CAPSTONE PROJECT

Machine Learning Engineer Nanodegree

I. Definition

Project Overview

In last decade we have seen significant progress in the field of computer vision. Much of this can be attributed to a resurgence of neural networks in machine learning. With the availability of faster processors especially in the form of GPU's, it is now possible to train programs over millions of images. These "learned" programs are then able to identify and predict new images with astonishing accuracy. Deep convolutional neural networks is one such form of neural network that has had lot of success in recent years. In this capstone project I intend to implement convolutional neural networks in the Tensorflow language, and use that to train the program to identify images of digits 0 to 9. The eventual goal is for the learned model to read snapshots of multiple digits like those of street house numbers.

Problem Statement

In this capstone project the objective is to make a deep learning program to identify images of sequence of digits correctly. For this we use the Street View House Numbers (SVHN) dataset, which is a real world collection of images obtained from Google Street View images. The challenge in learning from this dataset is that it the images are in real-world settings, as a result the numbers are displayed in various colors, patterns, fonts, sizes, lighting conditions, backgrounds and noise levels.

To describe the task at hand in more technical terms

1. Read the images from the SVHN single number dataset. These images are reformatted to arrays in the Tensorflow language.

2. Design a model based on convolutional neural network that gives a low loss and good accuracy in identifying single digits.

3. Adapt the single image to make a CNN model for sequence of multiple digits in an image. Here we learn how to classify a sequence without any additional information about length, position,

size etc. of digits.

4. Tune the model to get a good accuracy. Right from the beginning my goal is to develop a model that can identify multiple sequence of digits all together with around 90% accuracy. The benchmark here is the 96% accuracy achieved by Goodfellow et. al. This required substantial amount of computational time and hardware, both of which are beyond my reach.

Metrics

The most important metric to measure the performance of the model is the accuracy with which it identifies the sequence of digits in our sample. So we can simply define accuracy as

$$\boldsymbol{Acc_N} = \frac{\boldsymbol{Number\ of\ images\ correctly\ identified}}{\boldsymbol{Total\ number\ of\ image\ samples\ in\ the\ test\ set}}$$

Our test set will be the one on which we will never train our model. So this provides a good parameter to check the accuracy of the model by testing over a set which it has never seen before. While we will be training the model we will also keep a validation set and monitor its accuracy. Finally the trained model will be used to test the accuracy in identifying a test model. The accuracy of the test dataset will be our most important metric. In addition we will monitor the accuracy of our training dataset, and the loss/cross entropy while the training is happening.

We will also look into accuracy of each digit/label correctly guessed in addition to the previously defined accuracy. For instance if the image has the number 871 and our algorithm identified it as 879, it still got 2 out of 3 digits right. In real-world scenario this is not very helpful as these street numbers could be miles away from each other, but for algorithmic purposes it does show a more promising algorithm than if let’s say if it had identified 871 as 012.

$$\boldsymbol{Acc_D} = \frac{\boldsymbol{Number\ of\ labels/digits\ correctly\ identified}}{\boldsymbol{Total\ number\ of\ all\ the\ labels/digits}}$$

II. Analysis

Data Exploration

SVHN is a real-world image dataset for developing machine learning and object recognition algorithms with minimal requirement on data preprocessing and formatting. The dataset is obtained from house numbers in Google Street View images.

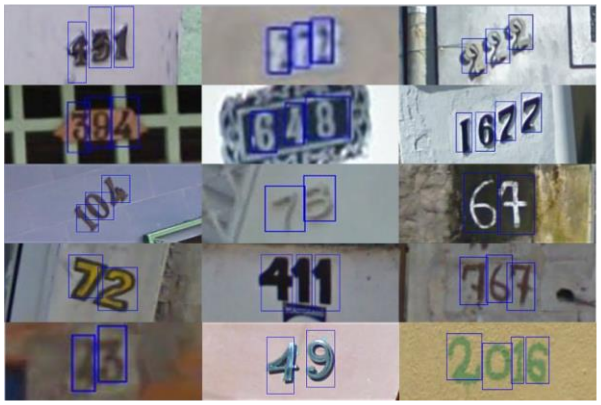


Figure 1: Sample images from the SVHN dataset. The dataset has original images with bounding boxes. Here each image can have multiple number of digits as is expected in a street number

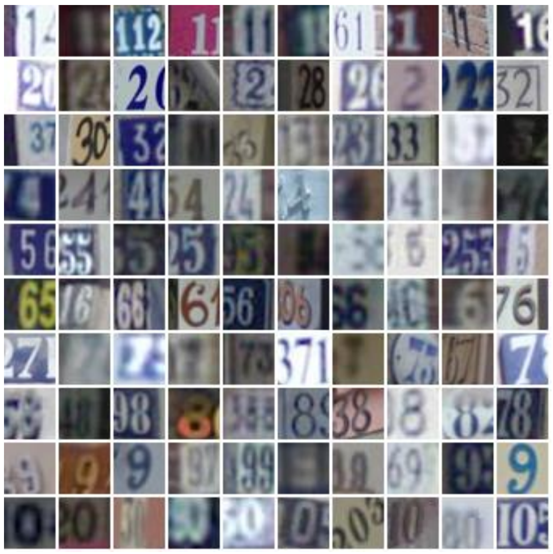


Figure 2: These are the cropped digits derived from the SVHN dataset, containing a single digit at center of a 32 by 32 pixel image.

Figure 1 and 2 show respectively the full images from street view and the cropped digits provided in a convenient format available at [1].

