Machine Learning Nanodegree Capstone Project
Deepa Mohan, Sep 2016

Digit Recognition from SVHN images using Deep Convolutional Networks

Introduction
I was excited to get started on the Capstone Project and I was drawn towards multiple topics as all of them seemed so interesting - Classifying stars and galaxies, classifying stars based on their brightness, Kaggle Competition problems, UC Irvine datasets like breast cancer detection, identifying religion based on the national flag, Picking the right school for kids, Accident statistics from DMV, etc. Though all of these have tons of applications and are extremely relevant, I got fascinated by Deep Learning and the immense potential it has and decided to pick a project in this area. Being a micro-architect at Intel Corp, designing server hardware for super-computing and deep learning/machine learning, I could realize how designing powerful systems could be so important for Deep Learning. I took the Deep Learning Course on Udacity and the last month or more has been a very challenging and an equally rewarding experience. I got to understand so many new concepts in Deep Learning, tensorflow, Cloud services like GCP and also experiment in the same areas.

Project Overview
Detecting and reading text from natural images is a hard computer vision task that is central to a variety of emerging applications. Related problems like document character recognition have been widely studied by computer vision and machine learning researchers and are virtually solved for practical applications like reading handwritten digits. Reliably recognizing characters in more complex scenes like photographs, however, is far more difficult: the best existing methods lag well behind human performance on the same tasks. In this project, we will focus on a restricted instance of the scene text problem: reading digits from house-number signs in street level images. If this problem is solved, more accurate maps can be built and navigation services can be improved.

Unfortunately recognizing characters in natural scenes is more difficult: characters and digits in photographs are corrupted by natural phenomena that are difficult to compensate for by hand, like severe blur, distortion, and illumination effects on top of wide style and font variations.After getting hands dirty with the nMNIST and MNIST tutorials, I decided to take up the Street View Housing Numbers dataset [SVHN] and build a deep learning network that can predict the digits in the images [Digit Recognizer]. SVHN is a real-world image dataset for developing machine learning and object recognition algorithms with minimal requirement on data preprocessing and formatting. It can be seen as 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).

The SVHN dataset was obtained from a large number of Google Street View images using a combination of automated algorithms and the Amazon Mechanical Turk (AMT) framework which is used to localize and transcribe the single digits. It can be downloaded freely from the web: http://ufldl.stanford.edu/housenumbers/ .The SVHN dataset, compared to many existing benchmarks, hits an interesting test point: since most digits come from printed signs the digits are artificial and designed to be easily read, yet they nevertheless show vast intra-class variations and include complex photometric distortions that make the recognition problem challenging for many of the same reasons as general-purpose object recognition or natural scene understanding.

Recently, there has been work done on the same problem. In [4], the authors describe how they were able to achieve ~90% accuracy with unsupervised learning techniques [linear SVM, K-means clustering] to solve this problem. In [2], the authors use convolutional neural networks (ConvNets) rather than prior hand designed vision approaches, They also used Lp pooling and establish a new state-of-the-art of 94.85% accuracy. In [1], the authors describe how in [4] or other traditional methods, they separate out the localization, segmentation, and recognition steps, but in their work they propose a unified approach that integrates these three steps via the use of a deep convolutional neural network that operates directly on the image pixels and apply a 11 layer convolutional network to achieve 96% accuracies.

In this project, we will try to solve the same problem by first building a small convolutional network, preprocessing the dataset for our needs, build and train the model using softmax regression, define and measure various metrics like accuracy, loss and build on the convnet architecture by utilizing pooling, dropout techniques. We will use tensor-flow for building the computational graph and implementing this project. We will then strive to improve high accuracies and build a robust model.

Data Exploration

SVHN dataset has 10 classes, 1 for each digit. Digit '1' has label 1, '9' has label 9 and '0' has label 10. 73257 digits for training, 26032 digits for testing, and 531131 additional, somewhat less difficult samples, to use as extra training data. It comes in two formats:
1. Original images with character level bounding boxes.
2. MNIST-like 32-by-32 images centered around a single character (many of the images do contain some distractors at the sides).
I decided to use the MNIST like 32x32 .mat format 2 for the digit recognition project.

Exploratory Visualization

I plotted some of the images from the dataset and this is how they look. As we can see from the training samples, they images are in different colors and fonts. They in general lack clarity, some of them are very blurry and hard for even the human eye to perceive. In this format, as the image is centered around a single character, they are less clear and have distractions on the sides.

