Machine Learning Engineer Nanodegree

Capstone Project

Derrick Chang
February 27th, 2017

I. Definition

Project Overview

Human brain is pretty good at gathering high-level information from images. However, these tasks are often challenging for machines. Computer vision is an interdisciplinary field seeks to automate tasks that the human visual system can do. One of these interesting tasks is to recognize digits in natural scenes.

The Street View House Numbers (SVHN) Dataset [1] is a collection of images contain house numbers in natural scenes created by Google. For Google, being able to read house numbers from images means improving map quality since every house number picture is geotagged. In this project, I will use deep learning approach based on Goodfellow [2] to recognize house numbers in the SVHN Dataset.

The reason I chose this project topic is because I want to learn more about deep learning, and image recognition is one of the fields which deep learning performs very well. My goal is to build an end-to-end system for house numbers recognition, so that I can have a concrete hand-on experience for deep learning.

Problem Statement

This problem is a supervised classification task. Given an image which contains a house number, the model should output a sequence of digits as a prediction. This problem is similar but more difficult than classifying digits for the MNIST dataset [3], because of the following two reasons:

1. The SVHN dataset contains images from natural scene, where perspective, lighting conditions, and other objects can cause distraction.

2. For a house number prediction to be considered as correct, all the digits in the house number should match the target label.

Generally speaking, house numbers range from 1 to 99999, and most of the house numbers in the SVHN dataset have 2-4 digits. Consider the size of our dataset and the distribution of house numbers' lengths, it would be impractical (way too many) if we define 99999 classes for this problem.

In this project, I will be assuming that house numbers are 1-5 digits long and defining 11 classes for each digit. Class [0-9] represents digit value [0-9], and class 10 represents N/A. The final model will take an image which contains a house number with 1-5 digits long, and output a sequence of 1-5 digits as a prediction. I will be building this model using deep convolutional neural networks based on Goodfellow's approach [2].

For this capstone project, the deliverables that I will be producing are:

1. Source Code
2. Project Report (this document)

The outline of tasks is as follow:

1. Collect SVHN data and parse metadata (bounding boxes and labels)
2. Preliminary analysis of data
3. Preprocess data and split the data into training, validation and test set
4. Build the CNN model
5. Fine tune the CNN model
6. Test and evaluation

Metrics

For a house number prediction to be considered as useful, it needs to be completely correct. No partial credits should be given if any of the digits in a house number is wrong. For example, a house number '3589' may be a few streets away from '9589' even though there is only a single digit mismatch.

As a result, a good metric for this project is defined as:

$$accuracy = \frac{\text{total instances of correctly predicted house number}}{\text{total house number}} * 100\%$$

Notice that the house number here means the entire house number sequence instead of a single digit.

For example, if we have 3 image samples with their labels and predictions shown as follows:

	Image#1	**Image#2**	**Image#3**
Labels	1234	1998	151
Predicts	1234	1999	51
Correctness	v	x	x

The accuracy is: 1/3 x 100%= ~33.33%

Here is the implementation of the accuracy function:

```python
def eval_accuracy(predictions, labels):
    return tf.reduce_mean( tf.reduce_min(tf.to_float(tf.equal(tf.to_int32(predictions), labels)), axis = 1))
```

The function takes two tensors as the input and output a tensor as the accuracy.
predictions: An Nx5 float tensor, contains prediction classes of digits in each sample
labels: An Nx5 integer tensor, contains prediction classes of digits in each sample
output: An 1x1 (scaler) float tensor, contains accuracy

II. Analysis

Data Exploration

The SVHN Data can be downloaded from: http://ufldl.stanford.edu/housenumbers/
The dataset comes in two formats:

Format 1: Original Images (*.png) with character level bounding boxes (Figure 1)

• Train Data:	33401
• Test Data:	13068
• Extra Data:	202353

For each set of images, there is a 'digitStruct.m' file stores metadata for every single digit in an image.

- Bounding boxes information: [Top, Left, Width, High]
- Digit class: [1-10], one for each digit. (Digit '1' has label 1, '9' has label 9, and '0' has label 10).

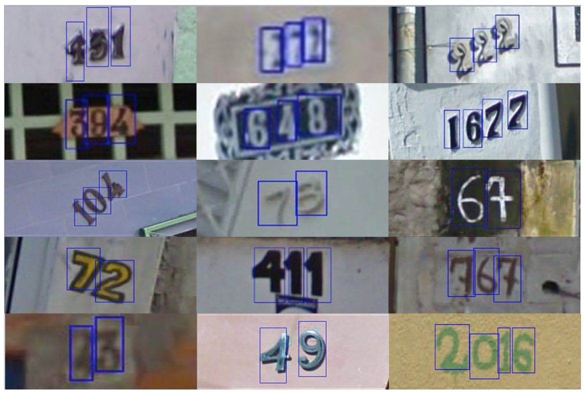


Figure 1: Examples of images from the SVHN dataset (format 1)

Format 2: MNIST-like 32x32 images centered around a single character (Figure 2)

• Train Data:	32x32x3x73257	Train Label:	73257x1
• Test Data:	32x32x3x26032	Test Label:	26032x1
• Extra Data:	32x32x3x531131	Extra Label:	531131x1

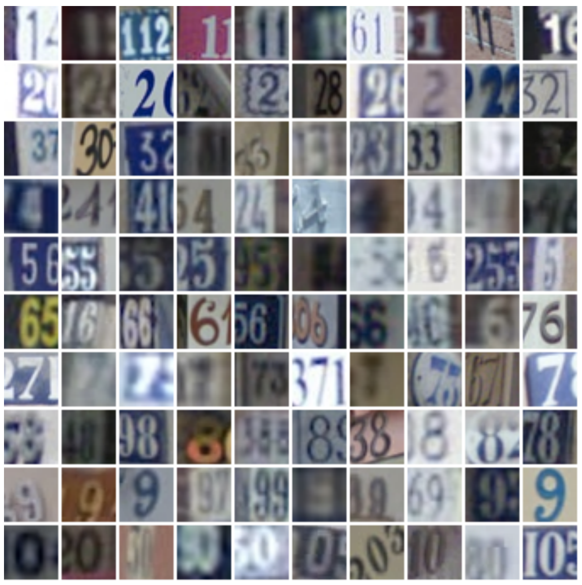


Figure 2: Examples of images from the SVHN dataset (format 2)

In this project, only the Training Data and Testing Data from the original images (Format 1) will be used.

