Name: Pratik Mohan Ramdasi

ECE 656: Machine Learning and Adaptive Systems

Computer Assignment 2

Date: 10/22/2015

Title: Multi-layer back-propagation neural network for pattern classification

1. Introduction

The purpose of this computer assignment is to design a multi-layer back-propagation neural network for pattern classification. The MATLAB neural network (NN) toolbox is used to train and test different BPNN classifiers. The data for the study is the ‘wine-recognition’ dataset.
This data was used in many papers for comparing various classifiers. In a classification context, it is a well-posed problem with somewhat "well-behaved" class structures. The dataset is the result of a chemical analysis of wines grown in a particular region in Italy but derived from three different cultivars (or classes). The analysis determined the quantities of 13 constituents found (i.e. 13-D feature space) in each of the three types of wines.
The attributes are:

1. Alcohol content
2. Malic acid
3. Ash
4. Alkalinity of Ash
5. Magnesium
6. Total Phenol
7. Flavonoid
8. Non-flavonoid Phenols
9. Proanthocyanins
10. Color intensity
11. Hue
12. OD280/OD315 of diluted wines
13. Proline

There are total 178 samples having 13 attributes each. Out of all these, 59 samples are for class1, 71 for class2 and 48 for class 3.The dataset is divided into training, validation and testing respectively.
These variables need to be standardized. Additional data samples are added to have equal number of samples for each class by adding random white Gaussian noise with zero mean and unity variance to each attribute to mimic error in measurements.
Using standard generalized delta rule with variable learning rate and a momentum factor and a fast learning algorithm like Levenberg-Marquet, along with two or three layers BPNN, training dataset samples are trained and the learning curve is monitored for each epoch to check if the network is properly training.
Validation samples are used to measure network generalization and to halt training when generalizations stops improving. Testing samples provide independent network performance measure during and after training.
Proposed BPNN approach for classification is compared using different learning rules based on some of the factors like: correct classification rate, number of hidden layers and number of neurons in each layer, associated confusion matrices etc.
Various advantages and disadvantages of NN based classification is provided at the end.

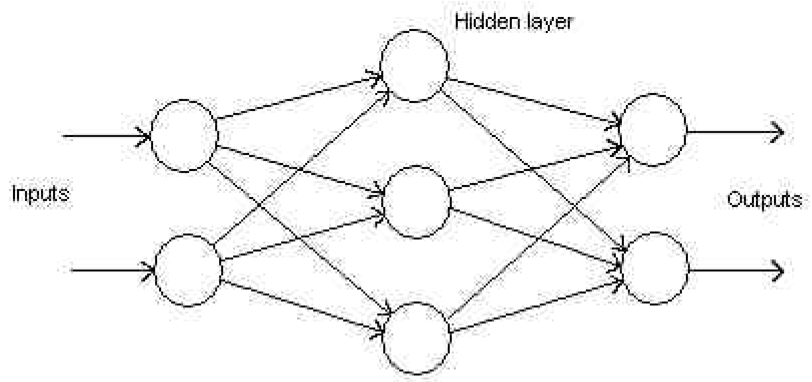
2. Theory

2.1) Multi-layer Perceptron (Feed Forward network):

<u>Multi-layered structure:</u>

Some input and output patterns can be easily learned by single-layer neural networks (i.e. perceptron). However, these single-layer perceptron cannot learn some relatively simple patterns, such as those that are not linearly separable. For example, a human may classify an image of an animal by recognizing certain features such as the number of limbs, the texture of the skin (whether it is furry, feathered, scaled, etc.), the size of the animal, and the list goes on. A single-layer neural network however, must learn a function that outputs a label solely using the intensity of the pixels in the image. There is no way for it to learn any abstract features of the input since it is limited to having only one layer. A multi-layered network overcomes this limitation as it can create internal representations and learn different features in each layer. 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 backpropagation 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.

A basic multi-layered structured is shown in the following figure;



How it works:

- Each neuron receives a signal from the neurons in the previous layer, and each of those signals is multiplied by a separate weight value.
- The weighted inputs are summed and passed through a limiting function, which scales the output to a fixed range of values.
- The output of the limiter is then broadcasted to all of the neurons in the next layer.

To use the network to solve a problem, we apply the input values to the inputs of the first layer, allow the signals to propagate through the network, and read the output values.

As we see from the structure of the multi-layer perceptron in previous figure, the connections between the units do not form a directed cycle. Hence, it is also called as the **multi-layer feed forward network.**

2.2) Back-Propagation learning through generalized delta rule:

Backpropagation, an abbreviation for "backward propagation of errors", is a common method of training artificial neural network used in conjunction with an optimization learning rule such as gradient descent, delta rule etc. Backpropagation requires a known, desired output for each input value in order to calculate the loss function gradient. Hence, usually it is considered as supervised training algorithm**.** It is a generalization of the delta rule to multi-layered feedforward networks, made possible by using the chain rule to iteratively compute gradients for each layer. Backpropagation requires that the activation function used by the artificial neurons be differentiable.

Sigmoidal activation function:
The backpropagation algorithm looks for the minimum of the error function in weight space using the method of gradient descent. The combination of weights, which minimizes the error function, is considered a solution of the learning problem. Since this method requires computation of the gradient of the error function at each iteration step, we must guarantee the continuity and differentiability of the error function. The most popular activation function used for back propagation is the sigmoidal activation function.
It is a real function $\left(\in (0,1)\right)$ given by the following equation:

$$s_c(x) = \frac{1}{1 + e^{-cx}}.$$

The constant c can be selected arbitrarily and its reciprocal $1/c$ is called the temperature parameter in stochastic neural networks. The shape of the sigmoid changes according to the value of c as shown in the figure,

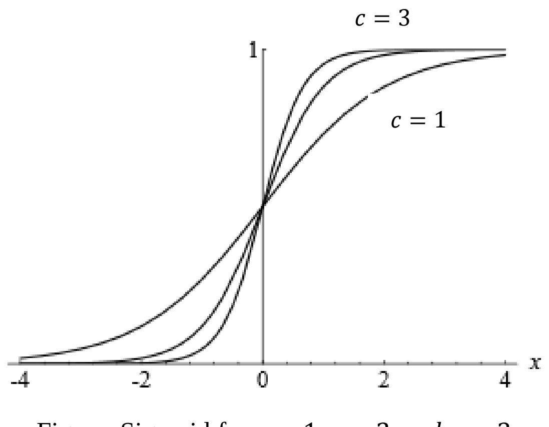


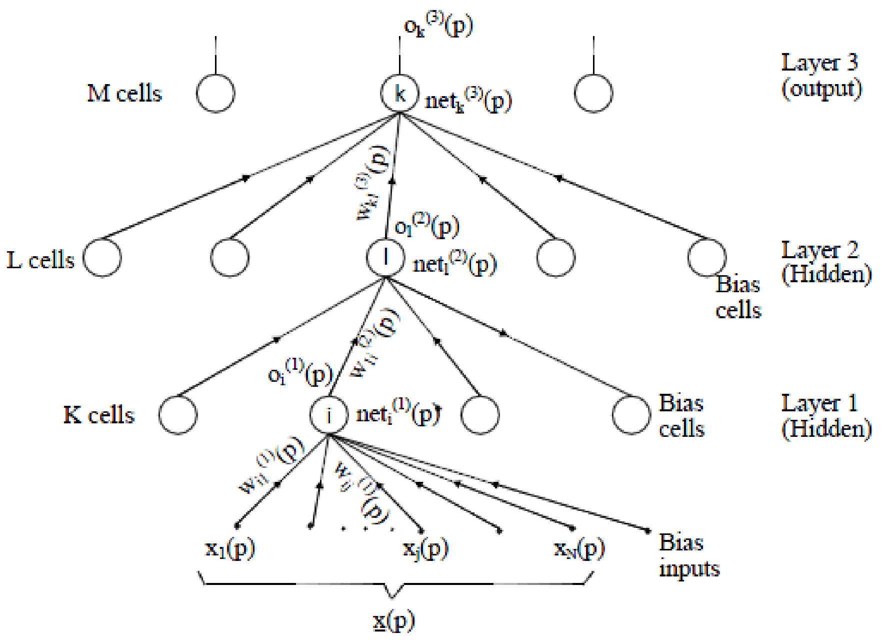
Figure: Sigmoid for $c = 1, c = 2 \; and \; c = 3$

Higher values of c bring the shape of the sigmoid closer to that of the step function and if the limit $c \to \infty$, the sigmoid converges to a step function at the origin. The derivative of the sigmoid function is can be written as:

$$\frac{d}{dx}s(x) = \frac{e^{-x}}{(1 + e^{-x})^2} = s(x)(1 - s(x))$$

<u>Generalized Delta Rule:</u>

Consider a 3-layer BPNN as shown below:



The training of BPNN consists of two processes: forward pass and backward pass.

