Capstone Project

Ilya Raykin

Machine Learning Engineer Nanodegree

December 23th 2016

Project Overview

Recent years have brought many advancements in deep learning, particularly in the field of image recognition. And as the amount of image data available about our world grows, as does the opportunity for using machine learning algorithms to extract useful information from this data. Applications include licence plate recognition for better traffic control[1], real-time object recognition for self-driving cars[2], and house number recognition for accurate mapping systems[3]. In this project, I explored this last example - recognizing house numbers in images.

The goal of this project was to use Convolutional Neural Networks to build a system that can recognize and read a number that is located somewhere within an input image. Since the vast majority of house numbers contain less than six digits, the system is designed to only deal with sequences of five digits or less.

Two datasets are used in this project. The primary data the classifiers are trained on is the SVHN (Street View House Number) dataset[4]. Additionally, the MNIST handwritten digit dataset[5] was used to generate synthetic data for the early stages of development.

Problem Statement

This problem can be broken down into the following high-level tasks:

1. Develop a system for pre-processing images into a form suitable for ingestion by a machine learning algorithm
2. Train a classifier that can output the digits of a number in an image, provided that the image is centered and cropped around the number
3. Develop a system to isolate the location of a number in a given image
4. Develop an algorithm that uses the classifiers from steps 1 and 2 to locate and read a number located anywhere in an arbitrary image
5. Write a web application that lets a user upload an image, and returns the number shown in the image along with the corresponding bounding box

While many classifiers exist that could conceivably be used in steps 2 and 3, Convolutional Neural Networks are currently among the best options for image classification problems[6]. In particular, they are an excellent choice for tasks that have a large corpus of labeled data available, as is the case here.

[1]H.Erdinc Kocera, K.Kursat Cevikb, Artificial neural networks based vehicle license plate recognition
[2]Alex Teichman and Sebastian Thrun, Practical object recognition in autonomous driving and beyond
[3]Ian J. Goodfellow et al. Multi-digit Number Recognition from Street View Imagery using Deep Convolutional Neural Networks
[4]http://ufldl.stanford.edu/housenumbers
[5]http://yann.lecun.com/exdb/mnist/
[6]Jiuxiang Gu et al. Recent Advances in Convolutional Neural Networks

The component in step 2 will be referred to in this paper as the *digit recognizer*. This is a neural network that, given an image of a number cropped to a reasonable bounding box and scaled to expected dimensions, will attempt to output the numbers represented in the image.

The component in step 3 will be referred to as the *number locator*. The number locator has the goal of locating a number in a provided image, without actually reading it. This is necessary for application to be able to work on images where the placement of the number is arbitrary. This has two sub-components:

1. A neural network that acts as a binary classifier to determine whether an input image contains a sequence of digits
2. An algorithm that uses this neural network to traverse the input image and identify potential number locations

Metrics

The metric I used throughout this project is *accuracy*, defined as the number of correct predictions divided by the total number of data points.

$$accuracy = correct / total$$

Accuracy is a commonly-used metric for machine learning systems because it is simple to calculate and intuitive to understand. There are three measures of accuracy that can be calculated in this system, each of which requires its own definition of correctness:

1. Accuracy of the number locator classifier
2. Accuracy of the digit recognizer classifier
3. Overall system accuracy

Correctness is easy define and calculate for the number locator: it is the proportion of the test images that are classified correctly into whether or not they contain a number. There are only two labels possible here, “yes” or “no”. Consequently, the the number of correct predictions is simply the sum of the of the *true positives* and the *true negatives*.

Calculating accuracy for the digit recognizer is slightly more involved. Given an input, the output is considered *correct* if and only if each digit in the sequence was read correctly. However, the classifier produces an output for each of the five digit positions even if there are less than five digits in the number, since “no digit” is one of the labels. That said, it should not matter if the fourth digit prediction for a three-digit number is wrong. To this end, a classification is considered *correct* if the first *n* digit predictions are correctly identified, where *n* is the sequence length prediction.

It should be mentioned here that accuracy is a poor metric to use when the underlying data has large class imbalances. In fact, as will become clear in the next section, the distribution of the SVHN data with respect to number length is heavily skewed. However, I did not deem this to be a critical issue because the digit classifier performance is more dependent on the digit-level classifiers, which are trained on well-balanced data. In other words, since length prediction is only one piece of the puzzle, I deemed accuracy to be a sufficient metric.

Lastly, we can calculate accuracy of the system as a whole. At this scale, we aim to determine how well the system can locate *and* accurately read a number in an image. This is the end goal of this project, and involves both components mentioned above. A *correct* result is one where the system can accurately determine the number located anywhere in the image. This is distinct from digit recognizer task accuracy, which benefits from known number bounding boxes.

Data Exploration

The dataset used to train the neural networks in both the number locator and digit recognizer is the SVHN dataset, released by Google[1]. The dataset is freely available at *http://ufldl.stanford.edu/housenumbers*. SVHN is a publicly available dataset of images of house numbers, extracted from Google Street View, roughly cropped around the numbers. The images vary widely in size, resolution, and quality.

The dataset is partitioned into three sections: *train, test, and extra*. The sections have 33,402, 13,068, and 202,353 images in them, respectively. Since Convolutional Neural Networks benefit from being trained on larger datasets, I used both the *train* and *extra* datasets for training the network, resulting in a training set size of 235,755 images.

In addition to the images themselves, the SVHN dataset provides metadata for each of the three datasets. For each image, the metadata contains the numerical value of each digit in the number, in order, as well as coordinates each individual digit's bounding box.

Exploratory Visualization

SVHN Data

In the following analyses, the "train" and "test" datasets have been combined together, since they are both used for training the classifier.

The SVHN dataset varies widely in resolution. The *Table 1* below summarizes the dimensions of the training and test data, while *Figure 1* and *2* visualize the associated distributions.

Table 1

Dimension	Minimum	Maximum	Mean	Std. Deviation
Training width	22	876	104	56
Training height	12	501	60	31
Test width	31	1,083	173	123
Test height	13	516	72	53

Figure 1

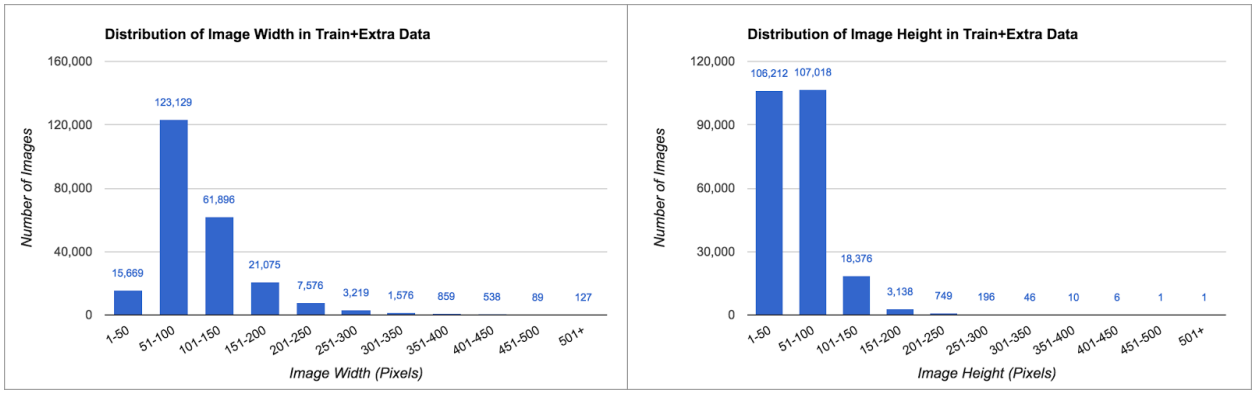
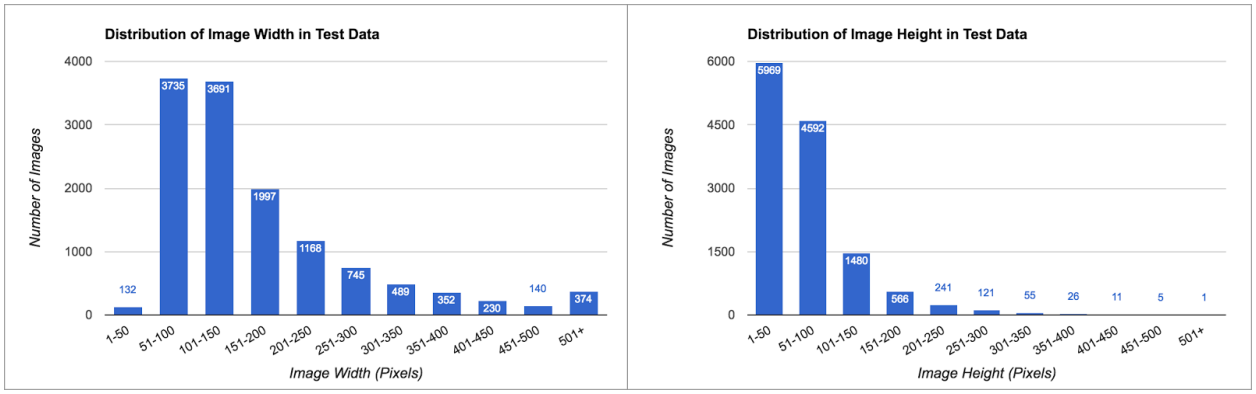


Figure 2



The presence of low-resolution images means that some of the numbers are difficult to read even for humans. For example, below is the smallest image from the training data, as measured by the number of pixels:

This image is labeled as the number “6”, despite being blurry and somewhat ambiguous. While this is an outlier, it tells us that there exist some images in the training set that are far less clear than others.
All SVHN images are in the RGB color space. Four images have an alpha channel in addition to the three RGB channels. This channel has all values set to 255, indicating 100% opacity. Figures 3 and 4 below show distributions of the three color channels across the train and test datasets. Figure 5 shows the average pixel intensity for both datasets, as a percentage of each dataset.

