Two Methods for Classifying the MNIST Fashion Data Set

Leonard Armstrong  
IST-718, Lab 09  
June 4, 2019

# Introduction

The name MNIST (Modified National Institute of Standards and Technology database) is often associated with a set of 70,000 labeled, gray-scale, 28 × 28 pixel, handwritten Arabic numerals used as a standard for comparing the performance of image classification applications. However, according to Zalando Research:

* MNIST is too easy. Convolutional nets can achieve 99.7% on MNIST. Classic machine learning algorithms can also achieve 97% easily. …. Most pairs of MNIST digits can be distinguished pretty well by just one pixel.
* MNIST is overused.
* MNIST cannot represent modern CV tasks.[[1]](#footnote-1)

To combat these issues, researchers at Zalando created the Fashion-MNIST data set. Designed as a drop-in replacement for the original MNIST digit data set, MNIST Fashion consists of 70,000 labeled, gray-scale, 28 × 28 pixel, images of clothing items. Since there are 10 distinct digit classes in MNIST, Fashion-MNIST contains 10 distinct classes of clothing articles.

In addition to serving as a next-generation replacement for the MNIST-digit data set, the classification of fashion items through machine learning has broader applications to problems such as the inventory management of clothing items in the retail sector. And adjusted versions of this problem could have military implications such as weapons identification.

Focusing on the retail question, a study was performed and is reported here. This study applies two machine learning models to the Fashion-MNIST classification problem. The goal of the study is to determine if clothing items can be automatically classified into one of a series of predefine fashion item labels. If a high percentage of clothing items can be successfully categorized, the impact on retailer inventory and product stocking could result in reduced costs by replacing lengthy manual labor with an automated process, giving the retailers the ability to apply labor to other, more pressing needs.

The first machine learning model applied to the data set is a multilayered perceptron-based neural network. The second model is a convolutional neural network.

# About the Data

The data set used for the analyses is the Fashion-MNIST data set. Fashion-MNIST consists of 70,000 images, the first 60,000 of which are intended to be a training data set and the last 10,000 are intended as a test set. Each image is labeled by one of ten distinct clothing classes: 0 for T-shirt/top; 1 for Trouser; 2 for Pullover; 3 for Dress; 4 for Coat; 5 for Sandal; 6 for Shirt; 7 for Sneaker; 8 for Bag; and finally, 9 for Ankle boot.

## Image Label Distributions

As shown in Figure 1, the data set has a uniform distribution of images with respect to the ten clothing labels.

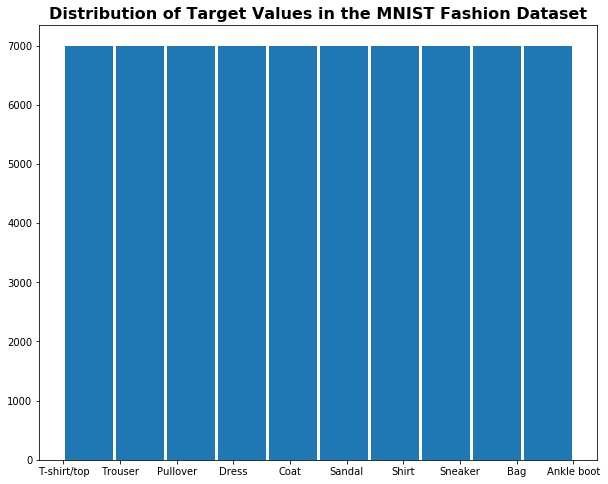


Figure - Distribution of Labels in Fashion-MNIST

Although not shown[[2]](#footnote-2), when viewed separately the training and test sets are also uniformly distributed.

## Sample Images and Pixel Value Distrubutions

Figure 2 shows a 6 × 6 grid of randomly selected images from Fashion-MNIST. The grid contains a variety of patterns that can easily seen by the naked eye. Some images are sparse. (E.g., the sandal in the bottom row). Some images are light. (E.g., the white shirt in the fourth row from the top.) Some images are full. That is, the image takes up a large percent of the number of pixels in the 28 × 28-pixel grid. (E.g., the shirt in the fourth column from the left, third row from the top.) Finally, there are images that are very dark to the naked eye. (E.g., the pullover in the second column from the left, second row from the top.)



