HYPERPARAMETER TUNING OF FULLY CONNECTED NEURAL NETWORKS FOR ROBUST IMAGE CLASSIFICATION

CHENAB

Applied and Computational Mathematics, University of Washington, Seattle, WA
chenab@uw.edu

Abstract. In the following paper, we present a Fully Connected Deep Neural Network (FCN /DNN) architecture to classify images in the FashionMNIST data set. We start with a baseline parameters for the optimizer, learning rate, drop out regularization etc, and then perform hyperparameter tuning to successfully get a peak test accuracy of 90.19% on the FashionMNIST and 98.4% accuracy on the MNIST.

1. Introduction and Overview

The FashionMNIST data set is a large database of fashion images, it contains 60,000 training images and 10,000 testing images. Each image is 28x28 pixels and gray-scale. We split the training data into Training and Validation sets.We define our training batch size and our validation batch size to be 512 and 256 respectively. Our neural network has input dimension 784 (for the pixels) and output dimension 10 for the classification. Moreover, it possesses an adjustable number of hidden layers, initially set to 2 layers with 512 and 256 number of neurons. We use ReLU as our activation function, and cross entropy loss to perform the classification. Moreover, we test various optimizers, including SGD, Adam and RMSProp, as well as various drop our regularization values, initilizations as well as batch normalization.

2. Theoretical Background

A **Neural Network** is a computational model inspired by the brain, useful for identifying complex patterns and making predictions from data. A **Fully Connected Neural Network** is one where each neuron in one layer is connected to every neuron in the next layer. They are essentially composed of several interconnected nodes, called neurons, organized into layers composed of simple units. The data is passed through the input layer, following which it goes through multiple hidden layers, and a final output layer. Each unit computes a weighted sum of its inputs, which can be passed through a nonlinear function. We use **ReLU** $f(x) = \max(0, x)$. It outputs the value x if it is positive, otherwise it outputs zero. The learning algorithm then changes the weights. The goal is to minimize some **loss function** (cross-entropy loss in our implementation). Cross entropy loss is given by $\mathcal{L}(\mathbf{y}, \hat{\mathbf{y}}) = -\sum_{c=1}^{M} y_c \log(\hat{y}_c)$

This is done through **back propagation**, an algorithm used to compute the gradient of the loss function. During the forward pass, each layer l, $z^l = Wa^{l-1} + b^l$ is computed. The backward pass involves computing the loss function at the output layer, then step-by-step computing the gradients with respect to the preceding layers. The process is repeated layer-by-layer until the input is reached.

$$\nabla_{\vec{w}} L(\hat{y}, y) = \frac{\partial L}{\partial \hat{y}} \frac{\partial \hat{y}}{\partial z} \nabla_{\vec{w}} z$$

Once we have the gradient, we update our weights. **Gradient Descent** is a popular algorithm. It iteratively updates the network weights based on the gradient of the loss function J with respect to the parameters. Here α is the learning rate, or step-size, representing how fast the model converges.

$$\vec{w}_{k+1} = \vec{w}_k - \alpha \nabla_{\vec{w}} J(\vec{w}_k; b)$$

Date: March 25, 2025.

$$b_{k+1} = b_k - \alpha \frac{\partial}{\partial b} J(\vec{w}; b_k)$$

Optimizers are algorithms that adjust the network parameters during training. We will be testing the following optimizers:

- **Stochastic Gradient Descent:** a modified version of the standard gradient descent algorithm where the gradient of the loss function is computed based on a smaller randomly selected sample (hence stochastic) instead of the entire data set. This makes computation much more efficient and feasible to run on large data sets.
- **AdaM:** Combines momentum and adaptive elements of RMSProp to provide a more robust algorithm. *"Momentum refers to accelerating the gradient descent process by incorporating an exponentially weighted moving average of past gradients"*. It also has dynamic learning rates, based on the values of past gradients.
- **RMSProp:** It is an adaptive algorithm that makes use of a moving average of squared gradients. RMSProp gives more weight to recent gradients, essentially scaling each parameter's learning rate, providing additional stability and better convergence

Dropout Regularization is a technique we will use to reduce overfitting in our neural network. During each training iteration, a random subset of neurons is "dropped", which means their output is set to 0. The probability of being dropped is calculated through a Bernoulli distribution. This essentially removes that neuron's effect for the particular training iteration. This prevents over-reliance on specific neurons and spreads the weights, forcing the network to learn redundant patterns, hence making it less dependent on the training data and more robust.

