# CI project: Milestone 1
# Comparison of Optimization Algorithms


**Team members**

| Name | ID |
|---|---|
| Waleed Khalid Mohamed Waleed Alzamil | 2002011 |
| Mohammed Khaled Kamal Ellithy | 1900568 |
| Mohammad saeed Ibrahim Dallash | 1900084 |


May 25, 2023

# Abstract

Deep learning has revolutionized the field of artificial intelligence, enabling machines to learn from data and make predictions with unprecedented accuracy. Optimization algorithms play a crucial role in improving the performance of deep learning models by minimizing the loss function and updating the model's parameters.

The field of deep learning has seen significant advancements in recent years, with optimization algorithms playing a crucial role in improving the performance of deep learning models. In this report, we compare the performance of several optimization algorithms commonly used in deep learning, including Stochastic Gradient Descent (SGD)[14, 13, 10], Gradient Descent with Momentum (GDM)[12], Adagrad[3, 11], Adadelta[2], and Adam[1, 8, 7, 6].

# Contents


**1 Introduction** **5**
- 1.1 Intro . . . 5
- 1.2 Problem Statement . . . 5

**2 Optimization Algorithms** **6**
- 2.1 Stochastic Gradient descent (SGD) . . . 6
- 2.2 Gradient descent (GDM) . . . 7
- 2.3 Gradient descent with Adaptive learning rate (Adagrad) . . . 7
- 2.4 Gradient descent with Adaptive learning rate (Adadelta) . . . 7
- 2.5 Gradient descent with Momentum and Adaptive learning rate (Adam) . . . 8
- 2.6 Enhancements . . . 8
  - 2.6.1 Line search algorithm (Adam, SGD and GDM) . . . 8
  - 2.6.2 Bias correction (Adam) . . . 9
  - 2.6.3 Learning Decay (SGD and GDM) . . . 11

**3 Test Functions** **14**
- 3.1 Many Local Minima . . . 14
  - 3.1.1 Definition . . . 14
  - 3.1.2 Input Domain . . . 14
  - 3.1.3 Global Minima . . . 14
- 3.2 Bowl-shaped . . . 15
  - 3.2.1 Definition . . . 15
  - 3.2.2 Input Domain . . . 15
  - 3.2.3 Global Minimum . . . 15
- 3.3 Plate-shaped . . . 16
  - 3.3.1 Definition . . . 16
  - 3.3.2 Input Domain . . . 16
  - 3.3.3 Global Minimum . . . 16
- 3.4 Valley-Shaped . . . 16
  - 3.4.1 Definition . . . 16
  - 3.4.2 Input Domain . . . 16
  - 3.4.3 Global Minimum . . . 16
- 3.5 Steep Ridges/Drops . . . 17
  - 3.5.1 Definition . . . 17
  - 3.5.2 Input Domain . . . 17
  - 3.5.3 Global Minimum . . . 17
- 3.6 Other . . . 18
  - 3.6.1 Definition . . . 18
  - 3.6.2 Input Domain . . . 18

3.6.3 Global Minimum . . . . . 18

**4 Experimental Results** **19**
4.1 Original algorithms . . . . . 19
4.2 Enhancements . . . . . 20

**5 Conclusion** **34**

# List of Algorithms


1 Stochastic Gradient Descent Algorithm . . . . . 7
2 Gradient Descent Algorithm with Momentum . . . . . 8
3 Adagrad Algorithm . . . . . 8
4 Adadelta Algorithm . . . . . 9
5 Adam Algorithm . . . . . 9
6 Line Search Algorithm . . . . . 10
7 Adam Algorithm . . . . . 10
8 Time Decay Algorithm . . . . . 12
9 Step Decay Algorithm . . . . . 12
10 Exponential Decay Algorithm . . . . . 12
11 Performance Decay Algorithm . . . . . 13

# Chapter 1

# Introduction

## 1.1 Intro

Optimization[5, 9, 4] is an important topic in many areas of science and engineering. There are a variety of optimization algorithms available, each with its own strengths and weaknesses. In this project, we compare the performance of five popular optimization algorithms: Stochastic Gradient Descent (SGD), Gradient Descent with Momentum (GDM), Adagrad, Adadelta, and Adam. We evaluate the performance of these algorithms on various test functions with different shapes and numbers of local minima.

Our goal is to provide insights into the strengths and weaknesses of these algorithms and to offer recommendations for future research in this area.

## 1.2 Problem Statement

The optimization problem that we considered was as follows:

$$\text{Minimize } f(x)$$

where $x \in \mathbb{R}^n$ is the decision variable, $f : \mathbb{R}^n \to \mathbb{R}$ is the objective function.

# Chapter 2

# Optimization Algorithms

## Theory

**A gradient-based learning rule** is a method for updating the parameters of a machine learning model by using the gradient of a loss function with respect to the parameters. The gradient is a vector that points in the direction of the steepest increase of the loss function, and by moving the parameters in the opposite direction, we can minimize the loss function and improve the model's performance.

A general form of a gradient-based learning rule is:

$$\theta_{t+1} = \theta_t - \eta \nabla_\theta \mathcal{L}(\theta_t)$$

where $\theta$ is the vector of parameters, $t$ is the iteration index, $\eta$ is the learning rate[1], and $\mathcal{L}$ is the loss function. The loss function measures how well the model fits the data, and it can be different depending on the task, such as mean squared error for regression or cross-entropy[2] for classification.

## 2.1 Stochastic Gradient descent (SGD)

1. **Stochastic Gradient descent (SGD)**: A method that computes the gradient of the cost function with respect to the parameters of the model, and then taking a step in the direction of the negative gradient.

[1] The learning rate controls how big of a step we take along the gradient direction, and it can be constant or adaptive

[2] Cross entropy is a concept from information theory that measures the difference between two probability distributions. It can be used to define a loss function in machine learning and optimization, especially for classification problems. The cross-entropy between a true probability distribution $p$ and an estimated probability distribution $q$ is defined as: $H(p, q) = -\sum_x p(x) log(q(x))$ where $x$ is a possible outcome, $p(x)$ is the true probability of $x$, and $q(x)$ is the estimated probability of $x$. The cross entropy is always non-negative, and it is zero if and only if $p$ and $q$ are the same.

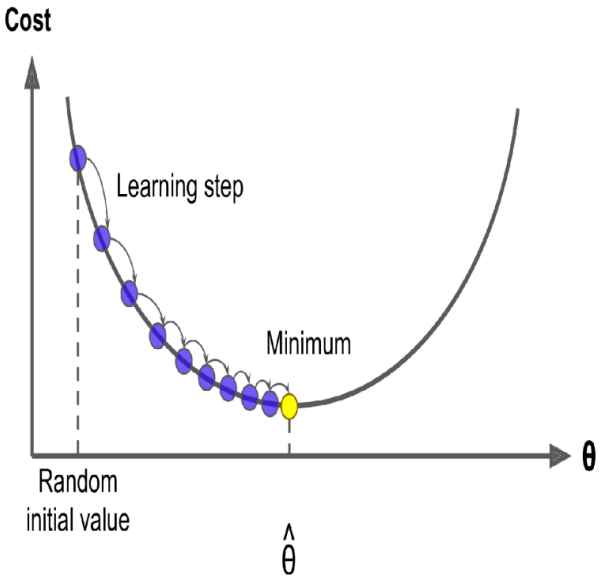



Figure 2.1: other loss functions

**Algorithm 1** Stochastic Gradient Descent Algorithm

**procedure** GRADIENTDESCENT($f, \nabla_x f, x_0, \eta, \epsilon$)
  $x \leftarrow x_0$
  **while** $||\nabla_x f(x)|| \geq \epsilon$ **do**
    $x \leftarrow x - \eta \nabla_x f(x)$
  **end while**
  **return** $x$
**end procedure**

$\epsilon$ and $\eta$ are constants, and $\nabla_x f(x)$ is the gradient of the function $f(x)$ evaluated at $x$. The algorithm updates model parameters $x$ iteratively based on the gradient of the loss function $f(x)$ computed on mini-batches of training data until the objective function reaches a minimum.

## 2.2 Gradient descent (GDM)

2. **Gradient descent with momentum (GDM)**: An extension for the GD method that adds a fraction of the previous update to the current update to reduce oscillations and accelerate convergence and avoid local minima. It is computed as a weighted average of past gradients, and it acts as a memory of the previous direction of the update.

## 2.3 Gradient descent with Adaptive learning rate (Adagrad)

3. **Adagrad**: A method that adapts the learning rate of each parameter based on the historical gradients of that parameter.

## 2.4 Gradient descent with Adaptive learning rate (Adadelta)

4. **Adadelta**: A method that adapts the learning rate using the momentum of each parameter based on the historical gradients of that parameter.

