Categorizing the Fashion MNIST data with a Neural Network

Authors: Yuunbum Lim, Sean Coulter, Jesus Moreno

Abstract

This report presents a solution to the Fashion MNIST classification problem via a convolutional neural network. Graphs and Matlab code are presented as visual tools to show the logic involved in solving this problem. Our techniques for approaching this problem, along with crucial and tangential findings, are broken down in the Methods and Results sections. We conclude that the performance of the proposed convolutional neural networks is a function of network architecture (e.g. layer design), and that with a carefully chosen architecture it is possible to achieve reasonably high classification accuracy. We found that having three convolutional layers mixed with dropout had the best results for accuracy.

Introduction

The objective of this work is to solve the MNIST Fashion classification problem using the proposed neural network. The problem aims to classify input data (represented as a 784 element feature vector) with output data (a 10-element vector) which contains the learned likelihood of the input's classification at a given index. The MNIST Fashion dataset can be classified into 10 different clothing items: t-shirt, trouser, pullover, dress, coat, sandal, shirt, sneaker, bag, ankle boot. In this report, we focus on four networks we created to solve the problem and discuss the tradeoffs offered by each. We pay special attention to how these so called "prototypes" optimize performance with respect to accuracy and loss. We attempted to find out what elements of the neural network, when modified, provided better results for accurately guessing the inputted fashion images.

This relates to the class in that we have been learning about ways to train a neural network. We started with basic algorithms, such as perceptrons, then went on to backpropagation, and now finally we investigated convolutional neural networks. Convolutional neural networks are a good fit for image classification because image classification involves so many different variables that need to be accounted for. These variables can be accounted for by the many hidden layers and their respective neurons using a convolutional architecture. We investigate different architectures and make modifications to them based on what we have learned in class.

Methods

The first thing we did was try to decide what neural networking tool to use to solve this problem. Matlab has a neural network toolkit, which gives one the ability to build a custom convolutional neural network by selecting which layers to include. Another tool we knew of was Tensorflow, which also allows one to build a custom convolutional network, using the Python coding language. Ultimately, we decided on using Matlab simply for its simplicity of use and our group members had more experience with Matlab, and minimal experience with Python.

We then needed to download the Fashion MNIST data, and import it into the project. The Fashion MNIST data was in .csv format, so we needed to parse the data into Matlab using the comma delimiter. We noticed that the the data was 60,001 rows by 786 columns. The first row simply had header information, so we ignored that row. The 1st column only labeled the iteration of the data (E.g. 1st clothing item, 2nd clothing item, etc) so we also ignored that. The 2nd column had the expected categorical output of the respective row. When importing the csv, we saved the 2nd column as a 1 X 60,000 vector, to save the expected output for each row. Then, since we know each row contains 784 elements (when ignoring the id and expected output columns), we saved a 60,000 X 784 matrix. The 784 elements represent the pixels of the 28 X 28 image.

Now that the data was loaded into matlab, we had to modify the data so that it could be analyzed by a convolutional neural network. First, we split the data into a training set and a validation set. This validation set would be used as a metric to analyze how the neural network performs as the network is training. The training set would be used to train the neural network. To begin, we split the image data so that the training set had 55,000 images, and the validation set consisted of the remaining 5,000 images. We also split the expected output vector into two vectors; one for the training set and one for the validation set, that were also 55,000 and 5,000 in length, respectively. The next step was to reshape the image data; instead of having a 1 X 784 vector for one image, we needed it to be a 28 X 28 matrix (why this is necessary is explained shortly). So, for all 60000 images, we transformed the vector into a 28 X 28 matrix. The last step that needed to be done was to transform the data into a 4D array. This 4D array would be in the dimensions: 28 X 28 X 1 X (Z), where Z is the number of “pages”. The number of pages would be 55,000 for the training set, and 5,000 for the validation set.

