ΠΑΝΕΠΙΣΤΗΜΙΟ
ΘΕΣΣΑΛΙΑΣ

ΣΧΟΛΗ ΘΕΤΙΚΩΝ ΕΠΙΣΤΗΜΩΝ

ΤΜΗΜΑ ΠΛΗΡΟΦΟΡΙΚΗΣ ΚΑΙ ΤΗΛΕΠΙΚΟΙΝΩΝΙΩΝ

Αλγόριθμοι Βελτιστοποίησης στη Μηχανική Ευφυΐα

Νικόλαος Λιάπης

ΠΤΥΧΙΑΚΗ ΕΡΓΑΣΙΑ

ΥΠΕΥΘΥΝΟΣ

Κωνσταντίνος Κολομβάτσος
Επίκουρος Καθηγητής

Λαμία Σεπτέμβριος 2024

ΠΑΝΕΠΙΣΤΗΜΙΟ
ΘΕΣΣΑΛΙΑΣ

ΣΧΟΛΗ ΘΕΤΙΚΩΝ ΕΠΙΣΤΗΜΩΝ

ΤΜΗΜΑ ΠΛΗΡΟΦΟΡΙΚΗΣ ΚΑΙ ΤΗΛΕΠΙΚΟΙΝΩΝΙΩΝ

Αλγόριθμοι Βελτιστοποίησης στη Μηχανική Ευφυΐα

Νικόλαος Λιάπης

ΠΤΥΧΙΑΚΗ ΕΡΓΑΣΙΑ

ΥΠΕΥΘΥΝΟΣ

Κωνσταντίνος Κολομβάτσος
Επίκουρος Καθηγητής

Λαμία Σεπτέμβριος 2024

UNIVERSITY OF
THESSALY

SCHOOL OF SCIENCE

DEPARTMENT OF COMPUTER SCIENCE & TELECOMMUNICATIONS

Optimization Algorithms in Machine Intelligence

Nikolaos Liapis

FINAL THESIS

ADVISOR

Konstantinos Kolomvatsos
Assistant Professor

Lamia September 2024

«Με ατομική μου ευθύνη και γνωρίζοντας τις κυρώσεις [(1)], που προβλέπονται από της διατάξεις της παρ. 6 του άρθρου 22 του Ν. 1599/1986, δηλώνω ότι:

*1. Δεν παραθέτω κομμάτια βιβλίων ή άρθρων ή εργασιών άλλων αυτολεξεί **χωρίς να τα περικλείω σε εισαγωγικά** και χωρίς να αναφέρω το συγγραφέα, τη χρονολογία, τη σελίδα. Η αυτολεξεί παράθεση χωρίς εισαγωγικά χωρίς αναφορά στην πηγή, είναι λογοκλοπή. Πέραν της αυτολεξεί παράθεσης, λογοκλοπή θεωρείται και η παράφραση εδαφίων από έργα άλλων, συμπεριλαμβανομένων και έργων συμφοιτητών μου, καθώς και η παράθεση στοιχείων που άλλοι συνέλεξαν ή επεξεργάσθηκαν, χωρίς αναφορά στην πηγή. Αναφέρω πάντοτε με πληρότητα την πηγή κάτω από τον πίνακα ή σχέδιο, όπως στα παραθέματα.*

*2. Δέχομαι ότι η αυτολεξεί **παράθεση χωρίς εισαγωγικά**, ακόμα κι αν συνοδεύεται από αναφορά στην πηγή σε κάποιο άλλο σημείο του κειμένου ή στο τέλος του, είναι αντιγραφή. Η αναφορά στην πηγή στο τέλος π.χ. μιας παραγράφου ή μιας σελίδας, δεν δικαιολογεί συρραφή εδαφίων έργου άλλου συγγραφέα, έστω και παραφρασμένων, και παρουσίασή τους ως δική μου εργασία.*

3. Δέχομαι ότι υπάρχει επίσης περιορισμός στο μέγεθος και στη συχνότητα των παραθεμάτων που μπορώ να εντάξω στην εργασία μου εντός εισαγωγικών. Κάθε μεγάλο παράθεμα (π.χ. σε πίνακα ή πλαίσιο, κλπ), προϋποθέτει ειδικές ρυθμίσεις, και όταν δημοσιεύεται προϋποθέτει την άδεια του συγγραφέα ή του εκδότη. Το ίδιο και οι πίνακες και τα σχέδια

4. Δέχομαι όλες τις συνέπειες σε περίπτωση λογοκλοπής ή αντιγραφής.

Ημερομηνία: ……/…../20……

Ο – Η Δηλ.

(1) «Όποιος εν γνώσει του δηλώνει ψευδή γεγονότα ή αρνείται ή αποκρύπτει τα αληθινά με έγγραφη υπεύθυνη δήλωση
του άρθρου 8 παρ. 4 Ν. 1599/1986 τιμωρείται με φυλάκιση τουλάχιστον τριών μηνών. Εάν ο υπαίτιος αυτών των πράξεων
σκόπευε να προσπορίσει στον εαυτόν του ή σε άλλον περιουσιακό όφελος βλάπτοντας τρίτον ή σκόπευε να βλάψει άλλον, τιμωρείται με κάθειρξη μέχρι 10 ετών.»

ΠΕΡΙΛΗΨΗ

Στόχος της παρούσας διατριβής είναι να παρουσιάσει μια ολοκληρωμένη ανάλυση ορισμένων αλγορίθμων βελτιστοποίησης που χρησιμοποιούνται στην εκπαίδευση μοντέλων μηχανικής νοημοσύνης. Η μελέτη αυτή θα παράσχει το απαραίτητο υπόβαθρο για την κατανόηση των λόγων και των μεθοδολογιών που κρύβονται πίσω από την απόδοση κάθε αλγορίθμου σε ένα δεδομένο πλαίσιο. Η λειτουργικότητα των αλγορίθμων θα περιγραφεί αναλυτικά και θα καταγραφεί ως προς τους ρυθμούς σύγκλισής τους, οι οποίοι θα συζητηθούν στο πλαίσιο της κυρτής και μη κυρτής βελτιστοποίησης. Εκτός από τη θεωρητική διερεύνηση, οι αλγόριθμοι θα εφαρμοστούν σε συνελικτικά νευρωνικά δίκτυα, σε σύνολο δεδομένων του πραγματικού κόσμου και θα αξιολογηθεί η απόδοσή τους.

ABSTRACT

The objective of this thesis is to present a comprehensive analysis of some of the optimization algorithms utilized in the training of machine intelligence models. This study will provide the background necessary to understand the reasons and methodologies behind each algorithm's perfomance in a given context. The functionality of the algorithms will be described analytically, and they will be recorded in terms of their convergence rates, which will be discussed in the context of convex and non-convex optimization. In addition to the theoretical investigation, the algorithms will be applied to convolutional neural networks on real-world dataset, and their performance will be evaluated.

“As a rule, when many options are available, man’s actions are guided by the need to choose the best possible way. Human activity, indeed, implicates solving (conciously or unconciously) optimization problems. Moreover, many laws of nature are of a variational character, even if it is inappropriate in this case to speak of the existence of a purpose.”

- Boris Polyak

Table of Contents

Chapter 1

Introduction

1.1 History

Since antiquity, people have been interested in optimization problems. According to *Pythagoras* of Samos (c. 570-495 BCE), who influenced later philosophers with his ideas, "mathematics is the path to enlightenment and understanding of the cosmos" [1]. In the same way, mathematicians such as *Euclid* of Alexandria (c. 325–265 BCE), *Archimedes* of Syracuse (c. 287-212 BCE), *Zenodorus* (c. 200-120 BCE), *Heron* of Alexandria (c. 10-85 CE) and *Pappus* of Alexandria (c. 290-350 CE) worked on optimization problems [2]. In contrast to ancient times, the rapid increase in computing power described by *Moore's-Law* and the algorithmic efficiency of the last decade [3] have allowed machine intelligence to make significant progress in recent years. With the integration of artificial intelligence, the availability of vast quantities of data, and the enhanced computing power, algorithms can be applied to solve various tasks including optimization problems. As a result, scientific and technological advancement is occurring at an accelerated pace, bringing new knowledge and challenges to society. In this context, deep learning models and their new developed architectures have the need for efficient implication and the optimization is one of the ways to achieve this. As automation continue to advance in our era, it is beneficial to understand the underlying principles of the optimization algorithms.

1.2 Optimization Algorithms

In optimization there are three categories of algorithms distinguished by the type of information they use to find the optimum of a function:

- Zeroth-order methods
- First-order methods
- Second-order methods

The zeroth-ones, also known as derivative-free or black-box methods, are based on function evaluation, rather than gradient information. These methods are suitable for complex and high-dimensional optimization problems, they can handle noisy and non-differentiable functions, but are generally slower compared to gradient-based methods. The next category, first-order methods, which this thesis is focused on, typically use the first derivative (the gradient) information of the function to guide the optimization process. They have a small per-iteration cost and they are widely utilised in machine intelligence problems, especially in large scale problems. The final category, second-order methods, are built from a combination of first and second derivatives of the function (Hessian Matrix). They are typically more accurate, but more

computationally intensive than first-order methods. Specifically, in second-order methods, the problem is the managing of the size of the inverse Hessian matrix. This problem can be solved with methods that approximate the Hessian matrix.

1.3 Thesis-Structure

Chapter 2
Presents the meaning and the basic idea of optimization, the challenges of optimization and different methods that are used in machine learning.

Chapter 3
Focuses on the first-order methods based on Gradient descent and its variants explaining the main ideas of how they work, the characteristics and their time complexity.

Chapter 4
Presents the results from the optimizers testing and their perfomance.

Chapter 5
Offers concluding remarks and future directions.

Appendix A & B
For better presentation, some basic definitions, illustrations are included in *Appendix A and Appendix B*.

Chapter 2

Introduction to Optimization

Optimizing is the process of finding a better solution in a complex system, with or without given constraints. In the context of machine intelligence, optimization algorithms are utilised in order to minimize the objective function, trying iteratively to find the optimal values of the model parameters. The challenge often lies in identifying an algorithm that can reach a possible solution, such as a stationary point[1] x^*. This is followed by the objective of accelerating the convergence of the algorithm and ensuring that it reaches a solution with a low objective value, in this case, the global minima. In practice, the goal may not always be to find the best solution for various reasons, including computational complexity, uncertainty and real-world constraints that limit the feasible solution space. Nevertheless, the final goal of a model is to make a function approximation which leads to the desired results. This approximation is evaluated using specific metrics that measure the model's perfomance. However, this goal is not solely dependent on the optimization techniques and algorithms employed.

2.1 Optimization in Machine Intelligence

In machine intelligence, there are two main branches of continuous optimization[2] that are widely used: unconstrained and constrained optimization. However, unconstrained optimization is encountered in many standard machine learning tasks, whereas constrained optimization is employed when the problem exhibits specific limitations or the structure of the solution space must be respected. For example, this occurs in Support Vector Machines (SVMs), where the margin constraints are fundamental to the problem formulation.

Definition 1
Unconstrained optimization involves the task of minimizing (or maximizing) a function $f(x)$ without any restrictions on the domain of x. Formally, we seek to find:

$$\min_{x \epsilon R^n} f(x)$$

$where\ f: R^n \rightarrow R$ is a continuously differentiable function.

Definition 2
Constrained optimization involves optimizing the function $f(x)$ subject to constraints on x:

$$\min_{x \epsilon R^n} f(x)$$

1 In optimization a stationary point of a function *f*, $where\ f: R^n \rightarrow R$ is considered a point where the gradient is zero ($\nabla f(x^*) = 0$). The stationary point can be a local minimum or a local maximum or a saddle point.

2 Continuous optimization involves finding the optimal value of a function, where variables can take on any value within a continuous range.

$$\text{Subject to } g_i(\text{x}) \leq c_i$$
$$h_j(x) = d_j$$

Where i =1,..,n are inequality constraints and j=1,..,m are equality contraints.

In machine learning, particularly in supervised learning during the training phase of a model, an overall loss function L(θ), or empirical risk, is typically implemented to achieve an objective.

$$L(\theta) = \frac{1}{n}\sum_{i=1}^{n} L\left(f(x_i;\theta), y_i\right)$$

Where, θ represents the parameters of a differentiable function f, n is the size of samples and L symbolizes the loss function that quantifies the difference between the predicted label $y_i' = f(x_i;\theta)$ and the true label y_i.

The objective of this formula is achieved by perfoming empirical risk minimization which finds the optimal solution $\theta^* \in \arg min\, \text{L}(\theta)$. In this unconstrained optimization the first and second derivatives are crucial for characterizing and computing the optimal solutions and more specifically the local minima and the global minima.

Definition 3

"A vector x* is an unconstrained local minimum of a function $f: \mathbb{R}^n \to \mathbb{R}$ if there an $\varepsilon > 0$ such that [4] :

$$f(\text{x}^*) \leq f(x), \quad \forall x \in \mathbb{R}^n \quad \text{with } \|x - x^*\| \leq \varepsilon$$

This means that x* is a local minimum if, within some neighborhood of radius ε, the value of the function at x* is less than or equal to the value of the function at any other point in that neighborhood.

A vector x* is an unconstrained global minimum of f if [4]:

$$f(\text{x}^*) \leq f(x), \quad \forall x \in \mathbb{R}^n$$

This means that x* is a global minimum if the value of the function at x* is less than or equal to the value of the function at any other point in the entire domain $\mathbb{R}^n$."

In general, the finding of the global minimum is regarded to belong to the class of NP-hard problems. However, for certain classes of functions f, there are some desirable properties with strong theoretical guarantees that allow efficient optimization, using algorithms such as gradient descent. These functions are referred to as convex functions. The fundamental concept is that when f is convex, there is an equivalence between $\|\nabla f(x)\| = 0$ and the fact that x $\in$ $\text{argmin}_{x \in R^d} f(x)$. For a more comprehensive understanding of the terms, definitions are provided in Appendix A.

The main algorithms for finding local and global minima using gradients in machine learning are gradient descent and its variants, which are described in the next chapter. The classic gradient descent (Vanilla gradient descent) algorithm finds applications in various optimization problems. Some notable applications include:

- **Linear Regression**: In linear regression, the gradient descent algorithm tries to find the optimal values for the regression coefficients that minimize a cost function such as the sum of squared errors.
- **Logistic Regression**: In logistic regression the minimizing of the logistic loss function using the gradient descent leads to find the optimal parameters that maximize the likelihood of the observed data.
- **Support Vector Machines**: Gradient descent is used in support vector machines to find the optimal hyperplane that separates data in different classes.
- **Neural Networks**: In neural networks, gradient descent is employed to update the weights and biases during the training process. This enables the network to learn and have better performance.

In optimization, algorithms such as gradient descent and its variants use the derivative information of the objective function *f(x)* to calculate the search direction. The search direction is crucial because it determines the direction in which the algorithm move from each iteration point to find a local minimum. Algorithms like exact line search need the search direction to find the optimal step size (see in Appendix A).

