COMP36212 EX3 ANN Gradient-Based Optimisation

Euan Liew

1 Introduction

In this report, different optimisation techniques of an Artificial Neural Network(ANN) will be explored for the image classification problem on the MNIST dataset. The MNIST dataset has 70,000 images of handwritten digits (0-9) in 28x28 pixels and has intensity values ranging from 0-255 [ref]. It will be split into 60,000 images for training and 10,000 for testing, while each sample will be flattened into an array of 784 elements.

With the main objective being to maximise the classification accuracy, the process will involve computing a prediction from a sample input, comparing the prediction of the network with the true label to formulate an objective function for optimisation given as $L = -\sum_k^N \hat{y}_k \log(P_k)$ and using optimisation techniques to minimise the objective function with respect to the network parameters.

1.1 ANN Definition

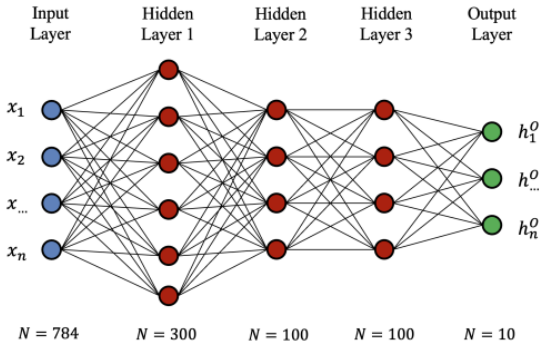

Figure 1: Architecture of the ANN used

The network takes image input flattened into 784 values (28×28 pixels) and passes it through three fully connected hidden layers with 300, 100 and 100 neurons respectively. Each neuron computes its activation using a weighted sum of inputs followed by an activation function. For all hidden layers, the Rectified Linear Unit (ReLU) activate function is used, which outputs zero for negative inputs and the input itself if positive. The output layer contains 10 neurons, each representing a digit class. It applies the softmax function to convert raw activation values into a probability distribution, enabling classification.

2 Stochastic Gradient Descent (SGD)

Stochastic Gradient Descent (SGD) is an optimisation algorithm that is used in the training of neural networks. Instead of calculating the gradient of the loss function using the entire dataset like in Gradient Descent, mini-batch approximations are used. The core idea is to iteratively adjust the parameters in the direction that decreases the overall loss [1] [10].

Optimisation rule is given as:

$$w_{ij} = w_{ij} - \eta \frac{1}{m} \sum_{k=1}^{m} \frac{\partial L_k}{\partial w_{ij}} \tag{1}$$

where w_{ij} is the weight being updated, η is the learning rate , m is the mini-batch size, and $\sum_{k=1}^{m} \frac{\partial L_k}{\partial w_{ij}}$ is the accumulated gradient for w_{ij} over the m samples. The implementation in the code is as follows:

Algorithm 1 Mini-Batch Stochastic Gradient Descent

Input: Training data D, learning rate η, batch size m, number of epochs E
Output: Trained neural network weights
Initialise network weights randomly **Validate** gradients using numerical differentiation
while *not completed all epochs* **do**
  for *each mini-batch of size m* **do**
    for *each example in mini-batch* **do**
      Forward pass: Compute network outputs and loss
      Backward pass: Calculate gradients $\frac{\partial L}{\partial w_{ij}}$ for all weights Accumulate gradients for batch update
    end
    for *all weights w_{ij} in network* **do**
      Apply update: $w_{ij} = w_{ij} - \eta \frac{1}{m} \sum_{k=1}^{m} \frac{\partial L_k}{\partial w_{ij}}$ then **Reset** accumulated gradients
    end
  end
  Calculate loss and accuracy on test set Record and report metrics
end

2.1 Gradient Calculation Validation

Numerical Differentiation : To validate the calculated gradients, three finite difference methods were used to verify the correctness of the analytical gradients, where $\epsilon = 10^{-8}$:

1.Forward Difference Method **2.Backward Difference Method** **3.Central Difference Method**

$$\frac{\partial L}{\partial w_{ij}} \approx \frac{L(w_{ij}+\epsilon) - L(w_{ij})}{\epsilon} \qquad \frac{\partial L}{\partial w_{ij}} \approx \frac{L(w_{ij}) - L(w_{ij}-\epsilon)}{\epsilon} \qquad \frac{\partial L}{\partial w_{ij}} \approx \frac{L(w_{ij}+\epsilon) - L(w_{ij}-\epsilon)}{2\epsilon}$$

Due to high computational costs of each method, the numerical gradients for a single sample were only computed once at the first training epoch and then compared to the analytical method. This approach balances implementation correctness and training efficiency without incurring computational cost during subsequent iterations. These methods were experimented using the values: learning rate $\eta = 0.1$ and batch size m = 10 for 10 epochs :

