
成绩

北京航空航天大學
BEIHANG UNIVERSITY

# Pattern Recognition and Machine Learning Experiment Report

院（系）名称 自动化科学与电气工程学院

专 业 名 称 自动化

学 生 学 号 15071129

学 生 姓 名 苗子琛

2018 年 6 月

# 4 Neural Networks and Back Propagation

## 4.1 Introduction

The learning model of Artificial Neural Networks (ANN) (or just a neural network (NN)) is an approach inspired by biological neural systems that perform extraordinarily complex computations in the real world without recourse to explicit quantitative operations. The original inspiration for the technique was from examination of bioelectrical networks in the brain formed by neurons and their synapses. In a neural network model, simple nodes (called variously "neurons" or "units") are connected together to form a network of nodes, hence the term "neural network".

Each node has a set of input lines which are analogous to input synapses in a biological neuron. Each node also has an"activation function"that tells the node when to fire, similar to a biological neuron. In its simplest form, this activation function can just be to generate a'1'if the summed input is greater than some value, or a'0'otherwise. Activation functions, however, do not have to be this simple - in fact to create networks that can do useful things, they almost always have to be more complex, for at least some of the nodes in the network. Typically, there are at least three layers to a feed-forward network - an input layer, a hidden layer, and an output layer. The input layer does no processing - it is simply where the data vector is fed into the network. The input layer then feeds into the hidden layer. The hidden layer, in turn, feeds into the output layer. The actual processing in the network occurs in the nodes of the hidden layer and the output layer.

## 4.2 Principle and Theory

The goal of any supervised learning algorithm is to find a function that best maps a set of inputs to its correct output. An example would be a simple classification task, where the input is an image of an animal, and the correct output would be the name of the animal. For an intuitive example, the first layer of a Neural Network may be responsible for learning the orientations of lines using the inputs from the individual pixels in the image. The second layer may combine the features learned in the first layer and learn to identify simple shapes such as circles. Each higher layer learns more and more abstract features such as those mentioned above that can be used to classify the image. Each layer finds patterns in the layer below it and it is this ability to create internal representations that are independent of outside input that gives multi-layered networks their power. The goal and motivation for developing the back-propagation algorithm was to find a way to train a multi-layered neural network such that it can learn the appropriate internal representations to allow it to learn any arbitrary mapping of input to output.

Mathematically, a neuron's network function $f(x)$ is defined as a composition of other functions $g_i(x)$ which can further be defined as a composition of other functions. This can be conveniently represented as a network structure, with arrows depicting the dependencies between variables. A widely used type of composition is the nonlinear weighted sum, where:

$$f(x) = (\sum_i w_i g_i(x))$$

where $K$ (commonly referred to as the activation function) is some predefined function, such as the hyperbolic tangent. It will be convenient for the following to refer to a collection of functions $g_i$ as simply a vector $g = (g_1, g_2, ..., g_n)$.Back-propagation requires a known, desired output for each input value in order to calculate the loss function gradient. It is therefore usually considered to be a supervised learning method.

The squared error function is:

$$E = \frac{1}{2}(t - y)^2$$

Where $E$ is the squared error, $t$ is the target output for a training sample, and $y$ is the actual output of the output neuron. For each neuron $j$, its output $o_j$ is defined as

$$o_j = \varphi(net_j) = \varphi(\sum_{k=1}^{n} w_{kj} x_k)$$

The input net to a neuron is the weighted sum of outputs $o_k$ of previous neurons. If the neuron is in the first layer after the input layer, the $o_k$ of the input layer are simply the inputs $x_k$ to the network. The number of input units to the neuron is $n$. The variable $w_{ij}$ denotes the weight between neurons $i$ and $j$.

The activation function $\varphi$ is in general non-linear and differentiable. A commonly used activation function is the logistic function, e.g.:

$$\varphi(z) = \frac{1}{1 + e^{-z}}$$

which has a nice derivative of:

$$\frac{\partial \varphi}{\partial z} = \varphi(1 - \varphi)$$

Calculating the partial derivative of the error with respect to a weight $w_{ij}$ is done using the chain rule twice:

$$\frac{\partial E}{\partial w_{ij}} = \frac{\partial E}{\partial o_j} \frac{\partial o_j}{\partial net_j} \frac{\partial net_j}{\partial w_{ij}}$$

We can finally yield:

$$\frac{\partial E}{\partial w_{ij}} = \delta_j x_i$$

with

$$\delta_j = \frac{\partial E}{\partial o_j}\frac{\partial o_j}{\partial net_j} = \begin{cases} (o_j - t_i)\varphi(net_j)(1-\varphi(net_j)) & \text{if } j \text{ is an output neuron} \\ (\sum_{l\in L}\delta_l w_{jl})\varphi(net_j)(1-\varphi(net_j)) & \text{if } j \text{ is an inner neuron} \end{cases}$$

## 4.3 Objective

The goals of the experiment are as follows:

(1) To understand how to build a neural network for a classification problem.

(2) To understand how the back-propagation algorithm is used for training a given a neural network.

(3) To understand the limitation of the neural network model (e.g., the local minimum).

(4) To understand how to use back-propagation in Autoencoder.

4.4 Contents and Procedures

### *Stage 1*

***(1) Given a dataset for classification, (E.g., Iris, Pima Indian and Wisconsin Cancer from the UCI ML Repository). Build a multi-layer neural network (NN) and train the network using the BP algorithm. We can start with constructing a NN with only one hidden layer.***

I trained different Neural Networks of different sizes for the three different datasets, though they share some attributes in common.

**1) Activation function**

I choose the Relu for activation function for all networks below because it is well recognized as a better activation function than Sigmoid for the reason that it avoids causing gradients vanishing problem in deep neural networks and it costs much less in computation.

**2) L2 regularization**

L2 regularization is a trick to improve model's generalization ability. It adds the L2 norm in the last loss function so as to squeeze the values in the weights,

$$C = C_0 + \frac{\lambda}{2n}\sum_w w^2,$$

And when updating, it updates in the way,

$$\begin{aligned} w &\rightarrow w - \eta\frac{\partial C_0}{\partial w} - \frac{\eta\lambda}{n}w \\ &= \left(1 - \frac{\eta\lambda}{n}\right)w - \eta\frac{\partial C_0}{\partial w}. \end{aligned}$$

**3) Learning rate decay**

Learning rate decay is the trick for better optimal performance. It diminishes the update steps in an appropriate time so as to let the loss continue to drop. In my network, this appropriate time is the time after two epochs, and learning rate decays follows the rule below,

$$\mathrm{lr}_{i+1} = \mathrm{lr}_i \times 0.8$$

**4) Mean subtraction**

It the most common form for data preprocessing. It involves subtracting the mean across every individual attribute in the data and has the geometric interpretation of centering the cloud of data around the origin along every dimension. And it should be noticed that the data mean should only be calculated on training data but mean-subtraction operation should be performed on whole dataset.

## *A. The Iris dataset*

The Iris dataset is a multiclass classification task. It has three output labels. Therefore, Instead of using the conventional sigmoid activation function, a softmax layer, an extension of the sigmoid activation function, is used as the last layer of the network for computing loss. The output of this layer is,

$$h_\theta(x) = \frac{e^{z_j}}{\sum_{k=1}^{C} e^{z_k}}$$

For Iris dataset, C is 3.