Exploratory Visualization

Length of labels

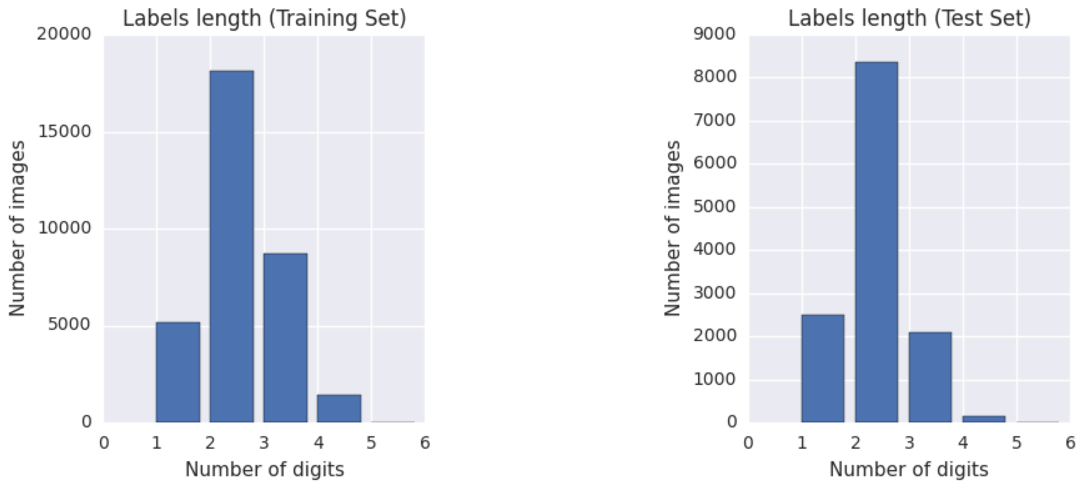


Figure 3: Distributions of house number length for the training set and test set

Label length in Train: {1: 5137, 2: 18130, 3: 8691, 4: 1434, 5: 9, 6: 1}
Label length in Test: {1: 2483, 2: 8356, 3: 2081, 4: 146, 5: 2}

Figure 3 indicates that most of house numbers have length in the range of 1-5 digits. Only 1 labels have 6 digits. I considered that data sample as an outliner and thus removed it from the dataset.

Digit Height

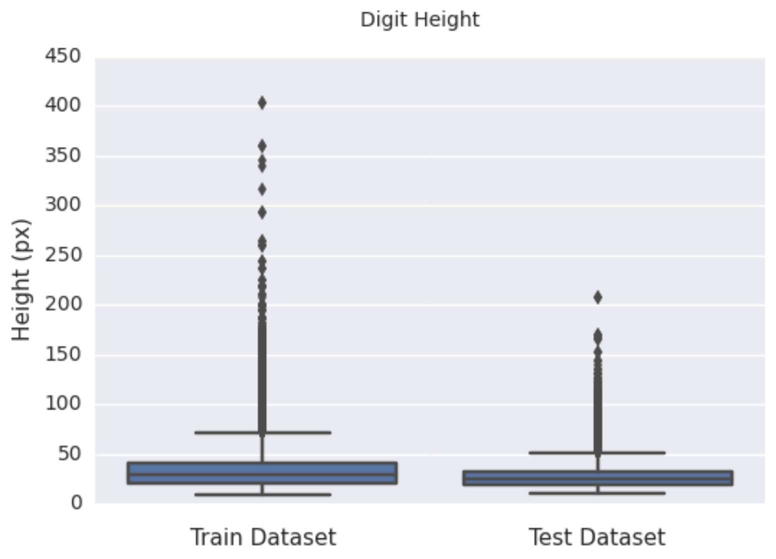


Figure 4: Box plots of digit height for the training set and testing set

Heights in Train: (mean, median, std) = (33.86, 29.00, 18.60)
Heights in Test: (mean, median, std) = (27.96, 24.00, 13.46)

Digit Width

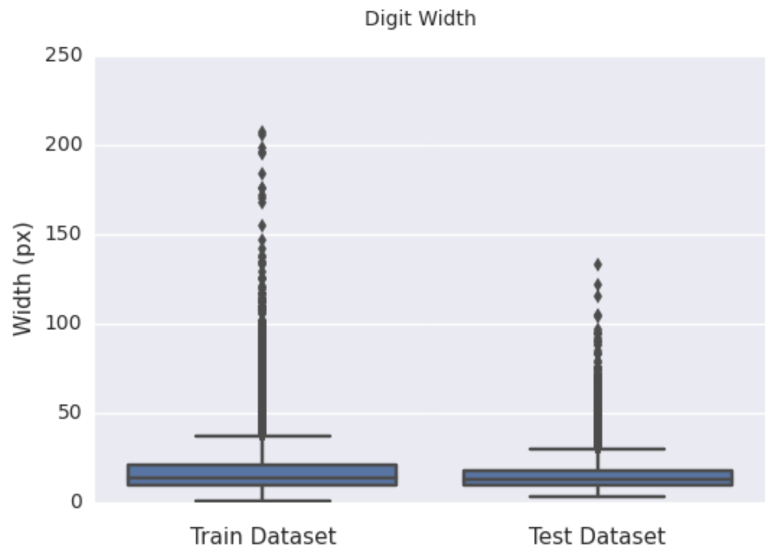


Figure 5: Box plots of digit width for the training set and testing set

Widths in Train: (mean, median, std) = (16.65, 14.00, 10.68)
Widths in Test: (mean, median, std) = (15.52, 13.00, 8.24)

Figure 4 and 5 show that the size of digits varies in a wide range, thus resizing should be performed during the preprocessing step.

