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 模式识别与智能控制

学生学号 14031259

学生姓名 孔昭宁

2017 年 5 月 28 日

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 gi(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:

$$\mathrm{f(x)} = (\sum_i w_i g_i(x))$$

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 gi as simply a vector g = (g1, g2, ..., gn). 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

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

The input net to a neuron is the weighted sum of outputs ok 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 φ 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

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

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

I conducted this experiment with Matlab, and visualizes the network with the Graphviz package. In order to achieve the best training results, a series of optimizing methods were used, including weight decay, adaptive learning rate, mean normalization, early stopping, etc.

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 all three datasets on ANNs of different size and number of layers. The results are as follows.

1) 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, whose output under

parameter θ is as follows.

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

where z_i is the input to node i. For the Iris dataset, we have K=3.

The output of the softmax function can be used to represent a categorical distribution.

I chose the cost function for the softmax output layer as:

$$J(\theta) = -\left[\sum_{i=1}^{m}\sum_{k=1}^{K} 1\{y^{(i)} = k\} log \frac{\exp\left(\theta^{(k)}\mathrm{x}^{(i)}\right)}{\sum_{j=1}^{k}\exp(\theta^{(j)}\mathrm{x}^{(i)})}\right] + \frac{\lambda}{2}\sum_{l=1}^{L-1}\sum_{i=1}^{s_l}\sum_{j=1}^{s_{l+1}}(\theta_{ji}^{(l)})^2$$

$$= -\left[\sum_{i=1}^{m}\sum_{k=1}^{K} 1\{y^{(i)} = k\} P(y^{(i)} = k|x^{(i)}, \theta)\right] + \frac{\lambda}{2}\sum_{l=1}^{L-1}\sum_{i=1}^{s_l}\sum_{j=1}^{s_{l+1}}(\theta_{ji}^{(l)})^2$$

where $1\{y^{(i)} = k\} = \begin{cases} 1\ if\ y^{(i)} = k \\ 0\ if\ y^{(i)} \neq k \end{cases}$.

In order to prevent the neural network from overfitting due to exploding parameters, a L-2 regularization term is appended to the cost function.

Taking the derivative w.r.t. θ of the function above yields

$$\frac{\partial J(\theta)}{\partial \theta} = -\sum_{i=1}^{m}[x^{(i)}\left(1\{y^{(i)} = k\} - P\left(y^{(i)} = k\middle|x^{(i)}, \theta\right)\right) + \lambda\theta]$$

which takes the simple form of the difference of the one-hot representation of a class, and the output of the softmax layer, in addition to the regularization term.

In order to attain the best training results, several other practical optimization methods have been used.

1) Weight Decay

In order to prevent the network from overfitting, an L-2 regularization term as follows is added to the cost function

$$\frac{\lambda}{2}\sum_{l=1}^{L-1}\sum_{i=1}^{s_l}\sum_{j=1}^{s_{l+1}}(\theta_{ji}^{(l)})^2$$

which penalizes the exceedingly big weight parameters.

2) Adaptive Learning Rate

A large learning rate expedites the training process, but fails locate an accurate minimum, whereas a small learning rate does just the opposite. To attain a balance between training speed and accuracy, an adaptive learning rate is generated as follows

$$\alpha = \alpha \times 0.99^n$$

where n increases by 1 after every 100 epochs. Therefore, the learning rate decays by 1% after every 100 epochs.

3) Mean Normalization

The gradient descent algorithm converges faster when all input parameters are on a similar scale. Therefore, the input parameters are preprocessed as follows:

$$X_i := \frac{X_i - \mu_i}{\sigma_i}$$

which normalizes each input feature with its mean and standard deviation.

4) Early Stopping

For some datasets, the network begins to over-fit after having trained for many epochs. Thus, a validation dataset was introduced to monitor the actual performance of the network. The training will be terminated, if error on the validation set does not decrease, for 100 epochs consecutively.

5) F1 Score

For binary-classification tasks, accuracy fails to measure the performance of the network when the dataset is very skewed. That is, there are considerably more samples of one class than the other. Such a case is very common in circumstances such as cancer datasets – only a fraction of patients actually has malign cancer. Consequently, a few other indices were introduced:

	Actual Class		
Predicted Class		1	0
	1	True Positive	False Positive
	0	False Negative	True Negative

From the terms defined in the table above, we have:

$$precision = \frac{\#(True\ Positive)}{\#(True\ Positive) + \#(False\ Positive)}$$

$$recall = \frac{\#(True\ Positive)}{\#(True\ Positive) + \#(False\ Negative)}$$

There has been a trade-off between precision and recall. For different tasks, we attach different importance to precision and recall. For instance, for cancer diagnosis tasks we desire a lower precision and higher recall, which avoids missing too many cases of cancer.

To compare precision and recall numbers, a unified F1 score is introduced as

follows:

$$F_1\ Score = 2\frac{precision \times recall}{precision + recall}$$

The network used to train this dataset is as follows

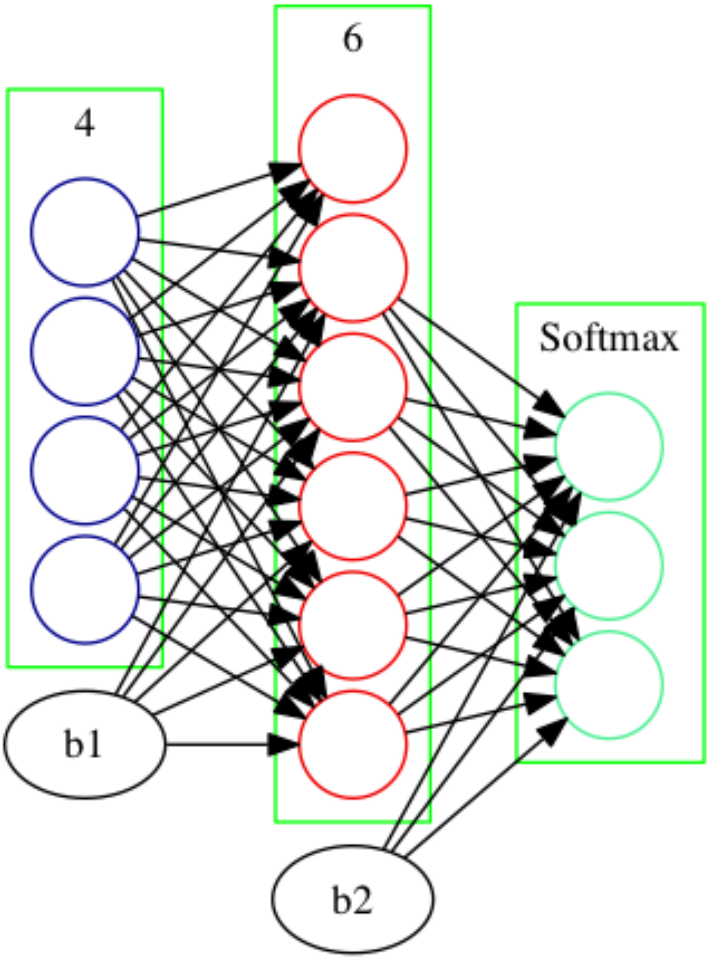

Figure 1 Iris Dataset Network Configuration

The specific parameters are chosen as:

- Proportion of training data: 80%
- Learning rate α: 0.5
- Learning rate decay: 0.99
- Weight decay λ: 0.1

Having trained for 2000 epochs, the training curve is plotted as follows.

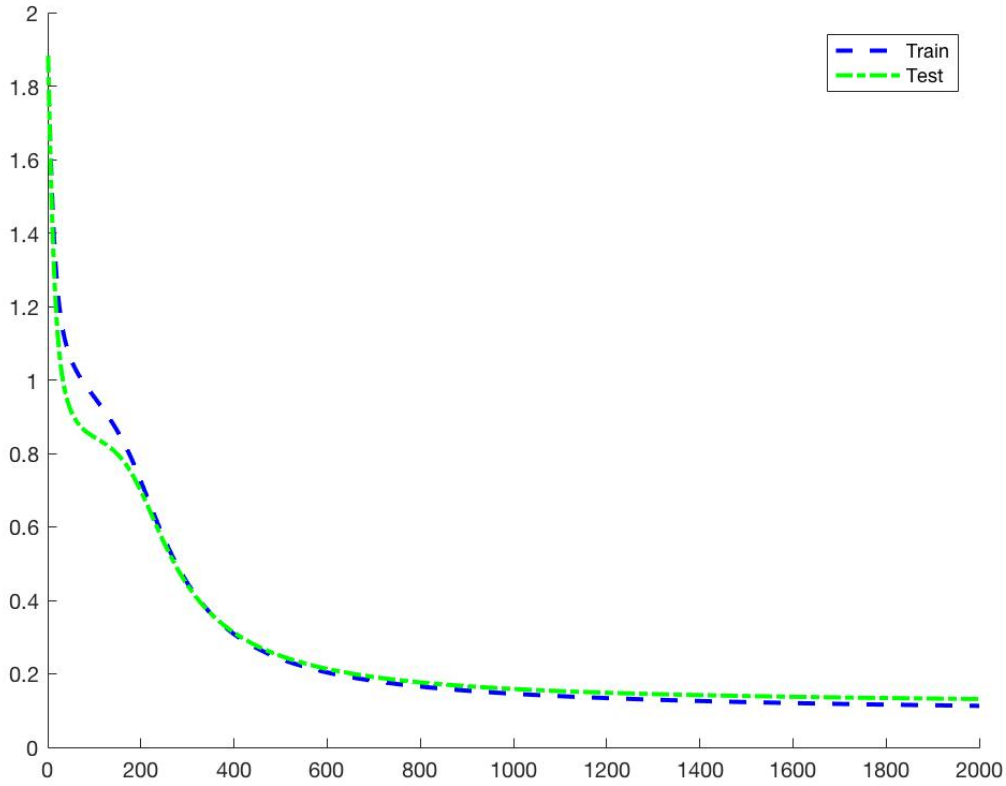

Figure 2 Iris Dataset Training Curve

The performance specifications are as follows:

$$J(\theta)_{train} = 0.112867$$

$$Accuracy(\theta)_{train} = 97.5\%$$

$$J(\theta)_{test} = 0.131480$$

$$Accuracy(\theta)_{test} = 100.0\%$$

2) The Pima Indian dataset

The Pima-Indian dataset is a binary classification task. The neural network configuration for this dataset is as follows, with two hidden layers.

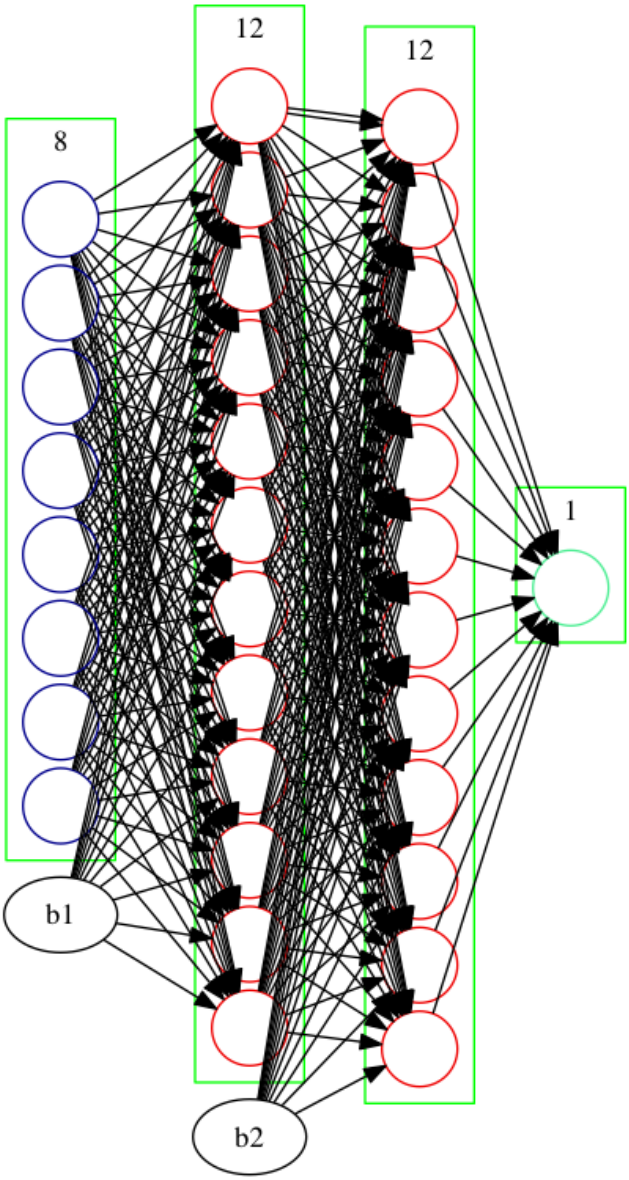

Figure 3 Pima Dataset Network Configuration

The network trained for 19568 epochs, before the early stopping mechanism was triggered. The training curve is as follows.

