

**Brody Anderson**  
**Math 5750/6880: Mathematics of Data Science**  
**Project #3 Final Report**  
**November 6, 2025**

My GitHub Project3 repository is located here:

<https://github.com/brodyanderson/math-6880-Project3>

### 1. Fashion-MNIST Image Classification Using sklearn

In order to properly experiment with each of the model training methods, I decided to focus on each one individually. I first created a very simple model, using all of the default parameterizations from sklearn's MLPClassifier. I used this model as a baseline to track changes in each of the training methods for the assignment. After experimenting with several parameterizations for these training methods, I incorporated some of the best performing individual methods into five models to see how they all worked together.

The table below displays different hidden layer and neuron combinations and their resulting accuracy on the test data, as well as the model's training time and the number of iterations to converge.

Table 1. Hidden Layers & Neurons

Structure	Test Accuracy (%)	Training Time (s)	Iterations
(100)	85.50	29.17	90
(200)	85.88	46.47	87
(100, 50)	85.47	26.24	71
(200, 100)	<b>86.40</b>	<b>37.89</b>	54
(200, 100, 50)	85.96	36.09	49
(512)	85.84	64.56	49
(512, 256)	84.58	76.45	42
(256, 128, 64)	<b>86.46</b>	<b>55.61</b>	52
(512, 256, 128)	83.84	105.46	48
(512, 256, 128, 64)	85.44	102.63	42

For every architecture with more than one hidden layer, I used a tapering structure. I started with generic numbers 50, 100 and 200, but later read about how powers of two for the number of neurons could prove to run faster (especially for GPUs). In response, I added more hidden layer and neuron combinations to test the training times. The most accurate networks had structures of (200, 100) and (256, 128, 64). The largest network with four hidden layers and the most neurons ran the slowest (no surprise there), and the single hidden layer network with 100 neurons trained the fastest. In general, the fewer neurons, the faster the model trained with no huge discrepancy in test accuracy. As a result, in the next section, I used the simpler structures to experiment with activation functions.

Four activation functions were compared with three different combinations of hidden layer and neuron structures. The table below displays the accuracy and training results.

Table 2. Activation Functions

Activation	Structure	Test Accuracy (%)	Training Time (s)	Iterations
relu	(200)	85.88	55.45	87
	(200, 100)	<b>86.40</b>	47.18	54
	(200, 100, 50)	85.96	49.07	49
logistic	(200)	85.73	92.66	134
	(200, 100)	85.29	68.08	89
	(200, 100, 50)	85.34	65.62	83
tanh	(200)	85.38	65.35	87
	(200, 100)	85.53	47.93	54
	(200, 100, 50)	85.95	53.45	47
identity	(200)	79.79	53.32	93
	(200, 100)	80.42	41.70	64
	(200, 100, 50)	80.31	75.84	87

On average, the most accurate activation function was relu. Relu also recorded the highest training accuracy of the group with 86.40% coupled with the (200, 100) hidden layer and neuron structure. Convergence and training speed were impacted more by the structure of the network than the activation function, but the logistic group recorded the longest training time and highest number of iterations.

Optimization methods and learning rates were all run with a structure of (200, 100). Adam and stochastic gradient descent were compared with several momentum values and differing learning rates.

Table 3. Optimization Methods &amp; Learning Rates

Optimizer	Momentum	Learning Rate	Test Acc (%)	Train Time (s)	Iterations
adam	-	0.001	<b>86.40</b>	35.03	54
	-	0.01	83.67	26.92	41
	-	0.1	10.00	9.15	13
sgd	0	0.001	83.05	121.41	200
	0	0.01	85.37	119.69	200
	0	0.1	85.60	56.29	96
sgd	0.5	0.001	84.53	120.01	200
	0.5	0.01	85.41	120.88	200
	0.5	0.1	85.87	37.67	62
sgd	0.9	0.001	85.37	119.60	200
	0.9	0.01	85.31	52.67	89
	0.9	0.1	<b>85.98</b>	39.32	67
sgd	0.99	0.001	85.48	61.57	102
	0.99	0.01	84.61	38.50	64
	0.99	0.1	10.00	6.10	12

Adam logged the single best accuracy but was heavily impacted by the learning rate. Sgd with a momentum value of 0.9 had the highest average accuracy overall and was very consistent across

learning rates. However, the biggest takeaway was not the optimization method or choice of momentum, but the drastic change in output from learning rate. A learning rate of 0.001 almost always resulted in longer training times compared to 0.01 and 0.1. The 0.1 learning rate trained significantly faster and with much fewer iterations but consequently decreased the accuracy overall.

