Capstone Project: Deep Learning

Shilpak Banerjee

September 21, 2016

1 Definition

The concept of deep learning has gained immense popularity in the recent years and is perhaps one of the major driving forces behind the whole boom in artificial intelligence. One of the greatest success of deep learning using neural networks is character recognition in unconstrained domains. I was avidly interested in this subject and hence I chose to do a project involving neural networks in the realm of digit recognition.

1.1 Project Overview

The purpose of this project is to construct a machine learning algorithm which, once trained, can recognize sequences of digits from a given image. The algorithms trains on a large collection of images of house numbers obtained from google street view images. Finally the fully trained algorithm can be used to identify sequences of digits from a random image. The algorithm involves so called *neural networks* and the code is written in Python. We use the tensorflow module to implement neural networks in our project.

1.2 Problem Statement

Sequence of digit recognition is a special kind of recognition problem. Given an image with a sequence of digits on it, we would like our model to identify all the digits on it in the correct order. This is important because if one wishes to use this model to identify digits from real-world images, like housing numbers, one needs to identify all digits correctly in the correct order to decipher the address.

In this project, we use layers of *convolutional neural networks* followed by fully connected layers applied to each digit to design our model. The model is then trained on a processed version of the *The Street View House Numbers (SVHN) Dataset* located at `http://ufldl.stanford.edu/housenumbers/` to get an optimal collections of weights. The fully trained model should be able to identify upto five digits contained in an image in the correct order.

1.3 Metrics

We will train our model to be able to identify all digits in an image in the correct order. We will train our model on the SVHN dataset which contains images of house numbers obtained from google street view images.So if the fully trained model is fed an image containing a number less than 100000 and if our model able to recognize all digits in the correct order, we call it a success. Note that even if the model identifies say 3 out of 4 digits in a 4 digit number, we count it as a failure. This way, if the model is used in real-world problems, for example, in navigation, the results will be reliable.

In our case, we evaluated the model against a testing set provided by the SVHN database. And the metric we used was the *accuracy* metric which measures the proportion of correct results (correct is as described in the previous paragraph) and expresses the result as a percentage.

2 Analysis

Now we are ready to give a high level but somewhat detail overview of the problem at our hand and also describe the general idea behind the algorithms we use. More details with detailed information regarding the specifics of the parameters used will be provided in the next section.

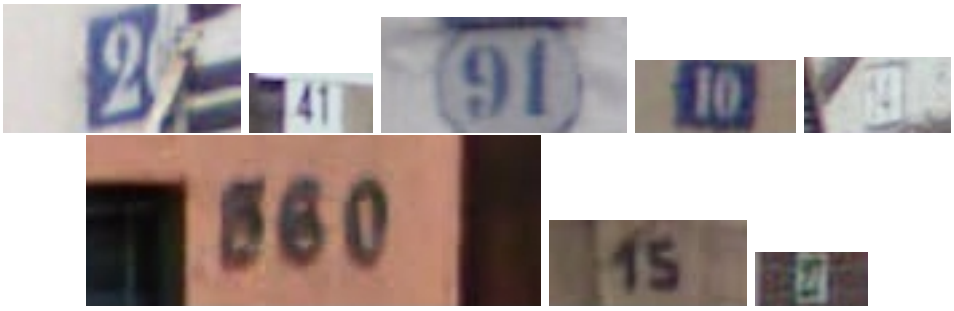

Figure 1: Sample of images from our data

We make a comment at this point of time. There can be many approaches to this problem. We adopt the one described in the 2013 article by *Goodfellow et all* (see [1]). We should note that for the sake of a better flow, in many places we refrain from citing this paper explicitly, it should be understood that our work is merely an implimentation of this paper in a somewhat restricted set up.

2.1 Data Exploration

The data we train our neural network model on was obtained from the street view house number (SVHN) dataset located at `http://ufldl.stanford.edu/housenumbers/`. This data is obtained from house numbers in google street view images. The data is provided in two formats. The first format is the one we used for training our final model. It contains variable size, variable resolution images of house numbers. The dataset also provides labeling for the images in the form of the digits pictured in correct order, number of digits and bounding boxes for the digits.

We now give a short description of the nature of this dataset. Once unzipped, this dataset has three folders: train, test and extra containing images of house numbers. These images are of variable width, height and resolution (and hence our algorithm resizes these images into a $32 \times 32 \times 1$ format before processing). Each image contains a house number made up of digits $(0, 1, \ldots, 8, 9)$. My analysis showed that the largest number of digits in a housing number pictured was six, but we restrict ourself and attempt to read only a maximum of five digits.

