U08280 – ADVANCED ARTIFICIAL INTELLIGENCE

Assessed Coursework

Categorization of a set of typical real data using a neural network

Guillaume Ardaud

09010297@brookes.ac.uk

Contents

Abstract 2

Introduction 2

1) Conception of the network 2

a) Constitution of the data set and representation of input and output patterns 2

b) Topology of the chosen network 5

c) Choice of network 7

2) The network in action 10

a) Effect of network constants on network training 10

b) Verifying the network 13

c) Determining iris species 13

d) Reaction of the network to noisy input 15

Conclusion 16

Bibliography 17

On the cover page, photographs from the species of Irises discussed in this paper- from left to right: *Iris setosa, Iris versicolor, Iris virginica.*

Source: Wikimedia Foundation.

Abstract

In this report, we devise a methodology for identifying the species of an iris amongst 3 based on 4 distinctive features. We first cover the constitution of the data set and input patterns. We then determine the layout and structure of the neural network we will use for the classification. We continue by testing the network in real life conditions. We conclude with a review of the methods used, and how they could be improved.

Introduction

We are dealt two CSV files: the first one, 'iristrainingdata.csv' containing the measures for 120 irises as introduced by Sir Ronald Aylmer Fisher (1), as well as the species to which they belong. This file will be used to train the neural network. The second file, 'iristestdata.csv', contains the measurements for 30 irises that have not been identified. Once our network is complete, we will identify the irises measured in this file.

1) Conception of the network

In this section, we will first detail how the data sets are built; we will then explain the methodology for creating the neural network in MATLAB and training it.

a) Constitution of the data set and representation of input and output patterns

The first step is to write a MATLAB function that will import the data contained in the CSV files into a matrix, which we will be able to use as the input pattern of our neural network. In the CSV files, the data is organized in the following manner:

- Each line corresponds to an iris.
- The values are comma separated.
- The first value is the number of the sample.
- The four following values are, in the following order: the Sepal length in cm, the sepal width in cm, the petal length in cm, the petal width in cm.
- The final value is a string corresponding to the species of iris.

We are going to write a MATLAB function which will import the data for an arbitrary number of iris in a matrix- to this end, we are going to use the *textread()* function.

```
[input{1:4}] = textread(file,'%*d %f %f %f %f %*s','delimiter',',',
'headerlines', 120-n);

input=cell2mat(input);
```

The second line is necessary; otherwise the data is stored in a cell-structure, which we cannot use for our network.

We also want to scale the data to the [0..1] range. MATLAB's variable editor tells us that the maximum value in the imported data is below 10; we can therefore safely assume that no iris has a

petal length/width or sepal length/width greater than 10 centimeters. What we are going to do is simply divide all the values in the matrix by 10, so that they fit in the [0..1] range. This operation is done in a very straightforward manner:

```
input=input/10;
```

Once all of these operations are achieved, our matrix has the following structure (where each letter corresponds to one input pattern and its subscript the index for each of its values):

$$\begin{pmatrix} a_1 a_2 \dots a_i \\ b_1 b_2 \dots b_i \\ \vdots \\ \alpha_1 \alpha_2 \dots \alpha_i \end{pmatrix}$$

However, in order to use the matrix with the neural net in MATLAB, we need it to be in the following form:

$$\begin{pmatrix} a_1 b_1 \dots \alpha_1 \\ a_2 b_2 \dots \alpha_2 \\ \vdots \\ a_i b_i \cdots \alpha_i \end{pmatrix}$$

Fortunately, this transformation is very easy to achieve, as it is just a transposition of the matrix. Therefore we just have to add the following line in MATLAB:

```
input=input';
```

The final thing we must do is generate the output pattern matrix from the CSV file. Unfortunately, for our output pattern, we need to use integers whereas the CSV file provides string- therefore we cannot directly use the values from the CSV file. What we are going to do is first create a matrix containing the values from the CSV file:

```
    strings=textread(file,'%*d %*f %*f %*f %*f %s','delimiter',',',
'headerlines', 120-n
```

What we then do is replace each string by an integer as follows:

Iris setosa	0
Iris versicolor	1
Iris virginica	2

And here is the resulting MATLAB code:

```
   output=[1:n];

    for i=1:length(strings
        if (strcmp(strings(i),'Iris-setosa'))
            output(i)=0;
        elseif (strcmp(strings(i),'Iris-versicolor'))
            output(i)=1;
        elseif (strcmp(strings(i),'Iris-virginica'))
            output(i)=2;
        end
    end
```

Our final *importIris()* function takes one variable as input: the number of iris samples we wish to import. It returns two variables: one input pattern for the neural network, and one output pattern.

