Gradient Based Optimization

Zihao Zeng

May 2, 2025

1 Introduction

This report investigates gradient-based optimisation techniques for training an Artificial Neural Network (ANN) to classify images of handwritten digits from the MNIST dataset. The network under consideration is a multi-layer perceptron (MLP) that consists of an input layer with 784 neurons, three hidden layers with 300, 100, and 100 neurons respectively, and an output layer with 10 neurons corresponding to the digit classes (0–9).

Each MNIST image, originally a 28×28 pixel grid, is first flattened into a vector

$$x \in [0,1]^{784}$$

by normalising the pixel intensities from 0 to 255. The forward propagation through the network is computed in two stages:

1. **Hidden Layers:** Each hidden neuron computes a weighted sum of the inputs from the previous layer followed by the application of a non-linear activation function. The Rectified Linear Unit (ReLU) is used for the hidden layers, defined as:
$$h_j^{(n)} = \max\left(0, \sum_{i=1}^{N_{n-1}} w_{ij}^{(n)}\, h_i^{(n-1)}\right) \quad \text{with} \quad h^{(0)} = x,$$
where $w_{ij}^{(n)}$ represents the synaptic weight connecting neuron i from layer $n-1$ to neuron j in layer n, and N_{n-1} denotes the number of neurons in the $(n-1)^{th}$ layer.

2. **Output Layer:** The output layer produces logits that are converted into a probability distribution using the softmax function:
$$S_j = \frac{e^{h_{O_j}}}{\sum_{k=1}^{N_O} e^{h_{O_k}}},$$
where S_j is the probability of the input image belonging to class j, h_{O_j} is the output neuron activation, and $N_O = 10$ is the total number of output neurons.

To ensure stable training, the network weights are initialised with a uniform distribution following the approach proposed by Glorot and Bengio [1]:

$$w_{ij} \sim \mathcal{U}\left(-\sqrt{\frac{6}{m+n}}, \sqrt{\frac{6}{m+n}}\right),$$

where m and n denote the numbers of neurons in the previous and current layers, respectively.

2 Part I: Stochastic Gradient Descent

Stochastic Gradient Descent [2] (SGD) is a fundamental optimisation algorithm widely used for training neural networks. Its primary goal is to minimise an objective function—in this case, the average loss over the training dataset—by iteratively updating the model's weight parameters. Unlike traditional gradient descent, which computes the gradient using the entire dataset, SGD estimates the gradient using a small subset of the data (a mini-batch), thereby significantly reducing the computational cost per update and introducing beneficial stochasticity into the learning process.

Assume the objective function is defined as the average loss over n training samples:

$$f(w) = \frac{1}{n} \sum_{i=1}^{n} L_i(x_i, w)$$

where $L_i(x_i, w)$ is the loss for the i^{th} sample x_i with model parameters w. The full gradient of this function is given by:

$$\nabla f(w) = \frac{1}{n} \sum_{i=1}^{n} \nabla L_i(x_i, w).$$

To overcome the computational overhead of processing the complete dataset at each update, SGD approximates this gradient using a mini-batch of m training samples. The corresponding weight update rule is formulated as:

$$w = w - \eta \left(\frac{1}{m} \sum_{i=1}^{m} \nabla L_i(x_i, w) \right),$$

where: η is the learning rate that governs the magnitude of the update and $\nabla L_i(x_i, w)$ represents the gradient of the loss for the i^{th} training sample.

The main reasons for employing SGD include:

- **Computational Efficiency**: By using only a fraction of the data for each update, SGD significantly reduces the amount of computation required per iteration [3].
- **Noise Regularization**: Stochastic updates use smaller learning rates to temper gradient noise, yielding smoother convergence [4]; this noise also serves as an implicit regularizer, and averaged stochastic gradients converge to the true gradient over time [5].
- **Flat Minima** By using small batches, SGD avoids the sharp minima that large-batch methods often converge to, and sharp minima that degrade model quality and impair generalization [6, 7].