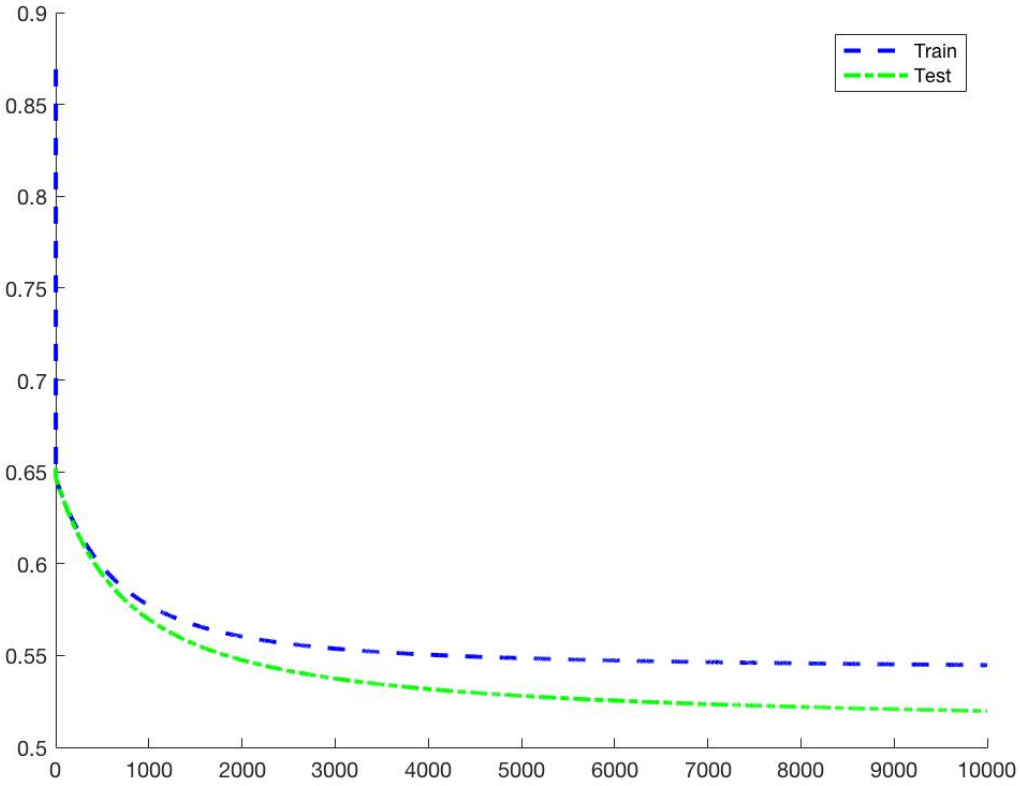

Figure 4 Pima Dataset Training Curve

The following results was attained.

