Machine Learning Engineer Nanodegree

Capstone Project

Paul Heraty
October 18th, 2016

I. Definition

Project Overview

For my capstone project, I will implement a model that can decode sequences of digits from natural images, and print the numbers it sees. I will begin by architecting a model that will use the Street House View Numbers (SHVN).

This project will use supervised Machine Learning techniques to recognize house number sequences in images. For example, for an image that contained the number of a house, the program would be able to determine what the actual house number is. This technology would be very useful for applications like map building from images. There have been other approaches to digit recognition in images previously, where digit sequences are segmented and recognized one by one, e.g. [Qadri, Asif, 2009]. In this project, I will perform the entire digit sequence recognition within the learned model. In particular, I will use techniques such as Deep Learning and Convolutional Neural Networks. I will implement this using Python, NumPy and TensorFlow.

The reason I chose this project is that I am very interested in computer vision applications, in particular object recognition in real time. I have previously worked on implementing low level vision processing kernels in OpenVX, and I am interested in moving up the stack to see how applications do things like object recognition.

Problem Statement

This project will implement a Machine Learning model that can decode sequences of digits from natural images, and print the numbers it sees. By using a labeled dataset of images that have numbers in them and labels that describe the numbers, we can apply supervised machine learning to train a model to recognize these numbers on future images. The labels will be a set of data like [number_of_detected_digits, digit1, digit2, ... digit5], where each digit is in order and in the range of 0-9. The input to the model will be an image that contains numbers, and the outputs will be the number of recognized digits and the individual digits themselves in sequence. As house numbers rarely exceed five digits, I will limit the project to recognizing sequences of up to five numbers.

I will use the SHVN dataset, which is a large scale dataset collected form house numbers in Google Street View. This dataset is challenging to work with, as the digits are not lined up neatly, have skews, different fonts and colors. Also, there are 604,388 training digit images in the dataset, and 226,032 test images. The SHVN dataset is an ideal dataset to use for solving the house number recognition problem, as it uses data from real world house numbers.

I will have to do some pre-processing of the input data in order to maximize the results I can achieve in terms of prediction accuracy. This will include converting the images to gray scale. Color is not a helpful trait when trying to recognize digits, and also reducing the RGB data to gray scale

will reduce the number of features input to the mode which will help with speed. I'll also be cropping the images to the regions of interest, i.e. the numbers themselves, and scaling the images to a standard size of 32x32 pixels.

Metrics

The best metric to use to evaluate my model would be the accuracy versus the labeled SVHN dataset, as I will use this to compare against results previously achieved by Goodfellow et al in their ICLR paper [3] . I will split that dataset into a training set, a cross validation set and a test set. Then, using the test set, I can compare the accuracy of my predictions versus the actual labels to see how effective my model is at predicting the numbers. I can also compare this to the accuracy numbers generated by the Google people at ICLR'14 in their paper “Multi-digit Number Recognition from Street View Imagery using Deep Convolutional Neural Networks”.

Since my classifier will be interpreting the digits individually, I can also look at individual digit recognition rates. However, this is not as important a metric for this project, as we want to be able to extract the full house number correctly for map-making purposes, and getting 'most' of the digits right is not acceptable. As such, the primary evaluation metric will be getting the complete number correct, and looking at individual digit recognition rates will be secondary.

To calculate accuracy, I will be calculating the prediction accuracy of each digit individually, and then calculating an overall accuracy by averaging the sum of the accuracies of each individual digit as well as the accuracy of the number of digits recognized.

Since accuracy as a metric relies on a good distribution of classes in the dataset, which as it happens is not the case in the SVHN dataset, I will also look at some other metrics. For the number of recognized digits and each individual digit, I will examine the Precision, Recall and F1-Scores. Precision is the number of correct positive results divided by the number of all positive results, and Recall is the number of correct positive results divided by the number of positive results that should have been returned. The F1 score is a measure of a test's accuracy that considers both the precision and the recall, and can be interpreted as a weighted average of the precision and recall, where an F_1 score reaches its best value at 1 and worst at 0.

II. Analysis

Data Exploration

I will primarily be using the Google Street View House Number dataset for this project (http://ufldl.stanford.edu/housenumbers). There are two datasets present on this website. The first are original, variable-resolution, color house-number images along with a datafile which has information on individual digit bounding boxes, labels and associated images. The second dataset has individual digits per image, and all digits have been resized to a fixed resolution of 32-by-32 pixels.

Given that I want to recognize multiple digits in a single pass, the second database will not be of any use for this purpose as it only has individual digits in each image. So the database that I will be using is the first one.

However, there are a number of issues with the first database that I will need to address in a data

pre-processing stage. Firstly, the images are or arbitrary size, so I will need to resize the images to a set size. Secondly, the images contain lots of pixels that do not have digits in them, and as such are noise. To counteract this, I decided to use the bounding box information to crop the images so that only the relevant digit information will remain.

There was also a single instance of an image that had 6 digits in it, so I excluded this from the dataset as one sample is not of any use, and I have limited my program to 5 digits maximum.

As an example of what I will do to pre-process the data, let's look at an example. Here is a sample of an original input image:

The first thing I will do is convert the image to gray scale, as color is not useful when trying to recognize digits. Also, by reducing the RGB images to gray scale, I am reducing the amount of input data by a factor of three, and this will make the model run faster.

Next, I will mean center the image. This involves calculating the average pixel value in the image, and subtracting this from every pixel value. This will help average out image brightness, essentially making all the images more alike, thus helping the model when its trying to find similar features.

Next, I will crop the images to the regions of interest. Using the individual digit bounding box information from the .mat file, I calculate a new bounding box that shrink-wraps the digits in the image. I then expand this by 30% in order to capture some of the surrounding context, leading to an image like the following:

I then resize this image to 32x32 pixels, and this will be used as the input features to my network:

Finally, I normalize the pixel data to the range -1.0 to 1.0 by dividing the values by 256.0.

For the labels, I create a list which contains:

[number_of_digits, digit1, digit2, ... ,digit5]

So there will be 6 elements in the labels list in total.

In total, there are 33,402 elements in the SVHN training database, and a further 13,068 in the test database. I further split the training database into training and validation sets of sizes 23,402 and 10,000 respectively.

