Indonesian Journal of Data and Science

Volume 6 Issue 1
ISSN 2715-9936
https://doi.org/10.56705/ijodas.v6i1.219

Research Article

Comparative Analysis of Gradient-Based Optimizers in Feedforward Neural Networks for Titanic Survival Prediction

I Putu Adi Pratama [1,*], Ni Wayan Jeri Kusuma Dewi [2]

[1] UHN IGB Sugriwa Denpasar, Bali, Indonesia, putuadi@uhnsugriwa.ac.id
[2] Institut Bisnis dan Teknologi Indonesia, Kota Denpasar, Bali 80225, Indonesia, wayan.kusumadewi@instiki.ac.id

Correspondence should be addressed to I Putu Adi Pratama; putuadi@uhnsugriwa.ac.id

Received 29 November 2024; Accepted 20 March 2025; Published 31 March 2025

Abstract:

Introduction: Feedforward Neural Networks (FNNs), or Multilayer Perceptrons (MLPs), are widely recognized for their capacity to model complex nonlinear relationships. This study aims to evaluate the performance of various gradient-based optimization algorithms in training FNNs for Titanic survival prediction, a binary classification task on structured tabular data. **Methods**: The Titanic dataset consisting of 891 passenger records was pre-processed via feature selection, encoding, and normalization. Three FNN architectures—small ([64, 32, 16]), medium ([128, 64, 32]), and large ([256, 128, 64])—were trained using eight gradient-based optimizers: BGD, SGD, Mini-Batch GD, NAG, Heavy Ball, Adam, RMSprop, and Nadam. Regularization techniques such as dropout and L2 penalty, along with batch normalization and Leaky ReLU activation, were applied. Training was conducted with and without a dynamic learning rate scheduler, and model performance was evaluated using accuracy, precision, recall, F1-score, and cross-entropy loss. **Results**: The Adam optimizer combined with the medium architecture achieved the highest accuracy of 82.68% and an F1-score of 0.77 when using a learning rate scheduler. RMSprop and Nadam also performed competitively. Models without learning rate schedulers generally showed reduced performance and slower convergence. Smaller architectures trained faster but yielded lower accuracy, while larger architectures offered marginal gains at the cost of computational efficiency. **Conclusions**: Adam demonstrated superior performance among the tested optimizers, especially when coupled with learning rate scheduling. These findings highlight the importance of optimizer choice and learning rate adaptation in enhancing FNN performance on tabular datasets. Future research should explore additional architectures and optimization strategies for broader generalizability.

Keywords: Binary Classification, Feedforward Neural Networks (FNNs), Gradient-based Optimisation Algorithms, Learning Rate Scheduler, Titanic Survival Prediction.
Dataset link: https://www.kaggle.com/datasets/tanlikesmath/diabetic-retinopathy-resized

1. Introduction

The Titanic dataset has long been recognised as a benchmark challenge in data science and machine learning, valued for its simplicity and historical significance. Originating from the 1912 Titanic disaster, the dataset offers structured data suitable for binary classification problems, making it ideal for testing algorithms and exploring predictive modelling techniques [1]–[3]. While numerous models, ranging from decision trees to support vector machines, have been applied to this dataset, neural networks, particularly Feedforward Neural Networks (FNNs), offer unparalleled capabilities in capturing complex, non-linear relationships within tabular data [4], [5].

Optimization in neural network training is pivotal, impacting convergence speed, model stability, and generalization. Gradient-based optimization algorithms are essential in this process, adjusting network parameters to minimize error. This includes algorithms like Batch Gradient Descent (BGD), Stochastic Gradient Descent (SGD), and more advanced methods like NAG, Heavy Ball Method, Adam, RMSprop, and Nadam, each designed to tackle different aspects of the optimization challenge [5]–[12].

Despite their extensive use in areas like computer vision and natural language processing, their comparative efficacy on structured datasets such as the Titanic survival prediction has not been thoroughly investigated until recently. Emerging research, however, is beginning to shed light on this domain, emphasizing the need for nuanced evaluations in the context of tabular data [13]–[15]. This study aims to address this gap by scrutinizing how these optimization strategies perform across different FNN architectures when confronted with the Titanic dataset.

This study seeks to address this gap by investigating the performance of various gradient-based optimisation algorithms on FNN architectures for Titanic survival prediction. Specifically, the evaluation includes three FNN architectures— [64, 32, 16], [128, 64, 32], and [256, 128, 64]—and eight optimisation strategies: BGD, SGD, Mini-Batch Gradient Descent, NAG, Heavy Ball Method, Adam, RMSprop, and Nadam. Additionally, the effect of a dynamic learning rate scheduler on training efficacy and model performance is explored.

The key contribution of this research lies in its comparative analysis of optimisers, training strategies, and FNN architectures, evaluated through multiple performance metrics including accuracy, precision, recall, F1 score, loss, and training time. By doing so, this study provides valuable insights into optimising FNNs for tabular datasets, a topic of growing importance as machine learning applications continue to expand across diverse domains.

2. Method:

This study applies machine learning techniques to predict the survival of Titanic passengers based on several features, including age, sex, class, and embarkation details. The process involves a series of key steps, from data pre-processing to model evaluation, with a focus on optimizing the model's performance using different machine learning algorithms and hyperparameters (**Figure 1**).

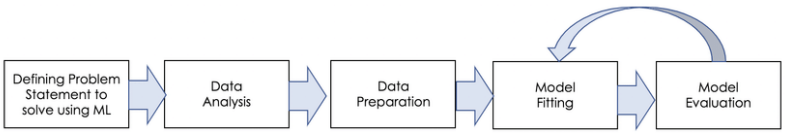

Figure 1. An overview of the process of the steps taken to create the Machine Learning models

Data Collection

This study utilizes the Titanic dataset, a widely used benchmark for binary classification tasks. Sourced from Kaggle, it comprises 891 passenger records with 12 features, including demographic data (age, sex, and class), socio-economic attributes (fare and ticket class), and familial relationships (number of siblings/spouses aboard). The target variable denotes survival status: 1 for survived and 0 for did not survive.

