Deep Learning Capstone Project
Digit Recognition in Natural Scene Images

§1 Introduction

§1.1 Project Overview

Optical character recognition (OCR) is the electronic conversion of images of typed, handwritten or printed text into machine-encoded text. This could for example be from a scanned document, a photo of a document or a scene-photo (for example the text on signs or billboards in a photo). Some applications include,

- Digitisation of printed or handwritten documents so they can be stored more compactly and are searchable
- Recognition of handwritten postal codes for automated mail sorting
- Recognition of vehicle registration plates for the automated production and distribution of penalty charge notices in the event of parking/moving traffic contraventions
- Automatic transcription of addresses from geo-located photographs of house numbers [2]

In this project, we design a system which, given a natural image of a digit, is capable of recognising the digit at the centre of it. It is intended as a first step to being able to recognise multi-digit numbers. To perform the digit recognition, we use deep (many layered) Neural Networks and more specifically TensorFlow - an open source software library for numerical computation using data flow graphs, particularly suited to machine learning with Deep Neural Networks.

To train and test our digit recognition system we work with the Street View House Numbers (SVHN [1]) dataset obtained from house numbers in Google Street View images. SVHN contains over 600,000 labelled digit images.

The data comes in two formats:

1. Original images with character level bounding boxes.
2. 32 x 32 x 3 (colour) images centred around a single character

In this paper, we focus on the latter dataset. The images are divided into three sets each provided in a separate file. The three sets are described on the website as follows:

1. 73,257 images and labels in test_32x32.mat “for training”
2. 26,032 images and labels in train_32x32.mat “for testing” and
3. 531,131 images and labels in train_32x32.mat “additional, somewhat less difficult samples, to use as extra training data”.

Because the images are taken from real photos of house numbers many of the images contain distractors at the sides (more than one digit in the image or partial digits surrounding the central digit).

§1.3 Problem Statement

In this paper, we design a model which performs optical digit (0-9) recognition from images in the 32 x 32 SVHN dataset.

Our strategy in solving this problem has followed the steps outlined below:

Data download, extraction, processing and exploration:

Our general approach is to process incrementally and check how our data looks at each stage

1. Download the data files to organised folders (0_SVHNDownloadExtract.ipynb).
2. Extract our data from the matlab file format it is provided in, to numpy arrays, a format which allows us to explore and process the data (1_SVHNExploreProcess32x32.ipynb).
3. Explore our data to understand the problem better:
    a. check the distribution of digits in the images to get a lower bound for our performance metric;
    b. take a random sample of 10 digits (one from each class) from each of the three datasets (train, test and extra) and look at them to check we have transformed the data to numpy arrays correctly and to get an idea of what difficulties we might encounter.
4. The dataset is big - there are in total 630,420 colour images, each one is a 32 x 32 x 3 array - we convert the images to greyscale, reducing the size of the dataset to a third of its original size and compare with their colour counterparts.
5. Normalise the dataset so it has approximately mean zero and standard deviation one to make our optimisation in training more stable.
6. Save our processed data to a format (pickle) which makes it possible to access our numpy arrays later.

Model training and prediction:

Our general approach is to start with a simple model and increase its complexity incrementally checking the performance at each stage. This approach allows us to understand the benefit of each 'bell and whistle' we add and makes finding an initial solution at each step easier

7. We start by looking at the 'extra_32x32' dataset which contains 531,131 "somewhat less difficult samples" and for interest we later see how our model performs on the train_32x32 and test_32x32 datasets
8. We first train an out-of-the box classifier (`sklearn.linear_model.LogisticRegression`) to see how it performs
9. We then move to a two-layer Neural Network
10. We add as many layers as we can, given the limitations of our computational power
11. We add techniques for preventing overfitting (L2 regularisation and Dropout [3]) if necessary

§1.2 Metrics

For our problem, we are only interested in predicting digits correctly and we consider no wrong prediction to be less bad than another. Because of this, the metric by which we will judge the performance of our model is simply accuracy,

$$Accuracy = \frac{Number\ of\ samples\ predicted\ correctly}{Total\ number\ of\ samples\ predicted}$$

in other words, the proportion of samples we predict correctly.

§2 Analysis

§2.1 Data Exploration

Each sample of our data consists of a colour image and its corresponding label. The samples are provided in three separate files,

1. 73,257 samples in train_32x32.mat,
2. 26,032 samples in test_32x32.mat,
3. 531,131 samples in extra_32x32.mat.

Because the images are taken from real photos of house numbers many of the images contain distractors at the sides (more than one digit or partial digits surrounding the central digit). Each image is a 32 x 32 x 3 array and the corresponding label is a single value between 1 and 10 inclusive. The label 1 corresponds to the digit 1, the label 2 corresponds to the digit 2 and so on until 9 and finally the label 10 corresponds to the digit 0.

In Figures 1 to 3, we take a random sample of 10 images (one from each class) from each of the three datasets and display them along with their labels. It's not immediately obvious just by looking at the samples why those contained in the extra_32x32 set are described as "easier"[1] and the datasets are too large to search for outliers manually so we take their word for it. We use all the images in the extra_32x32 dataset to train and test the performance of our model. At the end, we check how our final model performs on the other two datasets. We notice the distractors, we have samples which contain multiple digits in the same image, the label corresponds to the central one.

