Image Classification in the Street View House Numbers Dataset using TensorFlow

Group 7 Final Project Individual Report

INTRODUCTION / OBJECTIVE

The aim of this study is to perform a deep learning analysis on image data, measure, compare results and describe the findings. I am looking to train a model that would Classify as accurately as possible each of the digit images available.

In this project, I am using TensorFlow Framework that I encountered in class to classify the Street View House Numbers (SVHN) dataset using Convolution Neural Network (CNN) architecture. I chose this dataset because it gave me the opportunity to utilize Convolutional Layers and other topics that we covered in class. The SVHN is considerably more complicated than the MNIST dataset, as it contains confounding objects in the images, the digits appear in different angles in the images, and the dataset is in color.

DATA SET

Google Street View House Number(SVHN) Dataset

Souce: http://ufldl.stanford.edu/housenumbers/

SVHN is a real-world image dataset for developing machine learning and object recognition algorithms with minimal requirement on data preprocessing and formatting. It is similar in flavor to MNIST (e.g., the images are of small cropped digits), but incorporates an order of magnitude more labeled data (over 600,000-digit images) and comes from a significantly harder, unsolved, real world problem (recognizing digits and numbers in natural scene images). SVHN is obtained from house numbers in Google Street View images.

- Total 10 Classes, 1 for each digits i.e Label '9' for digit 9 and '10' for digit 0.
- 73,257 digits for training, 26,032 digits for testing
- Available in two different formats:
  Format 1: Original images with bounding box available for each character (may contain multiple characters in same images).
  Format 2: MNIST like 32x32 cropped images having single character in each image.

I worked on Format 2 dataset in this project using CNN Architecture on TensorFlow framework.

```
('Training Set', (73257, 32, 32, 3))
('Test Set', (26032, 32, 32, 3))

('Total Number of Images', 99289)
```

WHY TENSORFLOW

TensorFlow is an open source software library for numerical computation using dataflow graphs. Nodes in the graph represents mathematical operations, while graph edges represent multi-dimensional data arrays (aka tensors) communicated between them. The flexible architecture allows you to deploy computation to one or more CPUs or GPUs in a desktop, server, or mobile device with a single API.

TensorFlow as nothing but numpy with a twist. A major difference between numpy and TensorFlow is that TensorFlow follows a lazy programming paradigm. It first builds a graph of all the operation to be done, and then when a "session" is called, it "runs" the graph. It's built to be scalable, by changing internal data representation to tensors (aka multi-dimensional arrays). Building a computational graph can be considered as the main ingredient of TensorFlow.

The advantages of using TensorFlow are:

- It has an intuitive construct, because as the name suggests it has "flow of tensors". You can easily visualize each part of the graph.
- Easily train on CPU/GPU for distributed computing
- Platform flexibility. You can run the models wherever you want, whether it is on mobile, server or PC.

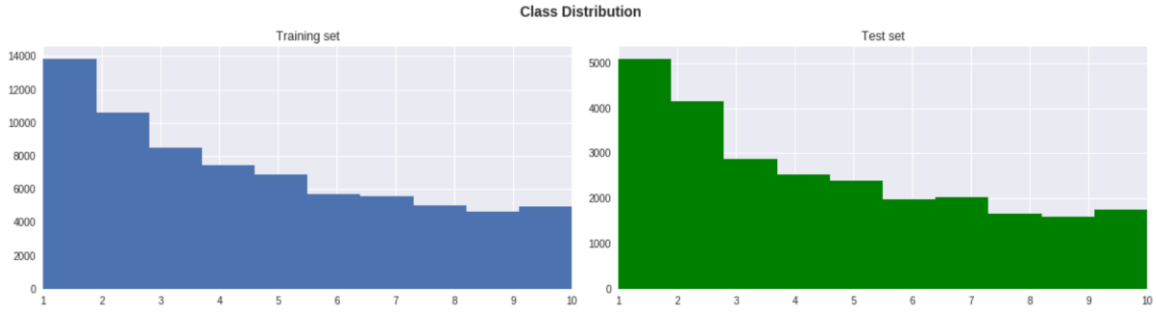
PREPROCESSING

Pre-processing the 32-by-32 images from the SVHN dataset centered around a single digit. In this dataset all digits have been resized to a fixed resolution of 32-by-32 pixels. The original character bounding boxes are extended in the appropriate dimension to become square windows, so that resizing them to 32-by-32 pixels does not introduce aspect ratio distortions.

