

# Deep Learning

A report on capstone project for Udacity ML Nanodegree

*Balamurugan V R*

## Definition

### Overview

This report introduces the problem of recognizing multi-digit sequences from arbitrary images and a deep learning based solution to this problem. This is a reasonably complex kind of Computer Vision problem. Computer Vision is a field of artificial intelligence where computers are trained to make sense of the world around us looking at it. Specific applications include image captioning, image tagging based on the objects in the images and many more. Multi-digit transcription is an application of computer vision where the algorithm attempts to extract the length of digits and the digits themselves from images of world around us. It is a fancy version of Optical Character Recognition (OCR) where digits are transcribed from printed/typewritten text from well known fonts. OCR finds applications commonly in data entry systems where documents are read and data loaded into computer systems. A most common example is reading information from official documents like passport, visa etc.,

Core OCR algorithms work by 2 basic types - Matrix matching and using extracted features. Matrix matching relies on the ability to extract the input glyph properly from the image, converting the glyph into matrix of pixels and then comparing the input to reference matrix. Comparing input to various glyphs, we can rank the glyphs based on the similarity and the glyph with maximum similarity is the most likely output. Using extracted features works by extracting features of the glyph like curves, lines, line intersections etc and then comparing the features to extracted features of the reference glyphs. If we have a database of such extracted features, we can compare the features of the input glyph to all the reference glyphs and identifying the input as a digit that has most match. This is nearest neighbors based approach and these type of algorithms spend virtually no time on training as there is no training involved but is costly in terms of storage and prediction time because we need to store virtually all the examples and compare with all of the examples during test time.

The above example introduces major components of a Machine Learning system. Basically, it starts with a set of input examples that represent the input domain of the problem. The data set can be either labelled or unlabelled. Labelled data can be used in a setting called supervised learning, where the system looks at the examples and their output and learns a model that can identify/calculate the output given input data. Unlabelled data may sound useless but unsupervised learning can be used to cluster the data into groups and segment it using some characteristic. It is intuitive that similar data points tend to cluster together and those clusters can represent some broad behavior of the data sources. Nearest neighbors based approach discussed above is a type of Machine Learning called Instance Based Learning.

Using Machine Learning in OCR makes OCR into a Intelligent Character Recognition system. Machine Learning extends the capability of OCR to identify characters from handwritten digits and images with digits from the world around us. The problem studied here is applying Deep Learning a type of Machine Learning for multi digit recognition which is a specific instance of more general arbitrary character recognition.

Deep Learning is a set of algorithms where the machine learning model attempts to learn abstractions from data. These abstractions are then used make predictions from the data and these predictions tend to be better and general because the various abstractions from data are captured in the Deep Learning models. This field is majorly inspired from neuroscientists' understanding of functioning of human brain where a cluster neurons work by mutually activating each other and the output of the neurons is driven by the interconnections of the activations. There are various architectures that implement this idea and I study the application of Convolutional Deep Neural Network for multi digit recognition from photographic images.

This solution can be extended to various applications. One baseline example application is where Google uses this to automatically update their maps database with metadata like door numbers, street numbers etc. from the street view images. Another application that uses a voice synthesizer in tandem with this can be an application to read numbers in the real world for the benefit of blind people. Extensions to this solution are possible that can be taught to read arbitrary words from sentences in the photographs.

I have used Street View Housing Numbers [dataset](#) by Netzer et al for this project. It is a huge repository of about 600000 labelled training samples preprocessed already to some extent. The dataset was generated by cropping appropriate sections from Google Street View images. It is an ideal candidate for optical recognition problems of Machine Learning. Goodfellow et al from Google have solved a [similar problem](#) with much bigger dataset, both public and internal and reported classification accuracy of about 97.84%.

## Problem Statement



(Image credit: Google)



**Output**

*Length: 3*  
*Digits: 666*

Formally, the problem can be defined as follows.

*Input: An image containing a sequence of numbers and their labels.*

*Output: Length of the sequence of numbers and the sequence of numbers contained in the image.*

It is a supervised classification problem. For practical reasons, the length of the sequence of numbers is limited to 5. This problem is formulated as classification problem with the following outputs. Length of the sequence of numbers has the sequence 1, 2, 3, 4, 5 as outputs. Each digit is labelled from 0-9 and thus contains 10 labels. Each of the individual digits can be identified using a individual classifier and the results can be concatenated to form the final sequence. One implementation detail is we need an additional label ('10' used here) to indicate absence of a digit since all the images don't contain sequences of length 5. So for the above example, the classifier will output a sequence [3, 6, 6, 6, 10, 10]. The result should be interpreted as a sequence '666' of length 3 ignoring 10s at the end.