Table 1: Comparison of Numerical Differentiation Methods with Analytical Gradients

Method	Mean Abs. Error	Mean Rel. Error	Max Abs. Error	Max Rel. Error
Forward Difference	2.10×10^{-8}	0.000201%	1.00×10^{-7}	0.001046%
Backward Difference	2.18×10^{-8}	0.000227%	1.00×10^{-7}	0.001733%
Central Difference	8.63×10^{-9}	0.000083%	1.00×10^{-7}	0.000702%

Ref to Table 1, using output data from the validate_gradient function implemented in the optimiser.c, error metrics were calculated for each method. The central difference method provides the most accurate approximation despite the computational cost, resulting in a mean absolute error approximately 2.4× smaller than forward or backward differences. This aligns with theoretical expectations that it is second-order accurate, while the other two are first-order accurate [11]. The low error rates between analytical and numerical gradients ($< 0.0003\%$) confirm the correctness of the backpropagation implementation. In addition, the analytical solution took 0.001s whereas numerical took 0.076s for the first training epoch, underscoring the computation efficiency of the analytical method.

2.2 Performance Analysis

To assess the performance of SGD, the following hyperparameters will be experimented on:

1. **Learning Rate** η is the step size of the SGD. Using a large rate may risk exponential changes, resulting in overshooting the minima while converging to a subobtimal solution. In contrast, a small learning rate would require running more epochs, risking local minima stagnation.

2. **Batch Size** m is the number of training data used to calculate the gradient at each iteration. Using a batch size of 1 enables frequent and noisy updates that may help escape local minima but slows convergence, while batch sizes of 10 or 100 reduces update variance for more stable training at the cost of increased memory.

For the first run, the learning rate was set to 0.1, the batch size set to 10 with epochs fixed at 10. The results were then plotted as shown in Figure 2. At the initial stage before Epoch 1, the loss function decreases sharply, while the test accuracy increases rapidly due to the relatively large learning rate used. The consistent decrease in loss and increase in accuracy confirms that the SGD implementation effectively optimises the ANN. The loss decreases monotonicallv from 0.180 to 0.022. with the rate of decrease gradually slowing over time.

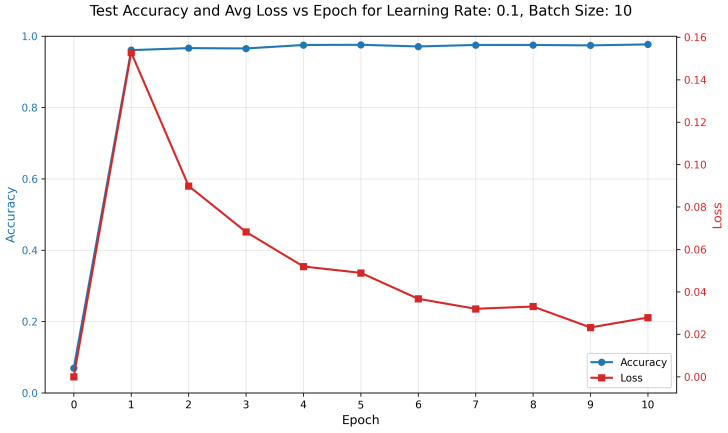

Figure 2: Loss and Accuracy plot for $m = 10$ $\eta = 0.1$ 10 epochs

Figure 3: Training Metrics

Epoch	Loss	Test Accuracy
0	0.180	0.683
1	0.116	0.962
2	0.090	0.970
3	0.075	0.974
4	0.058	0.978
5	0.050	0.980
6	0.042	0.981
7	0.035	0.982
8	0.032	0.984
9	0.026	0.985
10	0.022	0.985

2.2.1 Has the Algorithm Found an Optimum?

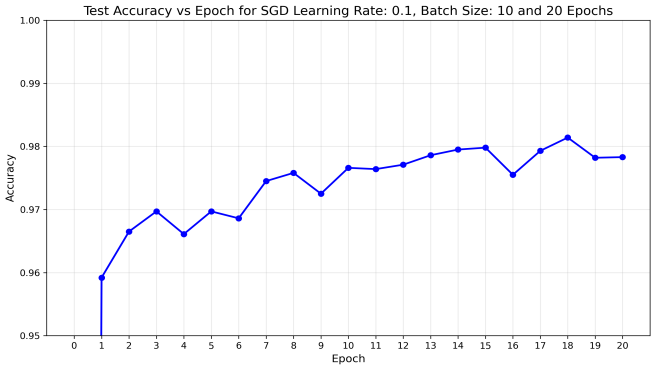

Figure 4: Accuracy plot for $m = 10$ $\eta = 0.1$ 20 epochs

To assess convergence, training was extended to 20 epochs. Fig 4 reveals that accuracy exhibits persistent oscillatory behavior throughout training. Despite reaching approximately 98.5% accuracy by epoch 10, performance fluctuates between epochs 13-20 with no consistent improvement beyond this threshold.

