Lab 1: Foundations of Deep Learning

Thi Hien Nguyen
School of Informatics, Computing, and Cyber Systems
Northern Arizona University
AZ, USA
tn598@nau.edu

The code can be accessed at `https://github.com/hienngt/IST597_Fall2019_TF2.0`.

1 Problem 1

The mean squared error (MSE) is defined as $\frac{1}{n}\sum_{i=1}^{n}(y_{\text{hat},i} - y_i)^2$. Mean absolute error (MAE) is defined as $\frac{1}{n}\sum_{i=1}^{n}|y_{\text{hat},i} - y_i|$. Hybrid loss is $\frac{1}{n}\sum_{i=1}^{n}\left(\alpha|y_{\text{hat},i} - y_i| + (1-\alpha)(y_{\text{hat},i} - y_i)^2\right)$ where $\alpha = 0.5$.

1.1 Loss Function

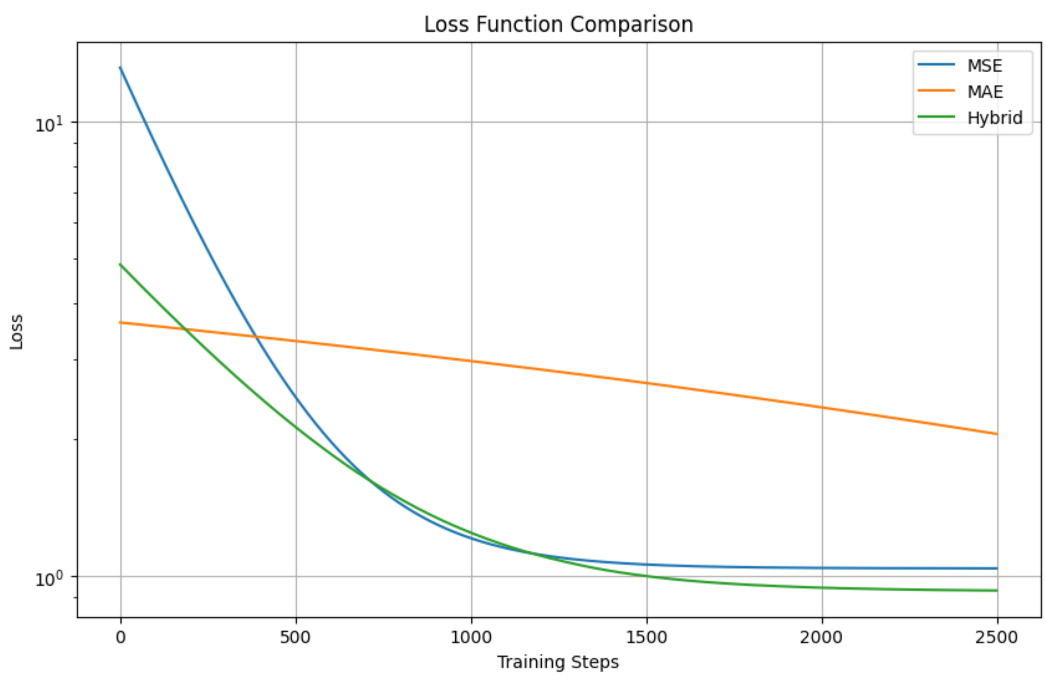


Figure 1: Performance of loss functions.

Based on Fig. 1, the hybrid loss function appears to outperform the other two loss functions, MSE and MAE, for several reasons. First, it demonstrates faster convergence, as the hybrid loss decreases more rapidly in the early stages of training, indicating that it achieves lower loss values more quickly, which is advantageous for efficient training. Additionally, the hybrid loss benefits from stability, as it combines the strengths of both MSE and MAE, balancing sensitivity to outliers from MSE with robustness from MAE. This balance likely contributes to its smoother and more stable convergence.

Finally, the hybrid loss reaches a lower final value compared to MAE, which decreases more slowly over time. Although MSE also converges to a low value, its initial sharp decline suggests a greater sensitivity to outliers early in training.