Figure 3

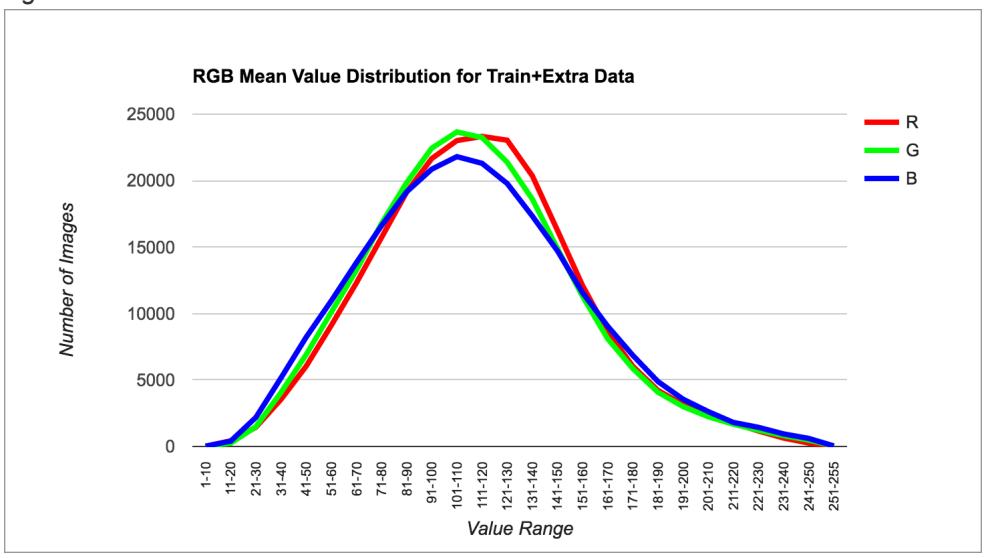


Figure 4

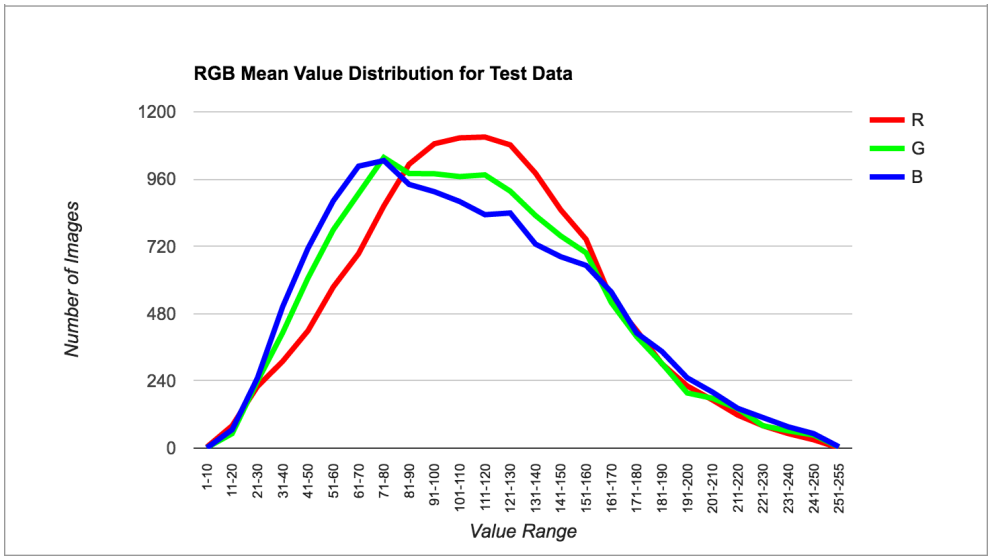
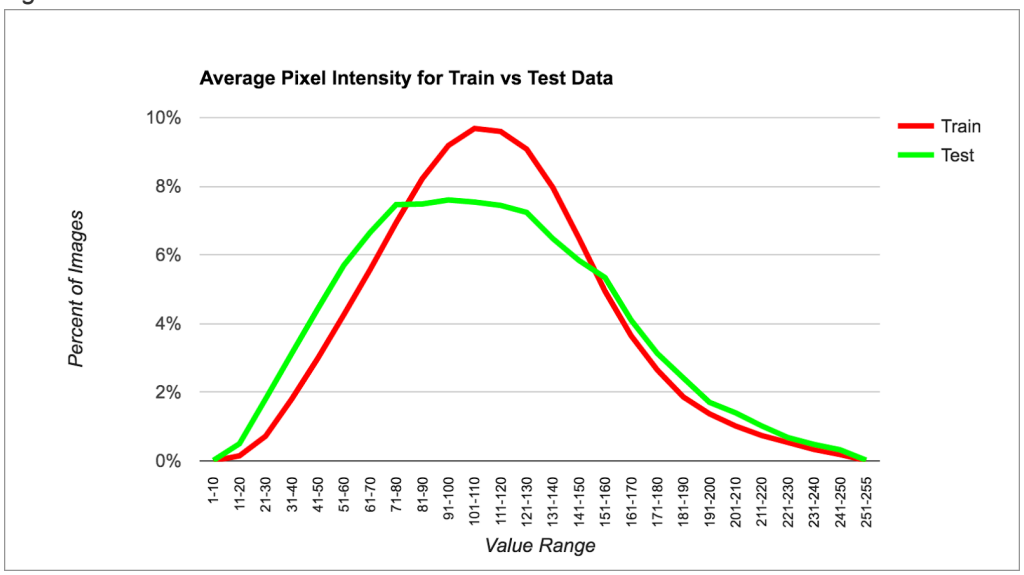


Figure 5

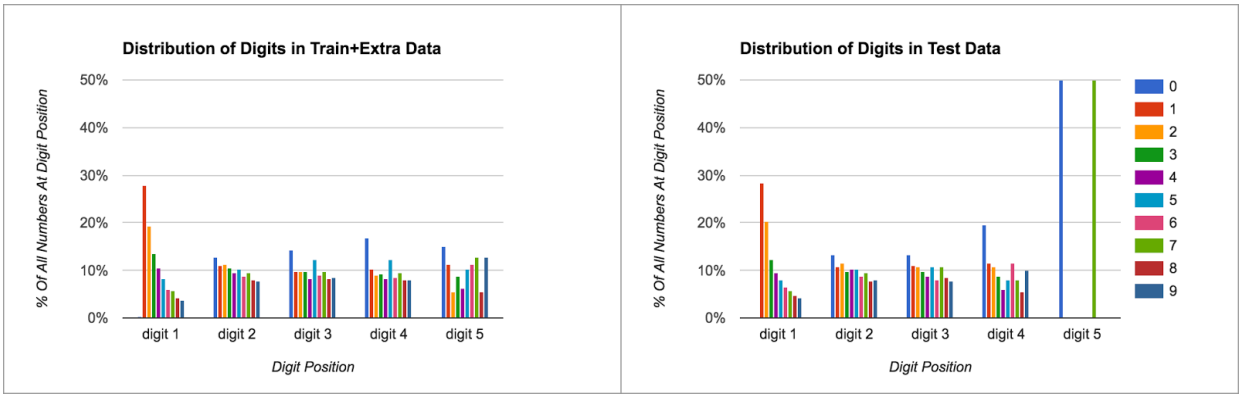


Since the images are converted to grayscale for training, Figure 5 is the most informative - it shows that the pixel intensities in the test dataset have a slightly wider distribution.

It is also important to understand the distributions of the numbers in the dataset, with respect to both the number length and individual digits. This is relevant because the digit recognizer neural network is trained on these labels, and can therefore be affected by the balance of labels.

Figure 6 shows that the digit values 0-9 at each digit position look to be fairly well-distributed. The distributions aren't exactly uniform - the all seem to be slightly weighted towards lower numbers - but in general the data is well-behaved. The only outlier seems to be the 1st digit position, which has very few examples of *0* and an overabundance of *1* and *2.* Overall, this should be sufficient for training the digit recognizers to read each digit position without much bias.

