Deep Learning Capstone Report

Xuanzhe Wang

Table of Content

I. Definition

a. Project Overview

Project is to solve an image classification problem. In machine learning, classification problem is to categorize inputs. For example, given several images, the machine learning agent need to tell which image is a car, which image is a person.

That is, given an input X (X is a vector), we need to find a function f that:

f(x) -> y, where y is the category/label of x

In this project, a special technical called convolutional neural network is applied. Convolutional neural network has been applied in image recognition, video analysis, drug discovery and other fields. For example, the famous AlphaGo by DeepMind used convolutional neural network.

b. Problem Statement

In this project, I trained a convolutional neural network to classify real world street digits.

The data set used is the Street View House Numbers (SVHN): A large-scale dataset of house numbers in Google Street View images.

The data set consists of 73257 training images and 26032 testing images. Each image is a 32x32 bitmap(array) with 3 color channels. Each image also has a label associated with it. These labels represent digits from 0 - 9.

My task is to train a model, as a function f, such that:

f(Image) -> label, where label ranges from 0 - 9

c. Metrics

Accuracy: the number of correctly classified inputs / the number of inputs

II. Analysis

a. Data Exploration

There are 73257 images in the training set and 26032 images in the testing set. Each image is represented as a 3-channel RGB 32x32 array.

b. Exploratory Visualization

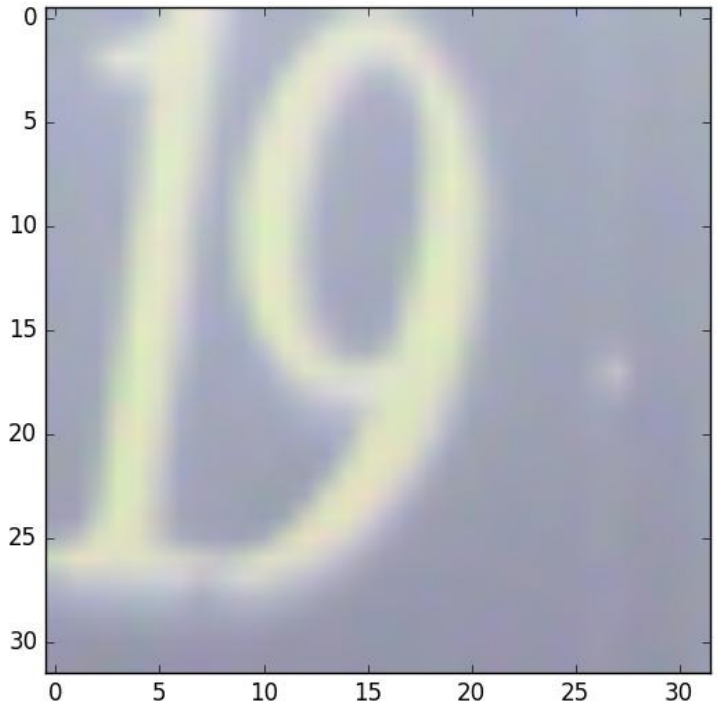


Figure 1, image_1_origin, label 9

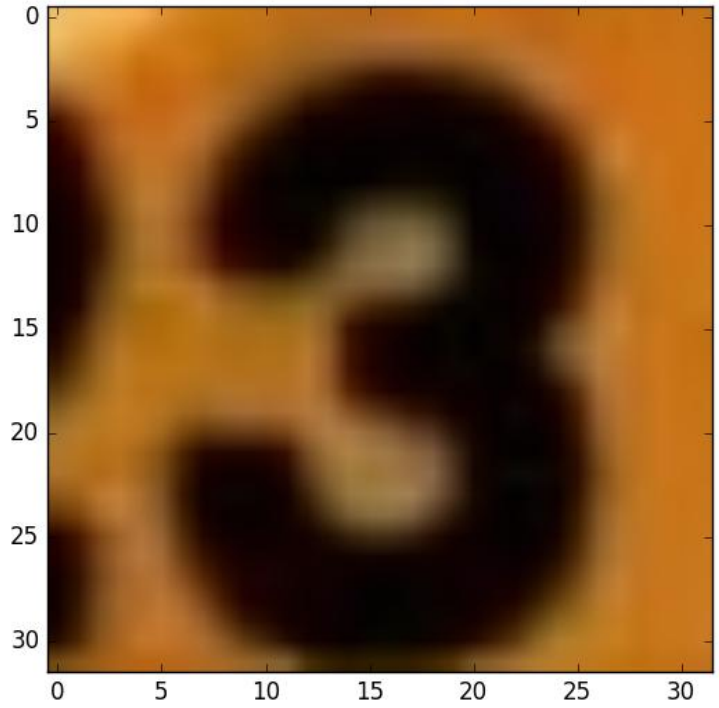
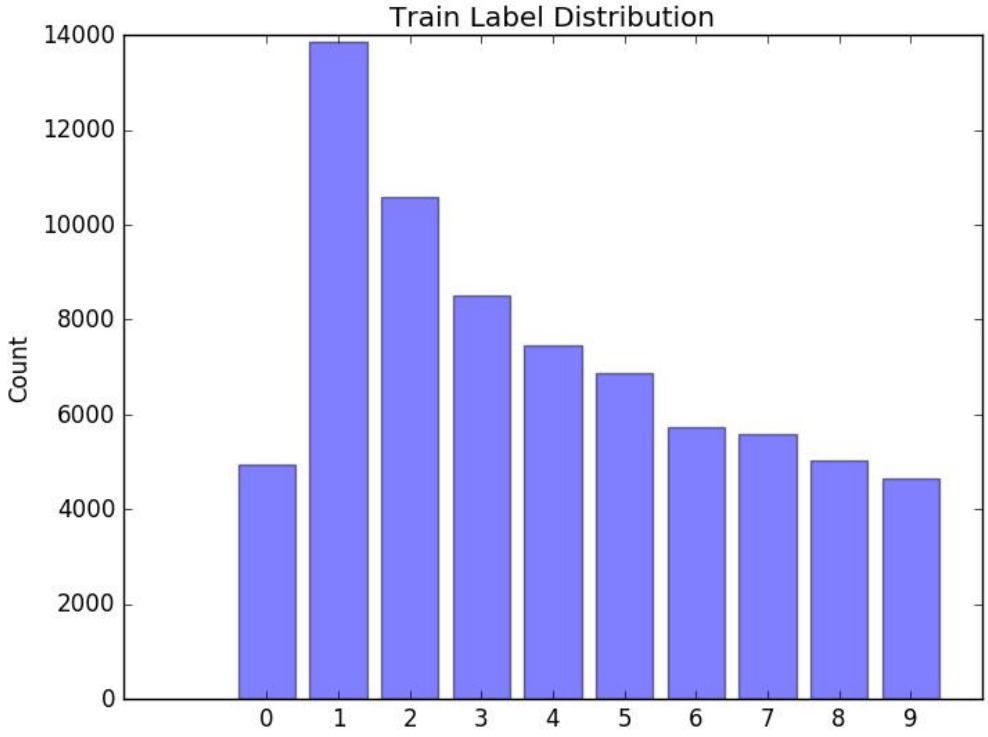
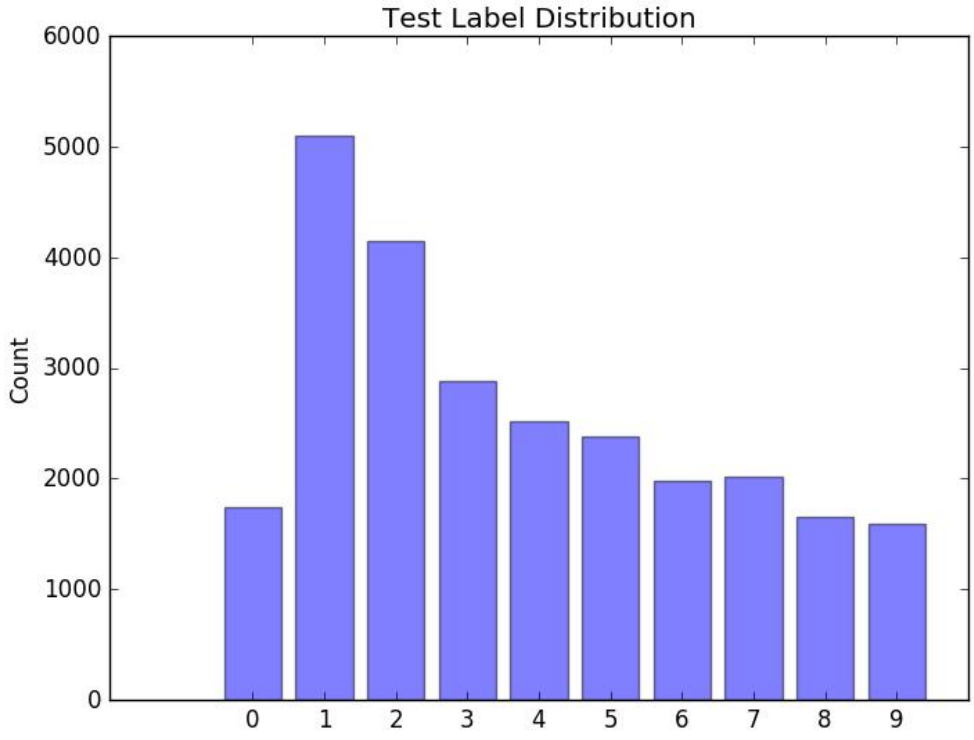


Figure 3, image_2_origin, label 3

Note that some images contains multiple digits. Right labels for such images are the digits closer to the center.