This pattern indicates the algorithm has identified a region containing a local optimum but the fixed learning rate ($\eta = 0.1$) prevents convergence. The oscillations suggest SGD with current hyperparameters has achieved practical convergence rather than true convergence where $\bigtriangledown L(w) \approx 0$.

The algorithm effectively orbits an optimum point rather than settling precisely at it. To determine the true convergence, alternative strategies such as learning rate decay or momentum-based methods will be explored in the following sections below to allow finer parameter adjustments.

3 Improving Convergence

Effectiveness of SGD is influenced by the choice of hyperparameters: learning rate (η) and mini-batch size (m). This section analyses different hyperparameter configurations to assess their effect on training performance. For this experiment, a grid search was conducted across the combinations of : $\eta \in \{0.001, 0.01, 0.1\}$ and $m \in \{1, 10, 100\}$ for 10 epochs. The test accuracy, average loss and execution time is then recorded to benchmark the performance.

3.1 Testing Different Hyperparameter Configurations

Table 2: Training Metrics

η	m	Avg Loss	Test Acc	t (s)
0.100	1	2.302	0.0980	1064.00
0.100	10	**0.0279**	**0.9771**	829.44
0.100	100	0.0186	0.9763	884.76
0.010	1	0.0240	**0.9787**	1031.98
0.010	10	0.0165	0.9763	922.87
0.010	100	0.1647	0.9512	908.72
0.001	1	**0.0162**	**0.9777**	1091.24
0.001	10	0.1646	0.9516	886.39
0.001	100	0.4777	0.8797	795.48

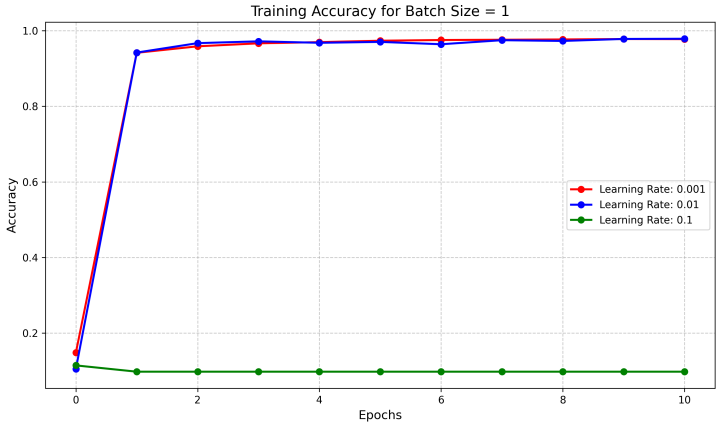

Figure 5: Accuracy for $m = 1$ and varying η

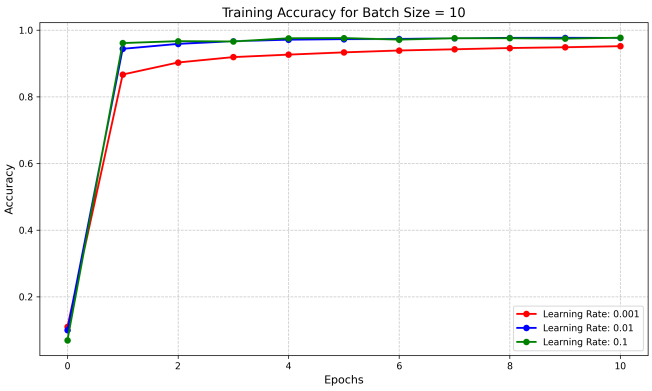

Figure 6: Accuracy for $m = 10$ and varying η

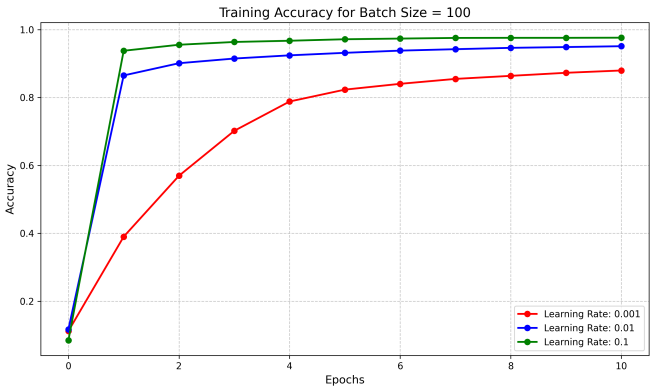

Figure 7: Accuracy for $m = 100$ and varying η

3.1.1 Experiment Observations

Impact of Learning Rate : Table 2 shows that for all batch sizes, a high learning rate ($\eta = 0.1$) produces the fastest convergence but is unstable when batch size = 1 (NaN loss and 9.8% accuracy) but for $\eta = 0.01$ and 0.001, it remains stable. At m = 1, reducing η from 0.1 to 0.01 decreases the average loss of NaN to 0.0240 and increases the precision from 9. 8% to 97. 87%. In addition, lowering η further to 0.001 improves loss slightly (0.0162) but comparable accuracy (97.77%) . This is because large step sizes lead to divergence [4], while small rates slow progress [2].

