Capstone Project

Machine Learning Engineer Nanodegree

Daliso Zuze
November 1st, 2016

Definition

Project Overview

With the introduction of Google Street View, a large collection of data has been made available for improving the accuracy of maps. The street view images taken of buildings can be analysed to extract house numbers which can be included in maps. This image analysis was done manually before the development of automated methods. The manual process is slow and expensive, limiting the quantity of data that can extracted from these images. With recent advances in automated methods, much larger volumes of Google Street View images can be used in map making.

See Fig 1 below for a sample of the house number images that are captured by Google Street View.

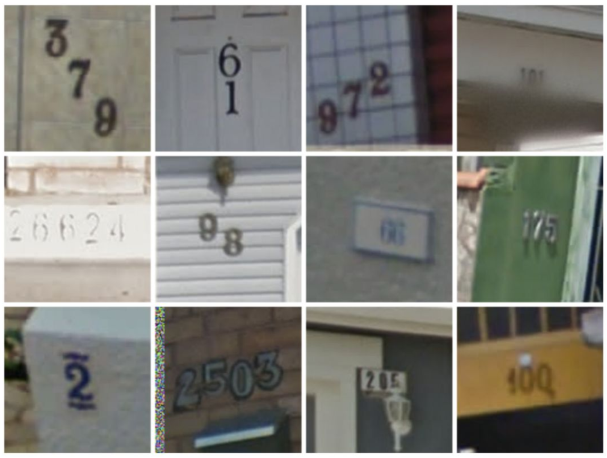

Fig 1: Examples of house number images from Google Streetview[1]

As can be seen from the example images above, there is a variation in quality of images, fonts, positioning of numbers which pose challenges that must be overcome in the computational approach to reading these house numbers accurately.

For this capstone project, I will be leveraging a deep learning architecture to process and recognise the number sequences in the publicly available SVHN dataset[2].

The Street View House Numbers (SVHN) dataset is a dataset of about 200k street numbers, along with bounding boxes for individual digits, giving about 600k digits total.

[1] Images from *Multi-digit Number Recognition from Street View Imagery using Deep Convolutional Neural Networks*
[2] http://ufldl.stanford.edu/housenumbers/

Problem Statement

The problem of identifying characters is similar to the character recognition problem from the notMNIST dataset tackled as part the of the Udacity Deep Learning course. It is however more complicated in that here we are attempting to classify sequences of digits. It is further complicated by the variety of styles of the images and image quality since these are images taken from the real world where factors such as shadows and wear-and-tear of signs come into play.

In this capstone project, I will develop a solution to the problem of multi-character digit recognition by training a deep convolutional neural networking using the Google Street View images dataset.

I will be basing my approach on the architecture demonstrated by GoodFellow, et Al in the paper *Multi-digit Number Recognition from Street View Imagery using Deep Convolutional Neural Networks*[3]. In this paper, they describe how traditional approaches to solve this problem typically separate out the localisation, segmentation, and recognition steps, however they propose a unified approach that integrates these three steps via the use of a deep convolutional neural network that operates directly on the image pixels.

My solution will be able to classify numbers up to 5 digits since most house numbers are within the range 1-5 digits. Given an image with a number that is 1-5 digits long, my solution will output an integer of 1-5 digits corresponding to what it predicts the number in the image to be.

For this capstone project, the deliverables that I will be producing are:

1. Project Report (this document)
2. Source Code
3. README for the source code

My approach will be as follows:
1. Collect the raw Google Street View House Number dataset
2. Do preliminary analysis to determine the preprocessing necessary.
3. Do the preprocessing needed, and split the datasets into training, validation and test sets
4. Generate a synthetic dataset for developing the first version of the model
5. Construct the deep convolutional network based on the architecture proposed by GoodFellow et Al [3]
6. Fine tune the model
7. Test the effectiveness of the model using new images from my environment

Metrics

The metric that I will use to measure the performance of my trained model is ‘accuracy’. In order to be considered accurate, each digit in the house number needs to be identified correctly. There is no partial accuracy for getting some of the digits in the house number correct. This is justified because an incorrect number can completely lead a map user astray, for example 915 vs 115 are likely to be in very different places geographically.

Accuracy = (number of exact matches)/(number of test samples)

Analysis

Data Exploration

The street view house number dataset comes in two formats:
1. Original images with character level bounding boxes.
2. MNIST-like 32-by-32 images centered around a single character (many of the images do contain some distractors at the sides).

I used the first format, since this provides training images in the real life context in which I will be testing the final model.

There are a total of 46,470 images split into a train set and a test set:

- Train set = 33,402 images; Test set = 13,068 images

Please see fig 2 below for examples. Note that the bounding boxes are not actually on the images, instead the train and test sets each come with a data file that contains the bounding box information for each digit in each image.

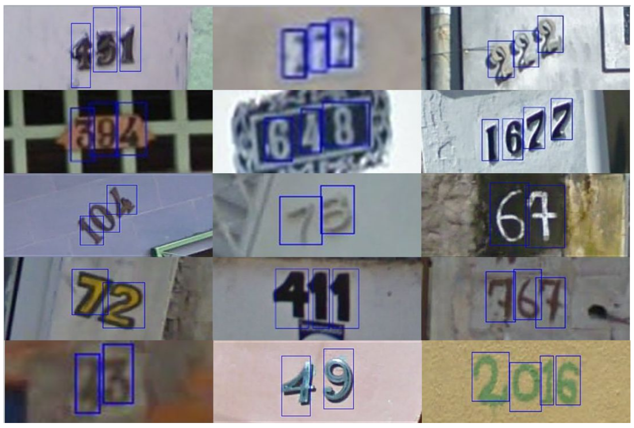

Fig 2: Examples of images from the Street View House Number dataset

It should be noted that the dataset does not come with labels in the form of an integer for the house number, instead we are provided information on each individual digit. For example in the first house number above, we are provided with the information that the image has three digits 4, 5 and 1 along with the bounding box information (Top, Left, Width, Height) for each digit in the image. Therefore, as part of pre-processing, the label 451 must be computed assuming that the number runs from left to right.

For the digits in the source data, there are 10 classes, 1 for each digit. Digit '1' has label 1, '9' has label 9 and '0' has label 10.

There are 73257 digits for training, 26032 digits for testing, and 531131 additional, somewhat less difficult samples, to use as extra training data[3]. I did not use the extra images, only the training and testing sets provided.

It should be noted also that the labels provided for the digits and the bounding boxes are not perfectly accurate. This dataset abnormality is possibly due to some human error that crept into the manual labelling process.

[3] As per the information obtained from http://ufldl.stanford.edu/housenumbers/

Exploratory Visualization

An important characteristic of the dataset is the distribution of the number of digits. From the chart below, we see that most house numbers have two digits with very few having 4 or 5 digits. Interestingly, 88 out of the 46,470 house numbers had 6 or 7 digits. There were no house numbers with more than 7 digits.

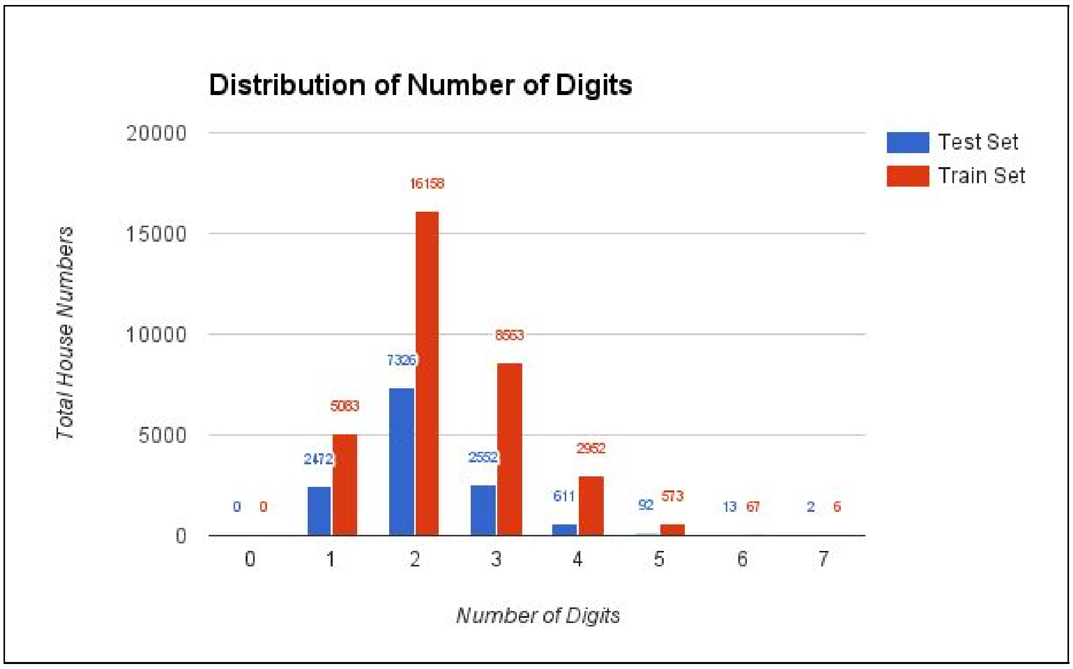

Fig 3: Distribution of Number of Digits on the Street View House Number Dataset

Furthermore, when one explores the distribution of the number of samples for each digit, we see that the vast majority is the digit 1 and the digits 0 and 9 have the fewest samples for both the training and the test set as illustrated in the chart below.

