Report of Deep Learning Capstone Project

Xuanzhe Wang

Definition

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

The data set consists of 73257 training images and 26032 testing images. Each image is a 32x32 bitmap(array) with 3 color channels.

Methodology

1. 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.

Let’s have a visual impression of several original and normalized images.

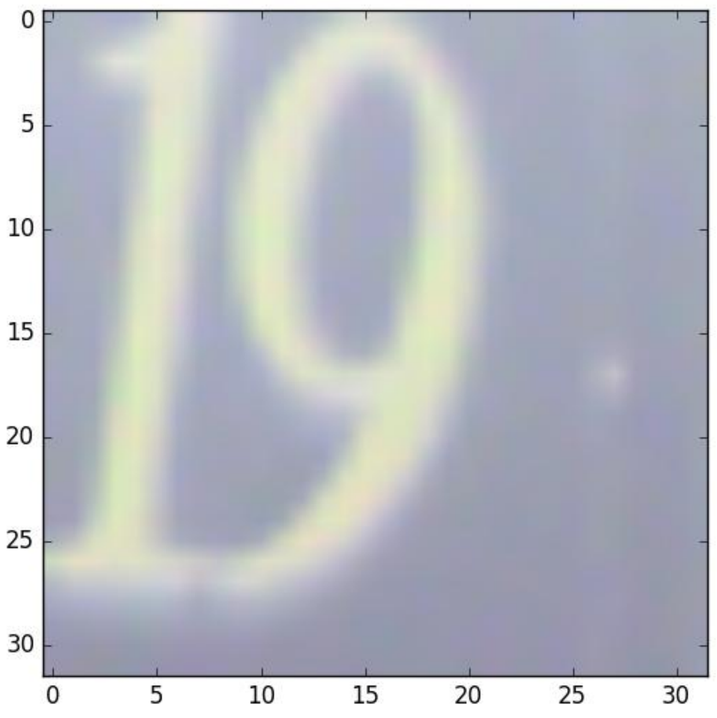


Figure 1, image_1_origin, label 9

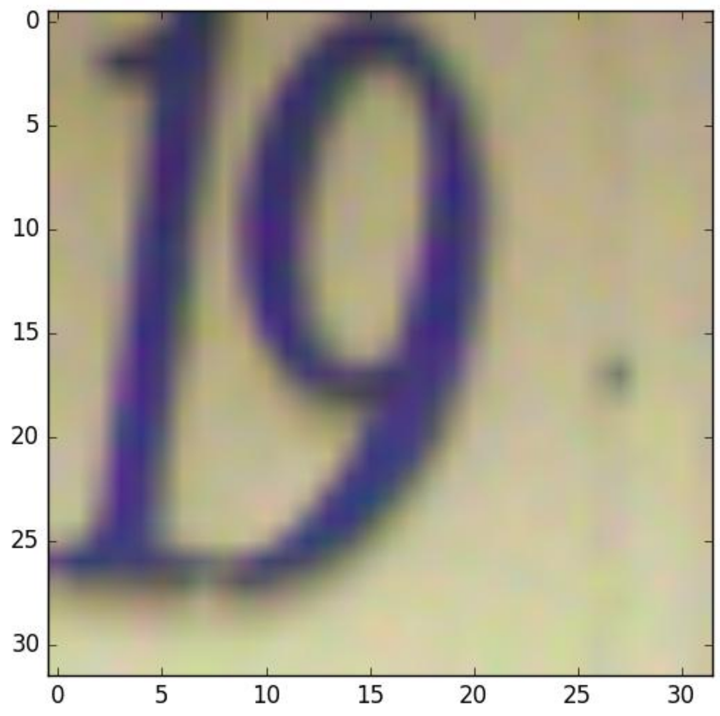


Figure 2, image_1_normal, label 9

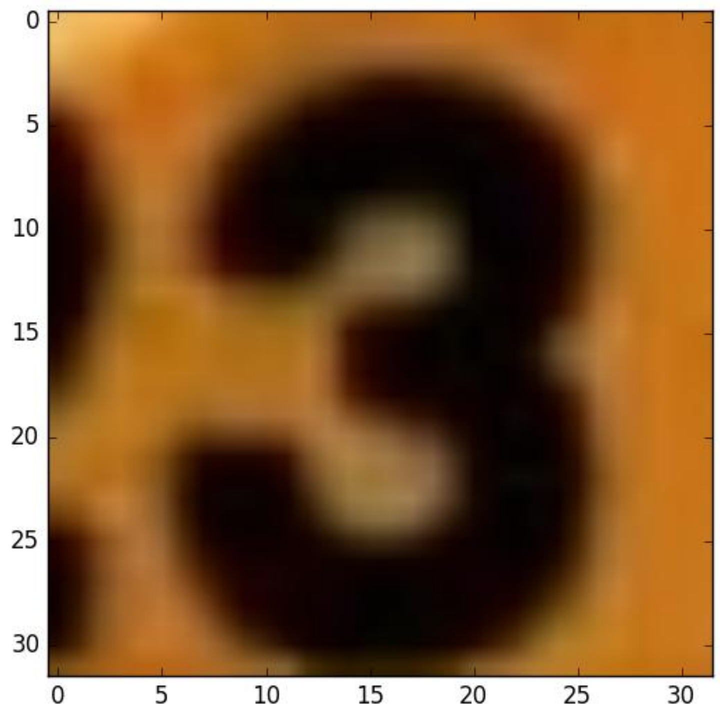


Figure 3, image_2_origin, label 3

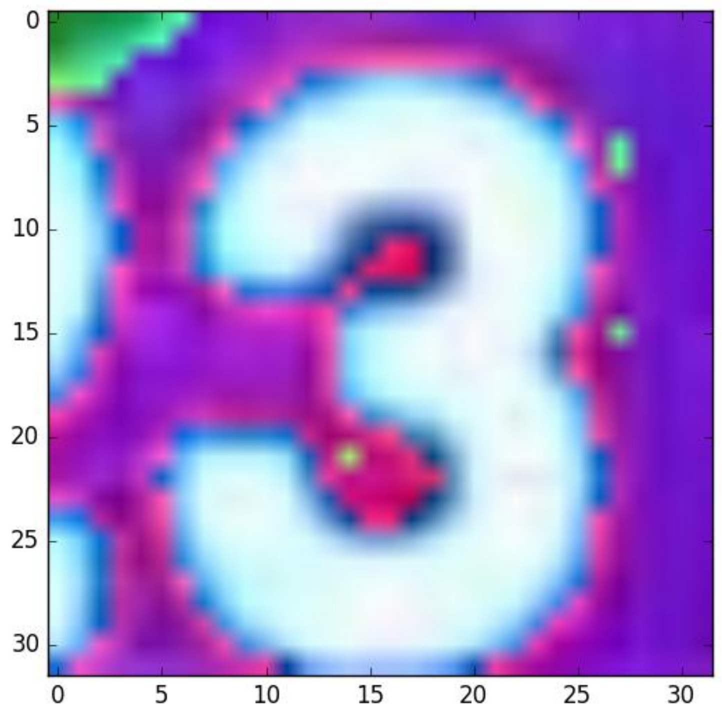


Figure 4, image_2_normal, label 3

As we can see, the normalized images have a stronger contrast. But t his virtual impression should not be considered as numeric evidence si nce images were drawn by matplotlib, the python library. The point is t hat by applying a linear mapping from [0 ~ 255] to [-1.0 ~ 1.0], we o btain the same information but have nicer floating point to work with. I will explain the meaning of "nicer" later.