$$J(\theta)_{train} = 0.495012$$

$$Accuracy(\theta)_{train} = 75.08\%$$

$$J(\theta)_{test} = 0.507420$$

$$Accuracy(\theta)_{test} = 74.03\%$$

$$F1\ Score = 0.6296$$

3) The Wisconsin Cancer dataset

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

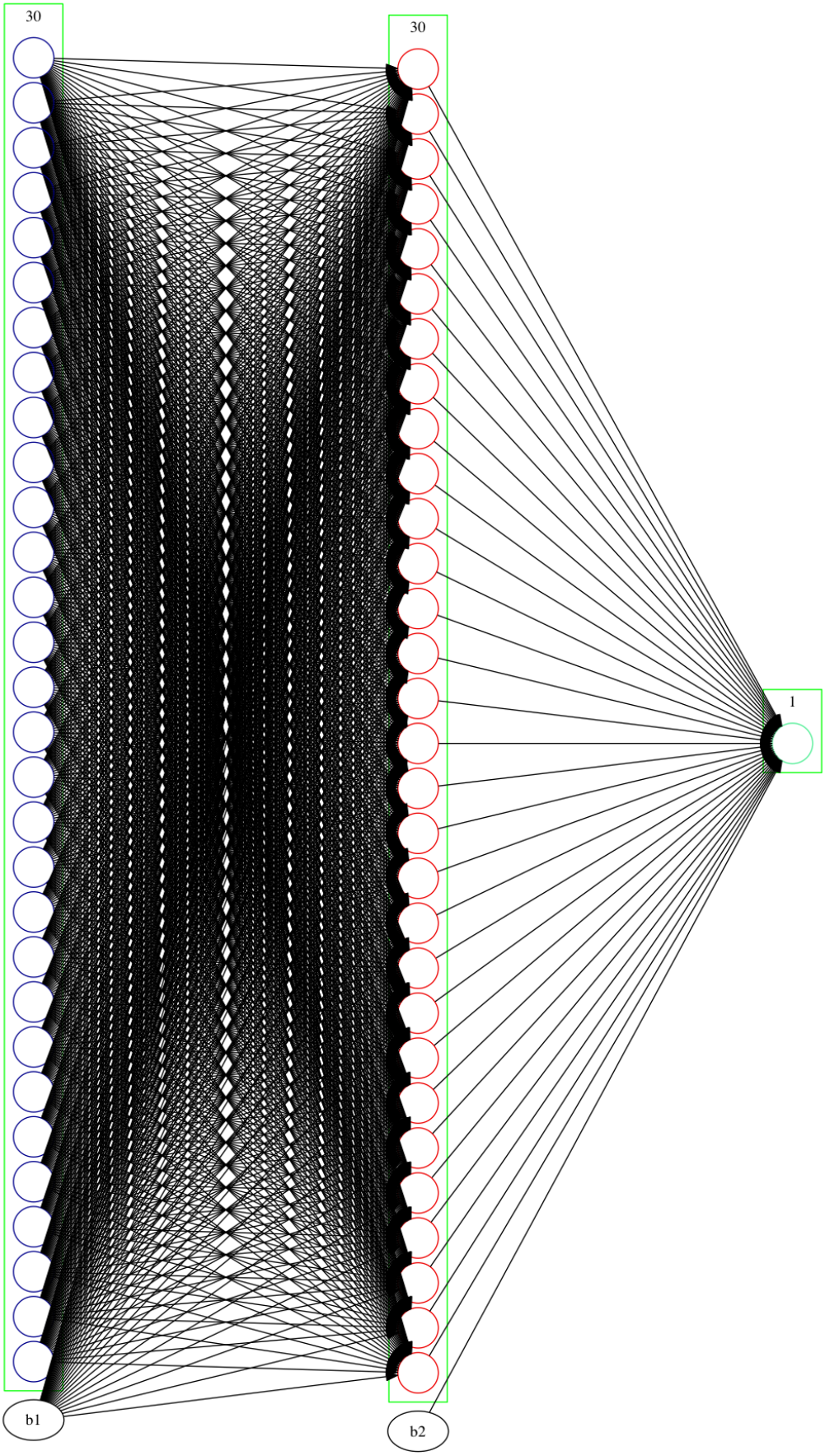

Figure 5 WDBC Dataset Network Configuration

The network trained for 284 epochs before the early-stopping mechanism intervened. The training curve is as follows.

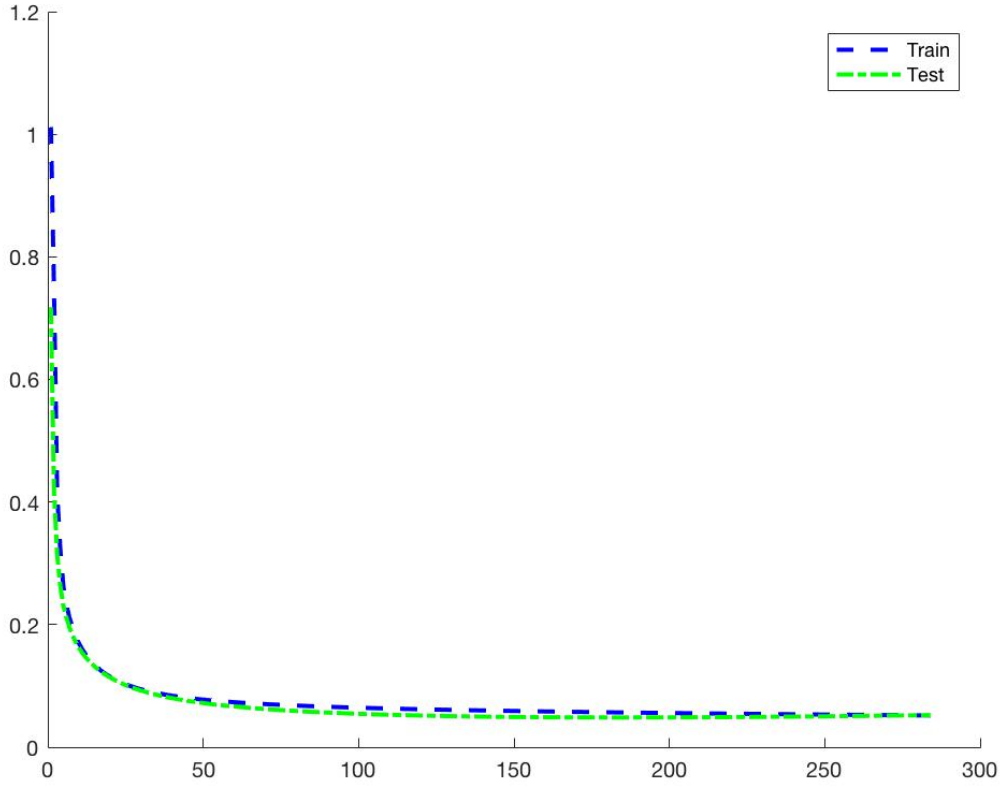

Figure 6 WDBC dataset training curve

The following results was attained.

$$J(\theta)_{train} = 0.051771$$

$$Accuracy(\theta)_{train} = 98.90\%$$

$$J(\theta)_{test} = 0.052349$$

$$Accuracy(\theta)_{test} = 100.0\%$$

$$F1\ Score = 1.000$$

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

I trained the Iris dataset with the perceptron provided by the Matlab built-in Neural Network Training Toolbox (nntraintool). The perceptron is a single-layer linear neuron depicted by the following graph.

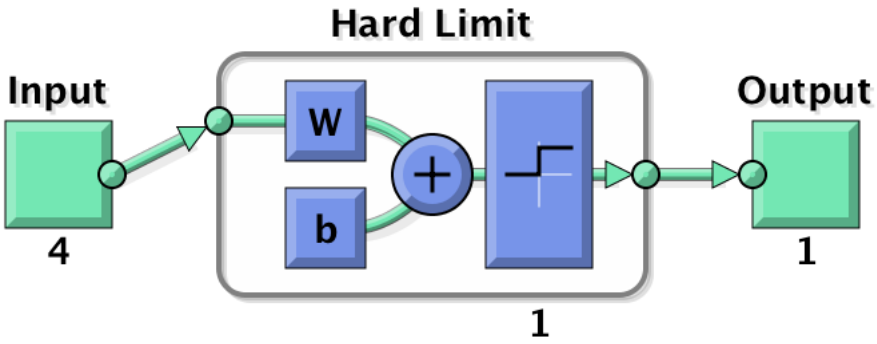

Figure 7 Perceptron

Having trained for 1000 epochs, the perceptron attained an accuracy of 82.33%, which is considerably lower than that of the one-hidden-layer neural network, which reached an accuracy of 100% on the test set.

Comparing the results of the neural network and perceptron, we can find that the neural network provides a much higher training accuracy, and is able to handle more complex datasets.

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

Having disabled learning rate decay and early-stopping, I trained the Iris dataset on the neural network for 3000 epochs, varying the learning rate. The cost on the test set is as follows.