Here is the corresponding code:

```
%imports the given amount of iris in a input matrix for the neural net,
%along with the corresponding output matrix
function [input,output]=importIris(n)
    file = 'iristrainingdata.csv'; %the file in which the data is located

    [input{1:4}] = textread(file,'%*d %f %f %f %f %*s','delimiter',',',
'headerlines', 120-n); %first we import the input for the number of iris
the user wants
    input=cell2mat(input); %then we put it in a matrix
    input=input'; %then we transpose the matrix so that we can use it as an
input for the neural net
    input=input/10; %scale the data

    strings=textread(file,'%*d %*f %*f %*f %*f %s','delimiter',',',
'headerlines', 120-n); %first we import the output for the number of iris
the user wants
    output=[1:n]; %we create a matrix in which we are going to put the
numerical values

    for i=1:length(strings)%then we replace the strings by numerical values
        if (strcmp(strings(i),'Iris-setosa'))
            output(i)=0;
        elseif (strcmp(strings(i),'Iris-versicolor'))
            output(i)=1;
        elseif (strcmp(strings(i),'Iris-virginica'))
            output(i)=2;
        end
    end
end
```

So for example, if one wanted to create an input pattern and output pattern for 40 irises, one would use the following MATLAB command:

```
[input, output]= importIris(20);
```

b) Topology of the chosen network

In order to determine the amount of hidden layers the network needs, we need to determine if the input space for the iris classification is linearly separable. To do that, we are going to write a MATLAB function that will plot on a graph the measures from the irises in the *iristrainigdata.csv* file. As there are 120 measures in this file, it should give us a pretty good idea of how the input space is divided.

Furthermore, we need to find a way to represent the data on a 2 dimensional graph- since we have 4 measures, we would need a 4 dimensional graph if we had wanted to plot all the measures separately.

After research, we found out that the petal length and width as well as the sepal length and width are correlated, as they are genetically determined and the ratio is similar within a single iris species, no matter the size of the iris.

Hence we wrote a function, *irisPlot()*, which plots the measures from the CSV file onto a graph. Here is the code for that function:

```
function irisPlot(input, output)
    hold on
    for i=1:length(input)
        sepalRatio=input(1,i)/input(2,i);
        petalRatio=input(3,i)/input(4,i);
        if (output(i)==0)
            plot(sepalRatio,petalRatio,'.');
        elseif (output(i)==1)
            plot(sepalRatio,petalRatio,'.g');
        elseif (output(i)==2)
            plot(sepalRatio,petalRatio,'.m');
        end
    end
    xlabel('Sepal length/width ratio');
    ylabel('Petal length/width ratio');
    title('Input space for the iris','FontSize',12);
    hold off
end
```

Figure 1 shows the graph obtained with the following MATLAB command:

```
[input,output]=importIris(120); irisPlot(input, output);
```