**Algorithm 2** Gradient Descent Algorithm with Momentum

**procedure** GradientDescentMomentum($f, \nabla_x f, x_0, \eta, \beta, \epsilon$)
    $x \leftarrow x_0$
    $v \leftarrow 0$
    **while** $||\nabla_x f(x)|| \geq \epsilon$ **do**
        $v \leftarrow \beta v + (1 - \beta) \nabla_x f(x)$
        $x \leftarrow x - \eta v$
    **end while**
    **return** $x$
**end procedure**

$\epsilon$,$\eta$ and $\beta$ are constants, and $\nabla_x f(x)$ is the gradient of the function $f(x)$ evaluated at $x$. The algorithm computes the gradient of the loss function $f(x)$ and updates the model parameters $x$ iteratively based on the momentum effect, computed as a weighted average of past gradients until the objective function reaches a minimum.

**Algorithm 3** Adagrad Algorithm

**procedure** Adagrad($f, \nabla_x f, x_0, \eta, \epsilon_1, \epsilon_2$)
    $x \leftarrow x_0$
    $s \leftarrow 0$
    **while** $||\nabla_x f(x)|| \geq \epsilon_1$ **do**
        $s \leftarrow s + \nabla_x f(x)^2$
        $x \leftarrow x - \frac{\eta}{\sqrt{s+\epsilon_2}} \nabla_x f$
    **end while**
    **return** $x$
**end procedure**

$\epsilon_1$, $\epsilon_2$ and $\eta$ are constants, and $\nabla_x f(x)$ is the gradient of the function $f(x)$ evaluated at $x$. The algorithm updates model parameters $x$ iteratively based on the historical sum of the squared gradients $s$, which acts as a measure of the second moment of the gradients, until the objective function reaches a minimum. The Adagrad algorithm adapts the learning rate of each parameter based on the historical sum of the squared gradients for that parameter.

## 2.5 Gradient descent with Momentum and Adaptive learning rate (Adam)

5. **Adam**: A method that combines the adaptive learning rate of Adadelta and the momentum of GDM to efficiently converge to a minimum.

## 2.6 Enhancements

In order to improve the gradient-based optimization method techniques. We first computed the gradient of the objective function and the constraints using automatic differentiation. Then, we implemented the following:

### 2.6.1 Line search algorithm (Adam, SGD and GDM)

6. **Line search**: A method to find the optimal step size along the descent direction. We terminated the algorithm when the norm of the gradient became smaller than a certain tolerance level.

**Algorithm 4** Adadelta Algorithm

**procedure** ADADELTA($f, \nabla_x f, x_0, \eta, \beta, \epsilon_1, \epsilon_2$)
  $x \leftarrow x_0$
  $s \leftarrow 0$
  **while** $||\nabla_x f(x)|| \geq \epsilon_1$ **do**
    $s \leftarrow \beta s + (1-\beta)\nabla_x f(x)^2$
    $x \leftarrow x - \frac{\eta}{\sqrt{s+\epsilon_2}}\nabla_x f$
  **end while**
  **return** $x$
**end procedure**

$\epsilon_1$, $\epsilon_2$, $\eta$ and $\beta$ are constants, and $\nabla_x f(x)$ is the gradient of the function $f(x)$ evaluated at $x$. The algorithm computes the gradient of the loss function $f(x)$ and updates the model parameters $x$ iteratively based on a moving average of the second moment of the gradients, computed as a exponential weighted average of past gradients until the objective function reaches a minimum.

**Algorithm 5** Adam Algorithm

**procedure** ADAM($f, \nabla_x f, x_0, \eta, \beta_1, \beta_2, \epsilon_1, \epsilon_2$)
  $x \leftarrow x_0$
  $v \leftarrow 0$
  $s \leftarrow 0$
  **while** $||\nabla_x f(x)|| \geq \epsilon_1$ **do**
    $v \leftarrow \beta_1 v + (1-\beta_1)\nabla_x f(x)$
    $s \leftarrow \beta_2 s + (1-\beta_2)\nabla_x f(x)^2$
    $x \leftarrow x - \frac{\eta}{\sqrt{s+\epsilon_2}} v$
  **end while**
  **return** $x$
**end procedure**

$\epsilon_1$,$\epsilon_2$, $\eta$, $\beta_1$, and $\beta_2$ are constants, and $\nabla_x f(x)$ is the gradient of the function $f(x)$ evaluated at $x$. The algorithm computes the gradient of the loss function $f(x)$ and updates the model parameters $x$ iteratively based on the momentum effect and the adaptive learning rate, computed as exponentially weighted moving averages of the past gradients and past squared gradients until the objective function reaches a minimum.

So, Line search algorithms are used to find the optimal learning rate at each step in this report. This shows how important this hyperparameter is for the performance of the algorithms. However, line search algorithms are not applied to all the algorithms because they are computationally expensive and may not be necessary for simple problems and functions. In such cases, the learning rate has to be manually selected and tuned. To understand how the learning rate affects the algorithms, see Figures 2.2 that illustrate the results for different values of the learning rate.

The learning rate is the most important hyperparameter. In this section, We have discussed its effects and challenges in detail. Other hyperparameters will be briefly introduced in the following chapters.

### 2.6.2 Bias correction (Adam)

7. **Bias correction**: A method that combines the adaptive learning rate of Adagrad and the momentum of GDM to efficiently converge to a minimum.

**Algorithm 6** Line Search Algorithm

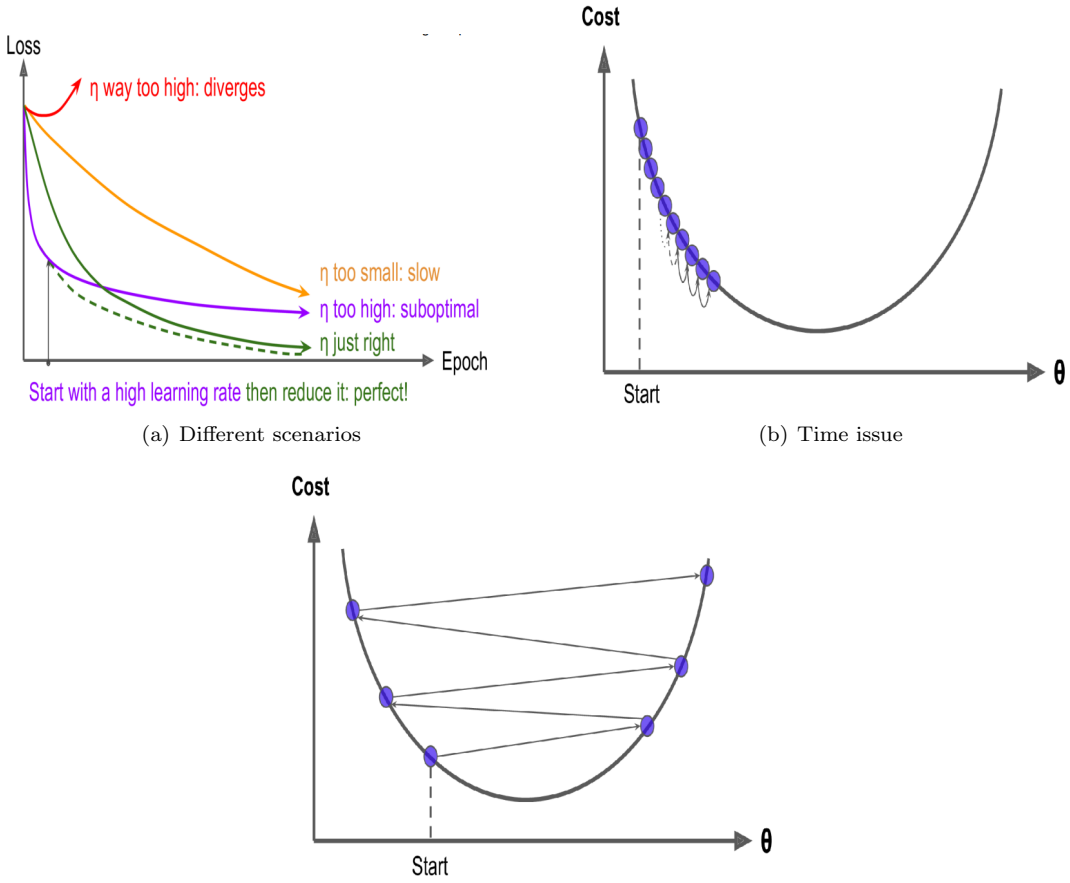
**procedure** LINE SEARCH($f(X)$, $X_0$, $\eta$, $\epsilon$)
    $Q(\eta) \leftarrow f(\mathbf{X_0} - \eta \nabla_{\mathbf{x}} \mathbf{f}(\mathbf{X_0}))$
    **while** $||\nabla_\eta Q(\eta)|| \geq \epsilon$ **do**
        $\eta \leftarrow \eta - \frac{\nabla_\eta Q}{\nabla_\eta^2 Q}$
    **end while**
    **return** $\eta$
**end procedure**

In this algorithm, $\epsilon$ is constant, and $\nabla_\eta Q(\eta)$ is the gradient of the function $Q(\eta)$ evaluated at $\eta$. The algorithm computes the optimal step size $\eta$ where this algorithm takes as input the objective function $f(X)$, an initial value $X_0$, an initial step size $\eta$, and a stopping criterion $\epsilon$ and iteratively updates the step size $\eta$ by computing the second derivative of the objective function and checking if the gradient of the objective function at the new step size satisfies the stopping criterion. Once the stopping criterion is met, the algorithm returns the optimal step size $\eta$ that minimizes the objective function.