Data Pre-processing

Before model development, the dataset was preprocessed to ensure cleanliness and suitability for analysis. Irrelevant columns (PassengerId, Name, Ticket, and Cabin) were removed, and missing values were handled by imputing the median for continuous variables (Age and Fare) and the most frequent category ('S') for the categorical variable Embarked. Categorical features (Sex and Embarked) were one-hot encoded to enable their inclusion in the model, and all features were standardized using StandardScaler to ensure uniform scaling, improving model convergence and performance.

Model Selection

Architecture Configurations

The architecture was experimented with across three configurations:

- Small: [64, 32, 16] – a compact model with fewer neurons, providing faster training but potentially lower accuracy.
- Medium: [128, 64, 32] – a balanced configuration that serves as a middle ground between model complexity and computational cost.
- Large: [256, 128, 64] – a larger model with more neurons, better capturing complex patterns but requiring more computational resources and training time.

Activation Functions (Leaky ReLU Activation)

In this study, the Leaky ReLU activation function, as shown in Equation 1, is employed to address the issue of dying neurons by allowing a small, non-zero gradient for negative inputs [16]. This activation function improves the model's ability to learn and generalize, particularly in deeper neural networks.

$$\boldsymbol{f(x) = x, if\ x > 0;\ \alpha x, if\ x \leq 0} \quad (1)$$

Where $\boldsymbol{f(x)}$ is the output of the activation function, $\boldsymbol{x}$ is the input to activation function, and α is the scale factor for negative values of x. Typically, α has a small value such as 0.01 or 0.1.

Regularization Techniques (L2 Regularization / Weight Decay)

In this study, L2 Regularization (Weight Decay), as illustrated in Equation 2 [4], is incorporated to penalize large weights in the model, promoting simpler models and reducing the risk of overfitting. This regularization technique improves generalization by constraining the magnitude of model parameters during training.

$$\boldsymbol{L}_{\text{reg}} = \boldsymbol{L}_{\text{original}} + \frac{\boldsymbol{\lambda}}{\boldsymbol{2}} \sum_{\boldsymbol{i}} \boldsymbol{w_i^2} \quad (2)$$

Where $\boldsymbol{L}_{\text{original}}$ is the original loss function (e.g., mean squared error or cross-entropy), w_i represents the individual model weights, and λ is the regularization strength, controlling the trade-off between minimizing the original loss and penalizing large weights.

Optimization Techniques

In this study, the following optimisation methods were employed: Batch Gradient Descent (BGD), Stochastic Gradient Descent (SGD), Mini-Batch Gradient Descent, Nesterov Accelerated Gradient (NAG), Heavy Ball Method, Adam (Adaptive Moment Estimation), RMSprop, and Nadam (Nesterov-accelerated Adaptive Moment Estimation). Each optimiser offers distinct advantages, allowing for a comprehensive evaluation of their performance in training models for the Titanic survival prediction task.

Basic Gradient Descent Algorithms

The basic update rule for gradient descent is expressed as Equation 3:

$$\boldsymbol{\theta_{t+1} = \theta_t - \eta \nabla J(\theta_t)} \quad (3)$$

Where θ_t is the model parameters at iteration t, $\nabla J(\theta_t)$ is the gradiant of the loss function $J(\theta)$ with respect parameters θ at iteration t, η it the learning rate, a hyperparameter that controls the step size of aech update, and θ_{t+1} is the updated parameters after the gradient descent step.

The update rule iteratively adjust the parameters θ in the direction of the negative gradient (steepest descent) to minimise the loss function $J(\theta)$.

In this study, Batch Gradient Descent (BGD), Stochastic Gradient Descent (SGD), and Mini-Batch Gradient Descent are employed to explore their distinct approaches to updating model parameters during training. BGD computes gradients using the entire dataset, ensuring precise updates but requiring significant computational resources and memory, making it less practical for large datasets. In contrast, SGD updates parameters using a single, randomly selected data point, enabling faster iterations and lower computational costs but introducing higher variance and less stable convergence. Mini-Batch Gradient Descent strikes a balance by using small subsets (mini-batches) of the dataset, combining BGD's stability with SGD's efficiency [17]. This hybrid approach reduces gradient variance, enables faster updates, and optimizes resource usage, making it ideal for large-scale models.

Momentum-Based Algorithms

The general formula for updating parameters θ with momentum is consist of compute the velocity (Equation 4) and update the parameters (Equation 5) [17].

Compute the velocity:

$$\boldsymbol{v_t = \beta v_{t-1} - \eta \nabla J(\theta_t)} \quad (4)$$

Where $\boldsymbol{v_t}$ is the velocity term (momentum-adjusted gradient), $\boldsymbol{\beta}$ is the momentum coefficient (typically between 0.5 and 0.9), $\boldsymbol{\eta}$ is the learning rate, and $\nabla J(\theta t)$ is the gradient of the loss function at time step $\mathbf{t}$.

Update the parameter:

$$\boldsymbol{\theta_{t+1} = \theta_t + v_t} \tag{5}$$

In this study, Nesterov Accelerated Gradient (NAG) and the Heavy Ball Method are employed to enhance optimization performance. NAG improves upon the momentum method by applying the velocity term to the anticipated future parameter position, enabling more accurate gradient updates and faster convergence in complex optimization landscapes. Similarly, the Heavy Ball Method emphasizes momentum to overcome local minima and accelerate convergence, particularly in quadratic loss functions, by effectively utilizing accumulated gradients [17].