Full Numbers: These are the full images of numbers. Images come in different sizes. There is lot of variety in the dataset in terms of length of numbers, lighting, colors etc.

- train.tar.gz: Contains 33401 SVHN images along with bounding box details
- test.tar.gz : Contains 13068 SVHN images along with bounding box details
- extra.tar.gz: Contains 202353 SVHN images along with bounding box details

In unpacking the tar-gzipped file we get a folder full of images of various size and a mat file containing the bounding box and label information for each digit in each image.

Cropped Digits: For convenience the cropped digits of sizes 32 by 32 pixel are provided in three matlab .mat files and which we can load into memory using python's scipy.io package as numpy arrays.

- train_32x32.mat: Contains 73257 cropped digits and labels.
- test_32x32.mat: Contains 26032 cropped digits and labels.
- extra_32x32.mat: Contains 531131 cropped digits and labels.

In each of the label files each non-zero digits is labelled by the same number (*i.e.* '1' for 1, '2' for 2 and so on). The digit '0' is labelled as 10. The extra dataset is collection of somewhat easier samples. Here are some samples with labels of cropped digits.

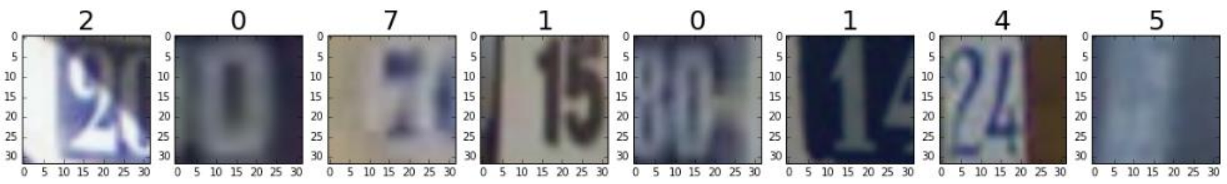


Figure 3: Cropped Image and labels

Exploratory Visualization

Cropped Digits: From the dataset for cropped digit images we can look for the distribution of labels in the dataset. In Figure 4 we can see that it is not a uniform distribution, and the digits 1 and 2 are more often visible in the dataset. In fact this sort of distribution is not uncommon in dataset found in real-world scenarios. This is often called the Benford's law *i.e.* frequency of digit 1 is highest and that of the subsequent digits is lesser sequentially.

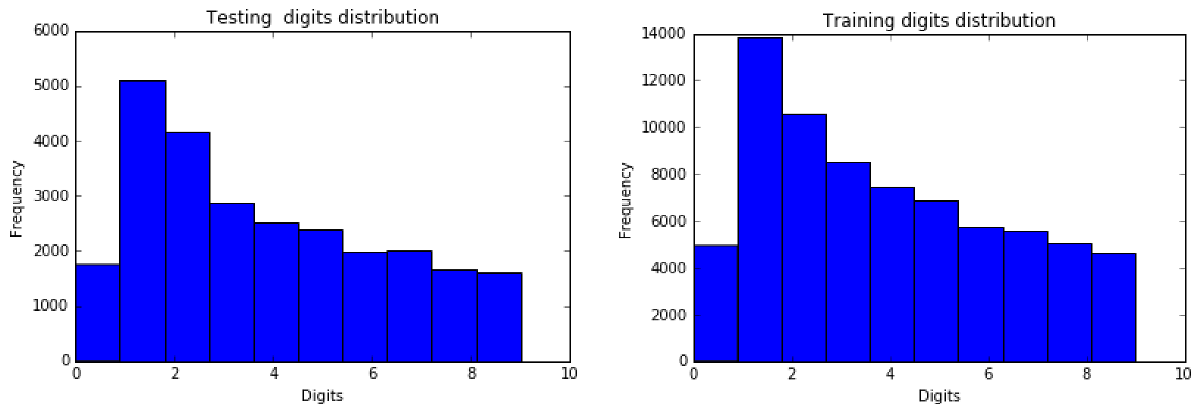


Figure 4: Distribution of digits in the cropped digits testing and training datasets are shown in the left and right respectively. The distribution is skewed towards higher representation of smaller digits.

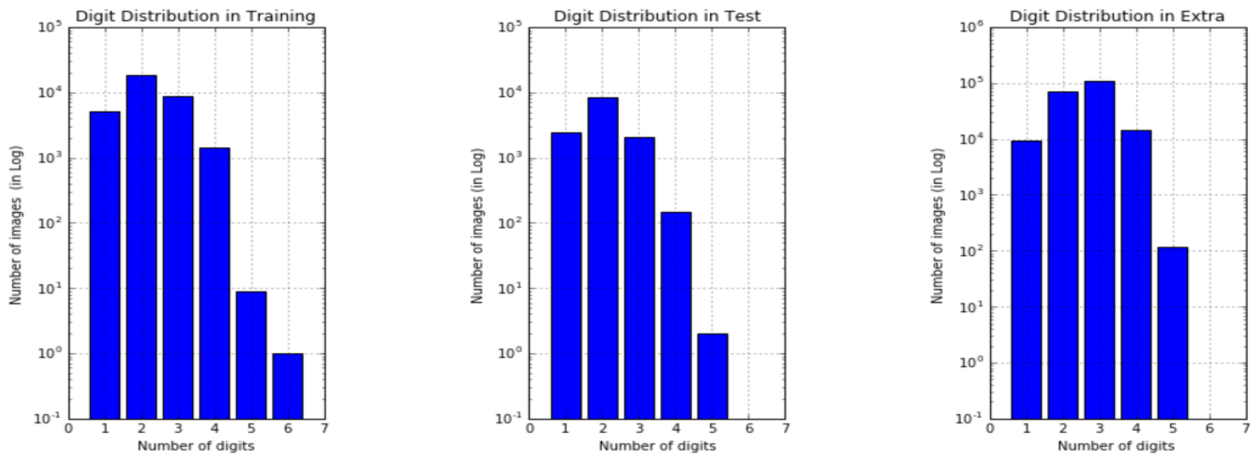


Figure 5: Distribution of the length of the entire number in SVHN dataset, divided into Test, Training and Extra groups. Please note that the plots are in semi-log scale so that the outlier could be shown clearly in the training dataset.

