Multi-Layer Perceptron

Zhuoran Liu

January 31, 2017

1 Introduction

In this report, we will consider the neural network multi-layer perceptron. Multi-layer perceptron is a neural network with multiple hidden layers and in each hidden layer there are multiple neurons. Between different neurons in adjacent layers, there exist weights to connect different neurons. In every hidden layer, there exist biases to tune some outcomes of neurons. The aim of training the network is to find the proper weights and biases which can be used to predict the new data.

1.1 Underlying Theory

The working mechanism of the MLP is like following. Given the neural network structure(weights matrix W and biases vector b) and input data vector a, we can calculate the feed forward process by formula

$$\mathbf{z} = g(\mathbf{w}a + \mathbf{b})$$

Here g is the activation function and output z is a vector. Use z as the input of the next layer, we can do similar calculation again until we get the output. Given the last output z, we will calculate the $argmax(z)$. The category of the $argmax(z)$ is the predication of the input a. Given the right category(training targets), we can tune the weights and biases to predict better and better. This is the learning process of MLP.

1.2 Learning Algorithm

Backpropagation algorithm is the learning algorithm we will use. It consists of 4 steps.

$$\delta^L = \nabla_a C \circ \sigma'(z^L)$$

$$\delta^L = ((w^{l+1})^T \delta^{l+1} \circ \sigma'(z^l)$$

$$\frac{\partial C}{\partial b_j^l} = \sigma_j^l$$

$$\frac{\partial C}{\partial w_{jk}^l} = a_k^{l-1} \sigma_j^l$$

Here symbol $\circ$ means Hadamard product.a_k^l means neuron k in the lth layer. These four step will backpropagate error and output the gradient change of cost function. The whole process is input, feedforward, backpropagate, update weights and biases, input again... Finally after many epoches we will get the final neural network with particular weights and biases, then we finish training.

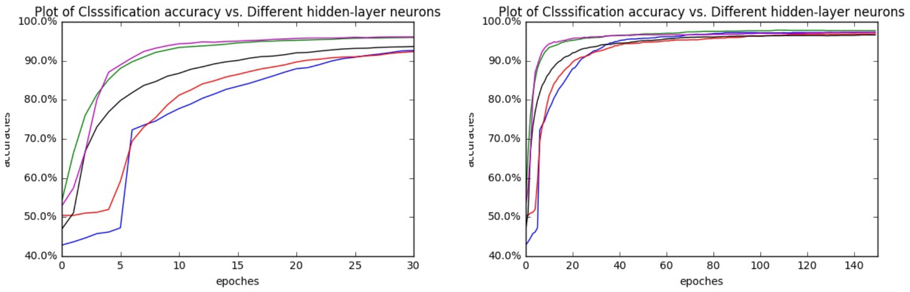


Figure 1: Different learning processess by different setting of neuron number with one hidden layer

2 Problem statement

1. The influence of different number of layers and different number of neurons.
2. The different influences of defferent initializations of the weights and biases before learning.
3. Influences by choosing different activation functions.
4. Compare two different learning method - Stochastic Gradient Descent vs. Momentum.

3 Results

3.1 The structure of Multi-Layer Perceptron

3.1.1 Different neurons

The first problem to investigate is the influence on classification accuracies from the number of neurons in hidden layers. We did the experiment with five different settings. All the settings have one hidden layer with different neurons. All five experiments used same input layer (784) and output layer, and used same epoches (150), same learning rate (0.05) and same mini-batch number (10). Figure 1. shows the results of five different setting.Blue line is 2 neurons, red line is 5 neurons, black line is 10 neurons, green line is 20 neurons and magenta line is 100 neurons.

In figure 1, the horizontal axis showed the epoches changed from 0 to 150, and the vertical axis showed the test accuracies in percentage. The formula

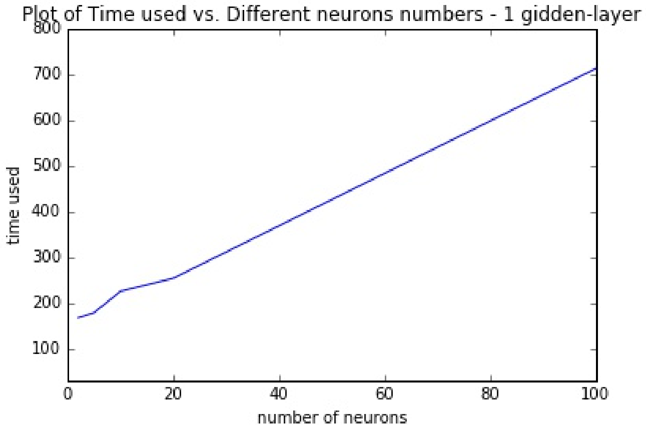


Figure 2: Different learning time by different setting of neuron number with one hidden layer

used here is

$$\text{accuracy} = \frac{\text{right classified number in test images}}{\text{total number in test images}}$$