Figure - 36 Random Fashion-MNIST Images

Figure 3 through Figure 6 show more detail about the four images from Figure 2 called out in the previous paragraph. Each image is shown alongside a distribution boxplot of the image’s 784 pixels. First, Figure 3 shows the pixel value distribution for an image that is sparsely populated, namely the sandal in the bottom row of Figure 2. Here we see that no less than ¾ of the 784 pixels are 0. This means that all non-zero pixels in the image are considered outliers.[[3]](#footnote-3) In fact, of all the clothing article images shown in Figure 2, the sandal shown in Figure 3 is the closest in pixel value distribution to a typical MNIST (digit) image.[[4]](#footnote-4)

|  |  |
| --- | --- |
| /var/folders/d5/7361vmxn78z85s16qplt88j40000gn/T/com.microsoft.Word/Content.MSO/8B6CA90E.tmp  Figure 3- Pixel Distribution of a Sparse Image | /var/folders/d5/7361vmxn78z85s16qplt88j40000gn/T/com.microsoft.Word/Content.MSO/54A6084C.tmp  Figure 4- Pixel Distribution of a Light Image |
| /var/folders/d5/7361vmxn78z85s16qplt88j40000gn/T/com.microsoft.Word/Content.MSO/3150303A.tmp  Figure - Pixel Distribution of a Full Image | /var/folders/d5/7361vmxn78z85s16qplt88j40000gn/T/com.microsoft.Word/Content.MSO/EA9EB355.tmp  Figure - Pixel Distribution of a Dark Image |

Figure 4 shows the light image, a white shirt, with a pixel distribution whose non-outliers all fall below the 100 value. At least ¼ of the 784 pixels are 0, as is evidenced by the fact that the first quartile is not distinctly shown. The outliers can be attributed to the small dark region in the neck area.

Moving next to the full image, the one that takes up a large volume of pixels, Figure 5 shows a shirt in medium-gray tones that has a wide range across its chest and belly regions made even wider by the wide arms extending from the sides. The pixel distribution for this image has four distinct quartiles and no outliers. The first and third quartiles are dense while the second and fourth quartiles have a wider spread of values. The densely populated third quartile – hovering near the midpoint of the 256 possible pixel values – is most likely due to the fact that the shirt is medium-gray in color. Comparing the boxplots of Figure 4 and Figure 5 shows that the distribution of pixel values is, in itself, an insufficient method to determine the image label.

Finally, Figure 6 shows a darker image. Here we see a boxplot with only three distinct quartiles. The lack of a distinct first quartile can be attributed to the narrowness and short bottom of the pullover allowing for a higher volume of 0-valued pixels in the image. The main difference between this plot and the other three is that the top two quartiles of this image are dense and have values that range between well over 200 and up to 255.

Finally, Figure 6 shows a darker image. Here we see a boxplot with only three distinct quartiles. The lack of a distinct first quartile can be attributed to the narrowness and short bottom of the pullover allowing for a higher volume of 0-valued pixels in the image. The main difference between this plot and the other three is that the top two quartiles of this image are dense and have values that range between well over 200 and up to 255.

## Data Cleansing and Serializing

Like its predecessor MNIST (image), Fashion-MNIST is a fairly reliable data set requiring little data cleansing. To ensure that all data was read as expected, tests were made to ensure that all values of the image and label data were present (not NULL nor NA) and were valid numerics. After verification passed on the images and labels, one small data type conversion was required. The labels are read, by default, as character data. Since the label values are the characters ‘0’ through ‘9’ the type of the label data was changed from character to integer. This simplifies the model validation steps allowing for direct comparisons to the values returned by the prediction methods of the model frameworks that were used.