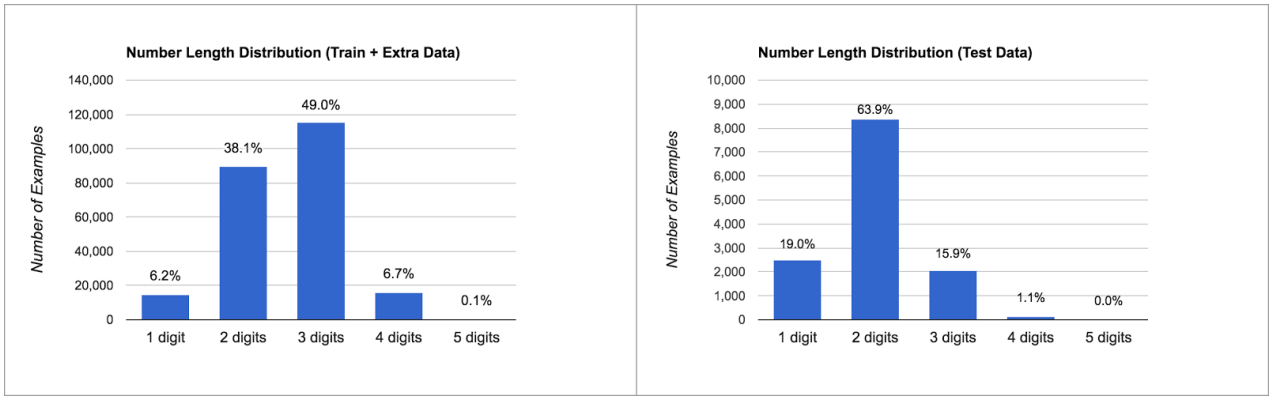
Figure 6



The distribution of number lengths, however, is less favorable. Looking at the charts below, it is apparent that the numbers are highly imbalanced with respect to the number of digits. The combined train+extra dataset is comprised of almost 50% 3-digit numbers, followed by 38% 2-digit numbers, with numbers of

lengths 1, 4, and 5 severely underrepresented. In particular, the training dataset has only 124 samples of five-digit numbers!
The distribution in the test dataset is somewhat different, but still very skewed. 3-digit numbers represent only 16% of the test data, with 2-digit numbers accounting for 64%. Only 2 samples in the test data have five digits.
Overall, this is somewhat problematic. If the training data has only 124 samples of numbers with five digits, the neural network may not have enough examples to learn what “five digits” looks like. Likewise, a mere two data points is meaningless for testing the accuracy of the length classifier when it comes to five-digit numbers. Unfortunately, we lack the data to properly train the classifier on long numbers.

Figure 7

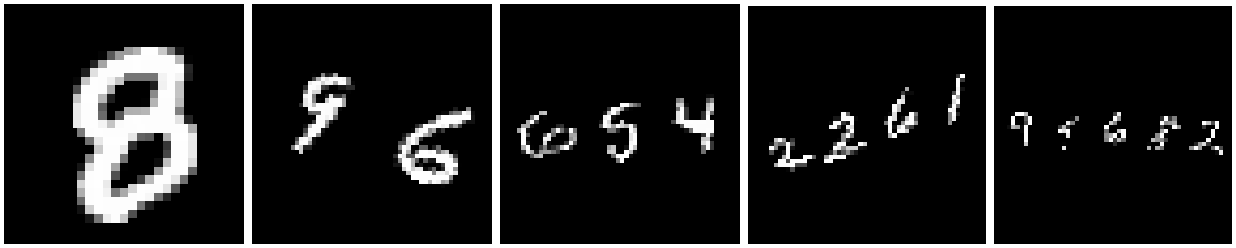


Lastly, one of the images in the training dataset had six digits. This data point is skipped during training.

Synthetic Data

In addition the SVHN dataset, I used the MNIST handwritten digit dataset to generate synthetic images of digit sequences that are simpler than those of the real-world data. The data is generated by stitching together, and then rotating, separate images from the MNIST dataset. Figure 8 below shows some examples.

Figure 8



The original MNIST images all have dimensions of 28x28 pixels. The generated images are four times larger, with dimensions of 56x56. All images have a single channel with values between 0 and 255.
The distribution of numbers and sequence lengths is uniform by design. The synthetic data has no set size, as I experimented with generating varying numbers of images in the early stages of the project.

Algorithms and Techniques

Digit Recognizer Design

A key question that arose early on was how to design a digit recognizer to read multiple digits in a single image. Training a network to output the entire number, up to five digits long, is clearly infeasible - this approach would require distinguishing between 99,999 possible output classes, and would therefore require the training data to sufficiently represent each possible digit combination. Not only is this inefficient, it is also infeasible due to data constraints.

A better approach is to read the number one digit position at a time, independent of the surrounding digits, and to also be able to tell how many digits the number contains. The digit recognizer would then only have to learn to output one of 11 classes (0 through 9, plus “no digit”) for each of the five potential digit positions, and one of five classes for the number length. These outputs would then be combined into the final answer - specifically, the final number would be the concatenation of the first *n* digit outputs, where *n* is the estimated length of the sequence.

There are two ways to approach building this kind of system. The first is to train a separate neural network for each of the tasks described above, resulting in six independent networks: one for the length, and one for each of the five digit positions. A supervisory process would then combine the outputs of these networks into the final output. A second approach is to combine these classifiers into a single neural network that produces six simultaneous outputs via six output layers.

The tradeoffs of these two methods have to do with training time and weight sharing. The single-network approach vastly reduces the time requirements to train the digit recognizer, which, given the hardware limitations, is a great benefit. The potential downside is that the sharing of the weights learned by the network may affect the accuracy of the effectiveness of the individual outputs. Conversely, training six individual networks for each output is much more time consuming, leading to slower network tuning, but may improve per-digit accuracy.

The codebase was developed with both of these options in mind; however, early on in experimentation I decided to use to the single-network approach. My initial testing showed that the joint network did not exhibit lower accuracy relative the individual per-digit networks, and the 5x reduction in training time due to the need to only train once per model made iterative development much more feasible.

Number Locator Design

The goal of the number locator is to find potential image patches in the input image that are likely to contain the number at a scale that resembles the data the digit recognizer was trained on. The primary component of the number locator is a binary classifier that returns “true” if the image contains a number and “false” otherwise, as well as a value representing the confidence of this result.

Neural Network Choice

While there are several types of neural networks architectures available that could theoretically be used for this problem to varying degrees of effect, I chose to use a Convolutional Neural Network architecture. CNNs are a class of neural networks that happen to be well-suited for image recognition tasks. To understand why they are so effective, it’s useful to first understand why a simpler architecture - the Multi-Layer Perceptron (MLP) - would be less appropriate.

In an MLP, each neuron in a given layer is connected to each neuron in the next layer, creating what is referred to as a “fully-connected” architecture. When trained, the MLP uses backpropagation to learn the weights for each connection between each neuron in adjacent layers. In the input layer, each neuron

corresponds to a pixel in the provided image. The upshot is that the network attempts to learn how relationships between *individual pixels* in the input image affects the final outcome. This is problematic when we consider how such a network would treat the following two images:

Although the two images are of fundamentally the same thing, simply moving the object in the image completely changes the underlying pixel values that are critical to recognizing it. In other words, fully-connected architectures fail when there is a lot of variance in positioning and scale.

A Convolutional Neural Network leverages the concept of *image kernels* often used in image processing. An image kernel, also referred to as a convolutional filter, is a matrix of values that is “slid” across an image; at each position, the kernel values are multiplied with the underlying pixel values and then added together to create a single output value. The process of applying this operation across the entire image is called a convolution.

A single layer of a CNN can contain any number of such filters, with subsequent layers generally having more filter depths. A process called *pooling* can then be applied to reduce the layer dimensions; in short, pooling breaks up the layer into a grid of cells (often 2x2 pixels) and for each one outputs a single value based on some criterion; this has the effect of reducing the layer’s dimensions and selecting the most relevant features. Different pooling functions exist, but *max pooling* is one of the most common approaches - a 2x2 max pooling layer will select the maximum of the 4 pixel values at each 2x2 patch in the underlying layer, halving its height and width.

Combining increasingly deeper convolution layers with pooling layers has the effect of reducing the image dimensions while increasing the depth of the information encoded in each layer. Essentially, such a network learns to distill *spatial* features into *conceptual* features that are independent of their placement in the image.

The benefit of this network architecture is that each layer learns progressively more abstract patterns in the data. To provide an oversimplified example, the digit recognizer may learn edge detection filters in the first layer, simple shape detectors in the second, and more complex shapes in the third.

The final convolution layer is fed into a full-connected set of layers that learn the relationship between the final convolutional layer and the output classes. In the above example, it may learn to distinguish the digits 0-9 based on the shapes identified in the third convolutional layer.

By contrast, an MLP would need to look at individual pixels for these relationships!

Other Network Features

There are other techniques that can be added to the CNN architecture to improve their performance. All of these are meant to reduce overfitting, one of the primary causes of poor accuracy, but some also have secondary benefits such as improved training time.

Dropout

Neural networks of all kinds, including convolutional networks, often fall victim to overfitting, wherein the learned weights fail to generalize to new data. Dropout is an approach that randomly drops neuron activation with the specified probability. The result is that during the training step, the network has to learn multiple internal representations of the data, making it better at generalizing to new inputs. It is an effective technique, and is used extensively in the digit classifier and number locator CNNs in this project.

1x1 convolutions

A relatively recent development in the field of CNNs is that of 1x1 convolutions. As its name implies, a 1x1 convolution is simply a convolutional layer with height and width of 1. While this may seem strange, it happens to be an effective technique for dimensionality reduction, which reduces overfitting and improves training time.

L2 normalization

L2 normalization is yet another way to reduce overfitting in a network. It works by introducing a parameter into the cost function that increases with the network complexity.

Benchmark

Goodfellow et al. at Google achieved 96.03% accuracy using their model, with 97.84% digit-level accuracy. Clearly, I do not aim to reach Google's accuracy metrics. Instead, I decided on a reasonable goal for the system by looking at its constituent parts.

