ANN_MLP_ Iris data

The number of hidden layer cells reached 4 through trial and error. In the first layer, we have the sigmoid function and in the second layer, we have the purelin linear function, which we first define in separate files so that we can easily use them in the main program.

:Purelin function

```
function [a] = TransferFcn_purelin(w,p,B)
a=w*p + B;
end
```

:Sigmoid function

```
function [a] = TransferFcn_logsig(w,p,B)
n=w*p + B;
a=1./(1+exp(-n));
end
```

Now let's look at the main codes.

LMS program :

In the main program, we first read the input and output values of the iris data set and store them in two variables, p and t. We define the maximum epoch and the evaluation and training performance vector. We also enter the number of hidden layer neurons. At the end of this section, we form the matrices w2, w1, b2, b1 (by random selection between 1 and -1)

```
close all
clear all
[p t] = iris_dataset;
maximum_epoch = 200;
epochs = 0:maximum_epoch;
learningrate = .001;
valRatio = 0.5;
trainRatio = 0.5;
hidden = 4;
vperformance = zeros(1, maximum_epoch+1);
performance = zeros(1, maximum_epoch+1);
w1 = 2*rand(hidden, size(p,1))-1;
b1 = 2*rand(hidden, 1)-1;
w2 = 2*rand(size(t,1), hidden)-1;
b2 = 2*rand(size(t,1), 1)-1;
```

The MSE value is calculated for each epoch.

```
vperformance_mat = power(t(:,valIndex) - TransferFcn_purelin(w, p(:,valIndex),
b),2);
vperformance(1) = sum(vperformance_mat(:))/length(vperformance_mat(:));
performance_mat = power(t(:,trainIndex) - TransferFcn_purelin(w,
p(:,trainIndex), b),2);
performance(1) = sum(performance_mat(:))/length(performance_mat(:));
```

Now we come to the part where we talk about the middle ground.

In each part, we first calculate a1 directly with the sigmoid function, and then with the pureline function, we obtain a2. Finally, the error must be calculated.

In the next part, we do the reverse and BP and calculate the sensitivity of the layers. Derivative_sigmoid is the derivative of the sigmoid function that we know is written as follows:

$$\dot{\mathbf{F}}^m(\mathbf{n}^m) = \begin{bmatrix} \dot{f}^m(n_1^m) & 0 & \dots & 0 \\ 0 & \dot{f}^m(n_2^m) & \dots & 0 \\ \vdots & \vdots & & \vdots \\ 0 & 0 & \dots & \dot{f}^m(n_{S^m}^m) \end{bmatrix}$$

In this matrix, each object on the principal diameter is the same sigmoid derivative obtained as y: 1 (1-y) Now we update the weight values and biases. The MSE is also calculated at the end of the general loop.

```
for epoch = 2:(maximum_epoch+1)

    for index = trainIndex

        a1 = TransferFcn_logsig(w1, p(:,index), b1);
        a2 = TransferFcn_purelin(w2, a1, b2);
        e = t(:,index) - a2;

        s2 = -2*e;
        Derivative_sigmoid=[a1(1)*(1-a1(1)) 0 0 0;0 a1(2)*(1-a1(2)) 0 0;0 0
a1(3)*(1-a1(3)) 0;0 0 0 a1(4)*(1-a1(4))];
        s1 = Derivative_sigmoid*w2'*s2;

        w2 = w2 - learningrate*s2*a1';
        b2 = b2 - learningrate*s2;
        w1 = w1 - learningrate*s1*p(:,index)';
        b1 = b1 - learningrate*s1;

    end
     vperformance_mat = power(t(:,valIndex) - TransferFcn_purelin(w2,
TransferFcn_logsig(w1, p(:,valIndex), b1), b2), 2);
    vperformance(epoch) = sum(vperformance_mat(:))/length(vperformance_mat(:));
    performance_mat = power(t(:,trainIndex) - TransferFcn_purelin(w2,
TransferFcn_logsig(w1, p(:,trainIndex), b1), b2), 2);
    performance(epoch) = sum(performance_mat(:))/length(performance_mat(:));

end
```

Batch program:

The difference between this part and the previous part is only in the learning part, in which case the variables are updated at the end of each epoch.

