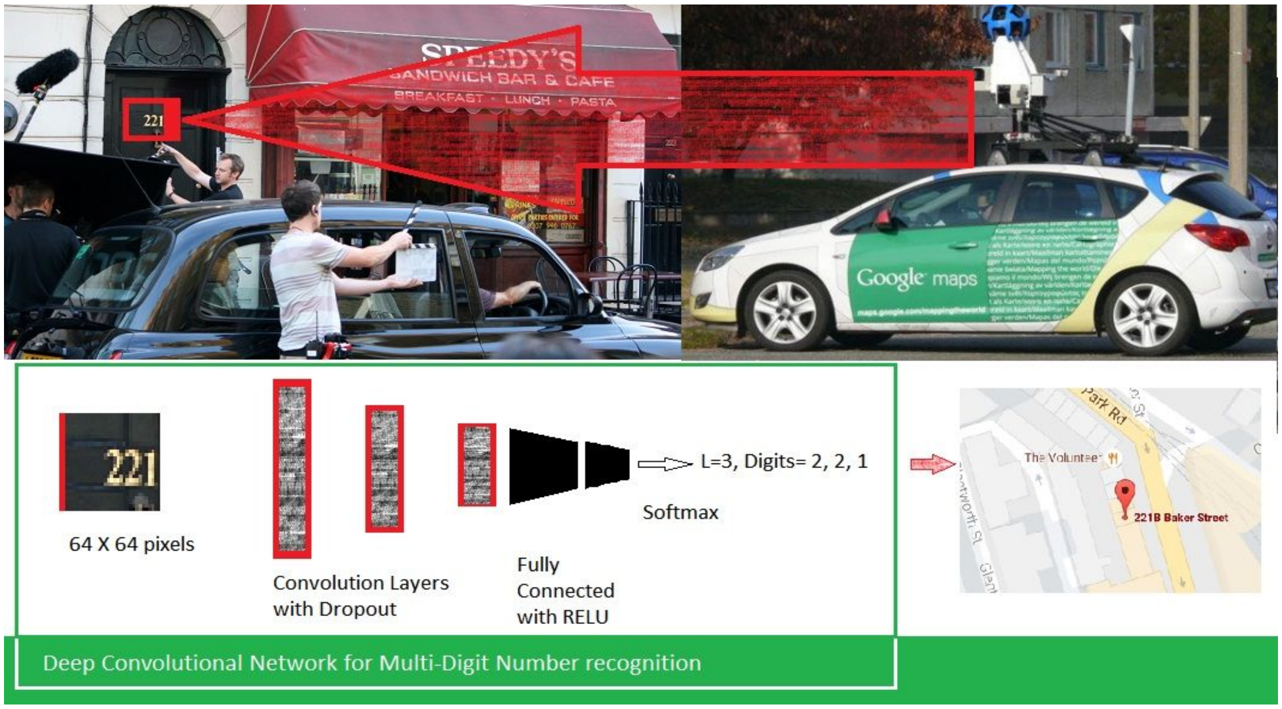
Capstone Project
Udacity Machine Learning NanoDegree

Deep Learning using TensorFlow to recognize house numbers in Street View House Number Images

- Implementation of Google's algo on multi-digit number recognition using Deep CNN

Sample use-case of recognizing sequence of numbers in house number images


221
64 X 64 pixels
Convolution Layers with Dropout
Fully Connected with RELU
L=3, Digits= 2, 2, 1
Softmax
221B Baker Street
Deep Convolutional Network for Multi-Digit Number recognition

Definition

Project Overview

Humans and machines capture images of their surroundings on a daily basis. Many of these images contain a sequence of digits, in a natural and unconstrained environment. These digits often do not conform to standard fonts or typography. These digits contain important metadata about their environment. And extracting digits from these images at a large scale requires that the process be automated, and has been a challenging task for decades. Recent developments in Machine Learning capabilities allow efficient solution to this problem, that is comparable to human capabilities in accuracy.

This project provides a basic implementation to automate recognition of such sequence of digits from images of house numbers, in natural light and surroundings, taken by Google's Street View Cars. Such an automated process would be helpful in mapping house numbers to geographical coordinates.

Problem Statement

Identification of house numbers in images involves core challenge of classifying a set of pixels into different digits from 0 to 9 and identifying their sequence in a specific image.

Classification tasks fall into the category of Supervised Learning. Feature Extraction is an important and tedious prelude to actual classification. Neural Network provide a way to bypass manual feature extraction, as well as a generic way of modelling real world inputs.

We intend to employ a specific form of Neural Network, that is geared towards classification of large image data, which is the input for this project - Convolutional Neural Network.

Data: In our endeavour to identify multiple digits in images of natural scenes, we will use Street view House Number data [REF: #1] for training of our model and also to test it. Implementation approach will take whole image as input, and try to identify digits in that image. ThIs is based on original images in Format #1 of publicly available SVHN data, and not on Format #2 of that data which has already cleaned image data.

Model: To learn of shapes of digits in unconstrained images, we will employ 8 Convolutional Neural Network layers as it allows spatial translation. And have 2 fully connected layer and six softmax classifiers to predict different digits and length of number sequence in each image.