2.1 Stochastic Gradient Descent Implementation

A function named `update_parameters()` completes the SGD implementation by updating the weight matrices using the rule:

$$w = w - \eta \frac{dw}{m},$$

where η (the learning rate) is set to 0.1 and m (the batch size) is set to 10. The function sequentially calls `update_weights()` for each weight matrix associated with the network layers:

- $W_{LI,L1}$ (Input to first hidden layer),
- $W_{L1,L2}$ (First to second hidden layer),
- $W_{L2,L3}$ (Second to third hidden layer), and
- $W_{L3,LO}$ (Third hidden to output layer).

Each update subtracts the averaged gradient from the current weight, and then resets the gradient accumulator to zero, thus ensuring that the parameter updates correctly reflect the gradient descent step derived from the mini-batch.

2.2 Validation through Numerical Differentiation

Numerical differentiation is used to validate the analytically computed gradients. Three methods are implemented:

- **Forward Difference:** Approximates the gradient by perturbing the weight positively:

$$\frac{\partial f}{\partial w} \approx \frac{f(w + \epsilon) - f(w)}{\epsilon}$$

- **Backward Difference:** Approximates the gradient by perturbing the weight negatively:

$$\frac{\partial f}{\partial w} \approx \frac{f(w) - f(w - \epsilon)}{\epsilon}$$

- **Central Difference:** Uses both forward and backward perturbations to obtain a more accurate estimate:

$$\frac{\partial f}{\partial w} \approx \frac{f(w + \epsilon) - f(w - \epsilon)}{2\epsilon}$$

In these formulas, ϵ is a small constant used to perturb the weights, with $\epsilon = 10^{-8}$. These methods provide numerical approximations that can be compared against the analytical gradients to ensure the correctness of the backpropagation implementation.

2.3 Results and Discussion

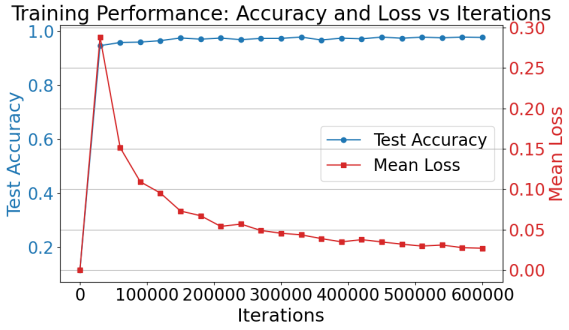

Figure 1: Training Performance: Loss and Accuracy over Iterations

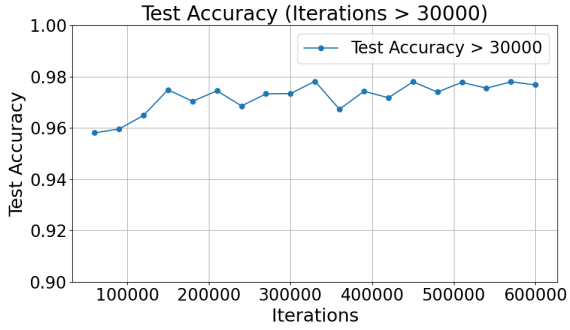

Figure 2: Detailed Accuracy after 30000 Iterations

Figure 1 compares the gradient validation results using forward, backward, and central differencing methods. The central difference method yields the lowest average relative difference because its symmetric formulation cancels the first-order error terms, providing a second-order accurate estimate [8]. In contrast, the forward and backward difference methods are only first-order accurate. This enhanced accuracy of the central difference method justifies its preference for gradient validation. Meanwhile, the average relative difference results are small enough to confirm that the analytical gradients are correct, as all methods yield values below 0.0003%.