**Algorithm 7** Adam Algorithm

**procedure** ADAM($f, \nabla_x f, x_0, \eta, \beta_1, \beta_2, \epsilon_1, \epsilon_2$)
    $x \leftarrow x_0$
    $v \leftarrow 0$
    $s \leftarrow 0$
    $t \leftarrow 0$
    **while** $||\nabla_x f(x)|| \geq \epsilon_1$ **do**
        $v \leftarrow \beta_1 v + (1 - \beta_1) \nabla_x f(x)$
        $s \leftarrow \beta_2 s + (1 - \beta_2) \nabla_x f(x)^2$
        $v^{corr} \leftarrow \frac{v}{1-\beta_1^t}$
        $s^{corr} \leftarrow \frac{s}{1-\beta_2^t}$
        $x \leftarrow x - \frac{\eta}{\sqrt{s^{corr}} + \epsilon_2} v^{corr}$
    **end while**
    **return** $x$
**end procedure**

$\epsilon_1$, $\epsilon_2$, $\eta$, $\beta_1$ and $\beta_2$ are constants, and $\nabla_x f(x)$ is the gradient of the function $f(x)$ evaluated at $x$. The algorithm computes the gradient of the loss function $f(x)$ and updates the model parameters $x$ iteratively based on the momentum effect and the adaptive learning rate, computed as exponentially weighted moving averages of the past gradients and past squared gradients until the objective function reaches a minimum.




(a) Different scenarios

(b) Time issue

(c) Divergence issue

Figure 2.2: other loss functions

### 2.6.3 Learning Decay (SGD and GDM)

In some situations, we may want to adjust the learning rate during the learning process without using the line search technique for some reason. For example, we may want to decrease the learning rate gradually. In this case, we can use some alternative methods that are less intelligent than the line search but also less computationally expensive. We will discuss some of these methods in this section.

**Time Based Decay**

8. **Time decay**: A type of learning rate decay where the learning rate is reduced based on the number of epochs or iterations by a certain factor to be decrease slowly over time.

**Step Based Decay**

9. **Step Decay**: A type of learning rate decay where the learning rate is reduced after a fixed number of epochs or iterations by a certain factor to be reduced at specific points during training.

**Algorithm 8** Time Decay Algorithm

**procedure** TIME DECAY($\eta, \nabla\eta, t$)
    $\eta \leftarrow \frac{\eta}{(1+\nabla\eta\ t)}$
    **return** $\eta$
**end procedure**

$\eta$, $\nabla\eta$ and $t$ are constants. The algorithm computes iteratively the learning rate $\eta$ with new iteration $t$ and it decreases with each iteration with the decay rate $\nabla\eta$ till it reach very small values of learning rate leading to almost no change in steps.

**Algorithm 9** Step Decay Algorithm

**procedure** STEP DECAY($\eta, \nabla\eta, t, b$)
    **if** $\mod \frac{t}{b} = 0$ **then**
        $\eta \leftarrow \eta\ \nabla\eta$
    **end if**
    **return** $\eta$
**end procedure**

$\eta$, $\nabla\eta$, $t$ and $b$ are constants. The algorithm computes the learning rate $\eta$ with every batch $b$ of iterations $t$ where the learning rate decreases with each successful batch with the decay rate $\nabla\eta$ till it reach very small values of learning rate leading to almost no change in steps.

**Exponential Based Decay**

10. **Exponential Decay**: A type of learning rate decay where the learning rate is reduced exponentially over time by a certain factor after every epoch or iteration to be reduced quickly in the beginning and then gradually over time.

**Algorithm 10** Exponential Decay Algorithm

**procedure** EXPONENTIAL DECAY($\eta, \nabla\eta, t$)
    $\eta \leftarrow \eta\ \nabla\eta^t$
    **return** $\eta$
**end procedure**

$\eta$, $\nabla\eta$, and $t$ are constants. The algorithm computes the learning rate $\eta$ with every new iterations $t$ where the learning rate decreases exponentially with each iteration with the decay rate $\nabla\eta$ till it reach very small values of learning rate leading to almost no change in steps.

**Performance Based Decay**

11. **Performance Decay**: A type of learning rate decay where the learning rate is reduced based on the performance of the model on a validation set when it does not improve after a certain number of epochs or iterations to ensure the model has stopped learning (Overfitting[3] issue).

[3]Overfitting is a problem that occurs when a machine learning model learns the patterns of the training data too well, and fails to generalize to new data. This means that the model performs well on the training data, but poorly on the test data or unseen data. Overfitting can happen when the model is too complex or has too many parameters compared to the amount of data available. Overfitting can lead to inaccurate predictions and poor performance.

**Algorithm 11** Performance Decay Algorithm

1: **procedure** PERFORMANCE DECAY($\eta, \nabla\eta, \mathcal{L}_t, \mathcal{L}_{t-1}$)
2: **if** $\mathcal{L}_t > \mathcal{L}_{t-1}$ **then**
3: $\eta \leftarrow \eta\ \nabla\eta$
4: **end if**
5: **return** $\eta$
6: **end procedure**

$\eta$ and $\nabla\eta$ are constants. The algorithm computes a new learning rate $\eta$ whenever the current loss $\mathcal{L}_t$ is greater than the previous loss $\mathcal{L}_{t-1}$ where the learning rate decreases by the time of occurrence of this difference with the decaying rate $\nabla\eta$ till it reach very small values of learning rate leading to almost no change in steps.

# Chapter 3

# Test Functions

## Intro

We used test functions from the SFU optimization test function library, including the Eggholder function, Trid function, Matyas function, Three-Hump Camel function, Michalewicz function, and Styblinski-Tang function. These test functions represent different types of objective functions, including those with many local minima, bowl-shaped functions, plate-shaped functions, Valley-Shaped, Valley-Shaped, and other shapes. Additionally, we tested each algorithm on each function in a range of search spaces with different dimensions.

## 3.1 Many Local Minima

### 3.1.1 Definition

1. **Many Local Minima**: The Eggholder function, defined as

$$f(\mathbf{x_1}, \mathbf{x_2}) = -(x_2 + 47)\sin\left(\sqrt{|\frac{x_1}{2} + x_2 + 47|}\right) - x_1\sin\left(\sqrt{|x_1 - (x_2 + 47)|}\right) \tag{3.1}$$

### 3.1.2 Input Domain

The function is usually evaluated on the square $x_i \in [-512, 512]$, for all $i = 1, 2$.

### 3.1.3 Global Minima

$f(X^*) = -959.6407$, at $X^* = \begin{bmatrix}512 & 404.2319\end{bmatrix}^T$

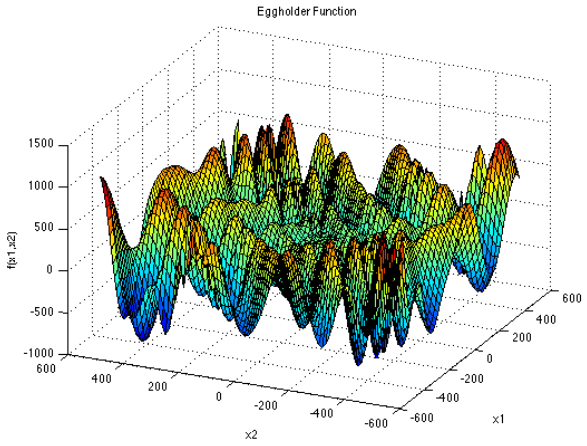



Figure 3.1: Eggholder

## 3.2 Bowl-shaped

### 3.2.1 Definition

2. **Bowl-shaped**: The Trid function, defined as

$$f(\mathbf{X}) = \sum_{i=1}^{d}(x_i - 1)^2 - \sum_{i=2}^{d} x_i x_{i-1} \tag{3.2}$$