Figure 1: Random sample of images and corresponding labels from the test_32x32 dataset

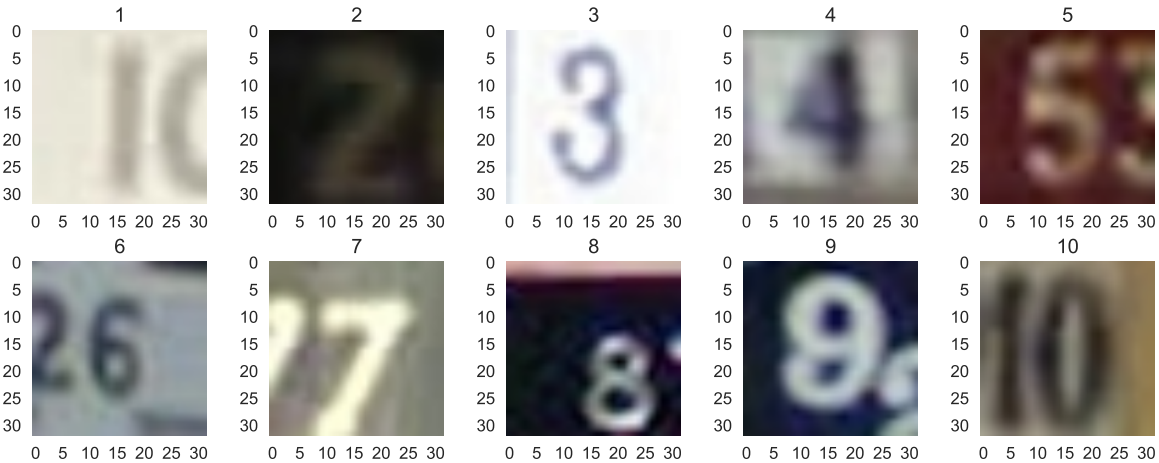

Figure 2: Random sample of images and corresponding labels from the train_32x32 dataset

[1] According to [1] all the images were produced using a dedicated sliding window house numbers detector which has a user specified threshold on the confidence score. The images in the extra_32x32 dataset were produced using a considerably higher detection threshold. We might expect that this translates to problems with the size, location or orientation of the digits in the image, the resolution of the image, the number or extent of the distractors, the relative colours of the central digit compared to the background among others but such issues aren't observed consistently enough in our selected samples from test_32X32 and train_32x32 when compared to extra32x32 to be able to separate them as they have been.

Figure 3: Random sample of images and corresponding labels from the extra_32x32 dataset

§2.2 Exploratory Visualization

Here we outline the algorithms and techniques we have used in solving our digit recognition problem.

- The main component of our model is a Deep Neural Network
- To find the optimal weights and biases for our model we use Stochastic Gradient Descent
- To initialise our weight matrices, we use Xavier weight initialisation
- To prevent over-fitting we use two different methods
    - L2 regularisation
    - Dropout [3]

§2.3.1 Deep Neural Networks

At a high level, a basic Neural Network (not Convolutional or Recurrent Neural Network) consists of layered function compositions. Each layer is essentially a linear transform with a non-linear (activation) function applied to it. Deep Neural Networks are simply Neural Networks where the number of layers is large. Though Neural Networks have been studied since the 80s it's only more recently that we have seen them gain in popularity thanks to the rise of GPU accelerated computing, the availability of large enough datasets to train them on and improvements in regularisation techniques. The general approach is to use a model architecture which is much bigger than you need and then use regularisation methods to deal with over-fitting. This might seem rather wasteful from an efficiency perspective but it works. Today, Deep Learning provides state-of-the-art solutions to computer vision and speech recognition problems.

For our problem, not only do we have a very large dataset to train on but we also know that Convolutional Neural Networks were recently used successfully in multi-digit number recognition using the SVHN full numbers dataset [2].

§2.3.2 Stochastic Gradient Descent

To calibrate (find the weights and biases) our Neural Network model we have to solve an optimisation problem which involves minimising a loss function. The method we use to do this is Stochastic Gradient Descent.

Gradient Descent is a numerical method for finding the minima of a given function. The basic idea algorithm is as follows:

1. Make an estimate (initial guess) for the location of the minima.

2. Calculate the function at the point we estimate to be the location of our minima to see how close we are
3. Calculate the slope (gradient) of the function (with respect to all its variables) at the point
4. Update our estimate for location of the minima with a new point which is a step of size learning rate in the direction where our slope is steepest downward (most negative)
5. Repeat steps 2 to 4 either for a fixed number of steps or until the value of our function is small enough that we feel we are close enough to the minima

Computing the gradient at each step for large systems like ours is computationally expensive enough to be prohibitive. The cost of computing the gradient is three times that of computing the loss. To solve this issue, we instead use Stochastic Gradient Descent. Rather than calculating the gradient for the full dataset at each step we calculate it for a random subset and use this to approximate the gradient. This makes our descent somewhat more erratic so we require an order of magnitude more steps to optimise but the gain in speed is significant.

A nice side effect of Stochastic Gradient Descent and other stochastic optimisation methods is their inherent ability to avoid overfitting in the models they optimise. Estimating our gradients using a random sample of our training data means that each step we are 'fitting' to a different random subset rather than the full training set. This means we take longer to optimise because we end up taking a more round-about route possibly ending up elsewhere altogether. Interestingly, a side effect of not being able to find the optima of the full dataset is a model with better generalisation properties.

§2.3.3 Xavier weight initialisation

With increasing depth of your Neural Network, the way one initialises the weights becomes increasingly important. If the weights are too small the output at each layer tends to diminish while if the weights are too large, they tend to explode. The explanation for why this happens assumes that one has used sigmoid activation functions at each layer. The reasoning is as follows. If the weights are too small the activations will be close to the central part of the sigmoid function (where it is approximately linear) and we lose the any benefit of having multiple layers. On the other hand, if the output at each layer is too large, then the sigmoid function becomes flat reaching saturation, at this point our gradients become zero making our activations meaningless.