1. Created a Balanced (Startified) 13% of data in Validation Set.
   Splitting to 13% in Val Set as it gives around 9500 data having minimum of 800 instances of each class.

```
('Training Set', (63733, 32, 32, 1))
('Validation Set', (9524, 32, 32, 1))
('Test Set', (26032, 32, 32, 1))
```

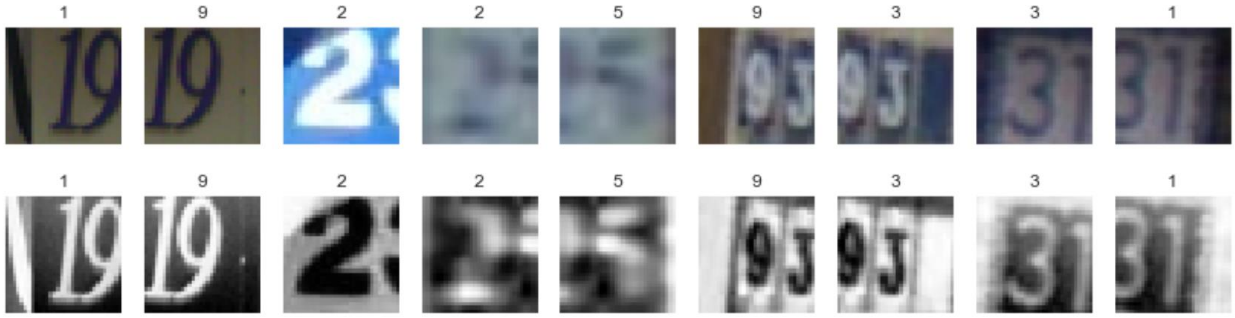
2. Initial Exploratory Analysis



All distributions have a positive skew, meaning that we have an underweight of higher digit values.

3. Converting the Label 10's to 0's
   So, we got a target label of [0 1 2 3 4 5 6 7 8 9]

4. RGB to Grayscale
   To speed up my experiments I convert the images from RGB to Grayscale, which reduced the amount of data to process. Since we are not concerned about the color of the image, just its shape (i.e. the number shown).
   Y = 0.2990R + 0.5870G + 0.1140B



Sample Images from my training set. RGB to Grayscale.

5. Normalization
   Normalization refers to normalizing the data dimensions so that they are of approximately the same scale. Divide each dimension by its standard deviation, once it has been zero-centered. Calculated the mean and standard deviation here.

6. One Hot Encoding
   Apply One Hot Encoding to make label suitable for CNN Classification.

```
('Training set', (63733, 10))
('Validation set', (9524, 10))
('Test set', (26032, 10))
```

7. Storing to Disk
   Used h5py package to store the numerical data, so that it can be easily manipulated using NumPy.

CODE WORKFLOW

The Work flow which I followed is broken down into 4 major phases

1. Making TensorFlow computational graphs (CNN Architecture)
2. Designing the TensorFlow Model
3. Execution of TensorFlow Model
4. Model Evaluation

Making TensorFlow computational graphs

I wrote functions for creating new TensorFlow Variables in the given shape and initializing them with random values. Followed by functions for stacking CONV-RELU layers and POOLING layers as well.

A convolutional layer produces an output tensor with 4 dimensions. We will add fully-connected layers after the convolution layers, so we need to reduce the 4-dim tensor to 2-dim which can be used as input to the fully-connected layer. So, I defined a flatten layer as well followed by the function to stack FC-RELU layer. Below are the functions that I created

```
def conv_weight_variable(layer_name, shape):
def fc_weight_variable(layer_name, shape):
def bias_variable(shape):
def conv_layer
def flatten_layer(layer):
def fc_layer
```

Designing the TensorFlow Model

First task was to define Placeholder Variables. They serve as the input to the graph that we may change each time we execute the graph. This allows us to change the images that are input to the TensorFlow graph.

```
x = tf.placeholder(tf.float32, shape=(None, img_size, img_size,
num_channels), name='x')
```

I implemented the following ConvNet architecture based on the layer patterns proposed in the CS231n notes:

```
INPUT -> [CONV -> RELU -> CONV -> RELU -> POOL] -> DROPOUT ->(Flatten) [FC ->
RELU] -> FC -> Softmax
```

Applied Optimization methods, computed cross-entropy as well

Execution of TensorFlow Model

Once the TensorFlow graph has been created, I created a TensorFlow session which is used to execute the graph.