2.2 Optimization Challenges

One of the main challenges in optimization is the existence of saddle points. Saddle points need to be avoided because their slopes have different directions, meaning that they have characteristics of both positive and negative curvature, where the gradient is zero. This can cause first order methods to get stuck or converge very slowly in suboptimal solutions. Therefore, it is important to navigate through them efficiently. Some ways to escape from these points are by injecting random perturbation or using the negative eigenvector of Hessian [5].

Moreover, local minima don't help the algorithms to find the best possible model parameters during optimization. Especially in non-convex optimization problems, local minima can trap the algorithms. Therefore, there is no general guarantee of finding a better local minimum or the global minimum. As a result, the optimization can be highly dependent on the initialization and algorithm parameters. In general, while first-order methods have good convergence guarantees for convex problems, these guarantees are generally weaker for non-convex problems.

Another challenge is the ill-conditioned Hessian matrix problems, which can lead to slow convergence and difficulty in finding the optimal solution. First order methods do not explicitly use the Hessian matrix, but they are affected by issues related to ill-conditioning in the

optimization landscape. When the Hessian matrix is ill-conditioned, it implies a large disparity between its eigenvalues, leading to steep regions with high curvature and flat regions with low curvature in the loss surface. This makes optimization challenging because steep regions require small step sizes to avoid overshooting, while flat regions require larger step sizes to make progress. Adaptive learning algorithms such as Adam and RMSprop, described in the next chapter, are often used to mitigate this problem.

An additional problem in optimization and very common in deep learning training is vanishing and exploding gradients. Especially in deep learning, the problem of vanishing or exploding gradients can occur, where gradients become extremely small or large as they propagate through many layers. This can lead to learning difficulties, and the methods that are typically used to address these issues are proper activation functions, adaptive optimization algorithms, batch normalization, residual connections and gradient clipping.

Furthermore, machine intelligence models often require large datasets. This means that, substantial computational resources are necessary in order to manage the number of the model parameters and the complexity of the algorithms. Thus, limited compute resources pose a significant challenge and can limit the number of experiments that can be performed, slowing down the research and development process. Consequently, efficient algorithms and hardware accelerators such as GPUs and TPUs are essential to overcome these limitations.

One of the most important parameters that needs to be properly adjusted in the optimization process is the learning rate. Besides the ability of adaptive algorithms to adapt the learning rate of each optimization problem, and the standard method of a fixed learning rate, there are some other strategies that adjust the learning rate during the training of machine learning models and they are called learning rate schedules. These schedules are designed to improve convergence and model performance. Some of these learning rate scheduling techniques are:

- **Step Decay**: This method reduces after a specific number of epochs, the learning rate. It is used especially when a high initial learning rate is required for fast learning.
- **Exponential Decay**: This technique decays the learning rate exponentially over time. It is used when a smooth and continuous decay is needed, and it doesn't allow the learning rate to become too small very quickly, unlike step decay.
- **Polynomial Decay**: In this method the learning rate is decreased following a polynomial function and can be more stable compared to exponential decay.
- **Cosine Annealing**: This technique gradually decrease the learning rate following a cosine function. It can be combined with restarts.
- **Warmup**: This method starts with a very low learning rate and gradually increases it to the the desired value [6]. It is also combined with other learning rate scheduling strategies, thereby enhancing the stability and convergence of the training process.
- **Cyclical Learning Rates**: This technique cycles the learning rate between a minimum and a maximum value, following a triangular or sinusoidal pattern.

By periodically increasing the learning rate, it helps to escape from local minima.

- **One-Cycle Policy**: This method consists of a single cycle of increasing and then decreasing the learning rate, sometimes finishing with an even lower rate than where it started. This policy helps to escape poor local minima and then settle in a better local minima with a lower learning rate.
- **Reduction on Plateau (ReduceLROnPlateau)**: This technique monitors a specific metric during training, usually validation loss. If, after a certain number of epochs, there is no improvement in the metric, the learning rate is reduced by a chosen factor to allow the model to learn. If the metric stops improving, it may indicate a plateau, and that's why the model continues to learn at a slower rate, increasing the likelihood of finding a better minima. Otherwise, the model may exhibit overfitting to the training data or oscillate missing out a more optimal solution.

The challenge of the above scheduling techniques is to use the one that fits better on the specific training scenario of the model with the desired rate of learning rate reduction. The scheduling techniques can also be combined with adaptive algorithms. The choice of the most appropriate technique depends on the specific characteristics of the training process and the model's behaviour.

The last important challenge in deep learning is the choice of the best combination between the architecture of the model and the optimizer. The architecture affects the perfomance of the optimizer. For example, the deep neural networks with many layers have bigger complexity than other models. In this way, the optimization process is more challenging because the optimizer need to navigate a more complex loss landscape with possible saddle points and local minima. Thus, the choice of optimizer for a specific architecture require testing and tuning.

Chapter 3

First-Order Optimization methods

First-Order optimization methods use the derivative information of the objective function to guide the optimization process. This chapter presents an analysis of the fundamental principles and the operation of the main algorithms.

3.1 Gradient Descent and its Variants

The earliest known reference to gradient methods dates back to *Augustin-Louis Cauchy* (1789-1857) who discussed gradient methods in his work on calculus and optimization [7]. Since that time, the algorithm has become a widely used tool in machine intelligence and is commonly used in the optimization of neural networks. Specifically, what it does is that it iteratively adjusts parameters leading to the direction of the steepest slope. But, because the gradient points towards the direction of steepest ascent, the negative term is introduced to ensure that it moves downhill, in order to minimize the function. Mathematically, the parameters are updated and calculating the derivative they give the slope in every point. In this way, the algorithm makes small changes in the input (weights, biases) to obtain the corresponding change in the output (predicted value). This section, will present the theoretical background of the three variants of the algorithm.

Batch Gradient Descent (BGD)

The first and classic version of gradient descent where the entire training dataset is used to compute the gradient of the cost function at each epoch is called Batch Gradient Descent. It is widely used in linear regression and logistic regression, but unlike the next two variants, not so much in deep learning where the models have many parameters and the training set is typically large. This algorithm is particularly useful for convex optimization, because in the case of a convex function, it is implied that any local minimum is also a global minimum (Appendix A). Specifically, for a differentiable function f, an initial point x_0 is defined by the following update rule:

$$x_{t+1} = x_t - n_t \nabla F(x_t), \qquad t = 0,1, \dots$$

Where x_{t+1} is the updated state of parameters after applying the above update rule, x_t is a parameter at iteration t, which represents the current state of the parameters being optimized (e.g. weights and biases), the value n_t is the *learning rate* or otherwise the *step size* and $\nabla F(x_t)$ is the gradient of the loss function for the *t*-th iteration. This gradient is a vector of partial derivatives and gives the rate of change of the function. In general, the learning rate is a crucial hyperparameter. It determines the rate at which the parameters are updated. If it is small, the algorithm will converge slowly, but if it is too large, the algorithm may oscillate

around the minimum or may even diverge. There are a number of methodologies that can be employed in order to select an optimal step size. Furthermore, there are techniques that are specifically designed for the purpose of finding the optimal step size in the context of optimization. These techniques are the backtracking line search and the exact line search which are described in Appendix A.

Algorithm: BGD

Inputs: Cost function J(x), learning rate n, iterations N

1. Initialize: Random x
2. **For** i = 1 to N
3. Compute the gradient with respect to x for all the data points:
4. gradient $= \nabla \mathbf{x}\, \boldsymbol{\Sigma}\, \boldsymbol{J}(\boldsymbol{x})$
5. Update the variable **x = x – n * gradient**
6. **End For**
7. **Return** model variable **x**

The upper bounds[3] on the convergence rate with the optimal step sizes of gradient descent for each property of the objective function are described by the following theorems [8] :

Theorem 1

" Let $f: \mathbb{R}^n \to \mathbb{R}$ be convex and L-Lipschitz. For T steps with step size:

$$n = \frac{\|x - x_*\|_2}{L\sqrt{T}}$$

Then the following holds:

$$f\left(\frac{1}{T}\sum_{t=1}^{T} x_t\right) - f(x_*) \leq \frac{\|x - x_*\| L}{\sqrt{T}} \text{ ,,}$$

Theorem 2

" Let $f: \mathbb{R}^n \to \mathbb{R}$ be convex and β-Smooth with step size n=1/β, then it holds:

$$f(x_t) - f(x_*) \leq \frac{\|x - x_*\|^2}{T-1} \text{ ,,}$$

In addition to being smooth, if f is strongly convex then there is a geometric decay which is 1-k, where k is the conditioning of the proplem and is equals to: $k = \frac{\alpha}{\beta}$ [4].

Theorem 3

" Let $f: \mathbb{R}^n \to \mathbb{R}$ be α-strongly convex and β-smooth. Then, gradient descent with step size η = 2 / α+β satisfies:

$$f(x_t) - f(x_*) \leq \frac{\beta}{2}\left(1 - \frac{\alpha}{\beta}\right)^{\mathrm{T}} \|x - x_*\|^2 \text{ ,,}$$

[3] An upper bound on the convergence rate is a valuable tool for the assessment of an algorithm's efficiency.

[4] Where α and β are, respectively, the strong convexity and the smothness constants.

TABLE I
CONVERGENCE RATE OF GRADIENT DESCENT

Convex and L-Lipschitz	$O(1/\sqrt{T})$
Convex and β-Smooth	$O(1/T)$
α-Strongly Convex and β-Smooth[5]	$O(e^{-T})$

According to the table, Lipschitz convex functions have a sublinear[6] convergence rate $O\left(1/\sqrt{T}\right)$, which is slower than the rate of smooth functions, but is useful for functions that are not differentiable at their minimum, such as subgradient[7] functions. Moreover, the best upper bound for gradient descent is achieved with the strong convexity and smoothness conditions which give exponential convergence[8].

Based on convex optimization problems, gradient descent guarantees convergence to the global minimum given a sufficient learning rate. However, in non-convex optimization, the algorithm may get trapped in saddle points or plateaus or converge to local minimum instead of the global minimum. As mentioned before, when the dataset is large is not so efficient to use this algorithm, because it uses all the training data to compute the cost function. Consequently, the computational expense and time required to compute are considerable. In such cases, it is preferable to utilize alternative optimization algorithms.

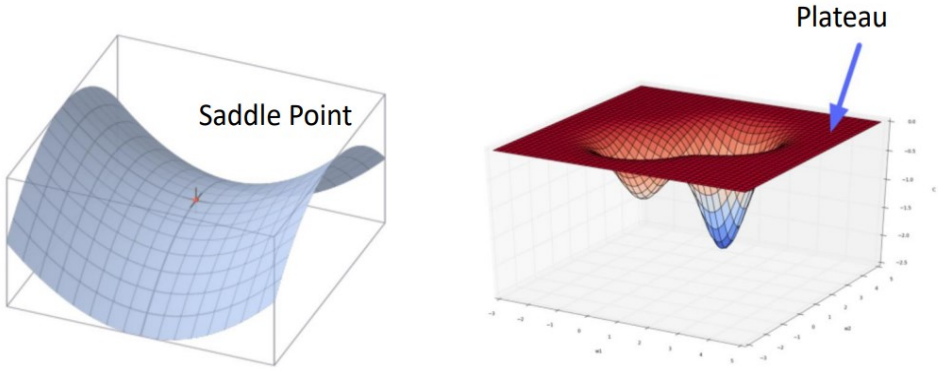

Fig. 1. Saddle point and Plateau as illustrated in [9].

[5] Under certain conditions $(1-k)^T$ can approximate e^{-T}. This is especially true when k is small.

[6] Sublinear means that the the rate at which an algorithm approaches the optimal solution decreases as the number of iterations increase. An iterative algorithm has sublinear convergence if the error ε at iteration κ satisfies: $\varepsilon_k \leq \frac{c}{k^\rho}$ for constants c, ρ.

[7] Subgradient is a generalization of the gradient concept to convex functions, which may not be differentiable (Appendix A).

[8] Exponential convergence is often referred to as linear convergence because the error decreases in a geometric progression over iterations in the context of iterative optimization algorithms. Suppose $\varepsilon_k = f(x_k) - f(x^*)$, if the error decreases as: $\varepsilon_{k+1} \leq \rho\varepsilon_k$, with constant 0<ρ<1 this implies that: $\varepsilon_k \leq \rho^k \varepsilon_0$.

Stochastic Gradient Descent (SGD)

Stochastic Gradient Descent selects a random mini-batch of training samples instead of processing the entire dataset during each iteration. In particular, the update rule for SGD is:

$$x_{t+1} = x_t - n_t \nabla F_i(x_t), \qquad t = 0,1, \ldots$$

where $F_i(x)$ represents the aggregate of the training loss for a given mini-batch.
The learning rate n_t should be properly tuned in order to guarantee efficient convergence without oscillations or divergence and it is common to use a diminishing step size.
In summary, the way the algorithm works is that at the beginning of each epoch, the training samples are randomly mixed and divided into multiple mini-batches. Then, at each iteration the gradient is computed and an update of weights is perfomed while one data instance is loaded into memory.

Algorithm: SGD

Inputs: Cost Function J(x), learning rate n, iterations N

1. Initialize: Random x
2. **For** i = 1 to N
3. Shuffle the data points
4. **For** each data instance (x_i, y_i)
5. Compute the gradient on a training instance (x_i, y_i):
6. gradient = $\nabla \mathbf{x}\, \boldsymbol{J}(\boldsymbol{x}, \boldsymbol{x_i}, \boldsymbol{y_i})$
7. Update parameters: **x = x – n * gradient**
8. **End For**
9. **End For**
10. **Return** model variable **x**

SGD is a computationally efficient method, although in theory it has a lower convergence rate (TABLE II). In practical terms, however, it is faster than gradient descent because it performs more iterations in the same amount of time. Moreover, due to its computational efficiency, it requires less memory per iteration and allows for the effective utilization of parallel computing resources, making it feasible for typically large datasets. However, the algorithm is characterized by high levels of noise and variance due to its stochastic nature, the use of small subsets and the fact that we do not take the average gradient over the entire data set. In particular, the noise causes fluctuations in the objective function, which leads to slower convergence, but sometimes this can help, especially in deep learning models, to obtain potentially better local minima, escape saddle points, and improve generalization capabilities. Consequently, the same step size will oscillate around the optimum. This is known as the Noise Ball effect. To avoid bouncing around the minimum, a decaying step size is used[9]. In terms of the updating cost of SGD, it is independent of the size of the dataset and can reach linear convergence. When applied to a convex function the upper bound is $\mathbf{O}(\mathbf{1}/\sqrt{\mathbf{T}})$ after T iterations and under strong convexity it is $\mathbf{O}(\mathbf{1}/\mathbf{T})$.