Using Rectified Linear Units (RELUs) as your activation function supposedly resolves this issue to some extent, since it is non-linear at zero and does not saturate for large positive values. However, in general we found that using Xavier weight initialisation was the difference between finding a good initial solution one could improve on and getting results where the loss exploded giving accuracies which were worse than guessing.

The procedure we used in our deep networks was to initialise our weights randomly from a truncated normal distribution and use Xavier's method to determine the variance of that distribution. The basic idea is to choose the variance of your weights such that applying the linear transform in that layer does not change it.

§2.3.4 L2 regularisation

One of the reasons for the success of Deep Neural Networks is the richness of the representations it is capable of displaying. This comes from the immense number of parameters we are using to fit our data. In our problem, the network has to learn not only to recognise digits but also ignore any distractors there might be and we demand a high level of accuracy of prediction. It also has to be able to fit to a huge training dataset.

With richness of representation comes the issue of overfitting. To combat this issue, we use regularisation. The general idea is to impose artificial constraints on our network that reduce the number of parameters without making it harder to optimise. L2 regularisation penalises large weights by adding a constant multiple of the sum of squared weights to the loss function.

§2.3.5 Dropout

Dropout is a recently published [3] technique that works very well at preventing overfitting. The basic idea is that at a given hidden layer one randomly zeros a fixed proportion, p, of the activations and scales all the others up by $1/p$ to ensure that on average the output of that unit remains the same. By losing activations randomly one forces the network to learn redundant representations, whilst this sounds inefficient, it actually makes the network robust since it learns not rely on the information provided by any given unit.

§2.4 Benchmark

We saw in Figure 4 that the distribution of digits in our data was unbalanced and that the digit 1 in all three datasets was the most frequently occurring label. The frequency of this digit then provides us with a lower bound for the accuracy of our model since employing a strategy of always predicting the digit 1 (regardless of the input image array) will result in an accuracy equal to the frequency of that digit in the dataset.

Human accuracy on the test_32x32 dataset is estimated to be around 98% [1]. Human performance unsurprisingly decreases with the size (height) of the digit in pixels. The main causes for human errors in the digit recognition task were found to be foreground/background confusion and low resolution/blurred images.

§3 Methodology

§3.1 Data Preprocessing

The first step in processing our data is to change the labels of the images of the digit zero from ten to zero so that our labels match the digit in the image.

Next we convert our images to greyscale, the main purpose being to save on computational expense. We found the size of the images large enough to cause issues with the memory while pre-processing[2] so converting to greyscale was an easy gain.

Greyscale images are produced by taking the weighted average of the red, green and blue values for each pixel. Different methods use different weights. The method we chose takes a simple average (all weights being one). Figures 1 to 3 show random samples of 10 images (one from each class) from each of the three datasets and display them along with their labels. Figures 5 to 7 show the greyscale counterparts of the same images.

[2] Note, 1_SVHNExploreProcess32x32.ipynb uses in excess of 32GB of memory, it is true that the sheet can be written somewhat more efficiently but since I have a machine that can cope this load, this was not necessary

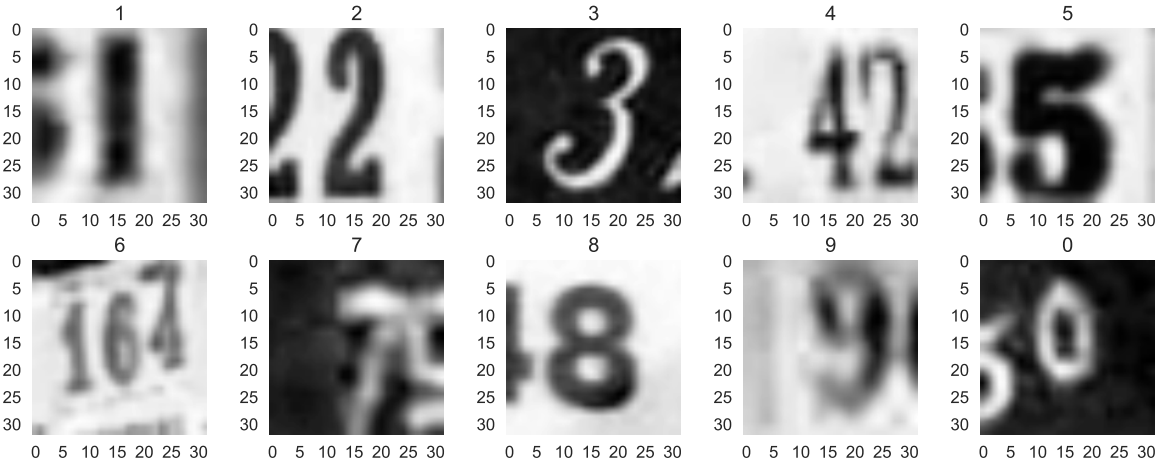

Figure 5: Random sample of greyscale images and corresponding labels from the train_32x32 dataset

