Artificial Neural Network – Iris Data

This is a partial listing of the Java code for the Artificial Neural Network program.

It includes all the code from the **Initialize** and **Evolution** sections, but does not include the variables' descriptions or the display section. The specification of the variables, for example the type of each variable and reasonable dimensions for the arrays, should be clear from the context.

Initialization section

```
/* Simple Neural network to learn the Iris data set. This program classifies the 150
members of the Iris data set to about 97% accuracy. In other words, the number
misclassified should be around 3-5 once a suitable number of cycles has passed. */

/* First set the parameters for the Neural Network.
Those who are unfamiliar with Java may not have come across instances, so for the
sake of simplicity, this program uses pre-set numbers of layers and nodes. */

ninputs=4;  // number of input signals into the NN

noutputs=3;  // number of outputs from the NN

nl1=2;   // number of units in the 1st hidden layer

nl2=2;   // number of hidden units in the 2nd hidden layer

cycle=0;

/* Here is the Iris data. The first four entries are values of physical parameters; the
final three (binary) values define whether this is an example of Iris type 1, 2 or 3. */

double iris[][] = {
{5.1,  3.5,  1.4,  0.2, 1,  0,  0}, //0
{4.9,   3,  1.4,  0.2, 1,  0,  0},
{4.7,  3.2,  1.3,  0.2, 1,  0,  0},
{4.6,  3.1,  1.5,  0.2, 1,  0,  0},
{  5,  3.6,  1.4,  0.2, 1,  0,  0},
{5.4,  3.9,  1.7,  0.4, 1,  0,  0},
{4.6,  3.4,  1.4,  0.3, 1,  0,  0},
{  5,  3.4,  1.5,  0.2, 1,  0,  0},
{4.4,  2.9,  1.4,  0.2, 1,  0,  0},
{4.9,  3.1,  1.5,  0.1, 1,  0,  0},
{5.4,  3.7,  1.5,  0.2, 1,  0,  0}, //10
{4.8,  3.4,  1.6,  0.2, 1,  0,  0},
```

