

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:

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:

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 is defined as

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 of the input layer are simply the inputs to the network. The number of input units to the neuron is n. The variable denotes the weight between neurons i and j.

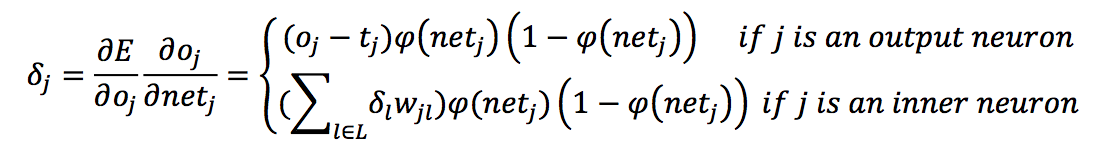
The activation function is in general non-linear and differentiable. A commonly used activation function is the logistic function, e.g.

which has a nice derivative of:

Calculating the partial derivative of the error with respect to a weight is done using the chain rule twice:

We can finally yield:

with



**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 is as follows.

where 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:

where .

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. of the function above yields

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

which penalizes the exceedingly big weight parameters.

1. 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

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

1. 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:

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

1. 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.

1. 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:

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:

The network used to train this dataset is as follows

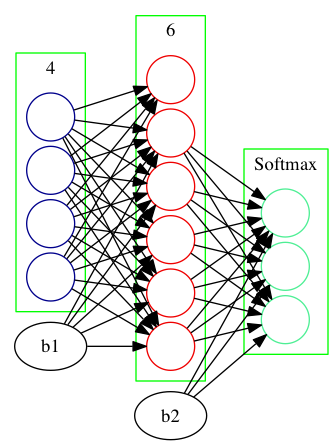


Figure 1 Iris Dataset Network Configuration

The specific parameters are chosen as:

* Proportion of training data: 80%
* Learning rate : 0.5
* Learning rate decay: 0.99
* Weight decay : 0.1

