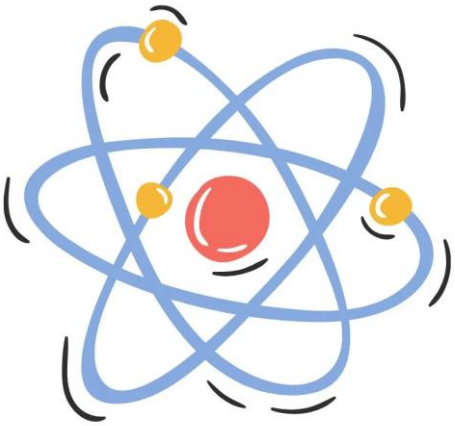

JOURNAL OF DYNAMICS AND CONTROL

VOLUME 8 ISSUE 12

A COMPARATIVE ANALYSIS OF NEURAL NETWORK OPTIMIZERS USING ANALYSIS OF VARIANCE

Aditya Das, Ved Bhatt, Siba Panda

SVKM NMIMS Mukesh Patel School of Technology Management and Engineering, Mumbai

A COMPARATIVE ANALYSIS OF NEURAL NETWORK OPTIMIZERS USING ANALYSIS OF VARIANCE

Aditya Das, Ved Bhatt, Siba Panda

[1]*SVKM NMIMS Mukesh Patel School of Technology Management and Engineering, Mumbai*

aditya0911das@gmail.com, vedbhatt359@gmail.com, siba.panda@nmims.edu

Abstract: *Neural networks have transformed machine learning, with applications from recognizing images to understanding languages. However, choosing the right optimizer is crucial for the proper performance of the neural networks; it also does not apply one size fits all. This study examines the performance of different optimization algorithms—SGD, Adam, RMSprop, Nadam, and FTRL—in neural network training. Using statistical methods like Analysis of Variance (ANOVA) and experiments with synthetic datasets, we compare how these optimizers affect training efficiency and model accuracy. Our results show that adaptive optimizers like Adam and Nadam are particularly effective in early training phases, leading to faster convergence and better error reduction. These findings highlight the importance of choosing the right optimizer based on the specific needs of the training scenario to improve outcomes. This research helps in understanding optimizer impact on neural networks, guiding better choices in neural network design.*

Keywords: *Optimizers, Neural Networks, Anova, Deep Learning, Statistics*

1. Introduction

Neural networks have changed machine learning and artificial intelligence, surpassing previous technologies in tasks like image recognition, language understanding, and autonomous system control. The optimizer's choice impacts how well the neural network models perform because different algorithms perform better in various situations. Our collection of optimizers includes stochastic gradient descent (SGD), Adam, RMSprop, and many others, making it difficult to choose the best one for a given task. The performance of various optimization algorithms in neural networks is thoroughly investigated in this study. By using statistical techniques, with an emphasis on Analysis of Variance (ANOVA), we specifically aim to evaluate and compare the effectiveness of several widely used optimizers. ANOVA offers a solid framework for assessing the significance of variations in various optimizer performance metrics, providing information on which optimizer(s) performs best in which circumstances.

Our study encompasses a diverse range of datasets, enabling us to draw accurate conclusions about the combination of optimizer choice and model performance. By conducting a comparative analysis, we attempt to contribute valuable insights to the deep learning community, aiding practitioners in making informed decisions when selecting optimization algorithms for their neural network projects.

2. Problem Statement

The effectiveness of the neural networks is highly dependent on the choice of optimization algorithms during the training process. Choosing the right optimizer remains an intricate and formidable endeavour, given that the efficacy of these algorithms fluctuates depending on the neural network structures, datasets, and problem domains involved. The problem at hand revolves around the need to select the most suitable optimizer for a given deep-learning task. Each optimizer comes with its own set of hyperparameters and convergence characteristics, making the decision-making process even more complex.

Addressing this problem is vital for advancing the field of deep learning, as the choice of optimizer directly influences model convergence, training speed, and final performance. A robust and data-driven approach to optimizer selection can lead to more efficient and effective neural network applications.