The image below is the distribution of train labels and test labels. As you can see, distributions in the training set and the testing set are similar.





c. Algorithms and Techniques

I applied three different architectures to the problem.

1) 2 Convolution + 2 Fully Connected

Conv -> Relu -> Max Pool -> Conv -> Reul -> Max Pool -> Dropout -> Fully Connected -> Fully Connected

2) 3 Convolution + 2 Fully Connected

Conv -> Relu -> Conv -> Relu -> Max Pool -> Conv -> Reul -> Max Pool -> Dropout -> Fully Connected -> Fully Connected

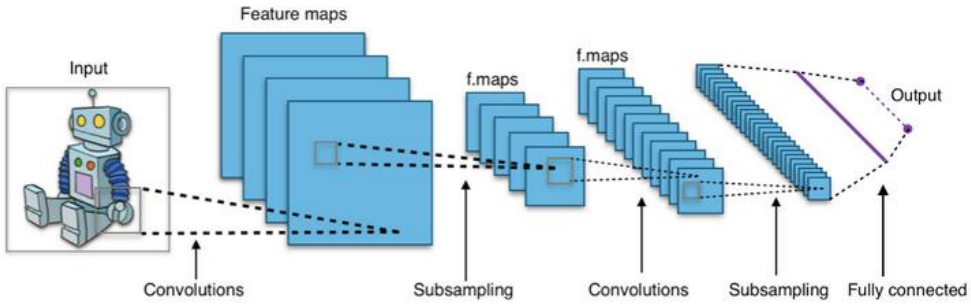
3) 4 Convolution + 2 Fully Connected

Conv -> Relu -> Conv -> Relu -> Max Pool -> Conv -> Reul -> Conv -> Relu -> Max Pool -> Dropout -> Fully Connected -> Fully Connected

Explanations of Different Layers:

Convolutional Layer:

Given an image with size N x N. A convolution is to slice through the image with a window size M x M, where M < N. In each slice, compute the matrix multiplication of the M x M cut of the image and a M x M weight matrix. All the outputs will form a new matrix and send to the next layer as the input.



One reason that convolution is good for image problems is that convolution preserves the local information of a image by the slicing window operation.

Relu:

A rectifier function function(X) = max(0, X)

This is a thresholding function that outputs all the positive inputs. If inputs are non-positive, outputs 0.

Max Pooling:

Max Pooling is just an image compression/down sizing step. If the stride is 2, then it means to downsize the image by a factor of 2. For example:

We have a single channel image

12, 34, 45, 03

08, 10, 52, 27

82, 99, 00, 43

66, 21, 02, 91

The max pooling function will only get the max value of every 2x2 cut of the original matrix. The output will be

34, 52

99, 91

Dropout:

Given a vector input X, randomly overwrite some values in X as 0 at a certain dropout rate. For example, if the dropout rate is 0.6:

<1, 3, 5, 7, 9> = dropout => <0, 3, 0, 0, 9>

Fully Connected Layer:

Normal neural network layer

Hyper Parameters are below:

1. Number of hidden nodes in fully connected layer
2. Depth of each convolutional layers
3. Patch size
4. Pooling stride
5. Drop out rate
6. Number of iterations to train
7. Base learning rate assuming exponential decay
8. Decay rate

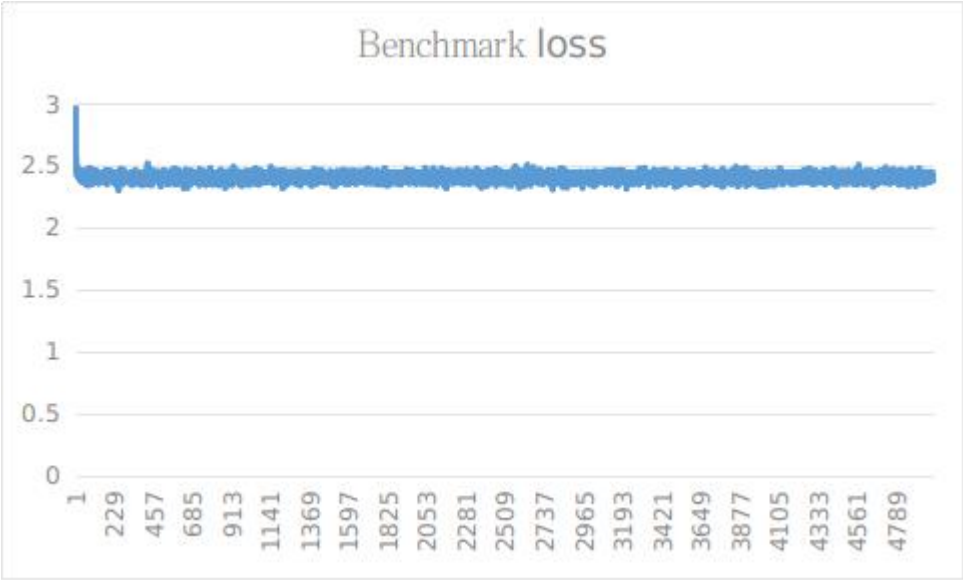
    1.

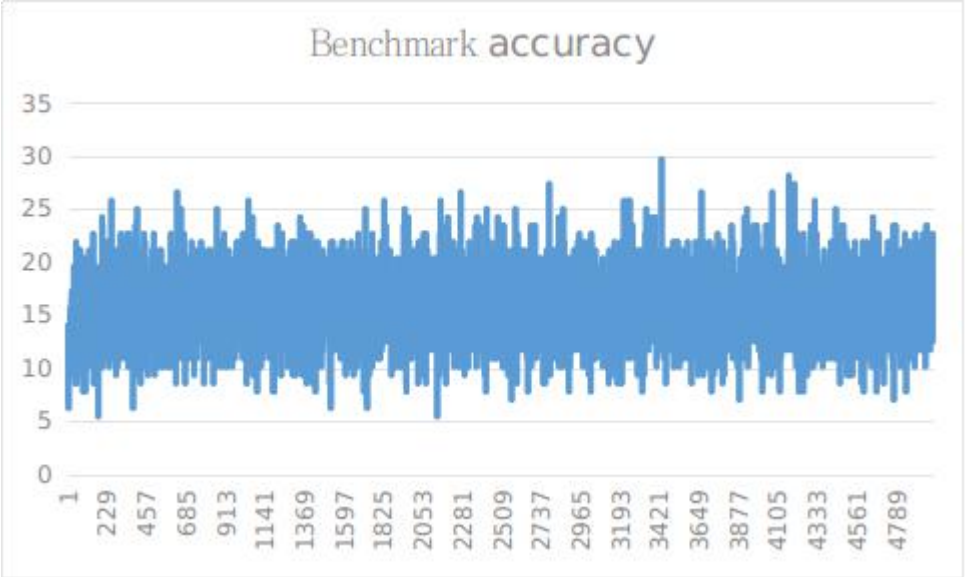
Number of iteration is not a hyper parameter of the network, but just a variable I use to control the training time. Batch size and number of iteration are complements. Basically I just set the batch size as big as memory capacity.

d. Benchmark

I started with the architecture number 1 mentioned in the Methodology section.
The initial parameter setting is:
Number of hidden nodes = 64
Batch_size = 128
Patch_size = 5
Convolutional layer depth = 16
Pooling Stride = 2
Dropout rate = 0.5
Base learning rate = 0.05
Decay rate = 0.95
Gradient Descent Optimizer

And the results are below:





As we can see, the model is useless. It converges prematurely.

The loss converges to 2.5 and the accuracy is 15% averagely.

III. Methodology

a. Data Preprocessing

Each pixel in the image has a range from 0 to 255. In order to let gradient descent work, first I normalized them into -1.0 to 1.0 as 32-bit float. This step was forgotten at first, and I got billions of losses in the training phase because gradient descent can't update big numbers effectively.

By applying a linear mapping from [0 ~ 255] to [-1.0 ~ 1.0], we obtain the same information but have nicer floating point to work with.

I also grayscaled all images to get a better memory performance. The results of grayscaled images and 3-channel images are very similar. Therefore grayscaling them is a wise choice.

b. Implementation

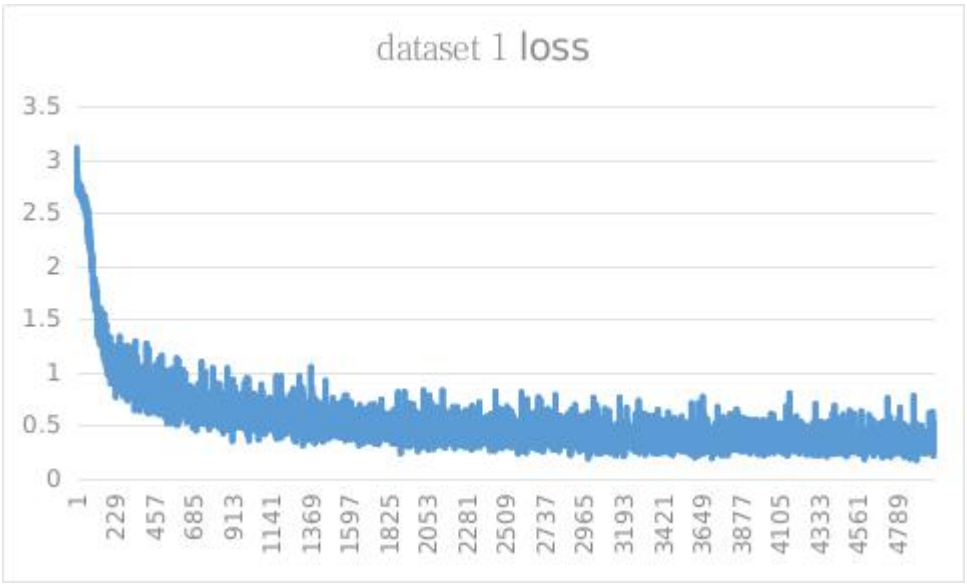
The strategy to solve this problem is an iterative approach. First to setup a basic model as the benchmark. Then iteratively scale out and scale up the model, which means to build a deeper model with more nodes in each layer. After finding a good size of the model, start to fine tuning hyper parameters.