1.2 Init Variable

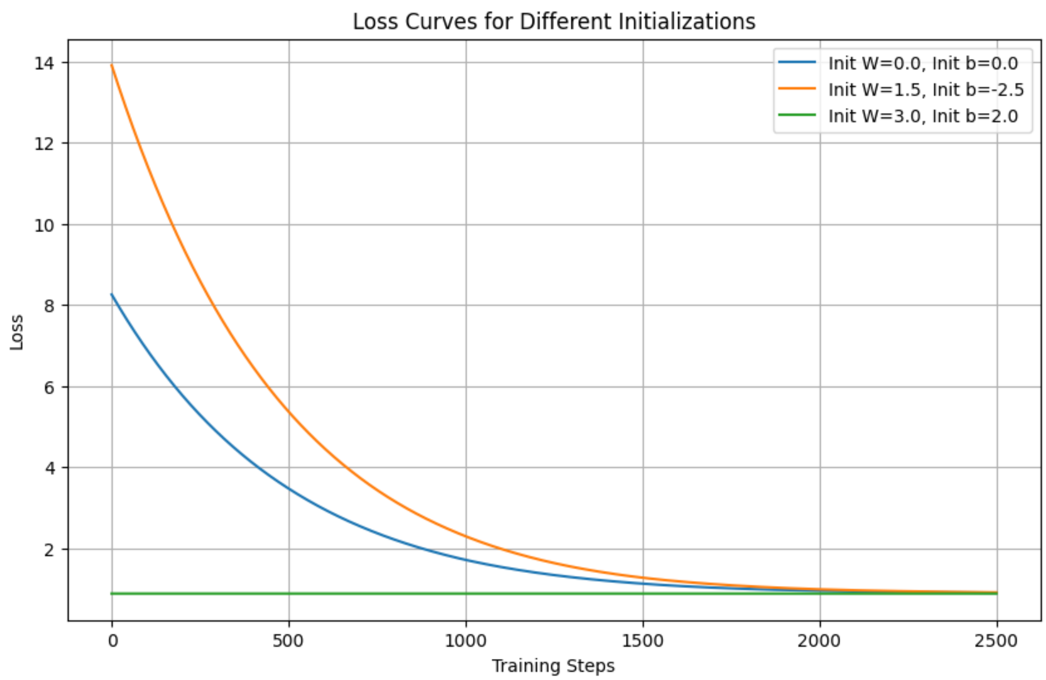


Figure 2: Loss curves for different initializations.

Fig. 2 shows the loss curves for different initialization. For the initialization $W = 3.0, b = 2.0$, the loss starts at a very low value and quickly stabilizes. Convergence is achieved almost immediately, with minimal training steps required. This demonstrates that initializing parameters close to their true values leads to faster convergence and minimal computational effort. In the case of initialization $W = 1.5, b = -2.5$, this initialization is farther from the true values. The loss starts at a much higher value compared to the green curve. Although convergence takes longer, it eventually reaches a similar final loss as the green curve. Starting farther from the true values requires more training steps for the model to converge, but it still reaches the same optimal solution eventually. For the initialization $W = 0.0, b = 0.0$, the loss starts at a high value and decreases steadily over time. Convergence takes the longest compared to the other two initializations. Poor initialization increases the number of training steps required but does not affect the final result as long as sufficient training steps are provided.

The key effects of changing initial values include the convergence speed, final result, and training efficiency. Initializing W and b closer to their true values significantly speeds up convergence, while poor initializations lead to slower convergence due to larger gradients and more updates needed to reach optimal values. Regardless of the initial values, all models eventually converge to similar final loss values $(W \approx 3, b \approx 2)$ if trained for sufficient steps, as gradient descent effectively optimizes convex loss functions like MSE or hybrid loss. Additionally, better initialization reduces computational effort by requiring fewer iterations for convergence; however, poor initialization increases training time without preventing convergence.

1.3 Noise

Noise can significantly affect model performance in various ways. When noise is added to the input data (X) and target labels (y), it increases variability in the training data. This variability can enhance generalization by encouraging the model to learn underlying patterns rather than memorizing specific