After reading the Fashion-MNIST data in its raw form the data is immediately saved to local disk using python’s pickle serialization library. All future reads use the pickle format file. Use of the pickle format over the OpenML data reduces the load time of Fashion-MNIST from 42 seconds down to 1 second.[[5]](#footnote-5)

# Analyses

Two separate models were used and compared on a single analytic goal: to predict the Fashion-MNIST test set with the highest possible accuracy. Both models were neural network-based. The first model used a multi-layer perceptron neural network (MLPNN) with adaptive learning rate. The second model is a convolutional neural network (CNN).

## Model 1 – Multi-layer Perceptron Neural Network

The first model used to predict Fashion-MNIST images was a multi-layer perceptron neural network with an adaptive learning rate. The MLPNN was not based on any popular framework but instead was implemented from the ground up.[[6]](#footnote-6) The framework was natively CPU-based and open source. The open source feature allows for alteration to the code’s internals. In this study the code was altered to collect information about the pattern of the adaptive learning rate.

The MLPNN framework allowed for the 10 hyperparameters shown in the parameter dictionary in Figure 7 to be specified. The additional two parameters, shown above the thick double line, are data description parameters, not tuning hyperparameters.

|  |  |  |
| --- | --- | --- |
| Parameter | Default | Description |
| **n\_output** | - | Number of possible output classes (labels). |
| **n\_features** | - | Number of variables for each observation. In the case of Fashion-MNIST images each image pixel is considered a variable/feature. |
| **n\_hidden** | 30 | Number of hidden nodes in the layer between the input layer and the output layer. |
| **l1** | 0.0 | Lambda value for L1 regularization.[[7]](#footnote-7) |
| **l2** | 0.0 | Lambda value for L2 regularization. |
| **epochs** | 500 | Number of learning iterations across the MLPNN. |
| **eta** | 0.001 | Initial learning rate. Since the learning rate is adaptive, eta will change across each epoch. |
| **alpha** | 0.0 | Momentum constant. Factor multiplied with the gradient of the previous epoch t-1 to improve learning speed.[[8]](#footnote-8) |
| **decrease\_const** | 0.0 | Value used to adapt the learning rate. |
| **shuffle** | True | Whether or not to shuffle the training data every epoch. |
| **minibatches** | 1 | Number of subsets into which the training data will be divided each epoch. |
| **random\_state** | None | Provides the ability to specify your own random seed, if desired. |

Figure - Parameter Dictionary for the MLPNN

### Model 1 Results

During the course of the MLPNN analysis five of the hyperparameters (n\_hidden, l1, l2, epochs, minibatches) were adjusted to find an optimal result – to maximize the accuracy rate of the model on the test data set. Figure 8 shows the parameter combinations used for thirteen separate test runs in the order they were performed, and the results of those tests. Three result values – test accuracy, train accuracy and time – are shown to the right of the thick double border line.[[9]](#footnote-9) The parameter combination that maximized test accuracy is highlighted in yellow(ish).[[10]](#footnote-10)

| Run | **n hidden** | **l2** | **l1** | **epochs** | **eta** | **alpha** | **decrease const** | **mini batches** | **time** | **test Accuracy** | **train Accuracy** |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 1 | 350 | 0.10 | 0.0 | 2500 | 0.00001 | 0.001 | 0.000001 | 100 | 5760 | 80.96 |  |
| 2 | 100 | 0.10 | 0.0 | 500 | 0.00001 | 0.001 | 0.000001 | 100 | 636 | 78.70 |  |
| 3 | 100 | 0.00 | 0.0 | 500 | 0.00100 | 0.001 | 0.000001 | 100 | 492 | 77.25 |  |
| 4 | 100 | 0.25 | 0.0 | 500 | 0.00100 | 0.001 | 0.000001 | 100 | 473 | 60.78 |  |
| 5 | 100 | 0.10 | 0.1 | 500 | 0.00001 | 0.001 | 0.000001 | 100 | 514 | 65.79 |  |
| 6 | 100 | 0.10 | 0.1 | 500 | 0.00001 | 0.001 | 0.000001 | 50 | 502 | 67.15 |  |
| 7 | 200 | 0.10 | 0.1 | 500 | 0.00001 | 0.001 | 0.000001 | 50 | 712 | 66.17 |  |
| 8 | 75 | 0.10 | 0.1 | 500 | 0.00001 | 0.001 | 0.000001 | 50 | 428 | 71.20 |  |
| 9 | 75 | 0.10 | 0.1 | 1000 | 0.00001 | 0.001 | 0.000001 | 50 | 806 | 72.43 |  |
| 10 | 150 | 0.10 | 0.1 | 1000 | 0.00001 | 0.001 | 0.000001 | 50 | 1057 | 74.51 |  |
| *11* | ***125*** | ***0.10*** | ***0.1*** | ***1250*** | ***0.00001*** | ***0.001*** | ***0.000001*** | ***50*** | ***1493*** | ***85.34*** | ***88.69*** |
| 12 | 125 | 0.10 | 0.1 | 1500 | 0.00001 | 0.001 | 0.000001 | 50 | 1823 | 85.33 | 88.72 |
| 13 | 150 | 0.10 | 0.1 | 1500 | 0.00001 | 0.001 | 0.000001 | 50 | 1823 | 85.33 | 88.72 |

