Benchmarking Optimizers on MNIST Classification Tasks

Po Hung, Cheng *Department of Electrical Engineering*
National Taiwan University
alexjeng0404@gmail.com

Abstract

This report investigates and compares the performance of different optimization methods in the MNIST handwritten digit classification task. We employ both logistic regression and multilayer perceptron (MLP) models. The experiments aim to analyze convergence behavior, training stability, and generalization performance across optimizers such as Adam, AdamW, RAdam, AdaBelief, AMSGrad, and others.

Index Terms

Optimization, MNIST, Neural Networks, Deep Learning, Machine Learning

I. Introduction

In machine learning, optimization methods play a crucial role in deep learning, where neural networks are trained to perform sophisticated tasks such as classification, regression, and image recognition. Frameworks such as PyTorch offer a rich library of these optimization algorithms.
In this paper, we delve into the mechanisms and properties of these common optimizers in PyTorch and present a thorough analysis of the **final error rate** and **precision** of simple test, logistic regression and MLP.

II. Background and Related Work

This section establishes the theoretical context for the models and optimization algorithms studied in this report, covering the foundational concepts of our baseline classifiers and the mechanisms of modern gradient-based optimizers.

A. Classification Models

1) Multinomial Logistic Regression: Logistic Regression (LR) is a linear classification model that serves as a simple yet interpretable baseline. In the multi-class setting with C classes, we use *multinomial* logistic regression, which models the posterior class probabilities using a linear score followed by a Softmax transformation.

Given an input feature vector $\mathbf{x} \in \mathbb{R}^d$ and a class label $y \in \{1, \dots, C\}$, multinomial LR computes a vector of class scores

$$\mathbf{z} = W\mathbf{x} + \mathbf{b},$$

where $W \in \mathbb{R}^{C\times d}$ is the weight matrix and $\mathbf{b} \in \mathbb{R}^C$ is the bias vector. The class posterior probabilities are obtained by applying the Softmax function:

$$p_\theta(y = c \mid \mathbf{x}) = \frac{\exp(z_c)}{\sum_{k=1}^{C} \exp(z_k)}, \quad c = 1, \dots, C,$$

where $\theta = \{W, \mathbf{b}\}$ denotes all trainable parameters and z_c is the c-th entry of $\mathbf{z}$.

For a dataset $\{(\mathbf{x}^{(i)}, y^{(i)})\}_{i=1}^N$, the negative log-likelihood (cross-entropy) loss is

$$\mathcal{L}_{\text{MLR}}(\theta) = -\frac{1}{N}\sum_{i=1}^{N} \log p_\theta\big(y^{(i)} \mid \mathbf{x}^{(i)}\big) = -\frac{1}{N}\sum_{i=1}^{N} \log \frac{\exp\big(z^{(i)}_{y^{(i)}}\big)}{\sum_{k=1}^{C}\exp\big(z^{(i)}_k\big)},$$

where $\mathbf{z}^{(i)} = W\mathbf{x}^{(i)} + \mathbf{b}$ and $z^{(i)}_{y^{(i)}}$ is the score corresponding to the true class $y^{(i)}$.

The gradient of the loss with respect to the parameters has a simple closed form. Denoting

$$\mathbf{p}^{(i)} = \big(p_\theta(y = 1 \mid \mathbf{x}^{(i)}), \dots, p_\theta(y = C \mid \mathbf{x}^{(i)})\big)^\top$$

and the one-hot encoding of the label as $\mathbf{e}^{(i)} \in \{0, 1\}^C$ (with $e^{(i)}_{y^{(i)}} = 1$), we have

$$\nabla_W \mathcal{L}_{\text{MLR}}(\theta) = \frac{1}{N}\sum_{i=1}^{N} \big(\mathbf{p}^{(i)} - \mathbf{e}^{(i)}\big)\,\mathbf{x}^{(i)\top}, \quad \nabla_{\mathbf{b}} \mathcal{L}_{\text{MLR}}(\theta) = \frac{1}{N}\sum_{i=1}^{N} \big(\mathbf{p}^{(i)} - \mathbf{e}^{(i)}\big).$$

Because the model is linear in $\mathbf{x}$ and the Softmax–cross-entropy objective is convex in θ (up to common regularization such as ℓ_2 weight decay), multinomial LR typically leads to a well-behaved optimization landscape. This makes it a natural choice as an interpretable baseline for evaluating gradient-based optimization algorithms.

