a i - j u n k i e

The CNeuralNet class

Let's get started on the neural network class, `CNeuralNet`. We want this class to be flexible so it can be used in other projects and as simple to use as possible. We need to be able to set up a neural network with any amount of inputs and outputs and any amount of neurons in any amount of hidden layers. So how do we do this? Well, first we need to define structures for a neuron and a neuron layer. Let's have a look at the definition of these structures… first the neuron:

```
struct SNeuron

{

   //the number of inputs into the neuron

   int m_NumInputs;



   //the weights for each input

   vector<double> m_vecWeight;



   //ctor

   SNeuron(int NumInputs);

};
```

This is very simple, we just need to keep a record of how many inputs there are into each neuron and a std::vector of doubles in which we will store all the weights. Remember, there's a weight for every input into the neuron. When a `SNeuron` object is created, all the weights are initialized with random values.

Programming note

std::vector is part of the STL and is a ready made class for handling dynamic arrays. A vector is created as seen above. Elements are added to it by using the method push_back(). i.e.

```
#include<vector>

std::vector<int> MyFirstVector;

for (int i=0; i< 10; i++)
{
        MyFirstVector.push_back(i);
        Cout << endl << MyFirstVector[i];
}
```

To empty a vector we just use the method: MyFirstVector.clear();

We can get the number of elements in a vector by: MyFirstVector.size()

That's it! No need to worry about memory management, std::vector does it all for you. I shall be using them throughout the program where appropriate

This is the constructor for `SNeuron`:

```
SNeuron::SNeuron(int NumInputs): m_NumInputs(NumInputs+1)

{

  //we need an additional weight for the bias hence the +1

  for (int i=0; i<NumInputs+1; ++i)

  {

    //set up the weights with an initial random value

    m_vecWeight.push_back(RandomClamped());

  }

}
```

This takes the number of inputs going into the neuron as an argument and creates a vector of random weights. One weight for each input.

What's that I hear you say? There's an extra weight there! Well I'm glad you spotted that because that extra weight is quite important but to explain why it's there I'm going to have to do some more maths. Remember that our activation was the sum of all the inputs x weights and that the output of the neuron was dependent upon whether or not this activation exceeded a *threshold* value (*t*)? And that this could be represented in equation form by

$$x_1w_1 + x_2w_2 + x_3w_3... + x_nw_n >= t$$

Because the network weights are going to be evolved it would be great if the threshold value could be evolved too. To make this easy I'm going to use a little trick to make it appear as a weight. All you have to do is subtract *t* from either side of the equation and we get:

$$x_1w_1 + x_2w_2 + x_3w_3... + x_nw_n - t >= 0$$

or we can write this another way:

$$x_1w_1 + x_2w_2 + x_3w_3... + x_nw_n + (-1)t >= 0$$

So you can see (hopefully) that we can treat the threshold as a weight that is always multiplied by an input of -1. This is usually referred to as the *bias*.

And that's why each neuron is initialized with one additional weight. Because now when the network is evolved we don't have to worry about the threshold value as it's built in with the weights and will take care of itself. Good eh?

Lets get on with the rest of the neural net code… The next structure defines a *layer* of neurons.

```
struct SNeuronLayer

{

  //the number of neurons in this layer

  int m_NumNeurons;



  //the layer of neurons

  vector<SNeuron> m_vecNeurons;



  SNeuronLayer(int NumNeurons, int NumInputsPerNeuron);

};
```

As you can see this just groups together a bunch of neurons into a layer. The `CNeuralNet` class is much more exciting, so let's move on and take a look at its definition:

```
class CNeuralNet

{

private:

  int m_NumInputs;


  int m_NumOutputs;


  int m_NumHiddenLayers;


  int m_NeuronsPerHiddenLyr;


  //storage for each layer of neurons including the output layer

  vector<SNeuronLayer> m_vecLayers;


public:


  CNeuralNet();


  //have a guess... ;0)

  void CreateNet();


  //gets the weights from the NN

  vector<double> GetWeights()const;


  //returns the total number of weights in the net
```

```
  int GetNumberOfWeights()const;


  //replaces the weights with new ones
  void PutWeights(vector<double> &weights);


  //calculates the outputs from a set of inputs
  vector<double> Update(vector<double> &inputs);


  //sigmoid response curve
  inline double Sigmoid(double activation, double response);
};
```

Most of this should be self explanatory. The main work is done by the method `Update`. Here we pass in our inputs to the neural network as a std::vector of doubles and retrieve the output as another std::vector of doubles. This is really the only method we use after the `CNeuralNetwork` class has been initialized. We can just treat it as a black box, feeding it data and retrieving the output as if by magic. Let's take a closer look at this method:

```
vector<double> CNeuralNet::Update(vector<double> &inputs)
{
  //stores the resultant outputs from each layer
  vector<double> outputs;


  int cWeight = 0;


  //first check that we have the correct amount of inputs
  if (inputs.size() != m_NumInputs)
  {
    //just return an empty vector if incorrect.
    return outputs;
  }
```

```
//For each layer....
for (int i=0; i<m_NumHiddenLayers + 1; ++i)
{
  if ( i > 0 )
  {
    inputs = outputs;
  }

  outputs.clear();

  cWeight = 0;

  //for each neuron sum the (inputs * corresponding weights).Throw
  //the total at our sigmoid function to get the output.
  for (int j=0; j<m_vecLayers[i].m_NumNeurons; ++j)
  {
    double netinput = 0;

    int NumInputs = m_vecLayers[i].m_vecNeurons[j].m_NumInputs;

    //for each weight
    for (int k=0; k<NumInputs - 1; ++k)
    {
      //sum the weights x inputs
      netinput += m_vecLayers[i].m_vecNeurons[j].m_vecWeight[k] *
                  inputs[cWeight++];
    }
```

```
      //add in the bias
      netinput += m_vecLayers[i].m_vecNeurons[j].m_vecWeight[NumInputs-1]
*
                  CParams::dBias;

      //we can store the outputs from each layer as we generate them.
      //The combined activation is first filtered through the sigmoid
      //function
      outputs.push_back(Sigmoid(netinput, CParams::dActivationResponse));

      cWeight = 0;
    }
  }
  return outputs;
}
```

After this method has checked the validity of the input vector it enters a loop which examines each layer in turn. For each layer, it steps through the neurons in that layer and sums all the inputs multiplied by the corresponding weights. The last weight added in for each neuron is the bias (remember the bias is simply a weight always tied to the value -1.0). This value is then put through the sigmoid function to give that neurons output and then added to a vector which is fed back into the next iteration of the loop and so on until we have our output proper.

The other methods in `CNeuralNet` are used mainly by the genetic algorithm class to grab the weights from a network or to replace the weights of a network.