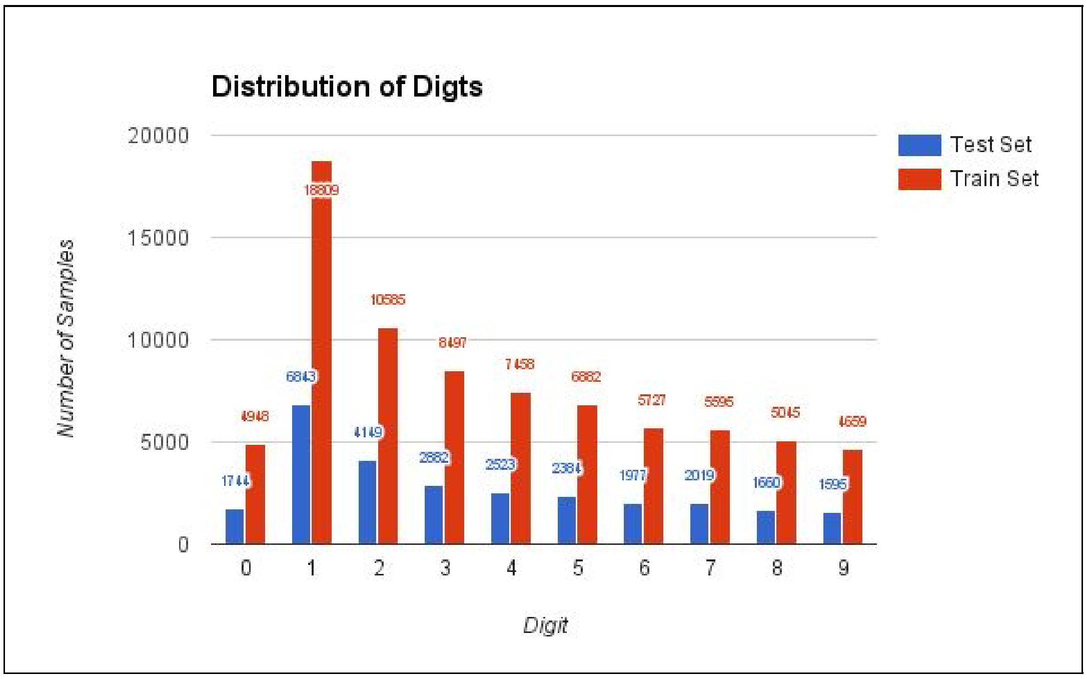

Fig 4: Distribution of Digit Samples on the Street View House Number Dataset

Algorithms and Techniques

My approach centred on making use of convolutional neural networks. Convolutional neural networks address an important characteristic of images in that they encode spatial information that we are not interested in for the purpose of classification, and which complicate the task of classification when using traditional neural networks. For example, say we have two images. One with the number 8 in the top left corner and another with the number 8 in the bottom right corner. For the sake of classification, it is an image of the number 8, but to a normal neural network that works directly on the pixels, these look like representations of two completely different things .

We need to help the neural network to summarise the image into a set of higher level concepts. We do this through a series of convolutions. The goal of the convolutions is to remove the spatial dimensionality of the image and replace this with a series of statistical facts about the image. Examples of the kind of facts that a network could learn about the image as a whole might be things like whether the image contains eyes, claws and whiskers. It does this by scanning the image for the small shapes that make up this eyes, claws and whiskers. This scanning makes use of the idea of weight sharing. Say you have a colour image that is 54 length x 54 width x 3 colour channels. Each convolutional layers might scan a 5x5x3 segment of the image and map this to a vector of dimension 16. In this case it is filtering the segment for 16 things. It then continues to scan other 5x5x3 segments of the image searching for the same 16 things, and in the end produces a new tensor that has a depth of 16 and a length and width based on how many pixels it skipped over horizontally and vertically when moving to the the next segment of the image to scan.

The image below is taken from the Udacity Deep Learning course and illustrates how this spatial reduction is achieved through successive convolutions of the original image on the left. The output of the final convolution is then fed into a regular deep neural network.

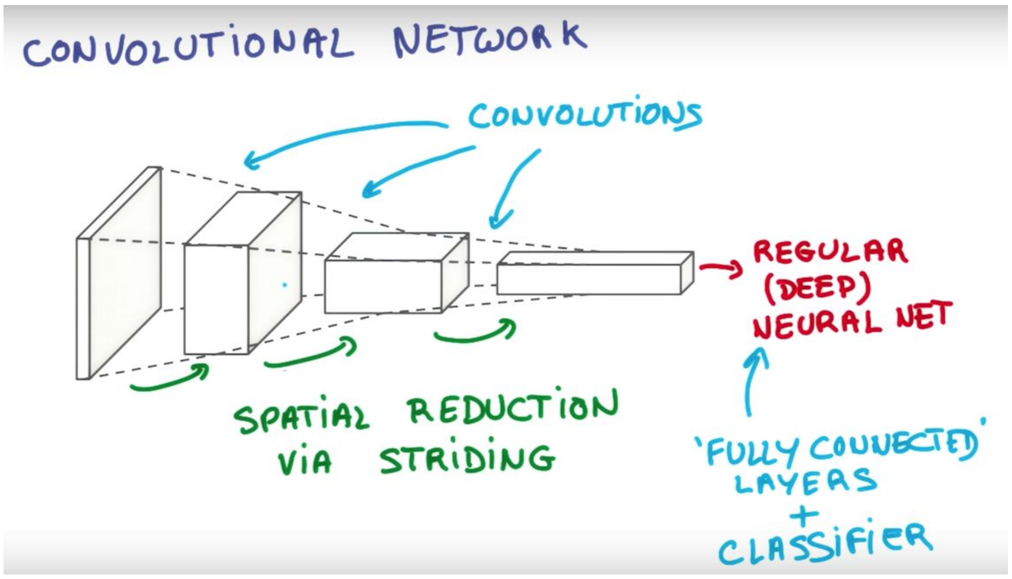
CONVOLUTIONAL NETWORK
CONVOLUTIONS
REGULAR (DEEP) NEURAL NET
SPATIAL REDUCTION VIA STRIDING
'FULLY CONNECTED' LAYERS + CLASSIFIER

Fig 5: Illustration of the mechanics of a convolutional network (image from Udacity Deep Learning Course)

My goal was to develop a deep convolutional neural network based on the model architecture below. This is inspired by the architecture used by Goodfellow, et Al [3] in their solution that achieved 96% accuracy.

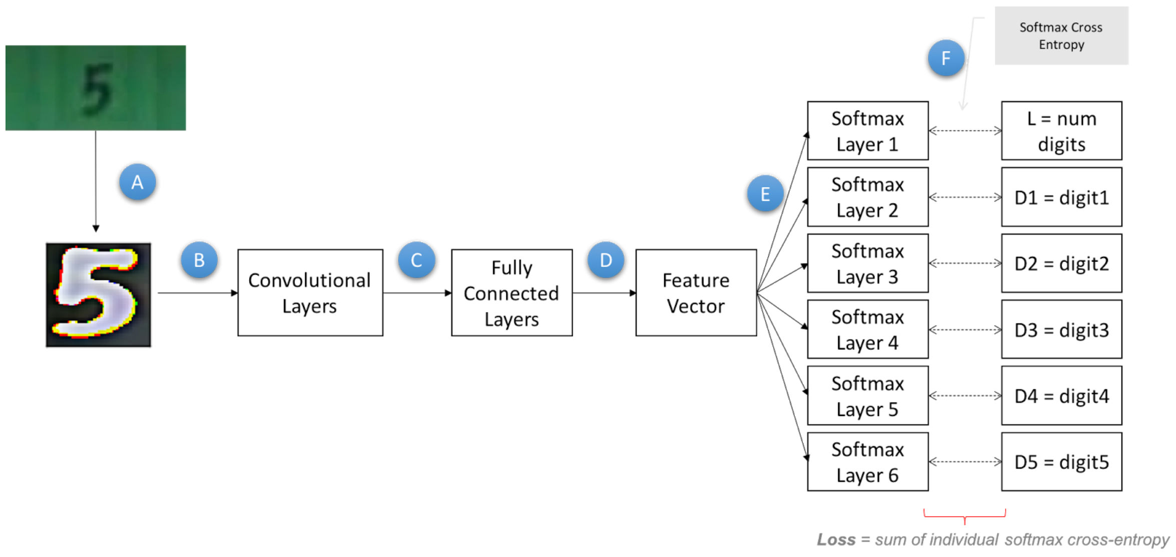

Fig 6: Model Deep Neural Network Architecture

Below is an explanation of each part of this architecture:

A) **Preprocess the raw images**
This is in order to obtain numpy arrays from the PNG files, cropped to uniform dimensions (54x54x3) and normalised (zero mean and equal variance) to make the computations easier. See Data Preprocessing section for more details.