Outline of the stages towards the implementation of the project is given below:

- Download the dataset from the source specified in the above sections
- Pre-process the data and ignore any images that have length of the sequence more than 5
- Shuffle the data and create training, validation and testing sets
- Train the model and tune the model hyper parameters as needed

Above steps are a well established road map towards solving any Machine Learning problem. First 3 steps will be fairly straightforward because the dataset I used here is very well preprocessed already. I expect that the last step above is going to be a much involved one with tons of iterations to settle on an architecture and set of optimal performing hyper parameters.

## Metrics

We need performance metrics to evaluate how well the model is performing and to guide the process of tuning the classifier.

### *Accuracy:*

Accuracy is the percentage of correctly classified samples. Say the neural network correctly identifies 80 letters in 100 samples, then we say that it is 80% accurate. I chose to use accuracy as a performance metric as it is a more intuitive measure of the performance of a classifier.

$$\text{Accuracy} = 100 * (\text{Number of correct predictions} / \text{Total number of samples})$$

Accuracy is not good performance metric for binary classification problems due to so called Accuracy Paradox. It is not a problem for this project because the prediction space is really huge as there are 5 options for length of the sequence and 11 options for each of the 5 digits. Accuracy paradox arises when the class labels are unbalanced and the classifier can reach higher accuracy by just predicting the most occurring label all the times. In case of this project, there are  $5 * 11 * 11 * 11 * 11$  unique predictions that can be made and there is no way that the classifier can get away with predicting a fixed output all the times. Looking at misclassified samples, what can be seen with respect to accuracy is, in most cases the prediction is not completely off. Either the length of the sequence or part of the digits themselves will be wrongly predicted with the most occurring classes due to the imbalance in the data. So, in some sense the classifier does a slightly better job than what

its accuracy says. Using full sequence comparison for accuracy calculation makes accuracy a robust metric for our purposes.

*Loss:*

Loss is a measure of how much the predictions deviate from the actual values. Typically, the distance between the predictions and the actual labels is the loss function of the classifier. There are different losses that can be used to measure the performance of a classifier.

Here, we use a type of loss called *cross entropy loss* which correctly quantifies how bad a classification is. Typically, classification works by estimating the probability a particular example might belong to certain class. The probability should be maximum for the correct label and minimum for incorrect labels. It is important to quantify the difference between the probabilities of correct and incorrect labels because a classifier that assigns much higher probability to correct labels compared to incorrect labels is a far better one compared to the one that assigns slightly higher probability to a correct label. Cross entropy loss makes sure that the classifier strongly classifies what it classifies and hence is the classification loss function used for this purpose. Also, it follows that, cross entropy loss function penalizes higher prediction errors more. So, the classifier can be more confident when it is certain and conservative when it is less certain during classification. The formula for cross entropy loss is as given below:

$$H(p, q) = - \sum p(x) \log(q(x))$$

In the above formula,  $p(x)$  is one-hot encoded class label which can be interpreted as probability mass being entirely on correct class.  $q(x)$  is true distribution indicating the probability distribution of classes after softmax-ing the scores.

## Data Exploration

Data source: <http://ufldl.stanford.edu/housenumbers/>

Street View House Numbers (SVHN) Dataset is a dataset of images from natural scenes containing multiple digits. This dataset is processed well and requires minimal cleanup and preprocessing. There are 2 kinds of data in this dataset.

1. Individual cropped digits

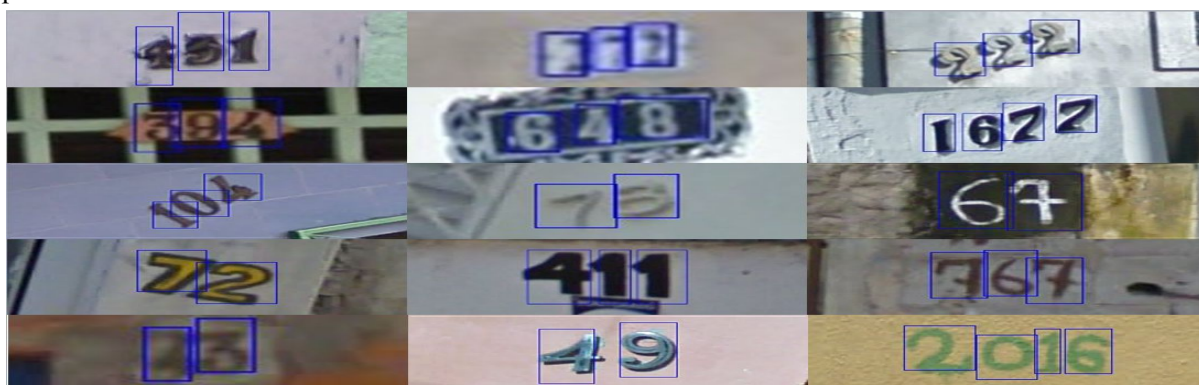
This dataset contains 32x32 images of individual digits extracted out of the natural scenes in SVHN data. They are labelled and are ideal for training a basic classifier to identify individual digits.



