A Neural Network Approach to Minimize Loss Functions using Optimizers

Rusmia Sattar[†]

[†] *Department of Mathematics and Statistics, Memorial University of Newfoundland, St. John's (NL) A1C 5S7, Canada*

E-mail: rsattar@mun.ca

Abstract

In the realm of neural network optimizers, we compared the effectiveness of six well-known neural network optimization techniques in this research. We analyzed the Stochastic Gradient Descent (SGD), Momentum, Nesterov Accelerated Momentum, AdaGrad, RMSProp, and Adam to minimize the loss of some complex functions. We used these techniques to determine global minima for the four specific functions: The Six-Hump Camel function, the Michalewicz function, The Ackley function, and the Bohachevsky function and the results were extremely insightful. We discuss ways to improve convergence and study the number of steps needed to attain the minimum by examining the effects of hyperparameters on optimization. Overall, our results show how flexible and resilient these optimizers are while navigating complex optimization environments and avoiding local minima, making a significant contribution to the fields of deep learning and real-world problem-solving.

Keywords: Neural Network, Optimizers, Loss Function, Global minima

1 Introduction

The fundamental concept of deep learning in machine learning is neural network optimization. It is the process of determining the optimum set of parameters to minimize a certain loss function.[1] The optimization algorithm utilized to properly evaluate this high-dimensional parameter space is a significant factor in attaining improved model performance.

The importance of optimizing neural networks lies in their ability to unlock the full potential of deep learning models, enabling them to perform better, train faster, and generalize more effectively. The implications are not only limited to the field of AI but extend to a wide range of practical applications that benefit from improved optimization techniques.[2]

In order to find the global minimum of complex functions and reduce the resulting loss, this research implements and investigates a variety of optimization techniques in neural networks.[3] Our goal is to provide insight into the behavior of various optimizers in the context of challenging conditions with complex loss functions, basically how the optimizers behave in certain cases.

We've chosen a group of optimization methods to implement on this project, including Adam, AdaGrad, RMSProp, Momentum, Nesterov Accelerated Momentum, Stochastic Gradient Descent (SGD). Modern deep learning models have benefited greatly from the training provided by these approaches.

Adam: It is a method for efficient stochastic enhancement that requires concise memory specification and first-order gradients. The Optimizer evaluates the first and second moments of the gradients to compute discrete flexible learning rates for various parameters.[1]

AdaGrad: AdaGrad is a slope-based advancement computation. It is a short form for Adaptive Gradient. It is a learning rate adaptation technique that has the advantage of being particularly simple to use. The square root of the sum of squares of the historical component-wise gradient is used to infer the learning rate. Its typical effect is to reduce the successful advance size as a function of time.[1]

RMSProp: It is shortly known for Root Mean Square Propagation. Similar to gradient descent with momentum, it makes use of the exponentially weighted average of the gradients, but parameter updates are different.[1]

Momentum: Momentum optimization is a method that typically achieves higher convergence rates. An explanation for this modification can be found in the optimization problem's physical aspect. Specifically, the loss can be understood as the altitude of a hilly area.[1]

Nesterov Accelerated Momentum: Nesterov Accelerated Momentum Optimization is an updated version of Momentum Optimization. It consistently performs somewhat better than standard momentum in practice and has stronger theoretical converge which guarantees for convex functions.[1]

SGD: Stochastic gradient descent or SGD as it is commonly known, is a monotonous design that improves a distinct unbiased function, a stochastic estimate of gradient descent intensification. It is referred regarded as stochastic because examples are selected randomly rather than in a group setting or according to how they appear in the training set.[1]

With the help of this study, we hope to empirically assess their performance in reducing complicated loss functions and analyze the trade-offs they present in terms of convergence speed, stability, and escape from local minima.

We use a variety of complex functions, including the Six-hump Camel function, the Michalewicz function, the Ackley function, and the Bohachevsky function, to test the optimizers. These are the best functions for assessing optimizer performance since they represent difficult environments with numerous local minima. Our analysis takes into account a number of variables, such as the rate at which each optimizer converges, the ease with which it may break out of local minima, and the overall number of steps needed to arrive at the global minimum. To get a complete picture of each optimizer's behavior, we run tests with a range of learning rates and starting points.