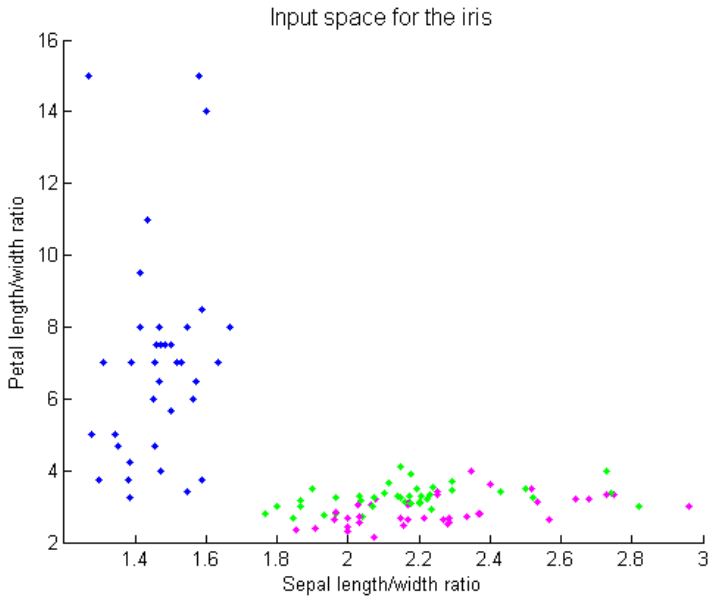

Figure 1 Plot of the iris measures used

What we want to know now is whether the input space is linearly separable or not; as this will directly influence the number of hidden layers we will need to use in the neural network. Here, it is clear that the input space is not linearly separable, as we have 3 different possible outputs.

We observe that the Iris Setosa (in blue) is divided from the rest; however the Iris Versicolor (in green) and the Iris Virginica (in magenta) seem to have overlapping values in some cases. This is however explained by the fact that some of the input is noisy- in reality we can still differentiate the two. Figure 2 shows the divided input space.

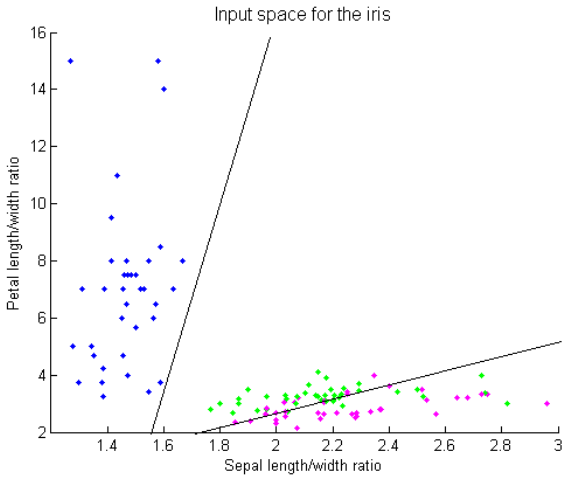

Figure 2 Separation of the input space

Another possibility would have been to plot all the graphs possible comparing each attribute to one another (for a total of 12 graphs). Such a collection of graphs is made available by the Wikimedia, and is shown on Figure 3. We shall note that the results are the same as the ones obtained previously: the input space is not linearly separable.

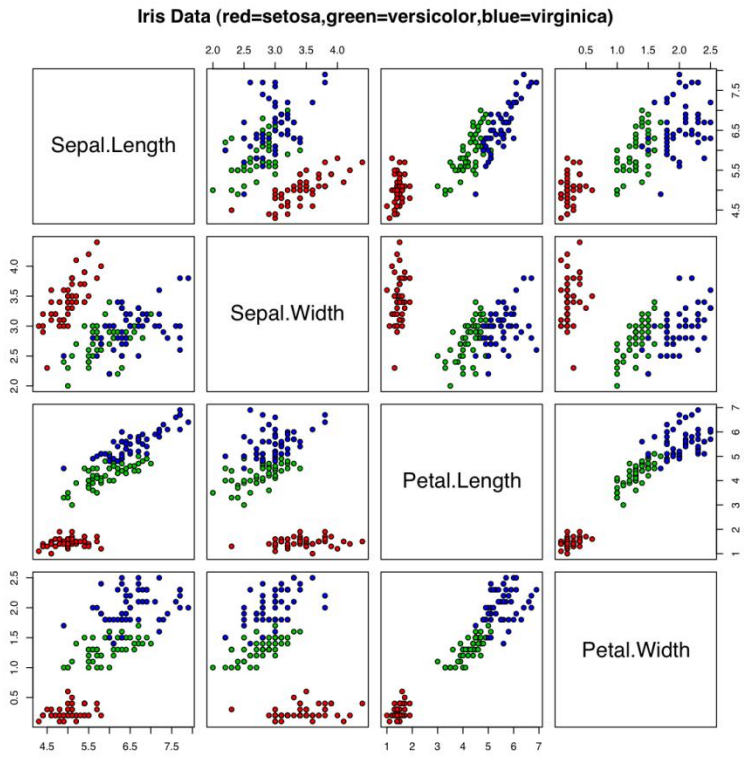

Figure 3 Scatterplot of the data set (source: Wikimedia foundation)

As a result, our network is going to have two hidden layers.

c) Choice of network

There are several types of neural network models. One that is often used in data classification is the feed forward back propagation model (2).