Initialization:

- **Ramdom Normal:** Uses a normal distribution with given mean and variance to sample values to initialize the weights.
- **Xavier Normal:** Factpors in the number of input and output units of the layer to keep similar variance across the layers of the network. $W \sim N(0, \frac{2}{n_{in}+n_{out}})$
- **Kaiming Uniform:** Well suited for networks using ReLU. Addresses the vanishing gradient problem, when gradients become extremely small as they are propagated backward through multiple layers. It uses a normal distribution with mean 0 and standard deviation $\sqrt{n}$, with n being number of inputs to the node.

Batch Normalization is a technique that involves normalizing the outputs of a layer each mini-batch during training, ensuring it has mean 0 variance 1. It reduces internal covariate shift technique, improving stability and speed of a neural network.

3. Algorithm Implementation and Development

The following algorithm was implemented when training and tuning for hyperparameters in our neural network.

1.) Initial Setup and Data Processing

We first load from the FashionMNIST database our training and test data. We then create a validation (10%) from our training set based on indices. We then define the batch size of our training and validation data to be 512 and 256 respectively, and load the data into each of training batch, validation batch and test batch using **DataLoader** function from **Pytorch** library

2.) Defining our Fully Connected Neural Network architecture We define our FCNN that has input dimension 784, output dimension 10. We design it to have an adjustable number of hidden layers. We set it to two hidden layers, with 512 and 256 number of neurons respectively. We define it to use the ReLU activation function and also enable funcionality for dropout regularization and batch normalization.

3.) Creating a Model training pipeline

This is the backbone of our implementation. We define a train model function that takes in parameters like model, epochs, learning rate, loss function, and the optimizer type, and trains the model through the following algorithm:

- (i) for each epoch, sets the model to training mode using **train** fuction of **pytorch**, initializing variable to keep track of the loss for the epoch.

- (ii) iterate though batches of training data, reshaping the input features from 2D 28×28 to 1D.
- (iii) clear previous gradients using **zero grad** from **pytorch** and perform forward pass to get predictions and calculate the loss between predictions and labels using **loss func**
- (iv) calculate gradients through backpropagation using **backward** function from python and update model parameters based on calculated gradients using **step** function, and update the epoch loss.
- (v) Calculate overall training loss for the epoch by averaging loss accumulated over all training batches
- (vi) Repeat step (ii) to (iv) for the validation batch.
- (vii) compute prediction accuracy for the validation batch
- (viii) Report training loss, validation accuracy for each epoch
- (ix) Set the model to **eval** mode and using **nograd** from**pytorch** to stop passing inputs. This enables us to compute the final **Test accuracy**

4.) Setting baseline then Hyper parameter tuning. We experiment with our hyperparameters: learning rate, epochs, initializations, drop out regularization and normalization to arrive at a baseline configuration that has 85% test accuracy. We sequentially apply hyper parameter tuning using the following algorithm:

- (i) We explore different combinations of learning rates and optimizers, keeping other parameters same as baseline, and then compare the validation accuracy, and the test accuracy for the best possible combination with the baseline model.
- (ii) Include dropout regularization and compare test accuracy.
- (iv) Consider different initializations, Random Normal, Xavier Normal, Kaiming (He) Uniform, and log their validation and test accuracies.
- (iv)Include batch normalization and compare the test accuracy.

4. Computational Results

We use the following baseline configuration to reach a **85.68%** test accuracy on the FashionMNIST dataset.

Table 1. Baseline Model Configuration

Parameter	**Value**
Epochs	50
Loss function	Cross-Entropy
Initialization	Default
Learning rate	0.01
Optimizer	SGD
Dropout regularization	No
Batch normalization	No
Test accuracy	**85.68%**