<u>Forward Pass:</u>

Inputs propagate through the layers of neurons in forward direction and form the outputs from the output layer cells. For cell i in the layer 1st layer at training iteration (training sample) p, we have

$$net_i^{(1)}(p) = \sum_{j=1}^{N+1} w_{ij}^{(1)}(p)\, x_j(p), \forall i \in [1, K]$$

Where, $w_{ij}^{(1)}$ connects input$j$ to cell i in the first layer, $w_{i,N+1}^{(1)} = b_i^{(1)}$i.e. the bias and $x_{(N+1)}(p) = 1$
If $f(.)$ Represents the activation function at cells,

$$o_i^{(1)}(p) = f(net_i^{(1)}(p)), \forall i \in [1, K]$$

Similarly for cell l in the second layer we have,

$$net_l^{(2)}(p) = \sum_{i=1}^{K} w_{li}^{(2)}(p)\, o_i^{(1)}(p), \forall l \in [1, L]$$

And

$$o_l^{(2)}(p) = f(net_l^{(2)}(p)), \forall l \in [1, L]$$

For the output layer cells,

$$net_k^{(3)}(p) = \sum_{l=1}^{L} w_{kl}^{(3)}(p)\ o_l^{(2)}(p)$$

And

$$o_k^{(3)}(p) = f(net_k^{(3)}(p)), \forall k \in [1, M]$$

The actual outputs $o_k^{(3)}(p)$ are compared with the desired outputs $d_k^{(3)}(p)$ for the pattern p and the error signal $e_k^{(3)}(p)$ is generated i.e.

$$e_k^{(3)}(p) = d_k^{(3)}(p) - o_k^{(3)}(p)\ , \forall k \in [1, M]$$

The main objective of learning is to adjust weights and biases in the network such that actual outputs are as close as possible to the desired outputs for all the output cells and all training samples. For each pattern,e, the sum squared error of all the output cells,

$$Ep = \frac{1}{2} \sum_{k=1}^{M} \left(d_k^{(3)}(p) - o_k^{(3)}(p) \right)^2$$

While the average system error over all the training samples can be written as:

$$E = \frac{1}{2P} \sum_{p=1}^{P} \sum_{k=1}^{M} \left(e_k^{(3)}(p) \right)^2$$

Now, using the delta rule, weights are updates as in proportion to $-\nabla$Ep wrt the weights.
To find the weight updating equation for the output layers, we form,

$$\frac{\partial Ep}{\partial\, w_{kl}^{(3)}(p)} = e_k^{(3)}(p) \frac{\partial e_k^{(3)}(p)}{\partial\, w_{kl}^{(3)}(p)} = -e_k^{(3)}(p) \frac{\partial o_k^{(3)}(p)}{\partial\, w_{kl}^{(3)}(p)}, \forall k \in [1, M]$$

Using the chain rule for partial derivatives,

$$\frac{\partial Ep}{\partial\, w_{kl}^{(3)}(p)} = -e_k^{(3)}(p) \frac{\partial\, f(net_k^{(3)}(p))}{\partial\, net_k^{(3)}(p)} \cdot \frac{\partial net_k^{(3)}(p)}{\partial\, w_{kl}^{(3)}(p)}$$

we have,

$$\frac{\partial\, net_k^{(3)}(p)}{\partial\, w_{kl}^{(3)}(p)} = o_l^{(2)}(p)$$

Thus, the weight update equation for the output weights become,

$$\Delta\, w_{kl}^{(3)}(p) = -\mu\, \nabla \mathrm{Ep} = \mu\, e_k^{(3)}(p) f'(net_k^{(3)}(p))\ o_l^{(2)}(p), \forall k \in [1, M]\ and\ \forall l \in [1, L]$$

We have used sigmoidal activation function as explained previously. Then the $f'(net_k^{(3)}(p))$ can be written as,

$$f'(net_k^{(3)}(p)) = (1 - o_k^{(3)}(p))\ o_k^{(3)}(p)$$

And the weight updating equation becomes,

$$w_{kl}^{(3)}(p+1) = w_{kl}^{(3)}(p) + \mu \; e_k^{(3)}(p)\left(1 - o_k^{(3)}(p)\right) o_k^{(3)}(p) o_l^{(2)}(p)$$

This can be represented as,

$$w_{kl}^{(3)}(p+1) = w_{kl}^{(3)}(p) + \mu \, r_k^{(3)}(p) \, o_l^{(2)}(p)$$

Where, learning signal, $r_k^{(3)}(p) = f'(net_k^{(3)}(p)) \; e_k^{(3)}(p) = e_k^{(3)}(p)(1 - o_k^{(3)}(p)) \; o_k^{(3)}(p)$

Problem occurs when we apply the same procedure to the hidden layers as there are no desired or target values for the cells in these layers.

<u>Backward Pass:</u>

To find the weight update expression for the 2nd layer cells, we have,

$$\frac{\partial Ep}{\partial \, w_{li}^{(2)}(p)} = \sum_{k=1}^{M} e_k^{(3)}(p) \frac{\partial e_k^{(3)}(p)}{\partial \, w_{li}^{(2)}(p)} = -\sum_{k=1}^{M} e_k^{(3)}(p) \frac{\partial o_k^{(3)}(p)}{\partial \, w_{li}^{(2)}(p)}$$

After solving, we get,

$$\frac{\partial Ep}{\partial \, w_{li}^{(2)}(p)} = -f'(net_l^{(2)}(p)) \, o_i^{(1)}(p) \sum_{k=1}^{M} w_{kl}^{(3)}(p) \, r_k^{(3)}(p)$$

And hence,

$$\Delta \, w_{li}^{(3)}(p) = \mu \, f'(net_l^{(2)}(p)) \, o_i^{(1)}(p) \sum_{k=1}^{M} w_{kl}^{(3)}(p) \, r_k^{(3)}(p)$$

As we see, error (learning signal) is back propagated through the weights (before updating) of the 3rd layer to determine the error signal at the output of the hidden layer cells.
Error signal is defined by,

$$r_l^{(2)}(p) = f'(net_l^{(2)}(p)) \sum_{k=1}^{M} w_{kl}^{(3)}(p) \, r_k^{(3)}(p)$$

So, the updating equation becomes,

$$w_{li}^{(2)}(p+1) = w_{li}^{(2)}(p) + \mu \, r_l^{(2)}(p) \; o_i^{(1)}(p)$$

Using the sigmoidal non-linearity

$$f'(net_l^{(2)}(p)) = (1 - o_l^{(2)}(p)) \, o_l^{(2)}(p)$$

To find the weights for the first layer, we repeat the same procedure to get,

$$\frac{\partial Ep}{\partial \, w_{ij}^{(1)}(p)} = f'(net_i^{(1)}(p)) x_j(p) \sum_{l=1}^{L} w_{li}^{(2)}(p) \, r_l^{(2)}(p)$$

if we define, $r_i^{(1)}(p) = f'(net_i^{(1)}(p)) \sum_{l=1}^{L} w_{li}^{(2)}(p)\, r_l^{(2)}(p)$

then,

$$r_i^{(1)}(p) = (1 - o_i^{(1)}(p))\, o_i^{(1)}(p) \sum_{l=1}^{L} w_{li}^{(2)}(p)\, r_l^{(2)}(p)$$

i.e. error signal at the output of the 1st hidden layer cells is obtained by back propagating the errors $r_l^{(2)}(p)$ from the outputs of 2nd hidden layer cells through the weights of the 2nd layer to the cells in the 1st layer. Thus the weight update equation for the 1st layer becomes,

$$w_{ij}^{(1)}(p+1) = w_{ij}^{(1)}(p) + \mu\, r_i^{(1)}(p)\, x_j(p)$$

BPNN learning algorithm steps can be summarized as follows:

1. Initialization:
   Choose μ, max. allowable error and initialize weights to small random values.

2. Forward Pass:
   Propagation of the training pattern $x(p)$ and computation of actual outputs, $o_k^{(3)}(p), \forall k \in [1, M]$.