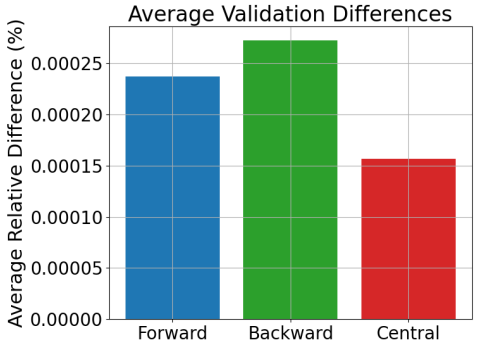

Table 1: Gradient Validation: Relative Difference Comparison

Regarding the compute time, the numerical methods incur significant computational cost since each estimate requires multiple function evaluations per parameter. Based on the empirical results, the forward differencing method takes approximately 0.52123 seconds per sample, the backward differencing method requires 0.52331 seconds per sample, and the central differencing method incurs a cost at 0.52957 seconds per sample. On the other hand, the analytical gradient computation, achieved through backpropagation, computes the full gradient vector in a single, efficient backward pass, making it substantially faster.

Figure 1 reveals that, starting near 10% at iteration 0, the test accuracy rapidly surpasses 90% by 30,000 iterations, reflecting the network's rapid acquisition of fundamental features. Beyond 30,000 iterations, the loss decreases from around 0.3 to below 0.05 as the optimiser fine-tunes the parameters. Figure 2 focuses on iterations exceeding 30,000, where the accuracy fluctuates between approximately 96% and 98%. This plateau indicates that SGD has converged to a (local) optimum.

3 Part II: Improving Convergence

In Part I, basic SGD shows instability near the optimum, with accuracy fluctuating between 96%–98%. This highlights the need to improve convergence and enable more stable and effective fine-tuning near the optimum.

3.1 Hyper-Parameter Analysis

In order to systematically investigate the convergence behavior of SGD, this experiment varies two key hyper-parameters: learning rate and batch size. The selected ranges are: **Learning Rates:** $\eta = 0.1, 0.01, 0.001$ and **Batch Sizes:** $m = 1, 10, 100$.

3.1.1 Results and Discussion

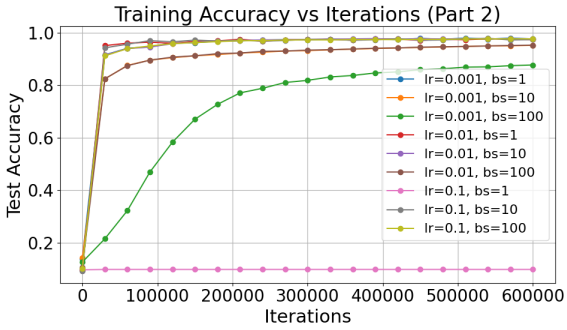

Figure 3: Accuracy across Iterations.

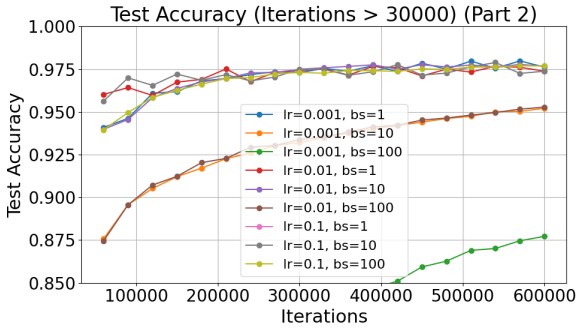

Figure 4: Accuracy after 30000 Iterations.

Figures 3 and 4 illustrated the training accuracy over iterations and a more detailed view of the accuracy after 30,000 iterations, respectively. The Table 2 summarizes the performance metrics for different combinations of learning rates and batch sizes.

Notably, when using a learning rate of 0.1 with a batch size of 1, the model records an extremely low accuracy (9.8%). In contrast, increasing the batch size to 10 or 100 with the same learning rate leads to significant improvements: the loss reduces to approximately 0.0253 and 0.0161, while accuracy reaches around 97.7%. These configurations also slightly reduce computation time, suggesting that larger batch sizes not only stabilize learning but also enhance computational efficiency.

