.5179	0.1147	0.4032	0.3637		0.3163	0.7434	
	RMSprop	1.0084	1.3327	0.3242		**0.7043**	1.1705		**0.9172**
4	Adam	0.7159	0.7085	0.0074	0.3184		0.7122	0.5966	
	AdaMax	0.6349	0.0857	0.5492			0.3603		
	Nadam	0.4913	0.7907	0.2994			0.6410		
	AMSGrad	0.4640	0.8817	0.4176			0.6728		
–	**AG-SGD**	**0.9714**	**0.9561**	**0.0153**	**0.0153**	**0.0153**	**0.6928**	**0.9638**	**0.9638**

5.5 Time Complexity

Referring to the pseudocode presented in Chapter 3, if the simple addition and multiplication are assumed to be $O(1)$, the time complexity of the proposed optimization algorithm is bounded and determined according to the vector dot-product (i.e., $V_{PG} \bullet V_{CG}$) in Equation 1. Since the dot-product performs an element-based multiplication between the two gradients, then the AG-SGD computational complexity is $O(n)$, where n represents the length of the two vectors.

Since all optimizers use *PG*s to calibrate *CG*s, they are all associated with a minimum time complexity of $O(n)$. This is due to the element-based addition between the two *G*s, making the use of the big *O* notation unusable for comparison purposes. Instead, to reveal the actual differences among the 10 optimizes, we measure their respective computational times. Since the optimizers calculate their calibration terms sequentially, where the practical time cost is determined by the number of computations with time complexities of $O(n)$. As a result, we measure the running or execution times for each of the 10 optimizers, as shown in Figure 5.7. Since the training task exhausts all available computation resources, the temperature of hardware (i.e., CPU/GPU) will be increased, which impacts optimization efficiency. Furthermore, if we orderly test (OT) the 10 optimizers, an early-tested optimizer will take the advantage of a more powerful and cooler hardware. To avoid this bias impacting the results, we perform the OT for three times (consecutively). Then, the result of each optimizer is averaging from its three sub-results. Particularly, if one of the sub-results deviates from the other two sub-results by 15% or more, it is discarded (i.e., assume to be affected by the performance decreasing due to the high temperature) and new tests will be conducted until the deviation is less

than 15%. To better compare the time costs between AG-SGD and the other optimizers, each averaged running time is divided by the result of AG-SGD. For example, AdaGrad is slower than AG-SGD by 41% (i.e., (1.41 – 1.00) / 1.00 = 0.41 / 1.00 = 0.41) in practice. It is evident that Vanilla is always the fastest because it does not compute any calibration term and solely updates the mode parameters by *CG*s. With respect to the other 9 optimizers, AG-SGD has a significant advantage in practical time cost.

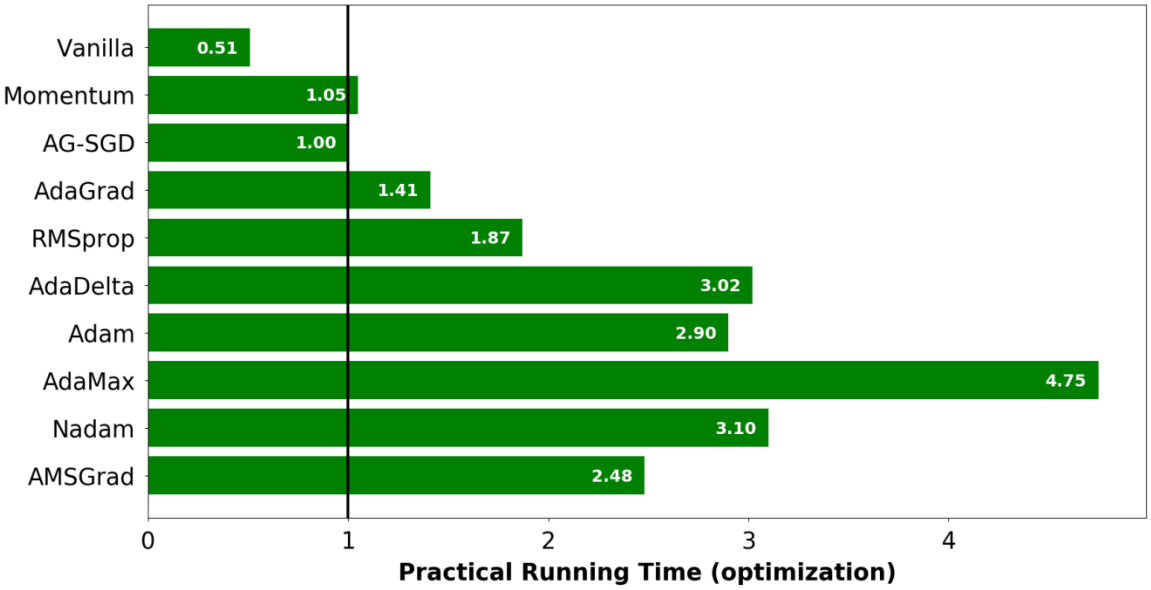


Figure 5.7 – Comparison of Practical Running Time
(result from optimization procedure only)

However, the advantage in optimization efficiency does not fully translate into an advantage in training efficiency, since the former is a subset of the latter. More specifically, since the time costs of the procedures other than optimization (e.g., backpropagation) are constant among all optimizers, it can be expected that the differences in training efficiency will be smaller than the differences in optimization efficiency, which is verified by the results shown in Figure 5.8. Although the advantages achieved by AG-SGD are less, it is still the most efficient optimizer (i.e., only slower

than Vanilla by 19% = (1.00 – 0.84) / 0.84). This means that a more complex artificial neural network can be trained employing AG-SGD, achieving higher accuracies.

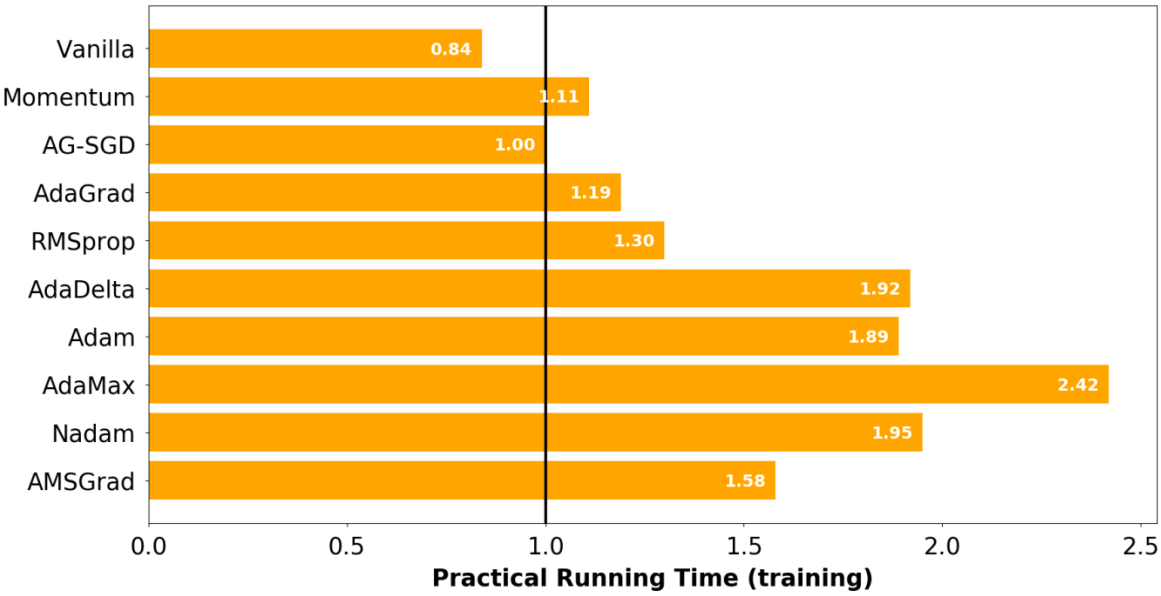


Figure 5.8 – Comparison of Practical Running Time
(result from all procedures of model training)

5.6 Summary

The group-based error rates of the two experiments (refer to the columns "Avg" under "Group" in Tables 5.2 and 5.4) show that complex optimizers (i.e., a larger group number) generally perform worse than simpler ones. These conclusions agree with our analysis in this chapter and other reviews such as [5.3]. Furthermore, according to the group-based costs and TRs of the two experiments (costs: under the columns "Avg" under "Group" in Tables 5.1 and 5.3, TR: under the columns "Group" under "Avg" in the same tables), high error rates achieved by complex optimizers may not result from higher costs, but instead from their lower TRs. These results are combined to show that the TR of an optimizer is inversely changing with the number of incorporated techniques. With respect to AG-SGD, its stable performance in TR (i.e., small difference between two TRs)

could be also attributed to the aforementioned reason because it behaves like a special Vanilla in practice. According to the group-based TRs in Tables 5.5–5.7, no distinct pattern that can be detected other than erratic values. Similar observations can be found on all individual TRs in addition to the extreme low/high TR achieved by AdaGrad/RMSprop. The above results are combined to show that AG-SGD has a significant advantage in consistently achieving high TRs when using a greater number of different cost functions. The results of time complexity indicate that AG-SGD is faster than all other optimizers, so its accuracy can be further improved by applying more complex models trained within the same time period.

CHAPTER 6

VERIFICATION, IMPROVEMENT, IMPLEMENTATION AND APPLICATION

6.1 Introduction

This chapter first verifies the Convergence Section (CS) that was proposed in Chapter 4 through the evaluation of AG-SGD on MNIST using various configurations. Because all measurements associated with AG-SGD rely on the new metric *a*, the method of computing *a* is presented with its corresponding principle. Next, the reliability of *a* and the associated principle of bypassing saddle points are verified by conducting statistical analysis of the MNIST experimental results that are covered in Chapter 5. These contents are combined to reveal the essential reasons of AG-SGD to be an effective method. Then, an improvement to the original AG-SGD is presented, which results from the variants of AG-SGD (i.e., creating alternative definitions of the two functions $F_{pg}(a,s_g)$ and $F_{\eta}(a,s_{\eta})$). Subsequently, two versions of ML models used in the experiments and the matrix-based multiplication that is implemented in the models will be briefly introduced. Finally, the intended application of AG-SGD and the associated approach for realizations are explained.