From the left plot we can easily see that, if there is one hidden layer, more hidden neurons will yield better results in small number of epoches.For example, the magenta line(100 neurons) did better than others before 30 epoches.
But from right side plot, by large amount of epches, intermediate number of neurons had better results. For example, green line (20 neurons) is better than others in right side plot.
Here the possible reason is over-fitting, since more neurons will yield more complex model. After training many steps, the model is too complex. So had a worse accuracies on test set.

We also need to consider the running time of different cases. In figure 2, it is obvious that more neurons will take more training time, and the relationship between neurons' number and time cost is almost linear. Since we used only one layer, it will just have more calculations by neurons. So it is reasonable here.

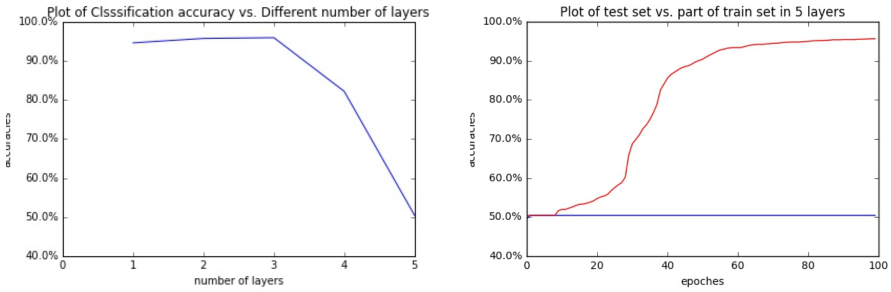


Figure 3: Different learning accuracies by different setting of layer numbers

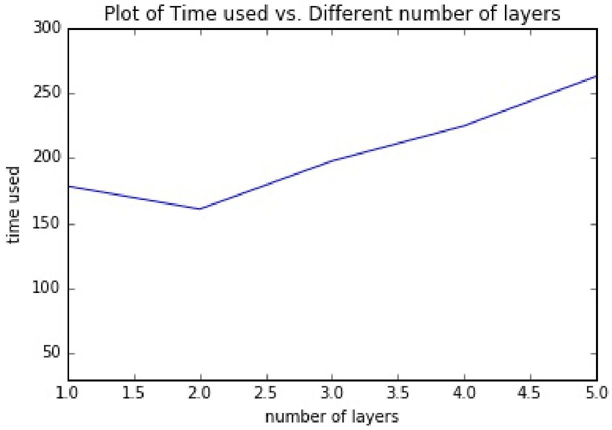


Figure 4: Different learning time by different setting of layer numbers

3.1.2 Different layers

The second problem to investigate is the influence on classification accuracies from the number of hidden layers. We did the experiment with five different settings. The settings of hidden layer number ranges from 1 to 5, and in each hidden layer we used 5 neurons. All five experiments used same input layer (784) and output layer, and used same epoches (100), same learning rate (0.05) and same mini-batch number (10). The figure 3 and figure 4 showed the results of five different settings.

From figure 3 left side, we can see that more hidden layers had a bad performance after layers' amout 3. It increased a little bit from layer number 1 to 3, then dropped after 3.
In figure 3 right side, we plotted the accuracies change between 5 hidden layers with test set and 5 hidden layers with part of training set. Here the

plot means there exist a over-fitting problem. Since for training data (red line), it is fine. But for test data. it is not good. The right side plot is not like this every training time, but it often happens.

More layers will take more training time, and the relationship between layers' number and time is also almost linear. The figure 4 is the running time of the five experiments we mentioned above. This relationship may not be the same, when we have more neurons in every layer and when we have more layers. It will be like exponential time, but not really. Since all the calculations are linear.

3.2 The initialization of the weights and biases

By default we choose the initialization of weights and biases using Gausssian random variables with mean 0 and deviation 1. Here we used another way to improve it such that we can have a better initialization and better results.

We mainly considered the initialization with a normalized Gaussian. The formula is

$$\text{initialization of weights} = \frac{\text{random}(y, x)}{\sqrt{x}}$$

This setting can make the distribution more sharply peaked which makes it less likely that neuron will saturate. We did experiment for both initializations and have the results below.Both experiments used same input layer (784) and output layer, and used same epoches (150), same learning rate (0.05) and same mini-batch number (10).Both with 1 hidden layer and 100 neurons. The plot below shows the results of two different settings.