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

2. Three Graph Architectures

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

Hyper Parameters are below:

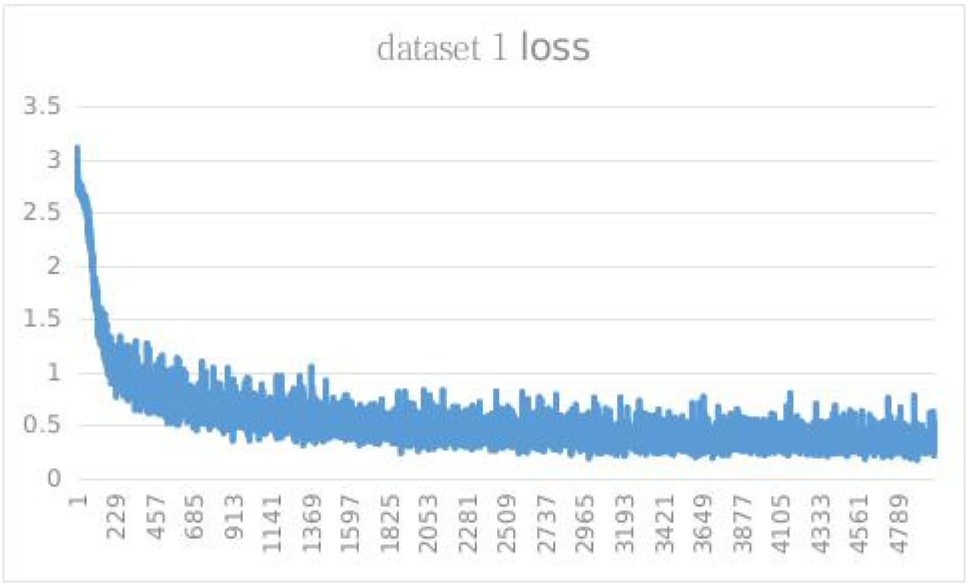
1. Number of hidden nodes in fully connected layer
2. Depth of each convolutional layers
3. Patch size
4. Pooling stride
5. Drop out rate
6. Number of iterations to train
7. Base learning rate assuming exponential decay
8. Decay rate

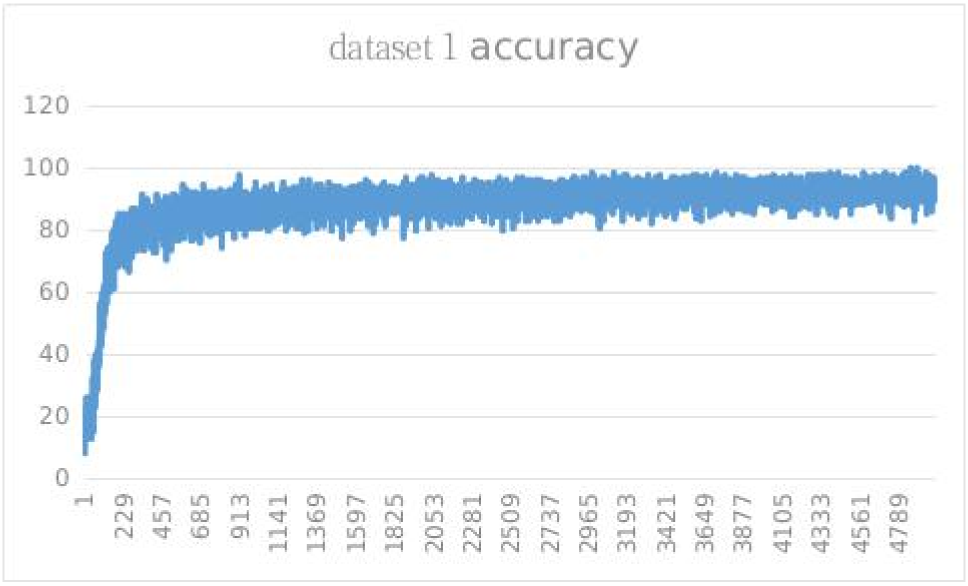
Number of iteration is not a hyper parameter of the network, but just a variable I use to control the training time. Batch size and number of iteration are complements. Basically I just set the batch size as big as memory capacity.

Analysis

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.

Let's look at training data.



