

## Neural Nets

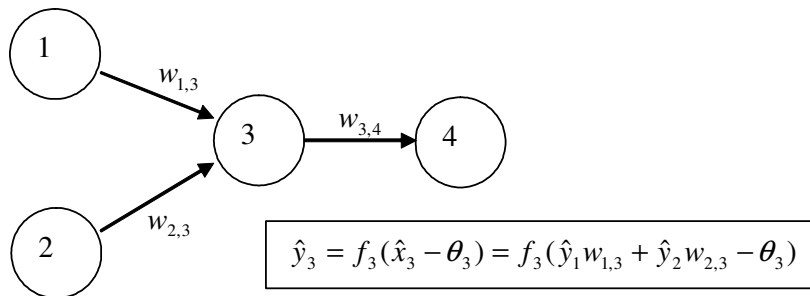
Neural Networks (NNs) can be used for regression, classification and clustering. Ordinary linear regression and logistic regression can both be formulated as special cases. The price of this flexibility is interpretability: NNs act like a black box, which gives an answer but with no intuition as to why.

NNs are described in terms of a graphical network, the nodes of which are the “neurons”. Every node  $j$  has an input potential  $\hat{x}_j$  and an output  $\hat{y}_j$ , which is obtained by applying a bias  $\theta_j$  and a function  $f_j$ , called the *activation function* (typically non-linear), to the input potential. That is,  $\hat{y}_j = f_j(\hat{x}_j - \theta_j)$ . An edge from node  $i$  to node  $j$  is given weight  $w_{i,j}$  (which can be negative). If there is no (directed) edge from  $i$  to  $j$  then  $w_{i,j} = 0$ .

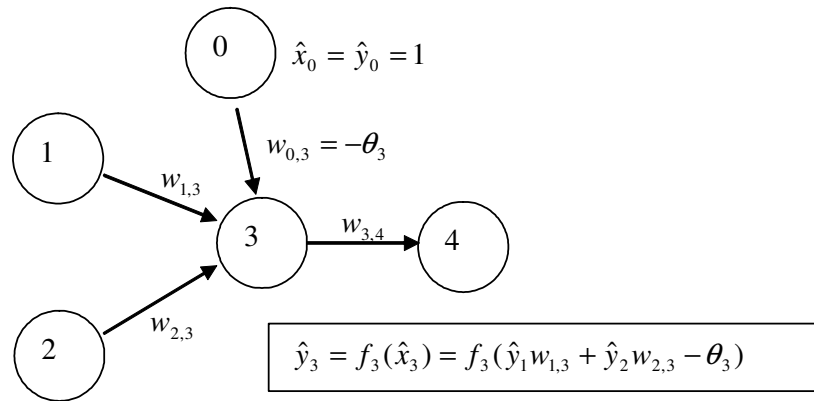
Input potentials are calculated iteratively. For a node  $j$  with no predecessors (called an input node) we assume that  $\hat{x}_j$  is known and we put  $f_j(x) = x$  and  $\theta_j = 0$ , so that  $\hat{y}_j = \hat{x}_j$ . For a node  $j$  with predecessors we put

$$\hat{x}_j = \sum_{i: i \rightarrow j} \hat{y}_i w_{i,j}$$

Provided that the network is acyclic, we can apply this rule to calculate  $\hat{x}_j$  and  $\hat{y}_j$  for all nodes  $j$ . Nodes with no successors are called output nodes.



We note that a bias  $\theta_j$  at node  $j$  is equivalent to having no bias and extra input node  $i$  with  $\hat{x}_i = 1$  and  $w_{i,j} = -\theta_j$ . Accordingly we will ignore bias terms from here on.