Impact of Batch Size : Figure 6 shows that small batches (m = 1) suffer from a high variance in the estimation of the gradient that requires moderate η (0.01–0.001) to stabilise learning, while m = 10 and 100 benefit from a higher η to offset the reduced update frequency. In Fig. 7 (m = 10), $\eta = 0.1$ reaches 97.7% accuracy by epoch 5, but in Fig. 8 (m = 100), $\eta = 0.01$ stalls at 95.1% and $\eta = 0.001$ at 87.9%, indicating that the effective step size η and m is too small. Increasing m reduces gradient noise but requires adjustments to η for avoiding sharp-minima traps [2].

Computational Efficiency : From Table 6 execution times, m = 1 incurs the greatest overhead (≈ 1,032–1,091 s) due to the updates per sample, while m = 10 reduces runtime by 20 % (≈ 829–922 s) and m = 100 adds only marginal gains (≈ 795–909 s). This reflects the diminishing returns of batch parallelism once GPU/CPU utilisation plateaus [3].

In conclusion, the configuration $\eta = 0.1$ with m = 100 balances as the optimal configuration (97.7% accuracy with 8–15% faster training than smaller batches and avoiding the instability seen at m = 1.

3.2 Decay LR

Learning rate decay is an optimisation technique that improves convergence by reducing the step size as training progresses, reducing it exponentially over time. This helps control convergence and prevents overshooting, helping models stabilise and converge smoothly [5]. This decay rate formula is defined as:

$$\eta_k = \eta_0(1 - \alpha) + \alpha\eta_N \quad \text{where} \quad \alpha = \frac{k}{N} \tag{2}$$

where η_0 = final learning rate, η_0 = initial learning rate, k = current epoch, and N = total number of epochs.

3.2.1 Experimentation and Analysis

For this experiment, various batch sizes are experimented where $m \in \{1, 10, 100\}$, different initial learning rates and final learning rates over 10 epochs. Referring to Table 3, starting with the initial rate of $\eta_0 = 0.01$ and final rate $\eta_N = 0.001$, it achieved a strong test accuracy of 0.9778 % and an average loss of 0.0288 at batch size 10. However, increasing the batch size to 100 under the same conditions worsened results as accuracy dropped while average loss increased, suggesting reduced update frequency hinders convergence at small learning rates [6].

When increasing the initial learning rate η_0 to a more aggressive 0.1, results were unstable (accuracy close to random and NaN loss) for batch size 1 likely due to noisy updates and volatile gradients. Interestingly by following Smith's 1-cycle policy where an aggressive initial rate (0.1) is used and then decaying to a fine-grained final rate (0.0001), this improved both accuracy and average loss [6].

Figure 8 demonstrates how the combinations of $\eta_0 = 0.01$ and $\eta_N = 0.001$ results in the smoothest curve. This stability reflects how smaller updates prevent oscillations at the cost of a lower final accuracy, highlighting the trade-off between stability and performance. In contrast, configurations with $\eta_0 = 0.1$ (blue and green lines) still oscillate between epochs 3-7. By epoch 10, the red line shows stable convergence with its monotonic trajectory, while blue and green still oscillate around the optimal solution. This demonstrates how learning rate parameters influence the balance between performance and stability.

η_0	η_N	m	Avg Loss	Test Acc
0.01	0.001	10	**0.0288**	**0.9778**
0.01	0.001	100	0.2101	0.9399
0.100	0.001	1	2.3026	0.0980
0.100	0.001	10	**0.0015**	**0.9834**
0.100	0.001	100	0.0320	0.9774
0.100	0.0001	1	2.3026	0.0980
0.100	0.0001	10	0.0018	0.9834
0.100	0.0001	100	0.0319	0.9783