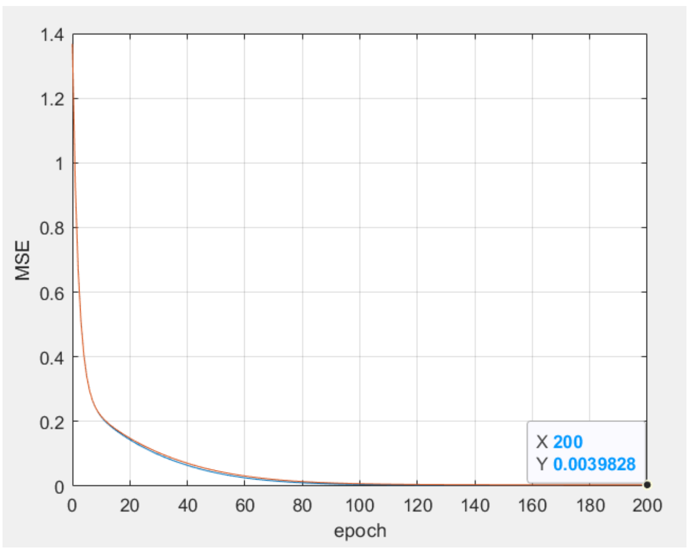
```
for epoch = 2:(maximum_epoch+1)
```

```
    dw2 = 0;
    dw1 = 0;
    db2 = 0;
    db1 = 0;
    for index = trainIndex

        a1 = TransferFcn_logsig(w1, p(:,index), b1);
        a2 = TransferFcn_purelin(w2, a1, b2);
        e = t(:,index) - a2;

        s2 = -2*e;
        Derivative_sigmoid=[a1(1)*(1-a1(1)) 0 0 0;0 a1(2)*(1-a1(2)) 0 0;0 0
a1(3)*(1-a1(3)) 0;0 0 0 a1(4)*(1-a1(4))];
        s1 = Derivative_sigmoid*w2'*s2;


        dw2 = dw2 + s2*a1';
        db2 = db2 + s2;
        dw1 = dw1 + s1*p(:,index)';
        db1 = db1 + s1;

    end

    w2 = w2 - (learningrate/length(trainIndex))*dw2;
    b2 = b2 - (learningrate/length(trainIndex))*db2;
    w1 = w1 - (learningrate/length(trainIndex))*dw1;
    b1 = b1 - (learningrate/length(trainIndex))*db1;

    vperformance_mat = power(t(:,valIndex) - TransferFcn_purelin(w2,
TransferFcn_logsig(w1, p(:,valIndex), b1), b2), 2);
    vperformance(epoch) = sum(vperformance_mat(:))/length(vperformance_mat(:));
    performance_mat = power(t(:,trainIndex) - TransferFcn_purelin(w2,
TransferFcn_logsig(w1, p(:,trainIndex), b1), b2), 2);
    performance(epoch) = sum(performance_mat(:))/length(performance_mat(:));

end
```