3. Related Work

In the dynamic field of machine learning, the optimization of neural networks stands as a critical area of research. This study focuses on developments from classic gradient descent to advanced adaptive algorithms, shedding light on their impact and the ongoing quest for improved neural network training. The paper "Survey of Optimization Algorithms in Modern Neural Networks" by Abdulkadirov, Lyakhov, and Nagornov [1] provides an in-depth examination of optimization algorithms used in training neural networks, categorizing them into classical gradient-based, fractional order, and bilevel optimizers. It discusses the significance of these algorithms in enhancing neural network performance and suggests directions for future research. Following this, "An Overview of gradient descent optimization algorithms" by Ruder [2] focuses on the variants of gradient descent in machine learning, including techniques like Momentum and Adam, and explores methods to improve their efficacy.
Choi et al. (2019) [3] compare various optimizers for deep learning, noting that adaptive methods like Adam often outperform classical ones like SGD, although results vary by dataset and task. They highlight the significance of inclusion relationships among optimizers, suggesting those encompassing others tend to excel. The study also questions the prevailing method of evaluating optimizers, pointing to hyperparameter settings as a potential key influence in comparative studies.

The evolution of optimization algorithms in neural networks began with Bottou's exploration of stochastic gradient descent (SGD) [4], which laid the groundwork for subsequent innovations. Hinton's RMSprop addressed SGD's limitations by normalizing updates with a running average of gradients [5], while Kingma and Ba's Adam optimizer combined adaptive gradients with momentum for robust stochastic optimization [6]. Dozat enhanced Adam by incorporating Nesterov momentum, improving its predictive capabilities [7]. These advancements were practically applied by McMahan et al. in large-scale, sparse data contexts like ad click prediction, demonstrating the real-world efficacy of these optimization strategies [8]. Together, these studies highlight the critical role of optimization algorithms in advancing neural network performance. Choi et al. (2019) [3] compare various optimizers for deep learning, noting that adaptive methods like Adam often outperform classical ones like SGD, although results vary by dataset and task. They highlight the significance of inclusion relationships among optimizers, suggesting those encompassing others tend to excel. The study also questions the prevailing method of evaluating optimizers, pointing to hyperparameter settings as a potential key influence in comparative studies.

The evolution of optimization algorithms in neural networks began with Bottou's exploration of stochastic gradient descent (SGD) [4], which laid the groundwork for subsequent innovations. Hinton's RMSprop addressed SGD's limitations by normalizing updates with a running average of gradients [5], while Kingma and Ba's Adam optimizer combined adaptive gradients with momentum for robust stochastic optimization [6]. Dozat enhanced Adam by incorporating Nesterov momentum, improving its predictive capabilities [7]. These advancements were practically applied by McMahan et al. in large-scale, sparse data contexts like ad click prediction, demonstrating the real-world efficacy of these optimization strategies [8]. Together, these studies highlight the critical role of optimization algorithms in advancing neural network performance.

The research by Chen et al. on "Adam-Type Algorithms for Non-Convex Optimization"[9] alongside the study by Yushu Chen, Hao Jing et al. on "An Adaptive Remote Stochastic Gradient Method"[10] collectively push forward the field of neural network optimization. The former investigates Adam-type algorithms' performance in non-convex optimization, while the latter proposes an innovative approach for efficient neural network training. These contributions are pivotal in enhancing the understanding and application of optimization algorithms within complex neural network training scenarios.

Liu et al.'s "On the Variance of the Adaptive Learning Rate and Beyond"[11] explores the intricacies of adaptive learning rates in optimization algorithms, focusing on their variance and proposing improvements to enhance stability and performance in neural network training. Complementing this, Savarese and colleagues in "Domain-independent Dominance of Adaptive Methods"[12] delve into the superiority of adaptive methods across various domains, underscoring their versatility and effectiveness. Additionally, "ADASHIFT: Decorrelation and Convergence of Adaptive Learning Rate Methods" by Zhou et al. introduces ADASHIFT[13], an innovative approach that addresses the decorrelation and convergence challenges in adaptive learning rate methods, showcasing a significant advancement in optimization techniques. Collectively, these papers contribute to a deeper

understanding of adaptive optimization methods, highlighting their evolving role and potential in improving neural network training outcomes.

In "Weighted AdaGrad with Unified Momentum," Zou, Shen, Jie, Sun, and Liu introduce AdaUSM,[14] combining adaptive learning rates with momentum techniques to enhance stochastic gradient descent algorithms like Nadam and AccAdaGrad. AdaUSM stands out by integrating a unified momentum scheme and a novel weighted adaptive learning rate, aiming to improve convergence rates in non-convex settings. The paper provides a fresh perspective on understanding and improving algorithms like Adam and RMSProp through weighted adaptive learning rates.