We want to assist practitioners in selecting the most appropriate optimization method for their deep learning projects. We are able to do that by analyzing neural network optimizers in-depth on complex functions. The interplay between mathematics, algorithms, and the search for the most effective way to increase model performance is also highlighted in this project as a testament to the art and science of optimization.

2 Methods

We examine the effectiveness of various neural network optimization strategies in this study. We analyze their efficiency in minimizing complicated loss functions that are created purposely to make it difficult for them to navigate through complex surfaces with several local minima. In order to achieve the best model performance, it is essential to comprehend how these optimizers behave during the training of deep neural networks.[4]

First, we start our experiment with a simple loss function:

$L(p) = sin2p + asin4p$

We define the loss function and its gradient. Then we define all six optimizers and implement to find the minima of this function. We take six optimization algorithms into account. These include Adam, AdaGrad, RMSProp, Momentum, Nesterov Accelerated Momentum, Stochastic Gradient Descent (SGD), and Momentum. Every optimizer has a unique set of guidelines and controls for altering model parameters, learning rates, and handling gradient data. We discover the advantages and disadvantages of various algorithms by analyzing them. For our experiment with intricate loss function, we check for each optimizer whether they are reaching their target error or not. We also keep a count of how many steps each of them requires to find the minimum of the loss function.

```
# Defining the loss function L(p) with a given value of a
def loss_function(p, a):
    return np.sin(2 * p) + a * np.sin(4 * p)

# Defining the gradient of the loss function
def gradient(p, a):
    return 2 * np.cos(2 * p) + 4 * a * np.cos(4 * p)
```

We define all six optimizers as follows:

```
    #Defining the Optimizer functions sgd, momentum, nesterov momentum, adagrad, rmsprop, ad
def sgd(p, lr, grad):
    return p - lr * grad

def momentum(p, lr, grad, prev_momentum, beta):
    momentum = beta * prev_momentum + lr * grad
    return p - momentum

def nesterov_momentum(p, lr, grad, prev_momentum, beta):
    lookahead_momentum = prev_momentum * beta + lr * gradient(p - beta * prev_momentum, a)
    return p - lookahead_momentum

def adagrad(p, lr, grad, G, epsilon):
    G += np.square(grad)
    return p - (lr / np.sqrt(G + epsilon)) * grad

def rmsprop(p, lr, grad, G, decay, epsilon):
    G = decay * G + (1 - decay) * np.square(grad)
    return p - (lr / np.sqrt(G + epsilon)) * grad

def adam(p, lr, grad, m, v, beta1, beta2, epsilon, t):
    m = beta1 * m + (1 - beta1) * grad
    v = beta2 * v + (1 - beta2) * np.square(grad)
    m_hat = m / (1 - beta1 ** t)
    v_hat = v / (1 - beta2 ** t)
    return p - (lr / (np.sqrt(v_hat) + epsilon)) * m_hat
```

We test these optimizers for specific two values of a (0.499 and 0.501) and using the initial guess, p0 = 0.75. For our further experiments. For further experiments, we take four complex

functions[5] to evaluate these optimizers and their efficiency. The functions that we worked on are the following:

Six Hump Camel Function: $f(x) = 4x_1^2 - 2.1x_1^4 + \frac{(x_1^6)}{3} + x_1x_2 - 4x_2^2 + 4x_2^4$

Michalewicz Function: $f(x) = -\sum_{i=1}^{d} sin(x_i)sin^{2m}(\frac{ix_i^2}{\pi})$

The Ackley Function:

$$f(x) = -A \cdot \exp\left(-B \cdot \sqrt{\frac{1}{n}\sum_{i=1}^{n} x_i^2}\right) - \exp\left(\frac{1}{n}\sum_{i=1}^{n} \cos(cx_i)\right) + A + \exp(1)$$