At this point, all the data is in the format we need, and we can now start experimenting with a neural network. To begin, for prototype 1, we used a basic multi-layered convolutional neural network. We used this neural network as a base because of its simplicity. We will outline the architecture of prototype 1, and then explain what each layer is doing, and why we chose to include it.

```
imageInputLayer([28 28 1])
convolution2dLayer(3,16,'Padding','same')
batchNormalizationLayer
reluLayer
maxPooling2dLayer(2,'Stride',2)
convolution2dLayer(3,32,'Padding','same')
batchNormalizationLayer
reluLayer
dropoutLayer
fullyConnectedLayer(10)
softmaxLayer
classificationLayer
```

Input Layer: This layer is simply the 28 X 28 X 1 input for each image. The 1 here is the channel size. A 1 indicates the image will be analyzed in grayscale. The 28 X 28 matrix is the matrix for the image, where each pixel is represented. This input layer will be passed through the neural network.

2 convolution layers: The first convolutional layer has a kernel size of 3 X 3, as does the second convolutional layer. The first convolutional layer has 16 filters, whereas the second one has 32 filters. For the padding, each convolutional layer has the padding size calculated by matlab. By setting padding to 'same', the padding is calculated behind the scenes so that the output has the same size as the input. Both convolutional layers are activated with the relu function. We chose to have two convolutional networks because convolutional layers gather the main features of the image. We needed to have several different filters so that the network can distinguish different elements of the image and hopefully be able to differentiate images. We increased the filters from 16 to 32 (from the first convolutional layer to the second convolutional layer) because as we made the neural network deeper, the network has to distinguish a larger scope of artifacts, so adding more filters helps to distinguish between them.

Batch Normalization Layers: The batch normalization layers help to normalize the output of the layer, thereby reducing the oscillations that may occur due to gradient descent. This thereby speeds up training, and reduces the impact of the layers for layers down the line.

Max Pooling Layer: The max pooling layer is done with a 2 X 2 matrix, with a stride of 2 in the X and Y direction. This helps to reduce the size of the feature maps, which can help with training time. Max pool was chosen because it gathers the maximum point, which is usually the point that has the greatest effect, and we wish to retain that information.

Dropout Layer: This layer randomly sets some of the inputs to 0. This is done to add some randomization to the network, so it isn't memorizing things. This should help with overfitting.

Fully connected layer: This is a 10-neuron layer that is fully connected to the output of the proceeding layer. It is 10 neurons because there are 10 different categories that can be picked from when analyzing an image. This layer needs to be fully connected so we can gather all of

the information that has been passed down the network, to help decipher what the image is classified as. This layer is activated by the soft max calculation, which gives each category of clothing a probability, based on what the neural network has learned. This creates a number from 0 to 1, where the higher the number, the more likely the category is correct classification for the image.

Classification layer: This layer receives the probabilities from the classification layer, and basically selects the category with the highest probability. The category with the highest probability is then turned into a 1, and the rest of the categories are turned into a 0. The 1 indicates that the network has selected that category as its guess for the image.

The layers specified above are specified in an array. Next, one needs to specify the options for the neural network. We set the initial learning rate to .01, and kept it static throughout. We trained using stochastic gradient descent with momentum. We used 30 epochs, with the data being randomly shuffled after every epoch. Last, we tested the validation data every 30 iterations, so we can see a visual of how the validation data is doing compared to the training accuracy. We then called MatLab's trainNetwork function, and awaited the training to complete. We trained the network across different machines using both GPU capabilities (GeForce GTX 1080 Ti with 11178MiB of memory) as well as CPU. Training on the GPU was more efficient and allowed our team to test different architectures in shorter amounts of time.

After training, we could begin testing. Under the circumstances of this homework, we only had access to the test data, but not the expected result of that test data. This test data was in the format of 10,000 rows (10,000 images), with a unique id, and 784 columns, where each column represents a pixel. We also imported this data into matlab the same way we imported the training set. Once the data was in the correct format, we ran the data though the neural network classifier, which in turn returns a prediction for each image. In this case, it will have 10,000 guesses, with a 1 X 10 vector. This vector will only have one 1, and the rest are zeros, and the index that has the 1 is the guess of the neural network. We can only see the performance of out neural network by uploading our network's guess to Kaggle, so we exported the guesses into a .csv file in the format that was specified, and submitted it to Kaggle. Kaggle knows the expected output of each of the 10,000 images in the test set.