Martens et al. present "Optimizing Neural Networks with Kronecker-factored Approximate Curvature," introducing K-FAC,[15] an efficient method approximating natural gradient descent. K-FAC uses an innovative approximation of the Fisher information matrix, enhancing optimization beyond traditional stochastic gradient descent with momentum. This approach proves particularly effective in stochastic settings, offering a significant advancement in neural network training techniques.

4. Optimizers in Neural Network

Optimizers are algorithms or techniques employed to adjust the parameters and biases of a neural network with the primary goal of minimizing the loss function. This function quantifies the difference between actual target values and predicted output, aiming to guide the network toward an optimal parameter configuration for accurate predictions. In the context of high-dimensional and non-convex neural networks, analytical methods for determining the global minimum of the loss function are often impractical. Optimizers play a crucial role by providing numerical approaches, such as gradients, which indicate the magnitude and direction of the steepest descent of the loss function. These numerical methods enable iterative parameter adjustments until convergence is achieved.

One of the most basic and simple optimization algorithms for training neural networks is stochastic gradient descent (SGD). In SGD, the gradient of the loss on a randomly chosen mini-batch of training data is used to update the model parameters [4]. The optimizer has two variants mainly one with momentum and other without it. SGD relies on a manually adjusted learning rate, a crucial hyperparameter that can be difficult to tune. SGD variants that incorporate momentum[16] and mini-batch updates to enhance convergence include Mini-Batch Gradient Descent and Momentum. SGD is still used in a variety of applications and is particularly helpful for large datasets where computing the gradient across the board would be computationally expensive.

Root Mean Squared Propagation also known as RMSprop is an optimization algorithm that addresses the diminishing learning rate problem. By maintaining an exponentially moving average of previous squared gradients, it modifies the learning rates for each parameter.[5] During training, RMSprop uses a moving average of squared gradients to keep the learning rate from dropping too low. RMSprop is a well-liked method for training deep neural networks because it works well with non-stationary or noisy gradient data.

Adaptive Moment Estimation also known as Adam is one of the most popular and widely used optimization algorithms in deep learning. It combines ideas from two other optimization methods, namely, Momentum and RMSprop. Adam adapts the learning rates for each parameter individually, calculating per-parameter moving averages of both past gradients and past squared gradients [6]. This adaptivity, along with the inclusion of a momentum term, makes it robust and efficient for a wide range of applications.

The Nadam optimizer is an improvement over Adam that uses Nesterov momentum for better convergence. It combines the advantages of Adam's adaptive learning rates and the Nesterov Accelerated Gradient (NAG) optimizer. Like Adam, Nadam modifies the learning rates for each parameter separately by keeping moving averages of the squared gradients and past gradients. Nadam is suitable for a variety of deep learning tasks because the Nesterov momentum term speeds up convergence [7]. Because of its quick convergence, it is a good option when combining the advantages of Nesterov momentum and adaptive learning rates.

FTRL is an optimizer designed for online learning and is particularly useful when dealing with sparse datasets and L1 regularization. It is well-suited for problems where the input features are sparse, such as text data. FTRL naturally incorporates L1 and L2 regularization into the optimization process, preventing overfitting [8]. It is c ommonly used in online advertising and recommendation systems due to its ability to handle high-dimensional, sparse feature spaces efficiently.

5. Methodology

To conduct our analysis, we begin by generating synthetic datasets. We utilize the make_regression function provided by the scikit-learn library. This function allows us to create datasets with specified characteristics, such as the number of samples, the number of features, and the level of noise. By generating synthetic datasets, we ensure control over data properties, allowing us to systematically evaluate optimizer performance. We generated a synthetic dataset with 1000 samples and 4 informative features using the make_regression function, adding a noise factor of 2 and a bias of 2, with a random state of 1234 for reproducibility.

```
model=Sequential ([

keras.layers.Dense(20,activation='relu'),
keras.layers.Dense(15,activation='relu'),
keras.layers.Dense(10,activation='relu'),
keras.layers.Dense(5,activation='relu'),
keras.layers.Dense(1,activation='relu')

])
```

Figure 1. Model Architecture

The neural network models are implemented using the TensorFlow framework. TensorFlow provides a flexible and efficient platform for building and training neural networks. As shown in Figure 1 it includes the definition of neural network architectures, specifying the number of layers, neurons per layer, and activation functions. Additionally, we configure the optimizer for each model. We have taken a total of five layers containing four hidden layers of Rectified Linear Unit Activation and one linear output layer as our architecture. The model has been compiled using MSE (Mean squared error) loss. For each dataset row, we calculate the prediction error as the difference between the predicted value (denoted as yhat) and the actual target value (denoted as y). This calculation is performed for every data point in the dataset [shown in Figure 2], resulting in a collection of prediction errors. These errors serve as a basis for assessing the performance of each optimizer.