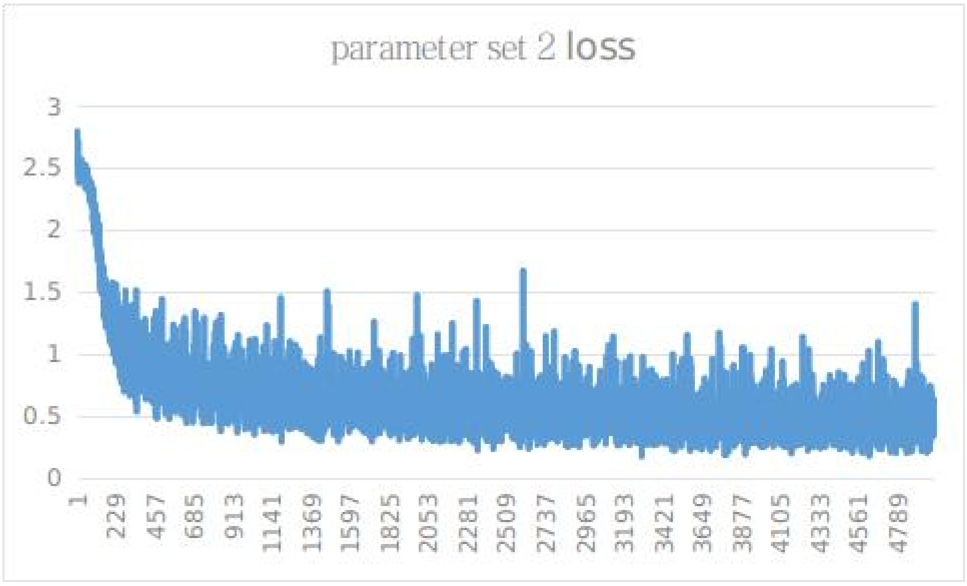


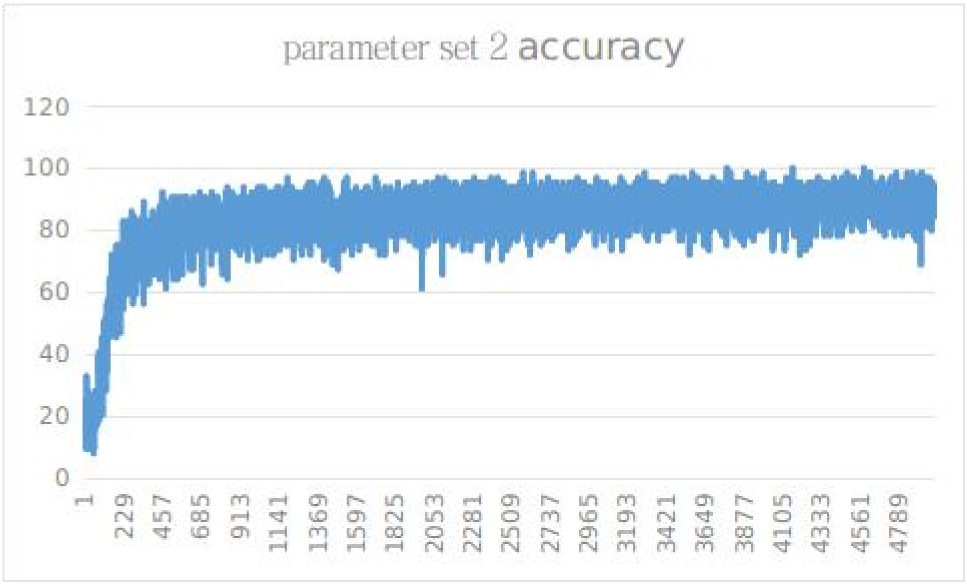
1. num_hidden=128
2. batch_size=128
3. patch_size=5
4. conv1_depth=32

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

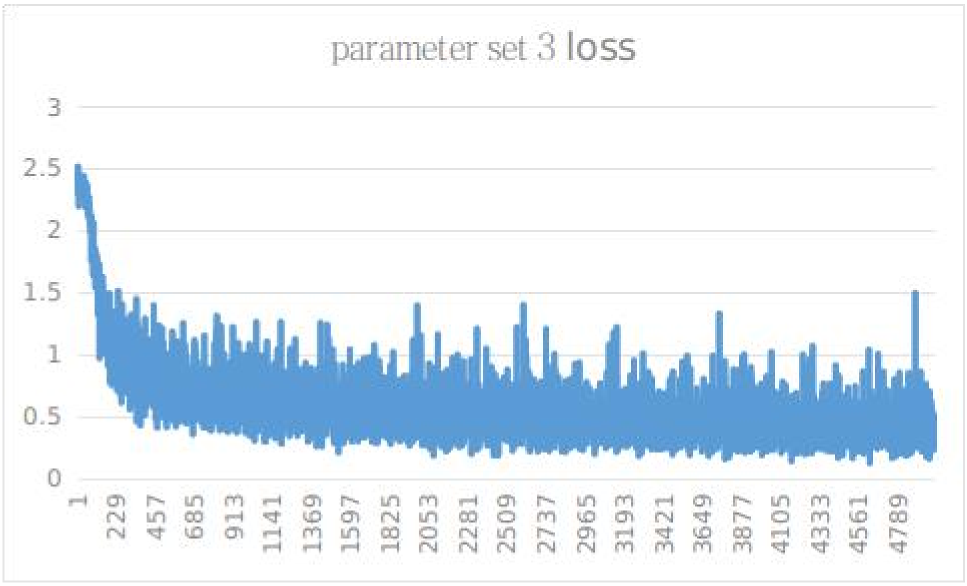
3. patch_size=5
4. conv1_depth=32
5. conv2_depth=32
6. pooling_stride=2
7. drop_out_rate=0.9
8. base_learning_rate=0.0013
9. decay_rate=0.99
10. optimizer='adam'

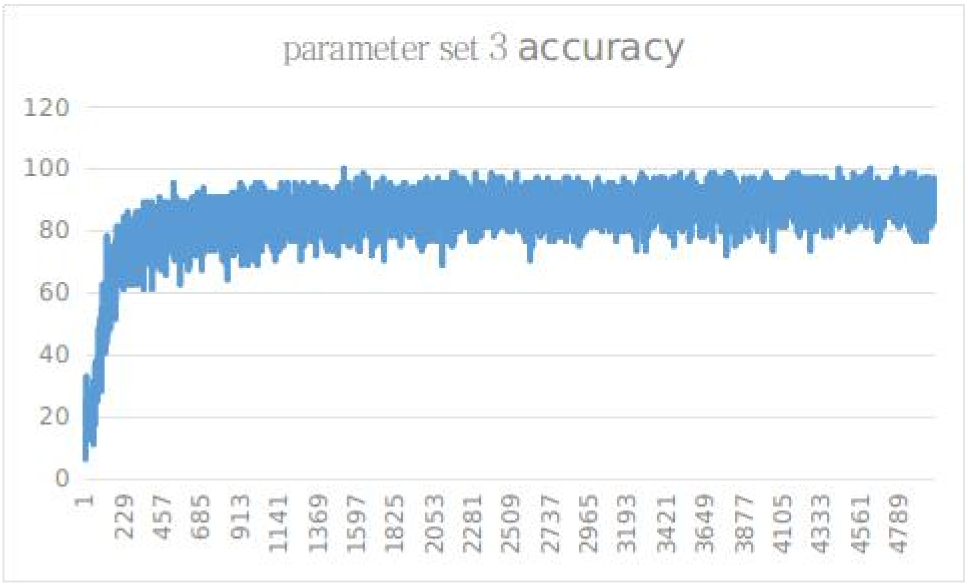
Here I changed num_hidden from 128 to 64. I also changed batch_size from 128 to 64, but this is purely for faster training.

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.

