PRELIMINARY NOTE

The following paper represents work completed for Project 1 as part of the course MATH 796: Machine Learning and Optimization (Fall 2025) at the University of Kansas under the supervision of Professor Erik Van Vleck. It is being submitted as a writing sample to demonstrate research methodology, technical analysis, and scientific communication skills.

During the course of this project and its subsequent extension within the class, several implementation limitations were identified and addressed. Specifically:

1. **Binary Classification Loss Function:** The original implementation used Mean Squared Error (MSE) for binary classification tasks (Higham Toy dataset). While this enabled valid comparative analysis under controlled conditions, MSE is theoretically suboptimal for probabilistic classification as it does not capture log-likelihood and can produce non-convex loss surfaces. This was corrected to Binary Cross-Entropy in the improved version.
2. **BFGS Implementation:** Computational constraints as well as implementation limitations restricted the initial use of BFGS in the project to shallow networks only. The implementation was subsequently enhanced for improved stability and broader applicability.
3. **Project Extension:** The framework with its improvements was expanded into **OptimizerArena**, a comprehensive platform featuring PCA-based loss landscape visualization, optimizer trajectory analysis, and enhanced optimization capabilities.

The results and analysis presented in this paper reflect the original implementation, ensuring all comparative evaluations were conducted under identical experimental conditions. The identified limitations do not invalidate the optimizer comparison analysis but rather represent natural evolution in understanding and implementation that occurred during the learning process.

Comparative Study of Learning-Rate Optimization Techniques and Network Depth in Neural Networks

Malek Kchaou
Computer Science Senior with Math Minor
The University of Kansas

(Submitted as part of MATH 796 – Machine Learning and Optimization, Fall 2025 with Prof. Erik Van Vleck)

Abstract

This paper presents a comparative analysis of classical and adaptive learning-rate optimization techniques applied to both shallow and deep neural network architectures. Building upon the framework proposed by Higham and Higham (2019), we evaluate the convergence, stability and generalization behavior of Gradient Descent methods, BFGS, Adam, Adagrad and RMSProp, both through custom NumPy implementations and Tensorflow's built-in optimizers through the Keras API. Using synthetic, Higham Toy, and Kaggle's Wall-Following datasets, the study aims to demonstrate that adaptive optimizers achieve faster and more stable convergence, particularly in deep architectures. Results also highlight the computational limitations second-order methods and the scalability advantage of adaptive techniques for real-world machine learning applications.

1. Motivation

Optimization lies at the heart of deep learning, governing how network parameters are iteratively updated to minimize loss functions and achieve generalization. The effectiveness of an optimizer directly influences both the convergence speed and the eventual generalization performance of a model. As the field of deep learning continues to expand exponentially, it becomes increasingly crucial to investigate in depth how different optimization algorithms behave under diverse training paradigms. Such investigation offers valuable insight into the trade-offs between convergence efficiency, numerical stability, and computational cost.

Classical methods such as Gradient Descent (GD), under its various formulations including full-batch, mini-batch, and stochastic variants, and Quasi-Newton methods such as BFGS provide strong theoretical guarantees and interpretability. However, they often struggle in high-dimensional and highly non-convex optimization landscapes typical of modern neural networks. In contrast, adaptive methods such as Adam, RMSProp, and Adagrad dynamically adjust learning rates based on gradient statistics, making them more resilient to noisy data and complex loss surfaces.

This study investigates several first-order and second-order optimization algorithms implemented both from scratch in NumPy and through TensorFlow's built-in Keras Optimizers API. The primary goal is to analyze how these different optimizers perform when training shallow and deep feedforward neural networks on a range of regression and classification problems.

The project is divided into two complementary subprojects:

- **Sub-Project 1:** Analyze and compare various learning-rate techniques and optimization methods across shallow and deep architectures using fully custom NumPy-based implementations.
- **Sub-Project 2:** Explore TensorFlow's built-in optimizers via the Keras API, validating the consistency, robustness, and efficiency of framework-level implementations relative to the custom, from-scratch approaches.

By combining theoretical insights with empirical evaluation across synthetic, benchmark, and real-world datasets, this project provides a holistic perspective on how the choice of optimization method and network depth jointly influence the training dynamics and the resulting performance of neural networks.

2. Methodology

This section outlines the architectural setup, datasets, optimization methods, and experimental configuration used in this study. Both custom (NumPy-based) and TensorFlow-based models were designed to ensure structural and hyperparameter consistency across frameworks, enabling a fair empirical comparison of optimization behavior.