Here are some of the criteria that indicate that a feed forward back propagation network is adapted (2):

1. A large amount of input/output data is available, but we're not sure how to relate it to the output.
2. The problem appears to have overwhelming complexity, but there is clearly a solution.
3. It is easy to create a number of examples of the correct behavior.
4. The solution to the problem may change over time, within the bounds of the given input and output parameters (i.e., today 2+2=4, but in the future we may find that 2+2=3.8).

In our cases, all of these criteria are satisfied:

1. We have several hundreds of measures (precisely 120*4=480 in our case), but we do not know the rules that relate them to the output
2. There is a clear solution, as it is possible to recognize the species to which a given iris belongs- however the complexity of the problem is far from trivial.

3. It is easy for us to create examples: all we have to do is measure the length and width of the petals and sepals of irises we can find in nature.
4. In the future, we might find certain irises that belong to a species but whose measures deviate from what we collected in our actual sample- yet they would still belong to that species.

We are going to use a feed forward back propagation network. A feed forward network is created in MATLAB as follows:

```
net = newff(input, output, units);
```

Where input is the *input* pattern, *output* is the output pattern, and *units* is the number of neurons we want to use in the hidden layers.

This leads us to ask ourselves how many neurons we need. It is hard to determine precisely what the right amount is, especially in cases like this with complex input patterns.

To this mean, we wrote a function, *variateHiddenUnits()*, which creates a network with n hidden units and runs i simulations. The function then displays what the classification success ratio is for that network. Here is the code for that function:

```
%this function runs i times a network with n hidden units
function variateHiddenUnits(n,loop)
    %we import the input and output patterns
    [input,output]=importIris(120);
    %we create the net
    net=newff(input,output,n);

    net.divideParam.trainRatio = 1; % use all inputs for training
    net.divideParam.valRatio = 0; % and none for validation
    net.divideParam.testRatio = 0; % or testing

    net.trainParam.min_grad= 1e-0100;%we set the minimum error gradient
    net.trainParam.lr=0.05; %we set the learning rate
    net.trainParam.epochs = 100; %we set the number of epochs

    total=0;
    for i=1:loop
        net=init(net);
        [net,errors] = train(net, input, output);%train the net
        result = sim(net, input);%run the net on the input pattern
        total=total+sum(round(result)==output); %display the number of
successes/total patterns
    end
    disp('Success rate:');
    rate=((total/loop)/120)*100;
    disp([num2str(rate),'%']);
end
```

We ran the function for several numbers of units, with a number of i=10 simulations each time.

Table 1:Success ratio for different numbers of hidden units

Number of hidden units	Success ratio
1	96.6667%
2	97.6667%
3	98.3333%
5	99%
7	99.6667%
10	99.9917%
13	100%
15	100%

As we can see, we have a very high success ratio with a relatively small amount of hidden units. Therefore, the choice lies in the compromise we wish to make between computation time (which increases dramatically as the number of hidden units rises) and overall efficiency.

In our case, we choose to use 5 hidden units- it guarantees a very good success ratio, and reduces the total computation time in our future tests and experimentation.