The network, single hidden layer network, that I devise for this dataset is shown below,

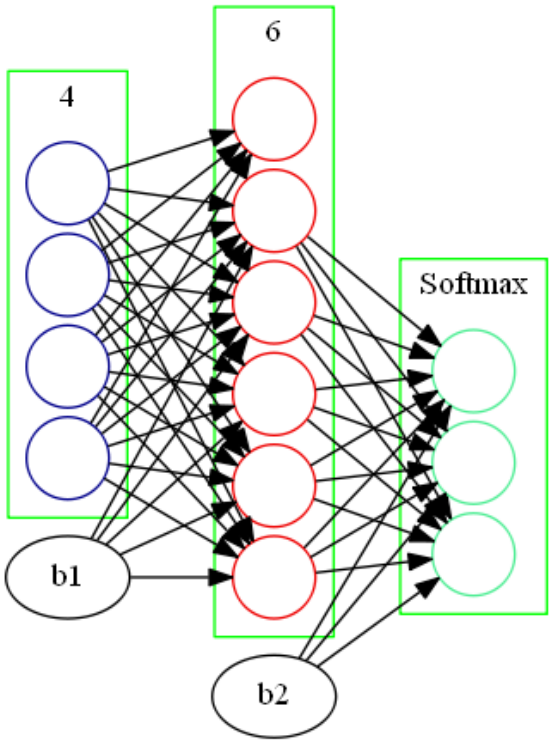



***Fig.1. NN for Iris dataset***

To train this network, I set hyper-parameter as,

```
%set hyper-parameters
neuron_num = [4 6 3];
learning_rate = 0.1;
batch_size = 15;
num_Epoches = 200;
learning_rate_decay = 0.8;
weight_decay = 5e-3;
```

and I partition the dataset as,

```
train_x = [data(1:30, :); data(51:80, :); data(101:130, :)];
train_y = [y(1:30, :); y(51:80, :); y(101:130, :)];
val_x = [data(31:40, :); data(81:90, :); data(131:140, :)];
val_y = [y(31:40, :); y(81:90, :); y(131:140, :)];
test_x = [data(41:50, :); data(91:100, :); data(141:150, :)];
test_y = [y(41:50, :); y(91:100, :); y(141:150, :)];
```

which means from the total 150 samples, I choose 90 samples to construct training set, 30 for validation set and the rest 30 for test set.

The training record incorporating training set and validation set accuracy record, training set loss and validation set loss record is shown below.

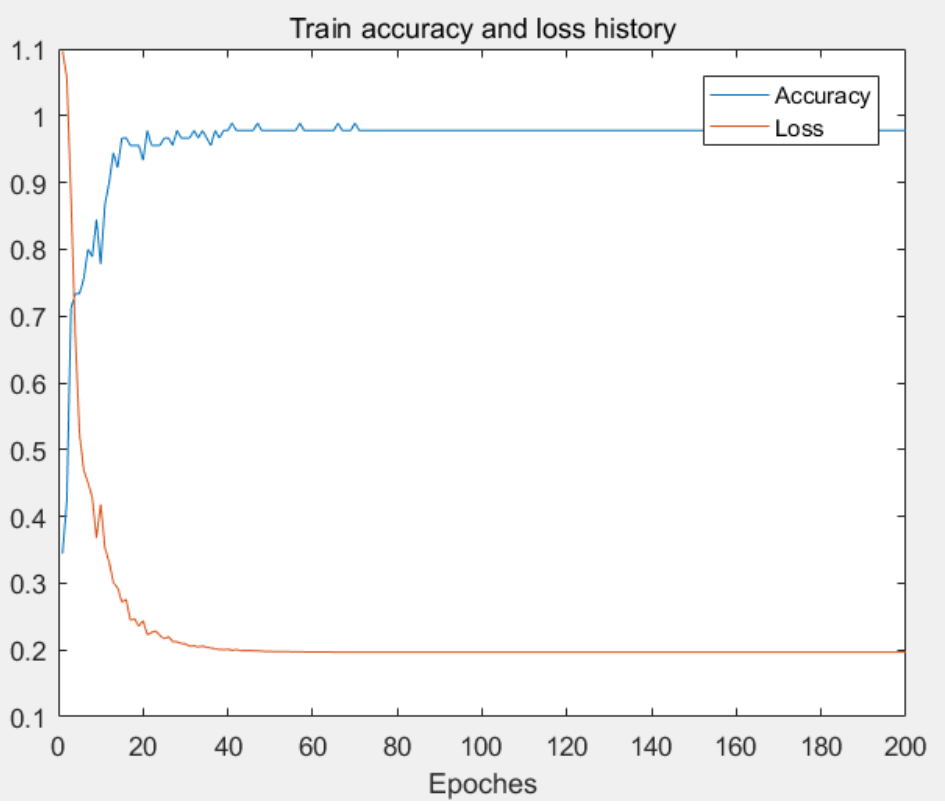



***Fig.2. Iris dataset training: training set performance***

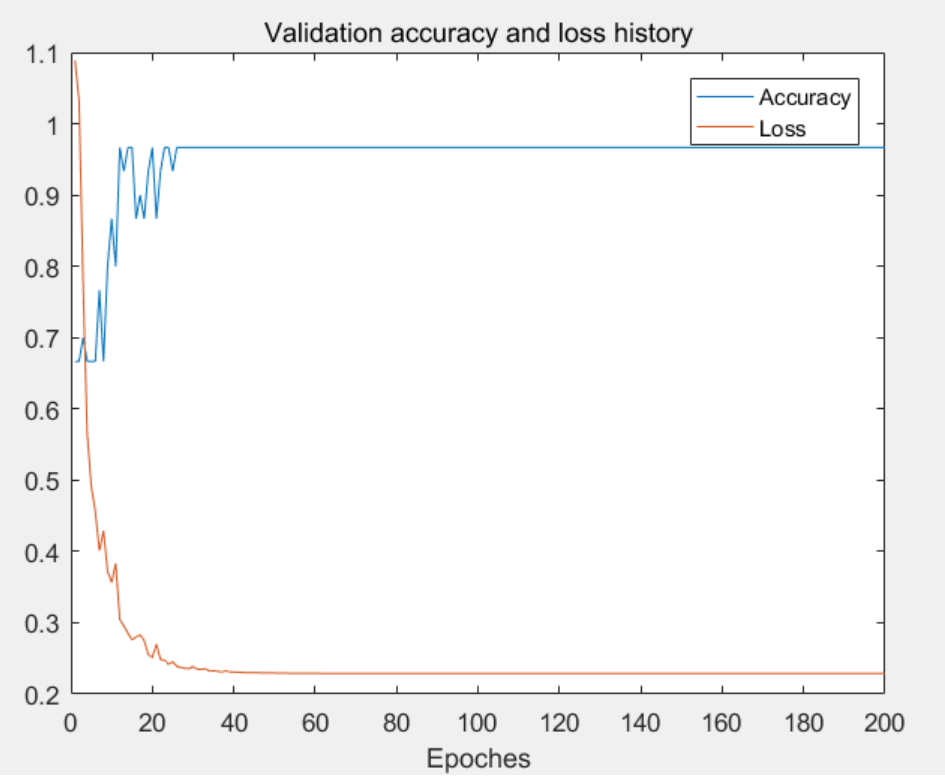



***Fig.3. Iris dataset training: validation set performance***

Then I test the network on test set, and got,

```
result on test set: test accuracy: 1.000000, test loss: 0.170807
```

which indicated that single hidden layer network performs really well on this easy dataset.

## *B. The Pima Indian Dataset*

The Pima-Indian dataset is a binary classification task. The neural network structure for this dataset is shown below.