For our second prototype, it had the same architectural structure as prototype 1 and followed the same procedures as described above. This time however, we changed some of the options. For this attempt, we set the learning rate to change as the training went on. The initial learning rate was 0.01, and every 5 epochs, the learning rate was multiplied by .02 (thus decreasing it). Also, we set the L2 Regularization optimization value to .0002. This prototype still used stochastic gradient descent and had 30 epochs, with each epoch being randomly shuffled. The reason for attempting this was because in the first prototype, the learning rate was static, and thus may be moving too fast throughout the training. We thought that by reducing the learning rate every 5 epochs, the global minimum would be found, thus increasing the accuracy of the neural network. We used L2 Regularization to attempt to minimize the error due to elements

with larger weights. L2 Regularization minimizes such cases. The results can be seen in figure 3 and figure 4:

For our third prototype, we had the following new architecture:

2 convolutional layers
1 dense layer with 1024 units
1 dropout layer
1 dense layer with 10 units
1 Softmax layer
1 classification layer

This architecture was inspired by the work of *(1)*. It follows the same procedure as Prototype 1 in the convolution layers, but notably it adds a dense layer of 1024 units whose activations are then regularized by dropout. The idea here is to collect a large number of high-level and low-level feature observations in a single, dense layer. We realized that the dropout layer in Prototype 1 was acting on fewer neurons, and as a result, was dropping out more information per neuron than we felt was appropriate. So, we let the network learn the mapping to 1024 neurons where the aim was to distribute information to more neurons so that dropout would only act on smaller chunks of information. This change was positively reflected as shown in Figure 5.

Our fourth prototype had 3 convolution 2d layers, 3 batch Normalization layers, 3 relu Layers, and 2 max Pooling 2d layers, which resulted in having more iterations than other prototypes. Adding one more of each layers ends up with 0.5% higher accuracy while the number of iterations increased fromm 7500s (prototype 3) to 10500s. Through this testing, we could also verify that adding more architectures does actually increase the amount of time to test. We not only added extra layers, but also modified the size of the kernels, the number of filters for second and third layer, which makes the iterations slower due to more process of computations.

imageInputLayer([28 28 1])
convolution2dLayer(3,16,'Padding','same')
batchNormalizationLayer
reluLayer
maxPooling2dLayer(2,'Stride',2)
convolution2dLayer(4,60,'Padding','same')
batchNormalizationLayer
reluLayer
maxPooling2dLayer(2,'Stride',2)
convolution2dLayer(5,60,'Padding','same')
batchNormalizationLayer
reluLayer
dropoutLayer
fullyConnectedLayer(10)

softmaxLayer
classificationLayer

Upon running this last prototype, it has the highest validation accuracy rate, and thus decided to choose this as our final submission for the Kaggle competition.

Results

We have found that the prototype with three layers produces the best results of the four prototypes. More generally, we find that changing network parameters through trial and error may lead to improved network performance. Table I below shows important information about our four prototype configurations which were used for result generation.

Table I: Key configuration data and results for the 4 proposed prototypes

Prototype	Conv layers	Pool layers	Dense layers	Epochs	Validation accuracy	Test accuracy
1	2	1	1	30	90.2	90.0
2	2	1	1	30	91.3	N/A
3	2	2	2	20	91.9	91.9
4	3	2	1	30	92.3	92.5

We present the results of our four prototypes below. Prototype 4 has the highest test accuracy, while Prototype 1 has the lowest. We begin with Prototype 1.