Algorithms and Techniques

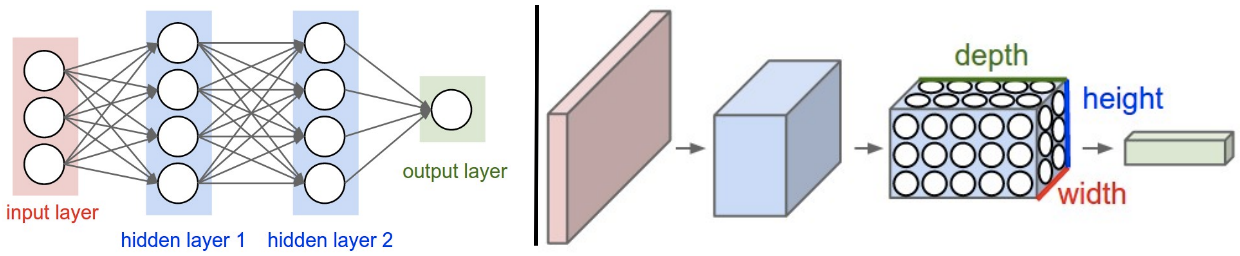
The model I used in this project is Convolutional Neural Networks (ConvNets).

The ConvNets are very similar to regular Neural Networks, they are made up of neurons that have learnable weights and biases. Each neuron receives some inputs, performs a dot product and optionally follows it with a non-linearity unit. The whole network will output a single differentiable score function and there is a loss function which can be used for model optimization.

The main different between ConvNets and regular Neural Networks is that: the ConvNet architectures make the assumption that the inputs are images, which makes the forward function more efficient to implement and vastly reduce the number of parameters in the network. As a result, ConvNet is chosen for this project.

The most common form of a ConvNet has the following pattern:

INPUT -> [[CONV -> RELU]*N -> POOL?]*M -> [FC -> RELU]*K -> FC(OUTPUT)
where the '*' indicates repetition, and the 'POOL?' indicates an optional pooling layer. Moreover, N >= 0 (and usually N <= 3), M >= 0, K >= 0 (and usually K < 3).



Left: A regular 3-layer Neural Network. Right: A ConvNet arranges its neurons in three dimensions (width, height, depth), as visualized in one of the layers. Every layer of a ConvNet transforms the 3D input volume to a 3D output volume of neuron activations. In this example, the red input layer holds the image, so its width and height would be the dimensions of the image, and the depth would be 3 (Red, Green, Blue channels). (Figures from Stanford CS231n [4])

INPUT: The input layer holds the raw pixel values of the image

CONV: The convolutional layer applies a set of filter on the input volume. Each filter scans through the entire input volume and computes dot products between their filter weights and small regions they are connected to in the input volume. The output is a stack of 2-dimensional activation maps for that image. The 2-dimension activation maps correspond to some type of feature at some spatial position.

RELU: The rectified linear unit (relu) layer applies an elementwise activation function: max(0,x) which thresholding at zero. It makes training faster and more efficient for deep neural architectures on large and complex datasets

POOL: Pooling is a regularization technique that performs non-linear down sampling. Its function is to progressively reduce the spatial size of the representation to reduce the amount of parameters and computation in the network, and hence to also control overfitting.

FC: Each neuron in the fully connected layer is connect to every neuron from the previous layer

OUTPUT: The output layer is a fully connected layer which output the class scores

Some other types of layers:

Local Response Normalization: The local response normalization layer performs a kind of "lateral inhibition" by normalizing over local input regions.

Batch Normalization: A technique to provide any layer in a Neural Network with inputs that are zero mean/unit variance.

Dropout: An extremely effective, simple and recently introduced regularization technique that complements the other methods (L1, L2, maxnorm). While training, dropout is implemented by only keeping a neuron active with some probability, or setting it to zero otherwise.

Benchmark

The benchmark that I will be using is the performance presented by Goodfellow et al. They reported an accuracy of 96% for multi-digit classification on the SVHN dataset. Also, human have 98% accuracy on this dataset.

III. Methodology

Data Preprocessing

Data size before preprocessing
Train data: 33402 images
Test data: 13068 images

Step 1: Extract digit labels and bounding boxes from metadata. Pickle these metadata into 'SVHN_metadata.pickle' for later use.

Step 2: Crop house numbers (both the training and testing set) and resize to 32x32
For each image, find a union of every digit's bounding box. Use the union bounding box to crop the image and resize the image to a 32 x 32 integer array. Also, in this step, house numbers longer than 5 digits will be removed.
Resulting dataset:

Train data: (33401, 32, 32, 3)	Train labels: (33401, 5)
Test data: (13068, 32, 32, 3)	Test labels: (13068, 5)

Step 3: Shuffle data
Shuffle data for the training and the testing data set

Step 4: Create a validation set
Split the training dataset: 80% as the training dataset and 20% as the validation dataset
Resulting dataset:

Train data: (26721, 32, 32, 3)	Train labels: (26721, 5)
Test data: (13068, 32, 32, 3)	Test labels: (13068, 5)
Validation data: (6680, 32, 32, 3)	Validation labels: (6680, 5)

Step 5: Transform images from RGB to grayscale
Since the color information is irrelevant for digit classification, it should be removed for simplicity. I choose the luminosity method which forms a weighted sum of the *R*, *G*, and *B* components to account for human perception (Figure 6).
Grayscale intensity = 0.2989 * R + 0.5870 * G + 0.1140 * B
Resulting dataset:

Train data: (26721, 32, 32)	Train labels: (26721, 5)
Test data: (13068, 32, 32)	Test labels: (13068, 5)
Validation data: (6680, 32, 32)	Validation labels: (6680, 5)

