ECE 1513 Graduate Project

Robert Agnoletto

April 10, 2019

1 Introduction

For this assignment I will be using the suggested sample dataset of house numbers. Specifically, the 32x32 cropped set as opposed to the varying size with multiple numbers set. I trained all models with 60,000 images with a 20,000 image validation set. The data set was shuffled before dividing them so it was different every time.

The three algorithms I will discuss and implement in this paper are Neural Networks (NN), Convolutional Neural Networks (CNN), and Principal Component Analysis (PCA).

2 Convolutional Neural Networks

To start, I first wanted to investigate how a simple convolutional neural network performed for this problem. This would give me a good baseline for the problem and seemed like the best choice because it was the most standard approach for the problem of classifying images.

2.1 Building off of Assignment 2

Since we tackled a similar problem for assignment 2 using CNNs, I decided to implement a similar model as that as a starting point to work off of. For my first model, I recreated the one suggested for assignment 2. However, I made a couple changes to it. First, I of course needed to change the size of the input tensor from 28x28 to 32x32x3 as these images had slightly higher resolution and were colour instead of black and white. I also changed the first fully connected layer from 784 to 1024. Using the same logic, I wanted an output for every pixel in my image. Finally, the last change I made was to include batch normalization after both hidden layers instead of just after the convolutional layer. Normalization is often done between every layer to help speed up convergence so I thought it would improve my model and probably wouldn't hurt. In assignment 2 my partner focused on the CNN section while I did more of the numpy NN section. This combined with the fact that our network was not very successful for that assignment meant I mostly had to start from square one for tuning. Since this network is even more complex than the one from a2 due to large images, this was probably the most time consuming part of the project.

I also opted to use the higher level tf.layers functions to have a cleaner and easier to read codebase and make it simpler to change my models. My initial CNN model was defined as such:

```
def buildGraph():
    print('1conv␣regular␣50␣ecpochs␣00005␣learning␣rate')
    X = tf.placeholder(tf.float32,[None, 32,32,3], name="X")
    y = tf.placeholder(tf.float32,[None,10], name="y")
    is_training = tf.placeholder_with_default(False, (), 'is_training')
    regularizer = tf.contrib.layers.l2_regularizer(scale=0.1)
    #32x32x32
    conv1 = tf.layers.conv2d(inputs=X,filters=32,kernel_size=[3, 3],
                             padding="same",activation=tf.nn.relu,kernel_regularizer
                                 ↪ =regularizer)

    # ceil(32/2) = 16x16x32
    pool1 = tf.layers.max_pooling2d(inputs=conv1, pool_size=[2,2], strides=2,
        ↪ padding="same")
    norm1 = tf.layers.batch_normalization(pool1, training=is_training, momentum
        ↪ =0.9)
    # 16x16x32 = 8192
    x1 = tf.layers.flatten(norm1)

    # 1024 = 32x32
    full2 = tf.contrib.layers.fully_connected(inputs=x1, num_outputs=1024,
        ↪ weights_regularizer=regularizer)
    norm2 = tf.layers.batch_normalization(full2, training=is_training, momentum
        ↪ =0.9)
    full3 = tf.contrib.layers.fully_connected(inputs=norm2, num_outputs=10,
        ↪ weights_regularizer=regularizer,activation_fn=None)

    y_pred = tf.nn.softmax(full3)

    CE = tf.reduce_mean(tf.nn.softmax_cross_entropy_with_logits_v2(labels = y,
        ↪ logits = full3), name='cross_entropy')
    l2_loss = tf.losses.get_regularization_loss()
    loss = CE+l2_loss

    optimizer = tf.train.AdamOptimizer(learning_rate = 0.000001, epsilon = 1e-04)
    update_ops = tf.get_collection(tf.GraphKeys.UPDATE_OPS)
    train = optimizer.minimize(loss=loss)
    train = tf.group([train, update_ops])
    return X, y, y_pred, full3, loss, CE, l2_loss, is_training, train
```

The update ops code is needed to make the batch normalization work. These moving means and standard deviations need to be updated while training. The is_training place holder is used to ensure that these values are only updating when training and not when evaluating the network. Working off our own code, I set learning rate = 1e-06, which was clearly much too small, and epsilon

to 1e-04. For 40 epochs with a batch size of 32, it took over 4 and a half hours for this model to be trained (on my 2015 macbook pro). While the model performed pretty well, it was clearly still learning and had not converged yet. See the plots below for the training and validation loss and accuracy.