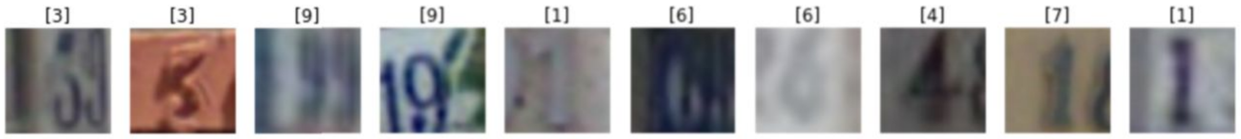


Figure1. Training dataset samples

Lets look at a few samples in the extra dataset. These are evidently somewhat less complicated and can help in training the model for a better accuracy.

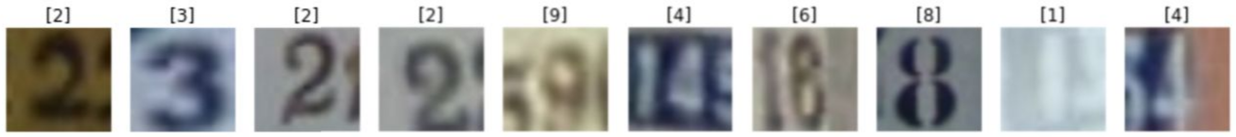
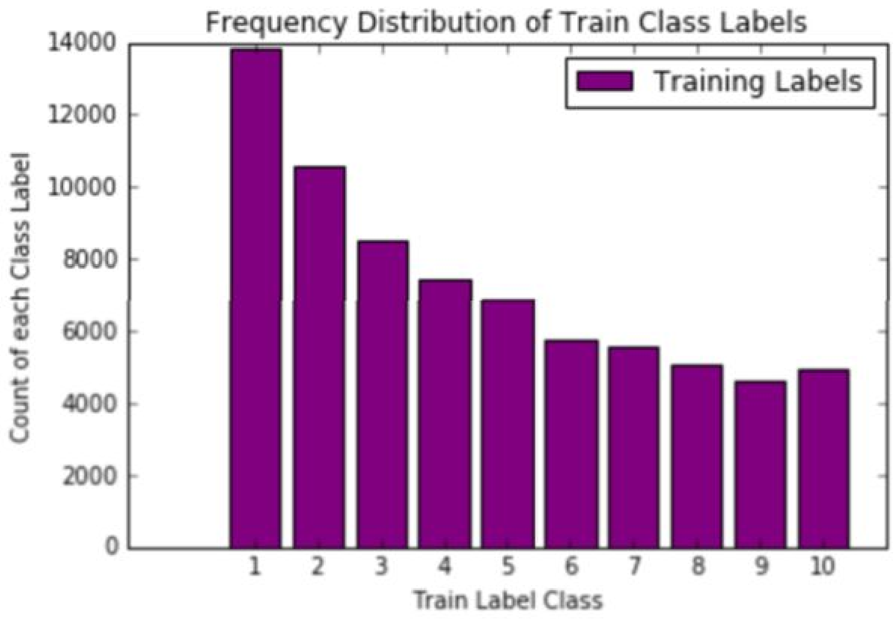
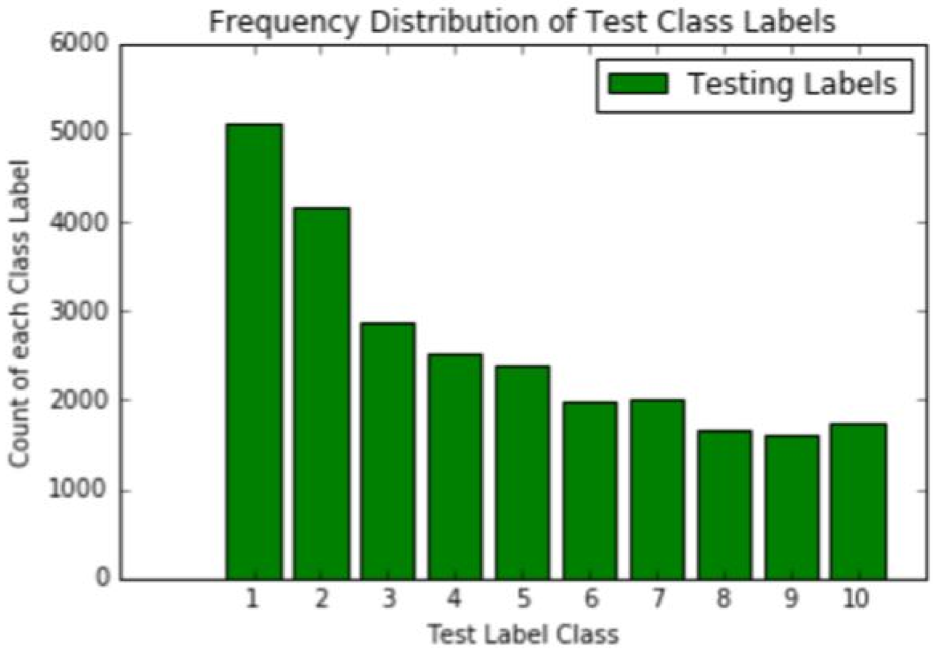