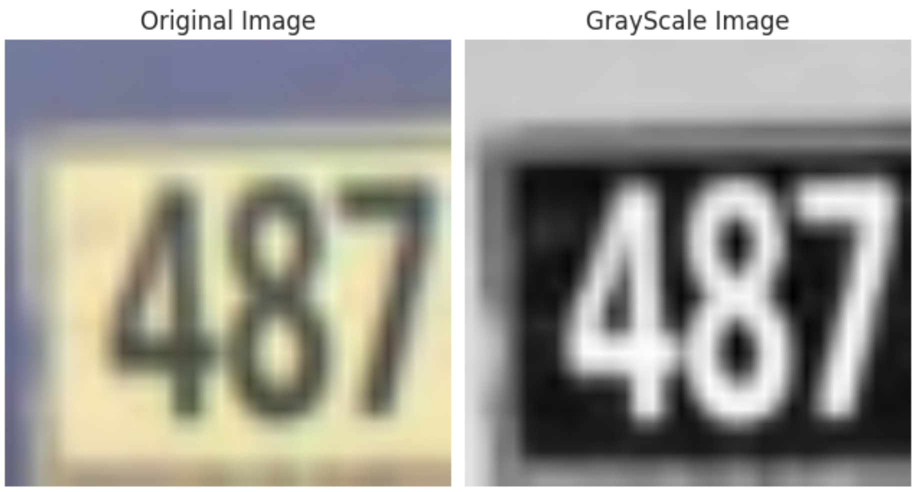


Figure 6: Grayscale transformation

Step 6: Normalize images
Normalize the image data so that each image has a zero mean and equal variance. This step is essential because proper normalization can improve the effectiveness of gradient search when performing optimizations.

Step 7: Reshape for CNN
Reshape the input data because the convolution layer take a volume as the input.
Resulting dataset:

Train data: (26721, 32, 32, 1)	Train labels: (26721, 5)
Test data: (13068, 32, 32, 1)	Test labels: (13068, 5)
Validation data: (6680, 32, 32, 1)	Validation labels: (6680, 5)

Step 8: Pickle the dataset into 'SVHN_data.pickle' for later use

Implementation

Version of Python: Python 2.7

Libraries: tensorflow, numpy, scipy, matplotlib, cPickle

Metadata:	SVHN_metadata.pickle
Data:	SVHN_data.pickle
Preprocessing:	1_SVHN_Preprocessing.ipynb
CNN Model Tuning:	2_SVHN_CNN_tuning.ipynb
CNN Final Model:	3_SVHN_CNN_final.ipynb
Predictions:	4_SVHN_Predict.ipynb

My initial implementation **ConvNet#1** has the following structure:

INPUT -> [[CONV -> RELU]*1 -> POOL]*2 -> DROPOUT -> FC/OUTPUT

Layer	**Resulting size**	**Parameters**
INPUT	[64x32x32x1]	batch size=64
CONV1+RELU	[64x32x32x16]	Patch size: 5x5, Stride:1, Padding: same, #of filters: 16
POOL1	[64x16x16x16]	Patch size: 2x2, Stride: 2, Padding: same, Pooling: max
CONV2+RELU	[64x16x16x32]	Patch size: 5x5, Stride:1, Padding: same, #of filters: 32
POOL2	[64x8x8x32]	Patch size: 2x2, Stride: 2, Padding: same, Pooling: max
DROPOUT	[64x8x8x32]	keep_probablity: 0.9
FC/OUTPUT	[64x11]	FC size: 2048 x 11 There are 5 of this layer, one for each digit.

ConvNet#1 is implemented as the following model function. The function outputs a tensor which contains 5 logits for each digit.

```
def model(data, keep_prob=1, isTraining = False):
    #CONV
    h_conv1 = tf.nn.conv2d(data,W_conv1, [1,1,1,1],padding='SAME', name='conv_layer1') + b_conv1
    h_conv1 = tf.nn.relu(h_conv1)
    h_conv1 = tf.nn.max_pool(h_conv1, [1,2,2,1], [1,2,2,1], 'SAME')

    h_conv2 = tf.nn.conv2d(h_conv1, W_conv2, [1,1,1,1], padding='SAME', name='conv_layer2') + b_conv2
    h_conv2 = tf.nn.relu(h_conv2)
    h_conv2 = tf.nn.max_pool(h_conv2, [1,2,2,1], [1,2,2,1], 'SAME')

    h_conv2 = tf.nn.dropout(h_conv2, keep_prob)

    #Reshape
    shape = h_conv2.get_shape().as_list()
    h_conv2 = tf.reshape(h_conv2, [shape[0], shape[1] * shape[2] * shape[3]])

    #OUTPUT
    logits1 = tf.matmul(h_conv2, W_o1) + b_o1
    logits2 = tf.matmul(h_conv2, W_o2) + b_o2
    logits3 = tf.matmul(h_conv2, W_o3) + b_o3
    logits4 = tf.matmul(h_conv2, W_o4) + b_o4
    logits5 = tf.matmul(h_conv2, W_o5) + b_o5
    return tf.pack([logits1, logits2, logits3, logits4, logits5])
```

For each digit, I calculated the cross entropy with the true label. And then I summed them up as the loss function.

```
loss = tf.reduce_mean(tf.nn.sparse_softmax_cross_entropy_with_logits(logits[0], tf_train_labels[:,0])) +\
       tf.reduce_mean(tf.nn.sparse_softmax_cross_entropy_with_logits(logits[1], tf_train_labels[:,1])) +\
       tf.reduce_mean(tf.nn.sparse_softmax_cross_entropy_with_logits(logits[2], tf_train_labels[:,2])) +\
       tf.reduce_mean(tf.nn.sparse_softmax_cross_entropy_with_logits(logits[3], tf_train_labels[:,3])) +\
       tf.reduce_mean(tf.nn.sparse_softmax_cross_entropy_with_logits(logits[4], tf_train_labels[:,4]))
```

The AdamOptimizer with default setting was chosen.

```
train_step = tf.train.AdamOptimizer().minimize(loss)
```