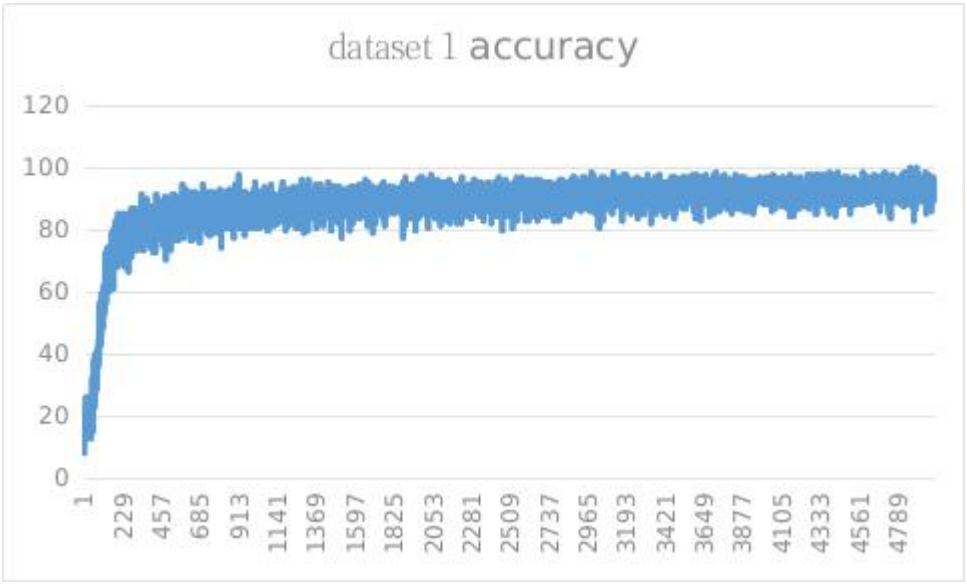
From a mathematical perspective, that is to try different loss functions, learning rate and many other parameters.

From a programming perspective, that requires to constantly re-factor code to adapt more advanced experiments.

c. Refinement

Through a lot of experiments and parameter tuning, I discovered that number of hidden nodes and convolutional depth have almost no influence. Patch size and dropout rate have more influence on loss and accuracy. Additionally, the choice of optimizer function, base learning rate and decay rate are vital to the result.



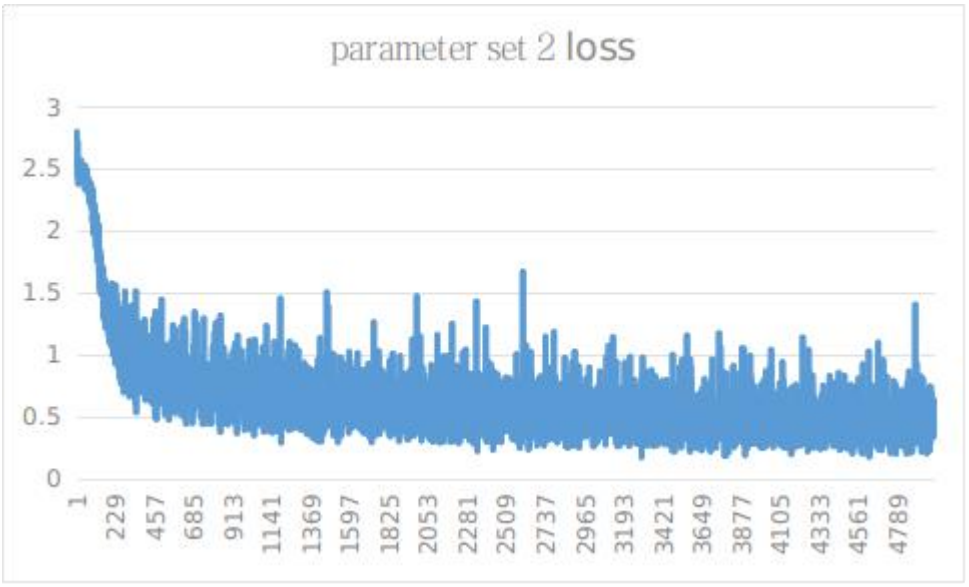


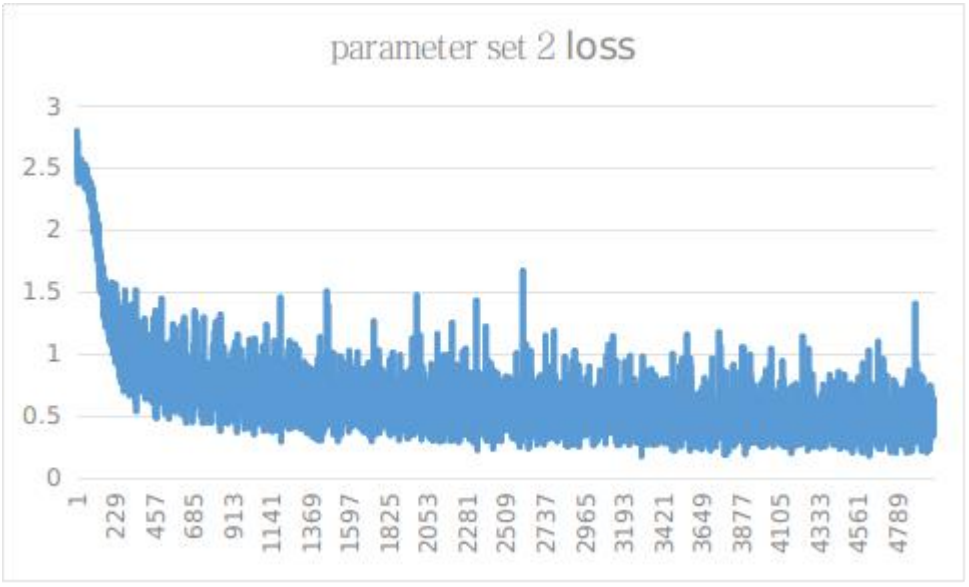
1. num_hidden=128
2. batch_size=128
3. patch_size=5
4. conv1_depth=32
5. conv2_depth=32
6. pooling_stride=2
7. drop_out_rate=0.9
8. base_learning_rate=0.0013
9. decay_rate=0.99

10. optimizer='adam'

This is the optimal configuration I have get so far. The loss converges to ~0.3 and the accuracy converges to ~93%.

Observation 1: Number of hidden nodes of fully connected layers has almost no influence.





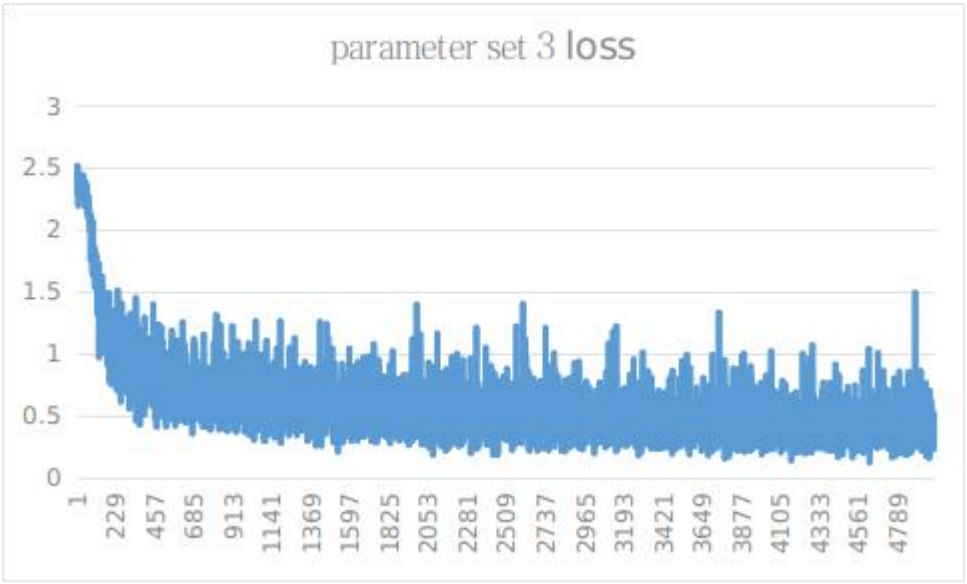
1. num_hidden=64
2. batch_size=64
3. patch_size=5
4. conv1_depth=32
5. conv2_depth=32
6. pooling_stride=2
7. drop_out_rate=0.9
8. base_learning_rate=0.0013
9. decay_rate=0.99
10. optimizer='adam'

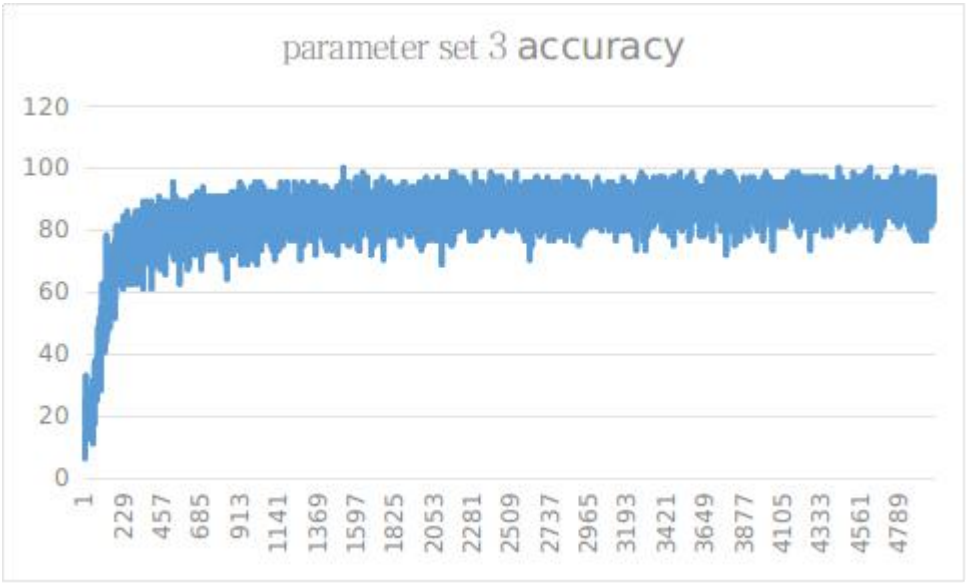
Here I changed num_hidden from 128 to 64. I also changed batch_size from 128 to 64, but this is purely for faster training.

