AMATH 482 Homework 4

Trevor Ruggeri

Due: March 17, 2025

This report documents the design, implementation, and optimization of a Fully Connected Neural Network (FCN) for classifying the FashionMNIST dataset. The FCN was built with adjustable architecture parameters, including the number of layers, neurons, learning rate, and training epochs. The project explored baseline configurations and hyperparameter tuning, testing various optimizers (SGD, Adam, RMSProp), regularization techniques (Dropout), weight initialization methods (Random Normal, Xavier, Kaiming), and Batch Normalization. Results demonstrate that hyperparameter choices significantly influence training loss, validation accuracy, and test accuracy. The final configurations achieved over 89% accuracy on the test set with effective generalization and stable training behavior.

1 Introduction and Overview

The FashionMNIST dataset consists of 70,000 grayscale images (28x28 pixels) classified into 10 categories of clothing. Developing machine learning models for this dataset poses challenges in generalization and computational efficiency. This project implements a Flexible Connected Neural Network (FCN) for classifying FashionMNIST images, enabling experimentation with architecture, optimization, and regularization techniques. The primary goals are:

1. To design a customizable FCN with adjustable architectural and hyperparameters.
2. To establish a baseline configuration yielding at least 85% test accuracy.
3. To explore hyperparameter tuning for improved training and generalization through experiments with optimizers, Dropout regularization, weight initialization, and Batch Normalization.

2 Theoretical Background

2.1 Fully Connected Neural Network (FCN)

An FCN consists of layers of neurons where each neuron is connected to every neuron in the subsequent layer. Activation functions (ReLU in this project) introduce non-linearities essential for learning complex representations.

2.2 Cross Entropy Loss:

The loss function measures the disparity between predicted and true class probabilities, guiding weight updates to minimize classification errors.

2.3 Optimizers:

Optimizers are algorithms used to update the weights of a neural network to minimize the loss function. They play a crucial role in determining how quickly and effectively the network learns.

1. **Stochastic Gradient Descent (SGD):** SGD is a simple and widely used optimizer that updates weights by computing the gradient of the loss with respect to the weights for each batch of training data. The weights are altered backpropogation by the following rule

$$\omega_{\text{new}} = \omega_{\text{old}} - \alpha \frac{\partial L}{\partial \omega},$$

where ω, L, α are the weights, loss function, and learning rate, respectively.

2. **Adam (Adaptive Moment Estimation):**

Adam is an adaptive learning rate optimizer that combines the benefits of two techniques: Momentum (which accelerates gradient descent in relevant directions) and RMSProp (which scales the learning rate based on recent gradients).

It maintains two moving averages for each weight: the first for the gradient m_t and the second for the squared gradient v_t:

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

where $g_t = \frac{\partial L}{\partial \omega}$ from above and β_1, β_2 are hyperparameters.

3. **RMSProp (Root Mean Square Propagation):** RMSProp modifies the learning rate for each weight based on a running average of the magnitude of recent gradients. This running average, $E[g^2]_t$, is updated as:

$$E[g^2]_t = \beta \cdot E[g^2]_{t-1} + (1 - \beta) \cdot g_t^2.$$

The weight update uses this average to normalize the gradient:

$$\omega_{\text{new}} = \omega_{\text{old}} - \frac{\alpha}{\sqrt{E[g^2]_t + \epsilon}} \cdot g_t,$$

where $\epsilon > 0$ is a small number for numerical stability and the other variables are the same as above.

2.4 Hyperparameter Tuning:

Critical parameters include learning rate, optimizer choice, weight initialization, and regularization (Dropout and Batch Normalization). Each affects model convergence, stability, and generalization ability.

2.5 Regularization Techniques:

- **Dropout:** Reduces overfitting by randomly deactivating neurons during training.
- **Batch Normalization:** Normalizes layer inputs to stabilize and accelerate training.

2.6 Weight Initialization:

- **Random Normal:** Simple distribution-based initialization.
- **Xavier:** Scales weights to improve gradient flow in networks with linear or sigmoid activations.
- **Kaiming:** Designed for ReLU activations, ensuring stable forward and backward passes.

3 Algorithm Implementation and Development:

- Input Layer: Dimension 784 (flattened 28x28 images).
- Output Layer: Dimension 10 (one-hot encoding for 10 classes).
- Adjustable Hidden Layers: Specified as a list of neuron counts for each layer (e.g., [128,64]).

3.1 Model Design:

- Input Layer: Dimension 784 (flattened 28x28 images).
- Output Layer: Dimension 10 (one-hot encoding for 10 classes).
- Adjustable Hidden Layers: Specified as a list of neuron counts for each layer (e.g., [128,64]).