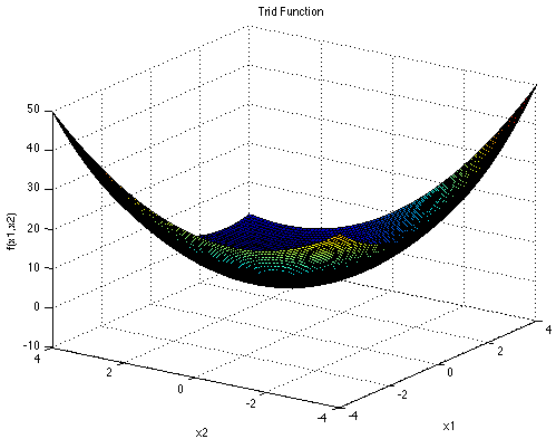



Figure 3.2: Trid

### 3.2.2 Input Domain

The function is usually evaluated on the hypercube $x_i \in [-d^2, d^2]$, for all $i = 1, \ldots, d$

### 3.2.3 Global Minimum

$f(\mathbf{X}^*) = \frac{-d(d+4)(d-1)}{6}$, at $x_i = i(d+1-i)$, for all $i = 1, 2, \ldots, d$

## 3.3 Plate-shaped

### 3.3.1 Definition

3. **Plate-shaped**: The Matyas function, defined as

$$f(\mathbf{x_1}, \mathbf{x_2}) = 0.26(x_1^2 + x_2^2) - 0.48x_1x_2 \tag{3.3}$$

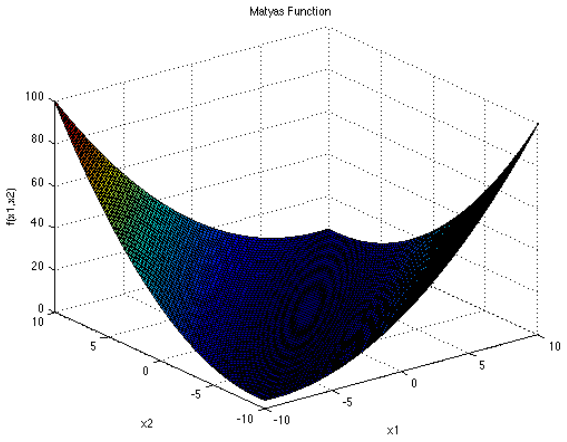



Figure 3.3: Matyas

### 3.3.2 Input Domain

The function is usually evaluated on the square $x_i \in [-10, 10]$, for all $i = 1, 2$.

### 3.3.3 Global Minimum

$f(\mathbf{X}^*) = 0$, at $X^* = \begin{bmatrix} 0 & 0 \end{bmatrix}^T$

## 3.4 Valley-Shaped

### 3.4.1 Definition

4. **Valley-Shaped**: The Three-Hump Camel function, defined as

$$f(\mathbf{x_1}, \mathbf{x_2}) = 2x_1^2 - 1.05x_1^4 + \frac{x_1^6}{6} + x_1x_2 + x_2^2 \tag{3.4}$$

### 3.4.2 Input Domain

The function is usually evaluated on the square $x_i \in [-5, 5]$, for all $i = 1, 2$.

### 3.4.3 Global Minimum

$f(\mathbf{X}^*) = 0$, at $X^* = \begin{bmatrix} 0 & 0 \end{bmatrix}^T$

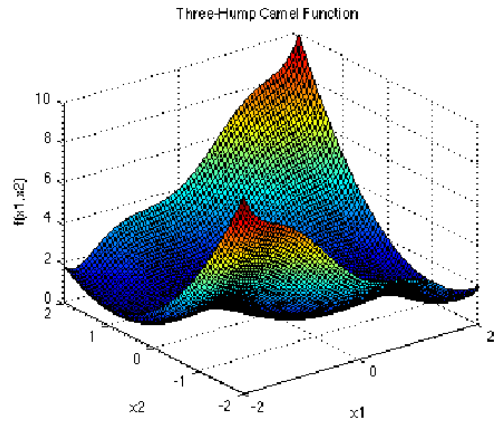



Figure 3.4: Three-Hump Camel

## 3.5 Steep Ridges/Drops

### 3.5.1 Definition

5. **Steep Ridges/Drops**: The Michalewicz function, defined as

$$f(\mathbf{X}) = -\sum_{i=1}^{d} \sin(x_i) \sin^{2m}\left(\frac{ix_i^2}{\pi}\right) \tag{3.5}$$

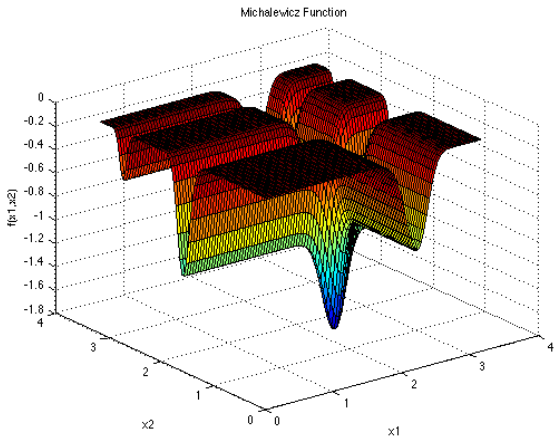



Figure 3.5: Michalewicz

### 3.5.2 Input Domain

The function is usually evaluated on the hypercube $x_i \in [0, \pi]$, for all $i = 1, \ldots, d$

### 3.5.3 Global Minimum

at $d = 2 : f(\mathbf{X}^*) = -1.8013$, at $X^* = \begin{bmatrix} 2.20 & 1.57 \end{bmatrix}^T$

at $d = 5 : f(\mathbf{X}^*) = -4.687658$

at $d = 10 : f(\mathbf{X}^*) = -9.66015$

## 3.6 Other

### 3.6.1 Definition

6. **Other**: The Styblinski-Tang function, defined as

$$f(\mathbf{X}) = \frac{1}{2}\sum_{i=1}^{d}(x_i^4 - 16x_i^2 + 5x_i) \tag{3.6}$$

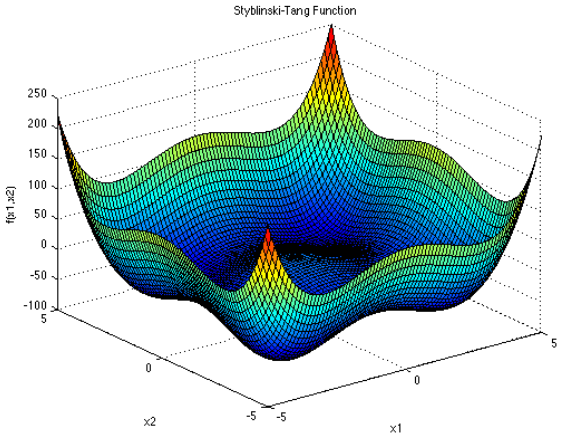



Figure 3.6: Styblinski-Tang

### 3.6.2 Input Domain

The function is usually evaluated on the hypercube $x_i \in [-5, 5]$, for all $i = 1, \ldots, d$.

### 3.6.3 Global Minimum

$f(\mathbf{X}^*) = -39.16599d$, at $X^* = \begin{bmatrix} -2.903534 & \ldots\ldots & -2.903534 \end{bmatrix}^T$

# Chapter 4

# Experimental Results

### experimental setup

In our experimental setup for the optimization project, we implemented various optimization algorithms using Python with the NumPy and Tensorflow libraries. For SGD and GDM, we used a fixed learning rate, while for Adagrad, Adadelta, and Adam, we experimented with different learning rates. Other hyperparameters, such as momentum coefficients, were set to commonly used values.

Each algorithm was run on each test function for 2000 iterations and the objective function value was recorded at each iteration.

It is important to note that The aim of this experiment was not to find the fastest algorithm, but to compare them under the same conditions

## 4.1 Original algorithms

In this section, we will see the performance of different algorithms without any extensions and under the same conditions.

1. In the first step, the step size for SGD Figure 4.1(a) is usually longer than for GDM Figure 4.1(b). This is due to the fact that GDM requires time to accumulate momentum, whereas SGD utilises the full gradient immediately. This feature can be useful at times as it reduces the risk of overshooting, when the system undergoes new sudden changes in gradient.

2. As shown in Figure 4.2(a) SGD got stuck in the local minima where the gradient was zero, and the updates stopped. On the other hand, in Figure 4.2(b) GDM allowed the optimization process to continue even when the gradient was zero because it adds a fraction of the previous gradient updates to the current one. Therefore, GDM can escape shallow local minima and move towards the global minimum.

   The performance of GDM varies depending on the initial point -initial values for $\nabla_x f(x)$-, as illustrated in Figure 4.2. In some cases, GDM may fail to converge if the initial values of the loss function derivatives

are unfavourable. To address this limitation, we will present more e robust methods that can guarantee convergence regardless of the initial point.

3. In the Figure 4.3(b) given, we have two variables $X_1$ and $X_2$, where $X_1$ has a steeper gradient than $X_2$. When using Adagrad, as shown in Figure 4.3(a) the algorithm adapts the learning rate separately for each weight as follows: For $X_1$, since it has a steeper gradient Adagrad reduced the effective learning rate to make sure that the variable doesn't receive too large of an update to prevent instability or divergence. For $X_2$, since it has a less steep gradient Adagrad increased the effective learning rate, to allow the weight to receive larger updates, which could help it converge faster.