{4.8, 3, 1.4, 0.1, 1, 0, 0},
{4.3, 3, 1.1, 0.1, 1, 0, 0},
{5.8, 4, 1.2, 0.2, 1, 0, 0},
{5.7, 4.4, 1.5, 0.4, 1, 0, 0},
{5.4, 3.9, 1.3, 0.4, 1, 0, 0},
{5.1, 3.5, 1.4, 0.3, 1, 0, 0},
{5.7, 3.8, 1.7, 0.3, 1, 0, 0},
{5.1, 3.8, 1.5, 0.3, 1, 0, 0},
{5.4, 3.4, 1.7, 0.2, 1, 0, 0}, //20
{5.1, 3.7, 1.5, 0.4, 1, 0, 0},
{4.6, 3.6, 1, 0.2, 1, 0, 0},
{5.1, 3.3, 1.7, 0.5, 1, 0, 0},
{4.8, 3.4, 1.9, 0.2, 1, 0, 0},
{ 5, 3, 1.6, 0.2, 1, 0, 0},
{ 5, 3.4, 1.6, 0.4, 1, 0, 0},
{5.2, 3.5, 1.5, 0.2, 1, 0, 0},
{5.2, 3.4, 1.4, 0.2, 1, 0, 0},
{4.7, 3.2, 1.6, 0.2, 1, 0, 0},
{4.8, 3.1, 1.6, 0.2, 1, 0, 0}, //30
{5.4, 3.4, 1.5, 0.4, 1, 0, 0},
{5.2, 4.1, 1.5, 0.1, 1, 0, 0},
{5.5, 4.2, 1.4, 0.2, 1, 0, 0},
{4.9, 3.1, 1.5, 0.1, 1, 0, 0},
{ 5, 3.2, 1.2, 0.2, 1, 0, 0},
{5.5, 3.5, 1.3, 0.2, 1, 0, 0},
{4.9, 3.1, 1.5, 0.1, 1, 0, 0},
{4.4, 3, 1.3, 0.2, 1, 0, 0},
{5.1, 3.4, 1.5, 0.2, 1, 0, 0},
{ 5, 3.5, 1.3, 0.3, 1, 0, 0}, //40
{4.5, 2.3, 1.3, 0.3, 1, 0, 0},
{4.4, 3.2, 1.3, 0.2, 1, 0, 0},
{ 5, 3.5, 1.6, 0.6, 1, 0, 0},
{5.1, 3.8, 1.9, 0.4, 1, 0, 0},
{4.8, 3, 1.4, 0.3, 1, 0, 0},
{5.1, 3.8, 1.6, 0.2, 1, 0, 0},
{4.6, 3.2, 1.4, 0.2, 1, 0, 0},
{5.3, 3.7, 1.5, 0.2, 1, 0, 0},
{ 5, 3.3, 1.4, 0.2, 1, 0, 0},
{ 7, 3.2, 4.7, 1.4, 0, 1, 0}, //50
{6.4, 3.2, 4.5, 1.5, 0, 1, 0},
{6.9, 3.1, 4.9, 1.5, 0, 1, 0},
{5.5, 2.3, 4, 1.3, 0, 1, 0},
{6.5, 2.8, 4.6, 1.5, 0, 1, 0},
{5.7, 2.8, 4.5, 1.3, 0, 1, 0},
{6.3, 3.3, 4.7, 1.6, 0, 1, 0},
{4.9, 2.4, 3.3, 1, 0, 1, 0},
{6.6, 2.9, 4.6, 1.3, 0, 1, 0},
{5.2, 2.7, 3.9, 1.4, 0, 1, 0},
{ 5, 2, 3.5, 1, 0, 1, 0}, //60
{5.9, 3, 4.2, 1.5, 0, 1, 0},

```
{ 6, 2.2,  4,  1, 0, 1, 0},
{6.1, 2.9, 4.7, 1.4, 0, 1, 0},
{5.6, 2.9, 3.6, 1.3, 0, 1, 0},
{6.7, 3.1, 4.4, 1.4, 0, 1, 0},
{5.6,  3, 4.5, 1.5, 0, 1, 0},
{5.8, 2.7, 4.1,  1, 0, 1, 0},
{6.2, 2.2, 4.5, 1.5, 0, 1, 0},
{5.6, 2.5, 3.9, 1.1, 0, 1, 0},
{5.9, 3.2, 4.8, 1.8, 0, 1, 0}, //70
{6.1, 2.8,  4, 1.3, 0, 1, 0},
{6.3, 2.5, 4.9, 1.5, 0, 1, 0},
{6.1, 2.8, 4.7, 1.2, 0, 1, 0},
{6.4, 2.9, 4.3, 1.3, 0, 1, 0},
{6.6,  3, 4.4, 1.4, 0, 1, 0},
{6.8, 2.8, 4.8, 1.4, 0, 1, 0},
{6.7,  3,  5, 1.7, 0, 1, 0},
{ 6, 2.9, 4.5, 1.5, 0, 1, 0},
{5.7, 2.6, 3.5,  1, 0, 1, 0},
{5.5, 2.4, 3.8, 1.1, 0, 1, 0}, //80
{5.5, 2.4, 3.7,  1, 0, 1, 0},
{5.8, 2.7, 3.9, 1.2, 0, 1, 0},
{ 6, 2.7, 5.1, 1.6, 0, 1, 0},
{5.4,  3, 4.5, 1.5, 0, 1, 0},
{ 6, 3.4, 4.5, 1.6, 0, 1, 0},
{6.7, 3.1, 4.7, 1.5, 0, 1, 0},
{6.3, 2.3, 4.4, 1.3, 0, 1, 0},