Figure2. Extra dataset samples

I plotted a bar graph to see how the distribution of the class labels is.
Train labels - {1: 13861, 2: 10585, 3: 8497, 4: 7458, 5: 6882, 6: 5727, 7: 5595, 8: 5045, 9: 4659, 10: 4948}



Test Class Labels - {1: 5099, 2: 4149, 3: 2882, 4: 2523, 5: 2384, 6: 1977, 7: 2019, 8: 1660, 9: 1595, 10: 1744}



We can see that the distribution of train and test class labels is pretty similar and the digit 1 occurs maximum number of times in both datasets. We also change training label 10 to a 0 and have the labels range from ‘0’ to ‘9’.

Validation dataset

We have seen how important cross-validation is to make sure we don't overfit on the test data. In the deep learning assignment, Vincent talks about the importance of having a validation dataset which is a subset of the training dataset and how we can use the validation dataset to train the model and measure the actual performance on the test data set. In [1], it is not clear how they come up with a validation dataset, however in [2], they describe this in detail. I followed the same method and composed the validation set with 2/3 from training samples (400 per class) and 1/3 from extra samples (200 per class), yielding a total of 6000 samples. This distribution allows to measure success on easy samples but puts more emphasis on difficult ones. I added the remaining extra dataset samples to the training dataset and also shuffled them.

These are the sizes after this step (#Samples, Image_size, Image_size, #Channels(RGB))

Train dataset	-(598388, 32, 32, 3)
Train labels	-(598388,)
Validation dataset	-(6000, 32, 32, 3)
Validation labels	-(6000,)
Test dataset	-(26032, 32, 32, 3)
Test labels	-(26032,)

We plot samples from the dataset to make sure, it still looks good after the shuffling and re-organizing.

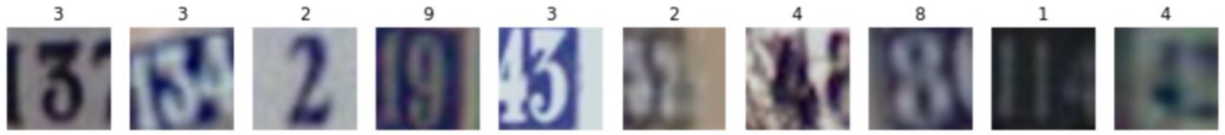


Figure3. Training dataset samples after merging with extra dataset

We need to now preprocess the dataset before we feed it to the deep network. Let's consider the system requirements before we dive into this.

Computing and System limitations/Sample set size

When I took the deep learning course on Udacity and attempted the assignments, I realized how high computational power and memory bandwidth requirements play a crucial role in the experiments we can using deep convolutional networks especially when we have to train the model on large datasets to achieve high accuracies. The final design choice is then a combination of all these factors, how well your model does in the given system constraints that also facilitate how many parameters you can tune and arrive at the optimal solution. I started doing the Deep Learning assignments using my 4GB no GPU laptop and soon ran into issues, then I chose Google Cloud Platform with a 2CPU,7.5GB,100GB virtual machine to run convolutional network experiments. In spite of this, I was not able to run sample sets greater than 50,000 as there were some limitations as to which virtual machine I can pick. The kernel would repeatedly die, although I can run a lot of preprocessing on large datasets, when I apply these to train the model, it would terminate the kernel. If I increase the dataset, I couldn't play around with the parameters to tune my model. Hence I decided to keep the train dataset sample size at 45,000, test data sample size at 15000 and validation data sample size at 6000. From various forum discussions, there was a general rule of thumb to at least have a 3:1 ratio for the training and test sets. I then randomly chose these number of samples from the entire length of the dataset. This enabled me to analyze how the model behaviors change from hyperparameter tuning and also allowed me to slowly build my convolutional network to attain high test accuracies.

Data Preprocessing

Normalization of data

In the deep learning course, Vincent talks about the importances of having well-conditioned data with zero mean and equal variance. The optimizer puts in a lot of effort on badly conditioned data to find an optimal solution. I researched online to see what the different techniques were to normalize image data. In [3], these techniques are discussed

1. Simple re-scaling is where we subtract and divide by 128 for each color channel - this is also used in the deep learning assignment.
2. Subtracting mean and dividing by standard deviation - [1] talks about this technique as they mention that they subtracted the mean but did not consider any local contrast normalization. [2] mention that they pre-processed the samples with a local contrast normalization (with a 7x7 kernel) on the Y channel of the YUV space followed by a global contrast normalization over each channel.
3. PCA/ZCA whitening - lot of kaggle competition folks suggested using this technique.