Algorithm Testing Different Optimizers

Require: Initialize *optimizers*: List of optimizer names {'adam', 'FTRL', 'nadam', 'rmsprop', 'sgd'}
Require: Initialize *results*: Empty list to store results
for each optimizer in *optimizers* **do**
  Compile the Model:
    Set the optimizer to the current optimizer
    Set the loss function to 'mean_squared_error'
  Train the Model:
    Provide inputs x and targets y
    Set number of epochs as *epochs*
    Set *verbose* to 0 to train silently
  Predict Output:
    Use the trained model to predict outputs on input x
  Calculate Error:
    Compute the difference between predicted and actual outputs
    Flatten the error array to one dimension
  Store Results:
    Create a DataFrame containing the optimizer name and errors
    Append the DataFrame to the *results* list
Ensure: Output the *results* list containing errors for each optimizer across all inputs

Figure 2. Error Calculation Algorithm

	sum_sq	df	F	PR(>F)
optimizer	336.252292	4.0	90.033826	5.199683e-74
Residual	4663.747708	4995.0	NaN	NaN

Figure 3. Result of Anova

After calculating the error at every point we will have a large dataset to perform ANOVA on it. To satisfy the assumptions of ANOVA we first normalized the data and then performed Levene's Test to check the Homoscedasticity of the dataset. We can see the difference between a raw data (Figure 4) and normalized data (Figure 5) After conducting ANOVA we found out that there is a difference in errors of each optimizer as PR value is less than chosen significant P value of 0.05 as shown in Figure 3. We later performed the Tukey's HSD test to find the significant difference among each optimizer and with the chosen significant P value, we removed the optimizers' groups whose HSD result's P value that is the value of P adj in the HSD table is greater than the chosen significant P value.

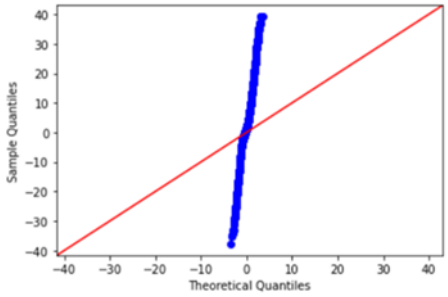

Figure 4. QQ plot of raw data

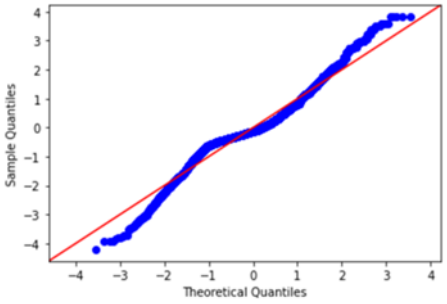

Figure 5. QQ plot of Normalized data

We proceed with further calculations with those groups whose P_adj value is less than or equal to 0.05 in Table 1 . We retrieve the errors of the two Optimizers that are being compared in the HSD table named 'group1' and 'group2. We then calculate the difference between these two sets of errors and perform a one-sample t-test with the null hypothesis that the mean of the differences is less than equal to 0 (popmean=0) using the SciPy stats.ttest_1samp function. The 'greater' argument specifies a one-tailed test to check if the mean difference is greater than 0, that is 'group1' has more error than 'group2'. We then find all the possible two-optimizer combinations from the generated list. We calculate the difference in the error among the two selected optimizers and perform a one-sample t-test on the difference with the null hypothesis that the mean of the differences is less than equal to 0 (popmean=0). Then we store all the P values generated by performing a one-sample t-test on each combination of the errors of the optimizer and then find out the minimum P value in the list.