2) Multi-Layer Perceptron (MLP): While multinomial LR learns only a linear decision boundary, many real-world classification tasks require modeling highly non-linear relationships. A Multi-Layer Perceptron (MLP) is the simplest form of a feed-forward neural network that addresses this limitation by stacking linear transformations with non-linear activation functions.

Consider an L-layer MLP (with $L-1$ hidden layers) that maps an input $\mathbf{x} \in \mathbb{R}^d$ to class logits $\mathbf{z}^{(L)} \in \mathbb{R}^C$. The forward propagation can be written as

$$\mathbf{h}^{(1)} = \sigma\big(W^{(1)}\mathbf{x} + \mathbf{b}^{(1)}\big),$$
$$\mathbf{h}^{(\ell)} = \sigma\big(W^{(\ell)}\mathbf{h}^{(\ell-1)} + \mathbf{b}^{(\ell)}\big), \quad \ell = 2, \dots, L-1,$$
$$\mathbf{z}^{(L)} = W^{(L)}\mathbf{h}^{(L-1)} + \mathbf{b}^{(L)},$$

where $W^{(\ell)}$ and $\mathbf{b}^{(\ell)}$ are the weight matrix and bias vector at layer ℓ, $\sigma(\cdot)$ is a non-linear activation function (e.g., ReLU), and $\mathbf{h}^{(\ell)}$ denotes the hidden representation at layer ℓ.

For multi-class classification, the output logits $\mathbf{z}^{(L)}$ are passed through a Softmax function to obtain class probabilities:

$$p_\Theta(y = c \mid \mathbf{x}) = \frac{\exp\left(z_c^{(L)}\right)}{\sum_{k=1}^{C} \exp\left(z_k^{(L)}\right)}, \quad c = 1, \dots, C,$$

where $\Theta = \{W^{(\ell)}, \mathbf{b}^{(\ell)}\}_{\ell=1}^{L}$ denotes all network parameters. The training objective is again the cross-entropy loss

$$\mathcal{L}_{\text{MLP}}(\Theta) = -\frac{1}{N}\sum_{i=1}^{N} \log p_\Theta\big(y^{(i)} \mid \mathbf{x}^{(i)}\big).$$

Unlike multinomial LR, the presence of non-linear activations and multiple layers makes $\mathcal{L}_{\text{MLP}}(\Theta)$ a highly non-convex function of the parameters. This results in a rich optimization landscape with local minima and saddle points, posing a significantly more challenging optimization problem. Consequently, MLPs provide a natural testbed for studying the behavior of advanced gradient-based optimizers (e.g., momentum methods, adaptive learning rate algorithms) beyond the convex setting.

B. Gradient-Based Optimization Algorithms

Gradient-based optimization algorithms form the foundation of neural network training. The goal is to minimize a loss function $L(\theta)$ by iteratively updating parameters θ in the direction of the negative gradient. Broadly, these methods fall into two families: (1) base and momentum-based optimizers, and (2) adaptive learning rate methods.

1) Base and Momentum-Based Methods: **SGD, Momentum, and Nesterov Accelerated Gradient (NAG)** are the simplest yet most fundamental optimizers.

- **SGD** [1]: Updates parameters using a fixed learning rate η:

  $$\theta_{t+1} = \theta_t - \eta \nabla L(\theta_t)$$

  While conceptually simple, SGD can be slow to converge, especially on ill-conditioned loss surfaces.
- **Momentum** [2]: Introduces a velocity term v_t to accumulate a moving average of past gradients:

  $$v_t = \beta v_{t-1} + (1-\beta)\nabla L(\theta_t), \quad \theta_{t+1} = \theta_t - \eta v_t$$

  The momentum term helps accelerate updates along consistent gradient directions and reduces oscillation in noisy regions.
- **Nesterov Accelerated Gradient (NAG)** [3]: Improves upon momentum by evaluating the gradient at a lookahead position, resulting in faster convergence for smooth loss landscapes:

  $$h_t = (1+\beta)v_t - \beta v_{t-1}, \quad \theta_{t+1} = \theta_t - \eta h_t$$

  Here, h_t acts as a corrected velocity that anticipates the next move.