After all data pre-processing is complete, a typical entry in the dataset looks like the following

dataset = *[32 x 32 pixel x 3 channels (RGB)]*

labels = *[number_of_digits, digit1, digit2, ... ,digit5]*

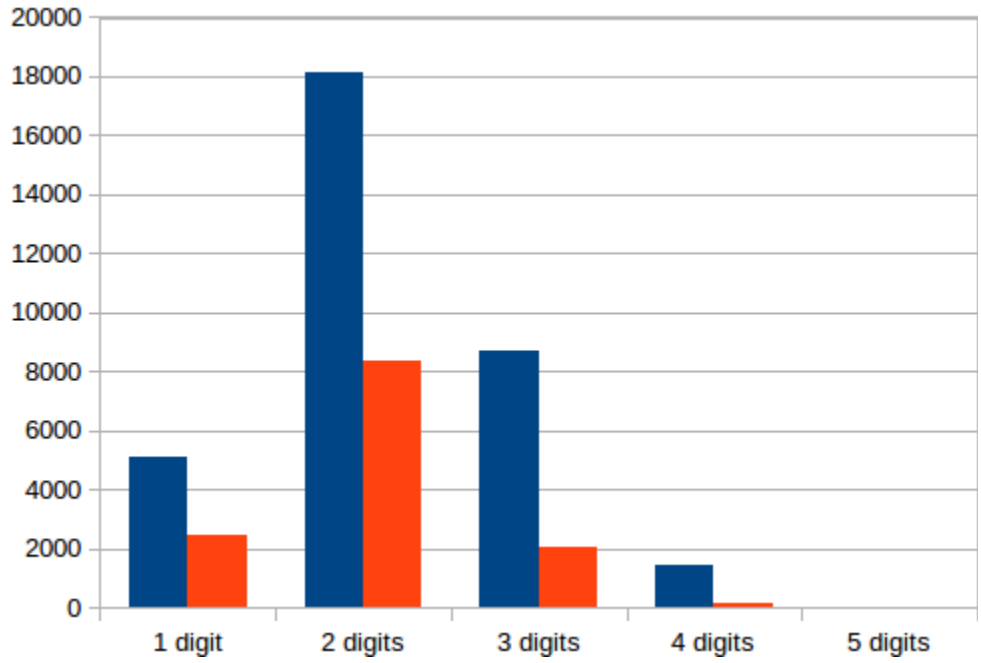
Finally I store these datasets in a pickle file, *svhm.pickle*, with the following names:

```
train_dataset,
train_labels,
valid_dataset,
valid_labels,
test_dataset,
test_labels
```

All of the above code for the above can be found in the preProcess.py file.

Exploratory Visualization

One remaining issue with the dataset is that there is not an equal distribution of samples across cases. Looking at the training and test datasets, you can see that the distribution is heavily skewed towards images with 2 and 3 digits, which likely reflects the real-world situation:



This leads me to worry that the model will be skewed towards recognizing images with 3 or less digits. It may be that I have to limit my program to 1-3 digits, based on the dataset skew. I will revisit this after I have a model up and running, and am able to evaluate the performance of 4& 5 digits cases.

Or I could just logically assume that 5 digits are not going to work at all, as there are only 9 samples in the training set and 2 in the test set. I could proceed to try our 4 digits and sees how well it performs, and remove them also if the performance is not in line with the other digit cases.

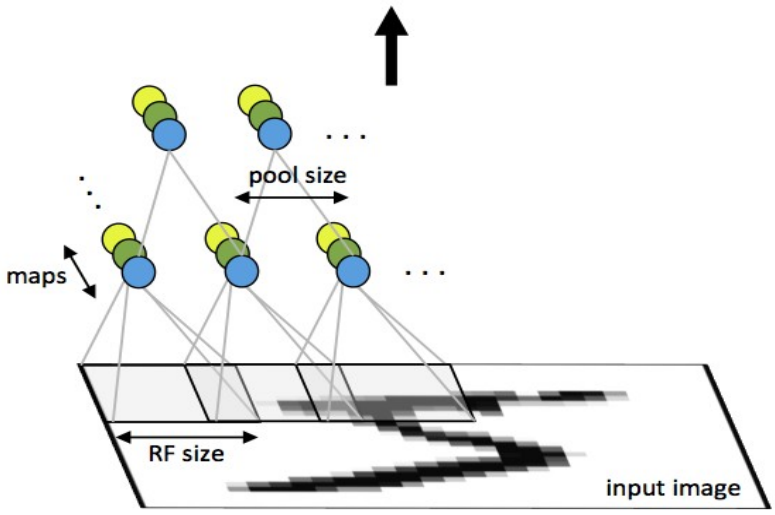
Need some advice here please? Is it acceptable to only do 1-3 digits?

Algorithms and Techniques

I will be using a Convolutional Neural Network (CNN) to create a model which can extract the digit

information from an image. A CNN is a type of feed-forward artificial neural network in which the connectivity pattern between its neurons is inspired by the organization of the animal visual cortex. Individual cortical neurons respond to stimuli in a restricted region of space known as the receptive field. The receptive fields of different neurons partially overlap such that they tile the visual field. The response of an individual neuron to stimuli within its receptive field can be approximated mathematically by a convolution operation. Convolutional networks were inspired by biological processes and are variations of multilayer perceptrons designed to use minimal amounts of preprocessing. They have wide applications in image and video recognition, recommender systems and natural language processing.

A CNN consists of a number of convolutional and subsampling layers optionally followed by fully connected layers. The input to a convolutional layer is a m x m x r image where m is the height and width of the image and r is the number of channels, e.g. an RGB image has r=3. The convolutional layer will have k filters (or kernels) of size n x n x q where n is smaller than the dimension of the image and q can either be the same as the number of channels r or smaller and may vary for each kernel. The size of the filters gives rise to the locally connected structure which are each convolved with the image to produce k feature maps of size $m-n+1$. Each map is then subsampled typically with mean or max pooling over p x p contiguous regions where p ranges between 2 for small images (e.g. MNIST) and is usually not more than 5 for larger inputs. Either before or after the subsampling layer an additive bias and sigmoidal nonlinearity is applied to each feature map. The figure below illustrates a full layer in a CNN consisting of convolutional and subsampling sublayers. Units of the same color have tied weights.



First layer of a convolutional neural network with pooling. Units of the same color have tied weights and units of different color represent different filter maps.

After the convolutional layers there may be any number of fully connected layers. The densely connected layers are identical to the layers in a standard multilayer neural network.

Convolutional neural networks are often used in image recognition systems. The ImageNet Large Scale Visual Recognition Challenge (ILSVRC) is a benchmark in object classification and detection, with millions of images and hundreds of object classes. In recent ILSVRCs, almost every

highly ranked team used CNN as their basic framework. As such, I chose to use a CNN as the foundational model to solve this challenge.

In order to train a CNN model to learn, I will be using Stochastic gradient descent (SGD). This is a stochastic approximation of the gradient descent optimization method for minimizing an objective function that is written as a sum of differentiable functions. In other words, SGD tries to find minimums by iteration. The objective function that I will be minimizing will be a loss function composed of the error between predicted classes and the actual labeled classes for the digits.

I will also use the Adam Optimizer. ADAM (Adaptive Moment Estimation) is a SGD algorithm in which the gradient used in each iteration is updated from the previous using a technique based in momenta. It tends to converge a lot faster than the standard GradientDecentOptimizer, and so I will be using it in this project.

I will also be using techniques such as regularization and dropout to help tune the mode and prevent overfitting. When the model fits the training data but does not have a good predicting performance and generalization power, we have an overfitting problem. Regularization is a technique used to avoid this overfitting problem.

Dropout is a regularization technique for reducing overfitting in neural networks by preventing complex co-adaptations on training data. It is a very efficient way of performing model averaging with neural networks. The term "dropout" refers to dropping out units (both hidden and visible) in a neural network.

Benchmark

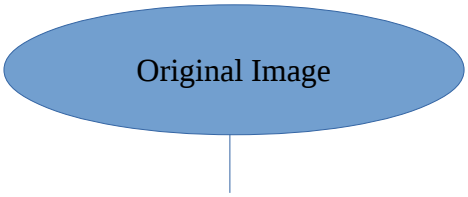
There is a previous solution to this problem posted at http://static.googleusercontent.com/media/research.google.com/en//pubs/archive/42241.pdf which I can use as a benchmark. This was presented at ICLR'14 by Goodfellow, Bulatov, Ibarz, Arnoud and Shet. They are achieving 96% accuracy using the SVHN dataset, and I can use this to compare my model accuracy to.