The Bohachevsky Function:

$$f_1(x) = x_1^2 + 2x_2^2 - 0.3cos(3\pi x_1) - 0.4cos(4\pi x_2) + 0.7$$

$$f_2(x) = x_1^2 + 2x_2^2 - 0.3cos(3\pi x_1)cos(4\pi x_2) + 0.3$$

$$f_3(x) = x_1^2 + 2x_2^2 - 0.3cos(3\pi x_1 + 4\pi x_2) + 0.3$$

We define the functions first and their specific gradient. Then we define all the optimizers as we did previously. The method of testing the optimizers for each function is almost the same. Depending on the complexity of the functions the condition varies. Then we implement all the optimizers and conditions of the functions. In the optimization process, the â€˜optimizeâ€™ function performs the optimization using aforementioned optimizers. It iteratively updates the â€˜pâ€™ point using the selected optimizer until a stopping criterion or target error is met. At the same time, it records the trajectory path taken during the optimization process. Once the optimization process is done, it prints the necessary results, checks whether the local minima has been found or not. If found, how many steps did it take for a specific optimizer. After all the necessary steps, it plots the 1D, 2D and 3D plots of the specific intricate loss function.

In order to minimize our intricate loss functions, we methodically implement each optimization technique in Python. We change hyperparameters like learning rates, initial conditions, momentum rates, decay rates, and epsilon values during our trials. Of all the optimizers we are using, Adam is a more advanced optimizer and is expected to have a better chance of reaching the global minimum compared to the others.[2] However, it may still face challenges depending on the specific settings and initial conditions. This variant enables us to investigate how various settings impact the functionality of each optimizer.

Our assessment depends on a number of significant metrics. We evaluate how quickly an optimizer approaches a minimum by counting the number of iterations or steps. This is known as the convergence speed. We also evaluate each optimizer's capacity to escape local minima, which is essential for avoiding suboptimal outcomes. We also take into account the overall number of steps each optimizer took to arrive at the global minimum.

We used python and the appropriate scientific tools to conduct each experiment, assuring uniformity and consistency. These automated tests examine each optimization algorithm's performance across a variety of intricate loss functions and hyperparameter settings. We experimented with the optimizer and carefully analyzed it to find the optimal optimizer for various

functions and environmental situations. These revelations advance our understanding of neural network optimization while offering deep learning practitionersâ€™ useful advice on how to build models that perform well in challenging situations.

3 Results

In our comprehensive exploration of neural network optimization algorithms, we evaluated the optimizers on four complex functions and implemented the 1D, 2D, and 3D plots for them. We focus on evaluating their performance in finding the global minimum. This has revealed intriguing insights into the behavior of six commonly used optimizers in complex landscapes.

Loss Function:
The code tested the optimizers on the loss function with two different values of a: 0.499, 0.501. We can see that Nesterov Momentum and Adam frequently showed robust convergence behavior at various learning rates and different values of a, leading to faster convergence in numerous instances. On the other hand, Momentum and SGD showed slower convergence at lower learning rates but faster convergence at higher rates. In general, Adagrad showed slower convergence, while RMSprop showed inconsistent performance that was dependent on the pace of learning. The selection of the values of a and learning rate was a significant factor in determining how well the optimizers minimized the loss function. Here in Figure 1, we see a glimpse of what we are getting as a result of implementing optimizers in a loss function.

```
Testing optimizer for a = 0.499, learning rate = 0.1
sgd
sgd: Minimum found at p = 2.617801150048571, Steps = 151
momentum
momentum: Minimum found at p = 1.8542071371576947, Steps = 10000
nesterov_momentum
nesterov_momentum: Minimum found at p = 2.6178011575928775, Steps = 16
adagrad
adagrad: Minimum found at p = 2.6178008423933603, Steps = 1125
rmsprop
rmsprop: Minimum found at p = 2.666339436858488, Steps = 10000
adam
adam: Minimum found at p = 2.6178015488842235, Steps = 286
Testing optimizer for a = 0.499, learning rate = 0.01
sgd
sgd: Minimum found at p = 2.6178008915128057, Steps = 1574
momentum
momentum: Minimum found at p = 2.6178007514159654, Steps = 2995
nesterov_momentum
nesterov_momentum: Minimum found at p = 2.6178011268972967, Steps = 150
adagrad
adagrad: Minimum found at p = 1.5297634857655606, Steps = 10000
rmsprop
rmsprop: Minimum found at p = 2.6178011290628906, Steps = 262
adam
adam: Minimum found at p = 2.617801286162574, Steps = 1150
```