   Thus changes smooth out the fluctuations and reduce the updates in directions that have little impact on the loss function as shown in Figure 4.3

4. in Figure 4.4(a) shows the problem of Adagrad as the accumulation of the sum of the full squares of past gradients can cause the effective learning rate to decrease too much as training progresses. Which caused the extremely slow convergence. In contrast, Adadelta takes a fraction of the squares of past gradients (exponentially moving average). This allows to maintain a more stable and appropriate learning rate throughout the training process.

   This idea lacks originality and has been already discussed in the literature. Can you specify the exact section or page number where you encountered it?

5. Figure 4.4 show the performance of the five algorithms on the EggHolder LossFunction 3.1 loss function. It is clear that the Adagrad algorithm has the highest error and the slowest convergence rate. On the other hand, the SGD and GDM algorithms converge faster but none of them can reach the global minimum

6. Figure 4.5 compares the performance of the ordinary gradient descent with other algorithms on different types of LossFunctions 3.2, 3.3 and 3.4. It shows that the ordinary gradient descent has the lowest error and the fastest convergence rate among all the algorithms. Moreover, the Adagrad algorithm has the slowest and worst performance on all the loss functions.

7. Figure 4.6(b) confirms the previous findings, but here we can see that the Adam and Adadelta algorithms converge faster than the SGD and GDM algorithms. on the other hand, As shown in Figure 4.6(a), the GDM and SGD algorithms have poor performance on these LossFunctions 3.5(Steep Ridges/Drops), while the Adadelta algorithm converges rapidly but exhibits some oscillations. The Adam algorithm achieves fast convergence and then the Adagdelta algorithm follows.

## 4.2 Enhancements

In this section, we will see the performance of different algorithms with some extensions.

1. It is possible that GDM may struggle to accelerate the learning process. When the learning process starts from an area with extremely small gradients see Figure 4.2, the momentum term may not accumulate to a large enough value to provide significant acceleration to the weight updates, leading to slow convergence or stagnation.

   One way to address this issue is to use adaptive learning rate methods such as line search as shown. It

managed to get to the global minimum with **EXACTLY ONE iteration** as shown in Figure 4.7(a). In Figure 4.7(b) illustrates the failure of GDM to reach the global minimum even after 2000 iterations.

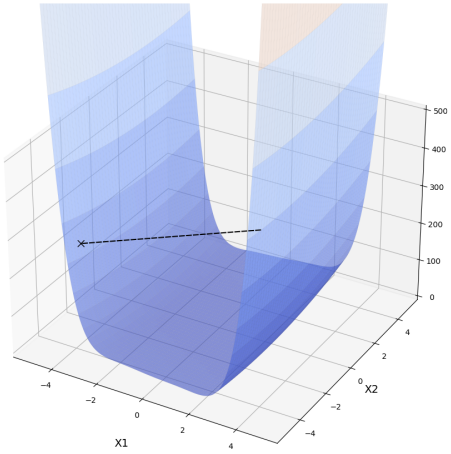
2. Figure 4.7(c) compares the performance of the five algorithms on EggHolder Function 3.1 for **200 iteration** without any extensions.

3. In Figure 4.7(d) and 4.8(b) show the superior performance of the Adam algorithm with Line Search Extension on EggHolder Function 3.1 for **200 iterations**, which exceeds the performance of the original Adam algorithm without Line Search Extension for **2000 iterations**.

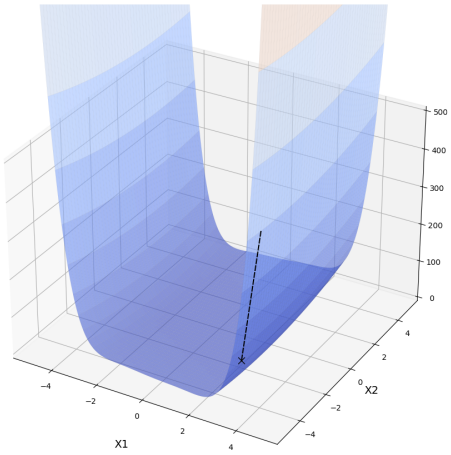
   Figure 4.7(d) Starting point at $X_{init} = \begin{bmatrix} 2.8 & 1.57 \end{bmatrix}^T$

   Figure 4.8(b) Starting point at $X_{init} = \begin{bmatrix} 300.0 & 250.0 \end{bmatrix}^T$

4. In Figure 4.8(a) We applied the bias correction technique to the Adam algorithm with Line Search Extension and observed that it improves the performance in the initial iterations.

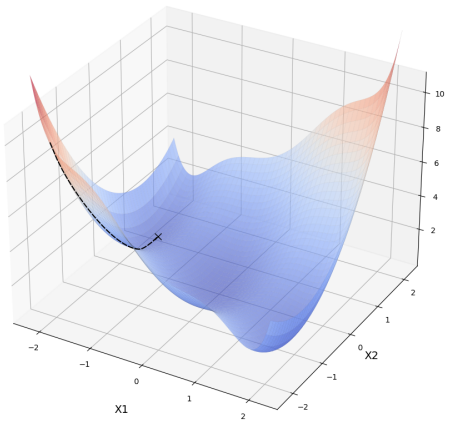



(a) SGD

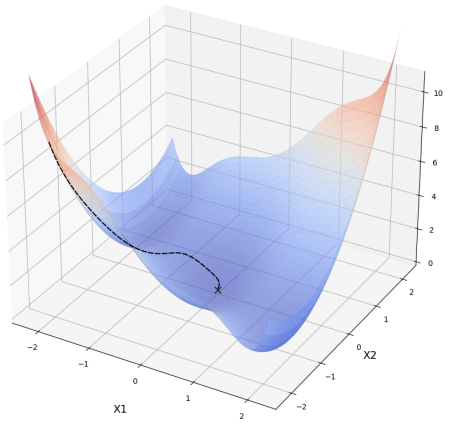



(b) GDM

Figure 4.1: other loss functions




(a) ThreeHumpCamel with SGD




(b) ThreeHumpCamel with GDM

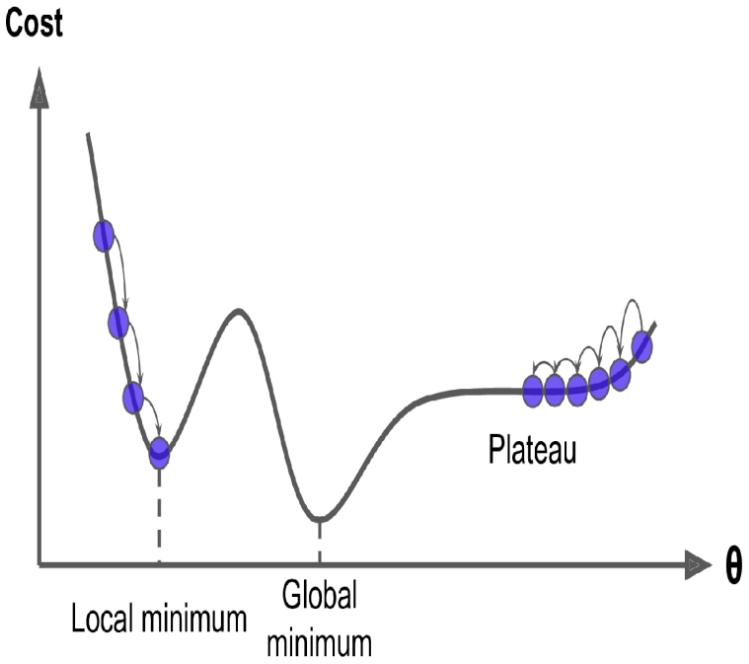
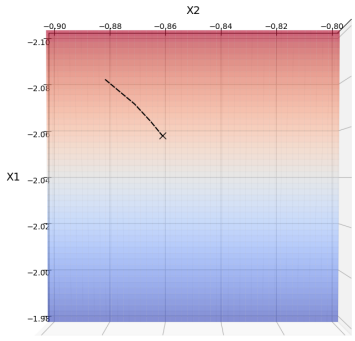
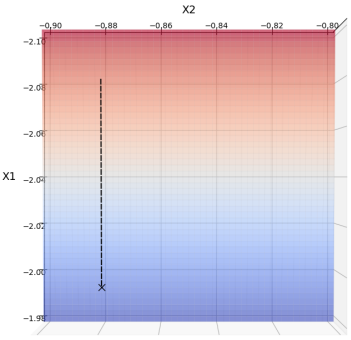