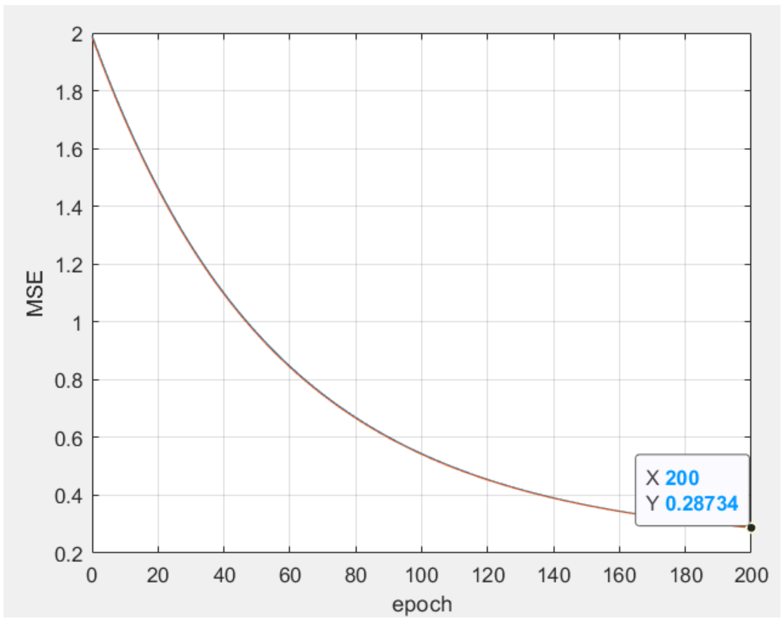
Isolation of Versicolor and Setosa data:

```
t = t([1 2],[1:50 51:100]);
p = p(:, [1:50 51:100]);
```

LMS:



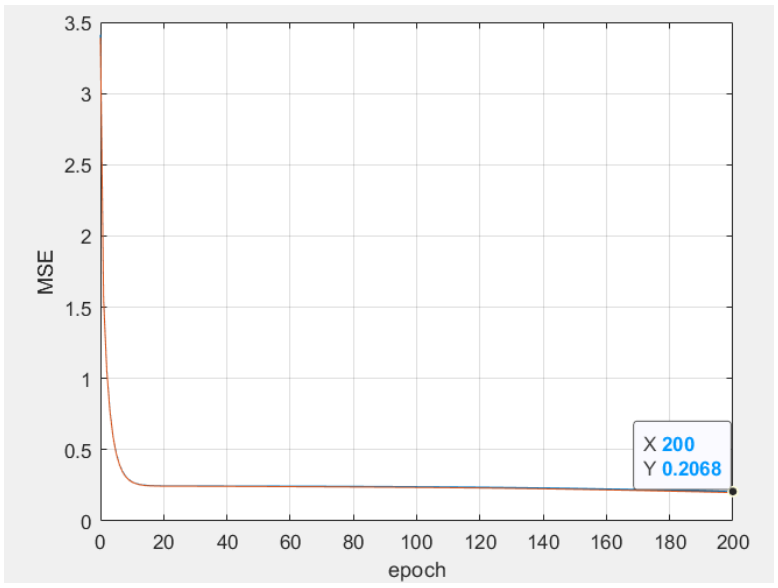
BATCH:



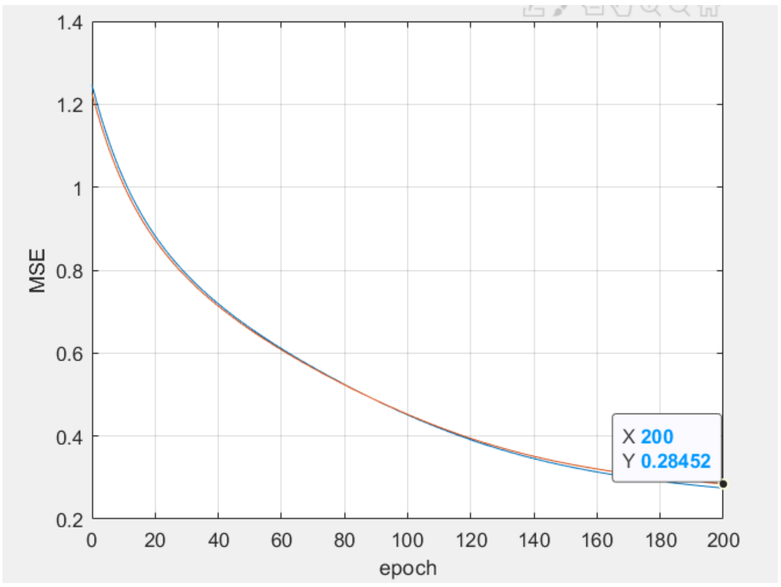
Isolation of Versicolor and Setosa data :

```
t = t([2 3],[51:100 101:150]);
p = p(:, [51:100 101:150]);
```