[9] In this case, the algorithm uses a decreasing step size $n_t = \frac{n_0}{t}$.

TABLE II

COMPARISON OF GRADIENT AND STOCHASTIC GRADIENT DESCENT CONVERGENCE RATES

Algorithm	GD	SGD
Convex	$O(1/\sqrt{T})$	$O(1/\sqrt{T})$
Convex and β-smooth	$O(1/T)$	$O(1/T)$
Strongly Convex	$O(e^{-T})$	$O(1/T)$

Mini-batch Gradient Descent (MBGD)

This is another variant that uses also a small subset of training examples at a time (mini-batch), but operates on small batches of examples with batch size greater than one. As a result, it has reduced variance compared to SGD (fig. 2) and more stable convergence, balancing the computational efficiency of SGD and the stability of gradient descent as it seems in fig. 3.

MBGD requires tuning of the batch size parameter. According to the authors of [10], " a batch size between 2 and 32 is suitable for optimal performance in deep neural networks, although larger batch sizes may be advantageous when parallelism is a priority ". In particular, if the batches are small, additional noise is added to the training process, which reduces the generalization error. While, in the case of larger batches, an improvement in stability and a reduction in variance are observed, but with a slower convergence towards the solution. The resulting accuracy is enhanced and the gradient error is diminished. In terms of generalization, the large batch size can maintain the perfomance.

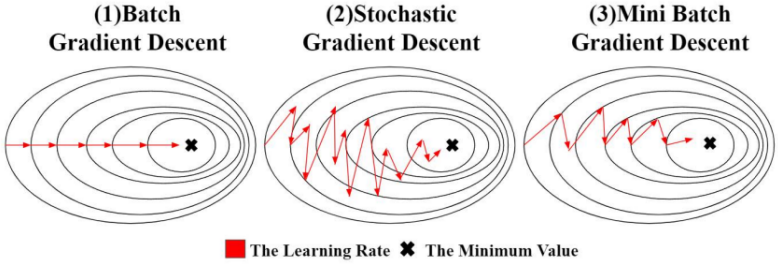

Fig. 2. Differences in training using Gradient Descent variants as illustrated in [11].

Algorithm: MBGD

Inputs: Cost Function J(x), learning rate n, iterations N, mini-batch size b

1. Initialize: Random x
2. **For** i = 1 to N
3. Shuffle the data points
4. Split the data into B mini-batches of size b:
5. **For** each mini-batch B
6. Compute the gradient with respect to x on the mini-batch B:
7. gradient $= \mathbf{1/b} * \nabla \mathbf{x} \sum \boldsymbol{J}(\boldsymbol{x}; \boldsymbol{B})$
8. Update parameters **x = x – n * gradient**
9. **End For**
10. **End For**
11. **Return** model variable **x**

MBGD is capable of handling data that is not uniformly distributed, as it processes a batch size of training examples in each iteration, thereby creating a more balanced representation of different classes within each mini-batch. However, it requires careful construction of mini-batches and potentially additional techniques such as data shuffling, stratified sampling, over-sampling and under-sampling.

The convergence rate of MBGD is between that of SGD and BGD, and it achieves linear convergence in strongly convex functions with well-chosen batch sizes and learning rates.

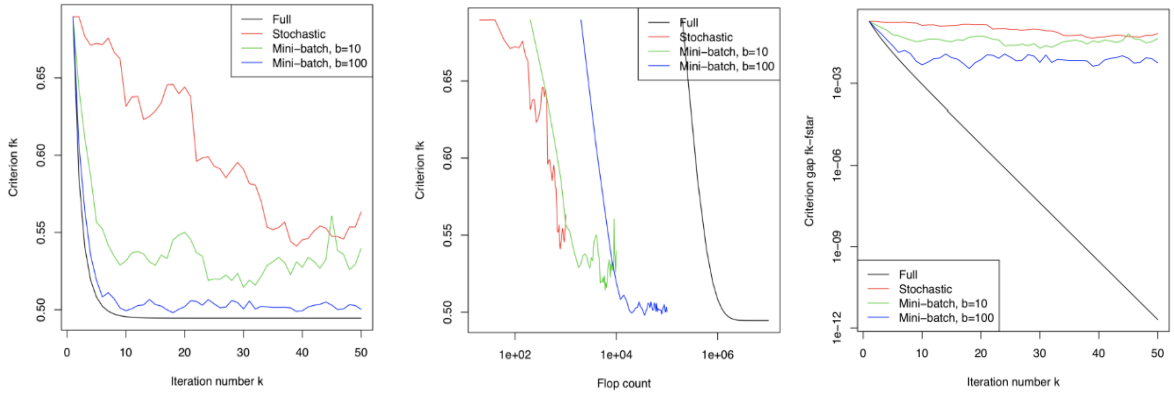

Fig. 3. Convergence rates between gradient descent, SGD and mini-batch [12].

In the above left figure it is described how each method reduces the objective function over time. The middle figure, which focuses on computational efficiency, shows convergence behavior relative to computational cost and the right figure provides a view of how each method converges to the optimal solution, especially when dealing with large differences in criterion. In the preceding algorithms, the presence of noise results in a reduction in the rate of convergence of the algorithm. The subsequent algorithms try to address this issue.

3.2 Acceleration Methods

These algorithms, as the name implies, are employed because they have a larger step size than the gradient descent algorithm, and thus achieve better convergence rates under certain specific conditions. The main idea here is that the previous gradients influence the current update, particularly in regard to its future trajectory. The newly introduced term, the momentum term, is of pivotal importance for the performance of these methods.

Momentum

Momentum is one of the most popular algorithms for large-scale machine learning problems. It is usually referred to as Heavy Ball (HB) method, which was introduced by Boris T. Polyak [13] and is considered as a simple type of momentum methods. The core idea behind the algorithm is that it uses an accumulated velocity vector that represents an exponential moving average of all of the gradients influenced by past gradients. The method can be applied to gradient descent, but in deep learning it is preferred to be combined with batch methods. Specifically, momentum uses the previous two iterates when computing the next one. This helps to smooth the noise of the SGD, and in particular is designed to speed up the convergence by dealing with high-dimensional spaces, small and noisy gradients [14]. Consequently, the method attempts to address the issue of variance, reducing the oscillations, and allows to use a

larger step size in regions with low curvature, thereby achieving better convergence and stability.

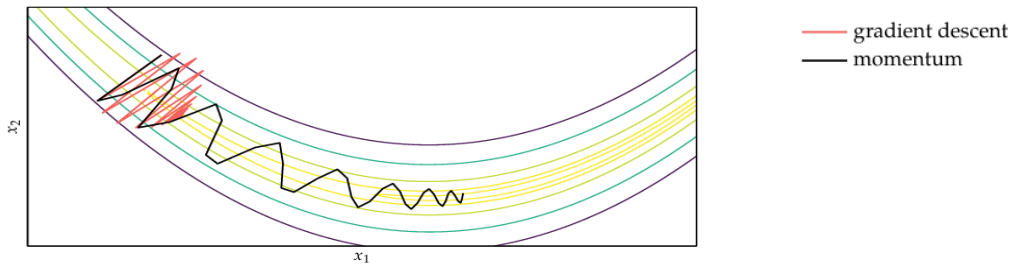

Fig. 4. Gradient descent and the momentum method compared on the Rosenbrock function as illustrated in [15].

The momentum update rule is:

$$v_{t+1} = \beta v_t - n\nabla f(x_t)$$

$$x_{t+1} = x_t + v_{t+1}$$

where v_{t+1} is the velocity at iteration t+1 and β is the momentum co-efficient which is typically between 0 and 1. Some studies suggest the values " 0.5, 0.9, and 0.99 as common settings " for this value [14] [16]. The largest step size when using momentum is described by the formula $\frac{n\|g\|}{1-\beta}$. This means that if the gradient g remains constant, the slope won't change, so the algorithm will accelerate until it reaches the terminal velocity. If β=0.9, the terminal velocity is equal to 1/(1-0.9) = 10 . Consequently, the rate of change is accelerated ten times faster the gradient descent method.

The interpretation of Polyak's heavy ball method rule is [17]:

$$x_{t+1} = x_t - n\nabla f(x_t) + \beta(x_t - x_{t-1})$$

where $x_t - x_{t-1}$ is the difference between the current and the previous position and β is the momentum co-efficient that heavy ball method use to determine the descent direction.

Algorithm: Momentum

Inputs: Cost Function J(x), learning rate n, momentum co-efficient ß

1. Initialize: Random x
2. Initialize: Δv=0
3. **For** i = 1 to N
4. Shuffle the data points
5. Compute the gradient $\mathbf{g = \nabla x}\, J(x, x_i, y_i)$
6. Update term **Δv=β*Δv+(1-β)*g**
7. Update parameters **x=x-n*Δv**
8. **End For**
9. **Return** model variable **x**

In a paper [18], it is proved a global sublinear convergence guarantee for HB for convex and smooth functions. While, Polyak proved that the HB method has linear convergence when minimizing strongly convex quadratic functions for good tuning of the step size n and the

coefficient value β. As a result, the algorithm exhibits a faster convergence rate than plain gradient descent.

Theorem 4 [19]
" Let f be a quadratic function which is β-smooth and α-strongly convex. For:

$$n = \frac{1}{\sqrt{\alpha\beta}}, \qquad \gamma = \left(\frac{\sqrt{\beta}-\sqrt{\alpha}}{\sqrt{\beta}+\sqrt{\alpha}}\right)^2$$

There exists a constant C such that for any t ∈ ℕ,

$$f(x_t) - f(x_*) \leq Ct^2 \left(\frac{\sqrt{\beta}-\sqrt{\alpha}}{\sqrt{\beta}+\sqrt{\alpha}}\right)^2 \|x - x_*\|^2 \text{ "}$$

The above theorem means that the optimal rate of convergence for strongly convex quadratic functions is $O\left(\left(1-\sqrt{\frac{a}{\beta}}\right)^T\right)$ but it doesn't work for all strongly convex and smooth functions. For example, there is a study [20] that shows that the heavy method fails to converge for certain strongly-convex and smooth functions and another that shows that there are simple quadratic problem instances that do not improve the convergence speed of SGD [21]. Yet another paper, reports that the convergence rate of Stochastic Gradient Descent with Momentum (SGDM) is as fast as SGD for smooth objectives under both strongly convex and nonconvex settings.

Nesterov Accelerated Gradient Descent

One other type of momentum methods is Nesterov's accelerated gradient (NAG), which have been found by Yurii Nesterov. The method has been enhanced through the utilisation of a lookahead step and the incorporation of the momentum term. In contrast to the Polyak algorithm, the gradient is evaluated at a future approximated point (after the current velocity is applied), and not in the current parameters x_t, which helps the algorithm to converge faster and more smooth because of the anticipation of future gradients which are used for early correction. Moreover, the algorithm converges for general convex functions, not only for some carefully built convex optimization problems.

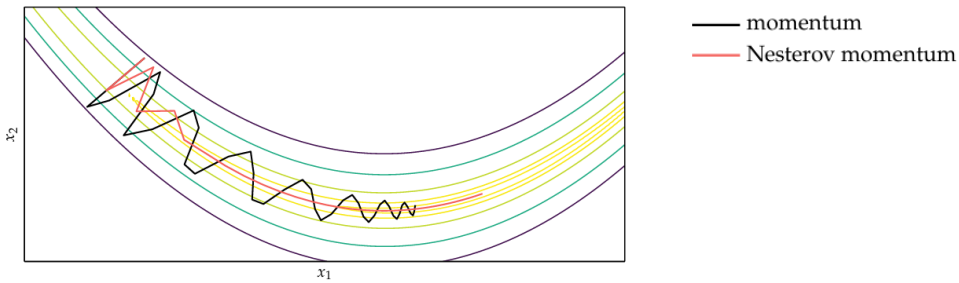

Fig. 5. Nesterov and the momentum method compared on the Rosenbrock function as illustrated in [15].

The iteration formula of NAG consists of the following steps :

$$v_{t+1} = \beta v_t - n\nabla f(y_t)$$

$$y_t = x_t + \beta v_t$$

$$x_{t+1} = x_t + v_{t+1}$$

Combining these sequences as one the update rule of the algorithm is [19]:

$$x_{t+1} = x_t - n\nabla f(x_t + \beta(x_t - x_{t-1})) + \beta(x_t - x_{t-1})$$

where, $y_t = x_t + \beta(x_t - x_{t-1})$ is the lookahead point at which the gradient is calculated and $\beta(x_t - x_{t-1})$ is the momentum term.

Algorithm: Nesterov

Inputs: Cost Function J(x), learning rate n, momentum co-efficient β

1. Initialize: Random x
2. Initialize: Δv=0
3. **For** i = 1 to N
4. Shuffle the data points
5. Compute lookahead $\hat{x}$ = **x +β*Δv**
6. Compute the gradient **g = ∇x** $J(\hat{x}, x_i, y_i)$
7. Update term **Δv=β*Δv-n*g**
8. Update parameters **x=x-n*Δv**
9. **End For**
10. **Return** model variable **x**

The following theorems describe the convergence rates of the algorithm in each case [19]:

Theorem 5

" Let f be an α-strongly convex and β-smooth function, then according to the iteration formula, for all $t \in \mathbb{N}$ with n=1/β and $\beta=\frac{\sqrt{\beta}-\sqrt{a}}{\sqrt{\beta}+\sqrt{a}}$:

$$f(x_t) - f(x_*) \leq 2\left(1 - \sqrt{\frac{\alpha}{\beta}}\right)^T \left(f(x) - f(x_*)\right)"$$

According to the above theorem, with the specific choice of parameters Nesterov's method converges at a linear rate with an upper bound complexity of $O\left(\left(1 - \sqrt{\frac{\alpha}{\beta}}\right)^T\right)$, which is true for all stronlgy convex functions and not just the quadratic ones as it happens in the heavy ball method.

While, in the case of smooth convex functions the convergence rate of Nesterov's method is $O\left(\frac{1}{T^2}\right)$ according to the next theorem:

Theorem 6

" Let f be an β-smooth convex function, with n=1/β, then for all $t \in \mathbb{N}$:

$$f(x_t) - f(x_*) \leq \frac{2L}{(T+1)^2} \|x - x_*\|^2 \text{ "}$$

The importance of NAG algorithm is based to the fact that it achieves the optimal convergence rates among gradient methods for both strongly convex and smooth convex functions. The table below presents the rates in contrast to the gradient descent algorithm.

