CS 599: Deep Learning - Lab 1

Neha Nagaraja
School of Informatics, Computing, and Cyber Systems
Northern Arizona University
nn454@nau.edu

1 Problem 1: Linear Regression

1.1 Objective

The goal of this problem is to implement linear regression using **gradient descent**. Instead of solving for a closed-form solution, we optimize the model parameters iteratively using **tf.GradientTape**.

1.2 Implementation

Target Function:

$$y = 3x + 2 + \text{noise} \tag{1}$$

where x is sampled from a normal distribution, and random noise is added.

Model: The hypothesis function is:

$$\hat{y} = Wx + b \tag{2}$$

where W and b are trainable parameters.

Gradient Descent Update Rule:

$$W = W - \alpha \frac{\partial L}{\partial W}, \quad b = b - \alpha \frac{\partial L}{\partial b} \tag{3}$$

1.3 Experimental Setup

Parameters and Hyperparameters

Data Generation The data is generated using the function:

$$y = 3x + 2 + noise, \quad noise \sim \mathcal{N}(0, 1). \tag{4}$$

where:

- **Samples:** $N = 10,000$
- **Inputs:** $X \sim \mathcal{N}(0, 1)$

Model

- **Architecture:** Linear regression model

$$\hat{y} = Wx + b \tag{5}$$

- **Initialization:** W and b initialized with small random values from $\mathcal{N}(0, 1)$.

Preprint. Under review.

Training

- **Optimizer:** Gradient Descent
- **Learning Rate:** $\alpha = 0.001$
- **Training Steps:** 2000
- **Early Stopping:** Patience = 100 steps

1.4 Loss Functions

Different loss functions were tested:

Mean Squared Error (MSE):

$$L_{\text{MSE}} = \frac{1}{n} \sum (y - \hat{y})^2 \tag{6}$$

L_{MSE}: The Mean Squared Error loss function.
n: The number of data points (samples).
y: The actual (true) value.
$\hat{y}$: The predicted value.
$\sum (y - \hat{y})^2$: The squared difference between the actual and predicted values, which penalizes larger errors more.

Mean Absolute Error (MAE):

$$L_{\text{MAE}} = \frac{1}{n} \sum |y - \hat{y}| \tag{7}$$

L_{MAE}: The Mean Absolute Error loss function.
n: The number of data points (samples).
y: The actual (true) value.
$\hat{y}$: The predicted value.
$\sum |y - \hat{y}|$: The absolute difference between the actual and predicted values, treating all errors equally.

1.4.1 Analysis on MSE vs. MAE

Experimental Results Comparison

Metric	MSE	MAE
Final Loss	0.9860	2.3641
Final W	2.9519	1.4961
Final b	1.9352	-0.0749

Table 1: Comparison of MSE and MAE Loss Functions

Key Observations

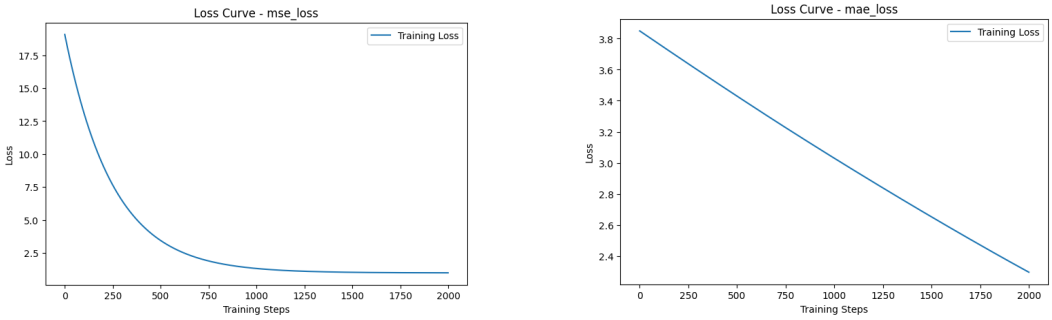
- MSE produced a lower final loss than MAE. Therefore MSE was a better model fit.
- MSE loss, parameters approached true values i.e *W* towards 3.0 and *b* towards 2.0.
- MAE struggled as parameters failed to approach true values.

Why was MSE better?

- The data included Gaussian noise. MSE is the maximum likelihood estimator under Gaussian noise, making it statistically optimal.
- MSE squares the errors, meaning larger errors contribute more to the loss making it accelerate gradient descent for big errors, leading to faster convergence.

