Capstone Project - Deep Learning

Bradley Stillwell
11 August 2016

Machine Learning Engineer Nanodegree

Definition

Project Overview

Digitally recognizing numbers in real life images has been a tough problem in artificial intelligence for many decades. The problem stems from the seemingly endless variations on fonts, colors, spacings, locations etc that these numbers can take within an image.

We can see this problem clearly in the street view house numbers (SVHN) dataset[1].

Recent advances in neural network techniques such as convolutional neural networks[2] and advances in computing power have made the task much easier as shown in academia[3]. But this does raise the question as to whether a system like this could be built on standard hardware such as a laptop, without the need for massive computing resources.

In this project I aim build an android application that will recognise a series of digits in real time using its camera stream. To recognise the numbers I will use a deep convolutional neural network. I will do the above on standard hardware, that being a quad-core laptop with 8gb of memory and without any specialised numerical processing hardware such as a gpu.

Problem Statement

There are a number of challenges in building an android application to recognise a number in an image. Firstly we'll need to build a system to recognise the digits, and secondly we'll need to to turn this into a realtime android application.

Number recognition

To recognise the number in the images, I will build a classifier using a deep convolutional neural network (cnn).

The performance of a cnn is highly dependant on it's structure, and we will spend a large part of our effort searching for the structure that gives us the best performance, this is known as a hyperparameter search.

Training a large neural network can take a large amount of time. It is common in academia for training to take a number of days for a deep cnn configuration. Due to available computational resources and large

---

[1] http://ufldl.stanford.edu/housenumbers/
[2] http://yann.lecun.com/exdb/publis/index.html#lecun-98
[3] http://static.googleusercontent.com/media/research.google.com/en//pubs/archive/42241.pdf

hyperparameter search space, I will limit the classifier training time to a small number of minutes for each configuration.

A scoring metric will be used to compare the classifiers in the hyperparameter search with the best performing classifier being used in the android application.

Android application

The android application will utilise our best performing classifier to recognise numbers. The application will do this by feeding it a live stream of images from the camera. The output of the classifier will be transformed into a string to be displayed on the screen.

To implement this we will use an existing demo application in the tensorflow github repository, and change it's code to use our classifier. The code itself is written in java and c++, and requires a functioning bazel build chain to compile. Getting all of this to working together is challenging.

The finished project will be a working classifier, running in an android device that correctly recognises numbers.

Metrics

In order to find the classifier that predicts the best, we will need to define a way to compare them. For this we will use the mean accuracy of digits predicted correctly.

For example if the classifier predicts 123 and the correct label is 120, the accuracy for this would be ⅔, as the digits 1 and 2 are correct and in the correct location, but the 0 is incorrect.

Another example predicts 12, and the correct label is 312, then the accuracy will be 0/3 as no digits are correct in the correct place.

The final score will be the accuracy averaged across the number of items in the batch.

Analysis

Data Exploration

The data I am using for training the classifier involves images which I will take from the SVHN dataset[4]. Simple statistics about the testing and training dataset are presented here.
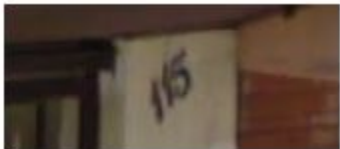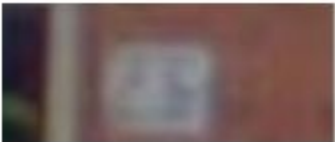
Dataset	Size	Samples	Max digit length	Min image width	Max image width
train.tar.gz	404mb	33402	6	9	423
test.tar.gz	276mb	13068	5	10	222

We have a large number of test and training samples which is good. Here are some samples taken from the dataset with their corresponding labels.

115	60	7

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

2	139	373

We can see that the numbers in the images have large variations. There are different font styles, colors, rotations, and the location of the number in the image also varies. We can see that some images show very poor quality such as the image labeled 2. We also see that this image has been labelled incorrectly and should be 25.

Exploratory Visualization

The number of digits we will need to deal with is important as it has a bearing on the complexity of our cnn structure.
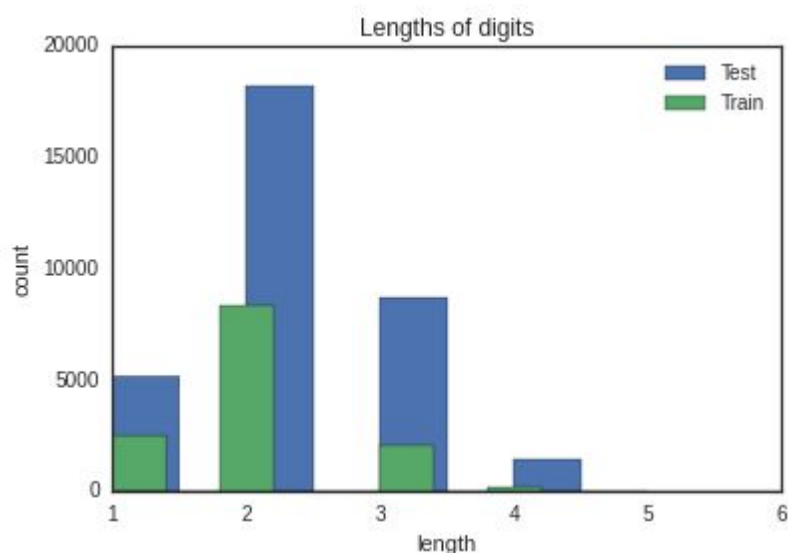
We can see that the distributions of image lengths for test and train are similar, but it appears that there are very few examples with 4 or 5 digits.

The quality of the images is important, if the images are too small they will not be good for use. I will investigate the quality of the image by using its height.

Heights of images
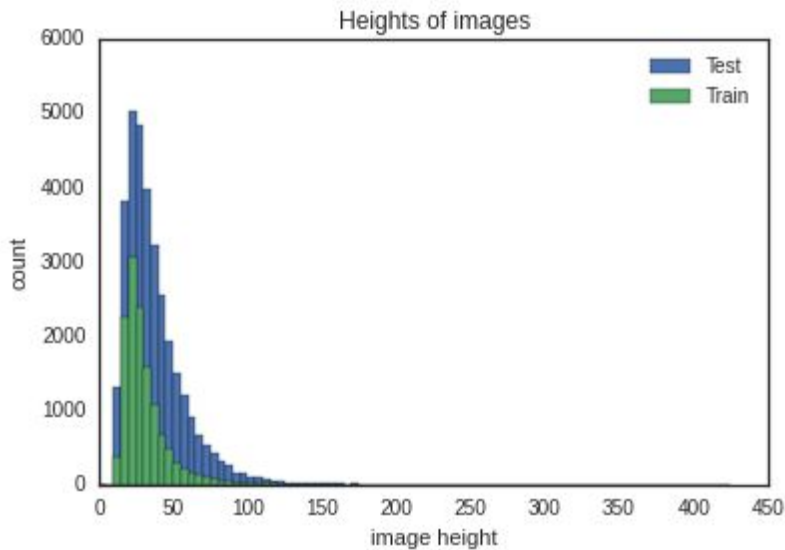
Again we see that the distributions between train and test are similar with both peaking around an image height of 25 pixels. We could cull the images with smaller heights to remove poor quality images.