2.1. Neural Network Architecture

A modular feedforward neural network was first developed entirely from scratch in NumPy, where all computations, including forward propagation, backpropagation, and parameter updates, were explicitly implemented. Hence, TensorFlow's built-in optimizers could not be directly applied to the custom NumPy-based network because TensorFlow operates on symbolic variables (tf.Variable) and computes gradients via automatic differentiation within its computation graph. Hence, to ensure comparability and validation, an identical architecture was mirrored in TensorFlow via the Keras API, using the same layer configurations, activation

functions, and loss formulations. This dual implementation allowed for a one-to-one comparison between custom optimization algorithms and TensorFlow's built-in optimizers under identical experimental conditions.
Each used model consisted of fully connected dense layers, non-linear activations and a configurable loss function. Two primary architectures were considered:

- **Shallow Network:** a single hidden layer consisting of 16 neurons.
- **Deep Network:** three hidden layers consisting of 64, 32 and 16 neurons respectively (64→32→16).

Hidden layers employed the tanh activation function chosen for its smooth gradient properties and its symmetry around zero. On the other hand, the output layer's activation function varied depending on the type of task at hand:

- For **regression**, a linear output activation was used with the Mean Squared Error (MSE) loss.
- For **classification**, a sigmoid activation was used for binary tasks with Mean Squared Error (MSE) loss, while softmax activation was used for multi-class tasks with Cross-Entropy loss.

Weight initialization followed **Xavier** initialization for tanh and sigmoid activations whereas bias terms were initialized to zero to ensure unbiased early learning behavior.

This consistent network design allows for reproducibility and for an isolated examination of optimizer performance under the two different architectures.

Throughout this paper, we refer to experimental configurations using the notation ***[Dataset] [Architecture]***. For instance, "Synthetic Deep" denotes the deep-network configuration on the Synthetic dataset, whereas "Higham Shallow" denotes the shallow-network configuration on the Higham toy dataset.

2.2. Datasets

Three different datasets of increasing complexity were picked to evaluate optimizers across regression and classification scenarios:

Dataset	Type	Task	Input Dim	Output Dim	Description
Synthetic Function	Regression	Continuous function fitting	1	1	$y = 2x + sin(x) + 0.3 * epsilon$ smooth function with Gaussian noise
Higham Toy Dataset	Binary Classification	Nonlinear decision boundary	2	1	Simple 2D dataset introduced by Higham & Higham (2019); interpretable yet nonlinear

Wall-Following Robot Sensors	Multi-Class Classification	Robotic navigation task	24	4	Real-world dataset from Kaggle where a robot learns wall-following behavior via sensor readings

The synthetic dataset served as a controlled environment for observing convergence trends, the Higham Toy dataset provided a simple non-convex classification setting, and the robot dataset introduced high-dimensional, noisy real-world data—ideal for testing optimizer robustness.

2.3. Optimization Algorithms

The selected optimizers span both first-order and second-order families, allowing an extensive comparison between simple gradient-based updates and curvature-informed quasi-Newton techniques. While Gradient Descent and its stochastic variants offer interpretability and stability under convex objectives, adaptive methods such as Adam and RMSProp exhibit resilience in high-dimensional and noisy regimes. Conversely, BFGS leverages curvature information and a line search mechanism satisfying Wolfe conditions, making it suitable for smooth, low-dimensional problems where precision is critical.

Optimizer	Type	Direction Update	Weight Update Equation	Step Size / Adaptivity Strategy	Key Advantages
Gradient Descent (GD)	First-order	$p_t = -\nabla f(\theta_t)$	$\theta_{t+1} = \theta_t + \eta p_t$	Fixed learning rate (η)	Simple, interpretable, baseline method
Mini-Batch / SGD	First-order stochastic	$p_t = -\nabla f(\theta_t; x_i)$	$\theta_{t+1} = \theta_t + \eta p_t$	Fixed learning rate (η) + Random samples or mini-batches	Faster convergence, good generalization
Adagrad	Adaptive first-order	Accumulate gradient squares	$\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{G_t + \epsilon}} g_t$	Learning rate scales inversely with gradient magnitude	Good for sparse data
RMSProp	Adaptive first-order	Exponential moving average of squared gradients	$\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{E[g^2]_t + \epsilon}} g_t$	Adaptive per-parameter rate	Stabilizes learning on non-stationary data
Adam	Adaptive first-order	Momentum + variance estimates	$\theta_t = \theta_{t-1} - \eta \frac{\hat{m}_t}{\sqrt{\hat{v}_t} + \epsilon}$	Adaptive learning rates with bias correction	Robust, widely used, efficient

BFGS	Quasi-Newton (second-order)	$p_k = -H_k \nabla f(x_k)$	$x_{k+1} = x_k + \alpha_k p_k$	Line search with Wolfe curvature condition	Fast convergence near optimum, curvature-aware

2.4. Training Configuration

Each network was trained using K-fold cross-validation (with K = 5) to evaluate both training and generalization performance.
Hyperparameters were kept fixed across all runs to isolate the effects of the optimization algorithm:

Parameter	Symbol	Value
Learning Rate	η	0.01
Epochs	—	300
Batch Size	—	Variable (full, 20 for mini-batch GD, or 1 for SGD)
Convergence Tolerance	ϵ	10^{-5}
Line Search (BFGS)	—	Wolfe conditions (Armijo + curvature)

2.5. Evaluation Metrics

For each training fold and at each epoch, the training losses were recorded into a loss history array (loss_history) to track the convergence of each optimizer across the different training folds. One final training loss as well as a validation loss were computed at the end of each iteration of the K-Fold cross-validation process. For classification problems, per-fold **metrics (accuracy for classification and MSE for regression)** were computed as an additional evaluation component. Hence, performance was assessed primarily using the following metrics:

- **Average Validation Loss:** primary indicator of generalization quality.
- **Standard Deviation of Validation Loss:** measures optimizer stability across folds.
- **Average Training Loss:** monitors convergence behavior.
- **Classification Accuracy:** evaluates predictive correctness of classification models (higher means better performance)
- **Mean Squared Error (MSE):** evaluates predictive correctness of regression models (lower means better performance)

Comparisons were then made both across optimizers and across framework implementations (custom vs. TensorFlow's Keras) to evaluate consistency and computational trade-offs relative to the given regression and classification problems.

2.6. Implementation Environment

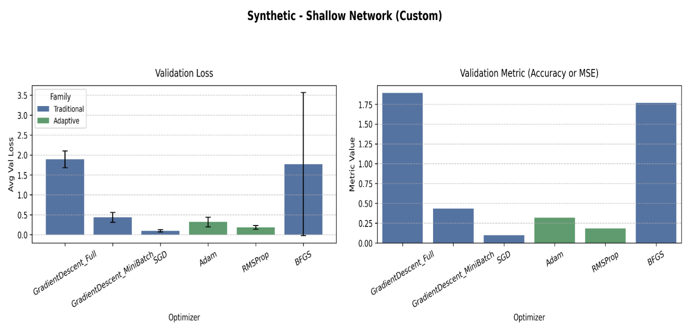
All experiments were conducted in a CPU based environment using python 3.10, NumPy 2.0.2 and Tensorflow 2.18.0. Random seeds were fixed during experiments for reproducibility and training was automated to iterate through all datasets, network architecture and optimizers exporting metrics into a unified CSV file (optimizers_comparison_summary.csv) for subsequent analysis and visualization.

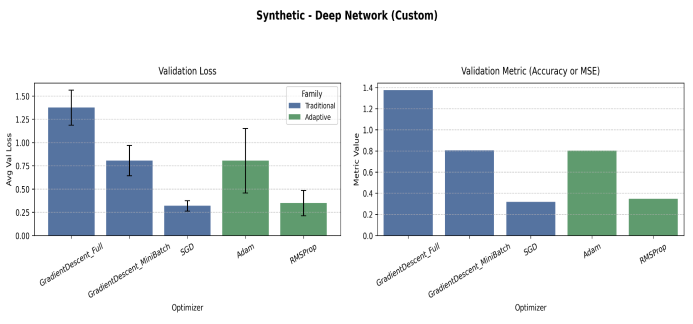
3. Results and Analysis — Sub-Project 1

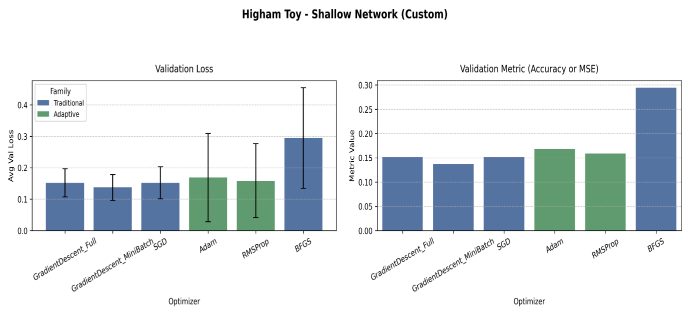
This section evaluates six optimization algorithms including three variations of the Gradient Descent (full-batch, mini-batch, and stochastic), BFGS, Adam and RMSProp, all of which were entirely implemented from scratch in NumPy. Experiments were then conducted across the three datasets highlighted in section 2.2 (Synthetic, Higham Toy, and Kaggle), each trained using the shallow and deep feed-forward networks specified in section 2.1. Performance was assessed via five-fold cross-validation mean training and validation losses and either mean squared error (mse) for regression tasks or accuracy for classification tasks.