As the .mat images of the SVHN dataset are color images in 32x32 format and they have 3 channels (RGB), I converted these to gray scale images first by computing the dot product using this formula. I had several issues here due to system limitations here and could only successfully do this computation for 45,000 samples.

```
gray = np.dot(image, [[0.2989],[0.5870],[0.1140]])
```

After this I applied global contrast normalization. I used the pylearn2 global_contrast_normalize() for reference. This is how the data looks after these two techniques

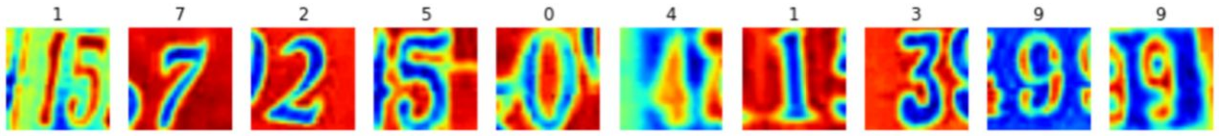


Figure4. Training dataset samples after GCN

Which machine learning model to use?

I ran a first pass simulation on logistic regression classifier and found that it not only took a really long time but also the accuracy came out to be very poor - around 0.17. I then tried experimenting with a fully connected neural network with L2 regularization and it didn't do good either and I got around the same accuracy. Hence it was obvious that convolutional network was the right choice for this dataset.

Convolutional Neural Networks take advantage of the fact that the input consists of images and they constrain the architecture in a more sensible way. In particular, unlike a regular Neural Network, the layers of a ConvNet have neurons arranged in 3 dimensions: width, height, depth.

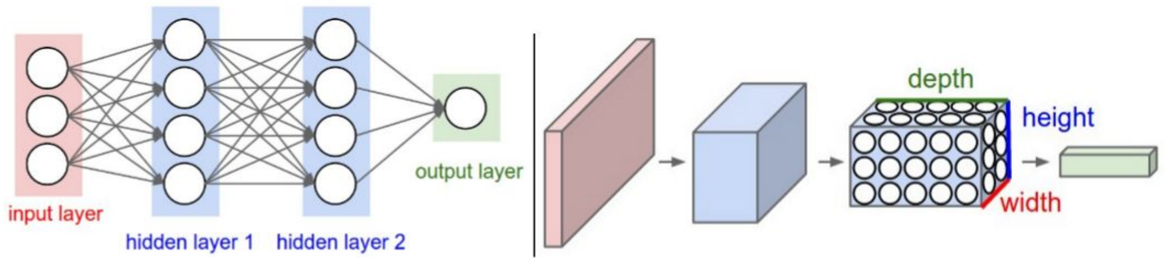


Figure5:Nueral Network and Conv Net

Algorithm and Technique

The input dimensions in the .mat format is 32x32x3 (width, height, depth respectively). Since we converted the image to grayscale, I will add a single channel and feed 32x32x1 input size to the convnet. We will use tensorflow to do the following

- Create a softmax regression function that is a model for recognizing SVHN digits, based on looking at every pixel in the image. This involves setting up weights and biases, creating nodes for input images and output classes.

Initializing weights and biases is a very important step. The weight matrix will be initialized using random valued following a normal distribution with standard deviation of 0.1. The normal distribution make it much easier for the model to learn and adjust them. The biases get initialized to zero for layer1, however for the other layers, I will use a initial constant value. This is to avoid dead neurons in the network [we will be using RELU] and also avoid a zero gradient descent.

```
layer1_weights = tf.Variable(tf.truncated_normal(
[patch_size, patch_size, num_channels, depth], stddev=0.1))
layer1_biases = tf.Variable(tf.zeros([depth]))
layer2_weights = tf.Variable(tf.truncated_normal(
[patch_size, patch_size, depth, depth], stddev=0.1))
layer2_biases = tf.Variable(tf.constant(1.0, shape=[depth]))
```

- We multiply the vectorized input images x by the weight matrix W, add the bias b, and compute the softmax probabilities that are assigned to each class.

```
y = tf.nn.softmax(tf.matmul(x,W) + b)
```

- Tensorflow can use automatic differentiation to find the gradients of the loss with respect to each of the variables. TensorFlow has a variety of built-in optimization algorithms that we will experiment with. We will use steepest gradient descent to descend the cross entropy. Use Tensorflow to train the model to recognize digits by having it "look" at thousands of examples. We split the training data into small batches and repeat the experiments over several steps so we can compute the loss function at every step and apply this to adjust the weights appropriately.
- Check the model's accuracy with our test data
- Strive to minimize the loss which is the cross entropy function between the target and the results.
- Build, train, and test a multilayer convolutional neural network to improve the results

The following picture from Deep Learning Udacity course neatly summarises the technique used. The vectorized input image is matrix multiplied with the variable weights and biases and scores are computed. The scores are transformed into probabilities using a softmax function and cross entropy then converts the scores into one hot output labels. Tensorflow [5] makes this a lot easier for us.