Algorithms and Techniques

Here I present the algorithms and techniques used in creating the classifier. I also detail default parameters as well as the values to be used in the hyperparameter search. The parameters used for benchmarking are highlighted in **bold**.

Training, validation and testing

To train the network I will use images formatted to 28x28 pixels, where the training images will be passed into the classifier in batches.

After initial investigation, a batch size of 64 has been chosen as smaller batch sizes have shown to cause overfitting in the network.

The size of the images was chosen after initial research which showed that larger images exhaust the computers memory. Another factor taken into account is that the image size is 1/8th the image size used in the android demo application, which will make it easier integrate when building this part of the system.

Also due to computational restrictions, the training will run for a set time. Once the training has completed for a cnn, a final score will be calculated based on a testing dataset.

A validation set will be kept as a reference and the accuracy for this will be calculated after a certain number of training steps.

Defaults:
- Image size 28x28 pixels
- Batch size 64 examples
- Measure validation accuracy every 500 steps

Deep convolutional neural networks

Deep convolutional neural networks have been shown to work well for image recognition problems[5]. They will be well suited to our problem as they are specifically useful for finding structure in images, where there is variation in the position of the item we want to recognise.

In our case the items to recognise will be digits and there is variation in tje location of these.
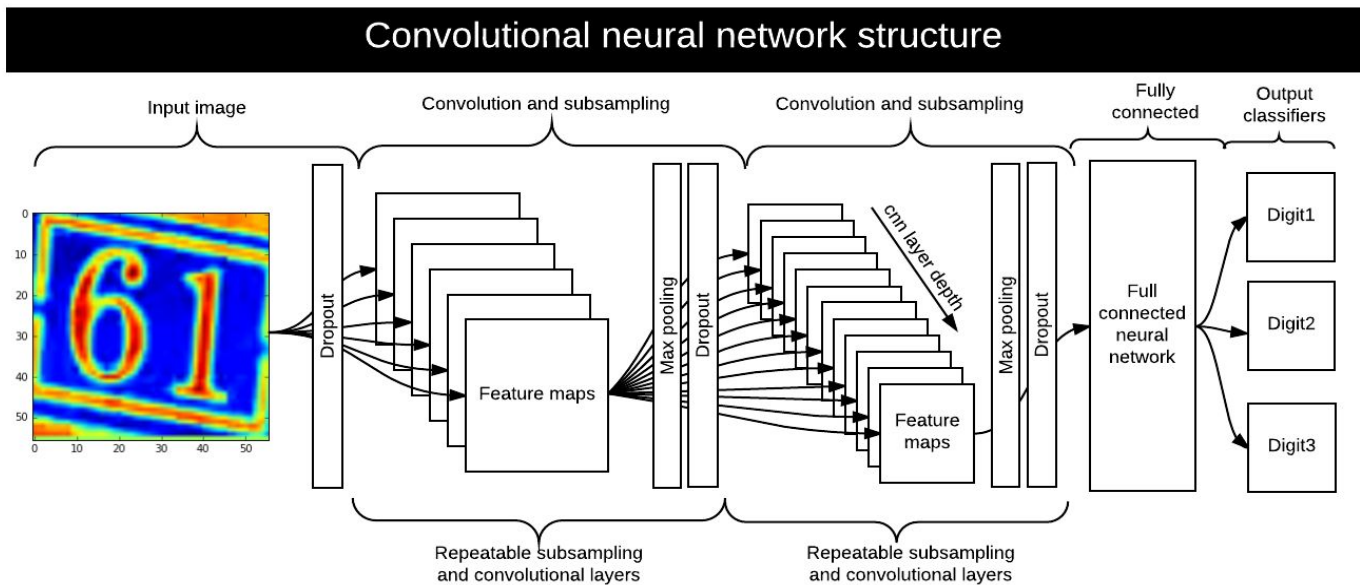

Figure 2 - Sample cnn configuration

A deep cnn is generally made up of an input layer, followed by a number of convolution and subsampling layers, followed by one or more fully connected layers which feed into output classifiers.

Convolutional layer

A convolutional layer[6] builds up a number of feature maps based on running a kernel across its input. The kernel itself is a small neural network, which moves across its input space at a certain step rate called a stride. At each stride it will take a patch of data surrounding the point, and feed this though it's internal neural network. The kernel will output a number of values, determined by a depth parameter. It is common that the depth increases with the number of layers.

Defaults
- Stride of 1
- Padding of type 'SAME'[7]
- Patch size of 5

Hyperparameter variables
- Depth of layer, [8, **16**,32]
- Number of layers [**1**, 2, 3]

---

[5] http://yann.lecun.com/exdb/publis/index.html#lecun-98

[6] https://en.wikipedia.org/wiki/Convolutional_neural_network

[7] https://www.tensorflow.org/versions/r0.10/api_docs/python/nn.html#conv2d

Rectified Linear Unit (Relu)

Neural networks are based on the firing of an activation[8] which surrounds each neuron in the network. There are many different activation functions, but it has been shown that Relu works well for large neural networks due to a number of its properties, such as being very fast to calculate and that it doesn't suffer from the vanishing gradient problem[9].

I will use Relu as the activation function around the convolutional layers and the fully connected layer.

Dropout

Dropout is a technique to improve the robustness of a neural network by randomly removing connections during training. This removal forces the neural network to not depend on any single weight for an activation, as during training that weight might not be available. This has been shown to improve performance in neural networks[10], and prevents overfitting.

After reading the research[11] I decided to apply dropout at a number of places in the network. The keep value is the percentage of input elements to keep in the network.
- Keep of 0.9 at the input
- Keep of 0.75 after each convolutional layer
- Keep of 0.5 after the fully connected layer

Hyperparameter variables
- Use dropout [True, **False**]

Max pooling

Max pooling is a technique used with cnn's which subsamples data in the feature maps. It is applied in a similar fashion to a cnn kernel, where it visits points in the input space, but the function it provides is to output the maximum value of the patch it is sampling. This effectively reduces the size of the feature maps.

Hyperparameter variables
- Use dropout [True, **False**]

Softmax

Softmax[12] is a function for normalising a number of inputs to a range between 0 and 1, where the outputs sum to 1. This is particularly useful for normalising the output of a neural network.

We will use softmax to normalise the outputs of each digit classifier.

Optimiser and loss function

For our neural network to learn we provide a function to apply small changes to the weights and biases during the backpropagation step.

The function I have chosen to do this is gradient descent. The amount of change applied to the weights and biases is determined by a loss function and a learning rate.

---

[8] https://en.wikipedia.org/wiki/Activation_function
[9] https://en.wikipedia.org/wiki/Vanishing_gradient_problem
[10] https://www.cs.toronto.edu/~hinton/absps/JMLRdropout.pdf
[11] https://www.cs.toronto.edu/~hinton/absps/JMLRdropout.pdf
[12] https://en.wikipedia.org/wiki/Softmax_function

Our loss function will be based on cross entropy loss. I have chosen this as it works well for multi class classification, meaning that there is only a single correct value in a set of possible values.

