COMPARATIVE ANALYSIS OF FCN AND CNN FOR IMAGE CLASSIFICATION: EFFECTS OF MODEL SIZE AND HYPERPARAMETER TUNING ON FASHIONMNIST

CHENAB

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

Abstract. In the following paper, we compare a Fully Connected Deep Neural Network (FCN /DNN) with different model architectures and weight constraints based on size with a Convolutional Neural Network (CNN)to classify images in the FashionMNIST data set. For both CNN and FNN, we perform hyperparameter tuning for the optimizer, learning rate, drop out regularization etc, and then evaluate the impact of model size on accuracy and training efficiency.

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 FCN has input dimension 784 (for the pixels) and output dimension 10 for the classification. We compare three FCN models: with 100k weights, 50k weights and 200k weights. We use ReLU as our activation function, and cross entropy loss to perform the classification. For our CNN, we compare four models with weights 100k, 50k, 20k and 10k. We make use of convolutional layers with max pooling, as well as FC layers. Moreover, we compare the effect of the number of weights on model accuracy and training efficiency.

2. Theoretical Background

A **Neural Network** is a brain-inspired computational model that identifies patterns and predicts from data. **A Fully Connected Neural Network(FCN)** connects each neuron to every neuron in the next layer. Data flows from input through hidden layers to output, with each unit computing a weighted sum 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.

A **Convolutional Neural Network(CNN)** is specialized for processing grid-like data by capturing patterns through convolutional operations, making it ideal for image classification. Its foundation lies in its **convolutional layer**. In the convolution operation, a kernel slides over the input data, creating a feature map. This operation excels at **Edge detection**, by detecting sharp changes in pixel intensity. Mathematically, it does elementwise multiplication between the input region I and the filter weights w as:

$$F[m,n] = I[m,n] * w[m,n] = \sum_{j=-\infty}^{\infty} \sum_{i=-\infty}^{\infty} I[i,j]w[m-i, n-j]$$

The following features are central to CNNs:

- **Kernel size:** The dimension (height vs width) of the kernel used for the convolutional operation
- **Stride:** It is the number of pixels, or the step size that the kernel moves through the input. Larger stride means smaller output size.
- **Padding:** Adds zeroes around the input borders, controlling the output size. Also solves the problem of edge pixels not being taken into account.

Date: March 25, 2025.

After each convolutional layer, we introduce activation fuctions like ReLU, to introduce non-linearity for complex pattern learning. Following which a **pooling** layer is used. It reduces the size of the feature map through dimensionality reduction and speeds up computation. The pooling we use is **Max Pooling**, which involves selecting the pixel with the maximum value as the filter moves through the input.

Our 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.

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

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

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:** Factors 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, and creating different sized models We define our FCN that has input dimension 784, output dimension 10. We design it to have an adjustable number of hidden layers. We define it to use the ReLU activation function and also enable funcionality for dropout regularization and batch normalization. Following hyperparameter tuning, we define 3 models having weights 50k, 100k and 200k respectively by adjusting the number of neurons in the three hidden layers.

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:

Algorithm 1 Model Training Pipeline

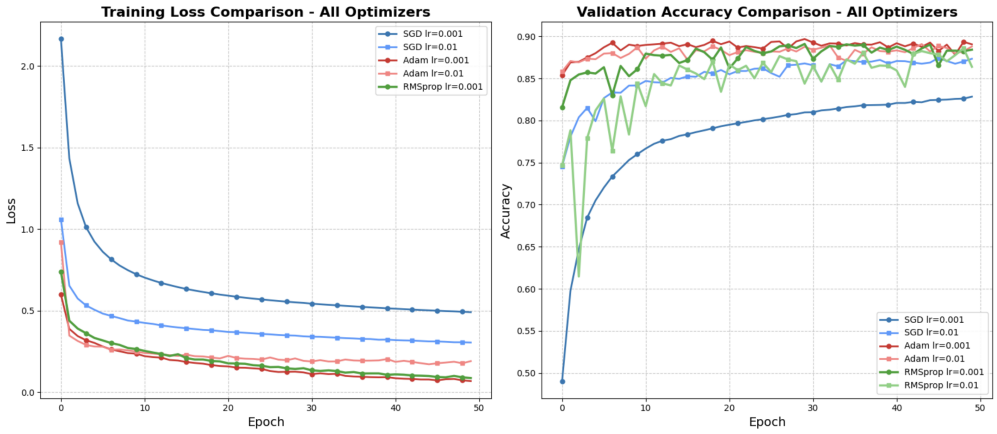
procedure TrainModel($model, epochs, lr, loss\_func, optimizer\_type$)
  for $epoch \leftarrow 1$ to $epochs$ **do**
    Set $model$ to training mode
    Initialize $epochLoss \leftarrow 0$
    for each batch in training data **do**
      Reshape input features from 2D (28×28) to 1D
      Zero previous gradients using PyTorch's `zero_grad`
      Compute predictions via a forward pass: $pred \leftarrow model(batch)$
      Compute loss: $loss \leftarrow loss\_func(pred, labels)$
      Backpropagate: perform `backward` on $loss$
      Update model parameters: call `step` on optimizer
      Accumulate loss: $epochLoss \leftarrow epochLoss + loss$
    end for
    Average $epochLoss$ over all training batches
    for each batch in validation data **do**
      Repeat forward pass and loss computation as above
    end for
    Compute validation accuracy
    Report training loss and validation accuracy for current $epoch$
  end for
  Set $model$ to evaluation mode
  Use PyTorch's `no_grad` to compute final test accuracy
end procedure