Figure 4.2: Some cases




(a) Michalewicz with Adagrad




(b) Michalewicz with SGD

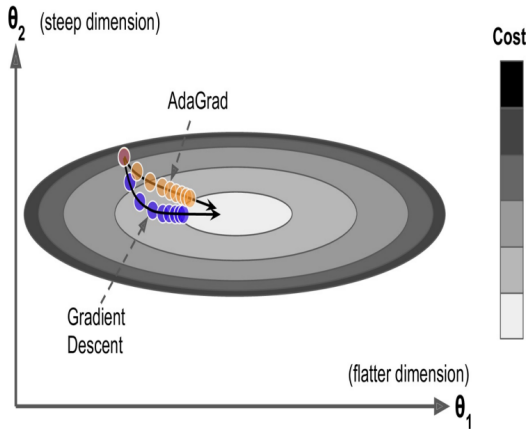
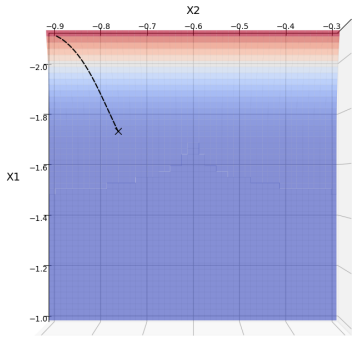
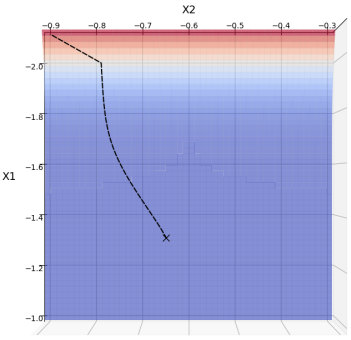


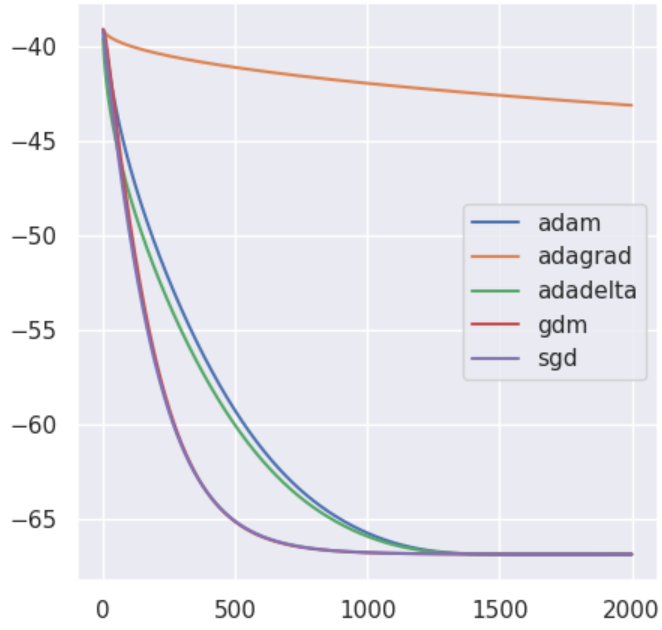

Figure 4.3: Observe the effect of the squared gradient in determining the direction for the stepsize



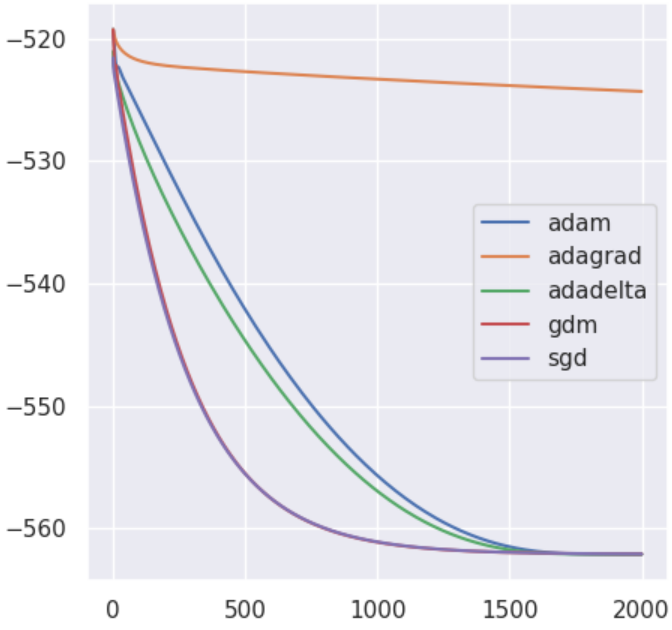

(a) Michalewicz with Adagrad




(b) Michalewicz with Adadelta

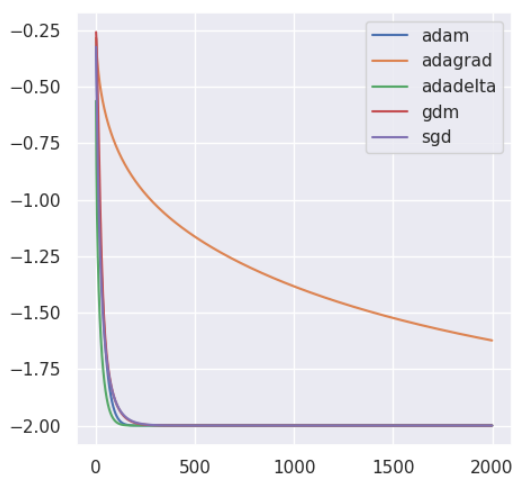



(c) EggHolder starting at (2.8, 1.1)

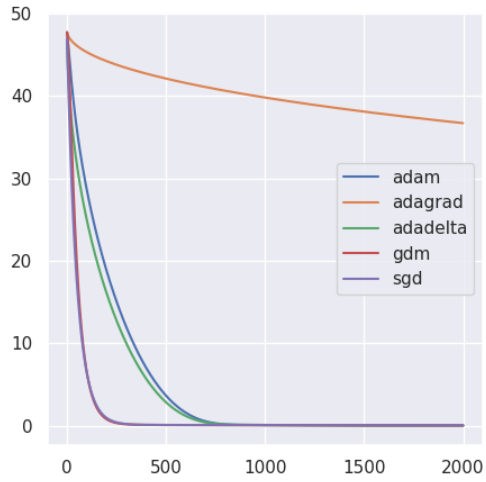



(d) EggHolder starting at (300.0, 250.0)