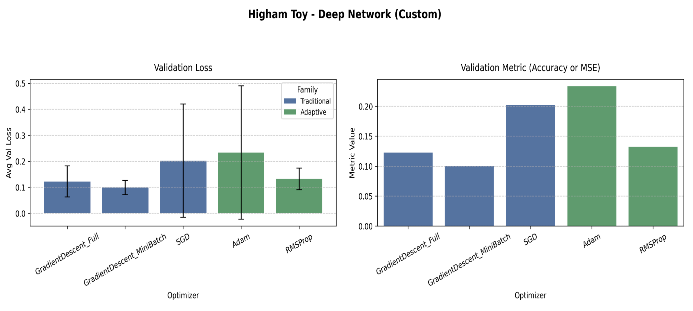
The bar plots (Figures 1-6) summarize each optimizer’s average validation loss (left-panel of figure) and task-specific metric (right-panel of figure). Furthermore, the errors represent the standard deviation across folds.

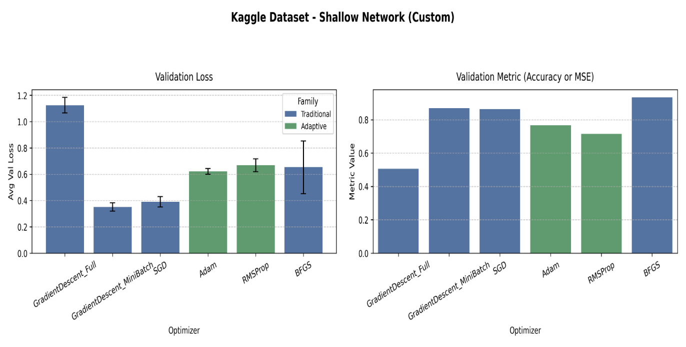
It is worth noting that for computational limitations BFGS was only used to train the Shallow network models.

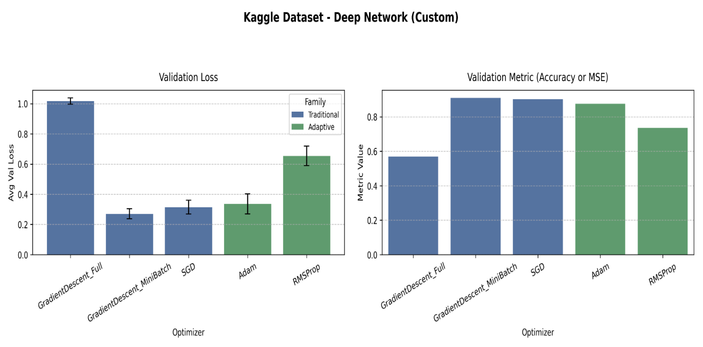












3.1. Traditional Gradient-Based Methods

Across all datasets, the baseline Gradient Descent variants reveal clear differences in convergence and generalization efficiency as the batch strategy changes:

- **Full-batch GD** often shows stable but slower convergence, usually reaching higher final validation losses especially in the Synthetic and Kaggle datasets, where it exceeds 1.0 in loss for both architectures. Its high bias but low variance highlights both its inefficiency on complex landscapes and its insensitivity to stochastic noise, respectively.
- **Mini-batch GD** consistently reduced both loss and variance compared to the full-batch variant. For instance, in Higham Toy Deep, the average loss fell below 0.10, while the standard deviation narrowed notably, demonstrating improved generalization and smoother gradient estimation.
- **Stochastic GD (SGD)** consistently achieved some of the lowest validation losses, e.g., it achieved a loss of approximately 0.3 in Synthetic Deep which was the lowest of all optimizers, but also occasionally displayed large error bars, confirming its high-variance nature.
- **BFGS** was only used on Shallow networks due to computational limitations and performed poorly in comparison to the first-order methods previously discussed as it often attained the highest losses amongst traditional algorithms, like in Higham Toy Shallow where it displayed the largest loss. It also displayed some of the largest standard deviations. Indeed, the high variability confirms numerical instability when approximating hessians especially in high-dimensional spaces.

The right panel metrics reinforce our loss-based observations:

- In Synthetic and Kaggle datasets, **Mini-Batch GD** and **SGD** consistently outperformed **Full_Batch GD** in both MSE (lower is better) and accuracy (higher is better). For instance, in Kaggle Deep, SGD achieved ≈ 0.90 accuracy, surpassing Full GD (≈ 0.57).
- Although **BFGS** attained the best accuracy (>0.9) on the Kaggle Shallow classification task, it consistently underperformed on regression problems, exhibiting the highest MSE on both the Synthetic Shallow and Higham Shallow datasets. This behavior suggests potential overfitting or failures in the line-search step.
- The correlation between low loss and good metric for all Gradient Descent variants, validating the consistency of both indicators.