6.2 Verification of Convergence Section

To compare the lowest costs that are result from parameters within and beyond the recommended CS, we test AG-SGD on MNIST by 10 different combinations of s_g and s_{η}, starting from 1.000 and with 0.025 step-size. Table 6.1 shows that the best achieved results when both parameters have the same value of 1.100. Also, the cost and the error

rate are (roughly) decreasing in [1.000, 1.100] and increasing in [1.100, 1.200]. These tendencies indicate that AG-SGD outperforms on the current dataset when these two parameters are chosen close to 1.100. Furthermore, since the results achieved in the section-averaged within [1.000, 1.100] are better than the results in [1.100, 1.2000], the optimal parameters are found in [1.000, 1.100] with a higher probability.

Table 6.1 – Best results with varied parameters

$s_g = s_\eta$	**Minimal**		**Section Average**	
	Cost (scaled by 10,000)	**Error Rate (percentage)**	**Cost (scaled by 10,000)**	**Error Rate (percentage)**
1.000	26.32	1.49	25.80	1.48
1.025	25.90	1.46		
1.050	25.31	1.48		
1.075	25.66	1.47		
1.100	25.03	1.41	–	–
1.125	25.74	1.48	26.31	1.51
1.150	26.28	1.52		
1.175	26.29	1.52		
1.200	26.94	1.53		

To further verify that the lower value parameters for the current experiment provide for more reliable configurations, we exam the varying trajectories of the cost and the error rate across all of the tested parameters. As shown in Figures 6.1 and 6.2, we observe that there are 3 curves that generate prominent peaks in both figures and are associated with an unstable convergence process. Since all 3 peaks are a result of the largest 3 parameter values of 1.150, 1.175, and 1.200, it indicates that [1.100, 1.2000] is not a good section for obtaining the optimal result. Whereas, all 4 curves resulting from [1.000, 1.100] represent a consistent stable optimization process. Therefore, there is a higher

probability for AG-SGD to achieve the optimal result when these two parameters are in [1.000, 1.100]. The experiment results verify that the cost convergence become unstable if the two parameters go beyond the upper bound of the recommended CS (i.e., 1.156).

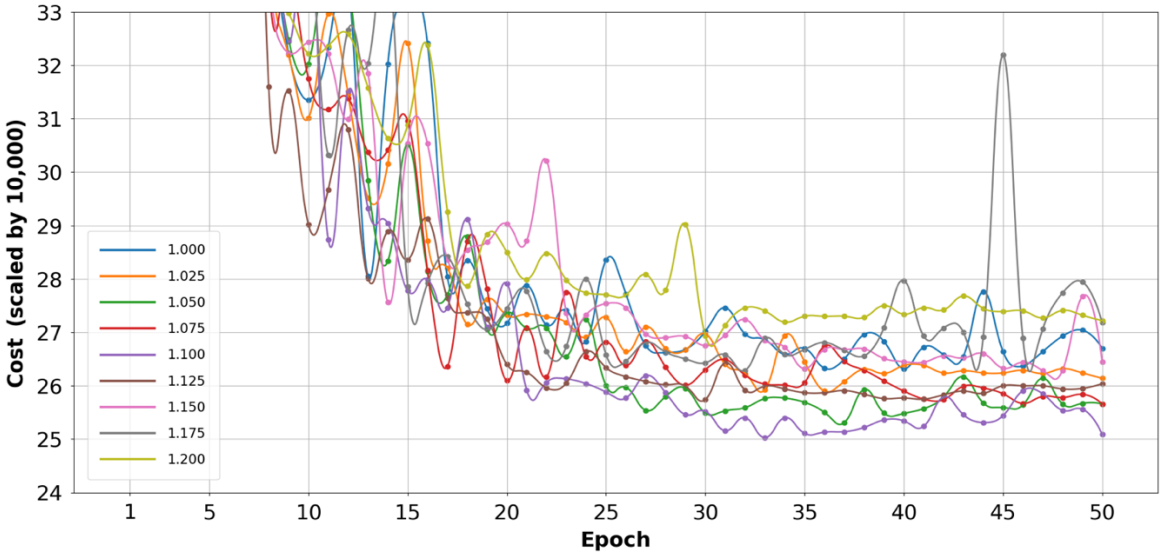


Figure 6.1 – The minimal costs with varied parameters (zoom-in, each dot represents the cost of specific configuration on the corresponding epoch)

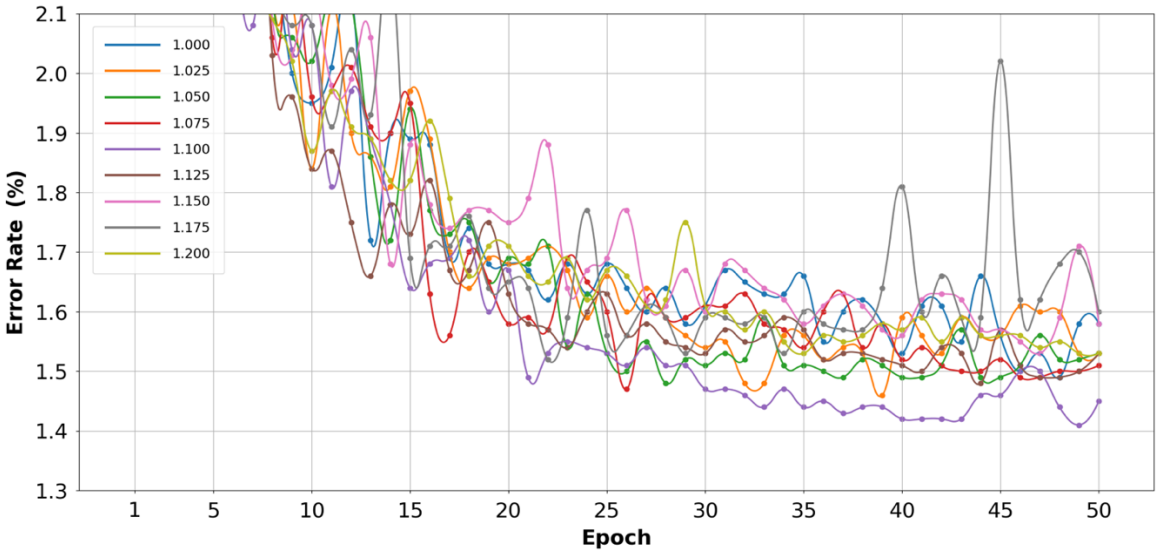


Figure 6.2 – The minimal error rates with varied parameters (zoom-in, each dot represents the error rate of specific configuration on the corresponding epoch)

6.3 Verification of The New Metric: Angle Between Consecutive Gradients

6.3.1 Computation of The Angle

Due to the fact that *NG* is used to update *PARAM* of the ML model, the former matches the latter in dimension, as shown in Figure 6.3 (note: when the element of *PARAM*/*NG* (i.e., $PARAM_{xy}/NG_{xy}$) refers to/is to update w_{xy} (i.e., the weight of a connection between two neurons), the data structure of $PARAM_{xy}/NG_{xy}$ is a vector instead of an element).

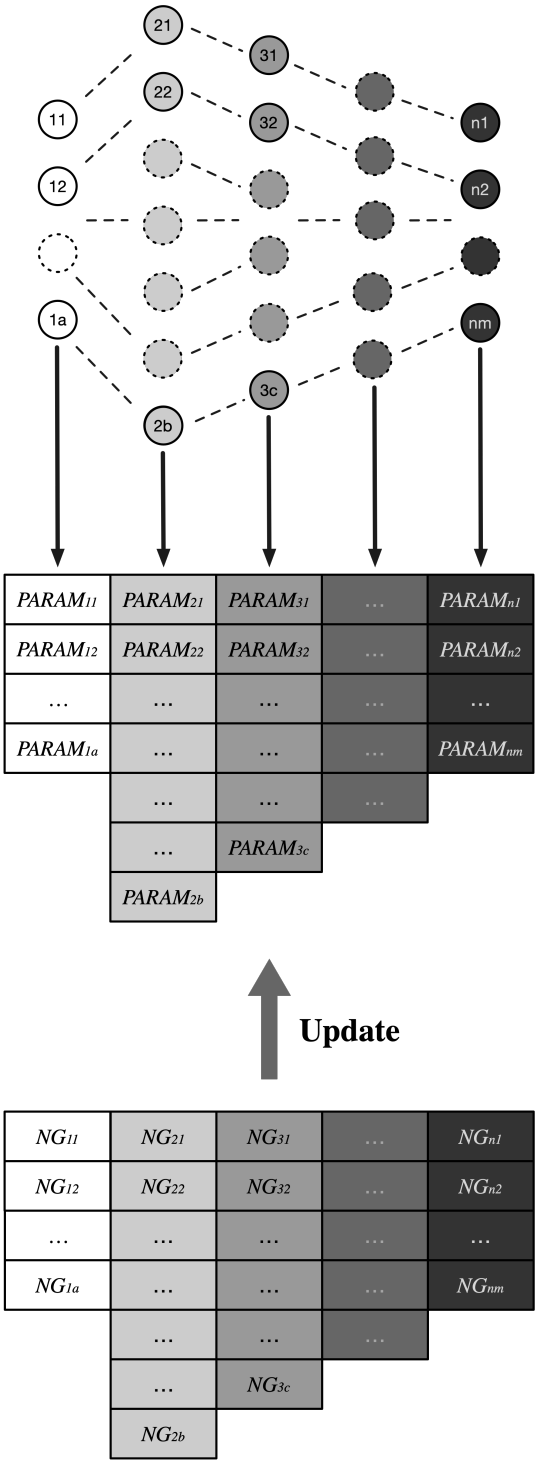


Figure 6.3 – Data structures of model parameters and the corresponding new gradients
(a lighter *NG* indicates a higher deviation associated on it;
a lighter *PARAM*/neuron means it is updated by a less accurate *NG*)

As a result, the two methods shown in Figure 6.4 can be used to calculate θ, which are:

1) Flattening the entire *PG* and *CG* into a vector, respectively. Then, computing θ between the flattened *PG* and *CG* using the presented equation (i.e., Equation 1 in Chapter 3).
2) Flattening each layer of *PG* and *CG* into a vector, separately. Then, we calculate each θ_x between the corresponding two vectors result from the x^{th} layer (i.e., the same layer). Finally, θ is obtained from averaging all θ_xs.