B) **Process the image through a series of convolutions**
The convolutional layers extract features from the image used to identify individual digits and infer the number of digits in the house numbers. This results in a representation of the features that does not depend on where exactly the number is located in the image. Max pooling is employed to reduce sensitivity of the model to small changes in the image. Activation layers are also used after each convolution.

C) **Flatten the image and do feature extraction through fully connected layers**
After transformation of the image through convolutions, the data can be reshaped to a vector and processed through fully connected layers.

D) **Map to a feature vector for performing the final linear classification**
The final fully connected layer is the feature vector from which linear classification is performed.

E) **Connect the feature vector to six individual softmax linear classifiers**
There are six linear classifiers. One to represent the number of digits, and five to represent each digit. These result in one-hot encoded vectors of length 11, positions 0-9 for the actual value of item and 10 if it is blank.

F) **Calculate the total loss for the prediction and do back propagation**
The loss is the sum of the individual softmax cross entropies for each classifier.

My plan was to implement this in TensorFlow and make use of TensorFlow functionality for automatically doing the back propagation.

The weights were to be initialised randomly. Training would be done using mini-batches and my default optimisation technique was the TensorFlow tf.train.GradientDescentOptimizer with an initial learning rate of 0.05 employing exponential decay.

Benchmark

The benchmark that I will be using is the performance obtained by GoodFellow et Al [3]. In their paper they describe obtaining an accuracy of over 96%.

Methodology

Data Preprocessing

The first step was to download the source files of the Street View House Number (SVHN) images from the Internet. The two files for train and test were obtained from the below URLs:

- http://ufldl.stanford.edu/housenumbers/train.tar.gz
- http://ufldl.stanford.edu/housenumbers/test.tar.gz

When extracted, these files each provide a folder with the raw images in PNG format and two additional files:

- digitStruct.mat
- see_bboxes.m

I used only the digitStruct.mat file. As described on the SVHN Dataset website:

> The digitStruct.mat file contains a struct called digitStruct with the same length as the number of original images. Each element in digitStruct has the following fields: name which is a string containing the filename of the corresponding image. bbox which is a struct array that contains the position, size and label of each digit bounding box in the image. Eg: digitStruct(300).bbox(2).height gives height of the 2nd digit bounding box in the 300th image

For my purposes, I needed a bounding box around the entire house number and not for each digit. I also needed a label dataset for the true house number values of each image. Therefore It was necessary for me to extract the data from the digit struct and transform it into a python dict that had the image filename as the key and for the value the bounding box for the entire house number along with an int representing the true house number itself.

Once I had the bounding boxes and house number labels, I proceeded to converting the raw images into a dataset that could be input into the deep convolutional neural network (CNN) for training. The steps involved in this conversion are as follows:

1. **Initialise two numpy arrays to hold the images and labels.**
   I chose to store each image in an array of dimensions 54 x 54 x 3. I chose 54 since this was what was used in the reference implementation by Goodfellow, et Al [3]. The dimension 3 represents the RGB colour channels. See the python code below:
```
dataset = np.ndarray(shape=(len(image_files), image_size, image_size,
                                    color_channels), dtype=np.float32)
labels = np.ndarray(shape=(len(image_files)),dtype=np.int)
```

2. **Read each image into the dataset array and normalise the image data so that the values have a zero mean and equal variance**
   We do this in order to improve the quality of computations since errors can be introduced when combining very small and very large numbers. It also improves the effectiveness of the optimisation by reducing the search space for the optimiser. This normalisation is done by transforming each value in the image tensor 'x' so that it becomes (x-128)/128

3. **Crop each image and scale each image so that the full house number is in the centre and image has length and height 54x54**
   I used the information in the bounding box dictionary when doing the cropping.
4. **Generate the corresponding labels array**
   At the same time as pre-processing each image I appended the labels array with the corresponding true house number value.
5. **Generate statistics of the data after preprocessing**
   Results were as below:

Train Set Stats	
Full Dataset Tensor	33402, 54, 54, 3
Mean	-0.047923665
Standard Deviation	0.1981962

Test Set Stats	
Full Dataset Tensor	13068, 54, 54, 3
Mean	-0.043531056
Standard Deviation	0.22369793

6. **Pickle the dataset for convenient use later when training**
   As part of this, I took out 20% of the training set to use as a validation set during training.

In addition to the raw images in the SVHN dataset, I also created a synthetic set for initial training and validation of my CNN design. I generated images using matplotlib. I generated 35,000 images for training and 15,000 images for testing. For each image I chose a random number of 1 to 5 digits and randomly varied the font family, colour, style, and rotation. I then put these images through the same preprocessing as described above for the raw images.

The statistics of this synthetic image set are below:

Synthetic Train Set Stats	
Full Dataset Tensor	35000, 54, 54, 3
Mean	0.43541774
Standard Deviation	0.21217383

Synthetic Test Set Stats	
Full Dataset Tensor	15000, 54, 54, 3
Mean	0.43539107
Standard Deviation	0.21227966

Implementation

General Workflow for Training the Model

Below is the general workflow I used for implementing the training of my convolutional neural network. I implemented a number of helper functions to support these steps. For details, please refer to the included IPython notebook 'SVHN-CNN-Implementation.ipynb'

- Setup directory and file names for saving log data from the training session
- Extract the dataset from the pickle file generated during the pre-processing stage
- Plot example images from the test set
- Reformat the labels to use one-hot encoding
- Define the TensorFlow CNN
- Initialise TensorFlow session
- Run N optimisation iterations (e.g. N = 5000)
- Calculate accuracy on test set
- Print examples of images misclassified by the model
- Close Tensorflow session

For defining the CNN in TensorFlow I initially used the standard method as described in the Udacity Deep Learning assignment. However, I later discovered a library named 'Pretty Tensor'[4] and based my implementation on the framework described in the tutorial by Hvass Labs [5]. In my explanations below, I lay out the model definition using Pretty Tensor.

Convolutional Neural Network Version 1

Training on Synthetic Dataset

For the first version of my CNN I used the synthetic set, since this should be much easier than the real dataset and would help me validate that my CNN model was on the right track.

Below are some example images from the synthetic test set after preprocessing.

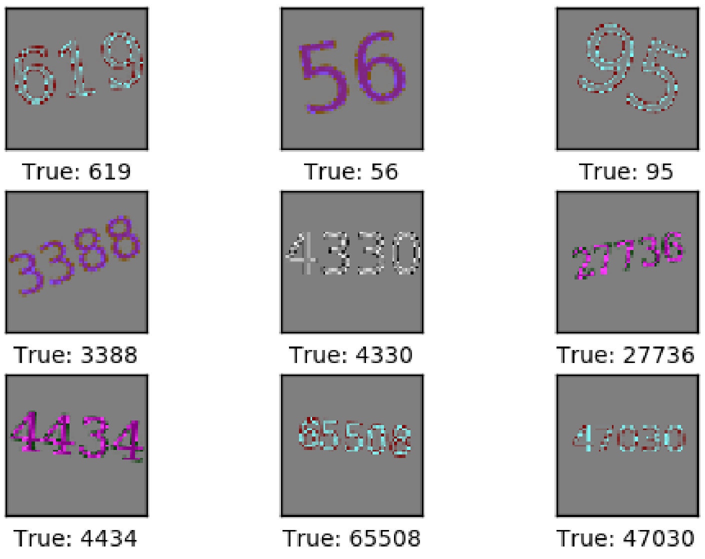

Fig 7: Examples from Synthetic Test Set (After Preprocessing)

For version 1 of my CNN I defined a model that had 2 convolutional layers with max pooling and ReLU on each. This was followed by 2 fully connected layers and finally 6 softmax classifiers at the end. One classifier was for predicting the number of digits and the remaining 5 were for predicting each of the digits.