Early stopping and the number of iterations without change were also explored.

Table 4. Early Stopping Settings

<b>Early Stopping</b>	<b>n_iter_no_change</b>	<b>Test Accuracy (%)</b>	<b>Training Time (s)</b>	<b>Iterations</b>
False	-	<b>86.40</b>	36.27	54
True	5	85.69	8.11	16
True	10	85.69	12.83	21
True	20	85.69	19.90	31

The best model did not use early stopping and recorded an accuracy of 86.40%. This was quite a bit better than any model that used early stopping; however, the training times and number of iterations when early stopping was employed were drastically lower. When setting early stopping to true and n\_iter\_no\_change=5, the training time was only 8.11 seconds, with 16 iterations and still resulted in 85.69% accuracy. If time or money were a serious factor in this setting, then the savings amassed by utilizing early stopping, with a relatively small trade-off in accuracy, would be substantial.

After experimenting with different training methods separately, several combinations were brought together in an attempt to achieve higher accuracy with speedy convergence.

Table 5. Final Models Using sklearn

<b>Structure</b>	<b>Activation</b>	<b>Solver</b>	<b>Learning Rate</b>	<b>Early Stopping</b>	<b>Test Acc (%)</b>	<b>Train Time (s)</b>	<b>Iterations</b>
(200, 100)	relu	adam	0.001	False	<b>88.75</b>	<b>256.93</b>	60
(256, 128, 64)	relu	adam	0.01	True (10)	<b>88.40</b>	<b>128.40</b>	26
(200, 100, 50)	logistic	adam	0.001	True (5)	87.97	85.89	20
(200, 100)	tanh	sgd	0.01	True (10)	87.82	323.42	66
(256, 128, 64)	tanh	sgd	0.01	False	88.01	357.87	54

The two best models shared relu as the activation function and adam as the optimization method. Both were fairly close to one another in terms of accuracy on the test set, but the learning rate for the second model coupled with early stopping led to a much faster training time, with significantly fewer iterations. The structure of the hidden layers and neurons had much less impact when compared to the learning rate in this instance.

The confusion matrices for the top two models are presented below. The two confusion matrices are very similar, with both models misclassifying shirts and coats most often.