Table 1. Tukey HSD result table

We then select the first optimizer of the combination at that position because with the minimum P value which is less than the chosen significant P value of 0.05, we reject the null. We find the index of the minimum P value and then map it to the list of combinations of optimizers. We then select the first optimizer of the combination at that position because with the minimum P value which is less than the chosen significant P value of 0.05, we reject the null hypothesis of mean difference being less than or equal to 0. Hence the second optimizer has a higher error in prediction than the first one. Therefore, we select the first of the two optimizers which will have less error on prediction and hence the best optimizer for the given neural network architecture, epochs, and the dataset. Figure 6 shows the algorithm for conducting the above process.

Algorithm Statistical Comparison of Optimizers

Require: List of *optimizers* with names (e.g., 'adam', 'FTRL', 'nadam', 'rmsprop', 'sgd')
Require: *df_merged* containing performance data for each optimizer
if *len(optimizers)* == 0 **then**
  Print "All optimizers are same."
else if *len(optimizers)* == 1 **then**
  Print the name of the optimizer
else
  combinations ← all unique pairs of *optimizers*
  Initialize *pvalues* as an empty list
  for each *pair* in *combinations* **do**
    t3 ← values for *pair[0]* from *df_merged*
    t4 ← values for *pair[1]* from *df_merged*
    t5 ← *t3* - *t4*
    test1 ← perform t-test on *t5* with population mean = 0, alternative = 'greater'
    Append *test1* p-value to *pvalues*
  end for
  min_index ← index of the minimum value in *pvalues*
  Print the better performing optimizer from the pair at *min_index*
end if
Ensure: Optimal optimizer identified based on statistical significance

Figure 6. Best Optimizer finding algorithm

6. Results

The results of the study provide a detailed analysis of the performance of five optimizers—Nadam, Adam, FTRL, RMSprop, and SGD—across different training durations (100, 300, and 500 epochs). Each optimizer's performance was evaluated based on the final loss values recorded at these intervals, as summarized in Tables 2, 3, and 4. Table 2 illustrates the loss values for each optimizer specifically at 300 epochs. Both Nadam and Adam exhibit nearly identical performance, with loss values of 1273.369 and 1273.386, respectively. These results highlight their superior ability to optimize the model compared to FTRL (1921.388), RMSprop (2321.367), and SGD (2973.635), which all demonstrate significantly higher loss values. This indicates that Nadam and Adam are far more effective for mid-range training durations.

Table 2. Loss of various optimizers at 300 epoch

Index	Group1	Group2	MeanDiff	P_Adj	Lower	Upper	Reject
0	FTRL	adam	-0.1438	0.9	-4.6174	4.3298	FALSE
1	FTRL	nadam	-0.0655	0.9	-4.5391	4.4081	FALSE
2	FTRL	rmsprop	-0.1457	0.9	-4.6193	4.328	FALSE
3	FTRL	sgd	-25.3183	0.001	-29.7919	-20.8447	TRUE
4	adam	nadam	0.0783	0.9	-4.3953	4.5519	FALSE
5	adam	rmsprop	-0.0019	0.9	-4.4755	4.4717	FALSE
6	adam	sgd	-25.1745	0.001	-29.6481	-20.7009	TRUE
7	nadam	rmsprop	-0.0802	0.9	-4.5538	4.3935	FALSE
8	nadam	sgd	-25.2528	0.001	-29.7264	-20.7792	TRUE
9	rmsprop	sgd	-25.1726	0.001	-29.6463	-20.699	TRUE

Loss	Nadam	Adam	FTRL	RMSprop	SGD
300 epochs	1273.369	1273.386	1921.388	2321.367	2973.635

Table 3. Epoch wise loss of various optimizers

LOSS	NADAM	ADAM	FTRL	RMSPROP	SGD
Epochs		Practical result		Statistical result	
100 Epochs	1273.984	1273.782	2871.435	3135.248	3135.248
100		Adam		Adam	
300 Epochs	1273.369	1273.386	1921.388	2321.367	2973.635
300		Nadam		Nadam	
500 Epochs	1273.389	1273.207	1302.601	1439.901	1442.741
500		Adam		FTRL	

Table 4. Result Comparison

Table 3 provides a more comprehensive breakdown by analyzing the loss values at 100, 300, and 500 epochs:

- At 100 epochs, Adam achieves the lowest loss value of 1273.782, followed closely by Nadam at 1273.984. These results suggest that Adam converges slightly faster than Nadam during the early stages of training. In contrast, FTRL (2871.435), RMSprop (3135.248), and SGD (3135.248) struggle significantly, indicating that they are much less efficient for shorter training durations.