Figure - Results of 13 Hyperparameter Combinations on the MLPNN

The three hyperparameters that were not adjusted – eta, alpha and decrease const – are shown in a lighter gray text to deemphasize their role in the table. Training set accuracy/error rates were not computed until the eleventh trail.

Although time – the training time of the analysis – is not an output with respect to accuracy, it is shown to provide insight on what parameters effect the training rate of the MLPNN. Since the MLPNN is CPU-based, training time can be costly.

Additionally, the results are shown graphically in Figure 9 that plots test accuracy versus number of epochs. Three additional dimensions are shown through the shape of the plotted data point (minibatch volume), the size of the plotted data point (number of hidden nodes) and color of the plotted data point (l1 value). Only the l2 value is not represented on the plot.

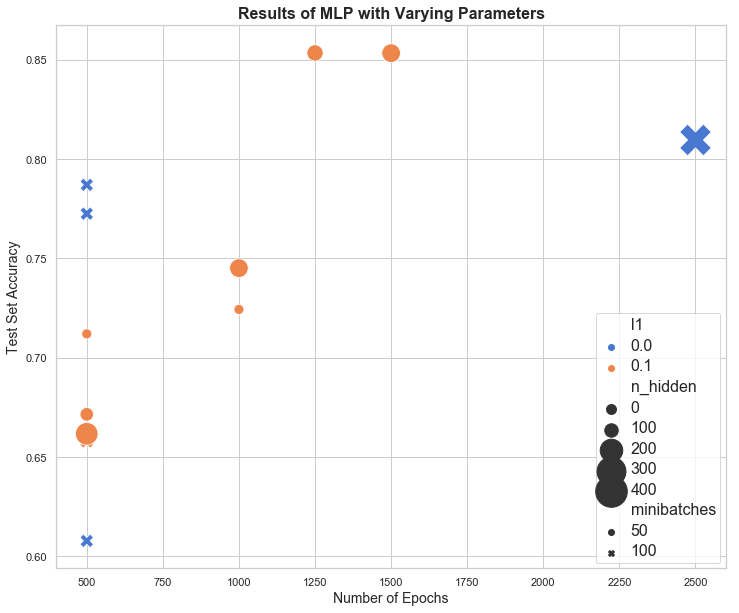


Figure - Results of 13 Parameter Combinations on the MLPNN

Clearly, adding more epochs and more hidden nodes is a not a simple formula for optimization of accuracy, as is evidenced by large blue × shown on the right side of the chart. Using the maximization of test accuracy as the single value for success, the optimal hyperparameter combination for Fashion-MNIST are those from row 11 of Figure 8, achieving an accuracy rate of over 85%.