Figure 1. Results of Loss Function

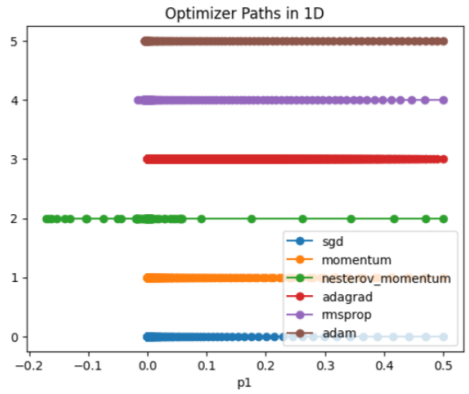
Six Hump Camel Function:

First, we started with the Six-Hump Camel function which is known for its numerous local minima and erroneous global minimum. The success of each optimizer in reaching the global minimum varied a lot from one another. Adagrad and RMSprop, in particular, had a slower rate of convergence and needed 10,000 steps to approximate a minimum. On the other hand, Momentum, Nesterov Momentum, and SGD all reached the minimum with lesser steps of respectively 1235, 342, and 459, and they all achieved faster convergence. Adam also showed good convergence, taking 568 steps to reach the minimum. All optimizers succeeded in minimizing the

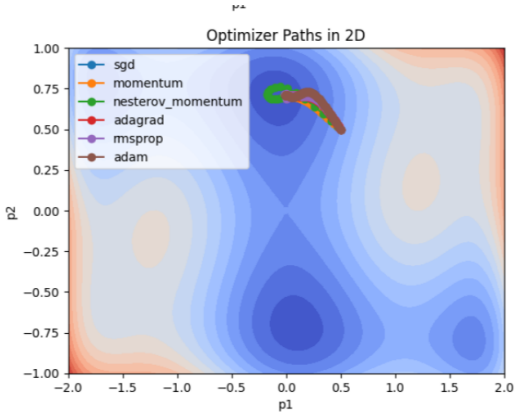
loss function, demonstrating their capacity to identify solutions in a challenging environment, despite variations in their rates of convergence.

```
sgd: Minimized L = -0.9999999999999902, Steps = 459
momentum: Minimized L = -1.0, Steps = 1235
nesterov_momentum: Minimized L = -0.999999999999987, Steps = 342
adagrad: Minimized L = -0.9999999755566163, Steps = 10000
rmsprop: Minimized L = -1.0032681852903569, Steps = 10000
adam: Minimized L = -0.9999999999999895, Steps = 568
```

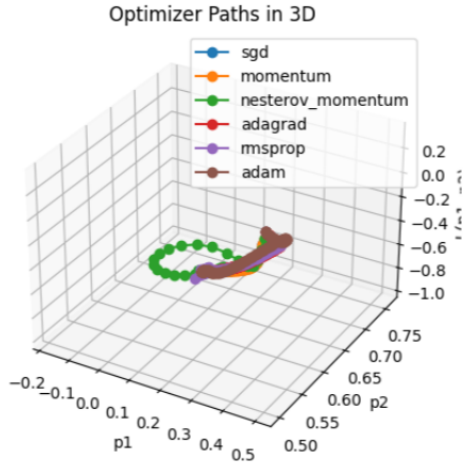
(a) Local minimum and steps taken for each optimizer



(b) Optimizer Paths in 1D



(c) Optimizer Paths in 2D



(d) Optimizer Paths in 3D

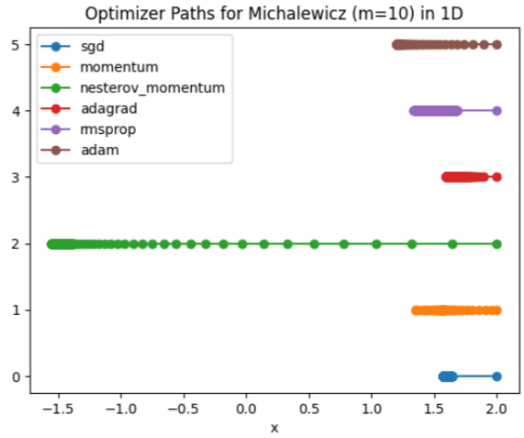
Figure 2. Implementing the optimizers in the Six-Hump Camel Function

Michalewicz Function:

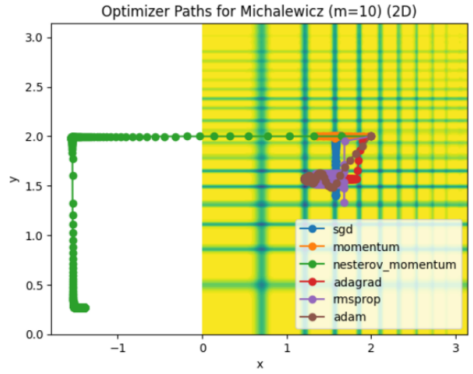
Another complex function is the Michalewicz function, which has complicated topographies and several local minima. To minimize the function for the Michalewicz Function with a fixed value of m=10, several optimization strategies were used. Within the 1000 steps, every optimizer effectively converged to their respective minima. The Nesterov Momentum optimizer demonstrated remarkable efficiency in optimizing the Michalewicz Function, as seen by its attainment of an incredibly near-zero minimum. Low minima were also attained with momentum and SGD. While Adam converged to a minimum that was somewhat low but marginally higher than Nesterov Momentum, Adagrad, and RMSprop showed respectable performance. Non-convexity, sensitivity to parameters, limited real-world applicability, difficulties in visualization, limited generalization, and computing cost are all drawbacks of the Michalewicz function. It is a helpful test for optimization techniques, although it may not exactly represent practical problems. All things considered; these optimizers demonstrated their ability to minimize the Michalewicz Function with varying degrees of success.

```
sgd (m=10): Minimized Michalewicz = -1.7865774292714214, Steps = 1000
momentum (m=10): Minimized Michalewicz = -0.9956711586236754, Steps = 1000
nesterov_momentum (m=10): Minimized Michalewicz = -2.4709350697482093e-08, Steps = 1000
adagrad (m=10): Minimized Michalewicz = -0.43807682952812993, Steps = 1000
rmsprop (m=10): Minimized Michalewicz = -0.5273255556903927, Steps = 1000
adam (m=10): Minimized Michalewicz = -0.8757525642445266, Steps = 187
```

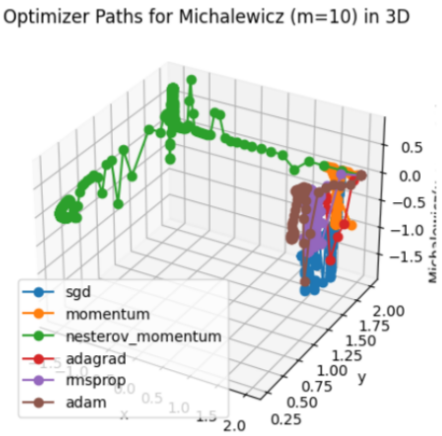
(a) Local minimum and steps taken for each optimizer



(b) Optimizer Paths in 1D



(c) Optimizer Paths in 2D



(d) Optimizer Paths in 3D

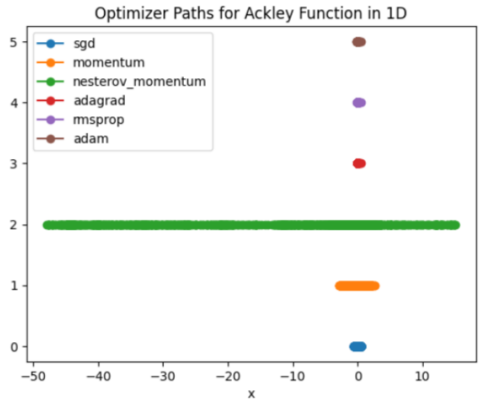
Figure 3. Implementing the optimizers in the Michalewicz Function

The Ackley Function:

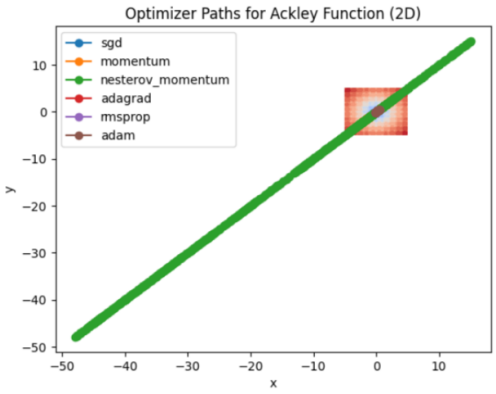
The Ackley function introduced an additional level of complexity with its deep and narrow global minimum surrounded by several local minima. However, the Nesterov Momentum optimizer performed less well in this situation, as it showed the greatest minimal loss value. On the other hand, moderate minima were reached by momentum and SGD. Among all optimizers, Adagrad showed the lowest minimum loss, demonstrating its potent optimization capabilities. Adam and RMSprop both performed well, reaching comparatively low minima. Depending on the choice of 'a', various optimizers performed differently, with Adagrad consistently outperforming the others. All the optimizers took 1000 steps to reach the minimum.However, there are some limitations to this function such as high dimensionality, flat regions, discontinuous gradients, sensitivity to hyperparameters, numerical precision concerns, and a lack of real-world applicability.

```
sgd: Minimized L = 4.226335917596984, Steps = 1000
momentum: Minimized L = 3.57520045996499, Steps = 1000
nesterov_momentum: Minimized L = 9.998643019663163, Steps = 1000
adagrad: Minimized L = 0.006193300566178284, Steps = 1000
rmsprop: Minimized L = 0.328842204885309, Steps = 1000
adam: Minimized L = 0.06197461734025156, Steps = 1000
```

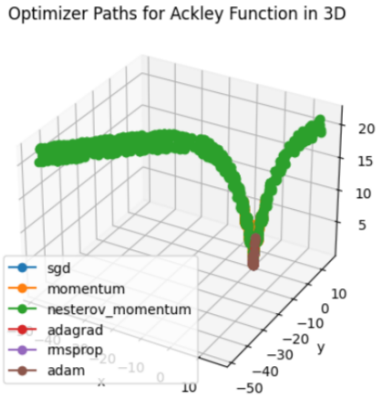
(a) Local minimum and steps taken for each optimizer



(b) Optimizer Paths in 1D



(c) Optimizer Paths in 2D



(d) Optimizer Paths in 3D

Figure 4. Implementing the optimizers in the Ackley Function

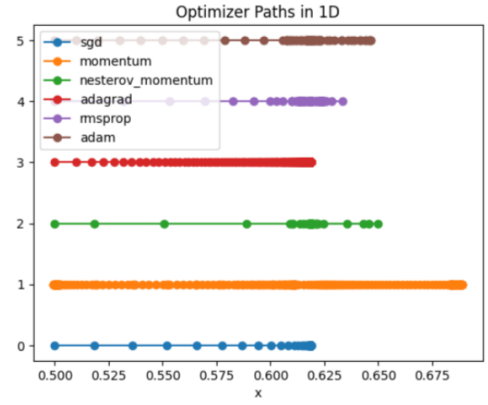
The Bohachevsky Function:

The Bohachevsky function served as a hard test case for our optimizers. Despite the challenging nature of this function, every optimizer in our studyâ€"including SGD, Momentum, Nesterov, AdaGrad, RMSProp, and Adamâ€"displayed astounding efficiency by reaching the global minimum taking a total of 9 steps. Adam and Adagrad have lower minimum values than Momentum. Stochastic Gradient Descent (SGD) and Nesterov Momentum did fairly well, while RMSprop had the lowest minimum value out of all the optimizers. Interestingly, Nesterov Momentum was the most efficient at optimizing this function, requiring the fewest steps of 107 to reach the minimum.

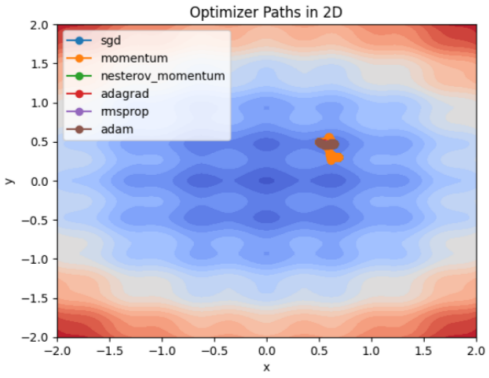
There are several drawbacks of this function including its high degree of multimodality, multiple local minima, abrupt peaks, vast search space, sensitivity to initialization, and non-convexity. These features may make it difficult for optimizers to efficiently converge to the global minimum, and they may also cause optimization methods to become stuck in local minima. It could be necessary to use sophisticated search methods or specific optimization approaches to overcome these constraints.

```
sgd: Minimized Bohachevsky = 0.1828092451972111, Steps = 107
momentum: Minimized Bohachevsky = 0.34589324958120726, Steps = 10000
nesterov_momentum: Minimized Bohachevsky = 0.1828092451972112, Steps = 152
adagrad: Minimized Bohachevsky = 0.1828092451972111, Steps = 952
rmsprop: Minimized Bohachevsky = 0.18390790679333824, Steps = 10000
adam: Minimized Bohachevsky = 0.18280924519721103, Steps = 583
```