η	m	**Loss**	**Accuracy**	**Time (s)**
0.1	1	2.3026	0.0980	589
0.1	10	0.0253	0.9774	555
0.1	100	0.0161	0.9768	544
0.01	1	0.0253	0.9738	588
0.01	10	0.0167	0.9769	551
0.01	100	0.1591	0.9528	547
0.001	1	0.0166	0.9762	591
0.001	10	0.1603	0.9521	558
0.001	100	0.4820	0.8772	546

Table 2: Final performance metrics at epoch 10

Meanwhile, Smaller learning rates (e.g., 0.001) produce smoother curves as each update nudges the weights more gently. Similarly, larger batch sizes (e.g., 100) average out noise across more samples, reducing gradient variance and stabilizing the learning trajectory. Conversely, larger learning rates (e.g., 0.1) introduce oscillations, which are also observed with small batch sizes (e.g., 1).

Moreover, pairing lower learning rates with a batch size of 100 yields higher losses (0.1591 for η=0.01 and 0.4820 for η=0.001) and a reduction in accuracy (roughly 95.3% and 87.7%, respectively). Combined with the plots, they indicate that the training process has not yet been given sufficient iterations to fully converge.

The optimal configuration in these experiments appears to be a high learning rate (0.1) combined with a large batch size (100), which achieves enough rapid convergence with a smooth trajectory and high overall accuracy, while managing computational time effectively.

3.2 Learning Rate Decay

A high initial learning rate can accelerate convergence by allowing the optimizer to take large steps, but it may hinder fine-tuning as the optimum is approached. When near a minimum, maintaining a high learning rate risks

overshooting the optimum and could cause oscillations rather than convergence. To address this issue, it is common to gradually reduce the learning rate—a process known as learning rate decay. This reduction allows the optimisation algorithm to take smaller, more precise steps during the later stages of training, effectively fine-tuning the parameters as the loss landscape flattens near the optimum [9].

3.2.1 Learning Rate Decay Implementation

In the implementation, the learning rate is linearly interpolated from an initial value $\eta_{\text{init}} = 0.1$ to a predefined final value $\eta_{\text{final}} = 10^{-4}$.

At the end of each epoch, the current epoch counter is incremented. When learning rate decay is enabled, the decay parameter α is computed as the ratio of the current epoch number to the total number of epochs:

$$\alpha = \frac{k}{N},$$

where k is the current epoch and N is the total number of epochs.

Using this decay parameter, the learning rate for the next epoch is updated using the following formula:

$$\eta_k = \eta_0(1 - \alpha) + \alpha\eta_N,$$

where η_0 is the initial learning rate, η_N is the final learning rate, η_k is the learning rate at epoch.

3.2.2 Results and Discussion

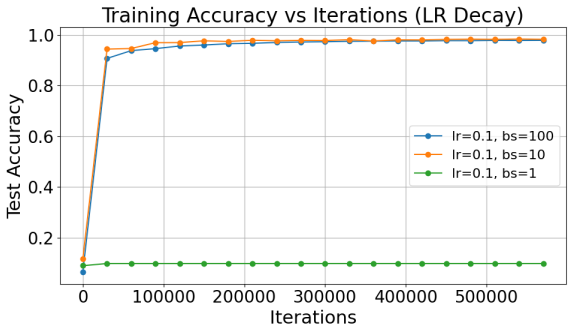

Figure 5: Accuracy across Iterations with Learning Rate Decay.

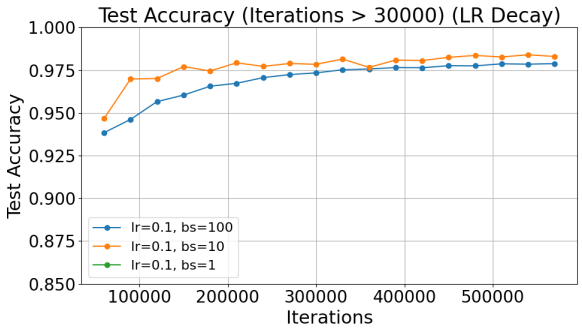

Figure 6: Accuracy after 30000 Iterations with Learning Rate Decay.

