AMATH 482: HOMEWORK 4

NICHOLAS NUGRAHA

Applied Math Department, University of Washington, Seattle, WA
`nickjn@uw.edu`

Abstract. This report explores the use of Fully Connected Neural Networks (FCNNs) for classifying the FashionMNIST dataset. Various optimizers, regularization techniques, and initialization strategies are analyzed to determine their impact on model performance. The study evaluates Stochastic Gradient Descent (SGD), RMSprop, and Adam optimizers, showing that Adam achieves the highest testing accuracy of 88.66%, outperforming SGD and RMSprop. Additionally, dropout regularization and different weight initializations, including Random Normal, Xavier Normal, and Kaiming Uniform, are tested to enhance model generalization. Results indicate that Random Normal initialization improves validation stability, while batch normalization unexpectedly reduces testing accuracy to 82.34%. This study highlights how different hyperparameter choices influence deep learning model performance on complex classification tasks.

1. Introduction and Overview

In this report, we will be training a fully connected deep neural network on the popular FashionMNIST dataset. The FashionMNIST dataset consists of 28×28 grayscale images that depict articles of clothing that belong to a class from one of 10 classes. The training set has 60K images and the testing set has 10K images. This task is more difficult than the MNIST digit classification because classifying articles of clothing requires more powerful and nonlinear models.

2. Theoretical Background

Fully connected neural networks are types of artificial neural networks where each neuron in one layer is connected to every neuron in the next layer. FCNNs are a fundamental architecture in deep learning and are commonly used for classification tasks like the FashionMNIT dataset. They have the ability to approximate complex functions and learn nonlinear relationships in the data through the multiple layers of interconnected neurons.

A FCNN consists of three main components, the first being the input layer, which takes in feature vectors $x \in \mathbb{R}^d$ where d is the input dimension. The second are the hidden layers, which transform the input layer through weighted connections and activation functions. The third component is the output layer, which produces the final predictions. It can be thought of as an actual human brain where we receive signals through our senses (input layer) and our brain activates specific neurons (hidden layers) which then form a thought or an action we perform (output layer). The mathematical formulation of a single layer transformation follows the equation:

$$z^{(l)} = W^{(l)} a^{(l-1)} + b^{(l)}$$

$$a^{(l)} = \sigma(z^{(l)})$$

Date: March 14, 2025.

where $z^{(l)}$ represents the weighted sum of inputs received by a neuron in layer l. The weight matrix $W^{(l)}$ contains the trainable parameters that determine the strength of the connections between neurons in successive layers, while the bias vector $b^{(l)}$ allows the network to shift the activations in order to improve learning flexibility. The activation function σ applies a nonlinear transformation to $z^{(l)}$, which is where our nonlinearity comes from. The output of this transformation, $a^{(l)}$, represents the neuron activations that are passed to the next layer. The choice of activation function depends on the problem at hand, however, we will be using the Rectified Linear Unit (ReLU) function, $ReLU(x) = max(0, x)$, which is a common function in deep networks.

In order for our model to actually learn, it has to know where it went wrong with its guess. This is why we use the loss function to measure the difference between predicted and actual values. Since we are classifying multiple categories of clothes, we will use the categorical cross-entropy loss function, used for multi-class classification.

Training a FCNN involved backpropagation and gradient descent optimization. During the forward pass, activations are computed layer by layer. The loss function is then used to quantify the error. In the backward pass, the gradients of the loss function (how much the weights change in relation to the error) with respect to the model parameters are computed. The model parameters are then updated using gradient descent:

$$W^{(l)} \leftarrow W^{(l)} - \eta \frac{\partial L}{\partial W^{(l)}}$$

where η is the learning rate. We will use various optimization algorithms, including Stochastic Gradient Descent (SGD), Adam (Adaptive Moment Estimation), and RMSprop. In order to improve generalization and prevent overfitting, we employ the dropout regularization technique which randomly drops a fraction of the neurons during training to prevent over-reliance on specific features. We also use batch normalization to stabilize training by normalizing layer activations, improving speed and performance.