Assumption here is that maximum number of digits in sequence will not be more than 5. The trained model will take an image as input, without information about individual digits. And it will be trained to predict the specific digits present in the image in a specific sequence, and length of this sequence.

Performance Metrics

For a classification problem, there are many ways to measure performance. Some of them are:

a. Accuracy: This is defined as number of correct predictions as a fraction of all predictions, for a subset of labels. This project will use Accuracy as measure of performance for sake of simplicity and humans' intuitional bias towards it, and also to compare it against the performance established by Google's model.

b. F1 / Precision and Recall: Precision is the fraction of True Positives out of all Positives . Recall is fraction of True Positives, over sum of True Positives and False Negatives. Since this project is for multi-class labels, we can average precision and recall over all labels. This is a better measure of classification performance than accuracy.

c. Log-loss: This is cross-entropy loss defines over probability estimates. It is defined as the negative log likelihood of classified output over given the ground truth labels. This is already used by model as part of softmax cross entropy function of TensorFlow, to tune the weights and biases of the network.

d. ROC Curve: This represents sensitivity of the model, and is a plot of True Positives Rate as a fraction of False Positive Rate, for a binary classifier. For projects like this one, which is a multi-label classification problem, Area Under ROC Curve for all labels can be used to calculate model performance. But Area Under Curve has recently been found to have serious deficiencies [REF # 6]. So we will stick to Accuracy as a simple measure of performance in this project.

So performance of our model will be based on accuracy in predicting length and digits of house numbers embedded in test image data.

Analysis

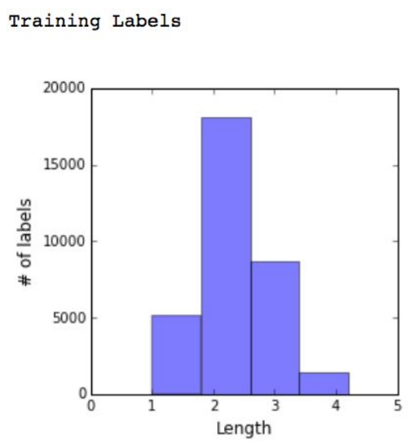
Data Exploration

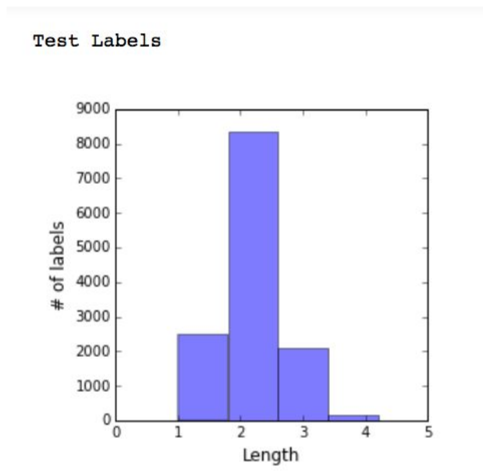
Distribution of digits and length of house numbers
House Numbers have different length, and have different digits at different places.
We want to compare the length of house numbers and distribution of digits in training images to that in test images.

There are 2 plots below. First plot is for training data and second plot is for test data.
Data points for training data and test data has been taken post normalization using pixel depth=255.

First set of plot below in blue color is a histogram of length.

Distribution of length across house numbers: Comparison of distribution of length of house numbers in Train and Test Data:

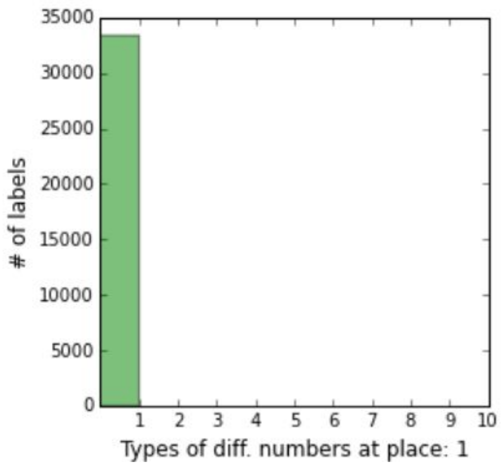


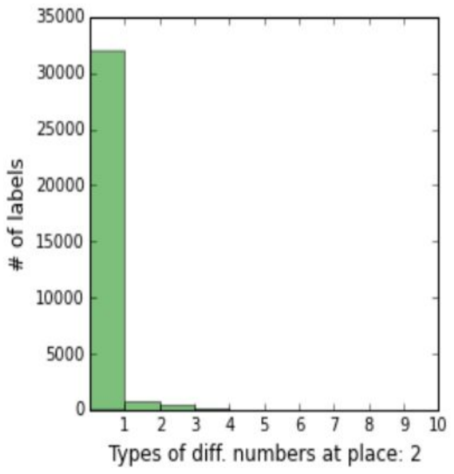


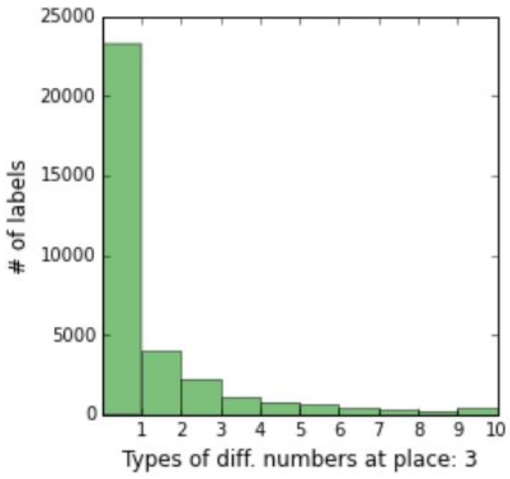
Above histograms show that most house numbers are of length 2, both in Train and Test data.
Also, there are almost no house numbers having length=0 and almost no house number having length=5.

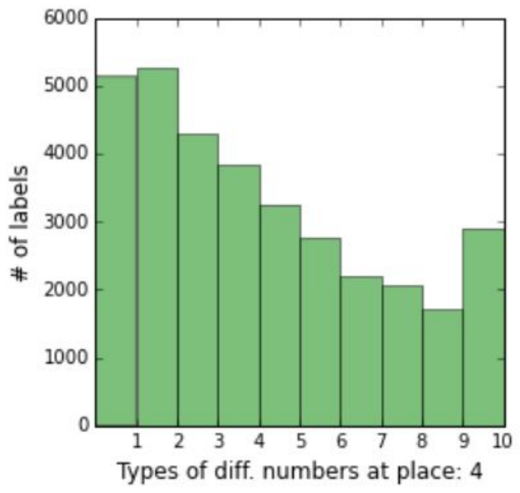
Distribution of 0-9 digits across house numbers:
Comparison of distribution of different digits (0,1,2,3,4,5,6,7,8,9) of house numbers in Train and Test Data.

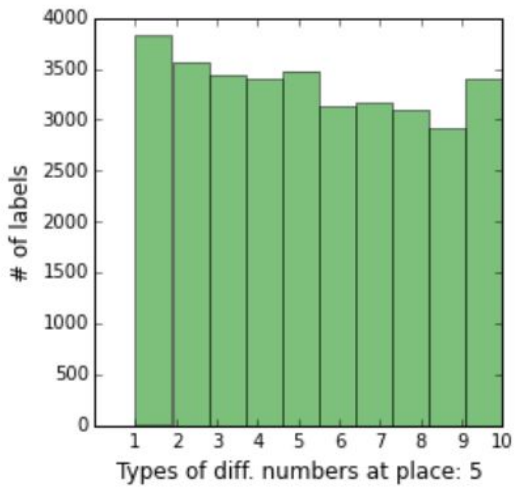
Training Data Histogram - 10 on X-axis represents digit 0 (ZERO).