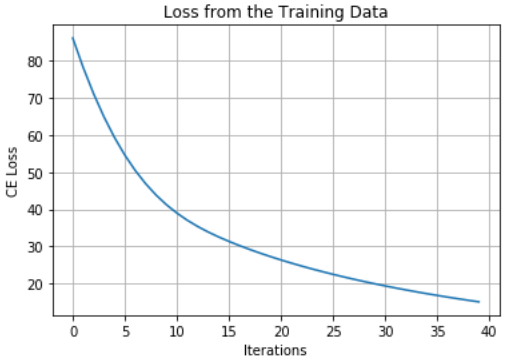

Figure 1: CE loss + 0.1 regularization for the training data 1e-06

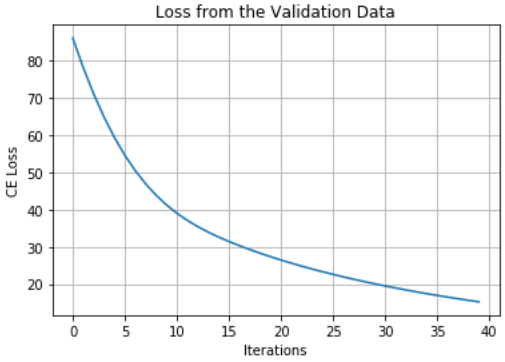

Figure 2: CE loss + 0.1 regularization for the validation data 1e-06

After some experimenting with running the model for a few epochs with different learning rates, I eventually settled on raising it to 5e-05 and leaving epsilon as the default for the next full training and running it for 50 epochs. This estimation was necessary to save time and make sure that each full training I ran counted. See below for the resulting plots of the loss over time.

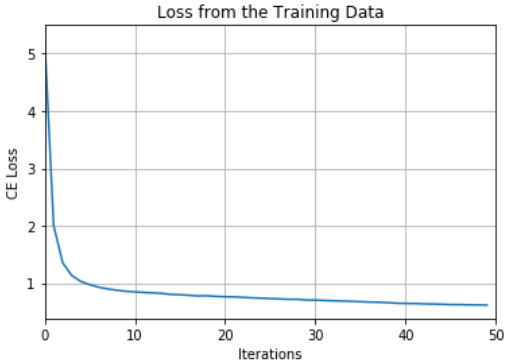

Figure 3: CE loss + 0.1 regularization for the training data 5e-05

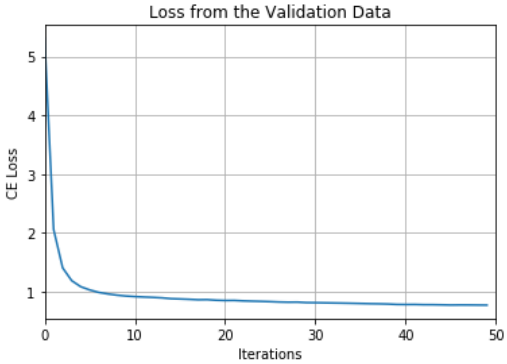

Figure 4: CE loss + 0.1 regularization for the validation data 5e-05

Here we see a nicer convergence and both losses reach much lower values. We can see the overall performance comparison in the following table.

Model	Training Loss	Validation Loss	Training Accuracy	Validation Accuracy
1e-6, 40 epochs	15.05961	15.20264	0.87335	0.81850
5e-05, 50 epochs	0.61756	0.76632	0.91582	0.86175

Table 1: The final results of the model with different learning rates

Clearly we see a large improvement in all categories. The validation accuracy is definitely lower than the training accuracy suggesting there may be some over-fitting, but when we look at the plots it is clear that something like early stopping would not help as the loss of the validation never grows. Some other options will be considered in a later section.