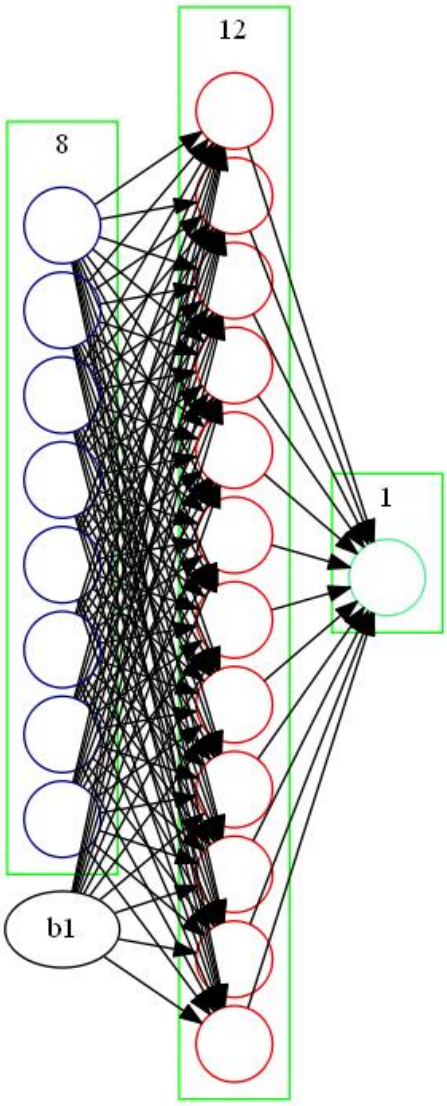



***Fig.4. NN for Pima Indians dataset***

To train this network, I set hyper-parameter as,

```
%set hyper-parameters
neuron_num = [8 12 1];
learning_rate = 0.05;
batch_size = 16;
num_Epoches = 200;
learning_rate_decay = 0.8;
weight_decay = 5e-3;
```

and I partition the dataset as,

```
train_x = data(1:32*16, :);
train_y = y(1:32*16);
val_x = data((32*16+1): 40*16, :);
val_y = y((32*16+1): 40*16);
test_x = data((40*16+1): 48*16, :);
test_y = y((40*16+1): 48*16);
```

which means from the total 768 samples, I choose 512 samples to construct training set, 128 for validation set and the rest 128 for test set.

The training record incorporating training set and validation set accuracy record, training set loss and validation set loss record is shown below.