The loss function was defined as the sum of the individual losses of each softmax classifier. The prediction of the model is a stack of all the individual predictions of each softmax classifier.

Please see definition of this model below:

Convolutional Neural Network - Version 1

```
with pt.defaults_scope(activation_fn=tf.nn.relu, phase=phase):
    network_base = x_pretty.\
        conv2d(kernel=5, depth=64, name='layer_conv1', batch_normalize=True).\
        max_pool(kernel=2, stride=2).\
        conv2d(kernel=5, depth=64, name='layer_conv2').\
        max_pool(kernel=2, stride=2).\
        flatten().\
        fully_connected(size=256, name='layer_fc1').\
        fully_connected(size=128, name='layer_fc2')
    y_pred1, loss1 = network_base.softmax_classifier(class_count=num_labels,
                                                     labels=y_true[:,0,:])
    y_pred2, loss2 = network_base.softmax_classifier(class_count=num_labels,
                                                     labels=y_true[:,1,:])
    y_pred3, loss3 = network_base.softmax_classifier(class_count=num_labels,
                                                     labels=y_true[:,2,:])
    y_pred4, loss4 = network_base.softmax_classifier(class_count=num_labels,
                                                     labels=y_true[:,3,:])
    y_pred5, loss5 = network_base.softmax_classifier(class_count=num_labels,
                                                     labels=y_true[:,4,:])
    y_pred6, loss6 = network_base.softmax_classifier(class_count=num_labels,
                                                     labels=y_true[:,5,:])
    loss = pt.create_composite_loss([loss1,loss2,loss3,loss4,loss5,loss6])
    y_pred = tf.pack([y_pred1,y_pred2,y_pred3,y_pred4,y_pred5,y_pred6],axis=1)
```

For optimisation in version 1, I made use of the Adagrad optimiser. During some early experimenting I compared the Gradient Descent optimiser to the Adagrad optimiser and found that Adagrad was faster at minimising loss and was able to achieve a lower level of loss over time.

Please see below for the definition of the optimiser for version 1:

```
optimizer = tf.train.AdagradOptimizer(learning_rate=0.005).minimize(loss,
                                                              global_step=global_step)
```

The results I obtained on the synthetic set are shown in the graphs below:

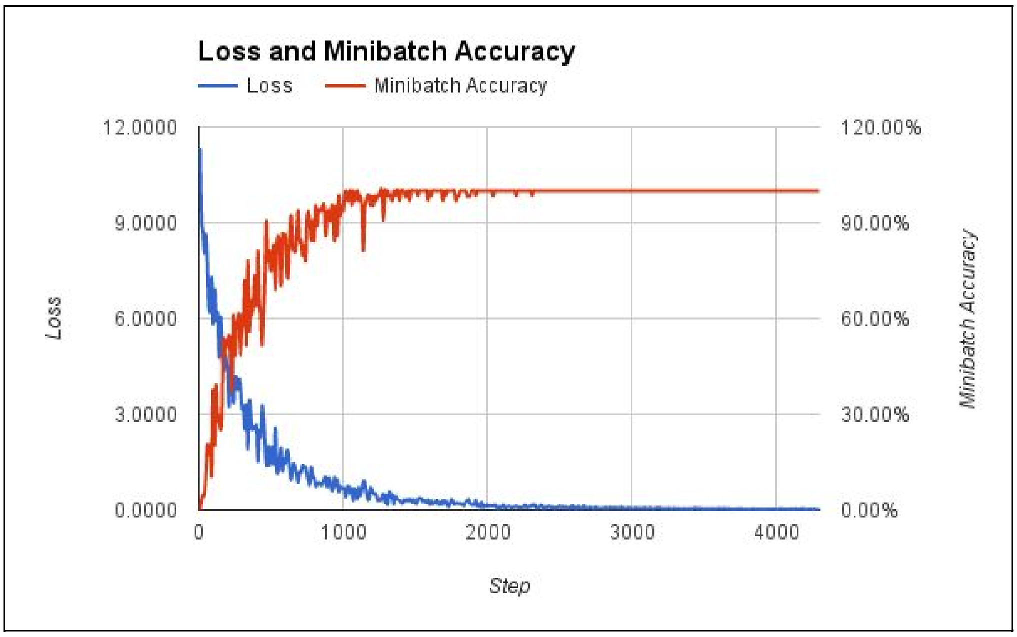

Fig 8: Loss and Minibatch Accuracy for Synthetic Dataset Using CNN version 1

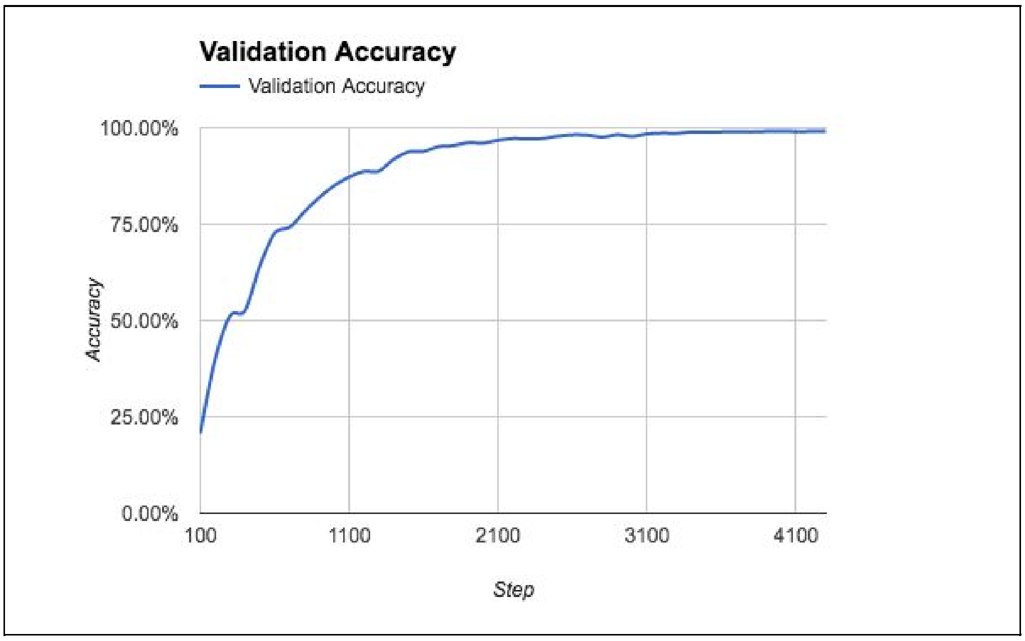

Fig 9: Validation Accuracy for Synthetic Dataset Using CNN Version 1

The accuracy on the test set came to 99.03%. Below are some examples of the images that it misclassified:

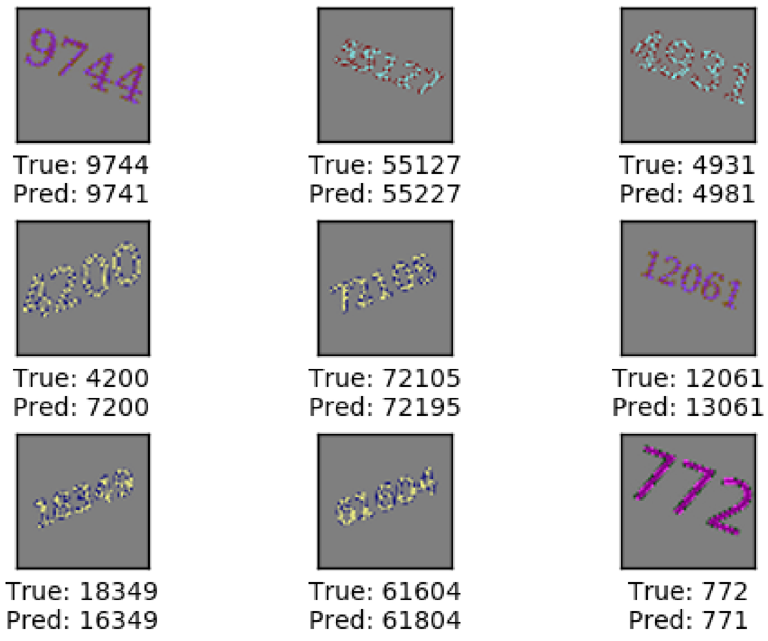

Fig 10: Examples of Incorrect Synthetic Test Set Predictions from CNN Version 1 after 4,000 Optimisation Iterations (99% accuracy on test set)

As we can see, CNN Version 1 performed quite well on the synthetic set. However, this was a relatively easy set of data to work with.

Given that the model performed well on the synthetic set, I then went on to using it to train on the real Street View House Number dataset.