2.2 Adding a Second Convolutional Layer

After doing some research, it became clear that most image classifiers use more than one convolutional layer to extract more features from the images. Therefore, I decided to start by adding a single extra convolutional layer to the model described in the previous section. I also added max pooling and batch normalization after this layer. The strides were kept the same but I made this layer have 64 layers compared to the previous layers 32. I was initially surprised to find that this model was actually faster to train. I assumed that adding an extra layer would increase the complexity. However, this was not the case as most of the complexity from the last model came from our first fully connected layer. This layer with 1024 outputs had 16x16x32 = 8192 inputs. However, with this new model we only have 8x8x64 = 4096 inputs. The source of this simplification was the additional max pooling layer. See below for the plots from this model as well as a comparison with the previous model with only one convolutional layer. Since a learning rate of 5e-05 and 50 epochs worked well last time we used this again here.

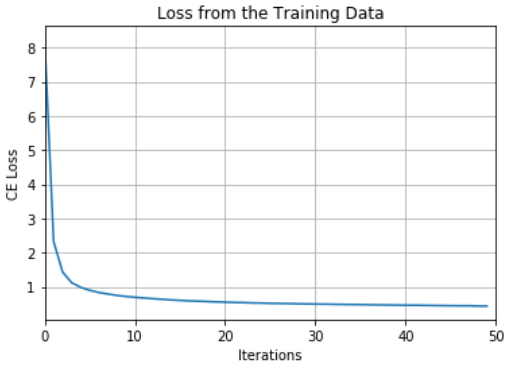

Figure 5: CE loss + 0.1 regularization for the training data with 2 conv layers

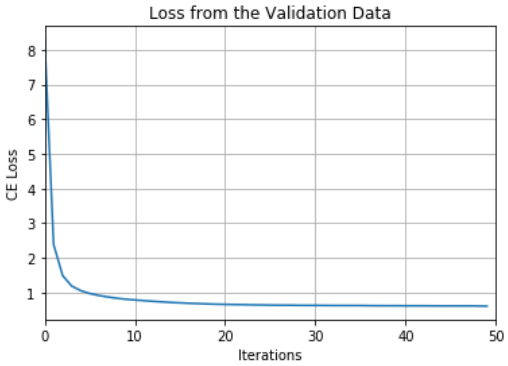

Figure 6: CE loss + 0.1 regularization for the validation data with 2 conv layers

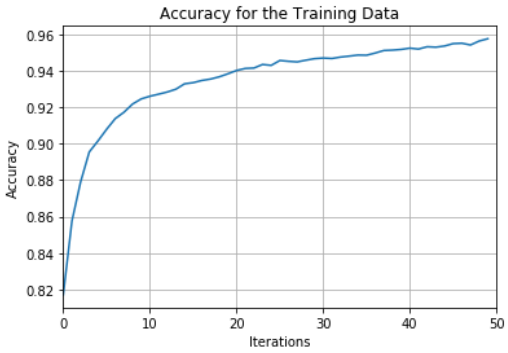

Figure 7: Accuracy of the training data with 2 conv layers

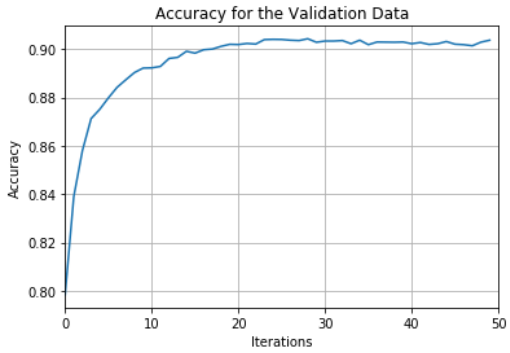

Figure 8: Accuracy of the validation data with 2 conv layers

Model	Training Loss	Validation Loss	Training Accuracy	Validation Accuracy	Run time
1 conv layer	0.61756	0.76632	0.91582	0.8617	5.7 hours
2 conv layers	0.43590	0.60612	0.95747	0.90370	5.3 hours

Table 2: The final results of the 1 and 2 layer models

A few things of note. From fig. 8 we can see that there is some over-fitting that could be corrected by early stopping. While the training accuracy is still going up, we see that the validation loss has converged and has decreased a little with continued training. Using early stopping would increase the accuracy of the validation set by 0.1% and cut the run time in half. Increasing the speed would be helpful, but since the accuracy of the validation plot is noisy and not very smooth, picking a stopping point based on its peak would be highly dependent on the validation set and would likely not translate to an overall increase in accuracy for unseen data. Overall, we see a similar discrepancy between the training and validation performance, indicating that adding an extra layer did not increase our over-fitting. However, since the performance in all categories has improved as well as a quicker training time, clearly this second layer offers an improvement.

