PREPRINT

PERTANIKA JOURNALS

SCIENCE & TECHNOLOGY

Journal homepage: http://www.pertanika.upm.edu.my/

A Comparative Study of Gradient Descent Methods in Deep Learning Using Body Motion Dataset

Zulfikar Sembiring[1,2]*, Khairul Najmy Abdul Rani[2,3] and Amiza Amir[2]

[1]*Faculty of Engineering, Universitas Medan Area,20223 Medan, North Sumatera, Indonesia*
[2]*Faculty of Electronic Engineering and Technology, Universiti Malaysia Perlis, 02600 Arau, Perlis, Malaysia*
[3]*Centre of Excellence, Advanced Communication Engineering, Universiti Malaysia Perlis, 02600 Arau, Perlis, Malaysia*

ABSTRACT

In this study, a Recurrent Neural Network (RNN) architecture model is used to analyse and compare the seven most widely used first-order stochastic gradient-based optimization algorithms. Adaptive Moment Estimation (ADAM), Root Mean Square Propagation (RMSprop), Stochastic Gradient Descent (SGD), Adaptive Gradient (AdaGrad), Adaptive Delta (AdaDelta), Nesterov-accelerated Adaptive Moment Estimation (NADAM), and Maximum Adaptive Moment Estimation (AdaMax) are the optimization techniques that have been studied. The study used the body motion datasets from the University of California-Irvine (UCI) Machine Learning (ML) datasets. This experiment demonstrates the capabilities of various combinations of optimizer models, long short-term memory (LSTM) architecture, activation functions, and learning rate. The main aim is to understand how good each optimizer performs in test accuracy and feasible training time behaviour over various learning rates and activation functions. The outcomes vary by setting, with some achieving higher accuracy and shorter training sessions than others. The AdaGrad model, which uses exponential and sigmoid activation functions with a learning rate of 0.001, has a training time of 17.1 minutes and a test accuracy of 78.31%, making it the top-performing configuration. The exponential function is an activation function that consistently outperforms other models and optimization algorithms. It consistently delivers high accuracy and minimal running time across numerous models and optimizers, while the Softmax activation function continuously underperforms.

Keywords: Accuracy, activation function, body motion datasets, gradient descent (GD), learning rate, long short-term memory (LSTM), Recurrent Neural Network (RNN), running time

ARTICLE INFO

Article history:
Received: 19 November 2024
Accepted: 08 April 2025
Published: 04 July 2025

DOI: https://doi.org/10.47836/pjst.33.4.13

E-mail addresses:
zulfikarsembiring@studentmail.unimap.edu.my (Zulfikar Sembiring)
khairulnajmy@unimap.edu.my (Khairul Najmy Abdul Rani)
amizaamir@unimap.edu.my (Amiza Amir)
*Corresponding author

e-ISSN: 2231-8526

INTRODUCTION

Selecting the optimal optimization algorithm is one of the effective approaches to enhance the learning process in deep learning (DL) algorithm. This work focuses on the application of optimization methods based on the gradient descent (GD) approach to optimize the DL architecture. DL relies heavily on optimization since the model optimizer continually updates and calculates the parameters that affect model training and output. The purpose is to follow the steepest descent direction, which is provided by the negative gradient, in order to approximate or reach the optimal value and optimize the objective function (Mehmood et al., 2023), DL optimization problems are complicated and call for more decomposition. Three stages can be identified in the growth of optimization. The first stage is to get the algorithm start running and have it arrived at a logical conclusion, like a stationary point. Making the algorithm converge as rapidly as possible is the second stage. Making sure the algorithm converges to a result with a low objective value, like a global minimum, is the third stage (Sun, 2019). It is important to remember that getting good test accuracy requires a further step that is outside the purview of optimization. The three categories of convergence, convergence speed, and global quality as in Figure 1 are used to group optimization problems.

Machine learning (ML) is a subset of artificial intelligence dedicated to creating algorithms that allow computers to learn from data and make predictions or judgments autonomously, without explicit programming (Ethem, 2020). The GD is widely employed in ML and other mathematical applications to optimize a cost function. The cost function calculates the discrepancy between a model projected and actual output. The objective of GD is to minimize the cost function by modifying the model's parameters, such as weights

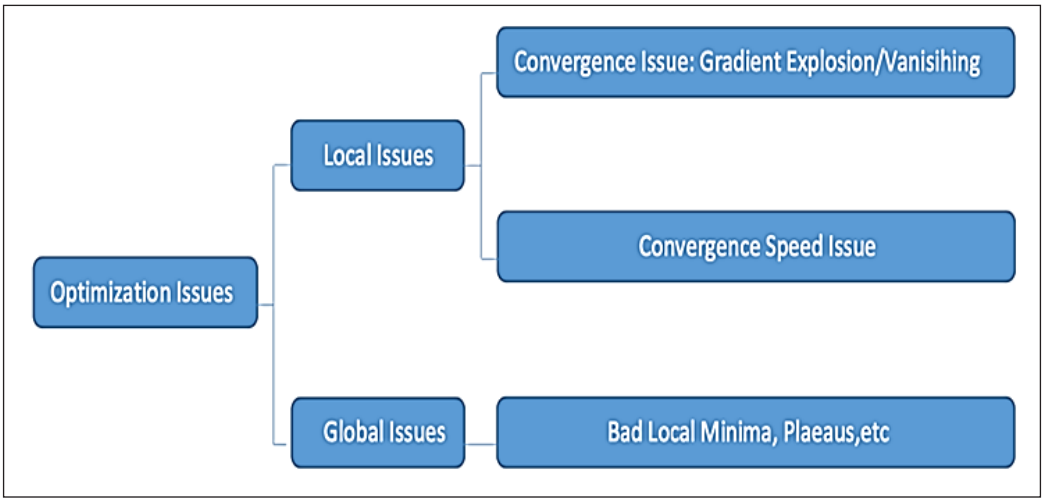

Figure 1. Optimization issues (Sun, 2019)

and biases (Chandra et al., 2022). The concept of GD is based on the idea of finding the steepest descent direction of the cost function and moving the parameters in that direction to reach a minimum value of the cost function (Seeli & Thanammal, 2024). This process involves computing the gradient of the cost function with respect to the parameters and taking a step in the opposite direction of the gradient by repeating this process iteratively, GD can converge to a minimum of the cost function (Haji & Abdulazeez, 2021).

Comparing different GD optimizers is crucial in DL because it allows for the selection of the most effective and efficient optimization algorithm for a specific problem. Each optimizer has its strengths and weaknesses, and understanding these differences can significantly impact the performance of the model. For instance, while Batch GD is simple and easy to implement, it can be slow and computationally expensive for large datasets. SGD, on the other hand, is faster but can be noisy and may not converge well (Schaul et al., 2013). Adaptive optimizers like ADAM and its variants, however, can adapt to the changing nature of the optimization problem, making them more robust and effective (Kingma & Ba, 2015). By comparing these optimizers, researchers and practitioners can identify the best approach for their specific problem, ensuring optimal performance and faster convergence. This comparison is particularly important in modern DL applications, where large datasets and complex models require efficient and effective optimization techniques.

Neural network (NN) is commonly employed to tackle non-convex problems, but choosing an appropriate optimization method can be difficult to locate the global optimum in these networks. This difficulty arises from the need to estimate a vast quantity of parameters within a high-dimensional search space. An ineffective optimization strategy may lead the network to remain trapped in local minimum while training, preventing any improvement (Dogo et al., 2018). Moreover, because of ADAM's stability and performance in a wide range of case scenarios, several NN researchers instinctively favour it in all circumstances. Consequently, it is essential to conduct a study to investigate how optimizers operate in relation to the model and dataset utilized to gain a deeper understand of their behaviours.

As a result, the contribution of this paper is an experimental comparison of the performance of seven well-known and widely used GD optimization algorithms on a RNN model. RNN is One type of neural network that can process sequential inputs, such as time series and natural language. The RNN model uses the body motion dataset from the UCI ML datasets with several learning rates, and activation functions. Using convergence speed, accuracy, and loss function as performance metrics, this comparison shows how well and consistently each optimizer handled the problem of locating the proper and ideal minima throughout training.

BACKGROUND OF STUDY AND RELATED WORK

GD is a first-order iterative optimization technique that moves in the opposite direction as the gradient to find a local minimum of a loss function (Chandra et al., 2022). On the other

hand, gradient ascent—moving in the direction of the gradient—leads to a local maximum (Hallén, 2017). By modifying weights during backpropagation to reduce loss and address the curse of dimensionality, GD plays a critical role in DL optimization (Goodfellow et al., 2016).

SGD, AdaGrad, AdaDelta, RMSprop, ADAM, AdaMax, and NADAM are prominent optimization algorithms utilized for training machine learning models, including neural networks, with SGD adjusting parameters according to gradients derived from individual or small batches of instances (Schaul et al., 2013). AdaGrad is an algorithm that adapts the learning rate of each model parameter based on historical gradient information (Duchi et al., 2011; Solanke & Patnaik, 2020). AdaDelta is a variant of AdaGrad that uses a moving window of gradient updates instead of accumulating all past gradients (Zeiler, 2012). RMSprop is a variant of the GD algorithm that uses a decaying average of partial gradients to adapt the step size for each parameter (Hinton & Tieleman, 2012; Zou et al., 2019). ADAM is an adaptive learning rate optimization algorithm that combines the advantages of both AdaGrad and RMSprop (Kingma & Ba, 2015; Yi et al., 2020). AdaMax is a variant of ADAM that is based on the infinity norm (Fatima, 2020). NADAM is a variant of ADAM with Nesterov momentum (Adem & Kiliçarslan, 2019; Sutskever et al., 2013). These algorithms are utilized in DL packages like Caffe, Lasagne, TensorFlow, and Keras. To obtain improved generality, researchers continue to create optimizers, as seen in (Lv et al., 2017).