For our final loss calculation we will add the averages of the cross entropy loss for each digit classifier after applying softmax to it.

Hyperparameter variables
● Learning rate [**0.05**, 0.025, 0.1]

Fully connected layer

A fully connected layer represents a neural network connected the all of the inputs and outputs. It is common to have one or more fully connected layers in a deep cnn.

Hyperparameter variables
● Fully connected size [**32**, 64, 128]

Training times

In order to investigate how training time affects the parameters of our classifiers, I will run the full hyperparameter search with maximum training times of 2 and 5 minutes.

Hyperparameter search

In order to find the optimal classifier, we will build a system to run a hyperparameter search across the variables listed in this section. I provide a summary here, with the values used for the baseline highlighted in **bold**.

Use dropout	True/**False**
Use max pooling	True/**False**
Number of layers and depth of cnn	(8), (**16**), (32), (8, 16), (16, 32), (8, 16, 32)
Fully connected layer size	**32**, 64, 128
Learning rate	**.05**, .10, .025

Benchmark

Due to the constraints on training time and computing power, there is no suitable benchmark for comparison, so I will create one.

The benchmark will be the accuracy score based on initial runs of the system using the default parameters as listed in the hyperparameter search. I present the benchmark here.

| **Training time**	**Accuracy score**
2 minutes	0.686
5 minutes	0.745

Methodology

Data Preprocessing

To have effective training, the input images will need to be sized and ranged to similar values.

Images with heights < 20px will be culled as the quality of them is poor.

The color range I will use is between -0.5 and 0.5. To transform the images into this focolor range, the colors are converted into grayscale, normalised and translated.

In choosing which part of the image we should crop too, we should take into account the usage of the android application and how a user would align digits. My guess is that the digits would be centered in the device screen, and so this is how we will crop our images.

To do this I calculate a bounding box around the digits by using the data taken from the labels. I then expand this box by 10% so we are not too tight around the number. I then expand the box to make it a perfectly square and crop the image. If the bounding box extends outside the image, I fill in the extra space with the mean color value of the image. Here are some examples.
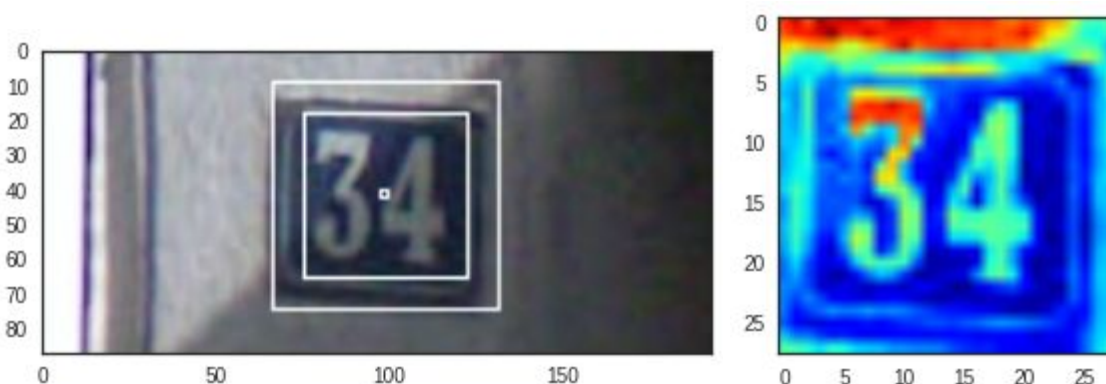
Here on the left we can see the original image with the bounding box of the digits. The box is expanded by 10% for the final crop. On the right we see the cropped image, with its colors flattened and scaled to a range between -0.5 and 0.5
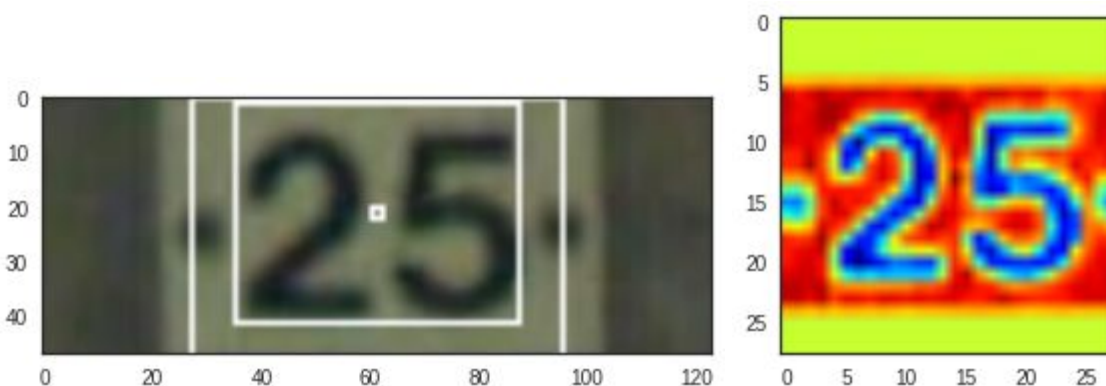
Here on the left you can see the bounding box extends the image dimensions, and on the right you can see the the empty spaces filled in with the images mean value, in this image that is shown as yellow.
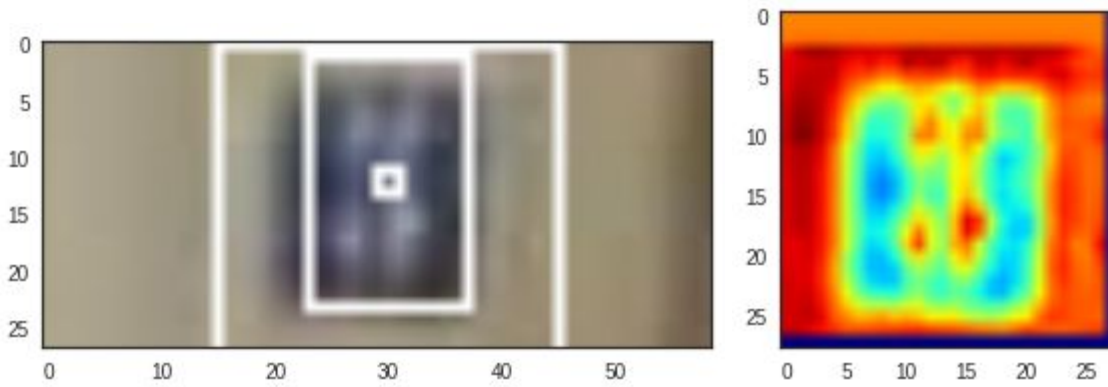
Here you can see a poor quality image, and the bounding box is made into a square.

Implementation

I will break the implementation of this project into two parts, the building of the classifier and the building of the android application. The code[13] has been commented to provide detailed documentation, and in this section I describe a few highlights and important features.