(Image credit: <http://ufldl.stanford.edu/housenumbers/>)

## 2. Ground truth information in variable resolution

This data set contains variable resolution images which are patches from natural scenes containing multiple digits. It is also accompanied by metadata that, for each images, informs about the bounding boxes for each individual digits and labels corresponding to each of the individual bounding box. For this project, I calculated the smallest bounding box containing all the digits, extracted that bounding box and resized the extracted image to resolution 32x32 pixels.

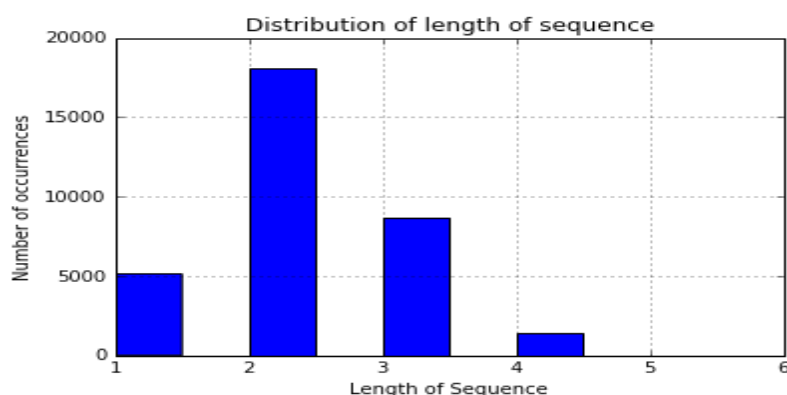


(Image credit: <http://ufldl.stanford.edu/housenumbers/>)

## Exploratory visualization

Number of training digits:	73257
Number of testing digits:	26032
Number of training images:	33364
Number of test images:	13060
Mean intensity of training images:	139.51
Standard deviation of intensity:	60.50
Image resolution:	arbitrary
Number of channels:	3 - RGB
Output features:	Length of the sequence of digits 'N' and 'N' digits

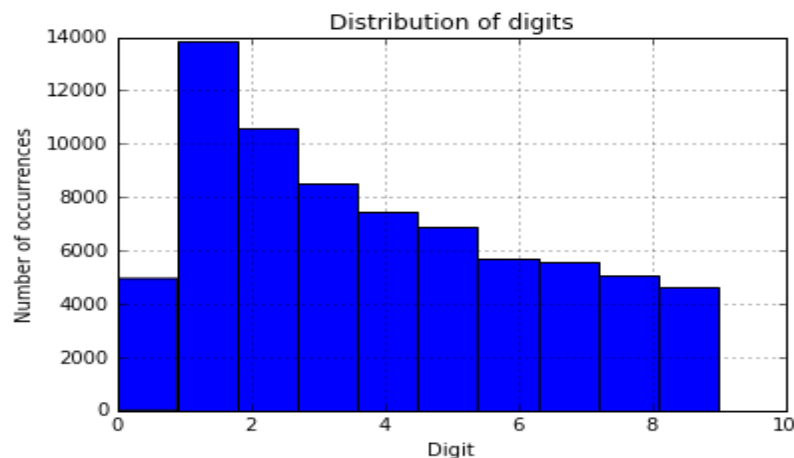
Distribution of length of sequences:



Length of sequence	Occurrences
1	5135
2	18126
3	8689
4	1433
5	9
6	1

From the distribution of length of sequence of digits above, apparently lengths  $> 4$  don't have enough training examples at least in the training dataset that is used in the project. So it is likely that the classifier will make mistakes when looking at images with number of digits more than 4.

Distribution of digits among the training images:



The distribution is slightly skewed towards left starting from 1. Interestingly enough, digits 0 and 9 seem to contain equivalent number of examples and since structurally they look similar, it is likely that the classifier makes mistakes confusing one with another. Except for digits 1 and 2, the number of other training digits seems comparable for the purpose of this project. It is also worth noting that since digits 1 and 2 contain significantly higher number of training examples, it is likely that the model predicts them often.