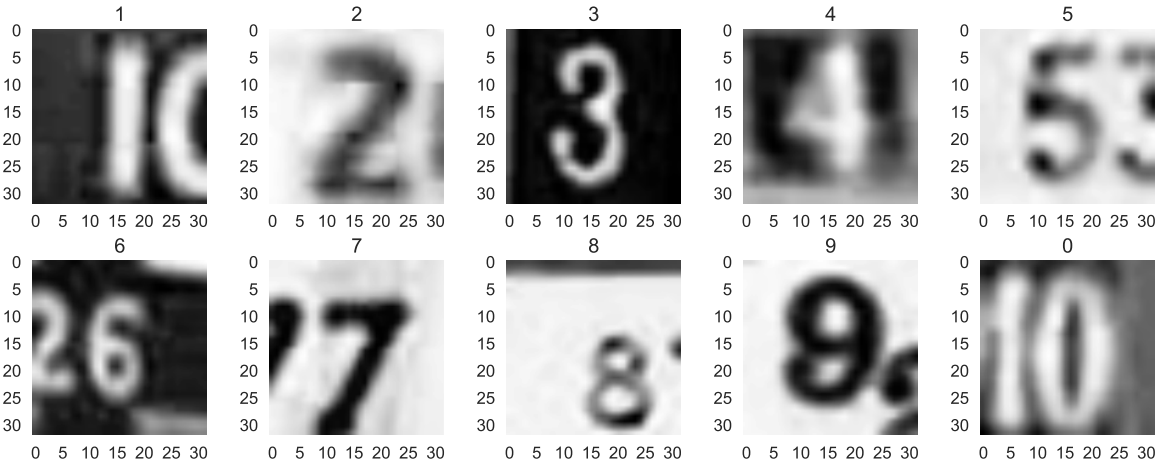

Figure 6: Random sample of greyscale images and corresponding labels from the test_32x32 dataset

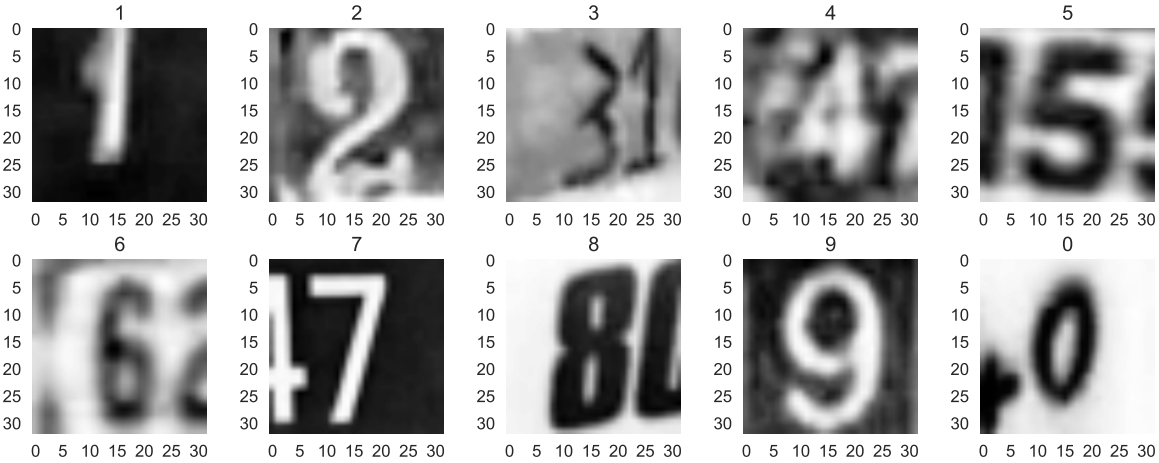

Figure 7: Random sample of greyscale images and corresponding labels from the extra_32x32 dataset

Next, we recentre and rescale the values in our image arrays so the full dataset has mean zero and standard deviation one using:

```
def normalise (image_array, mean, stdev):
    return (image_array-mean) / stdev
```

This is to ensure our optimisation problem is well specified and avoid numerical instability during training. In Tables 2 and 3 we show the mean and standard deviation of the greyscale images in each dataset before and after normalising.

	train_32X32	test_32x32	extra_32x32	total
Mean	115.112	116.781	110.589	111.370
Standard Deviation	49.4769	56.2277	48.7486	50.4876

Table 2: Mean and standard deviation of the greyscale image arrays before normalising

	train_32X32	test_32x32	extra_32x32	total
Mean	0.07610	0.11006	-0.01589	-1.143e-14
Standard Deviation	1.00635	1.14366	0.99154	1.000000

Table 3: Mean and standard deviation of the greyscale image arrays after normalising

Figures 5 to 7 show random samples of 10 greyscale images (one from each class) from each of the three datasets and display them along with their labels. We note that normalising the greyscale images has no observable impact on their appearance.

Finally, we randomly shuffle our datasets to ensure that when we separate them into training, testing and validation sets the distributions of labels in each are similar.

§3.2 Implementation

Our general approach has been to use existing library implementations of functions as much as possible. In our work, we have made use of tensorflow, numpy, pandas, matplotlib, pylab, seaborn, scipy, sklearn among others and were able avoid writing complicated algorithms.

The main complications we encountered during implementation were to do with the size of the data we were dealing with. Just processing the data used over 32GB of memory and once the model became complex enough, optimising the model parameters became very computationally expensive and we were constrained by our hardware.

Another complication was the TensorFlow documentation which is somewhat opaque and alone not enough to be able to use the library. Because it is a new and fast evolving library, examples on the internet are not necessarily for the version of the software one is running. We had particular issues with saving our trained model to make predictions outside of training. We also found making predictions outside of the model very slow.

§3.3 Refinement

Before one can optimise the parameters of a Neural Networks to fit their training and validation data one first must find an initial solution to actually improve on. The large numbers of parameters in a Deep Neural Network can make this difficult. Our general approach to solving this problem is to increase the model complexity incrementally. As the model grows in complexity so do the number of parameters which require tuning. Incrementally increasing the complexity means new parameters are introduced gradually which helps make it easier to find a first solution at each step.

