John Bullinaria's Step by Step Guide to Implementing a Neural Network in C

By John A. Bullinaria from the School of Computer Science of The University of Birmingham, UK.

This document contains a step by step guide to implementing a simple neural network in C. It is aimed mainly at students who wish to (or have been told to) incorporate a neural network learning component into a larger system they are building. Obviously there are many types of neural network one could consider using - here I shall concentrate on one particularly common and useful type, namely a simple three-layer feed-forward back-propagation network (multi layer perceptron).

This type of network will be useful when we have a set of input vectors and a corresponding set of output vectors, and our system must produce an appropriate output for each input it is given. Of course, if we already have a complete noise-free set of input and output vectors, then a simple look-up table would suffice. However, if we want the system to *generalize*, i.e. produce appropriate outputs for inputs it has never seen before, then a neural network that has *learned* how to map between the known inputs and outputs (i.e. our training set) will often do a pretty good job for new inputs as well.

I shall assume that the reader is already familiar with C, and, for more details about neural networks in general, simply refer the reader to the newsgroup *comp.ai.neural-nets* and the associated Neural Networks FAQ. So, let us begin...

A single neuron (i.e. processing unit) takes it total input *In* and produces an output activation *Out*. I shall take this to be the sigmoid function

```
Out = 1.0/(1.0 + exp(-In));        /* Out = Sigmoid(In) */
```

though other activation functions are often used (e.g. linear or hyperbolic tangent). This has the effect of squashing the infinite range of *In* into the range 0 to 1. It also has the convenient property that its derivative takes the particularly simple form

```
Sigmoid_Derivative = Sigmoid * (1.0 - Sigmoid) ;
```

Typically, the input *In* into a given neuron will be the weighted sum of output activations feeding in from a number of other neurons. It is convenient to think of the activations flowing through layers of neurons. So, if there are *NumUnits1* neurons in layer 1, the total activation flowing into our layer 2 neuron is just the sum over *Layer1Out[i]*Weight[i]*, where *Weight[i]* is the strength/weight of the connection between unit *i* in layer 1 and our unit in layer 2. Each neuron will also have a bias, or resting state, that is added to the sum of inputs, and it is convenient to call this *weight[0]*. We can then write

```
Layer2In = Weight[0] ;        /* start with the bias */
for( i = 1 ; i <= NumUnits1 ; i++ ) {        /* i loop over layer 1 units */
        Layer2In += Layer1Out[i] * Weight[i] ;        /* add in weighted contributions from layer
        1 */
}
```

```
Layer2Out = 1.0/(1.0 + exp(-Layer2In)) ;    /* compute sigmoid to give activation */
```

Normally layer 2 will have many units as well, so it is appropriate to write the weights between unit *i* in layer 1 and unit *j* in layer 2 as an array *Weight[i][j]*. Thus to get the output of unit *j* in layer 2 we have

```
Layer2In[j] = Weight[0][j] ;
for( i = 1 ; i <= NumUnits1 ; i++ ) {
        Layer2In[j] += Layer1Out[i] * Weight[i][j] ;
}
Layer2Out[j] = 1.0/(1.0 + exp(-Layer2In[j])) ;
```

Remember that in C the array indices start from zero, not one, so we would declare our variables as

```
double Layer1Out[NumUnits1+1] ;
double Layer2In[NumUnits2+1] ;
double Layer2Out[NumUnits2+1] ;
double Weight[NumUnits1+1][NumUnits2+1] ;
```

(or, more likely, declare pointers and use *calloc* or *malloc* to allocate the memory). Naturally, we need another loop to get all the layer 2 outputs

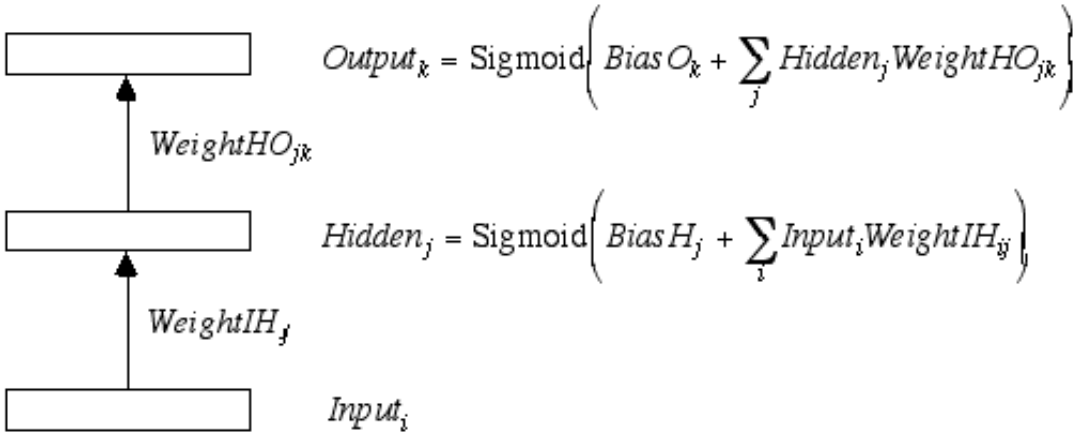
```
for( j = 1 ; j <= NumUnits2 ; j++ ) {
        Layer2In[j] = Weight[0][j] ;
        for( i = 1 ; i <= NumUnits1 ; i++ ) {
                Layer2In[j] += Layer1Out[i] * Weight[i][j] ;
        }
        Layer2Out[j] = 1.0/(1.0 + exp(-Layer2In[j])) ;
}
```