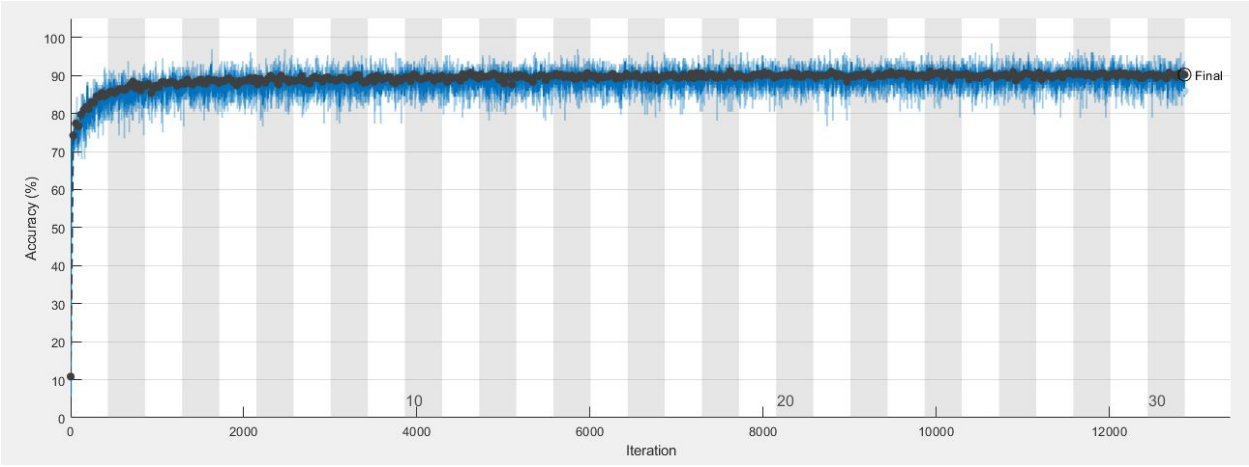

Figure 1: Training accuracy for prototype 1

Prototype 1 had a validation accuracy of 90.2%. (And test accuracy of 90.0%)

This graph is showing that the validation accuracy is not reaching the heights of the training accuracy. This indicates that the neural network is overfitting, and something needs to be done to reduce the overfitting. Perhaps the dropout layer was not having enough of an effect to help with overfitting. This graph also shows that validation accuracy progress seems to stagnate starting around the 6th epoch. This may be because the learning rate was constant, and thus struggled on getting past "saddle" points. After doing some research, some potential solutions to this are to start the learning rate at a somewhat high level, then decay the learning rate every epoch, and then every X epochs, reset it to the initial learning rate. We were not able to attempt such a solution because Matlab didn't have a function like this. It only allows you to decrease the learning rate every X epochs by multiplying it by some factor Y.