Table 3: Decay Results for 10 epochs

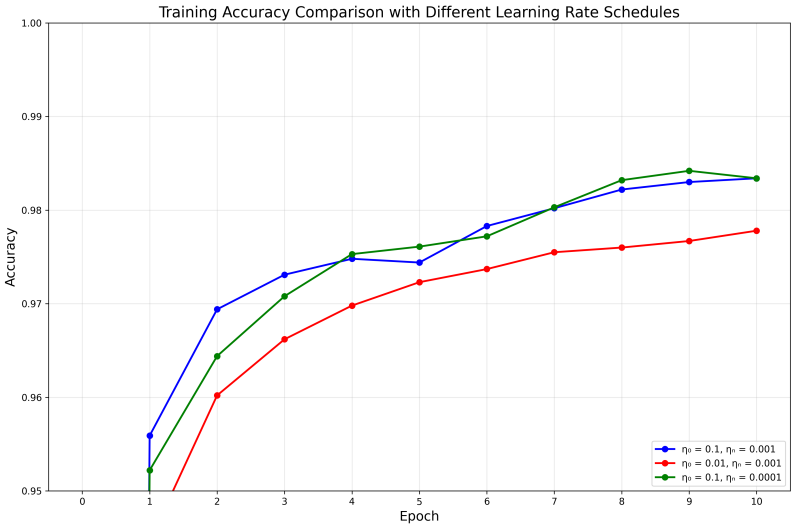

Figure 8: Combined Accuracy for different decays

3.3 Momentum

Another optimisation technique explored was applying SGD-with-momentum (SGDM) as analysed by Liu et al [7]. This techniques performs optimisation by tracking the accumulated gradients from previous time steps and continues moving in a consistent direction. This allows the algorithm to escape local minima and find the global minimum. Its equation can be defined as follows:

$$v_t = \alpha v_{t-1} + \eta \frac{1}{m} \sum_{i=1}^{m} \nabla L_i(x_i, w) \tag{3}$$

The weights are updated as follows:

$$w = w - v_t \tag{4}$$

where v_t is the velocity term at time t, α is the momentum coefficient, v_{t-1} is the velocity term at the previous time step, η is the learning rate, m the batch size, $\nabla L_i(x_i, w)$ is the gradient of the loss function for the i-th example in the batch $w.r.t$ weight w.

3.3.1 Experimentation and Analysis

For this experiment, the learning rate $\eta \in \{0.001, 0.01, 0.1\}$, batch size $m \in \{1, 10, 100\}$, and momentum coefficient $\alpha \in \{0.6, 0.9\}$ are varied to explore its effects on test accuracy and average loss. Referring to Table 4 below, when $\eta = 0.001$ and $m = 10$, the higher momentum value $\alpha = 0.9$ achieves better performance (97.76% accuracy, 0.0010 loss) compared to $\alpha = 0.6$ (96.84% accuracy, 0.0850 loss). This is because larger α values accelerates convergence with smaller step sizes by amplifying gradient directionality[7].

However, this is not a universal pattern across all configurations. When η is increased to 0.010 with the same batch size ($m = 10$), the relationship inverts. The lower coefficient $\alpha = 0.6$ slightly outperforms $\alpha = 0.9$ in both accuracy and loss. This is due to excessive momentum "overshoot" as step sizes grow, whereby a smaller η yields finer control near minima [7].

The batch size also plays a role in determining optimal momentum values. With a learning rate of $\eta = 0.100$ and batch size $m = 100$, $\alpha = 0.6$ achieves better results (97.62% accuracy, 0.0114 loss) than $\alpha = 0.9$ (97.35% accuracy, 0.0353 loss). Larger batches reduce gradient noise and thus require less momentum smoothing to maintain stable updates [6].

Interestingly, certain configurations fail with high learning rates ($\eta = 0.100$) and small batch sizes ($m = 10$). Both momentum coefficients result in poor convergence (9.80% accuracy, 2.3026 loss). This highlights the importance of balancing the hyperparameters momentum, learning rate, and batch size.

Overall, these results demonstrate that momentum improves SGD convergence across most configurations but its optimal value depends on both learning rate and batch size. Higher η offers rapid initial descent, while lower η provides smoother fine-tuned updates as mentioned by Liu et al [7].

Examining Figure 7, while $\alpha = 0.9$ achieved better accuracy throughout training, $\alpha = 0.6$ demonstrated smoother progression. The smoothness occurs because the lower momentum coefficient accumulates less velocity from previous gradients, resulting in more measured parametric updates. This creates a more stable optimisation trajectory lesser oscillations. In contrast, the $\alpha = 0.9$ configuration accumulates the gradient more aggressively, causing early acceleration but higher sensitivity towards local features in the loss landscape. Thus, higher momentum values offer faster initial convergence and better performance, while lower values provide smoother training dynamics.