Three layer networks are necessary and sufficient for most purposes, so our layer 2 outputs feed into a third layer in the same way as above

```
for( j = 1 ; j <= NumUnits2 ; j++ ) {       /* j loop computes layer 2 activations */
        Layer2In[j] = Weight12[0][j] ;
        for( i = 1 ; i <= NumUnits1 ; i++ ) {
                Layer2In[j] += Layer1Out[i] * Weight12[i][j] ;
        }
        Layer2Out[j] = 1.0/(1.0 + exp(-Layer2In[j])) ;
}
for( k = 1 ; k <= NumUnits3 ; k++ ) {       /* k loop computes layer 3 activations */
        Layer3In[k] = Weight23[0][k] ;
        for( j = 1 ; j <= NumUnits2 ; j++ ) {
                Layer3In[k] += Layer2Out[j] * Weight23[j][k] ;
        }
        Layer3Out[k] = 1.0/(1.0 + exp(-Layer3In[k])) ;
}
```

The code can start to become confusing at this point - I find that keeping a separate index *i, j, k* for each layer helps, as does an intuitive notation for distinguishing between the different layers

of weights *Weight12* and *Weight23*. For obvious reasons, for three layer networks, it is traditional to call layer 1 the *Input* layer, layer 2 the *Hidden* layer, and layer 3 the *Output* layer. Our network thus takes on the familiar form that we shall use for the rest of this document



Also, to save getting all the *In*'s and *Out*'s confused, we can write *LayerNIn* as *SumN*. Our code can thus be written

```
for( j = 1 ; j <= NumHidden ; j++ ) {       /* j loop computes hidden unit activations */
        SumH[j] = WeightIH[0][j] ;
        for( i = 1 ; i <= NumInput ; i++ ) {
                SumH[j] += Input[i] * WeightIH[i][j] ;
        }
        Hidden[j] = 1.0/(1.0 + exp(-SumH[j])) ;
}
for( k = 1 ; k <= NumOutput ; k++ ) {       /* k loop computes output unit activations */
        SumO[k] = WeightHO[0][k] ;
        for( j = 1 ; j <= NumHidden ; j++ ) {
                SumO[k] += Hidden[j] * WeightHO[j][k] ;
        }
        Output[k] = 1.0/(1.0 + exp(-SumO[k])) ;
}
```

Generally we will have a whole set of *NumPattern* training patterns, i.e. pairs of input and target output vectors,

```
Input[p][i] , Target[p][k]
```

labelled by the index *p*. The network learns by minimizing some measure of the error of the network's actual outputs compared with the target outputs. For example, the sum squared error over all output units *k* and all training patterns *p* will be given by

```
Error = 0.0 ;
for( p = 1 ; p <= NumPattern ; p++ ) {
        for( k = 1 ; k <= NumOutput ; k++ ) {
                Error += 0.5 * (Target[p][k] - Output[p][k]) * (Target[p][k] - Output[p][k]) ;
        }
}
```

(The factor of 0.5 is conventionally included to simplify the algebra in deriving the learning

algorithm.) If we insert the above code for computing the network outputs into the *p* loop of this, we end up with

```
Error = 0.0 ;
for( p = 1 ; p <= NumPattern ; p++ ) {        /* p loop over training patterns */
        for( j = 1 ; j <= NumHidden ; j++ ) {        /* j loop over hidden units */
                SumH[p][j] = WeightIH[0][j] ;
                for( i = 1 ; i <= NumInput ; i++ ) {
                        SumH[p][j] += Input[p][i] * WeightIH[i][j] ;
                }
                Hidden[p][j] = 1.0/(1.0 + exp(-SumH[p][j])) ;
        }
        for( k = 1 ; k <= NumOutput ; k++ ) {        /* k loop over output units */
                SumO[p][k] = WeightHO[0][k] ;
                for( j = 1 ; j <= NumHidden ; j++ ) {
                        SumO[p][k] += Hidden[p][j] * WeightHO[j][k] ;
                }
                Output[p][k] = 1.0/(1.0 + exp(-SumO[p][k])) ;
                Error += 0.5 * (Target[p][k] - Output[p][k]) * (Target[p][k] - Output[p][k]) ;     /* Sum
                Squared Error */
        }
}
```

I'll leave the reader to dispense with any indices that they don't need for the purposes of their own system (e.g. the indices on *SumH* and *SumO*).