From a global view, BP computes all NG_{xy}s (NG_{xy} is an update of a weight or a bias) in parallel based on features of each sample. In this sense, the first method is computing a sample-based θ. From the local perspective, BP generates all NG_{xy}s layer by layer, θ that is calculated by the second method is based on layer. The two methods are common in employing all G_{xy}s in computing θ. However, they will import deviation into θ and are infeasible to accomplish in practice due to the follow two reasons:

1) According to the principle of BP, the magnitude of each NG_{xy} varies with the activation that is received from the previous layer [2.1]. This means that a $PARAM_{xy}$ (e.g. w_{xy}) will not adequately be updated when the activation value is close to 0. This problem widely occurs on ML models that are employed on datasets with a great many 0 inputs. For example, when an ANN is employed on MNIST, its input layer would heavily suffer from this problem, as most input neurons receive 0 inputs (i.e., the white pixels). Furthermore, each neuron in later layers will receive multiple activations from the previous layer (instead of receiving only 1 input activation as the input neuron), and a neuron on a later layer is less likely to receive 0 input and this possibility decreases even more for later layers. As a result, the deviations of NG_{xy}s decay with the

corresponding layers becoming deeper, which are differentiated using different shades in Figures 6.3–6.5.

2) In terms of computation, the time spent on calculating θ from all G_{xy}s is unacceptable for training an ANN. For example, if we adopt the first method to calculate θ of the ANN that is evaluated in the previous chapter, there are 224,788,480 (each vectorized G, i.e., V_{PG} or V_{CG} in Figure 6.4 contains $784 \times 256 \times 112 \times 10$ elements) parameters need to be updated. Consequently, computing each θ needs almost 40 seconds on a 2.7 GHz Quad-Core Intel Core i7 CPU and each epoch needs to compute 7.5×10^3 θs. This computation time is much longer than the time required in training the model by one epoch if we do not calculate θ. Moreover, the computational time increases exponentially as the ML model becomes larger. Therefore, it can be anticipated that improving the computational algorithm or employing advanced hardware (e.g., CPU and GPU) alone may not resolve the problem. Due to the fact that the computation of θ is only a sub-process in BP computation, we need to reduce the time cost of each epoch at a level of 10^4 on small or medium ANNs and even more on larger ones.

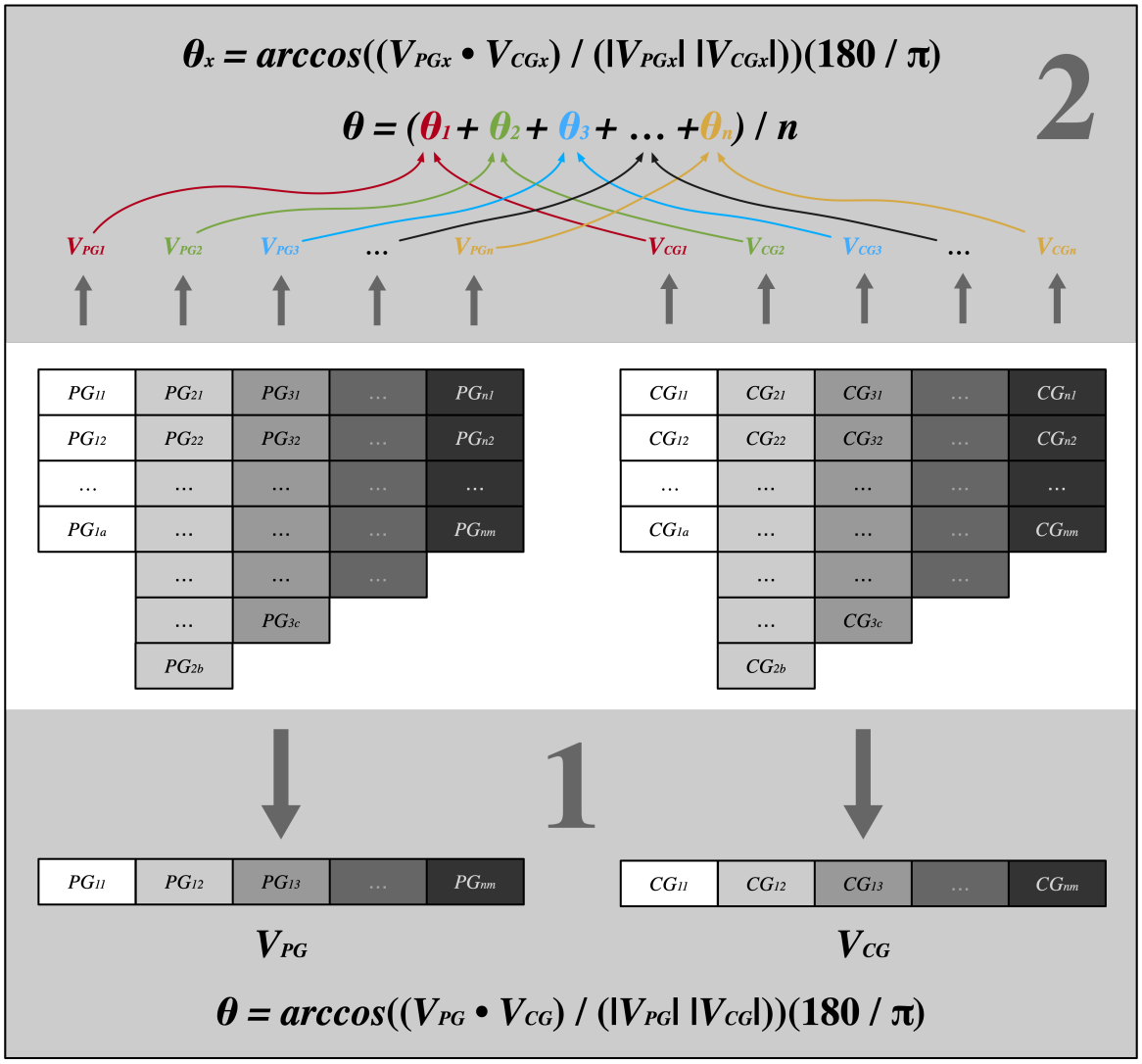


Figure 6.4 – Two abandoned methods of calculating *a* between consecutive gradients (a lighter PG_{nm}/CG_{nm} indicates a higher deviation associated on it)

As a result, adopting the aforementioned two methods will not only generate deviated θs, but are infeasible to accomplish in practice. To avoid the two shortcomings, we only employ G_{xy}s of the last layer to compute θs (i.e., the third method shown in Figure 6.5). The characteristics of this method are listed below:

1) During BP, G_{xy}s of each layer are computed based on those G_{xy}s of the previous layer [2.3], and all G_{xy}s are originally deriving from G_{xy}s of the last layer. Therefore, the difference between the latter at different moments (i.e., V_{PG}, V_{CG}) approximates the former (i.e., all G_{xy}s, the reliability will be verified by the correlation coefficient in the next subsection).
2) As we mentioned, G_{xy}s before the last layer have larger deviations compared with G_{xy}s of the last layer, and removing the former means removing large deviations from the computations of θs.
3) The computation of θ becomes much more efficient. For example, if we adopt the third method to calculate θ of the ANN evaluated in the previous chapter, then the number of elements in V_{PG} and V_{CG} is reduced to only 1,120 (i.e., 112 $\times$ 10). Then, each θ only costs about 5×10^{-4} seconds on the same CPU and improves the efficiency by 8×10^4 (i.e., $40 / 5 \times 10^{-4}$) times. As a result, the time of computing θ will meet the requirement of training DNNs.

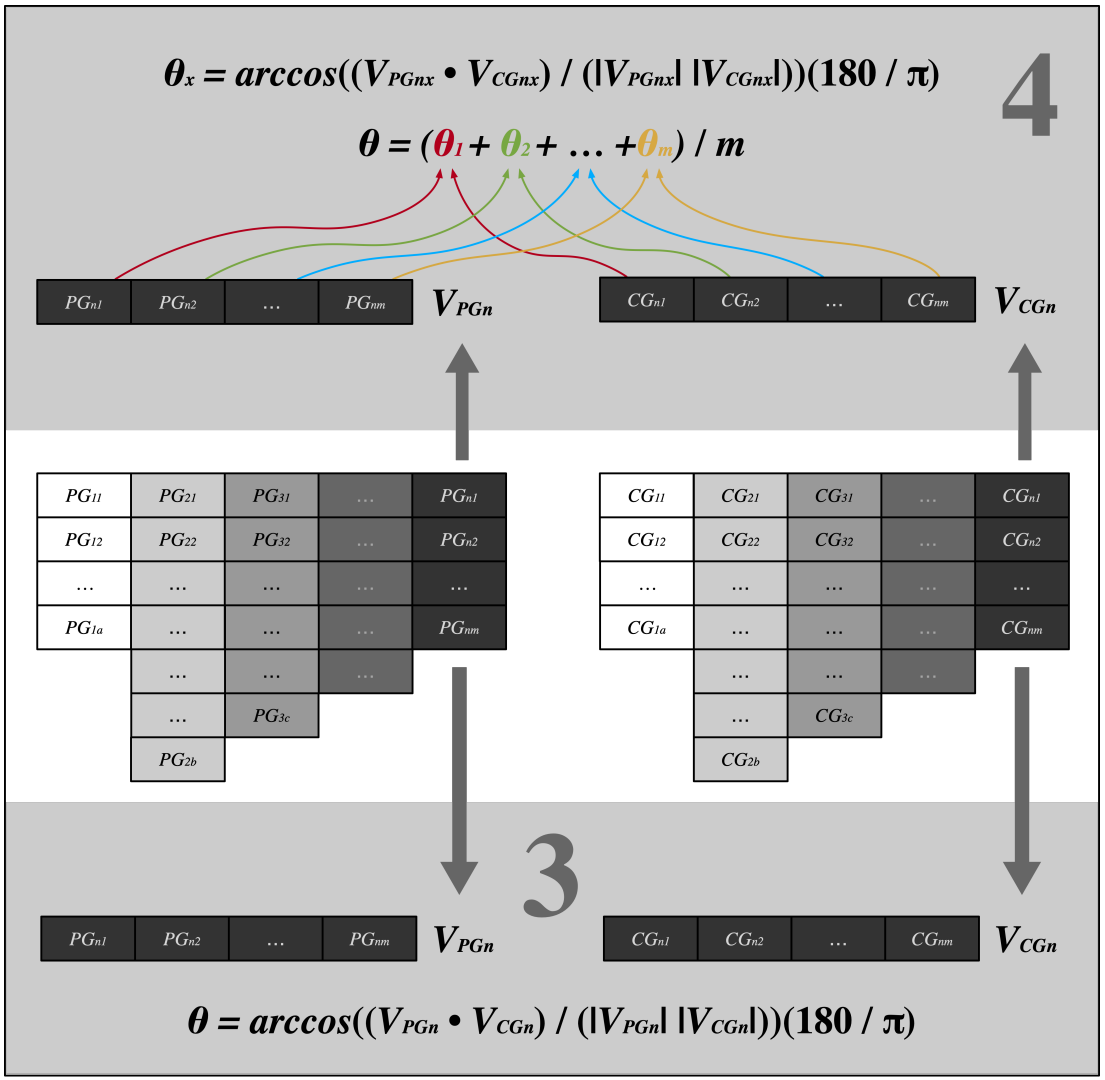


Figure 6.5 – The adopted method of calculating *a* between consecutive gradients (a lighter PG_{nm}/CG_{nm} indicates a higher deviation associated on it)

Due to the fact that all G_{ny}s (n represent the last layer) are computed in parallel during BP (i.e., layer-based computation), calculating layer-based θ between V_{PGn} and V_{CGn} aligns with the principle of BP (i.e., the third method). However, the computation of neuron-based θs between the corresponding G_{ny}s (i.e., the fourth method in Figure 6.5) is beyond the minimum layer-based computation that is supported by BP.