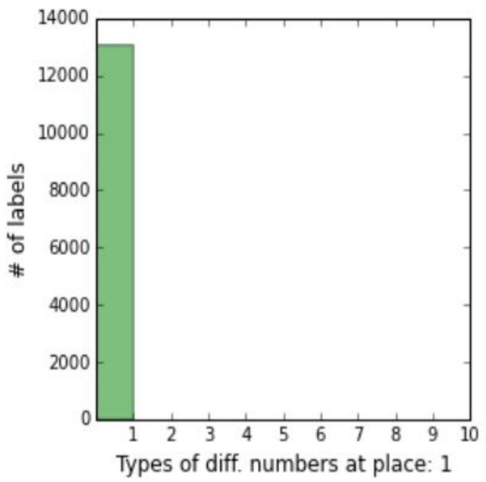


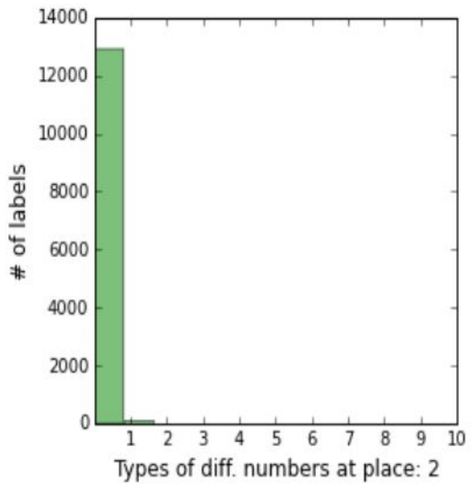


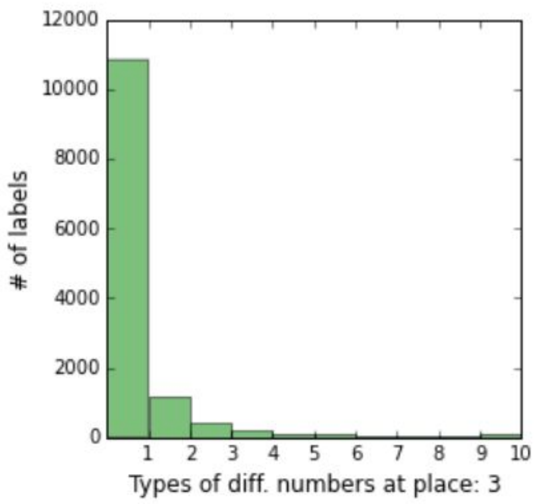


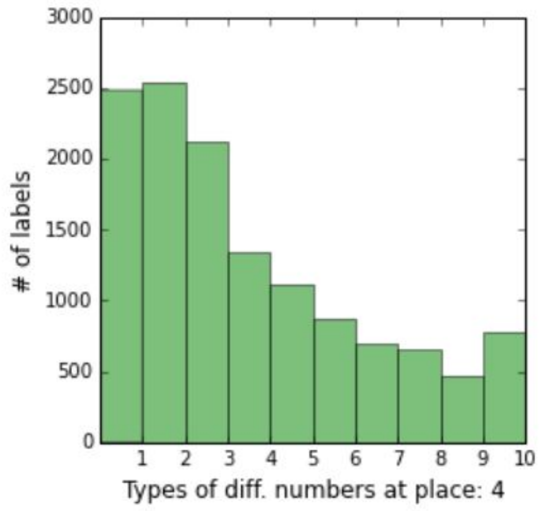


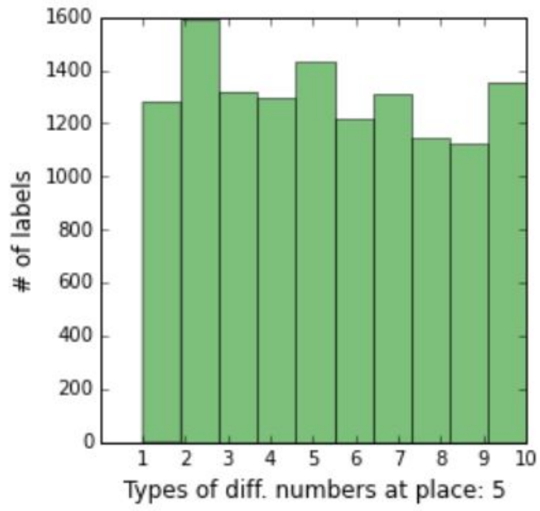
Test Data Histogram - 10 on X-axis represents digit 0 (ZERO).









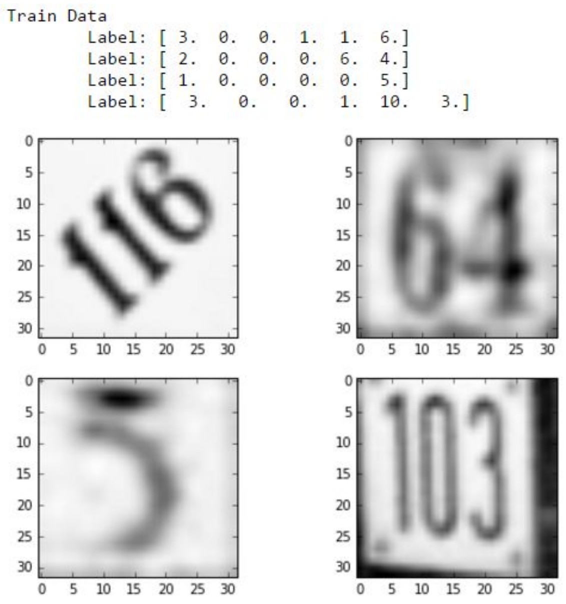


We can make following simple observations from the 2 sets of plots above:

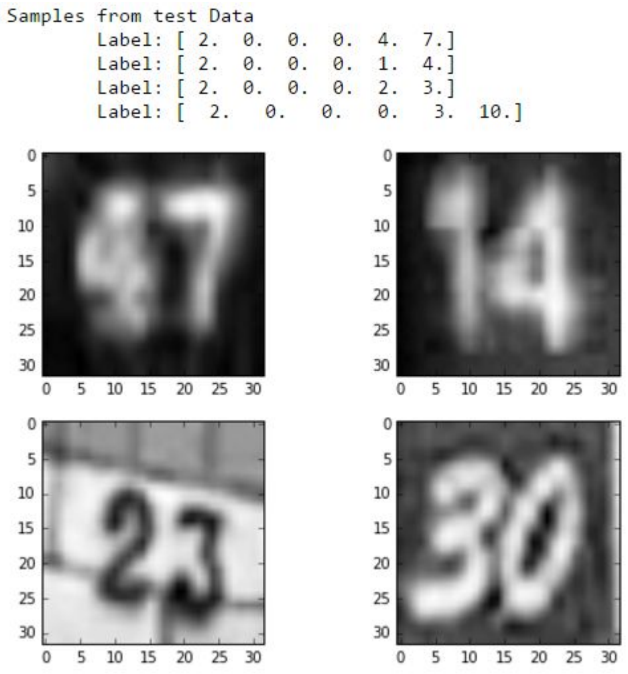
1. All digits from 0 to 9 occur at 4th and 5th place in house numbers.
2. Distribution of house numbers in test data is similar to that in training data.
3. Weights and Biases for Digits 1 through 3 will remain sparse.