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

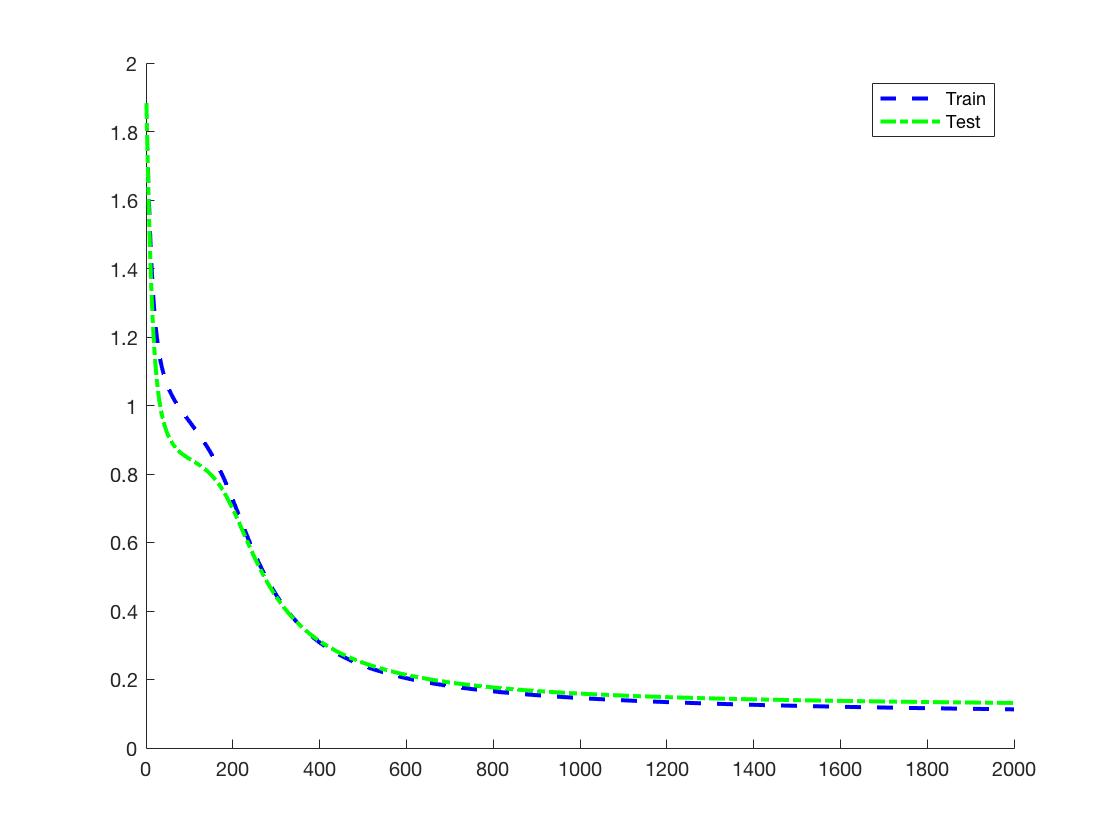


Figure 2 Iris Dataset Training Curve

The performance specifications are as follows:

**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.

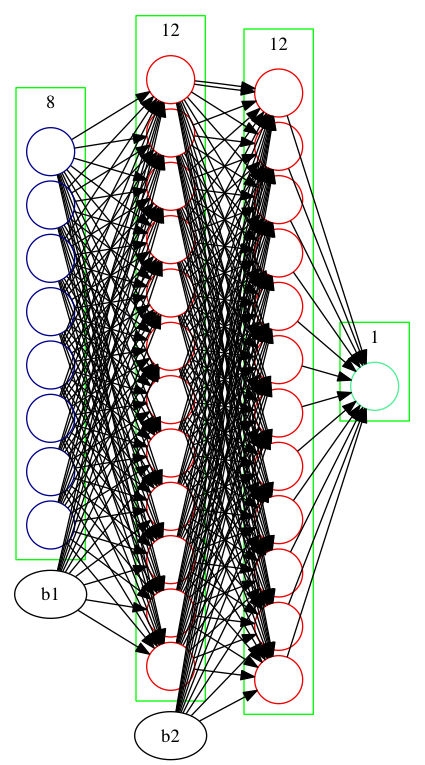


Figure 3 Pima Dataset Network Configuration

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

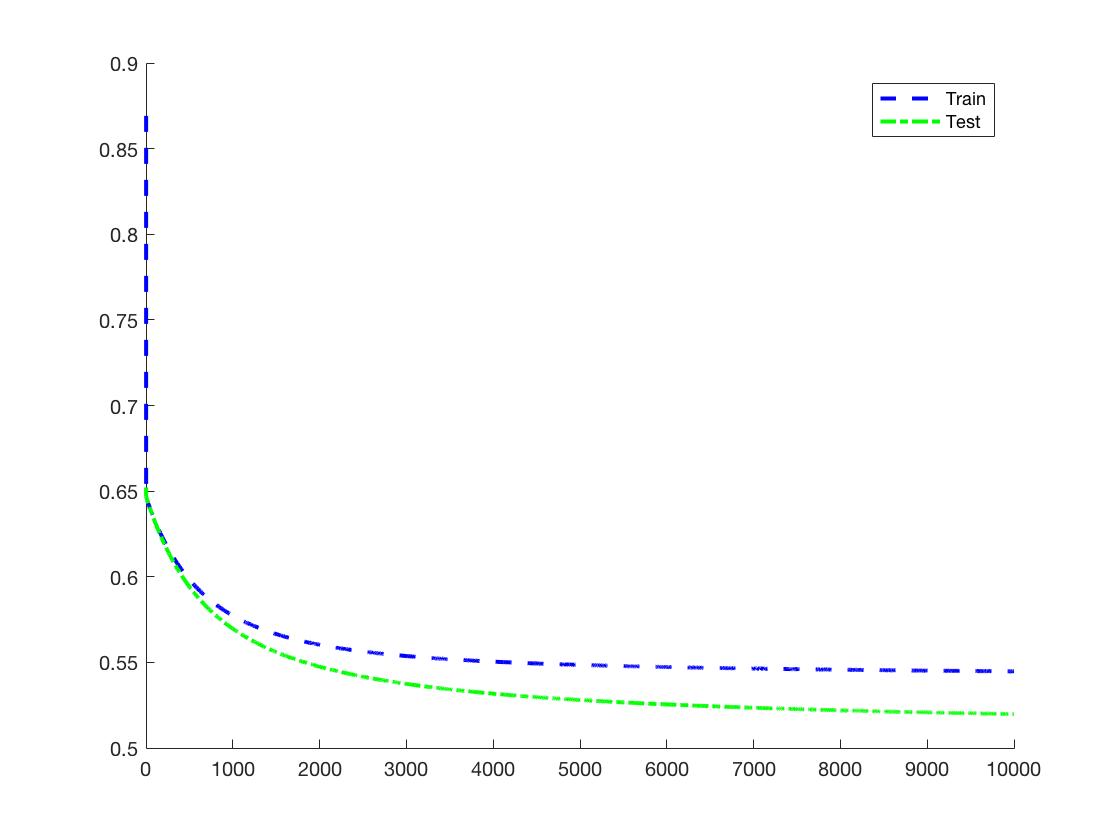


Figure 4 Pima Dataset Training Curve

The following results was attained.