TABLE III
COMPARISON OF GRADIENT DESCENT AND NESTEROV'S ALGORITHM CONVERGENCE RATES

Algorithm	GD	NAG
Smooth Convex Functions	$O(1/T)$	$O(1/T)^2$
Strongly Convex and Smooth Functions	$O\left(e^{-\frac{T}{k}}\right)$	$O\left(e^{-\frac{T}{\sqrt{k}}}\right)$

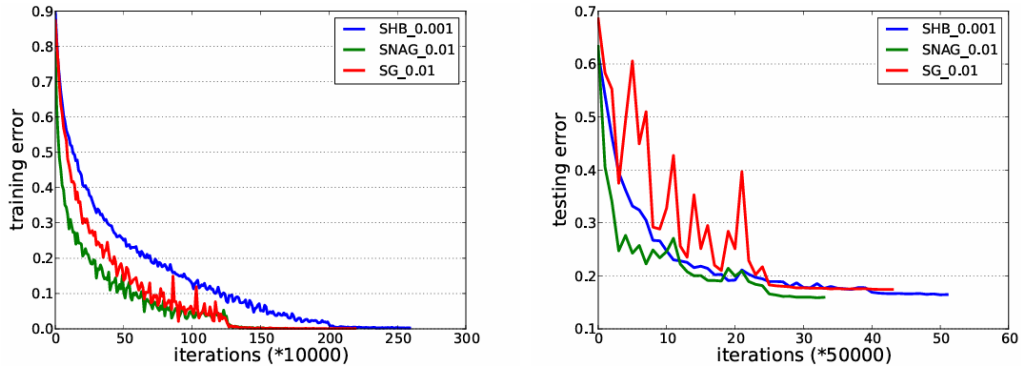

Fig. 6. Accelerated methods and SGD perfomance as illustrated in [21]

3.3 Adaptive Gradient Methods

The learning rate is an important hyperparameter for optimization algorithms. As discussed in previous sections, if the step size is very small, the updates will take a lot of time to update the parameters. Moreover, in some cases where the data is high-dimensional or sparse, the constant learning rate may not help the model converge because the learning update is inefficient. Therefore, it is necessary to dynamically change the learning rate to achieve convergence. That's the main goal of adaptive gradient methods. They aim to have a better efficiency than the other gradient-based optimization algorithms by adjusting the learning rate for each parameter. These algorithms approximate the diagonal elements of the Hessian matrix by calculating the squared gradients. In practical applications, the squared gradients can be used as a surrogate for curvature, providing a way to understand how steep or flat the loss function is. This approach can be beneficial in many scenarios, particularly in deep learning, where the calculation and utilisation of the full Hessian matrix is a computationally expensive process.

Adagrad (Adaptive Gradient)

Adagrad is an optimization algorithm that adjusts the learning rate for each parameter based on the sum of squared gradients calculated during training. This means that the historical gradients determine how small or large the learning rate is. Thus, if the parameters have larger historical gradients will have smaller learning rates and vice versa. The advantage of Adagrad, is that it helps deal with sparse data where the gradient vectors have zero or near-zero components by ensuring that the steps taken in each dimension are appropriately scaled based on the historical gradients.

The learning rate of the algorithm is adapted by this accumulated sum, of squared past gradients:

$$G_t = \sum_{t=1}^{N} \nabla f(x_t) \nabla f(x_t)^T$$

However, if this sum keeps growing, it results in exploding sums. This, in turn, causes a continual decay of learning rates that tend to zero, which lead to slow convergence and ultimately halts the learning process. This can cause the algorithm to stop making progress before reaching the optimal solution, especially in non-convex optimization problems or when training over long periods.
The iteration formula of Adagrad is [16]:

$$x_{t+1} = x_t - \frac{n}{\sqrt{G_t + \varepsilon}} \nabla f(x_t)$$

where: " G_t is a diagonal matrix, where the diagonal elements are the sum of the squares of the past gradients and ε is a small constant with a value usually set to 10^{-8} , which prevents division by zero."

Algorithm: Adagrad

Inputs: Cost Function J(x), learning rate n

1. Initialize: Random x
2. Initialize: matrix G=0
3. **For** i = 1 to N
4. Shuffle the data points
5. Compute the gradient $\mathbf{g} = \nabla \mathbf{x} \boldsymbol{J}(\boldsymbol{x}, \boldsymbol{x_i}, \boldsymbol{y_i})$
6. Update the matrix $\mathbf{G=G+g*g^T}$
7. Update parameters $\boldsymbol{x} = \boldsymbol{x} - \frac{\boldsymbol{n}}{\sqrt{\boldsymbol{diag(G)}+\boldsymbol{\varepsilon}}}\mathbf{g}$
8. **End For**
9. **Return** model variable **x**

In terms of convergence, Adagrad is almost certain to asymptotically convergent[10] in the non-convex problems, meaning in mathematical terms that: $\lim_{t\to\infty} \nabla_f(x_t) = 0$

[10] Asymptotic convergence means that as the number of iterations t approaches infinity, the parameters x_t approach a critical point of the objective function.

This implies that the gradients become small, indicating that the parameters are approaching a point where the objective function is flat and no further improvement can be made. In addition, Adagrad provides sublinear convergence $O\left(1/\sqrt{T}\right)$ in convex optimization, while for non-convex objectives a rate of $O\left(log(T)/\sqrt{T}\right)$ [22].

Adadelta

AdaDelta is an extension of Adagrad designed to address some of its limitations, in particular the problem of learning rate decay by using a moving average of squared gradients. The core idea is that it keeps track the historical gradients rather than accumulating them over time as AdaGrad does. In this way, learning progresses even after many iterations [23]. Adadelta is inspired by the exponential moving average that is used in momentum.

The update rule of Adadelta [16]:

$$E[g^2]_t = \rho E[g^2]_{t-1} + (1-\rho)g_t^2$$

Where $E[g^2]_t$is the exponentially decaying average of past squared gradients, ρ is the decay rate, which usually is set to 0.9 and g_t is the gradient at time step t.

So replacing the diagonal matrix G_t with the moving average of past squared gradients $E[g^2]_t$, the parameter update is equal to:

$$\Delta x_t = -\frac{n}{\sqrt{E[g^2]_t + \varepsilon}}\, g_t$$

Then, the update is:

$$E[\Delta x^2]_t = \rho E[\Delta x^2]_{t-1} + (1-\rho)\Delta x_t^2$$

$$RMS[\cdot]_t \cdot= \sqrt{E[\,.^2\,]_t + \varepsilon}$$

Knowing that Newton's Rearrangement Method is:

$$\Delta_x = \frac{\frac{\partial f}{\partial x}}{\frac{\vartheta^2 f}{\vartheta x^2}} => \frac{1}{\frac{\vartheta^2 f}{\vartheta x^2}} = \frac{\Delta x}{\frac{\vartheta f}{\theta x}}$$

Given that the $RMS[\Delta_x]_t$ is unknown, we assume that the curvature is locally smooth and we approximate it by taking the square root of the accumulated squared parameter updates from previous steps. So, the update parameter of Adadelta is:

$$\Delta_{x_t} = -\frac{RMS[\Delta_x]_{t-1}}{RMS[g]_t} g_t$$

$$x_{t+1} = x_t + \Delta_{x_t}$$

As it seems, Adadelta eliminates the need for an initial learning rate n by normalizing updates using the ratio of the RMS (Root Mean Square) of recent gradients to the RMS of recent parameter updates. This makes Adadelta more robust in models where manually tuning learning rates can be challenging.

The lack of strong theoretical convergence guarantees in Adadelta can be seen as drawback compared to some other optimization methods. But, its empirical performance often demonstrates faster convergence compared to other algorithms, particularly in non-convex optimization scenarios.

Algorithm: Adadelta

Inputs: Cost Function J(x), learning rate n

1. Initialize: Random x
2. Initialize: matrix G=0
3. Initialize matrix X=0
4. **For** i = 1 to N
5. Shuffle the data points
6. Compute the gradient $\mathbf{g} = \nabla \mathbf{x}\, \boldsymbol{J}(\, \boldsymbol{x}, \boldsymbol{x_i}, \boldsymbol{y_i})$
7. Update the matrix $\mathbf{G}=\boldsymbol{\rho}*\mathbf{G}+(1-\boldsymbol{\rho})\mathbf{g}*\mathbf{g}^{\boldsymbol{T}}$
8. Update variable $\boldsymbol{x} = \boldsymbol{x} - \frac{\boldsymbol{n}}{\sqrt{\boldsymbol{diag}(\boldsymbol{G})+\boldsymbol{\varepsilon}}}\mathbf{g}$
9. $\Delta\mathbf{x}=-\frac{\sqrt{\boldsymbol{diag}(\boldsymbol{X})+\boldsymbol{\varepsilon}}}{\sqrt{\boldsymbol{diag}(\boldsymbol{G})+\boldsymbol{\varepsilon}}}\boldsymbol{g}$
10. Update $\mathbf{X} = \boldsymbol{\rho}*\mathbf{X}+(1-\boldsymbol{\rho})*\mathbf{X}*\mathbf{X}^{\boldsymbol{T}}$
11. Update parameters $\mathbf{x} = \mathbf{x}+\Delta\boldsymbol{\chi}$
12. **End For**
13. **Return** model variable **x**

RMSprop

RMSprop (Root Mean Squared Propagation) is another extension of Adagrad, which proposed by Geoffrey Hinton. Actually, RMSprop is similar to the initial update vector of Adadelta in the previous section. The algorithm, like Adadelta, modifies the accumulation of past gradients by giving more weight to recent gradients and less weight to older gradients, thus preventing the learning rate from decreasing too fast. The update rule is [16]:

$$E[g^2]_t = \rho E[g^2]_{t-1} + (1-\rho) g_t^2$$

$$x_{t+1} = x_t - \frac{n}{\sqrt{E[g^2]_t + \varepsilon}}\, g_t$$

where ρ is recommended to "be set to 0.9 and a good default value for the learning rate η is 0.001" [16]. In contrast to Adadelta, RMSprop in his update rule requires the specification of an initial learning rate n, which needs to be tuned.

Algorithm: RMSprop

Inputs: Cost Function J(x), learning rate n

1. Initialize: Random x
2. Initialize: matrix G=0
3. **For** i = 1 to N
4. Shuffle the data points
5. Compute the gradient $\mathbf{g} = \nabla \mathbf{x}\, \boldsymbol{J}(\, \boldsymbol{x}, \boldsymbol{x_i}, \boldsymbol{y_i})$
6. Update the matrix $\mathbf{G}=\boldsymbol{\rho}*\mathbf{G}+(1-\boldsymbol{\rho})*\mathbf{g}\mathbf{g}^{\boldsymbol{T}}$
7. Update parameters $\boldsymbol{x} = \boldsymbol{x} - \frac{\boldsymbol{n}}{\sqrt{\boldsymbol{diag}(\boldsymbol{G})+\boldsymbol{\varepsilon}}}\mathbf{g}$
8. **End For**
9. **Return** model variable **x**

In terms of optimization, RMSprop achieves sublinear convergence rates in the context of non-convex stochastic optimization, for a given batch size. It also converge faster than other optimization algorithms, especially in deep neural networks with many layers. This happens because it can efficiently adjust the learning rate for each step, which is useful for complex with a wide number of feautures optimization models.

Adam

Adam (**Ada**ptive **m**oment Estimation) is another adaptive learning rate optimization algorithm. It is appropriate for big data problems, and it works as a combination of RMSprop and momentum. Specifically, Adam utilises the past squared first-order gradients, like RMSprop and the past first-order gradients similar to momentum [16]:

$$m_t = \beta_1 m_{t-1} + (1 - \beta_1) g_t$$

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

"Where: m_t and v_t are estimates of the first moment (mean) and the second moment (uncentered variance) of the gradients. The parameter β_1 is the decay rate of the first moment and β_2 is the decay rate of second moment which are typically set to around 0.999" [16]. These two estimates, make the algorithm to adjust the learning rate for each parameter.

Especially, the exponentially weighted moving average of the gradients (mean) provides information about the average direction in which the parameters should be updated and helps to smooth them, reducing the noise. While, the exponentially weighted moving average of the squared gradients (uncentered variance) measures the magnitude of the gradients, which is

essential for ensuring stability and preventing the step size from becoming too small or too large, during the optimization process.

The authors of Adam [24], observed that when these two vectors are initialized as zeros, they are biased towards zero. This problem is overcomed by computing the bias-corrected first and second moment estimates:

$$\widehat{m}_t = \frac{m_t}{1-{\beta_1}^t}\,,\ \widehat{v}_t = \frac{v_t}{1-{\beta_2}^t}$$

Then using these estimates yields the Adam update rule:

$$x_{t+1} = x_t - \frac{n}{\sqrt{\widehat{v}_t + \varepsilon}}\,\widehat{m}_t$$

The algorithm is stable because of the per-parameter adaptation of the learning rate and works well in non-convex problems with sparse or noisy gradients and high dimensional space. However, a study [25] shows that the exponential moving average in the Adam and RMSprop algorithms can cause non-convergence by providing an example of simple convex optimization problem. According to another study [26], Adam achieves the convergence rate of $O(1 / \sqrt{T})$ in the non-convex stochastic setting "by requiring the batch size to be the same order as the number of maximum iterations".

Algorithm: Adam

Inputs: Cost Function J(x), learning rate n,decay rates β1,β2

1. Initialize: vector m=0
2. Initialize: vector v=0
3. Initialize: matrix G=0
4. **For** i = 1 to N
5. Shuffle the data points
6. Compute the gradient $\mathbf{g} = \nabla \mathbf{x}\, \boldsymbol{J}(\, \boldsymbol{x}, \boldsymbol{x_i}, \boldsymbol{y_i})$
7. Update vector **m = β1*m+(1-β1)*g**
8. Update vector **v= β2*v+(1-β2)*gg**
9. Compute bias-corrected $\widehat{\boldsymbol{m}} = \boldsymbol{m}\ /\ (\mathbf{1} - {\boldsymbol{\beta_1}}^{\boldsymbol{T}})$
10. Compute bias-corrected $\widehat{\boldsymbol{v}} = \boldsymbol{v}\ /\ (\mathbf{1} - {\boldsymbol{\beta_2}}^{\boldsymbol{T}})$
11. Update parameters $\boldsymbol{x} = \boldsymbol{x} - \frac{\boldsymbol{n}}{\sqrt{\widehat{\boldsymbol{v}}+\boldsymbol{\varepsilon}}}\widehat{m}$
12. **End For**
13. **Return** model variable **x**

In this research [27], Adam is identified as a strong and reliable optimizer that consistently performs well across a variety of deep learning tasks. The study demonstrates that despite the emergence of newer optimization methods, Adam often remains competitive and is not significantly outperformed by these alternatives. Moreover it emphasizes the benefits of tuning[11] the algorithm and combining it with other optimizers like RMSProp and NAG for improved perfomance. If it is combined with NAG, then the following algorithm (Nadam) arises.

[11] Adjusting the hyperparameters of the optimizer (such as learning rate, momentum).

Nadam

Nadam (Nesterov-accelerated Adaptive Moment Estimation) is an adaptive algorithm that extends the Adam by incorporating Nesterov momentum. This implies that the algorithm benefits from the accelerated Nesterov method, which accelerates the convergence and smooths out the learning process, enhancing in this way the stability.