6.3.2 Verification of Using The Angle as A Reliable Metric

As introduced in Chapter 3, AG-SGD attempts to reduce the cost in one step by reversing *PG* (when needed), leading to a lower number of obtuse angles among the other SGD optimizers. Since the lower bound of the CS is 120˚ under the default configuration, the *PG* revision will occur only when the angle is larger than 120˚. To verify the effectiveness of angle reduction in the default CS, we count and compare the numbers of angles in [120˚, 170˚) for all of the 10 optimizers (note: [170˚, 180˚] is eliminated because no optimizer generates angle in this section) in the experiment on MNIST. As the data indicate in Table 6.2, AG-SGD has the least number of obtuse angles in all of the sections. Particularly, it has less than the second-least optimizer (i.e., AMSGrad) by 42.14% = (9371 – 5422) / 9371 and considering the averaged of all other optimizer by 54.17% = (11830 – 5422) / 11830 in total. Therefore, AG-SGD spends less steps on the high cost areas among the 10 optimizers, and the saved steps are used to find better minima.

Table 6.2 – Number of angles in different sections

GP	Optimizer	[120, 130)	[130, 140)	[140, 150)	[150, 160)	[160, 170)	Total			Reduction Rate
							Each	Group	Overall	
1	Vanilla	6527	3482	1153	138	4	11304	11831	**11830**	**54.17%**
	AdaDelta	5903	4423	1775	244	12	12357			
2	Momentum	5747	2884	814	70	3	9518	9518		
3	AdaGrad	6282	5166	2026	268	21	13763	13820		
	RMSprop	6381	4886	2207	391	11	13876			
4	Adam	7498	4951	2030	394	8	14881	12153		
	AdaMax	6083	3155	992	115	4	10349			
	Nadam	7165	4551	1917	372	7	14012			
	AMSGrad	5402	2912	902	134	21	9371			
–	**AG-SGD**	**3908**	**1289**	**205**	**18**	**2**	**5422**	**5422**	**5422**	

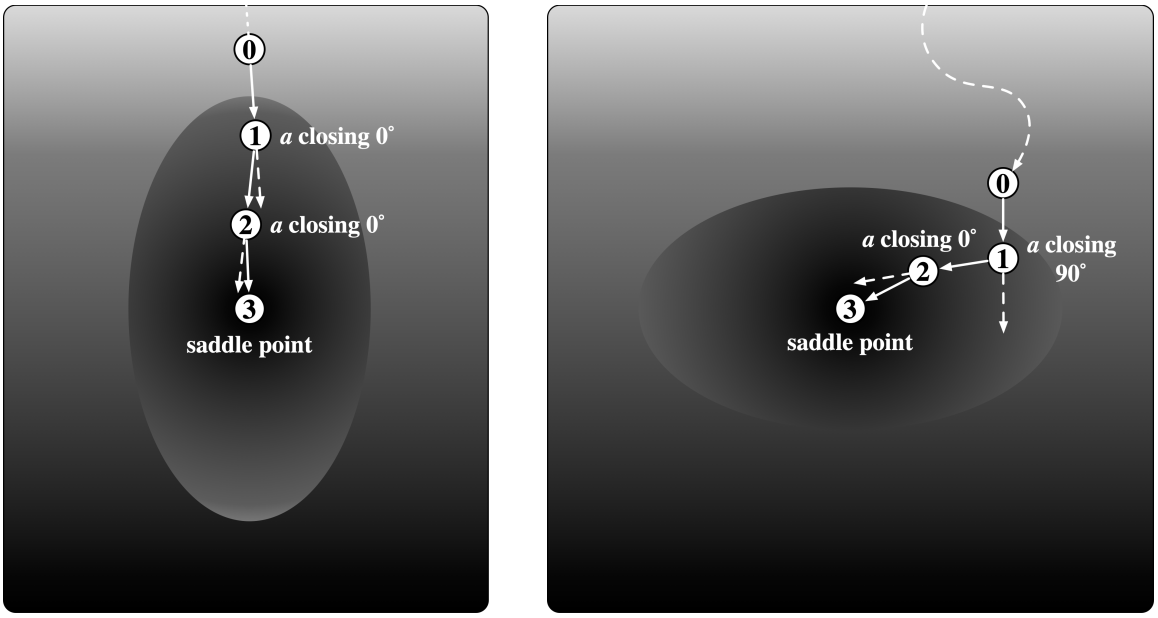
To check the relationship between the number of obtuse angles and the associated cost and error rate, we compute the correlation coefficients for all of the 10 optimizers, as shown in Table 6.3 Since the principle of SGD is to indirectly reduce the error rate via directly reducing the cost, the correlation coefficient associated with the cost reduction which is close to 0.8 as shown in Table 6.3 indicates a strong positive correlation between the two metrics. The reduction in correlation of error rate is due to the averaged TR that is lower than 1 among all of the optimizers presented in the previous chapter. Therefore, reversing *PG* when needed is an effective method and the angle between the gradients is a reliable metric in reducing the cost.

Table 6.3 – Correlation coefficients between obtuse angles and cost/error rate

GP	Optimizer	Total Angle	Average		Coefficient	
			Cost (scaled by 10,000)	Error Rate (percentage)	Cost (scaled by 10,000)	Error Rate (reduced by TR)
1	Vanilla	11304	27.64	1.62	**0.7923**	0.6900
	AdaDelta	12357	27.64	1.60		
2	Momentum	9518	27.94	1.61		
3	AdaGrad	13763	28.14	1.63		
	RMSprop	13876	28.43	1.59		
4	Adam	14881	29.43	1.66		
	AdaMax	10349	27.52	1.60		
	Nadam	14012	28.95	1.66		
	AMSGrad	13763	28.14	1.63		
–	**AG-SGD**	**5422**	**25.45**	**1.44**		

6.4 Verification of Bypassing Saddle Points

The proposed method may result in the cost being trapped into saddle points as shown in Figure 6.6. This problem occurs when the cost movement (the solid lines between positions 0 and 1) is aligning with (i.e., Figure 6.6(a)) or perpendicular to (i.e., Figure 6.6(b)) the saddle pit. As we concluded, *NG* is dominated by *CG* when *a* is close to 0˚ or 90˚ (the dashed lines are extensions of the previous gradients). As shown in both subfigures, *CG* of position 1 points to the centers of saddle pits, and the costs would guide the movement to position 2, finally reaching position 3 for the same reason. Although this problem would rarely occur in practice, it reveals a deficiency in the principle of the proposed optimizer. There are two methods to resolve this problem. (1) Conducting the experiment again may change the trajectory of the cost reduction and bypass the saddle pit. Even if the new trajectory is still crossing the saddle pit as in Figure 6.6 (b), the cost movement may not closely be perpendicular to the saddle pit. In this case, the weights of *PG*s will be increased, resulting in the cost escaping from the saddle pit. Although this result would not occur on the saddle pit in Figure 6.6 (a), the possibility of bypassing the saddle pit (a) would be higher than the saddle pit (b) because the former has a narrower cross-section than the latter in the direction of cost movement. (2) Employing a η warm restarts scheduler would enable the cost to escape from the saddle pit by increasing the magnitudes of *G*s, which is also applicable to other optimizers.



(a) **(b)**

Figure 6.6 – Two cases of trapping into the saddle points
(lighter colors indicate lower costs)

To verify the aforementioned problem would rarely occur in practice, we recorded the changing trajectory of the first 1,000 angles under the default setting (results from the experiment on MNIST), as shown in Figure 6.7. Our findings indicate that only 24 of 1,000 (2.4%) of the angles are larger than 120˚ and that there are no consecutive large angles. If these large angles result from saddle points, the data prove that the AG-SGD can achieve escaping from these saddle points in only one step (i.e., no consecutive larger angle). In line with an established principle, when a large angle (>120˚ under the default setting) is generated, w_{pg} will be less than zero and makes *PG* point in the direction that deviates from the saddle points (refer to Figure 3.2(a)). Although *CG* may point to the saddle points and w_{cg} is slightly larger than the $|w_{pg}|$ (i.e., only when the $\theta \in (120^{\circ}, 135^{\circ})$), the accumulated *PG*s will be much larger than the single *CG* in magnitude. As a result,

NG (i.e., η ($w_{pg}PG + w_{cg}CG$)) aligns in direction with *PG*, forcing the cost deviating from the saddle points (i.e., moving backward).