We should also mention that there is a second version of the data available from the same website. Here the images are more processed in the sense that individual digits are cut out into fixed size images. We did train an initial single digit recognizing version of our algorithm using this data set.

2.2 Exploratory Visualization

We would like to point to figure 1 to get an idea about the images we are using for our analysis. It is clear that the raw images have variable sizes and resolution. As part of our pre-processing we crop and resize the images to a uniform $(32 \times 32 \times 3)$ format. See figure 2 to get an idea about how the images appear after this processing. Also the folders train, test and extra contains the following number of images:

Train folder	33401
Test folder	13067
Extra folder	202352

2.3 Algorithms and Techniques

We use the technology of convolutional neural networks for our problem. So let us give a brief summary of this algorithm.

We start we describing the so called *logistic regression*. Here we have a feature vector, usually an $n \times d$ matrix X, where n is the number of samples and m is the number of features. Our job is to classify the data into d categories. So we choose an $m \times d$ matrix W, which we refer to as the *weight* matrix and a $d \times 1$ column vector b which we call the *bias* vector. We put

$$Y = WX + b$$

and then the output vector Y is fed into the *softmax* function

$$S\big((y_1,\ldots,y_d)\big) = \Big(\frac{e^{y_1}}{\sum_{i=1}^{d} e^{y_i}},\ldots,\frac{e^{y_d}}{\sum_{i=1}^{d} e^{y_i}}\Big)$$

which converts it into a sequence of probabilities. We minimize this final output with the one hot encoding of the output thereby obtaining an optimal set of weights.

This method of logistic regression works fairly well, but when applied directly without any modification, it is able to capture only a 'linear' approximation of the information contained in the training data. One modification would be to use two sets of weight matrices W_1 ($m \times h$ matrix) and W_2 ($h \times d$ matrix); two set of bias vectors b_1 ($h \times 1$ matrix) and b_2 ($d \times 1$ matrix) and do the following computation

$$Y = S(W_2(N(W_1X + b_1)) + b_2)$$

Where S is the previously described softmax function and N is some nonlinear function. Usually the *relu* function is often chosen as the non-linear function. Thus the resulting model is non-linear and is able to capture some nonlinear information from the data. The model we obtained here is often referred to as an *artificial neural network* model with one hidden layer with h hidden units. Of course this can be generalized into a model involving many hidden layers and what we get is a model for the so called deep learning architecture.

The deep learning architecture described above turns out to be very good when dealing with the problem of image recognition. There is however one method that can make this method even better. This is the concept of *convolutional neural networks*. The key idea at play here is the fact that often the same object can appear at different positions of the image and often multiple times and we would want to share this information. In other words we would like to share the weights and not let our neural net to treat the same image at different places as different objects. This is more efficient and saves a lot of computation. So we use a small window (called *filter*) and scan over the entire image. Say. We have an $m \times m$ image and we have a $k \times k$ filter, then we use a weight vector W (a $k \times h$ matrix) and a bias vector b (a $h \times 1$ matrix) to compute the vectors

$$Y_{i,j} = WX_{i,j} + b$$

where $W_{i,j}$ is a $k \times k$ submatrix at the (i, j) position. This allows us to capture the information encoded in a picture better. Of course our purpose here is to give a very loose description of the process and hence we avoid discussing what happens at the edges (i.e. SAME vs VALID padding).

This concept of convolutional neural networks when combined with operations line *pooling* and *dropouts* has turned out to be extremely robust in the realm of image recognition and hence we have decided to use the same for our digit recognition.

2.4 Benchmark

Here we describe a little about what is state of the art in recognizing housing numbers using from the housing number dataset. The best result is obtained from the work in [1] where an accuracy of 96.03% is obtained in sequence recognition. When restricted to just three convolution layers, it is stated in hte same paper that an accuracy of about 93% can be obtained.

3 Methodology

Now we are ready to give a more detailed description of the particular implementation we did of the algorithm described above.

3.1 Data Preprocessing

We did some preprocessing with the original dataset. As indicated earlier, the data consisted of images containing multi-digit housing numbers has variable size and resolution. For each image we do have access to some informations like the digits pictured along with bounding boxes containing individual numbers. We used these information to pre-process our data.