4.) Setting baseline then Hyper parameter tuning. We experiment with our hyperparameters: learning rate, optimizers drop out regularization and normalization to arrive at a baseline configuration that has 88% test accuracy for the 100k FCN, and compare it with 50k and 200k FCN

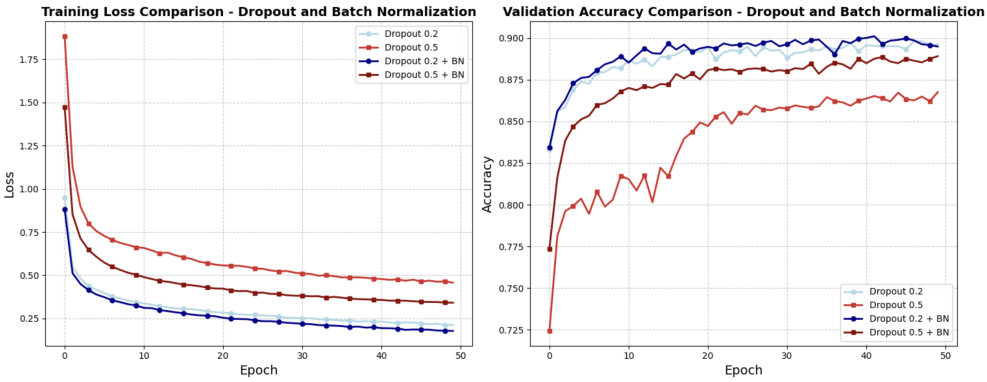
5. Repeat the process for CNN: For CNN, we define our model, on the same principles of FCN, with adjustable number of convolutional layers and FC laayers. We use ReLU activations after each convolutional step and then performing hyperparameter tuning on the 100k CNN model, then making three additional models with weights: 10k, 20k, 50k, and compare accuracy, as well as loss and validation curves

4. Computational Results

The following section will provide details of our models as well as comparisons between them. For FCN, we define a 100k model with **three hidden layers**, possessing [100, 100, 97] neurons respectively. The exact number of weights for this model are **99971**. We performed hyperparameter tuning by first comparing different learning rates and optimizer combinations, then adjusting dropout and batch normalization. For our tuning, we kept the initialization of **Kaiming Uniform** unchanged because we are using ReLU in our FCN and CNN, and kaiming uniform is specifically optimized to work best with ReLU activation fucntion. Moreover it also addresses the vanishing gradient problem. In the table can be seen the results of our hyperparameter tuning. And in the plots, the impact of our hyperparameters.



(A) Optimizer with different learning rates FCN 100k



(B) Dropout and Batch Normalization in FCN

FIGURE 1. Training loss and Validation Accuracy vs Epoch for both optimizers.

Clearly, the best configuration for 100k, turns out to be the one with Adam optimizer, 0.001 learning rate, dropout 0.2 with batch normalization, because of the superior validation accuracy (89.28%) and highest testing accuracy 88.7%, more stability and faster convergence as seen in the plots. However, a lot of instability can be seen in RMSProp, and the SGD simply isn't that accurate. Detailed performances are listed below.

