

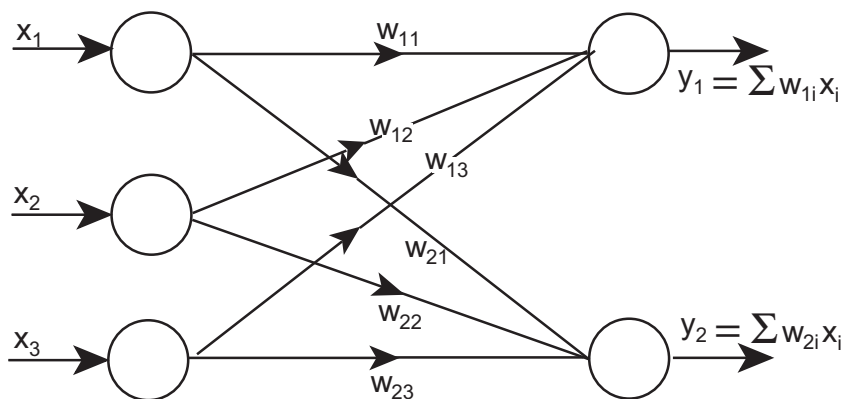
## FCS Seminar 4: Networks in Matlab

**Aims:** The aim of this worksheet is to reinforce the ideas of last week's lecture where we saw how matrices can be used to simplify network operations. In so doing it will introduce the concept of *network training* using the backpropagation algorithm. It will also help with learning matlab and highlight a few of its useful features.

**Learning Outcomes:** By the end you should be able to:

1. Write a matlab m-file
2. Transform inputs to a network into outputs in matlab using matrix-vector manipulations - as covered in last week's seminars
3. Perform network training using backpropagation
4. Investigate what functions can be 'learnt' by a network and why

### 1. Network Training



Suppose we have the single layer network pictured above, which takes in a 3 dimensional input vector  $\underline{x} = (x_1, x_2, x_3)^T$  and outputs a 2 dimensional output vector  $\underline{y} = (y_1, y_2)^T$ . NB: when I write vectors like:  $\underline{y} = (y_1, y_2)^T$  it is purely for convenience.  $\underline{y}$  is really a column vector, but to write it as such ruins the type-setting and takes up too much room.

Since the output is a weighted sum of the inputs, it depends on the weight matrix  $W$  where:

$$W = \begin{pmatrix} w_{11} & w_{12} & w_{13} \\ w_{21} & w_{22} & w_{23} \end{pmatrix}$$

(eg if all the  $w_{ij}$ 's are 0, the output will be 0 etc) and so changing the elements of  $W$  will change the output. Thus  $W$  defines the function (or mapping) the network performs and what functions it is *capable* of performing, as it defines how inputs are transformed into outputs.

Now suppose we have a set of input vectors (called *training vectors*) that we know the correct outputs for (the *target vectors*). The idea in network training is to use these 'examples' to alter the network weights so that by the end of training the network will output the correct target vector (or something 'close' to it) given any training vector. At this point the network is assumed to have 'learnt' the underlying (and unknown) mapping from training to target vectors and thus, given a novel input vector, will produce an output (often viewed as a prediction) consistent with the training data and, by implication, the mapping.

Whether this is possible in practice is another matter entirely and will depend not only on whether there is a relationship between training and target vectors, but also on the *representational capability* of the network used. Put simply, this means that simple networks can only perform simple mappings and will not be able to reproduce complex relationships between inputs and outputs.

### 2. The Task

In this worksheet, we will investigate the behaviour of the backpropagation algorithm. We will start with training-target sets that the network is capable of representing and then investigate what happens when it is

given a training-target combination it cannot reproduce<sup>1</sup>. As we need to control the complexity of the training-target mapping, we will therefore generate both sets ourselves, meaning that the underlying relationship is known (which clearly would not normally be the case).

The worksheet therefore starts by generating the training and target data, followed by performing network operations on these vectors, before coding the backpropagation training algorithm. Once this done, the questions in section 3 can be answered.