To start we take the extra_32x32 dataset (of supposedly “easier” examples) which contains 531,129 samples and split it into three parts. We use approximately 5% (26,556 samples) of it for validating our model (tuning parameters) and another 5% (26,556 samples) as a test set (to gauge the performance of our model on ‘unseen’ data) and the remaining 478,017 samples for training. Figure 8 shows the distribution of digits in our training, validation and testing datasets. From this we see that we have

approximately 17% as a lower bound for our accuracy (the accuracy we get if we always predict the most popular digit, 1).

	train data	valid data	test data
0	0.085886	0.084425	0.084840
1	0.170339	0.172127	0.171863
2	0.140792	0.140496	0.139592
3	0.114301	0.117224	0.113496
4	0.095279	0.094743	0.096852
5	0.100837	0.098885	0.100241
6	0.078311	0.079379	0.076819
7	0.082884	0.082543	0.082241
8	0.066537	0.067103	0.066652
9	0.064832	0.063074	0.067405

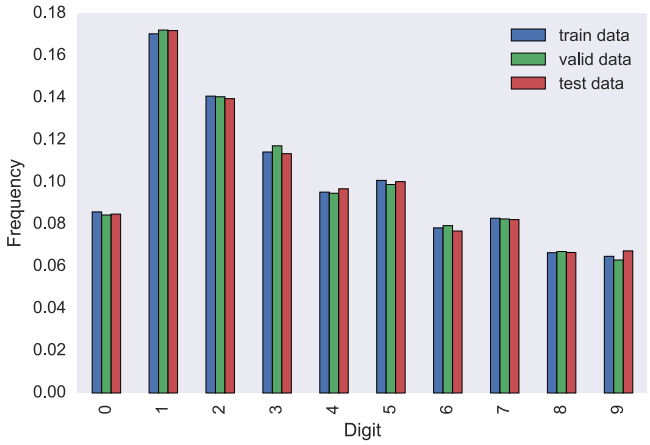

Figure 8: Distribution of labels in each of the three datasets

§ 3.3.1 Logistic Regression

The first model we used to solve our digit recognition problem was an out-of-the box logistic classifier (`sklearn.linear_model.LogisticRegression`). Looking at the performance of this model shown in Figure 9, it is clear that it does not display the richness of representation to be able to perform well on our dataset. Judging by the trend of the curves that increasing the number of training examples will not increase the accuracy on our validation data above 30%. With 20,000 training samples, we were able to get an accuracy of approximately 22.5% (a 5.5% improvement on our lower bound) on both our validation and test sets. We also note that our test and validation sets give very similar accuracy results indicating they are statistically similar.

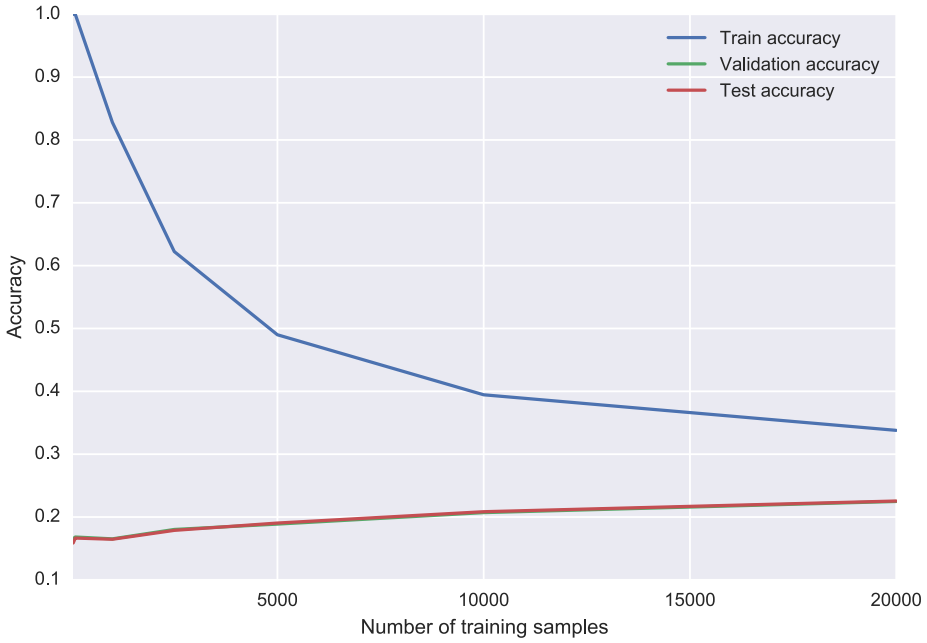

Figure 9: Accuracy results for `sklearn.linear_model.LogisticRegression` with increasing training sample size

§3.3.2 Two-layer Neural Network

Our next step was to train a two-layer Neural Network, essentially adding a hidden layer to an ordinary linear classifier model. We chose to use 1024 hidden nodes and use Rectified Linear Units as our activation function. We started with a flat learning rate and initialised our weights from a truncated standard normal distribution and used a flat learning rate. By using Xavier weight initialisation and varying the learning rate we were unable to find an initial solution. The loss function would either not decrease or explode. Employing an exponentially decaying learning rate we were able to find an initial solution giving approximately 85% accuracy after 3,000 steps. From here, keeping all else fixed we experimented with optimising the learning rate and changing the number of steps in our optimisation.

Exponentially decaying learning rate

We use TensorFlow's `tf.train.exponential_decay` to apply an exponentially decaying learning rate. The implementation of this function is,

```
decayed_learning_rate = learning_rate
                        * decay_rate ^ (global_step / decay_steps)
```

We see that there are three parameters here to tune just for the learning rate[3]. We optimise only two of these, `learning_rate` and `decay_rate`. From the above formula, we know that varying `decay_steps` has a similar effect on `decayed_learning_rate` to varying `decay_rate`. Figure 10 shows the `decayed_learning_rate` function for a variety of parameter choices.