Adaptive Gradient Descent Algorithms

Adam (Adaptive Moment Estimation)

The following formulas represent the key steps in the Adam optimizer algorithm: gradient calculation (Equation 6), running average of the gradients (first moment estimate, Equation 7), running average of the squared gradients (second moment estimate, Equation 8), bias correction (Equation 9 and 10), and parameter updates (Equation 11) [17].

$$\mathbf{g_t} = \nabla_{\theta_t} \mathcal{L}(\boldsymbol{\theta_t}) \tag{6}$$

$$\mathbf{m_t} = \boldsymbol{\beta_1}\, \mathbf{m_{t-1}} + (\mathbf{1} - \boldsymbol{\beta_1})\, \mathbf{g_t} \tag{7}$$

$$\mathbf{v_t} = \boldsymbol{\beta_2}\, \mathbf{v_{t-1}} + (\mathbf{1} - \boldsymbol{\beta_2})\, \mathbf{g_t^2} \tag{8}$$

$$\widehat{\mathbf{m_t}} = \frac{\mathbf{m_t}}{\mathbf{1} - \boldsymbol{\beta_1^t}} \tag{9}$$

$$\widehat{\mathbf{V_t}} = \frac{\mathbf{v_t}}{\mathbf{1} - \boldsymbol{\beta_2^t}} \tag{10}$$

$$\boldsymbol{\theta_{t+1}} = \boldsymbol{\theta_t} - \boldsymbol{\alpha}\, \frac{\widehat{\mathbf{m_t}}}{\sqrt{\widehat{\mathbf{v_t}}} + \boldsymbol{\epsilon}} \tag{11}$$

Where:

g_t : Gradient of the loss function $\mathcal{L}$ with respect to the parameters θ_t at time step t.
m_t : First moment estimate (exponential moving average of gradients).
v_t : Second moment estimate (exponential moving average of squared gradients).
β_1 : Decay rate for the first moment estimate (typically $\beta_1 = 0.9$)
β_2 : Decay rate for the second moment estimate (typically $\beta_2 = 0.999$)
$\widehat{m_t}$: Bias-corrected first moment estimate.
$\widehat{v_t}$: Bias-corrected second moment estimate.
α : Learning rate.
ϵ : Small constant added for numerical stability (e.g., 10^{-8}).
$\theta t + 1\theta_{t+1}$: Updated parameters at time step $t + 1$.

RMSprop

The following formulas represent the key steps in the RMSprop algorithm: gradient calculation (Equation 12), running average of squared gradients (Equation 13), and parameter updates (Equation 14), with the learning rate dynamically adjusted based on the accumulated information from the gradient history [17].

$$\boldsymbol{g_t = \nabla_{\theta_t \mathcal{L}}(\theta_t)} \tag{12}$$

$$\boldsymbol{v_t = v_{t-1} + (1 - \beta) g_t^2} \tag{13}$$

$$\boldsymbol{\theta_{t+1} = \theta_t - \frac{\alpha}{\sqrt{v_t + \epsilon}} g_t} \tag{14}$$

Where:

g_t : is the gradient of the loss function with respect to the parameters θt.
v_t : is the running average of the squared gradients.

β : is the decay rate.
α : is the learning rate.
ϵ : is a small constant for numerical stability.

Nadam (Nesterov-accelerated Adaptive Moment Estimation)

The following formulas outline the key components of the Nadam optimizer algorithm: the momentum update with Nesterov acceleration (Equation 15) and the parameter update (Equation 16). Nadam extends the standard Adam optimizer by integrating Nesterov momentum, which incorporates a lookahead mechanism that anticipates the future gradient [17]. This adjustment improves the update dynamics, potentially leading to more efficient parameter updates and faster convergence, particularly in optimization tasks involving deep learning models.

$$\widehat{m_t} = \beta_1 m_{t-1} + (1 - \beta_1) g_t \tag{15}$$

$$\theta_{t+1} = \theta_t - \alpha \frac{\widehat{m_t}}{\sqrt{\widehat{v_t}} + \epsilon} \tag{16}$$

Where:

$\widehat{m_t}$: is the bias-corrected first moment estimate (momentum term).
β_1 : is the exponential decay rate for the first moment estimate (usually close to 1).
m_{t-1}: is the previous first moment estimate.
g_t : is the gradient of the loss function with respect to the parameters at time step *t*.
θ_{t+1}: is the updated parameter at time step $t + 1$.
θ_t : is the parameter at time step *t*.
α : is the learning rate.
$\widehat{m_t}$: is the bias-corrected first moment estimate.
$\widehat{v_t}$: is the bias-corrected second moment estimate.
ϵ : is a small constant added to prevent division by zero (typically 10^{-8}).

Hyperparameter Tuning

In this study, several hyperparameters and techniques were utilized to optimize model training and prevent overfitting. Various learning rates were tested to identify the optimal balance between efficient convergence and stability, as overly large rates caused instability, while small rates led to slow or suboptimal convergence. A dropout rate of 0.2 was employed to deactivate 20% of neurons during training, enhancing generalization by reducing reliance on specific features. Batch sizes of 32 were used for most experiments, balancing efficiency and stable gradient estimation, while Batch Gradient Descent used the full dataset for updates, and Mini-Batch Gradient Descent maintained a fixed size of 32. Early stopping was applied, halting training if validation loss did not improve for 10 epochs, thereby preventing overfitting and ensuring better performance on unseen data.

Training and Evaluation

a. Accuracy: Accuracy measures the proportion of correct predictions made by the model out of the total predictions [18], [19]. It provides an overall assessment of model performance, but it may not be reliable for imbalanced datasets.

$$Accuracy \frac{TP + TN}{TP + TN + FP + FN} \tag{17}$$

b. Precision: Precision evaluates the fraction of predicted positive instances that are actually true positives [20]. It is a crucial metric when the cost of false positives is high, as it reflects the model's ability to make accurate positive predictions.

$$Precision \frac{TP}{TP + FP} \tag{18}$$

c. Recall: Recall measures the fraction of actual positive instances that the model correctly identifies [18], [21]. It is particularly important when missing positive instances (false negatives) has severe consequences, as it reflects the model's sensitivity.

$$Recall \frac{TP}{TP + FN} \quad (19)$$

d. F1-Score: The F1 Score is the harmonic mean of precision and recall, offering a balanced metric when there is a trade-off between the two [22]. It is especially useful for evaluating model performance on imbalanced datasets where both precision and recall are important.

$$F1 - Score = 2. \frac{Precision\ .\ Recall}{Precision + Recall} \quad (20)$$

e. Cross-entropy Loss: Cross-entropy loss quantifies the difference between the predicted probability distribution and the true labels [23], [24]. It is commonly used for classification tasks, guiding the model to output probabilities that closely match the actual class labels.

$$L = -\frac{1}{N}\sum_{i=1}^{N}\sum_{c=1}^{C} y_{i,c} \log(p_{i,c}) \quad (21)$$

Model Comparison: With and Without Learning Rate Scheduler

The performance of various neural network models was evaluated both with and without the use of a Learning Rate Scheduler (LRS) [25]. The primary objective of this comparison was to assess the effect of dynamically adjusting the learning rate during training on model performance, particularly in terms of convergence speed and generalization.