Digit Recognizer Benchmark

To come up with a benchmark estimate for the digit recognizer, I broke it down into its components. This neural network has six output layers: one for the length, and one for each digit position. A reasonable goal is to aim for 90% accuracy for each of these outputs. Knowing the distribution number lengths in the test dataset, ignoring 5-digit numbers due to their scarcity, we can calculate the expected accuracy of the digit recognizer:

$$0.9 \times [\, 0.19 \times 0.9 + 0.64 * 0.9^2 + 0.16 \times 0.9^3 \times 0.01 * 0.9^4 \,] = \mathbf{73\%}$$

Number Locator Benchmark

I was unable to find a benchmark for the binary classifier underlying the number locator, but I expected the accuracy to reach a minimum of **90%**.

Algorithm Benchmark

Determining a benchmark for end-to-end system accuracy is difficult. Given my goals of 65% test accuracy on the digit recognizer 90% accuracy on the number locator, and factoring in error introduced by the imperfect nature of locating a bounding box, I expect the system to reach **50%** accuracy on the task of locating and correctly reading SVHN house numbers. Note that this is more difficult, since at this stage we no longer rely on the bounding boxes provided by the metadata, but are instead finding our own.

Data Preprocessing

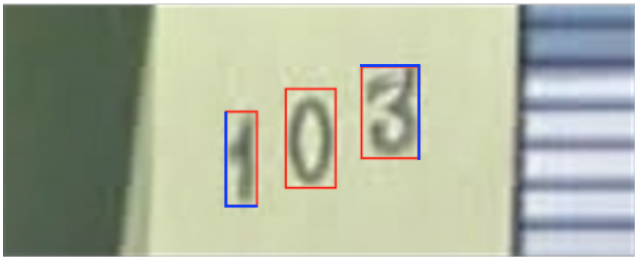
The SVHN dataset was used to train both the number locator and digit recognizer neural networks, although they required different pre-processing steps.

Metadata Preprocessing

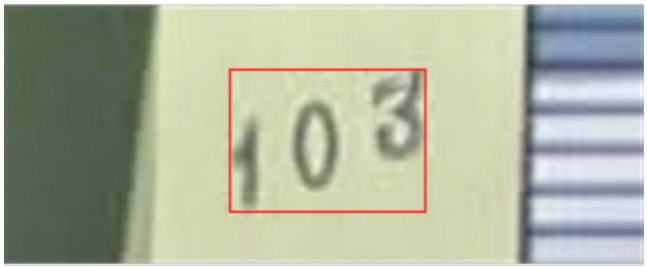
The metadata for each dataset resides in a single large file, *digitStruct.mat*, that is time consuming to open and read. Before any of the datasets could be used, the metadata had to be run through a process to transform it into a more easily-accessible format. To this end, I wrote a python script to extract this data into an easily-accessible dictionary data structure and persisted it to disk. After this one-time process was run for each of the three datasets, the metadata could be easily loaded and manipulated.

Digit Recognizer Training Data

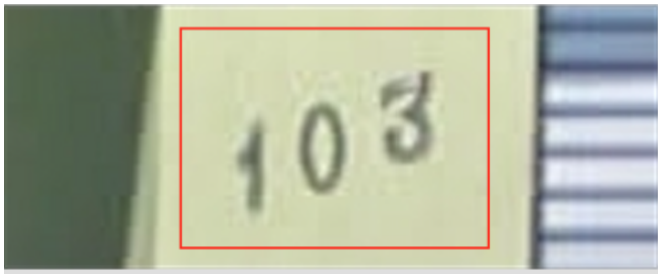
In order to use the SVHN dataset as training data for the digit recognizer, each image had to be cropped to the edges of the number contained within, in order to have more consistently positioned digits. In order to accomplish this, the per-digit bounding box data for each image had to be transformed into a bounding box for the whole number. This was done by first calculating the uppermost, lowest, leftmost, and rightmost edges across each digit bounding box:



Those extents were then used to create a bounding box for the number as a whole.



Finally, this bounding box was expanded by 30% in each direction.

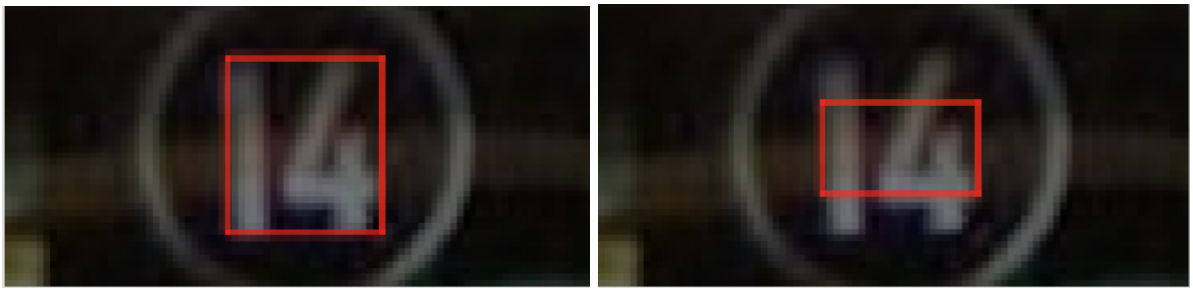


This last step was introduced in order to train the digit recognizer on images with a buffer zone between the digits and the image edges, which corresponds to the expected output of the number locator (described below). Without this buffer zone in the digit recognizer training data, the positioning of the digits in the image patches found by the number locator would differ from the digit positioning in the digit recognizer data, potentially harming performance.

Number locator training data

The positive examples for the number locator were extracted directly from the SVHN dataset using the same method described above. Generating negative examples, images without numbers, were also generated from the SVHN data, albeit with some extra processing. To generate negative examples, a window the same size as the bounding box calculated above was slid over each image in the dataset with a given horizontal and vertical step value. Whenever this window overlapped with the known bounding box, the image patch represented by the window was skipped. Otherwise, it could be assumed that the patch did not have any numbers in it, and could be used as a negative training sample. The image below shows an example of such image patches.

The 30% increase in bounding box dimensions was preemptively introduced in order to prevent the number locator from cutting off digit edges. I had a suspicion that training the number locator with bounding boxes that do not provide a buffer zone for the digits could lead to the algorithm not being able to differentiate between the following two scenarios:



In this example, the digit “1” looks similar in both bounding boxes, although the box on the left is preferable. I had hoped that training the classifier with a buffer zone around the digits would mitigate this.

Image Pre-Processing

The pre-processing procedure for images prior to inputting them into the digit recognizer network was as follows:

1. Convert the image to grayscale. This reduced the number of channels at each pixel to 1. This is done because color is largely irrelevant to reading the numbers, so we benefit from a 3x reduction in input size without loss of features.
2. Crop the image to the bounding box calculated using the steps described above. This places the digits at a consistent location and scale in each image.
3. Resize the cropped image to 64x64 pixels. Note that this may involve either increasing or decreasing the image dimensions, as well as changing the aspect ratio. For simplicity and performance, we use the *Image.NEAREST* resizing algorithm in the PIL module; while not ideal in most applications, it is sufficient for scaling the image for input into a classifier.
4. Normalize the pixel values by subtracting the mean pixel value from each pixel and dividing by 255. This constrains the data to between -1 and 1, making it more suitable for the neural network.
5. Flatten the 2-dimensional image data into a vector of 4096 values. This is necessary because the first layer of the neural network requires a single-dimensional array as input.

The pre-processing procedure for the number locator network is identical, except for the omission of step 2, as the input images are already cropped by the time they enter the pre-processing pipeline.

Implementation

The primary algorithm needed for the system to work ties together the number locator and the digit recognizer. At a high level the algorithm is as follows:

1. Extract potential number-containing image patches using the the number locator (detailed below)
2. Run each of the image patches from step 1 through the digit recognizer to extract potential numbers and corresponding confidence values
3. Drop numbers whose length is more than 1 less than the maximum number length from step 2. For example, if the longest prediction is 4 digits long, ignore all predictions of length 1 and 2.
4. From the set of numbers in step 3, return the number with the highest classification confidence.

Note that this algorithm yields more than one potential match at step 1. Although I had hoped that the number locator would result in one unambiguous answer, in reality it recognizes numbers at a variety of scales: for example, the highest confidence match may be an image patch that is “zoomed in” too far on the number, truncating several digits. To compensate for this, step 3 attempts to identify the “best” of these potential bounding boxes by the digit recognizer confidence and number length.

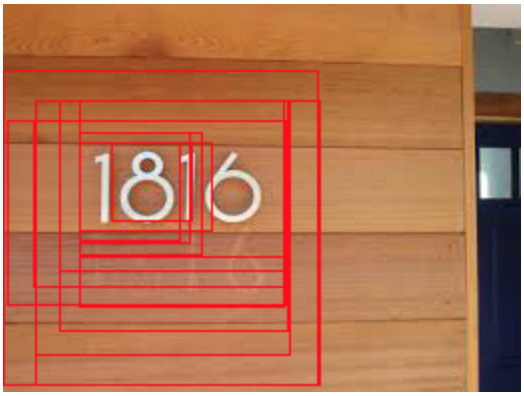
The method used by step 1 above to find candidate image patches takes a recursive approach:

1. Slide windows of various aspect ratios across the image. The windows have a height equal to the height of the input image, scaled down by a constant multiplier. The varying aspect ratios are meant to accommodate different lengths of numbers.
2. Choose the window for which the classifier returns “true” with the highest confidence, provided that this confidence passes a minimum threshold. This is the best window at this recursion level.
3. Recurse into step 1, this time using the window from step 2 as the input image. Repeat until the window height reaches a specified minimum threshold.