In his paper [28], Timothy Dozat demonstrates that Nadam outperforms other algorithms, including Adam, in terms of training and validation loss for a convolutional autoencoder.

The update rule of Nadam arises from few modifications of NAG and Adam [16] :

$$x_{t+1} = x_t - \frac{n}{\sqrt{\hat{v}_t t} + \varepsilon}\left(\beta_1 \hat{m}_t + \frac{(1-\beta_1) g_t}{1-{\beta_1}^t}\right)$$

Algorithm: Nadam

Inputs: Cost Function J(x), learning rate n, decay rates β1,β2

1. Initialize: m=0
2. Initialize v=0
3. **For** i = 1 to N
4. Shuffle the data points
5. Compute the gradient, gradient $= \nabla \mathbf{x}\, \boldsymbol{J}(\boldsymbol{x}, \boldsymbol{x_i}, \boldsymbol{y_i})$
6. Update vector **m = β1*m+(1-β1)*g**
7. Update vector **v= β2*v+(1-β2)*gg**
8. Compute bias-corrected $\hat{\boldsymbol{m}} = \boldsymbol{m} \,/\, (\mathbf{1} - {\boldsymbol{\beta_1}}^{\boldsymbol{T}})$
9. Compute bias-corrected $\hat{\boldsymbol{v}} = \boldsymbol{v} \,/\, (\mathbf{1} - {\boldsymbol{\beta_2}}^{\boldsymbol{T}})$
10. Update the parameters $\boldsymbol{x} = \boldsymbol{x} - \frac{\boldsymbol{n}}{\sqrt{\hat{\boldsymbol{v}}+\boldsymbol{\varepsilon}}}\,(\boldsymbol{\beta 1} * \hat{\boldsymbol{m}} + \frac{(\mathbf{1}-\boldsymbol{\beta_1})\boldsymbol{g}}{\mathbf{1}-{\boldsymbol{\beta_1}}^{\boldsymbol{t}}}$
11. **End For**
12. **Return** model variable **x**

Chapter 4

Experiments

In this chapter, we test the efficiency of the first-order algorithms on a very common benchmark dataset, Cifar-10 [29][30], using two different model architectures. The dataset contains 60.000 color images with 6.000 images per class. Each image is 32x32 pixels and the dataset is characterized by low image resolution and diversity within each class.

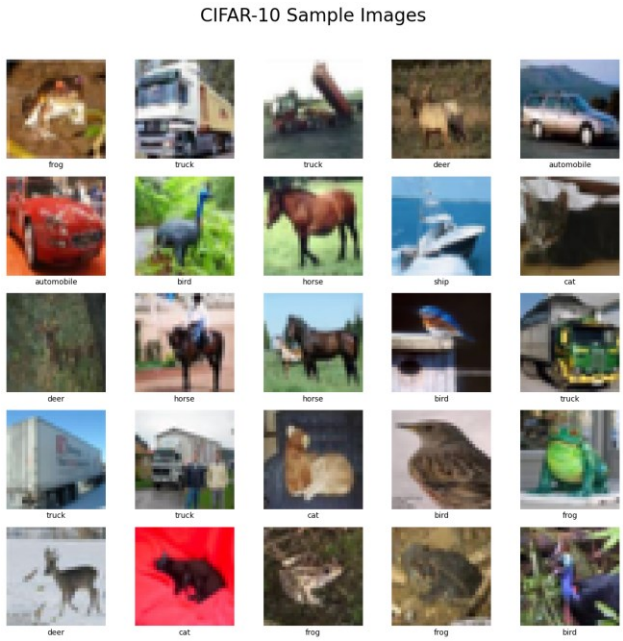

Fig. 7. Samples from Cifar-10 dataset.

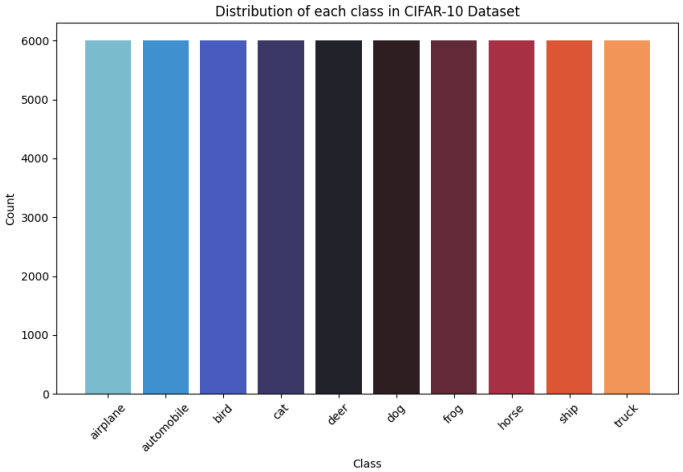

Fig. 8. The distribution of classes in Cifar-10 dataset.

The models below, are tested in order to evaluate each algorithm's perfomance. Every algorithm is examined in terms of accuracy, loss behavior, training time efficiency and learning rate perfomance. These information, allow to understand the differences between the algorithms. Specifically, by plotting training and validation loss over epochs it is described how quickly and effectively each optimizer converge. Each loss curve can describe also if there

is stability in the model. Moreover, some optimizers might have more fluctuations, which could impact the training stability. While, other optimizers in terms of learning rate are more sensitive. The figures that are presented in this chapter give an idea of how each optimizer behaves in the specific dataset.

4.1 Perfomance in Cifar-10 with a baseline Model Architecture

Initially, a basic CNN model is implemented in order to understand the perfomance between diferent optimizers.
The structure of the CNN model as shown in the figure below, consists of four Conv2D layers, with 32,64,64,128 filters respectively, each of size 3x3. Every convolutional layer is followed by a MaxPooling2D layer, except the last one. It has a GlobalAveragePooling2D layer which reduces the output of the last convolutional layer to a single vector. Finally, a fully connected (Dense) layer with 10 units is added to the model.

Model: "sequential_4"

Layer (type)	Output Shape	Param #
conv2d_16 (Conv2D)	(None, 32, 32, 32)	896
max_pooling2d_12 (MaxPooling2D)	(None, 16, 16, 32)	0
conv2d_17 (Conv2D)	(None, 16, 16, 64)	18,496
max_pooling2d_13 (MaxPooling2D)	(None, 8, 8, 64)	0
conv2d_18 (Conv2D)	(None, 8, 8, 64)	36,928
conv2d_19 (Conv2D)	(None, 8, 8, 128)	73,856
global_average_pooling2d (GlobalAveragePooling2D)	(None, 128)	0
dense_4 (Dense)	(None, 10)	1,290

Total params: 131,466 (513.54 KB)
Trainable params: 131,466 (513.54 KB)
Non-trainable params: 0 (0.00 B)

Fig. 9. The Simple CNN model architecture.

In this model, the training uses a batch size of 64 and will run for 30 epochs. To enhance the training process a couple of callback funtions are added.

Firstly, the early stopping technique has been implemented, which has resulted in some algorithms (in figure 10) exhibiting a reduced number of epochs or a reduced time. This ensures that the model will not waste time and computational resources. The parameter patience of the callback is set to 6.
Additionally, ReduceLROnPlateau, a type of learning rate scheduler[12] utilized to adjust the learning rate for better convergence and efficient training. The function's factor parameter is set to 0.5, the patience parameter is set to 4 and the minimum learning rate is 0.00001.
Each algorithm is tested at two different learning rates, and each plot shows the accuracy and the cross entropy loss in the training and validation sets. Plotting these metrics over epochs can also help to explain how well the model learns and generalizes.

[12] In Appendix B in figure B1 is also presented the CNN model curves without the learning rate scheduler.

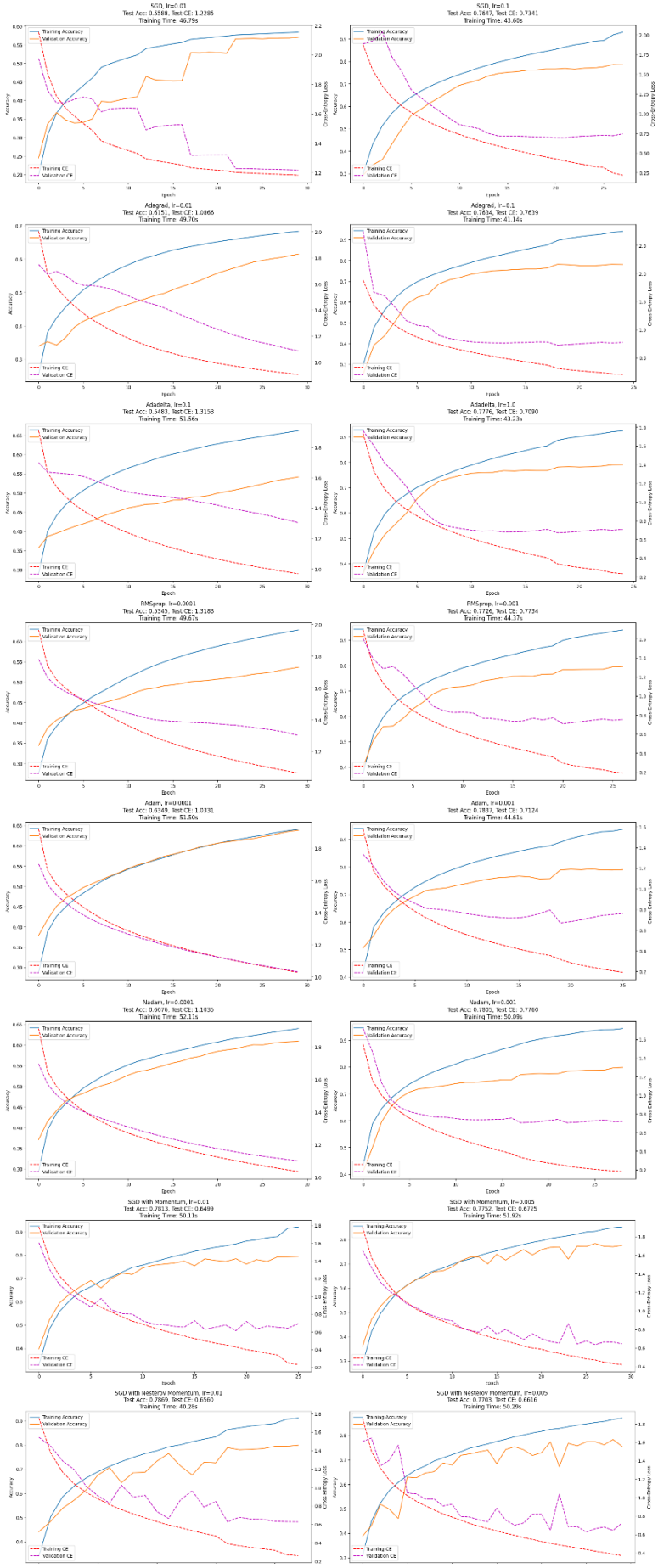

Fig. 10. The training accuracy and the cross entropy loss for each case.

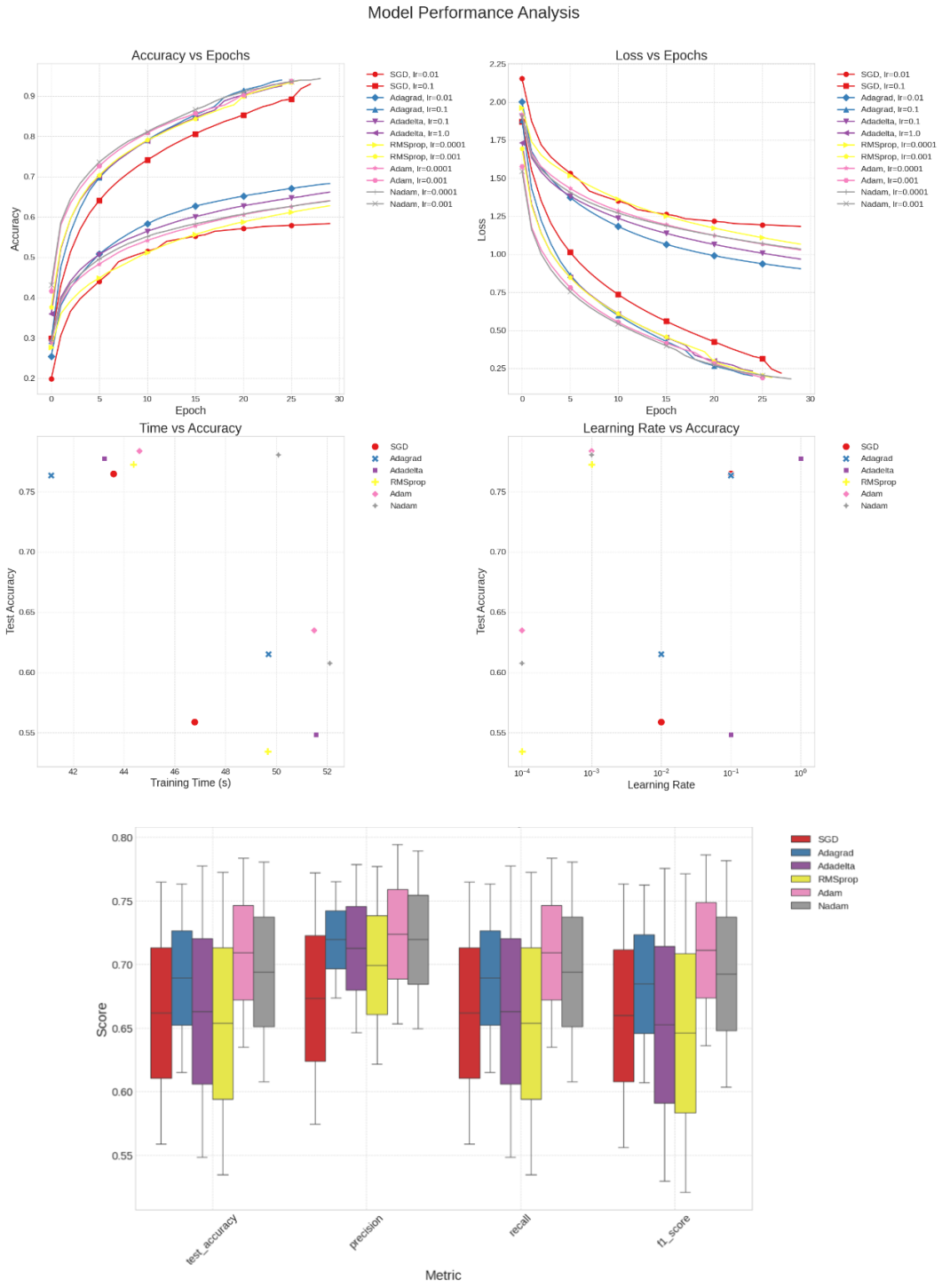

Fig. 10. The visualization analysis of the first CNN model perfomance in Cifar-10.