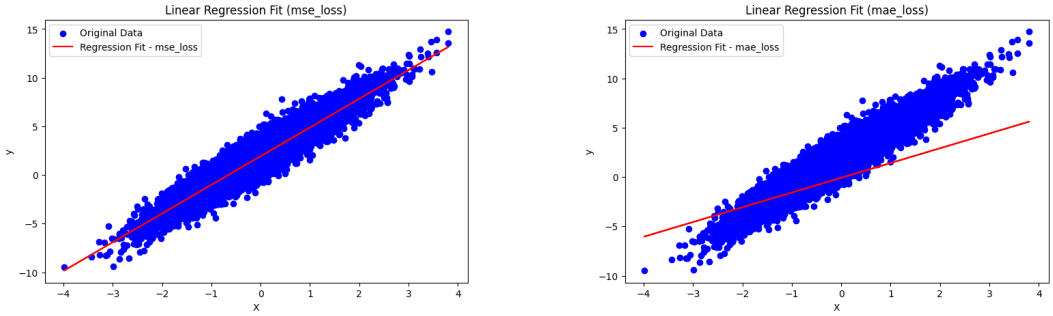
- MAE gives equal weight to all errors, so it does not push the model aggressively towards the best fit.
- Gradient of MAE is constant, while MSE has a smooth, varying gradient that helps optimization.

Cases when MAE would work better MAE is more robust to outliers, such as when the noise follows a Laplace distribution or contains extreme values. This is because its linear penalty function prevents large errors from disproportionately influencing the loss, unlike MSE, which squares errors and can overfit to outliers. If the dataset contains high noise or significant outliers, MAE would provide a more stable and reliable result by treating all errors equally.



(a) Loss curve for MSE (b) Loss curve for MAE

Figure 1: Comparison of loss curves for MSE and MAE during training.



(a) Linear Regression Fit using MSE (b) Linear Regression Fit using MAE

Figure 2: Comparison of Linear Regression fits optimized with MSE and MAE.

Based on the plots:
Loss Curves (Figure 1):

- The MSE loss (left) decreases steadily and flattens around a low value, indicating it converges closer to the true model.
- The MAE loss (right) decreases more slowly and remains relatively high compared to MSE.

Regression Fits (Figure 2):

- The MSE-based fit (left) closely aligns with the data, reflecting the underlying y=3x+2 relationship.
- The MAE-based fit (right) yields a flatter line, showing it did not converge to the true slope and intercept.

The next loss functions that I tried were the combination of the above functions; this can be done in two ways:

Huber Loss:

$$L_{\text{Huber}} = \begin{cases} \frac{1}{2}(y - \hat{y})^2, & \text{if } |y - \hat{y}| \leq \delta \\ \delta \cdot (|y - \hat{y}| - \frac{1}{2}\delta), & \text{otherwise} \end{cases} \tag{8}$$

L_{Huber}: The Huber loss function, which is a combination of MSE (for small errors) and MAE (for large errors).
y: The actual (true) value.
$\hat{y}$: The predicted value.
δ: The threshold that determines the transition between MSE and MAE behavior.
If $|y - \hat{y}| \leq \delta$, the loss behaves like MSE ($\frac{1}{2}(y - \hat{y})^2$).
If $|y - \hat{y}| > \delta$, the loss behaves like MAE ($\delta \cdot (|y - \hat{y}| - \frac{1}{2}\delta)$), reducing the effect of outliers.

Hybrid Loss (MSE + MAE):

$$L_{\text{Hybrid}} = \alpha L_{\text{MSE}} + (1 - \alpha) L_{\text{MAE}} \tag{9}$$

L_{Hybrid}: The hybrid loss function, which combines MSE and MAE.
L_{MSE}: The Mean Squared Error loss function.
L_{MAE}: The Mean Absolute Error loss function.
α: A weighting factor ($0 \leq \alpha \leq 1$) that determines the contribution of MSE vs. MAE.
If α is closer to 1, MSE dominates, making the loss more sensitive to large errors.
If α is closer to 0, MAE dominates, making the loss more robust to outliers.
This hybrid approach balances the benefits of both MSE (smooth optimization) and MAE (outlier resistance).

1.4.2 Analysis on Huber vs. Hybrid loss

Experimental Results Comparison

Metric	Huber Loss	Hybrid Loss ($\alpha = 0.5$, MSE + MAE)
Final Loss	1.9158	0.9911
Final W	1.5063	2.8085
Final b	-0.1021	1.7127

Table 2: Comparison of Huber Loss and Hybrid Loss ($\alpha = 0.5$).

Key Observations

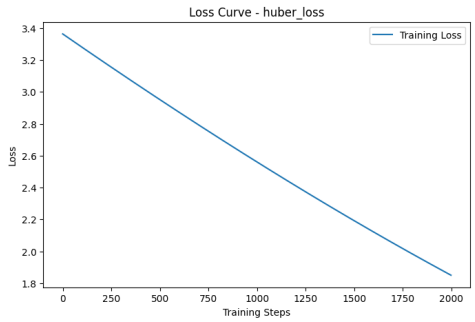
- Hybrid Loss achieved a lower final loss than Huber Loss. Hybrid Loss is a better fit.
- Hybrid Loss resulted in values much closer to the true parameters whereas Huber Loss struggled to reach the correct slope. Therefore was slower to adjust.