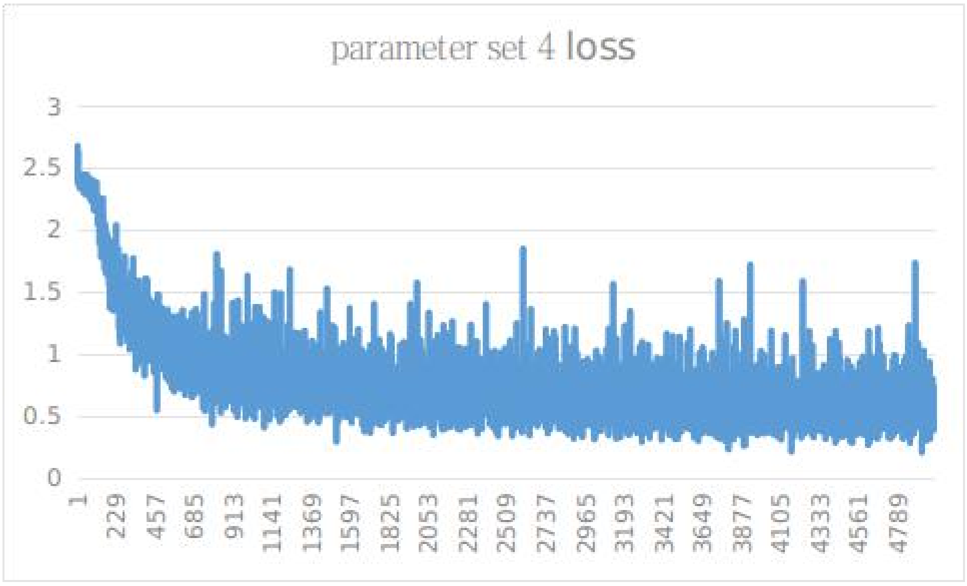


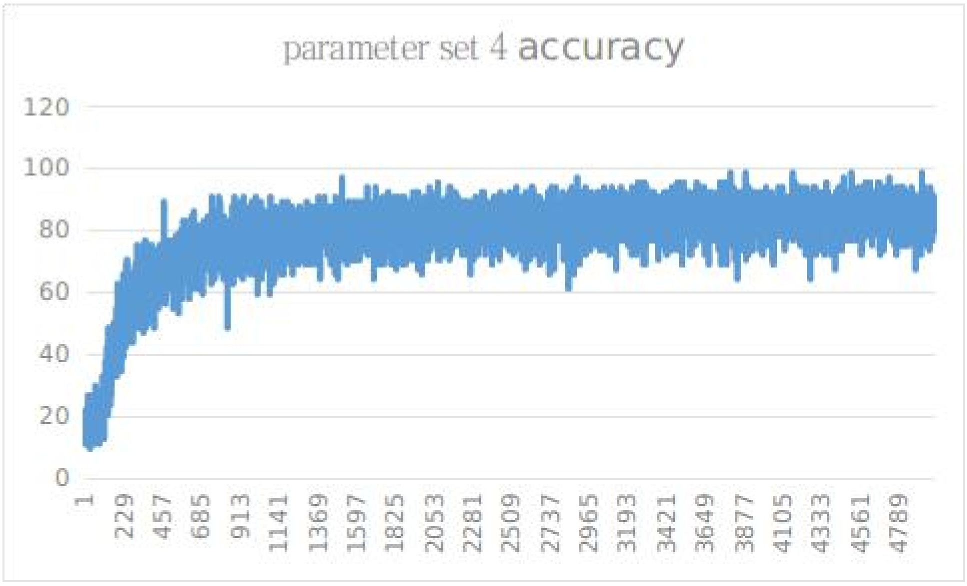


1.num_hidden=64
2.batch_size=64
3.patch_size=5
4.conv1_depth=16
5.conv2_depth=16
6.pooling_stride=2
7.drop_out_rate=0.9
8.base_learning_rate=0.0013
9.decay_rate=0.99
10.optimizer='adam'

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.

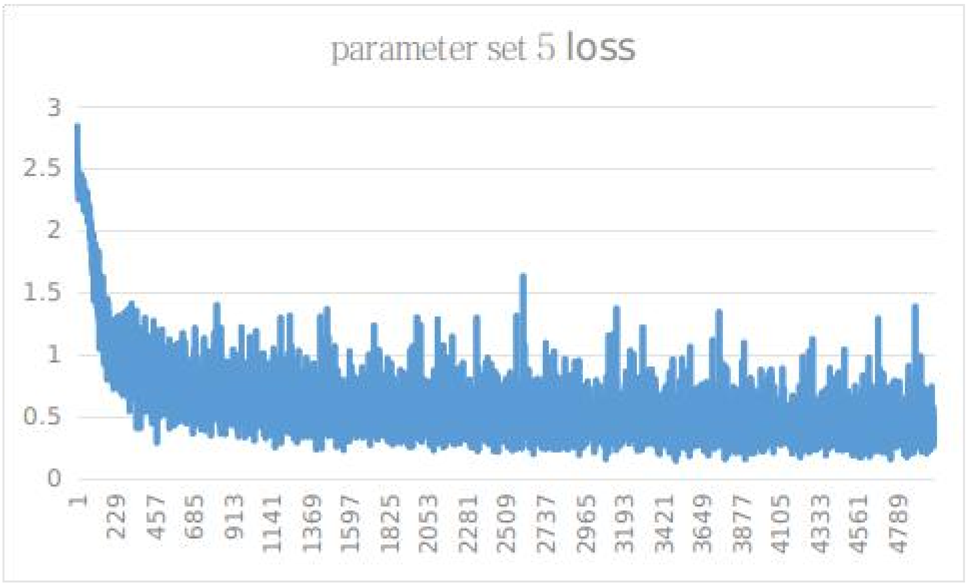
With a 0.5 dropout rate, half of the signal from the first fully connected layer to be randomly dropped. Therefore, at each training iteration, the second fully connected layer, which is the last layer in front of softmax function, can not rely on the inputs from fc1 totally.

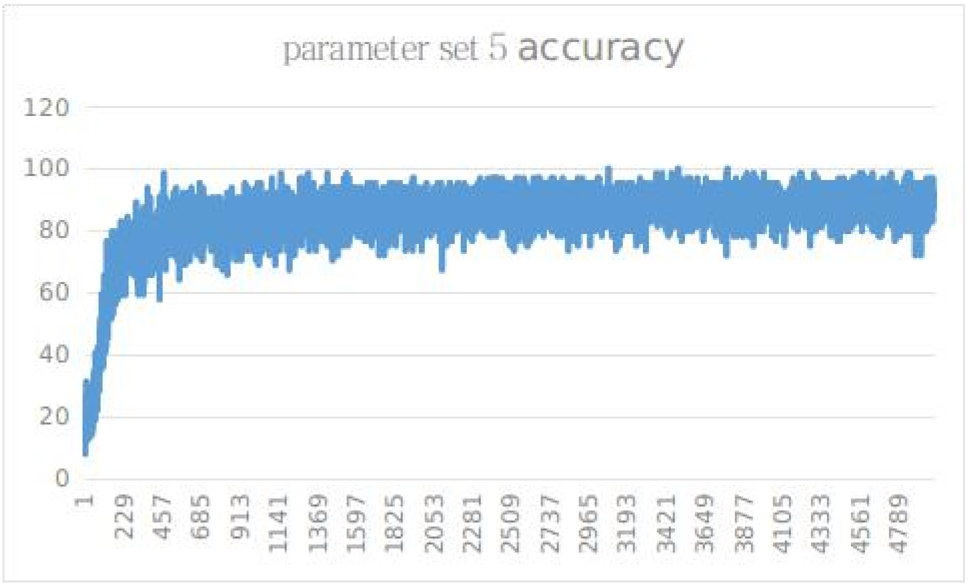
This effect is like to ask fc2 to make a decision based on less features. Additionally, each iteration will only have half gradients been back propagated through the network. The network learns more conservatively in some sense.

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.