Training Configuration

Models were trained under identical configurations, differing only by the presence or absence of the Learning Rate Scheduler. The initial learning rate for all models was set to 0.001. The Learning Rate Scheduler adjusted the learning rate as follows: 0.001 for the first 10 epochs, 0.0005 for epochs 11–20, and 0.0001 for epochs 21 and beyond.

In contrast, models without the Learning Rate Scheduler maintained a constant learning rate of 0.001 throughout the entire training process.

3. Results and Discussion

Results

The performance of various machine learning models was evaluated using different optimizers, architectures, and the inclusion or exclusion of a learning rate scheduler. The key metrics assessed include accuracy, precision, recall, F1 score, loss, and training time.

With Learning Rate Scheduler:

The models with the learning rate scheduler demonstrated improved consistency across performance metrics (Please refer to **Tables 1** through **6**). Among these, the Adam optimizer with the 128-64-32 architecture achieved the highest accuracy (82.68%), accompanied by strong precision (84.13%) and F1 score (77.37%). In contrast, the batch gradient descent (bgd) optimizer consistently underperformed, recording the lowest accuracy (41.90%) and a high loss (0.85). Notably, the RMSprop optimizer with the 256-128-64 architecture provided a balanced trade-off, achieving 81.01% accuracy and an F1 score of 75.71%. These results highlight the utility of fine-tuned learning rates for performance enhancement.

Table 1. Performance metrics for 128 - 64 – 32 architecture with learning rate scheduler

Optimizer	Accuracy	Precision	Recall	F1-Score	Loss	Training Time
adam	0.826816	0.841270	0.716216	0.773723	0.536194	7.720.988
rmsprop	0.782123	0.710843	0.797297	0.751592	0.540710	3.906.668
nadam	0.798883	0.779412	0.716216	0.746479	0.556580	4.984.274
sgd	0.648045	0.559140	0.702703	0.622754	0.747231	3.503.747
nag	0.575419	0.484848	0.432432	0.457143	0.595325	3.576.082
heavy_ball	0.743017	0.712121	0.635135	0.671429	0.604799	3.486.997

Optimizer	Accuracy	Precision	Recall	F1-Score	Loss	Training Time
b_gd	0.418994	0.412791	0.959459	0.577236	0.852602	2.833.640
mini_batch	0.564246	0.471429	0.445946	0.458333	0.771178	2.948.721

Table 2. Performance metrics for 256 - 128 – 64 architecture with learning rate scheduler

Optimizer	Accuracy	Precision	Recall	F1-Score	Loss	Training Time
adam	0.798883	0.771429	0.729730	0.750000	0.658257	4.484.004
rmsprop	0.810056	0.803030	0.716216	0.757143	0.653955	3.786.197
nadam	0.810056	0.794118	0.729730	0.760563	0.663502	5.161.725
sgd	0.581006	0.480000	0.162162	0.242424	0.841467	3.124.715
nag	0.715084	0.645570	0.689189	0.666667	0.726186	3.591.099
heavy_ball	0.715084	0.641975	0.702703	0.670968	0.722435	3.702.605
b_gd	0.407821	0.257576	0.229730	0.242857	0.970837	2.144.826
mini_batch	0.575419	0.472222	0.229730	0.309091	0.859582	3.072.036

Table 3. Performance metrics for 64 - 32 - 16 architecture with learning rate scheduler

Optimizer	Accuracy	Precision	Recall	F1-Score	Loss	Training Time
adam	0.815642	0.815385	0.716216	0.762590	0.496800	9.711.076
rmsprop	0.793296	0.703297	0.864865	0.775758	0.505699	4.765.330
nadam	0.726257	0.658228	0.702703	0.679739	0.510001	4.931.016
sgd	0.525140	0.420290	0.391892	0.405594	0.735625	2.880.930
nag	0.553073	0.453125	0.391892	0.420290	0.574668	3.483.740
heavy_ball	0.692737	0.787879	0.351351	0.485981	0.618826	3.547.900
b_gd	0.435754	0.357895	0.459459	0.402367	0.786849	2.136.626
mini_batch	0.469274	0.379310	0.445946	0.409938	0.696302	3.019.075

Without Learning Rate Scheduler:

In the absence of a learning rate scheduler, performance generally declined, with accuracy and F1 scores dropping for most configurations. The RMSprop optimizer with the 256-128-64 architecture performed best, achieving 81.01% accuracy and a competitive F1 score of 75.00%. However, without the scheduler, training times were slightly reduced across all models. Interestingly, models such as heavy ball and Nadam showed resilience with moderate metrics, indicating their adaptability even without learning rate modulation.