Training on Real SVHN Dataset

Below are some example images from the SVHN Test set after preprocessing.

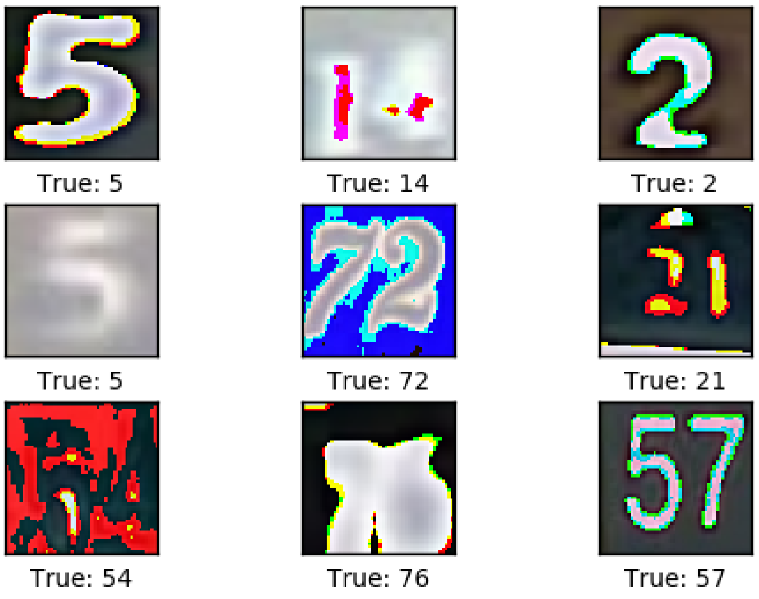

Fig 11: Examples from the SVHN Test Set (After Preprocessing)

For training on the real SVHN dataset, I modified the optimiser to use exponential decay on the learning rate. Below is the definition of the optimiser. The CNN model remained the same as before.

```
learning_rate = tf.train.exponential_decay(0.005, global_step, 500, 0.9, staircase=True)

optimizer = tf.train.AdagradOptimizer(learning_rate=learning_rate).minimize(loss,
global_step=global_step)
```

The results I obtained from training on the real SVHN dataset are shown in the graphs below:

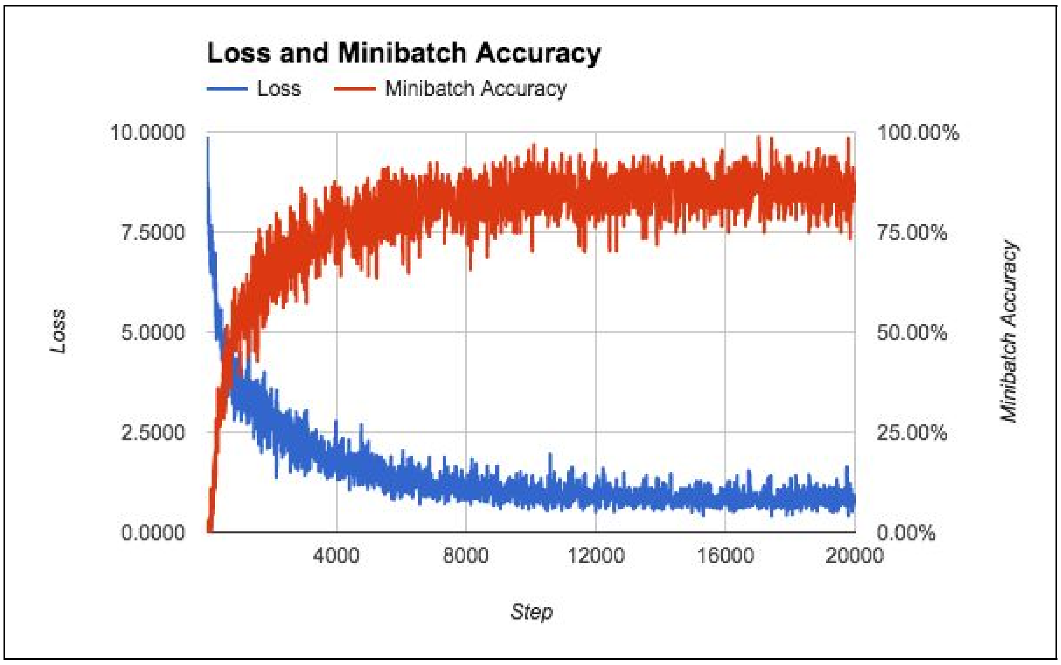

Fig 12: Loss and Minibatch Accuracy for SVHN Dataset using CNN Version 1

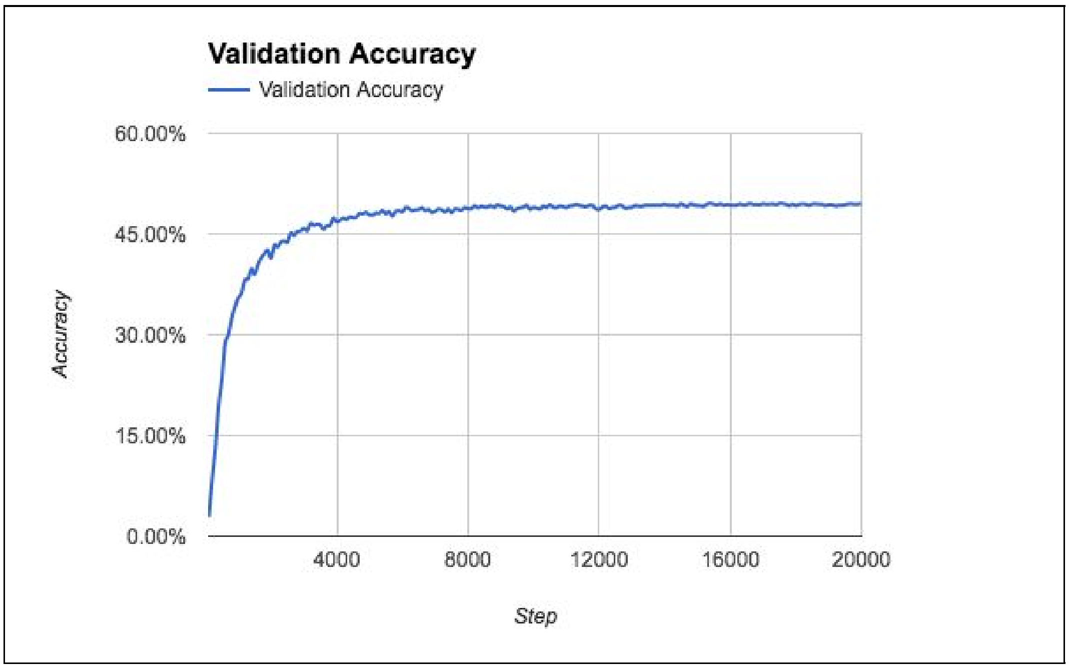

Fig 13: Validation Accuracy for SVHN Dataset using CNN Version 1

As can be seen, the loss and minibatch accuracy on each iteration did well, however the validation accuracy failed to get beyond 50% after 20,000 iterations. The best result on the validation set was 49.5%.

Similarly the performance on the test set was not good. The best result obtained on the test set was 47.28%

Below are some examples of misclassified images from the test set.