2) Adaptive Learning Rate Methods: These methods adjust the learning rate dynamically for each parameter using statistics from the gradient history. They include **Adagrad**, **RMSProp**, **Adam**, **Nadam**, and several of their refinements such as **AMSGrad**, **AdamW**, **RAdam**, and **AdaBelief**.

- **Adagrad** [4]: Adapts the learning rate based on the accumulated sum of squared gradients:

  $$G_t = G_{t-1} + g_t^2, \quad \theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{G_t + \epsilon}} g_t$$

It performs well for sparse features but often suffers from an overly small learning rate over time.

- **RMSProp** [5]: Fixes Adagrad's decay problem by replacing the accumulation with an exponential moving average:

$$v_t = \beta_2 v_{t-1} + (1-\beta_2) g_t^2, \quad \theta_{t+1} = \theta_t - \eta \frac{g_t}{\sqrt{v_t} + \epsilon}$$

RMSProp remains a popular choice for recurrent and non-stationary tasks.

- **Adam** [6]: Combines momentum (first-moment estimate) with RMSProp-style adaptive scaling (second-moment estimate):

$$m_t = \beta_1 m_{t-1} + (1-\beta_1) g_t, \quad v_t = \beta_2 v_{t-1} + (1-\beta_2) g_t^2$$

$$\hat{m}_t = \frac{m_t}{1-\beta_1^t}, \quad \hat{v}_t = \frac{v_t}{1-\beta_2^t}, \quad \theta_{t+1} = \theta_t - \eta \frac{\hat{m}_t}{\sqrt{\hat{v}_t} + \epsilon}$$

Adam's balance between stability and adaptivity makes it a strong default optimizer.

- **AdamW** [7]: Decouples weight decay from the gradient update:

$$\theta_{t+1} = \theta_t - \eta \left(\frac{\hat{m}_t}{\sqrt{\hat{v}_t} + \epsilon} + \lambda \theta_t \right)$$

This modification ensures true L_2 regularization and improves generalization.

- **AMSGrad** [8]: Maintains a non-decreasing second-moment estimate to fix Adam's theoretical convergence issue:

$$\hat{v}_t = \max(\hat{v}_{t-1}, v_t)$$

$$\theta_{t+1} = \theta_t - \eta \frac{m_t}{\sqrt{\hat{v}_t} + \epsilon}$$

- **Nadam** [9]: Combines Nesterov momentum with Adam's adaptive moments for smoother updates:

$$\hat{m}_t = \frac{\beta_1 m_t}{1-\beta_1^t} + \frac{(1-\beta_1) g_t}{1-\beta_1^t}$$

$$\theta_{t+1} = \theta_t - \eta \frac{\hat{m}_t}{\sqrt{\hat{v}_t} + \epsilon}$$

- **RAdam** [10]: Rectifies the variance of adaptive learning rates during early training:

$$r_t = \sqrt{\frac{(\rho_t - 4)(\rho_t - 2)\rho_\infty}{(\rho_\infty - 4)(\rho_\infty - 2)\rho_t}}$$

The scaling factor r_t stabilizes learning when moment estimates are still unreliable.

- **AdaBelief** [11]: Replaces the second-moment term with the squared deviation between the gradient and its mean:

$$s_t = \beta_2 s_{t-1} + (1-\beta_2)(g_t - m_t)^2$$

This allows the optimizer to trust gradients consistent with its belief while dampening outliers.

3) Advanced Techniques (Lookahead, CLR):

- **Lookahead Optimizer** [12]: Acts as a meta-optimizer that wraps around a base optimizer (e.g., SGD, Adam). Lookahead maintains two sets of weights: fast weights ϕ_t, updated every step by the inner optimizer, and slow weights θ_t, updated only every k steps by interpolating towards the fast weights. At each iteration t:

$$\phi_{t+1} = \text{InnerOptim}(\phi_t, g_t),$$

where InnerOptim denotes one update of the base optimizer using the gradient g_t. Every k steps, the slow and fast weights are synchronized via

$$\theta_{t+1} = \theta_t + \alpha(\phi_{t+1} - \theta_t), \quad \phi_{t+1} \leftarrow \theta_{t+1},$$

where $\alpha \in (0, 1]$ is the interpolation (lookahead) coefficient and k is the synchronization interval. This matches the common implementation pattern: fast weights are updated at every call to `step()`, and after every k steps, the slow weights move towards the fast weights by a factor α, after which the fast weights are reset to the new slow weights. By "looking ahead" in this way, the optimizer smooths the optimization trajectory and often improves stability and generalization across different base optimizers.