For Figures 5, 6, and Table 3, several important observations regarding the effect of learning rate decay can be drawn. First, the configuration employing a batch size of 1—whether with a decayed learning rate from 0.1 to 0.001 or with a constant learning rate of 0.1—consistently results in very low accuracy (9.8%).

η_0	η_N	m	**Loss**	**Accuracy**
0.1	0.0001	1	2.3026	0.0980
0.1	0.0001	10	0.0017	0.9827
0.1	0.0001	100	0.0311	0.9788

Table 3: Performance metrics at epoch 10 with learning rate decay.

In contrast, the configurations with larger batch sizes (10 and 100) display improvements in performance, comparing when no learning rate decay employed. Specifically, the configuration with a batch size of 10 achieves an accuracy of 98.30% (from 97.74%) with a loss of 0.0020, while the batch size of 100 yields an accuracy of 97.88% (from 97.68%) and a loss of 0.0311. The smoother training accuracy curves observed in these cases indicate that learning rate decay contributes to more gradual and stable convergence.

3.3 Momentum

Momentum is a technique designed to accelerate convergence and mitigate oscillations during optimization by incorporating an exponentially decaying moving average of past gradients [10]. By accumulating a history of gradient information, momentum helps the optimizer maintain consistent update directions, thereby smoothing the updates and enabling the algorithm to overcome small local minima.

3.3.1 Momentum Implementation

In implementation, momentum is incorporated through an additional velocity term v associated with each weight parameter w. The update rule for momentum is formulated as follows:

$$v = \alpha\, v - \eta \frac{1}{m} \sum_{i=1}^{m} \nabla L_i(x_i, w)$$

$$w = w + v$$

where η represents the learning rate, m is the batch size, $\nabla L_i(x_i, w)$ is the gradient of the loss with respect to the weight w for the i-th training example in the batch, α is the momentum coefficient.

The previous velocity is scaled by the momentum coefficient α and then decreased by the current gradient (averaged over the batch and scaled by the learning rate η). The computed velocity v is then added to the corresponding weight w, effectively smoothing the weight updates and promoting faster convergence.

3.3.2 Results and Discussion

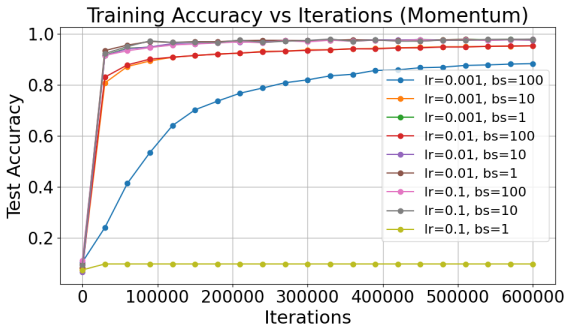

Figure 7: Accuracy across Iterations with Momentum.

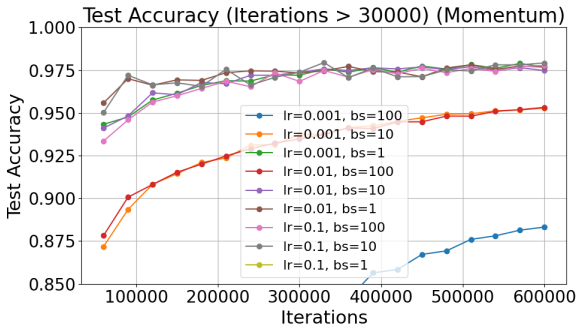

Figure 8: Accuracy after 30000 Iterations with Momentum.

In Figures 7, 8, and Table 4, several key observations can be drawn. Notably, regardless of whether momentum is applied, the configuration with a batch size of 1 and a learning rate of 0.1 again produces very low accuracy (9.8%). In contrast, all other configurations exhibit a slight improvement in accuracy compared to previous experiments without momentum, and their accuracy curves are slightly smoother. These findings support that a momentum coefficient of 0.1 is a sensible choice, and employing momentum can help to stabilize the training process.