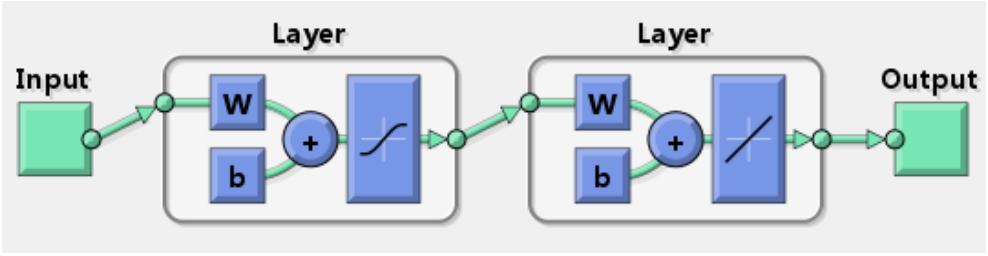

Figure 4 Architecture of the final neural network

Figure 1 shows the architecture of the complete feed forward back propagation neural network we are going to use for the classification, as made available by MATLAB's *nntool*.

2) The network in action

In the previous section, we covered the choices behind the architecture and layout of the network. In this section, we are going to review the decisions made concerning the training process of the network.

a) Effect of network constants on network training

When training the network, there are several constants we can modify. In this report, however, we will only study the effect of three of these constants:

- Learning rate
- Number of epochs
- The minimum error gradient

These three constants are chose for the proven direct impact they have on the network's training time and efficiency.

We are going to set each parameter individually through the following commands in the MATLAB console:

```
net.trainParam.lr=lr; %for the learning rate
net.trainParam.epochs = epochs; %for the number of epochs
net.trainParam.min_grad= min_grad; %for the minimum error gradient
```

After which we train the network using the following command:

```
[net,errors] = train(net, input, output);
```

The variable *net* being the trained network, and *errors* being a record that holds data that we can use for plotting the error graph.

It is important to note that for this coursework, we used MATLAB 2009b, which includes by default an early stopping behavior; in some cases, this can actually limit the performance of the network by stopping the network too early on datasets that are too small. In order to prevent this, we will completely override the early stopping behavior with the following commands:

```
net.divideParam.trainRatio = 1;
net.divideParam.valRatio = 0;
net.divideParam.testRatio = 0;
```

This adds to the training time, but ensures us that our network will provide reliable results.

The first parameter we are going to experiment with is the learning rate. In order to achieve that, we implemented a function, *plotLearningRate*(), which plots the error curve for 5 different learning rates. Here is the MATLAB code for *plotLearningRate()*:

```
%plots the error curve for various learning rates

function plotLearningRate()
    learningRates=[0.001 0.02 0.05 0.2 0.5];
    colors={'green','blue','red','magenta','black'};
```

```
    lStrings={};
    for i=1:length(learningRates)
        lStrings=[lStrings strcat('Learning rate:
',num2str(learningRates(i)))];
    end

    %set the graph's basic properties
    axis([0 1000 0 0.02])
    xlabel('Epochs');
    ylabel('Performance');
    title('Error graph for various learning rates','FontSize',12);
    %this is for plotting multiple curves on a same graph
    hold on;
    %import the inputs and outputs from the traning set
    [trainInput,output]=importIris(120);
    %create a network...
    net=createNetwork(trainInput,output);

    for i=1:length(learningRates)
        %reinitialize the weights
        net=init(net);
        %set the learning rate
        net.trainParam.lr=learningRates(i);
        %...and train the network
        [net,tr] = train(net, trainInput, output);
        %plot the error curve
        p=plot(tr.epoch,tr.perf,'Color',char(colors(i)));
    end

    legend(lStrings);

    hold off;
end
```