TABLE 1. Performance Comparison of FCN Models with Various Hyperparameters

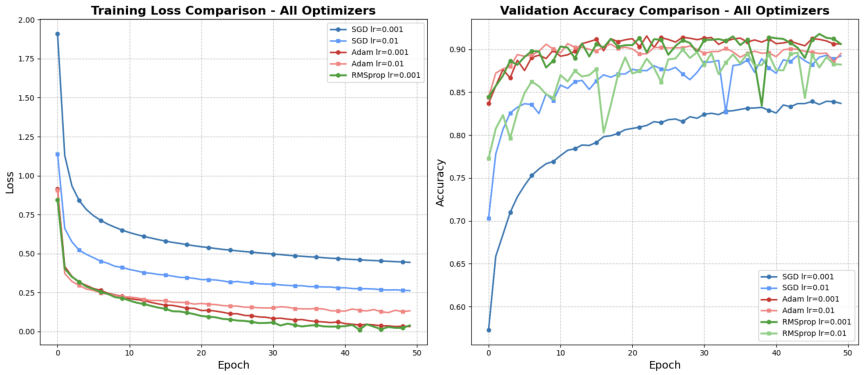
Type	Weights	Optimizer	LR	Dropout	BN	Val Acc (%)	Test Acc (%)	Time (s)
FCN	100k	SGD	0.001	No	No	82.83	81.72	N/A
FCN	100k	SGD	0.01	No	No	87.33	86.38	N/A
FCN	100k	SGD	0.1	No	No	87.40	86.78	N/A
FCN	100k	Adam	0.001	No	No	89.05	88.30	N/A
FCN	100k	Adam	0.01	No	No	88.28	87.07	N/A
FCN	100k	Adam	0.1	No	No	10.10	10.10	N/A
FCN	100k	RMSProp	0.001	No	No	86.95	86.23	N/A
FCN	100k	RMSProp	0.01	No	No	85.92	85.08	N/A
FCN	100k	RMSProp	0.1	No	No	10.20	10.20	N/A
FCN	100k	Adam	0.001	0.2	No	89.60	88.30	N/A
FCN	100k	Adam	0.001	0.2	Yes	89.48	88.70	203.15
FCN	100k	Adam	0.001	0.5	No	86.75	85.55	N/A
FCN	100k	Adam	0.001	0.5	Yes	88.90	88.09	N/A
FCN	50k	Adam	0.001	Yes	Yes	88.90	87.54	198.32
FCN	50k	Adam	0.01	Yes	Yes	88.42	87.72	196.21
FCN	**200k**	**Adam**	**0.001**	**Yes**	**Yes**	**89.75**	**89.39**	**209.62**
FCN	50k	Adam	0.1	Yes	Yes	88.48	87.48	209.11

TABLE 2. Performance Comparison of CNN Models with Various Hyperparameters

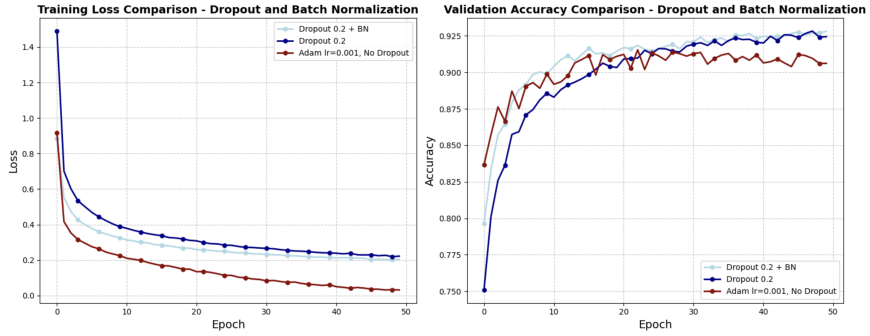
Type	Weights	Optimizer	LR	Dropout(0.2)	BN	Val Acc (%)	Test Acc (%)	Time (s)
CNN	100k	SGD	0.001	No	No	83.68	83.05	N/A
CNN	100k	SGD	0.01	No	No	89.22	88.18	N/A
CNN	100k	Adam	0.001	No	No	90.62	89.95	N/A
CNN	100k	Adam	0.01	No	No	89.50	88.94	N/A
CNN	100k	RMSProp	0.001	No	No	90.60	89.66	N/A
CNN	100k	RMSProp	0.01	No	No	88.22	87.43	N/A
CNN	**100k**	**Adam**	**0.001**	**Yes**	**Yes**	**92.82**	**92.20**	**716.84**
CNN	100k	Adam	0.001	Yes	No	92.45	91.90	N/A
CNN	50k	Adam	0.001	Yes	Yes	91.02	90.17	689.75
CNN	20k	Adam	0.001	Yes	Yes	87.52	86.78	507.21
CNN	10k	Adam	0.001	Yes	Yes	82.65	81.93	410.31

We gauge the effect of number of weights on our neural network model. We design a 50k weights neural network that also has three hidden layers but with a [50, 95, 50] configuration. The exact number of weights are **49795**. We also double the weights in our 200k model with a [200, 90, 200] configuration having **196280** weights. We train these models to have similar hyperparameters, in terms of optimizers, regularization normalization etc. The 200k FCN variant provides the best testing accuracy. This can be attributed to its greater capacity to learn complex patterns in the data set. Even though it is the most accurate, it comes at a cost because of the extra training time involved.