{5.6,  3, 4.1, 1.3, 0, 1, 0},
{5.5, 2.5,  4, 1.3, 0, 1, 0},
{5.5, 2.6, 4.4, 1.2, 0, 1, 0}, //90
{6.1,  3, 4.6, 1.4, 0, 1, 0},
{5.8, 2.6,  4, 1.2, 0, 1, 0},
{ 5, 2.3, 3.3,  1, 0, 1, 0},
{5.6, 2.7, 4.2, 1.3, 0, 1, 0},
{5.7,  3, 4.2, 1.2, 0, 1, 0},
{5.7, 2.9, 4.2, 1.3, 0, 1, 0},
{6.2, 2.9, 4.3, 1.3, 0, 1, 0},
{5.1, 2.5,  3, 1.1, 0, 1, 0},
{5.7, 2.8, 4.1, 1.3, 0, 1, 0},
{6.3, 3.3,  6, 2.5, 0, 0, 1}, //100
{5.8, 2.7, 5.1, 1.9, 0, 0, 1},
{7.1,  3, 5.9, 2.1, 0, 0, 1},
{6.3, 2.9, 5.6, 1.8, 0, 0, 1},
{6.5,  3, 5.8, 2.2, 0, 0, 1},
{7.6,  3, 6.6, 2.1, 0, 0, 1},
{4.9, 2.5, 4.5, 1.7, 0, 0, 1},
{7.3, 2.9, 6.3, 1.8, 0, 0, 1},
{6.7, 2.5, 5.8, 1.8, 0, 0, 1},
{7.2, 3.6, 6.1, 2.5, 0, 0, 1},
{6.5, 3.2, 5.1,  2, 0, 0, 1}, //110
{6.4, 2.7, 5.3, 1.9, 0, 0, 1},
```

```
{6.8,  3,  5.5,  2.1, 0,  0,  1},
{5.7,  2.5,  5,  2, 0,  0,  1},
{5.8,  2.8,  5.1,  2.4, 0,  0,  1},
{6.4,  3.2,  5.3,  2.3, 0,  0,  1},
{6.5,  3,  5.5,  1.8, 0,  0,  1},
{7.7,  3.8,  6.7,  2.2, 0,  0,  1},
{7.7,  2.6,  6.9,  2.3, 0,  0,  1},
{ 6,  2.2,  5,  1.5, 0,  0,  1},
{6.9,  3.2,  5.7,  2.3, 0,  0,  1},  //120
{5.6,  2.8,  4.9,  2, 0,  0,  1},
{7.7,  2.8,  6.7,  2, 0,  0,  1},
{6.3,  2.7,  4.9,  1.8, 0,  0,  1},
{6.7,  3.3,  5.7,  2.1, 0,  0,  1},
{7.2,  3.2,  6,  1.8, 0,  0,  1},
{6.2,  2.8,  4.8,  1.8, 0,  0,  1},
{6.1,  3,  4.9,  1.8, 0,  0,  1},
{6.4,  2.8,  5.6,  2.1, 0,  0,  1},
{7.2,  3,  5.8,  1.6, 0,  0,  1},
{7.4,  2.8,  6.1,  1.9, 0,  0,  1},  //130
{7.9,  3.8,  6.4,  2, 0,  0,  1},
{6.4,  2.8,  5.6,  2.2, 0,  0,  1},
{6.3,  2.8,  5.1,  1.5, 0,  0,  1},
{6.1,  2.6,  5.6,  1.4, 0,  0,  1},
{7.7,  3,  6.1,  2.3, 0,  0,  1},
{6.3,  3.4,  5.6,  2.4, 0,  0,  1},
{6.4,  3.1,  5.5,  1.8, 0,  0,  1},
{ 6,  3,  4.8,  1.8, 0,  0,  1},
{6.9,  3.1,  5.4,  2.1, 0,  0,  1},
{6.7,  3.1,  5.6,  2.4, 0,  0,  1},  //140
{6.9,  3.1,  5.1,  2.3, 0,  0,  1},
{5.8,  2.7,  5.1,  1.9, 0,  0,  1},
{6.8,  3.2,  5.9,  2.3, 0,  0,  1},
{6.7,  3.3,  5.7,  2.5, 0,  0,  1},
{6.7,  3,  5.2,  2.3, 0,  0,  1},
{6.3,  2.5,  5,  1.9, 0,  0,  1},
{6.5,  3,  5.2,  2, 0,  0,  1},
{6.2,  3.4,  5.4,  2.3, 0,  0,  1},
{5.9,  3,  5.1,  1.8, 0,  0,  1}};

nsamples=150;  // number of samples in total

/* It is common practice to scale sample data fed to help the ANN converge. We will
scale the four inputs to a range of 0.2 - 0.8, and the three outputs to values of either 0
or 0.8 */

for (int i=0; i<ninputs; i++)