3.2. Adaptive Gradient Methods

The adaptive methods, Adam and RMSProp, showed clear advantages in convergence speed and stability, especially during the early training phases. Their adaptive learning-rate mechanisms reduced the need for manual tuning and accelerated descent toward lower loss regions. However, despite their rapid convergence, they did not consistently achieve the best final validation losses or metrics across all tasks as performance varied significantly between datasets and network depths:

- **Adam** showed rapid initial convergence and consistently reduced training loss early, but its validation losses fluctuated more across folds than SGD or Mini-Batch GD.
  This higher variance is visible in the wider error bars in both the Higham Toy and Synthetic results. The cause lies in Adam's reliance on moving averages of past gradients and squared gradients, which can over-react to noisy updates in small-batch regimes.
- **RMSProp** behaved more conservatively, often converging more slowly than Adam but with slightly lower inter-fold variability. In regression-style datasets (like Synthetic), RMSProp's smoother exponential averaging helped stabilize updates, though its final losses were not always the lowest.

The right panel metrics observations reinforce the convergence and stability analysis once again:

- **Adam** attained great performance on several tasks including both shallow (Synthetic Shallow) and deep networks (Kaggle Deep) but not universally. Its fast learning occasionally led to over-adaptation, where the model converged to minima with higher validation loss, lower accuracy or higher MSE than other optimizers like mini-batch GD.
- **RMSProp** provided balanced behavior, maintaining moderate accuracy and MSE scores with less fold-to-fold variation.

In summary, while adaptive optimizers accelerated descent and reduced sensitivity to learning-rate tuning, they did not consistently minimize final validation loss nor ensure lower variance than stochastic methods such as SGD.

3.3. Comparative Discussion

Across all datasets, the comparative results reveal that adaptive optimizers (Adam, RMSProp) and traditional gradient-based methods (GD, Mini-Batch GD, SGD, BFGS) each exhibit distinct trade-offs between convergence speed, variance, and generalization. Adaptive methods, particularly Adam, showed faster initial convergence and reduced dependence on learning-rate tuning, yet they did not consistently achieve the lowest final validation losses or best task metrics. Adam frequently exhibited higher variance across folds compared to SGD or Mini-Batch GD, indicating sensitivity to gradient noise. RMSProp demonstrated steadier behavior but often converged to conservative, slightly suboptimal plateaus.

Traditional optimizers, though slower, achieved comparable or superior final results in several cases. Mini-Batch GD and SGD, in particular, combined efficiency with lower variance and stronger generalization, occasionally outperforming adaptive methods on shallower architectures and regression tasks. Overall, adaptive methods remain advantageous for deep or ill-conditioned models due to their dynamic learning-rate adaptation, but for smaller-scale or smoother problems, traditional gradient-based approaches offer more stable and interpretable convergence.

(All detailed epoch-wise training loss convergence plots for each optimizer, dataset, and architecture are available in the project's GitHub repository for reference and reproducibility.)

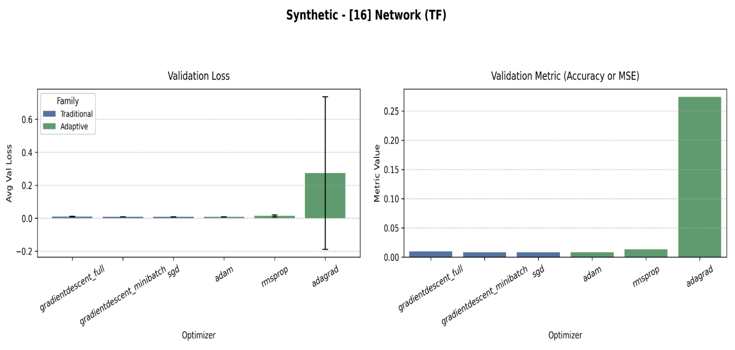
4. Results and Analysis — Sub-Project 2

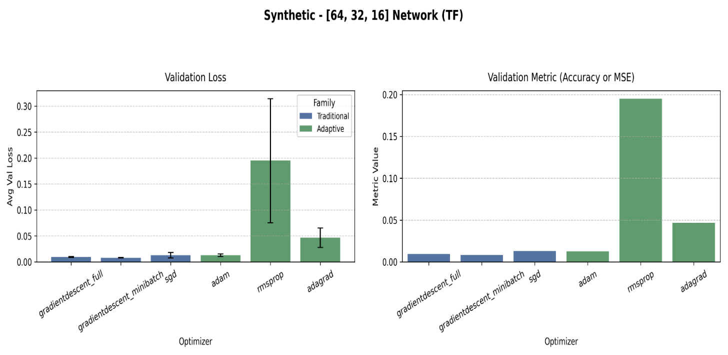
This section evaluates six Tensorflow-implemented Keras optimizers, including three Gradient Descent variants (full-batch, mini-batch, and stochastic), along with three adaptive ones, namely Adam, RMSProp, and Adagrad. Similarly to Sub-Project 1, each model was trained on all three datasets (Synthetic, Higham Toy, and the Kaggle one) using both the shallow and deep feed-forward network architectures defined in Section 2.1, but this time using the Keras dense layers instead of our from-scratch NumPy implementation.