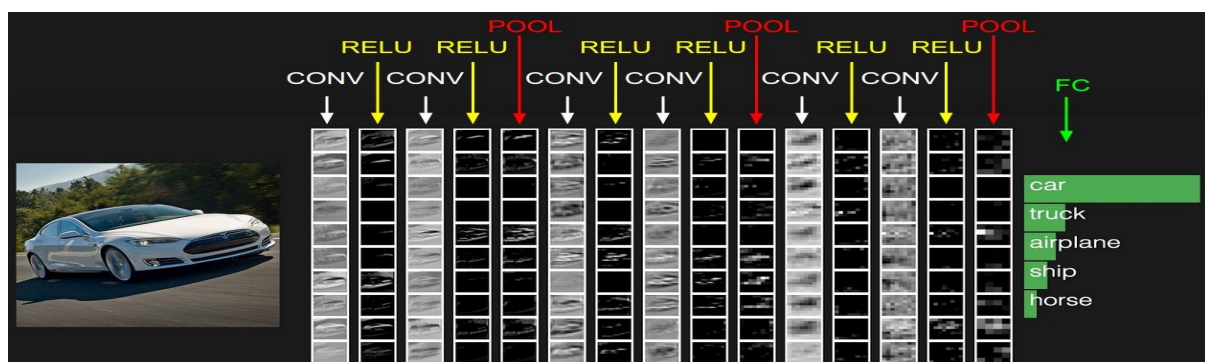
## Algorithms and Techniques

In this project, we use a particular type of Deep Learning called Convolutional Neural Networks (CNN). CNNs are the latest state of the art in visual and speech recognition and in variety of other Machine Learning tasks. They basically work by extracting abstractions of input features from a finer level to much higher level in the earlier layers thus providing a much richer input for the classifiers in the later stages of the pipeline. Particularly, they don't require hand crafting of features like those needed by statistical models and hence prove to be ideal where we have enough data and computing power at our disposal. I have used CNNs with L2 regularization and dropout to control overfitting. Also, I have used ADAM optimizer that is adapting required less control.

- Input layer  $[48 \times 48 \times 3]$  will hold the raw pixel values of the image, in this case an image of width 48, height 48, and with three color channels R,G,B.
- Convolutional layer will compute the output of neurons that are connected to local regions in the input, each computing a dot product between their weights and a small region they are connected to in the input volume. This may result in volume such as  $[48 \times 48 \times 16]$  if we decided to use 16 filters.
- ReLU layer will apply an elementwise activation function, such as the  $\max(0, x)$ . This leaves the size of the volume unchanged ( $[32 \times 32 \times 16]$ ).
- Pooling layer will perform a downsampling operation along the spatial dimensions (width, height), resulting in volume such as  $[24 \times 24 \times 16]$ .
- Dropout layer drops/keeps neurons with certain probability (a hyperparameter) during training to prevent neurons co-adapting too much and thus overfit the training data.
- There are 'N' stacks of above 4 layers. 'N' is governed by the computational power and good enough to fit the data properly and not overfit.
- Fully connected layer will compute the class scores, resulting in volume of size  $[1 \times 1 \times 5]$  (For length of sequence)], where each of the 5 numbers correspond to a class score, such as among the lengths of the sequence.
- Finding suitable weights and biases for the network so that it classifies images properly is optimization problem and Optimizer applies gradient descent algorithm to find appropriate weights and biases in iterative training steps.

Stacks of convolutional layers composed of convolutions, ReLU and pooling are efficient feature extractors thus more ideal candidates for this problem but are computationally costly and needs tons of training data.

An illustration of how CNNs work:



[Image source: <http://cs231n.github.io/assets/cnn/convnet.jpeg>]



## Benchmark

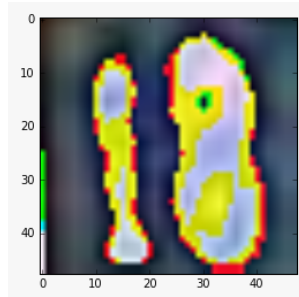
The most obvious benchmark for this problem are the results published by Google that achieved accuracy of about 97.84%. There are results published by individuals and practitioners working on the same problem and they achieve accuracy of about 93%. The benchmark by Google uses sophisticated distributed computing and weeks of training to achieve the results. Typically, deep learning systems are compute intensive and hence require GPUs for turnaround in reasonable time. Since I am restricted in hardware available at my disposal, I trained a simple neural network on my computer to get an idea about the accuracy I should strive for. The test accuracy came about 53% which sounds like a low bar to achieve. So, for this work I have decided to use a reasonable middle ground that's about 75%.