RNN utilize memory to integrate previous inputs, hence modifying the present input and result (Karna et al., 2024). Both unrollable finite impulse networks and temporally dynamic infinite impulse networks (Sherstinsky, 2020) are examples of RNN, which are characterized by infinite impulse response (Miljanovic, 2012). Like Gated Recurrent Units (GRUs), commonly referred to as feedback neural networks (FNN), these networks may have memory or gated states (Hochreiter & Schmidhuber, 1997).

Using different numbers of iterations and particular test function values on a stacked denoising autoencoder (SDA) architecture, based on convergence time, number fluctuations, and parameter update rate, authors in (Yazan & Talu, 2017) examined a comparison of optimization techniques based on SGD, specifically ADAM, AdaGrad, AdaDelta, RMSprop, SGD, and SGD with momentum. According to their experimental results in terms of rapid convergence, AdaDelta outperformed the other optimizers. The datasets they used for their tests are unknown. The author of (Papamakarios, 2014) used the logistic and Softmax regression on the synthetic and Modified National Institute of Standards and Technology (MNIST) handwritten digits datasets, respectively, to compare the performance of four GD-based variants on the limited convex objective fitting problem: GD, stochastic GD, semi-SGD, and stochastic average descent. In the authors' two trials, SGD outperformed SG in general, but the two hybrid forms achieved superior accuracy in more reasonable amounts of time. In a similar vein (Hallén, 2017) conducted a performance

comparison between GD and SGD using the MNIST ML dataset for linear regression and multinomial logistic regression, using accuracy, training, and convergence time as performance variables. In a recent study (Ruder, 2016), gave a simple overview of how modern GD optimization techniques behave.

While pointing out difficulties in optimizing GD (Shalev-Shwartz et al., 2017), experts advise flexible learning rates for complex neural network training. These difficulties include slow training, vanishing gradients as a result of insufficient conditioning, low signal-to-noise ratio (SNR), and limited gradient informativeness. However, a comprehensive evaluation of the impact of these popular optimization algorithms on an RNN architecture using image classification datasets appears to be lacking in existing studies.

Based on the related works, several types of studies can be conducted to further explore the optimization of neural networks. These studies can involve conducting a comprehensive comparison of various optimization algorithms on RNN using other datasets, developing a theoretical framework for selecting the most suitable optimizer for training RNN, and conducting experiments using various datasets and NN architectures to evaluate the performance of different optimizers. Additionally, studies can focus on investigating and developing new optimization techniques specifically designed for DL applications, analyzing the scalability and efficiency of different optimizers in large-scale neural network training, and optimizing NN with specific architectures. Furthermore, studies can explore the use of hybrid optimizers and ensemble methods to combine the strengths of different optimizers, investigate the performance of optimizers on non-convex problems, and study the performance of optimizers on sparse and large-scale datasets. These studies can help in understanding the performance of different optimizers, developing new optimization techniques, and improving the efficiency and accuracy of neural network training.

METHODOLOGY AND EXPERIMENTAL SETUP

This study used the LSTM networks, a version of RNN architecture specifically engineered to proficiently learn and retain long-range dependencies in sequential input (Hochreiter & Schmidhuber, 1997). On the developed model, the performance of seven well-known SGD optimization techniques was also demonstrated. This research utilized the Mobile Health (MHEALTH) dataset from the UCI ML repository, which is a body motion dataset intended to evaluate methodologies for human behavior analysis through multimodal body sensing (Banos et al., 2014) . The optimizers model examined were SGD, RMSProp, ADAM, AdaGrad, AdaDelta, AdaMax, and NADAM. For each trial, identical hyperparameter settings were applied. The body motion dataset was used to train the complete network across 100 epochs.

The MHEALTH dataset was utilised to train the LSTM architecture. The selected LSTM model underwent rigorous evaluation and analysis on the MHEALTH body motion dataset to determine its effectiveness in extracting meaningful information from the body motion.

Various metrics, including training time and test accuracy, were assessed to quantify the LSTM architecture performance. The impact by experimenting with various combinations of learning rates, activation functions, and LSTM architectures was investigated. This analysis involved training the LSTM model with different optimizers. Values on subsets of the MHEALTH body motion dataset and observing the corresponding training and test accuracy changes. Training dataset is presented to the model during training. Setting all the configurations with aims to evaluate and compare the performance of seven widely used first-order stochastic gradient-based optimization algorithms within a RNN framework. specifically focuses on how these optimizers — ADAM, RMSprop, SGD, AdaGrad AdaDelta, NADAM, and AdaMax—perform in terms of test accuracy and training time when applied to a body motion dataset. The objectives to be achieved are to identify which optimizer yields the best results by experimenting with various combinations of learning rates, activation functions, and LSTM architectures.

Data Collection - UCI Body Motion Dataset

In this study, the dataset used to measure and compare the performance of optimization algorithms was the motion dataset. The MHEALTH dataset was devised to benchmark techniques dealing with human behaviour analysis based on multimodal body sensing. This dataset consisted of 23 features (columns). The number of samples determined was 100,000 and there was no missing data in the dataset. The data collected for each subject was stored in a different log file. There were 10 people doing 12 common daily activities, such as standing still, sitting, and relaxing, lying down, walking, climbing stairs, waist bends forward, frontal elevation of arms, knees bending, cycling, jogging, running, and jump front/back, and each activity was recorded for 1 minute. For one object, it took 12 minutes to collect the data for 12 activities. Each file contained the samples (by rows) recorded for all sensors (by columns).

RNN Model Setup

Three LSTM network architecture models were created for this experimental investigation to observe the variations in the weight values produced by each architecture, which were 1) one LSTM as default, 2) two LSTMs, and 3) added dropout model between the two LSTMs. The LSTM architecture models in this study can be seen from Figures 2 till 4.

The network architecture in this study has 23 input features (m_1, m_2, m_3, ... , m_{23}) with input representation: $X_t \in \mathbb{R}^{23}$ at each time step, t. The following is a mathematical model that represents the LSTM network architecture Model-1 (Figure 2).

LSTM layer (hidden layer), the LSTM unit receives the input and the previous hidden state (t-1) and cell state (t-1). The mathematical model or equation for the main LSTM computation is using the equation (Hochreiter & Schmidhuber, 1997):

$$f_t = \sigma\,(W_f[h_t\text{-}1, Xt] + b_f) \quad [1]$$

$$i_t = \sigma\,(W_i[h_t\text{-}1, Xt] + b_i) \quad [2]$$

$$\tilde{C}_t = tanh\,(W_c[h_t\text{-}1, Xt] + b_c) \quad [3]$$

$$C_t = f_t \odot C_{t-1} + i_t \odot \tilde{C}_t \quad [4]$$

$$o_t = \sigma\,(W_o[h_t\text{-}1, Xt] + b_o) \quad [5]$$

$$h_t = o_t \odot tanh\,(C_t) \quad [6]$$

The LSTM accepts the current input vector X_t the previous hidden state h_t-1, and the previous cell state C_{t-1}. The forget gate, provided by $f_t = \sigma\,(W_f[h_t\text{-}1, Xt] + b_f)$ determines what parts of the previous memory e C_{t-1} must be saved. The input gate $i_t = \sigma\,(W_i[h_t\text{-}1, Xt] + b_i)$ and candidate cell state $\tilde{C}_t = tanh\,(W_c[h_t\text{-}1, Xt] + b_c)$ work together to decide what new information must be added to the memory. The new state of the cell is computed as $C_t = f_t \odot C_{t-1} + i_t \odot \tilde{C}_t$, where element-wise multiplication is represented by $\odot$. The output gate $o_t = \sigma\,(W_o[h_t\text{-}1, Xt] + b_o)$ then decides how much of the memory is to be given as output to the next time step. Finally, the new hidden state is calculated as $h_t = o_t \odot tanh\,(C_t)$, and this will be both the output of the present LSTM unit and also an input to the next step of the sequence. In these equations, σ is the sigmoid activation function (which outputs between 0 and 1), and tanh is the hyperbolic tangent function (which outputs between -1 and 1). The weight matrices W_f, W_i, W_c, W_o

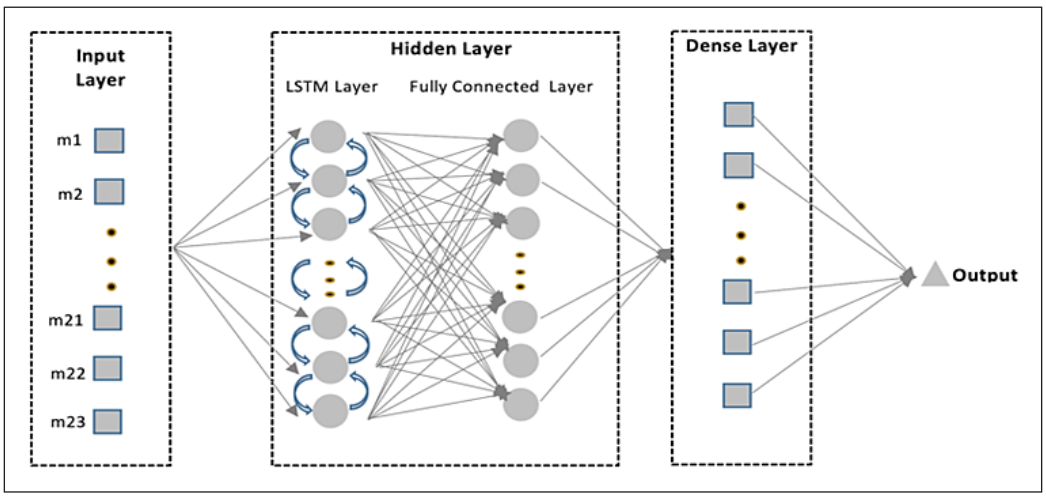

Figure 2. Model-1 (One hidden layer with one long short-term memory layer [LSTM])

are learned weight parameters for every gate, and b_f, b_i, b_c, b_o are corresponding bias vector.