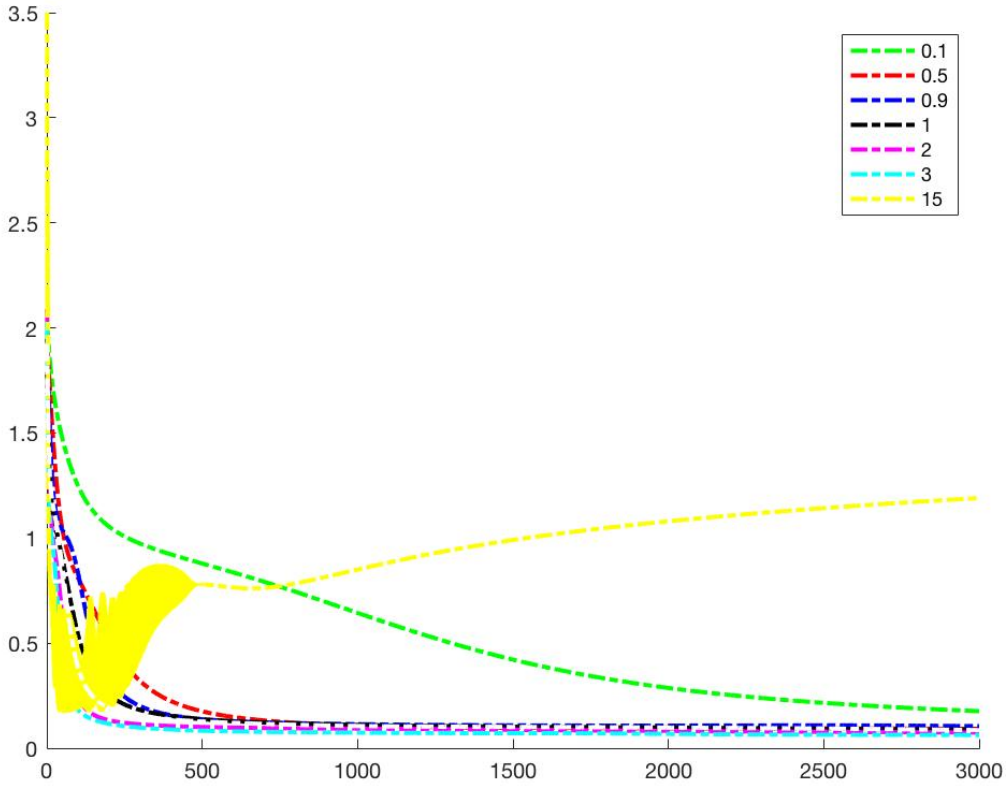

Figure 8 Performance under different learning rate

The accuracy on the test set is as follows.

α	0.1	0.5	0.9	1.0	2.0	3.0	15.0
Acc	98.7%	100.0%	100.0%	100.0%	100.0%	100.0%	90.0%

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.

Stage 2

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

Apart from the sigmoid activation function, I also used the purelin and tanh activation function. Both the two activation functions have very neat derivatives, which can be

determined by their output values.

$$f_1(x) = \tanh(x) = \frac{2}{1 + e^{-2x}} - 1$$

$$\frac{\partial f_1(x)}{\partial x} = 1 - f_1(x)^2$$

$$f_2(x) = x$$

$$\frac{\partial f_2(x)}{\partial x} = 1$$

The performance of the two activation functions on the Iris dataset is as follows.

	purelin	tanh
$J(\theta)_{train}$	1.135236	0.112827
$Accuracy(\theta)_{train}$	90.00%	100.0%
$J(\theta)_{test}$	1.157593	0.120325
$Accuracy(\theta)_{test}$	88.67%	100.0%

As it can be observed from the results, the tanh has a comparable performance on the Iris dataset with the sigmoid activation function. However, the purelin function performs poorly on the dataset, reaching an accuracy of only 88.67%.

(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 hold the view that although artificial neural networks were originally inspired by biological neural networks, there are considerable disparities between the two.

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, the mere fact that their structures are common does not necessarily prove that they work the same way. Biological neural networks perform in ways that people are yet

to discover, whereas the algorithms for artificial neural networks are deterministic. Meanwhile, despite the fact that artificial neural networks attain desirable results that other methods fails to achieve, the gap between neural networks and biological ones are still huge.

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

I trained the autoencoder with the Iris dataset. The network configuration for the autoencoder used is as follows.

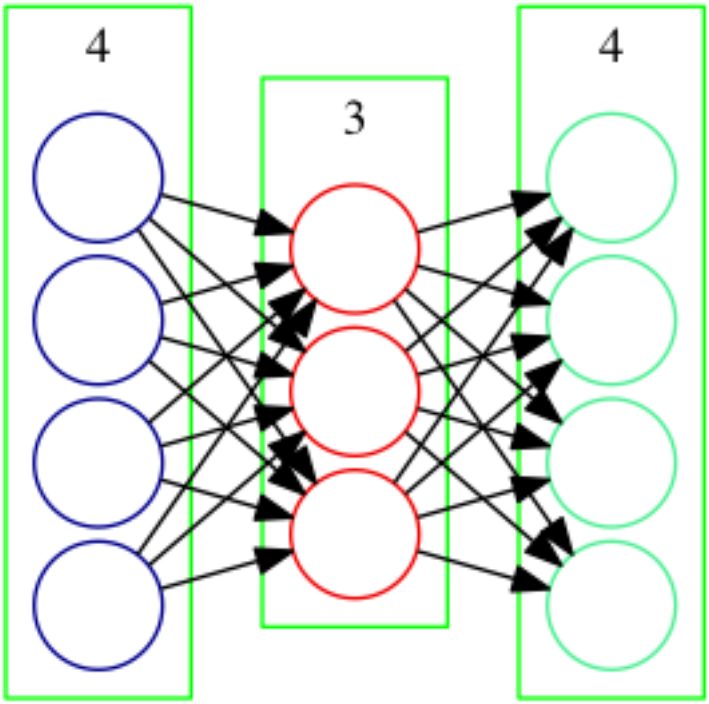

Figure 9 Network Configuration for Autoencoder

The input and output of the dataset are identical. However, in order to attain a better training result, they are normalized in different ways.