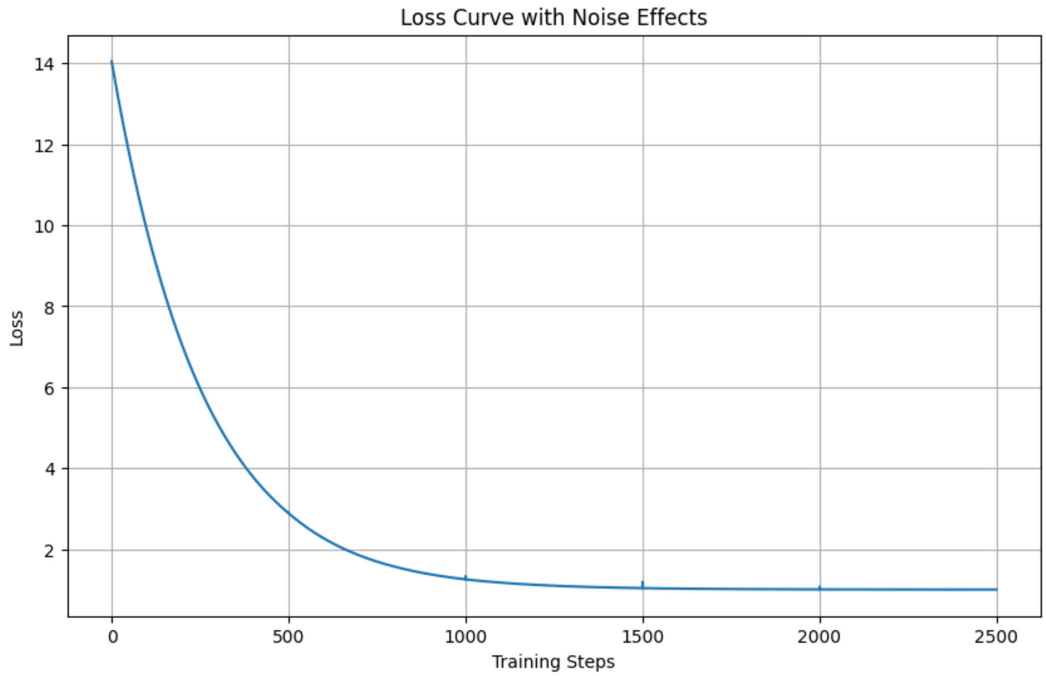


Figure 3: Loss curves with noise effects.

data points. However, it may also slightly slow down convergence, as the model must adjust for noisy inputs. Despite this added noise, the model still converged effectively, achieving a final loss of 1.0098.

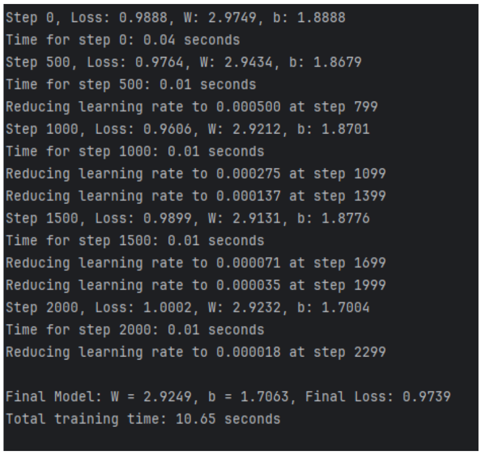
Similarly, introducing noise to the weights during training adds a degree of randomness to parameter updates. This randomness can help the model escape local minima by slightly perturbing the weights. However, excessive noise could destabilize the training process or hinder convergence. The weights (W) and bias (b) converged close to their true values, indicating that the noise applied to the weights was beneficial without being excessive.

Lastly, when noise is introduced to the learning rate, it creates variability in the step sizes during gradient descent. This variability allows the model to explore different regions of the loss surface more effectively. While some noise can be advantageous, excessive noise might lead to instability or oscillations during training. In this case, the adjustments to the learning rate did not hinder convergence, suggesting that the noise levels were well managed. Fig. 5 shows the effect of noise on the model.

I think when applied judiciously, these changes can similarly affect other classification problems and mathematical models. Controlled noise can improve generalization by acting as a regularizer, which helps prevent overfitting and enhances robustness. However, it may also slow down convergence due to increased variability during training; this trade-off often leads to better overall solutions.

Running the notebook multiple times without setting a random seed can yield different results. This variability arises from the noise and random initializations in the model, which involve stochastic processes that lead to different outcomes with each execution. Setting a random seed, ensures reproducibility, as it initializes the random number generator to a fixed state. Consequently, when running the notebook with the same seed, the random processes will consistently produce identical results.