Fully connected layer receives *t* from the LSTM layer and processes it with weights and biases.

$$H = \text{ReLU}\,(W_{fc} h_t + b_{fc}) \quad [7]$$

where, W_{fc} is referred to weight of fully connected layer and b_{fc} is bias.

Dense layer (output) takes the output of the fully connected layer and maps it to the output using the last weight and bias.

$$y = (W_{dense}\, H + b_{dense}) \quad [8]$$

where, W_{dense} is referred to weight of output layer and b_{dense} is bias.

A representation of the second network architecture that was utilized in the research is shown in Model -2, which can be found in this following Figure 3.

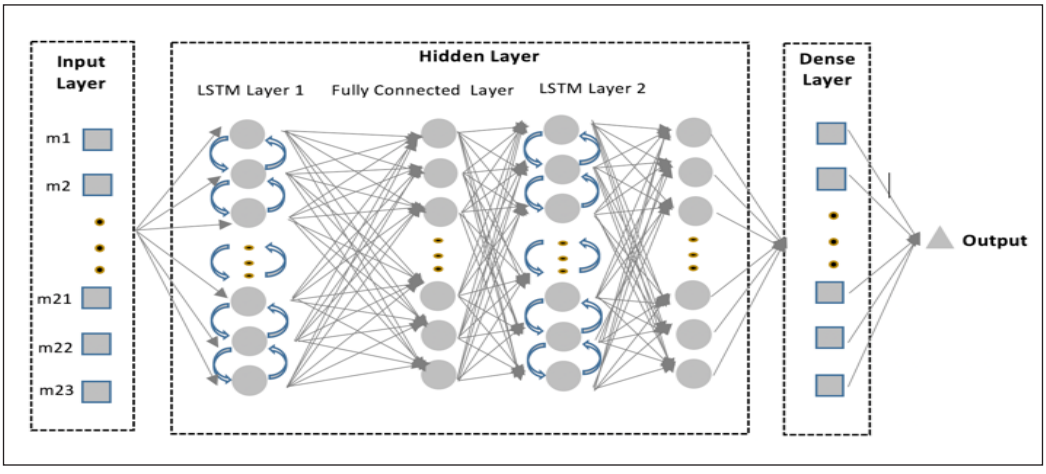

Figure 3. Model-2 (Two hidden layers with two long short-term memory [LSTM] layers)

The LSTM architecture models, Model-2 and Model-3 (Figures 3 and 4), are essentially the same as the LSTM architecture model, Model-1, for computations and mathematical models; however, they have an additional hidden layer. The following is a mathematical model that represents the LSTM network architecture Model-2 (Chen et al., 2022; Hochreiter & Schmidhuber, 1997; Van Houdt et al., 2020).

LSTM Layer 1: $$h_t^{(1)} = \sigma\left(W_1 X_t + U_1 h_{t-1}^{(1)} + b_1\right) \quad [9]$$

Fully Connected Layer: $$H = \text{ReLU}\,(W_{fc} h_t^{(1)} + b_{fc}) \quad [10]$$

LSTM Layer 2: $$h_t^{(2)} = \sigma\,(W_2 H + U_2 h_{t-1}^{(2)} + b_2) \quad [11]$$

Dense Layer (Output): $$y = (W_o h_t^{(2)} + b_o) \quad [12]$$

Model-2 enhances the standard LSTM by adding an extra hidden layer, allowing it to learn more complex patterns in sequential data. In the first LSTM layer, the hidden state $h_t^{(1)}$ is computed from the input X_t, previous hidden state $h_{t-1}^{(1)}$, input weights W_1, recurrent weights U_1, and bias b_1, using the sigmoid activation σ: $h_t^{(1)} = \sigma\left(W_1 X_t + U_1 h_{t-1}^{(1)} + b_1\right)$. This hidden state is transformed by a fully connected layer into a new feature H using ReLU activation: $H = \text{ReLU}\,(W_{fc} h_t^{(1)} + b_{fc})$ where W_{fc} and b_{fc} are the weights and bias of the dense layer. The second LSTM layer uses this H along with its previous hidden state $h_t^{(2)}$, weights W_2, U_2, and bias b_2 to compute: $h_t^{(2)} = \sigma\,(W_2 H + U_2 h_{t-1}^{(2)} + b_2)$. Finally, the model output is calculated as: $y = (W_o h_t^{(2)} + b_o)$ with W_o and b_o representing the output layer's weights and bias. This design enables the model to learn both short- and long-term dependencies effectively. Henceforth, the network architecture used in this study is LSTM model-3, which can be seen in the following Figure 4.

Figure 4 is the LSTM architecture Model-3, which is essentially identical to the previous one, with the exception that a dropout layer is used in place of the middle layer to avoid

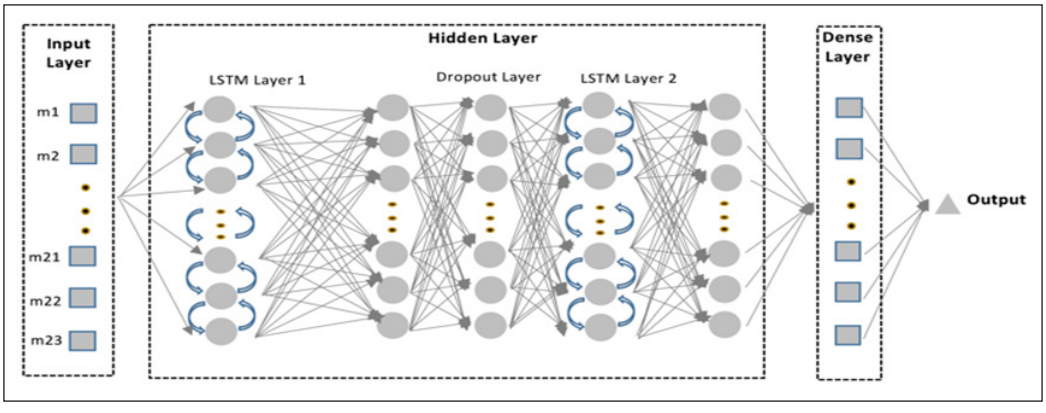

Figure 4. Model-3 (two hidden layers with two long short-term memory [LSTM] layers and one dropout layer)

overfitting. The following mathematical model represents the LSTM network architecture Model-3 (Chen et al., 2022; Hochreiter & Schmidhuber, 1997; Van Houdt et al., 2020).

LSTM layer 1: $$h_t^{(1)} = \sigma\left(W_1 X_t + U_1 h_{t-1}^{(1)} + b_1\right) \quad [13]$$

Dropout layer: $h_t^{(d)} = \text{Dropout}\left(h_t^{(1)}, p\right)$; p is dropout probability [14]

LSTM layer 2: $$h_t^{(2)} = \sigma\left(W_2 h_t^{(d)} + U_2 h_{t-1}^{(2)} + b_2\right) \quad [15]$$

Dense layer (Output): $$y = \left(W_o h_t^{(2)} + b_o\right) \quad [16]$$

where, the hidden state $h_t^{(1)}$ using the current input X_t, the previous hidden state $h_{t-1}^{(1)}$, weight matrices W_1 and U_1, and a bias term b_1. Then, a dropout layer is applied to $h_t^{(1)}$ producing a new intermediate state $h_t^{(d)}$ by randomly dropping units based on the dropout probability p. This result is fed into the second LSTM layer, which produces $h_t^{(2)}$ using W_2, U_2, and b_2. The final output y is calculated through a dense layer with weights W_o and bias b_o.

Optimization Techniques

Experiments are conducted by comparing seven different optimization methods, namely SGD, ADAM, RMSprop, AdaGrad, AdaDelta, NADAM, and AdaMax. Each GD based optimization method has a unique approach in updating the model parameters. The main equations for the methods used in this study are as follows:

The SGD method is a basic form of gradient-based optimization. The mathematical equation for all the GD method is using this following equations (Ruder, 2016):

$$\theta_{t+1} = \theta_t - \eta \nabla L\left(\theta_t\right) \quad [17]$$

where, the model parameters at a given iteration (denoted as θ_t) are updated to θ_{t+1} by moving in the opposite direction of the gradient of the loss function. This update is controlled by a learning rate (η), a small positive value that determines the size of each step towards minimizing the loss. The gradient, represented as $\nabla L\left(\theta_t\right)$, indicates how much the loss changes with respect to the current parameters, guiding the optimization process. AdaGrad adjusts the learning rate for each parameter based on the previous gradient:

$$g_t = \nabla L\left(\theta_t\right) \quad [18]$$

$$\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{G_t + \epsilon}} \odot g_t \quad [19]$$

$$G_t = G_{t-1} + g_t^2 \quad [20]$$

The AdaGrad optimizer improves the standard gradient descent method by adapting the learning rate for each model parameter based on the historical accumulation of gradients. At each training step, it calculates the gradient of the loss function with respect to the current parameters, noted as $g_t = \nabla L(\theta_t)$. Instead of applying a fixed learning rate, AdaGrad adjusts it by dividing the global learning rate η by the square root of the sum of all past squared gradients G_t, and a small constant ε to maintain numerical stability. This results in the update rule $\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{G_t + \epsilon}} \odot g_t$, where $\odot$ indicates element-wise multiplication. The cumulative term G_t is updated at each step as $G_t = G_{t-1} + g_t^2$. This approach naturally decreases the learning rate for parameters that frequently receive large updates, which helps improve learning efficiency and model performance, especially in scenarios involving sparse or high-dimensional data. RMSProp overcomes AdaGrad's weakness by using exponential averaging of squared gradients:

$$g_t = \nabla L(\theta_t) \quad [21]$$

$$E[g_t^2] = \rho E[g_{t-1}^2] + (1-\rho) g_t^2 \quad [22]$$

$$\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{E[g_t^2] + \epsilon}} \odot g_t \quad [23]$$

where, g_t is the gradient of the loss function with respect to the model parameters at iteration and $\nabla L(\theta_t)$ represents the direction and rate of increase of the loss function at parameters. Equation [22] defines the exponential moving average of the squared gradients, where $E[g_t^2] =$ represents the smoothed estimate of the squared gradient at iteration *t*. The term ρ is a decay factor, usually set between 0.9 and 0.99, which controls how much influence the past squared gradients g_{t-1}^2 have on the current estimate. Meanwhile, g_t^2 is the element- wise square of the current gradient. Equation [23] update the model parameters in RMSProp, where θ_t and θ_{t+1} are the current and updated parameters, respectively. η is the learning rate, ϵ is a small constant to avoid division by zero, and $\odot$ denotes element-wise multiplication. ADAM combines momentum and RMSProp for adaptive learning rate adjustment (Kingma & Ba, 2015; Ruder, 2016):

$$m_t = \beta_1 m_{t-1} + (1-\beta_1) g_t \quad [24]$$

$$v_t = \beta_2 v_{t-1} + (1-\beta_2) g_t^2 \quad [25]$$

$$\widehat{m_t} = \frac{m_t}{1-\beta_1^t} + \widehat{v_t} = \frac{v_t}{1-\beta_2^t} \quad [26]$$

$$\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{v_t+\epsilon}} m_t \quad [27]$$

In the ADAM optimization algorithm, parameter updates are computed using both the first and second moment estimates of the gradients. Equation [24] defines the first moment estimate, where g_t is the current gradient and *β1* is the exponential decay rate for the gradient moving average. Equation [25] computes the second moment estimate, where *β2* is the decay rate for the squared gradients. Since both m_t and v_t are initialized at zero, bias-corrected estimates are used as shown in Equation [26]. The model parameters are updated using Equation [27], where θ_t and θ_{t+1} are the current and updated parameters, η is the learning rate, and ϵ is a small constant added for numerical stability. This combination of momentum and adaptive learning rate adjustment makes Adam particularly effective for deep learning tasks. AdaDelta overcomes gradient accumulation in AdaGrad by using a moving average of the gradients (Ruder, 2016):

$$E[g_t^2] = \rho E[g_{t-1}^2] + (1-\rho) g_t^2 \quad [28]$$

$$\Delta\theta_t = -\frac{\sqrt{E[\Delta\theta_{t-1}^2]}}{\sqrt{E[g_t^2]+\epsilon}} + \epsilon \quad [29]$$

$$E[\Delta\theta_t^2] = \rho E[\Delta\theta_{t-1}^2] + (1-\rho)(\Delta\theta_t)^2 \quad [30]$$

The algorithm maintains a decaying average of past squared gradients as shown in Equation [28], where g_t is the gradient at the current time step, and ρ is a decay constant that controls how much weight is given to past values. Parameter updates are calculated using Equation [29], where $\Delta\theta_t$ represents the update applied to the model parameters. This formulation scales the raw gradient based on the ratio between the root mean square of previous updates and the current average of squared gradients, with ϵ (a small constanta) included to prevent division by zero and enhance numerical stability. Updates the moving average of squared parameter updates using Equation [30], where $\Delta\theta_t^2$ is the element-wise square of the current update. This recursive structure ensures consistent and adaptive parameter updates over time.

Adamax is an ADAM variant based on norm infinity ($\ell \bowtie$), the mathematical equation is using this following equations (Ruder, 2016) :

$$m_t = \beta_1 m_{t-1} + (1 - \beta_1) g_t \quad [31]$$

$$u_t = Max\,(\beta_2 u_{t-1}, |g_t^2|) \quad [32]$$

$$\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{u_t}} m_t \quad [33]$$

The first moment estimate is computed using an exponentially weighted moving average of the gradients, as shown in Equation [31], where m_t is the exponentially weighted average of gradients, g_t is the current gradient, and β_1 is the decay rate. Adamax uses the infinity norm, as defined in Equation [32], where u_t is the running maximum of the scaled gradient magnitudes, β_2 is the decay rate for this term, and $|g_t|$ is the element-wise absolute gradient. The parameters are then updated using Equation [33], where θ_t is the parameter at step *t*, η is the learning rate, and the denominator normalizes the update using the square root of the infinity norm. NADAM added Nesterov's momentum to ADAM's (Ruder, 2016).

$$m_t = \beta_1 m_{t-1} + (1 - \beta_1) g_t \quad [34]$$

$$u_t = \beta_2 u_{t-1} + (1 - \beta_2) g_t^2 \quad [35]$$

$$\widehat{m_t} = \frac{m_t}{1-\beta_1^t} + \frac{(1-\beta_1}{1-\beta_1^t} g_t \quad [36]$$

$$\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{\widehat{u_t} + \epsilon}} \widehat{m_t} \quad [37]$$

In the NADAM optimization algorithm θ_t represent for the model parameters at iteration *t,* and θ_{t+1} denote the update parameters.The gradient of the loss function at time t is denoted by g_t. The algorithm keeps track of two moving averages: m_t, which is the exponentially weighted average of past gradients (also known as the first moment), and u_t, which is the exponentially weighted average of past squared gradients (the second moment). These averages help smooth out the noise in the gradient updates. To improve stability early in training, NADAM uses a bias- corrected version of the first moment, noted as $\widehat{m_t}$, which also incorporates a lookahead gradient term inspired by Nesterov momentum. The constants β_1 and β_2 control the decay rates of these moving averages and are usually set close to 1. The learning rate, symbolized by η, determines how big each step should be when updating the parameters, and ϵ is a small value added to the denominator to avoid division by zero, ensuring numerical stability.

Activation Functions

Five activation functions—Sigmoid, Softmax, ReLU, tanh, and Exponential—were employed in this study to maximize the RNN model's performance. Particularly in deciding how information is processed and transmitted in each neuron layer, each of these functions has a distinct function and set of properties in the model training process. Sigmoid is perfect for binary classification because it compresses outputs between 0 and 1. Nevertheless, it slows down model updates due to the vanishing gradient issue. The function for all the activation function variant is using this following equations (Szandała, 2020):

$$\sigma\,(x) = \frac{1}{1 + e^{-x}} \qquad [38]$$

The sigmoid function σ (x) is the output of the activation function, mapping any real-valued input x to a value between 0 and 1. The input x is typically the weighted sum of inputs to a neuron, given by $x = w_1x_1 + w_2x_2 + ... + w_nx_n + b$, where w_i is the weight of the *i*-th input, x_i is the *i*-th input feature, and b is the bias term. The constant e is Euler's number, approximately 2.71828, and e^{-x} is the exponential function with exponent -x, which decreases rapidly as x increases.

Softmax transforms inputs into a probability distribution, generalizing sigmoid for multi-class classification. Its downside is overfitting, as it might give excessive probabilities. Softmax is defined as:

$$\text{Softmax}\,(x_i) = \frac{e^{x_i}}{\sum_j e^{x_i}} \qquad [39]$$

where, (x_i) is the output for the *i*-th class, representing the predicted probability that the input belongs to class *i*. The term x_i refers to the logit or raw output score from the model for the *i*-th class, (before the activation). The constant e denotes Euler's number, approximately equal to 2.71828. The expression e^{x_i} is the exponential of the logit x_i, which scales the output positively. The denominator $\sum_j e^{x_i}$ is the sum of the exponentials of all logits across all classes *j* used to normalizes the output so that they add up to 1.

By only activating positive values, the ReLU resolves the vanishing gradient problem and accelerates convergence. It has to deal with the fading ReLU issue, though, where neurons might stop working. This function is formulated as:

$$\text{ReLU}(x) = \begin{cases} x, & x > 0 \\ \alpha 0, & x \leq 0 \end{cases} \qquad [40]$$

where, ReLU(x) is the output of the activation function applied to the input x, which typically represents the weighted sum of inputs to a neuron. The symbol x is the input value, and α is a small positive constant (usually around 0.01) called the leakage coefficient, which allows a small, non-zero gradient when x is negative. This helps avoid the "dying ReLU" problem by ensuring that neurons can still update during training even when they receive negative inputs.

Similar to sigmoid, tanh improves representation by having an output range of -1 to 1. The vanishing gradient still affects it, particularly at extreme values. This function is defined as:

$$\tanh(x) = \frac{e^{x} - e^{-x}}{e^{x} + e^{-x}} \quad [41]$$

where, x represents the input value, typically the weighted sum of inputs to a neuron. The symbol e stands for Euler's number, approximately equal to 2.71828. The terms e^{x} and e^{-x} are the exponential functions of x and -x, respectively. The numerator $e^{x} - e^{-x}$ gives the difference of exponentials, while the denominator $e^{x} + e^{-x}$ ensures the output remains bounded between -1 and 1.

Exponential learning speeds up learning by exponentially magnifying input values, but it also runs the danger of inflating gradients, which can cause training instability. Exponential defined as:

$$\text{Exponential}\ (x) = e^{x} \quad [42]$$

where, x represents the input value, typically the weighted sum of inputs to a neuron. The symbol e stands for Euler's number (~2.71828) and the terms e^{x} grows rapidly with positive x and approaches zero as x becomes negative.

Experimental Configuration

Experiments were conducted to compare several optimization methods discussed, such as ADAM, RMSprop, SGD, AdaGrad, AdaDelta, NADAM, and AdaMax. The RNN model consisted of an input layer that was then forwarded to the LSTM architecture, and the last layer was the output layer. Based on Tabel 1, the training data was processed for a batch size of 32 sample records and 100 epochs. The training data was compiled with the learning rate of 0.1, 0.01, and 0.001, respectively. The network structure is shown in Table 1.