Finally, predictions and accuracies were evaluated.

```
def eval_accuracy(predictions, labels):
    return tf.reduce_mean( tf.reduce_min(tf.to_float(tf.equal(tf.to_int32(predictions), labels)), axis = 1))

# Predictions for the minibatch training, validation, and test data.
logits = model(tf_train_dataset, kp, tf_is_training)
train_prediction = tf.transpose(tf.argmax(logits, axis = 2))
valid_prediction =  tf.transpose(tf.argmax(model(tf_valid_dataset), axis = 2))
test_prediction =  tf.transpose(tf.argmax(model(tf_test_dataset), axis = 2))

train_accuracy = eval_accuracy(train_prediction, tf_train_labels)
valid_accuracy = eval_accuracy(valid_prediction, tf_valid_labels)
test_accuracy = eval_accuracy(test_prediction, tf_test_labels)
```

After 15000 iterations, ConvNet#1 reported the following results:

ConvNet #1	
Minibatch loss at step 15000	2.470681
Minibatch accuracy	50.0%
Validation accuracy	45.1%

At this point, I've set up my initial model and gotten a preliminary result. Although the resulting accuracy was still pretty coarse, everything else worked smoothly including preprocessing steps, data flow, and metrics for evaluation. Hence, I concluded that my implementation did fit the purpose of this project and there's no complications occurred that requires major changes.

Refinement

Since the performance of my initial implementation ConvNet#1 did not yield good results, I implemented ConvNet#2 by adding two additional CONV layers and one more FC layer before OUTPUT.

ConvNet#2 has the following structure:
INPUT -> [[CONV -> RELU]*1 -> POOL]*4 -> DROPOUT -> [FC -> RELU]*1 -> DROPOUT -> FC/OUTPUT

Layer	**Resulting size**	**parameter**
INPUT	[64x32x32x1]	batch size=64
CONV1+RELU	[64x32x32x16]	Patch size: 5x5, Stride:1, Padding: same, #of filters: 16
POOL1	[64x16x16x16]	Patch size: 2x2, Stride: 2, Padding: same, Pooling: max
CONV2+RELU	[64x16x16x32]	Patch size: 5x5, Stride:1, Padding: same, #of filters: 32
POOL2	[64x8x8x32]	Patch size: 2x2, Stride: 2, Padding: same, Pooling: max
CONV3+RELU	[64x8x8x64]	Patch size: 5x5, Stride:1, Padding: same, #of filters: 64
POOL3	[64x4x4x64]	Patch size: 2x2, Stride: 2, Padding: same, Pooling: max
CONV4+RELU	[64x4x4x96]	Patch size: 5x5, Stride:1, Padding: same, #of filters: 96
POOL4	[64x2x2x96]	Patch size: 2x2, Stride: 2, Padding: same, Pooling: max
DROPOUT	[64x2x2x96]	keep_probablity: 0.9
FC	[64x128]	FC size: 384x128
DROPOUT	[64x128]	keep_probablity: 0.9
FC/OUTPUT	[64x11]	FC size: 128x11 There are 5 of this layer, one for each digit.

ConvNet #2	
Minibatch loss at step 15000	1.533096
Minibatch accuracy	62.5%
Validation accuracy	51.8%

By adding more CONV and FC layers, the performance was improved.

ConvNet2+Batch Normalization: [CONV -> BN -> RELU]

BN	Yes	No
Minibatch loss at step 15000	0.312321	1.533096
Minibatch accuracy	87.5%	62.5%
Validation accuracy	47.3%	51.8%

I added a batch normalization layer between each CONV and RELU. The minibatch accuracy was improved while the validation accuracy was worse.

ConvNet2+Local Response Normalization (LRN): [CONV -> RELU -> LRN]

LRN	Yes	No
Minibatch loss at step 15000	0.43391	1.533096
Minibatch accuracy	80.2%	62.5%
Validation accuracy	65.1%	51.8%

I added a LRN layer for every CONV after the RELU layer. Both the minibatch accuracy and the validation accuracy were improved.

ConvNet2+LRN + L2 regularization

```
regularizers = tf.nn.l2_loss(W_conv1) + tf.nn.l2_loss(W_conv2) + \
               tf.nn.l2_loss(W_conv3) + tf.nn.l2_loss(W_conv4) + \
               tf.nn.l2_loss(W_fc1) + tf.nn.l2_loss(W_o1) + \
               tf.nn.l2_loss(W_o2) + tf.nn.l2_loss(W_o3) + \
               tf.nn.l2_loss(W_o4) + tf.nn.l2_loss(W_o5)
loss_regularized = loss + beta * regularizers
```

beta	0	0.001	0.01	0.1
Minibatch loss at step 15000	0.43391	2.615757	4.353477	5.87437
Minibatch accuracy	80.2%	87.5%	87.5%	3.1%
Validation accuracy	65.1%	65.6%	66.4%	2.0%

I added an L2 regularization term to the loss function. For small beta, L2 regularization did not bring much improvement to the accuracy, while larger beta made the performance much worse.

IV. Results

Model Evaluation and Validation

Based on the model refinement process, my final model has the following configuration:

CNN architecture: ConvNet#2
Weight initialization: truncated normal
Batch Normalization: No
Local Response Normalization: Yes
Dropout: Yes, with keep probability = 0.9
Optimizer: AdamOptimizer, with default parameters
L2 Regularization: No
Performance Evaluation: Whole sequence accuracy

Figure 7, 8 and 9 show the visualization of my final model via TensorBoard.