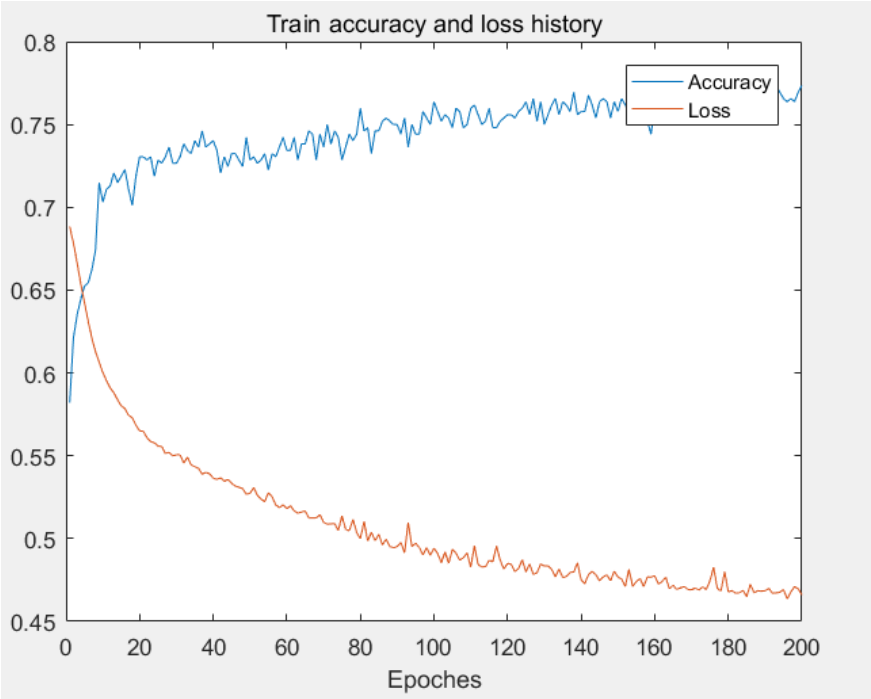



***Fig.5. Pima Indians dataset training: training set performance***

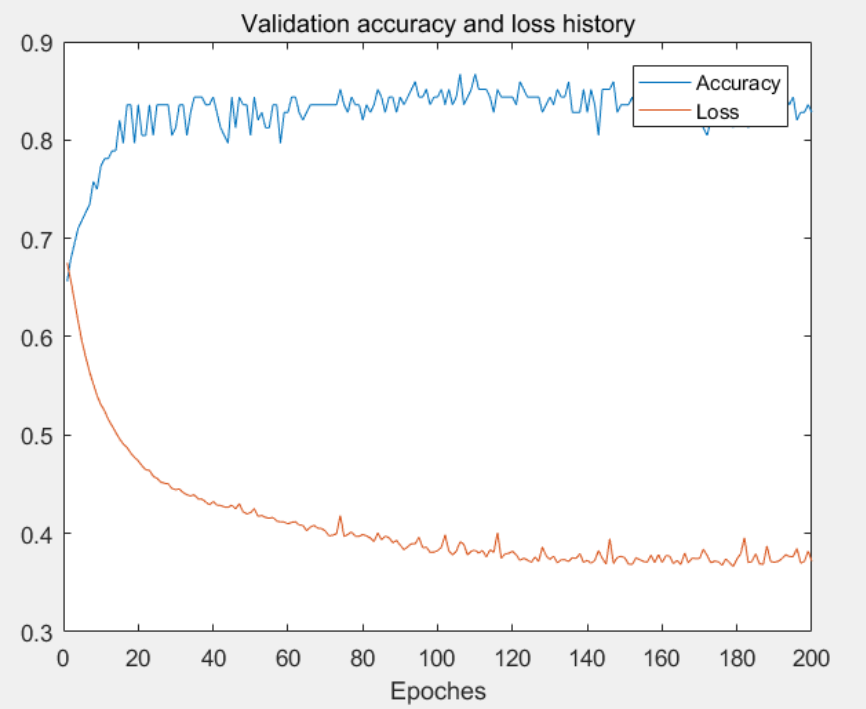


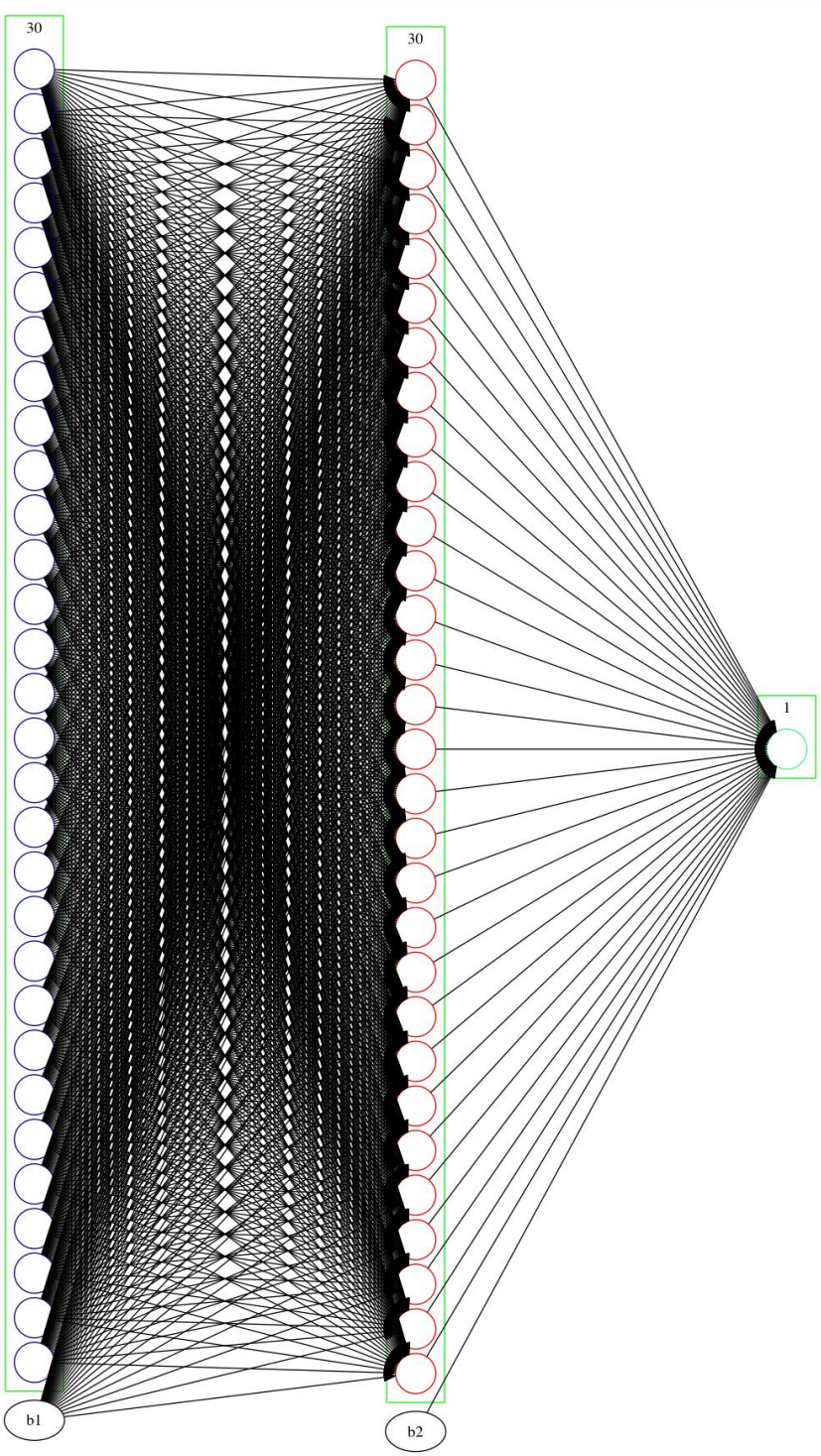

***Fig.6. Iris dataset training: validation set performance***

Then I test the network on test set, and got,

```
result on test set: test accuracy: 0.781250, test loss: 0.497134
```

### *C. The Wisconsin Cancer dataset*

The Wisconsin Cancer dataset is a binary-classification task. The neural network structure for this task is as follows.




***Fig,7 NN for Wisconsin Cancer dataset***

To train this network, I set hyper-parameter as,

```
%set hyper-parameters
neuron_num = [30 30 1];
learning_rate = 0.001;
batch_size = 20;
num_Epoches = 100;
learning_rate_decay = 0.8;
weight_decay = 5e-3;
```

and I partition the dataset as,

```
train_x = data(1:400, :);
train_y = y(1:400);
val_x = data(401: 469, :);
val_y = y(401: 469);
test_x = data(470: 569, :);
test_y = y(470: 569);
```

which means from the total 569 samples, I choose 400 samples to construct training set, 69 for validation set and the rest 100 for test set.

The training record incorporating training set and validation set accuracy record, training set loss and validation set loss record is shown below.

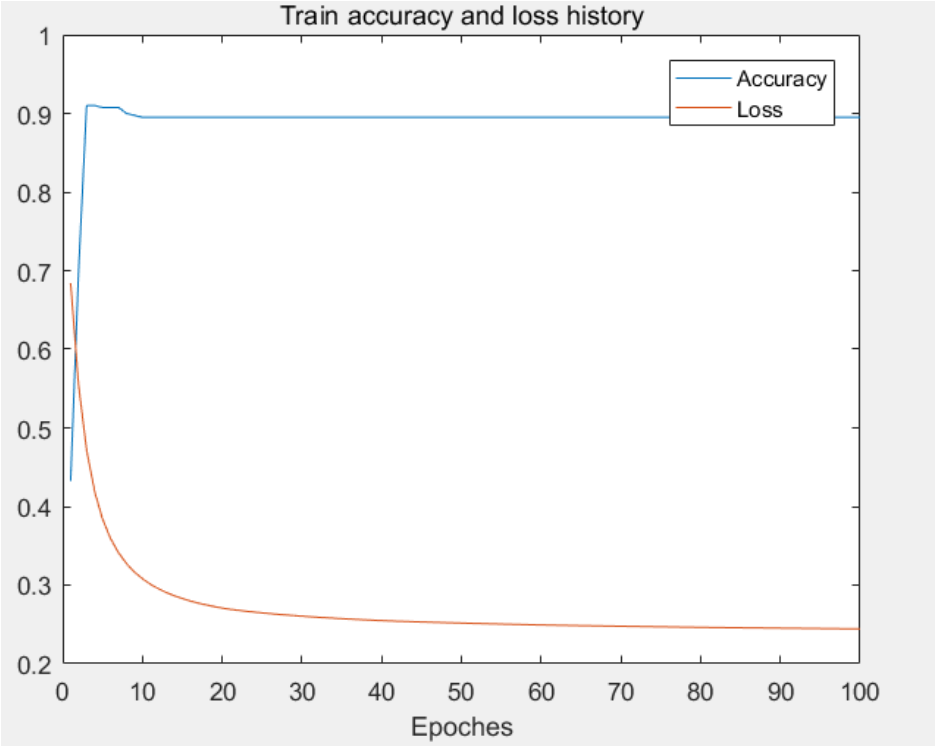



***Fig.8. Pima Indians dataset training: training set performance***

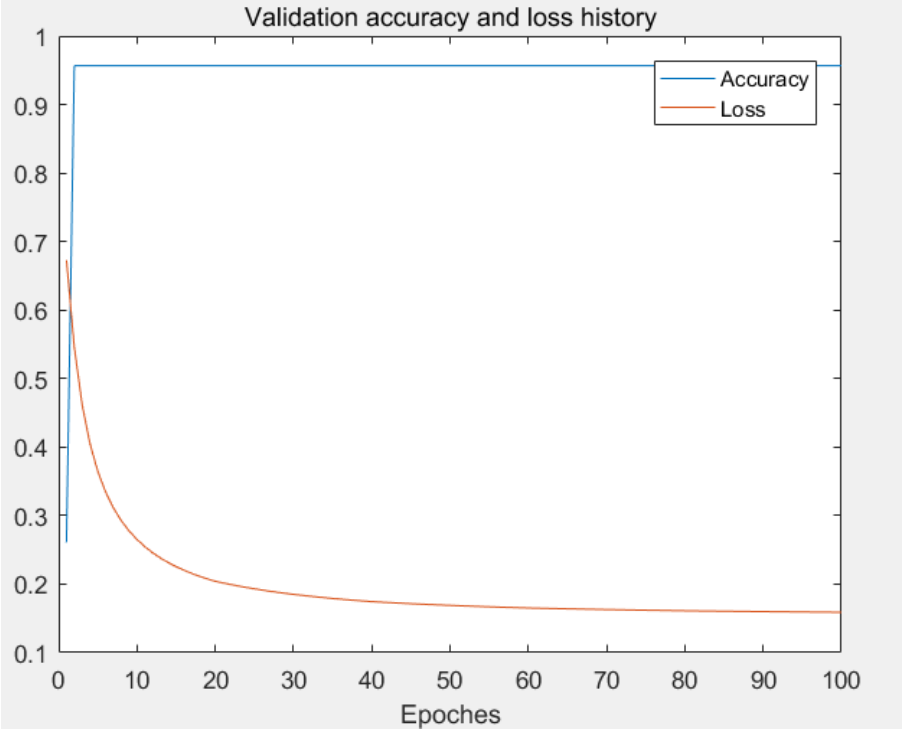



***Fig.9. Iris dataset training: validation set performance***

Then I test the network on test set, and got,

```
result on test set: test accuracy: 0.960000, test loss: 0.203224
```

### *(2) Compare the NN results to the results of Perceptron, which model is better in terms of efficiency and accuracy?*

I use the perceptron in experiment one and select the Wisconsin Cancer dataset for comparison.