### Model 1 – Additional Insights

#### Hyperparameter Tuning

The graph shown in Figure 9 provides some insight to the effect that certain hyperparameters have on the MLPNN. But additional insights can be gained by performing a linear regression across the set of five tuned hyperparameters as well. Two such regressions were performed, one using time as the dependent variable and another using test\_error**[[11]](#footnote-11)** as the dependent variable.[[12]](#footnote-12)

The regression summary for test\_error is shown in Figure 10. The adjusted R2 is very strong, showing that almost 97% of the variation in the model can be attributed to these hyperparameters. Also, all variables contribute to the model at the 95% confidence level except n\_hidden, the number of hidden nodes, which comes in closer to 94% confidence.

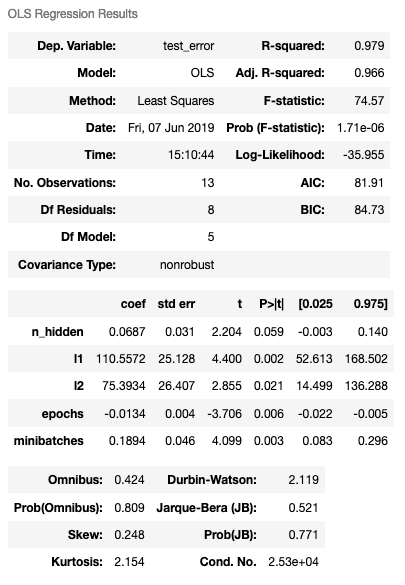


Figure - Test Error Regression of MLPNN

epoch has an overall negative effect on test\_error, despite the outlier previously highlighted in Figure 9. Intriguing and worthy of additional study are the effects of the l1 and l2 parameters on this regression. Both parameters have relatively large coefficients given that they will be multiplied by MLPNN parameter values typically equal to 0.1.

Figure 11shows the regression summary for time. Again, we have a very strong adjusted R2, near 97%, but this time only three of the predictor variables are above the 95% confidence level. Additionally, the confidence level on epoch is extremely strong, as one might expect since the number of epochs is the volume of times a common block of code will loop, thus heavily influencing the overall runtime. When epoch is used as a sole predictor variable in a time-based regression, it accounts for 87% of all variation.[[13]](#footnote-13)

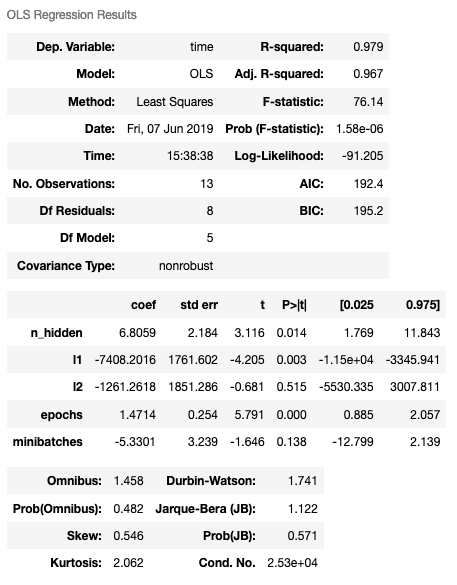


Figure - Time Regression of MLPNN

#### Adaptive Learning Rate

As mentioned previously, the (eta) parameter, also known as the learning rate parameter, is adaptive in the MLPNN implementation that was used. The decrease\_const plays a role in how eta evolves. As shown in Figure 12, the evolution of eta is sigmoidal in its evolution, starting at the initial eta parameter value and decreasing asymptotically to a limit of 0.