Why was Hybrid Loss better?

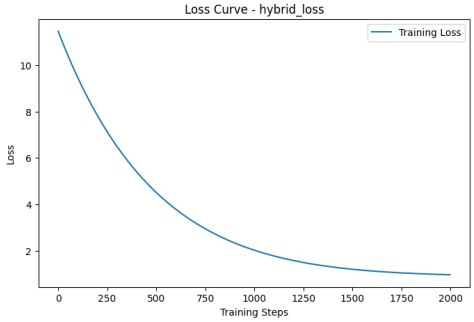
- Hybrid Loss takes advantage of both MSE and MAE at the same time, instead of switching dynamically like Huber.
- Since $\alpha = 0.5$, equal weight was given to MSE and MAE, helping to balance fast learning with robustness to noise.
- Huber Loss was more conservative and preventing rapid convergence.

When Should You Use Each?

- Use Hybrid Loss is a balance between precision and stability is needed. It learns faster than Huber while still being less sensitive to outliers.
- Use Huber Loss if your dataset has significant outliers because it automatically switches between MSE and MAE, ensuring stability.

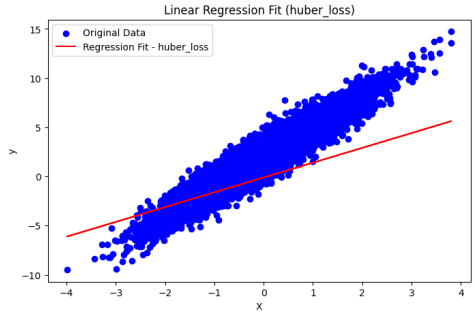


(a) Loss curve for Huber

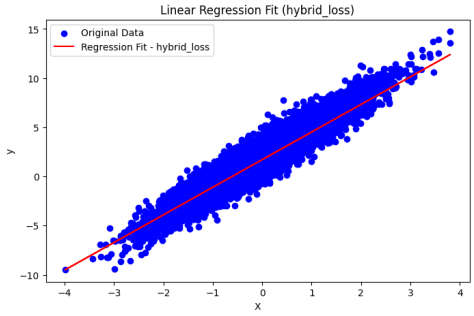


(b) Loss curve for Hybrid

Figure 3: Comparison of loss curves for Huber and Hybrid loss during training.



(a) Linear Regression Fit using Huber



(b) Linear Regression Fit using Hybrid

Figure 4: Comparison of Linear Regression fits optimized with MSE and MAE.

Overall, when comparing loss functions - MSE was the best it's final loss was the least the parameters were more close to the true ones.

1.5 Changing learning rate to 0.01

Impact of Learning Rate (0.01) on Loss Functions:

- All losses benefited from LR=0.01, highlighting the critical role of step size in gradient-based optimization.
- Increasing the learning rate (0.001 $\rightarrow$ 0.01) reduced MSE's convergence time by 3× (1,900 $\rightarrow$ 650 steps) and improved parameter accuracy (W=2.99,b=2.00).
- MAE, which failed to converge at LR=0.001, achieved usable parameters (W=2.98,b=1.99) with LR=0.01, though slower than MSE/Huber. Failed to achieve value close to previous parameter with previous LR.
- With LR=0.01, Huber achieved the lowest final loss (0.4168) and accurate parameters (W=2.99,b=2.01), leveraging adaptive L2/L1 dynamics. Failed to achieve value close to previous parameter with previous LR.
- Despite faster convergence than LR=0.001, the hybrid loss (L1+L2) plateaued at a higher loss (0.8828 vs. Huber's 0.4168) due to static weighting.
- Despite Huber's empirical success, MSE aligns with Gaussian noise assumptions and achieved near-ideal parameters (W=2.99,b=2.00) fastest.

1.6 Use patience scheduling [Whenever loss do not change, divide the learning rate by half].

Impact of using patience scheduling:

- MSE converged rapidly to near-true parameters (W=2.99,b=2.00) by step 300 but plateaued at Loss=0.9768, aligning with the Gaussian noise variance. Learning rate decay stabilized training but could not reduce loss further.
- Huber achieved the lowest loss (0.4168) and accurate parameters (W=2.99,b=2.01) by dynamically balancing L2 (for small errors) and L1 (for stability).
- Despite learning rate scheduling, MAE converged slower (1,473 steps) and achieved less accurate parameters (W=2.98,b=1.99) than Huber/MSE. Constant gradients and Gaussian noise mismatch hindered its efficiency, even with aggressive LR decay.
- The static $\alpha = 0.5$ weighting caused hybrid loss to plateau earlier (Loss=0.8828) than Huber. While closer to true parameters (W=2.98,b=2.00), its fixed design lacked the adaptability of Huber's dynamic L1/L2 switching.