{
   double biggest, smallest;
   biggest=iris[0][i];
   smallest=biggest;
```

```
    for (int is=0; is<nsamples; is++)
    {
       if (biggest<iris[is][i]) biggest=iris[is][i];
       if (smallest>iris[is][i]) smallest=iris[is][i];
    }
    for (int is=0; is<nsamples; is++)
    {
       iris[is][i]=(iris[is][i]-smallest)*(0.6/(biggest-smallest))+0.2;
    }
}

for (int i=4; i<=6; i++)
{
    for (int is=0; is<nsamples; is++)
    {
       iris[is][i]=iris[is][i]*0.8;
    }
}

/* Copy the Iris data into the patterns array */

for (int is=0; is<nsamples; is++)
{
    for (int nd=0; nd<ninputs+noutputs; nd++)
    {
       patterns[is][nd]=iris[is][nd];
    }
}

/* The patterns are divided up into training, validation and test sets in the ratio
training_fraction:validation_fraction:test_fraction. In this program all entries are used
for training, so there is no validation or test set. */

training_fraction=1.00;
validation_fraction=0.00;
test_fraction=1.0-training_fraction-validation_fraction;
ntraining=(int)(nsamples*training_fraction);
nvalidation=(int)(nsamples*validation_fraction);
ntest=nsamples-ntraining-nvalidation;

/* First set all the "sample_chosen" entries to false. Once an example has been chosen
for the training set, this value will be set to true. */

for (int is=0; is<nsamples; is++)
{
    sample_chosen[is]=false;
}

/* Select some (in this instance, all) samples for the training set */
```

```
int nt=0;
do
{
   for (int is=0; is<nsamples; is++)
   {
      if (!sample_chosen[is] && nt<ntraining)
      {
         if(Math.random()<training_fraction)
         {
            sample_chosen[is]=true;
            training_ids[nt]=is;
            nt++;
         }
      }
   }
}
while(nt<ntraining);

/* If there is no validation set, all remaining samples are members of the test set. */

if (validation_fraction == 0.0)
{
   int tt=0;
   for (int is=0; is<nsamples; is++)
   {
      if (!sample_chosen[is])
      {
         test_ids[tt]=is;
         tt++;
      }
   }
}

/* Initialize connection weights to random values. For simplicity we randomize every
entry in the connections arrays so that if the size of the network is changed, the
weights will already be initialized. */

restartwithnewparams=true;
alpha=1.0;
eta=0.085;
bestmisclassified=150;  // Smallest number of samples misclassified at this stage
```