2.3 Changing the Filter Size of the Convolutional Layers

From both models in this section, we achieved a very good level of accuracy on the validation data at just over 90% and 86% for the 2 layer and 1 layer CNNs respectively. However, with the Training accuracy around 5% higher for both, there is clearly a degree of over-fitting. This combined with the speed of the models being lacklustre (over 5 hours for both) suggests that we may benefit from simplifying the models. To achieve this, I decided to alter my models by decreasing the filter sizes by 50%. For our 1 layer CNN, we are left with a filter size of 16, and for the 2 layer one we now have 16 and 32. See below for a comparison of the performance of these new simplified models.

Model	Training Loss	Validation Loss	Training Accuracy	Validation Accuracy	Run time
1 layer original	0.61756	0.76632	0.91582	0.8617	5.7 hours
1 layer half filters	0.64546	0.78934	0.91080	0.85855	2.2 hours
2 layers original	0.43590	0.60612	0.95747	0.90370	5.3 hours
2 layers half filters	0.42487	0.60908	0.96052	0.90160	1.6 hours

Table 3: The final results of each model with their simplified counterparts

The results in table 3 are interesting. While the decrease in complexity did not have an affect on over-fitting, it also did not have an affect on the overall performance of the models. Therefore, we have been able to get the same results from models that take about one third of the time to train. Clearly this is a big improvement even though it did not help over-fitting.

2.4 Other Attempts to Improve Over-fitting

Continuing on with the 2 layer CNN with half filter sizes due to its superior performance, we again want to increase the validation accuracy. Since we have such a high training accuracy, we want to increase the validation accuracy at the expense of that. This is preferable as the validation accuracy is a better gauge for how we will perform on new unseen data.

A couple options were explored here. Increasing the regularization factor, removing regularization and using the drop-out method, and a combination of the two. Unfortunately, all results in the experiments I was able to run did not improve the validation set performance. While the tested changes did have more similar results for the training and validation data-sets, the actual validation accuracy and loss was worse. So while this indicates less over-fitting, the actual performance got worse so it was not something I went forward with.

3 Fully Connected Neural Networks

Due to the long training lengths of the initial CNN models I made, as well as the apparent over-fitting, I tried to model this problem with a traditional neural network hoping that the simpler model would be faster to train as well as resulting in less over-fitting.

The model I used was defined as follows:

```
def buildGraph1full():
    X = tf.placeholder(tf.float32,[None, 32,32,3], name="X")
    y = tf.placeholder(tf.float32,[None,10], name="y")
    is_training = tf.placeholder_with_default(False, (), 'is_training')
    regularizer = tf.contrib.layers.l2_regularizer(scale=0.1)
        # 3072 = 32x32x32
    x1 = tf.layers.flatten(X)
    # 1024 = 32x32
    full2 = tf.contrib.layers.fully_connected(inputs=x1, num_outputs=1024,
        ↪ weights_regularizer=regularizer)

    norm2 = tf.layers.batch_normalization(full2, training=is_training, momentum
        ↪ =0.9)

    full3 = tf.contrib.layers.fully_connected(inputs=norm2, num_outputs=10,
        ↪ weights_regularizer=regularizer,activation_fn=None)

    y_pred = tf.nn.softmax(full3)

    CE = tf.reduce_mean(tf.nn.softmax_cross_entropy_with_logits_v2(labels = y,
        ↪ logits = full3), name='cross_entropy')

    l2_loss = tf.losses.get_regularization_loss()

    loss = CE+l2_loss
    optimizer = tf.train.AdamOptimizer(learning_rate = 0.00001)
    update_ops = tf.get_collection(tf.GraphKeys.UPDATE_OPS)
    train = optimizer.minimize(loss=loss)
    train = tf.group([train, update_ops])
    return X, y, y_pred, full3, loss, CE, l2_loss, is_training, train
```

Here we see a similar model to the previous ones, but instead of convolutional layers, we simply flatten the image and then feed it directly into the single hidden fully connected layer. In addition to this, I used a slower learning rate as convergence was not smooth when using the learning rate I settled on for the CNNs.