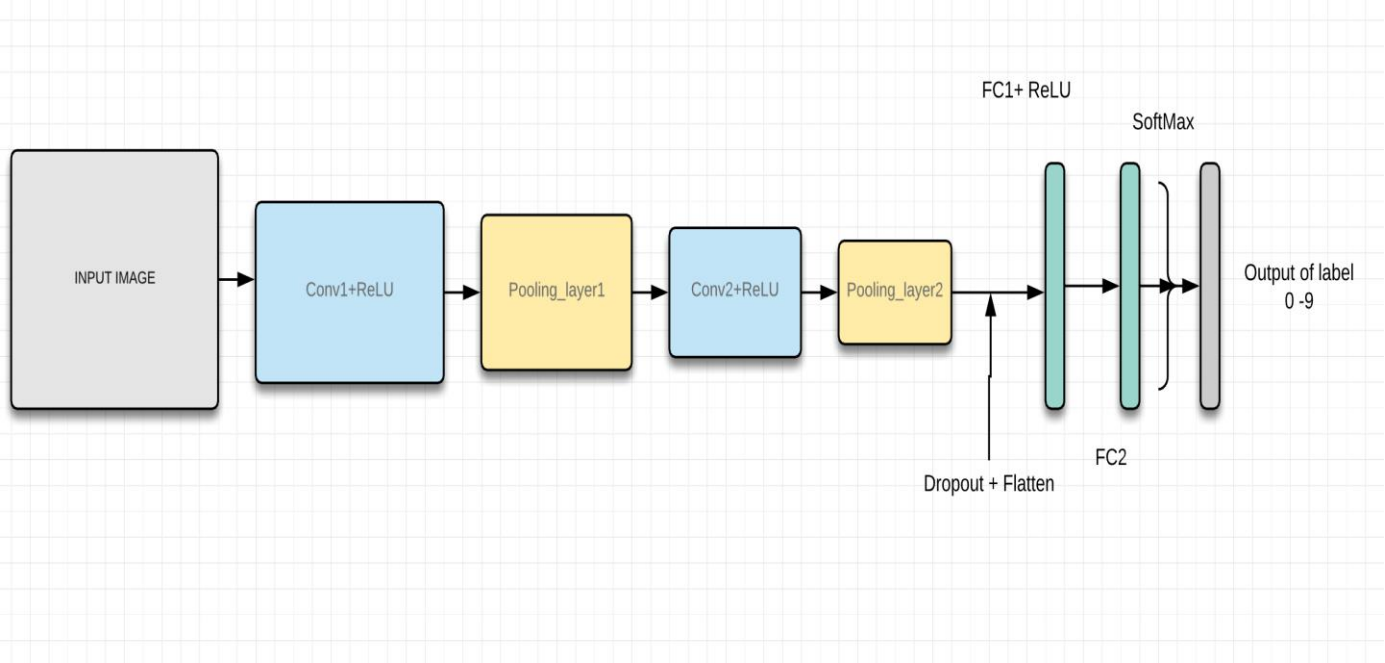
In order to save the variables of the neural network, I created a so-called Saver-object which is used for storing and retrieving all the variables of the TensorFlow graph. The saved files are often called checkpoints because they may be written at regular intervals during optimization. This is the directory used for saving and retrieving the data. After that, the execution starts on the training, validation set and testing set and we get the accuracy percentage for our model.

```
optimize(num_iterations=50000, display_step=1000)
```

Model Evaluation

Here I found and plotted some of the mis-classified examples in my testset and a confusion matrix showing how well our model is able to predict the different digits.