α	η	m	**Loss**	**Accuracy**
0.1	0.1	1	2.3026	0.0980
0.1	0.1	10	0.0250	0.9792
0.1	0.1	100	0.0136	0.9778
0.1	0.01	1	0.0272	0.9768
0.1	0.01	10	0.0145	0.9748
0.1	0.01	100	0.1630	0.9532
0.1	0.001	1	0.0145	0.9768
0.1	0.001	10	0.1487	0.9528
0.1	0.001	100	0.4821	0.8772

Table 4: Performance metrics at epoch 10 with momentum.

3.4 Combined Techniques: Learning Rate Decay with Momentum

Integrating learning rate decay with momentum might leverage the strengths of both approaches. Learning rate decay adapts the step size during training, grained updates when close to convergence. Concurrently, momentum incorporates a moving average of past gradients, which smooths the update trajectory and helps in surmounting shallow local minima or flat regions. The fusion of these techniques allows the optimizer to maintain directionality and stability, even as the step size diminishes, thereby reducing oscillations and overshooting near the minimum.

3.4.1 Results and Discussion

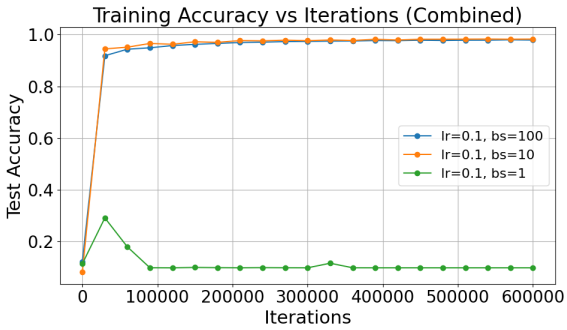

Figure 9: Accuracy across Iterations with Combined Techniques.

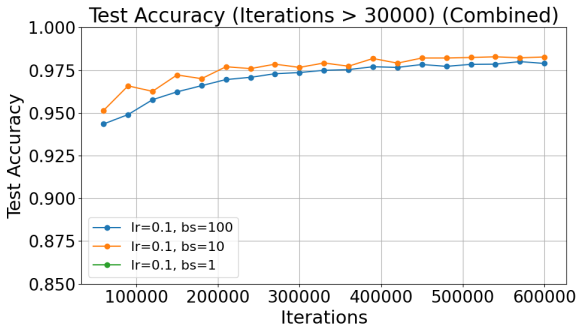

Figure 10: Accuracy after 30000 Iterations with Combined Techniques.

Figures 9, 10, and Table 5 demonstrate an improvement over previous experimental configurations. Notably, the configuration that yielded the highest performance—achieving 98.28% accuracy and a minimum loss of 0.0016—was obtained with a batch size of 10, an initial learning rate of 0.1 decaying to a final learning rate of 0.0001, and a momentum coefficient of 0.1. This optimal setting effectively balances rapid early convergence with the fine-tuning capabilities required in later stages of training, underscoring the benefits of integrating learning rate decay with momentum.

α	η_0	η_N	m	**Loss**	**Accuracy**
0.1	0.1	0.0001	1	2.3026	0.0980
0.1	0.1	0.0001	10	0.0016	0.9828
0.1	0.1	0.0001	100	0.0271	0.9789

Table 5: Performance metrics at epoch 10 with combined techniques.

4 Part III: Adaptive Learning

4.1 Choice of Adaptive Method

In this work, the Adam optimizer [11] is selected due to its effective combination of momentum and adaptive learning rate methods. Adam computes individual step sizes for each parameter by maintaining exponential moving averages of both past gradients and squared gradients. This adaptation helps to handle sparse gradients and noisy data while ensuring fast convergence and robust performance. Notably, as reported by Tato and Nkambou [12], CNNs with Adam achieved a test accuracy of 99.12% on the MNIST dataset, underscoring its practical benefits.

4.2 Mathematical Formulation

Adam's update mechanism is defined by the following equations:

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

where g_t is the gradient at time t, and the decay rates for the first and second moment estimates are $\beta_1 = 0.9$ and $\beta_2 = 0.999$, respectively.