η	m	Test Acc	Avg Loss	α
0.001	1	0.9764	0.0277	0.9
0.001	10	**0.9776**	**0.0010**	0.9
0.001	10	0.9684	0.0850	0.6
0.001	100	0.9519	0.1613	0.9
0.010	1	0.0980	2.3026	0.9
0.010	10	0.9756	0.0327	0.9
0.010	10	0.9769	0.0149	0.6
0.010	100	0.9757	0.0200	0.9
0.100	10	0.0980	2.3026	0.9
0.100	100	0.9735	0.0353	0.9
0.100	100	0.9762	0.0114	0.6

Table 4: Results for Momentum Coefficient ($\alpha = 0.9$ *and* 0.6)

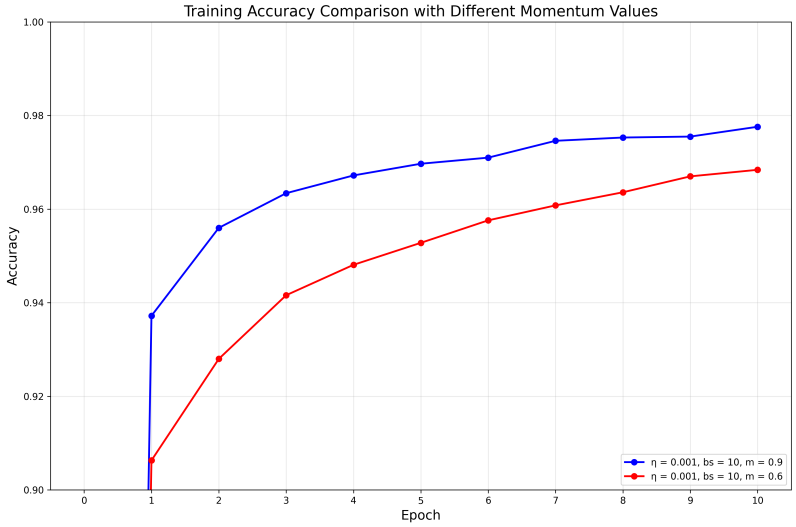

Figure 9: Combined Momentum Accuracy Plot

3.4 Combination

In this section, both decay learning rate and momentum techniques were combined together to optimise the SGD. This approach combines accumulation of gradients and fine-tunes the learning rate as training progresses, avoiding suboptimal loss regions and allowing finer adjustments in theory.

For this experiment, the most accurate decay learning rate combination ($\eta_0 = 0.1$, $\eta_N = 0.001$) from section 3.2 was used to experiment on batch sizes (1, 10, 100). Referring to Table 5, when the smaller batch sizes 1 and 10 were used, the results were unstable given the low test accuracy and NaN loss. However, as the batch size is increased to 100, the model achieved a good accuracy of 98.1% and a low loss of 0.032. These results were then plotted as shown in Figure 10.

m	Avg Loss	Test Acc	α
1	NaN	0.0980	0.9
10	NaN	0.0980	0.9
100	**0.0320**	**0.9812**	0.9
100	**0.0010**	**0.9819**	0.6

Table 5: Results combining Learning Rate Decay and Momentum

Examining Figure 10, the model displays unstable behaviour, given by the lack of smoothness when using the combinations of $\alpha = 0.9$, $\eta_0 = 0.1$ and $\eta_N = 0.001$. The zigzag pattern continues through the final epochs, though the model achieves 98.1% test accuracy at the end. These fluctuations reveal that the high momentum coefficient creates excessive updates, repeatedly overshooting the local minima although the loss curve decreases steadily.

The instability observed prompted further experimentation, so I reduced the momentum coefficient α to 0.6. As shown in Table 5, this modification improved accuracy by a small margin but decreased average loss by 68%. However when the results were plotted in Figure 11, the accuracy plot became significantly smoother without oscillations. While momentum accelerates convergence, its optimal value must be fine-tuned with the learning rate schedule to achieve stability and performance.