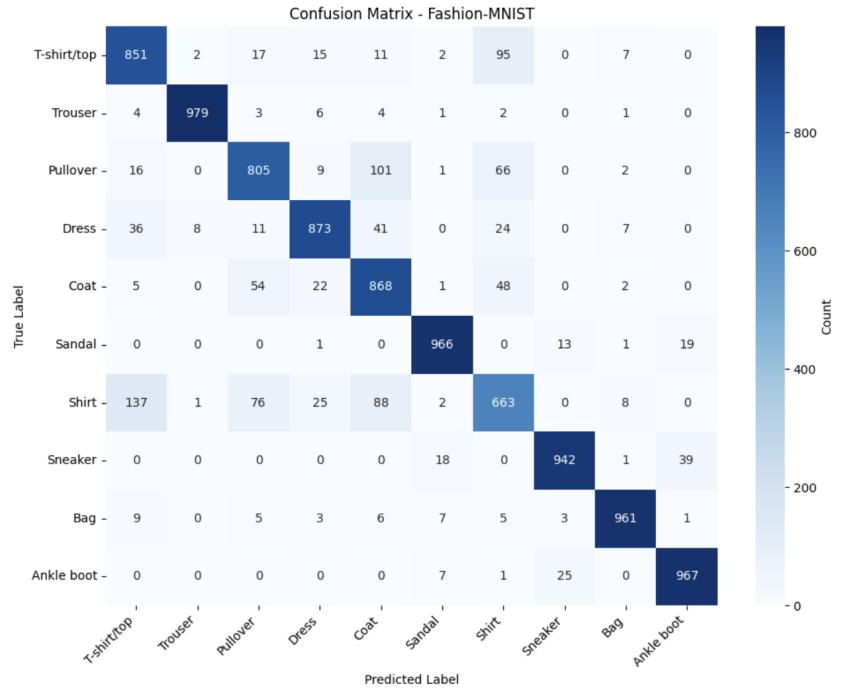


FIGURE 1. Model 1: (200, 100) Confusion Matrix

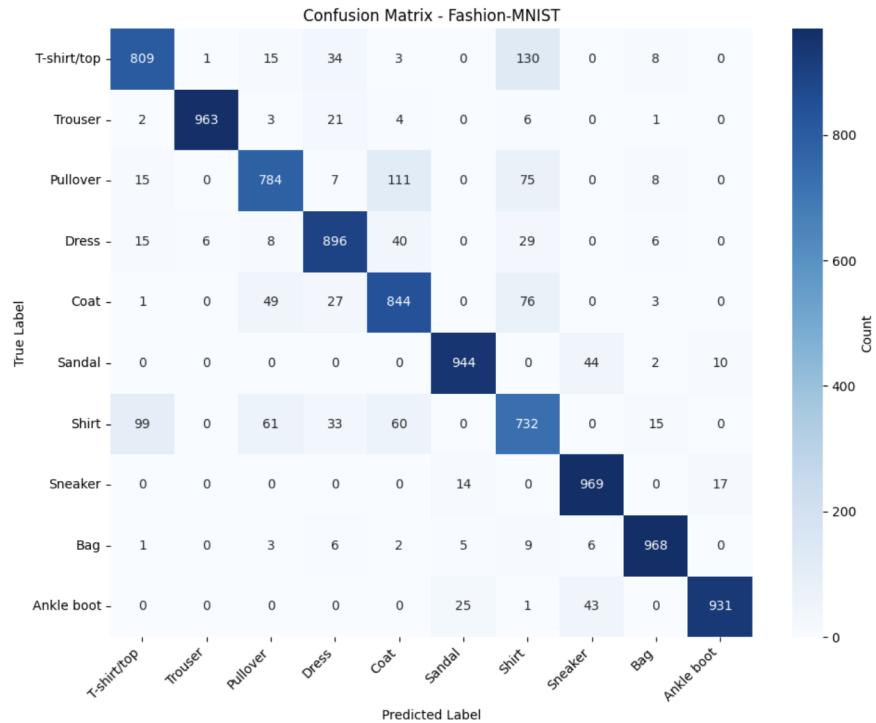


FIGURE 2. Model 2: (256, 128, 64) Confusion Matrix

## 2. PyTorch

PyTorch has so many more options than sklearn, but the code is considerably harder.

On a more serious note, I had never used PyTorch before and did not know anything about it. I learned that tensors are essentially multi-dimensional arrays that work similarly to NumPy arrays, but they can utilize GPUs to accelerate computation (which is a game changer). Instead of creating only simple, fully connected neural networks, PyTorch allows users to create deep networks that incorporate convolution and pooling layers to increase accuracy in a more efficient manner. There are an unlimited number of architectures that could be created, giving anyone complete control over how complicated or simple a neural network could be.

More practically, I learned how to set up a neural network and how to connect each layer to the next one. You have to make sure that the output size from one layer matches the input size of the following layer, especially when transitioning from the convolution phase into the fully connected phase. I also learned about different pooling methods (max, avg, global, etc.) and how they capture the most important information while decreasing the computational load and cutting down on the total number of parameters.

Simply put, I learned how to make a more accurate neural network for image classification.

## 3. Fashion-MNIST image classification using PyTorch

Previously, while using sklearn, (200, 100) was used as the default structure, relu was the activation function and adam was the solver if not specified otherwise. For PyTorch, I used stochastic gradient descent with a learning rate of 0.001 and momentum set to 0.9. The default structure was (512, 256) and cross entropy was the loss function. Furthermore, because PyTorch allows for GPUs to be used, I tried more combinations of structures, functions, optimizers, etc. and experimented with options not previously available in sklearn. With so much information that could be displayed, I have opted to simply write about the most accurate models with speedy convergence rather than provide even larger tables than those seen above.