We can see that the average loss and accuracy converges to almost the same value. The speed of convergence is also similar. Both parameter set 1 and parameter set 2 converges around 1000 iterations. One big difference is that both loss and accuracy oscillates much more, comparing to parameter set 1. This indicated that the convergence has a higher deviation.

My explanation is that a lower number of hidden layer nodes in the fully connected layer produce less final features to the softmax function, which gives softmax function more “pressure” to do the final decision(classification). However, because my convolutional layers have done a good feature extraction job, less final features in the fully connected layer is not a huge problem, as long as number of hidden nodes is not too small.

Observation 2: Convolutional depth has almost no influence.



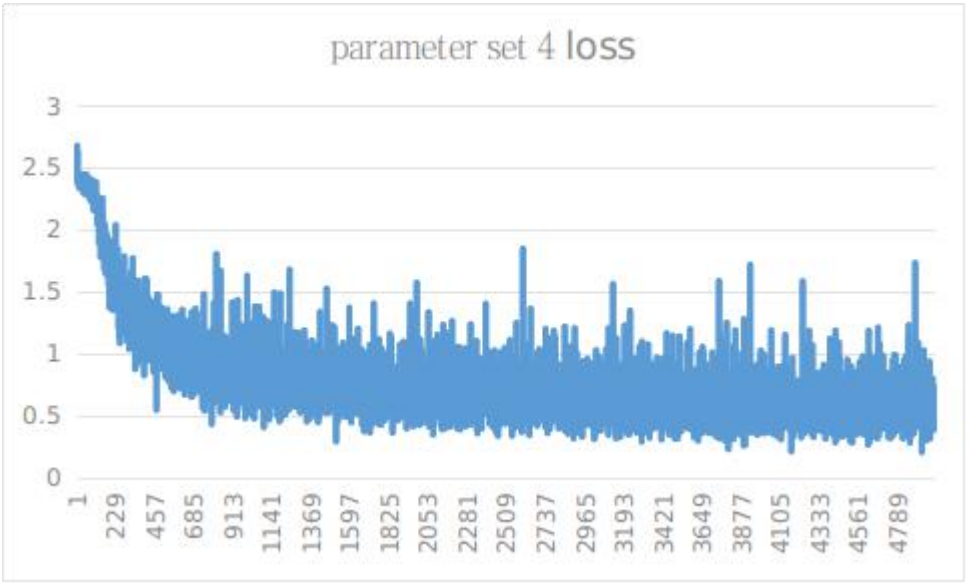


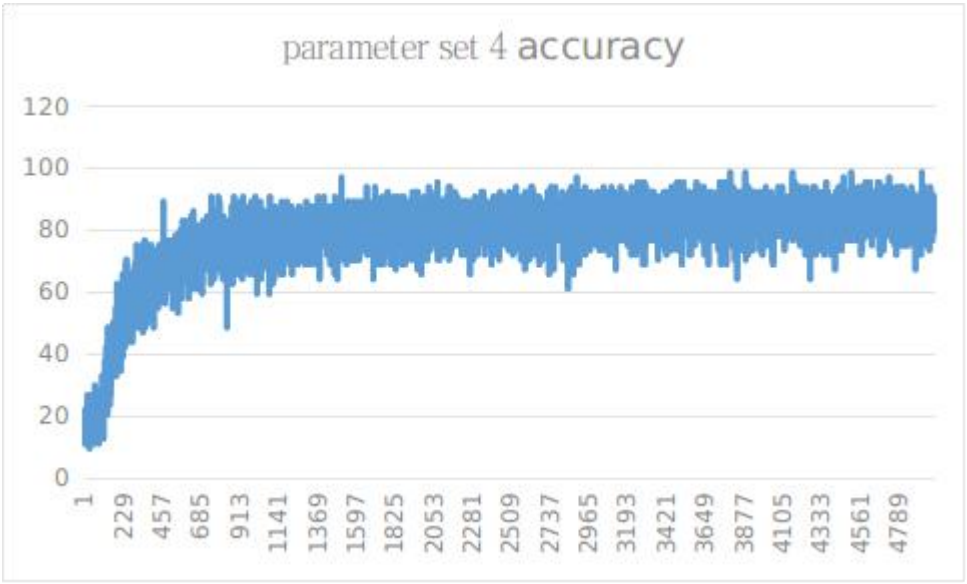
1. num_hidden=64
2. batch_size=64
3. patch_size=5
4. conv1_depth=16
5. conv2_depth=16
6. pooling_stride=2
7. drop_out_rate=0.9
8. base_learning_rate=0.0013
9. decay_rate=0.99
10. optimizer='adam'

Here I changed convolutional layers' depth from 32 to 16. However, the result is almost identical to parameter set 2's. Possibly, because the

problem domain is very narrow in turns of that inputs are just digit numbers, there are not many features to extra from the input, therefore, 16 depth is enough for this problem.

Observation 3: Drop out rate is essential for a good model.





1. num_hidden=64
2. batch_size=64
3. patch_size=5
4. conv1_depth=16
5. conv2_depth=16
6. pooling_stride=2
7. drop_out_rate=0.5
8. base_learning_rate=0.0013
9. decay_rate=0.99

`10.optimizer='adam'`

Here I changed drop out rate from 0.9 to 0.5. The loss converges to a higher value around 0.7 and the accuracy drops to around 80%. The speed of convergence becomes slower because we can see a more curve shape at the beginning.

One intuition is that drop out acts like a weak learner ensemble method in turns of that the final fully connected layer only gets the input from a subset of the whole network.

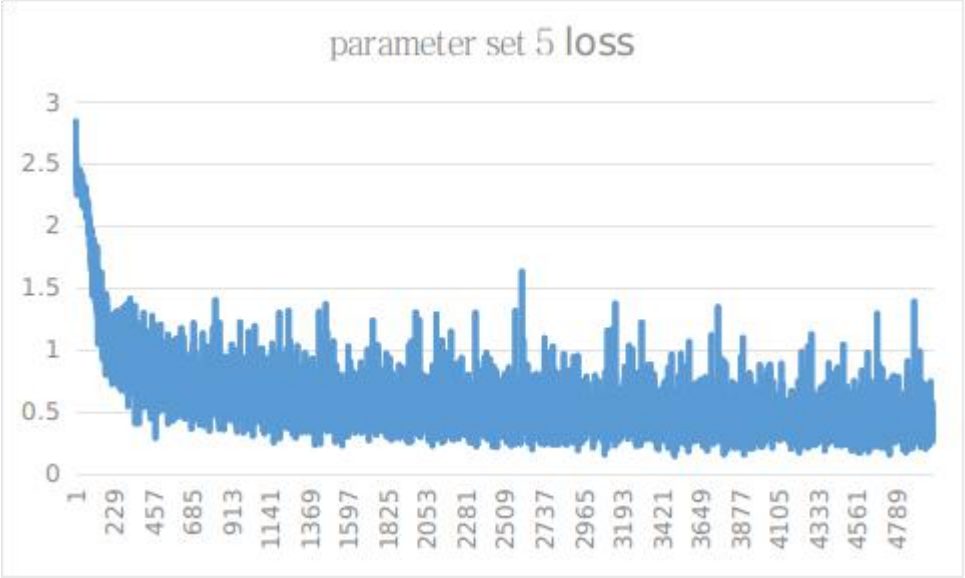
With a 0.5 dropout rate, half of the signal from the first fully connected layer to be randomly dropped. Therefore, at each training iteration, the second fully connected layer, which is the last layer in front of softmax function, can not rely on the inputs from fc1 totally.