Bias-corrected estimates are computed as:

$$\hat{m}_t = \frac{m_t}{1 - \beta_1^t},$$
$$\hat{v}_t = \frac{v_t}{1 - \beta_2^t}.$$

The parameter update rule becomes:

$$w_{t+1} = w_t - \eta \frac{\hat{m}_t}{\sqrt{\hat{v}_t} + \epsilon},$$

where η is the learning rate and ϵ is a small constant to prevent division by zero. This framework enables Adam to adjust learning rates for each parameter adaptively, facilitating efficient convergence and improved performance.

4.3 Results and Discussion

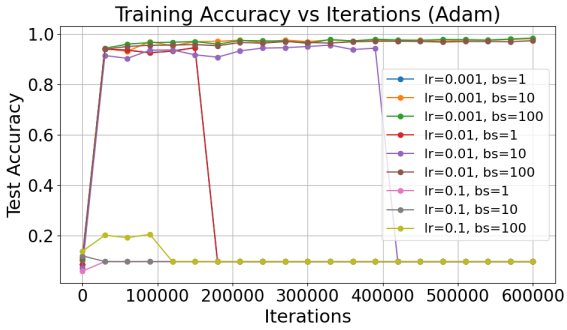

Figure 11: Accuracy across Iterations (Adam)

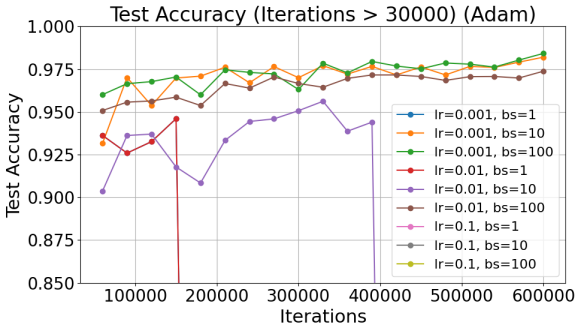

Figure 12: Detailed Accuracy after 30000 Iterations (Adam)

The results of the Adam optimizer are presented in Figure 11, 12, and Table 6. From the figures, the configuration with a learning rate $\eta = 0.01$ and a batch size $m = 100$, as well as that with $\eta = 0.001$ and $m = 10$ or 100, achieved convergence. However, these settings exhibited marked oscillatory behavior during training, suggesting that the adaptive updates still introduce significant fluctuations even when convergence is eventually attained.

η	m	**Loss**	**Accuracy**
0.01	100	0.0780	0.9738
0.001	10	0.0299	0.9820
0.001	100	0.0171	0.9842

Table 6: Performance metrics at epoch 10 of the Adam optimizer.

In contrast, other parameter combinations using Adam either failed to converge or experienced collapse during the convergence process. This behavior highlights the sensitivity of Adam to hyperparameter choices, especially in scenarios involving small batch sizes or aggressive learning rates [13].

When comparing with SGD, only the scenario with an large learning rate ($\eta = 0.1$) combined with a small batch size ($m = 1$), SGD failed to convergence. Overall, SGD demonstrated a higher degree of robustness under varied hyperparameter settings. And for the configuration with a batch size of 10, an initial learning rate of 0.1 decaying to a final learning rate of 0.0001, and a momentum coefficient of 0.1, SGD achieved the lowest loss of 0.0016 at epoch 10.

However, Adam's performance was superior in terms of final accuracy at epoch 10, as shown in Table 6. The best-performing configuration achieved an accuracy of 98.42%, surpassing the best SGD configuration (98.28%).

5 Conclusion

Limitations and Future Work This study identified several limitations for further investigation that could enhance performance and deepen understanding. Firstly, some experimental configurations did not fully converge within ten epochs; therefore, future work can consider extending the number of training epochs to determine whether these configurations eventually converge to superior results. Secondly, hyperparameter space are not fully explored, some hyperparameter optimization techniques like bayesian optimization [14] may further improve model performance. Thirdly, given the observed benefits of learning rate decay, increasing batch sizes appears promising [15]. Finally, the Nadam optimizer, which integrates Adam with Nesterov momentum, may yield enhanced performance [12].