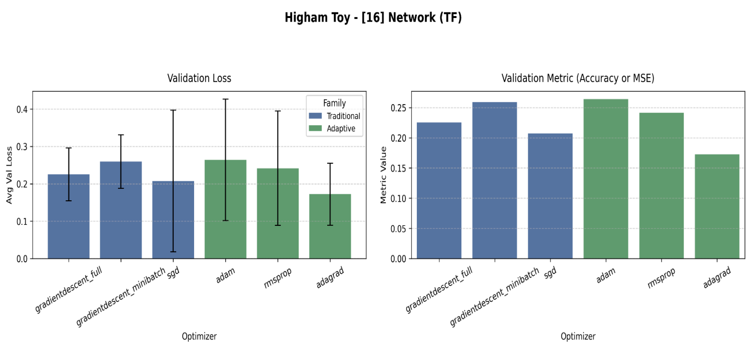
To ensure consistency between our custom experiments using our custom optimizer implementations and the tensorflow-based experiments, performance was assessed through five-fold cross-validation, measuring average training and validation losses and reporting mean squared error (MSE) for regression tasks or accuracy for classification tasks.

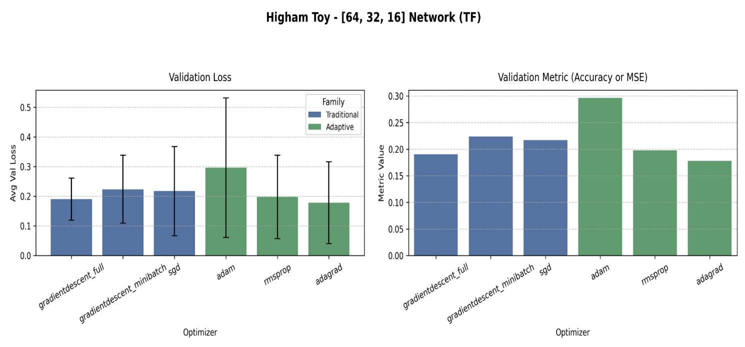
The following bar plots (Figures 7-12) summarize each optimizer's average validation loss (left-panel of figure) and task-specific metric (right-panel of figure). Error bars represent the standard deviation across folds, providing an indicator of optimizer stability.

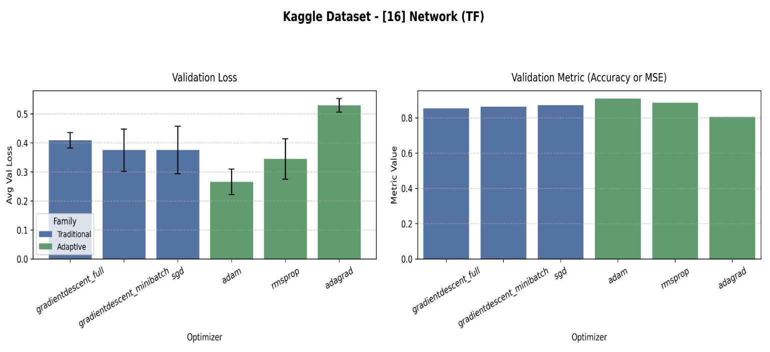
It is worth noting that we picked Adagrad to replace BFGS as the latter is not implemented in Keras. The choice of Adagrad was not made on the basis of similarity with BFGS, but was rather purely of random nature.

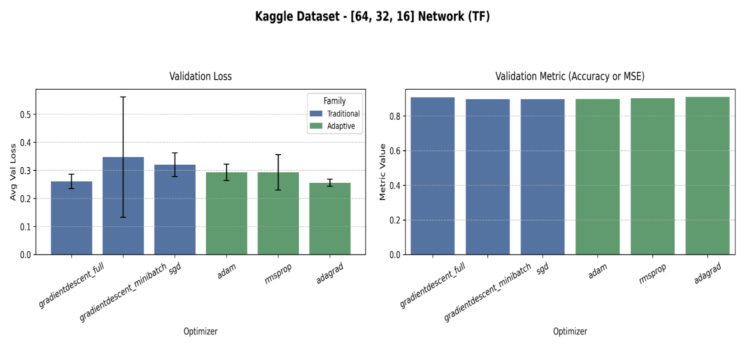












4.1. Traditional Gradient-Based Methods

Across all datasets, the Tensorflow implementations of the baseline Gradient Descent family displayed consistent and predictable behavior.