**3) The Wisconsin Cancer dataset**

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

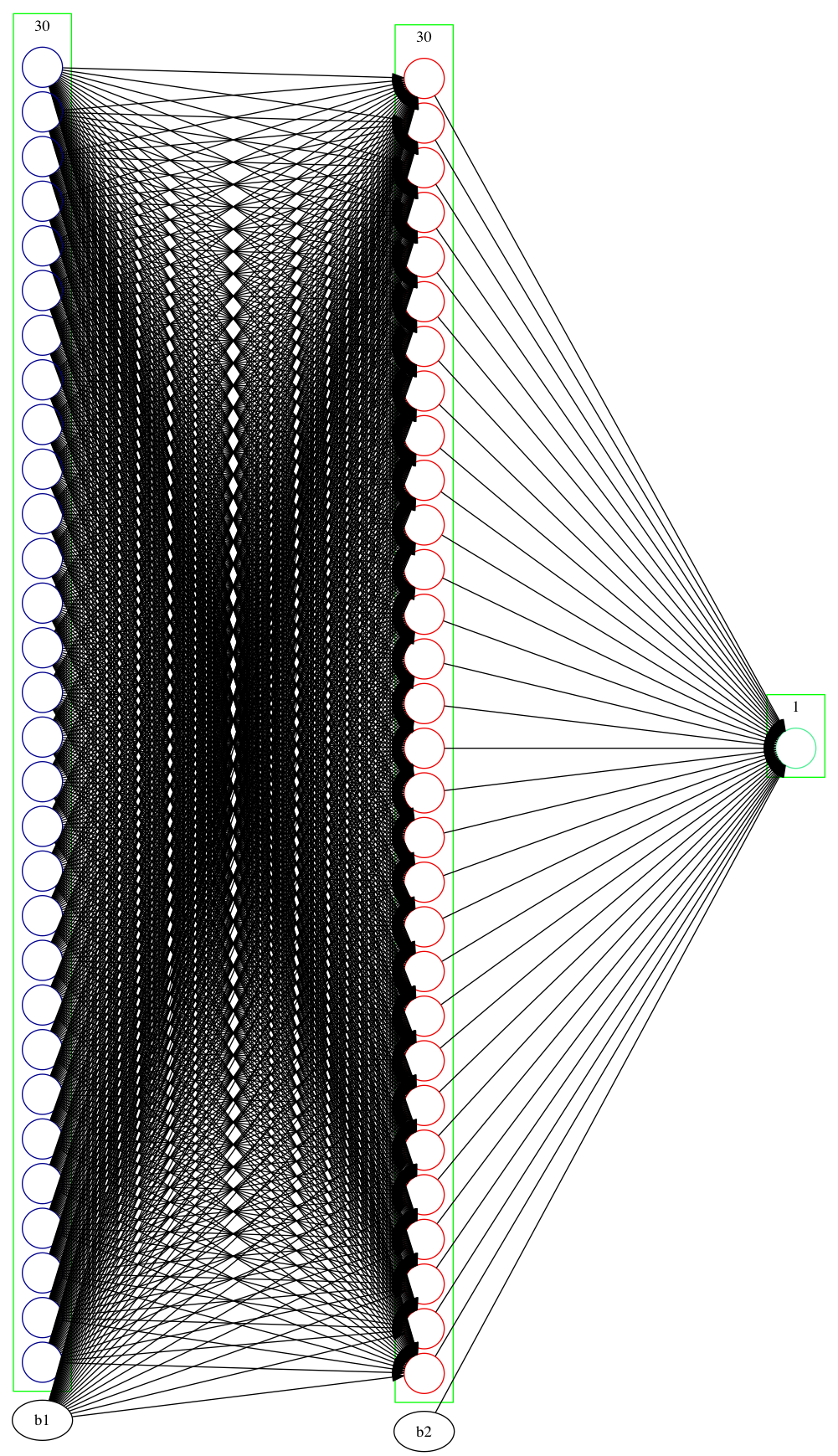


Figure 5 WDBC Dataset Network Configuration

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

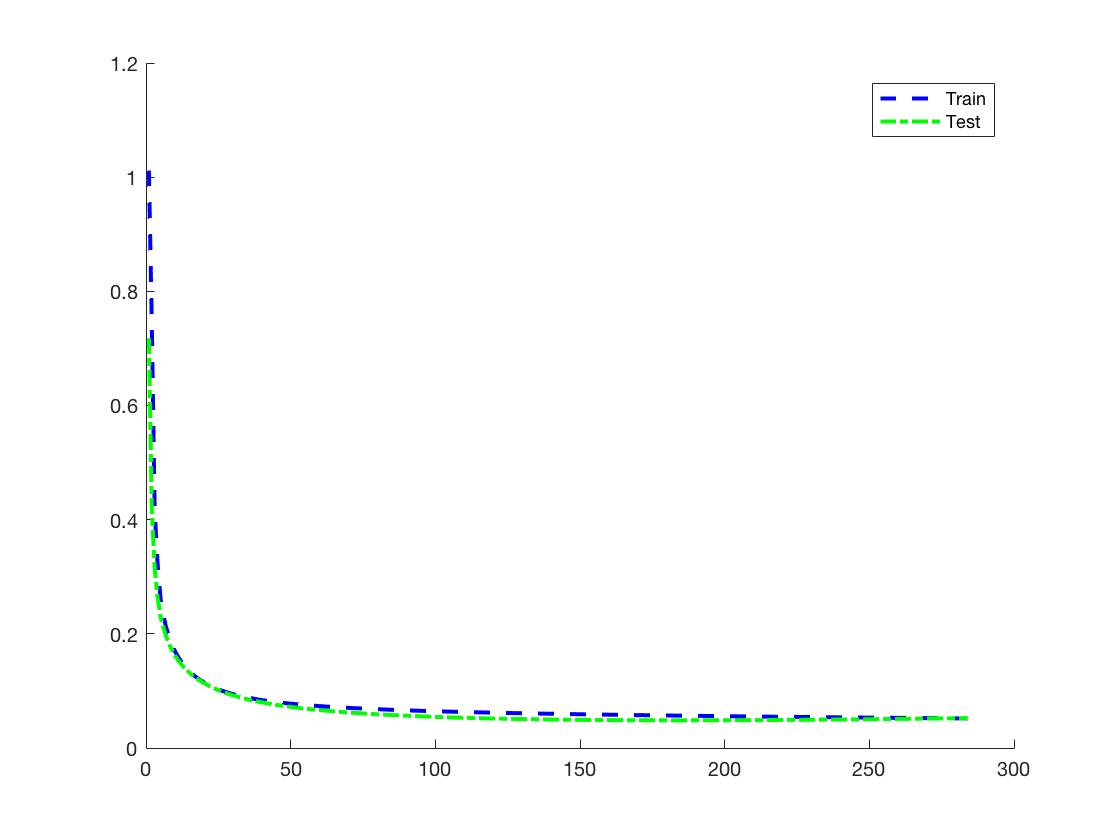


Figure 6 WDBC dataset training curve

The following results was attained.