Common choices for an activation function  $f$  are

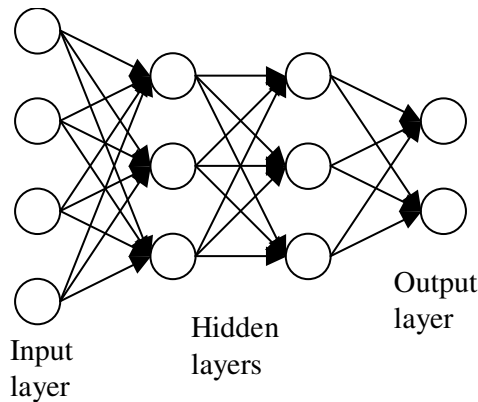
Logistic (or sigmoid):  $f(x) = \frac{1}{1 + e^{-x}}$

Threshold:  $f(x) = \begin{cases} 1 & x \geq 0 \\ 0 & x < 0 \end{cases}$

Linear:  $f(x) = x$

### Feed-forward networks

A feed-forward neural-network (FFNN) has nodes arranged in ordered layers. For example



The first layer is the input layer, followed by 0 or more hidden layers, then finishing with the output layer. Every node in layer  $k$  is connected to every node in layer  $k + 1$ . Note that some authors count the layers of *edges* rather than the layers of *nodes*.

We can consider a FFNN as a function  $F : \mathfrak{R}^k \rightarrow \mathfrak{R}^m$  where  $k$  is the number of input nodes and  $m$  is the number of output nodes. Suppose the input nodes are  $1, 2, \dots, k$  and the output nodes are  $k+1, \dots, k+m$ , then

$$F(\hat{x}_1, \dots, \hat{x}_k) = (\hat{y}_{k+1}, \dots, \hat{y}_{k+m})$$

Neural nets are called *black-box* models because the behaviour of  $F$  is very hard to predict based on the values of the edge weights. We just think of the function as a machine sitting in a black box, so that we can not see its workings. However, it can be shown that a feed-forward network with one hidden layer using a sigmoid activation function, can approximate any given function arbitrarily closely, provided the hidden layer has enough neurons. (Though this does not mean that we would never want to use more than 1 hidden layer: more layers can produce more efficient approximations.) See Ripley [4] for a proof.

Feed-forward networks are used for regression and classification. In the NN world this is called *supervised learning*. A significant drawback of neural network models is that they can not be easily interpreted. That is, the value of any single weight  $w_{i,j}$  tells you practically nothing about how the model works.

### Regression

For regression we have data of the form

$$(x_1(i), \dots, x_k(i), y_1(i), \dots, y_m(i)), i = 1, \dots, n.$$

The  $x_i$  are independent variables (inputs) and the  $y_i$  are dependent variables (outputs). We suppose that all are continuous, though we can easily accommodate ordinal variables. We take the input layer to have exactly  $k$  nodes and the output layer  $m$  nodes. We are free to choose the number of hidden layers and their size; our choice will have an impact on the performance of the network. 1 hidden layer with around  $k$  nodes is not unusual. Output nodes use the linear activation function and hidden nodes typically use the logistic function.

We initialise the network by setting the input  $\hat{x}_i$  at node  $i$  in the input layer equal to  $x_i$ . Our aim is to choose the weights  $w_{ij}$  so that the output levels at the output nodes match  $(y_1, \dots, y_m)$  as closely as possible. To do this we need some measure of “closeness”.

Let  $\hat{y}_i$  be the final output from node  $i$  in the output layer. Take as our training set the records

$$(\mathbf{x}(i), \mathbf{y}(i)) = (x_1(i), \dots, x_k(i), y_1(i), \dots, y_m(i)) \text{ for } i = 1 \text{ to } n.$$

For a given set of weights  $\mathbf{w} = (w_{ij})$ , the total error on the training set is

$$E(\mathbf{w}) = \sum_{i=1}^n d(\hat{\mathbf{y}}(i), \mathbf{y}(i))$$

where  $d(, )$  is some loss function, for example

$$d(\hat{\mathbf{y}}, \mathbf{y}) = \sum_{j=1}^m (\hat{y}_j - y_j)^2$$

(Minimising the squared error is equivalent to maximising the likelihood if  $\hat{y}_i = y_i + \varepsilon_i$  where  $\varepsilon_i \sim N(0, \sigma^2)$ . If we believe another error structure is more appropriate, then we should choose the loss function to be the appropriate minus log likelihood.)