3. Algorithm Implementation and Development

We utilize the already existing `PyTorch` library to build our model as they have all the functions mentioned in the previous section ready for use. Along with that, we use `numpy`'s package for mathematical functions, `matplotlib` to visualize our findings, and `sklearn` to split our data into training and validation sets.

Our neural network has an adjustable number of hidden layers and neurons in each layer. In our model the input layer has 784 neurons (28 x 28 images), and we choose to use 2 hidden layers, the first with 128 neurons and the second with 64 neurons. Our output layer has 10 neurons, each corresponding to one of the 10 classes. We initially use the SGD optimizer, but we also experiment with *RMSprop* and *Adam* as well.

4. Computational Results

We will first define a baseline that our model performs at using the SGD optimizer. We found that a learning rate of 0.08 and 50 epochs for the SGD optimizer were sufficient to gain a reasonable testing accuracy of 86.99% at an elapsed time of 2m 40s. The number of epochs will stay consistent at 50 epochs throughout all tuning processes.

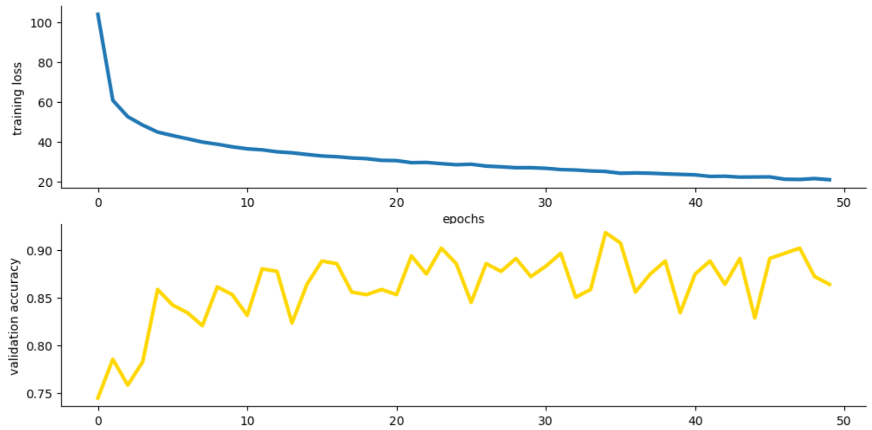

FIGURE 1. Training loss and validation accuracy curves for the SGD optimizer with learning rate = 0.08

Now that we have a baseline to compare to, we can perform hyperparameter tuning from this baseline by changing what optimizers we use, including regularization, considering different initializations, and including normalization.

The RMSprop optimizer yielded a test accuracy of 88.34% with learning rate 0.001 and an elapsed time of 2m 33s.

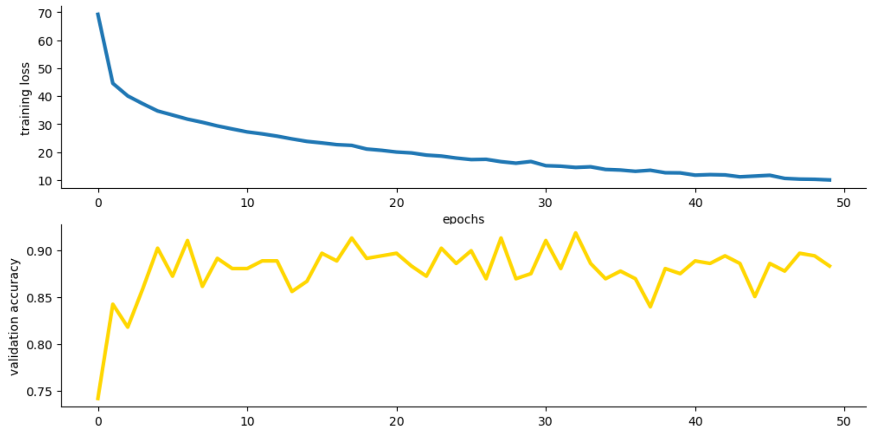

FIGURE 2. Training loss and validation accuracy curves for the RMSprop optimizer with learning rate = 0.001

The Adam optimizer yielded a test accuracy of 88.66% with learning rate 0.001 and an elapsed time of 2m 31s.