Having trained perceptron for 500 epochs, the perceptron attained an accuracy of 83.34% on the test set, which is considerably lower than that of the one-hidden-layer neural network, which reached an accuracy of 96% on the test set.

From the result one can tell the neural network with single hidden layer has more capacities comparing to the perceptron to fit the complex dataset. I think this better performance owes to the more non-linearity involved in single-hidden-layer neural network.

### *(3) Whether the performance of the model is heavily influenced by different parameters settings (e.g., the learning rate α)?*

I choose Iris dataset for comparison. I set the amount of neurons in the hidden layer at 6, and assign0.0001, 0.001, 0.1, 0.01, 0.5, 1 to the learning rate in turn and the results are shown below.

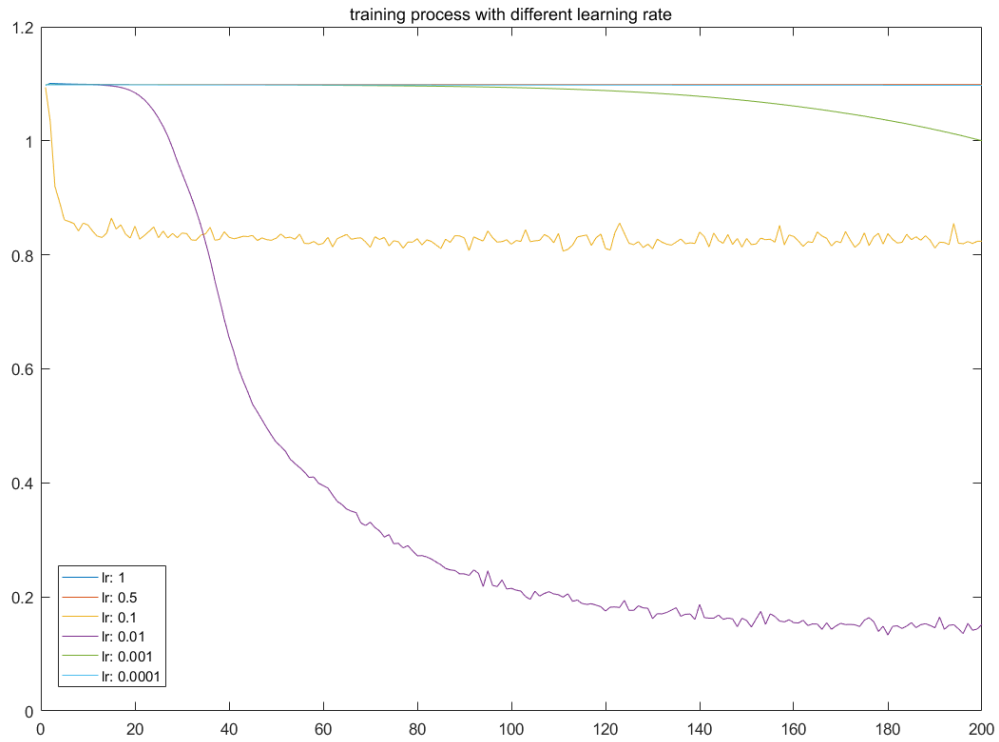



***Fig.10. training process with different learning rate***

***Table 1 test accuracy with different learning rate***

| Learning rate | 1 | 0.5 | 0.1 | 0.01 | 0.001 | 0.0001 |
|---|---|---|---|---|---|---|
| accuracy | 0.3333 | 0.3333 | 0.80 | 1.00 | 0.3333 | 0.3333 |

As it can be observed, a smaller learning rate slows down the training process. However, a large learning rate impedes the neural network from finding the minimum, and results in divergence in extreme cases. So, there is a tradeoff between training speed and training result. In this case, learning rate 0.01 is the optimal choice.

## *Stage 2*

### *(1) How the efficiency and accuracy will be influenced by different activation functions and more hidden layers.*

Due to the single-hidden-layer network performs really well on the Iris dataset and the Wisconsin Cancer dataset, I conduct this experiment on the Pima Indians dataset. The motivation is to explore the influence of network's depth on its performance. Thus, I will try networks with one, two, three and four hidden layers. To eliminate the disturbance of network's width, I set the number of neurons in each hidden layer to be 16. The result is shown below,

***Table 2 test accuracy with different number of hidden layers***

| # hidden layers | 1 | 2 | 3 | 4 |
|---|---|---|---|---|
| accuracy | 0.7734 | 0.7856 | 0.7039 | 0.4765 |

From the table, we can see there is a slight improvement when increase hidden layers from one to two. However, when I continue adding hidden layers, the performance went down. I suppose

it is the result of overfitting. When adding one hidden layer, the parameters increase a lot, meaning the network needs more data to train, but the size of the training set is constant. So, it more hidden layers does not mean higher performance. The number of hidden layers should be chosen carefully.

***(2) Do you think that biological neural networks work in the same way as our NN model? Can you provide any discussions to support your opinions?***

I believe that they are significantly different although artificial neural networks were originally inspired by biological neural networks.

Firstly, I admit their structural similarities. Artificial neural networks share similar structure with the biological ones: They both have several inputs and one output. With many neurons interlinked with each other, a complicated network is formed that is capable to perform complicated calculations.

However, they are different in the way that the biological neural network has no heavy computation, while ANN needs a large amount of computation to support it. Another fact is that biological neural network can performs work that ANN is incapable of.

(3) Using the BP algorithm to train an autoencoder.

Autoencoder is a network used to extract high semantic feature. It is trained to reconstruct its Input, which in my opinion, is a regression problem. So, I choose mean square error as the loss function and use Back Propagation (BP) to train the network. I simply build a single-hidden-layer network for this task, which is shown below. Besides, I conduct this experiment on the simple Iris dataset.

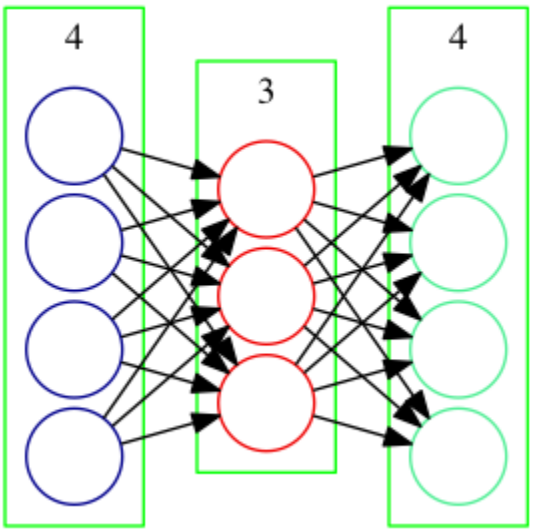



***Fig.11. structure of autoencoder***

the way that I partition the dataset is shown below,

```
train_x = [data(1:30, :); data(51:80, :); data(101:130, :)];
test_x = [data(31:50, :); data(81:100, :); data(131:150, :)];
```

This means that I use 90 samples for training and 60 for test.

The hyper-parameters I set are shown below,

```
%set hyper-parameters
neuron_num = [4 3 4];
learning_rate = 0.001;
batch_size = 5;
num_Epoches = 200;
learning_rate_decay = 0.9;
```

Notice that due to this dataset is too easy, so I discard the L3 regularization trick for sake of time efficiency.