Next we design a **Convolutional Neural Network**. We design it initially to have 100k weights. It has 3 convolutional layers with [16,32,64] and a fully connected layer having [128] neurons.We similarly experiment with different optimizers and learning rates. However, this time we have to carefully pick a subset of the possible combinations based on our best guess, since the CNN takes 3-4x more time than our FCN to train. Below can be seen the plots:

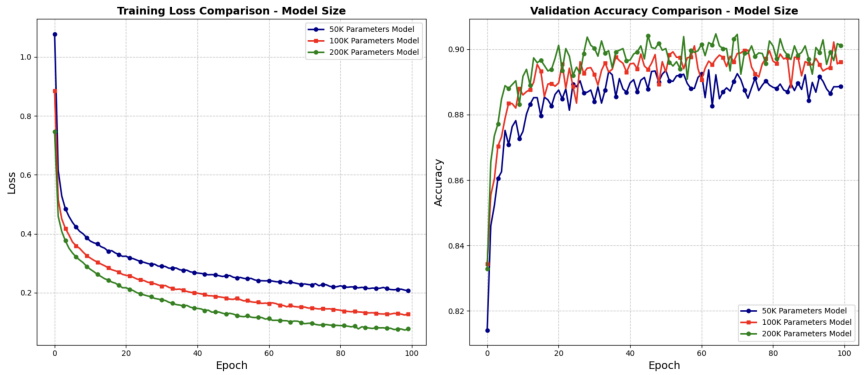


(A) Optimizer with different learning rates CNN 100k

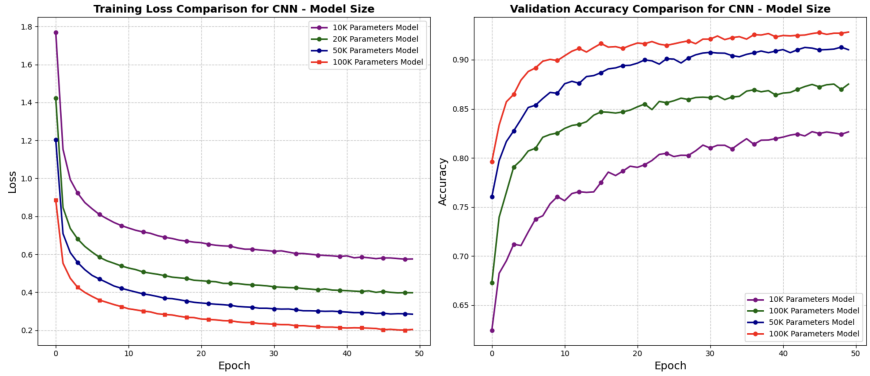


(B) Dropout and Batch Normalization in CNN

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



(A) FCN Model Size comparison(based on number of weights)



(B) CNN Model Size comparison(based on number of weights)

FIGURE 3. CNN Model Size comparison)

Clearly, we can see that our CNN without dropout is overfitting as can be seen by the dip in validation accuracy as training loss continues to decrease. This is fixed by adding dropout of 0.2 and batch normalization. A dropout of 0.5 was not added because CNNs do not need that high regularization. Our models now do not overfit the data, as can be seen from higher values of validation accuracies as training loss decreases. Now comparing 50k, 20k and 10k CNN models, we notice a clear trend: larger models converge faster and are more accurate, where as smaller models tend to underfit. However a surprising result is that CNN 50k, with half the weights as CNN 100K, possesses really similar validation accuracies (91.02 verses 92.82). Moreover, it is also much more efficient, finishing training in 689.75 seconds as opposed to 716.84 for the 100k CNN model. In terms of training time, there is a monotonic trend with number of weights and training time. This holds true for both CNNs and FCNs. Detailed training time comparison is given in the tables below.

Our Best CNN models are considerably better than our FCN models, showing their superiority in handling grid-like data and image classification. Their convolutional and pooling layers enable them to efficiently extract hierarchical patterns from the images. By far the best model accross CNN and FCNs, is the CNN 100K model, achieving a staggering **92.20** test accuracy. Detailed configurations can be seen in the table.

5. Summary and Conclusions

In conclusion, we effectively used a feed forward neural network and a convolutional neural network based architecture, performed hyperparameter tuning, and explored thheir differences in training performance and accuracy on FashionMNIST. Moreover, within each neural network type, we created models of several sizes and succesfully demonstrated their effect on accuracy, training time, overfitting situation etc to make the choices of our model more informative and guided by quantitative reasoning.

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] "What are Convolutional Neural Networks?" IBM Think. Accessed March 20, 2025. `https://www.ibm.com/think/topics/convolutional-neural-networks`.

[5] Instructor Notes and Helper Code.