## Methodology

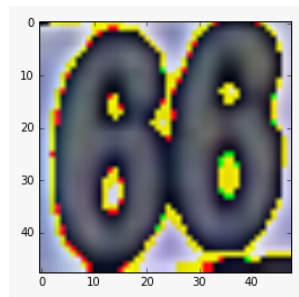
### Data preprocessing

The core dataset for this problem contains natural images that contain sequences of digits. There is associated metadata file that contains information about the bounding boxes for individual digits and also corresponding labels. Following preprocessing steps were applied:

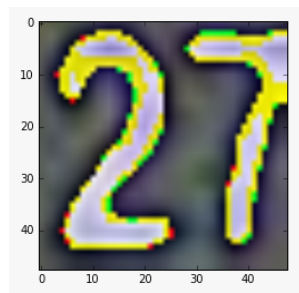
[ 1 1 8 10 10 10]



[ 1 6 6 10 10 10]



[ 1 2 7 10 10 10]



1. For each image, crop the image to the smallest bounding box containing all the digits.
2. Resize the images to 48x48. After some experiments, 48x48 seems to retain most of the useful information without much distortion.
3. Subtract mean pixel value from all the pixels of the image to center them around 0 and then divide them by pixel depth 255 so that they are in range (-0.5, 0.5) i.e. normalized.
4. For each of the image, generate a sequence containing length of the sequence of the digits (actually it is length-1 to make sure the classes start at 0) and the digits themselves. Fill the holes with label 10.
5. Randomly shuffle the training images set and do a 90:10 split to get training set and validation set. Random shuffling was to make sure that there is no further imbalance in the distributions of the minibatch.



6. Store the training, validation and test datasets in a dictionary and pickle the dictionary for later use..

I have preprocessed the data as explained above and generated a tarball and uploaded it to dropbox for use.

Link to processed data: <https://www.dropbox.com/s/yiogno0r2ac2aym/data.tar.gz?dl=1>

Code in ipython notebook “load\_data.ipynb” downloads and generates the pickle file suitable to run the training on.

## Implementation

Deep learning systems are typically composed of following layers:

**Input Layer:** Input images of size 48x48 and 3 channels(RGB)

### Feature extraction layers:

*Convolutional Layer* - 5x5 kernels convolved over the previous layer to generate ‘N’ feature maps.

*ReLU Activation Layer* -  $\max(0, x)$  activation applied to the inputs to introduce the non-linearity.

*Max Pooling Layer*- 2x2 kernels moved over the inputs and select max in every kernel.

*Dropout Layer* - Dropping some connections at a layer with some probability to reduce overfitting.

### Classification Layer:

*Readout layer* - Applies linear transformation to the values from the previous layer, then applies softmax to the output to get the probability distribution of the output classes.

### Hyperparameters:

BATCH\_SIZE - Number of training images to consider in each step - Set to 64 all through the implementation

LEARNING\_RATE - Amount of change to learnable parameters in each step - Started with 0.01, observed that the classifier took longer time to converge. Tuned it down and observed better behavior. Settled on 0.00005 finally.

LAMBDA - Regularization rate to control overfitting - Started with 0.00001 and figured that it is not enough because the validation loss started increasing after certain epochs during training. Increased it to 0.0005 and observed the model didn’t overfit any further.

NUM\_STEPS - Number of training steps to run - 150000 steps. The more training the more merrier!

CDEPTHX - Depth of convolutional layer ‘X’ - 3 layers of depth 16, 32 and 64 (Started with depths 32, 64 and 128 but that was computationally more complex)

Dropout probability - Probability with which to keep nodes in a layer - Started with dropout only in the last fully connected layer but later added dropout at all layers to prevent overfitting later.

## Optimizer - ADAM

1. Read the input pickle file created in the preprocessing step above.
2. Setup initializer functions for all the weights and bias variables of the model.
3. Setup the accuracy function
4. Setup the convolutional net using 3 convolutional layers and 6 readout classification layers.
5. Setup the loss function that calculates the mean cross entropy loss including the regularization loss.
6. Add image summaries for train, test and validation data, scalar summaries for all the weight, bias variables and training and validation accuracy and loss values.
7. All the above operations setup the tensorflow computation graph.
8. Setup a tensorflow session, create mini-batches of input data and run computation steps one at a time for each minibatch.
9. For every 10 steps, calculate the training and validation accuracy, loss and add all the summaries for visualization.