In order to expedite the training process, the input data are normalized by its mean and standard deviation.

$$x_{input} := \frac{x - \mu(x)}{\sigma(x)}$$

With the last layer being the sigmoid output function, the output data are normalized by its extremum values to [0, 1].

$$x_{output} := \frac{x - min(x)}{max(x) - min(x)}$$

Having trained for 10000 epochs, the loss on both the training set and validation set reached a desirable value. The learning curve for the autoencoder is as follows.

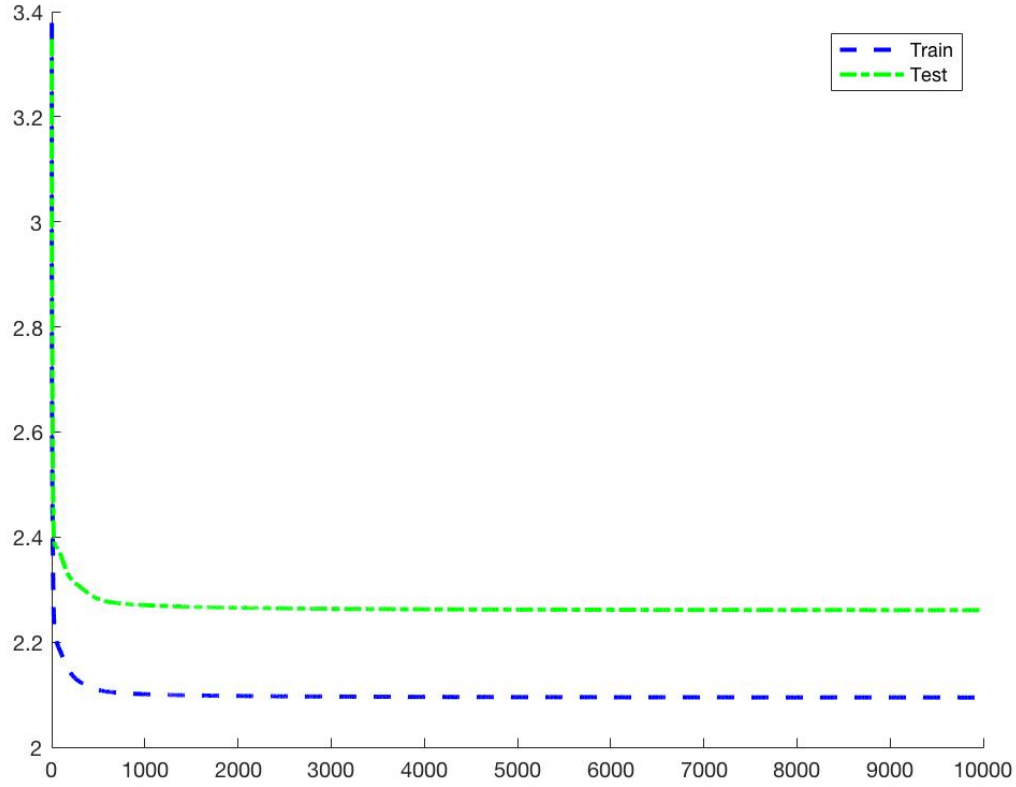

Figure 10 Learning Curve for Autoencoder

The results are as follows.

$$J(\theta)_{train} = 2.094629$$

$$J(\theta)_{test} = 2.260796$$

The performance on the test dataset is as follows.

	SL	SW	PL	PW		SL	SW	PL	PW
Input	6.9	3.2	5.7	2.3	Output	6.9305	3.2603	6.0065	2.2284
	5.6	2.8	4.9	2		5.4925	2.8158	5.0442	2.038
	7.7	2.8	6.7	2		7.4531	2.8357	6.221	2.0743
	6.3	2.7	4.9	1.8		6.2826	2.6826	5.3186	1.8598
	6.7	3.3	5.7	2.1		6.8397	3.3371	5.7841	2.1517
	7.2	3.2	6	1.8		7.3154	3.2332	5.8178	1.9385
	6.2	2.8	4.8	1.8		6.1698	2.7785	5.2075	1.872
	6.1	3	4.9	1.8		6.1256	2.9873	5.1195	1.913
	6.4	2.8	5.6	2.1		6.4754	2.8328	5.7522	2.1162

	7.2	3	5.8	1.6		7.263	3.0021	5.599	1.709
	7.4	2.8	6.1	1.9		7.3096	2.8132	6.0315	1.984
	7.9	3.8	6.4	2		7.6065	3.8079	5.9427	1.9981
	6.4	2.8	5.6	2.2		6.4336	2.8566	5.8008	2.16
	6.3	2.8	5.1	1.5		6.4643	2.7403	5.0303	1.6075
	6.1	2.6	5.6	1.4		6.3646	2.5375	5.0514	1.5587
	7.7	3	6.1	2.3		7.4432	3.0749	6.3091	2.2131
	6.3	3.4	5.6	2.4		6.2553	3.4443	5.7156	2.2725
	6.4	3.1	5.5	1.8		6.6201	3.0965	5.4351	1.9571
	6	3	4.8	1.8		5.9884	2.9866	5.027	1.9099
	6.9	3.1	5.4	2.1		6.9433	3.1421	5.8636	2.1186
	6.7	3.1	5.6	2.4		6.6764	3.1825	5.9735	2.2559
	6.9	3.1	5.1	2.3		6.8012	3.172	5.9131	2.1956
	5.8	2.7	5.1	1.9		5.7509	2.6996	5.1629	1.9683
	6.8	3.2	5.9	2.3		6.884	3.2567	6.007	2.2394
	6.7	3.3	5.7	2.5		6.6374	3.3714	5.9766	2.2938
	6.7	3	5.2	2.3		6.624	3.072	5.8719	2.198
	6.3	2.5	5	1.9		6.2227	2.5327	5.4523	1.9023
	6.5	3	5.2	2		6.5506	3.019	5.6006	2.0577
	6.2	3.4	5.4	2.3		6.16	3.4394	5.5711	2.2365
	5.9	3	5.1	1.8		5.9594	2.9809	5.0724	1.9439

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.

Meanwhile, in order to boost the performance of the network, I fine-tuned the network and training parameters for specific datasets, which enhanced my experience on designing neural network architectures.