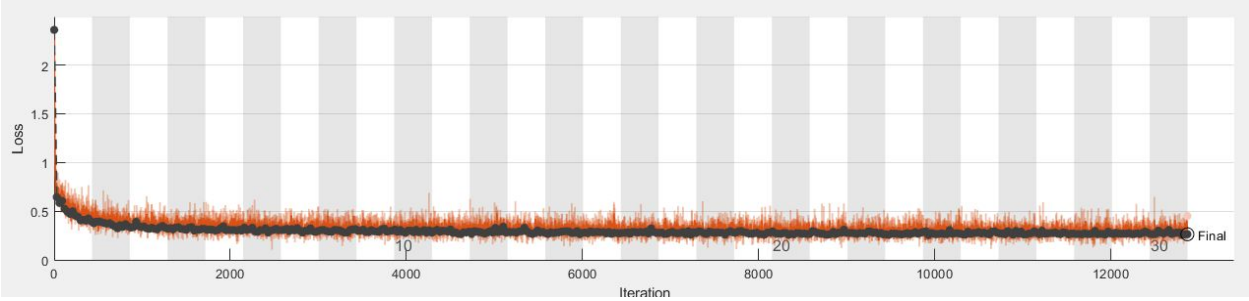

Figure 2: Loss plot for prototype 1

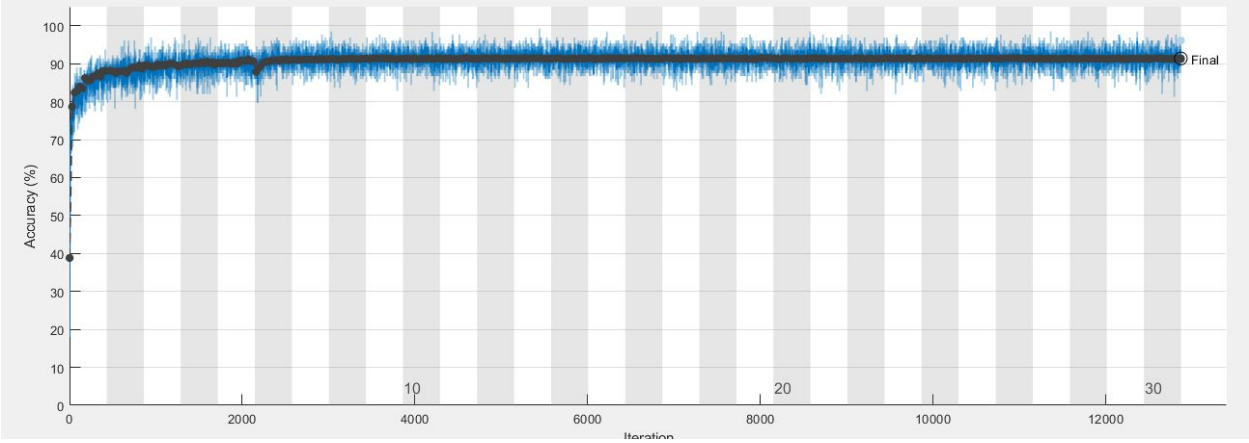

Figure 3: Training accuracy for prototype 2

Prototype 2 had a validation accuracy of 91.38%. We did not determine the test accuracy because we did not submit this to Kaggle so as to not waste a submission.

This plot looks very similar to the plot for prototype one, except that there is one interesting difference. After the 5th epoch, you can see that both the training and validation accuracy dip a little bit. This makes sense because the learning rate was decreased every 5 epochs, so this is the first time where the learning rate is now smaller. Due to the smaller learning rate, it took the network a bit more time to “get back on track”, so to speak, because it takes shorter steps. There weren’t any more dips on every 5th epoch iteration probably because the network started to converge on some point.

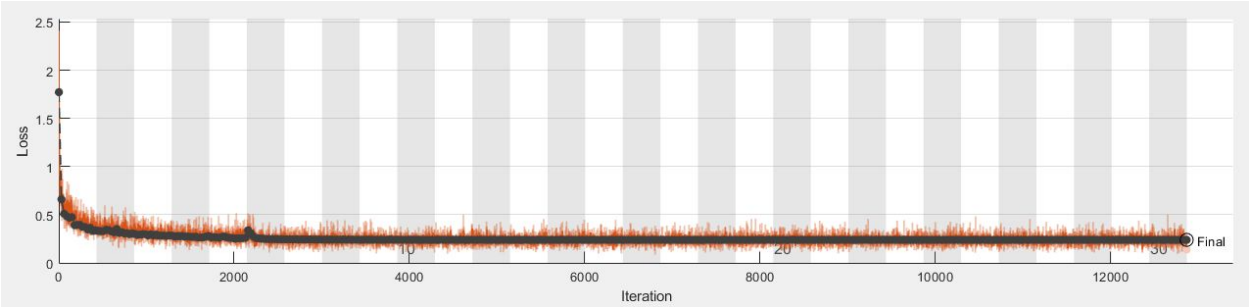

Figure 4: Loss plot for prototype 2

Just like in the training accuracy plot, there is a difference that can be seen shortly after the 5th epoch. The loss value slightly increases, probably due to the new learning rate.