### 2.1 Generate training-target vectors and store in matrices

Start by opening a new m-file (**BackPropEG.m**, for instance) in which you will generate 25 random training input vectors and corresponding target vectors. The input data  $\underline{x}$  is three dimensional:

$$\underline{x} = \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} \text{ where: } x_1, x_2, x_3 \in [0, 1]$$

so store the inputs in a  $3 \times 25$  matrix X, each column of which is an input vector (Tips: use the function 'rand' to generate the random input data). Next generate a  $2 \times 25$  matrix T each column of which is a target vector  $\underline{t} = (t_1, t_2)^T$ , one for each training vector. The mapping we want the network to reproduce is:

$$\begin{aligned} t_1 &= 2x_1 - 0.5x_3 \\ t_2 &= -3x_2 \end{aligned}$$

or, using matrix notation

$$\underline{t} = \begin{pmatrix} t_1 \\ t_2 \end{pmatrix} = \begin{pmatrix} 2x_1 - 0.5x_3 \\ -3x_2 \end{pmatrix} = \begin{pmatrix} 2 & 0 & -0.5 \\ 0 & -3 & 0 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} = \begin{pmatrix} 2 & 0 & -0.5 \\ 0 & -3 & 0 \end{pmatrix} \underline{x} = S\underline{x}$$

Enter the matrix S and use matrix multiplication to generate T from S and X (see last week's slides on network operations for more details).

### 2.2 Perform network operations

Generate some initial weights for the network and store them as a matrix W. These should be a matrix of random values in the range  $[-0.5, 0.5]$ . Remember that the weight matrix W has dimensions  $m \times d$  where  $m$  is the dimension of the output (target) vectors and  $d$  the dimension of the input (training) vectors. Thus use 'rand(m,d)' and remember that if  $x \in [0, 1]^2$  then  $x - 0.5 \in [-0.5, 0.5]$ . Now use these weights to generate a matrix of output vectors Y from the training and weight matrices X and W via matrix multiplication.

How closely do the output vectors match the target vectors (they shouldn't be particularly close)? Print the outputs and target vectors to the screen () and compare them. To examine this further, calculate the *sum of squares error* between all the output vectors and the target vectors. The square error between an output vector  $\underline{y}$  and target  $\underline{t}$  is  $E = \sum_j (y_j - t_j)^2$ . The sum of squares error for all the vectors is, as the name implies, the sum of these errors for each vector. The easiest way to do this is to calculate all the outputs for the new weights as a matrix  $Y = WX$ . Then get a matrix of differences with  $E = Y - T$ . Use 'E.^2' to square every element of this matrix and 'sum(sum(E))' to add them up.

Try running this a few times to see the size of the error and how it varies. So that the same training and target vectors are used each time write a subfunction: SumSqError(X,T,W) taking the training, target and weight matrices as inputs and returning the error, and use this in a loop eg:

```
for i = 1:10
    Generate random weight matrix W
    Err=SumSqError(X,T,W)
end
```

To visualise how close the network output is to the targets, write another sub-function PlotOutputs(X,T,W) which takes training, target and weight matrices as inputs and plots the network outputs together with the target vectors. The simplest way is to plot the first row of each matrix against the second row using plot(M(1,:),M(2,:),'bx'), since if no line style is specified, plot will do a scatter plot. Plot the outputs as blue crosses and the targets as red circles. Use the function 'hold' to see both lines on one axis (see 'help hold').

<sup>1</sup>Often in investigations it is more instructive when a technique fails, and to see how it fails, than it is to see it working

<sup>2</sup>The symbol  $\in$  means 'is a member of' so  $x \in [0, 1]$  is shorthand for 'x is a number between 0 and 1'

### 2.3 Network training via backpropagation

The backpropagation algorithm is one of the best known network training algorithms and is a good exemplar for understanding network training in general. It works by using the difference between the output vectors generated by a particular weight matrix and the target vectors as an error signal to direct the learning process and iteratively reduce the error. More precisely, the outputs and targets are used to determine the gradient of the error with respect to each weight ( $-\frac{\partial E}{\partial w_{ij}} = -2(y_i - t_i)x_j$ ), so that weights can be updated and error reduced via gradient descent.<sup>3</sup> It is therefore convenient to work out a matrix of weight updates (one update for each weight), as this can simply be added to the original weight matrix to generate the new one. The algorithm can be summarised as follows:

1. Randomise the order of the training vectors.
2. Loop through each input vector updating the weights via:

$$W_{new} = W_{old} - 2\eta(\underline{y} - \underline{t})\underline{x}^T$$

where:  $\underline{y} = W\underline{x}$  and  $\eta \in (0, 1)$  is the learning rate (start with  $\eta = 0.05$ ).