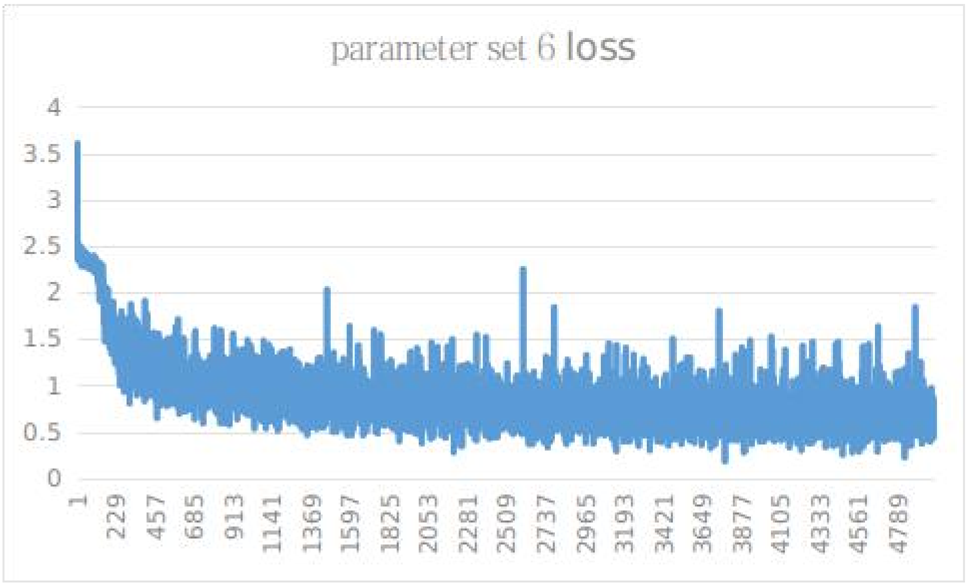


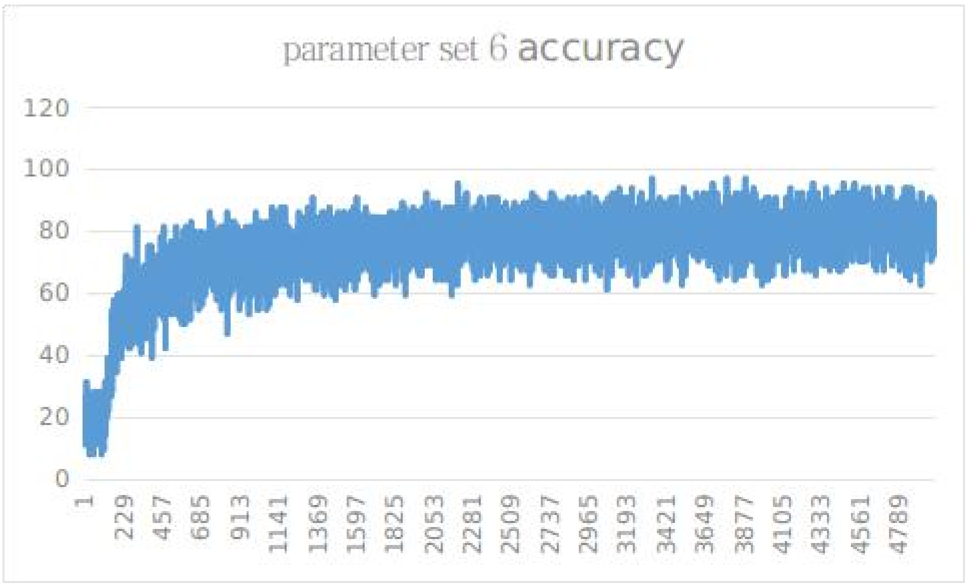
1.num_hidden=64
2.batch_size=64
3.patch_size=7
4.conv1_depth=16
5.conv2_depth=16
6.pooling_stride=2
7.drop_out_rate=0.9
8.base_learning_rate=0.0013
9.decay_rate=0.99
10.optimizer='adam'

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



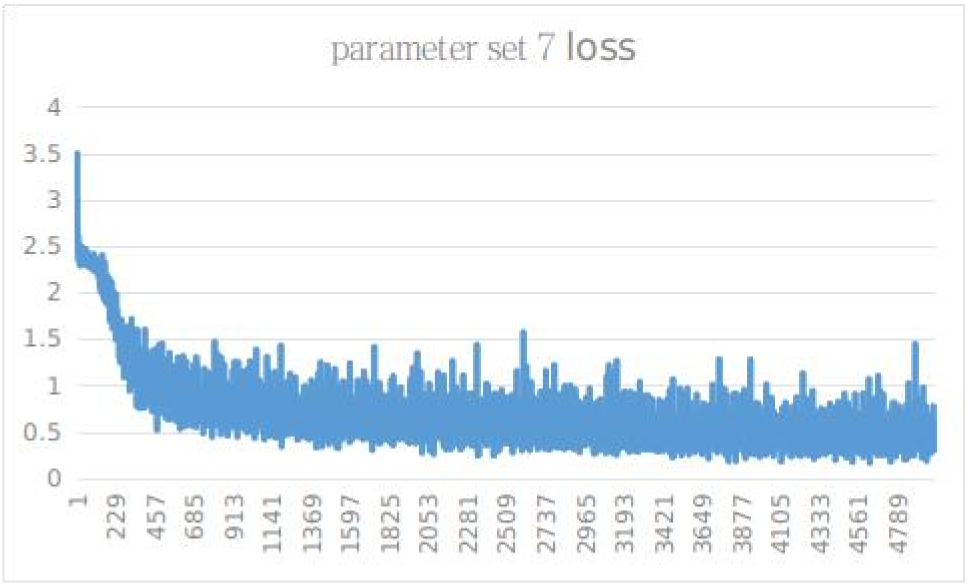


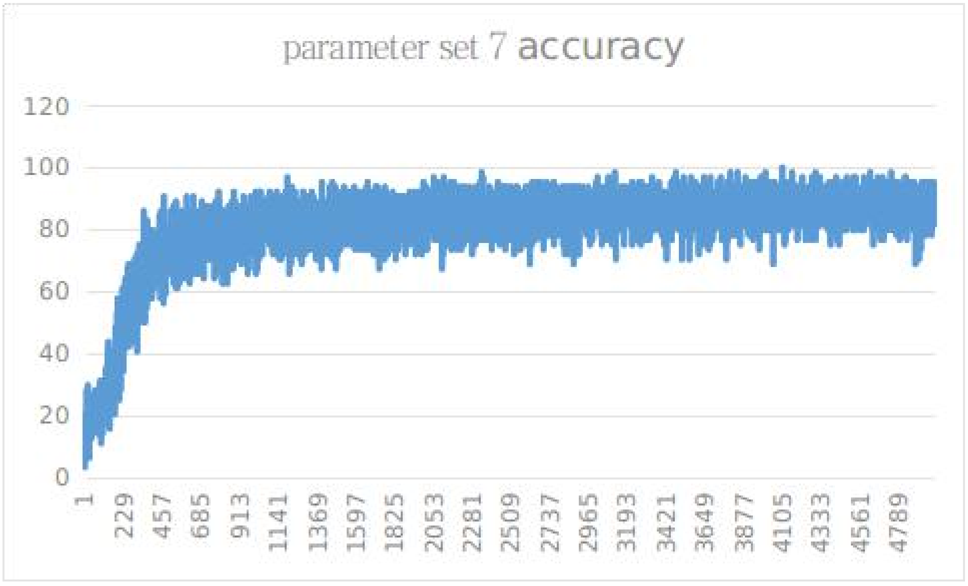
1.num_hidden=64
2.batch_size=64
3.patch_size=7
4.conv1_depth=16
5.conv2_depth=16
6.pooling_stride=2
7.drop_out_rate=0.9
8.base_learning_rate=0.005
9.decay_rate=0.99
10.optimizer='adam'

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.