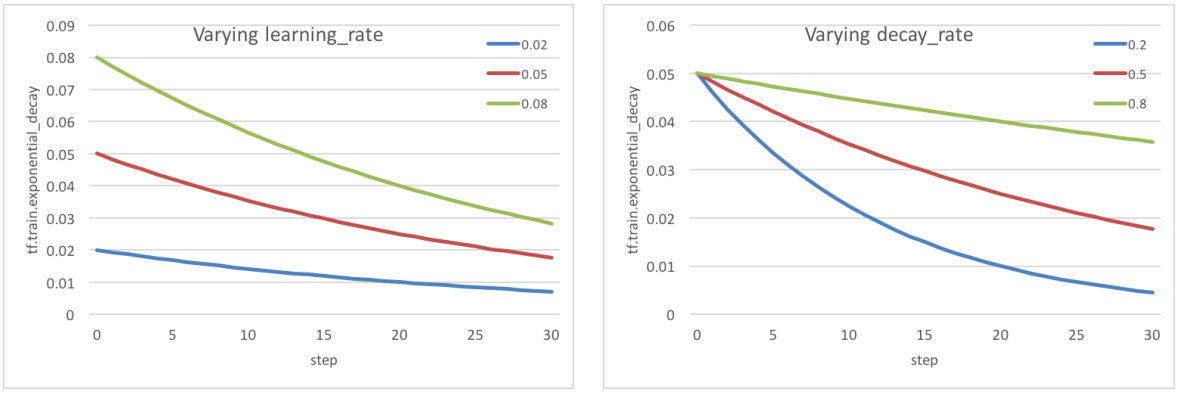

Figure 10: Behaviour of `tf.train.exponential_decay` for a range of values of the parameters. In each graph the red line is the function for `learning_rate = 0.05`, `decay_rate = 0.5` and `decay_steps = 10`

Through trial and error, we were able to increase our accuracy to 88%. After this we tried optimising for each of our two learning rate parameters separately. First optimising the initial learning rate, picking the best one and then optimising the decay rate. By doing this we were able to increase the accuracy on our test and validation sets to 89%.

Finally, we tried optimising over grids of values for the two learning rate decay parameters using 3,000 steps, picking the parameters which gave the best result and then increasing the number of steps to 10,000 to get a final solution. We found that such a strategy did not give the best performing model as there is interaction between the learning rate and number of steps. We then decided to use 10,000 steps on a larger grid to see what the validation accuracy surface looked like. This was extremely time consuming, nevertheless, using this method, we were able to increase our accuracy on the test set to 90%. Figure 11 shows our results for our best learning rate parameter choices.

[3] In reality there are more, one can choose to implement the decay as a step function where `decay_steps` determines the frequency of the step. In our implementation, we fix `staircase = True` to reduce the number of parameters which require tuning

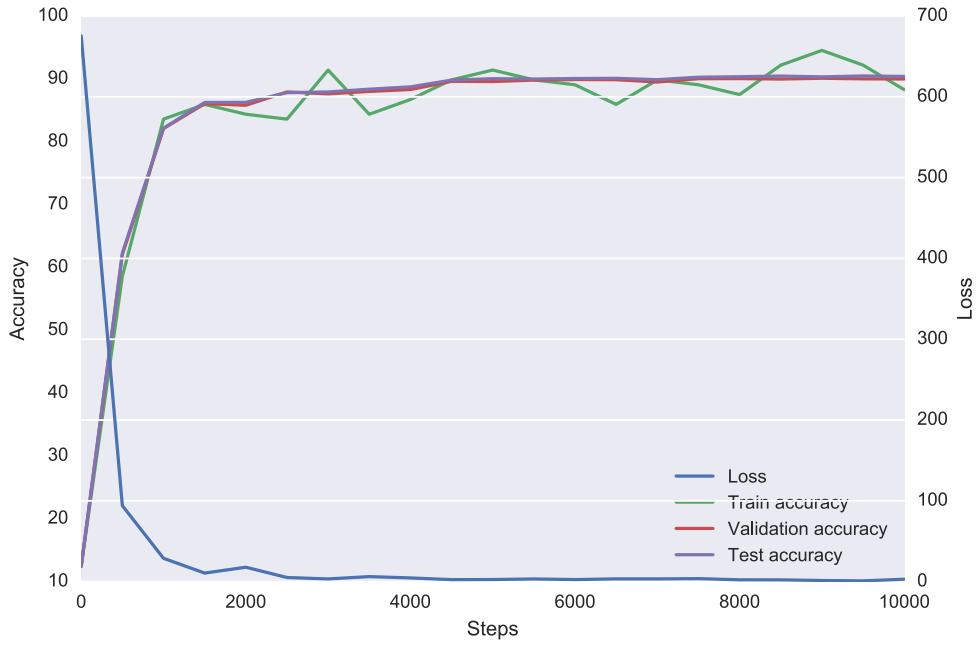

Figure 11: Loss and accuracy against the number of steps for our optimal learning rate in a two-layer Neural Network

We note that at this stage our model does not seem to have an issue with overfitting. In particular, we can see from Figure 11 that there is not a significant difference between the accuracy we see on our training, validation and testing sets.