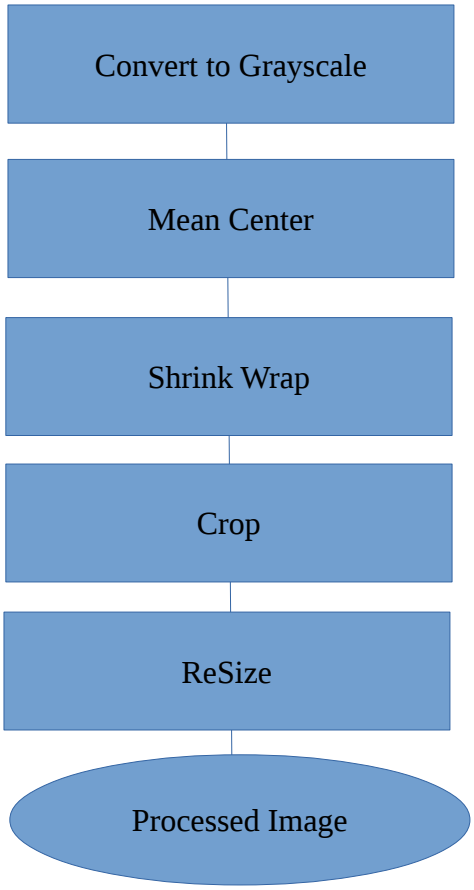
Goodfellow also uses a benchmark of 'human accuracy' in his paper, where coverage of 98% or better is required for map-making purposes, since this matches the accuracy obtained by human operators. They also achieve character-level accuracy of 97.84%, which I can use to compare against my solution.

III. Methodology

Data Preprocessing

The pipeline that I followed for data pre-processing looks as follows:





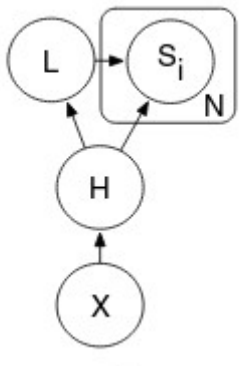
I take in the original image and immediately convert it to gray scale as color is not useful in this model. I then Mean Center the image. Using the individual digits coordinates form the original data, I then calculate a bounding box which 'Shrink Wraps' the digits. I scale this bounding box up by 30% in order to include some of the surrounding image data, and crop the image to this bounding box. I then resize this to 32x32 pixels, so that all input images will be of the same size. This resized image is what I use as input to my model.

Implementation

I will build a model using a CNN followed by a fully connected stage in order to correctly recognize single characters from the SVHN dataset.

I will explore different architectures; numbers and types of hidden layers, numbers of kernels for CNNs, parameters, and techniques such as dropout and regularization for tuning performance.

The overall architecture will look as follows



Here, X is the input image, and the number to be identified is a sequence of digits, S = s1, s2, ..., sN. Each si value can have 10 possible values, 0 through 9. For this project n will be 5, which is the maximum number of digits that can be recognized. H above represents the deep learning model. L is also an output which shows how many digits were recognized in the image. It can have only 7 values, 0 to 5 and a value that represents 'more then 5'. If L is less than N, then only s1..sL will have valid values.

I will test out techniques such as dropout and regularization in order to rune the performance of the model. Both can be very helpful to help prevent overfitting.

Refinement

As a first step to explore the dataset, I implemented a single layer neural network. This can be run by setting graph_num to 1 in recognizeDigits.pl. This graph has a very simple topology, with 3,072 inputs which represent an input image of 32 pixels * 32 pixels * 3 channels. The outputs are a little more complex. There are 6 different softmax outputs, and each are one shot encoded. The first output represents the number of digits recognized in the image, and the remain 5 are the digits themselves. The number of digits is a number between 1 and 5, and each digit can be in the range of 0 to 9. This means my combined output label list is of length 55; 5 entries for the number of digits and 5 digits of 10 entries each.

After training this network, a few things became immediately apparent. First and foremost, even for a single later neural network, it's pretty slow. It took 2,770 seconds on my quad core CPU to train. In order to address this, I will reduce the input datasize by converting the images to rescale. This will reduce the input data size from 3,072 to 1,024.

Secondly, as I guessed earlier, the lack of data for 4 and 5 digits makes it impossible to train a model to predict those digits. So I will remove all items with 4 and 5 digits from the database, and only run prediction on digits 1 to 3.

The accuracy scores from this first network, after 1000 training cycles, were as follows

Num Digits	Digit 1	Digit 2	Digit 3	Digit 4	Digit 5	Overall
19.4%	17.2%	9.2%	85.2%	88.3%	99.9%	0.2%

The second network I tested , recognizeDigits1.py, was also a single later NN, but this time the dataset is reduced to gray scale and also restricted to detecting images that have between 1 and 3 digits in the images. This now runs much faster, taking 758 seconds to train 1000 cycles.

The accuracy from this second run looks like the following

Num Digits	Digit 1	Digit 2	Digit 3	Overall
64.6%	9.4%	9.1%	85.9%	0.4%

I'm guessing at this stage that the network architecture is too simple to detect all items correctly. So my next step is to add a couple of hidden layers.

For this third network, the input data remains the same as the second network, but I'm going to use

stochastic gradient descent to train the network in order to speed up the process. This network, recognizeDigits2.py, now runs in 5.2 seconds, and has the following accuracy:

Num Digits	Digit 1	Digit 2	Digit 3	Overall
64.6%	27.3%	19.4%	86.0%	2.9%

Things appear to be moving in the right direction!

For the fourth network, the input data remains the same again, but I'm going to add a single hidden layer with the hope that increasing the network complexity will lead to an increase in accuracy scores. This network, recognizeDigits3.py, will have a hidden layer of 1024 a depth. I'm picking this numbers to match the size of the input feature array.

This network trains in 105 seconds and showed the following accuracy:

Num Digits	Digit 1	Digit 2	Digit 3	Overall
64.3%	28.6%	28.9%	84.4%	4.5%

So let's add another layer of the same size and see how that affects things. Training with 2 hidden layers caused some new issues. The first was that the loss was going to Nan pretty quickly. I addressed this in two ways. First, I used Xavier initialization of the weight matrices. Secondly, I also had to lower the learning rate from 0.5 to 0.001. This obviously impacted the rate at which the model would learn, so I had to run over many more cycles to get a comparative number. However, even with this learning rate, the minibatch loss went to Nan after a few thousand iterations. This network took 15,494 seconds to run and the accuracy looked like:

Num Digits	Digit 1	Digit 2	Digit 3	Overall
16.1%	58.9%	29.7%	86.0%	2.4%

I then added dropout and regularization to this network (recognizeDigits4.pl) and ran for 1000 cycles. This took 163 seconds and had the following accuracy

Num Digits	Digit 1	Digit 2	Digit 3	Overall
64.6%	27.7%	29.8%	86.0%	4.6%

What I think is happening is this: the first and second digit recognition rates are low due to the fact that they can be in various places in the image. For example, in a single digit image, digit one is centered. In a two digit image, digit one is typically in the left half of the image. And in a 3 digit image, digit one is typically in the left third. Whereas for digit 3, it will always be in the right third of the image, and so the recognition rate is high. That's my theory, and so let's try adding a few convolution layers at the start of our network to address this.