The next stage is to iteratively adjust the weights to minimize the network's error. A standard way to do this is by 'gradient descent' on the error function. We can compute how much the error is changed by a small change in each weight (i.e. compute the partial derivatives d*Error*/d*Weight*) and shift the weights by a small amount in the direction that reduces the error. The literature is full of variations on this general approach - I shall begin with the 'standard on-line back-propagation with momentum' algorithm. This is not the place to go through all the mathematics, but for the above sum squared error we can compute and apply one iteration (or '*epoch*') of the required weight changes *DeltaWeightIH* and *DeltaWeightHO* using

```
Error = 0.0 ;
for( p = 1 ; p <= NumPattern ; p++ ) {        /* repeat for all the training patterns */
        for( j = 1 ; j <= NumHidden ; j++ ) {        /* compute hidden unit activations */
                SumH[p][j] = WeightIH[0][j] ;
                for( i = 1 ; i <= NumInput ; i++ ) {
                        SumH[p][j] += Input[p][i] * WeightIH[i][j] ;
                }
                Hidden[p][j] = 1.0/(1.0 + exp(-SumH[p][j])) ;
        }
        for( k = 1 ; k <= NumOutput ; k++ ) {        /* compute output unit activations and errors
        */
                SumO[p][k] = WeightHO[0][k] ;
                for( j = 1 ; j <= NumHidden ; j++ ) {
                        SumO[p][k] += Hidden[p][j] * WeightHO[j][k] ;
                }
                Output[p][k] = 1.0/(1.0 + exp(-SumO[p][k])) ;
```

```
                Error += 0.5 * (Target[p][k] - Output[p][k]) * (Target[p][k] - Output[p][k]) ;
                DeltaO[k] = (Target[p][k] - Output[p][k]) * Output[p][k] * (1.0 - Output[p][k]) ;
        }
        for( j = 1 ; j <= NumHidden ; j++ ) {       /* 'back-propagate' errors to hidden layer */
                SumDOW[j] = 0.0 ;
                for( k = 1 ; k <= NumOutput ; k++ ) {
                        SumDOW[j] += WeightHO[j][k] * DeltaO[k] ;
                }
                DeltaH[j] = SumDOW[j] * Hidden[p][j] * (1.0 - Hidden[p][j]) ;
        }
        for( j = 1 ; j <= NumHidden ; j++ ) {       /* update weights WeightIH */
                DeltaWeightIH[0][j] = eta * DeltaH[j] + alpha * DeltaWeightIH[0][j] ;
                WeightIH[0][j] += DeltaWeightIH[0][j] ;
                for( i = 1 ; i <= NumInput ; i++ ) {
                        DeltaWeightIH[i][j] = eta * Input[p][i] * DeltaH[j] + alpha * DeltaWeightIH[i]
                        [j];
                        WeightIH[i][j] += DeltaWeightIH[i][j] ;
                }
        }
        for( k = 1 ; k <= NumOutput ; k ++ ) {       /* update weights WeightHO */
                DeltaWeightHO[0][k] = eta * DeltaO[k] + alpha * DeltaWeightHO[0][k] ;
                WeightHO[0][k] += DeltaWeightHO[0][k] ;
                for( j = 1 ; j <= NumHidden ; j++ ) {
                        DeltaWeightHO[j][k] = eta * Hidden[p][j] * DeltaO[k] + alpha *
                        DeltaWeightHO[j][k] ;
                        WeightHO[j][k] += DeltaWeightHO[j][k] ;
                }
        }
}
```

(There is clearly plenty of scope for re-ordering, combining and simplifying the loops here - I will leave that for the reader to do once they have understood what the separate code sections are doing.) The weight changes *DeltaWeightIH* and *DeltaWeightHO* are each made up of two components. First, the *eta* component that is the gradient descent contribution. Second, the *alpha* component that is a 'momentum' term which effectively keeps a moving average of the gradient descent weight change contributions, and thus smoothes out the overall weight changes. Fixing good values of the learning parameters *eta* and *alpha* is usually a matter of trial and error. Certainly *alpha* must be in the range 0 to 1, and a non-zero value does usually speed up learning. Finding a good value for *eta* will depend on the problem, and also on the value chosen for *alpha*. If it is set too low, the training will be unnecessarily slow. Having it too large will cause the weight changes to oscillate wildly, and can slow down or even prevent learning altogether. (I generally start by trying *eta* = 0.1 and explore the effects of repeatedly doubling or halving it.)