Exploratory Visualization

Here are visual samples for train and test data after normalizing original images and converting them to grayscale , that include labels and digits from house number.



```
Train Data
        Label: [ 3.  0.  0.  1.  1.  6.]
        Label: [ 2.  0.  0.  0.  6.  4.]
        Label: [ 1.  0.  0.  0.  0.  5.]
        Label: [  3.   0.   0.   1.  10.   3.]
```



```
Samples from test Data
        Label: [ 2.  0.  0.  0.  4.  7.]
        Label: [ 2.  0.  0.  0.  1.  4.]
        Label: [ 2.  0.  0.  0.  2.  3.]
        Label: [  2.   0.   0.   0.   3.  10.]
```

Algorithm and Techniques

Following techniques will be used for this project:

a. <u>Convolutional Neural Network (CNN) layer</u>

We know about our input data that it is composed of images of sequence of digits. But this sequence of digits can occur at any place within the image. This allows us to use the technique that is most suited to handle space invariance of input. Same digits occur at different places, our model should share weights. All these intuitions suggest that our model should use Convnets for training.

Model will have N layers of convnets, where N depends on semantic complexity of problem domain. Each successive layer will squeeze the amount of information to passed onto next layer.

We will experiment with different values of N, and choose one that has best accuracy and is sustainable on the computing infrastructure available with us.

b. <u>Max Pooling</u>

Our model will will have multiple layers which will increase spatial complexity of computations. To reduce this complexity, we will use a squeezing operation called pooling. We will use strides of 2 to reduce image size at certain layer. But by using strides of 2, we could use loose important information in the pixels that we skipped over. So to include info from those pixels we will use max pooling technique. Max-pooling will take maximum value of surrounding convolutions to represent the resulting convolution.

c. <u>Fully Connected Layer</u>

High-level reasoning is done by fully connecting each neuron at the end of convolutions to each neuron in fully-connected layer. This layer combined with classifier is responsible for combining spatial information from underlying convnets and classify them into different labels.

d. Dropout

Amount of input data we have is not large in comparison to projected size of our model, which will result in sparseness of model. This sparseness could lead to overfitting during training. To reduce overfitting we use regularization techniques that add synthetic constraints to model during training. We will our train our model by removing random inputs from convnet layers to fully connected layer. That way our model will be robust enough to learn a redundate representation, instead of depending on certain specific inputs.

e. Local Response Normalization

Output from each convnet layer is input for next convnet layer. This output is used to train the weights and biases of next layer. If these outputs are have high variations it could lead to poor generalization, high numerical instability, and model becomes difficult to train. It could also lead to problem of dead neurons. To avoid this problem of dead neurons, output from each convnet layer needs to be normalized, which we achieve using local response normalization function available in TensorFlow API.

f. Rectified Linear Units

Weights from different convnet layers may not necessarily add in a linear fashion to create different shapes. So as to allow non-linear combination of weights, we will use a nonlinear function between 2 layers. Rectified Linear Unit (ReLU) is one such function that is very simple and easily differentiable for back propagation.

g. Stochastic Gradient Descent Optimization

We need an optimizer function to train the loss function of our model. We will use Gradient Descent as our optimizer, and turn it into stochastic one by using mini batches. This will reduce complexity of computation. We will experiment with different batch sizes based on available infrastructure and time taken by it for computations.

Benchmark

Proposed model will predict both length and individual digits of house numbers. We will strive for the benchmark set by Google's Model on same SVHN data, which is of 96% accuracy. But due to limited computing resource at our disposal, we expect our model to reach an accuracy level lower than this benchmark.

Methodology

Data preprocessing

SVHN data is available in 2 formats:

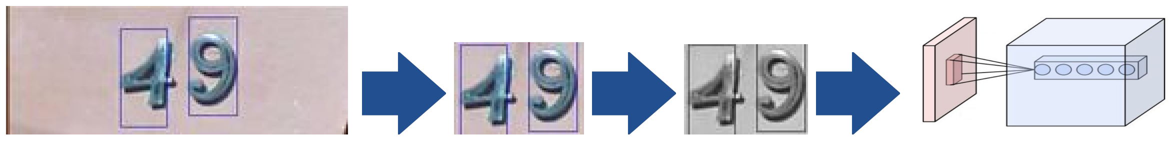
Format #1, is original variable-resolution image, along with with bounding box information.

Fotmat #2, is curated 32 X 32 pixels image, similar to MNIST data.

This implementation uses Format #1, to allow handling of more realistic image.

Pre-processing

SVHN Data in format #1 has 2 components: Images files, and Bounding Box information in matlab format. We use bounding box info in .mat file to localize whole sequence of number in an image, expand top, bottom, left, and right boundaries by 5%, and crop image to this dimension. Then we reshape it to 32 X 32 pixels, before converting to grayscale, and normalize it to zero mean, and unit standard deviation. This allows for better training.