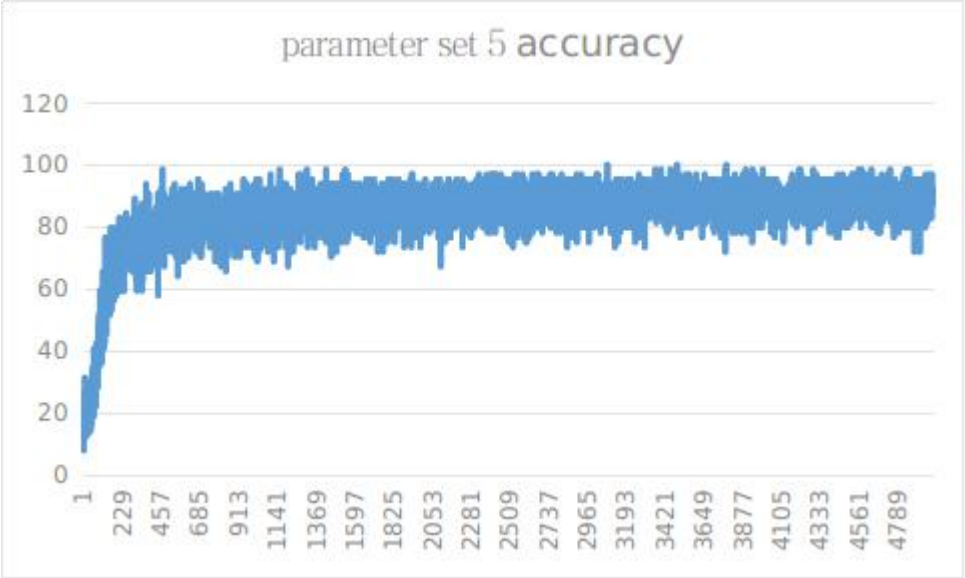
This effect is like to ask fc2 to make a decision based on less features. Additionally, each iteration will only have half gradients been back propagated through the network. The network learns more conservatively in some sense.

With a higher drop out rate such as 0.9, the network is much more conservatively. I am not saying that a high dropout rate is always better. But for the objectives of our problem, it is.

Also, in the testing, we don’t drop anything so that the fc layer can utilize all the features available.

Observation 4: Patch size has little influence as long as it doesn't change every much.



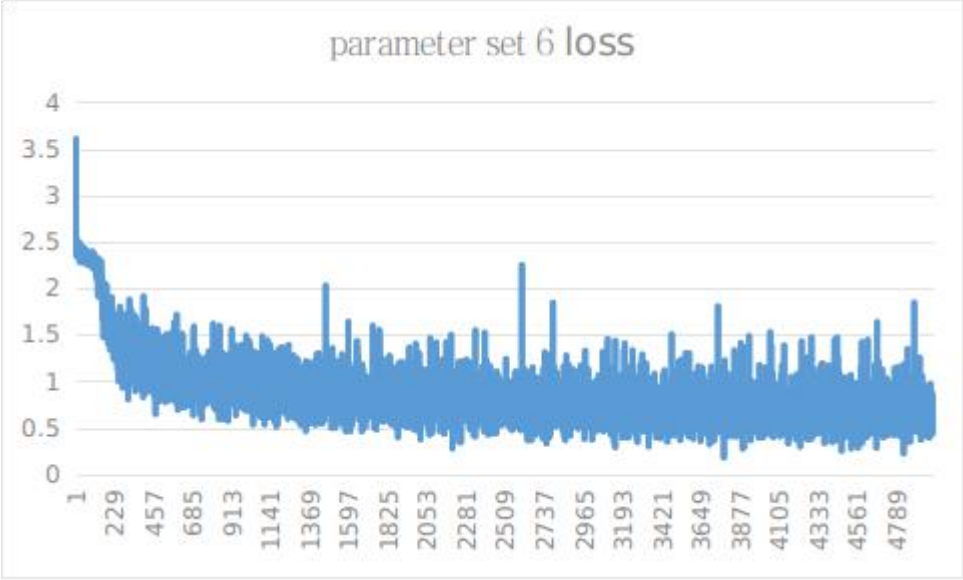


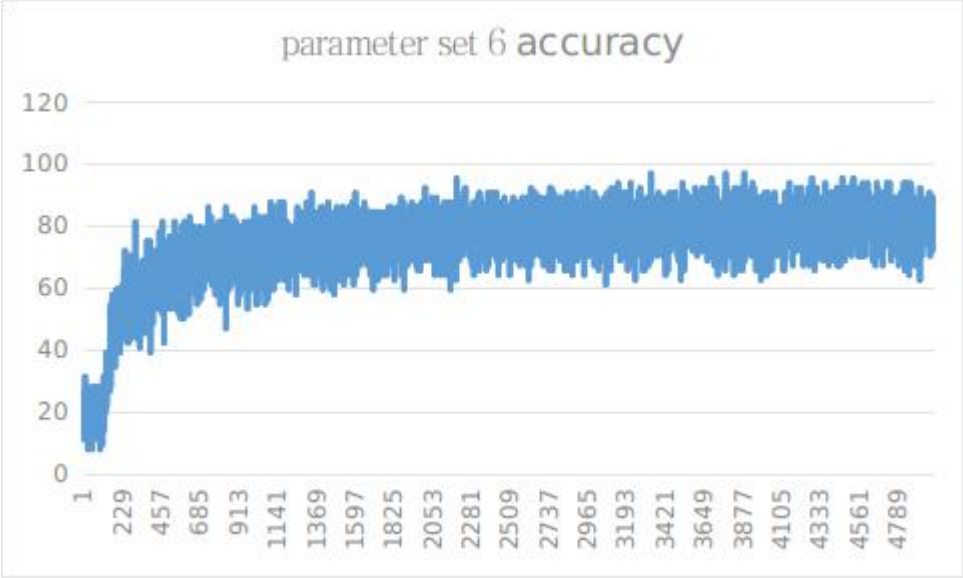
1. num_hidden=64
2. batch_size=64
3. patch_size=7
4. conv1_depth=16
5. conv2_depth=16
6. pooling_stride=2
7. drop_out_rate=0.9
8. base_learning_rate=0.0013
9. decay_rate=0.99
10. optimizer='adam'

Here the dropout rate is back to 0.9, but the patch size increases to 7 from 5. The idea here is that since a layer patch size compresses more/wider local information to a single point, the conluvotional layer will loss more locality. I expect a lower accuracy and high loss. However, I don't see much change.

If we take a look at the original images, we can see a some digits' font thickness takes about 1/5 of the image width. That is about 6 pixels. Therefore, 5 patch to 7 patch are every much the same. But, it is reasonable to assume that 5 patch and 15 patch would make a bigger difference.

Observation 5: Learning rate is very important





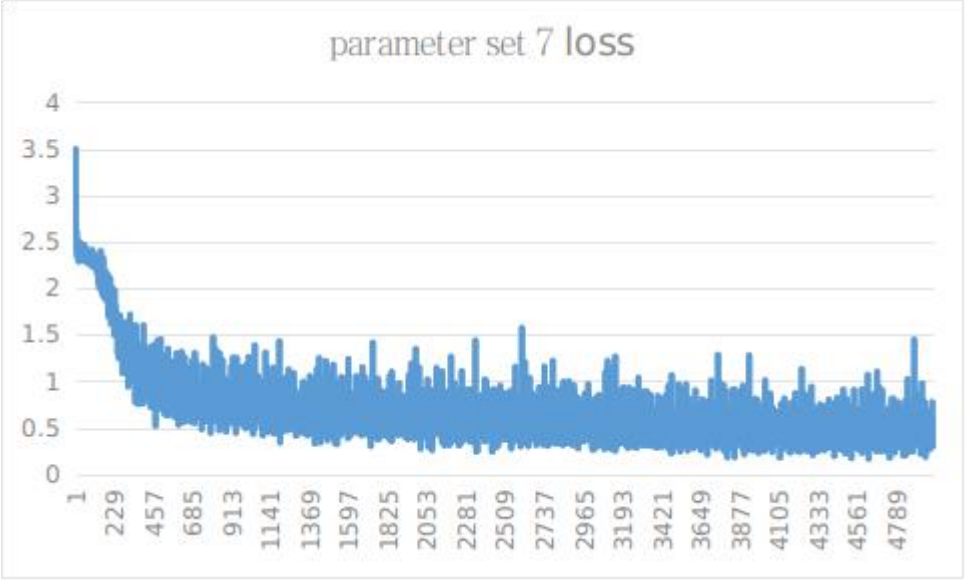
1. num_hidden=64
2. batch_size=64
3. patch_size=7

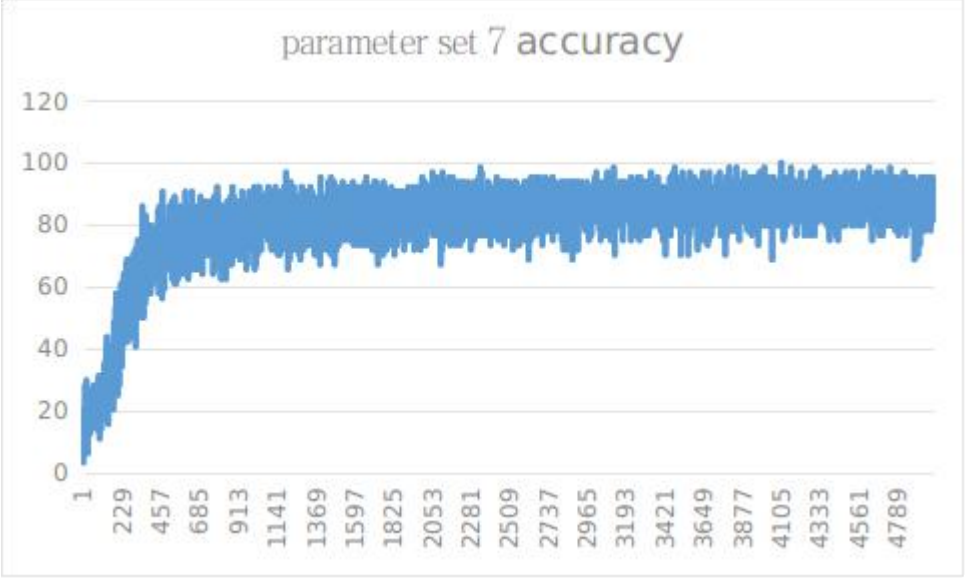
4.conv1_depth=16
5.conv2_depth=16
6.pooling_stride=2
7.drop_out_rate=0.9
8.base_learning_rate=0.005
9.decay_rate=0.99
10.optimizer='adam'

Note: Exponential decay is used in each iteration. Decay step is 100.