1.7 Train for longer Duration. Change the initial value for W and B.

Changing the $trainsteps = 5000$ Initial value of$W$ and b from random to 0.5

Impact:

- Setting W=0.5 and b=0.5 does helped the model converge faster. Initialization Closer to True Values Accelerates Convergence
- Training beyond a certain point did not improve results due to early stopping and learning rate reduction.
- Regardless of training duration or initialization, the model always converged close to the true parameters
- For MAE and Huber loss, initial values had less impact since they are robust to outliers.

1.8 Changing noise levels

Impact of Changing Noise Level (stddev = 0.1)

- Faster Convergence – Lower noise (stddev=0.1) allows the model to reach optimal W 2.9991, b 2.0004 much faster than higher noise settings.
- Lower Final Loss – Reduced noise leads to significantly lower loss values (MSE: 0.0098, Huber: 0.0049) compared to default noise (0.97).
- More Precise Estimates – The learned parameters closely match the true values (W=3, b=2), reducing deviations seen in noisier data.
- Less Frequent Learning Rate Reduction – With a stable loss early on, patience scheduling reduces the learning rate later, improving training efficiency.

Impact of Changing Noise Level (stddev = 2.0)

- Slower Convergence – More noise delayed convergence across all loss functions, requiring more steps to stabilize, unlike lower noise settings where loss plateaued quickly
- Higher Final Loss Values – Compared to lower noise scenarios, the final loss increased significantly (e.g., MSE 3.9071, Huber 1.1444), indicating that noise introduces greater variance in predictions.
- Huber Loss and Hybrid Loss Handled Noise Better – These loss functions achieved relatively lower final loss values and more stable convergence, as they are less sensitive to outliers compared to MSE and MAE.
- Greater Variability in Early Training Steps – The initial loss values fluctuated more compared to lower noise settings, indicating that gradient updates were noisier, making the optimization process less stable.

1.9 Using various type of noise

We have used Gaussian noise till now.

Impact of using Uniform Noise:

- Slower Convergence Higher Final Loss: MSE loss stabilized at 0.3302, higher than Gaussian noise (0.0098, std = 0.1), indicating uniform noise is more disruptive.
- MAE and Huber Are More Robust: Huber loss (0.1651) outperformed MAE (0.4962) and MSE, handling uniform noise better.
- Impact on Gradient Descent: Learning rate decay stabilized training, but loss plateaued earlier than with Gaussian noise.
- MSE Struggled More: Compared to Huber and Hybrid loss, MSE was more affected by uniform noise, leading to higher error.

Impact of using Laplacian Noise:

- Patience scheduling triggered learning rate reductions earlier than with Gaussian noise, showing that Laplacian noise affects convergence speed and requires more cautious step adjustments.
- Initial loss fluctuations were more pronounced, showing the high variance nature of Laplacian noise affecting gradient updates.
- Training took more steps to stabilize compared to Gaussian noise, highlighting the challenge of handling Laplacian noise.
- Huber (0.1892) and Hybrid loss (0.4557) handled heavy-tailed noise better, confirming their robustness against outliers.
- MSE loss stabilized at 0.3808, higher than Gaussian (0.0098, std = 0.1) and uniform noise (0.3302), showing Laplacian noise impacts MSE the most.

1.10 Analysis of the Effects of Noise on Data, Weights, and Learning Rate on Performance

Adding Noise to Data, Weights, and Learning Rate:

- The addition of noise to data creates a more robust model by preventing overfitting but can slow down convergence.
- Noisy weight updates help the model escape local minima but can make learning unstable.
- Adding noise to the learning rate introduces controlled randomness, which can enhance exploration early on but may lead to fluctuations in later training stages.

Differences in Loss Behavior:

- MSE: Prone to instability, especially when noise is added to learning rate and weights. Loss oscillations indicate overshooting, but final values are close.
- MAE: More stable but converges slower. Adding noise to weights and LR does not impact as much due to MAE's robustness to outliers.
- Huber: Most stable and efficient, as it balances MSE and MAE. Handles noise well and gives a reasonable approximation of W and b.
- Hybrid: Sensitive to high noise but generally stable. It combines MSE's sensitivity and MAE's robustness, making it a balanced option.