- **Full-batch GD** produced stable convergence with low variance but slower descent, often achieving moderately good loss minimization with the lowest or among the lowest validation losses. For instance, its deterministic nature allowed for a very smooth convergence, allowing it to reach smaller minima, particularly evident in the Higham Toy and Synthetic datasets across both architectures where the full-batch losses were often below those of mini-batch and SGD. Even in several deep-network runs, Full GD occasionally surpassed SGD, a rare outcome that indicates improved numerical precision and stability within TensorFlow's built-in gradient computations. However, its convergence speed remained slower, and its lower variance across folds confirms the absence of gradient noise.
- **Mini-Batch GD** produced slightly higher average losses in some experiments compared to its full-batch counterpart; this contrasts with expectations from the custom NumPy implementation. This indicates that while moderate stochasticity can aid generalization, TensorFlow's momentum-free version of Mini-Batch GD showed limited benefit here, likely due to smoother datasets and smaller batch-size regularization effects.
  Its performance remained consistent but less optimal than Full GD in both Higham Toy and Kaggle (deep) settings.
- **SGD** maintained competitive performance, occasionally approaching the best losses but usually with higher fold-to-fold variability reflecting the inherent noise of single sample updates. It is also worth noting that while SGD sometimes converged faster, it didn't consistently outperform mini-batch or full-batch GD approaches, especially in regression tasks.

Our figure wise right-panel metrics reinforce these findings:

- There is a clear positive correlation between lower losses and task-specific metric improvements across all optimizers, datasets and architectures.
- **Full-Batch GD** often achieved comparable and occasionally even better accuracy/MSE than its stochastic counterparts, confirming that its lower losses translated to meaningful metric improvements.
- **SGD** and **Mini-Batch GD** displayed slightly more metric variance than **Full-Batch** but similar overall averages, indicating that all three methods converged toward comparable basins in parameter space.

4.2. Adaptive Gradient Methods

The adaptive optimizers, Adam, RMSProp, and Adagrad, demonstrated rapid convergence and reduced dependence on explicit learning-rate tuning but did not uniformly outperform traditional methods in final validation loss or metrics:

- **Adam** demonstrated strong performance on complex datasets but showed dataset-specific limitations. In the Higham Toy dataset, Adam performed poorly, consistently recording the highest validation losses across both the shallow and deep networks.The dataset's small size and relatively simple loss landscape diminished the benefits of adaptive step scaling, leading Adam to overshoot optimal regions and oscillate near suboptimal minima. In contrast, on the Synthetic and Kaggle datasets, Adam demonstrated superior convergence, consistently achieving one of the lowest, if not the lowest, validation losses among all tested optimizers. Its momentum-based adaptive updates enabled efficient navigation of the higher-variance loss landscapes, particularly in deeper architectures where gradient magnitudes varied substantially between layers.
- **RMSProp** displayed highly dataset-dependent behavior. It outperformed Adam in the Higham Toy dataset across both shallow and deep networks, where its more conservative exponential averaging helped stabilize updates on the smooth, low-noise surface. However, in the Synthetic Deep configuration, RMSProp performed significantly worse, showing both high final losses and large variance across folds, suggesting difficulty adapting to the rapidly changing gradient scales of deeper nonlinear mappings. In the Kaggle dataset, RMSProp maintained competitive performance with Adam overall, nearly matching it in the deep configuration and trailing slightly in the shallow one, indicating effective generalization to noisier, real-world sensor data.
- **Adagrad** showed the weakest performance and limited adaptability. Its cumulative learning-rate decay caused rapid step-size reduction, leading to premature convergence and poor final validation losses across all datasets. While occasionally stable, it consistently underperformed Adam and RMSProp in both convergence speed and final metric scores.

The right-panel metrics clearly reflect our loss-based finding:

- **Adam** achieved some of the best overall validation metrics in both Synthetic and Kaggle datasets, confirming its strength on complex and noisy data.

- **RMSProp**, despite its regression on Synthetic Deep, maintained balanced metric performance elsewhere, validating its ability to generalize when gradients remain well-behaved.
- **Adagrad** lagged behind across all settings, failing to maintain effective step sizes in deeper network, thus achieving worse metrics on most datasets.

4.3 Comparative Discussion

The TensorFlow-based experiments reveal a complex trade-off between traditional and adaptive optimization strategies that depends heavily on dataset complexity and network depth. Full-Batch Gradient Descent consistently produced the most stable and often the lowest validation losses across all datasets, benefitting from TensorFlow's numerically precise gradient computations. Mini-Batch GD and SGD, while faster and slightly more stochastic, achieved comparable average metrics but exhibited higher fold-to-fold variance, reflecting their noisier update dynamics. In contrast, adaptive methods such as Adam and RMSProp converged more rapidly but showed clear dataset dependence: Adam excelled on the Synthetic and Kaggle datasets achieving the lowest validation losses and highest metrics, yet performed poorly on Higham Toy, while RMSProp outperformed Adam in Higham Toy but collapsed on Synthetic Deep and only remained competitive on Kaggle. On the other hand, Adagrad was consistently limited by its diminishing learning rate.