Here I change base learning rate to 0.005. We immediately see a worse result. The loss converges to around 0.8 and the accuracy converges below 80.

Then I want to see what happens if a super low base learning rate is applied.



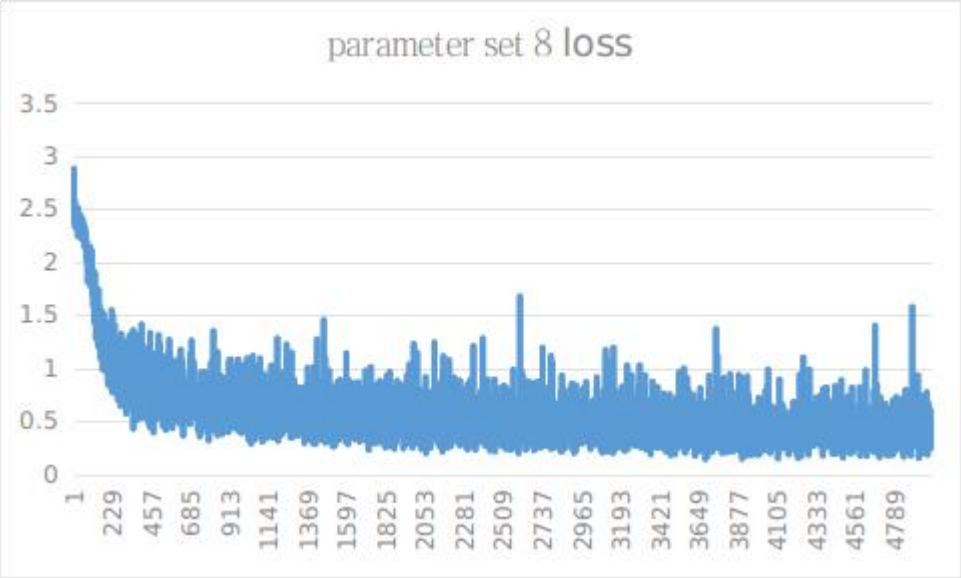


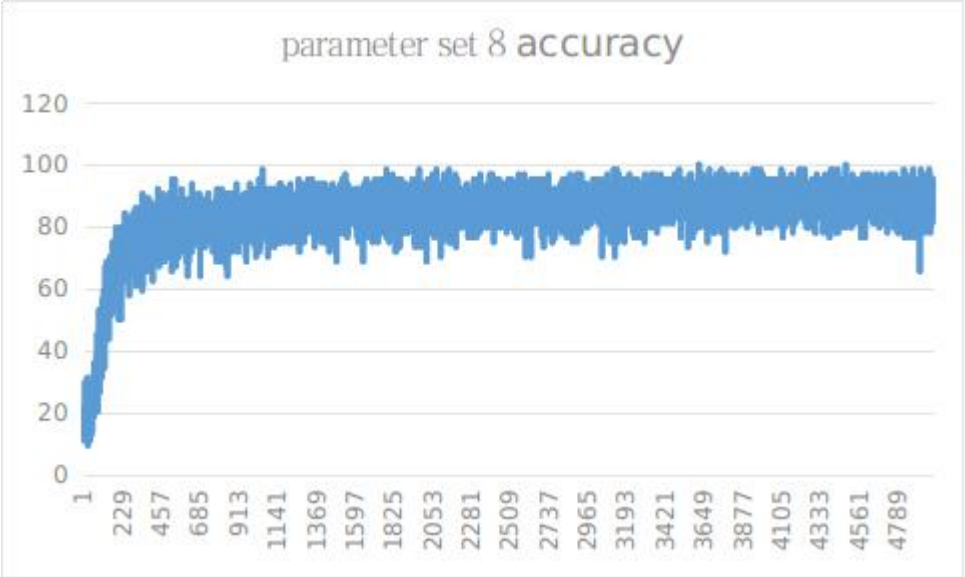
1. num_hidden=64
2. batch_size=64
3. patch_size=7
4. conv1_depth=16
5. conv2_depth=16
6. pooling_stride=2
7. drop_out_rate=0.9
8. base_learning_rate=0.0005
9. decay_rate=0.99
10. optimizer='adam'

The result becomes better, though it is still lower than optimal. But the oscillation is much smaller, which make sense because a smaller learning rate changes the weights much less.

Then after several times of trials, I found that 0.0013 is a good base learning rate.

Observation 6: Decay rate is not the most important

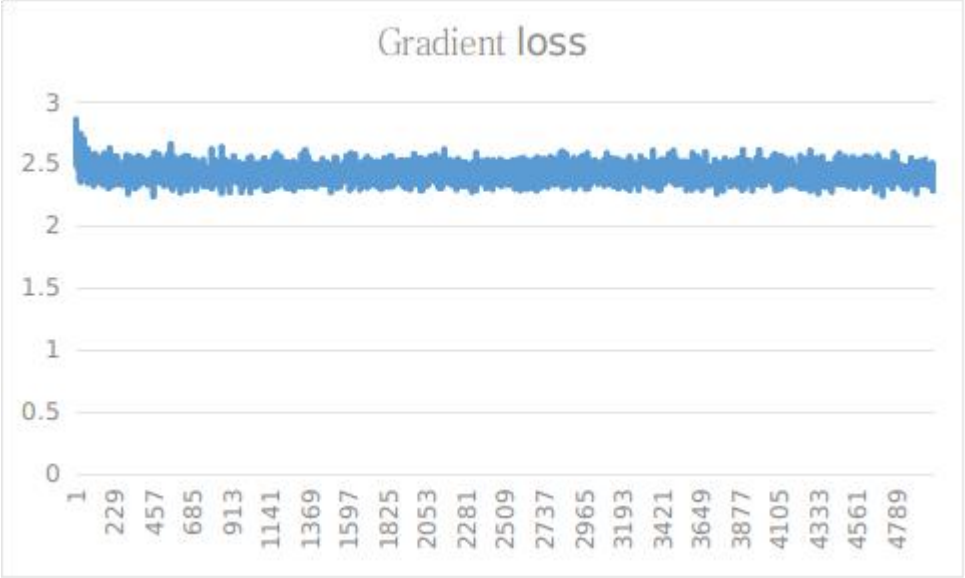


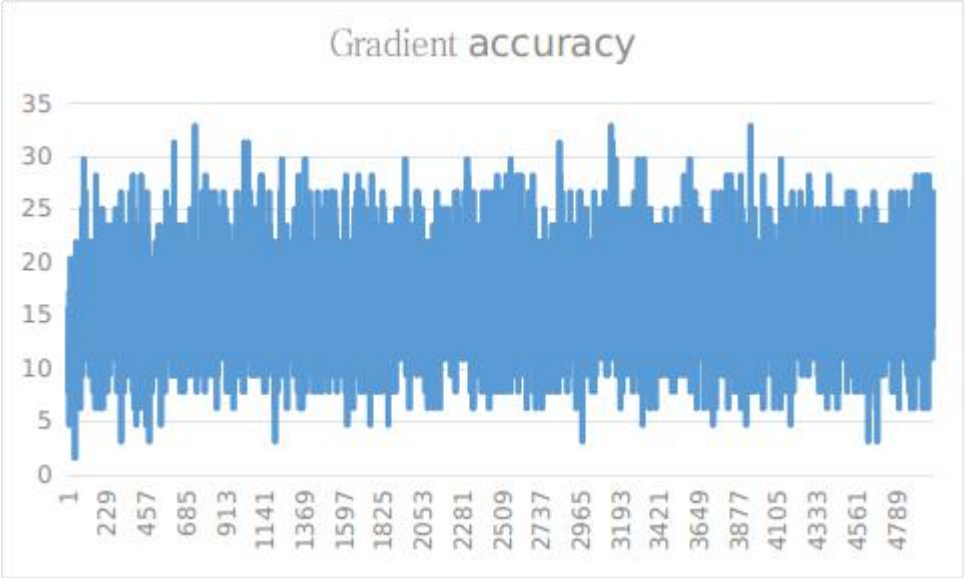


1. num_hidden=64
2. batch_size=64
3. patch_size=7
4. conv1_depth=16
5. conv2_depth=16
6. pooling_stride=2
7. drop_out_rate=0.9
8. base_learning_rate=0.0013
9. decay_rate=0.9
10. optimizer='adam'

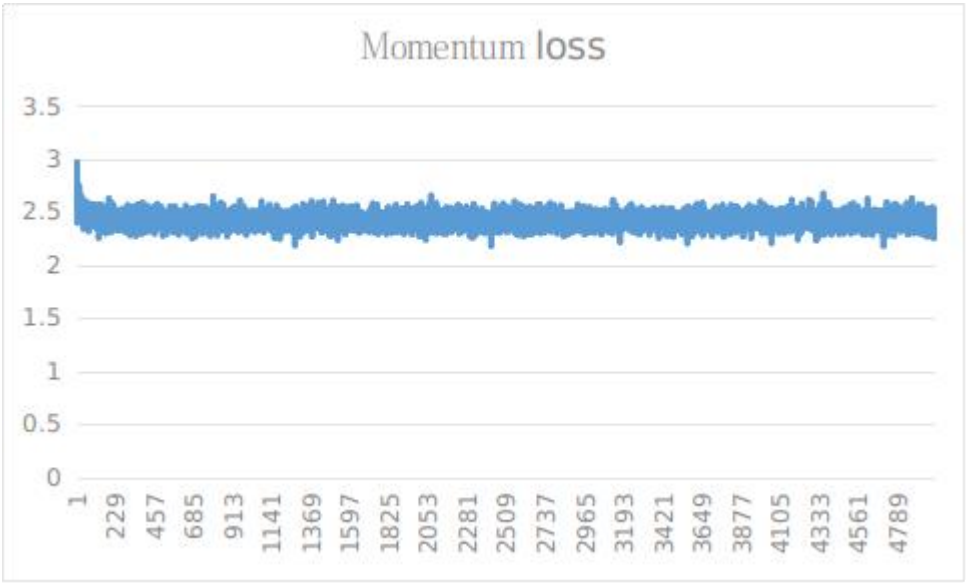
Based on parameter set 5, I changed decay rate to 0.9. The result is similar to parameter set 5. Intuitively, since the network converges very quickly, the decay rate doesn't have enough time to show its influence. If the network converges slower, then decay rate should be more observable.