These comparisons highlight the importance of hyperparameter tuning, such as learning rate and number of epochs. Especially, the learning rate is very crucial for the final result. In this model it is noticed that some algorithms overfit, but others such as Adam and Nadam have better stability. The SGD with momentum and SGD with Nesterov algorithms achieve good test accuracy and loss in both learning rate cases in contrast to the other algorithms. Also, the remaining algorithms are more sensitive to the change of learning rate and the majority of them have noise. However, the model performs relatively well depending on its size and the dataset, but for better performance a different architecture with more layers can be used.

4.2 Perfomance in Cifar-10 with a more complex Model Architecture

This architecture has some extra layers and it is presented how these changes in combination with some other add-ons affect the performance of each optimizer. Specifically, the model, consists of eight Conv2D layers, which have by two, 32,64,128,256 filters respectively. After each of these, a batch normalization layer is introduced to make the optimizer perform more consistently and to stabilize the learning process. Moreover, between them, four Maxpooling2D layers are imported and afterwards four dropouts. Dropouts improve generalization and reduce the risk of over-optimizing, preventing the algorithms to over-optimize specific paths in loss landscape that lead to poor generalization. Finally, a flatten layer and a fully connected layer complete the model architecture.

Layer (type)	Output Shape	Param #
conv2d (Conv2D)	(None, 32, 32, 32)	896
batch_normalization (BatchNormalization)	(None, 32, 32, 32)	128
conv2d_1 (Conv2D)	(None, 32, 32, 32)	9,248
batch_normalization_1 (BatchNormalization)	(None, 32, 32, 32)	128
max_pooling2d (MaxPooling2D)	(None, 16, 16, 32)	0
dropout (Dropout)	(None, 16, 16, 32)	0
conv2d_2 (Conv2D)	(None, 16, 16, 64)	18,496
batch_normalization_2 (BatchNormalization)	(None, 16, 16, 64)	256
conv2d_3 (Conv2D)	(None, 16, 16, 64)	36,928
batch_normalization_3 (BatchNormalization)	(None, 16, 16, 64)	256
max_pooling2d_1 (MaxPooling2D)	(None, 8, 8, 64)	0
dropout_1 (Dropout)	(None, 8, 8, 64)	0
conv2d_4 (Conv2D)	(None, 8, 8, 128)	73,856
batch_normalization_4 (BatchNormalization)	(None, 8, 8, 128)	512
conv2d_5 (Conv2D)	(None, 8, 8, 128)	147,584
batch_normalization_5 (BatchNormalization)	(None, 8, 8, 128)	512
max_pooling2d_2 (MaxPooling2D)	(None, 4, 4, 128)	0
dropout_2 (Dropout)	(None, 4, 4, 128)	0
conv2d_6 (Conv2D)	(None, 4, 4, 256)	295,168
batch_normalization_6 (BatchNormalization)	(None, 4, 4, 256)	1,024
conv2d_7 (Conv2D)	(None, 4, 4, 256)	590,080
batch_normalization_7 (BatchNormalization)	(None, 4, 4, 256)	1,024
max_pooling2d_3 (MaxPooling2D)	(None, 2, 2, 256)	0
dropout_3 (Dropout)	(None, 2, 2, 256)	0
flatten (Flatten)	(None, 1024)	0
dense (Dense)	(None, 10)	10,250

Total params: 1,186,346 (4.53 MB)
Trainable params: 1,184,426 (4.52 MB)
Non-trainable params: 1,920 (7.50 KB)

Fig. 11. The second CNN architecture.

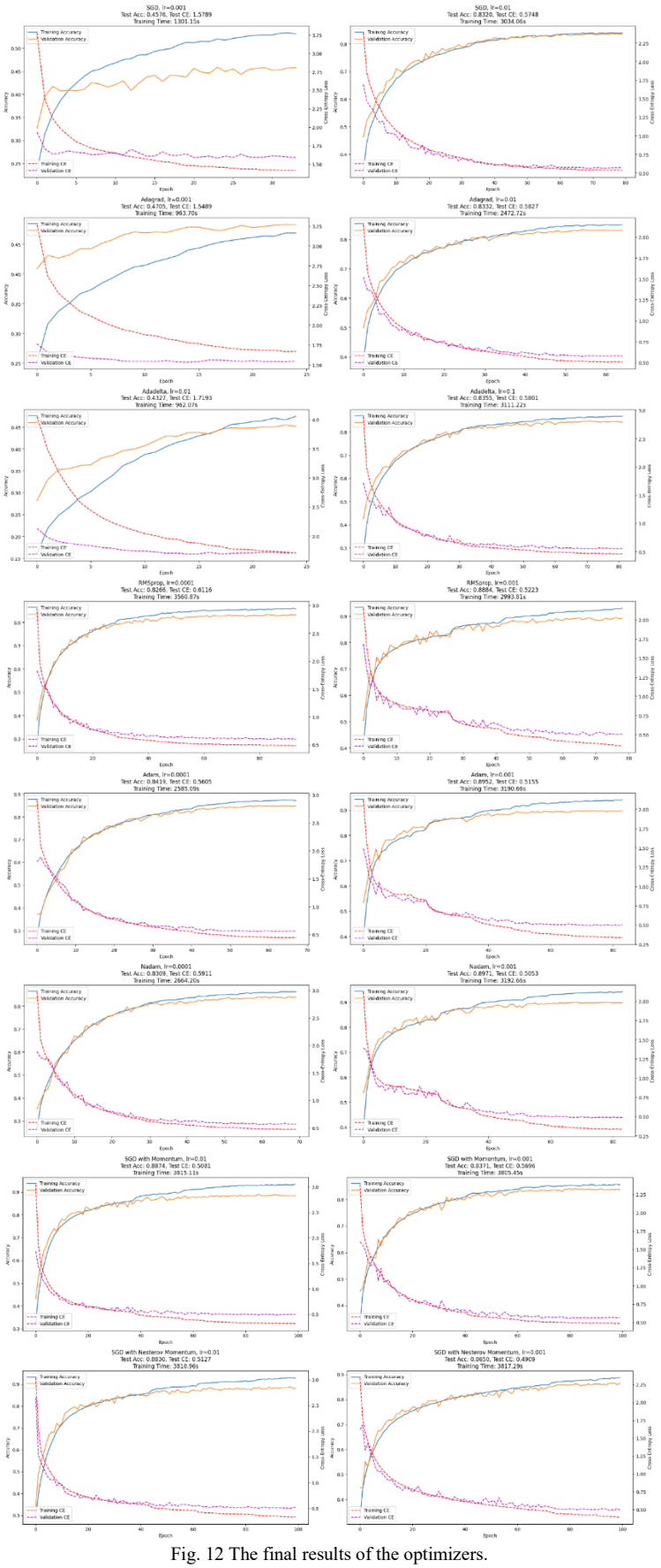

Fig. 12 The final results of the optimizers.

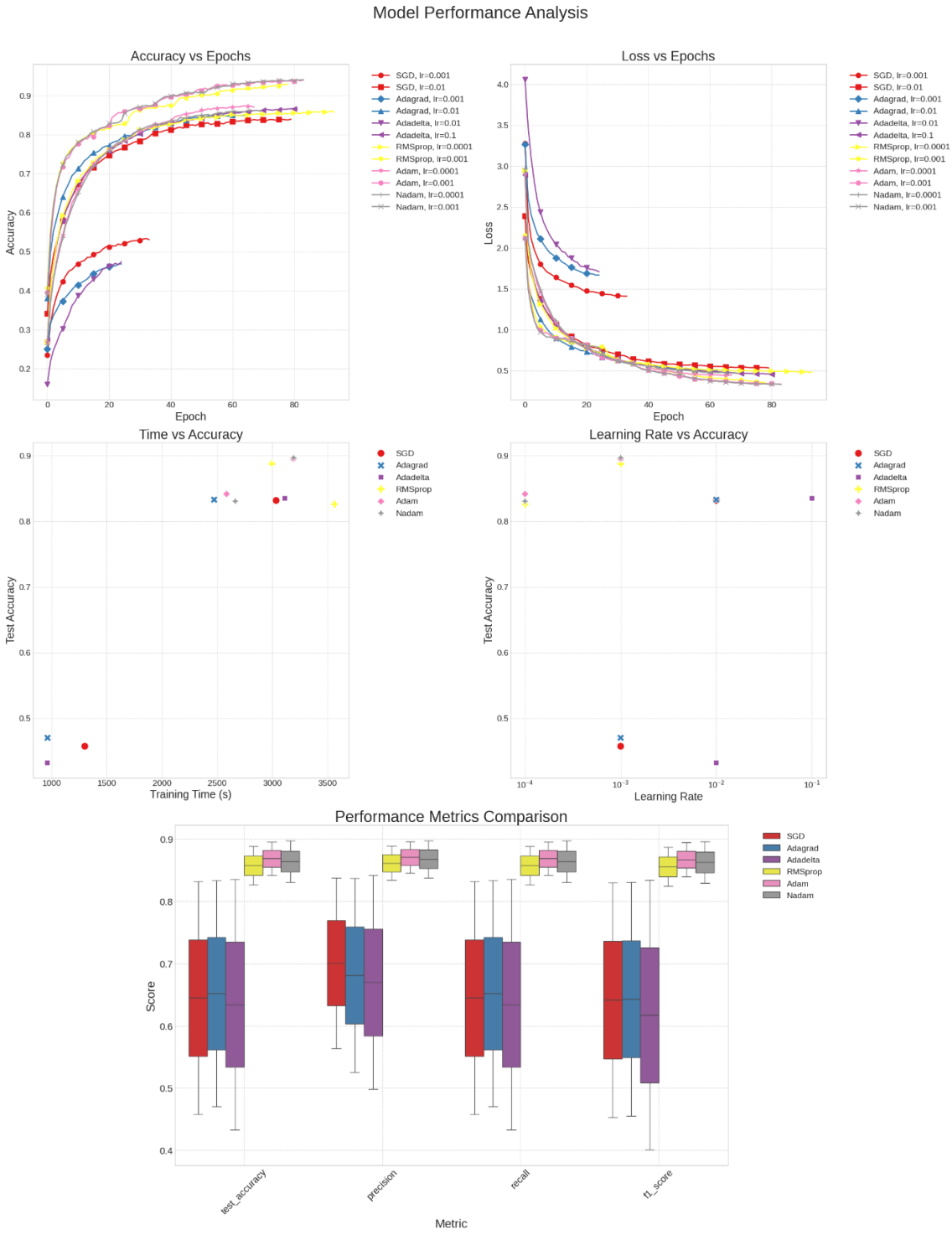

Fig. 13. The visualization analysis of the second CNN model perfomance in Cifar-10 dataset.

This model uses batch size of size 64, giving better stability, and the number of epochs is 100, including the callback earlystopping the learning rate scheduler. In the preprocessing step, standardization and data augmentation are employed, smoothing the loss landscape and improving generalization. In these graphs the optimizers have in general better perfomance than previously. According to the figure 12, the training and validation curves follow similar trajectories and converge, without the training metrics significantly outperforming the

validation metrics. This suggests that in most cases[13] the optimizers are stable and don't overfit or underfit. In terms of accuracy and cross entropy loss in this dataset, Nadam, Adam and RMSprop achieve good results. While, algorithms such as SGD, Adagrad and Adadelta with lower learning rates converge more slowly.

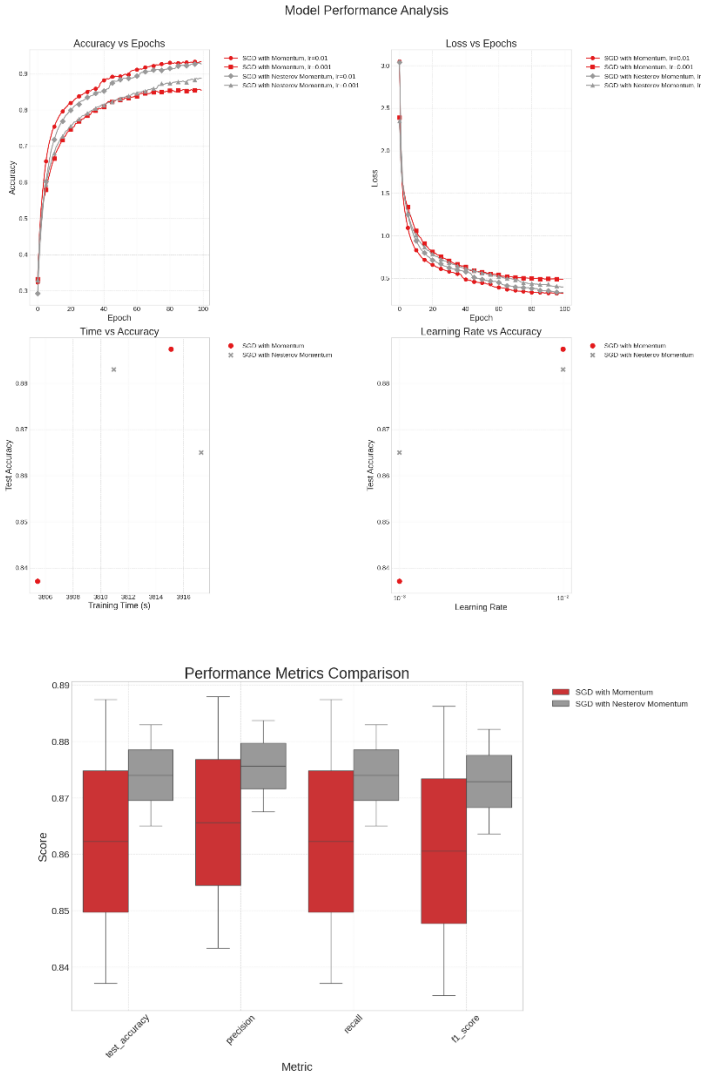

Fig. 14. The remaining algorithms analysis of the second CNN model

[13] In SGD and Adagrad with learning rate 0.001 and Adadelta with learning rate 0.01 there are exist signs of overfitting.

Chapter 5

Conclusions

According to the results of Chapter 4, there are many factors that affect the final performance of a machine learning model. The optimizers have different complexity and they can perform in a different way each time. However, there are many parameters and methods that are used to achieve faster training or better final performance. There are also cases, where the most used algorithms like Adam or SGD can't work efficiently on the dataset and there are other options that can be more effective. This means that no single optimizer consistently outperforms others across all problems and each optimizer according to the algorithmic design and the hyperparameter settings, may be better suited for certain types of problems. So depending on the nature of the data and the different dimensions of it in space landscape, there may be better options. The interesting point is that these algorithms can make combinations between them and give different results each time. Finally, in terms of optimization, it’s always important to evaluate a range of different techniques to get a better outcome.