Generalization to Other Problems Adam's adaptive learning capabilities generalize effectively across domains beyond image classification. It is the optimizer of choice for natural language processing models like Transformer [16]. It is also utilized in time series forecasting with LSTMs and GRUs [17]. Furthermore, Adam is widely adopted in reinforcement learning algorithms, where its adaptive learning rates and momentum terms facilitate stable and efficient training of policy and value networks in dynamic environments [18].

References

[1] Xavier Glorot and Yoshua Bengio. Understanding the difficulty of training deep feedforward neural networks. In Yee Whye Teh and Mike Titterington, editors, *Proceedings of the Thirteenth International Conference on Artificial Intelligence and Statistics (AISTATS)*, volume 9 of *Proceedings of Machine Learning Research*, pages 249–256, Chia Laguna Resort, Sardinia, Italy, May 2010. PMLR.

[2] Herbert Robbins and Sutton Monro. A stochastic approximation method. *The Annals of Mathematical Statistics*, 22(3):400–407, 1951.

[3] Léon Bottou. Large-scale machine learning with stochastic gradient descent. In Yves Lechevallier and Gilbert Saporta, editors, *Proceedings of COMPSTAT'2010: 19th International Conference on Computational Statistics*, pages 177–186, Paris, France, 2010. Physica-Verlag HD.

[4] D. Wilson and T. Martinez. The general inefficiency of batch training for gradient descent learning. In *Neural Networks*, 2003.

[5] I. Goodfellow, Y. Bengio, and A. Courville. *Deep Learning*. MIT Press, 2016.

[6] D. Mishkin, N. Sergievskiy, and J. Matas. Systematic evaluation of convolution neural network advances on the imagenet. In *Computer Vision and Pattern Recognition Workshops*, 2017.

[7] N. S. Keskar, D. Mudigere, J. Nocedal, M. Smelyanskiy, and P. T. P. Tang. On large-batch training for deep learning: Generalization gap and sharp minima. In *International Conference on Learning Representations*, 2017.

[8] Richard L. Burden and J. Douglas Faires. *Numerical Analysis*. Brooks/Cole, 7th edition, 2000.

[9] Kaifeng You, Mingsheng Long, Jianmin Wang, and Michael I. Jordan. How does learning rate decay help modern neural networks? *arXiv preprint arXiv:1908.01878*, 2019.

[10] Jingwen Fu, Bohan Wang, Huishuai Zhang, Zhizheng Zhang, Wei Chen, and Nanning Zheng. When and why momentum accelerates sgd: An empirical study. *arXiv preprint arXiv:2306.09000*, 2023.

[11] Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

[12] Ange Tato and Roger Nkambou. Improving adam optimizer. In *Workshop Track at the 6th International Conference on Learning Representations (ICLR)*, 2018.

[13] Shuaipeng Li, Penghao Zhao, Hailin Zhang, et al. Surge phenomenon in optimal learning rate and batch size scaling. *arXiv preprint arXiv:2405.14578*, 2024.

[14] P. I. Frazier. A tutorial on bayesian optimization. *arXiv preprint arXiv:1807.02811*, 2018.

[15] Samuel L. Smith, Pieter-Jan Kindermans, Chris Ying, and Quoc V. Le. Don't decay the learning rate, increase the batch size. *arXiv preprint arXiv:1711.00489*, 2017.

[16] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. In *Advances in Neural Information Processing Systems (NeurIPS)*, pages 5998–6008, 2017.

[17] A. Makinde. Optimizing time series forecasting: A comparative study of adam and nesterov accelerated gradient on lstm and gru networks using stock market data. *arXiv preprint arXiv:2410.01843*, 2024.

[18] K. Asadi, R. Fakoor, and S. Sabach. Resetting the optimizer in deep rl: An empirical study. In *Advances in Neural Information Processing Systems*, volume 36, pages 72284–72324, 2023.