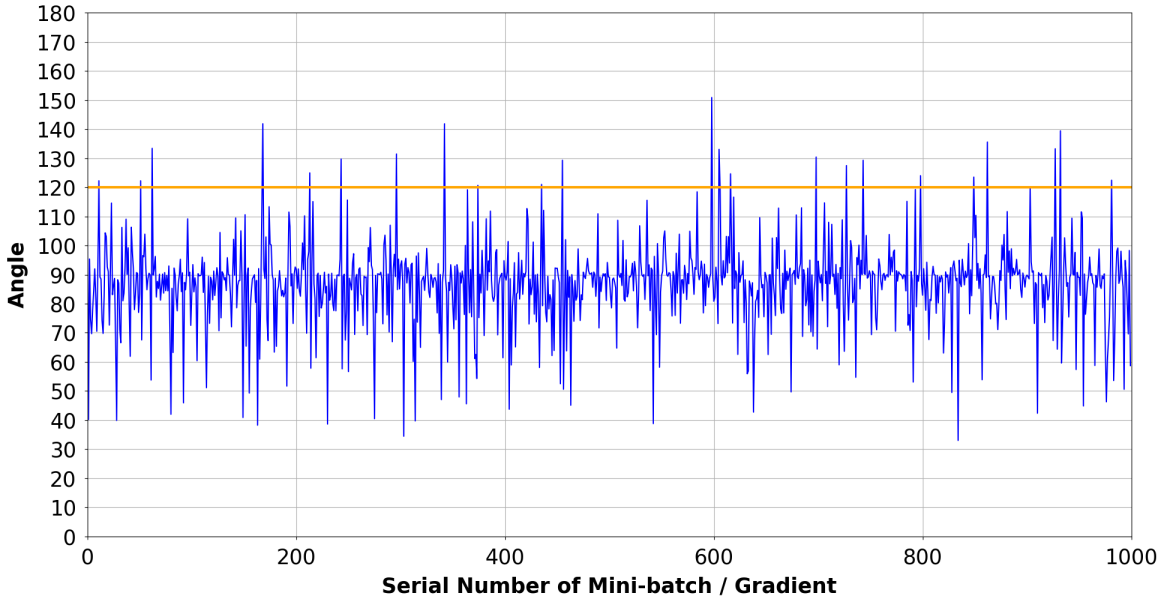
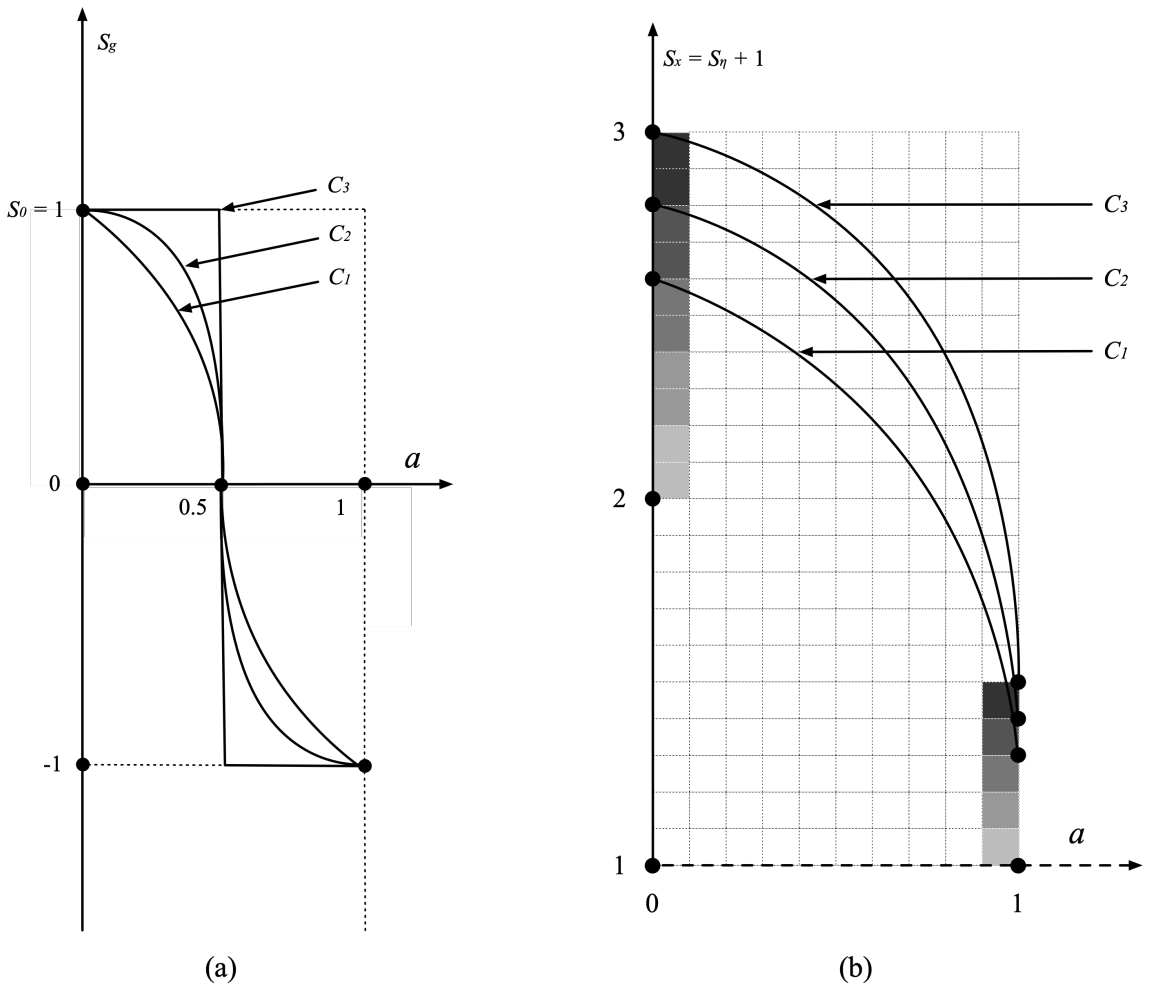


Figure 6.7 – Trajectory of the first 1,000 angles under the default setting
(results from the experiment on MNIST, only 24 of 1,000 are larger than 120˚)

6.5 Improvement from Variant

6.5.1 Criterion of Defining Variants

A curve can be an effective alternative to L_0 of $F_{pg}(a,s_g)$ and $F_\eta(a,s_\eta)$ in Figure 3.2, as long as it satisfies the following four requirements. They are: (1) a is limited within [0, 1]; (2) the curve has to be continuously and monotonically decreasing within [0, 1]; (3) the interception point on the horizontal axis is 0.5; (4) the intercept point on the vertical axis is not less than 1 for $F_{pg}(a,s_g)$ and 2 for $F_\eta(a,s_\eta)$. Accordingly, the curves C_1, C_2, C_3 in Figure 6.8 are promising alternatives for L_0, which may achieve better results on problems in certain fields.



(a) (b)

Figure 6.8 – Alternatives of the functions $F_{pg}(a,s_g)$ in (a) and $F_\eta(a,s_\eta)$ in (b) (curves C_1, C_2 and C_3 are alternative definitions that comply with the concept proposed in Chapter 3; the horizontal axis is the normalized angle between PG and CG; two intercept points on vertical axes s_g and s_η are parameters of the proposed method)

6.5.2 Study Case: Improvement from A Non-linear Variant

As mentioned in the previous chapter, AG-SGD does not show the best converging speed in the first 14 epochs. To determine the reason, we compute the averaged angles between the gradients of all 50 epochs. Referring to Figure 6.9, the angles during the first 14 epochs are distinctly smaller than other epochs, which means more acute angles are generated during earlier epochs. Since the magnitude of the NG variance decreases rapidly as the angle approaches to 0˚ (refer to Figure 3.2), we expect that the convergence

can be accelerated if we replace the simplistic linear line L_0 of $F_{pg}(a,s_g)$ with others which have higher slopes, such as the curves C_1, C_2, C_3 in Figure 6.8.

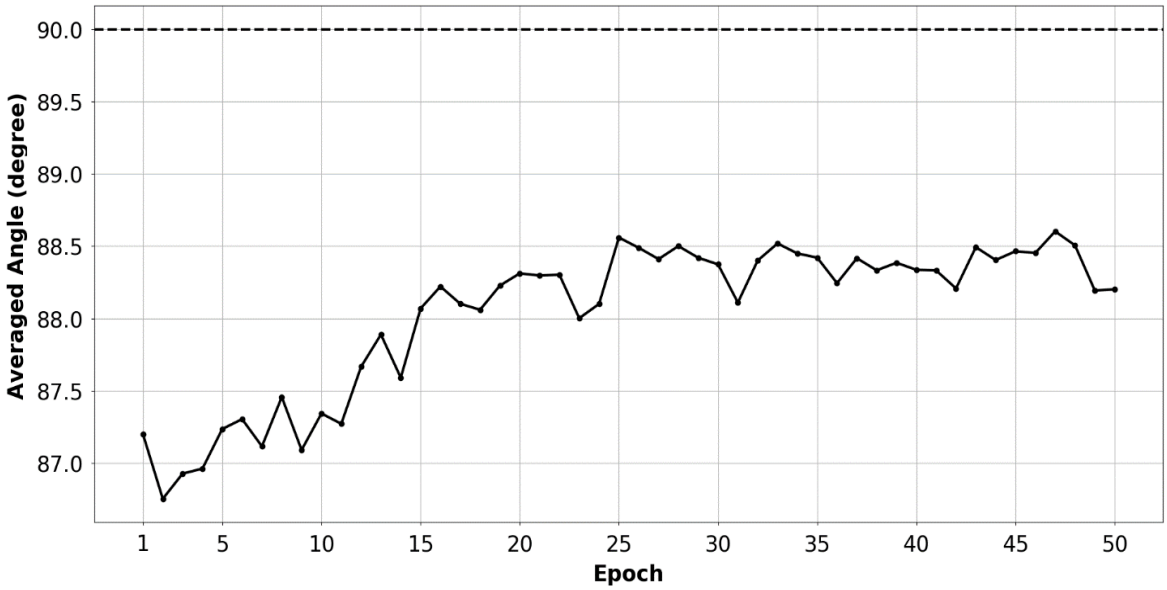


Figure 6.9 – Averaged angle between the consecutive gradients of each epoch (each dot denoted the averaged angles within each epoch)

To verify the feasibility of this solution, we have tested a non-linear definition $s_g cos(a/0.3183)$ of $F_{pg}(a,s_g)$ (setting 0.3183 such that the curve crosses the x-axis at 0.5). The results presented in Figure 6.10 indicate that the convergence speed effects from the non-linear definition are consistently faster than those from the linear ones. The distinction begins to manifest as of epoch no.11 (except for epochs no.18 and 20), which conforms with our expectation.