If we use a linear activation function and 0 hidden layers, then the NN performs least-squares linear regression.

### Classification

We suppose that we have data of the form

$$(x_1(i), \dots, x_k(i), y(i)), i = 1, \dots, n,$$

where  $y$  is categorical, taking values in a set of size  $m$ . Categorical input data is dealt with by recoding as binary inputs (this applies to regression as well as classification). For categorical output we distinguish between binary output,  $m = 2$ , and cases where  $m > 2$ .

For binary output we use a single output node with a logistic activation function. Hidden nodes also use the logistic activation function. As above we set the input  $\hat{x}_i$  at node  $i$  in the input layer equal to  $x_i$ . We interpret the output  $\hat{y}_j$  from the output node  $j$  as the probability  $y = 1$ . Given this interpretation, we can fit the weights using maximum likelihood, or equivalently use minus log-likelihood as the loss function:

$$d(\hat{y}, y) = -(y \log \hat{y} + (1 - y) \log(1 - \hat{y}))$$

This is analogous to logistic regression, and in the case with 0 hidden layers is precisely logistic regression. This loss function can also be interpreted as entropy.

In the case where  $y$  takes on  $m > 2$  possible values, we associate with each possible value a separate output node. The output class then corresponds to the node with the largest output. Number the output nodes  $k+1, \dots, k+m$ , then if  $y$  is in  $\{v_1, \dots, v_m\}$ , the output class is

$$\hat{v} = v_j \text{ where } \hat{y}_{k+j} = \max_i \hat{y}_{k+i}.$$

We use a logistic activation function for hidden nodes, but for the output nodes use the so-called *softmax* function:  $f(x) = \exp(x)$ . We then interpret the normed outputs as probabilities:

$$\tilde{y}_{k+i} := \frac{\hat{y}_{k+i}}{\sum_{j=1}^m \hat{y}_{k+j}} = \frac{\exp(\hat{x}_{k+i})}{\sum_{j=1}^m \exp(\hat{x}_{k+j})} = P(y = v_i)$$

This is analogous to multiple-logistic regression.

To maximise the likelihood we minimise the total error using the loss function:

$$d(\hat{y}, y) = -\sum_{i=1}^m I(y = v_i) \log \tilde{y}_{k+i}$$

## Training the network

For a given choice of loss function  $d(, )$ , we wish to find those weights  $\mathbf{w}$  which minimise  $E(\mathbf{w})$ . This is in general a non-trivial problem requiring numerical methods, which are complicated by the fact that NNs can have many local minima. Iterative local search methods are typically used, which have the advantage of scalability (with respect to  $n$ , the amount of training data), but can get stuck in local minima, so that the quality of solution depends on the initial weights chosen.

The first successful method for fitting FFNNs was the **backpropogation** algorithm. This is an iterative gradient based method, tailored to the layered structure of a feed-forward network [1]. We describe the method below, though observe that it is now outdated. We will assume that we use the sum-of-squares loss function.

Suppose that, for  $p = 1, \dots, n$ , we have inputs  $\mathbf{x}(p)$ , outputs  $\mathbf{y}(p)$  and model outputs  $\hat{\mathbf{y}}(p) = F(\mathbf{x}(p); \mathbf{w})$  where  $\mathbf{w}$  is a vector of parameters (weights and biases in the case

of a neural network). We wish to find  $\mathbf{w}$  which minimizes the sum of squared errors (SSE)  $E = E(\mathbf{w}) = \sum_{p=1}^n \|\hat{\mathbf{y}}(p) - \mathbf{y}(p)\|^2$ . Starting with some  $\mathbf{w}(0)$  we put