4.6 Codes

4.6.1 ex4.m

```
%% Set Parameters and load data
% load data/Iris.txt
% data = Iris;
% neuron_num = [4, 6, 3];
% load data/Pima_Indians.txt
% data = Pima_Indians;
% neuron_num = [8, 12, 12, 1];
```

```
% load data/wdbc.txt
% data = wdbc;
% neuron_num = [30, 30, 1];
train_prop = 0.8;
learning_rate = 10;
learning_rate_decay = 1.00;     % Learning rate decay after every 100 iters
lambda = 0.00;                   % Weight Decay
epoch_num = 3000;
softmax = false;
if(neuron_num(end) > 1)
    softmax = true;
end

%% Preprocess data
% Randomly Shuffle dataset
data = data(randperm(size(data, 1)), :);
x = data(:, 1 : end - 1);
y = data(:, end);
% Regularize x data
x = (x - mean(x)) ./ std(x);
% Transform to multivariate form
if(softmax)
    y_new = zeros(length(y), neuron_num(end));
    for i = 1 : length(y)
        y_new(i, y(i)) = 1;
    end
    y = y_new;
end
% Separate into train and test data
train_x = x(1 : int32(size(x, 1) * 0.8), :);
test_x = x(int32(size(x, 1) * 0.8) + 1 : end, :);
train_y = y(1 : int32(size(y, 1) * 0.8), :);
test_y = y(int32(size(y, 1) * 0.8) + 1 : end, :);

%% Train neural network
train_nn(train_x, train_y, test_x, test_y, neuron_num, learning_rate, epoch_num,
learning_rate_decay, lambda);
```

4.6.2 ex4_perceptron.m

```
%% Load data
load data/Iris.txt
data = Iris;

% Randomly Shuffle dataset
data = data(randperm(size(data, 1)), :);

% Separate into x and y data
x = data(:, 1 : end - 1)';
t1 = data(:, end)';
t2 = t1;

% t1 sets the first class as 1, others 0
t1(t1 ~= 1) = 0;
% t2 sets the second class as 1, others 0
```

```
t2(t2 ~= 2) = 0;
t2(t2 == 2) = 1;

%% Train perceptron
accuracy = 0;

net1 = perceptron;
net1 = train(net, x, t1);
view(net1)
y1 = net1(x);

net2 = perceptron;
net2 = train(net, x, t2);
y2 = net2(x);
```

4.6.3 ex4_autoencoder.m

```
%% Generate dataset
load data/Iris.txt
data = Iris;
x = Iris(:, 1 : end - 1);
neuron_num = [4 3 4];

%% Train autoencoder
train_prop = 0.8;
learning_rate = 1.0;
learning_rate_decay = 0.99;
lambda = 0.00;                    % Weight Decay
epoch_num = 10000;

% Normalize dataset
maxX = max(x);
minX = min(x);
meanX = mean(x);
stdX = std(x);
y = (x - minX) ./ (maxX - minX);
x = (x - mean(x)) ./ std(x);

% Separate dataset
train_x = x(1 : int32(size(x, 1) * 0.8), :);
test_x = x(int32(size(x, 1) * 0.8) + 1 : end, :);
train_y = y(1 : int32(size(y, 1) * 0.8), :);
test_y = y(int32(size(y, 1) * 0.8) + 1 : end, :);

train_nn(train_x, train_y, test_x, test_y, neuron_num, learning_rate, epoch_num,
learning_rate_decay, lambda, maxX, minX, meanX, stdX);
```