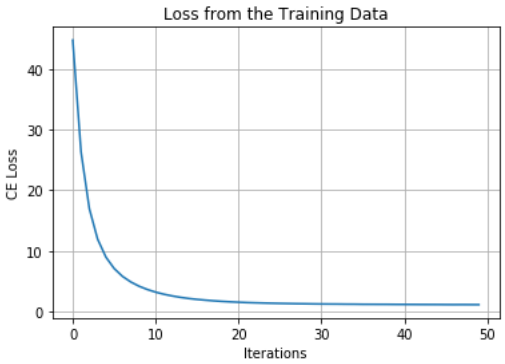

Figure 9: CE loss + 0.1 regularization for the training data with 1 hidden fully connected layer

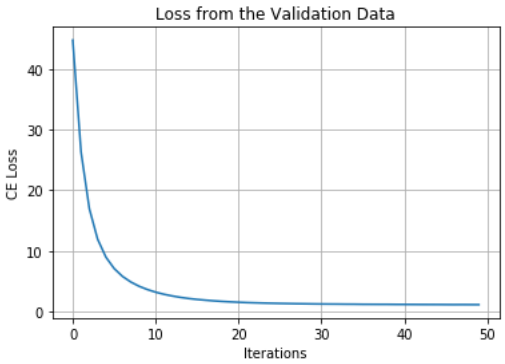

Figure 10: CE loss + 0.1 regularization for the validation data with 1 hidden fully connected layer

The final training and validation accuracy after 50 epochs was 80% and 79% respectively. In addition to this, it took roughly 1.6 hours for this model to be trained. I made this model after my initial attempts at CNN were so slow before I thought to try decreasing the filter size. As a result, the 1.6 hour training time was a significant improvement. While there is less over-fitting as we can see from the similar performance with out of sample data, the actual performance is significantly worse than the CNNs. While this offered me a faster model to experiment with, after decreasing the filter size on convolutional layers, the speed advantage this model had was lost, and we are left with a completely inferior model. Attempts at directly changing this model in terms of the size of the hidden layer were unsuccessful. It is expected that this model would perform worse since CNNs specifically excel when dealing with images. Here we are not looking for any higher level features in the image since we are considering each pixel element (3 for each pixel with colours, rgb) to be independent. This significantly hampers our ability to infer shapes in the images which is crucial if we want to know what the picture is of. While I did not have much success in coming up with a better pure NN, we will see some other improvements in the next section.

4 Principal Component Analysis (PCA)

Principal Component Analysis (PCA) is generally an exploratory tool used in the unsupervised learning context. As the name suggests, this algorithm seeks to extract the principal components of the data at hand to reduce the overall amount of data. This can be useful to determine the amount of components are actually in our data. Often, the variables we have to train on are not completely uncorrelated from one another and therefore we have redundant data and do not understand how the variables relate to one another. The result of PCA is an uncorrelated orthogonal basis which means we have separated the related data and have it in a more informative form. This ideally could solve some problems that traditional neural networks face. By treating every colour component of every pixel as its own unique variable, we are missing out on how the pixels relate to one another, and are faced with a high dimensional input. Using the result of PCA as the input of our neural

network, will ideally make it faster and more accurate. For this part of the project, I used sklearn's PCA functionality. The code that implements it can be seen below.

```
x_train, y_train, x_val, y_val, x_test, y_test, randIndx = loadData()

x_train_flat = x_train.flatten().reshape(len(x_train), 32*32*3)
x_val_flat = x_val.flatten().reshape(len(x_val), 32*32*3)
scaler = StandardScaler()
scaler.fit(x_train_flat)

x_train_pca = scaler.transform(x_train_flat)
x_val_pca = scaler.transform(x_val_flat)
pca = PCA(.95)
#pca = PCA(n_components=100)
pca.fit(x_train_pca)
lower_dimensional_train = pca.transform(x_train_pca)
lower_dimensional_valid = pca.transform(x_val_pca)

approximation_train = pca.inverse_transform(lower_dimensional_train)
approximation_valid = pca.inverse_transform(lower_dimensional_valid)
```