- **Cyclical Learning Rates (CLR)** [13]: CLR is not an optimizer itself but a learning rate scheduling strategy that cycles the learning rate between a lower and upper bound instead of monotonically decaying it. For example, a simple triangular policy updates the learning rate at iteration t as

$$\eta_t = \eta_{\min} + (\eta_{\max} - \eta_{\min}) \left(1 - \left|\frac{2(t \bmod T)}{T} - 1\right|\right),$$

where $\eta_{\min}$ and $\eta_{\max}$ are the minimum and maximum learning rates, and T is the length of one cycle. Such cyclic schedules can help the optimizer escape shallow local minima and saddle points while reducing the need for extensive manual tuning of the learning rate.

III. Experimental Setup

We evaluate optimization algorithms in a two-stage protocol of increasing complexity:

1) a one-dimensional toy optimization problem with a known analytic optimum; and
2) supervised image classification on MNIST using a linear baseline (multinomial Logistic Regression) and a non-linear baseline (Multi-Layer Perceptron, MLP).

This design enables us to separate the optimizer's behavior from the complexities of the model and data. The toy function serves as a controlled testing environment, while the ENIST experiments evaluate whether the identified convergence patterns hold true in practical learning scenarios.

A. Stage I: Toy Function Benchmark

1) Objective: As a controlled and interpretable testbed, we consider the quadratic function

$$f(x) = x^2 + 3x + 2,$$

a strictly convex function with a closed-form global minimum at

$$x^* = -1.5, \quad f(x^*) = -0.25.$$

This experiment serves two purposes:

- **Isolating optimizer characteristics**: We visualize and quantify the fundamental *convergence behavior* and *update trajectory* of different optimizers (e.g., SGD, Adam, AdaBelief) on a simple, known loss landscape.
- **Establishing a baseline**: The toy problem provides a sanity check and baseline expectation for how each optimizer behaves before scaling to high-dimensional neural network training.

B. Stage II: MNIST Classification

We next evaluate each optimizer on supervised image classification with MNIST, allowing us to study whether favorable properties observed in 1D carry over to practical tasks.

1) Dataset and Preprocessing: We utilize the standard MNIST dataset of handwritten digits: 60 000 training images and 10 000 test images of 28×28 grayscale digits (classes 0–9). All images are normalized to $[0, 1]$ and standardized using the dataset mean and standard deviation. We reserve a fixed subset of the original training set as a validation set (10% split) for hyperparameter selection and early stopping.

2) Models:

- **Logistic Regression Baseline**: A single linear layer (fully-connected) mapping the flattened input vector ($\mathbf{D_{in} = 784}$ features) directly to $\mathbf{10}$ class logits.
- **Multilayer Perceptron (MLP) Architecture**: A fully-connected feed-forward network with $\mathbf{L = 3}$ hidden layers.
- **Hidden Layer Dimensions**: The network employs the following configuration for the hidden layers (units $\rightarrow$ units):

$$\mathbf{784 \rightarrow 512 \rightarrow 256 \rightarrow 128 \rightarrow 10}$$

- Activation Function: The non-linear activation function is employed after each hidden layer is the Rectified Linear Unit (ReLU).

3) Optimizers: We implement and compare the performance of several optimizers, including: SGD, Momentum, Nesterov, Adagrad, RMSProp, Adam, AdamW, AMSGrad, RAdam, AdaBelief, and Lookahead (Adam). Refer to the tables for more detailed results.

C. Training Protocol and Metrics

1) Training Hyperparameters:

- **Epochs**: The maximum number of training epochs (10 for logistic regression, 5 for MLP).
- **Batch Size**: The mini-batch size for gradient computation was set to 32 for both logistic regression and MLP.
- **Learning Rate Schedule**: The parameters are listed below.

