AMATH 482/582: HOME WORK 4

SARAYU GUNDLAPALLI

Applied Math, University of Washington, Seattle, WA
saru07g@uw.edu

Abstract. This report presents the implementation and analysis of fully connected deep neural networks (FCNs) for image classification on the FashionMNIST dataset. The FashionMNIST dataset contains 60,000 images for training and 10,000 images for testing. The goal was to design an FCN architecture, that classifies the images into one of the ten categories of clothing (classes), with adjustable hyperparameters, and optimize it to achieve high accuracy. After establishing a baseline model, extensive hyperparameter tuning was performed by varying the optimizer, regularization, initialization schemes, and normalization methods. Through this comprehensive analysis, optimal parameters and strategies for training FCN models on FashionMNIST are identified and discussed.

1. Introduction and Overview

In this report, we explore deep neural networks, specifically fully connected networks (FCNs), for classifying images in the FashionMNIST dataset. FashionMNIST serves as a more complex replacement for the well-known MNIST dataset and better reflects real-world image classification challenges. This data set is composed of 70,000 28x28 photos of clothing, which are sorted into 10 groups.

We establish a baseline configuration of the FCN model by experimenting with different hyperparameters to achieve reasonable validation accuracy ($> 85\%$) training loss. Next, we conduct extensive hyperparameter tuning to optimize the performance of the model. This includes exploring different optimization algorithms, adjusting learning rates, implementing regularization techniques like dropout, and analyzing the effects of various weight initialization methods and normalization techniques.

2. Theoretical Background

2.1. **Neural Networks:** Neural networks are a powerful machine learning model inspired by biological neural networks in the brain. They excel at tasks like image classification by learning intricate patterns in the input data during training. A neural network consists of layers of interconnected nodes, where each connection has an associated weight. The first layer is the input layer, with one node for each input feature (e.g. 784 nodes for a 28x28 pixel image). The final layer is the output layer, with one node per class.

Between the input and output are one or more hidden layers that perform the learning and pattern recognition. Each node in a hidden layer calculates a weighted sum of the inputs from the previous layer, adds a bias term, and passes the result through an activation function. The network's weights and biases are iteratively adjusted during training to minimize a loss function between predicted and true outputs using optimization algorithms like gradient descent. This

Date: March 4, 2024.

layered, weighted architecture allows neural nets to model highly complex, non-linear functions and extract relevant features from high-dimensional inputs like images.

$$y = f(\sum_{i=1}^{n} x_i w_i + b) \tag{1}$$

in the above equation, f() is the activation function, w_i is the weights, x_i is the input data, b is the biases, and y is the output.

2.1.1. ***Forward Pass:*** The forward pass takes the inputs, passes them through the network, and allows each neuron to react to a fraction of the input. Neurons generate their outputs and pass them on to the next layer until eventually the network generates an output

2.1.2. ***Back Propagation:*** Backpropagation tracks the derivatives of the activation functions in each successive neuron, to find weights that bring the loss function to a minimum, which will generate the best prediction.

2.1.3. ***Batch:*** Training set could be divided into smaller sets called batches.

2.1.4. ***Epoch:*** When an entire dataset is passed both forward and backward through the NN once.

2.2. **Activation Function:** In equation (1), By plugging $(\sum_{i=1}^{n} x_i w_i + b)$ into the activation function, we can set the neuron to be active if the sum is greater than a threshold, and inactive if it is less than the threshold. It makes it easy for the model to generalize or adapt to a variety of data and to differentiate between the output.

2.2.1. *Rectified Linear Unit Function (ReLU):.* Applying ReLU to a layer of neurons means that all negative entries become zero, and all positive entries get left alone.

$$f = max(x, 0)$$

2.3. **Fully-Connected Neural Network:** Fully connected means that each neuron is connected to every other neuron in the previous layer. When each neuron is directly connected to the neuron in the previous layer it is a dense layer. A fully-connected neural network is made up of only dense layers.