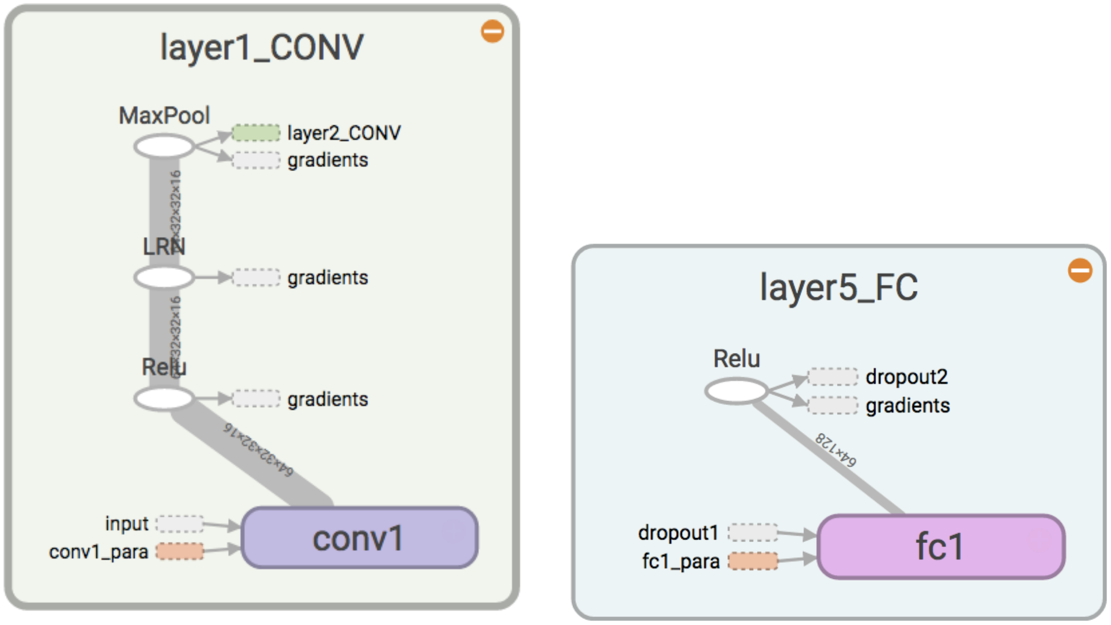


Figure 7: Zoom-in view of a CONV layer and a fully-connected layer

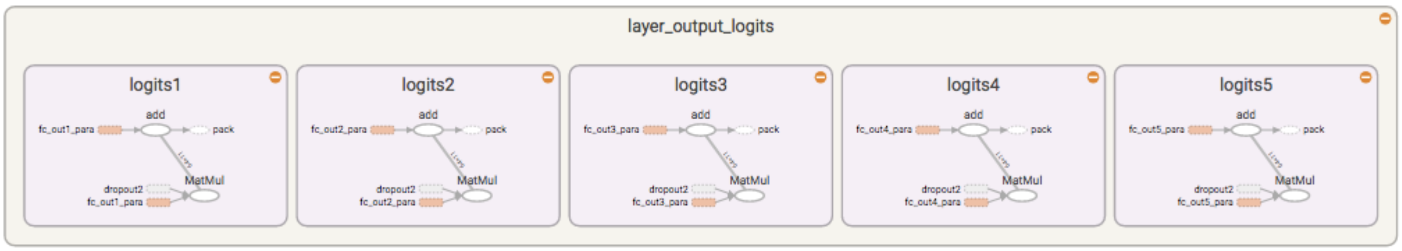


Figure 8: Zoom-in view of the output layer

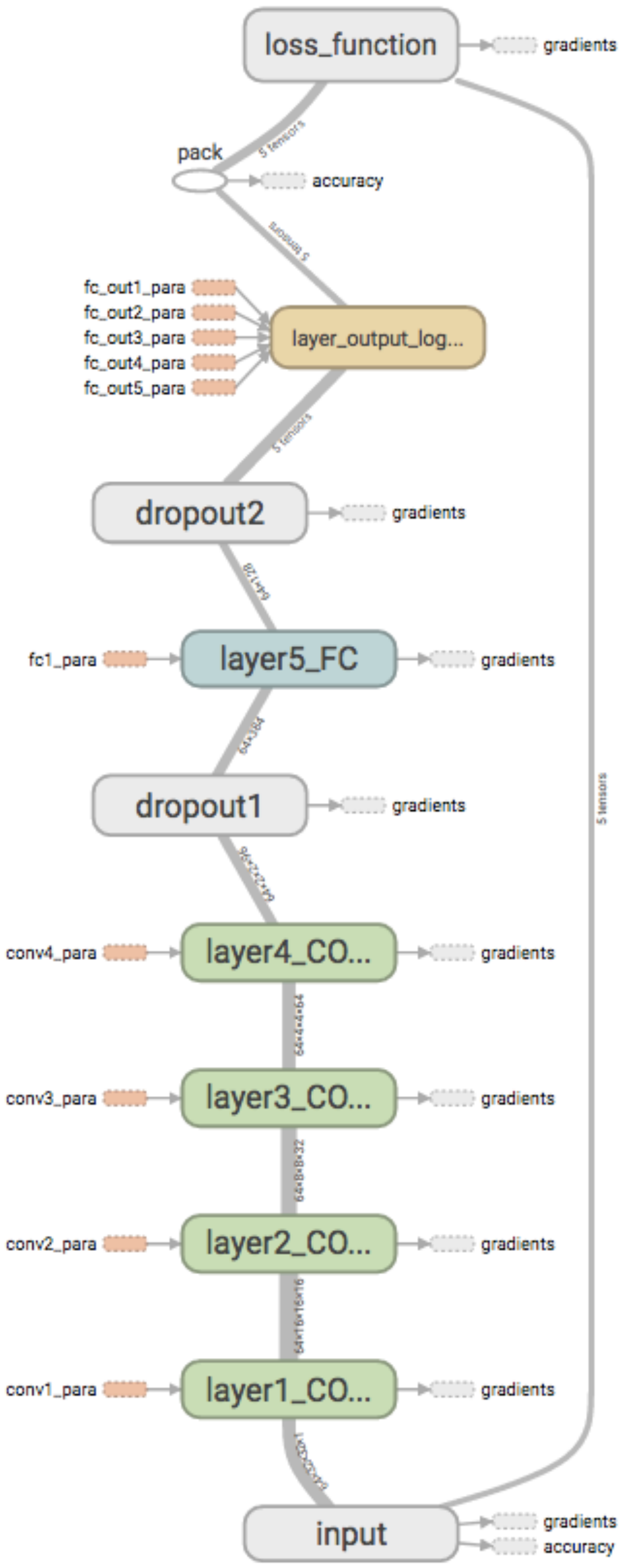


Figure 9: Visualization of the final model

Final result:
Training was stopped after 20k iterations, which represents approximately 48 epochs.