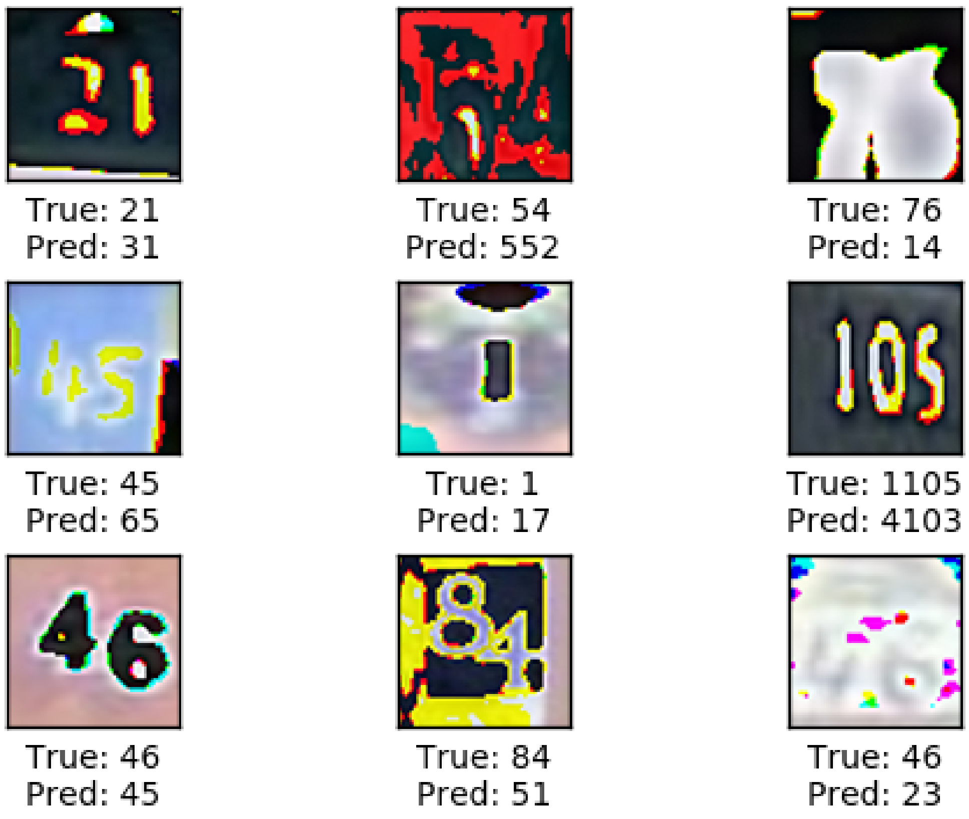

Fig 14: Examples of Incorrect Predictions from CNN Version 1 on SVHN Test Set after 20,000 Optimisation Iterations (47.14% accuracy on test set)

Convolutional Neural Network Version 2

After training CNN version 1 with the SVHN dataset and obtaining poor results, I created a version 2 of the CNN. The main changes were:

- Increased the depth by including 2 additional convolutional layers.
- Increased number of filters in the convolutional layers
- Increased the breadth of the fully connected layers
- Used batch normalisation on all convolutional layers
- Inserted dropout before each fully connected layer

Below is the Pretty Tensor definition on the CNN version 2

Convolutional Neural Network - Version 2

```
with pt.defaults_scope(activation_fn=tf.nn.relu, phase=phase):
       network_base = x_pretty.\
           conv2d(kernel=5, depth=48, name='layer_conv1', batch_normalize=True).\
           max_pool(kernel=2, stride=2).\
           conv2d(kernel=5, depth=64, name='layer_conv2', batch_normalize=True).\
           max_pool(kernel=2, stride=2).\
           conv2d(kernel=5, depth=128, name='layer_conv3', batch_normalize=True).\
           max_pool(kernel=2, stride=2).\
           conv2d(kernel=5, depth=160, name='layer_conv4', batch_normalize=True).\
           max_pool(kernel=2, stride=2).\
           dropout(0.95).\
           flatten().\
           fully_connected(size=2048, name='layer_fc1').\
           dropout(0.95).\
           fully_connected(size=1024, name='layer_fc2')
       y_pred1, loss1 = network_base.softmax_classifier(class_count=num_labels,
                                                        labels=y_true[:,0,:])
       y_pred2, loss2 = network_base.softmax_classifier(class_count=num_labels,
                                                        labels=y_true[:,1,:])
       y_pred3, loss3 = network_base.softmax_classifier(class_count=num_labels,
                                                        labels=y_true[:,2,:])
       y_pred4, loss4 = network_base.softmax_classifier(class_count=num_labels,
                                                        labels=y_true[:,3,:])
       y_pred5, loss5 = network_base.softmax_classifier(class_count=num_labels,
                                                        labels=y_true[:,4,:])
       y_pred6, loss6 = network_base.softmax_classifier(class_count=num_labels,
                                                        labels=y_true[:,5,:])
       loss = pt.create_composite_loss([loss1,loss2,loss3,loss4,loss5,loss6])
       y_pred = tf.pack([y_pred1,y_pred2,y_pred3,y_pred4,y_pred5,y_pred6],axis=1)
```

Below are the results I obtained on CNN version 2:

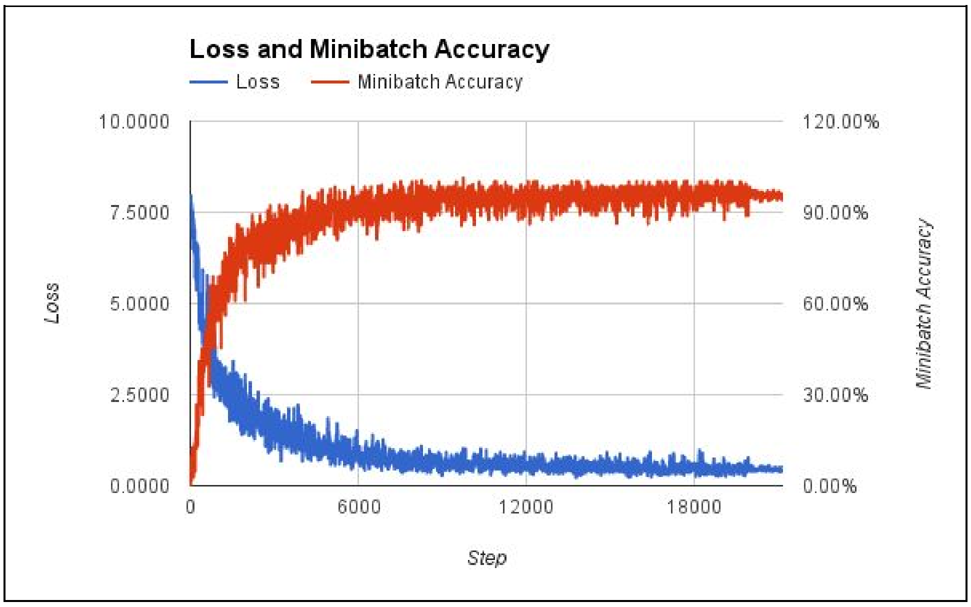

Fig 15: Loss and Minibatch Accuracy for SVHN dataset using CNN Version 2

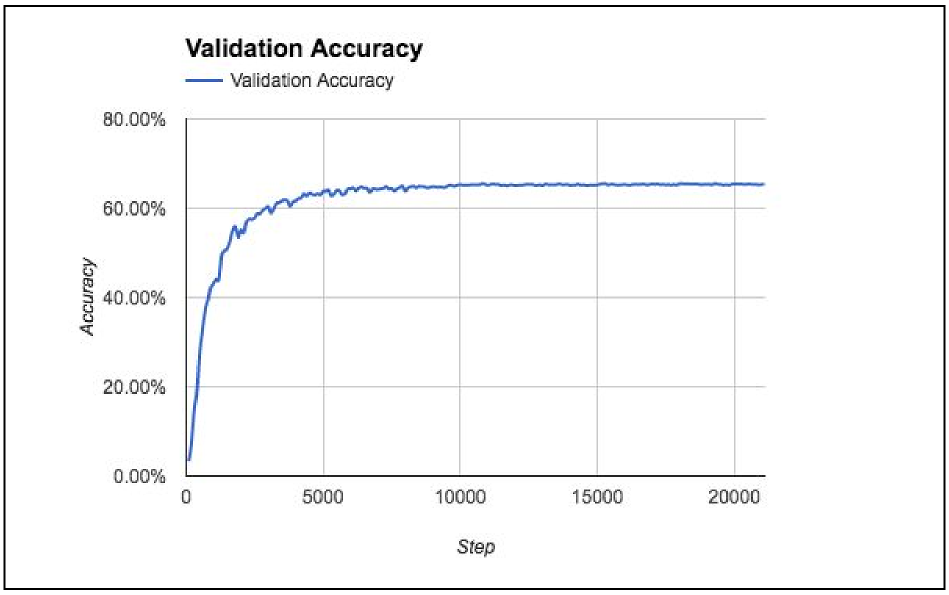

Fig 16: Validation Accuracy for SVHN Dataset using CNN Version 2

As can be seen, the validation accuracy for CNN version 2 did better than in CNN version 1, the best validation accuracy was 65.54%.