Evolution section

```
/* This is the start of the main neural network training loop.  Network configuration is
4 input nodes, two hidden nodes in layer 1, two hidden nodes in layer 2 and three
output nodes.
```

There are various ways in which an ANN might classify an Iris. One is to use a single output node which returns the value 0, 1, or 2 to indicate which of the three types of Iris the network believes it has identified. However, this biases the network towards recognizing Iris type 2 (whichever that is), since, if the output lies in the range 0-2, an output of 0-0.5 might be taken to indicate type 0, an output of 0.5-1.5 might be taken to indicate type 1 and an output of 1.5-2 to indicate type 2. The range of output corresponding to type 1 is thus twice as large as the other two ranges, so distorting the behaviour of the network.

There are several ways to tackle this, the simplest being to have three output nodes. The target output is then "1" from the node corresponding to the correct output and "0" from the remaining two nodes. This is the approach adopted here.

After a modest amount of training, this network typically classifies all except three or four samples correctly, though at times during training it may classify all but one or two correctly.

With alpha=1.0, and eta around 0.1, the network converges on reasonable network weights within a couple of hundred cycles.

No attempt has been made to optimise eta and there's no inclusion of momentum terms. Smaller values of eta lead to slower convergence, but still to good solutions. */

/* Initialize connection weights to random values. For simplicity we randomize every entry in the connections arrays so that if the size of the network is changed, the weights will already be initialized. This section of code, which was included above, appears here again so it can be executed if the parameters eta and alpha are changed. */

```
if (restartwithnewparams)
{
   for (int i=0; i<20; i++)
   {
      for (int j=0; j<20; j++)
      {
         input_to_h1_weight[i][j]=Math.random();
         h1_to_h2_weight[i][j]=Math.random();
         h2_to_output_weight[i][j]=Math.random();
      }
   }
   bestmisclassified=150;
   restartwithnewparams=false;
}

cycle++;  // Increment the epoch number

misclassified=0;  // Set the number of samples misclassified to zero
```