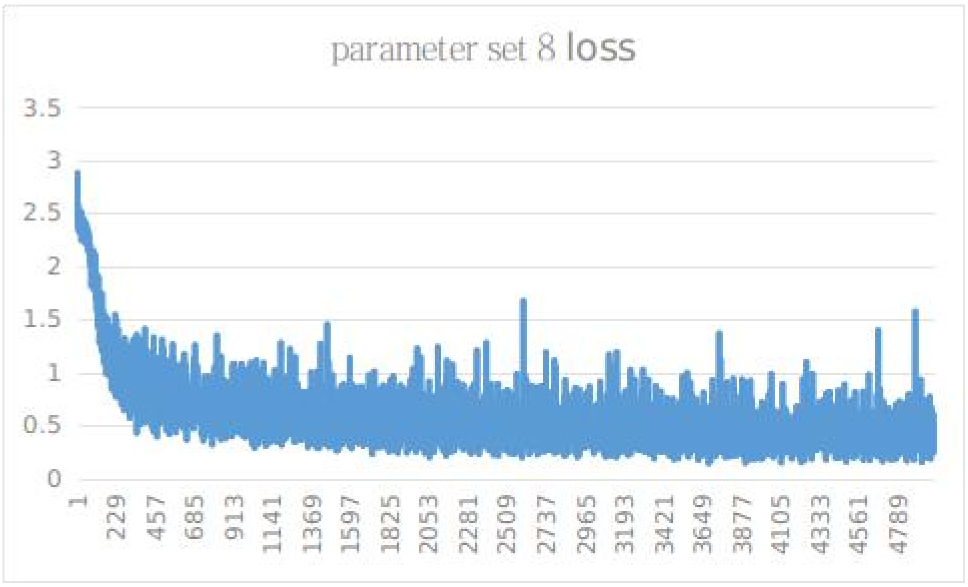
1.num_hidden=64
2.batch_size=64
3.patch_size=7
4.conv1_depth=16
5.conv2_depth=16
6.pooling_stride=2
7.drop_out_rate=0.9
8.base_learning_rate=0.0005
9.decay_rate=0.99

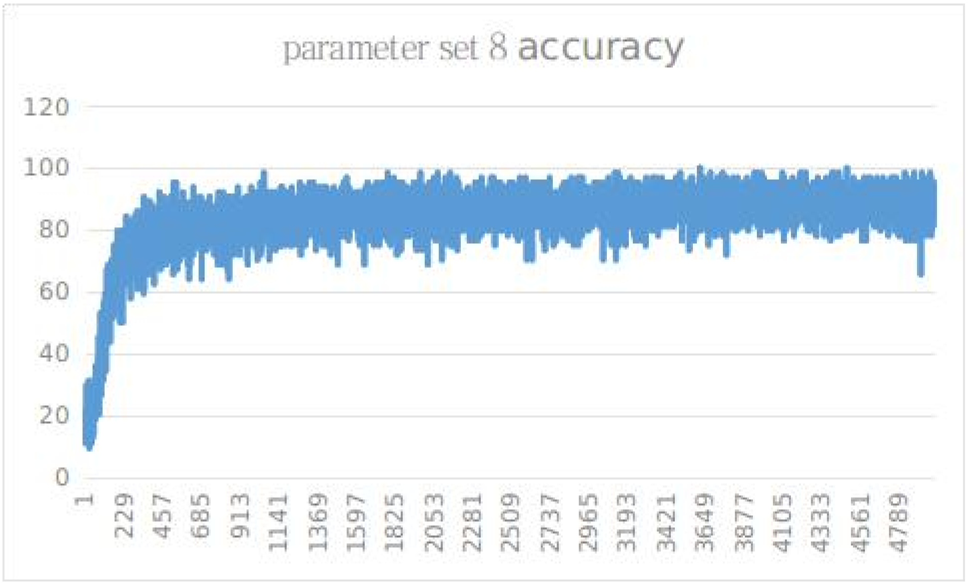
10.optimizer='adam'

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



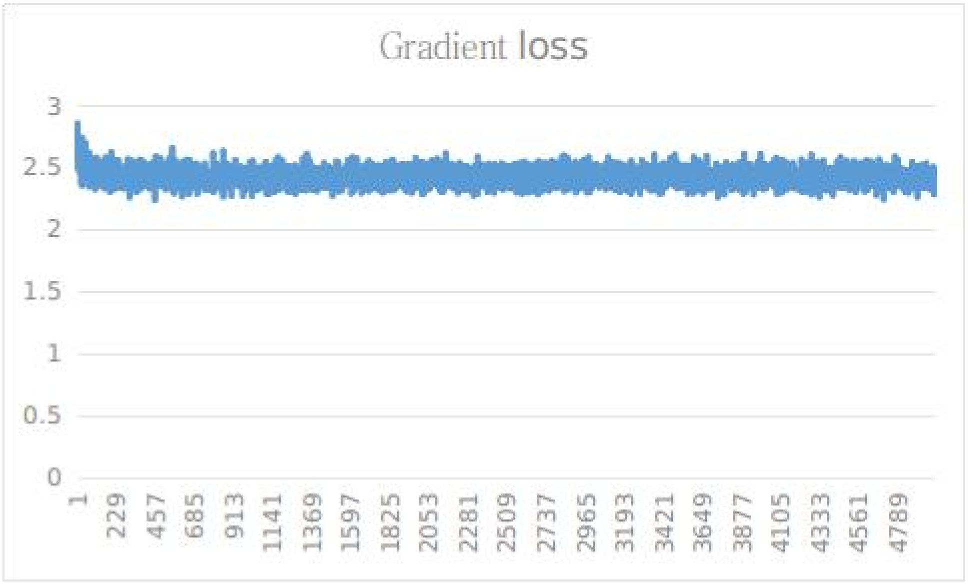


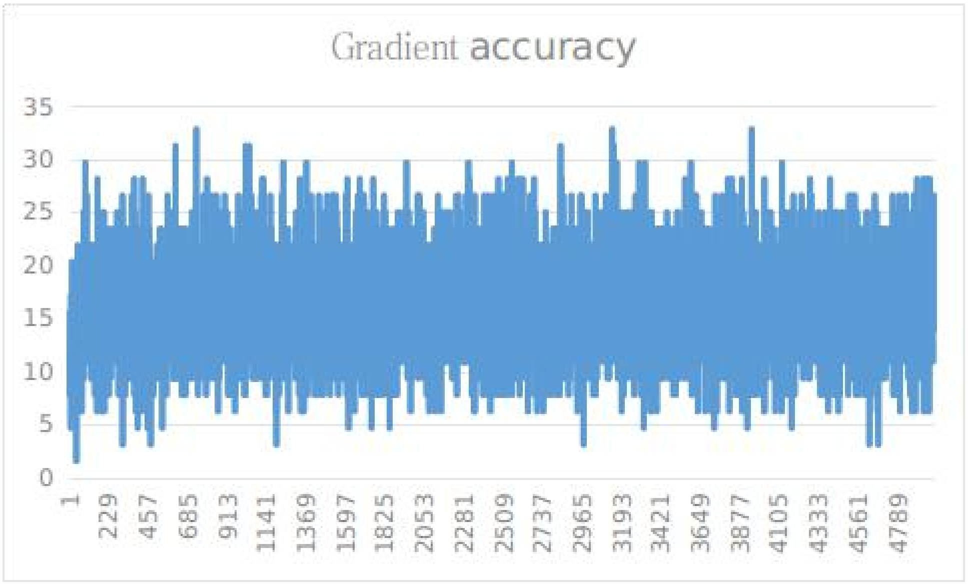
1.num_hidden=64
2.batch_size=64
3.patch_size=7
4.conv1_depth=16