Figure 2: Sample of processed images from our data

First we use the information about the bounding boxes to crop portion of the image containing the house number along with some additional portion of the image (the boundary expanded by approximately 30% from the best fit).

Then we resized the image into a $32 \times 32 \times 3$ image and renormalized the data to have zero mean. See figure 2 for an idea about the resulting image. We have used the same images pictured earlier in figure 1 in order to give the reader a clear idea regarding the result of such preprocessing.

There was a final preprocessing done which just converted the 3 channel RGB image to a gray scale image before being stored into a *.pickle* file.

We should note that there were three files labeled train, test and extra available at the website. The train file contained 33401 images, the test file contained 13067 images and the extra fle contained 202352 images. We combined the 'train' and 'extra' images and then extracted a validation set out of it. So the battle ready data sets had the following sizes:

Training dataset size	230752
Validation dataset size	5000
Test dataset size	13067

Now we describe how we preprocessed our labels. Each image contains a house number of variable width. So, first we labeled every image with a list representing the digits pictured in ordered from the left and using the number 10 to indicate a blank spot. So for example we have the following label conversions:

$$\begin{aligned} \text{Original house number} &\longrightarrow \text{New label with a list} \\ 12067 &\longrightarrow [1, 2, 0, 6, 7] \\ 245 &\longrightarrow [2, 4, 5, 10, 10] \\ 40 &\longrightarrow [4, 0, 10, 10, 10] \end{aligned}$$

3.2 Implementation

Now we discuss how our particular example of convolutional neural networks was implemented in our code.

Before we go into the description of the neural networks, we mention a few on the fly data processing that we did. First we converted our image array into a one channel (grayscale) array. This is because we are not interested in the color of the image and also it helps a lot with easing memory overload problems. Then we converted our list labeling to a one hot labeling. So each image is labeled by a 5×11 matrix.

The code essentially has two parts. The first part has the convolution and pooling layers. These layers are common for an image.

1. First the data (a $batch\_size \times 32 \times 32 \times 1$ matrix) is fed into a convolution layer with filter $5 \times 5 \times 64$ and SAME padding.
2. Then we do a max pooling with strides $[1, 2, 2, 1]$ and SAME padding.
3. Next is a convolution layer with filter $5 \times 5 \times 64$ and SAME padding.
4. Then we do a max pooling with strides $[1, 2, 2, 1]$ and SAME padding.
5. Next is a convolution layer with filter $5 \times 5 \times 64$ and SAME padding.
6. We perform a dropout operation for the output data. This dropout is very aggressively done (60% kept) to avoid overfitting.

Our hope after performing these convolution operations is that all the digits from the image has been 'separated' and two fully connected neural network layers, one for each of these five digits will be able to correctly identify all the digits pictured.

1. We feed the output from the convolution layers into a fully connected layer with 64 hidden units.
2. We perform a dropout operation of the output data. This dropout is very mild (95% kept).
3. Finally we feed the output from the previous layer to the final fully connected layer consisting of 11 output classes.

Now we talk a little about the loss function. The for each image, our output ($logits$) is a 5×11 matrix, where each row is our model's guess for the correct digit. We isolate each row and compute its distance (*cross entropy*) with the correct labeling (tf_train_labels) for that image. We sum these individual distances up to construct our *loss* function. More precisely,

$$\begin{aligned} loss = & tf.reduce\_mean(tf.nn.softmax\_cross\_entropy\_with\_logits(logits[0], tf\_train\_labels[:, 0, :])) + \\ & tf.reduce\_mean(tf.nn.softmax\_cross\_entropy\_with\_logits(logits[1], tf\_train\_labels[:, 1, :])) + \\ & tf.reduce\_mean(tf.nn.softmax\_cross\_entropy\_with\_logits(logits[2], tf\_train\_labels[:, 2, :])) + \\ & tf.reduce\_mean(tf.nn.softmax\_cross\_entropy\_with\_logits(logits[3], tf\_train\_labels[:, 3, :])) + \\ & tf.reduce\_mean(tf.nn.softmax\_cross\_entropy\_with\_logits(logits[4], tf\_train_labels[:, 4, :])) \end{aligned}$$

The loss function is then minimized using the *stochastic gradient descent* method. Each mini batch used in consisted of 64 images and their respective labels.