TABLE I: Hyperparameter in the Toy Function Experiment

toy function	lr	beta1	beta2	weight_decay	ϵ
SGD	0.1	x	x	x	x
Momentum	0.1	0.9	x	x	x
Nesterov	0.1	0.9	x	x	x
Adagrad	0.1	x	x	x	1e-8
RMSProp	0.1	x	0.99	x	1e-8
Adam	0.1	0.9	0.999	x	1e-8
AdamW	0.1	0.9	0.999	0.01	1e-8
AMSGrad	0.1	0.9	0.999	x	1e-8
RAdam	0.1	0.9	0.999	x	1e-8
AdaBelief	0.1	0.9	0.999	x	1e-8
Lookahead(Adam)	0.1	0.9	0.999	x	1e-8

TABLE II: Hyperparameter in the logistic regression

logistic regression	lr	beta1	beta2	weight_decay	ϵ
SGD	0.01	x	x	x	x
Momentum	0.01	0.9	x	x	x
Adam	0.001	0.9	0.999	x	1e-8
AdaBelief	0.001	0.9	0.999	x	1e-8
RMSProp	0.001	x	0.99	x	1e-8
Adagrad	0.1	x	x	x	1e-8

TABLE III: Hyperparameter in MLP

logistic regression	lr	beta1	beta2	weight_decay	ϵ
SGD	0.01	x	x	x	x
Momentum	0.01	0.9	x	x	x
Adam	0.001	0.9	0.999	x	1e-8
AdaBelief	0.001	0.9	0.999	x	1e-8
RMSProp	0.001	x	0.99	x	1e-8
Adagrad	0.1	x	x	x	1e-8
AdamW	0.1	0.9	0.999	0.01	1e-8

- **Loss Function**:The **objective function** is defined by the **toy function** ($f(x) = x^2 + 3x + 2$), **Cross-Entropy Loss** is employed to measure the error for both the Logistic Regression and MLP models.

2) Evaluation Metrics: We assess performance using the following metrics:

- **Training Loss Curves**: Analyze convergence speed and stability over 10 training epochs.
- **Test Accuracy**: The final classification accuracy on the MNIST test set after 10 epochs.
- **Convergence Speed**: Analyzed qualitatively by comparing loss curves across different optimizers.
- **Training Time**: Computational efficiency measured by total training time in seconds.

D. Implementation Details

All experiments are implemented using the **PyTorch** framework. Data loaders manage **dataset-specific preprocessing, batching, and train/validation splits**. A NVIDIA GPU (CUDA) is utilized for computation. Reproducibility is ensured by fixing the random seed for all optimizer runs within a specific model and dataset.

IV. Results and Discussion

A. Stage I: Toy Function Analysis

As shown in Fig. 1 and Fig. 2, all optimizers successfully converge to the global minimum of the quadratic toy function $f(x) = x^2 + 3x + 2$. Adaptive methods such as Adam, AdamW, and AdaBelief demonstrate smooth and stable convergence behavior.

In contrast, Adagrad converges more slowly due to its monotonically decreasing learning rate. Momentum-based methods (Momentum and Nesterov) exhibit slight oscillations after reaching a relatively low error, which can be attributed to the inertia introduced by the momentum term that causes the loss to fluctuate around the minimum.

These results confirm that while all optimizers can reach the global minimum in simple convex settings, adaptive algorithms achieve convergence more efficiently.

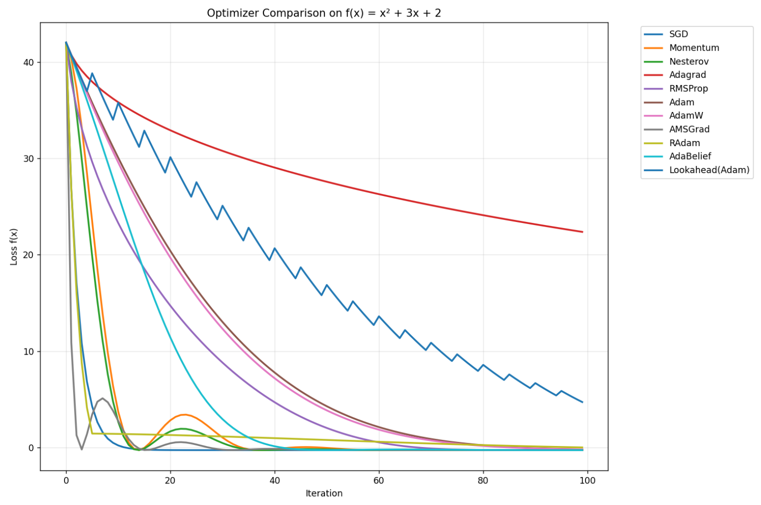

Fig. 1: Training loss of different optimizers on the toy quadratic function.