4.6.4 train_nn.mx

```
function model = train_nn(train_x, train_y, test_x, test_y, neuron_num,
learning_rate, epoch_num, learning_rate_decay, lambda, maxX, minX, meanX, stdX)
% Whether in autoencoder mode
autoencoder = false;
if(nargin >= 10)
```

```
    autoencoder = true;
end
autoencoder_input = [];
autoencoder_output = [];

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

% Use softmax in multivariate conditions
softmax = false;
if(neuron_num(end) > 1 && autoencoder == false)
    softmax = true;
end

% Construct neural Network
layer_num = length(neuron_num);
% Construct input and activation vectors
input = cell(1, layer_num);
activation = cell(1, layer_num);
for i = 1 : layer_num
    input{i} = zeros(neuron_num(i), 1);
    activation{i} = zeros(neuron_num(i), 1);
end
% Construct the weight matrices(including a bias vector as the last column)
weight = cell(1, layer_num - 1);
for i = 1 : layer_num - 1
    % Initialize the weight to be a random number in [-e, e]
    % where e is determined by the number of input and output neurons
    epsilon_init = sqrt(6) / sqrt(neuron_num(i) + neuron_num(i + 1));
    weight{i} = rand(neuron_num(i + 1), neuron_num(i) + 1) * 2 * epsilon_init -
epsilon_init;
end
% Construct the delta vectors(with the first element being empty)
delta = cell(1, layer_num);
for i = 2 : layer_num
    delta{i} = zeros(neuron_num(i), 1);
end


% Train Neural Network
fprintf('Training...\n')
best_testset_cost = 100;
best_accuracy = 0;
num_not_improving = 0;
train_cost = [];
test_cost = [];
for epoch = 1 : epoch_num
    autoencoder_input = [];
    autoencoder_output = [];
    fprintf('Iteration %d\n', epoch);
    % Initialize the accumulative derivative matrix corresponding to each
    % weight matrix
    accu = cell(1, layer_num - 1);
    for i = 1 : layer_num - 1
        accu{i} = zeros(neuron_num(i + 1), neuron_num(i) + 1);
    end
```

```
    % Traverse the train dataset
    num_correct = 0;
    J = 0;
    for i = 1 : size(train_x, 1)
        % Forward Propagation
        input{1} = train_x(i, :)';
        activation{1} = input{1};
        for j = 2 : layer_num
            input{j} = weight{j - 1} * [activation{j - 1}; 1];
            activation{j} = sigmoid(input{j});
        end
        % Softmax
        if(softmax)
            sum_of_probs = sum(activation{layer_num});
            activation{layer_num} = activation{layer_num} / sum_of_probs;
        end
        % Back Propagation
        delta{layer_num} = activation{layer_num} - [train_y(i, :)]';
        for j = layer_num - 1 : 2
            % Compute the delta terms on previous layers
            delta{j} = (weight{j}(:, 1 : end - 1))' * delta{j + 1} .*
(activation{j} .* (1 - activation{j}));
        end
        % Accumulate the delta terms on corresponding layers
        for j = 1 : layer_num - 1
            accu{j} = accu{j} + delta{j + 1} * [activation{j}; 1]';
        end
        % Accumulate the cost for this sample
        J = J - (train_y(i, :) * log(activation{layer_num}) + (1 - train_y(i, :)) *
log(1 - activation{layer_num}));
        % Compute accuracy
        if(softmax)
            [~, maxPos] = max(activation{layer_num});
            activation{layer_num} = zeros(size(activation{layer_num}));
            activation{layer_num}(maxPos) = 1;
        else
            activation{layer_num}(activation{layer_num} > 0.5) = 1;
            activation{layer_num}(activation{layer_num} <= 0.5) = 0;
        end
        if(autoencoder == false)
            if(isequal(activation{layer_num}, train_y(i, :)'))
                num_correct = num_correct + 1;
            end
        end
    end
    J = J / size(train_x, 1);
    % Calculate Weight Decay
    for i = 1 : layer_num - 1
        J = J + (lambda / 2) * sum(sum(weight{i} .* weight{i}));
    end
    train_cost = [train_cost, J];
    fprintf('Training Set Cost: %f\n', J);
    if(autoencoder == false)
        num_correct = num_correct / size(train_x, 1);
        fprintf('Training Set Accuracy: %f\n', num_correct)
    end

    % Update the weights
```

```
    for i = 1 : layer_num - 1
        weight{i} = weight{i} - learning_rate * ((1 / size(train_x, 1)) * accu{i} +
lambda * [weight{i}(:, 1 : end - 1), zeros(size(weight{i}, 1), 1)]);
    end

    % Traverse the test dataset
    num_correct = 0;
    J = 0;
    true_positive = 0;
    false_positive = 0;
    false_negative = 0;
    true_negative = 0;
    for i = 1 : size(test_x, 1)
        % Forward propagation
        input{1} = test_x(i, :)';
        activation{1} = input{1};
        if(autoencoder)
            x_actual = test_x(i, :) .* (stdX) + meanX;
            if(isempty(autoencoder_input))
                autoencoder_input = x_actual;
            else
                autoencoder_input = [autoencoder_input; x_actual];
            end
        end
        for j = 2 : layer_num
            input{j} = weight{j - 1} * [activation{j - 1}; 1];
            activation{j} = sigmoid(input{j});
        end
        if(autoencoder)
            x_predict = (activation{layer_num})' .* (maxX - minX) + minX;
            if(isempty(autoencoder_output))
                autoencoder_output = x_predict;
            else
                autoencoder_output = [autoencoder_output; x_predict];
            end
        end
        % Last Layer is Softmax
        if(softmax)
            sum_of_probs = sum(activation{layer_num});
            activation{layer_num} = activation{layer_num} / sum_of_probs;
        end
        % Accumulate the cost for this sample
        J = J - (test_y(i, :) * log(activation{layer_num}) + (1 - test_y(i, :)) *
log(1 - activation{layer_num}));
        % Compute accuracy
        if(softmax)
            [~, maxPos] = max(activation{layer_num});
            activation{layer_num} = zeros(size(activation{layer_num}));
            activation{layer_num}(maxPos) = 1;
        else
            activation{layer_num}(activation{layer_num} > 0.5) = 1;
            activation{layer_num}(activation{layer_num} <= 0.5) = 0;
        end
        if(autoencoder == false)
            if(isequal(activation{layer_num}, test_y(i, :)'))
                num_correct = num_correct + 1;
            end
        end
```

```
        % Compute F1 score for two-class classification cases
        if(~softmax && ~autoencoder)
            if(activation{layer_num} == 1 && test_y(i, :) == 1)
                true_positive = true_positive + 1;
            elseif(activation{layer_num} == 1 && test_y(i, :) == 0)
                false_positive = false_positive + 1;
            elseif(activation{layer_num} == 0 && test_y(i, :) == 1)
                false_negative = false_negative + 1;
            else
                true_negative = true_negative + 1;
            end
        end
    end
    J = J / size(test_x, 1);
    % Calculate Weight Decay
    for i = 1 : layer_num - 1
        J = J + (lambda / 2) * sum(sum(weight{i} .* weight{i}));
    end
    test_cost = [test_cost, J];
    precision = true_positive / (true_positive + false_positive);
    recall = true_positive / (true_positive + false_negative);
    F1_score = (2 * precision * recall) / (precision + recall);
    fprintf('Test Set Cost: %f\n', J);
    if(autoencoder == false)
        num_correct = num_correct / size(test_x, 1);
        fprintf('Test Set Accuracy: %f\n', num_correct);
    end
    if(~softmax && ~autoencoder)
        fprintf('Test Set F1 Score: %f\n\n', F1_score);
    else
        fprintf('\n')
    end
    if(mod(epoch, 100) == 0)
        learning_rate = learning_rate * learning_rate_decay;
    end
    % Record the best test data performance
    if(J < best_testset_cost)
        best_testset_cost = J;
        num_not_improving = 0;
        if(autoencoder == false)
            best_accuracy = num_correct;
        end
    % Early Stopping
    else
        num_not_improving = num_not_improving + 1;
        if(num_not_improving >= 10000)
            fprintf('Early Stopped at epoch %d\n', epoch);
            break
        end
    end
end

% Display results
if(autoencoder == false)
    fprintf('Best Test Accuracy: %f\n', best_accuracy);
end

% Plot learning curves
```

```
figure(1)
hold on
x = 1 : length(train_cost);
plot(x, train_cost, '--b', 'LineWidth', 2);
plot(x, test_cost, '-.g', 'LineWidth', 2);
legend('Train', 'Test')
% plot(x, test_cost, '-.g', 'LineWidth', 2);
% legend(num2str(learning_rate))

% Print autoencoder to excel
headers = ['a1', 'a2', 'a3', 'a4'];
xlswrite('input.xlsx', autoencoder_input)
xlswrite('output.xlsx', autoencoder_output)

% Save neural network
model.weight = weight;
end
```