4. Return the best windows for each recursion level.

Below is a visualization of this process, for the following example image:

First, the recursive number locator algorithm finds candidate image patches at various zoom levels. Note that it eventually “overshoots” the number and zooms in on a single digit.



Each of these image patches, including the entire image, is run through the digit recognizer, producing a digit sequence prediction with a corresponding confidence level between 0 and 1. Table 2 below shows each image patch, its prediction by the digit recognizer, and the corresponding confidence. The highlighted row corresponds to the image patch chosen as the final prediction.

Table 2

Image Patch	Prediction	Confidence
	10	0.483

	1979	0.618
	1519	0.814
	1814	0.819
	1816	0.984
	**1816**	**0.987**
	1816	0.888
	18	0.851
	18	0.953
	18	0.907
	8	0.913

The highlighted image patch is selected as the best choice, as it has the highest classification confidence out of the 4-digit predictions. It also happens that it has the highest confidence out of all predictions, but that is not always the case; for example, the confidence for the subsequence “18” could have been the highest overall. The step to drop shorter predictions exists for just such a scenario.

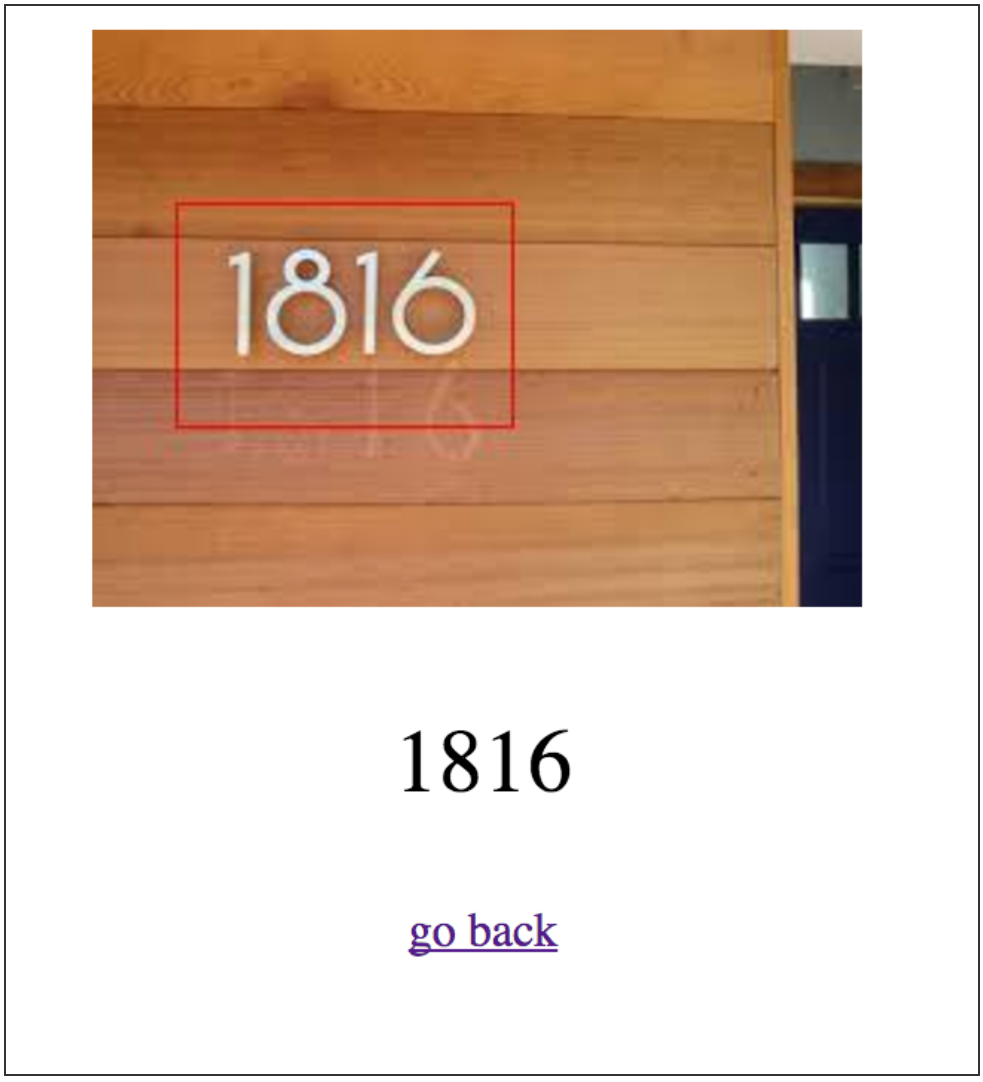
Classification Confidence

In order for the approach described above to work, the neural networks must have a way of outputting a confidence value along with their predictions. This is straightforward for the number locator: the logit values from the output layer are normalized with a softmax function, the largest resulting value is taken as the confidence for the prediction. Confidence for the digit recognizer requires an additional step: since this network has six output layers instead of one, the classification confidence is calculated as the mean of the six confidence values.

Web Application

The number recognition algorithm is exposed via a simple web interface. The first screen in the application prompts the user to upload an image containing a number:

After choosing a file and pressing “submit”, the application returns its prediction for the digit sequence highlighted, as well as a bounding box overlay of the image patch used for the prediction.



The web application can be launched via the *server.py* script included with the codebase.

Refinement

Digit Recognizer Refinement

Training a CNN on the full SVHN dataset is a time-consuming task on the hardware available for this project (2013 Macbook Pro), so the initial digit recognizer network architecture and supporting codebase was developed on a synthetic dataset much simpler than the one provided by SVHN. Working with this simpler dataset also allowed for easier debugging and quicker iteration during the early stages of development. The goal was to train a CNN that could perform reasonably well on this synthetic data before switching over to the SVHN data.
The synthetic data was generated by concatenating random digits from the MNIST dataset of handwritten digits, and encoding the data about the generated number in the filename. In order to introduce more variation, the dataset had the option of introducing a random rotation into numbers. This resulted in an images of numbers that were still diverse, yet without the complexity of the SVHN data.

The script was capable of generating images in two ways:

- By length: the script would generate *n* random images for lengths 1-5, resulting in 5*n* total images. This data was used early on in experimentation to develop a network capable of recognizing number length.
- By digit: the script would generate *n* random images with each digit 0-9 held constant at each of the 5 digit positions, resulting in 50*n* images. This was used to train digit-level classifiers, as it was guaranteed to provide an even number distribution.

The neural network architecture I began training on this synthetic data was a simple multi-layer perceptron, instead of a CNN. To further simplify the task, I opted to train it to recognize a single digit position at a time. Once the foundation was ready, I could modify it to learn all digit values, plus the length, at once.

The following neural network parameters were held constant for all runs at this stage:

- Rectified Linear Unit activation
- Trained over 25 training epochs with batch size of 250
- Weights initialized with a truncated normal distribution with a standard deviation of 0.1
- Biases initialized at 0.1
- Cost defined as the cross-entropy between the prediction and the known labels
- Weights learned with an Adam Optimizer with a learning rate of 0.001

Table 3 summarizes results from the first set of experiments.

Table 3

training size	*hidden layer sizes*	*dropout*	*accuracy*	*notes*
16000	2 x 5,000	0.5	83%	
16000	1 x 5,000	0.5	82%	
16000	1 x 10,000	0.66	86%	High dropout with wide hidden layer helps
8000	2 x 5,000	0.5	75%	More training data helps accuracy
8000	1 x 5,000	0.5	80%	Less layers may be better
8000	1 x 5,000	.25	76%	Lower dropout is worse
8000	1 x 5,000	0	70%	No dropout is even worse

The key takeaways from these runs is that dropout is a key ingredient to improving results. The best results achieved were with a high 66% dropout and a single hidden layer of 10,000 neurons.
My attempts at training an MLP to determine the length of the synthetic numbers was much more successful: any one of the networks in the above table determined the number length with *98-99%* accuracy.

The next step was to improve on these results by training a convolutional network on the synthetic data. For these runs, I increased the number of training images to 80,000. I also introduced L2 normalization to all models at this stage, with a *beta* value of 0.0005. Lastly, I updated bias initialization to 0. All convolutional layers have a 1x1 stride and use SAME padding algorithm.

Table 4 shows the two key results from those runs.

Table 4

Convolution layers	*Filter depths*	*FC layers*	*dropout*	*pooling*	*accuracy*	*notes*
10x10, 10x10, 10x10	16, 32, 64	1x1,000	0.5	max pool all layers	91%	much better than MLP
10x10, 10x10, 1x1	32, 64, 64	1x1,000	0.5	max pool all layers	95%	1x1 convolution helps a lot

My first attempt at a convolutional network yielded 91% per-digit accuracy on the synthetic dataset. Introducing a 1x1 convolution layer bumped this up to 95%. Not listed here are attempts at increasing the number and size of fully-connected layers; those runs yielded noticeably poorer accuracy partway through training and were terminated prior to completion to save time.

Next, I modified the best network above to learn all six output layers simultaneously. This required making some key changes to the codebase, since accuracy was no longer simple to calculate, as the value of the first output layer determined how many of the digits were evaluated. The batched nature of the validation process made this tricky; the solution involved aggregating intermediate data from each test batch and calculating the final test accuracy outside of the computation graph

.