Building the classifier

Create graph function
The create graph function is the key to a working hyperparameter search. The function takes a configuration of parameters and will return a graph for training. The creation of the graph was fairly straightforward using tensorflow.

The creation of the loss function was one of the more interesting sections here. I decided to create one classifier for each digit. To implement this, each output will be a one hot encoded array of the numbers 0-9 plus a character to indicate a space, making a total of 10 possible outcomes. The loss of each classifier needs to be calculated separately, and then joined together. For this to work I chose to add the output of each classifier together to create the following loss function.

```
46
47          #
48          # Create the loss function
49          #
50          loss_digits = [tf.reduce_mean(
51              tf.nn.softmax_cross_entropy_with_logits(
52                  logits[i],
53                  tf_train_labels[i]
54              ))for i in range(num_digits)]
55
56          loss = tf.add_n(loss_digits, name='tf_loss')
57
```

Another important feature of the create_graph function was the creation of tensors for use in the android application. The input for this tensor is different from our training tensors as it will take only 1 image with 3 channels, where in training we take a batch of 16 images with 1 color (grayscale).

To make this work I take in an input image, flatten the 3 color channels by averaging them using reduce_mean. I then translate the color into a range of -0.5 < x < 0.5 which is what is expected by the model.

---

[13] https://github.com/bvstillwell/udacity-mlnd-capstone

```
#
# Create a predictor for a single image, for use in the android app
#
tf_predict_single_dataset = tf.placeholder(tf.float32, shape=(1, img_height, img_width, 3), name='tf_predict_single_dataset')
# Average the color
tf_conv = tf.reduce_mean(tf_predict_single_dataset, 3)
# Shape it correctly
tf_conv = tf.reshape(tf_conv, tf_conv.get_shape().as_list() + [1])
# Move the color range
tf_conv = tf.add(tf.div(tf_conv, 255), -0.5)
tf_predict_single_output = tf.squeeze(
    tf.concat(1,
            [tf.nn.softmax(model(tf_conv)[i],
                      name="tf_test_prediction_%d" % i)
            for i in range(num_digits)]
            ),
    name='predict_single_output')
```

The prediction is returned as an array where each digit will take up 11 values.

Hyper parameter search

The hyperparameter search is an essential component in order to compare the different graphs. What it essentially does is runs through a list of graph configurations, where it will instantiate the graph, train it and give it a score for comparison. The training of the graphs is done in sequentially.

The implementation of the hyperparameter search was straightforward.

The search itself creates a number of outputs, these being

- A log file detailing the graphs training process. This includes its parameters, and training step information. The training step information includes the step number, learning rate, training batch accuracy score, validation accuracy score and final testing score.
- A saved model
- Results file in csv format, one row for each graph.

Building the android application

The construction of the android app was difficult due to the requirements of working with many different languages and the bazel build chain.

Updating the demo application code

The tensorflow git repo contains demo code to create an android application that performs predictions on a live stream of data. I used this as the basis to create my android application.

A number of steps were required to get this working, firstly I updated the code to use my classifier. To make this work I updated the java code to expect the image size, and standard deviation.

I then updated the c++ code to reformat the output of the classifier to be displayed on the screen. To transform the output of the classifier, I cut the array into lengths of 11, where each cut represents the prediction of a single digit. To extract the prediction I take the argmax of each cut.

To turn this into a string for display, I investigate the prediction of each digit one at a time. If the value of the prediction is poor, I do not show the digit. I chose the value of 0.3 as the threshold. If the value is above this I will display the digit, or and 'x' to indicate a space. The code for this is shown here.

```
40
41    float sum = 0.0;
42    for (int digit = 0; digit < max_digits; ++digit)
43    {
44      sum += confidence[digit];
45
46      if (confidence[digit] < 0.3)
47        ss2 << '.'; //Low confidence on this digit
48      else if (max[digit]<10)
49        ss2 << max[digit];
50      else
51        ss2 << 'x';
52    }
```

Refinement

A major refinement to the project was done in the training data. I initially was predicting for 5 digits, but I noticed that I was getting very high accuracy scores. This was due to the very low number of samples with 4 or more digits.
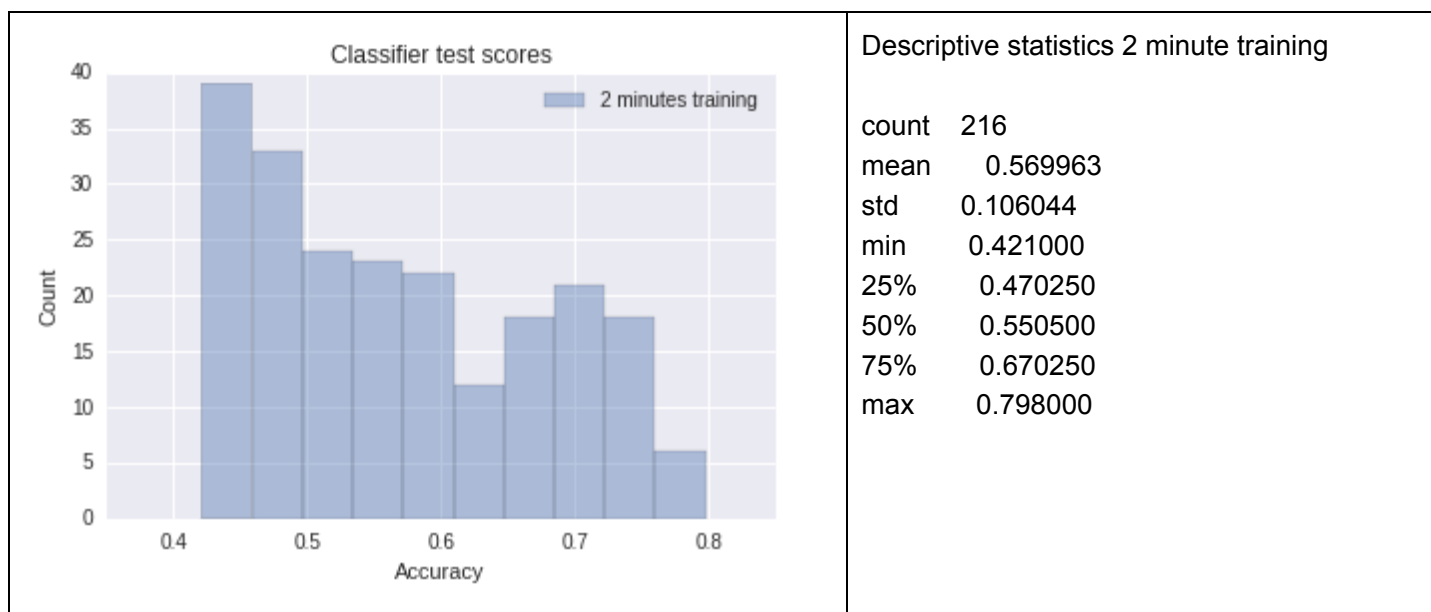
This manifested itself as the classifiers for digits 4 and 5 were always predicting the space character, which was nearly always correct. And as our scores are averaged, the score would always have a base of 40% correct.