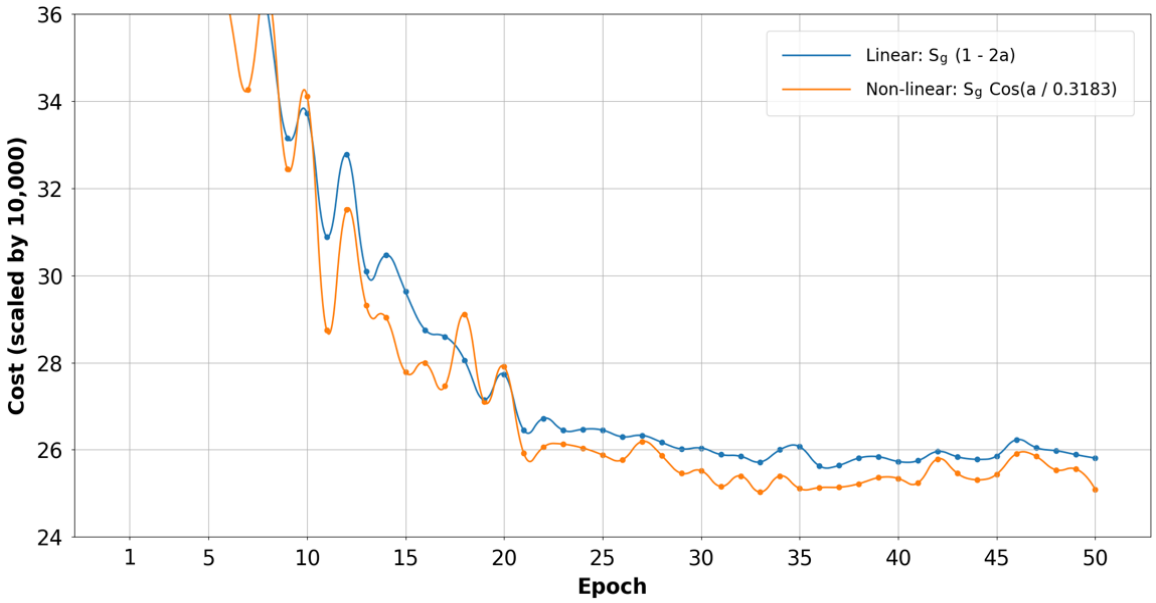


Figure 6.10 – Comparison of convergence speeds between linear (simplistic) and non-linear definitions of $F_{pg}(a,s_g)$

In addition, employing a non-linear definition does not increase the number of parameters. For example, s_g is the only parameter that exist in both the non-linear (i.e., $s_g cos(a/0.3183)$) and linear (i.e., $s_g(1\text{-}2a)$) definitions, as shown in the curves and lines in Figure 6.11 (i.e., when $s_g \in [1.0, 1.2]$, step by 0.1). It is important to note that mainstream deep-learning libraries (e.g., Tensorflow [1.18], Keras [1.19], Caffe [1.20], PyTorch [1.21]) identify the Adam and the AdaMax as two different optimizers, even though their only difference consist of the norms used (Adam/AdaMax uses l_2/l_∞ norm). Complying with the same criterion, if one/two functions (i.e., $F_{pg}(a,s_g)$ and $F_\eta(a,s_\eta)$) of AG-SGD is/are modified, a new optimizer with two parameters (i.e., s_g and s_η) is obtained (i.e., function modification should not be considered equivalent to parameter tuning). Compared with the Adam-family which consists of two optimizers, AG-SGD-family is able to generate numerous new optimizers by defining various definitions.

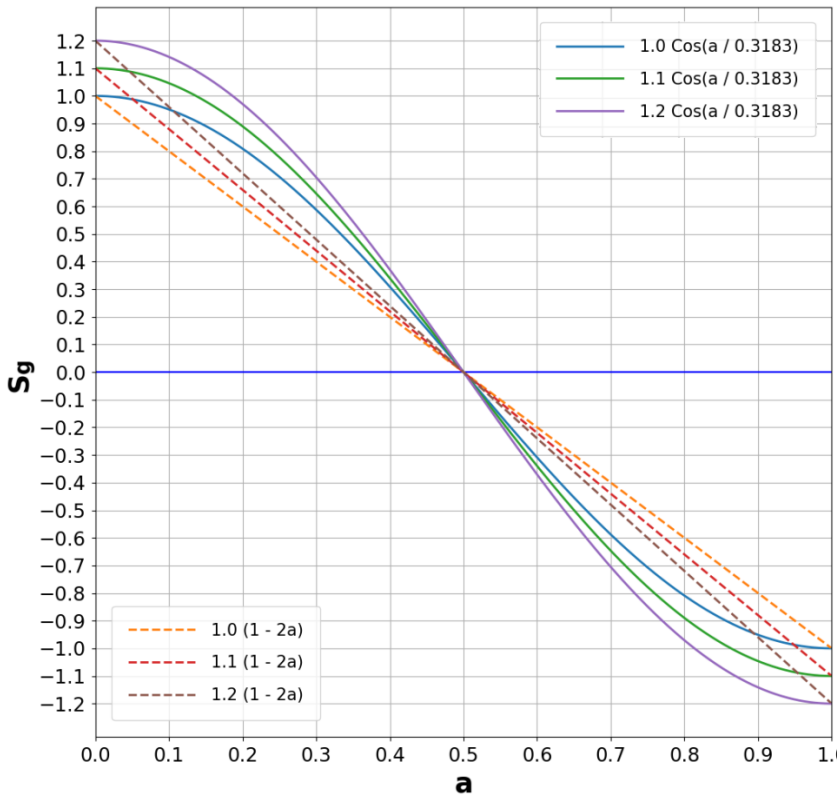


Figure 6.11 – Relationship among some representative linear and non-linear definitions of $F_{pg}(a,s_g)$ ($s_g \in [1.0, 1.2]$, step by 0.1)

6.6 Implementation

6.6.1 Matrix-based Multiplication

The employed matrix-based multiplication is implemented based on the method in [6.1], benefiting the computations of BP and FP. In the traditional looping approach, BP and FP are orderly executed on samples of the mini-batch. Conversely, the matrix-based approach treats all samples as a whole (i.e., batch-based computation) and generate the all results in parallel.

6.6.2 Two Versions: CPU and GPU

Each evaluated model has two versions. One is using a CPU and the other is using the power of a GPU. The GPU version is realized using Compute Unified Device

Architecture (CUDA) which is a parallel computing platform and programming model developed by NVIDIA [6.2]. Although the GPU version would be faster, the practical difference in speed between two versions is affected by many other factors. For example, the GPU version executes each BP with a cost of copying all data from the main memory to the GPU memory [6.3]. The additional overhead makes the GPU version slower than the CPU version when the batch size and/or the ML model is relatively small and/or simple. To reimplement ML models and reproduce results, one needs to install Numpy (Numerical Python) [6.4] for the CPU version and PyCUDA (accessing CUDA from Python) [6.5] for the GPU version.

6.7 Application

In addition to machine learning experts, the proposed algorithm can be used by non-professional users (e.g., programmers working for small businesses) who want to obtain insight from their data with minimum efforts. To better serve this group of users, AG-SGD is specifically designed/enhanced on the following four aspects. These are:

1) Due to the fact that non-professional users have limited ability of tuning parameters, AG-SGD sets the average of recommended section (i.e., (0.820, 1.156)) as the default configuration to better fitting data in most fields. Also, it abandons all measurements without finite parameter tuning spaces (e.g., the learning rate can be set to any magnitude in practice) because we are unable to determine a universal value that results in decent performances on various datasets (e.g., it is common to see the magnitude of learning rate change between levels of $10^{-1}/10^{-2}/10^{-3}/10^{-4}$). It is evident that non-professional users are unable to tune these types of parameters.

2) To improve the accuracy using the default setting, AG-SGD intends to guide non-professional users in training more complex models (i.e., more layers and/or more neurons in layers) instead of tuning various parameters. A more reasonable approach is that the former is not only more understandable (i.e., the larger the better), but also more effective than the latter because the complexity/architecture is the foundation of ML model.

3) Since non-professional users usually do not have high performance computing systems, we assume that powerful GPUs are not available to accelerate AG-SGD. Therefore, to reduce the time cost in training a complex model, it is important to establish a very small batch size and improve the corresponding accuracy. Since the experiment on the dataset NSL-KDD has shown that our method is effective on the smallest batch size (i.e., 1), its usage has a much lower requirement than other optimizers which have to set much larger batch sizes (e.g., 128/256/512...) to achieve decent results. As such, AG-SGD can train very complex models within the same time period as other methods.

4) We have shown that high accuracies can be obtained by setting the two parameters of AG-SGD to the same value v (i.e., $s_g = s_\eta = v$, where $v \in (0.820, 1.156)$). Since v determines the CS, a v that is close to 0.820/1.156 will result in a weaker/stronger convergence, indicating a more conservative/aggressive configuration (i.e., the correlation between the intensity of convergence and the magnitude of v is monotonous). This intuitive interpretation of the parameter tuning is completely understandable to non-professional users and provides sufficient guidelines to find the best result. For example, if a user obtains a higher accuracy by decreasing v from

1.1 to 1.0, the value 1.1 could be interpreted as a somewhat more aggressive configuration then a v that is smaller than 1.1 should be tried. Although other optimizers also have one parameter, these parameters have no monotonous correlation with the intensity of convergence. Consider, the parameter β of Momentum which can be difficult for non-professional to fully understand its affects. Particularly, the consequences of adjusting β in terms of convergence intensity are completely unpredictable. The usage difficulties of other optimizers (e.g., Adam) with multiple parameters (e.g., β, β_1, β_2, η, ε, etc.) are not comparable to the simplicity of using AG-SGD in a significant manner.

It is evident that none of the compared optimizers meet all four requirements. For example, AdaGrad has one parameter (i.e., learning rate), but its value changes too radically (i.e., cannot define a universal value as the default setting) and is obscure to non-professional users. Most importantly, its error rates are far beyond AG-SGD (e.g., the averaged error rates: 24.34% versus 13.71% in Table 5.4). Consequently, AG-SGD has a significant advantage on the intended application (i.e., used by non-professional users), which can be implemented on platforms that provide Auto-ML solutions (i.e., training ML models from built-in free-tuned algorithms), such as Amazon SageMaker Autopilot [6.6].

It needs to be re-emphasized that adjusting the two parameters together is the recommended configuration approach for non-professional users, but machine learning experts can achieve higher accuracies by tuning each parameter separately. However, the intended objective of AG-SGD is to fine tune and search for better definitions of $F_{pg}(a,s_g)$

and $F_\eta(a,s_\eta)$, by developing a new tuning section with a better CS for non-professional users in an effort to improve their results.

6.8 Summary

According to the verification of the correlation coefficient between a and the averaged costs from all optimizers, a would be an effective metric for improving the accuracy of *NG*. Also, the actual behaviors of AG-SGD comply with our intention based on the verification of CS. Verifications of CS and a are combined to show that a computed by the third method can accurately reflect the correlation between *PG* and *CG*, translating to a good accuracy that is realized in the adoption of a-based measurements in practice.

CHAPTER 7

CONCLUSIONS

7.1 Possible Improvements

7.1.1 Computation of The Angle

Due to the limitation on computing power, the current method in computing the angle is solely employing the gradient difference of the last layer. However, the fact is that the gradient differences of other layers can provide more useful information (at least for datasets without a lot of 0 inputs). To resolve the computation problem, one promising method is to divide the layer-scaled computation to a smaller unit (e.g., half-/quarter-layer-scaled, neuron-scaled), reducing the length/size of each variable and accomplishing all small-scaled computations in parallel using a GPU. However, the minimum calculation scale/unit of BP is layered, and we need to find supportive explanations for improvements that result from smaller scaled calculations. In addition, although we can remove 0 inputs from the dataset, the activation difference among input neurons remains unchanged, resulting in unbalanced updates among the neurons of different layers.