Bibliography

[1] J. J. Mark, “Pythagoras,” *World History Encyclopedia*. World History Encyclopedia, May 23,2019.[Online].Available:https://www.worldhistory.org/Pythagoras/#citation_info

[2] S. Kiranyaz, T. Ince, and M. Gabbouj, “Optimization Techniques: An Overview,” in *Multidimensional Particle Swarm Optimization for Machine Learning and Pattern Recognition*, vol. 15, in Adaptation, Learning, and Optimization, vol.15.,Berlin, Heidelberg:Springer Berlin Heidelberg,2014, pp.13–44.doi:10.1007/978-3-642-37846-1_2.

[3] D. Hernandez and T. B. Brown, “Measuring the Algorithmic Efficiency of Neural Networks,” May 08, 2020, *arXiv*: arXiv:2005.04305. Accessed: Apr. 22, 2024. [Online]. Available: http://arxiv.org/abs/2005.04305

[4] D. P. Bertsekas, *Nonlinear programming*, 3rd ed.Belmont, Mass: Athena scientific, 2016.

[5] Z. Allen-Zhu, “Natasha 2: Faster Non-Convex Optimization Than SGD,” Jun. 11, 2018, *arXiv*: arXiv:1708.08694. [Online]. Available: http://arxiv.org/abs/1708.08694

[6] R.-Y. Sun, “Optimization for Deep Learning: An Overview,” *J. Oper. Res. Soc. China*, vol. 8, no. 2, pp. 249–294, Jun. 2020, doi: 10.1007/s40305-020-00309-6.

[7] M. A. Cauchy, “Méthode générale pour la résolution des systèmes d’équations simultanées,” *Comptes Rendus de l'Académie des Sciences*, vol. 25, pp. 536–538, 1847.

[8] S. Bubeck, “Convex Optimization: Algorithms and Complexity,” *Found. Trends® Mach. Learn.*, vol. 8, no. 3–4, pp. 231–357, 2015, doi: 10.1561/2200000050.

[9] R. Grosse, "Lecture 7: Advanced Optimization," course slides, CSC321: Deep Learning, University of Toronto, [Online]. Available: https://www.cs.toronto.edu/~rgrosse/courses/csc321_2017/slides/lec7.pdf.

[10] D. Masters and C. Luschi, “Revisiting Small Batch Training for Deep Neural Networks,” Apr. 20, 2018, *arXiv*: arXiv:1804.07612. [Online]. Available: http://arxiv.org/abs/1804.07612

[11] S. Y. Khamaiseh, D. Bagagem, A. Al-Alaj, M. Mancino, and H. W. Alomari, “Adversarial Deep Learning: A Survey on Adversarial Attacks and Defense Mechanisms on Image Classification,” *IEEE Access*, vol. 10, pp. 102266–102291, 2022, doi: 10.1109/ACCESS.2022.3208131.

[12] R Tibshirani, “R. Tibshirani ‘Stochastic Gradient Descent,’ scribe notes, Convex Optimization course, Carnegie Mellon University,.” [Online]. Available: https://www.stat.cmu.edu/~ryantibs/convexopt/scribes/stochastic-gd-scribed.pdf

[13] B. T. Polyak, “Some methods of speeding up the convergence of iteration methods,” *USSR Comput. Math. Math. Phys.*, vol. 4, no. 5, pp. 1–17, Jan. 1964, doi: 10.1016/0041-5553(64)90137-5.

[14] I. Goodfellow, Y. Bengio, and A. Courville, *Deep learning*. in Adaptive computation and machine learning. Cambridge, Massachusetts: The MIT Press, 2016.

[15] M. J. Kochenderfer and T. A. Wheeler, *Algorithms for optimization*. Cambridge, Massachusetts: The MIT Press, 2019.

[16] S. Ruder, “An overview of gradient descent optimization algorithms,” Jun. 15, 2017, *arXiv*: arXiv:1609.04747. [Online]. Available: http://arxiv.org/abs/1609.04747

[17] IFT 6085, “Accelerated Methods - Polyak’s Momentum (Heavy Ball Method) lecture notes, [Online].” [Online]. Available: https://mitliagkas.github.io/ift6085-2019/ift-6085-lecture-5-notes.pdf

[18] E. Ghadimi, H. R. Feyzmahdavian, and M. Johansson, “Global convergence of the Heavy-ball method for convex optimization,” Dec. 23, 2014, *arXiv*: arXiv:1412.7457. [Online]. Available: http://arxiv.org/abs/1412.7457

[19] I. Waldspurger, "Advanced Gradient Descent," course notes, Université Paris Dauphine,[Online].Available:https://www.ceremade.dauphine.fr/~waldspurger/tds/22_23_s1/advanced_gradient_descent.pdf.
[20] L. Lessard, B. Recht, and A. Packard, “Analysis and Design of Optimization Algorithms via Integral Quadratic Constraints,” *SIAM J. Optim.*, vol. 26, no. 1, pp. 57–95, Jan. 2016, doi: 10.1137/15M1009597.
[21] R. Kidambi, P. Netrapalli, P. Jain, and S. M. Kakade, “On the insufficiency of existing momentum schemes for Stochastic Optimization,” Jul. 31, 2018, *arXiv*: arXiv:1803.05591. [Online]. Available: http://arxiv.org/abs/1803.05591
[22] A. Défossez, L. Bottou, F. Bach, and N. Usunier, “A Simple Convergence Proof of Adam and Adagrad,” Oct. 17, 2022, *arXiv*: arXiv:2003.02395. [Online]. Available: http://arxiv.org/abs/2003.02395
[23] M. D. Zeiler, “ADADELTA: An Adaptive Learning Rate Method,” Dec. 22, 2012, *arXiv*: arXiv:1212.5701. [Online]. Available: http://arxiv.org/abs/1212.5701
[24] D. P. Kingma and J. Ba, “Adam: A Method for Stochastic Optimization,” Jan. 29, 2017, *arXiv*: arXiv:1412.6980. [Online]. Available: http://arxiv.org/abs/1412.6980
[25] S. J. Reddi, S. Kale, and S. Kumar, “On the Convergence of Adam and Beyond,” Apr. 19, 2019, *arXiv*: arXiv:1904.09237. [Online]. Available: http://arxiv.org/abs/1904.09237
[26] F. Zou, L. Shen, Z. Jie, W. Zhang, and W. Liu, “A Sufficient Condition for Convergences of Adam and RMSProp,” Jun. 24, 2019, *arXiv*: arXiv:1811.09358. [Online]. Available: http://arxiv.org/abs/1811.09358
[27] R. M. Schmidt, F. Schneider, and P. Hennig, “Descending through a Crowded Valley - Benchmarking Deep Learning Optimizers,” Aug. 10, 2021, *arXiv*: arXiv:2007.01547. [Online]. Available: http://arxiv.org/abs/2007.01547
[28] T. Dozat, "Incorporating Nesterov Momentum into Adam," in *Proceedings of the 4th International Conference on Learning Representations (ICLR)*, San Juan, Puerto Rico, May 2016.
[29] A. Krizhevsky, "Learning Multiple Layers of Features from Tiny Images," M.S. thesis, Dept. Comput. Sci., Univ. of Toronto, Toronto, ON, Canada, 2009.
[30] Unknown, “CIFAR-10.” UCI Machine Learning Repository, 2009. doi: 10.24432/C5889J.
[31] B. Gartner and M. Jaggi, “Optimization for Machine Learning Lecture Notes CS-439, Spring 2023”.
[32] S. Boyd and L. Vandenberghe, *Convex Optimization*, 1st ed. Cambridge University Press, 2004. doi: 10.1017/CBO9780511804441.
[33] M. Hardt, "EE227C: Convex Optimization and Approximation," Lecture Notes, Dept. Elect. Eng. and Comput. Sci., Univ. of California, Berkeley, CA, USA, 2015. [Online]. Available: https://www.eecs.berkeley.edu/~hardt/ee227c/
[34] A. A. Amini, "Lecture 7: Convexity II," *ORF 523: Convex and Conic Optimization*, Princeton University, Princeton, NJ, 2017. [Online]. Available: https://www.princeton.edu/~aaa/Public/Teaching/ORF523/ORF523_Lec7.pdf.
[35] A. Anderson, "Subgradient optimization," Optimization in Chemical Engineering, Cornell University, [Online]. Available: https://optimization.cbe.cornell.edu/index.php?title=Subgradient_optimization.

Appendix A

Convex Sets

Definition 1 "A set C is convex if the line segment between any of two points of C lies in C. If for any $x, y \in C$ and any λ with $0 \leq \lambda \leq 1$ we have [31]:

$$\lambda x + (1 - \lambda)y \in C"$$

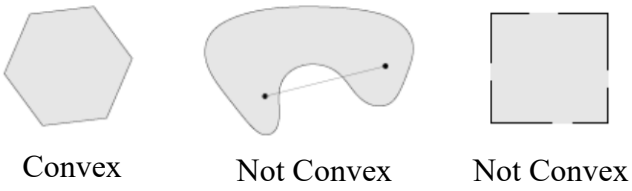

Fig. A1. Convex and nonconvex sets as illustrated in [32, Fig. 2.2]

Convex Functions

Definition 2 "A function $f : dom(f) \rightarrow \mathbb{R}$, $\mathrm{dom(f)} \subseteq \mathbb{R}^{\mathrm{d}}$ is convex if dom(f) is a convex set and for all $x, y \in dom(f)$ and λ with $0 \leq \lambda \leq 1$ we have [31]:

$$f(\lambda x + (1 - \lambda)y) \leq \lambda f(x) + (1 - \lambda)f(y), \qquad \forall x, y \in dom(f)$$

Otherwise, if we have:

$$f(\lambda x + (1 - \lambda)y) \geq \lambda f(x) + (1 - \lambda)f(y), \qquad \forall x, y \in dom(f)$$

the function is concave".

The Convexity of *f* means geometrically, that the line segment connecting two points (x,f(x)) to (y,f(y)) on the graph lies above or on the graph of *f*.

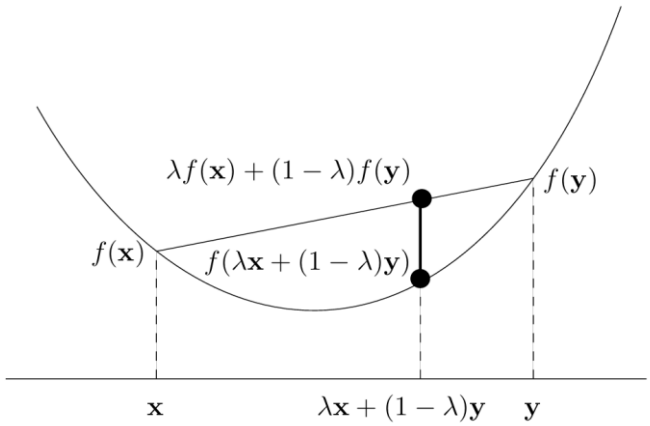

Fig. A2. A Convex function as illustrated in [31, Fig. 1.3].

Proposition
Let f be convex. If x is a local minimum of f then x is a global minimum of f.
Suppose that x ∈ $dom(f)$ is a local minimum of $f : dom(f) \rightarrow \mathbb{R}$ meaning that any point in a neighborhood around x has larger function value ($f(y) \geq f(x)$). Now, for every y ∈ $dom(f)$ we can find a λ ∈ [0,1] such that:

$$f(y) \geq f(\lambda x + (1-\lambda)y) \geq f(x)$$

First and Second order characterization of Convex functions

Definition 3 Suppose $f : dom(f) \rightarrow \mathbb{R}$ is differentiable over an open domain. In particular, the gradient (vector of partial derivatives) $\nabla f(x) = \left(\frac{\vartheta f}{\vartheta x_1}(x), \ldots, \frac{\vartheta f}{\vartheta x_d}(x)\right)$ exists at every point x ∈ $dom(f)$. Then, f is convex if and only if dom(f) is convex and [33]:

$$f(y) \geq f(x) + \nabla f(x)^T(y-x), \qquad x,y \in dom(f)$$

Geometrically, this means that the function lies above its tangent hyperplane as in figure A3.

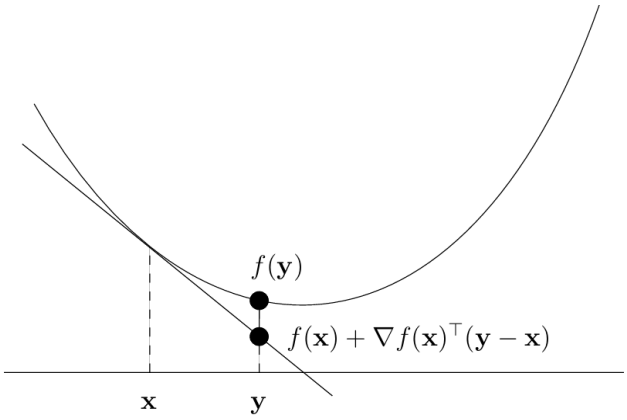

Fig. A3. First-order characterization of convexity as illustrated in [31, Fig. 1.5].

Definition 4 "While, if f *is twice differentiable; in particular the Hessian (matrix of second derivatives)*

$$\nabla^2 f(x) = \begin{bmatrix} \frac{\partial^2 f}{\partial x_1^2} & \frac{\partial^2 f}{\partial x_1 \partial x_2} & \cdots & \frac{\partial^2 f}{\partial x_1 \partial x_d} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial^2 f}{\partial x_d \partial x_1} & \frac{\partial^2 f}{\partial x_d \partial x_2} & \cdots & \frac{\partial^2 f}{\partial x_d^2} \end{bmatrix}$$

exists at every point x ∈ $dom(f)$ and is symmetric, then f is convex if only dom(f) is convex and for all x ∈ $dom(f)$ we have [31]:

$$\nabla^2 f(x) \geq 0"$$

In general, the second derivative measures the speed that the slope of a function can change. Similarly, the Hessian represents how fast the curvature of a function changes. More

specifically, the Schwarz theorem[14] implies that the Hessian matrix is always a symmetric matrix and according to the spectral theorem, any symmetric matrix, including the Hessian, can be decomposed into the form:

$$A = V \cdot \Lambda \cdot V^T$$

where the matrix of $V = [v_1, \dots, v_n]$ is orthogonal ($V^T V = VV^T = I$), and contains the eigenvectors of A, while the diagonal matrix Λ contains the eigenvalues of A.

Lipschitz Continuity

Definition 5 A function $f: dom(f) \to \mathbb{R}$ is L-Lipschitz continuous if there exists a constant L > 0 such that for all x,y ∈ $dom(f)$ [33]:

$$|f(x) - f(y)| \leq L\|x - y\|$$

Where $\|\cdot\|$ denotes the Euclidean norm.
A Lipschitz continuous function is bounded in how fast it can change.