MODEL ARCHITECTURE



Convolutional Layer 1 & Pooling Layer 1

Create the first convolutional layer. It takes x as input and creates num_filters1 different filters, each having width and height equal to filter_size1.

Convolutional Layer 2 & Pooling Layer 2

Create the second convolutional layer, which takes as input the output from the first convolutional layer. The number of input channels corresponds to the number of filters in the first convolutional layer. Finally, we wish to down-sample the image so it is half the size by using 2x2 max-pooling.

Dropout Layer

To reduce overfitting, we will apply dropout after the pooling layer.

Flatten Layer

The convolutional layers output 4-dim tensors. I wanted to use these as input in a fully-connected network, which requires for the tensors to be reshaped or flattened to 2-dim tensors.

Fully-Connected Layer 1

Add a fully-connected layer to the network. The input is the flattened layer from the previous convolution. The number of neurons or nodes in the fully-connected layer is fc_size. ReLU is used so we can learn non-linear relations.

Fully-Connected Layer 2
Add another fully-connected layer that outputs vectors of length 10 for determining which of the 10 classes the input image belongs to. Note that ReLU is not used in this layer.

Softmax
The second fully-connected layer estimates how likely it is that the input image belongs to each of the 10 classes. However, these estimates are a bit rough and difficult to interpret because the numbers may be very small or large, so we want to normalize them so that each element is limited between zero and one and the 10 elements sum to one. This is calculated using the so-called softmax function and the result is stored in y_pred.

Calculate Cross-entropy
To make the model better at classifying the input images, we must somehow change the variables for all the network layers. To do this we first need to know how well the model currently performs by comparing the predicted output of the model y_pred to the desired output y_true.

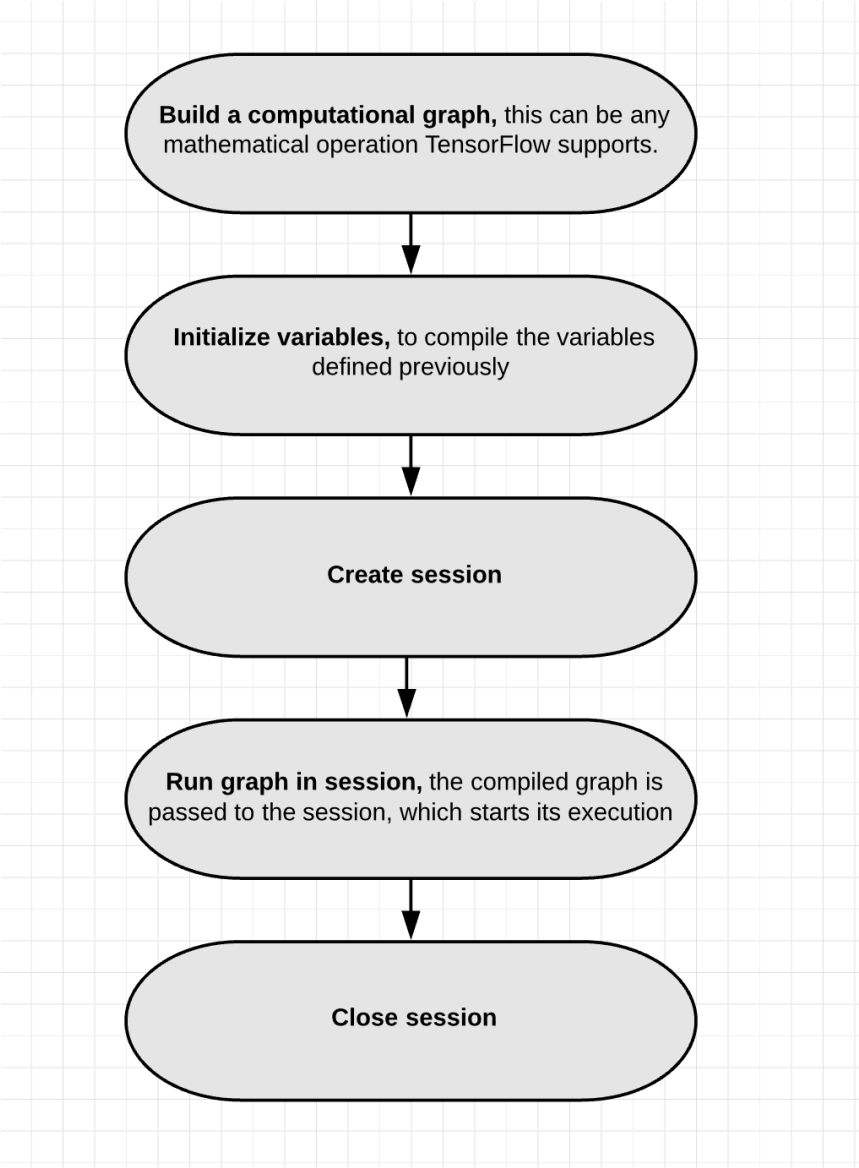
Optimization Method: Adam
Now that we have a cost measure that must be minimized, we can then create an optimizer. In training deep networks, it is usually helpful to anneal the learning rate over time. There are three common types of implementing the learning rate decay. Here I used exponential decay and Adagrad optimizer.