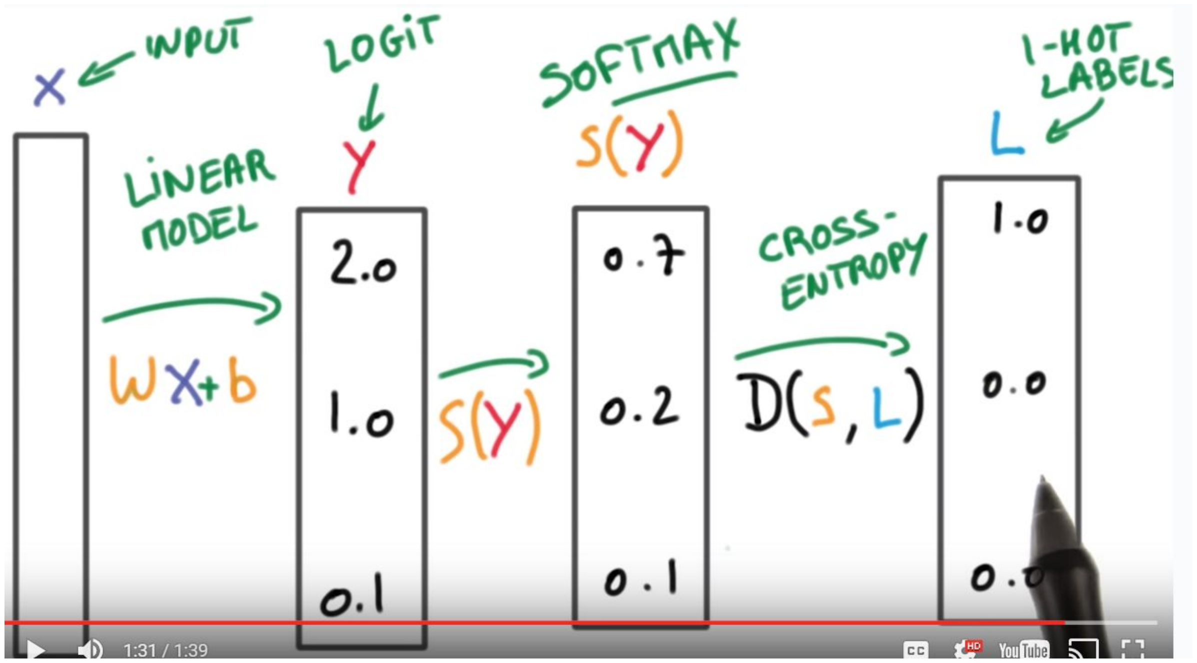


Figure5:Technique for solving the problem of image digit recognition.

Metrics - Accuracy and Loss

The model we will build will try to attain maximum accuracy and minimum loss

Accuracy

```
def accuracy(predictions, labels):
  return (100.0 * np.sum(np.argmax(predictions, 1) == np.argmax(labels, 1))
```

/ predictions.shape[0])

Loss
Our loss function is the cross-entropy between the target and the model's prediction
loss = tf.reduce_mean(tf.nn.softmax_cross_entropy_with_logits(logits, tf_train_labels))

Building and Training the model

I first built a small network with two convolutional layers, followed by one fully connected layer. **Convolutional networks** are more expensive computationally, so we'll limit its depth and number of fully connected nodes. A simple ConvNet architecture will have the following components

- INPUT [32x32x1] will hold the raw pixel values of the image, in this case an image of width 32, height 32, and 1 channel for grayscale. Initializing the weights with a normal distribution was a critical aspect along with making the data well-balanced. Tensorflow does a lot of these things for us.
- CONV layer will compute the output of neurons that are connected to local regions in the input, each computing a dot product between their weights and a small region they are connected to in the input volume.
- RELU layer will apply an elementwise activation function, such as the max(0,x)max(0,x) thresholding at zero. This leaves the size of the volume unchanged ([32x32x12]). However by using a patch_size of 5 and stride of 2, we reduce dimensionality. The output at this stage is 8x8x16. We reshape and flatten this this before we apply it to the FC layer.
- FC (i.e. fully-connected) layer will compute the class scores, resulting in volume of size [1x1x10], where each of the 10 numbers correspond to a class score, such as among the 10 numbers from 0-9. As with ordinary Neural Networks and as the name implies, each neuron in this layer will be connected to all the numbers in the previous volume.

I got an accuracy of **71.9%** here and I will use this as a starting point and improve my model from this baseline. In [1] an **accuracy of 97.4%** was achieved for single digit recognition which can be treated as a **benchmark** for multi-digit recognition, however it is not possible to meet this due in my model due to sample-set limitations (only 45000 training dataset). So I tried to achieve at least 90% accuracy on the test dataset, however my intuition was that greater than 90% accuracy could be achieved with advanced computer vision data pre-processing techniques like identifying duplicates, different normalization techniques etc which are also listed in the areas of improvement.