## A timeline of model architecture evolution

Description	Observations
<p>Initial model: Used grayscale images, 3 convolutional layers (depths 32, 64 and 128), pooling at second and third layer, 2 fully connected layers with dropout after first FC layer</p> <p>CDEPTH1 = 32; CDEPTH2 = 64; CDEPTH3 = 128</p> <p>LEARNING_RATE = 0.001</p> <p>DROPOUT = 0.8 LAMBDA = 0.00001</p> <p>Weight initialized from random distribution with std dev: 1</p>	<p>Test accuracy: 30%</p> <p>Prolonged training times</p> <p>Spent significant amount of time trying to debug stuff here but to no luck</p>
<p>Noted from the forums that if a human can't make sense of an input image, CNN can't make sense of it either. Observed that grayscale conversions distorted the images so much that the algorithms could barely learn anything. Removed grayscale conversion from the process.</p>	
<p>Same architecture with RGB images as inputs</p>	<p>Validation accuracy shot to about 60% in 8 epochs. Prolonged training hours indicated unnecessarily complex architecture.</p>
<p>Reduced the depth of convolution layers by half. Removed one FC layer at the end and introduced dropout after third convolution layer.</p> <p>CDEPTH1 = 16; CDEPTH2 = 32; CDEPTH3 = 64</p> <p>LEARNING_RATE = 0.00005</p> <p>DROPOUT = 0.8 LAMBDA = 0.0001</p> <p>Weights initialized from normal distribution with stddev <math>\sqrt{2/\text{\#input neurons}}</math></p>	<p>Much better training times. Model was converging pretty rapidly. After certain epochs of training, started overfitting. Test accuracy of about 64%.</p>

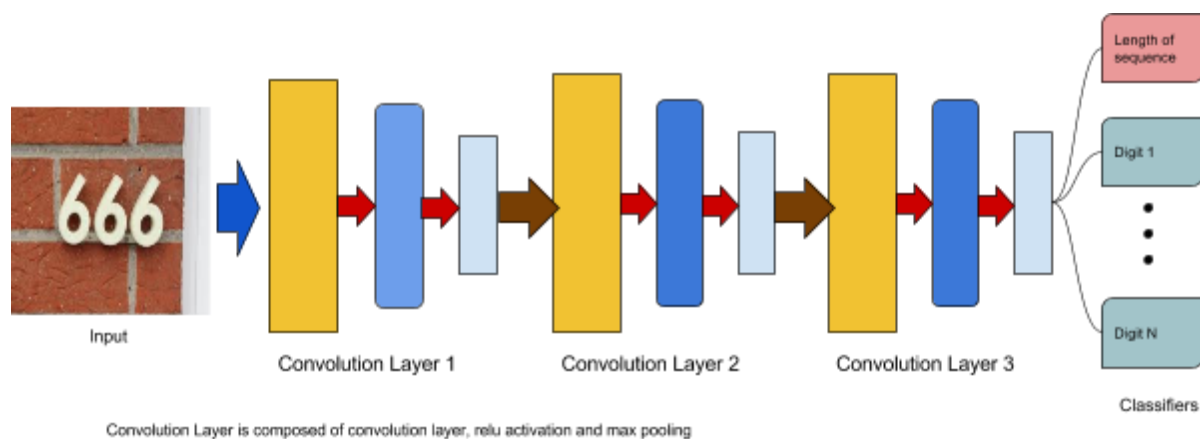
Refinement	
Introduced dropouts with keep probabilities 0.8, 0.6 and 0.5 after each convolutional layers respectively and increased regularization rate to 0.0005.	After training for longer times, no indication of overfitting. Achieved the benchmark of 75% test accuracy in 300 epochs. Errors were marginal.

## Refinement

Implemented the architecture and achieved a test accuracy of 64% after about 100 epochs. Training accuracy went to about 93% whereas validation accuracy stayed at about 70% indicating a condition of overfitting. Introduced dropout after final convolution layer and increased the regularization rate. After training for about 100 epochs again, the test error raised to about 68%. This proved that regularizing further will help improve the situation. Introduced dropout after all the convolutional layers, and increased regularization rate further. With about 200 epochs, the testing error reached about 74.5%. Another way to control overfitting is to use a lot of training examples. The data source contains extra dataset that contains a huge number of training examples but my hardware limitations didn't allow me to use it. Given that we can see that the situation is classical overfitting scenario, using the additional extra dataset will improve the testing accuracy significantly.