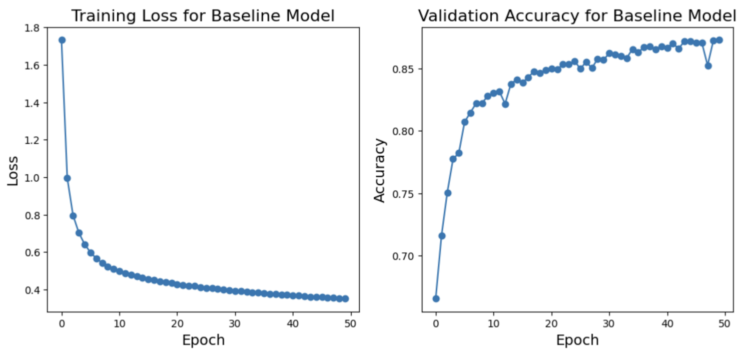


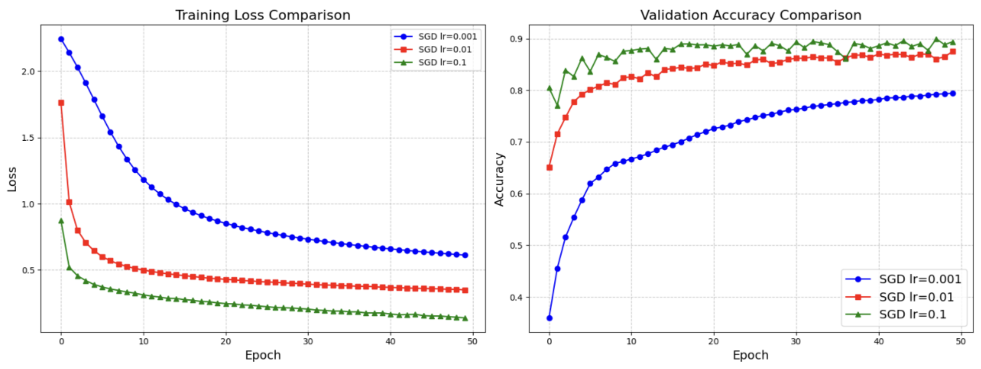
Figure 1. Training loss and Validation Accuracy vs Epoch for Baseline Configuration

We then perform hyperparameter tuning, starting with adjusting the optimizer and learning rates, keeping every other parameter **the same** as baseline and get the following results:

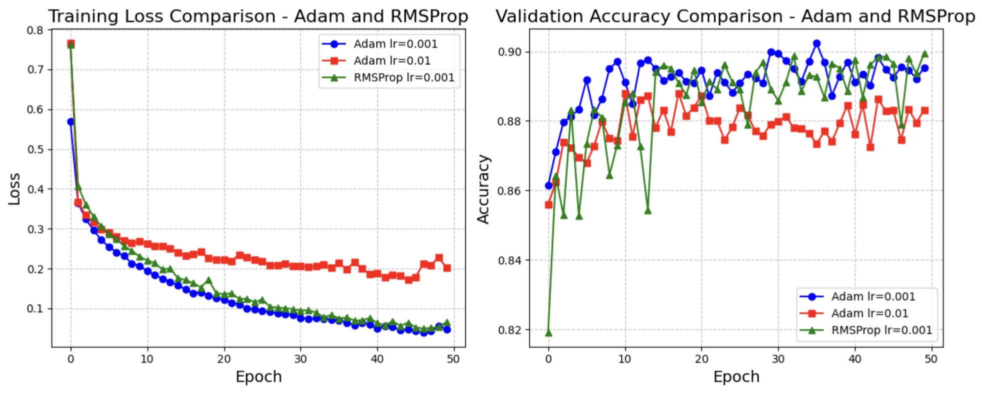
Table 2. Comparison of Optimizer and Learning Rates

Optimizer	**Learning Rate**	**Validation Accuracy (%)**	**Test Accuracy (%)**
SGD	0.001	79.43	78.32
SGD	0.01	87.30	85.68
SGD	0.1	89.4	88.76
AdaM	**0.001**	89.53	**89.11**
AdaM	0.01	88.32	87.5
AdaM	0.1	19.32	19.32
RMSProp	0.001	89.03	88.94
RMSProp	0.01	87.55	86.62
RMSProp	0.1	10.12	10.24

Clearly, the AdaM optimizer with learning rate 0.01 provides the highest **89.11%** test accuracy. This can be explained by the superior method of adaptive learning rate with momentum, which could make it easier propel past local minima, making it converge faster with more stability.