In this code we are first scaling the input, which the library requires, and then fitting our PCA model to our training set. We also want to specify something about how we want our data to be transformed. We can do this by specifying the number of components we want our transform to have, or be telling it how much of the variance we want it to capture. We can then transform our data to this lower dimension form to be fed into our model, and we can even transform it back to get an idea of how much information we lose in this process. Example outputs for each of the models specified can be seen below.

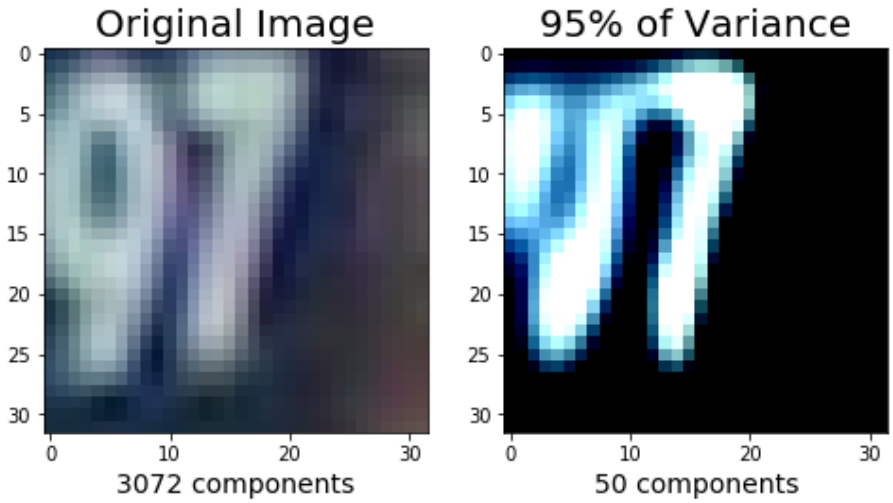

Figure 11: Original image and PCA transform with 95% retained variance specified

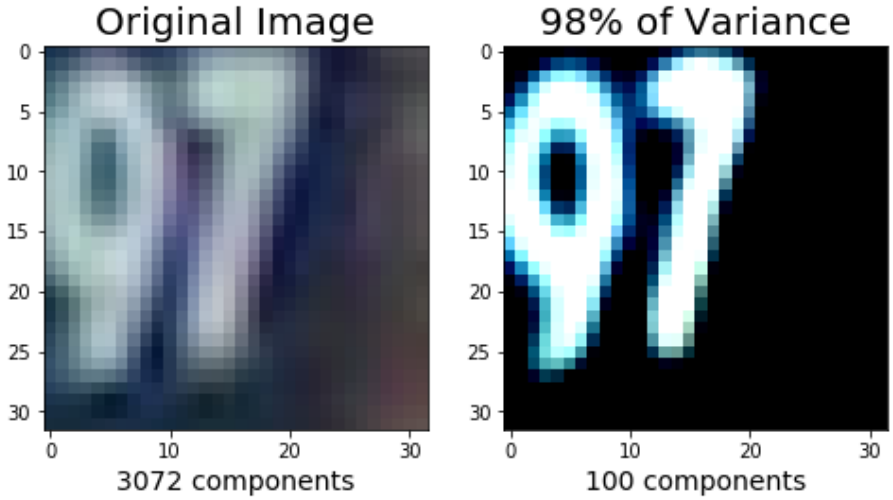

Figure 12: Original image and PCA transform with n_component=100 specified

In figure 11 we see that sklearn's pca library was able to retain 95% of the variance in our data while lowering our dimension from 32x32x3=3072 to only 50 components. Then, in figure 12 we see the results where 100 components are specified and it is able to retain 98% of the variance with that restriction. While both inverse transformations show that there is a lot we can still make out from the image with such a large reduction, the benefit of more components is clearly seen in the

second figure.