1.11 Impact on Other Classification Problems and Mathematical Models:

Impact on Classification Problems

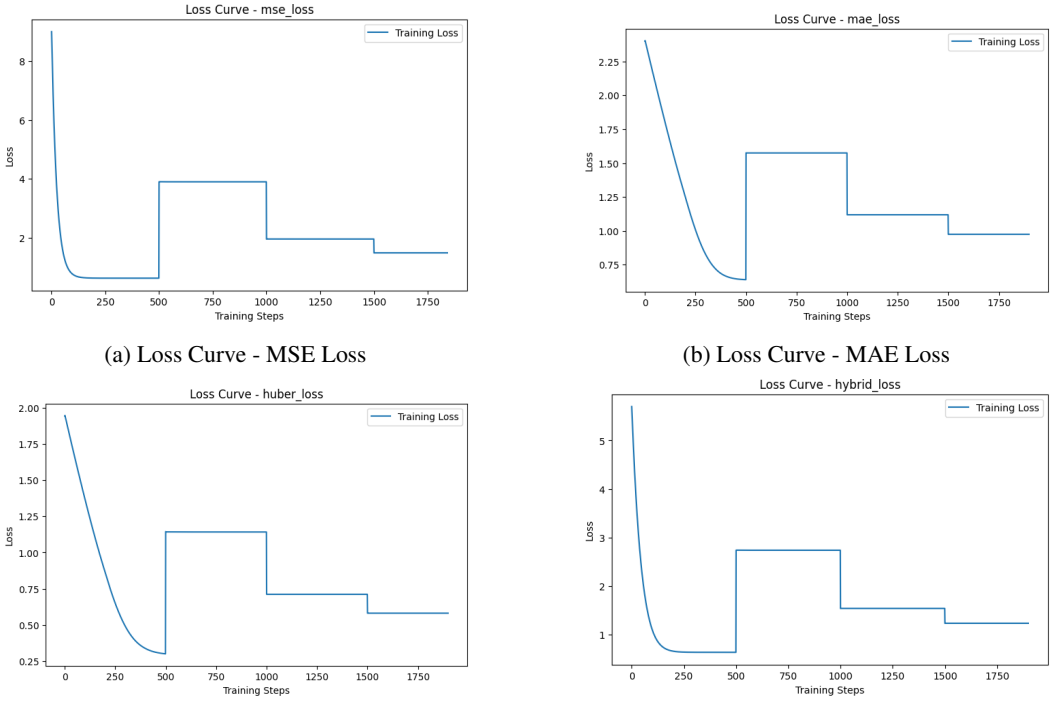
- Noise in Data: Improves generalization by preventing overfitting, especially with small datasets. Excessive noise can distort features, leading to misclassification. Example: In image classification, moderate Gaussian noise aids robustness, but excessive noise may obscure key patterns.

- Noise in Weights: Acts as a form of regularization, encouraging better generalization. Too much noise leads to unstable convergence. Example: Neural networks benefit from stochastic weight updates (e.g., Stochastic Weight Averaging (SWA)), but excessive weight noise can prevent the model from learning stable representations.
- Noise in Learning Rate: Helps escape sharp local minima, improving optimization. Excessive variation in learning rates can destabilize training.

Impact on Other Mathematical Models

- Regression Models: Small noise aids generalization, but large noise disrupts accuracy.
- Optimization-Based Models (e.g., Genetic Algorithms, Reinforcement Learning): Noise enhances exploration, helping to escape local minima. Uncontrolled noise may slow convergence.
- Probabilistic Models (e.g., Bayesian Inference): Bayesian models naturally incorporate uncertainty, making additional noise unnecessary. Excessive noise can distort posterior estimates.

1.12 Plot the different result - Noise on data, weights, learning rate on different losses



(a) Loss Curve - MSE Loss

(b) Loss Curve - MAE Loss

(c) Loss Curve - Huber Loss

(d) Loss Curve - Hybrid Loss

Figure 5: Comparison of loss curves for different loss functions with noise on data, weights and learning rate.

1.13 Do You Get the Exact Same Results If You Run the Notebook Multiple Times Without Changing Any Parameters? Why or Why Not?

Yes, I always get the exact same results, as I have fixed seed is set for random number generation. In my case, I have set the random seed **seed value = sum(ord(c) for c in "Neha")**. So each time I run, I get the same results. This ensures reproducibility by initializing the same sequence of random numbers across runs, making experiments repeatable.

Significance of Setting a Seed

- Machine learning experiments often rely on stochastic processes (e.g., weight initialization, batch selection, noise generation). Setting a seed ensures that these processes yield the same results across runs, making experiments repeatable.
- A fixed seed allows researchers to isolate the impact of model changes, preventing randomness from influencing performance comparisons.
- In optimization algorithms, random initialization affects gradient updates. Keeping a fixed seed ensures models converge similarly across different runs.

1.14 Can you get an model which is robust to noise? Does model lead to faster convergence? Do you get better local minima? Is noise beneficial?

Yes, the model is robust to noise – Techniques like Huber loss and regularization help handle noisy data. Adding noise to weights and data can improve generalization and prevent overfitting.

No, noise slows convergence – While controlled noise can prevent overfitting, excessive noise makes training unstable. It may require more iterations or adaptive learning rate strategies to maintain stability.