The record of loss in training process is shown below,

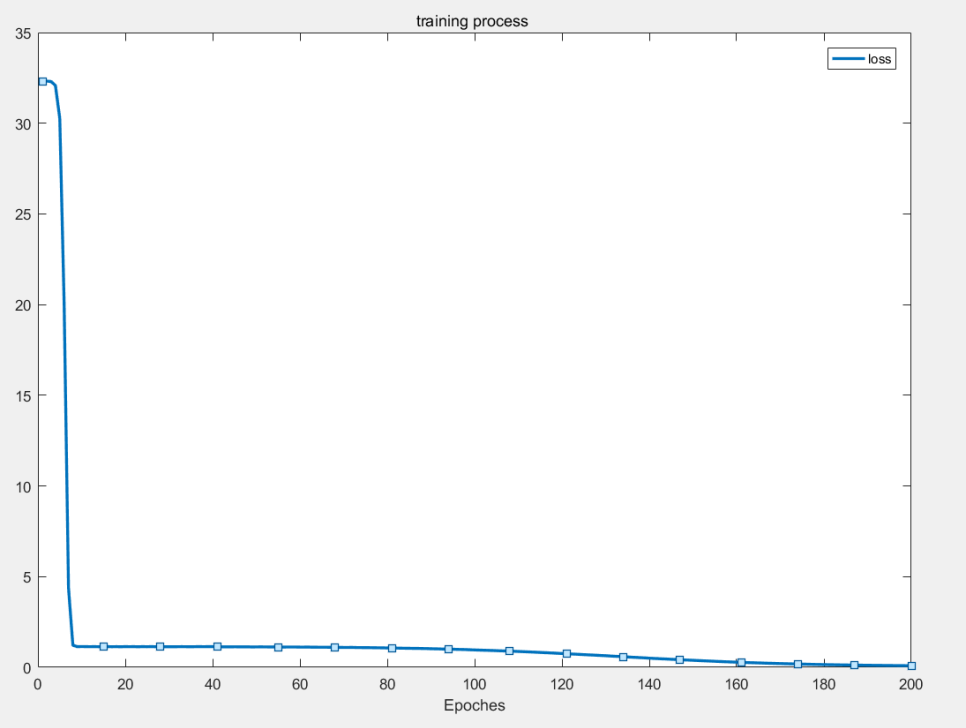



***Fig.12. training process of autoencoder***

During test time, I output the loss and compare the output values to input values to test the performance of the Autoencoder.

```
result on test set: test loss: 0.093510
```

***Table 3 test of Autoencoder***

| Input Values | | | | Output Values | | | |
|---|---|---|---|---|---|---|---|
| SL | SW | PL | PW | SL | SW | PL | PW |
| 4.8000002 | 3.0999999 | 1.6000000 | 0.20000000 | 4.6764712 | 2.9616776 | 1.8251613 | 0.42435083 |
| 5.4000001 | 3.4000001 | 1.5000000 | 0.40000001 | 5.1522427 | 3.3064415 | 1.9146945 | 0.42503917 |
| 5.1999998 | 4.0999999 | 1.5000000 | 0.10000000 | 5.2860732 | 3.5183527 | 1.6855429 | 0.31285834 |
| 5.5000000 | 4.1999998 | 1.4000000 | 0.20000000 | 5.4932652 | 3.6777534 | 1.7040405 | 0.30410367 |
| 4.9000001 | 3.0999999 | 1.5000000 | 0.20000000 | 4.7160702 | 3.0130658 | 1.7823950 | 0.40221998 |

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 5 | 3.2000000 | 1.2000000 | 0.20000000 | 4.7492251 | 3.1257994 | 1.5923291 | 0.31553376 |
| 5.5000000 | 3.5000000 | 1.3000000 | 0.20000000 | 5.1992164 | 3.4252484 | 1.7359494 | 0.34222582 |
| 4.9000001 | 3.5999999 | 1.4000000 | 0.10000000 | 4.8829479 | 3.2225788 | 1.6177641 | 0.31584662 |
| 4.4000001 | 3 | 1.3000000 | 0.20000000 | 4.3210220 | 2.8009253 | 1.5440133 | 0.32917026 |
| 5.0999999 | 3.4000001 | 1.5000000 | 0.20000000 | 4.9568210 | 3.2055256 | 1.7878641 | 0.38496709 |
| 5 | 3.5000000 | 1.3000000 | 0.30000001 | 4.8925619 | 3.2183428 | 1.6443641 | 0.32681394 |
| 4.5000000 | 2.3000000 | 1.3000000 | 0.30000001 | 4.1161900 | 2.5672867 | 1.6940268 | 0.41218606 |
| 4.4000001 | 3.2000000 | 1.3000000 | 0.20000000 | 4.3985758 | 2.8764393 | 1.5158643 | 0.31039694 |
| 5 | 3.5000000 | 1.6000000 | 0.60000002 | 4.9735818 | 3.1561596 | 1.9271395 | 0.44513428 |
| 5.0999999 | 3.8000000 | 1.9000000 | 0.40000001 | 5.2111630 | 3.2915101 | 2.0533080 | 0.48146948 |
| 4.8000002 | 3 | 1.4000000 | 0.30000001 | 4.5968685 | 2.9385750 | 1.7336566 | 0.39042449 |
| 5.0999999 | 3.8000000 | 1.6000000 | 0.20000000 | 5.1345391 | 3.3447597 | 1.7981787 | 0.37500489 |
| 4.5999999 | 3.2000000 | 1.4000000 | 0.20000000 | 4.5456066 | 2.9438338 | 1.6301693 | 0.34888837 |
| 5.3000002 | 3.7000000 | 1.5000000 | 0.20000000 | 5.1975713 | 3.3979855 | 1.7933331 | 0.36771423 |
| 5 | 3.3000000 | 1.4000000 | 0.20000000 | 4.8332233 | 3.1399684 | 1.7114798 | 0.36131585 |
| 5.5000000 | 2.4000001 | 3.8000000 | 1.1000000 | 5.3775015 | 2.6346664 | 3.8048961 | 1.2417899 |
| 5.5000000 | 2.4000001 | 3.7000000 | 1 | 5.3504949 | 2.6553941 | 3.7106376 | 1.2023499 |
| 5.8000002 | 2.7000000 | 3.9000001 | 1.2000000 | 5.7074685 | 2.8459928 | 3.9284697 | 1.2694308 |
| 6 | 2.7000000 | 5.0999999 | 1.6000000 | 6.1208010 | 2.7479181 | 4.8860965 | 1.6587733 |
| 5.4000001 | 3 | 4.5000000 | 1.5000000 | 5.7238111 | 2.7033212 | 4.2734742 | 1.4205302 |
| 6 | 3.4000001 | 4.5000000 | 1.6000000 | 6.2565746 | 3.0829816 | 4.3878999 | 1.4275604 |
| 6.6999998 | 3.0999999 | 4.6999998 | 1.5000000 | 6.6165395 | 3.2322166 | 4.7026262 | 1.5372081 |
| 6.3000002 | 2.3000000 | 4.4000001 | 1.3000000 | 5.9808607 | 2.8250327 | 4.4647079 | 1.4840226 |
| 5.5999999 | 3 | 4.0999999 | 1.3000000 | 5.7489967 | 2.8475535 | 3.9994247 | 1.2973882 |
| 5.5000000 | 2.5000000 | 4 | 1.3000000 | 5.4702916 | 2.6309681 | 3.9793384 | 1.3112836 |
| 5.5000000 | 2.5999999 | 4.4000001 | 1.2000000 | 5.5951157 | 2.6304824 | 4.2040682 | 1.4003787 |
| 6.0999999 | 3 | 4.5999999 | 1.4000000 | 6.1774960 | 2.9776213 | 4.4793649 | 1.4744335 |
| 5.8000002 | 2.5999999 | 4 | 1.2000000 | 5.6913028 | 2.7964418 | 4.0091567 | 1.3064017 |
| 5 | 2.3000000 | 3.3000000 | 1 | 4.9102254 | 2.4668417 | 3.3390305 | 1.0741314 |
| 5.5999999 | 2.7000000 | 4.1999998 | 1.3000000 | 5.6552768 | 2.7224884 | 4.1082606 | 1.3531326 |
| 5.6999998 | 3 | 4.1999998 | 1.2000000 | 5.8294215 | 2.8842874 | 4.0622377 | 1.3185704 |
| 5.6999998 | 2.9000001 | 4.1999998 | 1.3000000 | 5.7950406 | 2.8375967 | 4.1039581 | 1.3398128 |
| 6.1999998 | 2.9000001 | 4.3000002 | 1.3000000 | 6.1287012 | 3.0237744 | 4.2898021 | 1.3946648 |
| 5.0999999 | 2.5000000 | 3 | 1.1000000 | 4.9865527 | 2.6083987 | 3.1625361 | 0.98991424 |
| 5.6999998 | 2.8000000 | 4.0999999 | 1.3000000 | 5.7336531 | 2.8116338 | 4.0514202 | 1.3216151 |
| 7.4000001 | 2.8000000 | 6.0999999 | 1.9000000 | 7.2698121 | 3.1952546 | 5.9549327 | 2.0371470 |
| 7.9000001 | 3.8000000 | 6.4000001 | 2 | 8.0408592 | 3.7264807 | 6.1609025 | 2.0651569 |
| 6.4000001 | 2.8000000 | 5.5999999 | 2.2000000 | 6.5478463 | 2.8314803 | 5.4663453 | 1.8802570 |
| 6.3000002 | 2.8000000 | 5.0999999 | 1.5000000 | 6.3418117 | 2.9133918 | 4.9159150 | 1.6538914 |
| 6.0999999 | 2.5999999 | 5.5999999 | 1.4000000 | 6.2484961 | 2.7086525 | 5.2017884 | 1.7878238 |
| 7.6999998 | 3 | 6.0999999 | 2.3000000 | 7.5515795 | 3.3538170 | 6.1089063 | 2.0821571 |
| 6.3000002 | 3.4000001 | 5.5999999 | 2.4000001 | 6.7270894 | 3.0005608 | 5.4133439 | 1.8421952 |
| 6.4000001 | 3.0999999 | 5.5000000 | 1.8000000 | 6.6239829 | 2.9922800 | 5.2469258 | 1.7770897 |