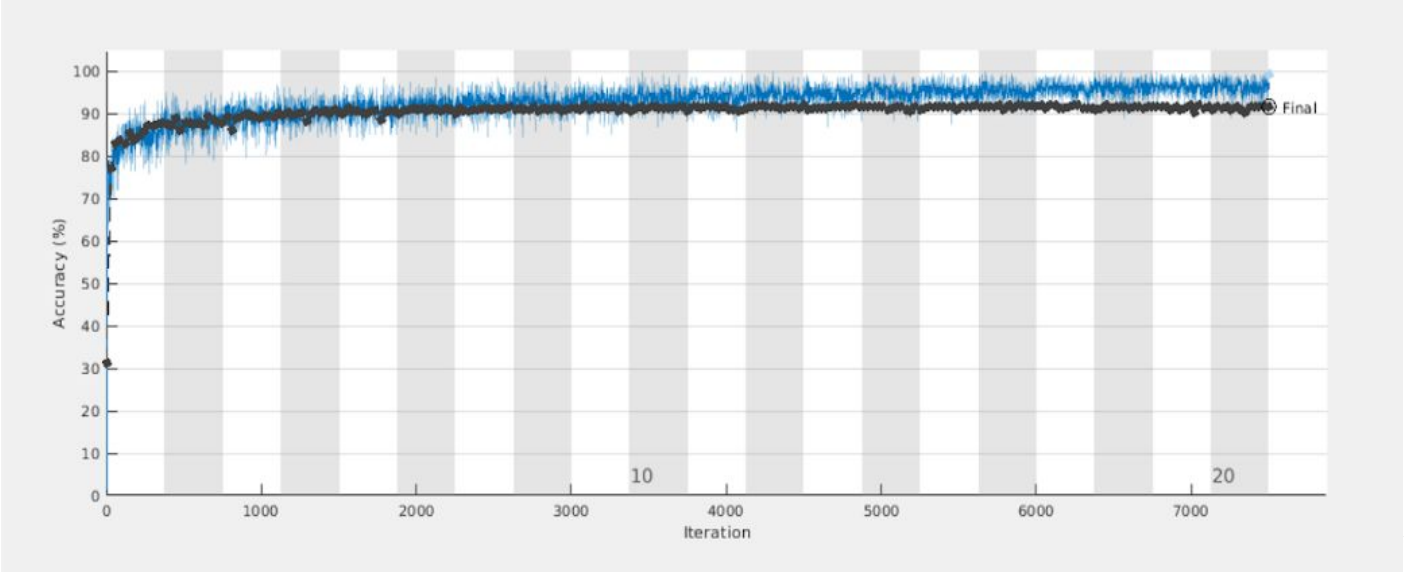

Figure 5: Training accuracy for architecture prototype 3

In Figure 5, an interesting trend can be observed starting around iteration 3000. Here, the training accuracy (blue) continues its upward trend while the validation accuracy (black) decreases its rate of change. This possibly indicates overfitting of the

training data thus giving a false sense of increased accuracy. The network completes its run after 20 epochs with the validation accuracy approximately 5% lower than the training accuracy. Upon submitting this Prototype's test results to Kaggle, we found that the validation accuracy measure was a better representation of results (91.59% validation accuracy to 91.96% test accuracy). We can conclude that the validation accuracy is a good estimate of the network's performance in a test setting, i.e. a more realistic scenario.