Yes, better local minima – Noise helps escape sharp minima and reach flatter, more generalizable ones. This prevents the model from overfitting to spurious patterns in the training data.

Yes, noise is beneficial – If controlled properly, it improves generalization; too much can be harmful. Adding moderate noise can help the model learn more robust decision boundaries and reduce variance.

2 Problem 2: Logistic Regression

2.1 Objective

The objective of this task is to implement a logistic regression model using TensorFlow with eager execution to classify images from the Fashion-MNIST dataset. The dataset consists of 60,000 training samples and 10,000 test samples, with each image being a 28×28 grayscale representation of an article of clothing, categorized into one of 10 classes. Given the complexity of Fashion-MNIST compared to standard MNIST, achieving high accuracy (above 90%) is challenging. This implementation will focus on optimizing loss functions, regularization, and optimizers to improve performance while adhering to the constraint of not using Keras.

2.2 Plot Images - Adam

The plot images function is responsible for visualizing 9 test images from the Fashion MNIST dataset in a 3x3 grid along with their true and predicted labels. Refer Figure 6.

2.3 Plot Weights - Adam

The plot weights function visualizes the learned weights of the logistic regression model for each of the 10 output classes. Refer Figure 7.

2.4 Optimizer Comparison: Adam vs SGD vs RMSProp

Optimizer	Final Test Accuracy (%)
Adam	83.81%
SGD	80.77%
RMSProp	79.68%

Table 3: Comparison of different optimizers on Fashion-MNIST using Logistic Regression.

Convergence Speed and Stablilty

- Fastest Convergence: Adam Loss drops quickly in early epochs. Stable training, but plateaus early. Final loss is lowest among all optimizers. Most stable.

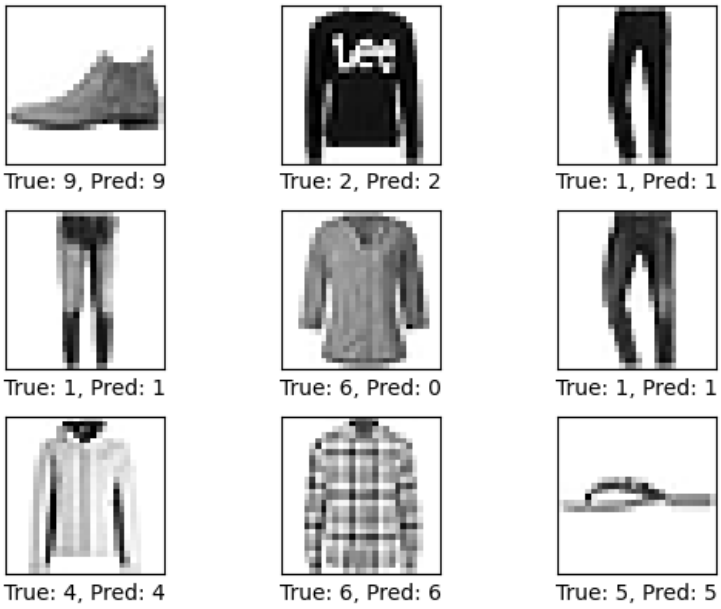


Figure 6: Sample test images with true and predicted labels (Adam).

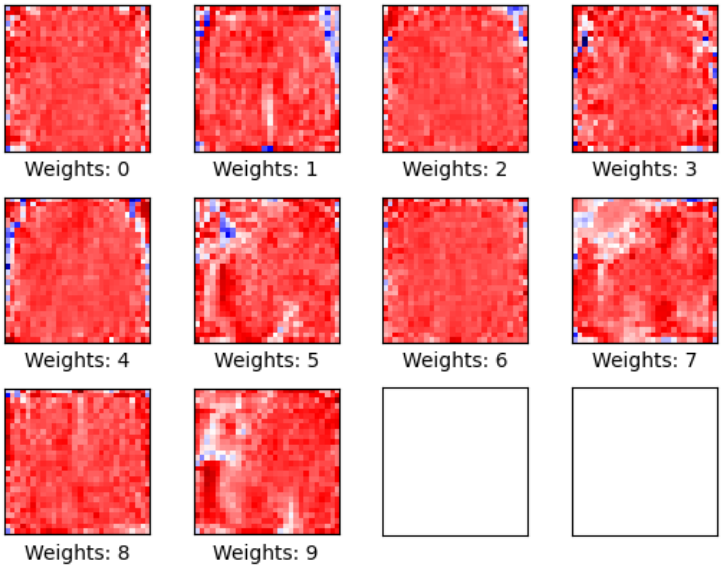


Figure 7: Visualization of learned weights for different classes (Adam).