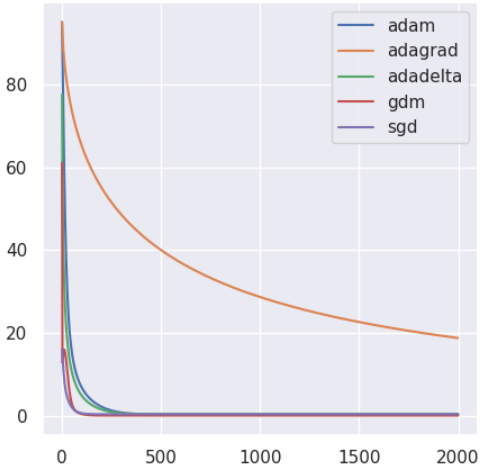
Figure 4.4: EggHolder




(a) Trid

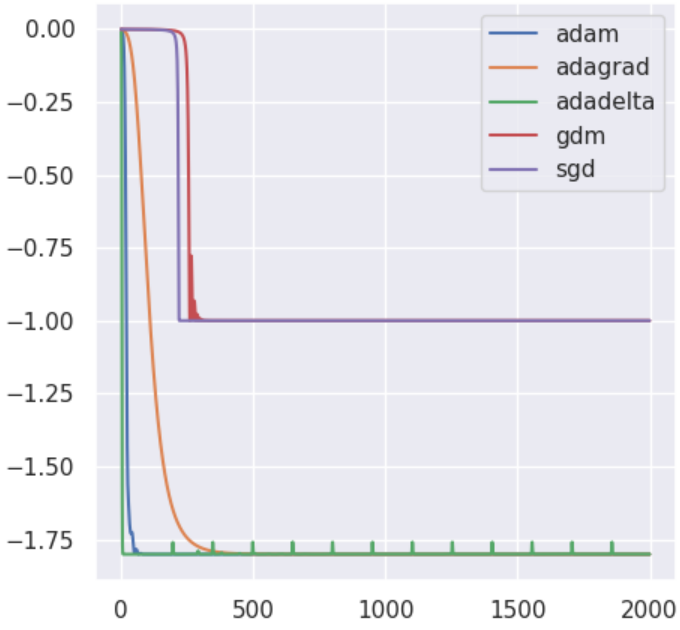



(b) MATYAS

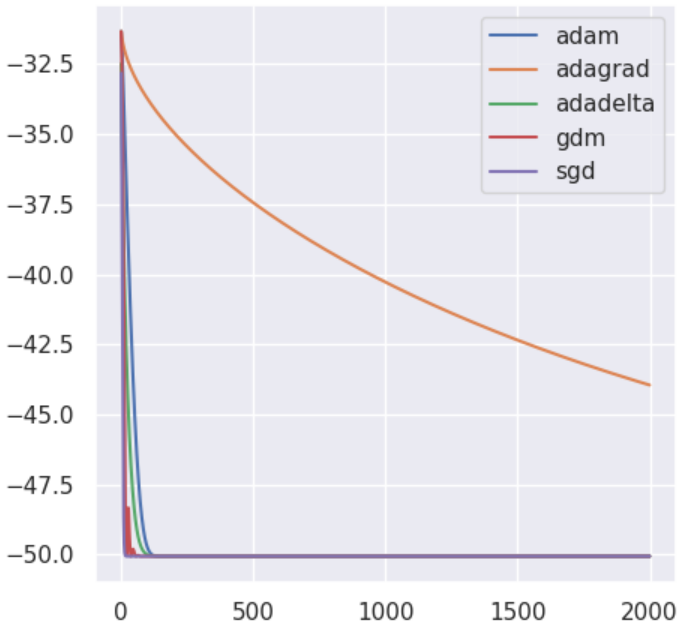



(c) ThreeHumpCamel

Figure 4.5: other loss functions

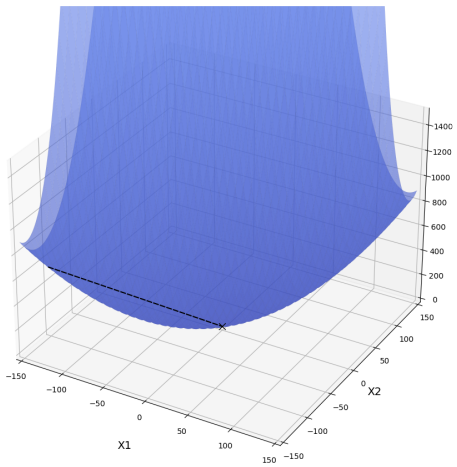



(a) Michalewicz

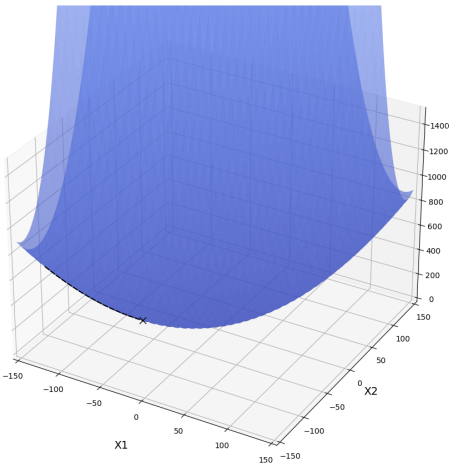



(b) StyblinskiTang

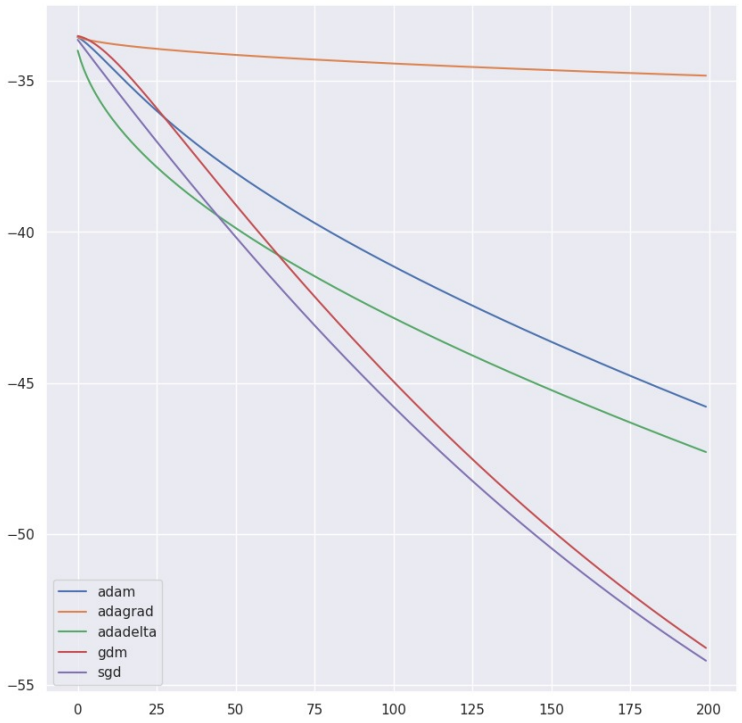
Figure 4.6: other loss functions



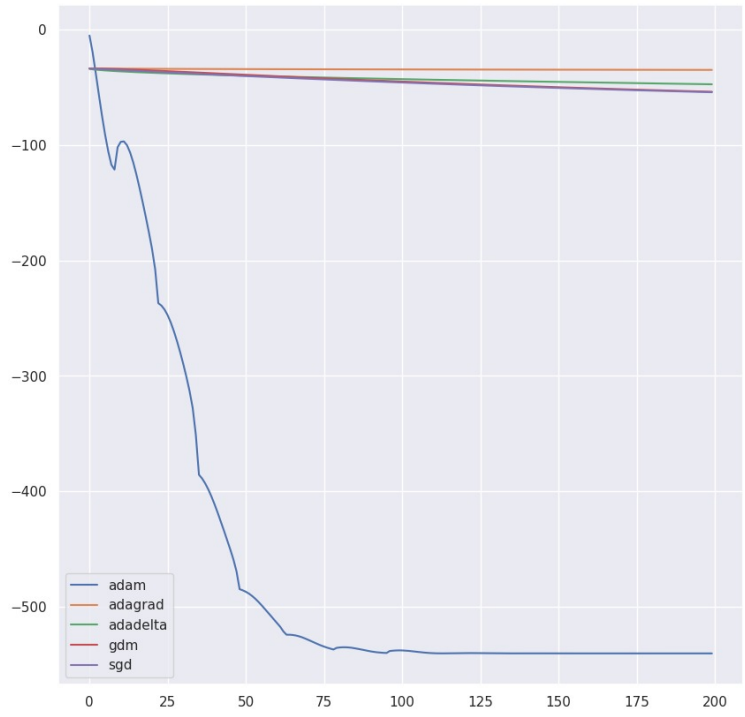

(a) MATYAS with SGD




(b) MATYAS with GDM

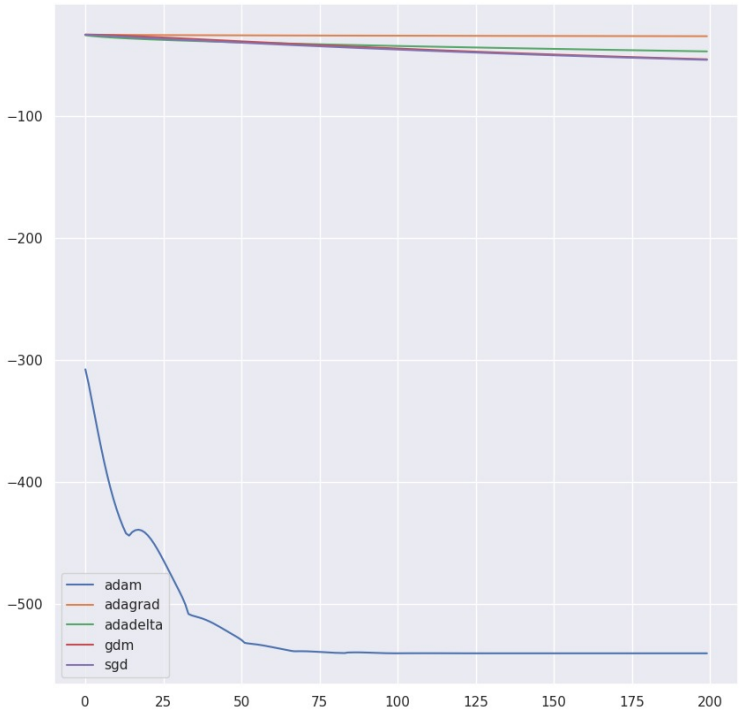



(c) Line Enhancement

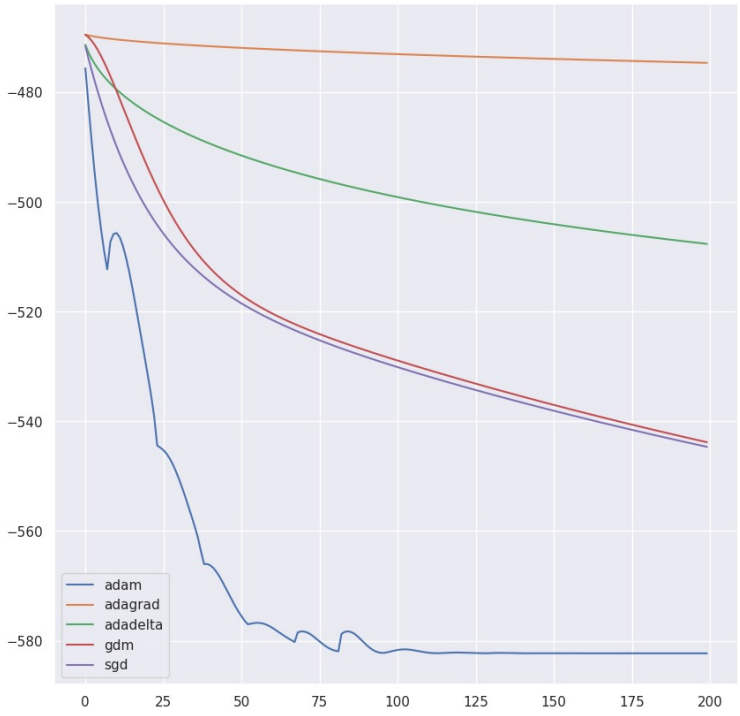



(d) Line Enhancement

Figure 4.7: EggHolder




(a) Line Enhancement




(b) Line Enhancement

Figure 4.8: EggHolder

# Chapter 5

# Conclusion

1. **Adam (Adaptive Moment Estimation)**: Adam is an adaptive learning rate optimization algorithm that combines the advantages of both AdaGrad and RMSProp (Adadelta). It computes adaptive learning rates for each parameter by considering the first and second moments of the gradients. Adam is known for its fast convergence and robustness to different types of loss functions.

2. **Adadelta**: Adadelta is an extension of AdaGrad that addresses the diminishing learning rates problem. It uses a moving average of the squared gradients to adapt the learning rate for each parameter. Adadelta is more robust to different types of loss functions compared to AdaGrad.

3. **Adagrad (Adaptive Gradient Algorithm)**: Adagrad adapts the learning rate for each parameter based on the past gradients. It performs well on sparse data and is less sensitive to the choice of the initial learning rate. However, it may suffer from a diminishing learning rate, making it less suitable for some types of loss functions.

4. **GD with momentum**: Gradient Descent with momentum is a modification of the standard Gradient Descent algorithm that incorporates momentum to accelerate convergence. It helps to overcome local minima and oscillations in the optimization process. It performs well on loss functions with shallow valleys and plateaus.

5. **Stochastic Gradient Descent (SGD)**: SGD is a variant of Gradient Descent that updates the parameters using a random subset of the data (mini-batch) instead of the entire dataset. It is computationally efficient and can escape local minima, but it may have a slower convergence rate compared to other algorithms.

Please **NOTE** that when applying these methods to deep learning problems, you may encounter the following output after the first epoch: loss: nan - mae: nan. This may indicate that your network is experiencing exploding[1] gradients. This is a common issue if you used SGD as an optimizer and chose a

[1]Exploding gradients are a problem that occurs when the error gradients in a neural network become very large during training. This can cause the network to become unstable and unable to learn from the data. The weights of the network can also become so large that they overflow and result in NaN values. Exploding gradients are more likely to happen in deep networks or recurrent networks, where the gradients can accumulate and grow exponentially over many layers or time steps.

learning rate that is too high. To resolve this issue, you may try reducing the learning rate or using Adam with the default learning rate.

Performance on different loss functions:

1. **Eggholder (Many Local Minima)**: Adam and Adadelta are likely to perform better on this type of loss function due to their adaptive learning rates and ability to escape local minima.

2. **Trid (Bowl-Shaped)**: All algorithms should perform well on this type of loss function, but Adam and GD with momentum may converge faster due to their adaptive learning rates and momentum, respectively.

3. **Matyas (Plate-Shaped)**: GD with momentum and Adam are expected to perform well on this type of loss function due to their ability to overcome plateaus and accelerate convergence.

4. **Three-Hump Camel (Valley-Shaped)**: Adam, Adadelta, and GD with momentum should perform well on this type of loss function, as they can navigate the valleys efficiently.

5. **Michalewicz (Steep Ridges/Drops)**: Adam and Adadelta are likely to perform better on this type of loss function due to their adaptive learning rates, which help them navigate steep gradients.

6. **Styblinski-Tang**: Adam, Adadelta, and GD with momentum should perform well on this type of loss function, as they can handle the complex landscape and avoid getting stuck in local minima.

We also found that the performance of all algorithms degraded as the dimensionality of the search space increased. This suggests that algorithm performance should be carefully evaluated on high-dimensional problems, which are common in many real-world optimization problems.

In this project, we compared the performance of five popular optimization algorithms on a range of test functions. We found that Adam consistently outperforms the other algorithms but also observed some differences in performance on different types of functions and also we saw the effect of some extensions like line search algorithm, bias correction and etc. These findings have important implications for researchers and practitioners in the field of optimization and can inform the development of new and improved algorithms in the future.