Table 4. Performance metrics for 128 - 64 - 32 architecture without learning rate scheduler

Optimizer	Accuracy	Precision	Recall	F1-Score	Loss	Training Time
adam	0.765363	0.686047	0.797297	0.737500	0.557728	4.464.359
rmsprop	0.804469	0.774648	0.743243	0.758621	0.552221	3.603.284
nadam	0.770950	0.698795	0.783784	0.738854	0.558508	5.010.076
sgd	0.463687	0.401786	0.608108	0.483871	0.715697	2.988.414
nag	0.731844	0.671053	0.689189	0.680000	0.602635	3.469.756
heavy_ball	0.698324	0.666667	0.540541	0.597015	0.596084	3.437.803
b_gd	0.424581	0.418079	1.000.000	0.589641	0.834218	2.202.016
mini_batch	0.407821	0.375000	0.648649	0.475248	0.764953	2.966.333

Table 5. Performance metrics for 256 - 128 - 64 architecture without learning rate scheduler

Optimizer	Accuracy	Precision	Recall	F1-Score	Loss	Training Time
adam	0.798883	0.806452	0.675676	0.735294	0.676148	4.449.394
rmsprop	0.810056	0.822581	0.689189	0.750000	0.666917	4.573.398

Optimizer	Accuracy	Precision	Recall	F1-Score	Loss	Training Time
nadam	0.810056	0.785714	0.743243	0.763889	0.659236	5.134.668
sgd	0.368715	0.365517	0.716216	0.484018	0.844684	3.035.601
nag	0.737430	0.642105	0.824324	0.721893	0.712914	3.640.618
heavy_ball	0.759777	0.712329	0.702703	0.707483	0.707158	3.597.630
b_gd	0.497207	0.136364	0.040541	0.062500	0.967988	2.100.201
mini_batch	0.441341	0.401515	0.716216	0.514563	0.877617	3.029.622

Table 6. Performance metrics for 64 - 32 - 16 architecture without learning rate scheduler

Optimizer	Accuracy	Precision	Recall	F1-Score	Loss	Training Time
adam	0.765363	0.735294	0.675676	0.704225	0.512029	4.192.075
rmsprop	0.798883	0.763889	0.743243	0.753425	0.503299	3.598.486
nadam	0.681564	0.585859	0.783784	0.670520	0.511419	4.989.890
sgd	0.430168	0.367925	0.527027	0.433333	0.722079	2.967.985
nag	0.664804	0.567308	0.797297	0.662921	0.575418	3.448.653
heavy_ball	0.687151	0.586538	0.824324	0.685393	0.555722	3.372.678
b_gd	0.586592	0.000000	0.000000	0.000000	0.814174	2.131.908
mini_batch	0.614525	0.545455	0.405405	0.465116	0.700288	2.883.502

Discussion

The results from the comparative evaluation of optimizers, network architectures, and the influence of a learning rate scheduler reveal critical insights into model performance. The discussion is structured to analyze the impact of the learning rate scheduler, the efficacy of different optimizers, and the role of network architectures on key performance metrics, including accuracy, precision, recall, F1 score, and loss.

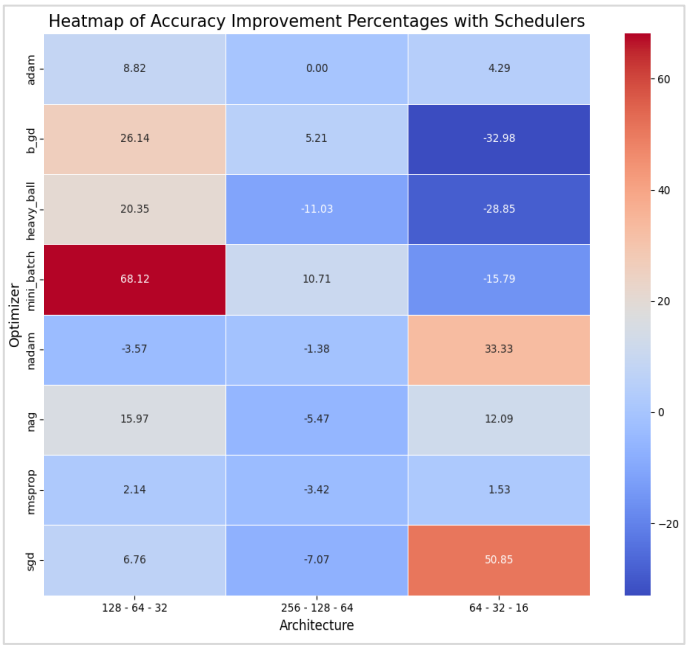

Figure 2. Heatmap of Accuracy Improvement Percentages with Schedulers

The heatmap in Figure 2 illustrates the accuracy improvement percentages when learning rate schedulers are applied. Key observations include:

- Mini-batch Gradient Descent with the 128 - 64 - 32 architecture shows the highest improvement at 68.12%, indicating a substantial benefit from the scheduler.
- Batch Gradient Descent with the 64 - 32 - 16 architecture experiences the largest decrease at -32.98%, suggesting that the scheduler was detrimental in this scenario.

- Adam optimizer generally exhibits positive improvements across all architectures, with the 128 - 64 - 32 architecture showing an improvement of 8.82%.
- Nadam with the 64 - 32 - 16 architecture shows a notable increase of 33.33%, highlighting the efficacy of the scheduler in this configuration.
- SGD presents mixed results, with a significant improvement of 50.85% for 64 - 32 - 16 but a decrease for 256 - 128 - 64.