Smoothness

Definition 6 In optimization a function f is β-smooth, if its gradient is Lipschitz continuous with Lipschitz constant β [33]:

$$\|\nabla f(x) - \nabla f(y)\| \leq \beta\|x - y\| \quad \forall x, y \in dom(f)$$

This condition restricts the speed of gradient's change, and β essentially measures the maximum rate of change of the gradient.
Also, the Hessian satisfies:

$$H(x) \leq \beta I$$

meaning that the eigenvalues of H(x) are at most β ($\lambda\max(H(x)) \leq \beta$).

Some useful implications of smoothness are:
1. If f is β-smooth then the function $\frac{\beta}{2}\|x\|^2 - f(x)$ is convex.

2. If f is β-smooth then, there is a quadratic upper bound on the function:

$$f(y) \leq f(x) + \nabla f(x)^T(y - x) + \frac{\beta}{2}\|y - x\|^2$$

[14] The second-order partial derivatives satisfy the identity $\frac{\vartheta}{\vartheta x_i}\left(\frac{\vartheta f}{\vartheta x_j}\right) = \frac{\vartheta}{\vartheta x_j}\left(\frac{\vartheta f}{\vartheta x_i}\right)$.

Strict Convexity

Definition 7 "A Function $f: dom(f) \to \mathbb{R}$ is strictly convex if for all $x \neq y \in dom(f)$ and all $\lambda \in (0,1)$ [31]:

$$f(\lambda x + (1-\lambda)y) < \lambda f(x) + (1-\lambda)f(y)"$$

Or if:

$$f(y) > f(x) + \nabla f(x)^T (y - x) \text{ [34]}$$

As evidenced by the definitions, if f is strictly convex, then f is convex. In general, the strict convexity of the optimization process enhances its efficiency, resulting in improved convergence rates and stability. This stability ensures that small changes in the input lead to predictable changes in the output.

Strong Convexity

Definition 8 "A function is α-strongly convex if $\exists \alpha > 0$ constant such that the modified $g(x) = f(x) - \alpha\|x\|^2$ is convex" [34].
This means that if $g(x)$ is convex, then $f(x)$ must be sufficiently "curved upwards" and more strongly curved than the quadratic term $\alpha\|x\|^2$. Furthermore, there are several significant implications of strong convexity [33]:

"1. If f is strongly convex then an equivalent definition is that it satisfies the following inequality:

$$f(y) \geq f(x) + \nabla f(x)^T (y - x) + \frac{\alpha}{2}\|y - x\|^2, \quad \text{x, y} \in \text{dom(f)}$$

This definition ensures that the function f has a quadratic lower bound, which means it curves upwards more steeply than a standard convex function.

2. If f is twice differentiable, an equivalent characterization is:

$$\nabla^2 f(x) \geq \alpha I"$$

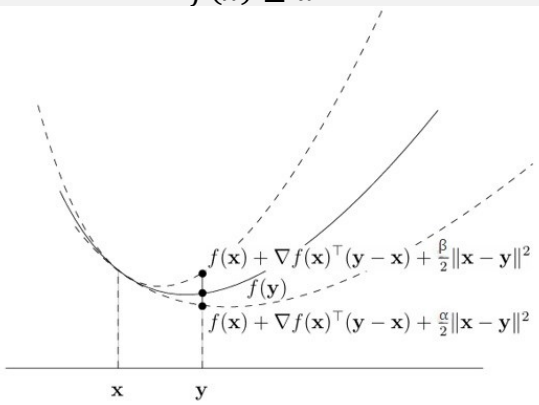

Fig. A4. Smooth and Strongly convex function as illustrated in [31, Fig 2.3].

The steepness of strong convex functions have a significant impact on the convergence behavior of gradient-based algorithms, making the gradient steps more efficient.
The above inequality $\nabla^2 f(x) \geq \alpha I$ demonstrates that the eigenvalues of the Hessian matrix are at least α $(\lambda min(H(x)) \geq \alpha)$, which ensures that the Hessian is well-conditioned. This implies that the function avoid directions with very small curvature which cause issues like slow convergence or overshooting.

In general, the rule is as follows: "Strong convexity ⇒ Strict convexity ⇒ Convexity
But, the converse is not true" [34].

Definition 9 Subgradients are convex functions which are not necessarily differentiable but they preserve convexity, such as the max-operation. A subgradient of a function f at a point x is a vector g such that:

$$f(y) \geq f(x) + g^T(y - x), \quad y \in \text{dom(f)}$$

The set of all subgradients at x is called the subdifferential and is denoted by ∂f(x). Also, If f is differentiable at x, the subdifferential $\partial f(x)$ contains exactly one element, which is the gradient $\nabla f(x)$.

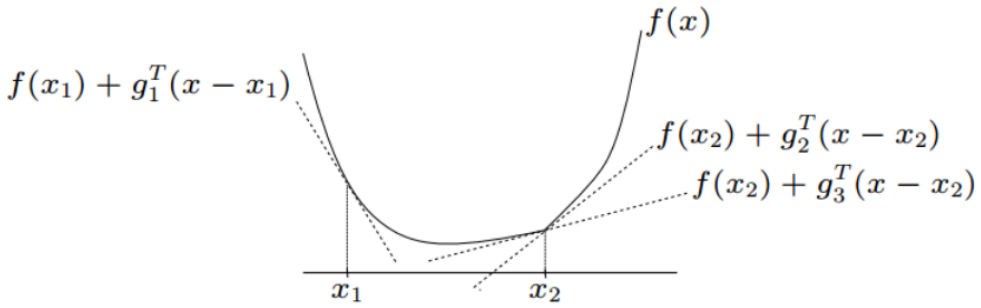

Fig. A5. The subgradients of a non-differentiable convex function at point x_2 as illustrated in [35]

Definition 10 Let A be a real matrix. The condition number is the ratio of its largest and smallest eigenvalues:

$$\text{k(A)} = \frac{\lambda max(A)}{\lambda min(A)}$$

The condition number measures the dynamic range of curvatures of the objective function and it is an indicator of the sensitivity in the input data. Moreover, it is of great significance as it provides insight into the convergence behavior of gradient descent algorithms. It helps to determine the optimal step size for efficient and stable convergence, and it indicates how well-conditioned the optimization problem is, which in turn affects the speed of finding the solution. The ideal condition number is equal to 1 or at least close to 1. In cases where k>>1, the function has steep curvature in some directions and flat curvature in others, which means that the algorithm will have slow convergence and in practice will need more iterations to converge and a small step size to maintain stability.

Backtracing Line Search

Definition 11 Backtracing line search is an inexact method that is used to find an efficient step size that decreases the objective function. The goal is to iteratively reduce the step size n, until an appropriate value is found [32].

The stopping condition for the backtracking line search is:

$$f(x + n\Delta x) \leq f(x) + \alpha n \nabla f(x)^T \Delta x$$

Where α is a factor that adjusts *f*, and takes values between 0 and 0,5. These are the values that can guarantee convergence and sufficient progress. As it seems in the below figure when factor α is reduced the upper dashed line moves downward. Also, for any n, the value of $f(x+\mathrm{n}\Delta x)$ must lie below this new lower line in order for the condition to be satisfied.

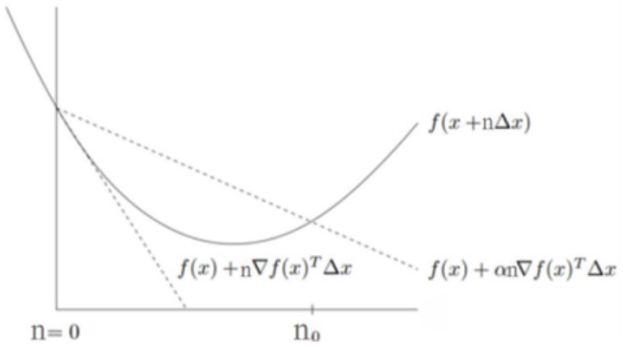

Fig. A6. The backtracking search method as illustrated in [32, Fig 9.1].

"Since Δx is a descent direction, $\nabla \mathrm{f(x)}^{\mathrm{T}}\Delta \mathrm{x} < 0$, so for small enough n, by the Taylor expansion series it is implied [32]:

$$f(x + n\Delta x) \approx f(x) + n\nabla f(x)^T \Delta \mathrm{x}$$

and combining the above formula with the backtracking condition:

$$f(x) + n\nabla f(x)^T \Delta x < f(x) + \alpha n \nabla f(x)^T \Delta x$$"

This method is considered to be less computationally intensive, dealing with large-scale problems and is preferred when the objective function is noisy.

Exact Line Search

Definition 12 Exact line search is used to find the optimal step size n, that has the maximum decrease in the objective function *f*, along a specified search direction Δx. The goal is to select n that minimizes the function $f(\mathrm{x} + \mathrm{n}\Delta \mathrm{x})$ solving the one dimensional problem [32]:

$$n = argmin_{\mathrm{a}\geq 0}\, f(\mathrm{x} + \mathrm{a}\Delta \mathrm{x})$$

This method is useful when the cost of calculating the step size n is low compared to determining the direction, which is the computationally intensive part of the optimization

process. It is preferred when the objective function is smooth for better calculation of the derivatives, which helps to find the exact step size. Exact line search typically needs an algorithm to find the search direction first. Some common methods for determining the search direction include:

- **Gradient Descent:** The search direction d_t at iteration t is the negative gradient of the objective function -$\nabla F(x_t)$ as it is described in Chapter 3.

-While in some other cases:

- **Newton's Method:** In this case the search direction uses the Hessian matrix and it is $d_t = -{H_t}^{-1}\nabla F(x_t)$.

- **Quasi-Newton Methods:** In these algorithms, such as BFGS and L-BFGS the Hessian matrix is approximated and this approximation A_t is used to calculate the search direction: $d_t = -{A_t}^{-1}\nabla F(x_t)$.

- **Conjugate Gradient Method:** In this case **t**he search direction is a combination of the negative gradient and the previous search direction adjusted by a factor β_t as it follows: $d_t = -\nabla F(x_t) + \beta_t d_{t-1}$.

Appendix B

The following tables present a summary of the statistical results obtained from the experiments conducted in sections 4.1 and 4.2, for the two distinct learning rates that employed in each instance.

TABLE B1
TEST ACCURACY (%)

Optimizer	Count	Mean	Min	Max
SGD	2	0.64480	0.5588	0.7647
SGD WITH MOMENTUM	2	0.77825	0.7752	0.7813
SGD WITH NESTEROV	2	0.77860	0.7703	0.7869
ADAGRAD	2	0.68925	0.6151	0.7634
ADADELTA	2	0.66295	0.5483	0.7766
RMSPROP	2	0.65355	0.5345	0.7726
ADAM	2	0.70930	0.6349	0.7837
NADAM	2	0.69405	0.6076	0.7805

TABLE B2
TRAINING TIME (SEC)

Optimizer	Count	Mean	Min	Max
SGD	2	45.19	43.59	46.79
SGD WITH MOMENTUM	2	51.06	50.11	51.92
SGD WITH NESTEROV	2	45.28	40.27	50.28
ADAGRAD	2	45.41	41.13	49.69
ADADELTA	2	47.39	43.22	51.56
RMSPROP	2	47.01	43.59	49.66
ADAM	2	48.05	44.61	51.49
NADAM	2	51.09	50.08	52.11

TABLE B3
TEST ACCURACY (%)

Optimizer	Count	Mean	Min	Max
SGD	2	0.64480	0.4576	0.8320
SGD WITH MOMENTUM	2	0.86225	0.8371	0.8874
SGD WITH NESTEROV	2	0.87400	0.8650	0.8830
ADAGRAD	2	0.65185	0.4705	0.8332
ADADELTA	2	0.63410	0.4327	0.8355
RMSPROP	2	0.85750	0.8266	0.8884
ADAM	2	0.86855	0.8419	0.8952
NADAM	2	0.86400	0.8309	0.8971

TABLE B4
TRAINING TIME (SEC)

Optimizer	Count	Mean	Min	Max
SGD	2	2167	1301	3034
SGD WITH MOMENTUM	2	3810	3805	3815
SGD WITH NESTEROV	2	3814	3810	3817
ADAGRAD	2	1718	963	2472
ADADELTA	2	2036	962	3111
RMSPROP	2	3277	2993	3560
ADAM	2	2887	2585	3190
NADAM	2	2928	2664	3192

In the followimg figure the model perfomance have more noise and inferior metrics compared to this from the section 4.1.

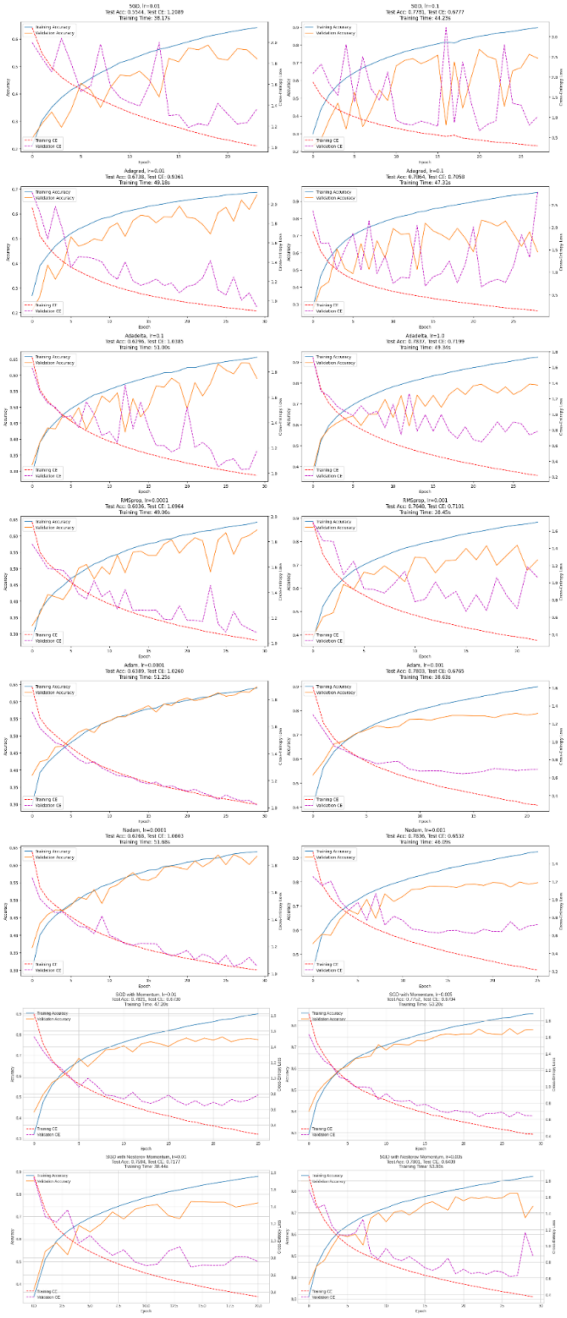

Fig. B1. The Cnn Model from section 4.1 without learning scheduling.