In recognizeDigits5.pl, I added a 2 layer CNN at the start of the model. I also added max pooling between each layer. The input data needs to be formatted differently to work with a CNN, so now it's a 4 dimensional array of the form [sampleNumber, pixlesX, pixelsY, channels]. When I train

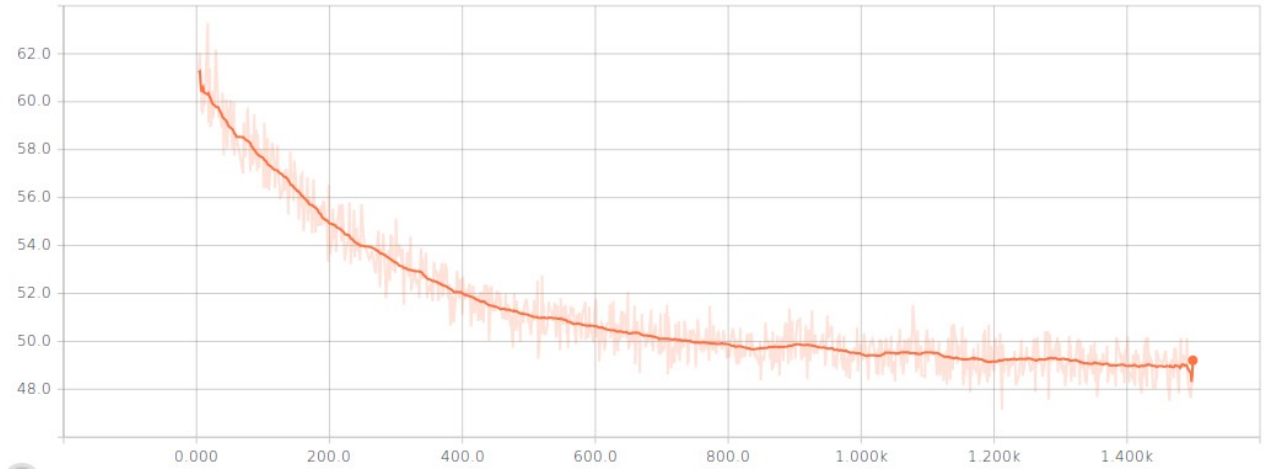
this network over 1000 epochs, it takes 469 seconds, and I get the following accuracy:

Num Digits	Digit 1	Digit 2	Digit 3	Overall
64.6%	27.7%	29.8%	86.0%	4.6%

It's identical to the previous network, except it takes longer to train. Something must be wrong!

Aside: I changed my overall accuracy to be the average of the sum instead of the product of all digits and number of digits.

I set the learning rate to 0.00005, as I was seeing divergence in the losses, and then ran for 10,000 epochs. After 1,500 epochs it seems we maxed out at around 46% validation accuracy with a loss of around 49.



The final accuracies were:

Num Digits	Digit 1	Digit 2	Digit 3	Overall
64.6%	27.7%	9.7%	86.0%	47.0%

I'm still not convinced that the CNN piece is adding anything to the model... running without the CNN piece gives:

Num Digits	Digit 1	Digit 2	Digit 3	Overall
64.6%	27.7%	29.7%	86.0%	52.0%

I believe I have an issue with training, where the model is trying to predict digits 2 and 3, even when only one digit is in the image. I'd imagine this is causing the model to train incorrectly. The label will be '0' for digits 2 & 3 when there is only one digit, so the weights will be getting modified to try to predict a '0' when it really should not be doing anything.

In order to address the latter problem, I extended the number of possible digit labels from 10 to 11,

and modified the format() function accordingly. Now each digit label will be of the format

[NoDigit, 1, 2, ...10]

where NoDigit means there is no digit and 10 is the digit 0. So a full one-shot-encoded label for the number '20' would look like:

```
[0, 1, 0,                          # Two digits
 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0,  # First digit is 2
 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1,  # Second digits is 0
 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]  # No third digit
```

I hoped this would make a noticeable difference to the recognition rates, however after 1500 epochs, I got:

Num Digits	Digit 1	Digit 2	Digit 3	Overall
64.6%	27.7%	19.2%	83.8%	48.8%

Hrmmm. The loss had converged to 51 after about 900 epochs, so I don't think further training would help much.

I decided to try a different optimizer, so instead of the GradientDescentOptimizer, I decided to try the AdamOptimizer. Running the same network for 1500 epochs gave:

Num Digits	Digit 1	Digit 2	Digit 3	Overall
64.6%	27.7%	8.7%	83.8%	46.2%

It does get the loss down to 40, and it appears to converge a little faster too. But the results are no better, in fact a little worse.

Let's try some different learning rates now. I had been using 0.000005, as I had earlier problems with the loss increasing to Nan. However, now I see that when I increased the learning rate to 0.00005 on my current network, after 1500 epochs, it got the loss down to 25.8 and had an overall accuracy of 48.8%. Increasing the learning rate even further to 0.005 and decaying the learning rate over time gave a final loss of 7.5, with an overall accuracy of 50.4%:

Num Digits	Digit 1	Digit 2	Digit 3	Overall
67.5%	28.7%	21.5%	83.8%	50.4%

Next step is to add more layers to the CNN. Let's try adding one more layer. I added just the conv2d and relu to the start of the graph, no pooling. It took 7,332 seconds to train, roughly 2 hours, which was ~16x slower than before. Wow. My machine was effectively hung during this time, so I think I'm gonna have issues when adding more layers. The final loss was 6.6 and the accuracy improved to

Num Digits	Digit 1	Digit 2	Digit 3	Overall
85.4%	51.5%	49.2%	85.4%	67.9%

At this point, I decided to try a few more tests. I was concerned that my loss was still pretty high, at

6.6. So I ran some tests using my network with a smaller sample size of 130 samples, in order to help me tune my hyper-parameters. I turned off regularization and dropout, and found that a learning rate of 0.001 work well and the network quickly converge to a very small loss of 0.001. This gave me an indication that the network architecture and coding was ok, since it was able to fit to a small sample of the images.

I did note that multiple runs of the same network gave different results. A assume this is due to two reasons. Firstly the network has not converged, as shown by the high loss, so it's incomplete. And secondly, I assume the randomness is due to the random initialization of the network weights and biases, in tandem with the fact that the network has not converged.

I then switched back to the full training database size of 23,968 images and noticed that after 1500 epochs, my loss was around 2.2. But it was still shrinking at this stage, so I upped the number of training epochs to 2500. This gave me a final loss of 0.66 after 2,796 seconds with the following test accuracy:

Num Digits	Digit 1	Digit 2	Digit 3	Overall
91.5%	73.4%	69.4%	89.8%	81.0%

Not bad! I even ran a single inference on a test image and it correctly identified that there was a single digit and that the digit was 5!

And this is with dropout and regularization turned off. Also, the loss still appeared to be dropping so I ran again with 3500 epochs with dropout and regularization on. That ran in 5154 seconds, but slightly reduced the accuracy:

Num Digits	Digit 1	Digit 2	Digit 3	Overall
91.1%	70.9%	65.8%	88.2%	79.0%

So either regularization and/or dropout is not particularly helping. The final loss after 3500 epochs was 4.88, with it being 5.5 after 2500 iterations. So it looks like the extra epochs were still moving things in the right direction. However, the loss was still decreasing, so perhaps I can use a slightly higher learning rate.