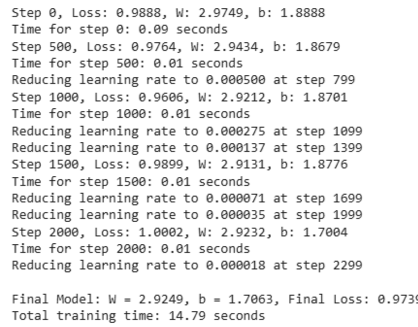
The model achieves a final loss of 0.9739 on both CPU and GPU, demonstrating its robustness to noise, even with disturbances introduced in the data, weights, and learning rate. The periodic addition of Gaussian noise did not hinder convergence, indicating that the model can generalize effectively under noisy conditions. While noise does not accelerate convergence, it slightly delays the process of reaching lower loss values due to its stochastic nature. However, this trade-off is acceptable as it



```
Step 0, Loss: 0.9888, W: 2.9749, b: 1.8888
Time for step 0: 0.04 seconds
Step 500, Loss: 0.9764, W: 2.9434, b: 1.8679
Time for step 500: 0.01 seconds
Reducing learning rate to 0.000500 at step 799
Step 1000, Loss: 0.9606, W: 2.9212, b: 1.8701
Time for step 1000: 0.01 seconds
Reducing learning rate to 0.000275 at step 1099
Reducing learning rate to 0.000137 at step 1399
Step 1500, Loss: 0.9899, W: 2.9131, b: 1.8776
Time for step 1500: 0.01 seconds
Reducing learning rate to 0.000071 at step 1699
Reducing learning rate to 0.000035 at step 1999
Step 2000, Loss: 1.0002, W: 2.9232, b: 1.7004
Time for step 2000: 0.01 seconds
Reducing learning rate to 0.000018 at step 2299

Final Model: W = 2.9249, b = 1.7063, Final Loss: 0.9739
Total training time: 10.65 seconds
```

(a) CPU.



```
Step 0, Loss: 0.9888, W: 2.9749, b: 1.8888
Time for step 0: 0.09 seconds
Step 500, Loss: 0.9764, W: 2.9434, b: 1.8679
Time for step 500: 0.01 seconds
Reducing learning rate to 0.000500 at step 799
Step 1000, Loss: 0.9606, W: 2.9212, b: 1.8701
Time for step 1000: 0.01 seconds
Reducing learning rate to 0.000275 at step 1099
Reducing learning rate to 0.000137 at step 1399
Step 1500, Loss: 0.9899, W: 2.9131, b: 1.8776
Time for step 1500: 0.01 seconds
Reducing learning rate to 0.000071 at step 1699
Reducing learning rate to 0.000035 at step 1999
Step 2000, Loss: 1.0002, W: 2.9232, b: 1.7004
Time for step 2000: 0.01 seconds
Reducing learning rate to 0.000018 at step 2299

Final Model: W = 2.9249, b = 1.7063, Final Loss: 0.9739
Total training time: 14.79 seconds
```

(b) GPU.

Figure 4: Result on CPU and GPU.

enhances the model's robustness and generalization capabilities. Noise can help the optimizer escape sharp or suboptimal regions of the loss surface, leading to better local minima. Both CPU and GPU yield identical results for model parameters and final loss, confirming that TensorFlow's operations remain deterministic when using a fixed random seed. GPUs typically provide faster training times than CPUs for larger datasets or more complex models due to their parallel processing capabilities. For this small-scale problem, however, the difference in training time between CPU and GPU is likely negligible. Fig. 4 shows the details when running this model on CPU and GPU.

2 Problem 2

Optimizer

Fig. 5 shows the effect of the optimizer on loss. SGD exhibits slow convergence, with training accuracy increasing from 13.58% to 30.69% over 10 epochs. The validation accuracy fluctuates, peaking at 27.80% in epoch 9. In contrast, the Adam optimizer achieves fast convergence, with training accuracy jumping to 45.02% in the first epoch and stabilizing around 59% by the second epoch. Validation accuracy rapidly exceeds 60% and remains stable. RMSprop demonstrates moderate convergence speed; training accuracy reaches 36.30% in the first epoch and stabilizes around 61% by epoch 3, with validation accuracy quickly surpassing 57% and continuing to improve. Among these optimizers, Adam converges the fastest, followed by RMSprop, while SGD shows the slowest convergence. In terms of final performance, SGD achieves a training accuracy of 30.69% and a validation accuracy of 22.05%, indicating poor performance and potential underfitting. Adam reaches a final training accuracy of 59.14% and a validation accuracy of 60.73%, demonstrating good performance with consistent generalization. RMSprop outperforms both, achieving a final training accuracy of 61.09% and a validation accuracy of 65.13%, reflecting the best performance and effective generalization. RMSprop not only reaches the best local minima but also generalizes better, as indicated by its highest validation accuracy. While Adam performs well with consistent generalization, SGD struggles to find an effective solution. RMSprop converges relatively quickly while achieving the best local minima and superior generalization. Adam, despite converging the fastest, settles for slightly lower performance. SGD shows poor results in both convergence speed and final performance for this particular problem.