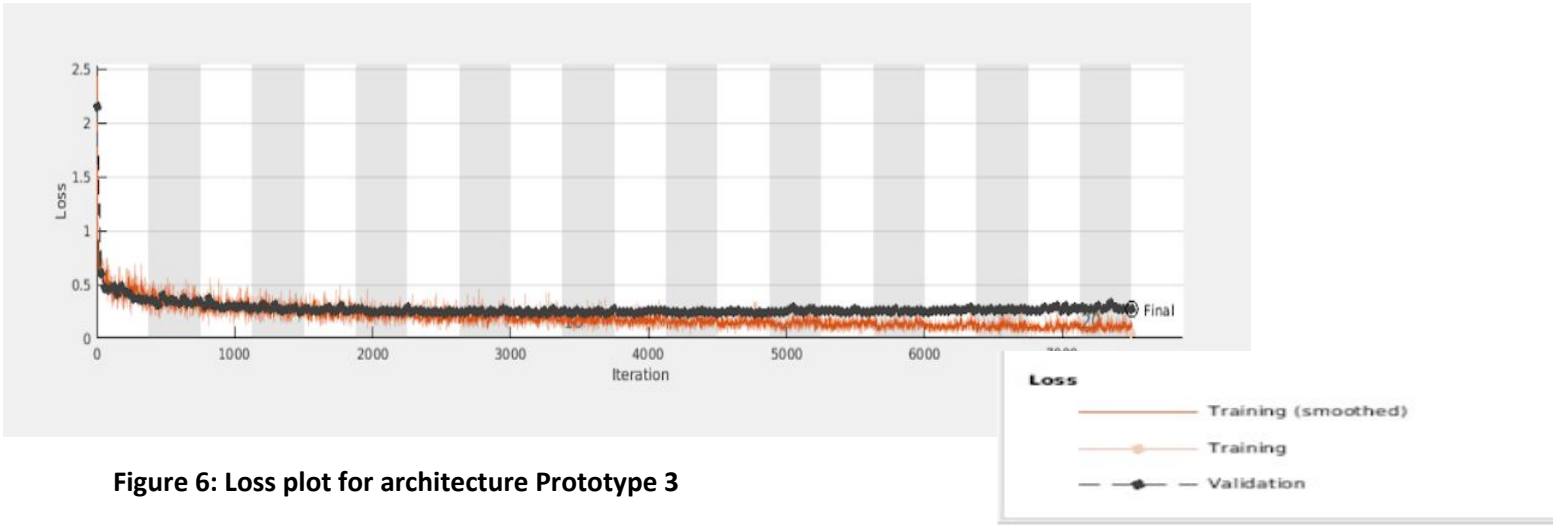

Figure 6: Loss plot for architecture Prototype 3

We can see from Figure 5 that the same phenomenon observed in Figure 6 occurs. The loss appears subject to inconsistent decrease around iteration 3000 until the final iteration. This may be because the loss optimization algorithm (stochastic gradient descent with momentum) is taking too large a "step" in the direction of steepest descent. The blame then lies with the learning rate of our model (constant at 0.01), which in this case was too large for later stages of training. Future work may benefit from combining our idea from prototype 2, where we gradually decrease the learning rate every X epochs, where X varies depending on the problem.

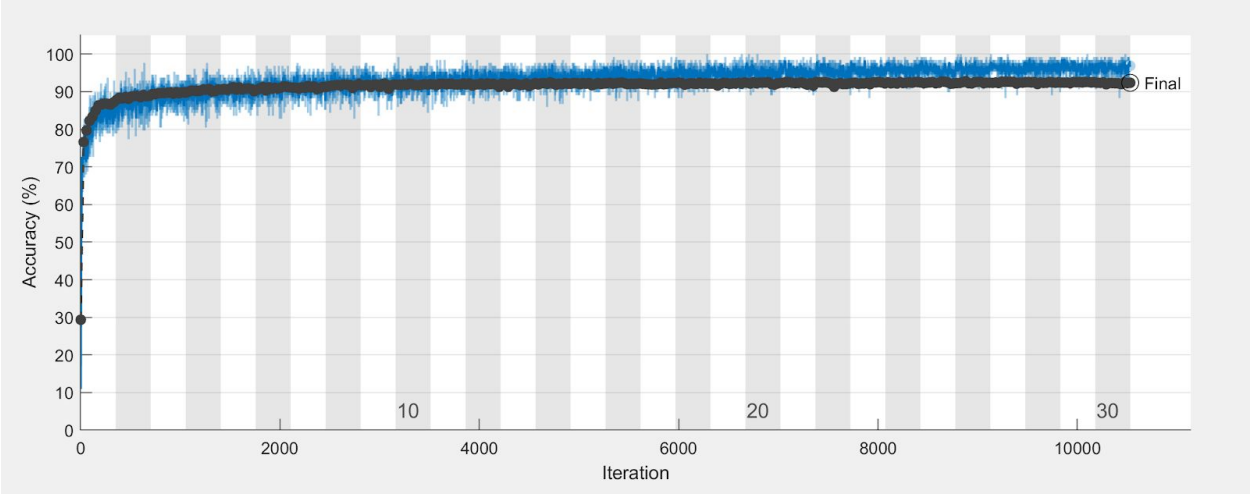

Figure 7: Training accuracy for prototype 4

Had a validation accuracy of 92.35%. When it was tested against the Kaggle test set, it had an accuracy of 92.5%.

This plot looks very similar to prototype 3's accuracy plot, however, the difference between the training accuracy and validation accuracy is smaller than that of prototype 3. This shows that a marginal improvement was made through this neural network. However, due to the training accuracy being higher than the validation accuracy, the neural network is still overfitting.