3.2 Baseline Training Process:

1. Model initialization using PyTorch.
2. Weight initialization using selected methods.
3. Training using Cross Entropy Loss with the SGD optimizer.
4. Validation after each epoch to monitor generalization.
5. Testing after training to evaluate final accuracy.

3.3 Hyperparameter Tuning Experiments:

- **Optimizers:** Tested SGD, Adam, and RMSProp with varying learning rates.
- **Regularization:** Dropout applied after each hidden layer to mitigate overfitting.
- **Weight Initialization:** Evaluated Random Normal, Xavier Normal, and Kaiming Uniform.
- **Normalization:** Incorporated Batch Normalization after each layer for stable gradients.

3.4 Visualization and Analysis:

- Loss and accuracy curves plotted for each experiment.
- Results tabulated to compare methods quantitatively.

4 Computational Results

4.1 Baseline Configuration:

- **Parameters:** [128,64] hidden neurons, 10 epochs, learning rate 0.01.
- **Results:** Final Training Loss: 0.1623, Final Validation Accuracy: 88.90%, Mean Test Accuracy: 87.35%.

Figure 1 shows the training loss and validation accuracy curves for the baseline configuration.

4.2 Hyperparameter Tuning Results:

1. **Optimizer Comparison:** See Table 1 and Figures 2 and 3 below.

LR/Optim	SGD MTA	SGD SD	Adam MTA	Adam SD	RMSProp MTA	RMSProp SD
0.001	68.99%	3.28%	89.12%	2.02%	87.37%	2.34%
0.01	82.72%	2.02%	86.62%	1.85%	84.73%	2.04%
0.05	86.61%	1.86%	83.36%	2.98%	12.50%	2.13%

Table 1: Mean Training Accuracies and Standard Deviation Across Training Batches

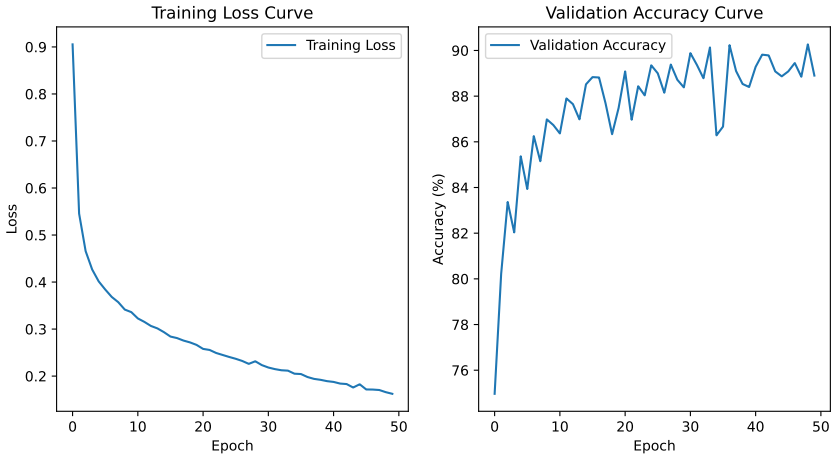

Figure 1: Training Loss and Validation Accuracy Curves for Baseline Configuration

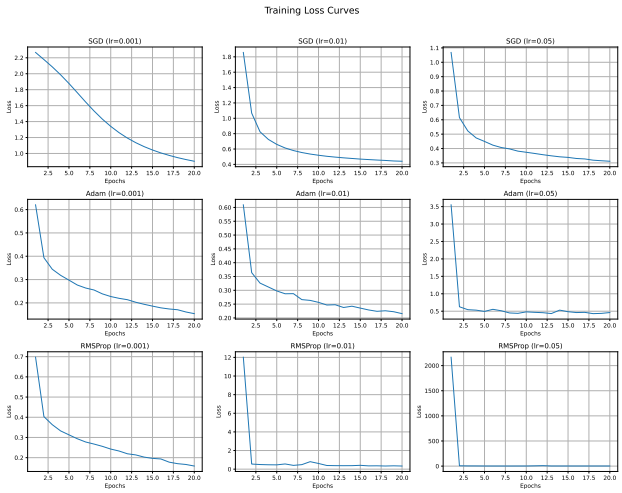

Figure 2: Training Loss Curves for Different Optimizers

2. **Dropout Regularization:** See Figure 4 below.
3. **Weight Initialization:** See Figure 5 below.
4. **Batch Normalization:** See Figure 6 below.

5 Summary and Conclusions

This project successfully implemented and optimized an FCN for FashionMNIST classification. The baseline model achieved 87.35% test accuracy, and further hyperparameter tuning improved mean training accuracy results to 89.49%. Key insights include:

- Adam and RMSProp are adaptive optimizers that enable faster convergence compared to SGD. However, higher learning rates cause instability, resulting in chaotic validation accuracy and high training loss, as seen in Figures 2 and 3. Lower learning rates stabilize updates and improve training performance.