3. Error Computation:
   Compute squared error at all the cells using:
   $$Ep = \frac{1}{2} \sum_{k=1}^{M} \left(d_k^{(3)}(p) - o_k^{(3)}(p)\right)^2$$

4. Error Back-propagation:
   For output layer,
   $$r_k^{(3)}(p) = e_k^{(3)}(p) \left(1 - o_k^{(3)}(p)\right) o_k^{(3)}(p), \forall k \in [1, M]$$
   for 2nd hidden layer,
   $$r_l^{(2)}(p) = (1 - o_l^{(2)}(p))\, o_l^{(2)}(p) \sum_{k=1}^{M} w_{kl}^{(3)}(p)\, r_k^{(3)}(p), \forall l \in [1, L]$$
   for 1st hidden layer,
   $$r_i^{(1)}(p) = (1 - o_i^{(1)}(p))\, o_i^{(1)}(p) \sum_{l=1}^{L} w_{li}^{(2)}(p)\, r_l^{(2)}(p), \forall i \in [1, K]$$

5. Weight Updating:
   Use the weight updating equations give below for each layer:
   $$w_{kl}^{(3)}(p+1) = w_{kl}^{(3)}(p) + \mu\, r_k^{(3)}(p)\, o_l^{(2)}(p), \forall k \in [1, M]$$
   $$w_{li}^{(2)}(p+1) = w_{li}^{(2)}(p) + \mu\, r_l^{(2)}(p)\, o_i^{(1)}(p), \forall l \in [1, L]$$
   $$w_{ij}^{(1)}(p+1) = w_{ij}^{(1)}(p) + \mu\, r_i^{(1)}(p)\, x_j(p), \forall i \in [1, K]$$

6. Iteration and Error Check:
   If $p < P, increment\ p$ and go to step (2); otherwise check if error < max. allowable error.
   If yes, terminate the training. If not, restart from $p = 1$ and initiate a new training session and go to step (2).

2.3) Practical Consideration and Issues:

Several important factors impact the learning performance of the BPNN. These are:

1. Weight initialize:

   Large initial weights lead to saturation, which causes small gradients and ultimately slow/no training.
   It is recommended to use,
   $$w_{ij}^{(k)}(0) \sim uniform[\frac{-\sqrt{3}a}{N_k}, \frac{\sqrt{3}a}{N_k}]$$
   where N_k is number of inputs to the kth layer and $1 \leq a \leq 1.5$

2. Learning Parameters:

   a) *Step size (learning rate) μ:*
   It presents tradeoff between accuracy and convergence speed.
   $\mu \propto \frac{\mathbf{1}}{\boldsymbol{fan-ins}}$ where $fan - ins$ is number of incoming weights to a layer**.**

   b) *Momentum factor:*
   Used to accelerate the convergence.

$$w_{ij}^{(k)}(p+1) = w_{ij}^{k}(p) + \Delta w_{ij}^{(k)}(p)$$

$$\Delta w_{ij}^{(k)}(p) = \mu\, r_i^{(k)}(p)\, o_j^{(k-1)}(p) + \alpha \Delta w_{ij}^{(k)}(p-1)$$

Where, $0 < \alpha < 1$ is the momentum parameter.
$\alpha = 0 \Rightarrow standard\ generalized\ delta\ rule$

The effects of momentum become important during the later stages where it introduces the forgetting factor to the weight increments.

   c) *Local minimum:*
   If BPNN stops learning before reaching an acceptable error goal, some remedies can be applied such as restarting weight initialization, changing the number of hidden layer neurons, changing learning parameter values etc.

d) *Information content:*

Training samples should be chosen according to their information content. Avoid choosing similar samples and overtraining.

e) *Network structure:*

The number of hidden layer cells determines existence of the solution and accuracy in approximating a mapping function. Number of hidden layer neurons, K are approximately,
For 2-Layer BP network: $K \sim 2 - 2.5N$ where N = dimension of the input vector,
For 3-Layer BP network: $K \sim 2 - 2.5N$ for 1st hidden layer and $L \sim 1 - 1.5N$ for 2nd hidden layer

f) *Data representation:*

Data is standardized prior to training.
1. Subtract sample mean vector using all the training data from each sample.
2. Divide by standard deviation of the data vector for each sample.

g) *Function approximation:*

Universal approximation theorem states that an MLP network with a single hidden layer containing a finite number of hidden cells is a universal approximator among continuous functions on compact subsets, under mild assumptions on the activation function.

h) *Scaling:*

There are many different parameters that can be scaled including: number of
(a) Hidden layers; (b) hidden layer cells; (c) fan-in (incoming) and fan-out (outgoing) of the hidden layer cells; and (d) training sample and complexity of mapping.

2.4) Levenberg-Marquardt algorithm:

The Levenberg–Marquardt algorithm, which was independently developed by Kenneth Levenberg and Donald Marquardt, provides a numerical solution to the problem of minimizing a non linear function. It is fast and has stable convergence. In the artificial neural-networks field, this algorithm is suitable for training small- and medium-sized problems.
It gives a good exchange between the speed of the Newton algorithm and the stability of the steepest descent method.

<u>The Levenberg-Marquardt (LM) method:</u>

In the back propagation algorithm, the performance index (denoted as F(w)) to be minimized is sum squared error between target outputs and network's simulated outputs and it can be written as:

$$F(w) = e^T e$$

Where $w = [w1, w2, \dots, wN]$ consists of all weights of the network, e is the error vector comprising the error for all the training examples. When training with the LM method, the increment of weights Δw can be obtained as follows:

$$\Delta w = [J^T J + \mu I]^{-1} J^T e$$

Where J is the Jacobian matrix, μ is the learning rate, which is to be updated using the β depending on the outcome. In particular, μ is multiplied by decay rate β (0 < β <1) whenever $F(w)$ decreases, whereas μ is divided by β whenever $F(w)$ increases in a new step.

The standard LM training process:
1. Initialize the weights and the parameter μ (= 0.01).
2. Compute the sum of squared errors for all the inputs $F(w)$.
3. Solve 2 to obtain increment of weights Δw
4. Recomputed the sum of squared errors $F(w)$. Use $w = w + \Delta w$ and monitor the sum till its lesser then the maximum allowable value.

3. Results and Discussions

In this computer assignment, we will be training 2 and 3 layer BPNNs for the given wine dataset using generalized delta rule and Levenberg-Marquet rule. The task is to correctly classify the given data into number of classes.
The wine recognition dataset is from the website- http://archive.ics.uci.edu/ml/datasets/Wine.
As mentioned earlier, there are total 178 samples with 13 attributes each. These are divided into 3 classes such that 59 samples represent class1, 71 samples represent class2 and remaining 48 samples represent class3.
We will be using equal number of samples for each class. Number of samples from each class can be made equal by adding random white Gaussian noise with zero mean and unity variance. Hence, our new wine data set contains:
Class1 = 60 samples
Class2 = 60 samples
Class3 = 60 samples
Total number of samples will be 180.

Pre-processing:

Since the attribute values are very different from each other, it is necessary to do preprocessing on the dataset before using it for training.
Procedure for standardizing the dataset:

- 13 Attributes corresponding to total 180 samples are arranged column wise. Calculate the mean and standard deviation value of each attribute.
- Subtract the mean value from each value and divide by the standard deviation.

The operation can be shown mathematically as,

$$x = (x - \mu)/\sigma$$

Where, $x = input\ sample,\ \mu = mean\ value,\ \sigma = std\ deviation$
Newly formed dataset is used for training and testing.