- Slowest Convergence: RMSProp Steady but slow improvement per epoch. Takes more epochs to reach a good state.
- Medium Convergence: SGD Faster than RMSProp but less stable than Adam. Loss fluctuates

2.5 Train for longer epochs.

Key Insights from Training for 20 Epochs

- Best Final Accuracy: SGD (82.59%) outperformed Adam (80.46

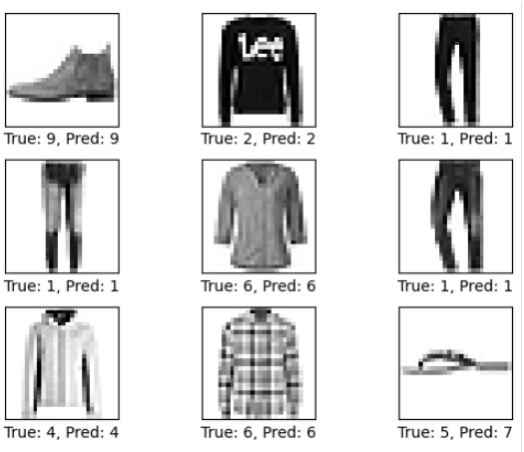


Figure 8: Sample test images with true and predicted labels (SGD).

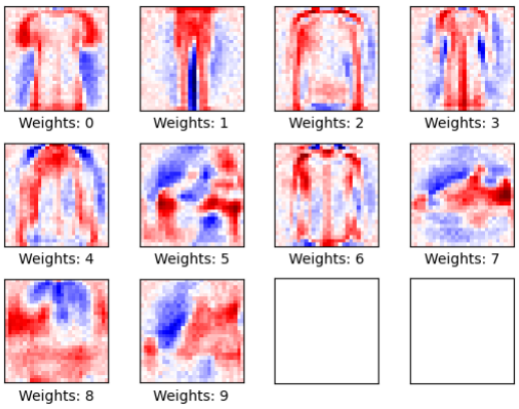


Figure 9: Visualization of learned weights for different classes (SGD).

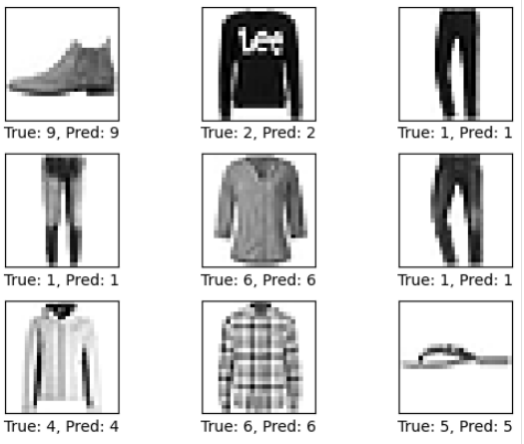


Figure 10: Sample test images with true and predicted labels (RMSProp).

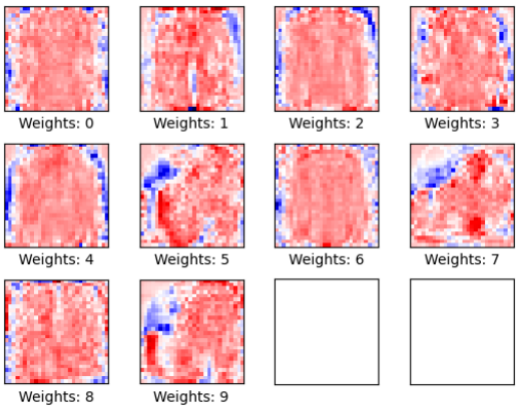


Figure 11: Visualization of learned weights for different classes (RMSProp).

Optimizer	Final Test Accuracy (%)
Adam	80.46%
SGD	82.59%
RMSProp	79.98%

Table 4: Test accuracy of different optimizers on Fashion-MNIST using Logistic Regression (20 epochs).

- Adam Converges Fast but Plateaus: Reached peak accuracy early (10 epochs) but started fluctuating.
- SGD Improves Steadily: Slow but consistent accuracy increase, best suited for longer training.
- RMSProp is Unstable: Fluctuating loss and accuracy, making it unreliable for extended training.
- Best for Short Training: Adam (10-15 epochs) if fast convergence is needed.
- Best for Long Training: SGD (20+ epochs) for stable and better generalization.

Changing n_epochs back to 10

2.6 Observing Performance Changes with Different Train/Validation Splits

We compare 80/20 vs. 90/10 train/validation splits to observe performance drop/gain across different optimizers.

Optimizer	80/20 Split (%)	90/10 Split (%)	Change (Drop/Gain)
Adam	83.81%	82.57%	$\downarrow$ 1.24% Drop
SGD	80.77%	81.16%	$\uparrow$ 0.39% Gain
RMSProp	79.68%	82.12%	$\uparrow$ 2.44% Gain

Table 5: Impact of different train-validation splits on optimizer performance.