2.4. **Loss function:** The Loss function is a method of evaluating how well your algorithm is modeling your dataset. Here we use the cross-entropy loss function:

$$L(\hat{y}, y) = -(ylog\hat{y} + (1 - y)log(1 - \hat{y}))$$

2.5. **Gradient Descent:** Training a neural network requires using a gradient descent algorithm for optimization. The Stochastic Gradient Descent Algorithm is an iterative algorithm that finds the minimum of a convex function.

3. Algorithm Implementation and Development

- Initialize the FCN model with adjustable parameters: number of hidden layers, neurons per layer, ReLU activation, and input/output dimensions.
- Use Cross Entropy loss for classification.
- Optimize with SGD and adjust the learning rate.
- Train for an adjustable number of epochs, monitor training loss, validate accuracy on the validation set, and evaluate on the testing set.
- Test RMSProp, Adam, and SGD with different learning rates.
- Apply Dropout regularization to the baseline model with the most suitable optimizer and learning rate.
- Try Random Normal, Xavier Normal, and Kaiming (He) Uniform initialization.

- Implement Batch Normalization.

Packages used:

- NumPy[1]
- scikit-learn [4] for mathematical calculations
- Matplotlib[2] for visualizations.
- Pytorch[3] for building deep learning models.
- seaborn[5] for data visualisation.

4. Computational Results

For task 1, we tested several parameters (epoch = 10, LR = 0.001) which gave us an accuracy of around 70%. We arrived at the baseline NN model with the given parameters1 to obtain a validation accuracy of 86.016% (above 85%) and a testing accuracy of 85.44%.

HL	HL dimensions	Learning rate	Epochs
2	400, 400	0.05	15

Table 1. Baseline parameters for validation accuracy $> 85\%$

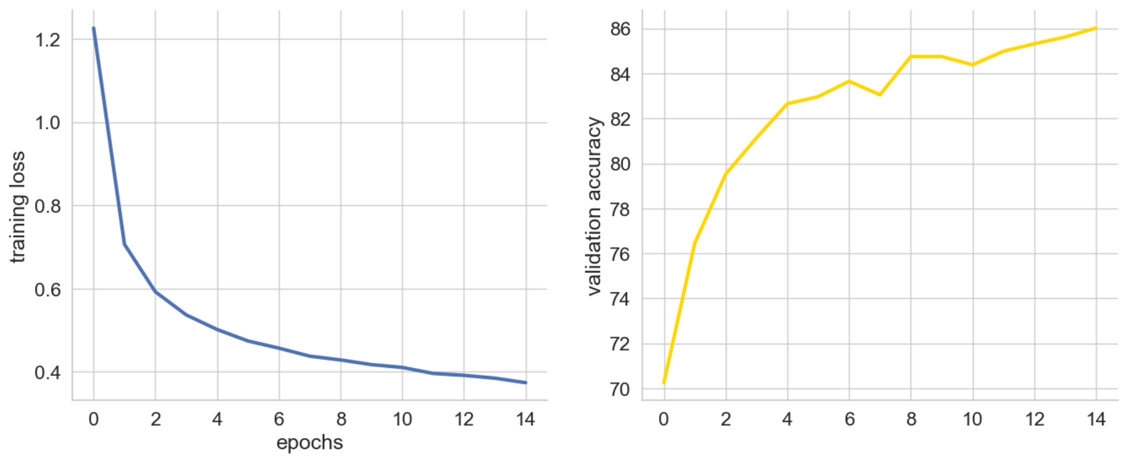

Figure 1. Training loss and validation accuracy for the baseline model.

We performed hyperparameter tuning from the baseline by varying the optimizer. We considered different optimizers including RMSProp, Adam, and SGD with different learning rates. The training loss, validation, and test accuracy for each are shown 235.

We see that the optimizer Adam at LR = 0.001 is the most suitable for our FCN model i.e. gives the highest accuracy. Adam might be the better optimizer algorithm for our FCN because its adaptive learning rate mechanism allows it to handle sparse gradients effectively, leading to faster convergence and better generalization. Adam typically converges faster compared to RMSprop and SGD, especially in the early stages of training.

Further, we analyze the overfitting/underfitting situation of our model with the best optimizer and see that there is clear overfitting (high variance)3. In particular, the model performs well on training data but poorly on the new data in the validation set.