Another factor in the removal of digits greater than 3 was that the size of the images were small, there was not enough space for more than 3 digits.

Results

Model Evaluation and Validation

We start by investigating the 2 minute and 5 minute training data, we have 216 results for each. Initial analysis showed one result in the 5 minute dataset had converged to NaN and so I re-ran this model to get a fair result.

Descriptive statistics 2 minute training	
count	216
mean	0.569963
std	0.106044
min	0.421000
25%	0.470250
50%	0.550500
75%	0.670250
max	0.798000

Here we see a histogram of test scores for the 2 minute training dataset. We can see the scores are all between accuracies of 0.42 and 0.80. There is a steady decline in the scores with most frequent being approximately 40 with a score of 0.45 and the lowest having an approximate frequency of 5 with an accuracy of 0.8.

Descriptive statistics 5 minute training

```
count   216
mean       0.694394
std     0.098310
min        0.436000
25%        0.650250
50%        0.725500
75%        0.763000
max        0.822000
```
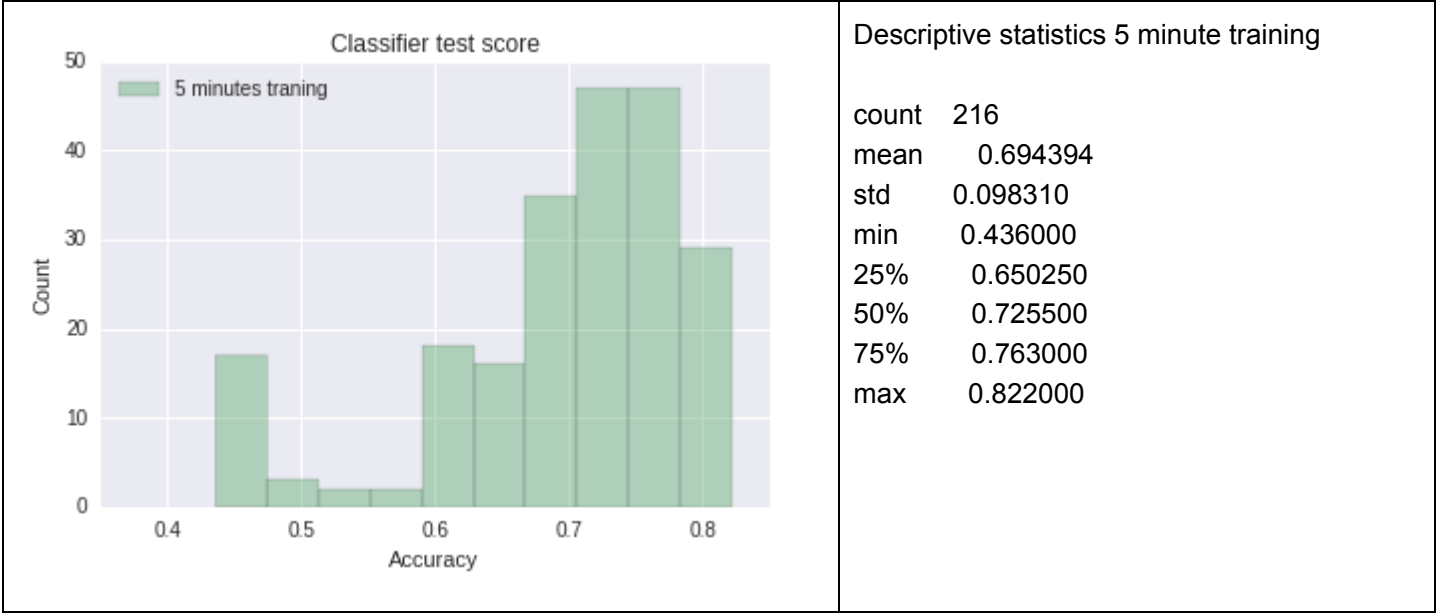
The 5 minute training graph shows a shift the frequencies towards the higher end. There is a small group of approximately 20 outliers below 0.5. The rest of the scores look normally distributed around the accuracy of 0.75 with the highest accuracy recorded being 0.822

Which classifier is the best?

We can now investigate which variables make up the best scoring classifiers. I'll do this by examining the top 5% of scores for each training period.

```
*****************************************
* Top 5% test scores 2 minutes training
*****************************************
Total rows:10

Scores
[ 0.798  0.782  0.778  0.772  0.772
0.761  0.76   0.757  0.751  0.747]

Parameter frequencies
layers
[16, 32]        2
[16]            1
[32]            1
[8, 16, 32]     1
[8, 16]         4
[8]             1
dtype: int64

learning_rate
```

```
*****************************************
* Top 5% test scores 5 minutes training
*****************************************
Total rows:10

Scores
[ 0.822  0.82    0.817  0.817  0.815
0.814  0.812  0.811  0.81    0.81 ]

Parameter frequencies
layers
[16, 32]        6
[8, 16, 32]     3
[8, 16]         1
dtype: int64

learning_rate
0.05     6
0.10     4
dtype: int64
```

```
0.05     1
0.10     9
dtype: int64


num_hidden
32       1
64       2
128      7
dtype: int64


use_dropout
False    10
dtype: int64


use_max_pool
True     10
dtype: int64
```

```
num_hidden
32       3
64       3
128      4
dtype: int64


use_dropout
False    10
dtype: int64


use_max_pool
True     10
dtype: int64
```

We can see some commonalities which make up the best classifiers. Firstly in both time periods, we see that all results have the settings use_max_pool=True and use_dropout=False. We can see that using num_hidden=128 has the highest frequency in both result sets.

The learning rate shows differences where using the higher learning rate showed up 9/10 times in the 2 minute training samples, whereas in the 5 minute samples 0.05 turned up more often with the frequency 6 out of 10.

Lastly we look at the layering of the cnn. The 2 minute training results show high variation, with the most effective being layering of [8, 16] which turns up 4/10 times. In the layers for the 5 minute results we can see that a layering of [16, 32] turned up the most at 6/10.

Best scoring classifier 2 minute period	Best scoring classifier 5 minute period
layers=[8, 16] learning_rate=0.1 num_hidden=128 use_dropout=False use_max_pool=True	layers=[16, 32] learning_rate=0.05 num_hidden=128 use_dropout=False use_max_pool=True

If we look at the best classifier for the 2 minute period we see that it's parameters match those which are reported the most commonly in the top 5%. We notice the same thing when comparing the best 5 minute classifier against it's common parameters.

From this I will select the best classifier from the 5 minute period over the 2 minute classifier as the best performing.