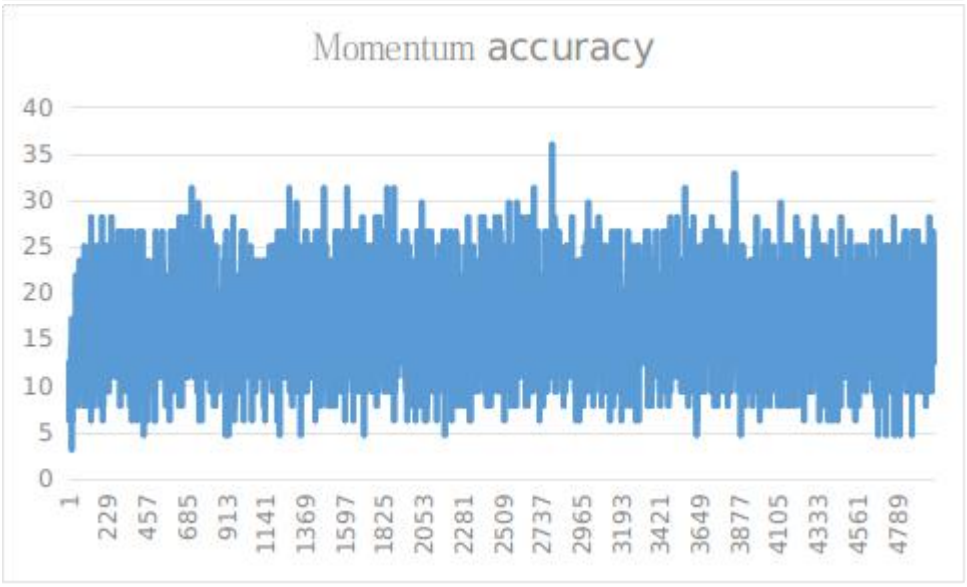
Observation 7: Loss function / Optimizer function is probably the most important factor.





The parameter set is optimal except for that Gradient Optimizer is used.

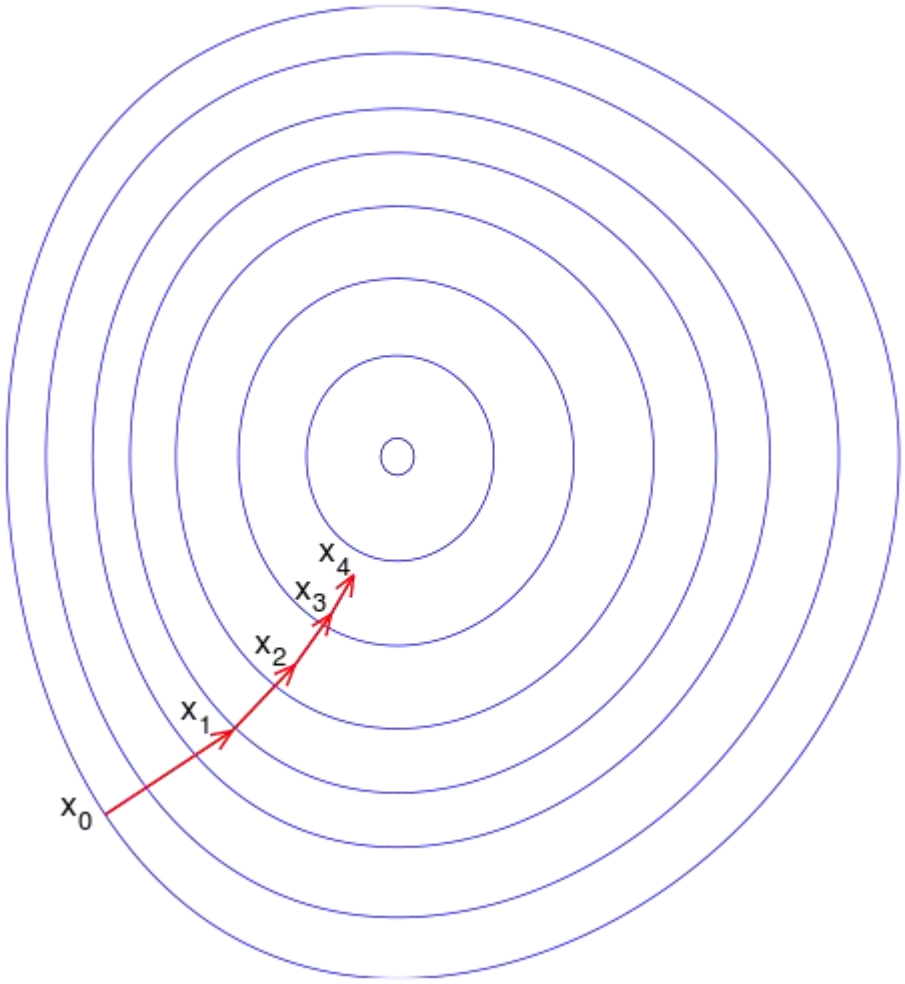




The parameter set is optimal except for that Momentum Optimizer is used.

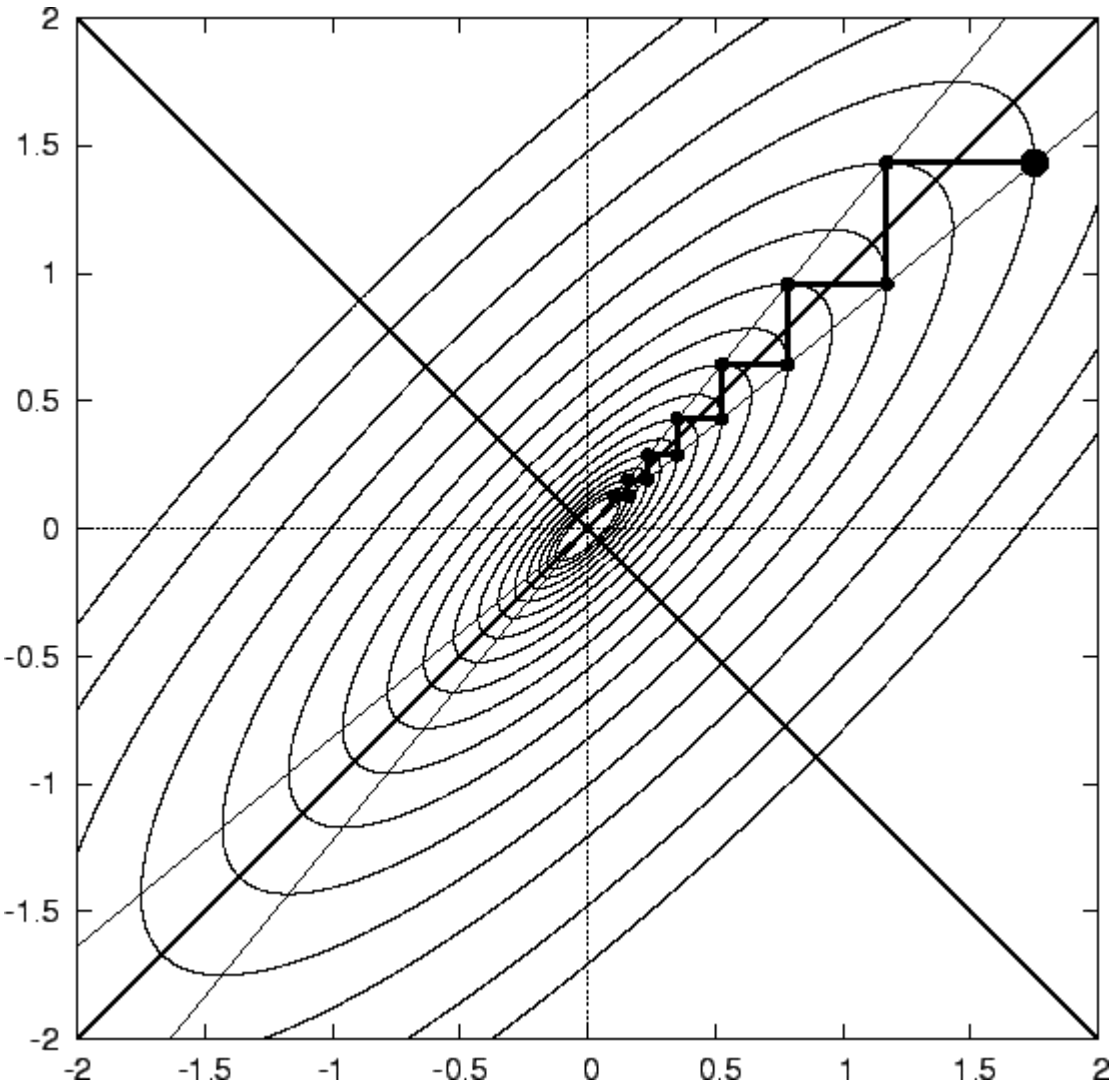
Normal gradient descent and momentum just fail.