3. Calculate the sum square error between the input vectors and targets, as described in section 2.2. If the error is less than a user-specified tolerance value (say, 0.01) STOP. Otherwise repeat from 1.

Note that question 6 from last week's problem sheet showed that the matrix equation in step 2 calculates the correct weight updates ie that:

$$\text{If } D = -2(\underline{y} - \underline{t})\underline{x}^T \quad \text{then: } d_{ij} = -\frac{\partial E}{\partial w_{ij}} = -2(y_i - t_i)x_j$$
$$d_{ij} = -\frac{\partial E}{\partial w_{ij}} = -2(y_i - t_i)x_j$$

In pseudo-code the algorithm will look something like:

```
Generate random training vectors
Generate target vectors from training vectors
Generate initial random network weights
Calculate network output and sum square error for these weights
while(sum square error > 0.01)
    for each training vector in random order
        Update weights
        Calculate and output sum square error
        Plot outputs and targets
        Pause so you can see the output generated
    end
end
Output weights
```

The code you wrote in section 2.1 generates the training and target vectors and initial weights. You also wrote a subfunction which calculates the error for a set of weights and one which plots outputs and targets in section 2.2. The only thing that needs to be done is to write a subfunction that takes the old weights, an input vector and a target vector as inputs and returns the new weights, and to wrap it all inside a while and for loop. Remember  $\underline{y} = W\underline{x}$  and update the weights using:  $W_{new} = W_{old} - 2\eta(\underline{y} - \underline{t})\underline{x}^T$ ; where  $\eta \in (0, 1]$  (start with  $\eta = 0.05$ ). To do the loops, look at section 3 of the matlab notes and, in particular, task 3. For ways to pause to view the output, see **PlottingExamples.m**. NB if you get bored of pausing after every training vector is used, move the plotting and pausing commands to outside the 'for' loop. This will now pause after the whole set of training vectors has been used (known as an *iteration* of the training algorithm).

PTO for questions.

---

<sup>3</sup>Full details of gradients, differentiation and gradient descent will be given next week. It is not important to understand these concepts in order to complete this worksheet

#### 4. Questions

1. What happens to the weight matrix after network training?
2. What effect does changing the learning rate have on the number of training iterations needed?
3. What effect does changing the stopping clause in the while loop have (eg if you did 'while(sum square error > 2)')?
4. What happens if you add some uniformly distributed random noise  $s$  in the range  $[-0.05, 0.05]$  to the target values so that:

$$\underline{t} = \begin{pmatrix} t_1 \\ t_2 \end{pmatrix} = \begin{pmatrix} 2x_1 - 0.5x_3 + s \\ -3x_2 + s \end{pmatrix}$$

The easiest way to do this is to generate target values as before but then use  $0.1 * \text{rand}(\text{size}(\underline{T})) - 0.05$  to generate some noise which can be added to the target matrix. What happens to the output-target pictures? Does the algorithm ever stop (if stopping clause is 'while(sum square error > 0.01')? What happens if you increase the stopping clause so that the algorithm stops? Can you suggest a better way to make it stop? Briefly highlight pros and cons of this method.

5. What happens if we make  $t_2 = -3x_2^2$ ? Again the easiest way to do this is to generate the targets as before but then do:  $\underline{T}(2,:) = -3 * (\underline{X}(2,:).^2)$ . Why can't the network approximate this function? Tip: To examine network output more systematically, it can be better to generate a new set of input vectors to test the network with which cover the input range evenly,  $\text{NewX} = [0:0.05:1; 0:0.05:1; 0:0.05:1]$ ; for instance. While you still train the network on the random training vectors, you can test it by generating and plotting targets and network output for  $\text{NewX}$ . This can help you see if there is a range of values which cannot be approximated, or that are approximated well.

Marks out of 10: 1 mark for questions 1 and 3, 2 for question 2 and 3 for questions 4 and 5.