In the above baseline model, there is some information loss and to compensate this we add a POOLing layer at the output of each conv layer. POOL layer will perform a downsampling operation along the spatial dimensions (width, height), resulting in volume such as [8x8x16]. This increased the accuracy to **75.5%.** I tried avg vs max pooling and for my model, avg pooling gave better results.

This model had a depth = 8 and num_hidden = 48, when I increased depth = 16 and num_hidden = 64, the accuracy increased to **85.8%**. I tried to vary depth parameter for the first and second Conv Layer, but I had kernel issues and hence decided to keep the same value.

This was the CNN architecture, the corresponding shapes.

C1 - (16, 32, 32, 1)
S2 - (16, 28, 28, 16)
C3 - (16, 14, 14, 16)
S4 - (16, 10, 10, 16)
F5 - (16, 5, 5, 16)
Tensor("Reshape:0", shape=(16, 400), dtype=float32)
F6 - 400 x 10

Hyper-parameter tuning

Learning Rate

When training a model, it is often recommended to lower the learning rate as the training progresses. I set this up as a decaying function.

learning_rate = tf.train.exponential_decay(0.05, global_step, 1000, 0.85, staircase=True)

Dropout is an extremely effective, simple and recently introduced regularization technique. While training, dropout is implemented by only keeping a neuron active with some probability p (a hyperparameter), or setting it to zero otherwise. I implemented 2 dropouts one after each FC layer and observed the **accuracy increase to 87.1%**. I ran experiments for various values of dropout 0.1-0.9 and noticed that after a value of 0.85, it didn't make an impact on the accuracy.

Batch_size and num_steps

The experiment so far was with 10000 steps with a **step_size** of 500. I also tried a step_size of 50 but it didn't change the final test accuracy much, so decided to use 500 as the results were obtained much faster. I also tried a batch size of 32, with 80000 steps, I got a test accuracy of around 89.3%

I decided to add another Conv layer and ran the experiment over 100000 steps with a batch_size of 16. This gave a **test accuracy of 90.4%**. We can see that the training minibatch accuracy reaches 100% at around 7000 steps and validation accuracy reaches a maximum value of 90.5% and stays more or less around the same value.

Optimizer

The optimizer used in this case was **GradientDescent**Optimizer. I also ran experiments with AdamOptimizer, AdaGradOptimizer and RMSOptimizer but GradientDescentOptimizer was about 1% more accurate than AdamOptimizer, AdaGradOptimizer.

optimizer = tf.train.GradientDescentOptimizer(learning_rate).minimize(loss, global_step=global_step)

Model	Test Accuracy(%)	Depth	Num_Hidden	Num_Steps	Drop-Out	Batch_Size	Step_Size
LinearRegressor	17						
2 Conv + FC with RELU	71.9	16	64	1000		16	50
2 Conv + FC with RELU, POOLing	75.5	8	48	1000		16	100
2 Conv + 2 FC with RELU, POOLing	85.8	16	64	9000		16	500
2 Conv + 2 FC with RELU, POOLing, 2 DropOut	87.1	16	64	10000	0.1-0.95, picked 0.85/0.95	16	50,500 picked 500
3 Conv + FC with RELU, Pooling, Dropout	89.3	16	128	100000	0.95	32	500

3 Conv + FC with RELU, Pooling, Dropout	90.4	16	128	100000	0.95	16	500

The below table summarize all the experiments I have tried and the green highlight shows the configuration of the final solution.

Final Solution

A 6 layer network with 3 convolutional layers network with a depth of 16, followed by a dropout layer and a fully connected layer with 128 hidden layers.

(C1, 28, 28, 16)
(S2, 14, 14, 16)
(C3, 10, 10, 16)
(S4, 5, 5, 16)
(C5, 1, 1, 128)
(F6, 128)
Output Layer 128x10

The structure closely resembles the LeNet Architecture, the difference being the parameter sizes.