Switching to a single-network architecture resulted in 90.5% overall accuracy on the synthetic dataset, which I found to be promising. At this point, I decided to move on to the SVHN data. This introduced another challenge - the code had previously been loading the data into a single large numpy array in memory and accessing batches of it for training. This approach did not scale well to the SVHN dataset, as loading all the data into memory became intractable. To get around this, I factored out data access to a *datasource* interface with multiple implementations. The synthetic data would still be accessed from memory, while SVHN data would be lazily loaded from disk as needed.

Table 5 shows the results of the last set of runs.

Table 5

Convolution layers	*Filter depths*	*FC layers*	*dropout*	*pooling*	*accuracy*	*notes*
10x10, 10x10, 1x1	32, 64, 32	1x1,000	0.66 all layers	max pool all layers	72%	2 days to train
3x3, 3x3, 1x1, 3x3, 3x3, 1x1	8, 16, 32, 32, 64, 64	1x1,000	0.66 all layers	max pool layers 3-6	71%	1 day to train

5x5, 5x5, 5x5, 1x1	16, 32, 64, 32	1x1000	0.66 all layers	max pool all layers	66%	worst model so far
5x5, 5x5, 5x5, 1x1	16, 32, 64, 32	1x1000	0.5 all layers	max pool all layers	70%	reducing dropout helps
3x3, 3x3, 1x1, 3x3, 3x3, 1x1	8, 16, 32, 32, 64, 64	1x5000	0.2 conv layers, 0.66 FC layers	max pool layers 3-6	72%	less dropout on conv layers; more FC neurons
5x5, 5x5, 5x5, 5x5, 1x1	8, 16, 32, 64, 32	2x1000	0.5 FC layers	max pool 2-5	75%	Best model achieved

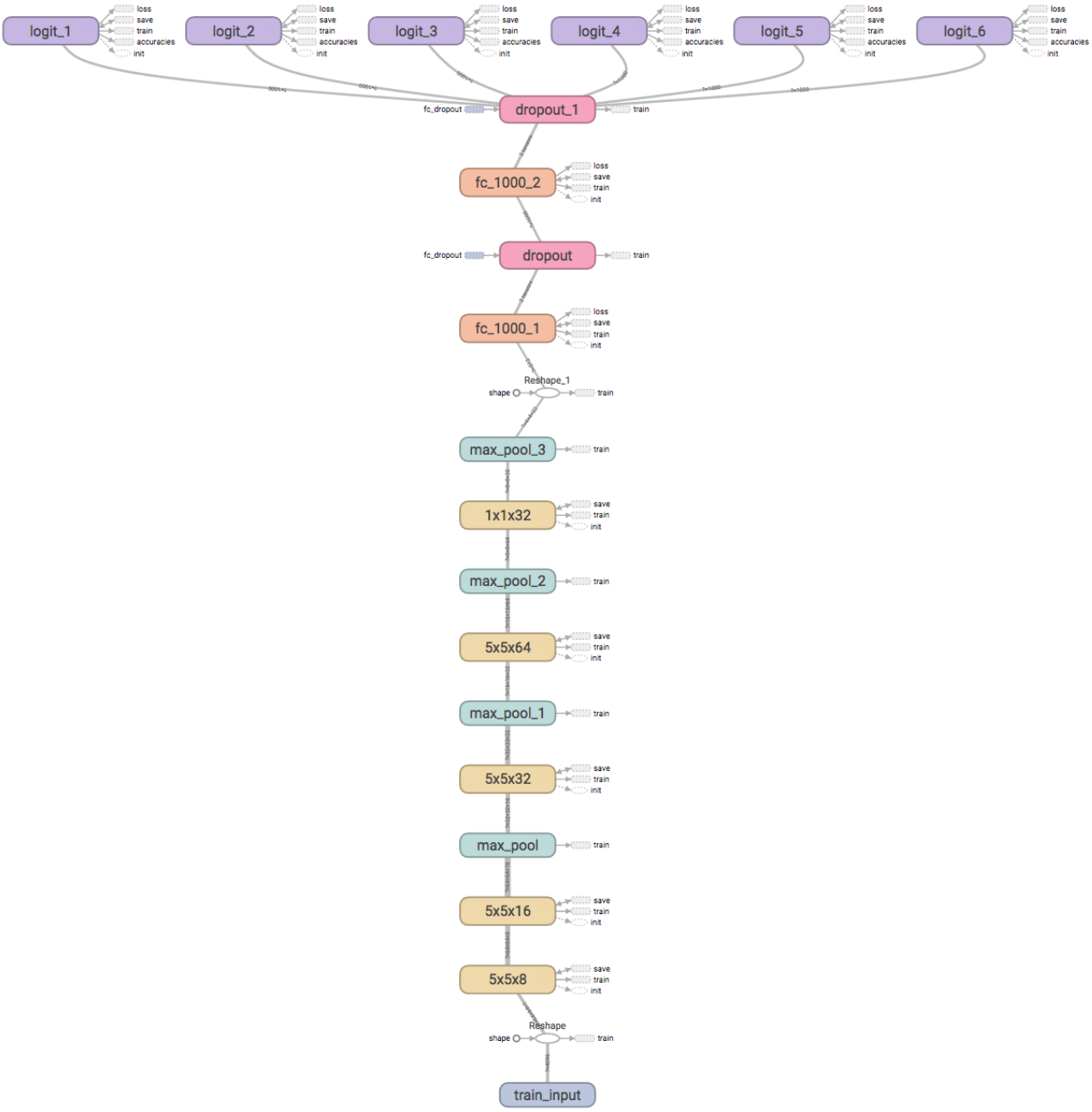
The first thing I noticed is that 10x10 convolution layers vastly increase the training time on the real-world dataset. To compensate, I reduced the convolutions to 3x3, added more layers, and introducing a second 1x1 convolution. While this reduced training time by half, the accuracy unfortunately went down.
Increasing kernel dimensions to 5x5 and reducing network depth fared even worse, with 66% overall accuracy. Finally, knowing from past experiments that results are sensitive to dropout settings, I reduced the dropout probability for the convolution layers. I also increased the size of the fully-connected layer to 5000. This improved results somewhat.
Finally, I removed dropout completely from the convolution layers, picked four 5x5 kernels, and introduced two 1000-neuron fully connected layers. This architecture fared well, yielding an overall accuracy of 75%. Given the training time of 1+ days for each model, I decided that this was sufficient.

The final CNN used by the digit recognizer was as follows:

- Four 5x5 convolution layers, learning 8, 16, 32, and 64 filter
- One 1x1 convolution reducing the number of filters to 32
- 2x2 max pooling on convolutional layers 2-5
- 1x1 stride with SAME padding on all convolutional layers
- Two 1000-neuron fully-connected layers with 50% dropout
- Rectified Linear Unit activation on all layers
- All weights initialized with a truncated normal distribution with a standard deviation of 0.1
- Biases initialized at 0
- Cost defined as the cross-entropy between the prediction and the known labels, plus L2 cost with beta value of 0.0005
- Weights learned with an Adam Optimizer with a learning rate of 0.001
- Six output layers
- Trained over 25 training epochs with a batch size of 250

Figure 9 below is a visualization of the network. Notice the six logit output layers: logit_1 outputs the length of the number, while logits 2 through 6 output the value at each digit position.

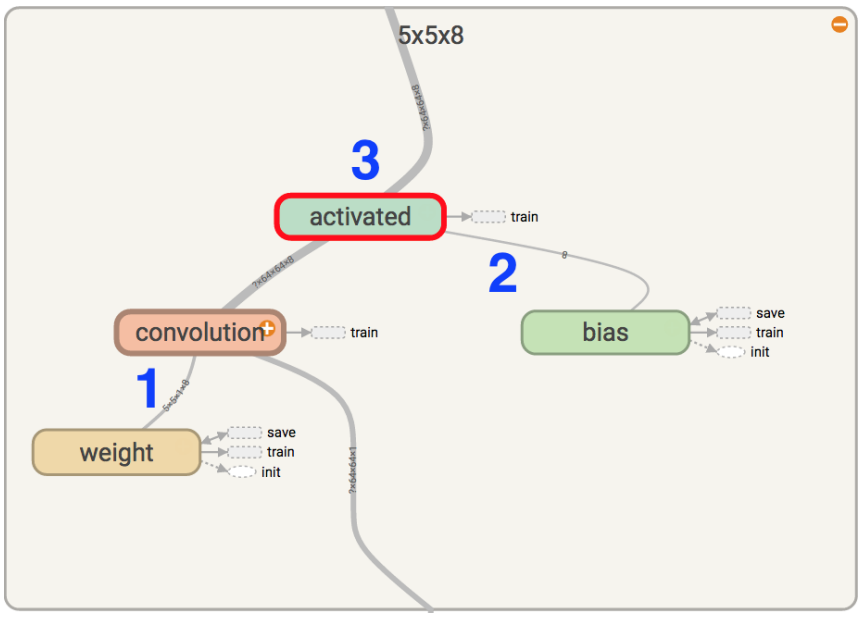
Figure 9



Zooming in on one of the convolution layers, we can see in more detail how each layer is structured:

1. The weights from the previous layer are convolved
2. A bias vector is added
3. The resulting sum is run through a ReLU activation function

Figure 10



Number locator refinement

I used the final digit recognizer model above as the basis for the number locator model. The training data for this model was generated dynamically using the process described earlier in the paper, and fed to the model in batches using yet another implementation of the datasource interface. Anticipating that the binary classification task would be a simpler problem, I slightly simplified the model parameters before attempting the initial run. The resulting model had the following parameters:

- Three 5x5 convolutional layers with depths of 8, 16, and 32
- A 1x1 convolution after the last 5x5 layer, with a depth of 32.
- 2x2 max pooling after each convolutional layer
- 1x1 stride with SAME padding on all convolutional layers
- One fully-connected layer with 1000 neurons
- 50% dropout on the fully-connected layer
- All weights initialized with a truncated normal distribution with a standard deviation of 0.1
- Biases initialized at 0
- Cost defined as the cross-entropy between the prediction and the known labels, plus L2 cost with beta value of 0.0005
- Weights learned with an Adam Optimizer with a learning rate of 0.001
- Trained over 10 training epochs with a batch size of 500

Within 4 epochs, it had achieved 97% accuracy. I found this sufficient, and did not iterate on this model further.