(A) SGD Optimizer with different learning rates

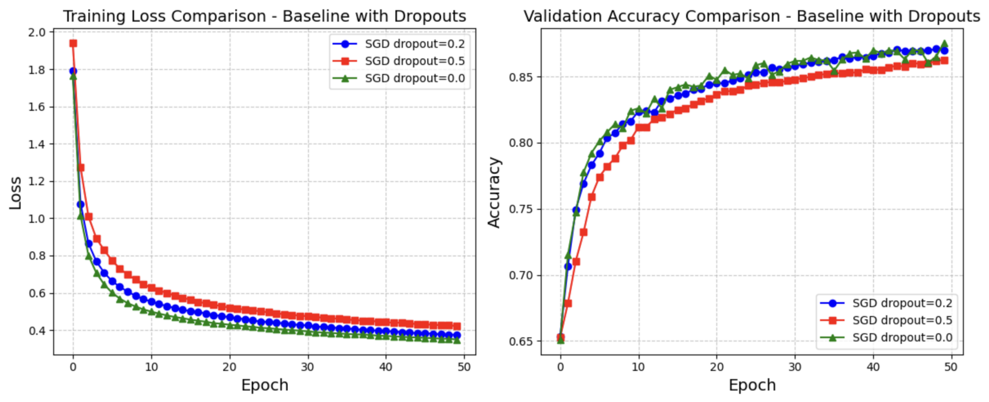


(B) Adam and RMSProp Optimizers with different learning rates

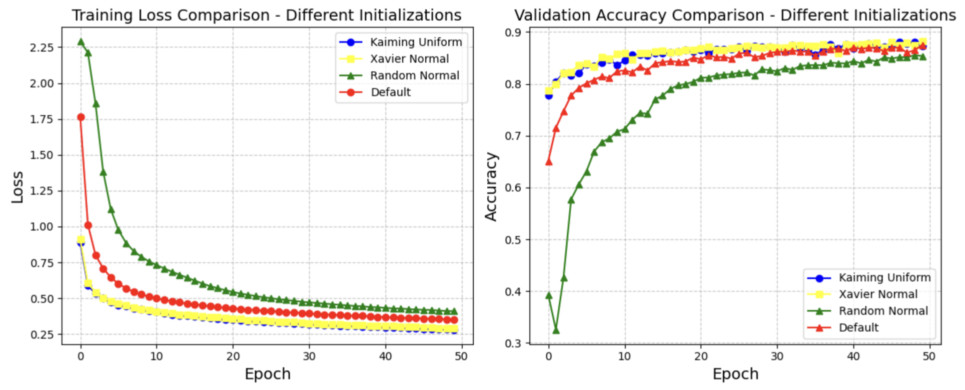
Figure 2. Training loss and Validation Accuracy vs Epoch for both optimizers.

Analyzing the **Overfitting** situation of our baseline model (showin in red in figure (A), we notice that as the training loss curve decreases, the validation accuracy increases over epochs. And as the training loss curve plateaus, a similar plateuing is noticed in the validation accuracy curve. Since the validation accuracy does not decrease as the training loss decreases, we can say that our model is not overfitting in the particular scenario. Also, since there is not a significant discrepancy between the test and validation accuracies, it gently suggests there isn't much overfitting.

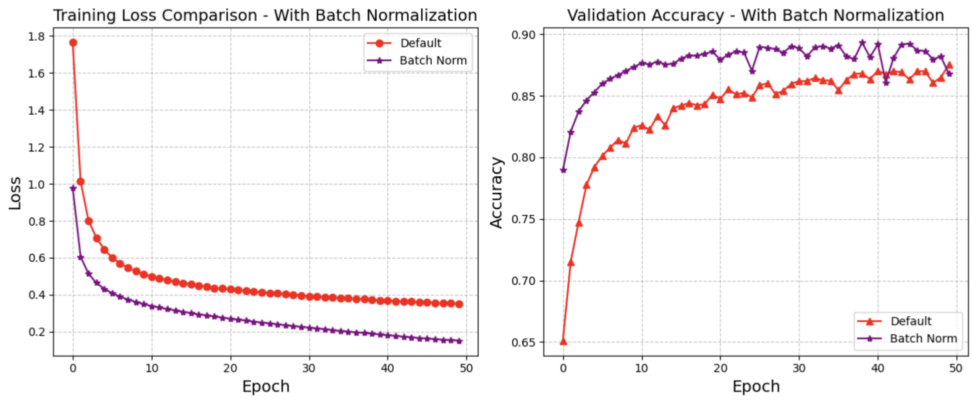
However, we will now include the dropout regularization and see how it affects our model's performance. The following are the training loss and validation accuracy curves:



(A) Baseline Model with different dropout regularization values



(B) Baseline Model with different Initializations



(C) Batch Normalization on Baseline Model

FIGURE 3. Baseline Model performance for varying dropout values, initializations, and batch normalization.

(A) Dropout Rates on Baseline Model

Dropout	**Val Acc (%)**	**Test Acc (%)**
0.0	**87.30**	**85.68**
0.2	86.98	85.64
0.5	86.25	84.95

(B) Initialization Methods

Initialization	**Val Acc (%)**	**Test Acc (%)**
Default	87.30	85.68
Kaiming Uniform	**88.33**	**87.28**
Xavier Normal	88.17	87.12
Random Normal	85.38	84.11

Table 3. Model Comparisons

As expected, including **dropout regularization** provides no additional benefit in our model, with the validation accuracy and test accuracy being extremely close to each other for dropout regularization values 0.0 and 0.2. We also notice that in our model, that upon increasing the dropout value by too much, it actually negatively impacts our validation accuracy, showing it is now underfitting by a small margin. The benefit of dropout might be more apparent in more complicated larger data sets, where there is a risk of overfitting to the training data.