Investigating our best classifier

We can investigate whether our best classifier has finished training by looking at it's loss, and validation accuracy.
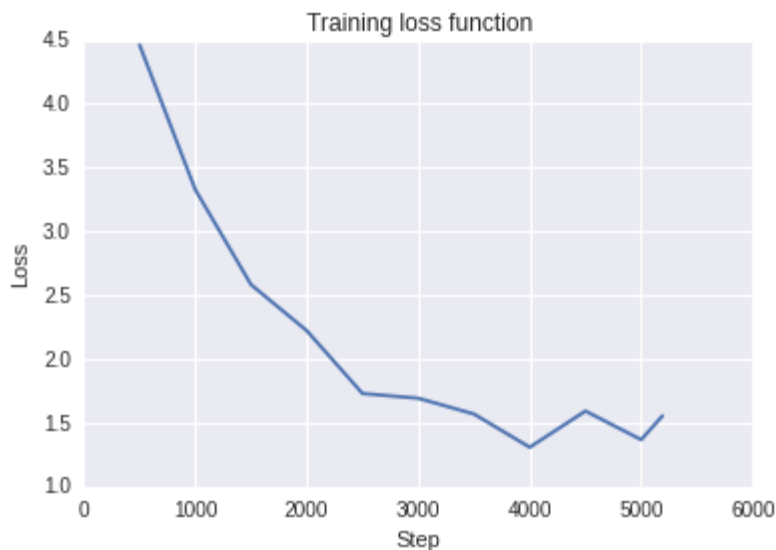
Here we see the loss function drops quickly as expected, but we can see that the loss levels out at a value of 1.5 from step 3500 onwards. This suggests that the classifier is no longer learning well after this period. This could be due to a number of reasons, but a common one would be that the learning rate is to high.

We can investigate if the classifier has overfit by investigating the training, validation and test accuracy.

We can see that the training accuracy is consistently higher than the validation accuracy, which suggests overfitting. This might not be the case though as the sizes of these datasets are considerably different, with the training dataset using batches of 64 images and the validation using 2000.

We can also see that the training and validation accuracy scores level out at step 4000, which matches approximately where our loss functions levels out. We can finally see that the validation score matches the final test score, which due to the size differences in these datasets suggests that the validation dataset is a good representation of the population.

Using the android application

We now will investigate the robustness of our best model. To do this we will load it into the android app and see how well it works.

We'll first examine the app with a number from a runners bib. Straight away we I found high variations in the predictions, with none of them being correct. This is a bad sign that our classifier does will not work well.

After a large amount of alignment our classifier eventually predicts numbers correctly.

| 45x: 0.859 | 52.: 0.469 | 433: 0.746 |

Fig a | Fig b | Fig c
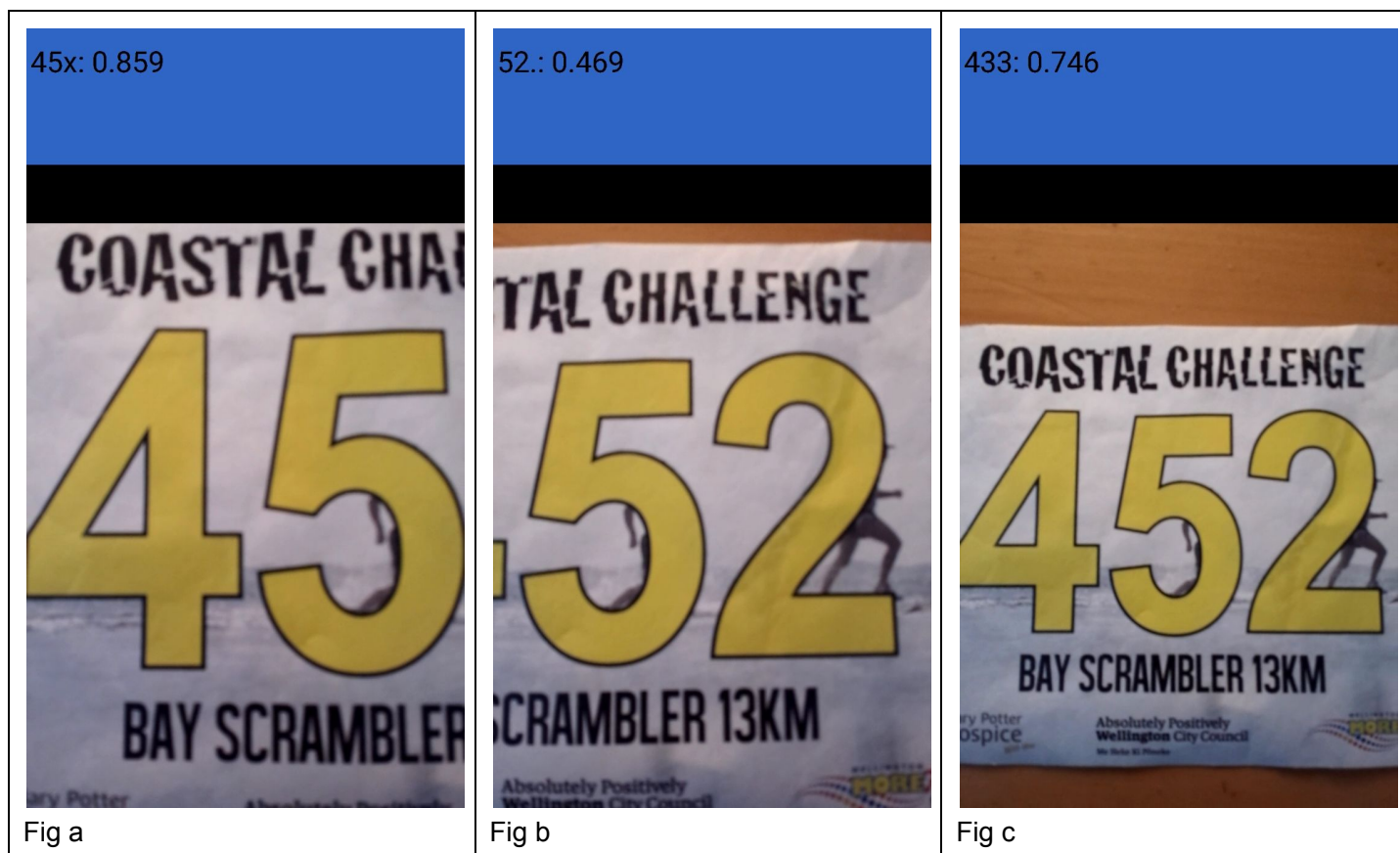
Here we see the classifier working when the numbers are aligned properly. Fig a correctly predicts 45 with a space. Fig b predicts 52 with the last digit not having enough confidence to be shown, and fig c shows that the classifier has got the first digit correct, but the other 2 are incorrect. We can see the zoom has changed considerably in this last image.