Algorithm Refinement

The algorithm used to determine the bounding boxes went through several rounds of tuning as well. Tuned parameters included:

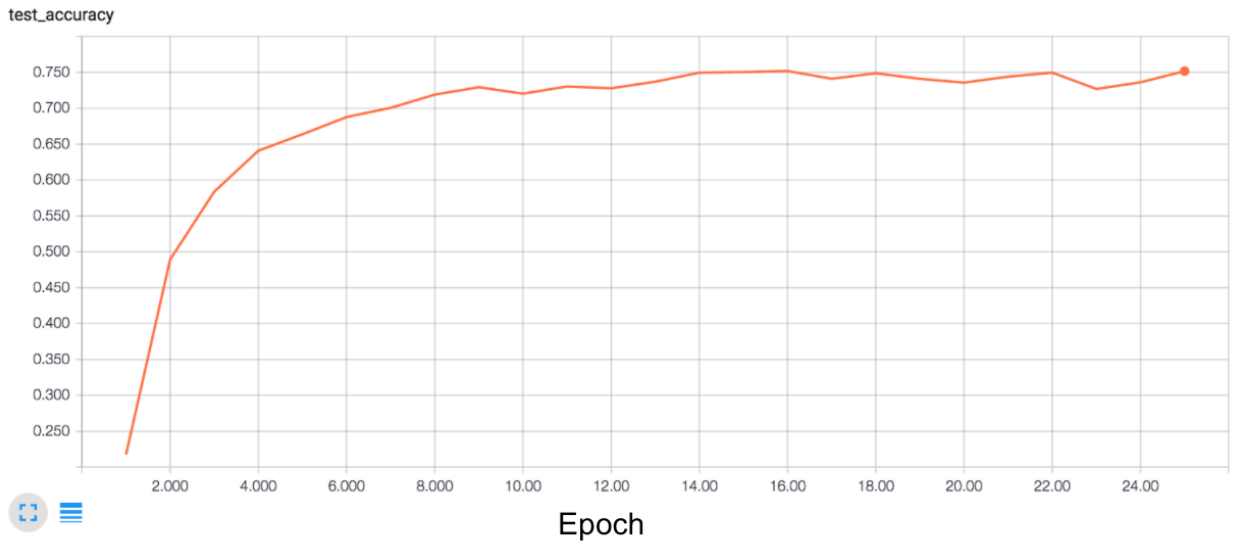
- The scaling factor, which determines the window height at each recursive step. In the final algorithm, this is set to 0.9 - each window is 90% the height of the previous one. Smaller values were attempted to attempt to speed up the algorithm, but often resulted in the algorithm missing an optimal bounding box size.
- Various bounding box aspect ratios were attempted. A higher number of aspect ratios attempted corresponds to a better potential fit to the number in the image, but also requires more processing time. The final algorithm attempts values of 1 (square), 1.5, 2, 3, and 4. This seems to capture a wide variety of number sizes, without adding too much to processing time.
- The horizontal and vertical step sizes taken by sliding window during image patch evaluation. Smaller sizes lead to better matches, but take longer to calculate. This final value is set to 10% of the window height or width, whichever is smaller. Smaller step sizes yielded better potential fits, as more positions are attempted; however this was not deemed worth the increase in algorithm runtime.
- The minimum height *H* of an image patch. This gives the recursion a terminating condition, and determines how small the numbers to be read can be. This is set to either 20% of the height of the input image, or 10 pixels, whichever is larger.
- The minimum locator classification confidence *C*. A higher value of *C* can reduce false positives, but too high a value can introduce false negatives. The value of this was set to 0.99 after examining the confidence level of various example images and seeing that the confidences tend to to run high.

Model Evaluation and Validation

Digit recognizer results

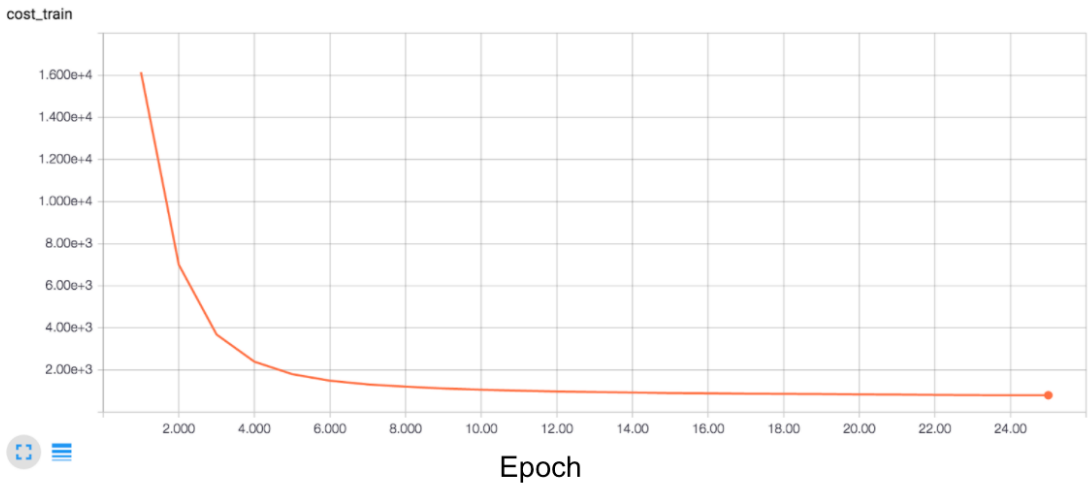
The test accuracy of the final digit recognizer model was 75%, which it achieved within 14 training epochs.

Figure 11



Looking at the training cost graph in figure 12, we can see that within 10 epochs the network had achieved close to the minimum cost value, and by the end of the 25 epochs there was barely any progress being made.

Figure 12



The graph of the test cost at each epoch (Figure 13) confirms that the network did not suffer from overfitting, which is characterized by an increase in testing cost due to network parameters being fit too closely to the known data.

Figure 13

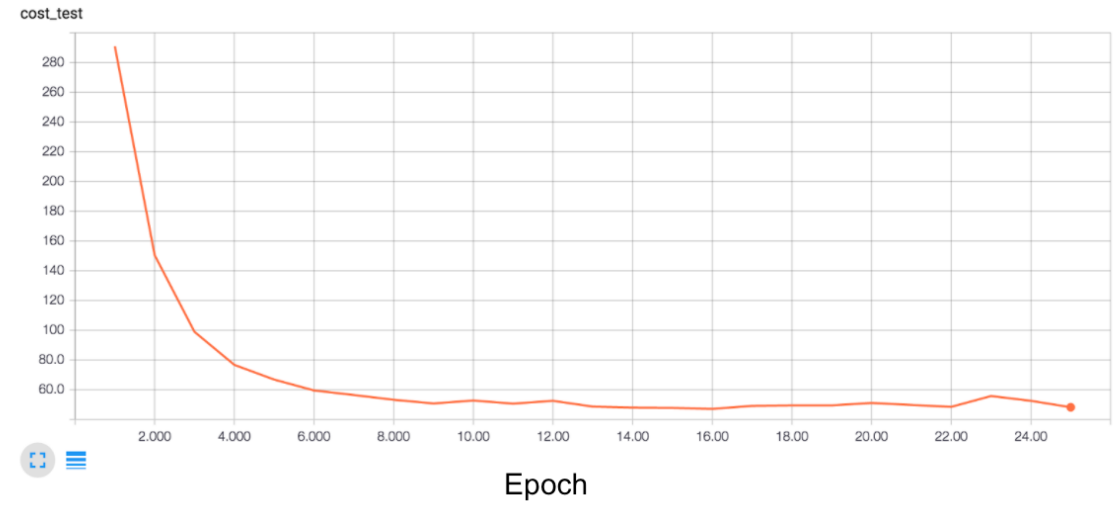
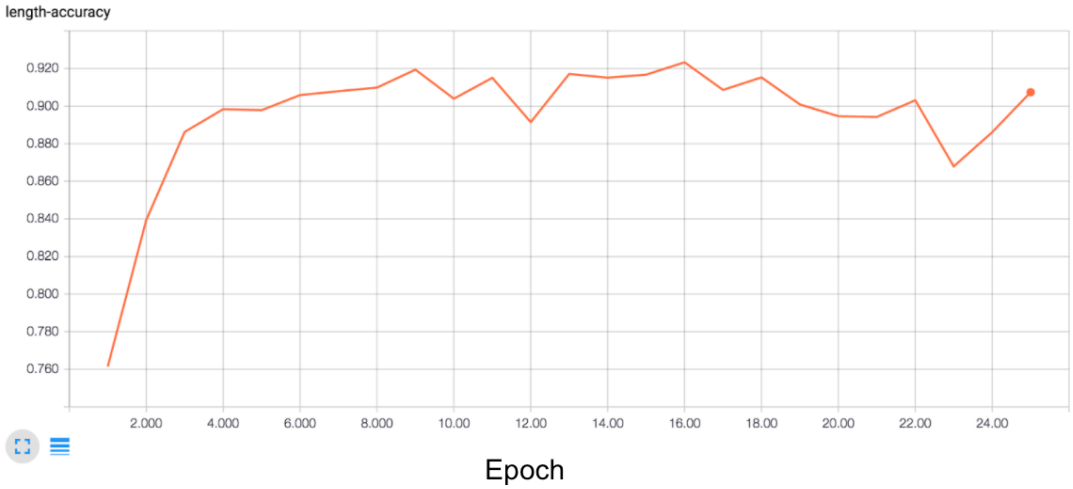


Figure 14 shows that the network correctly determined the length of the number sequence with 91% accuracy at the end of the 25 epochs.