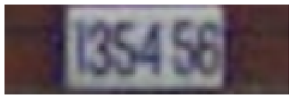


Figure 6: This is the outlier in the training dataset with 6 digits in house number (135456)

Full Images: In the full images dataset each image can have multiple digits in it. It is interesting to see the distribution of lengths of house numbers in dataset. In figure 5 we can see that in the entire dataset the largest house number has six digits (this was in training dataset). That outlier is shown in Figure 6.

Algorithms and Techniques

In recent times Convoluted Neural Network (CNN) is considered the best option for image recognition problems. For our problem we will develop a multi-layered deep CNN in the Tensorflow language and feed our dataset of images to it. A classical neural network has one hidden layer but a deep neural network has many layers. The nesting of many layers is what makes it 'deep' and helps to identify complex patterns in images during training. We would not be building any algorithm from scratch but rather use the various API's provided by Tensorflow to implement our CNN.

Let's look into the breakdown of the algorithm. We can broadly break into two steps, first is the feed forward phase having the following components. A pictorial summary is given in Figure 5, this is not to be taken as the standard structure of a CNN

- **Convolution Layer**: The input to the learning algorithm is the raw pixels of the image. These input images are fed to the convolution layer. As the name suggests, in this layer we find the convolution of the image with a smaller filter width of weights and depth. This set of filters usually scans through the entire image with a stride and sums up the result. One can imagine this as an input image of certain volume is transformed into another volume of new sizes.

- **Max Pooling**: Is a technique used to reduce size (only height and width) of the intermediate volumes created after a previous convolution layer. For example a max pooling of an image of 32 by 32 pixels done with a stride of 2 steps will reduce it to a 16 by 16 pixels image. This is simple done by replacing a 2 by 2 square of pixels with the maximum value in that square. The obvious advantage is that memory consumption is immediately reduced.

- **Rectified Linear Units (ReLU):** These serve as the activation function in our deep neural network and introduce the non-linearity in the neural network. ReLU basically adds a multiplication factor h to the previous layer values a as defined by the equation$h = \max(0, a)$. ReLU has one major advantage over other non-linear functions like sigmoid, tanh, etc., in that it reduces the likelihood of gradients (of weights and biases) to vanish. In fact due to this same reason, ReLU also helps in learning faster over other activation functions. Non-linearity introduced by the activation function plays a key role in learning. If we had several linear layers of weights and biases then they would sum to just a single equivalent linear layer. These several layers of non-linearity in essence helps in fine tuning our decision boundaries beyond the simple line of a linear approximation.

- **Normalization**: Because ReLU neurons have unbounded activations it was suggested and demonstrated in [4] that a normalization after ReLU stage will improve accuracy. A Local Response Normalization (LRN) does precisely that.

- **Dropout**: Dropout helps in preventing overfitting which as we know is quite important for any learning algorithm.

- **Full Connected Layer:** The output of the neurons/activations of the previous stages finally need to be reduced to the number of labels we are classifying into. This is done by a single or a series of several fully connected layers which have connections to each of the previous neuron.

- **Softmax**: Softmax function converts the output of the final fully connected layer of CNN into normalized probabilities (logits) that sum up to 1. Amongst the output labels one with the highest probability is the prediction of the algorithm for the given input image.