We try to combat this by including a *Dropout* regularisation of 0.5 in the model and notice that the model was indeed overfitting before, and dropout helped mitigate this issue without sacrificing accuracy on the training data (the accuracy remains almost the same32. Dropout prevents the model from relying too heavily on specific features or complex interactions between features in the

RMSProp				
Learning Rate	0.001	0.01	0.05	0.1
Validation Accuracy	89.6166	86.4666	19.4833	26.0166
Test Accuracy	87.8399	84.77	19.0	25.9199

TABLE 2. Results for Optimizer: RMSProp at different learning rates.

Adam				
Learning Rate	0.001	0.01	0.05	0.1
Validation Accuracy	90.6333	89.35	85.4333	37.8666
Test Accuracy	89.01	87.31	83.64	38.47

TABLE 3. Results for Optimizer: Adam at different learning rates.

SGD				
Learning Rate	0.001	0.01	0.05	0.1
Validation Accuracy	65.5666	81.95	86.2833	87.8166
Test Accuracy	63.93	80.63	84.49	85.77

TABLE 4. Results for Optimizer: SGD at different learning rates.

training data. This leads to a reduction in overfitting, which can be observed by better performance on unseen data (validation or test set).

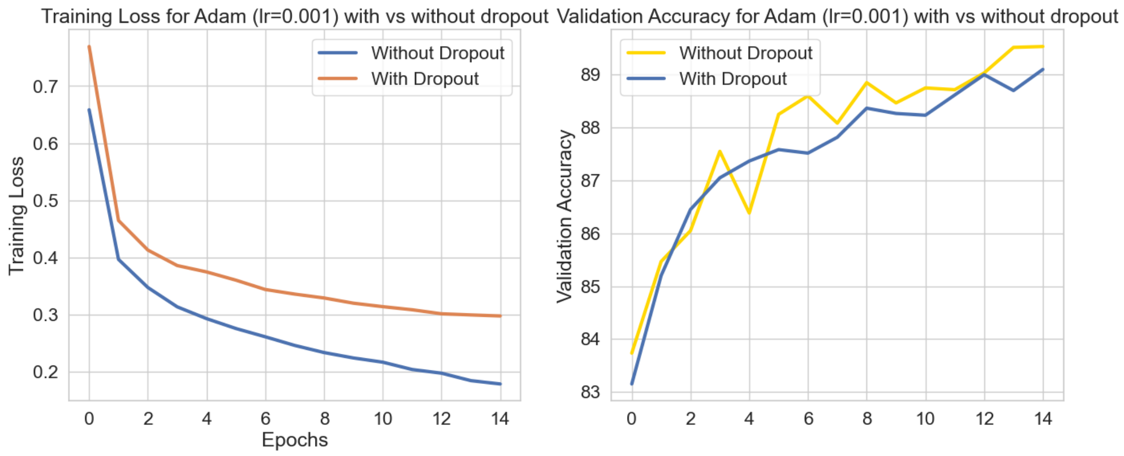

FIGURE 2. Training loss and validation accuracy for Adam optimizer (LR = 0.001).

We consider different initializations, such as Random Normal, Xavier Normal, and Kaiming (He) Uniform on the base model (no dropout). We observe that the accuracy decreases by almost 8% when using random normal initialization compared to other initialization methods. On the other hand, we observe that the accuracy remains consistent when using Xavier normal and Kaiming uniform initialization methods. This consistency suggests that these initialization methods are more effective in initializing weights that facilitate learning and lead to higher accuracy45.

Despite the decrease in accuracy, we notice that random normal initialization works well for reducing overfitting, while the other initialization methods (Xavier normal and Kaiming uniform) do not perform as well in this aspect. This suggests that random normal initialization might introduce