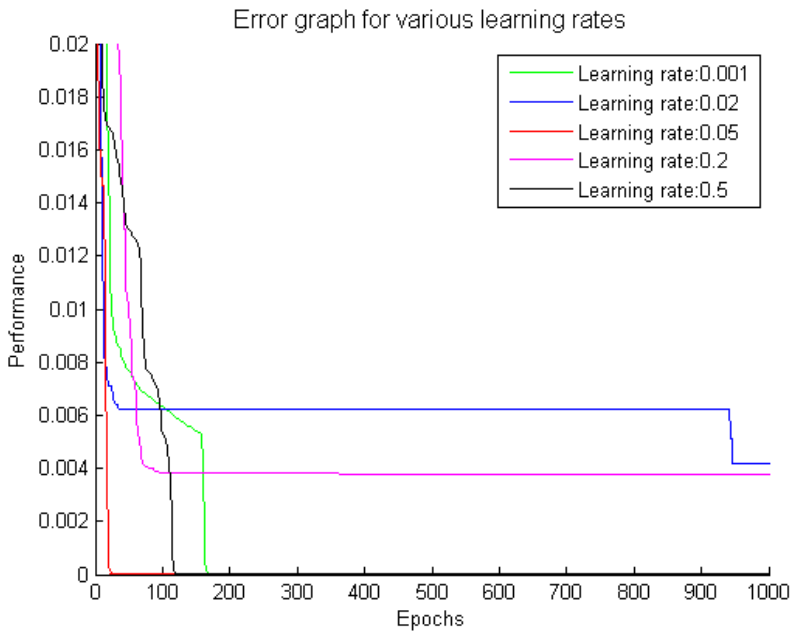

Figure 5 Error graph for various learning rates

Figure 5 shows the graph generated by said function. We observe that all training rates provide a very good performance; however it seems like a learning rate of 0.02 provides the best and fastest performance. We are therefore going to use a training rate value of 0.02 in our neural network.

The second factor we will vary is the number of epochs used in the network's training. We have implemented a function, *variateEpochs()*, which works in a way similar to *variateHiddenUnits()* (which we have detailed in the previous section), and have ran it for 3 different epochs values. The results are shown in table 2.

Number of epochs	Success ratio
100	99.3333%
1000	99.5833%
10000	99.75%

Table 2: Success ratio for different epochs

Here, we can see that the higher the number of epochs, the higher the success rate. However, this is again done at the cost of computation time. The success ratio given by 100 epochs is more than enough in our case, and we will therefore use this value for the neural network.

Finally, the last constant to determine is the minimum error gradient. Again, we have written a function, *variateMinGrad()*, that works in a similar manner as *variateEpochs()* and *variateHiddenUnits()*, which displays the success ratio for 3 different values for the minimum error gradient.

Value of the minimum error gradient	Success ratio
1e-010	99.4167%
1e-0100	99.5%
1e-01000	99.75%

Table 3: Success ratio for different error gradients

We can see here that the success ratio augments with the minimum error gradient used. However, so does the computation time- on the system we used, training a network with a minimum error gradient of 1e-01000 took several minutes. We find that a minimum error gradient of 1e-0100 is sufficient for our purposes, and will use this value for the training of our network.

Now that we have determined all the optimal constants for the network, we wrote a function, *createNetwork()*, which creates and trains a network for iris classification. The code is below.

```
%creates a neural network for iris classification
function [net]=createNetwork(input,output)
    units=5; %number of hidden units
```

```
    lr=0.01; %learning rate for the network
    epochs= 1000; %number of epochs
    mg=1e-0100; %minimum error gradient

    net = newff(input, output, units); %we create a feed forward network

    net.divideParam.trainRatio = 1; % use all inputs for training
    net.divideParam.valRatio = 0; % and none for validation
    net.divideParam.testRatio = 0; % or testing

    net.trainParam.lr=lr; %we set the learning rate
    net.trainParam.epochs = epochs; %we set the number of epochs
    net.trainParam.min_grad= mg; %we set the minimum error gradient

    %we train the network
    [net,errors] = train(net, input, output);
end
```

b) Verifying the network

We have determined the optimal constants for our network and trained it; we can now verify its reliability using the data provided in *iristrainingdata.csv*. We have written a MATLAB function for this purpose, *checkNetwork()*, which takes one parameter: the number of simulations to run. It runs the given number of simulation using the values determined in the previous sections, and displays the average success rate for the given amount of simulations.

The structure of the function is similar to the other simulation functions (*variateEpochs(), variateHiddenUnits(), variateMinGrad()*).

Running the simulation 10 times displays the following:

```
>> checkNetwork(10)
Success rate:
99.75%
```

Our neural network has an average success rate of 99.75%, which is an excellent value- we can therefore safely say that the network we trained is very reliable in the categorization of irises by species.

c) Determining iris species

Now that we have determined the optimal way to train our network and have verified its reliability, we can use it to classify the irises from *iristestdata.csv.* To achieve that, we have written a MATLAB function, *classifyIris()*, which does not take any parameter: it simply creates and trains a network with the values previously discussed, and then classifies the iris from the *iristestdata.csv* file by species. It returns an array with the results, which we can pass to the function *nameIrises()* to have the full species name for each iris.