Any optimization problem is just a searching problem in a gradient space. We try to find the minimum point.



Picture from Wikipedia https://en.wikipedia.org/wiki/Gradient_descent

Above picture is just a demonstration in a 2-dimensional space. In reality, we are searching in multi-dimensional space. Because learning rate is applied to every dimension, if a dimension has much more slope than another dimension, then the simple gradient descent will just step more in the more slope dimension, as the next picture demonstrate.



Picture from http://komarix.org/ac/papers/thesis/thesis_html/node10.html

This leads to unnecessary oscillation and if we are not lucky, we may not even reach the optimal point.

This is where momentum method comes from. If we image the search space as a physical space, we can apply a resistance to counter the slope. Let's take a look at the formula.

```
# Momentum update
v = mu * v - learning_rate * dx # integrate velocity
x += v # integrate position
```

Formulate from http://cs231n.github.io/neural-networks-3/ Note: every variable is a vector

Here we have another momentum * velocity term against the regular gradient descent.

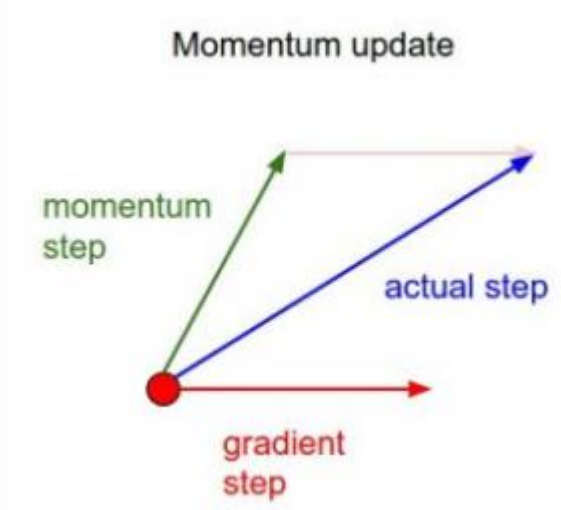


Image from http://cs231n.github.io/neural-networks-3/

Therefore, the update wound be strongly influenced only by gradient descent anymore. The network belongs a more “careful” learner in some sense.

However, if momentum is good, why we also get terrible results from the momentum optimizer?

Let’s take a look at the simplified Adam update.

```
m = beta1*m + (1-beta1)*dx
v = beta2*v + (1-beta2)*(dx**2)
x += - learning_rate * m / (np.sqrt(v) + eps)
```

Formulate from http://cs231n.github.io/neural-networks-3/

Adam takes in the first order momentum and the second order(gradient of gradient) momentum into count. Basically, we still want the momentum to influence the update, but additionally, divided by second order. Intuitively speaking, if first time network updates too greedy, then next time it will be more conservative. If next time it is too conservative, then next next time it will try to be more greedy. After a while it can dynamically find a right ‘rhythm’. Therefore, it is much smarter than simple momentum updates.

Actually, I also tried Adam update without exponential decay. It worked very well too.

Why does deeper network help little?

One interesting observation is that deeper network doesn’t improve the result very much, possibly because 2 convolutional layers are enough to catch all the information in the input since the input are not complex.

With the optimal parameters, performances are below:

Accuracy in Testing

Network 1 (2 convolutions): 89.89%

Network 2 (3 convolutions): 90.08%

Network 3 (4 convolutions): 92.17%

IV. Result

a. Model Evaluation and Validation

The final accuracy is 92.17%. The confusion matrix is:

	0	1	2	3	4	5	6	7	8	9	r
0	1632	29	10	5	2	5	25	5	16	12	0.94
1	32	4850	53	29	21	13	12	66	13	4	0.95
2	12	54	3942	46	15	9	6	40	6	15	0.95
3	11	71	54	2547	6	45	18	15	36	74	0.89
4	18	133	21	19	2268	4	17	11	13	16	0.9
5	7	19	17	65	9	2164	67	5	17	12	0.91
6	59	24	9	15	3	24	1785	6	44	5	0.90
7	5	67	36	10	4	4	0	1878	2	10	0.93
8	30	17	17	29	2	8	60	3	1471	21	0.89
9	48	19	45	15	7	6	2	9	17	1427	0.89
a	0.88	0.92	0.94	0.92	0.97	0.95	0.90	0.92	0.90	0.89	

Recall is slightly better than accuracy.

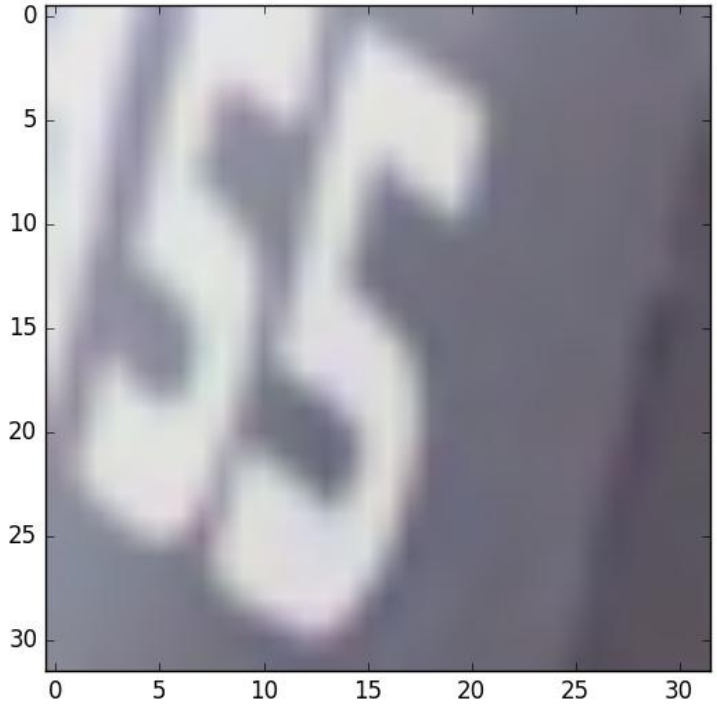
b. Justification

Comparing to the benchmark, this is a huge improvement. The testing result is about 1% lower than the training result. There should be no overfitting.

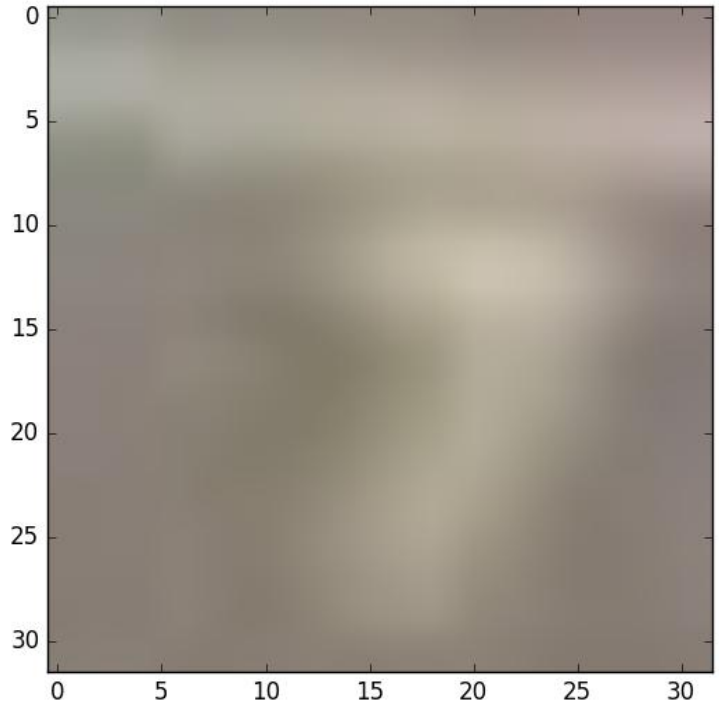
V. Conclusion

a. Free-Form Visualization

Let’s see 2 predictions



The label is 5 and the prediction is also 5.



The label is 8 but the prediction is 3. However, this one is not recognizable by human.

b. Reflection

Understanding the mathematical properties of convolutional neural network and the API usages of TensorFlow is the two most essential parts of the success of the project.

At first I didn’t really understand the mathematical meanings of different layers. Therefore, I can’t understand the sample code from TensorFlow website. I directly took the sample and tried to apply it on this problem. This approach didn’t work. After several days of study, I finally understood the basic building blocks of a convolutional neural network. Then I was able to customize the sample code into my needs.

Because of the sample code was just a proof of concept. I had to re-factor most of it to fit my needs. The documentation saved me a lot of time. The most important concept to understand is the TensorFlow is a low level computation framework. It doesn’t do machine learning by itself. But, by defining a computational graph, machine learning algorithm can be easily implemented.

c. Improvement

There are mainly 3 aspects to improve.

1. Better Code Structure: Configuration vs. Convention
Right now, when I try to modify the computational graph, I still need to modify the source code. A better API should be exposed. I should organized my code in a manner that providing a configurable API for me to run a set of experiments easily.\

2. More pre-processing
Sometimes training data are limited. For images, it is a good idea to create more data by applying rotation, blur, and other image processing algorithms on original data. Not only doing this will produce more training data, but also help the model to fight against noises and to prevent overfitting.

3. Regression: Locate Images
Be able to locate where each digit is in the image. This can be achieved by upgrade our model to a regression model. The position of the digit is the value that model regresses against.