Train/Val split

With a 0.15 split, the final training accuracy is slightly lower at 58.54% compared to 61.09% with a 0.1 split. This indicates that the model learns the training set marginally better with a smaller training data split. The 0.1 split consistently achieves higher validation accuracies throughout the training process. The final validation accuracy for the 0.1 split is 65.13%, notably higher than the 61.66% achieved with the 0.15 split. Additionally, the highest validation accuracy for the 0.1 split is better at 65.53% compared to 63.23% for the 0.15 split. The 0.1 split exhibits more fluctuation in validation accuracy

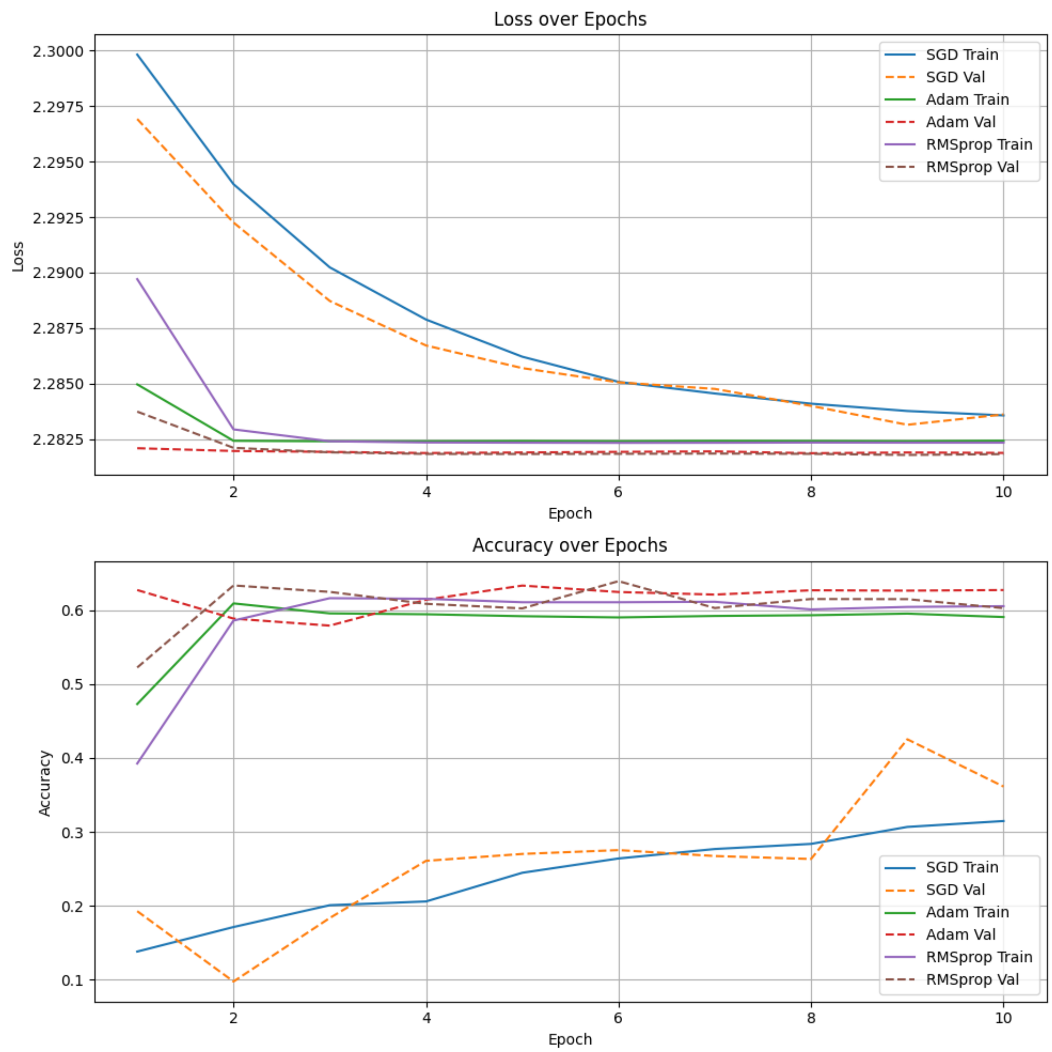
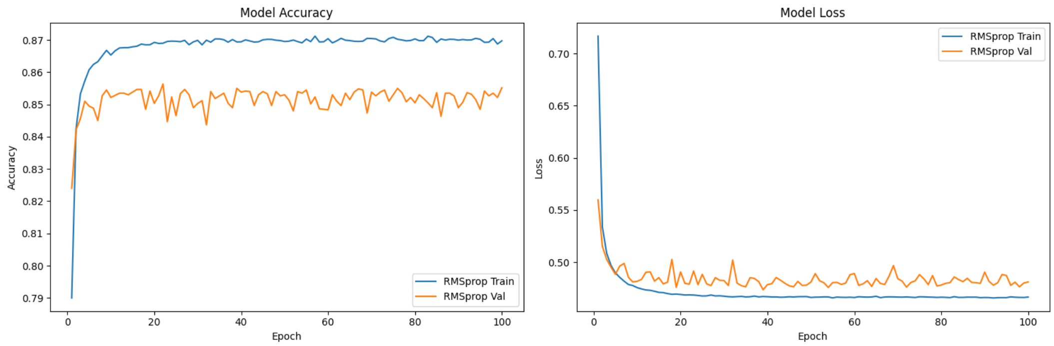