***(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.

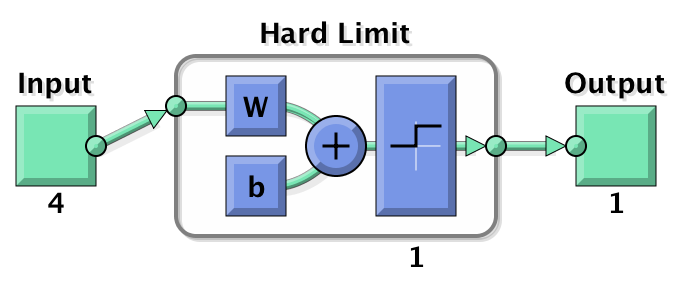


Figure 7 Perceptron

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.

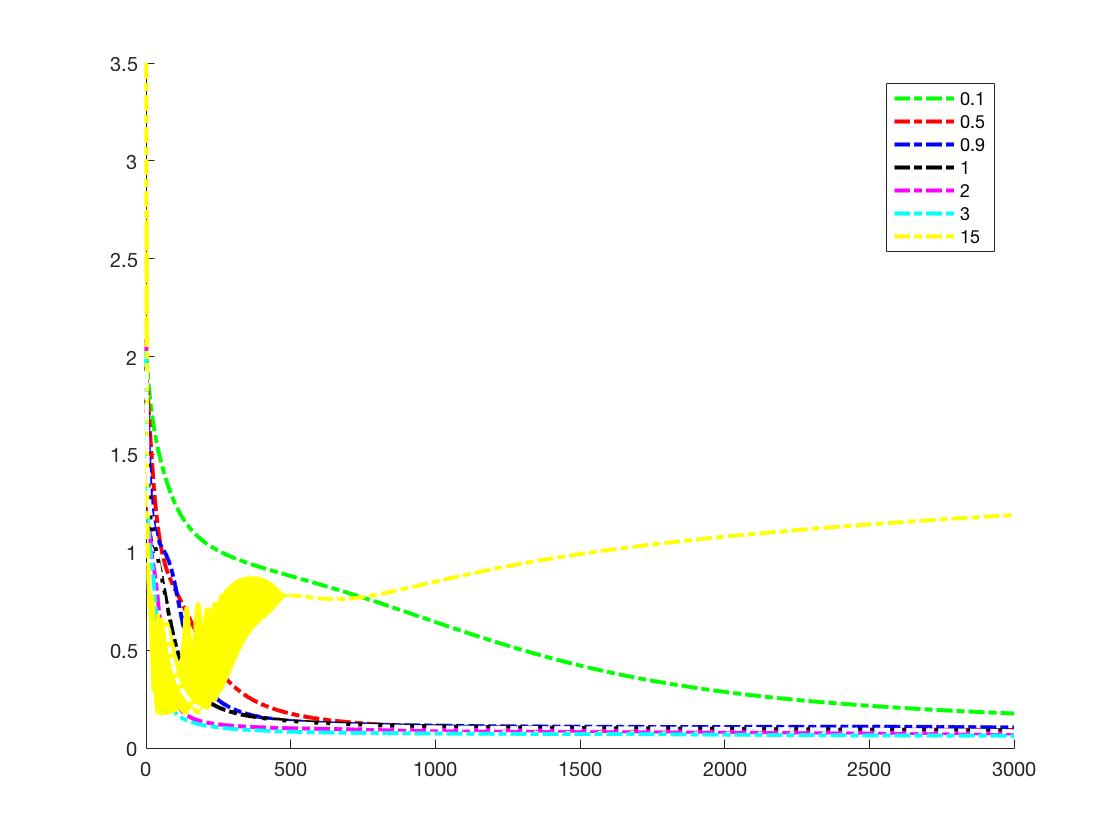


Figure 8 Performance under different learning rate

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
|  | 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% |

The accuracy on the test set is as follows.