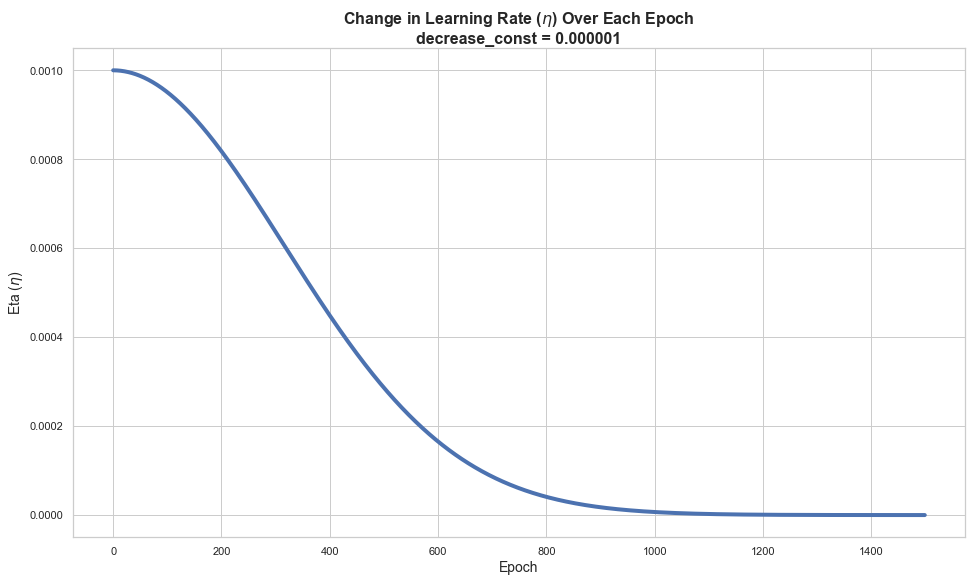


Figure - Evolution of the Learning Rate Value

The curve looks to level off to a near indistinguishable change near the 1100 epochs mark.

#### Default Hyperparameters

One additional trail run was performed but not reported using the default values for all non-required hyperparameters. This was an invocation of the NeuralNetMLP class of the form:

nn = NeuralNetMLP(n\_output=10, n\_features=X\_train.shape[1])

This run came back consistently with a 90% test\_error rate which is the same value as would be expected for a system that consistently picks the same label for all inputs. When checked, all labels were being predicted as 3 when using the default parameter set.

The same results were achieved when explicitly naming the parameters using the default values:

nn = NeuralNetMLP(

n\_output=10, n\_features=X\_train.shape[1], n\_hidden=30,

l1=0.0, l2=0.0, epochs=500, eta=0.001, alpha=0.0, decrease\_const=0.0,

shuffle=True, minibatches=1, random\_state=None)

After individually testing a number of the parameters, the culprit seems to be the use of 1 as the default for minibatches. When the minibatch value was changed to a larger number (10 was the value used) the network stopped predicting a single value for all test cases.

## Model 2 – Convolutional Neural Network

A second model was implemented using the same Fashion-MNIST data set and the same goal: to maximize model accuracy in predicting the test set of data. The second model used a convolutional neural network and was implemented using the PyTorch library. PyTorch is a Python machine learning library that provides tensor computing with GPU acceleration and deep neural network implementations. PyTorch tensors are similar to NumPy arrays, but tensors can also be used on a GPU that supports Nvidia’s CUDA API.[[14]](#footnote-14)

PyTorch (developed, in part, by Facebook[[15]](#footnote-15)) is considered to be a more direct and easier to implement modeling framework than TensorFlow (developed initially by Google). In PyTorch, one defines the network as a class of type nn.module, and feed the input data through it. This is similar to the orientation used in Model 1 and the NeuralNetMLP class. The code is easier to read and more intuitive than, say TensorFlow, and because of its runtime-execution model, it is easy to debug the code as the data passes through the model.[[16]](#footnote-16)

The PyTorch model used in the analysis is based on work published by Ravish Chalwa in his online publication “*Intro to PyTorch with image classification on a Fashion clothes dataset*.”[[17]](#footnote-17) The defined network structure is shown in Figure 13. The input is, as expected, the 28×28 Fashion-MNIST images. The first layer has 16 5×5 convolution filters with 2 pixels of padding. The second layer increases this to 32 filters of the same 5×5 dimension and padding. Both of the first two layers also used a normalization stage, a rectified linear unit stage, and a pooling stage for dimensionality reduction.



Figure - PyTorch CNN Neural Network Model

Data is passed from the input to the first layer, sequentially through the first layer’s four stages, and then on to the second layer. The data again travels through the second layer’s stages in sequence and is then send the third layer that performs a linear transformation on its input resulting in a mapping to one of the output labels.