Figure 5: Loss over Epochs.

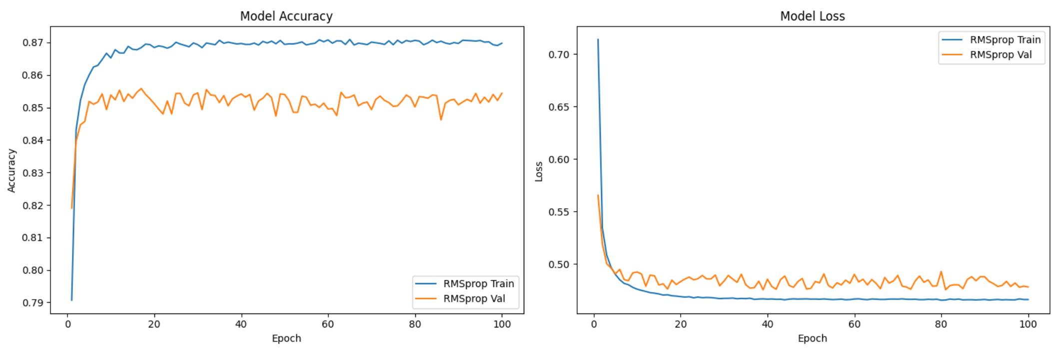
across epochs, while the 0.15 split demonstrates more stable validation performances, likely due to the larger validation set. There is a noticeable performance gain when utilizing the 0.1 validation split as opposed to the 0.15 split, evident in both training and validation accuracies. Possible reasons for this performance improvement include a larger amount of training data, as a smaller validation set (0.1) allows the model to learn from more examples. Furthermore, a 0.1 split provides a sufficient number of validation samples to reliably assess the model's performance. However, it is important to note that the smaller validation set (0.1 split) may yield a less robust estimate of the model's generalization capability compared to the larger validation set (0.15 split).

The model was trained using the RMSprop optimizer for 100 epochs. Below are the results:

- Training Accuracy: The model achieved a maximum training accuracy of approximately 87%, showing good performance on the training data.
- Validation Accuracy: The model achieved a maximum validation accuracy of approximately 85%, indicating slight overfitting but still strong generalization.
- Training and Validation Loss: Both losses decreased significantly in the first few epochs, stabilizing after around epoch 20.



(a) CPU.



(b) GPU.

Figure 6: Train/validation accuracy and loss over time.

The model performed well on both training and validation datasets, achieving high accuracy. In Fig. 6, the left graph shows the accuracy over epochs for both training and validation datasets and the right graph shows the loss over epochs for both training and validation datasets.