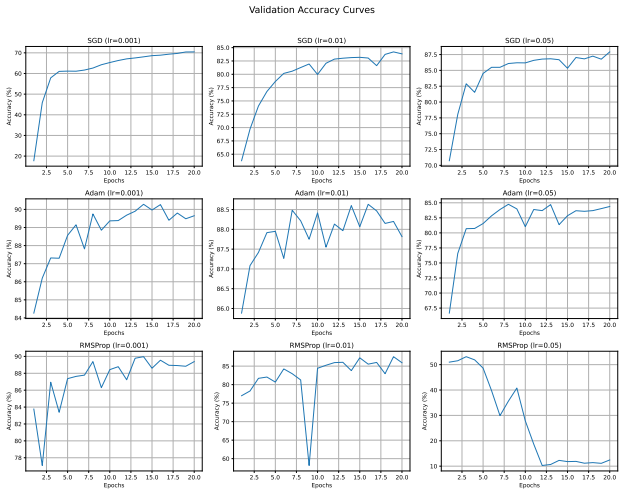

Figure 3: Validation Accuracy Curves for Different Optimizers

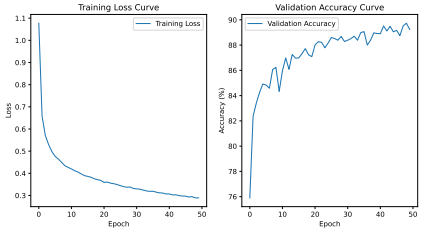

Figure 4: Training Loss and Validation Accuracy Curves with Dropout Regularization

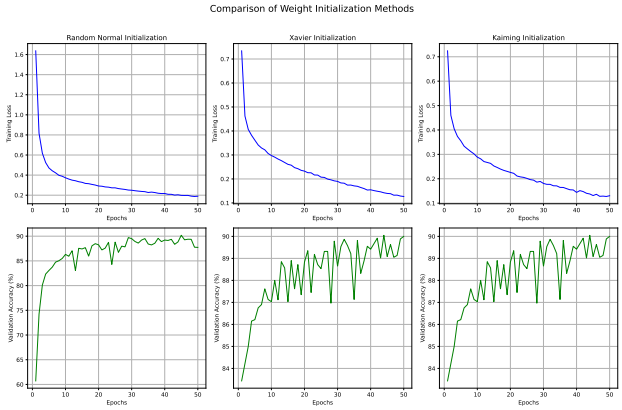

Figure 5: Training Loss and Validation Accuracy Curves for Different Weight Initializations

- Dropout reduces overfitting by randomly deactivating neurons during training, forcing the model to learn more robust and generalized features. This is evident in reduced gaps between training loss and validation accuracy in Figure 4, leading to better performance on unseen data.
- Xavier and Kaiming initializations improve gradient flow, enabling faster convergence and higher validation accuracy. However, their aggressive optimization can cause fluctuating validation accuracy in early training, as seen in Figure 5, which stabilizes as the model converges.
- Batch Normalization stabilizes gradient flow, but in this project, it did not significantly improve training

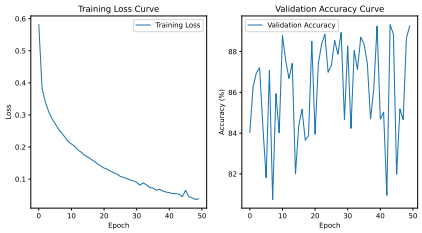

Figure 6: Training Loss and Validation Accuracy Curves for Batch Normalization

loss and led to fluctuating validation accuracy due to batch-dependent normalization, which may not fully align with the dataset's overall distribution. This is evident in Figure 6.

- Future work could explore advanced architectures (e.g., Convolutional Neural Networks) and automated hyperparameter search methods.

5.1 Acknowledgements

I would like to thank Rohin Gilman for their guidance and support throughout this project. Special thanks to my classmates for their valuable discussions and feedback.

5.2 References

1. LeCun, Y., Bengio, Y., Hinton, G. "Deep learning." Nature 521.7553 (2015): 436-444.

2. Srivastava, N., et al. "Dropout: A simple way to prevent neural networks from overfitting." Journal of Machine Learning Research 15.1 (2014): 1929-1958.

3. He, K., et al. "Delving deep into rectifiers: Surpassing human-level performance on ImageNet classification." Proceedings of the IEEE ICCV (2015).

4. Ioffe, S., Szegedy, C. "Batch normalization: Accelerating deep network training by reducing internal covariate shift." arXiv preprint arXiv:1502.03167 (2015).