The best architecture that gave me the best results:

Layer	**Description**
Input Layer	We input a batch of 64 images (64, 32, 32 ,1)
Conv1+ReLU1	32 Filters, 5x5 Kernal size, Stride 1, Zero Padding
Pooling1	2x2, Stride 2
Conv2+ReLU2	64 Filters, 5x5 Kernal size, Stride 1, Zero Padding
Pooling2	2x2, Stride 2
Dropout	0.5
FC	256 nodes

FRAMEWORK WORKFLOW

The following is the workflow which I followed while creating the CNN model on TenserFlow framework

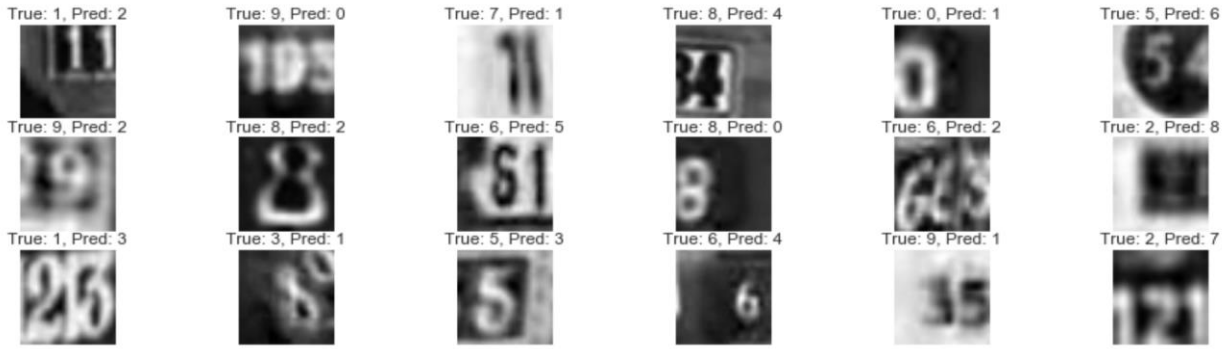


RESULTS & CONCLUSIONS

With the mentioned Architecture I got good Test accuracy of **92.96 %** at 50000 iterations which is good enough I think.

```
Minibatch accuracy at step 49000: 1.0000
Validation accuracy: 0.9022
Test accuracy: 0.9296
Time usage: 0:06:50
```

I had some of the misclassified images which I found so I dig in deeper to see how good the model works



So, I calculated the confusion matrix to show how well our model is able to predict the different class values. The Accuracy % of all digits seem high.

INTERNET CODE

I followed the cs231n notes Stanford CS class CS231n: Convolutional Neural Networks for Visual Recognition for the architecture to get the broad idea and suggestions. For TensorFlow Framework setup I referenced Stack overflow and the official website of TensorFlow

Percentage of code found/copied (by the defined formula): 30%

REFERENCES

[1] A.Karpathy, http://cs231n.github.io/convolutional-networks.

[2] Multi-digit Number Recognition from Street View Imagery using Deep Convolutional Neural Networks (https://arxiv.org/abs/1312.6082)

[3] https://www.tensorflow.org/

[3] Stack Overflow