§3.3.3 Three-layer Neural Network

Next we add another hidden layer. This time we setup an architecture with 2048 and 512 hidden nodes in the first and second hidden layer respectively. We used Xavier weight initialisation and otherwise keep our parameter setup the same as in our optimal solution using two layers. With this setup, we are able to achieve 93% accuracy. Figure 12 shows the results with this setup.

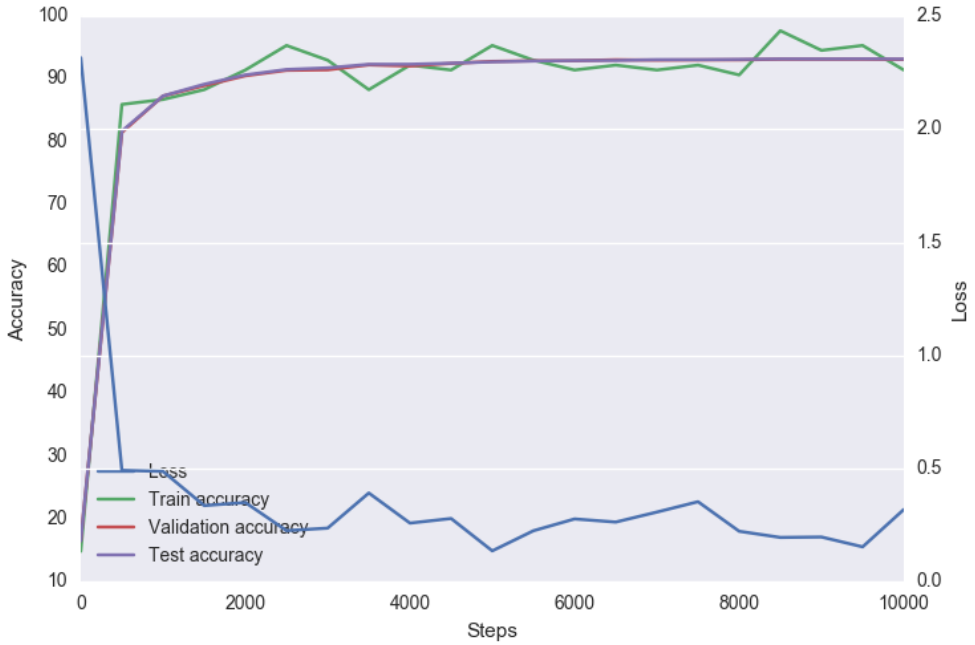

Figure 12: Loss and accuracy against the number of steps for our three-layer Neural Network

We notice a big gain in accuracy (3%) with very little effort, here. We've made minimal changes to the code and spent no effort optimising the learning rate again. All we have done is add a layer to our Neural Network and used the learning rate parameters that we optimised for our two-layer Network. We still observe no signs of over-fitting.

As a next step this time we go straight ahead and increase the number of layers again rather than trying to optimise the learning rate.

§3.3.4 Four-layer Neural Network

This time we add another hidden layer between our previous two with 1024 hidden nodes i.e. we have an architecture with 2048, 1024 and 512 hidden nodes in that order. Again, we used Xavier weight initialisation and otherwise keep our parameter setup the same. With this setup, we achieve only a very small gain in accuracy over our three-layer model, around 0.2% and with no sign of overfitting there is nothing to be gained by adding L2 regularisation or dropout. Because of this minimal gain in accuracy, we chose the three-layer Neural Network as our final solution.

Figure 13: Loss and accuracy against the number of steps for our four-layer Neural Network

§4 Results

§4.1 Model Evaluation and Validation

Our final trained model is a three-layer Neural Network with 2048 and 512 hidden nodes in the first and second hidden layers respectively. We arrived at this model by incrementally increasing the complexity, starting with a logistic regression, then training a two-layer Neural Network and finally using our best parameters in a three-layer model. We were able to get a slight improvement by adding another layer but decided that the gain in accuracy was not worth the additional computational expense. We optimised our model parameters based on the results of our validation dataset and evaluated its performance on an 'unseen' test for each model to be sure. For all our Neural Network models, we found that the training, validation and test sets gave similar results indicating that we are not over fitting. We did not use k-fold cross validation because our dataset was large and the computational expense would have been too great.

With our final model, we have been able to achieve an accuracy of prediction on our 'unseen' test set of over 93%.

To test the robustness of the model we use it to make predictions on data from the test_32x32dataset. These are described as harder examples because they were produced using a lower detection threshold on the sliding window house numbers detector than the images we have trained our model on [1]. Sure enough, our model performs worse on these images, we get an accuracy of 86.4%. This tells us that the method used to detect the digits must be consistent for our accuracy measures to be reliable.

§4.2 Justification

In section 2.4 we gave two benchmarks for prediction accuracy, a lower and upper bound. From Figure 4, the lower bound for the data in train_32x32, test_32x32 and extra_32x32 were 18.9%, 19.6% and 17% respectively. This is the accuracy we would get if we always predict the most frequently occurring digit in

the dataset (1). To get an idea of what an upper bound might look like, [1] estimates human accuracy on the test_32x32 dataset to be 98%.

Using Neural Networks, we have been able to get an accuracy on our digit recognition task of 93%. When applied to "harder" dataset test_32x32 we were able to get an accuracy of 86.4%. While the results are significant progress on the lower bound, human accuracy is still some distance from where we are and likely for any real world applications human accuracy or better would be required as a minimum.

§5 Conclusion

§5.1 Free-Form Visualization

Figure 14 shows the accuracy achieved on our test set against the number of layers in our Neural Network along with our lower and upper bounds for accuracy (described in section 2.4). Note that here we have extrapolated the results for our one layer Neural Network (logistic regression) which was only trained on 20,000 training points. Bearing in mind that for our three and four layer models we did not optimise the model parameters at all, this graph gives an idea of the potential of Neural Networks, given enough data and enough computational power.