We'll now take an easier example and see how it fairs with numbers, black on white.

| 12x: 0.848 | 23x: 0.955 | 123: 0.897 |

Fig d | Fig e | Fig f
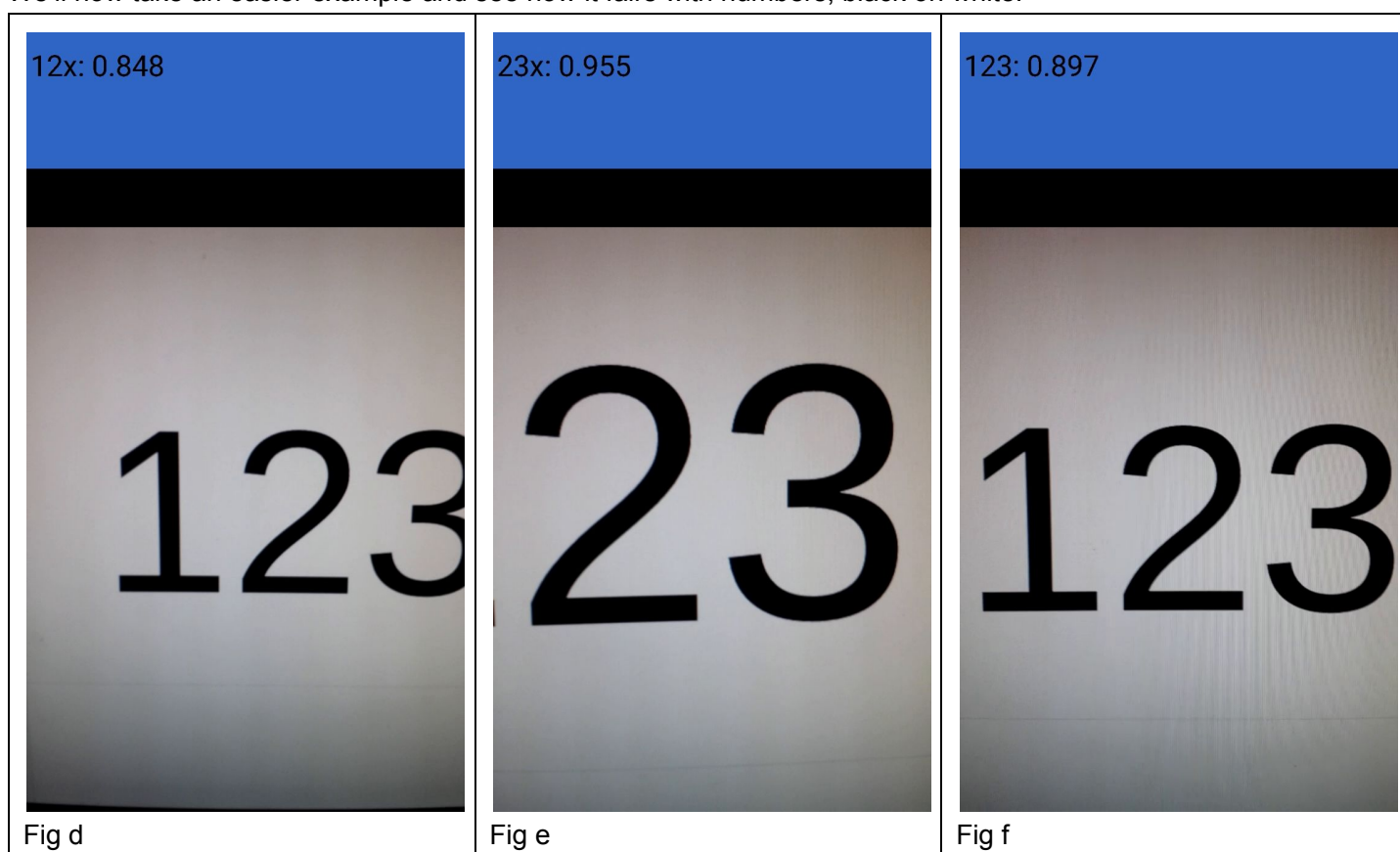
We can see that in fig d it picks up 12, where it could have got 123. In fig e we are very certain it is 23 as noted by the 0.955 acuracy, and in figure f we are fairly sure it is 123.

We can investigate how well the application works in the real world, we've seen it can predict, but lets try the with some random objects.

| .xx: 0.568 | 8xx: 0.975 | 22x: 0.719 |
| Fig g | Fig h | Fig i |
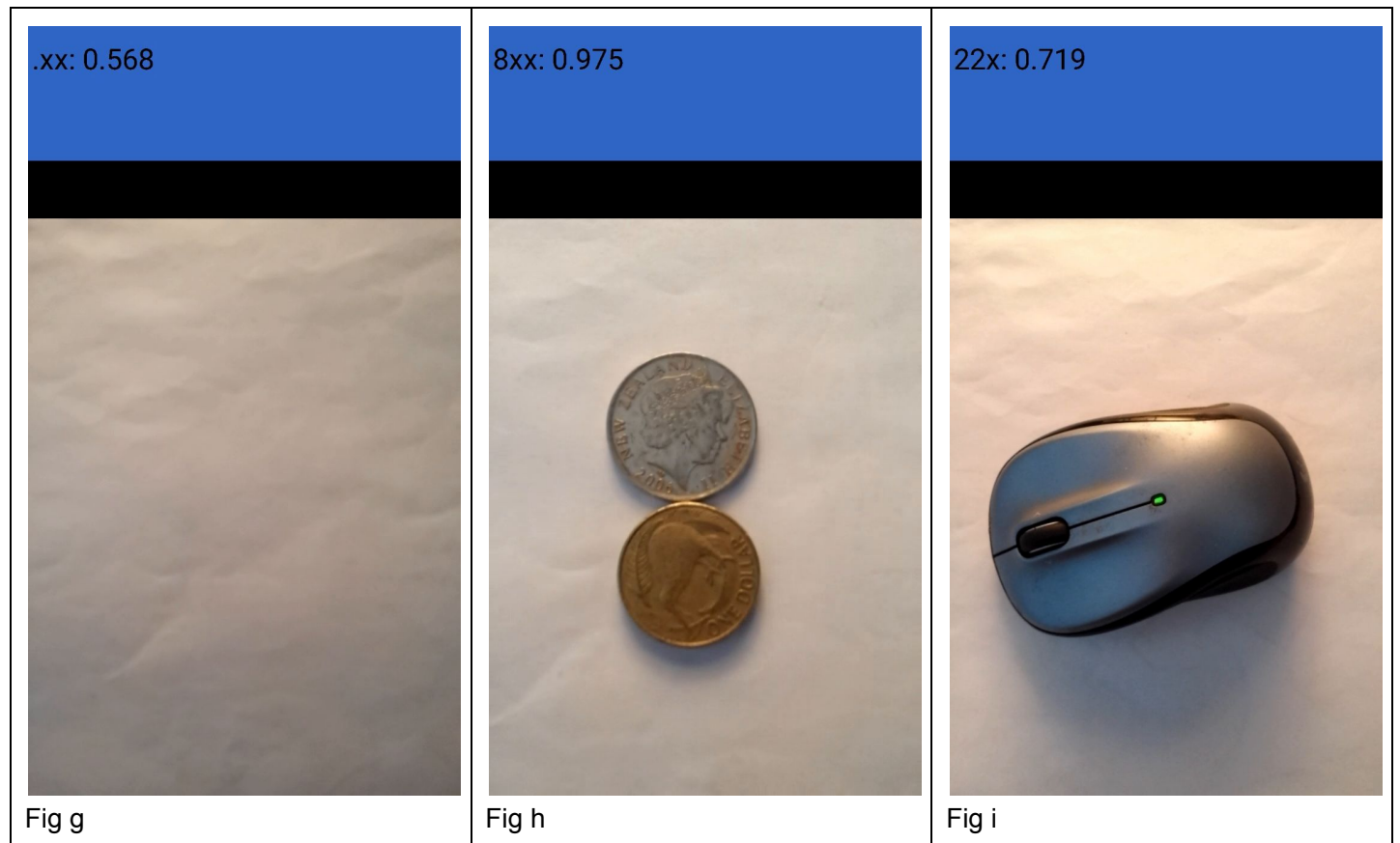
In fig g we can see that the classifier can spot when there is no image, although the first digit is recording no confidence. This brings to attention that each item in the training set was a positive example, i..e all examples had a number to classify. Fig h classifies the number 8 correctly which is great to see, but unfortunately we can see the classifier performing poorly in figure i.