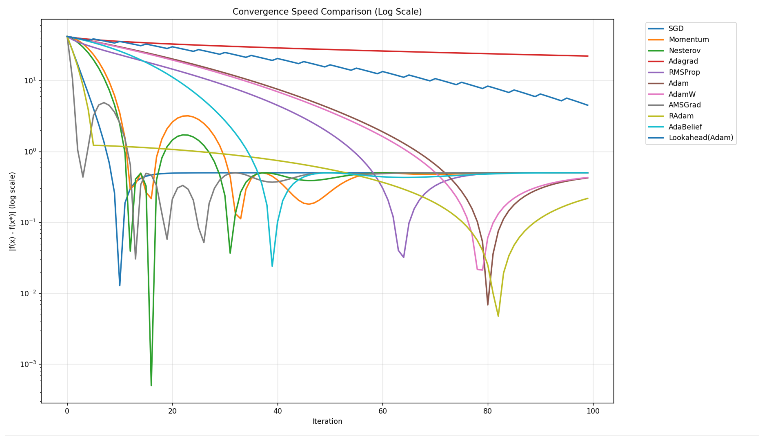

Fig. 2: Convergence speed comparison (log scale) on the toy function.

B. Stage II: Logistic Regression Performance

For the logistic regression model trained on the MNIST dataset, Fig. 3 shows that all optimizers consistently reduce the training loss within ten epochs. The corresponding test accuracy curves are presented in Fig. 4. All optimizers achieve test accuracies between 91Among them, Adam and AdaBelief exhibit slightly faster convergence and higher final accuracy, confirming that adaptive optimizers retain their advantages even in simple models.

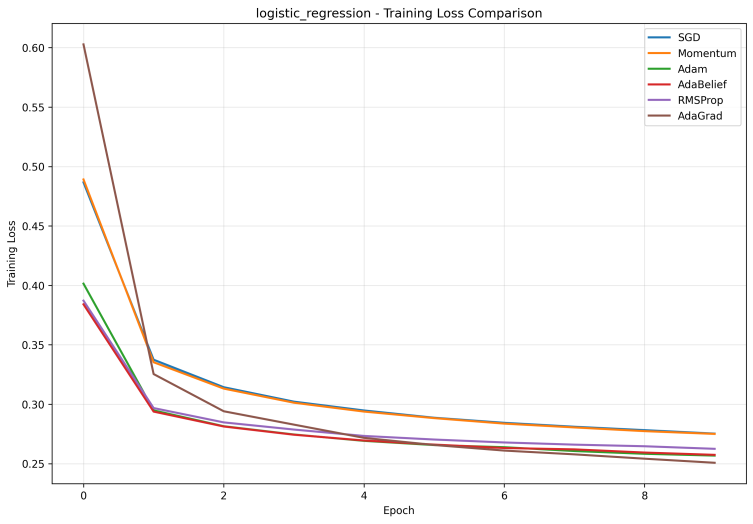

Fig. 3: Training loss comparison of optimizers on logistic regression.

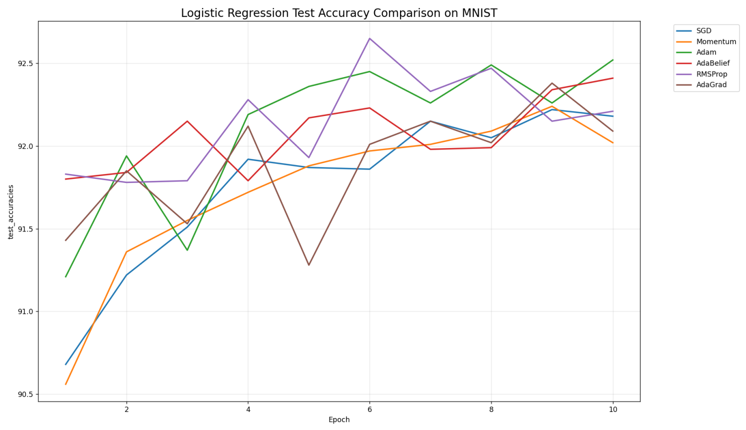

Fig. 4: Test accuracy comparison of optimizers on logistic regression.