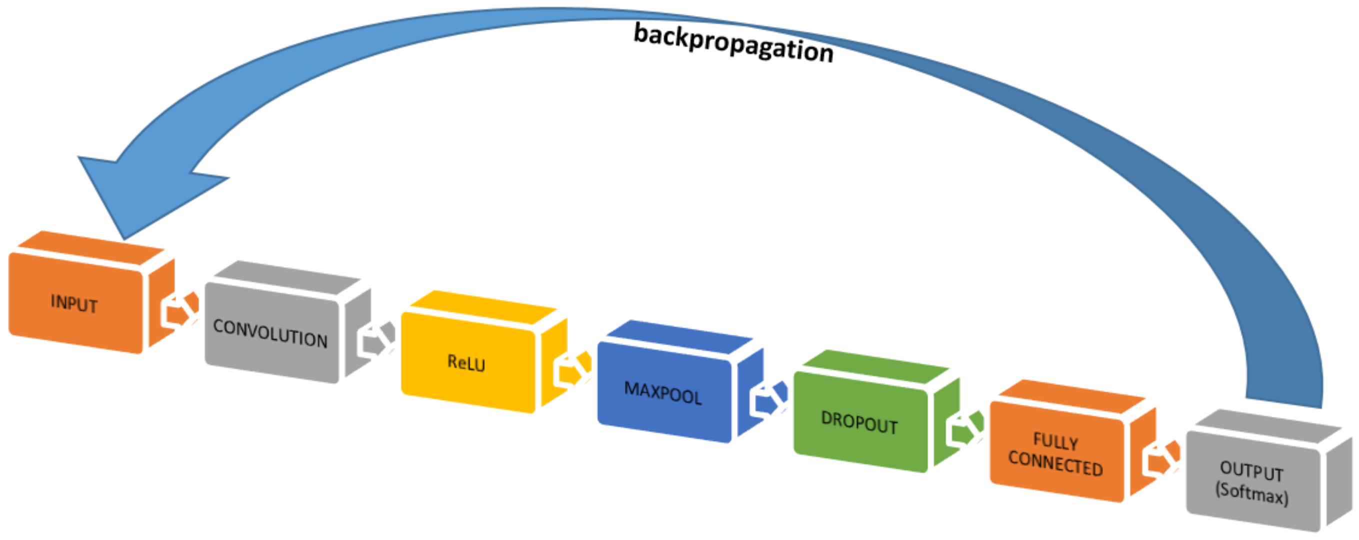


Figure 7

In the feed forward stage the input is taken and sent forward through the layers of convolution, max pooling and fully connected stage. Functions like ReLU, dropout and normalization are also applied. We initialize the weights and biases for the neurons with a constant or a normal distribution. When the learning happens the values of weights and biases change during the learning phase. One can imagine that some kind of feedback is needed once the output prediction is made by the CNN. This is where the next stage of the process, backpropagation comes into picture. In essence backpropagation is similar to the minimizing of a function $f(x)$ in calculus which is done by calculating$\frac{df(x)}{dx} = 0$. Of course our function here would be far more complex and depends on all the weights and biases of each layer. So the two main entities of backpropagation are

- **Entropy or Loss**: We need to define entropy or loss function which reflects the difference between the predicted labels and the actual output label of the training set. This difference acts as a feedback for the network to correct the weights and biases so that this loss or entropy is minimized during the next iterations of feed-forward phase. This could be a function as simple as the squared differences between predictions and actual true labels.

- **Optimizer**: The feedback from the loss function is back propagated through the CNN in reverse. This is done through a gradient optimizer. The gradient optimizer uses the loss value to calculate the correction for weights of the previous layer, and this is done by calculation of gradients of loss (or the weights/bias, whichever is the current layer). Subsequent layers are also corrected in the same manner until we reach the first convolutional layer. With these new corrected weights and biases for the whole network, we accept the next input image and keep repeating the process while minimizing the final loss or entropy and monitoring the accuracy which should be growing as the iterations increase.

Benchmark

As previously mentioned the benchmark that I will be using is the performance mentioned by Goodfellow et al in their paper [2]. They reported an accuracy of 96% for multi-digit classification of entire number on the SVHN dataset which is really an excellent achievement as humans have a 98% accuracy over the same dataset.

III. Methodology

Data Preprocessing

- **Remove outlier**: In the preprocessing stage we can decide certain things based on what we saw in the data exploration stage. As noticed in the data exploration stage there was a single outlier with 6 digits in the SVHN training dataset. We remove this outlier (Figure 6) from our training sample. This will reduce the number of weights and biases and the memory consumption of our program.

- **Convert to Grayscale**: In addition, the color of the image really does not help much in identifying the digits. For this purpose we transform the image from RGB to grey scale. Formula for Luminance from [5] can be used to convert RGB to grayscale. Figure 8 shows some sample of images converted to grayscale and displayed by using matplotlib.

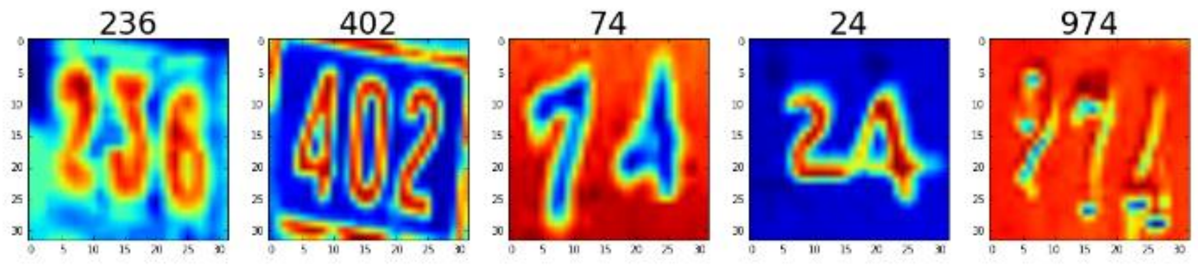


Figure 8

- **Resize Multi-digit dataset**: Another major pre-processing we need to do is to convert the various size images of SVHN to a fixed size of 32 by 32 pixels. For cropped images we are already provided with that, but we need to do the same for the multi digit images of SVHN. Thankfully the bounding box information from the **digitStruct.mat** file can be used to find the smallest rectangle surrounding all the digits and discarding rest of the image. The image is then resized to 32 by 32 pixels. I have used source from http://www.a2ialab.com/ for this conversion which used h5py library.