3.2.1 different initialization of weights and biases

From the figure 5, we can easily see that. Optimized initialization with a normalizing component will make the initial accuracy munch higher than the default gaussian distribution. Also it showed a better result than the default setting. Since the distribution is more sharply peaked, it less likely that neuron will saturate.

3.2.2 different activation functions

By default we used sigmoid function as the activation function. Here we compared the sigmoid function with tanh function.

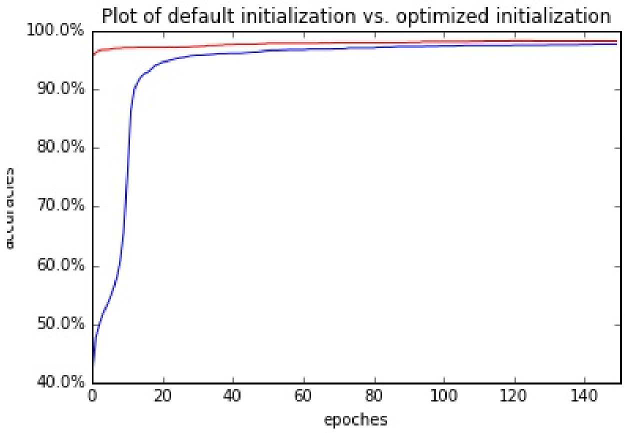


Figure 5: Different initializations of weights and biases

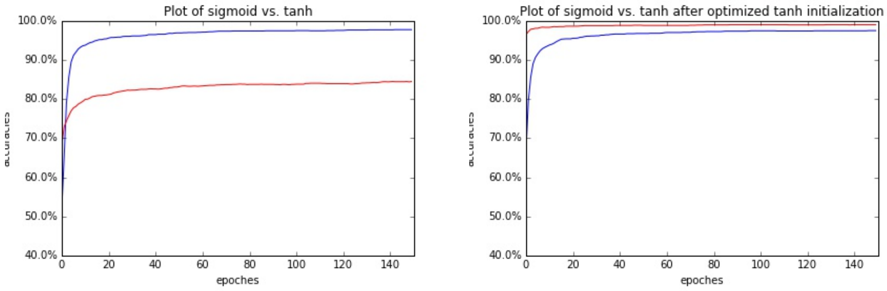


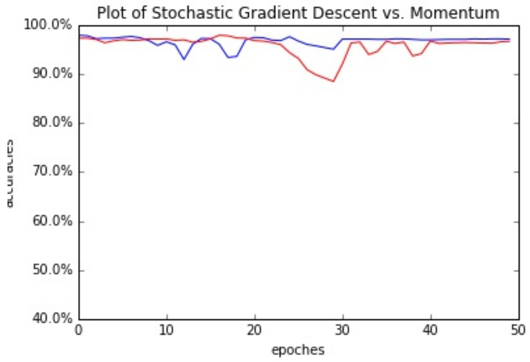
Figure 6: Different activation functions in same structure

From figure 6 left side, we can see that sigmoid did better. Here should not be so much difference, so we optimized the initialization of tanh function. The result is on the right side plot. It showed that tanh activation function did the process better. We can conclude that different activations functions have different performances, and it really depended on the initializations of weights and biases. Because at initialization we want the weights to be small enough around the origin so that the activation function operates in its linear regime, where gradients are the largest.

3.3 Different learning methods

3.3.1 Stochastic gradient descent vs. Momentum

Momentum is a method to escape from local minimum, and it did speed up the learning process by changing gradient direction more straight forward to



the target. We used two formulas to implement the momentum learning.

$$v' = \mu v - \eta \nabla C$$

$$w' = w + v'$$

After implementation we got the results compared with stochastic gradient method as below. In figure 7, red line showed the process of stochastic gradient descent and blue line showed the process of momentum. The momentum convergence faster than SGD. Since it changed more in every step than SGD, it fluctuated a little bit.

4 Discussion and Conclusion

Multi-layer perceptron is perceptron which can be trained by various patterns. It is not easy to be trained well. This training process depended on many aspects. For example number of neurons, number of layers, initialization of weights and biases, learning methods, learning rates, size of mini-batches and so on.

From what we have explored, more hidden layers and more neurons involved may have better results, but it will also take more time and cause over-fitting. The time has a similar linear relationship with the number of neurons and the number of hidden layers. A good initialization can obviously improve the initial accuracy of the learning process, and it also help tune the weights and biases better. Different activation functions have different influences on the accuracies of test, and they have also different running time. The choose of learning rate, epoches and mini batch sizes are also import parts. But they can be investigated by some experiments and choose the comparable better one.

5 Appendix