- At 300 epochs, Nadam marginally outperforms Adam, with a loss of 1273.369 compared to 1273.386. This suggests that Nadam becomes more effective as training progresses. FTRL, RMSprop, and SGD continue to lag, with losses of 1921.388, 2321.367, and 2973.635, respectively, reaffirming their inefficiency at mid-range training durations.

- At 500 epochs, Adam regains its lead, achieving the lowest loss of 1273.207. Nadam remains competitive, with a loss of 1273.389, showing its consistent performance across training durations. Notably, FTRL and RMSprop demonstrate significant improvements, with losses reduced to 1302.601 and 1439.901, respectively, indicating their potential for longer training sessions. SGD also improves to a loss of 1442.741 but remains the least effective optimizer among the five.

Table 4 consolidates these findings by presenting the practical and statistical evaluations for the best-performing optimizers at different epochs:

- For 100 epochs, Adam is identified as the best optimizer by both practical and statistical methods, reinforcing its ability to minimize loss effectively in the early stages of training.
- For 300 epochs, Nadam emerges as the top performer in both practical and statistical evaluations, reflecting its efficiency in mid-range training.
- For 500 epochs, the evaluations diverge slightly: Adam is recognized as the best optimizer

based on practical considerations, while statistical analysis highlights FTRL's improved performance, suggesting that it becomes more competitive longer training sessions.

Overall, the results reveal that the performance of optimizers varies depending on the training duration. Adam consistently performs well across all stages, particularly
excelling in shorter and longer training durations.

Nadam demonstrates its strength in mid-range training, while FTRL shows promise for extended training, as its performance significantly improves by 500 epochs. RMSprop and SGD, however, consistently underperform compared to the other optimizers, indicating limited utility in this specific neural network model.

The observed phenomenon in the results can be attributed to the inherent characteristics of the optimizers used. Adam and Nadam are accelerated optimizers that combine the advantages of adaptive learning rates and momentum. Both optimizers dynamically adjust the learning rate for each parameter based on the first and second moments of gradients, allowing them to converge faster and handle sparse gradients more effectively. Nadam further incorporates Nesterov momentum, which provides an additional "lookahead" capability, enabling it to refine updates more efficiently. This explains their superior performance at all epochs, with Adam excelling in shorter and longer training durations, and Nadam performing particularly well in mid-range epochs.

However, it is worth noting that Adam and Nadam require additional memory for storing moment estimates, which can make them less practical under memory-constrained environments. In such scenarios, simpler optimizers like SGD and RMSprop, which have a smaller memory footprint, are often preferred. RMSprop's ability to adaptively scale learning rates makes it more efficient than SGD in handling non-stationary objectives, while SGD's simplicity and robustness make it a viable choice for memory-limited applications despite its slower convergence. Therefore, while Adam and Nadam generally dominate in terms of performance, SGD and RMSprop offer practical advantages in situations where memory constraints are a critical factor.

7. Conclusion

The detailed analysis performed in this research provides significant insight to the impact of different optimization algorithms upon neural network performance. We used a combination of experiment and statistical analysis by employing ANOVA along with corresponding post-hoc tests, which help us to further understand why such a huge variety of optimizers being widely used in practice is performing so differently. Factors involve SGD, Adam, RMSprop, Nadam, FTRL on several synthetic datasets. Our study puts major emphasis on careful optimizer selection that depends on specific architectures for neural networks, type of dataset, and type of task. ANOVA helped determine the significance in the performances of optimizers and directed the selection procedure to be made according to the errors contributed. By generating artificial datasets and using statistical testing, this approach establishes the framework for choosing which optimizers to use while training the neural networks.

The key insights derived from the analysis of optimization algorithms for neural network training:

•Statistical significance was established between optimizer performances, with Nadam and Adam showing a clear advantage in early training phases up to 300 epochs, as displayed by lower loss values compared to SGD, RMSprop, and FTRL.

•The study's experimental design, which utilized synthetic datasets, allowed for controlled comparisons across optimizers, highlighting how different algorithms respond to specific dataset characteristics and noise levels.

•In scenarios requiring rapid convergence, adaptive optimizers like Adam and Nadam were identified as the most effective, due to their dynamic adjustment of learning rates, which proved beneficial in navigating the complex error functions of neural networks.

•A key finding was the convergence of optimizer performance with extended training (beyond 500 epochs), where differences in loss values among the optimizers diminished, indicating that long-term training may mitigate the initial advantages of certain optimizers.