From investigating the application, we can see that in general the model performs poorly. From usage in the android application we can see it does not perform well to unseen data.

Justification

We can finally compare our results against our benchmark. The differences in the parameters is highlighted in **bold**.

Benchmark classifier	**Best classifier**
Accuracy 0.738	**Accuracy** 0.822
Parameters layers=[16] learning_rate=0.05 num_hidden=32 use_dropout=False use_max_pool=False	**Parameters** layers=**[16, 32]** learning_rate=0.05 num_hidden=**128** use_dropout=False use_max_pool=**True**

We can see that our best classifier outperforms the benchmark by 11.4%

We can see that our best classifier has an extra convolution layer of 32 nodes, increases the number of hidden nodes in the fully connected layer from 32 to 128. It also uses max pooling.

Conclusion

From our results we can see that we can train a moderately successful classifier for recognizing digits on standard hardware. Although the android application does not work as spectacularly as hoped, it is amazing that we can train a functional classifier in 5 minutes.

Free-Form Visualization

Here I will investigate the weightings learned by the convolutional neural networks.

Convolutional layer 1 weights   Convolutional layer 2 weights
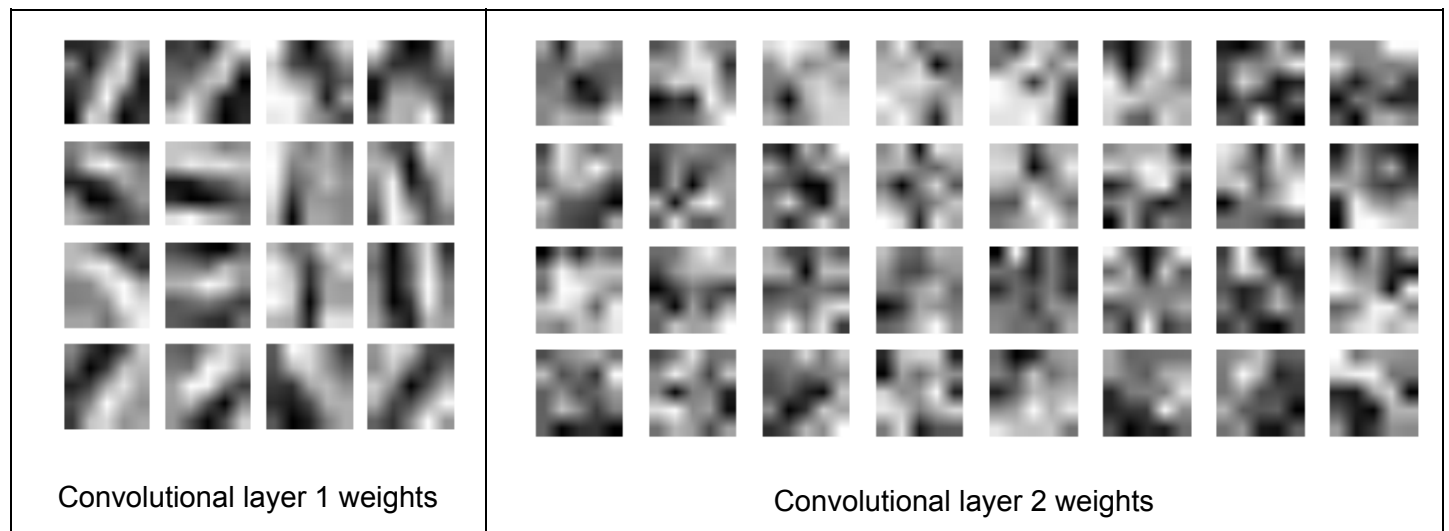
On the left we show the 16 weights in the first convolutional layer. We can see some structure here as many of the images look like lines. These lines indicate the the activations would fire when they found edges in the image, and so we could summarize that the first cnn layer is used for edge detection.

In the second layer we have 32 images, and we can see that the images are noisy and do not have clear structure. This is expected as the steps between the output of the first cnn and the second cnn mix up the picture. These steps are the addition of the biases, relu activation and max pooling.

Reflection

This was a difficult project. Coming into it I had little experience with neural networks and so a large amount of time was put into learning and playing around with them.

The process of building the project was as follows
- Complete the udacity tensorflow tutorials
- Use this code to test convolutional neural networks
- Get a feel of what works and what doesn't for cnns
- Setup tensorflow and build the demo android application
- Limit the scope of the project to something achievable
- Build the code to run the hyperparameter search
- Record results and write the report

My final project was built on top of many failures and wrong directions.

Hurdles building the classifier

Building the software for the classifier was an evolutionary process. To get started I copied the udacity tutorials from tensorflow and started to modify them. I had many failings in my initial work. I first tried synthesizing numbers from a series of digits, which caused a lot of grief as I used image sizes that were far too large. This caused slowdowns and crashes in my computer. In my second attempt I tried to use the images from the SVHN dataset, but again I came across the problems of image sizes being too large. After I realised this I went for very small image sizes, which I could get some comparable results from.

Another realisation I had in the initial research phase was that there were so many variables to make up a graph that a hyperparameter search will be the only way to find what works well. This would require running many classifiers, and could potentially take a very long time, so I decided to limit training time to bring some sanity to the project.

Hurdles building the android application

Our goal to build an android app was helped by the presence of demo code included in the tensorflow github repository. The task of getting this building was tough as there were a large number of dependencies required. These included the bazel build system, android sdk/ndk,loading of the inception model, and finally updating the WORKSPACE file in tensorflow to join all of this together This was not an easy task. I initially failed by using the wrong android API.

The most frustrating part of the project was that there was a major bug in the android camera api in android 5.0.1. This bug went unnoticed, as the android app built and installed correctly. I picked it up when I started logging the images to the sdcard, and could see that they were all green. Details of the bug are listed here bug. To overcome this I had to root and flash a new image onto my android device. After this the logged images were full color and were working correctly.

Improvement

My project could be improved upon in many ways. One improvement would be to create a classifier that is not limited by the number of digits. The classifier I created requires a classifier for each output digit. This works fine when we have a small number of digits, but I can imagine a common scenario would be to read a long number of digits such as those on a credit card.

A way to implement this would be to use a different approach such as using a recurrent neural network. This kind of network could be used to read digits of arbitrary lengths, by using the network in a similar way that we read. In this I mean to run a classifier across an image from left to right, and output each digit when it reads across it.