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 6 | 3 | 4.8000002 | 1.8000000 | 6.1780910 | 2.8787041 | 4.6993270 | 1.5715717 |
| 6.9000001 | 3.0999999 | 5.4000001 | 2.0999999 | 6.9256096 | 3.1752450 | 5.3824821 | 1.8123403 |
| 6.6999998 | 3.0999999 | 5.5999999 | 2.4000001 | 6.8595986 | 3.0456672 | 5.5509524 | 1.8921692 |
| 6.9000001 | 3.0999999 | 5.0999999 | 2.3000000 | 6.8665690 | 3.1927600 | 5.2379360 | 1.7532986 |
| 5.8000002 | 2.7000000 | 5.0999999 | 1.9000000 | 6.0095687 | 2.6419284 | 4.9213419 | 1.6834334 |
| 6.8000002 | 3.2000000 | 5.9000001 | 2.3000000 | 7.0240221 | 3.0965700 | 5.7329159 | 1.9591335 |
| 6.6999998 | 3.3000000 | 5.6999998 | 2.5000000 | 6.9641590 | 3.1004534 | 5.6170616 | 1.9128360 |
| 6.6999998 | 3 | 5.1999998 | 2.3000000 | 6.7259831 | 3.0640202 | 5.2709308 | 1.7793626 |
| 6.3000002 | 2.5000000 | 5 | 1.9000000 | 6.2204537 | 2.7761803 | 5.0021095 | 1.7018900 |
| 6.5000000 | 3 | 5.1999998 | 2 | 6.5883756 | 3.0116322 | 5.1403003 | 1.7328883 |
| 6.1999998 | 3.4000001 | 5.4000001 | 2.3000000 | 6.6152620 | 2.9934883 | 5.2286267 | 1.7697171 |
| 5.9000001 | 3 | 5.0999999 | 1.8000000 | 6.1837134 | 2.8037276 | 4.8753185 | 1.6488713 |

### 4.5 Experience

Conducting this experiment have strengthened my understanding of neural networks. Having coded the back-propagation algorithm on my own, I had a deeper impression on the details of the algorithm and the influence of those parameters.

### 4.6 Part of Code

The whole version of code can be seen on:
https://github.com/michaelHaha/BUAA_PRML_Experiment