The best test accuracy obtained from this CNN version 2 was 62.77%

Some examples of incorrectly classified images are below:

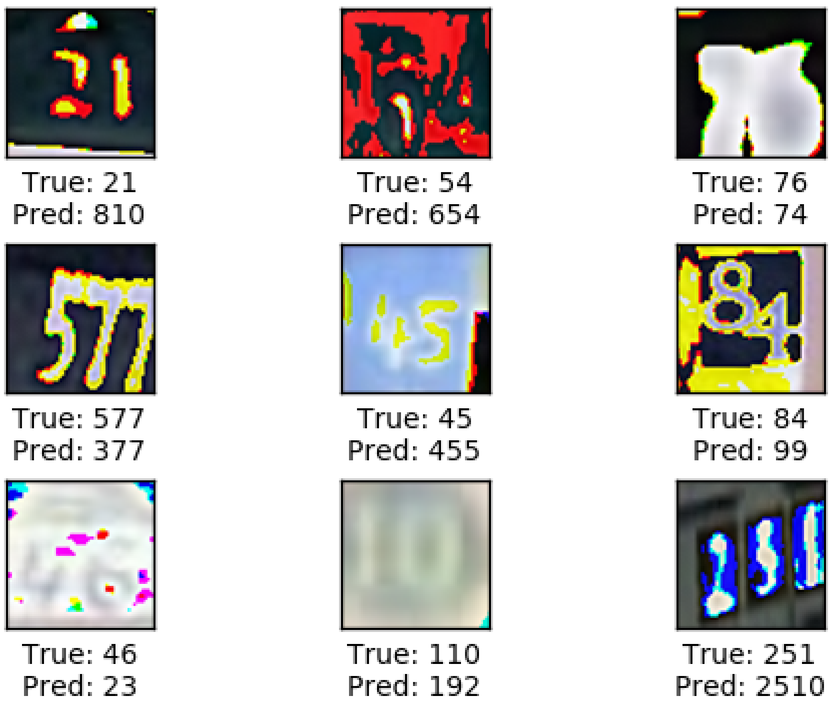

Fig 17: Examples of Incorrect Predictions from CNN Version 2 on SVHN Test Set after 22,000 Optimisation Iterations (62.8% accuracy on test set)

Refinement

Convolutional Neural Network Version 3

In order to refine my model to get better performance, I converted images from RGB to grayscale before the first convolutional layer. My theory was that the color channels contained redundant information making it more complicated than needed for the model to optimise.

See below for the addition of the grayscale conversion.

Convolutional Neural Network - Version 3

```
    with pt.defaults_scope(activation_fn=tf.nn.relu, phase=phase):
        network_base = x_pretty.\
            apply(tf.image.rgb_to_grayscale).\
            conv2d(kernel=5, depth=48, name='layer_conv1', batch_normalize=True).\
            max_pool(kernel=2, stride=2).\
            conv2d(kernel=5, depth=64, name='layer_conv2', batch_normalize=True).\
            max_pool(kernel=2, stride=2).\
            conv2d(kernel=5, depth=128, name='layer_conv3', batch_normalize=True).\
            max_pool(kernel=2, stride=2).\
            conv2d(kernel=5, depth=160, name='layer_conv4', batch_normalize=True).\
            max_pool(kernel=2, stride=2).\
            dropout(0.95).\
            flatten().\
            fully_connected(size=2048, name='layer_fc1').\
            dropout(0.95).\
            fully_connected(size=1024, name='layer_fc2')
        y_pred1, loss1 = network_base.softmax_classifier(class_count=num_labels,
                                                         labels=y_true[:,0,:])
        y_pred2, loss2 = network_base.softmax_classifier(class_count=num_labels,
                                                         labels=y_true[:,1,:])
        y_pred3, loss3 = network_base.softmax_classifier(class_count=num_labels,
                                                         labels=y_true[:,2,:])
        y_pred4, loss4 = network_base.softmax_classifier(class_count=num_labels,
                                                         labels=y_true[:,3,:])
        y_pred5, loss5 = network_base.softmax_classifier(class_count=num_labels,
                                                         labels=y_true[:,4,:])
        y_pred6, loss6 = network_base.softmax_classifier(class_count=num_labels,
                                                         labels=y_true[:,5,:])
        loss = pt.create_composite_loss([loss1,loss2,loss3,loss4,loss5,loss6])
        y_pred = tf.pack([y_pred1,y_pred2,y_pred3,y_pred4,y_pred5,y_pred6],axis=1)
```

I also changed my optimiser from Adgrad to Adam. I considered trying different learning rates and exponential decay with Adagrad, however, when I saw that the TensorFlow Adam optimiser has a default for the learning rate and all other parameters I reasoned that working with these defaults might be better than any guess I might make. Additional information on the Adam optimiser can be found at the link below:
https://arxiv.org/abs/1412.6980

Below are results from CNN version 3:

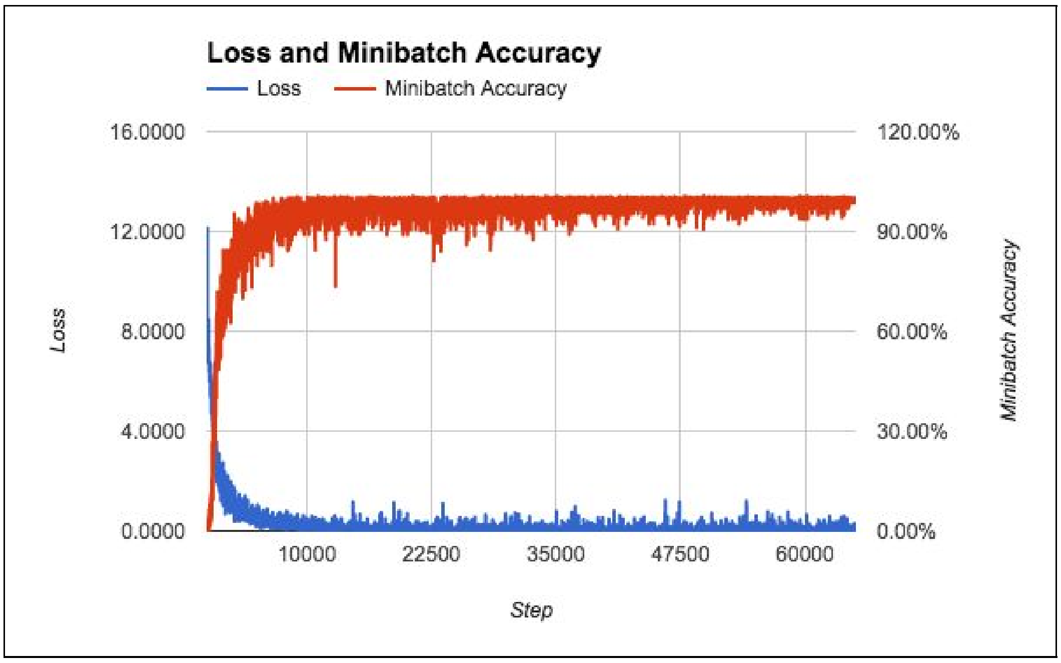

Fig 18: Loss and Minibatch Accuracy for SVHN Dataset using CNN Version 3

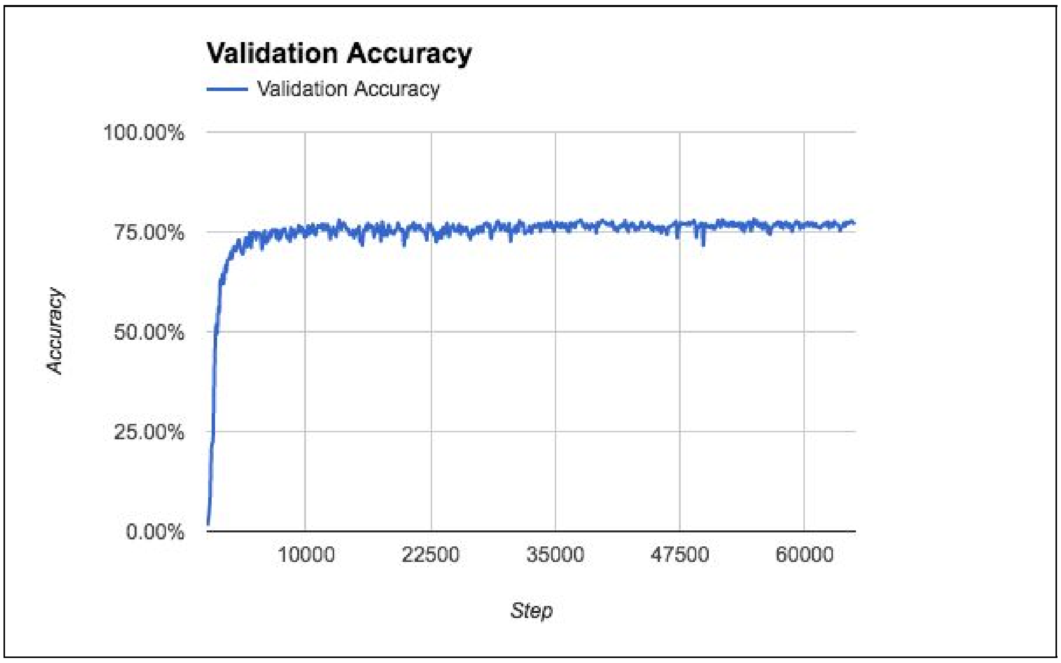

Fig 19: Validation Accuracy for SVHN Dataset using CNN Version 3

As can be seen, CNN version 3 performed significantly better with loss getting close to 0 and minibatch accuracy reaching 100%. For the validation test, the best result obtained was 78.28%

The test set also performed significantly better, with the best result being 76.61%. Some examples of misclassified images are below.