7.1.2 Angle-based Learning Rate Scheduler

As an external technique, the learning rate scheduler (e.g., annealing/decay and warm restarts) may be combined with AG-SGD to further reduce the cost. Based on our preliminary experiments on MNIST, if the researchers decay the learning rate then the lowest, average and stability of the cost can be improved. However, the traditional/tested learning rate scheduler is incompatible with AG-SGD, as it adjusts the learning rate based

on epoch instead of the angle. Therefore, we are testing multiple angle-based learning rate methods to significantly improve AG-SGD.

7.2 Significant of The Work

The proposed AG-SGD would be the first optimizer that utilizes the angle between consecutive gradients to improve ML models. From the perspective of the research, the utilization of angle largely reduces the difficulty of creating new measurements. Now, researchers can employ the previous gradient, angle, or both in their improvements. Particularly, the angle-based measurements can increase the TR of the model, which is unachievable by measurements that solely rely on the previous gradients. This improvement implies that the accuracy associated with existing optimization algorithms can be further improved by incorporating the proposed or designing new angle-based measurements. Internal conflicts that would otherwise confiscate benefits resulting from different measurements are nonexistent because both old and new measurements rely on different metrics. It is possible to design and develop new angle-based optimizers from scratch, and creating AG-SGD variants through defining alternative definitions of $F_{pg}(a,s_g)$, $F_{\eta}(a,s_{\eta})$ or both is the easiest way for achieving the same goal. There is an unlimited number of functions that can be chosen from, and the modified AG-SGD would result in good performances for specific problems in a variety of fields. Most importantly, a new variant will inherit all advantages (e.g., cost awareness ability) that come with the original AG-SGD, which may be unachievable using newly-designed measurements. With respect to the application, the most important inheritable property is the CS (i.e., a limited parameter tuning section). Enabling the CS provides a contribution

that allows a non-professional user to find the optimal configuration with much less effort. Furthermore, the utilization of CS indicates new variants that use the default settings, providing accurate results and largely broadening the application of the model in various scenarios.

LIST OF REFERENCES

[1.1] O. I. Abiodunab, A. Jantana, A. E. Omolarac, K. V. Dadad, N.A. Mohamede, H. Arshadf, "State-of-the-art in artificial neural network applications: A survey," Heliyon, vol. 4, 2018.

[1.2] L. F. Guilhoto, "An overview of artificial neural networks for mathematicians," 2018. Available: http://math.uchicago.edu/~may/REU2018/REUPapers/Guilhoto.pdf.

[1.3] KDnuggets, "The 8 neural network architectures machine learning researchers need to learn," Available: https://www.kdnuggets.com/2018/02/8-neural-network-architectures-machine-learning-researchers-need-learn.html.

[1.4] A. Krizhevsky, I. Sutskever, G. E. Hinton, "ImageNet classification with deep convolutional neural networks," Conference on Neural Information Processing Systems, Lake Tahoe, United States, December 2012.

[1.5] R. Salakhutdinov, G. Hinton, "Deep Boltzmann Machines," 25th International Conference on Artificial Intelligence and Statistics, Florida USA, April 2009.

[1.6] K. K. Al-jabery, T. Obafemi-Ajayi, G. R. Olbricht, D. C. Wunsch II, "Deep Learning for Power System Data Analysis" in *Computational Learning Approaches to Data Analytics in Biomedical Applications*, 1st ed. Elsevier, 2020.

[1.7] H, Lee, R, Grosse, R. Ranganath, A. Y. Ng, "Convolutional deep belief networks for scalable unsupervised learning of hierarchical representations," In Proceedings of the 26th International Conference on Machine Learning, Montreal, Canada, 2009.

[1.8] G. Hinton, "Deep Belief Nets," 2007. [Online]. Available: https://www.cs.toronto.edu/~hinton/nipstutorial/nipstut3.pdf.

[1.9] B. T. Polyak, "Some methods of speeding up the convergence of iteration methods," USSR Computational Mathematics and Mathematical Physics, vol. 4, pp. 1–17, 1964.

[1.10] D. P. Kingma, J. Ba, "Adam: a method for stochastic optimization," 3rd International Conference on Learning Representations, San Diego, May 2015.

[1.11] I. Loshchilov, F. Hutter, "SGDR: Stochastic gradient descent with warm restarts," In Proceedings of International Conference on Learning Representations, Toulon, France, April 2017.

[1.12] Keras, "Regression losses." [Online]. Available: https://keras.io/api/losses/regression_losses/#meansquarederror-class.

[1.13] Keras, “CategoricalCrossentropy.” [Online]. Available: https://www.tensorflow.org/api_docs/python/tf/keras/losses/CategoricalCrossentropy.

[1.14] Keras, “Regression losses: huber,” [Online]. Available: https://keras.io/api/losses/regression_losses/#huber-class.

[1.15] Keras, “Regression losses: cosinesimilarity,” [Online]. Available: https://keras.io/api/losses/regression_losses/#cosinesimilarity-class.

[1.16] Keras, “Layer activation functions.” [Online]. Available: https://keras.io/api/layers/activations/.

[1.17] P. Golik, P. Doetsch, H. Ney, “Cross-entropy vs. squared error training: a theoretical and experimental comparison,” Interspeech, 2013.

[1.18] Tensorflow, [Online]. Available: https://www.tensorflow.org.

[1.19] Keras, [Online]. Available: https://keras.io.

[1.20] Caffe, [Online]. Available: https://caffe.berkeleyvision.org.

[1.21] PyTorch, [Online]. Available: https://pytorch.org.

[1.22] M. D. Zeiler, “ADADELTA: An adaptive learning rate method,” 2012. [Online]. Available: arXiv: https://arxiv.org/abs/1212.5701.

[2.1] M. Nielsen, Neural Networks and Deep Learning, 2015. [Online]. Available: http://neuralnetworksanddeeplearning.com/index.html.

[2.2] H. Robbins, S. Monro, “A stochastic approximation method,” The annals of mathematical statistics, vol. 22, pp. 400–407, September 1951.

[2.3] D. E. Rumelhart, G. E. Hinton, R. J. Williams, “Learning representations by back-propagating errors,” Nature, pp. 533–536, 1986.

[2.4] Z Wang, X Xu, W Zhao, Y Zhang, S He, “Optimizing sparse matrix-vector multiplication on CUDA,” International Conference on Education Technology and Computer, Shanghai, China, 2010.

[2.5] Y. Bengio, J. Louradour, R. Collobert, “Curriculum learning”. Proceedings of the 26th Annual International Conference on Machine Learning, pp. 41–48, 2009.

[2.6] W. Zaremba, I. Sutskever, “Learning to Execute,” 2014. [Online]. Available: arXiv: 1410.4615.

[2.7] J. Zhang, I. Mitliagkas, C. Ré, “YellowFin and the art of Momentum tuning,” 2017. [Online]. Available: arXiv: 1706.03471.

[2.8] M. Denkowski, G. Neubig, “Stronger baselines for trustable results in neural machine translation,” 2017. [Online]. Available: arXiv: 1706.09733.

[2.9] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, N. A. Gomez, I. Polosukhin, “Attention is all you need,” In Advances in Neural Information Processing Systems, CA, USA, May 2017.

[2.10] L. S. Smith, J. P. Kindermans, V. Q. Le, “Don’t decay the learning rate, increase the batch size,” 2017. [Online]. Available: arXiv: 1711.00489.

[2.11] J. Dean, S. G. Corrado, R. Monga, K. Chen, M. Devin, V. Q. Le, Y. A. Ng, “Large scale distributed deep networks,” Neural Information Processing Systems, pp. 1–11, 2012.

[2.12] J. Pennington, R. Socher, D. C. Manning, “Glove: global vectors for word representation,” Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing, pp. 1532–1543, 2014.

[2.13] Y. Dauphin, R. Pascanu, C. Gulcehre, K. Cho, S. Ganguli, Y. Bengio, “Identifying and attacking the saddle point problem in high-dimensional non-convex optimization,” 2014. [Online]. Available: arXiv: 1406.2572

[2.14] K. Kawaguchi, “Deep learning without poor local minima,” In Advances in Neural Information Processing Systems, Barcelona, Spain, December 2016.

[2.15] C. Zhang, S. Bengio, M. Hardt, B. Recht, O. Vinyals, “Understanding deep learning requires rethinking generalization,” In Proceedings of International Conference on Learning Representations, Toulon, France, April 2017.

[2.16] S. N. Keskar, D. Mudigere, J. Nocedal, M. Smelyanskiy, P. T. P. Tang, “On large-batch training for deep learning: generalization gap and sharp minima,” In Proceedings of International Conference on Learning Representations, Toulon, France, April 2017.

[2.17] L. Dinh, R. Pascanu, S. Bengio, Y. Bengio, “Sharp minima can generalize for deep nets,” In Proceedings of the 34th International Conference on Machine Learning, Sydney, Australia, August 2017.

[2.18] S. R. Sutton, “Two problems with backpropagation and other steepest-descent learning procedures for networks.” In Proceedings of 8th Annual Conference on Cognitive Science Society, London, UK, 1986.

[2.19] N. Qian, “On the momentum term in gradient descent learning algorithms,” Neural Networks : The Official Journal of the International Neural Network Society, vol. 12, pp. 145–151, 1999.

[2.20] J. Duchi, E. Hazan, Y. Singer, “Adaptive subgradient methods for online learning and stochastic optimization,” Journal of Machine Learning Research, vol. 12, pp. 2121–2159, 2011.

[2.21] John C. Duchi, Michael I. Jordan, H. Brendan McMahan, “Estimation, optimization, and parallelism when data is sparse,” Neural Information Processing Systems, Nevada, United States, December 2013.

[2.22] Z. Chen, Y. Xu, E. Chen, T. Yang, “SADAGRAD: strongly adaptive stochastic gradient methods,” Proceedings of the 35th International Conference on Machine Learning, vol. 80, pp. 913-921, 2018.

[2.23] X. Liu, E. Chow, M. Smelyanskiy, P. Dubey, “Efficient Sparse Matrix-Vector Multiplication on x86-Based Many-Core Processors,” Proceedings of the 27th international ACM conference on International conference on supercomputing, pp. 273–282, June 2013.