Table 1
Recurrent Neural Network structure specification

Description	Value
Learning rate	0.1; 0.01; 0001
Number of epoch	100
Batch size	32
Number of training	80% : 80,000
Number of validation	20% : 20,000
Optimizer	ADAM, RMSprop, SGD, AdaGrad, AdaDelta, NADAM, and AdaMax
Activation function	Sigmoid, Softmax, ReLU, tanh, and Exponential

Note. ADAM = Adaptive Moment Estimation; RMSProp = Root Mean Square Propagation; SGD = Stochastic Gradient Descent; AdaGrad = Adaptive Gradient Algorithm; AdaDelta = Adaptive Delta; NADAM = Nesterov-accelerated Adaptive Moment Estimation; AdaMax = Adaptive Moment Estimation with Infinity Norm

RESULTS AND DISCUSSION

The purpose of the experiment was to analyze each configuration of optimizers, which included ADAM, RMSprop, SGD, AdaGrad, AdaDelta, NADAM, and AdaMax with three LSTM network architecture models, five activation functions, and three learning rates (α), respectively. The comparison results shown in Table 2 are the training process or training time in minutes, abbreviated by "TT", and the test accuracy in percentages, abbreviated by "TA", for every configuration.

This study systematically evaluated the performance of various optimization algorithms using RNN architecture, specifically LSTM networks. The implementation was carried out using the TensorFlow DL library and Python programming language. The experiments were performed on a Dell personal computer equipped with an Intel Core i7 processor, 16 GB of RAM, and an NVIDIA GeForce GTX 1660 graphics card, ensuring a robust computational setup. Performance evaluation was based on test accuracy and training time, with results systematically compiled into tables summarizing the impact of different optimizers and hyperparameter configurations such as learning rates (0.1, 0.01, and 0.001) and five activation functions (Sigmoid, Softmax, ReLU, tanh, and Exponential). Each trial maintained a consistent batch size of 32 samples, ensuring uniform data processing across all experiments for comparability.

Table 2 indicates that testing accuracy is determined by the model evaluation outcomes following training with the test dataset. The test accuracy value is calculated by splitting the dataset into two sections: the training process (80%) and the testing process (20%). The model is trained using training data with various parameters (optimizer, learning rate, activation function) during the training process. Following training, the model is tested using the test dataset (test set). By contrasting the actual label in the test dataset with the model prediction, the degree of accuracy, also known as test accuracy, is determined.

Table 2
Performance comparisons of studied optimization algorithms

Optimizer model	LSTM architecture	Activation function	α = 0.1		α = 0.01		α = 0.001	
			TT (min)	TA (%)	TT (min)	TA (%)	TT (min)	TA (%)
ADAM	Model-1	Sigmoid	81.56	78.31	73.73	76.38	76.43	78.29
		Softmax	17.96	1.83	78.33	1.83	80.18	1.83
		ReLU	79.83	78.31	79.24	78.31	79.67	78.31
		tanh	17.8	78.31	17.91	78.31	17.87	78.31
		Exponential	17.84	78.31	17.89	78.31	17.95	78.31
	Model-2	Sigmoid	32.04	78.31	31.8	78.31	31.66	78.31
		Softmax	31.74	1.83	31.81	1.83	31.78	1.83
		ReLU	31.97	78.31	32.47	78.31	34.4	78.31
		tanh	34	78.31	34.28	78.31	34.14	78.31
		Exponential	34.03	78.31	33.53	78.31	34.29	78.31
	Model-3	Sigmoid	32.05	75.09	31.17	75.10	31.37	75.11
		Softmax	31.53	1.83	31.66	1.83	31.51	1.83
		ReLU	31.46	78.31	31.63	78.31	31.6	78.31
		tanh	31.49	78.31	31.47	78.31	31.55	78.31
		Exponential	31.6	78.31	31.55	78.31	31.56	78.31
RMSprop	Model-1	Sigmoid	17.81	78.31	17.72	78.31	18.03	78.31
		Softmax	18.02	1.83	18.02	1.83	18.1	1.83
		ReLU	18.01	78.31	18.01	78.31	17.89	78.31
		tanh	17.9	78.31	17.99	78.31	17.99	78.31
		Exponential	18	78.31	18	78.31	18.02	78.31
	Model-2	Sigmoid	34.28	78.31	33.84	78.31	34.16	78.31
		Softmax	34.03	1.83	33.91	1.83	33.16	1.83
		ReLU	34.18	78.31	33.87	78.31	34.78	78.31
		tanh	34.35	78.31	33.87	78.31	33.04	78.31
		Exponential	33.05	78.31	33.11	78.31	33.08	78.31
	Model-3	Sigmoid	31.01	78.31	30.97	75.07	30.9	75.06
		Softmax	31.01	1.83	31.04	1.83	31.05	1.83
		ReLU	31.05	78.31	31.11	78.31	31.21	78.31
		tanh	31.09	78.31	31.39	78.31	31.07	78.31
		Exponential	31.12	78.31	31.25	78.31	31.52	78.31

Table 2 *(continue)*

Optimizer model	LSTM architecture	Activation function	α = 0.1		α = 0.01		α = 0.001	
			TT (min)	TA (%)	TT (min)	TA (%)	TT (min)	TA (%)
SGD	Model-1	Sigmoid	18	78.31	18.01	78.31	18.04	78.31
		Softmax	18.01	1.83	18.01	1.83	18.01	1.83
		ReLU	18	78.31	18	78.31	18	78.31
		tanh	18.01	78.31	18.01	78.31	18.01	78.31
		Exponential	18.01	74.76	18.01	74.85	18.01	74.87
	Model-2	Sigmoid	34.16	78.30	34.12	78.30	33.83	78.30
		Softmax	33.35	1.83	33.03	1.83	33.03	1.83
		ReLU	33.02	78.31	32.74	78.31	30.10	78.31
		tanh	30.2	78.31	30.12	78.31	30.07	78.31
		Exponential	30.25	78.28	30.16	78.29	30.33	78.29
	Model-3	Sigmoid	31.13	78.24	30.99	78.26	30.98	78.27
		Softmax	30.76	1.83	31.07	1.83	31.43	1.83
		ReLU	31.32	78.31	31.64	78.31	31.7	78.31
		tanh	32.01	78.31	32.37	78.31	32.61	78.31
		Exponential	32.89	78.31	32.30	78.31	31.97	78.31
AdaGrad	Model-1	Sigmoid	17.06	78.31	17.02	78.31	17.02	78.31
		Softmax	17.01	1.83	17.08	1.83	17.26	1.83
		ReLU	17.02	78.31	17.23	78.31	17.16	78.31
		tanh	17.05	78.31	17.04	78.31	17.01	78.31
		Exponential	17.03	78.31	17.02	78.31	17.01	78.31
	Model-2	Sigmoid	34.47	78.30	34.36	78.31	34.96	78.31
		Softmax	34.82	1.83	34.95	1.83	35.01	1.83
		ReLU	34.98	78.31	35.2	78.31	35.11	78.31
		tanh	35.18	78.31	33.77	78.31	33.83	78.31
		Exponential	33.85	78.29	34.02	78.29	36.26	78.29
	Model-3	Sigmoid	32.76	78.31	32.88	78.31	32.83	78.31
		Softmax	32.29	1.83	31.99	1.83	31.99	1.83
		ReLU	32.78	78.31	32.77	78.31	32.01	78.31
		tanh	31.88	78.31	31.86	78.31	32.06	78.31
		Exponential	32.02	78.31	32.1	78.31	32.13	78.31

Table 2 *(continue)*

Optimizer model	LSTM architecture	Activation function	α = 0.1		α = 0.01		α = 0.001	
			TT (min)	TA (%)	TT (min)	TA (%)	TT (min)	TA (%)
AdaDelta	Model-1	Sigmoid	20.63	78.31	21.4	78.31	21.03	78.31
		Softmax	21.03	1.83	21.04	1.83	21.1	1.83
		ReLU	21.04	78.31	21.09	78.31	21.03	78.31
		tanh	21.06	78.31	21.03	78.31	21.03	78.31
		Exponential	21.10	74.92	21.06	74.93	21.09	74.94
	Model-2	Sigmoid	34.58	78.31	34.17	78.31	34.33	78.31
		Softmax	34.67	1.83	35.03	1.83	34.71	1.83
		ReLU	34.90	78.31	34.94	78.31	34.93	78.31
		tanh	35.01	78.31	33.06	78.31	33.37	78.31
		Exponential	33.3	78.29	33.58	78.29	35.92	78.29
	Model-3	Sigmoid	31.48	78.31	31.53	78.31	31.13	78.31
		Softmax	31.62	1.83	31.85	1.83	31.63	1.83
		ReLU	32.77	78.31	32.8	78.31	32.69	78.31
		tanh	32.15	78.31	31.68	78.31	31.42	78.31
		Exponential	31.16	74.73	31.42	74.66	31.42	74.66
NADAM	Model-1	Sigmoid	21.99	78.31	21.17	78.31	21.05	78.31
		Softmax	21.08	1.83	21.06	1.83	21.09	1.83
		ReLU	21.06	78.31	21.11	78.31	21.13	78.31
		tanh	21.08	78.31	21.11	78.31	21.09	78.31
		Exponential	21.1	78.31	21.28	78.31	20	78.31
	Model-2	Sigmoid	33.38	78.31	33.37	78.31	33.46	78.31
		Softmax	33.98	1.83	34.05	1.83	34.05	1.83
		ReLU	34.05	78.31	34.03	78.31	34.04	78.31
		tanh	34.53	78.31	35.58	78.31	35.52	78.31
		Exponential	35.18	78.31	34.99	78.31	34.90	78.31
	Model-3	Sigmoid	31.97	75.06	31.17	75.10	31.28	75.09
		Softmax	31.54	1.83	31.61	1.83	31.73	1.83
		ReLU	31.64	78.31	31.46	78.31	31.36	78.31
		tanh	31.65	78.31	31.8	78.31	31.94	78.31
		Exponential	31.14	78.31	31.36	78.31	32.11	78.31