The grid of weight visualizations shown in Fig. 8 provides insight into how the model's learned features change throughout training. Each image represents the activations corresponding to different learned weights as shown in Fig. 7. As training progresses, the patterns in these visualizations become more defined, suggesting that the model is honing in on relevant features in the dataset. Overall, the model exhibits promising performance, with a strong training accuracy and a healthy validation accuracy trend. The visualizations of weights and predictions reinforce the model's capability in feature learning and classification. Future work could focus on enhancing generalization through additional regularization techniques or exploring different model architectures to further improve performance.

Batch size significantly impacts model performance, training dynamics, and generalization. Smaller batch sizes lead to longer training times due to more frequent parameter updates and often result in better generalization because of noisier gradient estimates that help the model escape local minima. Conversely, larger batch sizes can speed up training but may require more memory and can lead to overfitting due to sharper minima.

The choice of batch size also affects the learning rate; larger batches often necessitate a higher learning rate for optimal performance. While smaller batches introduce variability that aids in generalization, they may create instability during training. Ultimately, finding the right batch size is essential, and experimentation is key to determining the best setting for a specific problem and dataset.

In Fig. 6, the accuracy plot reveals that training accuracy, represented by the blue line, consistently improves and stabilizes around 87%. In contrast, validation accuracy, shown by the orange line, stabilizes at a slightly lower level of approximately 85%, exhibiting some fluctuations during training. The small but noticeable gap of about 2% between training and validation accuracy indicates mild overfitting. In the loss plot, both training and validation loss decrease sharply in the initial epochs,

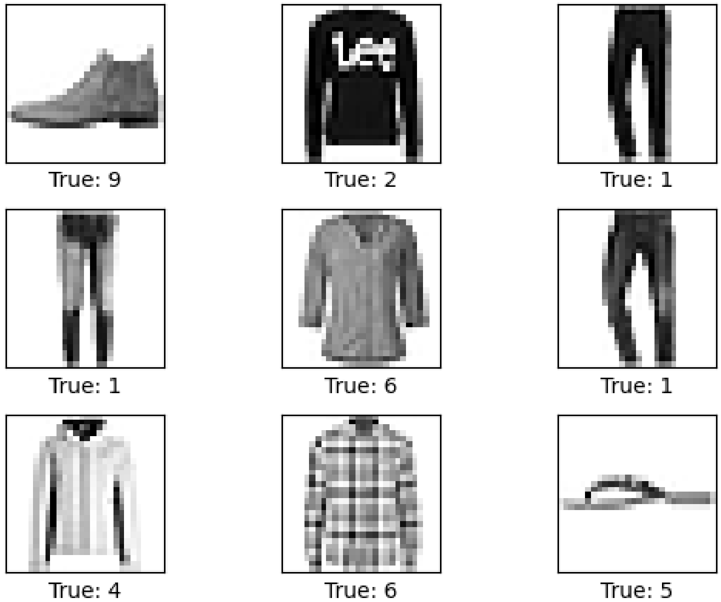


Figure 7: Sample predictions on GPU.

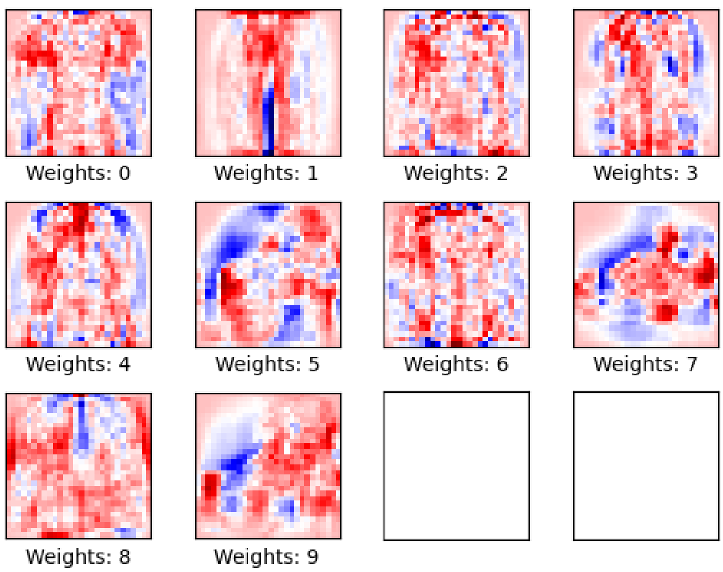


Figure 8: Model weights visualization on GPU.

stabilizing after around epoch 20. However, validation loss fluctuates more than training loss, further suggesting mild overfitting. Regarding weight visualization in Fig. 8, the weights for each class display distinct patterns, demonstrating that the model has effectively learned meaningful features for classification. Nevertheless, the presence of some noisy regions in the weight heatmaps hints at potential overfitting to specific details in the training data. So I think the model overfits. Overfitting occurs when the model learns patterns in the training data that do not generalize well to unseen data. In this case, potential reasons for overfitting include:

- Logistic regression is a simple model and may struggle to generalize complex patterns in the Fashion MNIST dataset.