Similar to the first section of this paper, the number of hidden layers and neurons were experimented with first. I tried more combinations of one, two and three layer structures with varying numbers of neurons. Almost all of the combinations were powers of two, as this should reduce computation lengths when using GPUs. The best model had a structure of (512, 256) and returned an accuracy of 83.87% on the validation data. As mentioned previously, the number of neurons had the greatest effect on the speed of training. The training times ranged from 15.01 seconds (128,) all the way up to 110.77 seconds (1024, 512, 256).

For activation functions, relu, sigmoid, tanh, leaky\_relu, elu and identity were all trained with the following structures: (128,), (512, 256), (784, 392, 196). Relu with a structure of (512, 256) reported the highest validation accuracy of 83.93%, but most of the other activation functions were not far off (sigmoid was notably the worst). The activation functions each recorded similar convergence speeds with the only real variation coming from the structures rather than the functions themselves.

For optimization methods, I looked at stochastic gradient descent with several momentum values and a wider range of learning rates. I also added 'adamw' which is adam with weight decay and should be an improved alternative to the standard adam solver. In the end, adamw with a learning

rate of 0.001 returned the best validation accuracy, but both adam optimizers were significantly slower than sgd. Learning rates of 0.0001 and 0.1 performed terribly when compared to more moderate values.

PyTorch has a tolerance option when configuring early stopping settings. I looked at tolerance values of 0, 0.0001 and 0.001 with n\_iter\_no\_change values of 5, 10 and 20. The more tolerant parameterizations with higher n\_iter\_no\_change values trained more slowly than less tolerant models. The difference in validation accuracies between all of the models was negligible.

I also experimented with alternative loss functions, but that was not as interesting as I thought it would be. They were fairly equivalent in terms of speed and accuracy with cross entropy edging out the other options.

Penultimately, I used convolution neural network layers and MaxPool2d layers to improve convergence speed and accuracy. This was probably the most interesting part of the project. I trained four models each with a different number of layers. First, was a fully connected network with no convolution layers. The next model had one convolution layer, the third model had two layers and the last model had three layers. I used MaxPool2d(2,2) for each pooling layer and used the format of each convolution layer being followed by a pooling layer in every model. Each fully connected portion was the same and used the (512, 256) structure as experimented with previously. Using the information gained from before, I also used adamw as the optimizer with the learning rate set to 0.001 and relu as the activation function. The output for each of the models is presented below.

Table 6. CNN Models on Validation Set Using PyTorch

Model	Validation Accuracy (%)	Training Time (s)	Total Parameters
Dense NN	89.35	50.51	535,818
1 Layer CNN	91.71	537.44	2,903,626
2 Layer CNN	<b>91.76</b>	470.54	972,426
3 Layer CNN	89.58	453.75	292,618

The two layer CNN model was the most accurate, by a slim margin over the one layer CNN model; however, the training time was quite a bit less in comparison. The one layer CNN model had the most parameters, which makes sense because it had fewer pooling layers to cut dimensions. Following this same logic, the three layer CNN had the fewest total parameters at 292,618. Even though the fully connected neural network had the worst validation accuracy (not by much), it recorded the fastest training time.

With pretty good accuracy on the validation set, I used the one layer and two layer models on the test data. This was the only time that the test data was touched. The resulting table and confusion matrices are presented below.