Table 2 *(continue)*

Optimizer model	LSTM architecture	Activation function	α = 0.1		α = 0.01		α = 0.001	
			TT (min)	TA (%)	TT (min)	TA (%)	TT (min)	TA (%)
AdaMax	Model-1	Sigmoid	17.07	78.24	17.06	78.27	17.03	78.29
		Softmax	17.83	1.83	17.86	1.83	18.03	1.83
		ReLU	18.04	78.31	18.07	78.31	18.69	78.31
		tanh	18.97	78.31	20.01	78.31	20.1	78.31
		Exponential	20.74	78.31	20.97	78.31	21.45	78.31
	Model-2	Sigmoid	33.90	78.29	33.27	78.25	33.53	78.27
		Softmax	33.72	1.83	33.86	1.83	33.94	1.83
		ReLU	33.83	78.31	33.72	78.31	33.89	78.31
		tanh	34.28	78.31	35.49	78.31	35.34	78.31
		Exponential	34.74	78.30	34.6	78.28	34.63	78.28
	Model-3	Sigmoid	31.25	78.21	31.13	77.74	31.23	78.20
		Softmax	31.68	1.83	31.5	1.83	31.63	1.83
		ReLU	31.36	78.31	31.50	78.31	31.39	78.31
		tanh	31.26	78.31	31.45	78.31	31.5	78.31
		Exponential	31.69	75.46	31.98	76.28	31.92	76.56

Note. TT = Training time; TA = Training accuracy; ADAM = Adaptive Moment Estimation; RMSProp = Root Mean Square Propagation; SGD = Stochastic Gradient Descent; AdaGrad = Adaptive Gradient Algorithm; AdaDelta = Adaptive Delta; NADAM = Nesterov-accelerated Adaptive Moment Estimation; AdaMax = Adaptive Moment Estimation with Infinity Norm

Enclosed is the comprehensive summary of the analysis conducted on each optimizer:

1. The best performance of the ADAM optimizer is found using the LSTM architecture Model-1 with the tanh activation function, and the learning rate of 0.1, respectively. Moreover, the ADAM optimizer performs well using the sigmoid, tanh, and exponential activation functions, exhibiting the lowest running time and highest test accuracy. However, the ADAM optimizer shows significantly higher running time and lower test accuracy using the Softmax and ReLU activation functions compared to the tanh, sigmoid and exponential activation functions.
2. The RMSprop produces the best performance using the LSTM architecture Model-1 with the sigmoid activation function and the learning rate of 0.01. However, RMSprop shows poor performance in terms of high running time and low accuracy value for each configuration using the Softmax activation function.

3. The SGD performs well in most combinations of activation functions and learning rates, wherein the lowest running time and highest test accuracy achieved using the sigmoid and ReLU activation functions. However, the SGD exhibits higher running time and lower test accuracy using the Softmax activation functions compared to the tanh and exponential activation functions.
4. The AdaGrad performs well using the sigmoid, tanh and exponential activation functions. However, the AdaGrad shows the worst performance using the Softmax activation function followed by the ReLU activation function with the highest running time and lowest test accuracy.
5. In general, the AdaDelta performs well using the sigmoid, ReLU and tanh activation functions in each configuration. The AdaDelta performs poorly, with the highest running time and lowest test accuracy using the Softmax activation function.
6. NADAM has quite poor performance compared to other optimizer models for each configuration.
7. The AdaMax performs well using the sigmoid and ReLU activation functions with a learning rate of 0.01 and 0.001, exhibiting the lowest running time and highest test accuracy. However, AdaMax shows higher running time and lower test accuracy for each configuration using the Softmax activation function compared to others.

In sum, the two best optimizers in terms of less training time and high accuracy are AdaGrad and SGD optimizers. Both AdaGrad and SGD optimizers perform well across a wide range of activation functions, Precisely, both optimizers achieve high accuracy and low running time across a wide range of models and optimizers, particularly effective with exponential and sigmoid. While SGD is an easy and widely used method, AdaGrad offers more adaptive and efficient updates, making it particularly suitable for sparse data and tasks where the learning rate needs to be adjusted dynamically (Alzubaidi et al., 2021; Duchi et al., 2011). Where its application is suitable for the case in this study which has large amounts of data or high-dimensional data. AdaGrad and SGD perform better than others due to their enhanced efficiency and robustness against noisy data. Additionally, they dynamically adjust the learning rate for each model parameter, which helps to avoid the issue of vanishing gradients. This combination of efficient updates and adaptive learning rates enables these optimizers to effectively optimize DL models (Solanke & Patnaik, 2020).

Moreover, the two best activation functions are sigmoid and exponential. The choice of sigmoid or exponential activation function significantly impacts the training time of neural network models. Both sigmoid and exponential activation functions provide a continuous output range, which facilitates efficient updates by the optimizers (Yi et al., 2020). This is particularly important for SGD and AdaGrad, which rely on the gradient of the loss function calculated for a single training example or a small batch of examples. The continuous output

range of sigmoid and exponential activation functions also helps in reducing the impact of noisy data on the model's performance (Yi et al., 2020), which is crucial for optimizers like SGD and AdaGrad. Additionally, the continuous output range helps in adapting the learning rate for each model parameter, which is particularly important for AdaGrad.

Both sigmoid and exponential activation functions consistently perform well across different optimizers and activation functions, providing a continuous output range that facilitates efficient updates and robustness to noisy data. In contrast, activation functions like Softmax and ReLU may result in poorer performance due to their discrete output ranges. The interaction between optimizers and activation functions is crucial, and selecting the right combination of optimizer and activation function is essential for a specific task. Overall, the choice of sigmoid or exponential activation function plays a significant role in the performance of NN models, and it is crucial to experiment with different optimizers and activation functions to find the best combination for a specific task and dataset (Zou et al., 2019). In summary, the sigmoid and exponential activation functions are particularly effective with AdaGrad and SGD due to their continuous output range, non-linearity, robustness to noise, efficient updates, learning rate adaptation, robustness to local minima, performance across activation functions, and theoretical analysis. These characteristics make them well-suited for DL models and ensure that they can learn and represent complex patterns in the data, effectively (Mehmood et al., 2023).

Meanwhile, the Softmax activation function has very poor performance due to not providing optimal performance across different models and optimizers. This is because the special properties of the Softmax function, which are non-sparsity and potential vanishing gradients hinder the optimization process, hence generate the overfitting problem. Consequently, the Softmax function in the LSTM architecture does not provide a clear separation between classes, leading to poor generalization performance.

In addition to that, the Softmax activation function consistently performs poorly across different models due to several reasons. Firstly, the Softmax activation function produces output values between 0 and 1, leading to a loss of information and a decrease in the model's ability to learn complex patterns in the data. Additionally, the Softmax activation function is not as non-linear as other activation functions like sigmoid, tanh, and exponential, which can lead to a loss of information and a decrease in the model's ability to learn complex patterns in the data. The gradient calculation for Softmax can also be more complex and sensitive to the input values, leading to a loss of information and a decrease in the model's ability to learn complex patterns in the data (Shen et al., 2023; Szandała, 2020). Furthermore, the optimization algorithms used in the experiments, such as ADAM, RMSprop, SGD, and AdaGrad, may not be well-suited for the Softmax activation function, leading to poor performance.

The results for the accuracy of the different optimization algorithms used in this study are simply laid out from Figures 5 to 11, each giving insight into how well the models

performed with the same experimental setup. Starting with Figure 5, it shows how the ADAM optimizer performed, giving a baseline to compare against. Figure 6 goes on with RMSProp results, and Figure 7 presents the accuracy obtained using Stochastic Gradient Descent (SGD). Figure 8 then presents how AdaGrad fared, and Figure 9 goes on with results from AdaDelta. Figure 10 presents NADAM's performance, which is essentially an enhanced version of ADAM with Nesterov momentum. Lastly, Figure 11 presents the accuracy results for AdaMax, yet another infinity norm-based variant of ADAM. Collectively, these figures facilitate the comparison and visualization of the strengths of each optimizer in a consistent way. The comparison of accuracy results for each model or optimization algorithm using the experiment's current setups is shown graphically in the following.

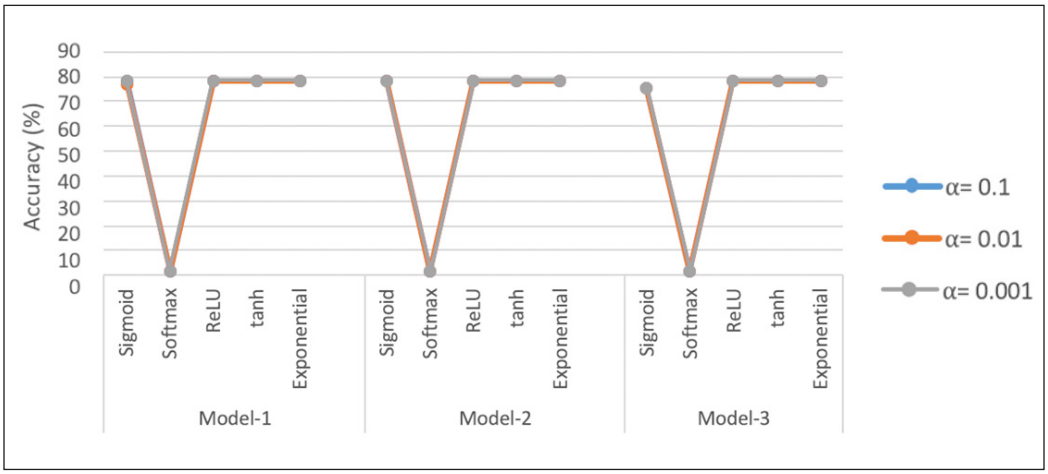

Figure 5. Accuracy result (ADAM)