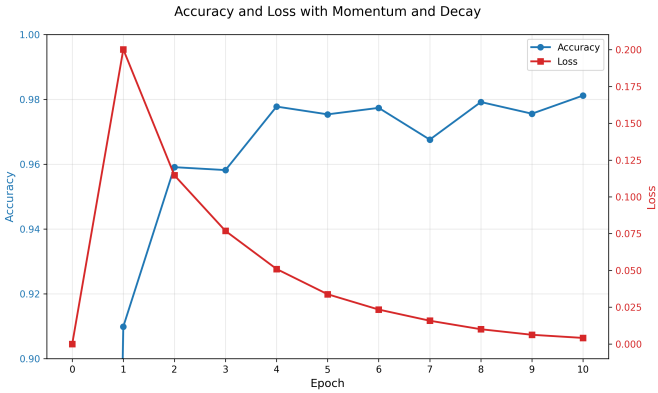

Figure 10: Plot for $\alpha = 0.9$, $\eta_0 = 0.1$, $\eta_N = 0.001$

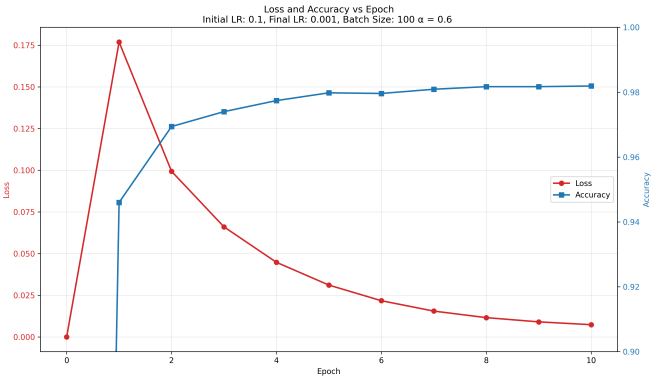

Figure 11: Plot for $\alpha = 0.6$, $\eta_0 = 0.1$, $\eta_N = 0.001$

4 Adaptive Learning Rate (AdaGrad)

In this section, the AdaGrad technique was used to optimise the SGD. AdaGrad adapts the learning rate of each parameter weight individually by accumulating historical squared gradients instead of using a global learning rate for all weights. This approach improves stability in training and has demonstrated effectiveness in handling sparse gradients and noisy updates [8]. The algorithm below is a summary of the mathematical implementation where η is the base learning rate and m is the batch size:

Algorithm 2 AdaGrad Optimisation Algorithm

Input: Training Data, η, m
Initialise network weights randomly , **Initialise** gradient accumulation $G_{ij} = 0$, **Validate** gradients
while *not completed epochs* **do**
  for *each batch of* m **do**
    for *each example in batch* **do**
      Forward pass: Compute network outputs and loss
      Backward pass: Calculate gradients $\frac{\partial L}{\partial w_{ij}}$ for all weights Accumulate gradients for batch update
    end
    for *all weights* w_{ij} *in network* **do**
      1. Compute the squared gradient: $g_{ij}^2 = (\frac{1}{m}\frac{\partial L}{\partial w_{ij}})^2$
      2. Update gradient accumulation: $G_{ij} = G_{ij} + g_{ij}^2$
      3. Calculate adaptive learning rate: $\eta_{ij} = \frac{\eta}{\sqrt{G_{ij};+\epsilon}}$ where $\epsilon = 10^{-8}$
      4. Apply update: $w_{ij} = w_{ij} - \eta_{ij} \cdot \frac{\partial L}{\partial w_{ij}}$
      Reset accumulated gradients
    end
  end
  Calculate loss and accuracy on test set Record and report metrics
end

Referring to Algorithm 2, weights and gradient accumulators are first initialised, then iteratively updated using mini-batches. In 1. and 2. for each parameter, it computes the squared gradient and adds it to an accumulator. Subsequently in 3., the learning rate is then scaled inversely by the square root of this sum. A small constant ϵ is added to the denominator to prevent zero division errors. The final step 4. applies the update to the weight parameter. This implementation allows frequently updated parameters to receive smaller adjustments while infrequent ones receive larger updates.

AdaGrad distinguishes itself from the previous SGD optimisers discussed in 3.2 and 3.3 by dynamically adjusting learning rates for each parameter based on accumulated past gradients. There is no need to manually update the learning rate as it varies with iterations adaptively, providing stable convergence [8] [9]. For this experiment, a grid search was conducted across the combinations of : $\eta \in \{0.001, 0.01, 0.1\}$ and $m \in \{1, 10, 100\}$ for 10 epochs similar to 3.1. The results are as follows:

η	m	Test Acc	Avg Loss
0.001	1	0.9529	0.1613
0.001	10	0.9731	0.0665
0.001	100	0.9798	0.0138
0.01	1	**0.9810**	0.0108
0.01	10	0.9794	**0.0033**
0.01	100	0.9708	0.0191
0.1	1	0.098	NaN

Table 6: Results for Momentum Coefficient ($\alpha = 0.9$ *and* 0.6)

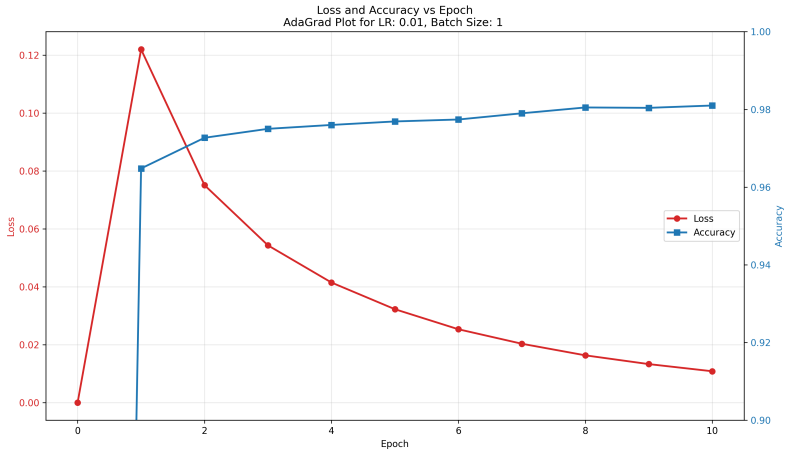

Figure 12: Loss and Accuracy plot for $\eta = 0.01$ and m = 10

4.1 AdaGrad Analysis

The failure observed with $\eta = 0.1$ and $m = 1$ can be explained by the **initialisation sensitivity** of adaptive gradient methods. AdaGrad is vulnerable to large initial learning rates because the accumulated squared gradient term G_{ij} starts near zero in early iterations. This causes the effective learning rate $\frac{\eta}{\sqrt{G_{ij}+\epsilon}}$ to be extremely high, leading to missing global minimas and experiences issues where the learning rate decreases and slowly converges [9]. This phenomenon often causes "NaN" loss values as shown by table 6, preventing the network from learning anything beyond random guessing [9].