3.3 Refinement

Let us briefly recall the natural evolution of this project. I took the Deep learning course from Udacity and as a part of the course developed a single digit recognizer that trains on the *NOT MNIST* data set. As a practice, I also wrote a single digit recognizer for the *MNIST* dataset and the processed *SVHN* dataset (where cropped digits were stored in *.mat* files.

The accuracy estimated on the respective testing sets for the above projects were very high and in almost all cases I did not use convolutional networks, rather used normal neural networks with many hidden layers.

In the multi-digit recognization problem, just doing many hidden layers did not produce a good result. Using convolutional neural nets improved the training process a lot. However I realized these are very taxing on the processor and the memory of my computer. So my initial attempts to go many layers deep were not successful. In fact computing validation without going through a batch feed and without crashing the computer turned out to be very challenging. I had to break down the computation into 10 parts so that it would fit in the memory of by computer.

Another aspect which stumped me initially was that I refused to convert my images into gray scale believing that convolutional networks will automatically take care of it. Perhaps this assumption is true, but, converting images to gray scale freed up more memory and I could work with more data which translated into better training.

Also I incorporated an automatically decaying learning rate in my model and it did improve performance. I also added a couple of dropout layers to eliminate overfitting to a certain extent.

4 Results

Now we are ready to describe the outcome of all experiments performed.

4.1 Model Evaluation and Validation

The final parameters for the final model was chosen after playing with several parameters. First I made a decision to not go beyond the 3 convolution layers because of computational restrictions of my computer (everyday laptop with No GPU). I used three convolution layer, each with depth 64 and a two max pooling layer with strides $[1, 2, 2, 1]$ following the first and the second convolution layers. Then I did a 40% dropout followed by two fully connected layers with 6 hidden units separated by a 10% dropout layer.

I optimized using stochastic gradient descent, with mini batch sizes of 64. I ran 150001 steps (30 hours or so in my computer) to obtain the following result:

	TYPE OF DATA	ACCURACY
1.	Last mini batch accuracy (with dropout)	81.2%
2.	Training accuracy (without dropout)	87.0%
3.	Validation accuracy (with dropout)	80.3%
4.	Validation accuracy (without dropout)	76.2%
5.	Test accuracy with dropout	60.5%
6.	Test accuracy without dropout	66.5%

Item 1. is accuracy calculated on a random selection of training data of size 64. We kept repeating this computation ever 1000 steps and comparing with validation error to get an idea of overfitting or underfitting happening on the fly. Item 2. is accuracy calculated on a random selection of 13067 training data without the dropouts. We did not use the full training data set because of memory limitations. This should give an estimate about the accuracy on the training set.

Item 3. is accuracy calculated on a validation set with the dropouts. We recall that the validation set is a set of size 5000 carved from the training set and kept aside. Our training algorithm does not directly see this data. Item 4. is accuracy calculated on the same validation set without the dropouts. We do this to get a better idea about the performance of our algorithm on an unseen data. Dropouts are good for avoiding overfitting during training but when one uses the trained model, dropouts should be removed for better performance.

Item 5. and 6. are accuracy calculation done on the test set of size 13067 with and without the dropouts. These test set was kept hidden from the training algorithm and I did not use it to tune any parameter and hence item 6. (test accuracy without dropouts) should be considered to be the best estimate about the performance our model in real world data.

4.2 Justification

The results I was able to obtain was far weaker than the benchmark of about 92% - 93% indicated by the works of Goodfellow et al. The model works fine with housing numbers with less digits but is very unreliable when it comes to four or five digit numbers.

5 Conclusion

We finish this project write up with a brief recap of how the works was done and some thoughts on how the project could be improved.

5.1 Summary

We provide a flow chart describing how our learning algorithm works. We begin by recalling the operations done in 0-*image-processor.ipynb*

Image (arbitrary size $\times$ arbitrary size $\times$ 3)

$\downarrow$

Resized Image ($32 \times 32 \times 3$)

$\downarrow$

Grayscale Image ($32 \times 32 \times 1$)

$\downarrow$

image_folder.pickle

Now we briefly describe how the model in 0-*SVHN-multi-digit-train* is organized. We use the following scheme to train our model. However when using the model to do actual predictions or computing training accuracy, we skip the two dropout steps.