Pre-processing Implementation

Pre-processing code is in SVHN_Data_Prep notebook, in get_labels_and_dataset() method.
After loading data from images, and corresponding bounding box data from .mat files, following steps are performed:
Code snippet to increase bounding box to be included in image: if(left==0 or tmp_left < left):

```
 if(left==0 or tmp_left < left):
            left=tmp_left*0.95;
if(top==0 or tmp_top < top):
            top=tmp_top*0.95
if(bottom ==0 or tmp_bottom > bottom):
            bottom=tmp_bottom*1.05
if(right==0 or tmp_right > right):
            right=tmp_right*1.05
```

Code that resized image, and converts to 32x32 and grayscale:

```
im_orig = im.crop((left, top, right, bottom)).resize([desired_image_size,desired_image_size]).convert('L')
```

After this Images are normalized for better training:

```
dataset_basic = (dataset_basic.astype(float) - pixel_depth / 2) / pixel_depth
```

After this Train and Test images are pickled for later processing:

```
train_pickle_file = 'SVHN_basic_train_labels.pickle'

 try:
    F  = open(train_pickle_file, 'wb')
    save = {
      'train_image_labels': train_image_labels
   }
    pickle.dump(save, f, 1)
    f.close()
except Exception as e:
    print('Unable to save train label data to', train_pickle_file, ':', e)
```

Implementation of Model

Platform

This project uses TensorFlow v0.9 deployed on following configuration:

Ubuntu 14 with Nvidia Drivers, Cuda 7.5 Toolkit, cuDNN 5.1, TensorFlow 0.9, scikit-learn, Python 2, and Jupyter.
BitFusion's AMI [REF. #7] was used for quick setup on Amazon g2.8x GPU instance.

Implementation Steps

Following are the steps followed in implementation:

1. Data loading from pickled files.
   Unpickle the Training and Test data from the pickled versions earlier.
   Sample of Training set
   Shape: *(33402, 32, 32)*
   Sample Labels:

```
[[ 2.  0.  0.  0.  1.  9.]
 [ 2.  0.  0.  0.  2.  3.]]
```

2. Reformat to required size, that is 32 X 32 X 1 format
   Data loaded in previous step is formatted into 32x32x1 format, for single feature map:
   Sample output:

```
train_dataset.shape: (33402, 32, 32, 1) , train_labels.shape: (33402, 6)
test_dataset.shape: (1000, 32, 32, 1) , test_labels.shape: (1000, 6)
```

3. Calculate digit masks
   This was among the important and complicated pieces to be understood and implemented, along with implementation of accuracy function.
   Each image can have a maximum of 5 digits. But most images have few digits missing.

For example
If a batch has 3 numbers, say 123, 45, and 6789.

So for each image in input batch, we calculate masks, digit wise, for missing digits.
So mark for above batch if input images,
Masks would be as follows:
Digit 1: 000 (None of the above 3 images has 1st digit)
Digit 2: 001 (Third image has 2nd digit from left)
Digit 3: 101 (FIrst and Third images have a digit at 3rd place)
Digit 4: 111
Digit 5: 111

This mask is calculated in explode() function call, digit wise, for , and is used to mark back prop from missing digits when calculating loss.
Here is code snippet for masking back prop for missing 1st-place digits in the tensor batch being processed.

```
loss_digit_1 = tf.nn.sparse_softmax_cross_entropy_with_logits(logit_digit_1, tf_train_labels[:,1])
loss_digit_1 = loss_digit_1 * tf_digit_masks_1
```

4. Define Model and train it.
We use batch size of 64 and convolutional kernel of size 5.
Model uses 10 convnet layers for training.
Depths of these 10 convnet layers are as specified in code snippet below:

```
depth1 = 16
depth2 = 24
depth3 = 32
depth4 = 48
depth5 = 64
depth6 = 128
depth7 = 128
depth8 = 256
depth9 = 512
depth10 = 1024
```

Model also uses 2 Fully Connected layers, having depths of 1024 and 512 respectively.

```
num_hidden1 = 1024
num_hidden2 = 512
```

To reduce overfitting, the Dropout layer uses 0.5 as pass through rate:

```
dropout = 0.50
```

Since there can be 10 different types of digits, number of digit labels is 10.

```
num_labels = 10 # 10 for 0-9
```

But length can take values ranging from 0 to 5, number of labels for length is 6 .

```
len_labels=6 #6 for lenghts0-5, not yet for more than 5
```

Model uses max-pooling layer defined here:

```
def max_pool_2x2(x):
        return tf.nn.max_pool(x, ksize=[1, 2, 2, 1],strides=[1, 2, 2, 1], padding='SAME')
```

Max-pooling layer is used 5 times. It is not used on first input layer to pass maximum info to second conv layer.
And it has not been added after each conv layer to keep input size larger than kernel size.

To connect output from Convolutions to fully-connected layer, it is resized here:

shape = h_pool10.get_shape().as_list()
*conv_result_reshape = tf.reshape(h_pool10, [shape[0], shape[1] * shape[2] * shape[3]])*

Then Fully Connected layers are defined here:
hidden = tf.nn.relu(tf.matmul(conv_result_reshape, fc_weights_1) + fc_biases_1)
hidden = tf.nn.relu(tf.matmul(hidden, fc_weights_2) + fc_biases_2)

Dropout is added now, and logits are calculated.
Loss is then calculated, considering the masks to be applied for missing digits.
This loss is returned from call to model() function.