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'

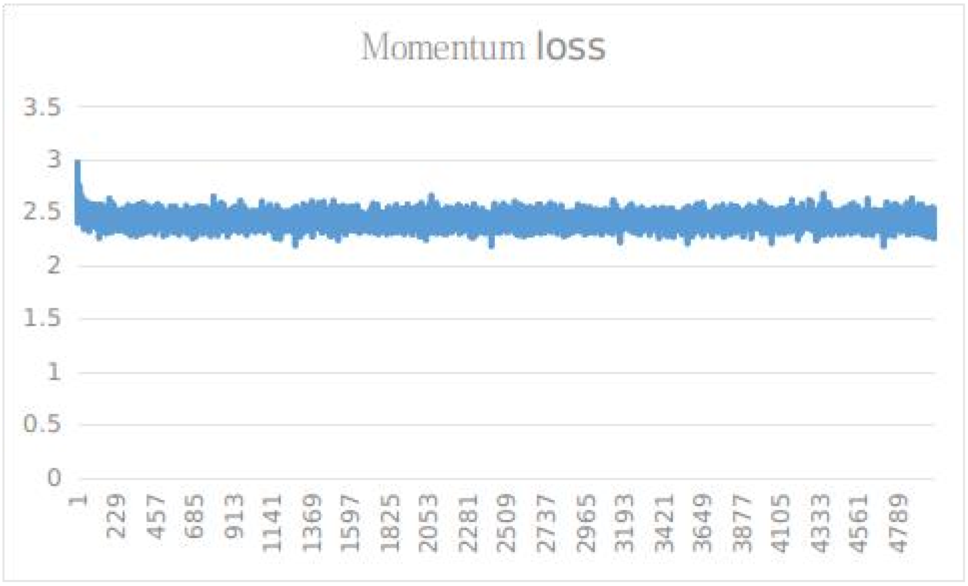
Based on parameter set 5, I changed decay rate to 0.9. The result is similar to parameter set 5. Intuitively, since the network converges very quickly, the decay rate doesn't have enough time to show its influence. If the network converges slower, then decay rate should be more observable.

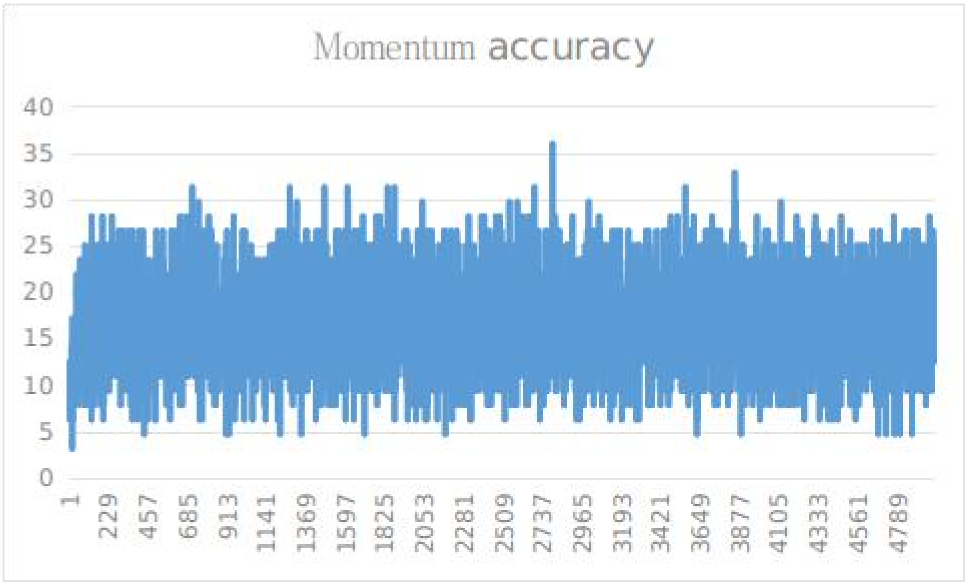
Observation 7: Loss function / Optimizer function is probably the most important factor.





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

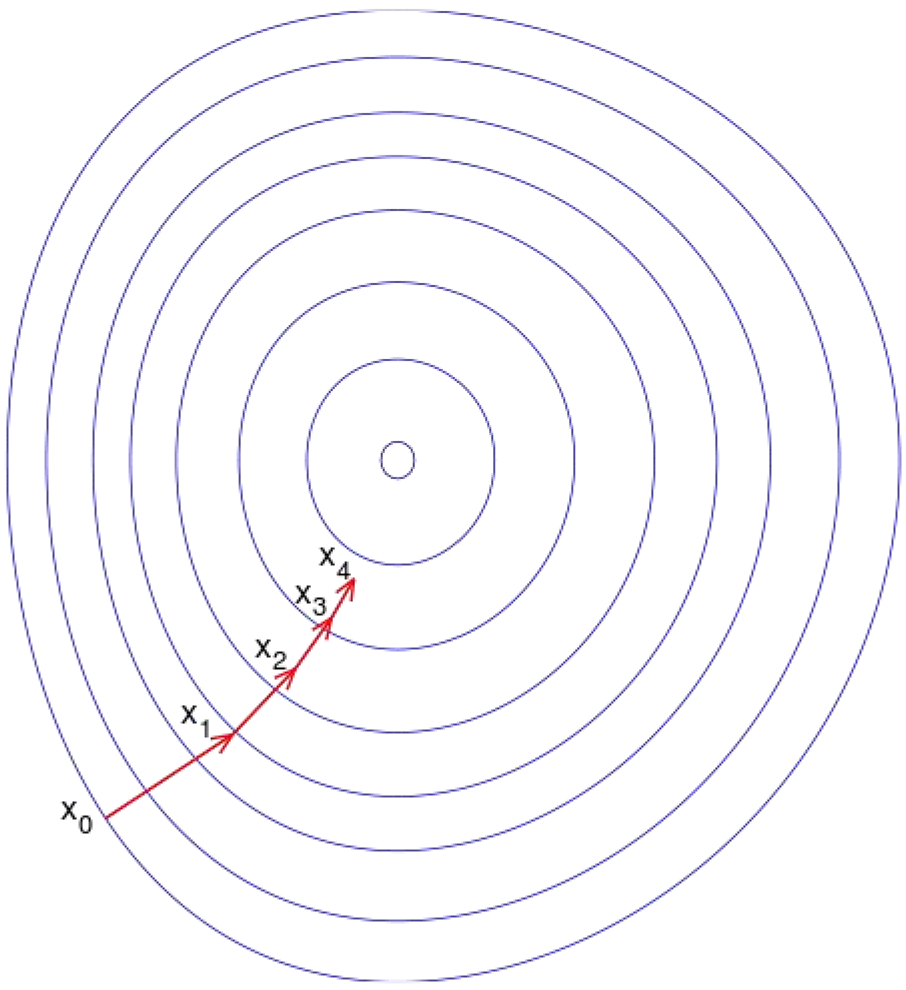




The parameter set is optimal except for that Momentum Optimizer is used.