To obtain the name of irises from the *iristestdata.csv* file, we run the following MATLAB command:

```
names=nameIrises(classifyIris());
```

The results obtained, along with the values from the CSV file and the raw results, are shown in table 4.

Sample n°	Sepal length	Sepal width	Petal length	Petal width	Result	Species
1	6.2	3.4	5.4	2.3	2.24	'Iris-virginica'
2	6.7	3.3	5.7	2.5	2.03	'Iris-virginica'
3	5	3.3	1.4	0.2	0.00	'Iris-setosa'
4	6.7	3.1	5.6	2.4	2.02	'Iris-virginica'
5	6.5	2.8	4.6	1.5	1.20	'Iris-versicolor'
6	4.4	3.2	1.3	0.2	0.00	'Iris-setosa'
7	5	3.5	1.6	0.6	0.02	'Iris-setosa'
8	4.9	2.4	3.3	1	0.89	'Iris-versicolor'
9	5	3.5	1.3	0.3	0.00	'Iris-setosa'
10	5.7	2.8	4.5	1.3	0.86	'Iris-versicolor'
11	4.6	3.2	1.4	0.2	0.00	'Iris-setosa'
12	5.2	2.7	3.9	1.4	0.85	'Iris-versicolor'
13	5.3	3.7	1.5	0.2	0.00	'Iris-setosa'
14	6.9	3.1	5.1	2.3	1.95	'Iris-virginica'
15	5.8	2.7	5.1	1.9	1.97	'Iris-virginica'
16	4.8	3	1.4	0.3	0.00	'Iris-setosa'
17	6.7	3	5.2	2.3	1.96	'Iris-virginica'
18	5.5	2.3	4	1.3	1.00	'Iris-versicolor'
19	6.9	3.1	4.9	1.5	1.11	'Iris-versicolor'
20	6.8	3.2	5.9	2.3	2.02	'Iris-virginica'
21	5.1	3.8	1.9	0.4	0.01	'Iris-setosa'
22	6.3	2.5	5	1.9	2.02	'Iris-virginica'
23	5.1	3.8	1.6	0.2	0.00	'Iris-setosa'
24	4.5	2.3	1.3	0.3	0.26	'Iris-versicolor'
25	6.9	3.1	5.4	2.1	1.92	'Iris-virginica'
26	6.3	3.3	4.7	1.6	1.27	'Iris-versicolor'
27	6.6	2.9	4.6	1.3	1.03	'Iris-versicolor'
28	6.5	3	5.2	2	1.83	'Iris-virginica'
29	7	3.2	4.7	1.4	0.90	'Iris-versicolor'
30	6.4	3.2	4.5	1.5	1.10	'Iris-versicolor'

Table 4: Classification of the irises from the “iristestdata.csv” file

We can observe that very few samples were classified with ambiguous values- most results are within ±0.20 from the output values (0, 1 and 2). This shows that our network can be used to classify irises with high confidence, and is confirmed by the extremely high success rates obtained in the previous section.

d) Reaction of the network to noisy input

An interesting thing to study is the reaction of the network to noisy input; in a real life situation, it is possible that the system or person in charge of taking the measures on the irises would make mistakes. Studying how the network copes with that gives further insight into how reliable the network is if it were used in real life.

What we are going to do is take 10 measures from the *iristestdata.csv* file, and write a script that will randomly generate a variate for this measure, within a range of ±5, ±10, ±15, ±20 or ±50%.