Key Observations

- Adam Performs Better with More Training Data (80/20 Split): 83.81% $\rightarrow$ 82.57% (-1.24%): Slight drop when using fewer training samples (90/10 split). Adam converges quickly, and more training data helps generalization.
- SGD Shows Slight Improvement in 90/10 Split: 80.77% $\rightarrow$ 81.16% (+0.39%): Small improvement. More validation data helped prevent overfitting.

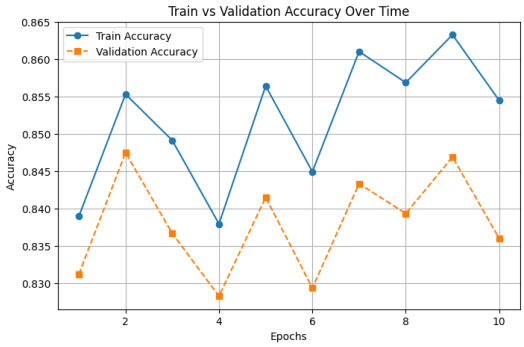


Figure 12: Train/Val accuracy over time - Adam (80/20 split)

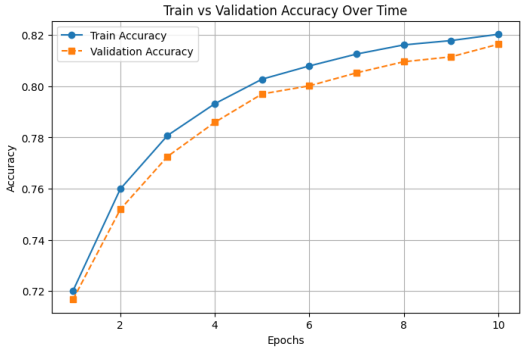


Figure 13: Train/Val accuracy over time - SGD (80/20 split)

- RMSProp Gains the Most in 90/10 Split: 79.68% $\rightarrow$ 82.12% (+2.44 %): Biggest improvement!. RMSProp was unstable in 80/20 but improved with more validation samples.

Changing to 80/20 split

2.7 Train/Val accuracy over time (80/20 split)

Refer Figure 12, 13, 14.

2.8 Does Batch Size Affect Model Performance?

Yes! Batch size significantly impacts performance in terms of accuracy, convergence speed, memory usage, and generalization. **When batch size = 32**

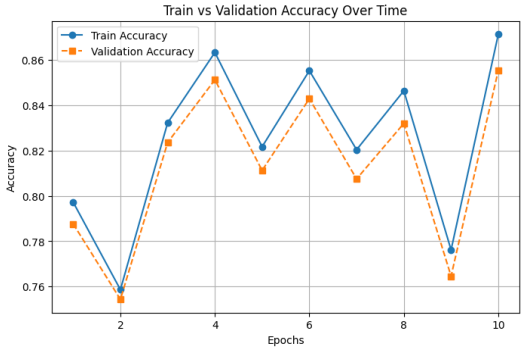


Figure 14: Train/Val accuracy over time - RMSProp (80/20 split)

Batch Size	SGD Accuracy (%)	Adam Accuracy (%)	RMSProp Accuracy (%)
32	83.07% (Higher than batch=128)	79.17% (Lower than batch=128)	80.11% (Higher than batch=128)
128	80.77%	83.81% (Higher than batch=32)	79.68%

Table 6: Impact of batch size on optimizer accuracy for Fashion-MNIST classification.

Small Batch (32) → Better for SGD RMSProp

SGD (83.07%) RMSProp (80.11Smaller batches lead to more frequent updates, improving generalization. Less stable updates → may improve in scenarios with noisy data. Larger Batch (128) → Better for Adam

Adam performed best with batch=128 (83.81Adam benefits from larger batches since it uses adaptive learning rates. More stable updates with fewer weight oscillations. Trade-off Between Stability Generalization

Batch=32 generalizes better (higher test accuracy in SGD/RMSProp). Batch=128 stabilizes training (smoother loss curve, better for Adam).

2.9 Does the Model Overfit?

Yes, overfitting is possible. If Train Accuracy is greater than Validation Accuracy, the model memorizes training data instead of generalizing.

Example from Adam, 90/10 split result: Train Acc (Epoch 9): 86.89% Validation Acc (Epoch 9): 85.13% Train Acc (Epoch 10): 85.07% Validation Acc (Epoch 10): 83.08% (Performance drop!) Sign of Overfitting: Validation accuracy stagnates or drops while train accuracy increases.

Overfitting: Train loss continues to decrease, but validation loss stops decreasing or increases. Good Generalization: Train and validation loss decrease together.

Measures Taken to Avoid Overfitting

- Train/Validation Split Adjustment: Used 80/20 & 90/10 splits to evaluate performance on unseen data. 80/20 gave better generalization for Adam/
- Batch Size Optimization: Used optimum size 128.
- We can add early-stopping, weight regularization.
- Change learning parameters