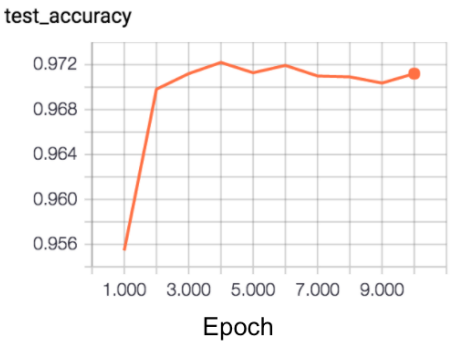
Figure 14



Number locator results

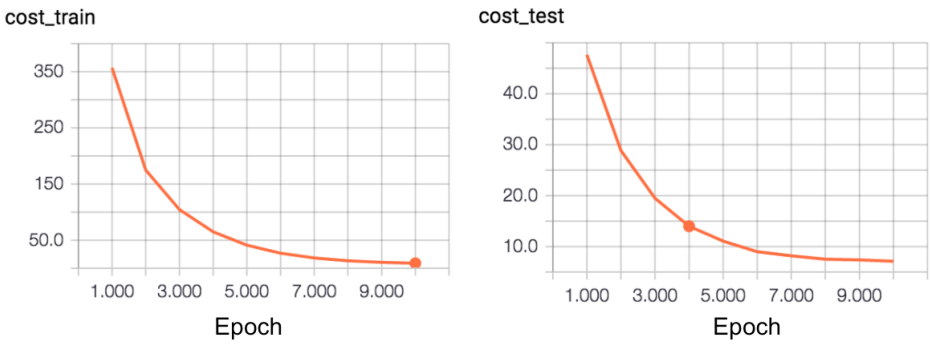
The number locator classifier achieved 97% accuracy within 3 training epochs.

Figure 15



The train and test costs level out by the 10th epoch, implying that the network has been trained to its limit.

Figure 16



Overall algorithm results

Finally, we can calculate the accuracy of the entire number identification system. This involves attempting to identify the numbers in the images *without* first cropping them to the known bounding boxes. Running the SVHN test dataset through the system yielded **58%** accuracy.

Justification

The final result compares favorably to my benchmark. The biggest surprise was the performance of the number locator CNN, which achieved a classification accuracy of over 97%.
The digit also recognizer performs better than the goal I set out for it - at 75% accuracy, it achieves slightly better performance than the 73% calculated estimate.

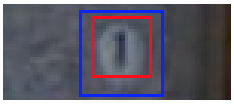
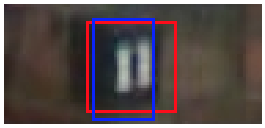
Lastly, the overall algorithm accuracy of 58% surpassed my 50% estimated benchmark goal. Considering all the sources of error inherent to the process of localizing *and* reading a number, I find 58% accuracy to be a success.

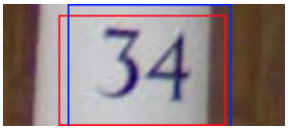
Free-Form Visualization

With all the pieces in place, we can now visualize how well the system performs the task of locating *and* reading numbers from the SVHN test dataset. The images in Table 6 below show two bounding boxes: the one calculated from the provided metadata and the one determined by the number locator. The number returned by the classifier is listed underneath each image, with correct results in green.

Table 6

Red bounding boxes were determined by the number locator
Blue bounding boxes are calculated from the provided metadata
The number below each image is the prediction
Green background represents a correct prediction
Red background represents an incorrect prediction

1.	2.	3.	4.
			
5	**21**	**16**	**1**
5.	6.	7.	8.
			
9	**11**	**83**	**180**

9.	10.	11.	12.
			
144	**16**	**34**	**2**

Looking at the incorrect predictions in the table above, we can see that the prediction was often correct when the number locator estimated a bounding box close to that of the one provided by the metadata.

Interestingly, several common source of error can be identified in these examples:

1. Errors due to the number locator mistaking extraneous features in the image for extra digits (seen in image 3)
2. Errors due to the number locator truncating the digit sequence (seen in image 7)
3. Errors due to the number locator failing to identify the number location (seen in image 8)
4. Errors due to the digit recognizer incorrectly reading an otherwise well-positioned bounding box (seen in images 2 and 12).

Reflection

To summarize, the task of reading numbers in an image required combining the solutions to two separate problems: identifying bounding boxes for digit sequences, and then interpreting each digit. In order to train a neural network to recognize digits, I began by first using a synthetic dataset that made quick iteration easier. Once I had the basics implemented, I switched to using the SVHN dataset and refined the neural network architecture accordingly. The network had six output layers: one for the number length and five for the digits; the predicted length determined now many of the digit outputs were read.

The neural network used in the number locator was based off the digit recognizer architecture, albeit slightly simplified. With this network, I was able to write an algorithm to recursively search the image for a patch that most likely contained the target digit sequence. Upon reflection, the number locator was one aspect of the project that I had not fully considered at the start. Although the neural network achieved 97% accuracy in recognizing whether a given image contained a number, actually using this classifier to identify an ideal bounding box turned out to be one of the most interesting parts of the project. For example, I had not foreseen the problem of the system identifying a bounding box that only spanned a subset of the digits - this is the reason behind the system evaluating multiple potential image patches and then choosing the best one based on the sequence length and confidence.

Overall, I believe that this is the correct method for solving this problem. With better hardware and more experimentation it is possible to build more accurate neural networks, but the two-step approach of localizing and then interpreting the number is effective. All in all, I am happy with the results.

Improvement

Several aspects of the system described in this paper could be improved.

Primarily, the digit recognizer CNN has the potential to yield much better results than the 75% accuracy achieved here. This is partially due to hardware limitations: a 25-epoch training round took 1 to 2 days on a Macbook Pro. This severely limited how much experimentation and refinement I could accomplish within reason. Likewise, it put larger, potentially more powerful convolutional network architectures out of my reach, as I did not have the computational resources necessary to train them. Goodfellow et al. achieve the best results with upwards of 11 hidden layers - given more computational power, I would attempt to train networks closer to the ones described in that paper. One caveat is that larger networks require more data to train; to compensate, I could augment the existing training data by cropping multiple differently-placed image patches out of each source image.

I also believe that the number locator had room for improvement. Although the underlying neural network was trained on the SVHN dataset with consistently-sized bounding boxes, the system still had a tendency to return positive for image patches that were too small, often resulting in only a part of the image being read. It may be possible to improve on this by including negative training examples that represent truncated numbers.

The class imbalance with respect to number lengths undoubtedly caused issues in training digit-level classifiers. In particular, the network did not have enough data to learn how to read the 5th digit. To correct for this, I actually attempted to create a synthetic dataset that stitched together SVHN digits. However, training the classifier on this artificial data yielded very poor results. Either there was a bug in the code, or the jarring transitions introduced by joining images with different backgrounds introduced complexity into the data that the CNN could not account for.
This imbalance also makes accuracy a somewhat problematic metric to rely on for the length classifier, since a high accuracy value may have limited predictive power in heavily skewed datasets. If I had to do it over again, I would use the *F1* metric instead, defined as defined as

$$2 * (precision * recall) / (precision + recall) .$$

Unlike accuracy, this metric is resilient to skewed data, and may have given me more insight into the model performance.

Technology notes

This project was developed using Python 2.7. I used the *Tensorflow* library to develop and train the underlying neural networks. The training was done on a 2013 Macbook Pro. Image manipulation was done with the help of the *Python Imaging Library (PIL)*. The web application was built using the *web.py* web framework, and deployed with uWSGI on Amazon EC2 instance.

References

- H.Erdinc Kocera, K.Kursat Cevikb, Artificial neural networks based vehicle license plate recognition
- Alex Teichman and Sebastian Thrun, Practical object recognition in autonomous driving and beyond
- Ian J. Goodfellow et al. Multi-digit Number Recognition from Street View Imagery using Deep Convolutional Neural Networks
- Jiuxiang Gu et al. Recent Advances in Convolutional Neural Networks