/* It is important to select members of the training set in a random order each cycle. To do this, we first create a list of random numbers. */

```
for (int is=0; is<ntraining; is++)
{
   randomnos[is]=Math.random();
}

// Now sweep through the random numbers, arranging them in descending order

double biggest;
int biggest_location;
for (int is=0; is<ntraining; is++)
{
   biggest=randomnos[0];
   biggest_location=0;
   for (int js=1; js<ntraining; js++)
   {
      if(randomnos[js]>biggest)
      {
         biggest=randomnos[js];
         biggest_location=js;
      }
   }
order[is]=biggest_location;
randomnos[biggest_location]=-1.0;
}

/* The array order[] now contains all entries in the training set from 0 to ntraining in a
random order. */

for (int is=0; is<ntraining; is++)  // This is the start of the nn training loop
{
   int next_sample=order[is];  // Select the next data sample

/* Feed the sample into the input nodes. The first node in each layer (other than the
output layer) is the bias node, so fill that first. */

   input_node_output[0]=1.0; // Wasteful to do this every cycle, but there's little
computational cost
   for (int iinput=1; iinput<=ninputs; iinput++)
   {
      input_node_output[iinput]=patterns[next_sample][iinput-1];
   }

/* Zero the total input at each node in hidden layer 1. As with the input layer, node[0]
in this hidden layer is the bias node, so receives no input. */

   for (int ih1=1; ih1<=nl1; ih1++)
   {
      h1_node_input[ih1]=0.0;
   }
```

```
// Calculate the input to nodes in hidden layer 1

   for (int ii=0; ii<=ninputs; ii++)
   {
      for (int ih1=1; ih1<=nl1; ih1++)
      {

h1_node_input[ih1]=h1_node_input[ih1]+input_node_output[ii]*input_to_h1_weight
[ii][ih1];
      }
   }

// Calculate outputs from hidden layer 1 using the logistic function

   for (int ih1=1; ih1<=nl1; ih1++)
   {
      h1_node_output[ih1]=1.0/(1.0+Math.exp(-h1_node_input[ih1]));
   }

// Calculate inputs to hidden layer 2. First zero the inputs

   for (int ih2=1; ih2<=nl2; ih2++)
   {
      h2_node_input[ih2]=0.0;
   }

   h1_node_output[0]=1.0;  // Set the input to layer 2 from the bias node

   for (int ih2=1; ih2<=nl2; ih2++)
   {
      for (int ih1=0; ih1<=nl1; ih1++)
      {

h2_node_input[ih2]=h2_node_input[ih2]+h1_node_output[ih1]*h1_to_h2_weight[ih1
][ih2];
      }
   }

// Calculate outputs from hidden layer 2

   for (int ih2=1; ih2<=nl2; ih2++)
   {
      h2_node_output[ih2]=1.0/(1.0+Math.exp(-h2_node_input[ih2]));
   }

// Calculate inputs to output layer. First set the signals at the outputs to zero

   for (int io=0; io<noutputs; io++)
   {
```

```
      output_node_input[io]=0.0;
   }

   h2_node_output[0]=1.0;  // Set the bias node feeding into the output layer

   for (int ih2=0; ih2<=nl2; ih2++)
   {
      for (int io=0; io<noutputs; io++)
      {

output_node_input[io]=output_node_input[io]+h2_node_output[ih2]*h2_to_output_
weight[ih2][io];
      }
   }

// Calculate outputs from output layer

   for (int io=0; io<noutputs; io++)
   {
      output_node_output[io]=1.0/(1.0+Math.exp(-output_node_input[io]));
   }

/* Determine whether the sample is correctly classified; this is particularly clunky
code, but easy to understand. */

   int irisclass=1;
   if (patterns[next_sample][5]>0.79) irisclass=2;
   if (patterns[next_sample][6]>0.79) irisclass=3;

   if (irisclass==1 && (output_node_output[0]<output_node_output[1] ||
output_node_output[0]<output_node_output[2]))misclassified=misclassified+1;
   if (irisclass==2 && (output_node_output[1]<output_node_output[0] ||
output_node_output[1]<output_node_output[2]))misclassified=misclassified+1;
   if (irisclass==3 && (output_node_output[2]<output_node_output[0] ||
output_node_output[2]<output_node_output[1]))misclassified=misclassified+1;

// Start Backpropagation. First calculate the error at each output node

   for (int io=0; io<noutputs; io++)
   {
      output_error[io]=(patterns[next_sample][ninputs+io]-
output_node_output[io])*output_node_output[io]*(1.0-output_node_output[io]);
   }

// Now find the error at the hidden nodes. First zero the error for layer 2 nodes

   for (int ih2=1; ih2<=nl2; ih2++)
   {
      h2_error[ih2]=0.0;
   }
```

```
// Calculate the error at each hidden node in layer 2

   for (int ih2=1; ih2<=nl2; ih2++)
   {
      for (int io=0; io<noutputs; io++)
      {
         h2_error[ih2]=h2_error[ih2]+h2_node_output[ih2]*(1.0-
h2_node_output[ih2])*h2_to_output_weight[ih2][io]*output_error[io];
      }
   }

// Now do the same for nodes in the first hidden layer

   for (int ih1=1; ih1<=nl1; ih1++)
   {
      h1_error[ih1]=0.0;
   }

   for (int ih1=1; ih1<=nl1; ih1++)
   {
      for (int ih2=1; ih2<=nl2; ih2++)
      {
         h1_error[ih1]=h1_error[ih1]+h1_node_output[ih1]*(1.0-
h1_node_output[ih1])*h2_error[ih2]*h1_to_h2_weight[ih1][ih2];
      }
   }

/* We have the errors for all layers, so we can do the BP.  First we do the weights
update for weights into the output layer. */

   for (int io=0; io<noutputs; io++)
   {
      for (int ih2=0; ih2<=nl2; ih2++)
      {

h2_to_output_weight[ih2][io]=h2_to_output_weight[ih2][io]*alpha+eta*output_error
[io]*h2_node_output[ih2];
      }
   }

// Update the weights from hidden layer 1 to hidden layer 2

   for (int ih1=0; ih1<=nl1; ih1++)
   {
      for (int ih2=1; ih2<=nl2; ih2++)
      {

h1_to_h2_weight[ih1][ih2]=h1_to_h2_weight[ih1][ih2]*alpha+eta*h2_error[ih2]*h1
_node_output[ih1];
```

```
      }
   }

// Finally update the weights to hidden layer 1 from the input layer

   for (int ii=0; ii<=ninputs; ii++)
   {
      for (int ih1=1; ih1<=nl1; ih1++)
      {

input_to_h1_weight[ii][ih1]=input_to_h1_weight[ii][ih1]+eta*h1_error[ih1]*input_n
ode_output[ii];
      }
   }
}  // This is the end of the training loop, i.e. one complete epoch

if (bestmisclassified>misclassified) bestmisclassified=misclassified;
if (cycle>5000) System.exit(0); // Here we just run for a predefined number of cycles
```