Overall, we found that Adam consistently outperformed the other algorithms in terms of convergence speed and final objective function value on all test functions. However, we also observed some differences in performance on different types of functions. For example, Adagrad and Adadelta performed better than SGD and GDM on test functions with many local minima, while Adam performed better on test functions with complex plate-shaped objective functions. And we also made some extensions to make the algorithms more intelligent and we saw how the line search algorithm made a huge difference in the performance. One of the main disadvantages of the extensions we made is the time required to finish them

For future work in this field, we propose to extend these algorithms with the line search strategy to improve their suitability and efficacy for real-world applications. We also recommend exploring more

To prevent exploding gradients, one can use techniques such as gradient clipping, which limits the size of the gradients during training.

| Algorithm | Description | Advantages | Disadvantages | Hyperparameters |
|---|---|---|---|---|
| Adam | Combines Momentum and AdaGrad, using exponential moving averages of past gradients and past squared gradients | Adaptive learning rates, fast convergence, robust to noisy gradients | Requires more memory and computation than simple SGD, may not generalize well on some data sets | $\eta$: Learning rate, $\beta_1$: is the momentum coefficient that controls how much of the past gradients are retained, $\beta_2$ is the momentum coefficient that controls how much of the past squared gradients are retained, $\epsilon$: Adding a negligible constant to avoid the undefined case of zero denominators. |
| Adadelta | Extension of AdaGrad that uses a fixed window of past gradients instead of all of them | Adaptive learning rates, faster convergence than AdaGrad, robust to noisy gradients | Requires more memory and computation than simple SGD, may not generalize well on some data sets | $\eta$: Learning rate, $\beta$: is the momentum coefficient that controls how much of the past squared gradients are retained, $\epsilon$: Adding a negligible constant to avoid the undefined case of zero denominators. |
| Adagrad | Adapts the learning rate to each parameter, performing larger updates for infrequent and smaller updates for frequent parameters | Adaptive learning rates | Aggressive and monotonically decreasing learning rate, may stop learning too early | $\eta$: Learning rate, $\epsilon$: Adding a negligible constant to avoid the undefined case of zero denominators. |
| GD with momentum | Accelerates the convergence of gradient descent by adding a fraction of the previous update to the current one | Faster convergence, overcome local minima and oscillations | Requires more memory and computation than simple SGD, may overshoot the optimum | $\eta$: Learning rate, $\beta$: is the momentum coefficient that controls how much of the past gradients are retained. |
| Stochastic gradient descent (SGD) | Updates the parameters in the opposite direction of the gradient of the objective function with respect to a randomly selected subset of the data | Simple and widely used, can handle large datasets and online learning154 | Slow convergence, sensitive to learning rate and initial values, prone to local minima and oscillations | $\eta$: Learning rate |

Table 5.1: Comparison between algorithms Adam, Adadelta, Adagrad, GD with momentum and Stochastic gradient descent

powerful and reliable second-order methods that can cope with large-scale and non-convex optimization problems. Furthermore, we suggest examining how different optimization methods affect the performance of various deep learning models, such as supervised, unsupervised, and hybrid ones.

# Bibliography


[1] Baeldung. Adam optimizer — baeldung on computer science, 2021.

[2] Papers With Code. Adadelta explained — papers with code, 2021.

[3] Papers With Code. Adagrad explained — papers with code, 2021.

[4] Dive into Deep Learning. Optimization — dive into deep learning 1.0.0-beta0 documentation - d2l, 2021.

[5] Keras. Optimizers - keras, 2021.

[6] Diederik P Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

[7] Andrew Ng. Adam optimization algorithm - optimization algorithms — coursera, 2017.

[8] John Pomerat, Aviv Segev, and Rituparna Datta. Adam, 2020.

[9] PyTorch. Optimizers — pytorch 2.0 documentation, 2021.

[10] scikit learn. 1.5. stochastic gradient descent — scikit-learn 1.2.2 documentation, 2022.

[11] Virginia Tech. Adagrad – optimization in machine learning, 2018.

[12] Cornell University. Momentum - cornell university computational optimization open textbook initiative, 2020.

[13] Cornell University. Stochastic gradient descent - cornell university computational optimization open textbook initiative, 2020.

[14] Wikipedia. Stochastic gradient descent - wikipedia, 2022.