C. *Stage II: MLP Performance*

For the MLP model, the trends become more pronounced. As shown in Fig. 5, Adam, AdamW, AdaBelief, and RMSProp achieve much lower training loss within only a few epochs, while SGD and Momentum converge slowly. Adagrad remains the slowest due to its aggressive learning rate decay.

Fig. 6 presents the test accuracy over epochs. AdamW achieves the best generalization performance, closely followed by AdaBelief and Adam, all reaching around 97.5–98% accuracy. In contrast, SGD and Momentum remain below 95% after the same number of epochs. These results suggest that adaptive optimizers not only converge faster but also generalize better in non-linear models such as MLPs.

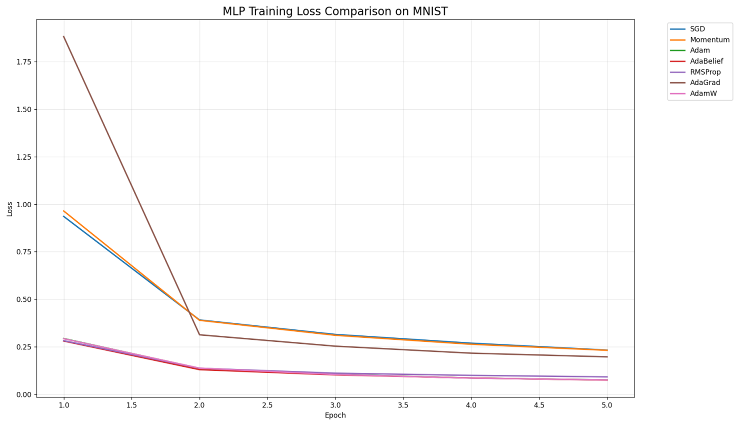

Fig. 5: Training loss comparison on MNIST (MLP).

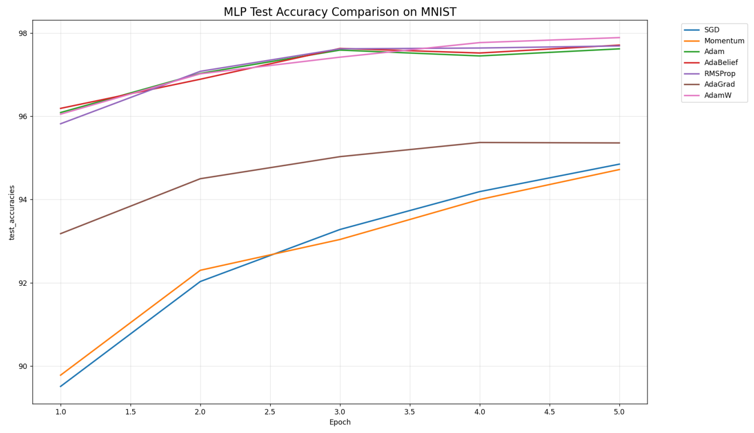

Fig. 6: Test accuracy comparison on MNIST (MLP).

D. Comparative Analysis

Across all experiments, adaptive learning rate algorithms (Adam, AdamW, AdaBelief, and RMSProp) consistently outperform traditional methods (SGD and Momentum) in both convergence speed and stability. Among them, AdamW demonstrates the best overall balance between rapid convergence and robust generalization, likely due to its decoupled weight decay formulation that mitigates overfitting. AdaBelief shows comparable results, offering smoother optimization dynamics by adapting its step size to the “belief” in gradient consistency.

RAdam and AMSGrad also perform reliably, though their advantages are less pronounced on small-scale datasets such as MNIST. These findings highlight that while simple methods like SGD remain competitive for large-scale, fine-tuned training, adaptive optimizers are generally more effective and efficient for standard deep learning tasks.