Network Parameters:

After pre-processing the data, it is important to define the network parameters before using the network. Some of the parameters are selected as follows:

1. *Learning rate:*
   For a 2-layer BPNN, number of incoming weights to a hidden layer, $fan-ins = 13$
   We know, learning rate $\mu \propto \frac{1}{fan-ins}$ and hence we choose $\mu = \frac{1}{13} = 0.07$

2. *Number of neurons in hidden layers:*
   For 2-layer BPNN: $K \sim 2 - 2.5N$
   For our network, $N = 13$
   consider, $K = 30$
   For 3-Layer BPNN:
   For 1st hidden layer - $K = 30$
   For 2nd hidden layer - $L \sim 1 - 1.5N$, consider $L = 15$

3. *Weight Initialization:*

Consider $\alpha = 1$ and number of inputs to the first layer, $N_k = 13$

We choose initial weights to be in uniform distribution between $\left[\frac{-\sqrt{3}}{13}, \frac{\sqrt{3}}{13}\right] = [-0.13, 0.13]$

Finally, 70% of the dataset is used for training, 15% for validation and 15%for testing. We are using NN toolbox from MATLAB to train and test our wine dataset.

3.1) 2-layer BPNN

<u>Learning using gradient descent:</u>

'traingdx' is a network training function that updates weight and bias values according to gradient descent momentum and an adaptive learning rate.

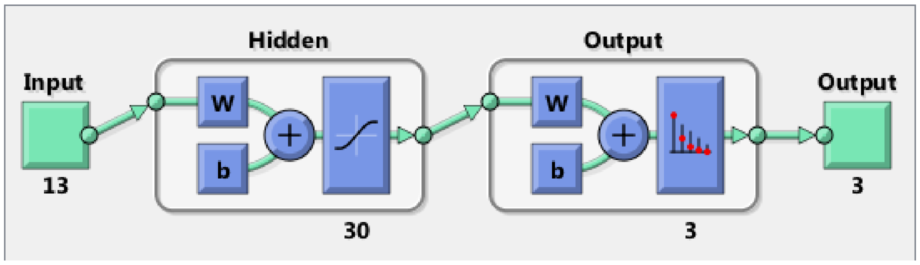


Figure1: block structure of 2-layer BPNN

2-layer BPNN contains 30 neurons in the hidden layer and 3 neurons in the output layer. Momentum factor for the learning rule is 0.9 and learning rate (μ) is 0.07.

Learning curve:

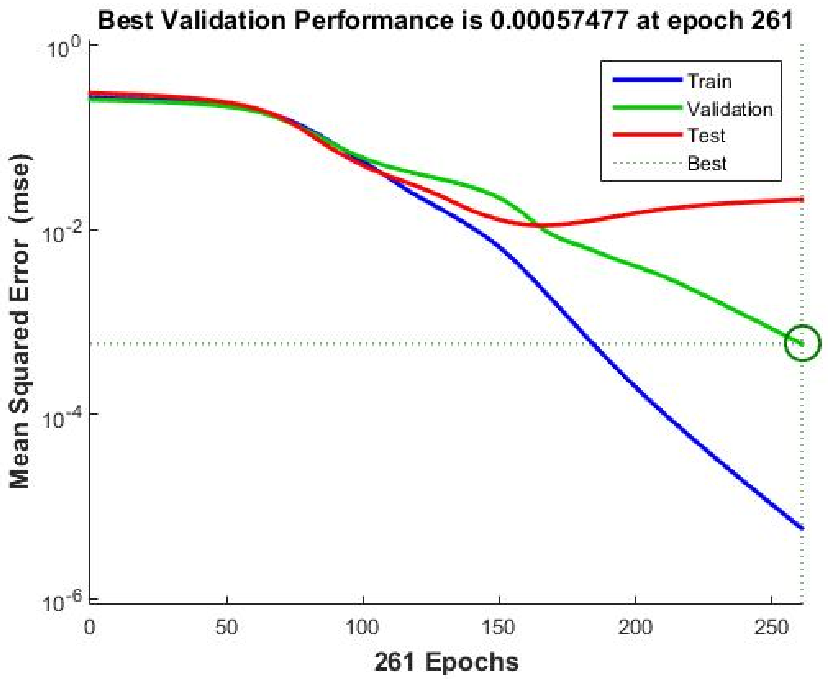


Figure 2: learning curve for gradient descent, $\mu = 0.07$

Confusion matrix:

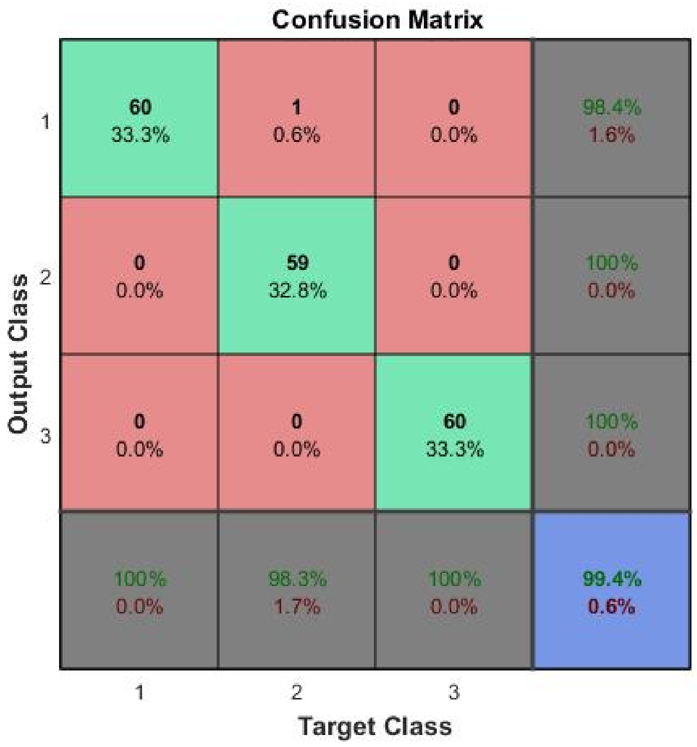


Figure 3: confusion matrix for gradient descent

Observations:

- Network is trained number of times using random initial weight values and training is stopped when the error decreases below 0.001 (maximum allowable error) and the training process is considered successful.
- The gradient descent-training rule with momentum and adaptive learning rate gives the best performance on validation after 261 epochs as can be seen from the learning curve above.
- It was observed that MSE converges after many epochs with respect to the same learning rate and momentum term. MSE values for validation, training and testing are given in following table:

	MSE
Training	$5.74 * 10^{-4}$
Validation	$5.79 * 10^{-6}$
testing	$2.09 * 10^{-2}$

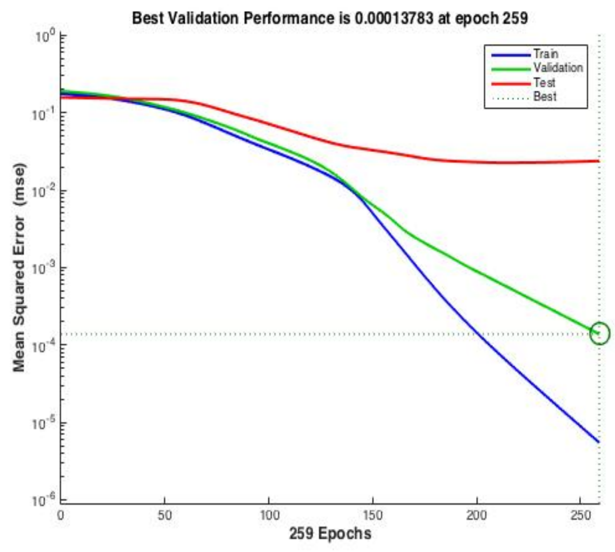
- Effect of changing momentum factor:

Momentum factor	Epochs
0	325
0.5	275
0.9	261

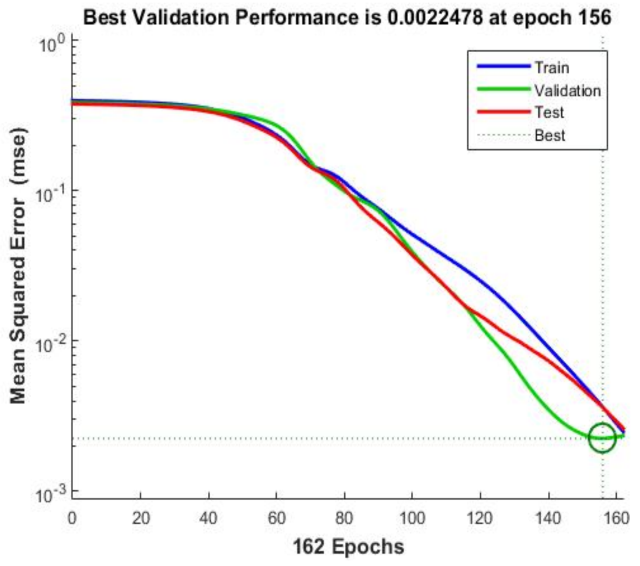
As we see, the momentum factor accelerates the process.