Implementation

Cropped digit:

- We start with the already provided 32 by 32 pixel images. We read each of the three mat files to memory as np arrays. Then we merge the training and extra dataset to a single array and then randomize the merged and separate that to a new training and validation dataset. While splitting the merged array I roughly took 1% of it and made that into validation set and rest was the training set. These are the final sizes

```
Train data: (600000, 32, 32, 1), Train labels: (600000,)
Valid data: (4388, 32, 32, 1), Valid labels: (4388,)
Test data: (26032, 32, 32, 1), Test labels: (26032,)
```

- The layers for the CNN can be summarized as

```
Conv. Layer 1 shape (BATCH_SIZE, 14, 14, 16)
Conv. Layer 2 shape (BATCH_SIZE, 5, 5, 32)
Conv. Layer 3 shape (BATCH_SIZE, 1, 1, 64)
After Flattening Layer 3 shape (BATCH_SIZE, 64)
Fully connected Layer 1 shape (BATCH_SIZE, 32)
Final Fully connected Layer 2 shape (BATCH_SIZE, 10)
```

For convolution layers 1, 2 and 3 we do local response normalization after the convolution. For convolution layers 1 and 2 we also do the max pooling which reduces size from 28x28 to 14x14, and from 10x10 to 5x5. A dropout of 90% percent is applied after layer 3 of convolution.

Multiple digit:

- In pre-processing stage we had resized each image to 32 by 32 pixel images, each of which could have 1 to 5 digit numbers. We read each of the three mat files to memory as np arrays. Then we merge the training and extra dataset to a single array and then randomize the merged set and separate it to a new training and validation dataset. While splitting the merged array I roughly took 1% of it and made that into validation set and rest was the training set. These are the final sizes

```
Train data: (600000, 32, 32, 1), Train labels: (600000,)
Valid data: (4388, 32, 32, 1), Valid labels: (4388,)
Test data: (26032, 32, 32, 1), Test labels: (26032,)
```

- The layers for the CNN can be summarized as

```
Conv. Layer 1 shape (BATCH_SIZE, 28, 28, 16)
Conv. Layer 2 shape (BATCH_SIZE, 24, 24, 32)
Conv. Layer 2 shape (BATCH_SIZE, 12, 12, 32) ← MAXPOOLING
Conv. Layer 3 shape (BATCH_SIZE, 8, 8, 64)
Conv. Layer 4 shape (BATCH_SIZE, 4, 4, 128)
Conv. Layer 4 shape (BATCH_SIZE, 2, 2, 128) ← MAXPOOLING
After Flattening Layer 4 shape (BATCH_SIZE, 512)
Fully connected Layer 1 shape (BATCH_SIZE, 256)
Fully connected Layer 2 shape (BATCH_SIZE, 64)
```

```
FINAL Fully connected Layer 3 shape - length (BATCH_SIZE, 11)
FINAL Fully connected Layer 3 shape - digit1 (BATCH_SIZE, 11)
FINAL Fully connected Layer 3 shape - digit2 (BATCH_SIZE, 11)
FINAL Fully connected Layer 3 shape - digit3 (BATCH_SIZE, 11)
FINAL Fully connected Layer 3 shape - digit4 (BATCH_SIZE, 11)
FINAL Fully connected Layer 3 shape - digit5 (BATCH_SIZE, 11)
```

For convolution layers 1, 2 and 3 we do local response normalization after the convolution. For convolution layers 1 and 2 we also do the max pooling which reduces size from 28x28 to 14x14, and from 10x10 to 5x5. A dropout of 90% percent is applied after layer 3 of convolution. There are two fully connected layer in the end. The last one has 6 sets of weights and biases since we want to identify the digits in a multiple digit number. The 6 logits can be used to express a 5 digit number in the following way.

Logit0	- Number of digits (this can be 1,2,3,4 or 5)
Logit1 through **Logit5**	- The actual digit predicted (0 to 9 or 10 for no digit)

Logit1 - Logit5 have 11 possibilities. It is because of these 11 possibilities that we have 11 for the number of classes in the final fully connected layer. The BATCH_SIZE for our final model is 25.

Refinement

The primary refinement was in the choice of which optimizer to choose. For this I did testing with the cropped digit datasets. After playing around with several I decided to go with the adaptive gradient decent optimizers. Here I want to show the refinement achieved by simply switching from the adaptive delta optimizer to adaptive gradient optimizer with the same learning rate $\alpha = 0.1$ for the cropped digit training dataset.