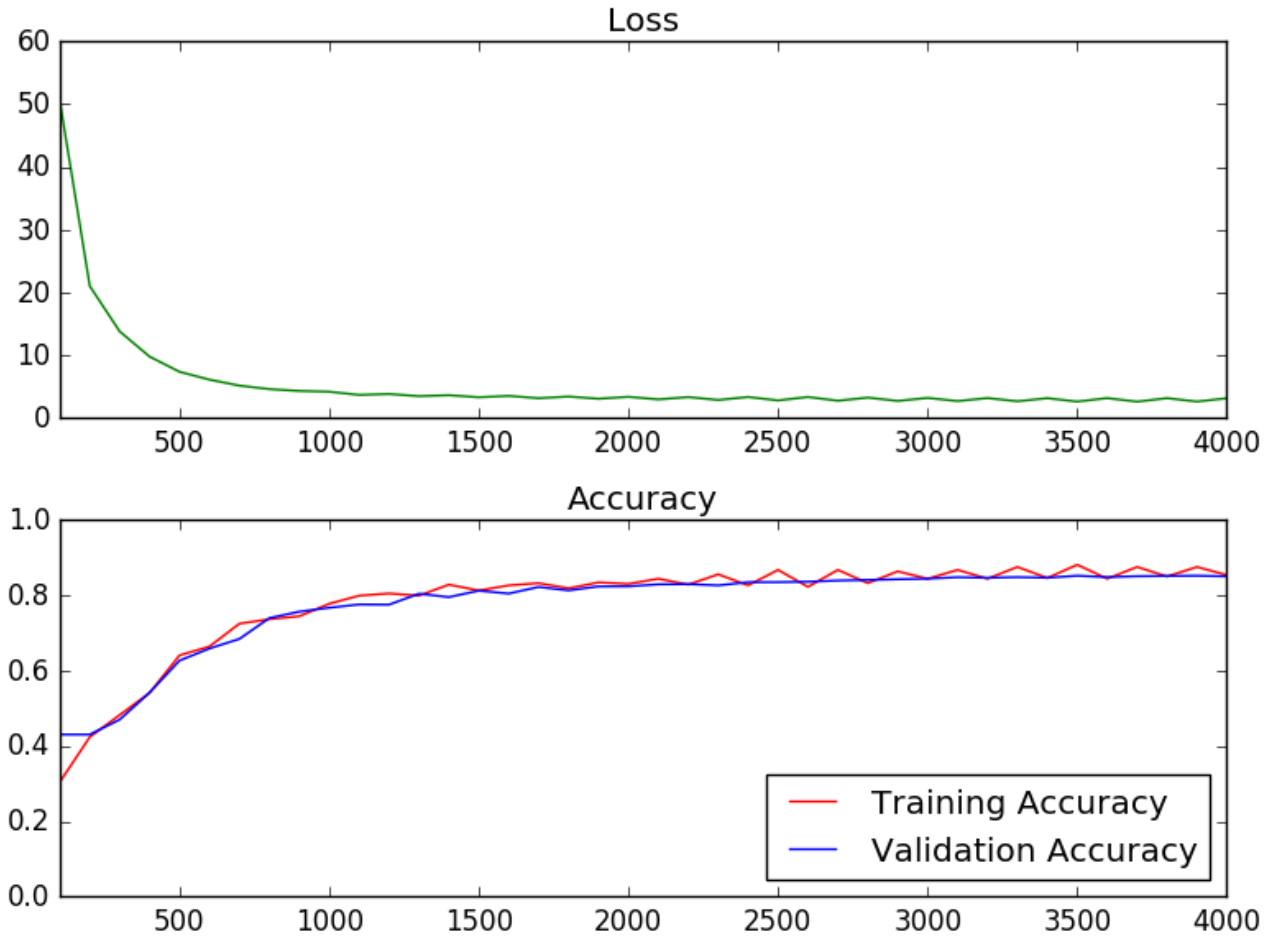
Lets turn off dropout and up the learning rate to 0.002. At this stage, I also made a lot of changes to the recogniseDigits.py file to improve performance, add log files, and also change how the runtime iterated over epochs.

I also modified the preProcess2.py file to find each input image mean pixel value (after conversion to gray scale), and then subtract this from each pixel value. This should have the effect of mean centering the image, which should help normalize things like lighting conditions across the multiple images, making them easier for the learning process to manage as they will be more 'alike'.

Now, after running over 20 epochs, I get a loss of 2.99 after 2,866 seconds. I get the following test accuracy:

Num Digits	Digit 1	Digit 2	Digit 3	Overall
93.1%	79.9%	75.4%	91.5%	85.0%

The plot of loss and accuracy (batch and validation set) over time looks like:



When I run on a test image with a single digit '5', here's what I get as the predictions:

[[**5.16639805**, 1.01836765, -2.37923241]
[[-4.40307474, -0.97034192, -0.20101976, 2.39751792, 0.74679005,
4.42687225, 3.31125593, -0.36889815, 0.97152537, 0.92122066,
-1.0388298]]
[[**8.28252602**, 3.00666666, -1.08322525, 1.00289452, -1.20612705,
1.982723 , -1.40155029, 2.62438297, -1.63069427, -0.24124342,
-0.61320817]]
[[**8.94338226**, 1.07958436, 0.52324212, 0.40189534, -0.12975776,
-0.58580232, 0.29284585, -0.73762536, 0.65268481, -0.11255205,
0.62284458]]

The first set shows that it decisively thinks that there is one single digit. It also predicts the highest likelihood that the digit is '5', and that there are no second and third digits in the image.