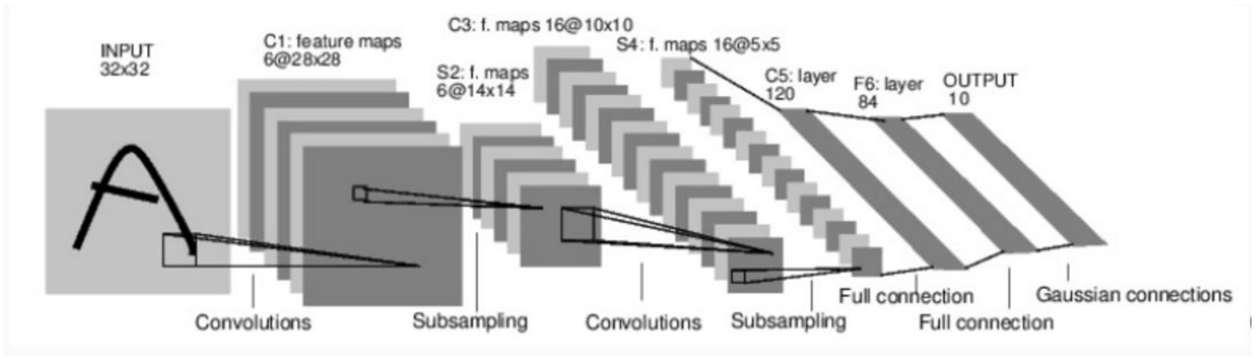


Figure 7: LeNet Architecture

Results

Accuracy

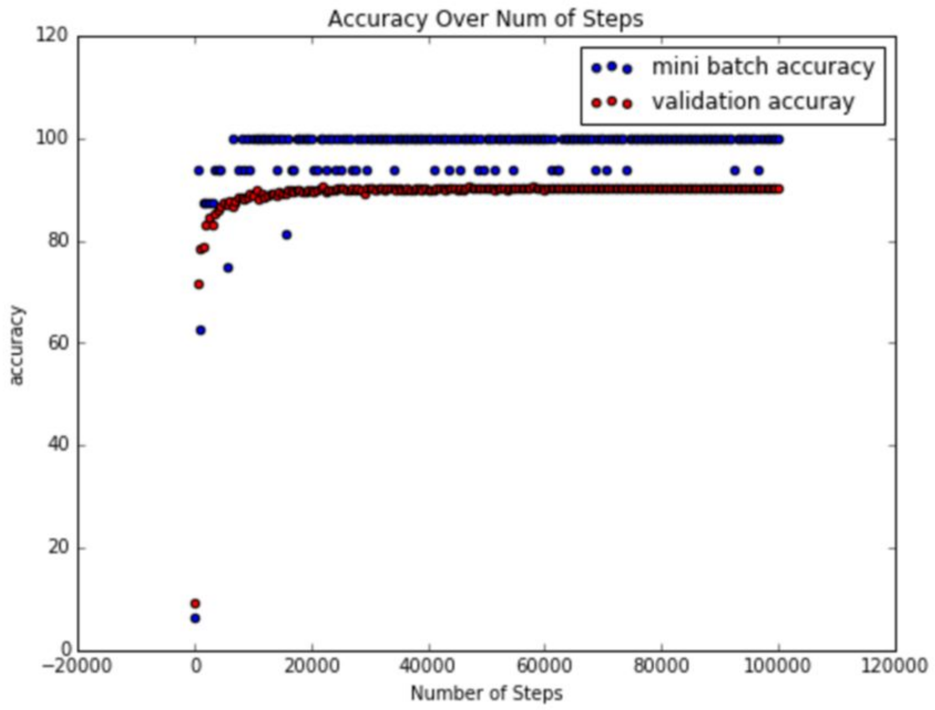


Figure 8: Accuracy vs train steps

The plot on the final model shows how the validation accuracy reaches around 89% and minibatch accuracy reaches around 100% and stabilizes around 20000 steps. The model is learning pretty well.

The mean and max values of validation_accuracy are 89.2 and 90.48 respectively.

Loss Function

The minibatch loss value is computed at every step and we can see that as the model learns the loss value at the step starts to decrease and reaches a very low value.