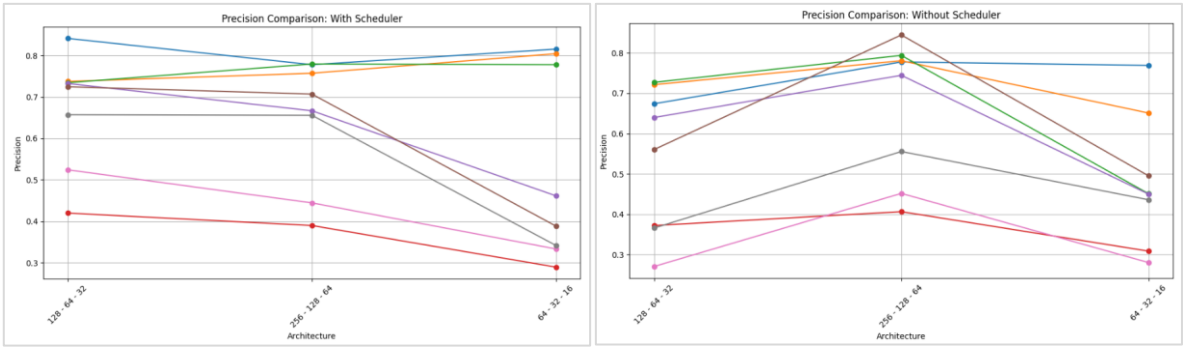

Figure 3. Precision comparison with and without learning rate scheduler

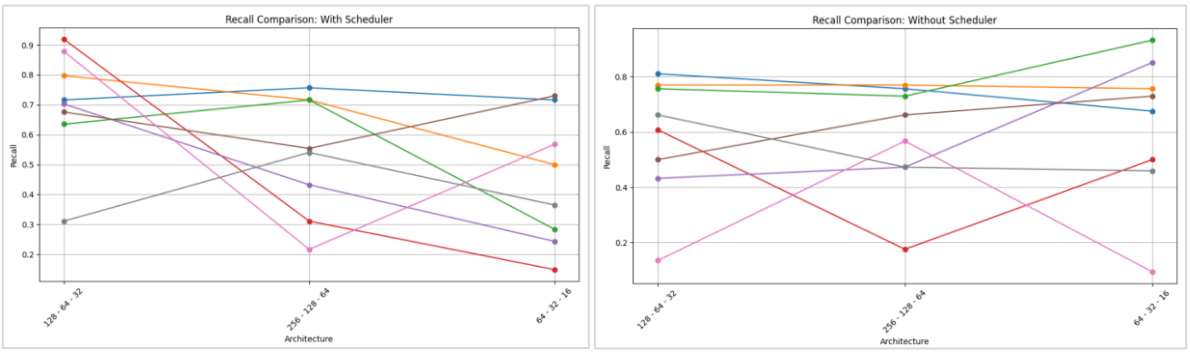

Figure 4. Recall comparison with and without learning rate scheduler

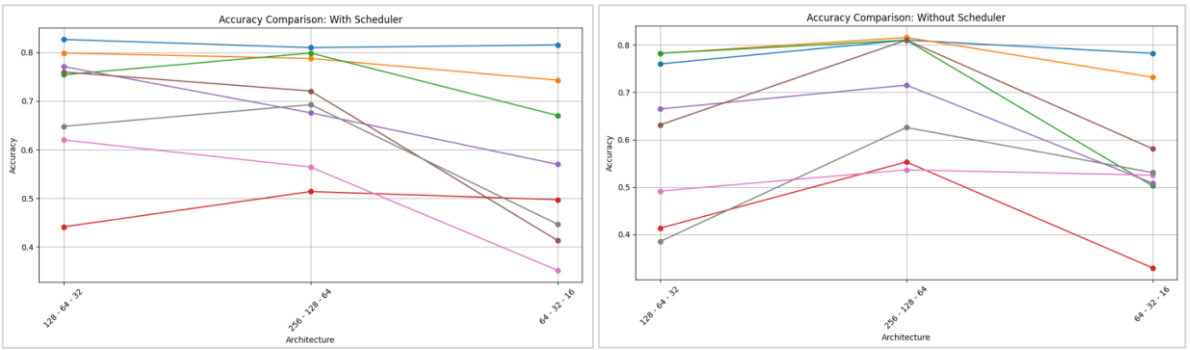

Figure 5. Recall comparison with and without learning rate scheduler

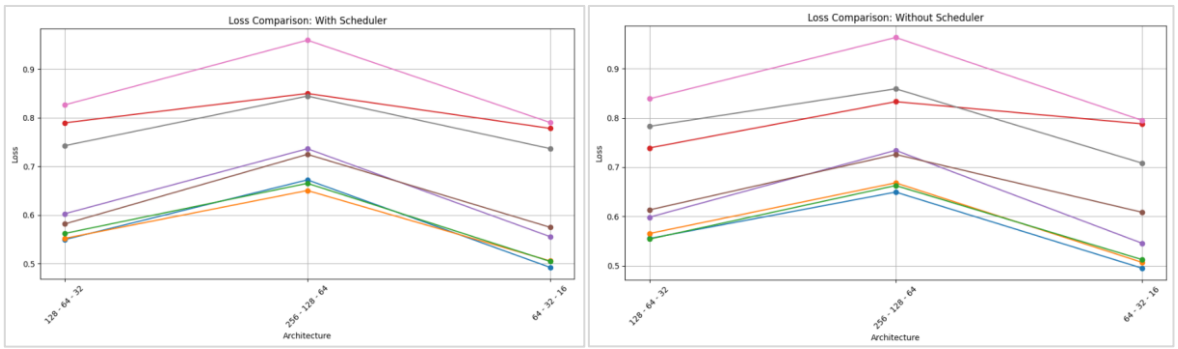

Figure 6. Loss comparison with and without learning rate scheduler

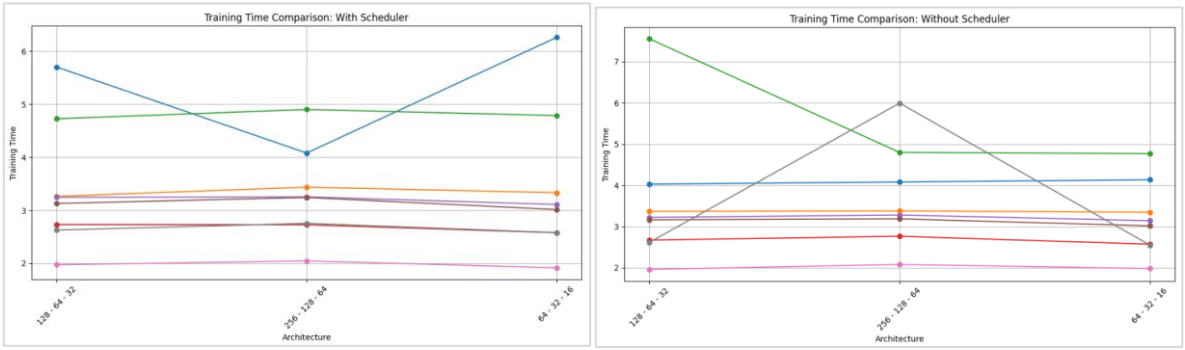

Figure 7. Training time comparison with and without learning rate scheduler

Impact of Learning Rate Scheduler

The integration of a learning rate scheduler consistently enhanced the performance of most optimizers across various architectures (Please refer to Figure 1 through 7). For example, when combined with the Adam optimizer, the learning rate scheduler led to a notable improvement in accuracy, achieving 82.68% with the 128-64-32 architecture, compared to 76.54% without the scheduler. Additionally, other performance metrics, such as F1 score and recall, exhibited significant improvements with the use of the scheduler.