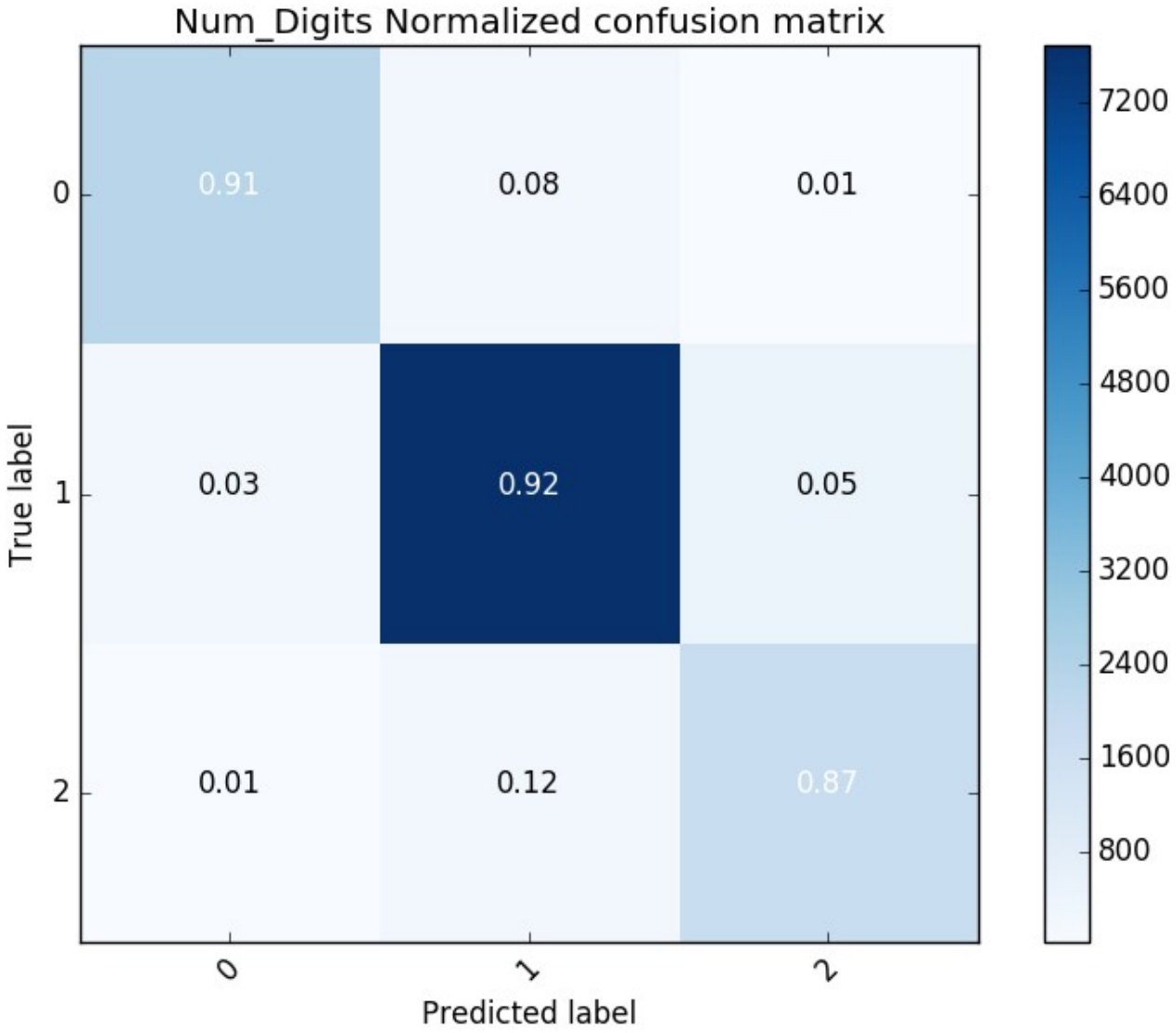
IV. Results

Model Evaluation and Validation

As can been seen above, I end up with a model that has an 85% accuracy rate. Given that I am running on CPU (not GPU), I am somewhat limited in how many layers I can add to the model while still having reasonable run times. All in all, I'm quite happy with the accuracy achieved.

Let's look at how each part of the model is performing individually.

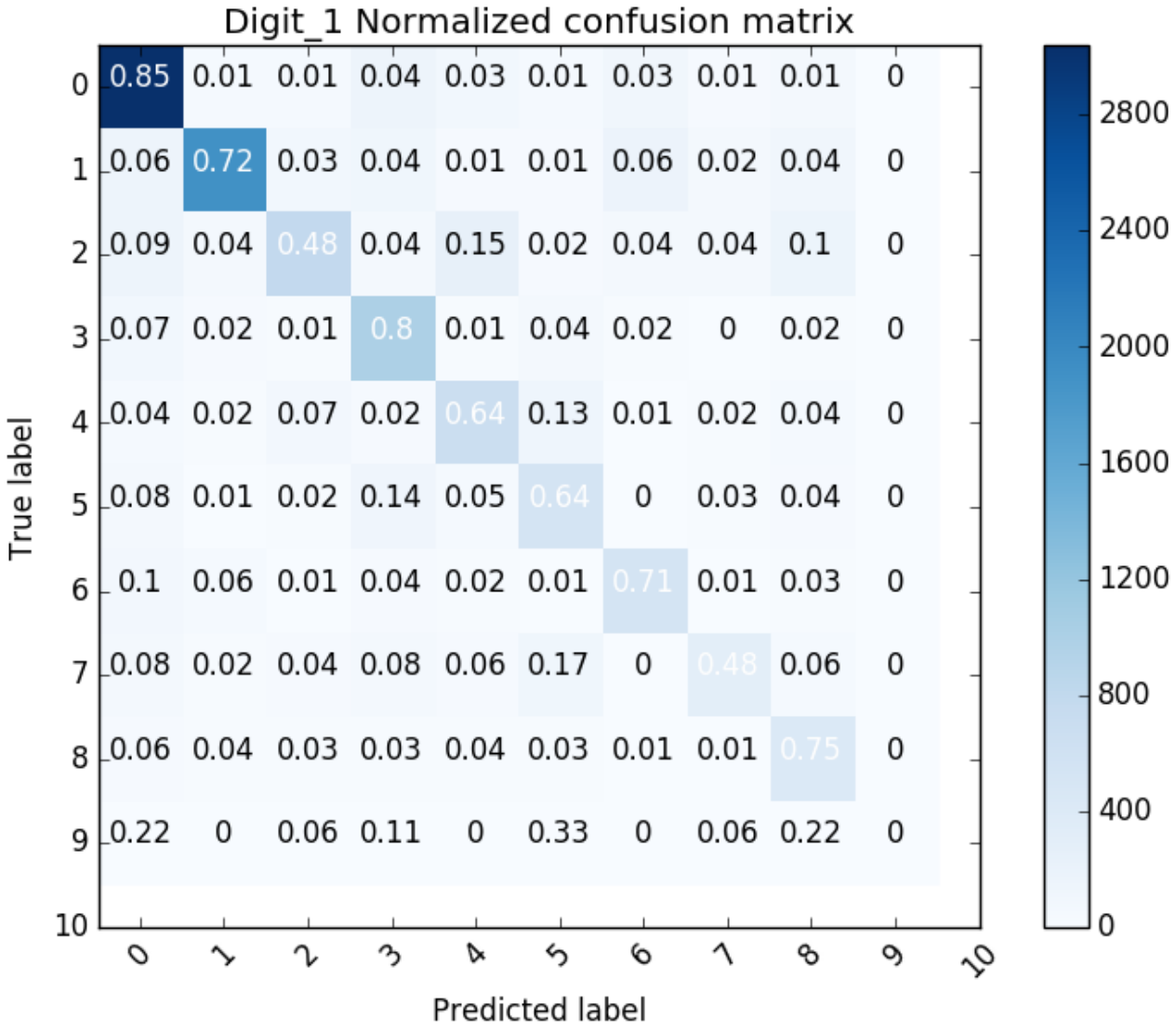
For the Number of Digits, I get the following confusion matrix:



And here is what the Precision, Recall and F1 scores look like:

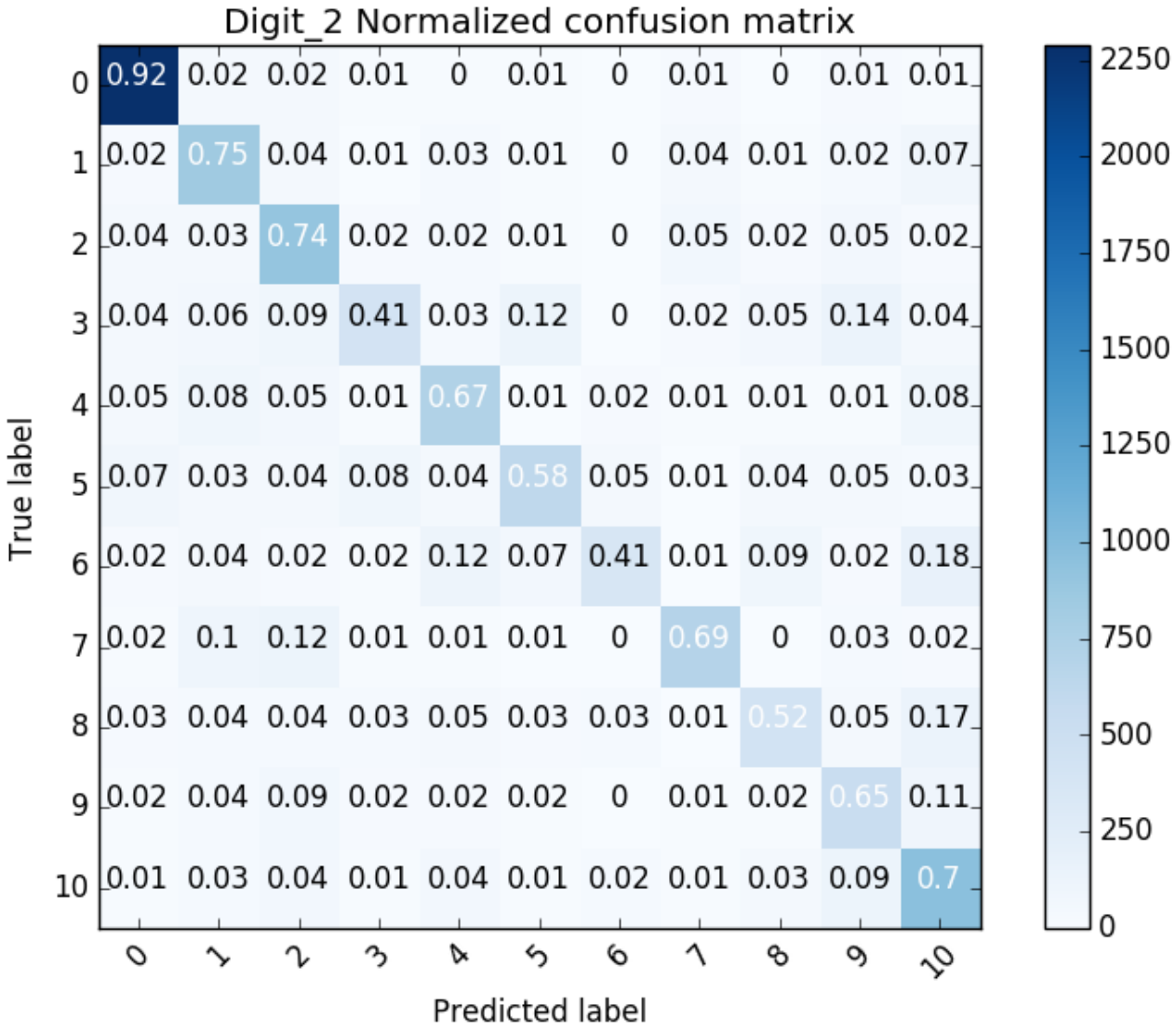
Numdigits	Precision	Recall	F1-Score	Support
0	0.90	0.91	0.90	2483
1	0.95	0.92	0.93	8356
2	0.80	0.87	0.83	2081
Avg/Total	0.91	0.91	0.91	12920

For Digit 1:



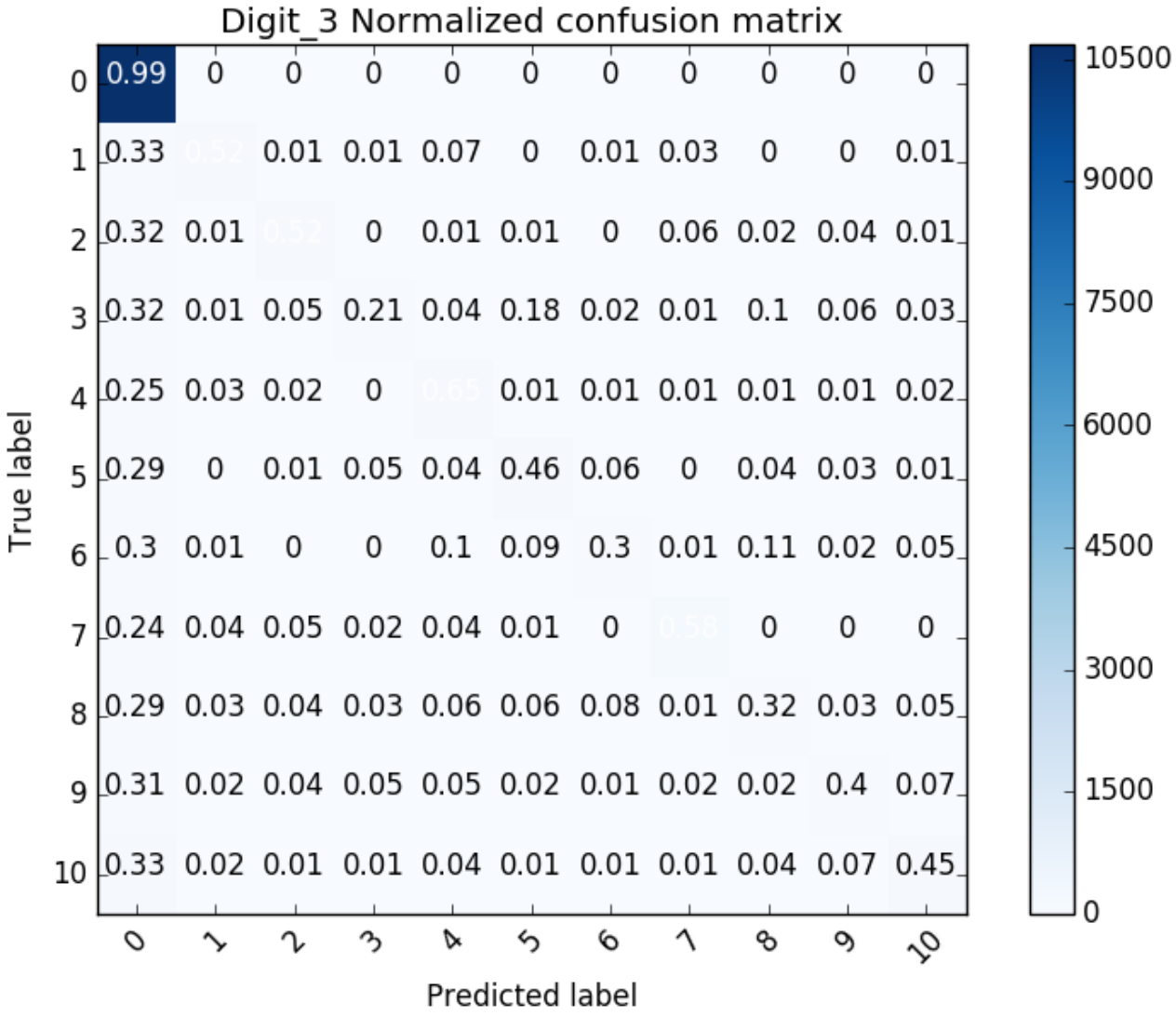
Digit 1	Precision	Recall	F1-Score	Support
No Digit	0.00	0.00	0.00	0
1	0.82	0.85	0.83	3585
2	0.89	0.72	0.80	2632
3	0.73	0.48	0.58	1621
4	0.63	0.80	0.71	1232
5	0.58	0.64	0.61	1052
6	0.56	0.64	0.69	855
7	0.59	0.71	0.64	745
8	0.59	0.48	0.53	624
9	0.47	0.75	0.58	556
0	0.00	0.00	0.00	18
Avg/Total	0.73	0.71	0.71	12920