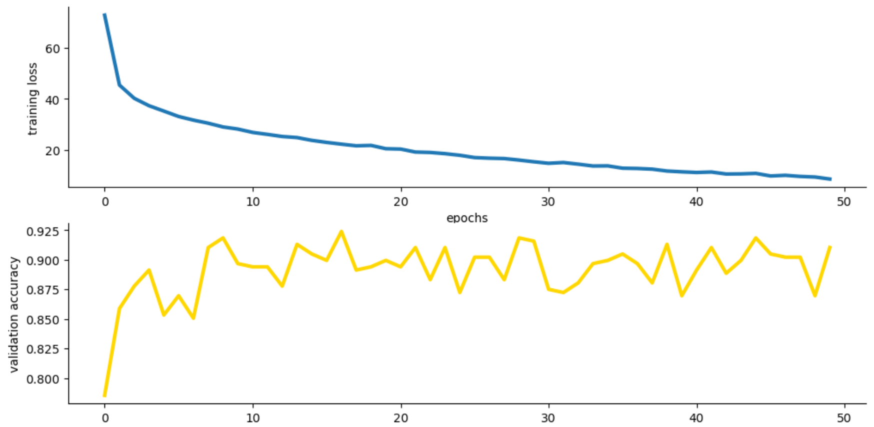

FIGURE 3. Training loss and validation accuracy curves for the Adam optimizer with learning rate = 0.001

Optimizer	**Learning Rate**	**Testing Accuracy**
SGD	0.08	86.99%
RMSprop	0.001	88.34%
Adam	0.001	88.66%

TABLE 1. Learning rate and testing accuracy of different optimizers

Across the three different optimizers we explored, the Adam optimizer had the highest testing accuracy, although not by a lot compared to the RMSprop optimizer. All three optimizers have reasonable training loss curves and the RMSprop/Adam optimizers show a consistently higher validation accuracy in Figures 2 and 3, compared to the baseline SGD optimizer as seen in Figure 1. Since the dataset is medium sized and contains a lot of noise, Adam performs the best because of its ability to handle noisy gradients well, as well as not over-relying on the chosen batch size.

By including dropout regularization on the SGD optimizer, the testing accuracy increases to 88.25%, compared to the 86.99% accuracy without dropout regularization.

Now we will consider different initializations, starting with the Random Normal.

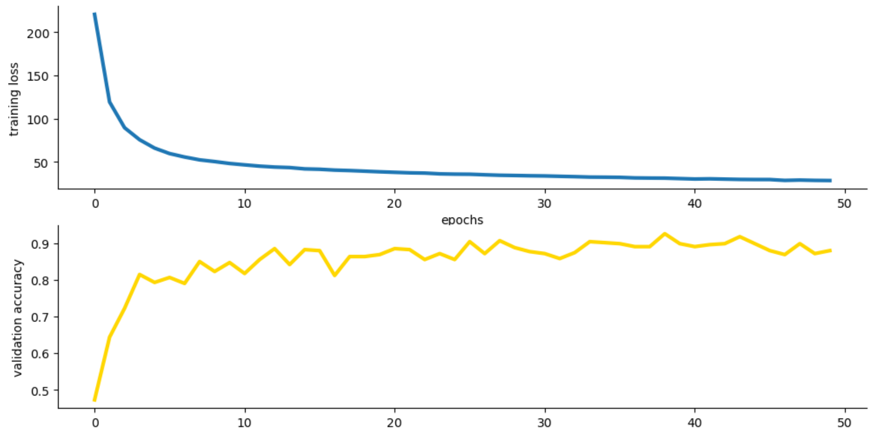

FIGURE 4. Training loss and validation accuracy curves for SGD optimizer with random normal initialization