**Train_NN.m:**

```
function weight = Train_NN(train_x, train_y, val_x, val_y, neuron_num,
learning_rate, batch_size, num_Epoches, learning_rate_decay, weight_decay)
%%
%Inputs:
%training_set, labels: train_x, train_label
%hyperparameters: learning_rate, batch_size, number of epoches to
%train, learning_rate_decay, weight_decay

%Outputs;
%Weights; both weight matrice and biases

%% preparation for training
% Ensure neuron number match input data
assert(size(train_x, 2) == neuron_num(1));
assert(size(train_y, 2) == neuron_num(end));

%get number of inputs and layers
Num_Inputs_trn = length(train_x);
Num_Inputs_val = length(val_x);
layer_num = length(neuron_num);
```

```
%compute iterations per epoch
iters_per_epoch = (Num_Inputs_trn/batch_size);

%outputs_before_activation of every layer, used for BP
outputs_before_activation = cell(1,layer_num);

%augment training set
train_x_aug = [train_x ones(Num_Inputs_trn, 1)];

%initiallize Augmented weights with Gaussian distribution with mean 0,
deviation 0.01
% Notice: weight{i} corresponds to the (i+1) layer
weight = cell(1, layer_num - 1);
for i = 1:layer_num-1
    if i ~= layer_num-1
        weight{i} = randn(neuron_num(i)+1, neuron_num(i+1)+1)/100;
        weight{i}(neuron_num(i)+1, :) = 0;
    else
        weight{i} = randn(neuron_num(i)+1, neuron_num(i+1))/100;
        weight{i}(neuron_num(i)+1, :) = 0;
    end
end
% for i = 1:layer_num-1
%     if i ~= layer_num-1
%         weight{i} = zeros(neuron_num(i)+1, neuron_num(i+1)+1)/10;
%     else
%         weight{i} = zeros(neuron_num(i)+1, neuron_num(i+1))/10;
%     end
% end

% D_weight: restore the gradient for each weight matrix in weight(layer: 1 -
layer_num-1)
D_weight = cell(1,layer_num-1);

% D_output:restore the gradient for output in each layer(layer: 2 - layer_num-
1)
D_output = cell(1,layer_num-1);
% for i = 1: layer_num-1
%     D_weight{i} = zeros(neuron_num(i)+1, neuron(i+1));
% end

% record:
loss = 0.0;
```

```
accuracy = 0.0;
global_step = 0;

trn_acc_rec = zeros(num_Epoches,1);
trn_los_rec = zeros(num_Epoches,1);
val_acc_rec = zeros(num_Epoches,1);
val_los_rec = zeros(num_Epoches,1);

%% start training
fprintf('Training...\n');

for epoch = 1 : num_Epoches
    %shuffle data every epoch
    shuffle = randperm(size(train_x_aug, 1));
    train_x_aug_shuf = train_x_aug(shuffle,:);
    train_y_shuf = train_y(shuffle, :);
    fprintf('strating training %d epoch\n', epoch);

    acc_per_epoch = 0;
    los_per_epoch = 0;

    for iter = 1:iters_per_epoch
        % get batch and its labels
        batch = train_x_aug_shuf(((batch_size*(iter-
1))+1):(batch_size*(iter)), :);
        y = train_y_shuf(((batch_size*(iter-1))+1):(batch_size*(iter)), :);

        %feed forward and record inputs(outputs) of each stage
        outputs_before_activation{1} = batch;
        for i = 2: layer_num
            %first feed the outputs of last layer to activation function
            %and then feed it to this layer
            if i == 2
                outputs_before_activation{i} = (outputs_before_activation{i-
1})*weight{i-1};
            else
                outputs_after_activation =
activation_Fn(outputs_before_activation{i-1});
%                 outputs_after_activation = [outputs_after_activation
ones(length(outputs_after_activation), 1)];
                outputs_before_activation{i} = outputs_after_activation *
weight{i-1};
            end
        end
```

```
        % output and compute Loss
        output = outputs_before_activation{layer_num};
        % two kinds of loss: sigmoid loss(one-class classification) and
        % cross-entropy loss(multi-class classification)

        if size(output, 2) == 1
            %accuracy
            accuracy = (sum((output > 0) == y))/batch_size;

            %sigmoid loss
            loss = sum((log(sigmoid(output))).*y)+sum((ones(size(y))-
y).*(log(ones(size(output))-sigmoid(output))));
            loss = -loss/batch_size;

            % preparing update for the weights
            delta_output = (output-y)/batch_size;
            D_output{end} = delta_output;
        else
            %accuracy
            [~,max_position] = max(output, [], 2);
            [~, max_position_y] = max(y, [], 2);
            accuracy = sum(max_position == max_position_y)/batch_size;

            %cross-entropy loss
            P = exp(output);
            P_sum = sum(exp(output), 2);
            for i = 1: size(output, 2)-1
                P_sum = [P_sum sum(exp(output), 2)];
            end
            P = P./P_sum;
            loss = sum(-log(sum((y.* P), 2)))/batch_size;

            %preparing update for the weights
            P_cut = y;
            delta_output = (P-P_cut)/batch_size;
            D_output{end} = delta_output;
        end
        % need to add L-2 loss to the total loss
        for i = 1: layer_num-1
            dis = weight{i}.^2;
            loss = loss+sum(dis(:))*0.5*weight_decay;
        end
```

```
        %back propagation: compute the gradient for each weight matrix
        for i = layer_num-1:-1:1
            if i ~= 1
                outputs_after_activation =
activation_Fn(outputs_before_activation{i});
%                 outputs_after_activation = [outputs_after_activation
ones(length(outputs_after_activation), 1)];
            else
                outputs_after_activation = outputs_before_activation{i};
            end
            D_weight{i} = outputs_after_activation' * D_output{i};
            D_weight{i} = D_weight{i} + weight_decay * weight{i};
            if i ~= 1
                D_output{i-1} = D_output{i} * (weight{i}');
                %let the gradient pass through the activation function:
                %relu function, partial derivative is 1 for positive inputs
                %and 0 for negative outputs
                D_output{i-1} = (outputs_after_activation>0) .* D_output{i-1};
            end
        end

        %update weights
        for i = 1:layer_num-1
            weight{i} = weight{i} - learning_rate * D_weight{i};
        end

        %record total iterations
        global_step = global_step+1;

        %report result every 100 iterations
        if mod(global_step, 2) == 0
            fprintf('step: %d, learning rate: %f, accuracy: %f, loss: %f\n',
[global_step, learning_rate, accuracy, loss]);
        end

        %record accuracy and loss for this iteration
        acc_per_epoch = acc_per_epoch + accuracy;
        los_per_epoch = los_per_epoch + loss;
    end

    % AFTER ONE EPOCH
    % learning rate decay: learning rate decay every 2 epoches
    if mod(epoch, 20) == 0
        learning_rate = learning_rate*learning_rate_decay;
```

```
    end

    %test on validation set for evaluation every epoch
    x = [val_x ones(length(val_x),1)];
    for i = 1: layer_num-1
        %augment data
        x = x*weight{i};
        if i ~= layer_num-1
            x = activation_Fn(x);
        end
    end

    %compute loss and accuracy
    if size(x, 2) == 1
        %accuracy
        accuracy_val = (sum((x > 0) == val_y))/Num_Inputs_val;
        %loss
        loss_val = sum((log(sigmoid(x))).*val_y)+sum((ones(size(val_y))-
val_y).*(log(ones(size(x))-sigmoid(x))));
        loss_val = -loss_val/Num_Inputs_val;
    else
        %accuracy
        [~, m1] = max(x, [], 2);
        [~, m2] = max(val_y, [], 2);
        accuracy_val = sum(m1==m2)/Num_Inputs_val;
        %cross-entropy loss
        P = exp(x);
        P_sum = sum(exp(x), 2);
        for i = 1: size(x, 2)-1
            P_sum = [P_sum sum(exp(x), 2)];
        end
        P = P./P_sum;
        loss_val = sum(-log(sum((val_y.* P), 2)))/Num_Inputs_val;
    end
    %print the result
    fprintf('result of %d epoches: val accuracy: %f, val loss: %f\n', [epoch,
accuracy_val, loss_val]);

    %record the trn accuracy/loss and validation accruacy/loss
    trn_acc_rec(epoch) = acc_per_epoch/iters_per_epoch;
    trn_los_rec(epoch) = los_per_epoch/iters_per_epoch;
    val_acc_rec(epoch) = accuracy_val;
    val_los_rec(epoch) = loss_val;
end
```

```
%% When finish training, plot train and validation record
%plot(trn_los_rec); hold on; title('training process with different learning
rate'); legend(['lr:' num2str(learning_rate)]);
% figure(1);
% plot(trn_acc_rec); hold on; plot(trn_los_rec); title('Train accuracy and loss
history'); legend('Accuracy', 'Loss');xlabel('Epoches');
% figure(2);
% plot(val_acc_rec); hold on; plot(val_los_rec); title('Validation accuracy and
loss history'); legend('Accuracy', 'Loss');xlabel('Epoches');
End
```

Iris.m:

```
%% load Iris dataset and partition data
dataset = load('./dataset/Iris.txt');
% partition the dataset to training set, validation set and test set with
% proportion of 3:1:1 homogeneously
data = dataset(:, 1:4);
label = dataset(:, 5);
y = zeros(150, 3);
for i = 1:150
    y(i, label(i)) = 1;
end

train_x = [data(1:30, :); data(51:80, :); data(101:130, :)];
train_y = [y(1:30, :); y(51:80, :); y(101:130, :)];
val_x = [data(31:40, :); data(81:90, :); data(131:140, :)];
val_y = [y(31:40, :); y(81:90, :); y(131:140, :)];
test_x = [data(41:50, :); data(91:100, :); data(141:150, :)];
test_y = [y(41:50, :); y(91:100, :); y(141:150, :)];

%% train neural network
%set hyper-parameters
neuron_num = [4 6 3];
learning_rate = 0.5;
batch_size = 5;
num_Epoches = 200;
learning_rate_decay = 1;
weight_decay = 5e-3;

%train the network
weight = Train_NN(train_x, train_y, val_x, val_y, neuron_num, learning_rate,
batch_size, num_Epoches, learning_rate_decay, weight_decay);
```

```
% test on test_set
x = [test_x ones(length(test_x),1)];
layer_num = size(neuron_num, 2);

for i = 1: layer_num-1
    %augment data
    x = x*weight{i};
    if i ~= layer_num-1
        x = activation_Fn(x);
    end
end
%compute loss and accuracy
%accuracy
[~, m1] = max(x, [], 2);
[~, m2] = max(test_y, [], 2);
accuracy_tst = sum(m1==m2)/30;
%loss
P = exp(x);
P_sum = sum(exp(x), 2);
for i = 1: size(x, 2)-1
    P_sum = [P_sum sum(exp(x), 2)];
end
P = P./P_sum;
loss_tst = sum(-log(sum((test_y.* P), 2)))/30;
%print the result
fprintf('result on test set: test accuracy: %f, test loss: %f\n',
[accuracy_tst, loss_tst]);
```