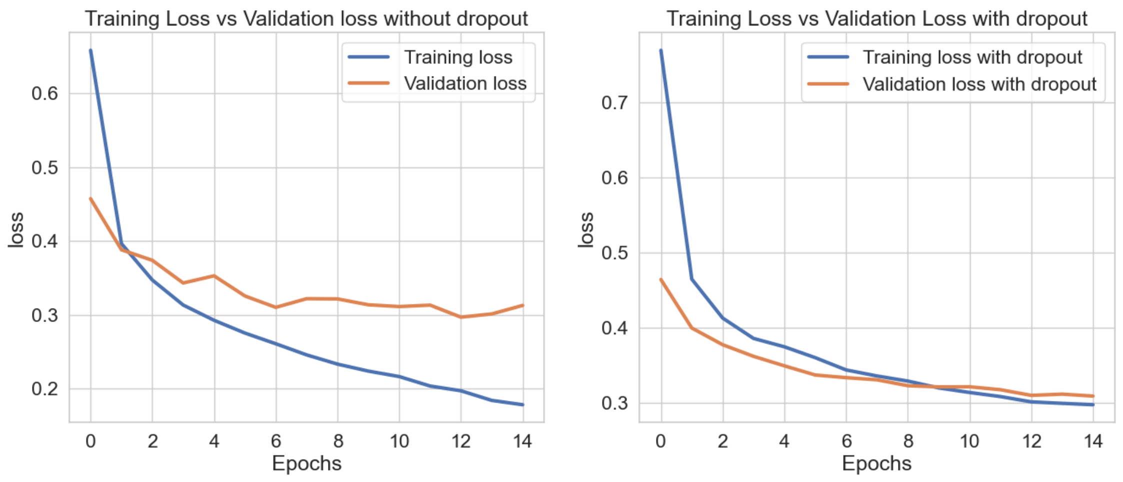

Figure 3. Training loss vs validation loss with and without dropout.

sufficient noise in the model's parameters, helping prevent it from memorizing the training data too closely4.

Initialization	Random Normal	Xavier Normal	Kaiming (He) Uniform
Validation Accuracy	81.8333	89.0666	89.0833
Test Accuracy	81.5899	88.69	88.7

Table 5. Results for different initializations.

Finally, we include normalization such as Batch Normalization (also on the base model, using Adam optimizer). The Batch normalization did not have much effect on the accuracy, suggesting that the model was not suffering from issues related to internal covariate shift, to begin with. It is also noticed that it does not have much effect on the overfitting situation. In some cases, if the model is highly complex and has a large number of parameters, batch normalization alone may not provide enough regularization to prevent overfitting. Adding additional regularisation like dropout or weights might be necessary to reduce the overfitting in the model.

5. Summary and Conclusions

Neural Networks are an incredibly versatile tool for solving a wide variety of optimization problems and they are especially practical for image classification. We implemented an adjustable FCN architecture in PyTorch and performed comprehensive hyperparameter tuning experiments to maximize accuracy. This included evaluating different network architectures, optimizers, learning rates, regularization techniques, initialization schemes, and other settings. Our best model achieved 90.6% testing accuracy on FashionMNIST by using Adam Opimiser at a learning rate of 0.001. Further hyperparameter tuning was done on this base model to analyze how they affect the overfitting and overall training process and accuracy.

Acknowledgements

The author is thankful for the suggestions/advice about the algorithm implementation from peers in AMATH 482/582. The author is thankful to the Teaching Assistants, Saba Heravi and Juan Felipe Osorio Ramírez, for their invaluable time and fruitful discussions about setting up the model. Further, the author is thankful to Prof. Eli Shlizerman for useful discussions about the course content.