The complete training process will consist of repeating the above weight updates for a number of epochs (using another *for* loop) until some error crierion is met, for example the *Error* falls below some chosen small number. (Note that, with sigmoids on the outputs, the *Error* can only reach exactly zero if the weights reach infinity! Note also that sometimes the training can get stuck in a 'local minimum' of the error function and never get anywhere the actual minimum.) So, we need to wrap the last block of code in something like

```
for( epoch = 1 ; epoch < LARGENUMBER ; epoch++ ) {
        /* ABOVE CODE FOR ONE ITERATION */
        if( Error < SMALLNUMBER ) break ;
}
```

If the training patterns are presented in the same systematic order during each epoch, it is possible for weight oscillations to occur. It is therefore generally a good idea to use a new random order for the training patterns for each epoch. If we put the *NumPattern* training pattern indices *p* in random order into an array *ranpat[]*, then it is simply a matter of replacing our training pattern loop

```
for( p = 1 ; p <= NumPattern ; p++ ) {
```

with

```
for( np = 1 ; np <= NumPattern ; np++ ) {
        p = ranpat[np] ;
```

Generating the random array *ranpat[]* is not quite so simple, but the following code will do the job

```
for( p = 1 ; p <= NumPattern ; p++ ) {        /* set up ordered array */
        ranpat[p] = p ;
}
for( p = 1 ; p <= NumPattern ; p++) {        /* swap random elements into each position */
        np = p + rando() * ( NumPattern + 1 - p ) ;
        op = ranpat[p] ; ranpat[p] = ranpat[np] ; ranpat[np] = op ;
}
```

Naturally, one must set some initial network weights to start the learning process. Starting all the weights at zero is generally not a good idea, as that is often a local minimum of the error function. It is normal to initialize all the weights with small random values. If *rando()* is your favourite random number generator function that returns a flat distribution of random numbers in the range 0 to 1, and *smallwt* is the maximum absolute size of your initial weights, then an appropriate section of weight initialization code would be

```
for( j = 1 ; j <= NumHidden ; j++ ) {        /* initialize WeightIH and DeltaWeightIH */
        for( i = 0 ; i <= NumInput ; i++ ) {
                DeltaWeightIH[i][j] = 0.0 ;
                WeightIH[i][j] = 2.0 * ( rando() - 0.5 ) * smallwt ;
        }
}
for( k = 1 ; k <= NumOutput ; k ++ ) {        /* initialize WeightHO and DeltaWeightHO */
        for( j = 0 ; j <= NumHidden ; j++ ) {
                DeltaWeightHO[j][k] = 0.0 ;
                WeightHO[j][k] = 2.0 * ( rando() - 0.5 ) * smallwt ;
        }
}
```

Note, that it is a good idea to set all the initial *DeltaWeights* to zero at the same time.

We now have enough code to put together a working neural network program. I have cut and

pasted the above code into the file nn.c (which your browser should allow you to save into your own file space). I have added the standard #includes, declared all the variables, hard coded the standard XOR training data and values for *eta*, *alpha* and *smallwt*, #defined an over simple *rando()*, added some print statements to show what the network is doing, and wrapped the whole lot in a *main(){ }*. The file should compile and run in the normal way (e.g. using the UNIX commands 'cc nn.c -O -lm -o nn' and 'nn').

I've left plenty for the reader to do to convert this into a useful program, for example:

- Read the training data from file
- Allow the parameters (*eta*, *alpha*, *smallwt*, *NumHidden*, etc.) to be varied during runtime
- Have appropriate array sizes determined and allocate them memory during runtime
- Saving of weights to file, and reading them back in again
- Plotting of errors, output activations, etc. during training

There are also numerous network variations that could be implemented, for example:

- Batch learning, rather than on-line learning
- Alternative activation functions (linear, tanh, etc.)
- Real (rather than binary) valued outputs require linear output functions
  ```
  Output[p][k] = SumO[p][k] ;
  DeltaO[k] = Target[p][k] - Output[p][k] ;
  ```
- Cross-Entropy cost function rather than Sum Squared Error
  ```
  Error -= ( Target[p][k] * log( Output[p][k] ) + ( 1.0 - Target[p][k] ) * log( 1.0 - Output[p][k] ) ) ;
  DeltaO[k] = Target[p][k] - Output[p][k] ;
  ```
- Separate training, validation and testing sets
- Weight decay / Regularization

But from here on, you're on your own. I hope you found this page useful...

This page is maintained by John Bullinaria. Last updated on 18 November 2002.