Digit 2



Digit 2	Precision	Recall	F1-Score	Support
No Digit	0.88	0.92	0.90	2483
1	0.63	0.75	0.68	1127
2	0.61	0.74	0.67	1223
3	0.65	0.41	0.50	1018
4	0.65	0.67	0.66	1072
5	0.68	0.58	0.63	1069
6	0.73	0.41	0.52	910
7	0.77	0.69	0.73	1013
8	0.60	0.52	0.56	818
9	0.50	0.65	0.57	817
0	0.59	0.70	0.64	1370
Avg/Total	0.69	0.68	0.68	12920

And Digit 3



Digit 3	Precision	Recall	F1-score	Support
No Digit	0.94	0.99	0.96	10839
1	0.68	0.52	0.59	229
2	0.66	0.52	0.58	228
3	0.44	0.21	0.28	199
4	0.50	0.65	0.57	185
5	0.54	0.46	0.50	232
6	0.53	0.30	0.38	172
7	0.74	0.58	0.65	221
8	0.41	0.32	0.36	179
9	0.51	0.40	0.44	162
0	0.67	0.45	0.54	274
Avg/Total	0.89	0.90	0.89	12920

The above data shows some interesting insights. Because the data come from real house numbers, there are some patterns that we can expect, but may not foresee. Let's look at Digit 1 for example. From the precision/recall table, we can see that the data distribution is heavily skewed towards the lower digits. For example, there are 3585 samples which have the number '1', but only 556 that have the number '9'. If you think about it, it more likely in the real world that houses with two digits will begin with a lower number, i.e. there are going to be more houses with the numbers '1x' as opposed to '9x', as many housing estates will not have 90+ houses. The effect of this is that we get better F1-scores for the lower numbers of Digit 1, as there are more training samples. Likewise, for Digits 2 & 3, there are many more images where there are 'No Digits', leading to lower F1-Scores for individual numbers 1-0.

Also, in digit1 we can see that there are 18 labeled cases of Digit1 being '0'. This should never be the case, as you should not have a house number '012' for example. This leads me to think that there may be some incorrectly labeled images in the dataset, which would be confusing to the model during training, and hence lead to lower accuracies.

Justification

In the paper by Goodfellow [3], they achieved a final accuracy metric of 97.84%. My model achieves between 80-85%, as subsequent runs produce slightly differing results. Using my maximum accuracy of 85%, I believe I compare favorably to the Goodfellow results for a couple of reasons.

Firstly, I am limited in the compute resources at my disposal. I am training my model on CPU only and using a single laptop with a quad core CPU. This imposes restrictions on the complexity of the model that I can train. Goodfellow used an eight layer CNN, whereas my model is using just a three layer CNN.

Secondly, Goodfellow trained his model for six days using 10 replicas on DistBelief. DistBelief is a Google software framework that can utilize computing clusters with thousands of machines to train large models. My model trained on a quad-core CPU in 2,866 seconds, or ~48 minutes.

V. Conclusion

Free-Form Visualization

I believe I have addressed this section above in the final parts of Model Evaluation and Validation.

Reflection

I must say that I found this whole problem extremely interesting and challenging. It was a huge jump to go from the MLND material into a problem of this magnitude. I had to go 'offline' to really learn how CNNs and Neural Networks worked, and I used the CS231N lectures on YouTube for this purpose. They gave me a good insight into how CNNs and NNs really work. The Udacity material on these topics is way to light and high-level to be of any use.

One area that I am still not clear on is how to determine the best architecture for both the CNN and NN portions, i.e. what depths of CNN to extract, how many layers is 'enough', how wide the NN layers should be etc. The values I used were values that I had seen in the nonMNIST training section of the course. It happened that these gave acceptable results, but I've no way of knowing if

they are too little, too much, etc. Perhaps this is just something you learn by doing testing?

The solution I deployed is pretty complex, more than I though would be needed when I first started on this challenge. Up front, there was a lot of work to be done on pre-processing the data to get it into a usable format for a machine learning model. This involved converting to gray scale,mean centering the image, cropping the image to the relevant portion, and then resizing this to a common 32x32 pixel size so that all input images will be of the same size.

From there, I had to architect a CNN/NN network that would be able to learn from the input data. This was challenging for a number of reasons. Firstly, since there were multiple predictions happening in parallel, the outputs of the model were more complex that models we had trained in the class material. Secondly, the challenge of determining multiple objects in one pass meant that the network would be more complex than others I had used. It took a lot of effort to end up with the network architecture, and in particular the hyper-parameters to use. This really was the most beneficial part of the exercise, as you can really only learn this stuff by using a real example and figuring out what's working (or broken) as you go.

I also got to learn a lot about TensorFlow, which I would not have learned otherwise. You really only learn something like this by applying a real problem to it, and I've got to learn many tricks and good routines along the way. I feel I have a good base solution now which I can apply to future challenges.

Another eye-opener for me was how important data pre-processing is. I guess the old saying of 'garbage in, garbage out' hold true for Machine Learning also :) It really is worth spending time making sure you understand the data, and that it is in a good format for modeling before you start training models on it.

Finally, I noticed that there is variability in my model. If I run the training process multiple times, I get final accuracies that range from 80% to 85%. I'm guessing this is due to how the weights are randomly initialized, as my losses are never identical. Perhaps if I trained for longer periods, this would all converge to a common accuracy.

Improvement

If I were to look to achieve higher accuracies for this model, the first thing I would look to do would be to add more convolution layers. This would enable the model to extract deeper features and learn more from the image dataset. For this, I believe I would need to change hardware to a machine with more memory and also with an Nvidia GPU in order to accelerate the training time.

I also would look to generate more data samples. One approach I would take would be to synthetically generate more samples of house number for cases where the existing dataset has fewer. By looking at the number of support cases in the F1-score tables, I could identify cases where we have fewer samples. I could then take the existing samples for these cases, and use OpenCV to generate new samples by transforming or skewing the existing images. This would help round of the distribution of samples, and lead to higher accuracies for the cases where we currently have least samples, in turn leading to a higher overall accuracy

---

References

[LeCun et al., 1998]

Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, november 1998

[Qadri, Asif, 2009]

Muhammad Tahir Qadri, Muhammad Asif. Automatic Number Plate Recognition System for Vehicle Identification Using Optical Character Recognition. *ICETC*, April 2009

[Goodfellow et al. 2014]

Ian J. Goodfellow, Yaroslav Bulatov, Julian Ibarz, Sacha Arnoud, Vinay Shet Multi-digit Number Recognition from Street View Imagery using Deep Convolutional Neural Networks. ICLR 2014