- Effect of changing the learning rate (μ):

- It is found that high learning rate leads to rapid learning but weights may oscillate while lower value of learning rate slows the weight updation process.
  Validation learning curve for $\mu = 0.001 \; and \; 0.1$ is shown below:



learning curve, $\mu = 0.001$



learning curve $\mu = 0.1$

We observe that convergence is faster when we increase the learning rate value. However, if the value is too big (eg.1), then the weights may oscillate and the network will not be stable.

- The visualization of the performance of the algorithm can be shown by looking at the confusion matrix above.
- We see that all the samples from class1 are correctly classified while one sample from class2 is misclassified as class1. All the samples from class3 are correctly classified.
- The overall classification rate of out gradient descent BPNN algorithm is 99.4%.

Learning using Levenberg-Marquardt:

Learning curve:

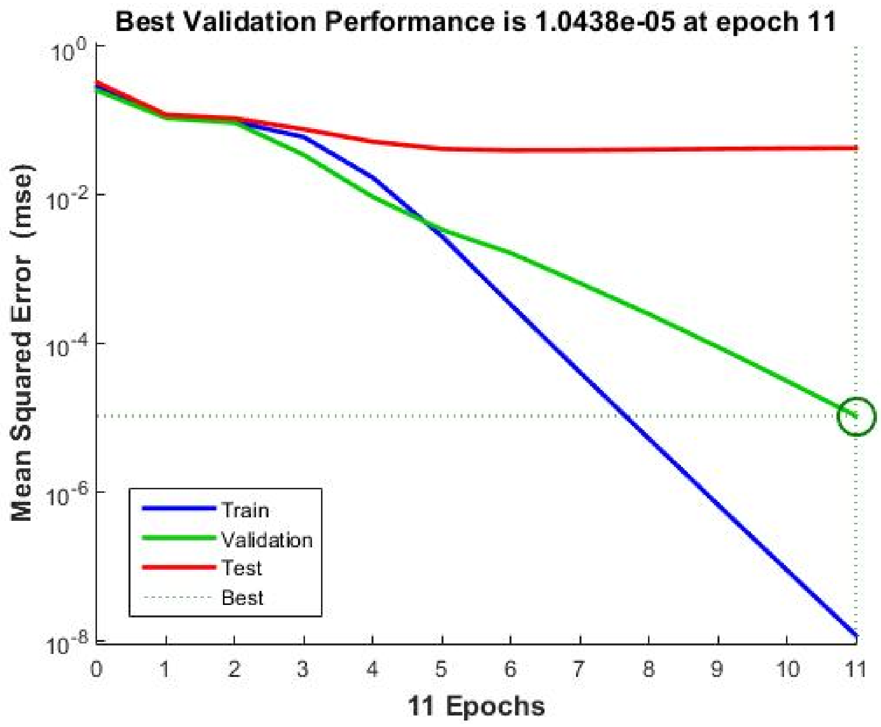
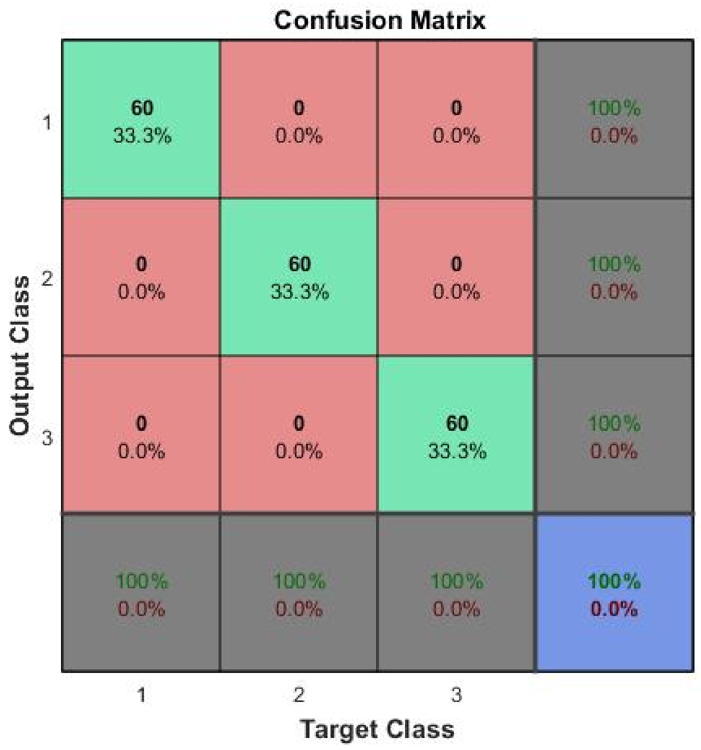


Figure 4: learning curve for Levenberg-Marquardt

Confusion matrix:



	1	2	3	
1	60 33.3%	0 0.0%	0 0.0%	100% 0.0%
2	0 0.0%	60 33.3%	0 0.0%	100% 0.0%
3	0 0.0%	0 0.0%	60 33.3%	100% 0.0%
	100% 0.0%	100% 0.0%	100% 0.0%	100% 0.0%

Figure 5: confusion matrix for Levenberg-Marquardt

Observations:

- As we see form the learning curve for Levenberg-Marquardt algorithm, the convergence is faster than the normal gradient descent learning. This is evident from the figure as it takes only 11 epochs to get the minimum value of MSE for best validation whereas usual gradient descent takes 261 epochs.
- MSE values for training, validation and testing are given in the following table:

	MSE
Training	$1.04 * 10^{-5}$
Validation	$1.16 * 10^{-8}$
testing	$4.26 * 10^{-2}$

- Though the algorithm is faster, it may take large memory space.
- We can see from the confusion matrix that all the samples are correctly classified and the classification rate is 100%.
- We can say that performance of Levenberg-Marquardt algorithm is much better than gradient descent learning rule.

3.2) 3-Layer BPNN

Learning using gradient descent:

‘

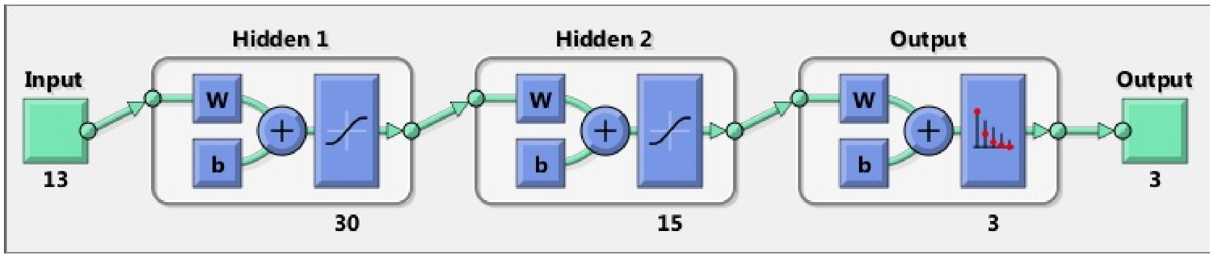


Figure 6: Block structure for 3 layer BPNN

3-Layered BPNN contains 30 neurons in the first hidden layer, 15 neurons in the second hidden layer and 3 neurons in output layer. Learning rate (μ) is 0.07 and momentum factor is 0.9.