For the CNN analysis, three hyperparameters, described in Figure 14, were tuned to find an optimal test accuracy result. In this case the **Default** column values are not system defaults, but rather the values used by Ravish Chalwa in his paper.

| Parameter | Default | Description |
| --- | --- | --- |
| **NUM\_EPOCHS** | 5 | Number of learning iterations across the MLPNN. |
| **BATCH\_SIZE** | 100 | Number of items in each minibatch. This is different than the MLPNN where a number of minibatches was specified and batch size was computed. |
| **LEARN\_RATE** | 0.001 | The initial learning rate parameter. |

Figure - Hyperparameters for PyTorch CNN

In total, 11 separate trail runs were performed using nine unique hyperparameter combinations. One combination (epochs=6, batch\_size=50, learning\_rate=0.005) was executed three times to observe the range of accuracy values produced by the CNN.[[18]](#footnote-18) The results of these trails are shown in Figure 15. Timing data was not captured for this model as PyTorch’s use of the GPU minimized timing concerns. Again, the hyperparameter combination that maximized test accuracy is highlighted.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Epochs** | **Batch Size** | **Learning Rate** | **Train Accuracy** | **Test Accuracy** |
| 15 | 100 | 0.005 | 96.11 | 91.28 |
| ***6*** | ***100*** | ***0.005*** | ***94.16*** | ***92.06*** |
| 6 | 50 | 0.005 | 92.89 | 90.59 |
| 93.60 | 91.36 |
| 93.40 | 90.97 |
| 6 | 150 | 0.005 | 93.28 | 90.76 |
| 6 | 75 | 0.005 | 92.42 | 90.59 |
| 6 | 60 | 0.005 | 92.64 | 90.13 |
| 6 | 60 | 0.008 | 93.39 | 91.20 |
| 5 | 60 | 0.008 | 92.90 | 91.15 |
| 9 | 75 | 0.003 | 95.13 | 91.54 |

Figure - Results of 11 Hyperparameter Combinations on the CNN

And the same results are shown graphically in Figure 16

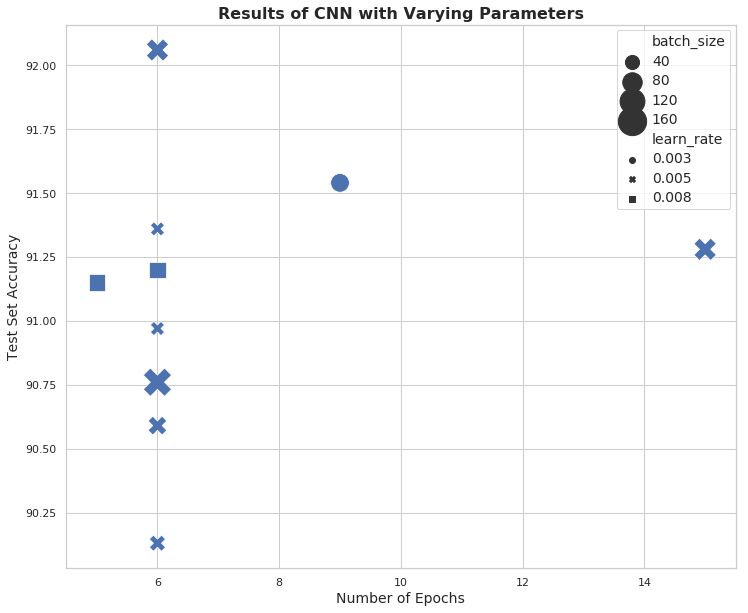


Figure - Results of 11 Parameter Combinations on the CNN

## Overall Results

The three-stage CNN was able to perform at a higher level of accuracy that the MLPNN achieving an accuracy rate of over 92% on the Fashion-MNIST test set. Additionally, the use of the GPU in the PyTorch implementation of the CNN made the network faster to train.

# Conclusions

A 92% accuracy rate means that 8 out of every 100 articles of clothing would be misclassified. The question of “is this rate sufficient?” is more difficult to determine. Imagine this means that a pullover sweater winds up being delivered to the tee shirt department. It seems much less costly for a clerk in the tee shirt department to identify erroneous deliveries and return them to the right department than for a group of employees to identify and properly sort every article of clothing individually. If, on the other hand, the model was adapted to military weapons identification where the lives of servicepeople were on the line, then 92% accuracy may not be sufficient.