These findings underscore the importance of learning rate schedulers in optimizing model convergence and mitigating overfitting. Specifically, for optimizers such as Adam and RMSprop, the learning rate scheduler plays a crucial role in fine-tuning the learning rate throughout training, leading to more efficient convergence and improved generalization. The results suggest that the dynamic adjustment of learning rates, as facilitated by schedulers, is particularly beneficial in deep learning tasks where adaptive optimization strategies are essential.

Optimizer Performance

Among the optimizers evaluated, Adam demonstrated the most robust performance across all architectures, consistently achieving a favorable balance between accuracy, precision, recall, and F1 score. For example, with the 256-128-64 architecture, Adam achieved an accuracy of 81.56% and an F1 score of 76.25% when combined with the learning rate scheduler, highlighting its ability to optimize model performance effectively.

RMSprop also performed strongly, often approaching or even matching Adam's performance, particularly in smaller architectures such as 64-32-16. In this configuration, RMSprop achieved an accuracy of 79.33% and an F1 score of 77.58%, demonstrating its effectiveness in certain settings where Adam's performance was comparable.

In contrast, gradient descent-based optimizers, such as Basic Gradient Descent (BGD) and Stochastic Gradient Descent (SGD), exhibited significantly lower performance metrics across all tested architectures. These results suggest that modern optimizers, particularly Adam and RMSprop, are better equipped to handle the complexities of contemporary deep learning models, which involve dynamic learning rate adjustments and more intricate loss landscapes.

Influence of Network Architecture

The size and complexity of the network architecture significantly impacted model performance. In general, medium-sized architectures, such as the 128-64-32 configuration, demonstrated the best balance between computational efficiency and accuracy, outperforming both smaller (64-32-16) and larger (256-128-64) architectures. While larger architectures, such as the 256-128-64 configuration, achieved comparable accuracy, they incurred notably longer training times. For instance, when using the RMSprop optimizer, the 256-128-64 architecture required an average of 3.78 seconds per epoch, whereas the smaller 64-32-16 architecture required only 2.88 seconds per epoch.

These results highlight the importance of finding an optimal balance between architectural complexity and computational cost. While deeper and more complex architectures may provide greater representational power, they come with the trade-off of increased computational demands, which can impact training efficiency and overall model performance. Future research may explore strategies for optimizing network architecture to achieve a better balance between accuracy and efficiency.

General Observations on Loss and Training Time

The observed loss values across the experiments were consistent with the performance metrics, showing a clear correlation between lower loss values and higher accuracy as well as F1 scores. Specifically, the Adam and RMSprop optimizers demonstrated superior performance, consistently yielding lower loss values compared to the more traditional Stochastic Gradient Descent (SGD) and Batch Gradient Descent (BGD) optimizers. This aligns with the known efficiency of adaptive learning rate methods in minimizing the loss function.

Training time varied significantly across different model configurations, with more complex network architectures and optimizers, such as Adam, requiring longer training durations. For example, when using the 128-64-32 architecture with Adam and a learning rate scheduler, the training time was 7.72 seconds per epoch. This indicates that while optimizers like Adam may offer superior performance, they often come at the cost of increased computational expense, reflecting the trade-off between model performance and training efficiency.

Limitations and Future Work

Although the current study offers valuable insights into the performance of different optimizers and architectures, it is limited in scope by the specific set of optimizers and network architectures evaluated. Future research could broaden this scope by exploring additional optimization algorithms, such as newer variants or hybrid approaches, to further understand their impact on model performance.

Furthermore, alternative learning rate schedulers, activation functions, and regularization techniques were not fully explored in this study. Investigating the effects of these factors could provide deeper insights into optimizing model training and improving generalization.

Expanding the evaluation to include a wider variety of datasets and tasks, particularly those with different data characteristics or from other domains, would facilitate the generalization of the findings and contribute to a more comprehensive understanding of the factors influencing model performance across different contexts

4. Conclusion

We have developed and evaluated a novel model for Titanic survival prediction that employs a structured neural network architecture with varying configurations and optimizers. The model was tested on the Kaggle Titanic dataset, which comprises 891 records with 12 demographic, socio-economic, and familial relationship features. Using a systematic evaluation framework, we compared model performance across architectures and optimizers, both with and without learning rate schedulers.

Our results demonstrate that models using the Adam optimizer consistently achieve higher accuracy, with the best-performing configuration (128-64-32 architecture, Adam, with learning rate scheduler) achieving an accuracy of 82.68%. This configuration outperformed others in terms of precision (84.13%), recall (71.62%), and F1-score (77.37%). The integration of learning rate schedulers was found to enhance overall performance metrics, particularly for optimizers like Adam and RMSprop, underscoring their efficacy in stabilising training and improving convergence.

Additionally, our experiments reveal that deeper architectures (e.g., 256-128-64) do not necessarily yield better performance without careful tuning of hyperparameters such as learning rates and optimizers. Lightweight configurations (e.g., 64-32-16) showed competitive results, indicating the potential for resource-efficient implementations.

Future work will focus on incorporating feature engineering techniques, including interaction terms and non-linear transformations, to enhance model interpretability and predictive power. We also plan to experiment with ensemble methods and more advanced neural network architectures, such as attention-based models, to further

improve survival prediction. By extending our work to other datasets, we aim to generalize our findings and validate the robustness of our proposed methodologies.

References:

[1] Y. Ai, "Predicting Titanic Survivors by Using Machine Learning," *Highlights in Science, Engineering and Technology*, vol. 34, pp. 360–367, 2023.