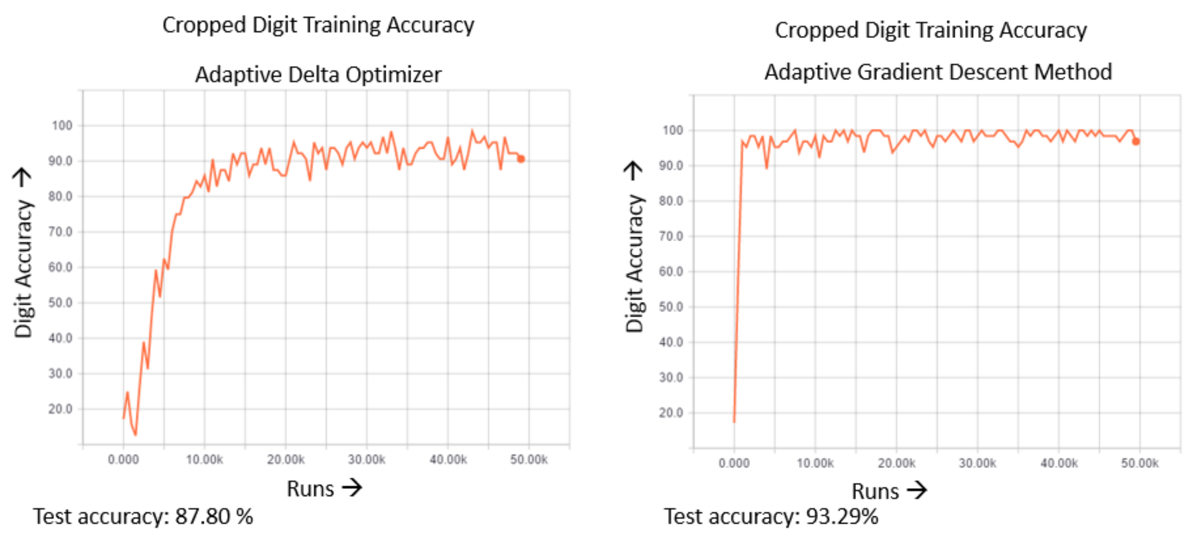


Figure 9

With the adaptive gradient descent method a training set accuracy of 93.29% accuracy was achieved after 50000 iterations with a batch size of 64 in each dataset. For adaptive delta method the training set accuracy was 87.80%. Hence we can see (Figure 9) the convergence to a higher accuracy with the same learning rate for adaptive gradient descent optimizer over adaptive delta optimizer. I found adaptive gradient optimizer was giving better results than other optimizers and also quickly. So this was the choice of optimizer when tests were done with the multiple digit dataset.

IV. Results

Model Evaluation and Validation

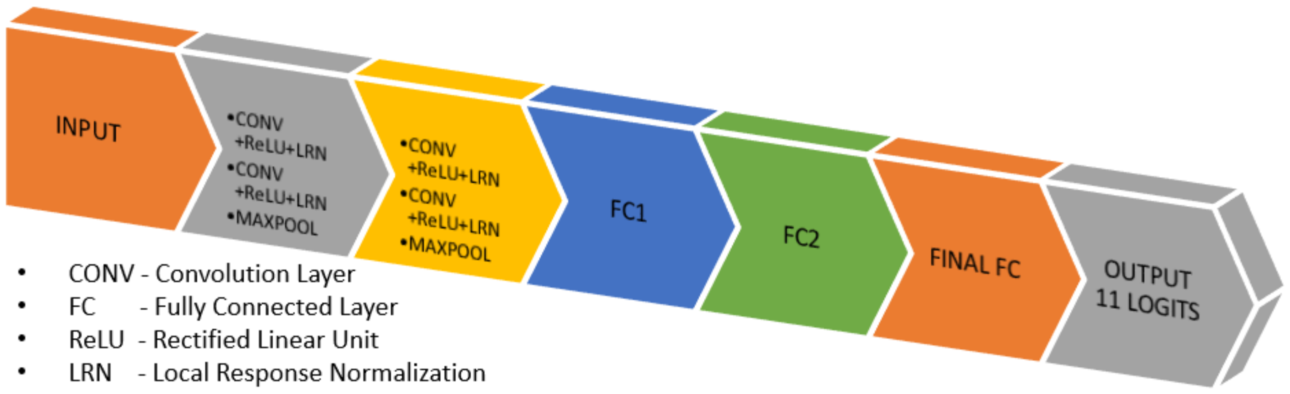
Results from cropped image datasets are not discussed anymore as we saw in previous section that very good accuracy of close to 95% was achieved on training. Here we will present results from the final model used for training over the multiple digits dataset and discuss its performance.

In selecting the model presented earlier I had used the reference from [3], where it is advised that the general ConvNet architecture to follow the pattern

INPUT -> [[CONV -> RELU]****N*** -> POOL?]****M*** -> [FC -> RELU]****K*** -> FC

Where, the * indicates repetition, **FC** the fully connected layer, and the **POOL** indicates an optional pooling layer. Moreover, N >= 0 (and usually N <= 3), M >= 0, K >= 0 (and usually K < 3).

For the model I used, $N = 2$ $M = 2$ and $K = 2$. Hence the layers of model look like


INPUT
•CONV +ReLU+LRN •CONV +ReLU+LRN •MAXPOOL
•CONV +ReLU+LRN •CONV +ReLU+LRN •MAXPOOL
FC1
FC2
FINAL FC
OUTPUT 11 LOGITS
• CONV - Convolution Layer
• FC - Fully Connected Layer
• ReLU - Rectified Linear Unit
• LRN - Local Response Normalization

$N = 1$ always gave accuracy below 80%, but with N=2 and having two convolution layers back to back before applying the destructive maxpool action improved accuracy to more than 80% for training set.

Let us look into the result coming from this trained model. For this particular model, we used a dropout of 0.9 and a learning rate of 0.01 with the optimizer. A batch size of 25 and 300000 iterations were done over the training set. In figure 10 find the training set accuracy evolution for digit and the whole number. We can see that digit accuracy approaches very close to 100 and the number accuracy is in the early 90's.