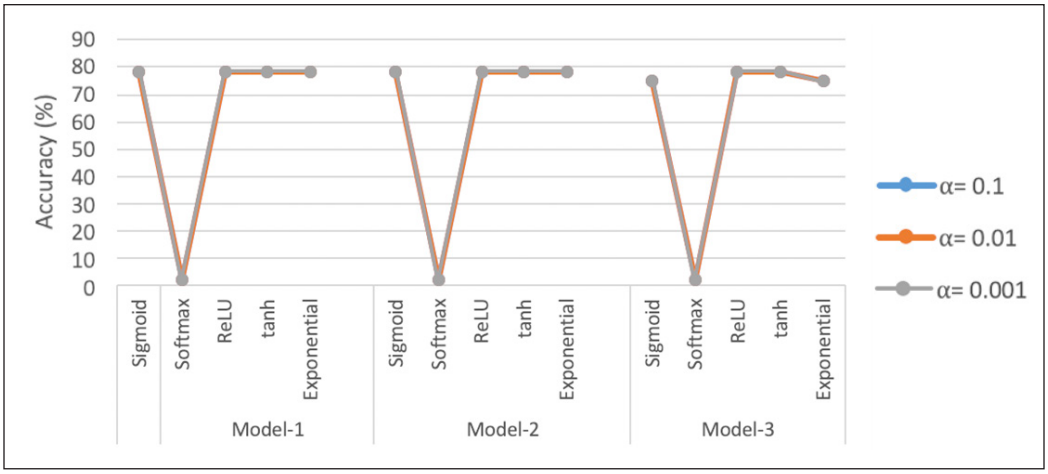

Figure 6. Accuracy result (RMSProp)

Based on the analysis of the results of the studies and experiments that have been presented, some things that can be highlights are the importance of choosing the right optimizer model and hyperparameters for a specific problem. It also emphasizes the need for further research to better understand the interactions between these factors and their impact on model performance. Identify potential future directions for research in this area, such as exploring new optimization algorithms or activation functions, or integrating optimization algorithms with other DL techniques. Furthermore, the performance of the optimization algorithms and activation functions can be compared possibly using other methods.

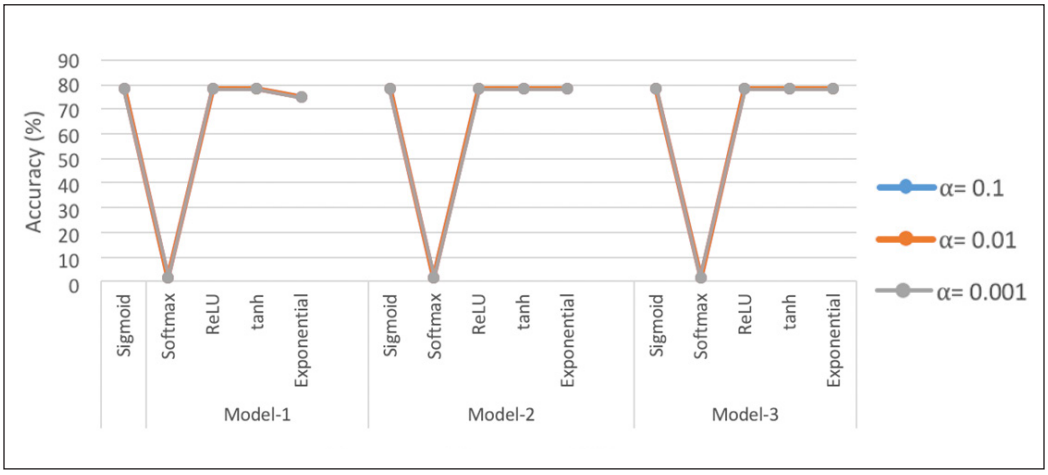

Figure 7. Accuracy result (SGD)

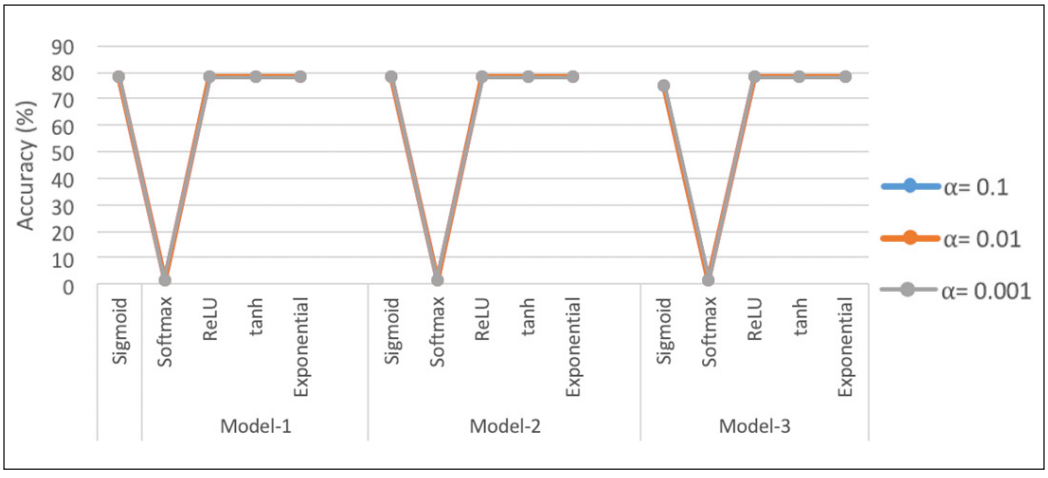

Figure 8. Accuracy result (AdaGrad)

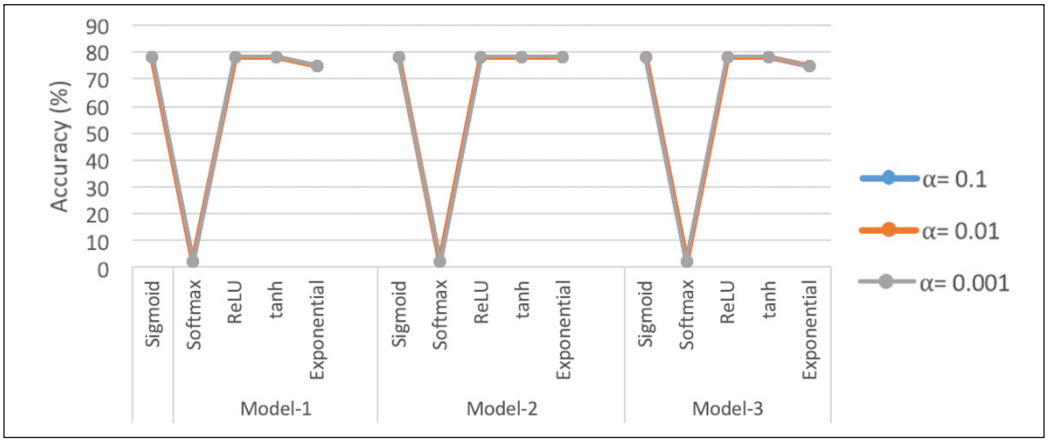

Figure 9. Accuracy result (AdaDelta)

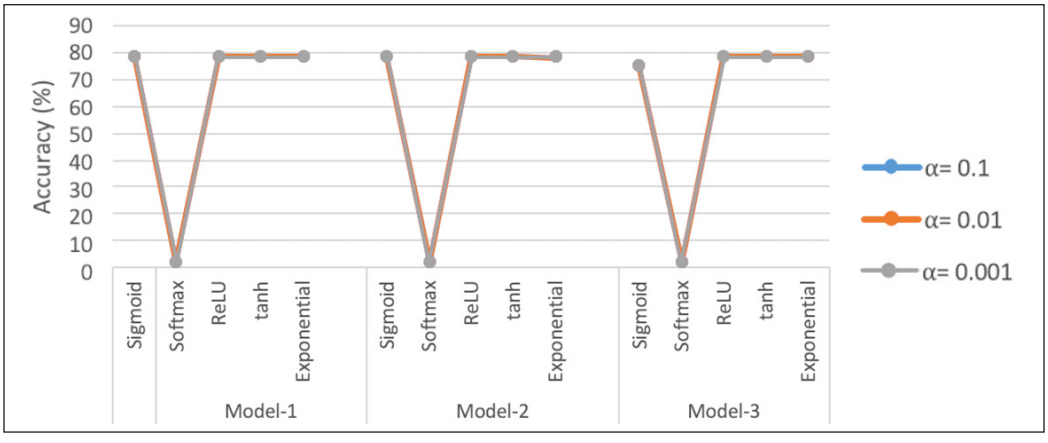

Figure10. Accuracy result (NADAM)

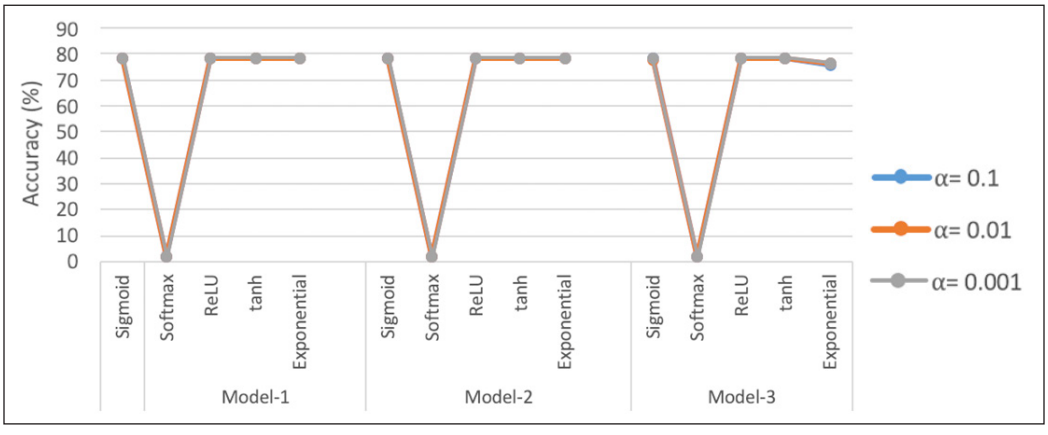

Figure 11. Accuracy result (AdaMax)

CONCLUSION

This study aims to compare the performance of seven widely used GD optimization algorithms on RNN architecture using body motion datasets from the UCI ML repository. The optimization techniques compared are ADAM, RMSprop, SGD, AdaGrad, AdaDelta, NADAM, and AdaMax. Precisely, the study is done across different combinations of optimizer models, LSTM architectures, activation functions, and learning rates. The results show that the AdaGrad model with both exponential and sigmoid activation functions and a learning rate of 0.001 has the best performance, with a training time of 17.1 minutes and a test accuracy of 78.31%, respectively.