4.1 Using PCA Transform as a NN Input

As mentioned earlier, our end goal, aside from making some interesting images, was to improve our NN by feeding it more informative and concise information about the images. To do this we reused our 1 full model with a slight modification.

```
def buildGraph1fullpca(pca):
    X = tf.placeholder(tf.float32,[None, pca], name="X")
    y = tf.placeholder(tf.float32,[None,10], name="y")
    is_training = tf.placeholder_with_default(False, (), 'is_training')
    regularizer = tf.contrib.layers.l2_regularizer(scale=0.1)

    # 1024 = 32x32
    full2 = tf.contrib.layers.fully_connected(inputs=X, num_outputs=1024,
        ↪ weights_regularizer=regularizer)
    norm2 = tf.layers.batch_normalization(full2, training=is_training, momentum
        ↪ =0.9)
    full3 = tf.contrib.layers.fully_connected(inputs=norm2, num_outputs=10,
        ↪ weights_regularizer=regularizer,activation_fn=None)
    y_pred = tf.nn.softmax(full3)

    CE = tf.reduce_mean(tf.nn.softmax_cross_entropy_with_logits_v2(labels = y,
        ↪ logits = full3), name='cross_entropy')

    l2_loss = tf.losses.get_regularization_loss()
    loss = CE+l2_loss

    optimizer = tf.train.AdamOptimizer(learning_rate = 0.00001)
    update_ops = tf.get_collection(tf.GraphKeys.UPDATE_OPS)
    train = optimizer.minimize(loss=loss)
    train = tf.group([train, update_ops])
    return X, y, y_pred, full3, loss, CE, l2_loss, is_training, train
```

The main difference here is that we are feeding it an already flattened array, and that we are not always sure of the dimensions of X at runtime (when we specify variance instead of components). For this reason we need to pass in the resulting number of components from our model, but otherwise everything else is the same. I tried many variations of this model as it was quick to train, and you can see the results in the table below.

Layers	Learn Rate	Var, N_comp	T Loss	V Loss	T Acc	V Acc	Run time
1024	1e-5	0.95,50	0.89993	0.94791	0.83705	0.81870	12 minutes
1024	5e-5	0.95,50	0.77009	0.84421	0.85372	0.82810	12 minutes
1024,512	5e-5	0.95,50	0.64504	0.93576	0.88310	0.81230	37 minutes
1024	5e-5	0.98,100	0.69669	0.79612	0.88685	0.85065	16 minutes

Table 4: The final results of each model of NN with PCA input

There are a few interesting things to note from these results. First, by transforming our data to a better form, our model performs better with a higher learning rate than before. Also, we get better results from this compared to giving the model our full image. Then, we actually get a worse result when we tried to increase the complexity of our model by adding an extra layer. Here we see a higher degree of over-fitting and overall we see worse performance on our validation set. Finally, by increasing the number of components, and therefore also our retained ratio of variance, we get even better results that start to get close to what our CNNs achieved. In addition to this, the speed at which this model can be trained is a massive step up from our previous models. Even including the time it took for the PCA to train on and transform our data which was only about a minute or two. See the plots from the last model in the table (best performing) below.

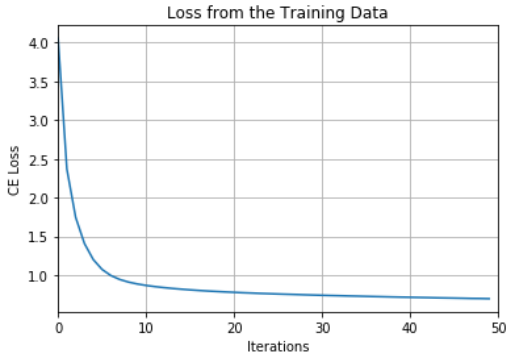

Figure 13: CE loss + 0.1 regularization for the training data with PCA input into NN