Learning curve:

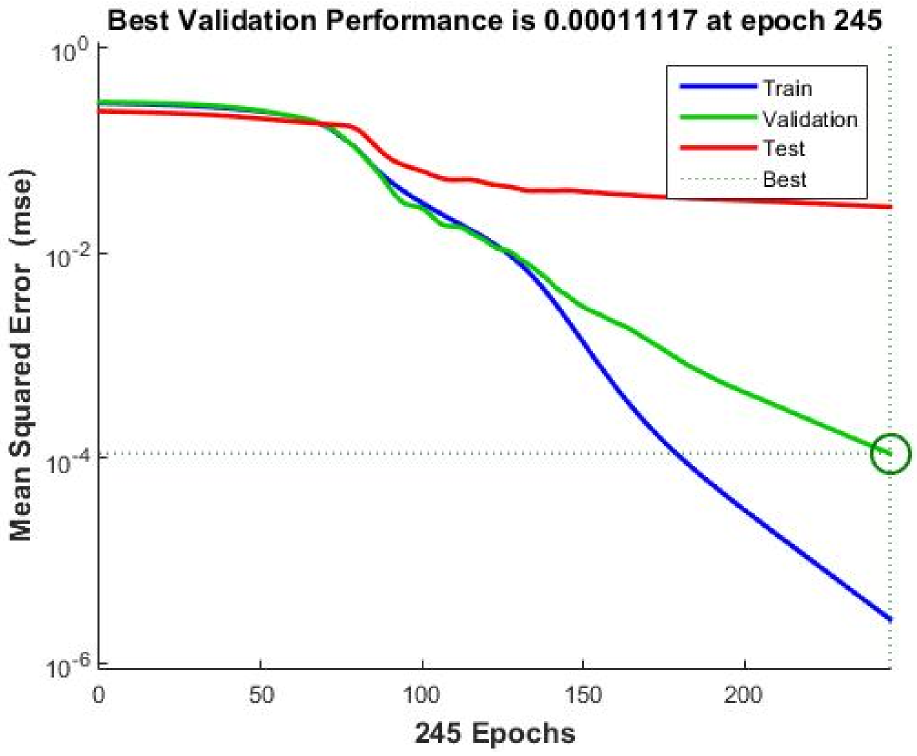
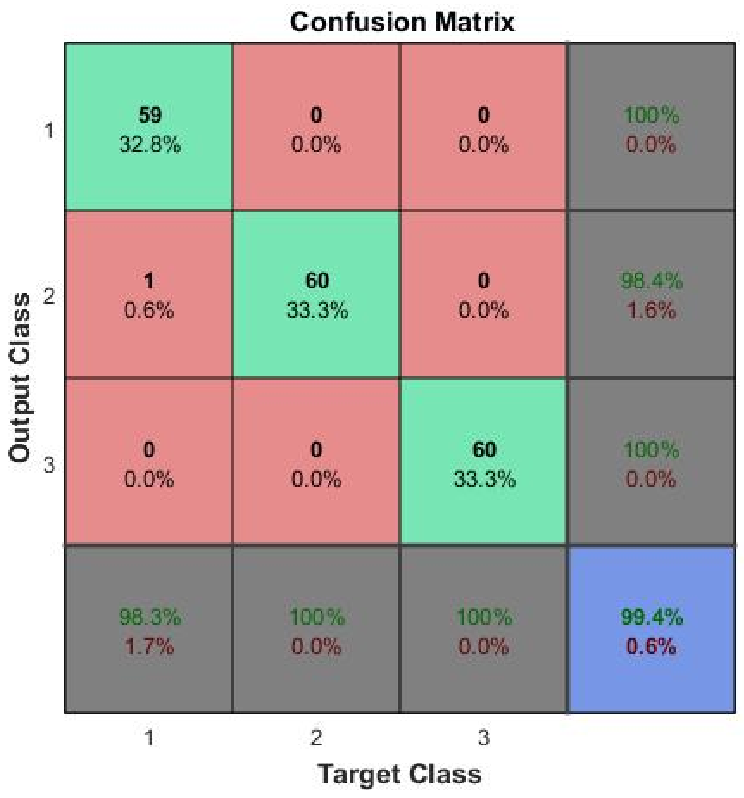


Figure 7: learning curve for gradient descent, $\mu = 0.07$

Confusion matrix:



Confusion Matrix
Output Class
Target Class

	1	2	3	
1	**59** 32.8%	**0** 0.0%	**0** 0.0%	100% 0.0%
2	**1** 0.6%	**60** 33.3%	**0** 0.0%	98.4% 1.6%
3	**0** 0.0%	**0** 0.0%	**60** 33.3%	100% 0.0%
	98.3% 1.7%	100% 0.0%	100% 0.0%	**99.4%** **0.6%**

Figure 8: confusion matrix for gradient descent

Observations:

- 3-layer BPNN is trained number of times using random initial weight values till the error goes below the maximum allowable error (0.001) and then the training is stopped.
- The gradient descent-training rule with momentum and adaptive learning rate gives the best performance on validation after 245 epochs as can be seen from the learning curve above.
- MSE for training, validation and testing is shown in the following table:

	MSE
Training	$1.11 * 10^{-4}$
Validation	$2.7 * 10^{-6}$
testing	$2.79 * 10^{-2}$

- We observe that 3-layer BPNN gives better performance as compared to 2-layer BPNN for training and validation phase.
- The visualization of the performance of the algorithm can be shown by looking at the confusion matrix above.
- We see that 59 samples from class1 are correctly classified while one sample from class1 is misclassified as class2. All the samples from class2 and class3 are correctly classified.
- The overall classification rate of out gradient descent 3-layer BPNN algorithm is 99.4%.

Learning using Levenberg-Marquardt:

Learning curve:

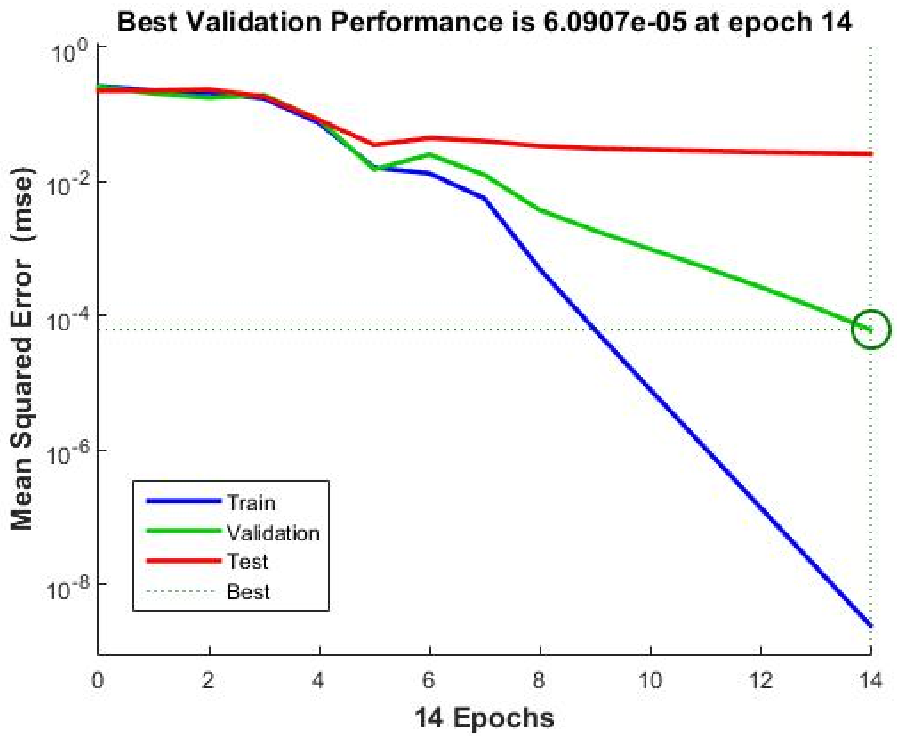


Figure 9: learning curve for Levenberg-Marquardt

Confusion matrix:

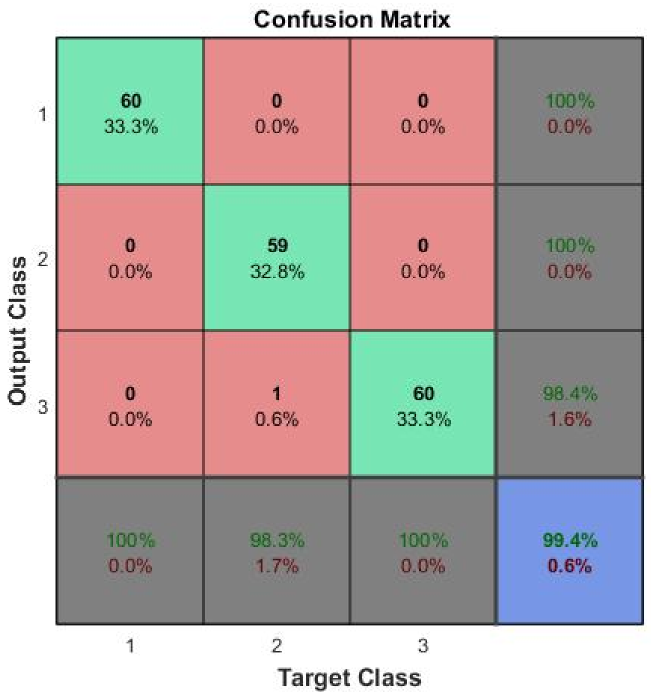


Figure 10: confusion matrix for Levenberg-Marquardt

Observations:

- As we see form the learning curve for Levenberg-Marquardt algorithm, the convergence is faster than the gradient descent learning. This is evident from the figure as it takes only 14 epochs to get the minimum value of MSE for best validation whereas usual gradient descent takes 245 epochs.
- Value of MSE for training, validation and testing is as shown in the table.

	MSE
Training	$6.09 * 10^{-5}$
Validation	$4.8 * 10^{-9}$
testing	$2.38 * 10^{-2}$

- Though the algorithm is faster, it may take large memory space.
- The visualization of the performance of the algorithm can be shown by looking at the confusion matrix above.
- We see that all the samples from class1 are correctly classified while one sample from class2 is misclassified as class3. All the samples from class3 are correctly classified.
- The overall classification rate of out Levenberg-Marquardt 3-layer BPNN algorithm is 99.4%.

3.3) Discussion and comments

Generalization ability of the neural networks

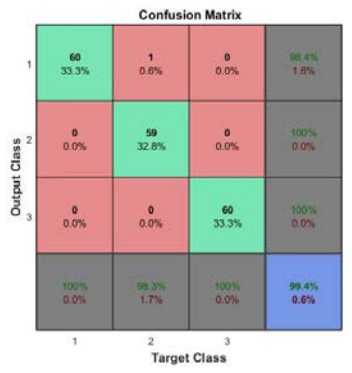
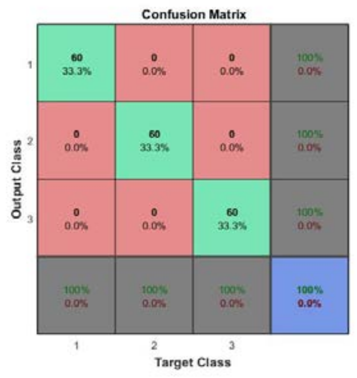
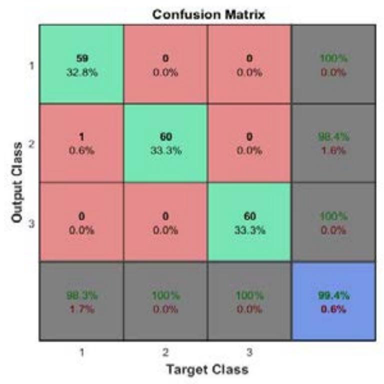
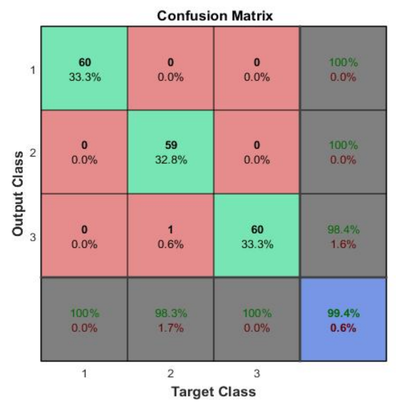
- To assess the generalization performance of our BPNN algorithm trained on different training set sizes, we conducted some experiments varying the size of the training set. The results are depicted in the following table:
  For a single hidden layer network (2-layer BPNN):

Training	Validation	Testing	MSE
40%	30%	30%	0.0203
50%	30%	20%	0.012
60%	20%	20%	0.0053
70%	15%	15%	$5.74 * 10^{-4}$
80%	10%	10%	$6.8 * 10^{-5}$
90%	5%	5%	$3.98 * 10^{-7}$

- Table shows the generalization MSE versus the size of the training set for a single hidden layer network with 30 neurons in it.
- As we can see, the generalization error goes on decreasing as the size of training data set goes on increasing and the minimum error will occur when the size of the training data set will be very high (90% or 100%). Perhaps one of the reason why this could be so is that the full training set is more representative of the problem space that the altered variations of it.
- In reality, we do not need to train our network with large amount of training data. Various experiments have been conducted to observe this and it has been found out that beyond certain size of training data, generalization do not improve, pointing that an excess of training patterns is no gain.
- As we can see from the table, we selected 70% training data as the minimum MSE is lower than maximum allowable error and we have achieved good results based on our choice of training data size. This also gave us sufficient data for validation and testing our algorithm. The same trend on MSE follows for 3-layer BPNN network.

Comments

- In this study, we have applied 2 and 3 layer back propagation neural networks for the pattern classification problem.
- We gave generated the plots showing the classification rate, generated MSE while training, validation and testing. Below is the table showing the comparison between different (2 and 3 layer) networks and different learning rules we used based on several factors.
  These factors include:
  1. Number of hidden layer neurons
  2. Number of layers
  3. Learning rule
  4. Correct Classification rate
  5. Associated confusion matrix

Number of layers	Number of hidden layer neurons	Learning Rule	Correct classification rate	Associated confusion matrix
2	30	Generalized delta rule	99.4	
2	30	Levenberg-Marquardt rule	100	
3	1st hidden layer- 30 2nd hidden layer - 15	Generalized delta rule	99.4	
3	1st hidden layer- 30 2nd hidden layer- 15	Levenberg-Marquardt rule	99.4	

Advantages/Disadvantages of NN based classification:

- Neural networks are models of intelligence that consists of large numbers of simple processing units that collectively are able to perform very complex pattern matching tasks. These models perform stimulus response mapping.
- Classification is a process of learning rules or models from training data to generalize the known structure and then to classify new data with these rules.
- NN algorithm can estimate complex target concepts locally and differently for each new instance to be classified.
- Neural network classifiers are data-driven and self-adaptive.
- They are universal function approximator.
- They have high accuracy and noise tolerance.
- Some other advantages include: quick learning, generalization accuracy on many domains, robustness to noisy training data, easy to understand
- Some of the disadvantages of neural networks include; lack of transparency, longer learning time (for typical generalized learning algorithms) also defining classification rules is difficult and time consuming.
- They have large storage requirement and accuracy of a neural network decreases with increase in irrelevant attributes.