Batch size effects were learning-rate dependent: For $\eta = 0.001$, larger batches improved performance (95.29% at $m = 1$ to 97.98% at $m = 100$). While for $\eta = 0.01$, smaller batches performed better. The highest accuracy (98.10%) was achieved with $\eta = 0.01$ and $m = 1$, which was then plotted on Figure 12.

Analysing Figure 12, it shows rapid initial improvement in both metrics followed by stable convergence. The loss continues to decrease even after accuracy plateaus around epoch 5. Compared to the SGD plot in Figure 2, AdaGrad's loss function oscillates less and its smoother, showing better image classification capabilities compared to plain SGD.

5 Conclusion

Method	Test Accuracy	Avg Loss	Computation t(s)
SGD	0.9787	0.0240	1031.98
SGD (Decay)	**0.9834**	0.0015	3167.474
SGD (Momentum)	0.9776	**0.0010**	7467.632
SGD (Momentum+Decay)	0.9819	**0.0010**	1020.505
Adagrad	0.9810	0.0108	5418.19

Table 7: Performance Comparison of Optimisation Methods

- **Best overall performer**: SGD (Decay) achieved the highest test accuracy (98.34%). This suggests that for this classification task, a well-tuned learning rate decay schedule can outperform other adaptive methods.
- **Most computationally efficient**: Among the high-performing methods, SGD with momentum and decay provided the best balance between accuracy (98.19%) and computational efficiency (1020.505 seconds), making it a feasible option for practicality.
- **Easiest to implement and tune**: AdaGrad required the least hyperparameter tuning while still achieving competitive accuracy (98.10%), making it a practical. However it required a relatively longer computation time compared to others. Hence, its usage depends on whether performance or accuracy is prioritised.

In conclusion, while SGD with learning rate decay achieved the highest accuracy on this classification task. The choice between these methods would ultimately depend on specific application requirements, balancing factors such as accuracy, training efficiency, implementation complexity, and generalisation capabilities. For future applications, other optimisation algorithms such as Adam and RMSProp should be explored as well, to carry out a more comprehensive hyperparameter search.

References

[1] Amari, S.I., 1993. Backpropagation and stochastic gradient descent method. Neurocomputing, 5(4-5), pp.185-196.

[2] N. Keskar et al., "On Large-Batch Training for Deep Learning: Generalization Gap and Sharp Minima," ICLR Workshop, 2017.

[3] J. Dean et al., "Large Scale Distributed Deep Networks," Adv. in Neural Information Processing Systems, 2012.

[4] L. Bottou, "Stochastic Gradient Descent Tricks," in Neural Networks: Tricks of the Trade, Springer, 2012.

[5] You, K., Long, M., Wang, J. and Jordan, M.I., 2019. How does learning rate decay help modern neural networks?. arXiv preprint arXiv:1908.01878.

[6] Smith, L.N., 2018. A disciplined approach to neural network hyper-parameters: Part 1–learning rate, batch size, momentum, and weight decay. arXiv preprint arXiv:1803.09820.

[7] Liu, Y., Gao, Y. and Yin, W., 2020. An improved analysis of stochastic gradient descent with momentum. Advances in Neural Information Processing Systems, 33, pp.18261-18271.

[8] Dogo, E.M., Afolabi, O.J., Nwulu, N.I., Twala, B. and Aigbavboa, C.O., 2018, December. A comparative analysis of gradient descent-based optimization algorithms on convolutional neural networks. In 2018 international conference on computational techniques, electronics and mechanical systems (CTEMS) (pp. 92-99). IEEE.

[9] Haji, S.H. and Abdulazeez, A.M., 2021. Comparison of optimization techniques based on gradient descent algorithm: A review. PalArch's Journal of Archaeology of Egypt/Egyptology, 18(4), pp.2715-2743.

[10] Bottou, L., 2010. Large-scale machine learning with stochastic gradient descent. In Proceedings of COMPSTAT'2010: 19th International Conference on Computational StatisticsParis France, August 22-27, 2010 Keynote, Invited and Contributed Papers (pp. 177-186). Physica-Verlag HD.

[11] Karpathy, A., 2017. CS231n Convolutional Neural Networks for Visual Recognition-Convolutional Neural Networks (CNNs/ConvNets).