LMS:

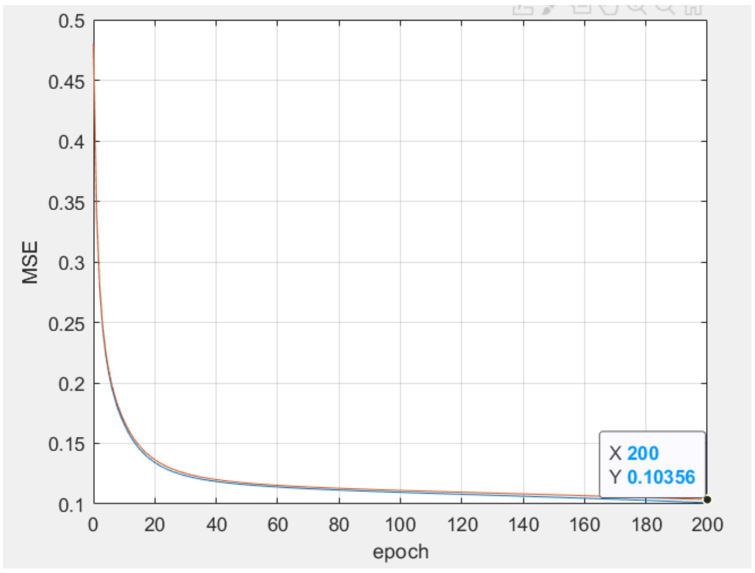


BATCH:

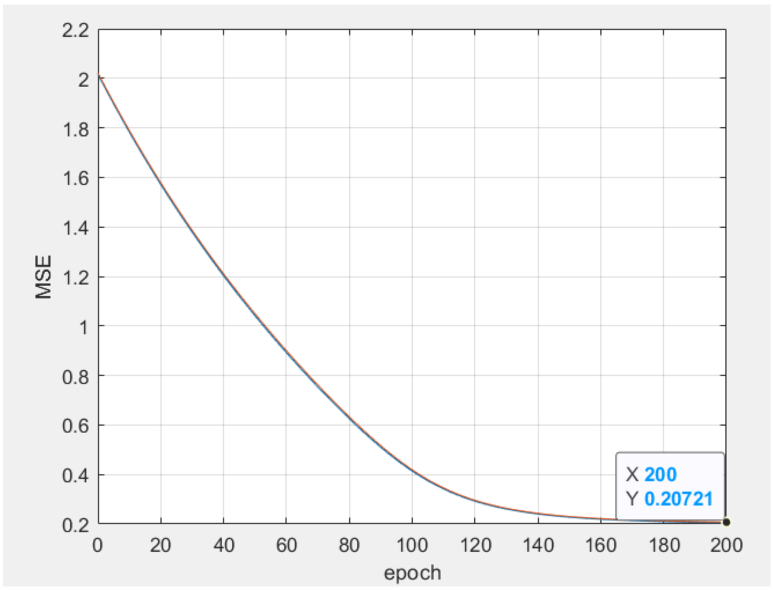


Data separation of all three classes:

LMS:



BATCH:



Versicolor values are not categorized correctly.

In sections B and C, because we compare versicolor data and this data is not linearly separable, the MSE increases.

In general, in batch, we have less MSE and it is more convenient and faster.

LMS has a better response and may be due to the fact that the batch gets stuck in local min but not lms.

Comparison of results obtained from MLP, Linear, Adline and SLP networks:

SLP has a relatively poor performance. Because iris data is nonlinear.

Adaline also does linear separation, but because it reduces MSE, it performs better than SLP.

MLP can also separate nonlinear data because it has a hidden layer, so it is more convenient.

If we use linear data, the result will certainly be the opposite, and SLP is the most appropriate network; Because it is simpler and suitable for linear data