The network, as designed, is only one in a wide array of CNN models that could be applied to the problem. Still unknown are the effects of different convolution kernel sizes, adding more convolution stages to the network, and additional hyperparameter combinations. It is possible that alterations to the network design could increase the accuracy of the model for applications requiring higher degrees of certainty.

One goal of Fashion-MNIST was to be an acceptable and viable drop-in alternate to MNIST (digits). As reported earlier, conventional models can easily achieve between 97% and 99.7% accuracy on MNIST (digits). A 2017 Kaggle competition using Fashion-MNIST resulted in many solutions achieving between 92% and 94% accuracy. Therefore, Fashion-MNIST appears to be a reasonable next step in the evolution of MNIST-structured image classification problems.

1. Zalando Research. Fashion-MNIST. <https://research.zalando.com/welcome/mission/research-projects/fashion-mnist/>. [↑](#footnote-ref-1)
2. Mainly to avoid showing the world’s most boring histogram in triplicate. Honestly, a UPC code on the bottom of a box of Cap’n Crunch is more exciting than this histogram. [↑](#footnote-ref-2)
3. Since at least three quarters of the pixels are 0, the inter-quartile range (IQR) would also be 0, as would 1.5 × IQR. Thus, all non-0 pixels would be outliers. [↑](#footnote-ref-3)
4. Based on work done by the author in SYR-IST-707 in February 2019. [↑](#footnote-ref-4)
5. To help other students reduce time waste, the pickle serialization code was shared by the author on a class Slack channel. Therefore, the use of common pickle serialization code may be seen in other’s work as well. [↑](#footnote-ref-5)
6. The author believes the original implementer of the MLPNN framework that was used in this analysis was Professor John Fox of Syracuse University. [↑](#footnote-ref-6)
7. L1 and L2 regularization involves “adding an extra term to the loss function, which penalizes certain parameter configurations.” (<http://deeplearning.net/tutorial/gettingstarted.html#l1-l2-regularization>) [↑](#footnote-ref-7)
8. Taken directly from the framework documentation. [↑](#footnote-ref-8)
9. The MLPNN software provided error rates which have been translated to accuracy rates for this chart by the simple formula accuracy = (1-error). [↑](#footnote-ref-9)
10. The author is color challenged. [↑](#footnote-ref-10)
11. As noted earlier, test error and test accuracy are direct functions of each other. This regression analysis used test error instead of test accuracy as it was the value directly output from the MLPNN process. [↑](#footnote-ref-11)
12. Tests on the predictor variables’ interdependences were not performed so the predictor variables in these regressions may have some degree of collinearity. [↑](#footnote-ref-12)
13. Results of that regression are not shown here. [↑](#footnote-ref-13)
14. KDnuggets. “Introduction to PyTorch for Deep Learning.” Mwiti, Derrick. November, 2018. <https://www.kdnuggets.com/2018/11/introduction-pytorch-deep-learning.html>. [↑](#footnote-ref-14)
15. Author note: I deleted my Facebook account in April 2018 due to my lack of trust with the way Facebook was handling customer data. However, after using the Prophet and PyTorch libraries developed by Facebook, I need to give kudos to the machine learning software development parts of the Facebook organization for providing some fantastic products and making them open source and freely available. I still have no intention to recreate my Facebook account, though. [↑](#footnote-ref-15)
16. Medium.com. “Intro to PyTorch with image classification on a Fashion clothes dataset.” Chawla, Ravish. 2018-May–01. <https://medium.com/ml2vec/intro-to-pytorch-with-image-classification-on-a-fashion-clothes-dataset-e589682df0c5>. [↑](#footnote-ref-16)
17. Ibid. [↑](#footnote-ref-17)
18. These trails were performed in unison with attempts to find how PyTorch’s random seed was set, to produce consistent results on repeated trials. [↑](#footnote-ref-18)