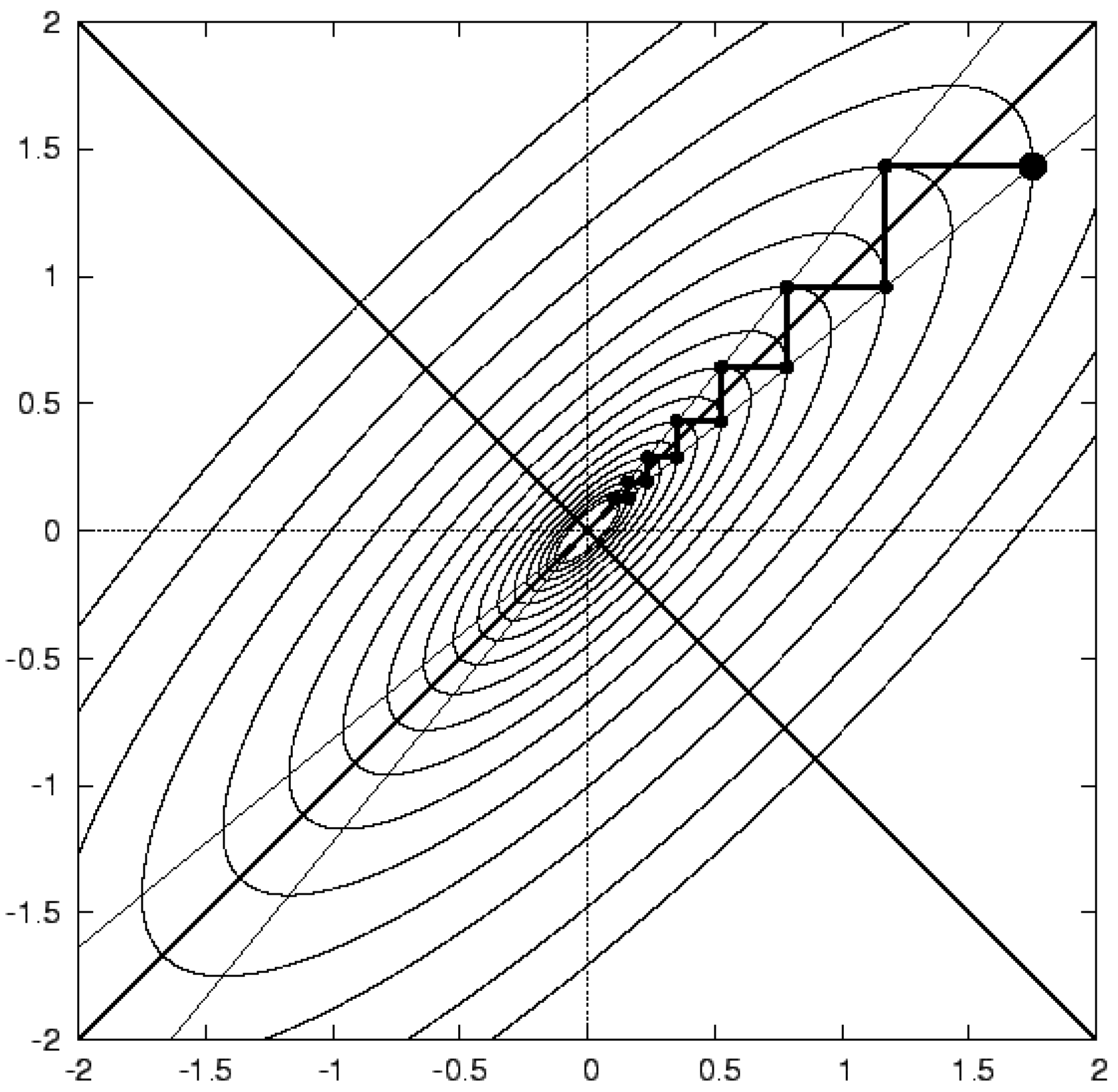
Normal gradient descent and momentum just fail.

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

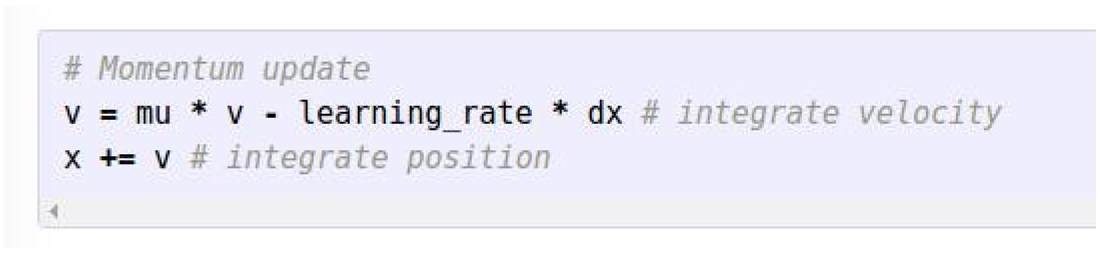
Above picture is just a demonstration in a 2-dimensional space. In reality, we are searching in multi-dimensional space. Because learning rate is applied to every dimension, if a dimension has much more slope than another dimension, then the simple gradient descent will just step more in the more slope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

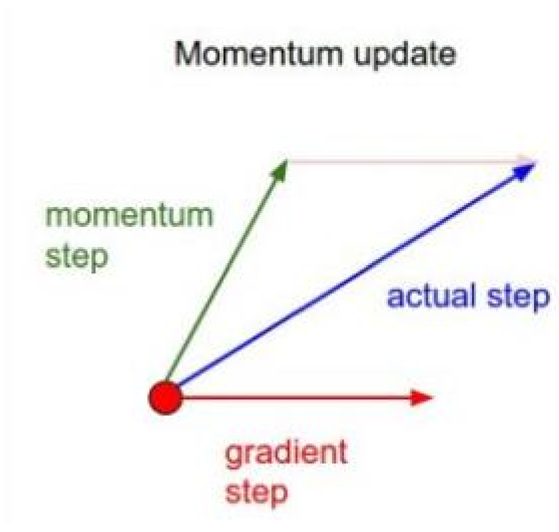
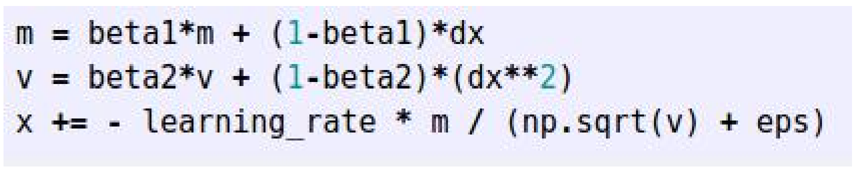


Image from http://cs231n.github.io/neural-networks-3/

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 doesn’t deeper network help?

Due to limited computing power, I cannot vertically scale deeper. One interesting observation is that deeper network doesn’t improve the result at all, possibly because 2 convolutional layers are enough to catch all the information in the input since the input are not complex.

Because of this, it makes little sense to try out more pooling with different strides.

Results

The best loss is about 0.3. The best accuracy is about 93% at training. The testing gives me a better accuracy of 96%.

Conclusion

Convolutional neural network is a powerful and flexible feature extraction method. Loss function is vital to a good result. Other parameters are also important to tune according to the specific problem. In this problem, dropout and learning rate are the two most important parameters. A deeper and larger network may help, but not necessary to some problems.