Table 7. Final Models Using PyTorch

Model	Test Accuracy (%)	Training Time (s)	Total Parameters
1 Layer CNN	<b>91.44</b>	573.95	2,903,626
2 Layer CNN	90.66	569.79	972,426

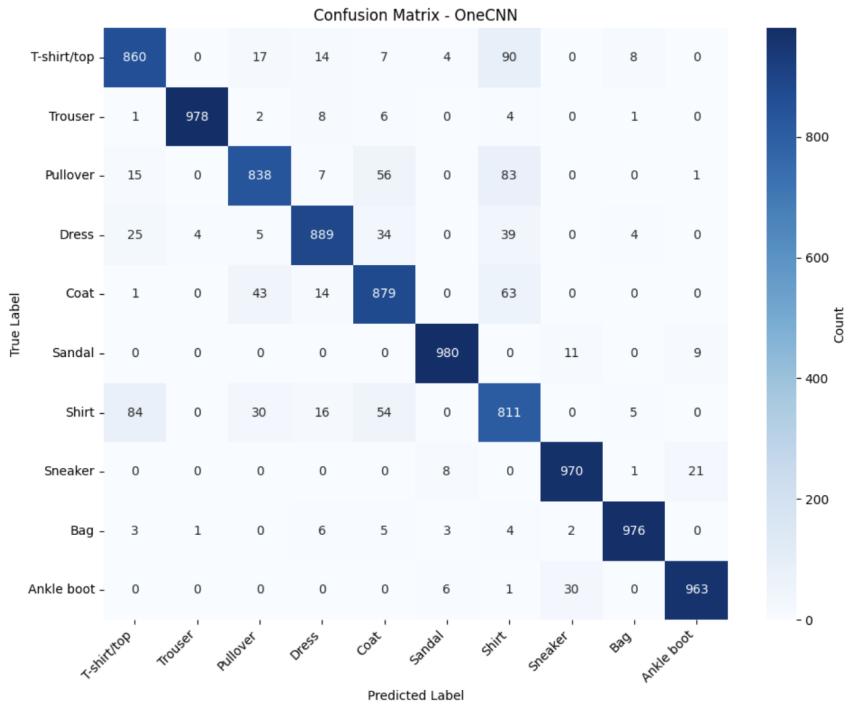


FIGURE 3. One Layer CNN Confusion Matrix

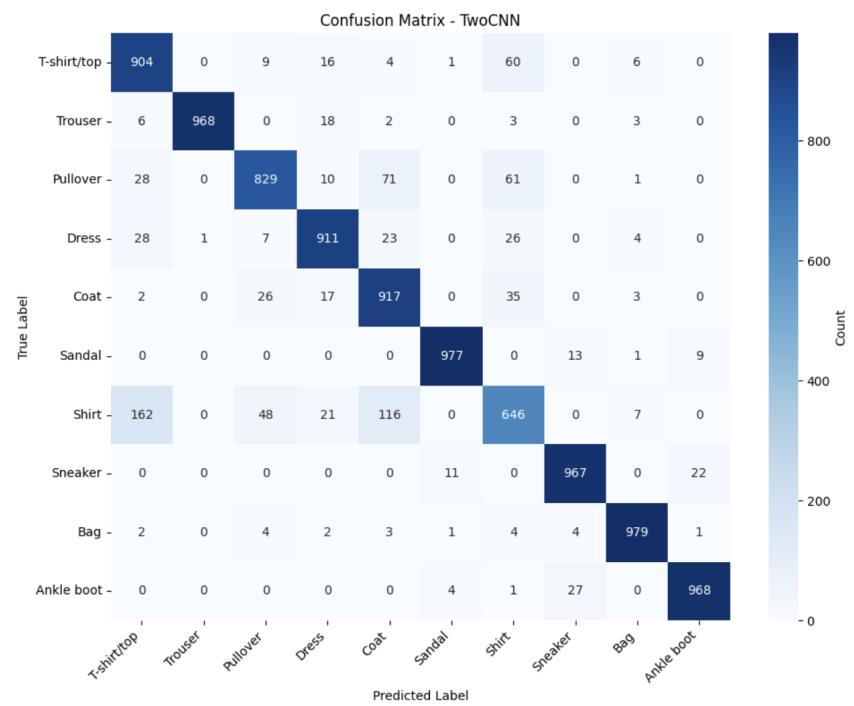


FIGURE 4. Two Layer CNN Confusion Matrix

The one layer CNN model actually performed the best with an accuracy of 91.44% and trained faster than the two layer network. Looking at the confusion matrices, both networks had the hardest time classifying shirts.

While the proposition of fine tuning a pretrained model seemed fun, I was ultimately not up for the challenge.