After the refinement,  $\text{LAMBDA} = 0.0005$ ;  $\text{DROPOUT1} = 0.8$ ;  $\text{DROPOUT2} = 0.6$ ;  $\text{DROPOUT3} = 0.5$ ;  $\text{NUM\_STEPS} = 150000$

## Final architecture



# Results

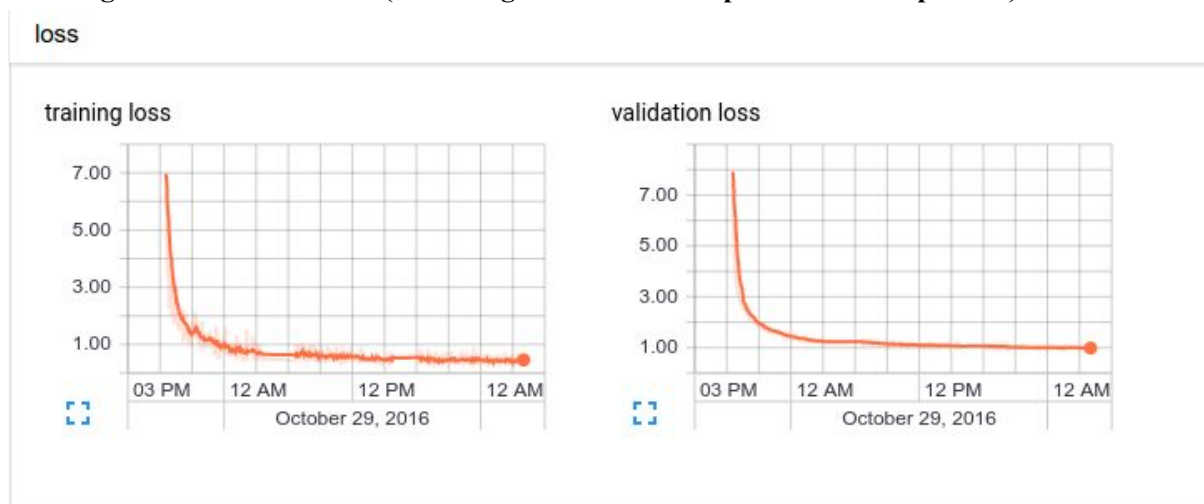
## Model Evaluation and Validation

The layer configurations, parameter values and the architecture was finalized because they performed best among the tried configurations in terms of training times, generalizing and producing adequately accurate results. A validation set of about 200 images was kept aside to monitor the performance of the classifier during training and stop earlier of the model starts diverging.

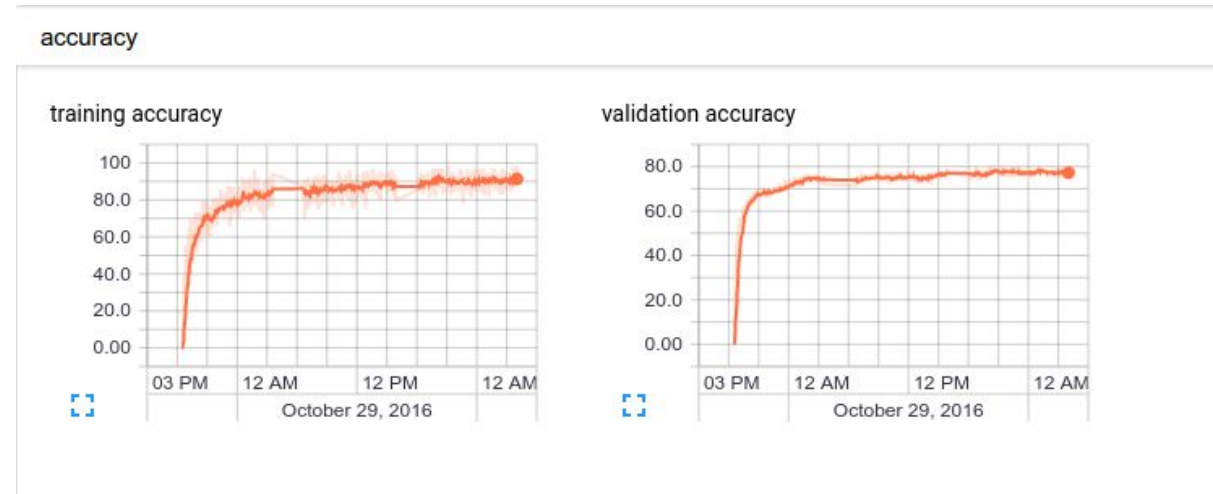
- The model contains 3 convolutional layers of depth 16, 32 and 64 respectively.
- All the weights were sampled from truncated normal distribution with standard deviation  $\sqrt{2/\text{number of input connections}}$
- All the bias units were initialized to be a constant value of 0.01.
- ReLU activation is used after each convolutional layer.
- Max pooled with strides 2 after each convolution thus cutting down the spatial dimensions of the previous layer by 2.
- To control overfitting, I regularize aggressively using L2 regularization and also employed dropout. During training, about nodes are dropped with probability 20% after first layer, with 40% after second layer and with 50% after final layer.
- 1000 steps is roughly 2 epochs here and trained the model for 300 epochs in 3 sessions.