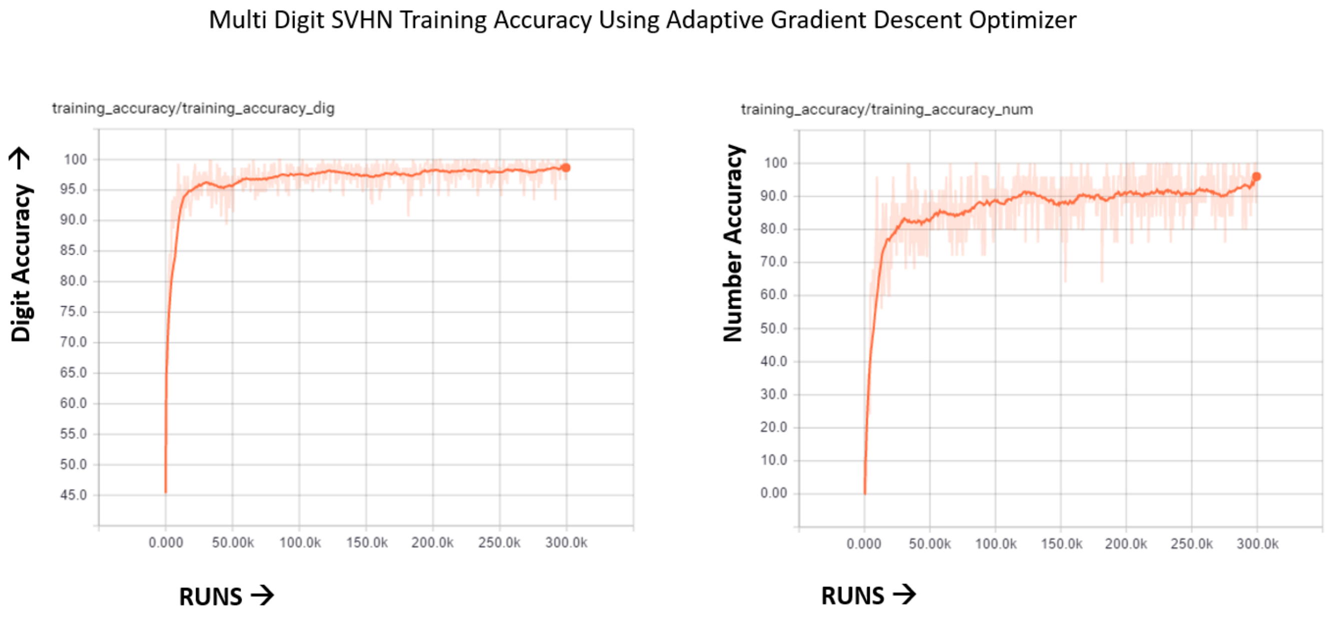


Figure 10: Digit and whole number training accuracy plot with respect to iteration number.

The validation set accuracy also shows a similar plot and is shown in figure 11. We can see the relatively less spikes in the validation dataset, which is expected because we are always testing the same sample with improved weights. The digit accuracy reaches close to 96% and the whole number accuracy approaches 86%.

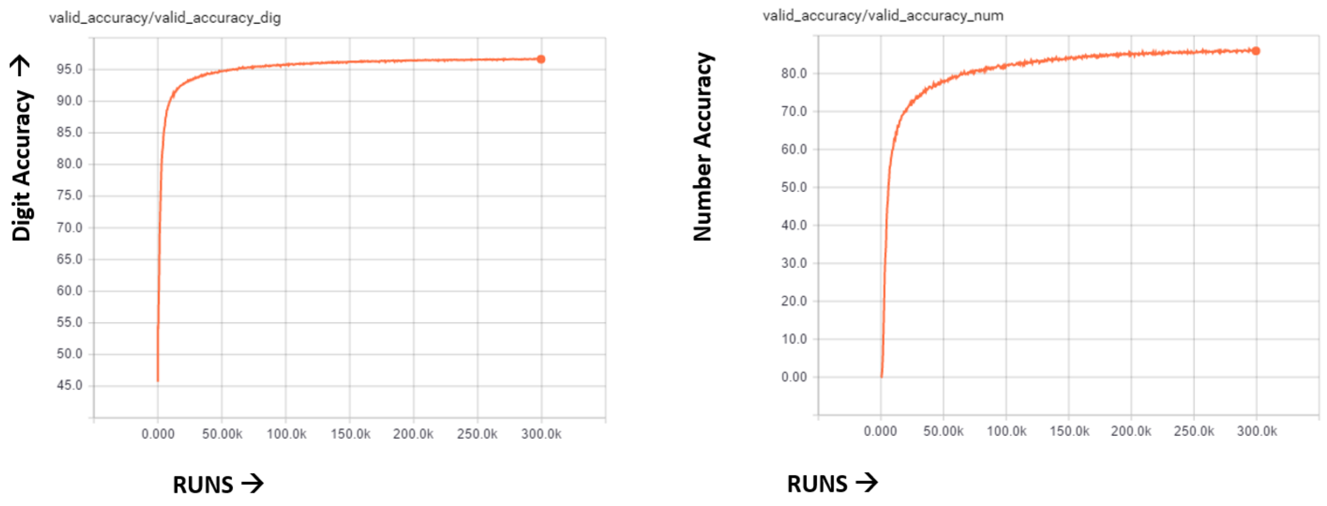
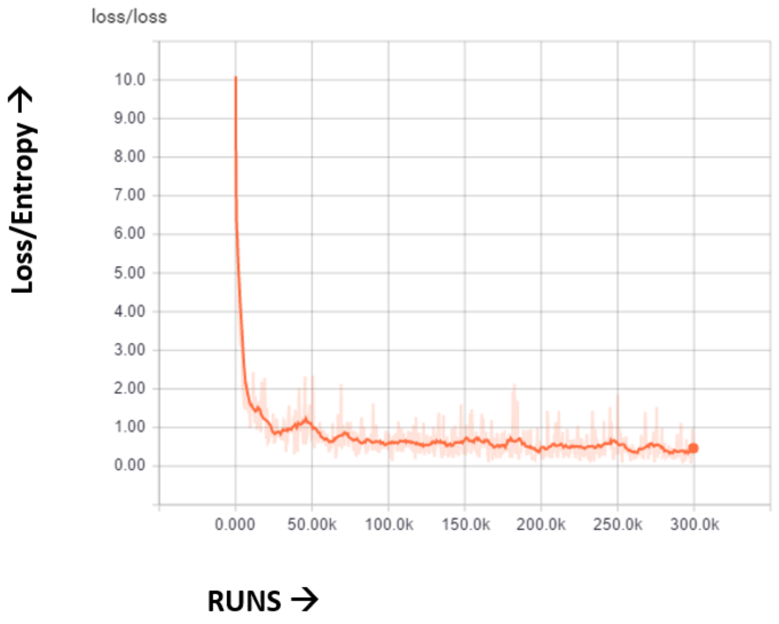


Figure 11: Digit and whole number validation dataset accuracy plot with respect to iteration number.