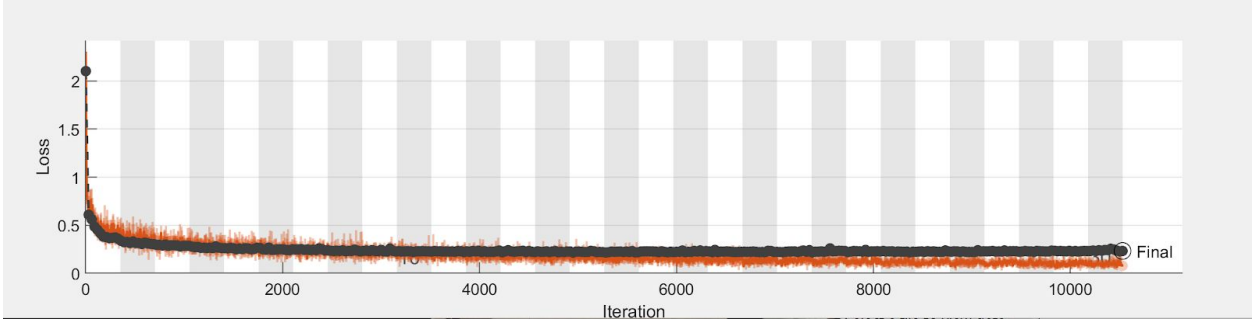

Figure 8: Loss plot for prototype 4

We present an alternate view of the code used to run Prototype 4 below in Figure 9.

```
for i = 1:15000
    validationMatrix = reshape(train2(i+45000,:),[28,28])';
    validationMultiMatrix(:,:,1,i) = validationMatrix;
end

cell = {validationMultiMatrix, expectedOutputvalidation};

layers = [
    imageInputLayer([28 28 1])
    convolution2dLayer(3,16,'Padding','same')
    batchNormalizationLayer
    reluLayer
    maxPooling2dLayer(2,'Stride',2)
    convolution2dLayer(4,32,'Padding','same')
    batchNormalizationLayer
    reluLayer
    maxPooling2dLayer(2,'Stride',2)
    convolution2dLayer(5,32,'Padding','same')
    batchNormalizationLayer
    reluLayer
    dropoutLayer
    fullyConnectedLayer(10)
    softmaxLayer
    classificationLayer];

options = trainingOptions('sgdm','InitialLearnRate',0.01,'MaxEpochs',30,'Shuffle','every-epoch','plots','training-progress','ValidationData',cell,'ValidationFrequency',30);

trained = trainNetwork(multimatrix, categorical(expectedOutput),layers,options);


prediction = classify(trained, testMultiMatrix);
```

Figure 9: Prototype 4 layers code

Conclusion

We have explored the Fashion MNIST classification problem using deep neural networks and highlighted the performances of 4 Prototype networks. By analyzing layer configurations and Matlab plots for these four Prototypes, we have attempted to explain performance tradeoffs and gains and have suggested reasons for deviations from desired results. Through the improvement from Prototype 1 to Prototype 4, we understand that the trial and error process of network optimization is a robust way to improve accuracy, but is time-consuming and hardware dependent. At the same time, we learned that neural network performance optimization by holding parameters constant and changing others doesn't necessarily lead to monotonic improvement and that the principles of convolution, pooling, regularization, and other operations commonly found in deep networks may have to be extended or retracted in order to achieve the most satisfactory results.

The main element that seemed hold our proposed neural networks back from higher accuracy was that our neural networks kept overfitting while training. This pattern was seen in prototypes 3 and 4. It seems like when we implemented a mechanism to reduce this overfitting, such as in prototype 2, the difference between the training and validation accuracy was minimized, but did not yield an increase in accuracy. This led us to believe that in the future, we would focus on investigating what can be done to minimize overfitting. Theoretically, if we are able to reduce overfitting, the neural network would be able to perform better on data that is new to it. One mechanism we attempted (but did not include in the method section because the code for it wasn't working) was to augment images randomly when training the neural network. We were trying to change images randomly by flipping the image X degrees, randomly

shifting pixel, and randomly zeroing some of the pixel values. These would minimize overfitting because it is like introducing a new input to the training that is not in the training set, thus increasing the data count and improving the neural network.

Something else we learned is that training neural networks takes a high portion of time. Some of our architectures we attempted took 3+ hours. We learned that in cases like this, it may be useful to use more pooling to decrease the size of the 2D-array that is being learned, as well as using more compute power.

Overall, the fourth prototype we introduced produced the highest accuracy for both the test set on Kaggle and the validation set. We believe it performed better because it had 3 convolutional layers (versus 2 in the other implementations) and each convolutional layer had more filters. This increase in filters allowed the network to recognize more facets of the images, thus increasing accuracy.

References

1. https://www.tensorflow.org/tutorials/estimators/cnn
2. https://www.mathworks.com/help/deeplearning/examples/create-simple-deep-learning-network-for-classification.html
3. https://www.mathworks.com/help/deeplearning/ug/layers-of-a-convolutional-neural-network.html
4. https://www.mathworks.com/help/deeplearning/ug/setting-up-parameters-and-training-of-a-convnet.html