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.

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

|  |  |  |
| --- | --- | --- |
|  | purelin | tanh |
|  | 1.135236 | 0.112827 |
|  | 90.00% | 100.0% |
|  | 1.157593 | 0.120325 |
|  | 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.

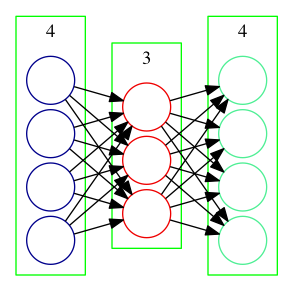


Figure 9 Network Configuration for Autoencoder

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.

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

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.

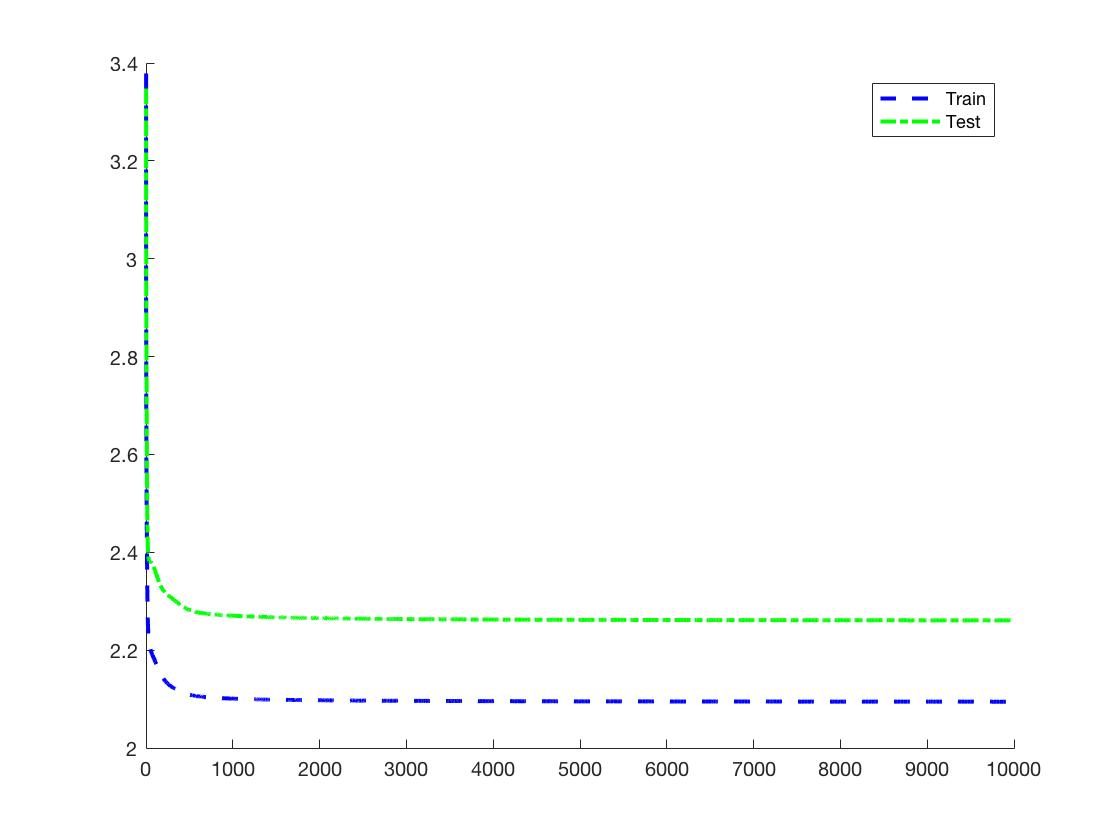


Figure 10 Learning Curve for Autoencoder

The results are as follows.

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