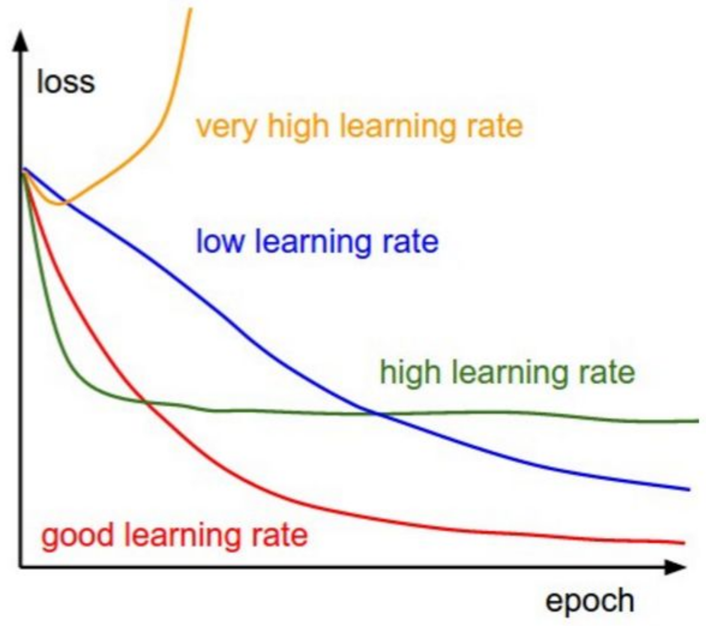


Figure 9. Loss function over epoch size

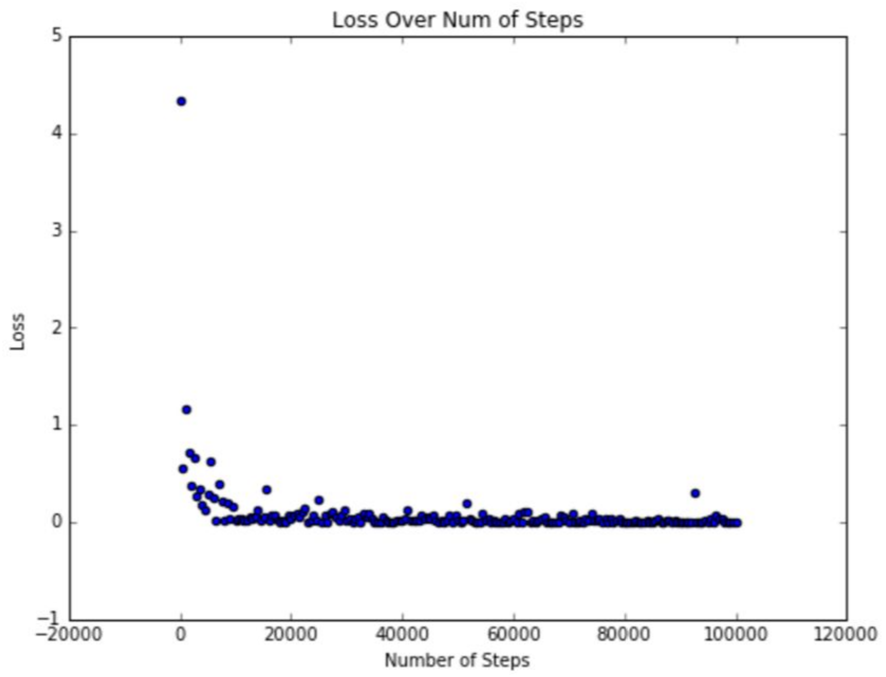


Figure 10. Loss vs train steps

The mean value of Loss is 0.0838742 and the min value is 9.26075e-06

Robustness

We need to check how well our model performs on unseen data. When we create our test dataset, I sample 15,000 samples from the SVHN test dataset of 26,032. I ran the experiments multiple times regerating the test dataset, indicating the 15,000 samples could be over the entire testdataset space. This is a good representation of unseen data. I consistently noticed same/similar values of minibatch accuracy, testing accuracy and validation accuracy indicating the model is robust.

Conclusion/Areas of Improvement

The deep network discussed in the final solution section with a test accuracy of 90.4% is my final model. To improve the accuracy, I could do the following experiments but was limited due to system constraints and knowledge needed of GCP and AWS virtual machine services and how I can deploy them to perform large scale experiments.

1. Increase sample-set size
2. Vary depth beyond 8 and 16, hidden parameters for various layers, I had kernel issues here
3. Experiment with patch_size. I tried to do this but kernel would repeatedly die.
4. Learn more on computer vision techniques to preprocess data better
5. Improve ConvNet architecture, the model here closely resembles LeNet but other models could be studied
6. Would like to investigate more on the Optimizer options.
7. Would like to understand more on tensorflow knobs and how we can finetune them.

Is the model good enough? The model accuracies need to be higher than 99% to beat humans, however this could be achieved with MNIST and nMNIST datasets as they are a bit more perceptible. However the SVHN dataset is complicated and due to distortions, color, zooming, irregularities in font, orientation of numbers, blurring it is hard to achieve higher accuracies on small system. Some of them are even hard for the human eye. The google implementation in [1] achieved 96% on multi-digit and 97% on single digit but this was also in support of the tremendous infrastructure and how deep the model can be laid out. They had 11 layers in their network while my model has only 6. I believe my model is definitely good but not ready to be deployed especially for scalability. It can certainly do better if the known limitations captured above are addressed.

Acknowledgements
I am very grateful to my fellow Udacity Forum members for being so helpful and shedding light and hope in the compute intensive project and Udacity to offer this unique learning platform.

References

[1] https://arxiv.org/pdf/1312.6082.pdf
[2] https://arxiv.org/pdf/1204.3968.pdf
[3] http://ufldl.stanford.edu/wiki/index.php/Data_Preprocessing
[4] http://ufldl.stanford.edu/housenumbers/
[5] http://cs231n.github.io/neural-networks-3/#loss
[6] https://www.tensorflow.org/versions/r0.10/tutorials/mnist/pros/index.html#deep-mnist-for-experts