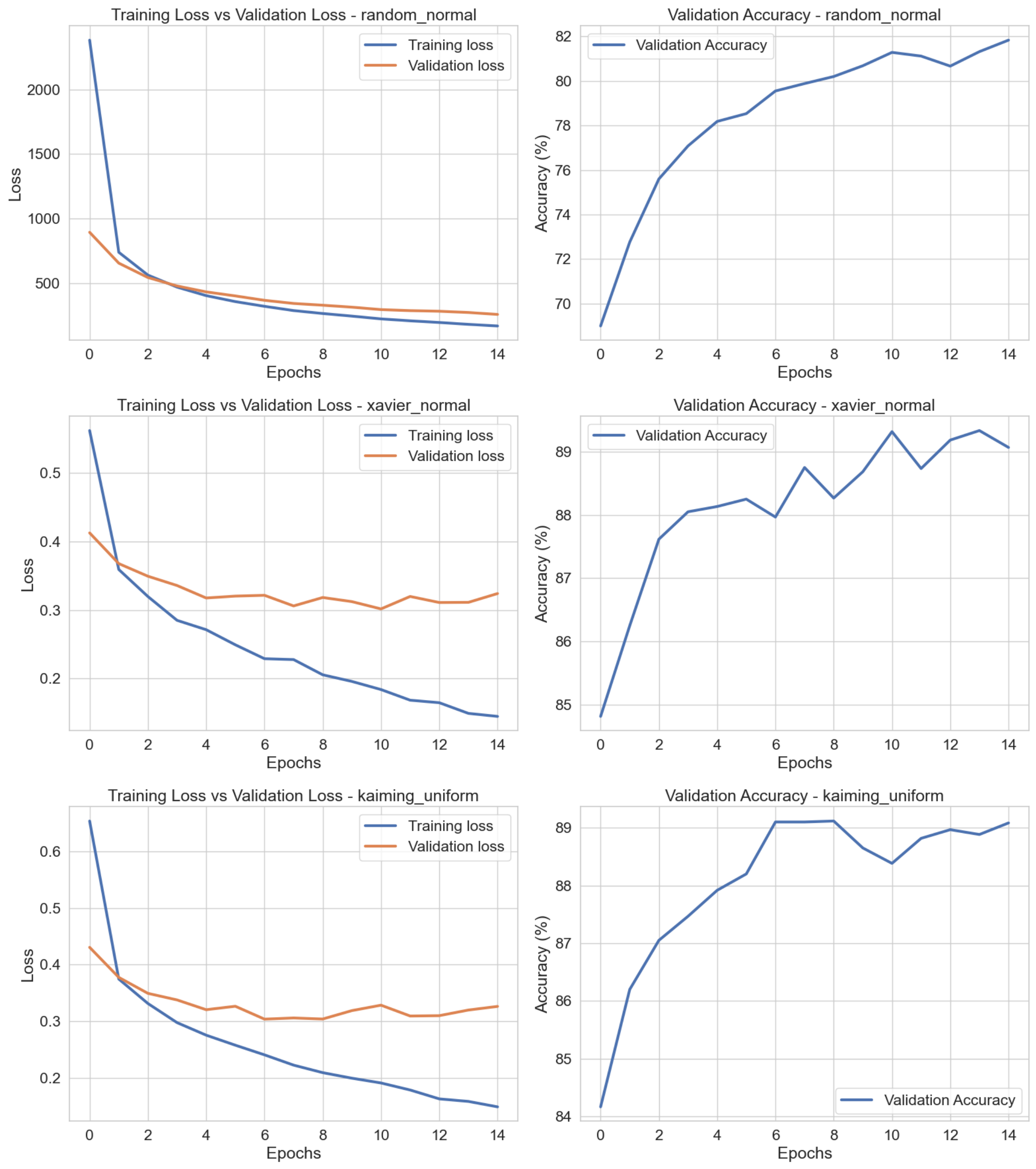

FIGURE 4. Different initializations and how they affect the model.

REFERENCES

[1] C. R. Harris, K. J. Millman, S. J. van der Walt, R. Gommers, P. Virtanen, D. Cournapeau, E. Wieser, J. Taylor, S. Berg, N. J. Smith, R. Kern, M. Picus, S. Hoyer, M. H. van Kerkwijk, M. Brett, A. Haldane, J. F. del Río, M. Wiebe, P. Peterson, P. Gérard-Marchant, K. Sheppard, T. Reddy, W. Weckesser, H. Abbasi, C. Gohlke, and T. E. Oliphant. Array programming with NumPy. *Nature*, 585(7825):357–362, Sept. 2020.

[2] J. D. Hunter. Matplotlib: A 2d graphics environment. *Computing in Science & Engineering*, 9(3):90–95, 2007.

[3] J. Kutz. *Methods for Integrating Dynamics of Complex Systems and Big Data*. Oxford, 2013.

[4] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12:2825–2830, 2011.

[5] M. L. Waskom. seaborn: statistical data visualization. *Journal of Open Source Software*, 6(60):3021, 2021.