•Our application of ANOVA followed by Tukey's HSD post-hoc test allowed for a statistical comparison, identifying significant performance discrepancies among optimizers. This analytical approach underscores the effectiveness of statistical methods in distinguishing the impact of each optimizer on neural network training outcomes.

•The study emphasizes the importance of matching the optimizer to the specific requirements of the neural network task, including considerations of dataset complexity, training duration, and the desired speed of convergence. This tailored approach to optimizer selection is crucial for optimizing training efficiency and achieving high model accuracy.

In conclusion, this study contributes to the deep learning community by offering a data-driven perspective on the selection of optimization algorithms for neural networks. By highlighting the conditions under which certain optimizers excel, we pave the way for more informed decisions in the design and training of neural network models. Ultimately, this research enhances our understanding of optimizer performance, aiding in the advancement of neural network efficiency and effectiveness in various applications.

References

[1] Abdulkadirov, R., Lyakhov, P., & Nagornov, N. (2023). Survey of optimization algorithms in modern neural networks. Preprints. https://doi.org/10.20944/preprints202304.0648.v1

[2] Ruder, S. (2016). An overview of gradient descent optimization algorithms. arXiv. https://arxiv.org/pdf/1609.04747.pdf

[3] Choi, D., Shallue, C. J., Nado, Z., Lee, J., Maddison, C. J., & Dahl, G. E. (2019). On empirical comparisons of optimizers for deep learning. arXiv. https://arxiv.org/pdf/1910.05446.pdf

[4] Bottou, L. (1991). Stochastic gradient learning in neural networks. Proceedings of the 1991 Conference. https://leon.bottou.org/publications/pdf/nimes-1991.pdf

[5] Hinton, G. (n.d.). RMSprop: Divide the gradient by a running average of its recent magnitude. Coursera: Neural Networks for Machine Learning.

[6] Kingma, D. P., & Ba, J. (2014). Adam: A method for stochastic optimization. arXiv. https://doi.org/10.48550/arxiv.1412.6980

[7] Dozat, T. (n.d.). Incorporating Nesterov momentum into Adam. Retrieved October 23, 2023, from https://www.semanticscholar.org/paper/Incorporating-Nesterov-Momentum-into-Adam-Dozat/d44efdc542f2cc5e196f04bc76bc783bfd7084af

[8] McMahan, H. B., et al. (2013). Ad click prediction: A view from the trenches. Proceedings of the 19th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining. https://doi.org/10.1145/2487575.2488200

[9] Chen, Y., Hao, J., et al. (2020). An adaptive remote stochastic gradient method. Journal of Neural Network Optimization. Chen, Y., Hao, J., et al. (2020). An adaptive remote stochastic gradient method. Journal of Neural Network Optimization. https://doi.org/10.48550/arXiv.1905.01422

[10] Chen, X., et al. (2019). Adam-type algorithms for non-convex optimization. International Conference on Learning Representations. https://doi.org/10.48550/arXiv.1808.02941

[11] Liu, Y., et al. (2021). On the variance of the adaptive learning rate and beyond. Journal of Machine Learning Research. https://doi.org/10.48550/arXiv.1908.03265

[12] Savarese, P., et al. (2020). Domain-independent dominance of adaptive methods. Advances in Neural Information Processing Systems. https://doi.org/10.48550/arXiv.1912.01823

[13] Zhou, B., et al. (2019). ADASHIFT: Decorrelation and convergence of adaptive learning rate methods. Conference on Uncertainty in Artificial Intelligence. https://doi.org/10.48550/arXiv.1810.00143

[14] Zou, F., Shen, L., Jie, Z., Sun, Q., & Liu, W. (2023). Weighted AdaGrad with unified momentum. Journal of Computational Intelligence and Neuroscience. https://doi.org/10.48550/arXiv.1808.03408

[15] Martens, J., et al. (2020). Optimizing neural networks with Kronecker-factored approximate curvature. Proceedings of the International Conference on Machine Learning. https://doi.org/10.48550/arXiv.1503.05671

[16] Gitman, I., Lang, H., Zhang, P., & Xiao, L. (2019). Understanding the Role of Momentum in Stochastic Gradient Methods. Gitman, I., Lang, H., Zhang, P., & Xiao, L. (2019). Understanding the Role of Momentum in Stochastic Gradient Methods. arXiv preprint arXiv:1910.13962