4. Conclusion

- In this computer assignment, 2 and 3-layers Back Propagation neural network is applied to the pattern classification problem.
- Back propagation neural network is successfully constructed using Matlab Neural Network toolbox.
- Wine recognition dataset is used as the input data to train the network and classification is performed.
- To train the back propagation neural network, generalized delta rule and the Levenberg-Marquardt learning rule is applied and results are plotted and tabulated.
- We observed that Levenberg-Marquardt learning rule is faster training rule than generalized gradient descent with momentum and adaptive learning rate. However, it does not take into account the momentum factor, which accelerates the conversion process.
- As the number of neurons in hidden layer increase, learning is slowed down on an average. The main reason for this will be, increase in hidden layer neurons increase number of epochs needed for convergence.
- Choosing learning rate,$\mu \propto \frac{1}{fan-ins}$, significantly reduced number of epochs. The most difficult part in the training process is to determine the appropriate learning rate and the momentum factor.
- Large initial weights lead to saturation, which causes small gradients and ultimately slow/no training.
- From the results, it is evident that Levenberg-Marquardt learning rule is faster and most popular for pattern classification applications as compared to generalized delta rule with gradient descent. However, it takes large storage space.
- 3-layer network works faster and with better efficiency than 2-layer network however, both the networks give approximately similar correct classification rate (around 99%).
- Comparison with respect to several characteristics for both the networks is tabulated and generalization ability of the networks is discussed.

5. References

- Chien-Cheng Yu, and Bin-Da Liu, "A Back Propagation Algorithm with Adaptive Learning Rate And Momentum Coefficient".

- The Backpropagation Algorithm,
  http://page.mi.fu-berlin.de/rojas/neural/chapter/K7.pdf

- Error Backpropagation
  https://www.willamette.edu/~gorr/classes/cs449/backprop.html

- Derrick Unguyen, Bernard Widrow, "Improving the learning speed of 2-layer neural networks by choosing initial values of the Adaptive Weights".

- Amir Abolfazl Suratgar, Mohammad Bagher Tavakoli, and Abbas Hoseinabadi, "Modified Levenberg-Marquardt Method for Neural Networks Training".

- S. Sapna, Dr.A.Tamilarasi, M. Pravin Kumar, "Back Propagation Learning Algorithm based on Levenberg-Marquardt Algorithm".

- Vusumuzi Moyo, Khulumani Sibanda. "Training Set Size for Generalization Ability of Artificial Neural Networks in Forecasting TCP/IP Traffic Trends".

- Lecture notes.

- Wikipedia

- MATLAB codes are attached for reference.

2-Layer Backpropagation network MATLAB code:

```matlab
clear all;
clc;

% Get original wine-recognition dataset
x1range = 'B:N';
data = xlsread('wine_data.xlsx',x1range);
class1 = data((1:59),:);
class2 = data((60:130),:);
class3 = data((131:178),:);

% random noise component
noise = sqrt(0.01)*rand([12 13]);

% make the size of datasets equal(60 samples) by adding random noise components
class1_new = [class1;noise(1,:)];
class2_new = [class2((1:60),:)];
class3_new = [class3;noise((1:12),:)];
data_new = [class1_new;class2_new;class3_new];

%standardize the data
s=[];
for i =1:size(data_new,2)
    A = data_new(:,i);
    s(i)=std(A);
end
for j = 1:size(data_new,2)
    for i=1:size(data_new,1)
        data_new(i,j)= (data_new(i,j)- mean(data_new(:,j)))/s(j);
    end
end

% desired values
desired = zeros(180,3);
desired((1:60),1) = 1;
desired((61:120),2) = 1;
desired((121:180),3) = 1;

% This script assumes these variables are defined:
%   data_new - input data.
%   desired - target data.
x = data_new';
t = desired';

% Choose a Training Function
% 'trainlm' is usually fastest.
% 'traingdx' is slower but considers adaptive learning rate and momentum
%  factor
trainFcn = 'traingdx';
net.trainparam.lr = 0.07; % learning rate
net.trainparam.mc = 0.9; % momentum factor

% Choose a Performance Function
net.performFcn = 'mse';  % Mean Square error

% Create a Pattern Recognition Network
```

```
hiddenLayerSize = 30;
net = patternnet(hiddenLayerSize,'traingdx','mse');

%set the weight vectors to random value
net = setwb(net,(-0.13 + (0.26)*rand(30,1)));

% Setup Division of Data for Training, Validation, Testing
net.divideFcn = 'dividerand';  % Divide data randomly
net.divideMode = 'sample';  % Divide up every sample
net.divideParam.trainRatio = 70/100;
net.divideParam.valRatio =15/100;
net.divideParam.testRatio = 15/100;

% Choose Plot Functions
% For a list of all plot functions type: help nnplot
net.plotFcns = {'plotperform','plottrainstate','ploterrhist', ...
    'plotconfusion', 'plotroc'};

% Train the Network
[net,tr] = train(net,x,t);

% Test the Network
y = net(x);
e = gsubtract(t,y);
performance = perform(net,t,y);
tind = vec2ind(t);
yind = vec2ind(y);
percentErrors = sum(tind ~= yind)/numel(tind);

% Recalculate Training, Validation and Test Performance
trainTargets = t .* tr.trainMask{1};
valTargets = t .* tr.valMask{1};
testTargets = t .* tr.testMask{1};
trainPerformance = perform(net,trainTargets,y);
valPerformance = perform(net,valTargets,y);
testPerformance = perform(net,testTargets,y);

% View the Network
view(net)

% Plots
figure; plotperform(tr);
figure; plottrainstate(tr);
figure; ploterrhist(e);
figure; plotconfusion(t,y);
figure, plotroc(t,y);
```

3-layer Backpropagation network MATLAB code:

```
clear all;
clc;

x1range = 'B:N';
data = xlsread('wine_data.xlsx',x1range);
class1 = data((1:59),:);
class2 = data((60:130),:);
class3 = data((131:178),:);

%random noise component
noise = sqrt(0.01)*rand([12 13]);

% make the size of datasets equal(60 samples) by adding random noise components
class1_new = [class1;noise(1,:)];
class2_new = [class2((1:60),:)];
class3_new = [class3;noise((1:12),:)];
data_new = [class1_new;class2_new;class3_new];

% standardize the data
s=[];
for i =1:size(data_new,2)
    A = data_new(:,i);
    s(i)=std(A);
end
for j = 1:size(data_new,2)
    for i=1:size(data_new,1)
        data_new(i,j)= (data_new(i,j)- mean(data_new(:,j)))/s(j);
    end
end

% desired values
desired = zeros(180,3);
desired((1:60),1) = 1;
desired((61:120),2) = 1;
desired((121:180),3) = 1;

% This script assumes these variables are defined:
%   data_new - input data.
%   desired - target data.
x = data_new';
t = desired';

% Choose a Training Function
% 'trainlm' is usually fastest.
% 'traingdx' is slower but considers adaptive learning rate and momentum
%  factor
trainFcn = 'traingdx';
net.trainparam.lr = 0.07; % learning rate
net.trainparam.mc = 0.9; % momentum factor

% Choose a Performance Function
net.performFcn = 'mse';  % Mean Square error

% Create a Pattern Recognition Network
hiddenLayerSize = [30 15];
```

```
net = patternnet(hiddenLayerSize,'traingdx','mse');

% Setup Division of Data for Training, Validation, Testing
net.divideFcn = 'dividerand';  % Divide data randomly
net.divideMode = 'sample';  % Divide up every sample
net.divideParam.trainRatio = 70/100;
net.divideParam.valRatio = 15/100;
net.divideParam.testRatio = 15/100;

% Choose Plot Functions
net.plotFcns = {'plotperform','plottrainstate','ploterrhist', ...
    'plotconfusion', 'plotroc'};

% Train the Network
[net,tr] = train(net,x,t);

% Test the Network
y = net(x);
e = gsubtract(t,y);
performance = perform(net,t,y);
tind = vec2ind(t);
yind = vec2ind(y);
percentErrors = sum(tind ~= yind)/numel(tind);

% Recalculate Training, Validation and Test Performance
trainTargets = t .* tr.trainMask{1};
valTargets = t .* tr.valMask{1};
testTargets = t .* tr.testMask{1};
trainPerformance = perform(net,trainTargets,y);
valPerformance = perform(net,valTargets,y);
testPerformance = perform(net,testTargets,y);

% View the Network
view(net)

% Plots
% Uncomment these lines to enable various plots.
figure, plotperform(tr);
figure, plottrainstate(tr);
figure, ploterrhist(e);
figure, plotconfusion(t,y);
figure, plotroc(t,y);
```