Overall, the results emphasize that no optimizer universally dominates. Traditional methods like Full GD retain superior stability and generalization on smoother, low-noise problems, whereas adaptive optimizers, especially Adam, excel in complex, noisy, or deeper networks where gradient magnitudes vary widely. This complementarity underscores that optimizer choice should remain context-driven, balancing convergence speed against stability and variance depending on dataset and architecture characteristics.

(All detailed epoch-wise training loss convergence plots for each optimizer, dataset, and architecture are available in the project's GitHub repository for reference and reproducibility.)

5. Conclusion

This work presented a systematic comparison of optimization algorithms for training feedforward neural networks using both custom NumPy implementations and TensorFlow's built-in optimizers. Sub-Project 1 focused on traditional and adaptive methods implemented from scratch, highlighting how learning-rate strategies, batch processing schemes, and optimizer formulations influence convergence, generalization, and computational efficiency. Sub-Project 2 extended this investigation to TensorFlow's high-level environment, validating the empirical consistency of framework implementations and revealing subtle differences introduced by automatic differentiation and numerical precision. Across both studies, no single optimizer proved universally superior: traditional gradient-based methods offered stable and interpretable convergence on smoother problems, whereas adaptive methods such as Adam and RMSProp excelled in complex, noisy, or deeper networks. Collectively, the findings underscore that optimizer choice remains a context-dependent design decision that must balance speed, stability, and generalization.

6. Future Work

Future extensions of this research will focus on scaling experiments to larger architectures and datasets, integrating second-order methods such as L-BFGS and hybrid adaptive quasi-Newton schemes, and incorporating learning-rate schedulers for more dynamic training control. Additional directions include analyzing optimizer behavior in convolutional or recurrent networks, performing hyperparameter sensitivity studies, and investigating mixed-precision training to quantify numerical stability effects. Such extensions would deepen understanding of how optimization dynamics evolve under realistic deep learning workloads and guide the design of more efficient, robust training algorithms.

References

1. C. F. Higham and D. J. Higham, "Deep Learning: An Introduction for Applied Mathematicians," *arXiv preprint arXiv:1801.05894*, 2018. [Online]. Available: https://arxiv.org/abs/1801.05894
2. D. P. Kingma and J. Ba, *"Adam: A Method for Stochastic Optimization,"* in *Proceedings of the 3rd International Conference on Learning Representations (ICLR)*, 2015.
3. G. Hinton, N. Srivastava, and K. Swersky, *"Neural Networks for Machine Learning – Lecture 6a: Overview of Mini-Batch Gradient Descent,"* University of Toronto, 2012.
4. J. Duchi, E. Hazan, and Y. Singer, *"Adaptive Subgradient Methods for Online Learning and Stochastic Optimization," Journal of Machine Learning Research (JMLR)*, vol. 12, pp. 2121–2159, 2011.
5. T. Tieleman and G. Hinton, *"RMSProp: Divide the Gradient by a Running Average of Its Recent Magnitude,"* Coursera: Neural Networks for Machine Learning, 2012.
6. J. Nocedal and S. J. Wright, *Numerical Optimization*, 2nd ed., Springer, 2006.
7. M. Ruder, *"An Overview of Gradient Descent Optimization Algorithms,"* arXiv preprint arXiv:1609.04747, 2016.
8. T. Hastie, R. Tibshirani, and J. Friedman, *The Elements of Statistical Learning: Data Mining, Inference, and Prediction*, 2nd ed., Springer, 2009.
9. TensorFlow Developers, *TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems*, Google, 2015. [Online]. Available: https://www.tensorflow.org/
10. F. Chollet et al., *Keras API Reference Documentation*, Google Research, 2015. [Online]. Available: https://keras.io/
11. N. Higham, *Accuracy and Stability of Numerical Algorithms*, 2nd ed., SIAM, 2002.
12. L. Bottou, F. E. Curtis, and J. Nocedal, *"Optimization Methods for Large-Scale Machine Learning," SIAM Review*, vol. 60, no. 2, pp. 223–311, 2018.
13. UCI Machine Learning Repository, "Wall-Following Robot Navigation Data Set," *Kaggle Datasets*, 2017. [Online]. Available: https://www.kaggle.com/datasets/uciml/wall-following-robot

14. M. Kchaou, "OptimizerArena: Loss Landscape Visualization and Optimizer Analysis Platform," GitHub, 2025. [Online]. Available: https://github.com/MK720-dev/OptimizerArena
(Note: Contains the enhanced implementation of experiments described in this paper)