Final Model	
Minibatch loss at step 20000	0.296226
Minibatch accuracy	87.5%
Validation accuracy	66.2%
Test accuracy	64.2%

Figure 10, 11 and 12 show the progress of each iterations. I notice that there is a large gap between the minibatch accuracy and validation accuracy. This indicates there may be some problem of overfitting.

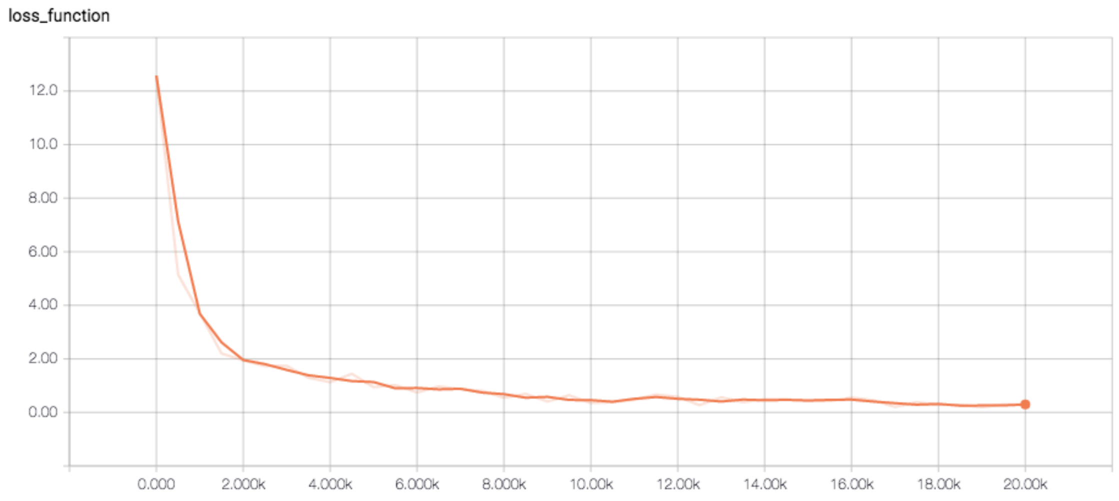


Figure 10: Loss function for the final model (20k iterations)

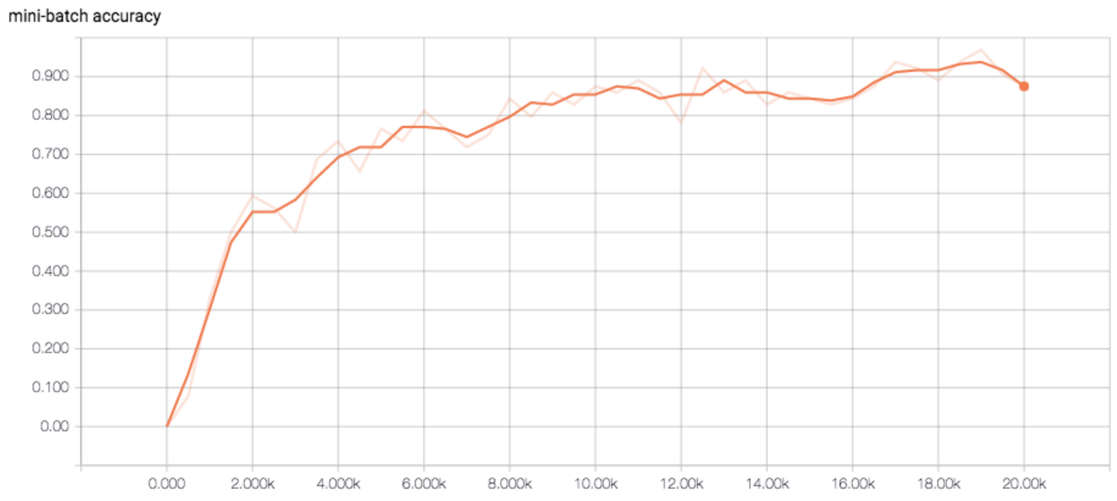


Figure 11: Minibatch accuracy for the final model (20k iterations)

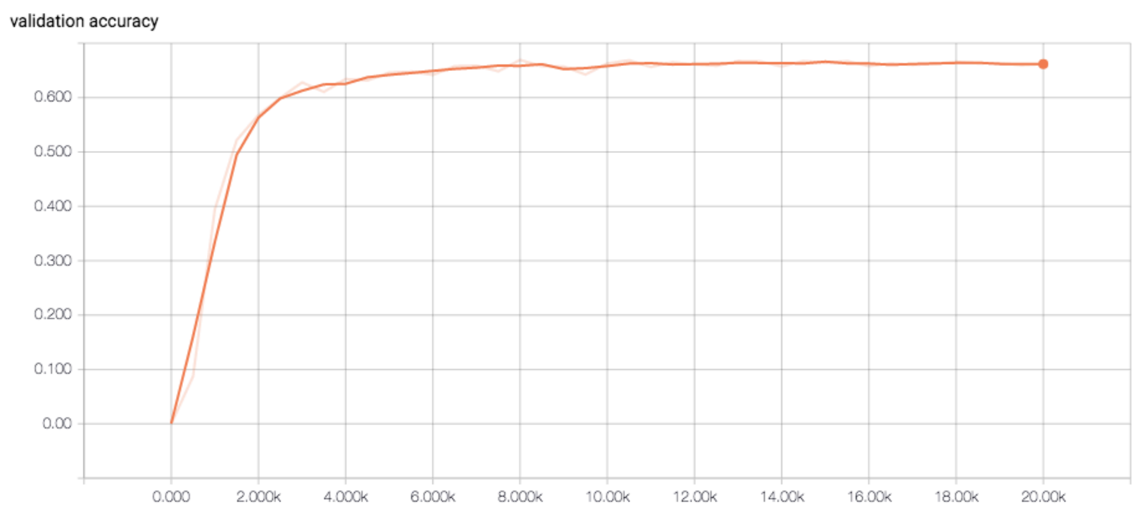


Figure 12: Validation accuracy for the final model (20k iterations)