Now, we consider performing hyperparameter tuning on different initializations to examine their impact on our model and how they differ from the baseline configuration. The **Kaiming Uniform** initialization demonstrates superior performance compared to all other methods, with Xavier Normal following as a close second. An additional advantage of Kaiming Uniform is its significantly faster convergence rate compared to default or random normal initializations, making our model more efficient. Kaiming Uniform's exceptional performance can be attributed to its optimization for ReLU activation functions, effectively mitigating the vanishing gradient problem. It also maintains variance during the forward pass, which enhances overall stability.

Finally, we perform the final step in our hyperparameter tuning: **Batch Normalization**. We find that batch normalization has had significant difference in our model's behavior. It makes the model converge much faster and the training process more stable. It also provides a validation accuracy: 86.80% and a **test accuracy: 86.31%**, which is better than the test accuracy of 85.68 for unnormalized baseline configuration.

5. Bonus

Using insights from our hyperparameter tuning process, we have come up with our final model configuration. We can explain this configuration by saying that most of our critical hyperparameters, including learning rate, optimizer, initializer etc. were determined through hyperparameter tuning. We had to do some additional tuning to reach the 90% accuracy, because of the peculiarities of our dataset and some randomness involved. This involved setting a dropout to 0.2 and not doing batch normalization.

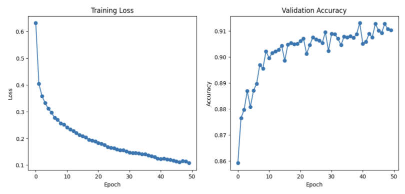
We show the loss curves and accuracy curves for the final configuration. The performance improvement is clearly noticable through higher values of the validation accuracies, as well as higher overall test accuracy of **90.19**, hence satisfying the BONUS component.

On the simpler **MNIST** data set, this model achieved a staggering **98.4%** test accuracy, outperforming the previous best KNN with 97.6% test accuracy.

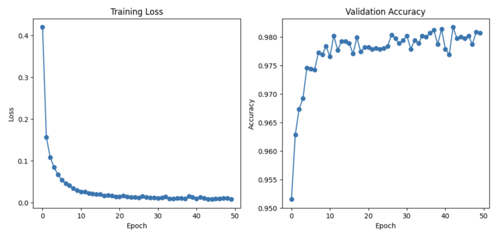
Moreover, applying the previous best method for the MNIST, which was the KNearestNeighbors classifier with 5 neighbours, we achieved an accuracy of **85.40%**, which is much lower than our 90.19 achieved using a neural network. This shows that neural networks are superior to the KNN classifier because they better interpret non-linear features in the dataset.

Table 4. Final Model Configuration post Hyperparameter tuning

Parameter	**Value**
Epochs	50
Loss function	Cross-Entropy
Initialization	Kaiming Uniform
Learning rate	0.001
Optimizer	AdaM
Dropout regularization	0.2
Batch normalization	No
Test accuracy	**90.19%**



(a) Training loss and Validation Accuracy vs Epoch for our (best) Final Model



(b) Training loss and Validation Accuracy vs Epoch on MNIST

Figure 4. Training loss and Validation Accuracy vs Epoch for both optimizers.

6. Summary and Conclusions

In conclusion, we effectively used a feed forward neural network, and performed hyperparameter tuning to succesfully distinguish images from the FashionMNIST data set, achieving a peak accuracy of 91.19%. We also researched the effect of different hyperparameters on the learning process and accuracy, and showed several informative visualizations to corroborate our findings.

References

[1] "Adam Optimizer." *GeeksforGeeks.* Accessed March 17, 2025. `https://www.geeksforgeeks.org/adam-optimizer/`.

[2] "Kaiming Initialization in Deep Learning." *GeeksforGeeks.* Accessed March 17, 2025. `https://www.geeksforgeeks.org/kaiming-initialization-in-deep-learning/`.

[3] "The Glorot Normal Initializer, Also Called Xavier Normal Initializer." *Keras3 Documentation, version 1.2.0.* Posit. Accessed March 17, 2025. `https://keras3.posit.co/reference/initializer_glorot_normal.html`.

[4] Instructor Notes and Helper Code.