[2.24] G. Hinton, N. Srivastava, K. Swersky. [Online]. Available: http://www.cs.toronto.edu/~tijmen/csc321/slides/lecture_slides_lec6.pdf

[2.25] V. Bushaev, “Adam–latest trends in deep learning optimization,” 2018. [Online]. Available: https://towardsdatascience.com/adam-latest-trends-in-deep-learning-optimization-6be9a291375c

[2.26] T. Dozat, “Incorporating Nesterov Momentum into Adam,” International Conference on Learning Representations (ICLR) Workshop, Puerto, Rico, 2016.

[2.27] Y. Nesterov, “A method for unconstrained convex minimization problem with the rate of convergence o(1/k2),” Soviet Math Docl, vol. 269, pp. 543– 547, 1983.

[2.28] S. J. Reddi, S. Kale, S. Kumar, “On the convergence of Adam and beyond,” Conference on Learning Representations (ICLR), Vancouver, Canada, 2018.

[2.29] I. Loshchilov, F. Hutter, “Decoupled weight decay regularization,” In Proceedings of International Conference on Learning Representations, LA, USA, May 2019.

[2.30] J. Ma, D. Yarats, “Quasi-hyperbolic momentum and Adam for deep learning,” In Proceedings of International Conference on Learning Representations, LA, USA, May 2019.

[2.31] J. Lucas, S. Sun, R. Zemel, R. Grosse, "Aggregated Momentum: Stability through passive damping," In Proceedings of International Conference on Learning Representations, LA, USA, May 2019.

[2.32] T. Dozat, D. C. Manning, C. D, "Deep biaffine attention for neural dependency parsing," In Proceedings of International Conference on Learning Representations, Toulon, France, April 2017.

[2.33] S. Laine, T. Aila, "Temporal ensembling for semi-supervised learning," In Proceedings of International Conference on Learning Representations, Toulon, France, April 2017.

[2.34] F. Korzeniowski, "Experiments with AMSGrad," 2018. [Online]. Available: https://fdlm.github.io/post/amsgrad/

[2.35] G. Huang, Z. Liu, Q. K. Weinberger, L. Maaten, "Densely connected convolutional networks," In Proceedings of Conference on Computer Vision and Pattern Recognition, Hawaii, United States, July, 2017.

[2.36] Y. Wu, M. Schuster, Z. Chen, V. Q. Le, M. Norouzi, W. Macherey, J. Dean, "Google's neural machine translation system: bridging the gap between human and machine translation," 2016. [Online]. Available: arXiv: https://arxiv.org/abs/1609.08144.

[2.37] X. Chen, S. Liu, R. Sun, M. Hong, "On the convergence of a class of adam-type algorithms for non-convex optimization," In Proceedings of International Conference on Learning Representations, LA, USA, May 2019.

[2.38] L. N. Smith, "A disciplined approach to neural network hyper-parameters: part 1 – learning rate, batch size, momentum, and weight decay," 2018. [Online]. Available: arXiv :https://arxiv.org/abs/1803.09820

[2.39] L. N. Smith, "Cyclical learning rates for training neural networks," 2017. [Online]. Available: arXiv: https://arxiv.org/abs/1506.01186

[2.40] J. Bergstra, Y. Bengio, "Random search for hyper-parameter optimization," Journal of Machine Learning Research, vol. 13, pp. 281–305, 2012.

[2.41] G. Melis, C. Dyer, P. Blunsom, 2017. [Online]. "On the state of the art of evaluation in neural language models," Available: arXiv: https://arxiv.org/abs/1707.05589.

[2.42] S. Merity, N. K. Shirish, R. Socher, 2017. [Online]. "Regularizing and Optimizing LSTM Language Models," Available: arXiv: https://arxiv.org/abs/1708.02182

[3.1] S. Ruder, "An overview of G descent optimization algorithms," 2017. [Online]. Available: arXiv: 1609.04747.

[3.2] M. Nielsen, Neural Networks and Deep Learning, 2015. [Online]. Available: http://neuralnetworksanddeeplearning.com/index.html.

[3.3] S. M. R. Arnold, P. A. Manzagol, R. Babanezhad, I. Mitliagkas, N. L. Roux, "Reducing the variance in online optimization by transporting past gradients," 2019. [Online]. Available: https://arxiv.org/abs/1906.03532.

[3.4] M. Hardt, B. Recht, Y. Singer, "Train faster, generalize better: stability of stochastic gradient descent," International Machine Learning Conference (ICML), New York City, NY, USA, 2016.

[3.5] Y. Wu, M. Schuster, Z. Chen, et al, "Google's neural machine translation system: bridging the gap between human and machine translation," 2016. [Online]. Available: https://arxiv.org/abs/1609.08144.

[3.6] D. Choi, C. J. Shallue, Z. Nado, J. Lee, C. J. Maddison, G. E. Dahl, "On empirical comparisons of optimizers for deep learning," 2020. [Online]. Available: https://arxiv.org/abs/1910.05446.

[4.1] R. Gylberth, R. Adnan, S. Yazid, T. Basaruddin, "Differentially private optimization algorithms for deep neural networks," 9th International Conference on Advanced Computer Science and Information Systems (ICACSIS), Bali, Indonesia, 2017.

[5.1] Y. LeCun, C. Cortes, C. J.C. Burges, "The Mnist Database of handwritten digits." [Online]. Available: http://yann.lecun.com/exdb/mnist/.

[5.2] M. Tavallaee, E. Bagheri, W. Lu, A. A. Ghorbani, "A Detailed Analysis of the KDD CUP 99 Data Set," 2nd IEEE Symposium on Computational Intelligence for Security and Defense Applications (CISDA), Ottawa, ON, Canada, 2009.

[5.3] A. C. Wilson, R. Roelofs, M. Stern, N. Srebro, B. Recht, "The Marginal Value of Adaptive Gradient Methods in Machine Learning," 2018. [Online]. Available: arXiv: 1910.05446.

[5.4] N. Srivastava, G. Hinton, A. Krizhevsky, I. Sutskever, R. Salakhutdinov, "Dropout: A Simple Way to Prevent Neural Networks from Overfitting," Journal of Machine Learning Research, vol. 15, pp. 1929–1958, 2014.

[5.5] A. Krogh, J. A. Hertz, "A Simple Weight Decay Can Improve Generalization," Conference on Neural Information Processing Systems (NeurIPS), Denver, Colorado, USA, 1991.

[5.6] R. Ge, S. M. Kakade, R. Kidambi, P. Netrapalli, "The Step Decay Schedule: A Near Optimal, Geometrically Decaying Learning Rate Procedure For Least Squares,"

Conference on Neural Information Processing Systems (NeurIPS), Vancouver, Canada, 2019.

[5.7] Kaggle, “Learn computer vision fundamentals with the famous MNIST data.” [Online]. Available: https://www.kaggle.com/c/digit-recognizer/discussion/61480.

[5.8] Y. Pei, “Linear Principal Component Discriminant Analysis,” IEEE International Conference on Systems, Man, and Cybernetics, Budapest, Hungary, January 2016.

[5.9] S. Rawat, A. Srinivasan, V. R, “Intrusion detection systems using classical machine learning techniques versus integrated unsupervised feature learning and deep neural network.” [Online]. Available: arXiv:1910.01114.

[5.10] Y. Ding, Y. Zhai, “Intrusion Detection System for NSL-KDD Dataset Using Convolutional Neural Networks,” Proceedings of the 2018 2nd International Conference on Computer Science and Artificial Intelligence, Shenzhen, China, 2018, pp. 81–85.

[5.11] X. Zhang, J. Ran; J. Mi, “An Intrusion Detection System Based on Convolutional Neural Network for Imbalanced Network Traffic,” IEEE 7th International Conference on Computer Science and Network Technology (ICCSNT), Dalian, China, 2020.

[6.1] H. Sellouk, “Matrix based back-propagation.” [Online]. Available: https://medium.com/@hindsellouk13/matrix-based-back-propagation-fe143ce2b2df

[6.2] D. B. Davidson, “EM programmer's notebook,” IEEE Antennas and Propagation Magazine, vol. 52, June 2010.

[6.3] J. Ghorpade, J. Parande, M. Kulkarni, A. Bawaskar, “GPGPU processing in CUDA architecture,” 2012. [Online]. Available: arXiv: 1202.4347.

[6.4] Numpy. [Online]. Available: https://numpy.org.

[6.5] PyCUDA. [Online]. Available: https://documen.tician.de/pycuda/.

[6.6] Amazon SageMaker Autopilot, [Online]. Available: https://aws.amazon.com/sagemaker/autopilot/.

VITA

CHONGYA SONG

2015-2017	Master Student College of Engineering Florida International University Miami, Florida
2017	M.S., Computer Engineering Florida International University Miami, Florida
2017-2019	Research Assistant College of Engineering Florida International University Miami, Florida
2019-2021	Doctoral Candidate College of Engineering Florida International University Miami, Florida
	Teaching Assistant College of Engineering Florida International University Miami, Florida

PATENTS, PUBLICATIONS AND PRESENTATIONS

Song, C., Pons, A. (2021), *Systems and Methods for Network-Based Intrusion Detection* (US Patent Issued: Serial No. US 10, 911, 471 B1).

Song, C., Pons, A, Yen, K. (2021), *AG-SGD: Angle-based Stochastic Gradient Descent*. IEEE Access.

Song, C., Pons, A, Yen, K. (2020), *Optimizing Stochastic Gradient Descent Using the Angle Between Gradients*. IEEE International Conference on Big Data.

Song, C., Pons, A, Yen, K. (2020), *Sieve: An Ensemble Algorithm Using Global Consensus for Binary Classification*. AI, 1(2), 242-262.

Song, C., Yen, K. Pons, A, Liu, J. (2020), *Rank-Based Chain-Mode Ensemble for Binary Classification*. International Conference on Machine Learning and Applications

Song, C., Pons, A, Yen, K. (2018), *AA-HMM: An Anti-Adversarial Hidden Markov Model for Network-Based Intrusion Detection*. Applied Science, 8, 2421.

Song, C., Pons, A, Yen, K. (2016). *Building a Platform for Software-Defined Networking Cybersecurity Applications*, IEEE International Conference on Machine Learning and Applications.

Song, C., Li, A (2012). *QT Software Development*. New Construction Version of Modern Property Management.

Song, C., (April, 2018). *Cybersecurity Protection for Software-defined Networking Applying Machine Learning.* Presented at Florida Center for Cybersecurity Research Symposium, Tampa, Florida.

Song, C., (December, 2016). *Building a Platform for SDN-based NIDS*. Paper presented at IEEE International Conference on Machine Learning and Applications, Anaheim, California.