5. Optimization Function
We use Gradient Descent to optimize loss function and use learning rate decay to avoid numerical fluctuations, and maintain momentum. Learning rate decay penalizes higher weights and biased after model has learned considerably over period of time.

6. Length and Digit predictions
Length labels have a shape of 6 while digit predictions have a shape of 10. Hence length values after after softmax is padded with 4 ZEROS before packing to be returned.

7. Accuracy
Argmax is applied on length and digit prediction tensors returned after each iteration, to get the actual digits.
Predicted length values used to set missing digits as zero, to allow for correct comparision of predicted values to actual values in pre-processed data.
Now accuracy is calculated as % of images for which both length and digits were predicted correctly, as a fraction of number of input images in each batch.

Here is the code snippet from accuracy() method:
*100.0 * (np.sum([np.all(row) for row in complete_accuracy])) / len(labels)*

Problems faced and refinement done

1. Saved by RELUs: During early phase of model creation and training, I observed a pattern in predictions. All digits being predicted at an individual place, say all digits at place 3, were same. After much analysis, I was able to conclude this behaviour was emanating from inability of model to differentiate between different shapes. This is when RELU layers were added, to allow learning of non-linear shapes. After addition of RELUs, model was able to predict different digits at same place.
Before using ReLUs: accuracy was 0%. Output labels, for a batch of 5 images, looked like following:

```
[[ 2.  0.  0.  0.  1.  1.]
 [ 2.  0.  0.  0.  1. 1.]
[ 3.  3.  1.  3.  3.  3.]
 [ 4.  4.  1.  4.  4. 4.]
[ 3.  5  5.  1.  5.  1.]]
```

There was no differentiation in different digits shapes. Results were completely weird.

After adding ReLUs, output looked something like this:

```
[[ 2.  0.  0.  0.  1.  9.]
 [ 2.  2.  0.  0.  2. 3.]
[ 3.  0.  0.  2.  1.  10.]
 [ 4.  0.  3.  1.  6. 9.]
[ 3.  0.  0.  4.  7.  7.]]
```

Although these were not accurate, but was model able to learn different shapes for different digits.
B**ut still training accuracy was stuck at 69% and test accuracy at a miserable 39.2%, even after 100,000 iterations**.

2. Skip missing digits: Goodfellow's paper mentions about skipping feedback from missing digits. Earlier attempt to implement this failed miserably, when I tried to mask output from FC layers while calculating logits. That attempt resulted in model unable to learn at all. After multiple readings of the paper and further analysis of code, I figured out that instead of logits, the loss being calculated in softmax layer had to be masked for missing digits. After this correction, model was able to improve further.

3. Use predicted length: Model being created included prediction of all 5 digits and length in calculating accuracy. Even after having 8 Convolutional layers and 2 fully connected layers, training accuracy was stuck at 82%, and test accuracy was stuck at 69%. Problem was that model was not using predicted length to consider only available digits for comparison. Instead the accuracy function was comparing prediction for all 5 digits, existing and missing, against the actual digits. This would not work since it would be hard to learn if a space at the beginning of house number image is for 1 missing digit or 2 missing digits; moreover these since length being used was predicted from model, so using that length to trim only those many digits is absolutely correct thing to do.

Once loss from missing digits was masked, and predicted length was used to compare predictions, training accuracy moved beyond 88%, and test accuracy reached 75%.

Refinement proposed

1. Train and learn more from examples that are have higher loss rate and lower accuracy during training..
2. Shuffle training examples after each epoch.
3. Add component to calculate bounding box.
4. Implement on multiple GPUs simultaneously.
5. Implement transfer of learning across multiple GPUs.

Results

Model Evaluation and Validation

Infrastructure Details

Above described model was run on a AWS instance using GPUs. The new **g2.8xlarge**[REF # 5] instance has the following specifications:

- Four NVIDIA GRID GPUs, each with 1,536 CUDA cores and 4 GB of video memory and the ability to encode either four real-time HD video streams at 1080p or eight real-time HD video streams at 720P.
- 32 vCPUs.
- 60 GiB of memory.
- 240 GB (2 x 120) of SSD storage.

Pre-processing results

Image and bounding box data was loaded into python arrays, and images are scaled around bounding box of all digits combines, and not around individual digits. This means that model will have to learn to locate digits in image and also identify them from their shapes.