[2] M Yasser H, "Titanic Dataset."

[3] N. K. Pandey and A. Jain, "Predicting survival on titanic through exploratory data analysis and logistic regression: Unraveling historical patterns and insights," *The Pharma Innovation Journal*, vol. 8, no. 4, pp. 30–37, Jan. 2019.

[4] I. Goodfellow, Y. Bengio, and A. Courville, *Deep Learning*. MIT Press, 2016. Available: http://www.deeplearningbook.org

[5] I. Sutskever, J. Martens, G. Dahl, and G. Hinton, "On the importance of initialization and momentum in deep learning," in *Proceedings of the 30th International Conference on Machine Learning*, 2013.

[6] D. P. Kingma and J. Ba, "Adam: A Method for Stochastic Optimization," in *International Conference on Learning Representations (ICLR)*, Dec. 2015. Doi: 10.48550/arXiv.1412.6980

[7] S. Ruder, "An overview of gradient descent optimization algorithms," Sep. 2016. doi: 10.48550/arXiv.1609.04747

[8] Y. Bengio, "Practical recommendations for gradient-based training of deep architectures," in *Neural Networks: Tricks of the Trade*, 2012, pp. 437–478. doi: 10.48550/arXiv.1206.5533

[9] S. Y. Zhao, C. W. Shi, Y. P. Xie, and W. J. Li, "Stochastic normalized gradient descent with momentum for large-batch training," *Science China Information Sciences*, vol. 67, no. 11, pp. 1–15, Nov. 2024, doi: 10.1007/s11432-022-3892-8.

[10] P. Gou and J. Yu, "A nonlinear ANN equalizer with mini-batch gradient descent in 40Gbaud PAM-8 IM/DD system," *Optical Fiber Technology*, vol. 46, pp. 113–117, Dec. 2018, doi: 10.1016/j.yofte.2018.09.015.

[11] I. Dagal, K. Tanriöven, A. Nayir, and B. Akın, "Adaptive Stochastic Gradient Descent (SGD) for Erratic Datasets," *Future Generation Computer Systems*, Dec. 2024, doi: 10.1016/j.future.2024.107682.

[12] M. S. Shamaee, S. F. Hafshejani, and Z. Saeidian, "New logarithmic step size for stochastic gradient descent," *Front Comput Sci*, vol. 19, no. 1, pp. 1–10, Jan. 2025, doi: 10.1007/s11704-023-3245-z.

[13] V. Borisov, T. Leemann, K. Sebler, J. Haug, M. Pawelczyk, and G. Kasneci, "Deep Neural Networks and Tabular Data: A Survey," *IEEE Trans Neural Netw Learn Syst*, vol. 35, no. 6, pp. 7499–7519, Jun. 2024, doi: 10.1109/TNNLS.2022.3229161.

[14] K.-Y. Chen, P.-H. Chiang, H.-R. Chou, T.-W. Chen, and T.-H. Chang, "Trompt: Towards a Better Deep Neural Network for Tabular Data," in *Proceedings of the 40th International Conference on Machine Learning*, Jul. 2023, pp. 4392–4434.

[15] A. S. Nazdryukhin, A. M. Fedrak, and N. A. Radeev, "Neural networks for classification problem on tabular data," in *Journal of Physics: Conference Series*, IOP Publishing Ltd, Dec. 2021, pp. 1–8. doi: 10.1088/1742-6596/2142/1/012013.

[16] A. L. Maas, A. Y. Hannun, and A. Y. Ng, "Rectifier Nonlinearities Improve Neural Network Acoustic Models," in *Proceedings of the 30th International Conference on Machine Learning*, 2013.

[17] A. Géron, *Hands-On Machine Learning with Scikit-Learn, Keras & TensorFlow*, 2nd Edition. O'Reilly, 2017.

[18] M. A. Arshed, "Multi-Class Skin Cancer Classification Using Vision Transformer Networks and Convolutional Neural Network-Based Pre-Trained Models," *Inf.*, vol. 14, no. 7, 2023, doi: 10.3390/info14070415.

[19] S. Deshmukh, "Skin Cancer Classification Using CNN," *International Conference on Applied Intelligence and Sustainable Computing, ICAISC 2023*. 2023, doi: 10.1109/ICAISC58445.2023.10199440.

[20] R. Bharti, "Comparative Analysis of Potato Leaf Disease Classification Using CNN and ResNet50," *2024 International Conference on Data Science and Its Applications, ICoDSA 2024*. pp. 87–91, 2024, doi: 10.1109/ICoDSA62899.2024.10651649.

[21] G. B. Alghanimi, "CNN and ResNet50 Model Design for Improved Ultrasound Thyroid Nodules Detection," *2024 ASU International Conference in Emerging Technologies for Sustainability and Intelligent Systems, ICETSIS 2024*. pp. 1000–1004, 2024, doi: 10.1109/ICETSIS61505.2024.10459588.

[22] M. Singla, "ResNet50 Utilization for Bag Classification: A CNN Model Visualization Approach in Deep Learning," *2024 IEEE International Conference on Information Technology, Electronics and Intelligent Communication Systems, ICITEICS 2024*. 2024, doi: 10.1109/ICITEICS61368.2024.10624847.

[23] J. Girija, "Innovative Precision Medicine: An Explainable AI- Driven Biomarker-Guided Recommendation System with Multilayer FeedForward Neural Network Model," Lecture Notes in Networks and Systems, vol. 1071. pp. 438–447, 2024, doi: 10.1007/978-3-031-66410-6_35.

[24] H. Li, "Research on music signal feature recognition and reproduction technology based on multilayer feedforward neural network," Appl. Math. Nonlinear Sci., vol. 9, no. 1, 2024, doi: 10.2478/amns.2023.2.00647.

[25] V. Chandrabanshi, "A novel framework using 3D-CNN and BiLSTM model with dynamic learning rate scheduler for visual speech recognition," Signal, Image Video Process., vol. 18, no. 6, pp. 5433–5448, 2024, doi: 10.1007/s11760-024-03245-7.