Of course the most important result is the TEST set accuracy. **For our TEST dataset we had the entire number accuracy at 83.36% and the digit recognition accuracy at 95.94%.**

Let's also take a look at the evolution of loss to decreasing values, as the accuracy improves during training. The plot is shown in Figure 12.



Justification

Our final test set accuracy for "entire numbers" was around 84% and accuracy on getting the digits correct was around 96%. This is clearly not as good as the benchmark accuracy of 96% for reading "entire number" of multiple digits. However that benchmark was achieved over running a deeper model for multiple days over several GPU's. My result does fall below the initial expectation I had set of 90%. However this is still pretty good and moreover we are getting the digit or label correctly with an accuracy of 96%. I think this is a satisfactory result considering the resources I had at disposal.

V. Conclusion

Free-Form Visualization

For visualization of the result I took 10 random images from the test dataset. In Fig 12 we can see the image as it is and in Fig 13 we can how the cropped greyscale version of the images look like.

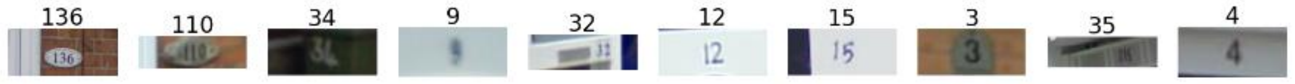


Figure 12: Ten Random images from test dataset with the correct labels

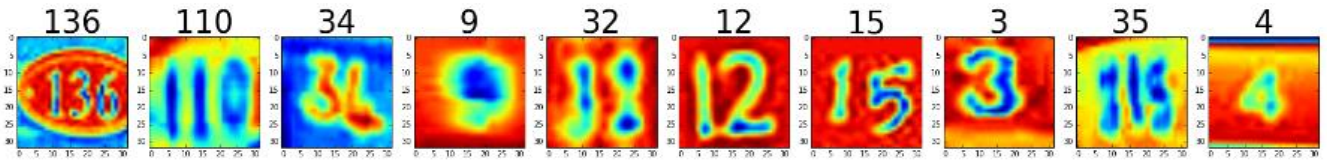


Figure 13: The input images to the model with the correct labels

Now with the saved weights and biases from our model presented earlier we try to make the prediction on digits. The predicted results from the model are given in Figure 13. We can see that we got an accuracy of 80% in getting the full number correct, which is in line with the accuracy we calculated on the test dataset. We got all but three digits wrong which is a very good result.

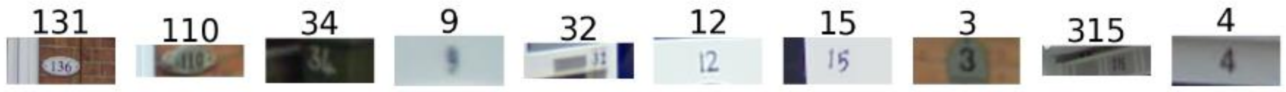


Figure 14: The predicted numbers are shown above the image.

Reflection

So in summary we first download the mat files from [1] for cropped digits and train our model to a good accuracy with that dataset. This can be used as a reference to see if the model is working as per our expectation and giving good accuracy. Digit accuracy as high as 95% was observed with the test dataset.

Whether it be single cropped digit or multiple digits dataset (composed of the tar.gz files of house number images in PNG format), we start by merging the extra and training set, shuffle the merged set, and generate a new training and validation dataset from it. The test dataset is used as it is. This training dataset is trained over a multilayered convolution deep neural network and optimized with an adaptive gradient algorithm.

Our final output is a well-trained model (the weights and biases of the network) with a measured accuracy over test dataset. This model can then be used to predict numbers up to 5 digits from an arbitrary image containing a number.

Improvement

- The improvements I would like to see is in trying out multiple parameters tuning for the optimizer like we did in our assignments for nanodegree. The only thing that stopped me from doing were time and money constraint.
- All my training was done on the GPU supported machine p2.xlarge, with 61 GB memory, in AWS (Amazon Web Services) cloud. Cutting edge options like more GPU's, faster processors or access to Google's TPU (Tensor Processing Unit) will certainly speed up the training part.
- I think careful addition of more layers to CNN and faster hardware will help in improving accuracy of the model further.

References

1. Source of datasets: *http://ufldl.stanford.edu/housenumbers/*
2. Ian J. Goodfellow, Yaroslav Bulatov, Julian Ibarz, Sacha Arnoud, Vinay Shet, Multi-digit Number, *Recognition from Street View Imagery using Deep Convolutional Neural Networks*, 2014
3. Course notes from Stanford CS class CS231n: Convolutional Neural Networks for Visual Recognition: *http://cs231n.github.io/*
4. Krizhevsky et al., ImageNet classification with deep convolutional neural networks (NIPS 2012).
5. Recommendation ITU-R BT.709-6 *https://www.itu.int/dms_pubrec/itu-r/rec/bt/R-REC-BT.709-6-201506-I!!PDF-E.pdf*