We then normalized it for better training. Following is metadata about the samples to be used to train and test CNN model:

Training Data:	Test Data:
image_labels: (33402, 6) image_labels_array: (33402, 6) Mean: -0.0566795406976 Standard deviation: 0.19967425856	image_labels: (13068, 6) image_labels_array: (13068, 6) Mean: -0.0510183666569 Standard deviation: 0.225976380396

Results from Model Iterations

Model 1: 3 Convs, 0 FC, No ReLU	Accuracy: 0%
Model 2: 5 Convs, 2 FCs, No ReLU	Accuracy: 0%
Model 3: 5 Convs, 2FC, each with ReLU	Accuracy: 39.2%
Model 4: 10 Convs, 2FC, each with ReLU	Accuracy: 69.2%
Final Model: 10 Convs, 2FC, each with ReLU, No backprop from missing digits, and Accuracy only includes digits	Accuracy: 78.9%

upto predicted length

Final Execution Results:

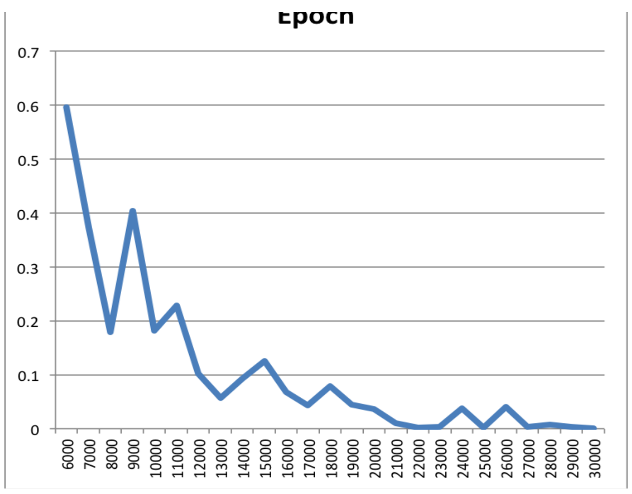
Following results are for Final Model, described above, after 30,000 iterations for batch size of 64:

Here is the result of model execution:

```
Minibatch loss at step 25000: 0.001621
Minibatch accuracy: 100.0%
Minibatch loss at step 26000: 0.040764
Minibatch accuracy: 98.4%
Minibatch loss at step 27000: 0.003704
Minibatch accuracy: 100.0%
Minibatch loss at step 28000: 0.007553
Minibatch accuracy: 100.0%
Minibatch loss at step 29000: 0.003003
Minibatch accuracy: 100.0%
Minibatch loss at step 30000: 0.000352
Minibatch accuracy: 100.0%
Test accuracy: 78.9%
```

Multiple iterations of final model over images that were slightly rotated or had digits on edges, gave similar accuracy of 78% +/- 0.5%. This proves robustness of model on different inputs.

Here is a plot of reducing loss as training proceeds:



Increasing iterations further to 100K and more conv layers, and tuning dropout and initial learning rate, could improve accuracy and bring it closed to benchmark value of 96%.

Conclusions

This project involved analyzing images of house numbers and extraction of exact house number from these images. There were several complexities associated including theoretical, tooling, and infrastructure challenges.

First the publicly available image data has to be structured in a form that could be operated upon by Python library, and in a efficient manner. Then this data had to be normalized for better results during Model training and Evaluation. Current implementation combines both these tasks in a single loop to reduce execution time. Only Train data set has been used, and Extra data set has been pickled but not used for modelling. Using Extra data, and generating more data by rotating images could improve accuracy further.

Creating an appropriate model posed several challenges. Although the paper suggests using 5 CNN layers, I experimented with variable number of layers to see the results and let the model be derived from requirements, instead of from theory. Starting with 2 CNN layers resulted in learning of missing digits, but the model predicted only similar numbers, like all 1s even when a digit is actually 7.

Real benefit came from adding RELU activations after CNN layers, which resulted in model being able to identify different shapes at different places in same house number. Adding Dropout helped reduce overfitting.

Current model uses 30,000 iterations, each having a batch size of 64. Kernel size was 5 for all layers.

There are 10 CNN layers and 2 FC layers. Here are their depths:

depth1 = 16
depth2 = 24
depth3 = 32
depth4 = 48
depth5 = 64
depth6 = 128
depth7 = 128
depth8 = 256
depth9 = 512
depth10 = 1024
num_hidden1 = 1024
num_hidden2 = 512

Executing such a large graph took 6 hours on AWS g2.8x large instance.

I intend to improve this further with additional synthetic data from same seed image data, by adding rotation and randomly trimmed images. Also plan to add plots of tensorflow execution summaries, for a better view into runtime.

References

1. Format 1 of Street View House Number (SVHN) from Stanford data: http://ufldl.stanford.edu/housenumbers/
2. Google's paper - Multi-digit Number Recognition from Street View Imagery using Deep Convolutional Neural Networks; http://static.googleusercontent.com/media/research.google.com/en//pubs/archive/42241.pdf
3. MNIST Data of Handwritted Digits: http://yann.lecun.com/exdb/mnist/
4. Dropout to prevent overfitting: http://www.jmlr.org/papers/volume15/srivastava14a.old/source/srivastava14a.pdf
5. AWS GPU INstances: https://aws.amazon.com/blogs/aws/new-g2-instance-type-with-4x-more-gpu-power/
6. Alternative to the area under the ROC curve: https://link.springer.com/article/10.1007/s10994-009-5119-5
7. BitFusion TensorFlow AMI: https://aws.amazon.com/marketplace/pp/B01EYKBEQ0