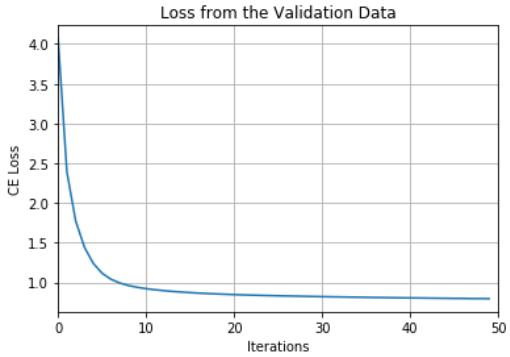

Figure 14: CE loss + 0.1 regularization for the validation data with PCA input into NN

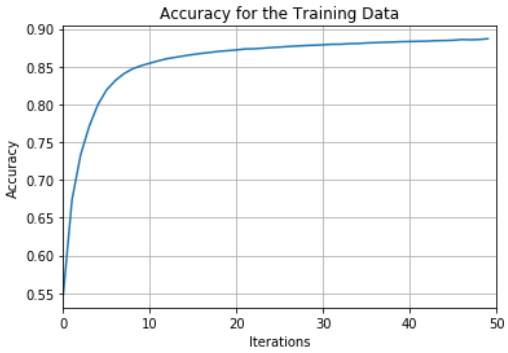

Figure 15: Accuracy of the training data with PCA input into NN

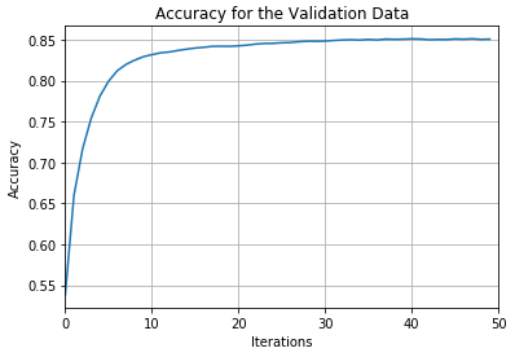

Figure 16: Accuracy of the validation data with PCA input into NN

4.2 Using PCA Transform as a CNN Input

Next, I tried using the PCA transforms as our input layer of our CNNs to hopefully see an improvement. To do this, I needed to specify the number of components to a value that I know can be reshaped into a square (I chose 100). This is required as the CNN expects to receive an image. The results for this, however, were unsuccessful. Overall the performance on the validation set was worse than with the regular NN unlike when we fed the same models the full image. I thought perhaps removing the pooling step would help as we could have been throwing away too much data (it was already only 10x10), however this just resulted in over-fitting where the training accuracy went up to about 90%, but the validation accuracy was still below that of the fully connected NN (83% vs. 85%). This was not completely unexpected however, as the components returned by the PCA are not really equivalent to the pixels of an image. When reshaped as such, neighbours do not actually have a specific elation to each other because the components are simply ordered by decreasing variance. Therefore, the CNN is making inferences with respect to the higher level shape of the "image" which do not have any real meaning.

5 Conclusion

In conclusion, the best model to deal with this problem of digit recognition proved to be the 2 layer convolutional CNN. It ran faster than the model with 1 convolutional layer, and out-performed it in every category. In addition to this, decreasing the number of filters in these models proved to have nearly no affect on the performance while increasing the speed considerably. There did seem to be a decent degree of over-fitting, in that it performs much better on the training set compared to with a regular NN. However, I was not able to correct this over-fitting in a way that resulted in better performance on the validation data.

PCA also proved to be massively useful in how it could represent so much of the image in such a small amount of data. Using the PCA results as an input for the traditional NN not only resulted

in a massive speed increase, but also resulted in better performance due to a better representation of the data. While the performance was still definitely worse than our best CNN, the difference in speed could make it a good option depending on the application. However, these benefits did not carry over to CNNs which performed worse with this input, which is to be expected given the form PCA outputs data in, and the assumptions CNNs work on to better interpret images.

Some future work in this direction would be exploring more options for over-fitting with the CNNs, as well as perhaps adding another layer if these problems are solved. On the PCA end I would have liked to investigate more if it was possible to reach the performance of my CNNs with PCA + NN. Some options include feeding it higher dimensions PCA results as well as increasing the complexity of the network. While increasing the complexity did not help it initially and resulted in over-fitting, perhaps with more full data from PCA this would not be the case.