- Although L2 regularization was used, it might not be strong enough to prevent overfitting entirely.
- Fashion MNIST is a relatively challenging dataset with subtle differences between some classes, which can lead to overfitting if the model tries to memorize specific details.

To avoid overfitting, we can apply early stopping, early stopping could have been used to halt training once validation performance stopped improving, further reducing overfitting, also, we can normalize data before the training model.

Compare performance with random forest and svm

In my analysis of the Fashion MNIST dataset, I compared three machine learning models: logistic regression implemented in tensorflow, random forest, and support vector machine (SVM) using scikit-learn. The accuracy results reveal a clear hierarchy in performance, with random forest achieving the highest accuracy at 87.64%, followed by SVM at 84.64%, and logistic regression at 83.88%. Random forest's superior performance stems from its ability to handle non-linear relationships and capture complex interactions between features. As an ensemble method, it is robust to overfitting and excels at identifying intricate patterns in image data, making it well-suited for the fashion MNIST dataset. While SVM outperformed Logistic Regression, it fell short of random forest's accuracy. Its strengths lie in handling high-dimensional spaces and using various kernel functions, but a linear kernel may have limited its ability to capture complex non-linear patterns. Nevertheless, SVM showed good class separation in feature space. Logistic regression, the simplest model, delivered competitive results despite its linear nature. Its interpretability is an advantage, and its performance indicates that even linear models can capture significant aspects of the Fashion MNIST dataset.

Cluster the weights for each class

Figure 9 visualizes the clustering of weights using k-means with five clusters. The t-SNE plot represents the logistic regression model's weight vectors for each of the 10 Fashion MNIST classes in a reduced two-dimensional space. Each point corresponds to a weight vector for a specific class, and the colors indicate the clusters identified by k-means. The plot reveals distinct clusters, suggesting that classes with similar learned weights are grouped together. This clustering demonstrates how the model differentiates between classes and highlights relationships among them. The separation of clusters suggests that the model has learned meaningful representations for each class, while overlapping clusters may indicate challenges in distinguishing certain classes.

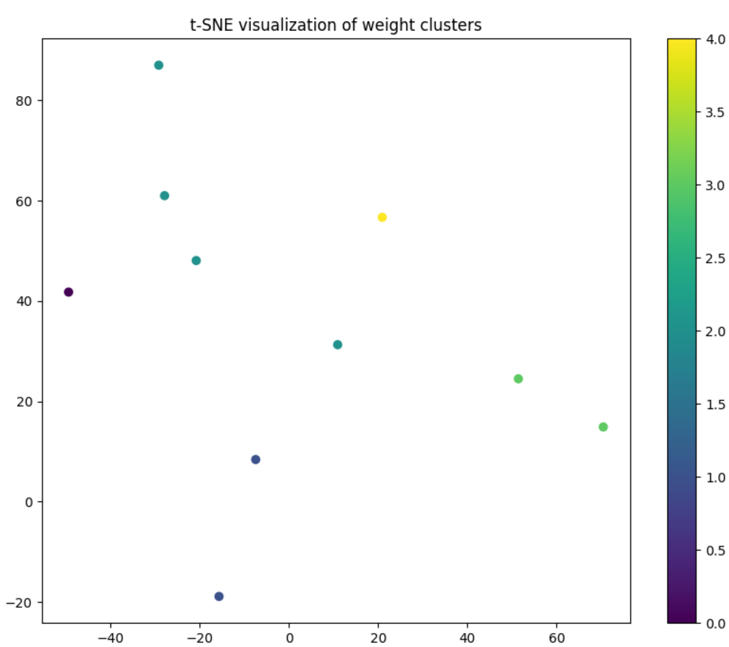


Figure 9: t-SNE visualization of weight clusters.