Moreover, the study also highlights the importance of selecting the optimal optimization algorithm for RNN training given specific properties of the training data. The findings suggest that the exponential activation function consistently performs well across different models and optimizers in most cases, whereas the Softmax activation function consistently performs poorly in all cases. Finally, this study also contributes to the understanding of how different optimizers handle the challenge of determining the best and most suitable training parameters utilizing accuracy and training duration as performance metrics. The results provide valuable insights for researchers and practitioners in the field of DL and optimization.

ACKNOWLEDGMENTS

The author sincerely thanks Assoc. Prof. Ts. Dr. Khairul Najmy Rani and Assoc. Prof. Ts. Dr. Amiza Amir for all the insights and suggestions in realizing this paper.

REFERENCES

Adem, K., & Kiliçarslan, S. (2019). Performance analysis of optimization algorithms on stacked autoencoder. In *3rd International Symposium on Multidisciplinary Studies and Innovative Technologies* (pp. 1-4). IEEE. https://doi.org/10.1109/ISMSIT.2019.8932880

Alzubaidi, L., Zhang, J., Humaidi, A. J., Al-Dujaili, A., Duan, Y., Al-Shamma, O., Santamaría, J., Fadhel, M. A., Al-Amidie, M., & Farhan, L. (2021). Review of deep learning: Concepts, CNN architectures, challenges, applications, future directions. *Journal of Big Data*, *8*, 53. https://doi.org/10.1186/s40537-021-00444-8

Banos, O., Garcia, R., Holgado-Terriza, J. A., Damas, M., Pomares, H., Rojas, I., Saez, A., & Villalonga, C. (2014). mHealthDroid: A novel framework for agile development of mobile health applications. In L. Pecchia, L. L. Chen, C. Nugent, & J. Bravo (Eds.), *Ambient assisted living and daily activities* (pp. 91–98). Springer. https://doi.org/10.1007/978-3-319-13105-4_14

Chandra, K., Xie, A., Ragan-Kelley, J., & Meijer, E. (2022). Gradient descent: The ultimate optimizer. In *NIPS'22: Proceedings of the 36th International Conference on Neural Information Processing Systems* (pp. 8214-8225). Associaton for Computing Machinery.

Chen, W., Zheng, F., Gao, S., & Hu, K. (2022). An LSTM with differential structure and its application in action recognition. *Mathematical Problems in Engineering*, *2022*(1), 7316396. https://doi.org/https://doi.org/10.1155/2022/7316396

Dogo, E. M., Afolabi, O. J., Nwulu, N. I., Twala, B., & Aigbavboa, C. O. (2018). A comparative analysis of gradient descent-based optimization algorithms on convolutional neural networks. In *2018 International Conference on Computational Techniques, Electronics and Mechanical Systems* (pp. 92-99). IEEE. https://doi.org/10.1109/CTEMS.2018.8769211

Duchi, J., Hazan, E., & Singer, Y. (2011). Adaptive subgradient methods for online learning and stochastic optimization. *Journal of Machine Learning Research*, *12*, 2121-2159.

Ethem, A. (2020). *Introduction to machine learning* (4th ed.). The MIT Press.

Fatima, N. (2020). Enhancing performance of a deep neural network: A comparative analysis of optimization algorithms. *Advances in Distributed Computing and Artificial Intelligence Journal*, *9*(2), 79–90. https://doi.org/10.14201/adcaij2020927990

Goodfellow, I., Bengio, Y., & Courville, A. (2016). *Deep learning*. The MIT Press.

Haji, S. H., & Abdulazeez, A. M. (2021). Comparison of optimization techniques based on gradient descent algorithm: A review. *PalArch's Journal of Archaeology of Egypt / Egyptology*, *18*(4), 2715–2743.

Hallén, R. (2017). *A study of gradient-based algorithms*. http://lup.lub.lu.se/luur/download?func=downloadFile&recordOId=8904399&fileOId=8904400

Hinton, G., & Tieleman, T. (2012). Lecture 6.5 - Rmsprop: Divide the gradient by a running average of its recent magnitude. *Neural Networks for Machine Learning*, *4*, 26-31.

Hochreiter, S., & Schmidhuber, J. (1997). Long short-term memory. *Neural Computation*, *9*(8), 1735-1780. https://doi.org/10.1162/neco.1997.9.8.1735

Karna, V. V. R., Karna, V. R., Janamala, V., Devana, V. N. K. R., Ch, V. R. S., & Tummala, A. B. (2024). A comprehensive review on heart disease risk prediction using machine learning and deep learning algorithms. *Archives of Computational Methods in Engineering*, *32*, 1763-1795. https://doi.org/10.1007/s11831-024-10194-4

Kingma, D. P., & Ba, J. L. (2015). *ADAM: A method for stochastic optimization*. arXiv. https://doi.org/10.48550/arXiv.1412.6980

Lv, K., Jiang, S., & Li, J. (2017). *Learning gradient descent: Better generalization and longer horizons*. arXiv. https://doi.org/10.48550/arXiv.1703.03633

Mehmood, F., Ahmad, S., & Whangbo, T. K. (2023). An efficient optimization technique for training deep neural networks. *Mathematics*, *11*(6), 1360. https://doi.org/10.3390/math11061360

Miljanovic, M. (2012). Comparative analysis of recurrent and finite impulse response neural networks in time series prediction. *Indian Journal of Computer Science and Engineering*, *3*(1), 180–191.

Papamakarios, G. (2014). *Comparison of modern stochastic optimization algorithms*. University of Edinburgh.

Ruder, S. (2016). *An overview of gradient descent optimization algorithms*. arXiv. https://doi.org/10.48550/arXiv.1609.04747

Schaul, T., Antonoglou, I., & Silver, D. (2013). *Unit tests for stochastic optimization*. arXiv. https://doi.org/10.48550/arXiv.1312.6055

Seeli, D. J. J., & Thanammal, K. K. (2024). Quantitative analysis of gradient descent algorithm using scaling methods for improving the prediction process based on Artificial Neural Network. *Multimedia Tools and Applications*, *83*, 15677–15691. https://doi.org/10.1007/s11042-023-16136-9

Shalev-Shwartz, S., Shamir, O., & Shammah, S. (2017). *Failures of gradient-based deep learning*. arXiv. https://doi.org/10.48550/arXiv.1703.07950

Shen, K., Guo, J., Tan, X., Tang, S., Wang, R., & Bian, J. (2023). *A study on ReLU and Softmax in Transformer*. arXiv. https://doi.org/10.48550/arXiv.2302.06461

Sherstinsky, A. (2020). Fundamentals of Recurrent Neural Network (RNN) and Long Short-Term Memory (LSTM) network. *Physica D: Nonlinear Phenomena*, *404*, 132306. https://doi.org/10.1016/j.physd.2019.132306

Solanke, A. V., & Patnaik, G. K. (2020). Intrusion detection using deep learning approach with different optimization. *International Journal for Research in Applied Science and Engineering Technology*, *8*(5), 128–134. https://doi.org/10.22214/ijraset.2020.5022

Sun, R. (2019). *Optimization for deep learning: Theory and algorithms*. arXiv. https://doi.org/10.48550/arXiv.1912.08957

Sutskever, I., Martens, J., Dahl, G., & Hinton, G. (2013). On the importance of initialization and momentum in deep learning. *Proceedings of the 30th International Conference on Machine Learning*, *28*(3), 1139-1147.

Szandała, T. (2020). Review and comparison of commonly used activation functions for deep neural networks. In A. K. Bhoi, P. K. Mallick, C.-M. Liu, & V. E. Balas (Eds.), *Bio-inspired neurocomputinge* (Vol. 903, pp. 203-224). Springer. https://doi.org/10.1007/978-981-15-5495-7_11

Van Houdt, G., Mosquera, C., & Nápoles, G. (2020). A review on the long short-term memory model. *Artificial Intelligence Review*, *53*, 5929–5955. https://doi.org/10.1007/s10462-020-09838-1

Yazan, E., & Talu, M. F. (2017). *Comparison of the stochastic gradient descent based optimization techniques*. In *2017 International Artificial Intelligence and Data Processing Symposium (pp. 1-5). IEEE.* https://doi.org/10.1109/idap.2017.8090299

Yi, D., Ahn, J., & Ji, S. (2020). An effective optimization method for machine learning based on ADAM. *Applied Sciences*, *10*(3), 1073. https://doi.org/10.3390/app10031073

Zeiler, M. D. (2012). *ADADELTA: An adaptive learning rate method*. arXiv. https://doi.org/10.48550/arXiv.1212.5701

Zou, F., Shen, L., Jie, Z., Zhang, W., & Liu, W. (2019). A sufficient condition for convergences of ADAM and RMSProp. In *2019 Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition* (pp. 11119–11127). IEEE. https://doi.org/10.1109/CVPR.2019.01138