V. Conclusion

In this work, we briefly review several commonly used optimizers in the PyTorch library and evaluate their performance on both a toy function dataset and the MNIST dataset. For the toy function experiment, all optimizers successfully converge to zero, although Adagrad shows a relatively slower convergence rate. In the MNIST experiments, adaptive learning rate algorithms, such as Adam and AdamW, consistently outperform traditional methods like SGD and Momentum in terms of both convergence speed and stability. Overall, our results suggest that Adam and AdamW are generally the most effective and reliable choices for practitioners seeking efficient model training in typical deep learning applications.

While this study provides empirical insight into the optimization behaviors on controlled benchmarks and moderate-scale datasets, several directions remain open for further exploration:

- **Evaluation on More Complex Datasets:** Extending experiments to larger-scale and higher-dimensional datasets such as CIFAR-10, CIFAR-100, or ImageNet would help assess optimizer scalability and robustness in more realistic scenarios.
- **Exploration of Deeper and More Complex Architectures:** Applying the same comparative analysis to convolutional neural networks (CNNs), recurrent networks (RNNs), and transformer-based architectures could reveal how optimizer characteristics interact with distinct gradient dynamics.

- **Custom Learning Rate Schedules:** Incorporating advanced scheduling strategies—such as cyclical learning rates, cosine annealing, or warm restarts—may yield deeper insight into the interplay between optimizers and learning rate dynamics.
- **Theoretical Analysis:** Future work may also include formal convergence and stability analysis of adaptive methods under non-convex loss landscapes to better explain their empirical success.

In summary, this work highlights the trade-offs between convergence speed, stability, and generalization among modern optimizers. Continued research on larger-scale tasks, deeper architectures, and theoretical understanding will further clarify the design principles underlying efficient gradient-based optimization.

VI. Acknowledgment

References

[1] H. Robbins and S. Monro, "A stochastic approximation method," *The Annals of Mathematical Statistics*, vol. 22, no. 3, pp. 400–407, 1951.

[2] D. E. Rumelhart, G. E. Hinton, and R. J. Williams, "Learning representations by back-propagating errors," *Nature*, vol. 323, no. 6088, pp. 533–536, 1986.

[3] Y. Nesterov, *Introductory Lectures on Convex Optimization: A Basic Course*, ser. Applied Optimization. Kluwer Academic Publishers, 2004, vol. 87.

[4] J. Duchi, E. Hazan, and Y. Singer, "Adaptive subgradient methods for online learning and stochastic optimization," in *Advances in Neural Information Processing Systems (NeurIPS)*, vol. 24, 2011, pp. 257–264.

[5] T. Tieleman and G. Hinton, "Lecture 6.5—rmsprop: Divide the gradient by a running average of its recent magnitude," Coursera: Neural Networks for Machine Learning, 2012, university of Toronto.

[6] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," *International Conference on Learning Representations (ICLR)*, 2015.

[7] I. Loshchilov and F. Hutter, "Decoupled weight decay regularization," in *International Conference on Learning Representations (ICLR)*, 2019.

[8] S. J. Reddi, S. Kale, and S. Kumar, "On the convergence of adam and beyond," in *International Conference on Learning Representations (ICLR)*, 2018.

[9] T. Dozat, "Incorporating nesterov momentum into adam," in *ICLR Workshop*, 2016.

[10] L. Liu, H. Jiang, P. He, W. Chen, X. Liu, J. Gao, and J. Han, "On the variance of the adaptive learning rate and beyond," in *International Conference on Learning Representations (ICLR)*, 2020.

[11] J. Zhuang, T. Tang, Y. Ding, S. Tatikonda, N. Dvornek, X. Papademetris, and J. Duncan, "Adabelief optimizer: Adapting stepsizes by the belief in observed gradients," in *Advances in Neural Information Processing Systems (NeurIPS)*, vol. 33, 2020, pp. 18 795–18 806.

[12] M. R. Zhang, J. Lucas, J. Ba, and G. E. Hinton, "Lookahead optimizer: k steps forward, 1 step back," in *Advances in Neural Information Processing Systems (NeurIPS)*, vol. 32, 2019.

[13] L. N. Smith, "Cyclical learning rates for training neural networks," in *IEEE Winter Conference on Applications of Computer Vision (WACV)*, 2017, pp. 464–472.