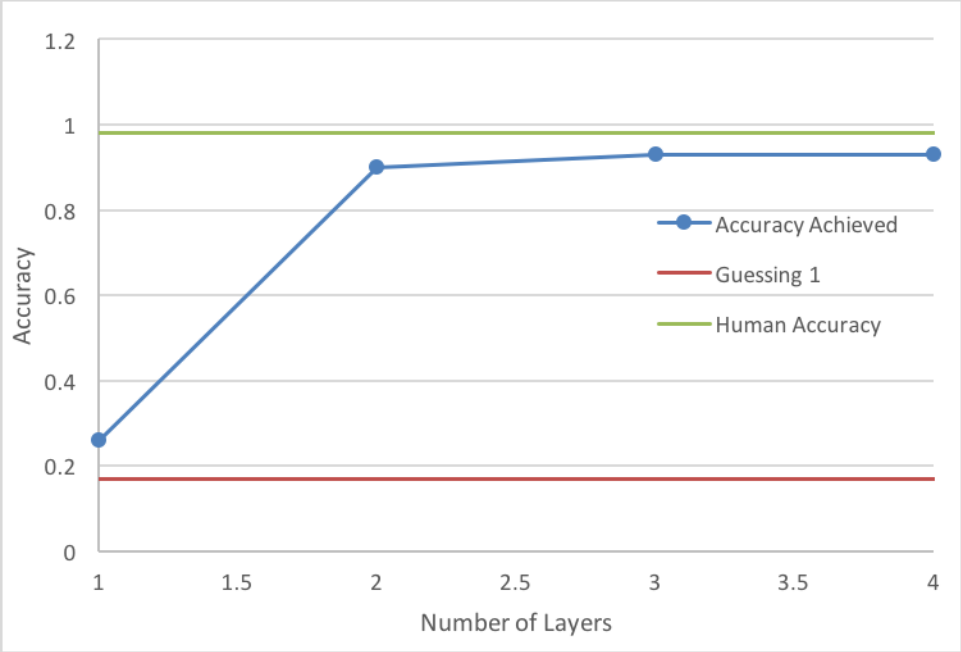

Figure 14: Accuracy as a function of the number of layers

§5.2 Reflection

In this project, we have used Deep Neural Networks to perform optical character recognition on the SVHN 32 x 32 dataset. The first step in solving this was to reduce the size of our dataset by reducing them to greyscale. We then started with a simple linear regression model moving to a two-layer Neural Network and gradually increased the complexity of our Network improving the accuracy of its predictions at each stage.

The main difficulty of Deep Neural Networks is the sheer number of parameters all of which can have a significant impact on the performance of the model and so need to be optimised. For a basic Neural Network model with many layers, we have a choice over the initial weights and biases; there are also architecture related choices to make, like the number of hidden layers and the number of nodes in each

layer. There are parameters related to the learning rate, a decaying learning rate is typical for which there are at least two parameters, the initial learning rate and learning decay rate, but possibly more[4]. Then there are parameters related to overfitting prevention, the regularisation coefficient and if you use dropout the probability of keeping any given output from each layer. Even the number of steps in your optimisation is important. Many of the parameters are not independent, i.e. optimising one while keeping the other fixed does not result in an optimal solution. We observe strong interdependence between the two learning rate parameters and the number of steps in the optimisation.

Another issue with optimising is the computational expense because our data is large and choosing the parameters which give the best model performance by training on an n dimensional grid where n is the number of parameters is not feasible. Moreover, even if you are able to do such an optimisation it is difficult to know that you are not stuck in a local optimum.

Some of the ways in which we tried to deal with this issue included

- Fixing parameters in some cases for example,
    - Xavier weight initialisation
    - Making fixed architecture choices
    - Fixing two of the four learning rate parameters in `tf.train.exponential_decay`
- Incrementally increasing model complexity

Ultimately to be able to reach or human accuracy on this problem one needs more layers, better optimisation for the model parameters and more computational power.

§5.3 Improvement

The most obvious way in which we could improve our implementation is to employ better methods to find the optimal parameter choices for a given model architecture. Incrementally increasing the complexity of the model and using grids to find the best parameter choices is very time consuming and computationally expensive because we have a lot of parameters. One possible option would be a simplex method like Nelder Mead which has a search space which evolves with the function evaluations within it. We note that there is an implementation of this available in scipy. This way we could spend less time incrementally increasing the model complexity and fitting each time and really approach this problem the way Neural Networks have been shown to work best - by choosing a much bigger architecture than needed and employing regularisation methods to prevent overfitting. Such an approach would likely be much less labour intensive than what we have done.

Another avenue of investigation would be to see if the extra information in the colour photos allows us to make more accurate predictions.

§6 References

[1] Yuval Netzer, Tao Wang, Adam Coates, Alessandro Bissacco, Bo Wu, Andrew Y. Ng
Reading Digits in Natural Images with Unsupervised Feature Learning, *NIPS Workshop on Deep Learning and Unsupervised Feature Learning 2011*

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

[3] Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, Ruslan Salakhutdinov
Dropout: A Simple Way to Prevent Neural Networks from Overfitting, Journal of Machine Learning Research 15 (2014) 1929-1958

[4] Xavier Glorot, Yoshua Bengio

[4] TensorFlow includes the option of having a step function for your learning rate, this adds two more parameters

Understanding the Difficulty of Training Deep Feedforward Neural Networks, *DIRO, Universite de Montréal, Montréal, Québec, Canada*