Justification

My final model reports test set performance of 64.2%, which is not as good as the 96% benchmark by Goodfellow et al. [2], however their model took 6 days to train on a powerful machine. Given that I was using more limited resources, I believe the result I got is acceptable but still has room for improvement.

V. Conclusion

Free-Form Visualization

I randomly chose 10 samples from the test dataset, made predictions with my final model and compared the results with true labels. Figure 13 shows that 7 out of 10 image were correctly predicted, which roughly corresponds with the reported accuracy at 64.2%. It is worthy to note that, in most of the cases when wrong predictions were made, there is no more than one wrong digit. For example, 31=>91, 120=>126 and 282=>262. However, for our application, the prediction is considered as fail even though there is only a single mis-classified digit. I also observed that the model tends to make mistakes that human may make. For example, the digit '3' in the first example looks like 9 to human.

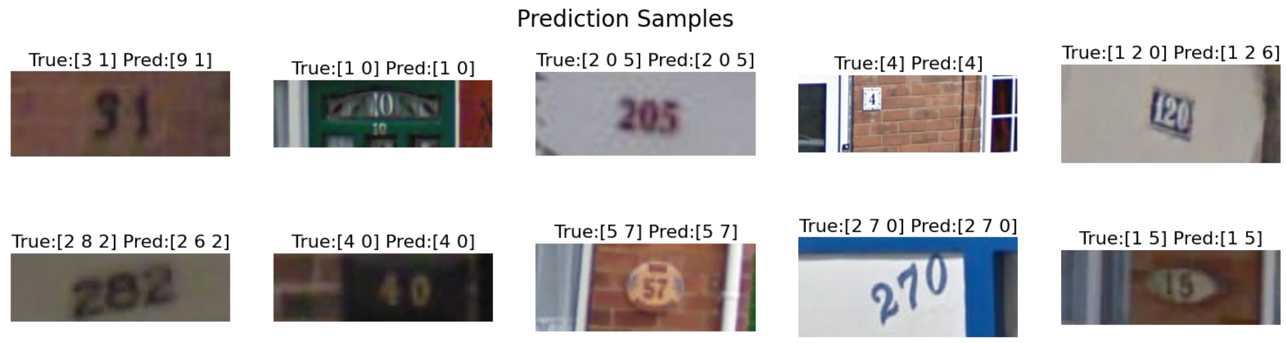


Figure 13: Random samples of prediction results vs. label

Reflection

In this project, I implemented an end-to-end system for recognizing house numbers in natural scenes using ConvNets. I started from gathering the SVHN dataset and performing preliminary data analysis. Based on what I observed during the data exploration, I pre-processed the data and organized them into the right format for building models. And I designed my model based on the general pattern of ConvNets. I went through multiple rounds of refinement and came out with my final model. Finally, I evaluated the performance of my model and provided explanations.

This project helps me learn a lot about deep learning, convolutional neural networks, and Tensorflow. In the beginning of this project, I realized that my knowledge about deep learning (which I learned from Udacity) was insufficient to start this project. As a result, I went through the course notes of Stanford's CS231n[4] to get the required background knowledge for this project.

The most interesting but also difficult part is designing the architecture of ConvNets and refining the model. I spent a significant amount of effort in trying different configurations. Since I was working on a device with limited computation power, every trial took a huge amount of time to process.

A couple of tricky issues arouse when I was manipulating tensors in Tensorflow. I ran into many bugs because I didn't set the dimension right. For example, the accuracy function took me a bit of time to get it right.

I also learned how to use TensorBoard, which is an amazing tool that allows me to visualize models and monitor variables.

Overall, although the accuracy of my final model did not meet my expectations, I enjoyed working on this project and I believe I've learned a lot.

Improvement

Some potential improvements could be made in the following areas:

1. As mentioned in the result section, the gap between the minibatch accuracy and the validation accuracy may indicate overfitting. Thus, better approach for generalization should be explored. One possible solution is to artificially expanding the training data, such as augmenting the training dataset with rotation and scaling.

2. In my implementation, I didn't use the length of house numbers as a label to train the model. Adding the length information may better exploit the training data so that the model can learn better.

3. Figure 11 shows that when the performance plateau, the minibatch accuracy does not reach near 100%. This indicates that more complexity should be added to the model. Possible solution is to modify the CNN architecture and add more layers. An alternative solution of developing architectures from scratch is to use some popular CNN architectures such as GoogLeNet, VGGNet or ResNet.

4. In this project, limited computation resource is a big challenge for me. When I designed the CNN architecture, I need to compromised with the model complexity. It would be worthwhile to consider cloud computing services such as Google Cloud ML, Amazon ML or Microsoft Azure.

Reference

1.The Street View House Numbers (SVHN) Dataset: http://ufldl.stanford.edu/housenumbers/
2. Ian J. Goodfellow, Yaroslav Bulatov, Julian Ibarz, Sacha Arnoud, Vinay Shet (2014). Multi-digit Number Recognition from Street View Imagery using Deep Convolutional Neural Networks. https://arxiv.org/pdf/1312.6082.pdf
3. MNIST dataset: http://yann.lecun.com/exdb/mnist/
4. Course notes from Stanford CS class CS231n: Convolutional Neural Networks for Visual Recognition: http://cs231n.github.io/