The learning curves below compare training and validation loss for about 300 epochs run in 3 different batches of 100 epochs each. Model was checkpointed and restored for each training session. It can be clearly seen from the training loss curve below from the smooth sections of the curve. The model converged pretty early in the training but took really longer training sessions to improve the accuracy. There is no overfitting now as the validation error never increases or gets worse. Further improving the model's accuracy depends on using additional training data, increasing the number of layers in the network and training for still longer hours.

### Training and validation error (3 training sessions of 100 epochs/50000 steps each)



## Training and validation accuracy ((3 training sessions of 100 epochs/50000 steps each))

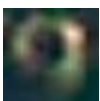




## Justification

I could achieve 75% test accuracy, achieving the benchmark defined earlier, on 1000 test samples, training for only 300 epochs and using only about 30000 training samples. Given that the model reaches the defined benchmark using a simple architecture and low end hardware, I can justify that the solution is adequate.

Implementation	Accuracy
Logistic Regression on cropped images	23%
ANN on ground truth data	53%
Earlier architecture on gray scale images	30%
With RGB images and conservative regularization	64%
With RGB images, aggressive regularization and dropout aka Final architecture	75%










Looking at the mis-classified samples, I figured that the images were marginally misclassified due to either poor contrast, distorted, background noise confusing the classifier etc. There were some gross misclassifications as well where most of the cases were to do with orientation of the digits in the images. Some samples given below:

Image	Prediction	Actual
	[0 0 10 10 10 10]	[0 9 10 10 10 10]

	[0 1 0 10 10 10]	[1 1 0 10 10 10]
	[0 1 7 10 10 10]	[0 7 10 10 10 10]

## Conclusion

### Free form visualization

								
Some examples of correctly classified samples. We can see that the classifier identified the length and digits though there were different fonts, slight distortions and different orientation.								

## Reflection

Following steps summarize the process used for this project:

1. The initial problem was decided based on the available options in the Udacity capstone projects list.
2. Learnt basic deep learning from the Google's deep learning course.
3. The data was downloaded and explored. Initially, a CNN was trained on the cropped images to get an idea how a tentative architecture will look like. Reached test accuracy of about 91% with some tuning of the hyper parameters.
4. A benchmark was created based on existing literature and some tests in the available hardware.
5. The architecture was augmented for additional classifiers for the length of sequence and digits themselves.
6. It was then trained using the data (multiple times, until a good set of parameters were found)

Steps 3 and 6 were more time consuming in 2 dimensions 1) learning curve involved and 2) technical limitations due to available hardware. Started using subsets of data for validation and training to handle the memory constraints I had. Learnt a great deal about deep learning in general, weights initialization, tuning learning rate and dropout, model evaluation and validation, setting up and debugging tensorflow models. The aha! moment occurred when I looked at the grayscale training samples in the earlier stages, then suddenly realized - "wait, if I can't make sense of this image how come the CNN is going to". The training and tuning after this realization was like a breeze with sensible feedback from the system. Debugging before that realization was a nightmare. The most

frustrating experience was prolonged feedback times for each tuning iteration as the hardware I had was horrendously slow.

## Improvements

As noted in many places in the sections above, hardware posed a major challenge in the implementation. So, one immediately obvious improvement to the project is to use GPU banks to run the computational graphs on. Also with enough main memory, we can leverage all the training data which will definitely help improve the accuracy of the classifier. Other functional improvements can be to utilize the feature detectors in the earlier stages of the network to perform localizations by adding regression units instead of classifiers. One can then estimate the bounding boxes of the digits from arbitrary images. This is important to utilize this classifier in real world applications.

## References

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

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*. ([PDF](#))

<https://github.com/priijip/Py-Gsvhn-DigitStruct-Reader>

<http://cs231n.github.io/>

[https://en.wikipedia.org/wiki/Optical\\_character\\_recognition](https://en.wikipedia.org/wiki/Optical_character_recognition)

[https://www.tensorflow.org/versions/r0.11/api\\_docs/](https://www.tensorflow.org/versions/r0.11/api_docs/)