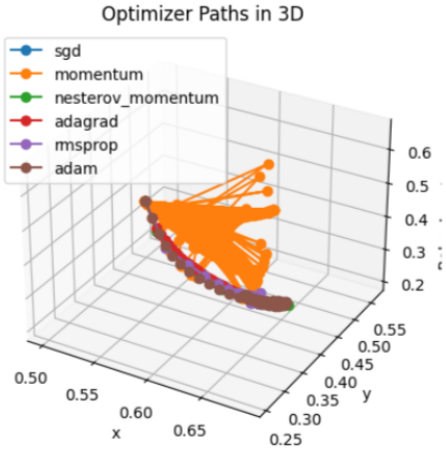
(a) Local minimum and steps taken for each optimizer



(b) Optimizer Paths in 1D



(c) Optimizer Paths in 2D



(d) Optimizer Paths in 3D

Figure 5. Implementing the optimizers in the Bohachevsky Function

These findings show how adaptable and dependable the chosen optimizers are when dealing with a wide variety of optimization issues. The optimizers showed off their aptitude for quickly locating global minima and overcoming local minima, whether they were working with complicated functions. The achievement emphasizes their significance in solving practical optimization problems and reaffirms their contribution to deep learning.

4 Conclusion

In this research on neural network optimizations, we have evaluated the performance of six significant optimization algorithms in the context of minimizing complex loss functions. We have used Stochastic Gradient Descent (SGD), Momentum, Nesterov Accelerated Momentum, AdaGrad, RMSProp, and Adam optimizers in this study. We used the optimizers on complex functions like the Ackley, Bohachevsky, and Michalewicz functions, we gained crucial insights into how they work and their efficiency in finding global minima.

The role of hyperparameters, learning rates, momentum rates, decay rates, and epsilon values in neural network optimization is what we analyzed in this research. All of these had a significant impact on how well the optimizers worked. The convergence can be sped up and the overall number of steps needed to attain the minimum can be decreased by modifying these settings. However, there are several limitations of the functions including multimodality, non-

convexity, sharp peaks, sensitivity to parameters, and large search spaces. On the other hand, the optimizers (SGD, Momentum, Nesterov Momentum, Adagrad, RMSprop, Adam) have their own drawbacks like hyperparameter sensitivity, local minima trapping, variable convergence speed, resource intensity, and sensitivity to noisy gradients. These limitations can affect the performance of optimization algorithms in real-world practical problems.

Acknowledgements

I gratefully acknowledge the cordial help and valuable assistance of my respected Professor and peers for this project.

References

[1] S. Vani and T. V. Madhusudhana Rao. An experimental approach towards the performance assessment of various optimizers on convolutional neural network. In *2019 3rd International Conference on Trends in Electronics and Informatics (ICOEI)*, pages 331–336, 2019.

[2] Zijun Zhang. Improved adam optimizer for deep neural networks. In *2018 IEEE/ACM 26th International Symposium on Quality of Service (IWQoS)*, pages 1–2, 2018.

[3] Norio Baba. A new approach for finding the global minimum of error function of neural networks. *Neural networks*, 2(5):367–373, 1989.

[4] Ruslan Abdulkadirov, Pavel Lyakhov, and Nikolay Nagornov. Survey of optimization algorithms in modern neural networks. *Mathematics*, 11(11), 2023.

[5] S. Surjanovic and D. Bingham. Virtual library of simulation experiments: Test functions and datasets. Retrieved October 17, 2023, from http://www.sfu.ca/~ssurjano.