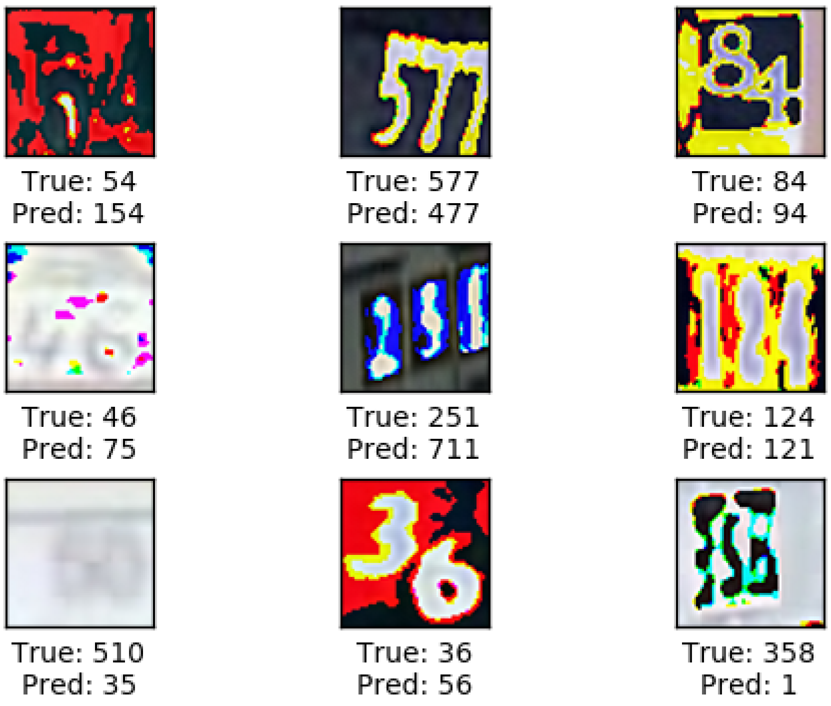

Fig 20: Examples of Incorrect Predictions from CNN Version 3 on SVHN Test Set after 65,000 Optimisation Iterations (75.6% accuracy on test set)

Results

Model Evaluation and Validation

My final model achieved an accuracy of 75.6% on the SVHN Test Set.

The overall model for CNN Version 3 follows the general pattern proposed by Goodfellow et Al. The optimisations I made, such as converting the images to grayscale contributed to simplifying the model. Furthermore, by using max pooling the model is less sensitive to small perturbations in the training data.

In order to further validate my model I used it to predict the numbers from some images I took outside with my phone. Below are the results.

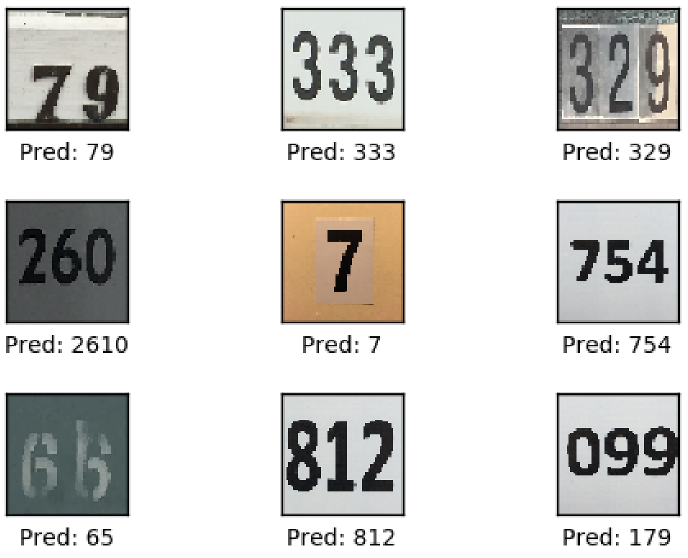

Fig 21: Predictions using CNN Version 3 of Images Taken on My Phone

There are a few noticeable errors, however, in general I would trust the results from my CNN Version 3 model if the original image quality were relatively good and cropped to show only the house number.

Justification

My CNN version 3 model test set performance of 75.6% is not as good as the 96% benchmark by Goodfellow et Al [3] , however the benchmark took 6 days to train on very powerful hardware, and given that I was using more limited resources, I believe that this represents a decent result.

My solution seems good enough to have solved the problem where the image quality is relatively good.

Conclusion

Free-Form Visualization

The two charts below present an interesting story about my results. On the chart on the left, we see that the minibatch loss got very close to 0 and the accuracies on the minibatches got very close to 100% (in some iterations actually achieving 100). However the chart on the right shows the validation set accuracy plateauing at around 75%.

This shows that reducing loss does not always yield better accuracy on unseen data, and potentially some regularisation is needed on the network. Also, perhaps the loss function could be enhanced to ensure that the weights generalise well to unseen data.

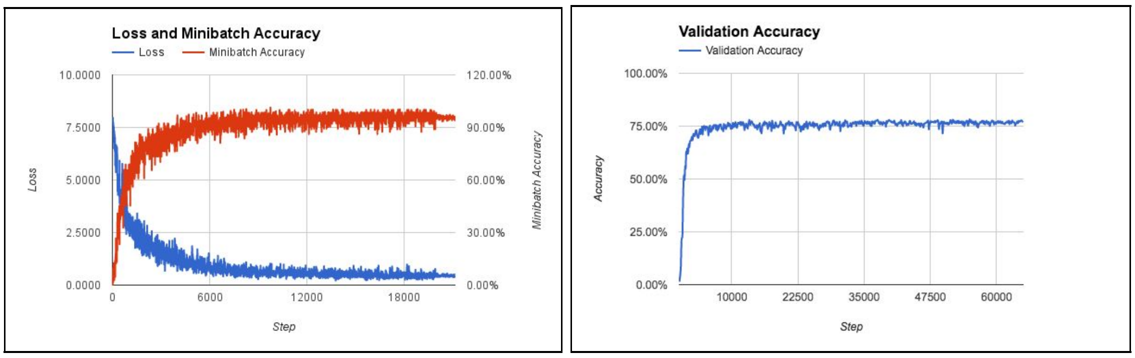

Fig 22: Comparison - minibatch vs validation performance

Reflection

This project helped me learn a lot about deep learning, convolutional neural networks and TensorFlow.

The preprocessing stage took a significant amount of effort but was a very important part of the overall workflow. Building the synthetic dataset proved very useful in validating my early versions of the CNN and furthermore, the good results I got at the beginning were a great motivation boost.

One thing I learned while implementing this is that memory utilisation needs to be taken into consideration when developing a solution. I found myself utilising a high memory virtual machine in the cloud in order to execute my scripts. Splitting up the processing into smaller batches can help to ensure that memory utilisation is contained.

I also learned that logging is very important. I used Python’s logging function to generation csv files from my optimisation runs. I was then able to process this data easily in a spreadsheet.

A couple of tricky parts to implementing this arose from the fact that I had 6 classifiers at the end of the network instead of 1. Defining the loss and prediction as a composite of the results of these 6 classifiers took a bit of effort to get right. Also, defining the accuracy function took a bit of time to get right.

Discovering Pretty Tensor and the tutorials on Hvass Labs significantly helped to accelerate my ability to experiment with different CNN models.

My expectations for the solution have been met, and it could be used in a general setting as long as fairly good quality images are used.

Improvement

Some potential areas of further improvement are below:

- Using smaller patches (3x3) may yield faster optimisation as recommended in Stanford course 231n , lecture 11
- It could be worth re-designing the loss function, since I observed that loss was able to reduce to almost zero yet the CNN accuracy plateaued. This leads me to believe that the current definition of the loss function might not be perfectly aligned with the factors that drive accuracy.
- In the paper by Goodfellow et Al [3] , they mention that adding more layers yields better results, therefore adding more layers to my model might yield improved accuracy results.
- Using transfer learning from an existing model (e.g. Google's Inception model) might achieve greater accuracy in less time.

References

[1] MNIST. http://yann.lecun.com/exdb/mnist/

[2] Y Netzer, T Wang, A Coates, A Bissacco, B Wu, AY Ng, *Reading Digits in Natural Images with Unsupervised Feature Learning*, 2011

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

[4] SVHN Dataset: http://ufldl.stanford.edu/housenumbers/

[5] Google Pretty Tensor: https://github.com/google/prettytensor

[6] Hvass Labs TensorFlow Tutorial: https://github.com/Hvass-Labs/TensorFlow-Tutorials