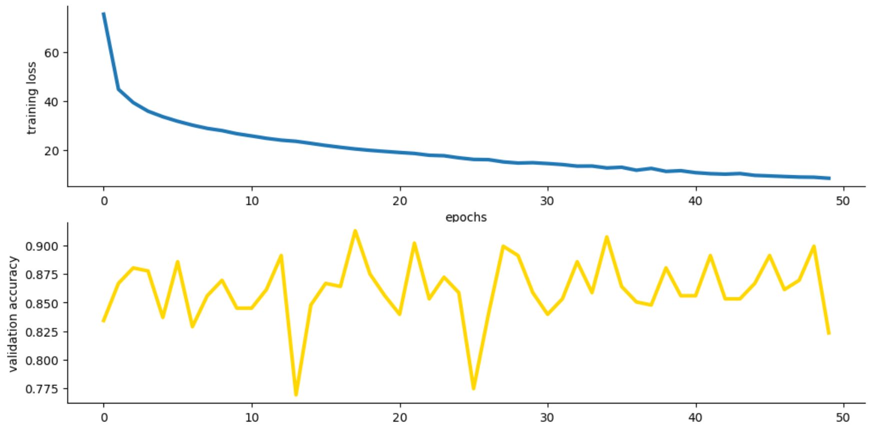

Figure 5. Training loss and validation accuracy curves for SGD optimizer with batch normalization

The testing accuracy of SGD with random normal initialization is 87.39%. An interesting difference we noticed was that the validation accuracy was much more stable in figure 4 compared to the curve without initialization.

The Xavier Normal initialization yielded a testing accuracy of 87.28%, while the Kaiming Uniform initialization yielded a test accuracy of 87.50%. We didn't notice the same validation accuracy stability with these two initializations as we did with the random normal initialization. Overall, the initializations improved the accuracy of the model, and random normal seemed to have a pretty stable validation accuracy.

Now we will include batch normalization with our SGD optimizer. With batch normalization, the testing accuracy decreased to 82.34%. The validation accuracy curve in figure 5 also fluctuates a lot more than the baseline in figure 1.

5. Summary and Conclusions

In this report, we investigated different methods to improve the classification accuracy of a Fully Connected Neural Network (FCNN) on the FashionMNIST dataset.

Our baseline model used the SGD optimizer with a learning rate of 0.08, achieving a testing accuracy of 86.99%. We then explored RMSprop and Adam optimizers, with Adam slightly outperforming the other two at 88.66% accuracy. The training loss curves and validation accuracy plots confirmed that Adam consistently produced higher validation accuracy and faster convergence, making it the most effective optimizer for this dataset.

Further improvements were tested by adding dropout regularization, which increased the test accuracy of SGD from 86.99% to 88.25%, confirming that dropout helps prevent overfitting. We also explored different weight initialization methods and found that Random Normal initialization resulted in the most stable validation accuracy, improving model reliability. However, batch normalization unexpectedly reduced test accuracy to 82.34%, suggesting that it may not be well-suited for this specific architecture or dataset.

Acknowledgements

The author is thankful to Professor Natalie Frank for useful lectures about neural networks and classification methods. I also thank TA Rohin Gilman for assisting in office hours. I acknowledge that artificial intelligence was used to make edits to this document, fix minor bugs, and gain a better understanding of the concepts.

References

[1] T. N. Community. *NumPy Documentation*, 2025. Accessed: 2025-01-25.
[2] N. Frank. Amath 482: Computational methods for data analysis. Lecture notes, University of Washington, 2025.
[3] J. N. Kutz. *Data-driven modeling & scientific computation: methods for complex systems & big data*. OUP Oxford, 2013.
[4] OpenAI. Chatgpt. `https://chat.openai.com/`, 2025.
[5] PyTorch Developers. *PyTorch: An Open Source Machine Learning Framework*, 2024. Accessed: 2025-03-13.