Image $(32 \times 32 \times 1)$

↓

First convolution layer: $(5 \times 5 \times 64)$ filter with SAME padding

↓

First Max Pooling layer: $([1, 2, 2, 1])$ strides with SAME padding

↓

Second convolution layer: $(5 \times 5 \times 64)$ filter with SAME padding

↓

Second Max Pooling layer: $([1, 2, 2, 1])$ strides with SAME padding

↓

Third convolution layer: $(5 \times 5 \times 64)$ filter with SAME padding

↓

Dropout Layer

↓	↓	↓	↓	↓
Fully conn	**Fully conn**	**Fully conn**	**Fully conn**	**Fully conn**
↓	↓	↓	↓	↓
Dropout	**Dropout**	**Dropout**	**Dropout**	**Dropout**
↓	↓	↓	↓	↓
Fully conn	**Fully conn**	**Fully conn**	**Fully conn**	**Fully conn**
↓	↓	↓	↓	↓
Digit 0	**Digit** 1	**Digit** 2	**Digit** 3	**Digit** 4

We finally provide a list of sample prediction, both correct and incorrect predictions made by our model. We refer to figure 3 to get an idea about the predictions made by our model. Also the code $0 - random - test.ipynb$ can be used to produce random predictions on the test set and visualize the results.

5.2 Improvement

To be completely honest, I am far from being satisfied with my results. My accuracy is far below the maximum possible as obtained in Goodfellow et all. I have decided to submit the project at this time because of time and resources. It took me a long time to run the project on a dual core computer with no GPU and each parameter tuning required a lot of time. There were many experiments that I still want to to do and I believe they may be able to improve the model. I list them below.

1. I did not play around much with the training set. I would like to see if increasing the number of images in the training set using multiple copies of the same image, each with some distortions and different centering would produce a better result or not. I avoided this because increasing the training data seems to slow down my computer al lot and cause it to crash. May be partially loading only a portion of the training set, training and saving the model, deleting the training set and loading the next chunk and improving the trained model would be a solution for a resource crunched computer. Did not have time to do it.

2. I would like to explore more about the data. And introduce 'fake' distorted copies of images with less represented number of digits.

3. I am extremely unhappy with the validation set size of 5000 that I used. In my opinion and also from the Udacity deeplearning class I took, it seems a bigger ($\sim$ 30000) validation set is more appropriate. Memory problems while calculating validation accuracy motivated me to use a smaller set.

4. There is overfittingand I would like to test more aggressive ($\sim$ 50%) dropouts.

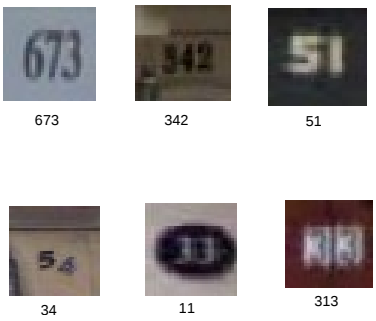

Figure 3: The first row depicts images which were correctly predicted by our model. The second row depicts images incorrectly predicted by our model. We include the predictions our model made.

5. I would like to see if one can play around with the way variables are initialized and see if it makes a difference.
6. Building a live camera app as suggested in the Udacity project description seems to be interesting. Since this would involve making minor modification to an existing code, I decided to skip it and focus more on the deep learning aspect of the project.

6 Acknowledgement

I used have gone through the Udacity forums many time and have used many ideas from the discussions that happened there. I thank user Hang_Yao in particular whose comments were particularly insightful.

References

[1] Ian J. Goodfellow, Yaroslav Bulatov, Julian Ibarz, Sacha Arnoud, Vinay Shet *Multi-digit Number Recognition from Street View Imagery using Deep Convolutional Neural Networks* arXiv: `http://arxiv.org/abs/1312.6082`

Appendix: How to use the code

There are three python codes. We explain how to use these.

1. First one needs to download the three files *train.tar.gz*, *test.tar.gz* and *extra.tar.gz* from the SVHN dataset.
2. Then one runs 0-*image-processor.pynb* which creates the raw image folders $train, test$ and $extra$. Next it creates the processed image folders $train\_processed, test\_processed$ and $extra\_processed$. Next it pickles the info into six files: *train_processed.pickle*, *train_processed_labels.pickle*, *test_processed.pickle*, *test_processed_labels.pickle*, *extra_processed.pickle* and *extra_processed_labels.pickle*. This files are provided in the git repository.
3. Then one runs 0-*SVHN-multi-digit-train* which uses the six *.pickle* files listed above. The output is the model *model*1.*ckpt*. This is provided in the git repository.
4. One can play with image prediction using *model*1.*ckpt* using the code 0-*random-test.ipynb*