The script, *noiseValues,* is as follows:

```
%this function adds noise to the values by a margin of +/-n%
function [randMatrix] = noiseValues(values,n)
    n=n/100; %we divide by 100 to have a percentage
    s=size(values);
    randMatrix=zeros(s);

    for i=1:s(1) %we loop through the matrix
        for j=1:s(2)
            noiseValue=values(i,j)*n*2; %see line n+2 for explanation of
why we multiply by 2
            noise=noiseValue*rand; %we generate a random value in that
interval
            noise=noise-(noiseValue/2); %we divide the value we substract
by two (to haev a negative lower bound)
            randMatrix(i,j)=values(i,j)+noise; %we add the noise to the
current value
        end
    end

end
```

We train the network with non-noised data, and then do simulations with the noised data as input.

Table 5 shows the results obtained for various amounts of noise (the red cells indicate a divergence from the original result greater than 0.5, yellow cells greater than 0.2, and green lower than 0.2).

S lgth	S wdth	P lgth	P wdth	Result	Result - 5%	Result - 10%	Result - 15%	Result - 20%	Result - 50%
6.3	2.3	4.4	1.3	1.00	1.04	0.29	3.11	3.21	0.95
5.5	4.2	1.4	0.2	0.00	-0.39	-0.34	-0.36	0.49	0.30
7.1	3	5.9	2.1	2.00	2.32	2.21	1.30	2.77	2.82
6.1	2.8	4	1.3	1.00	1.06	1.01	0.94	0.57	0.72
6	2.9	4.5	1.5	1.00	1.04	0.94	1.12	1.86	0.89
4.8	3.1	1.6	0.2	0.00	-0.01	0.23	-0.05	-0.26	2.16
5.8	2.7	3.9	1.2	1.00	1.00	0.78	1.16	1.47	1.93
6	2.2	5	1.5	2.00	2.03	1.66	1.35	1.55	2.22
5.4	3.9	1.3	0.4	0.00	0.54	-0.59	0.83	-0.72	1.34
7.2	3.2	6	1.8	2.00	2.07	1.45	1.84	2.60	0.87

Table 5: Results of a classification with noisy input

We observe that the network deals rather well with a noise of up to 5%. However, the reliability drops immensely for a 10% error rate. Past that point, the correctness of the results is varied, which

leads us to think that any correct answer is a mere coincidence. We believe that past 20%, the margin of error is too great for the network to be able to cope with it decently, and even a human expert would make erroneous conclusions. However, it would be interesting to explore the possibilities which would allow us to deal better with the cases where there is around 10% of noise.

Conclusion

In this report we have seen how to design and implement neural networks in MATLAB to very successfully classify iris according to their species, with only 4 measures. We have also covered the question of how different factors influence the reliability of the network.

While the actual process of identifying the irises was extremely successful, we found out that the network's performance dropped quickly when the input data was noisy.

There are numerous approaches to this problem of data classification, and the iris data set has been used for many if them (3) (4) (5). It would be interesting to expand upon the work done in this coursework to develop a hybrid system for iris classification, for example by using decision trees (6).

Another interesting area of investigation could be the ability for the system to detect noisy input, when detecting inconsistencies with the general pattern of the data it was trained with. However, this exceeds by far the capabilities of a simple feed forward back propagation neural network, and would require major additions to the system.

Bibliography

1. **Wikipedia.** Iris Flower Data Set. [Online] http://en.wikipedia.org/wiki/Iris_flower_data_set.

2. **McCollum, Pete.** An Introduction to Back-Propagation Neural Networks. *Seattle Robotics.* [Online] http://www.seattlerobotics.org/encoder/nov98/neural.html.

3. **Geordie.** Classifying the Iris data set . [Online] http://dwave.wordpress.com/2009/08/15/classifying-the-iris-data-set/.

4. **Xuchen, Yao.** Iris Data Set Classification Using Support Vector Machine. [Online] http://sites.google.com/site/yaoxuchen/Home/iris_svm_en.

5. **Nikolaev, Nikolay Y.** Iris classification using probabilistic neural networks. [Online] http://homepages.gold.ac.uk/nikolaev/311pnn.htm.

6. **MATHWORKS.** Iris classification using decision trees. [Online] http://www.mathworks.com/products/demos/statistics/classdemo.html.

7. **MATLAB Documentation.** Architecture of backpropagation neural networks. [Online] http://www.mathworks.com/access/helpdesk/help/toolbox/nnet/backpro4.html#32553.