$$\mathbf{w}(k+1) = \mathbf{w}(k) - \eta \nabla E(\mathbf{w}(k)) \quad (*)$$

where  $\eta > 0$  is a user supplied tuning parameter. The algorithm iterates until the  $\mathbf{w}(k)$  converge. If the algorithm does converge to some  $\mathbf{w}$ , then we must have  $\nabla E(\mathbf{w}) = 0$ . This will happen provided  $E(\mathbf{w})$  is smooth enough and you start “close enough” to a local minimum.<sup>1</sup> (Methods that make use of the second derivatives, that is the Hessian, typically perform better.)

### Example (least-squares linear regression)

Suppose that we have 1-dimensional output ( $m = 1$ ) and suppose that  $F(\mathbf{x}, \mathbf{w}) = \mathbf{x}'\mathbf{w}$ , then  $E(\mathbf{w}) = \sum_{p=1}^n (\mathbf{x}'(p)\mathbf{w} - y(p))^2 = \|\mathbf{X}\mathbf{w} - \mathbf{y}\|^2$  where  $\mathbf{X}$  is the matrix whose  $p$ -th row is  $\mathbf{x}'(p)$  and  $\mathbf{y}$  is the vector whose  $p$ -th element is  $y(p)$ . In this case we have  $\nabla E(\mathbf{w}) = 2\mathbf{X}'(\mathbf{X}\mathbf{w} - \mathbf{y})$  so  $\mathbf{w}(k+1) = \mathbf{w}(k) - 2\eta\mathbf{X}'(\mathbf{X}\mathbf{w}(k) - \mathbf{y})$ .

Note that putting  $\nabla E(\mathbf{w}) = \mathbf{0}$  we get  $\mathbf{X}'\mathbf{X}\mathbf{w} = \mathbf{X}'\mathbf{y}$  which we can solve directly (provided  $\mathbf{X}'\mathbf{X}$  is non-singular), so this iterative method is not recommended in this case. None-the-less, you do see variants of this method used to solve constrained linear least-squares problems. For example, to minimize  $E(\mathbf{w})$  subject to  $\mathbf{w} \geq 0$  you can use

$$\mathbf{w}(k+1) = \max(\mathbf{w}(k) - \eta \nabla E(\mathbf{w}(k)), \mathbf{0})$$

(here the max is taken componentwise).

### Row and block updating

A variant of the algorithm (\*), called row updating, is given by

$$\mathbf{w}(k+1) = \mathbf{w}(k) - \eta \nabla E_p(\mathbf{w}(k))$$

where  $E_p(\mathbf{w}) = \|\hat{\mathbf{y}}(p) - \mathbf{y}(p)\|^2$  is the  $p$ -th component of  $E(\mathbf{w})$  and  $p = (k \bmod n) + 1$ .

More generally if  $I(1), \dots, I(m)$  form a partition of  $\{1, \dots, n\}$  then for

$$E_{I(q)}(\mathbf{w}) = \sum_{p \in I(q)} \|\hat{\mathbf{y}}(p) - \mathbf{y}(p)\|^2$$
 we can take

$$\mathbf{w}(k+1) = \mathbf{w}(k) - \eta \nabla E_{I(q)}(\mathbf{w}(k))$$

for  $q = (k \bmod m) + 1$ . This is called block updating.

The order in which rows/blocks are used can be randomized rather than deterministic.

An advantage of row/block updating is that  $\nabla E_p$  and  $\nabla E_{I(q)}$  are faster to calculate than  $\nabla E$ . The disadvantage is that convergence is less certain.

Note that this iterative scheme, with row or block updating, can be used for any error of the form  $E = E(\mathbf{w}) = \sum_{p=1}^n d(\hat{\mathbf{y}}(p), \mathbf{y}(p))$  provided  $d(\hat{\mathbf{y}}(p), \mathbf{y}(p))$  is non-negative and differentiable w.r.t.  $\mathbf{w}$ .

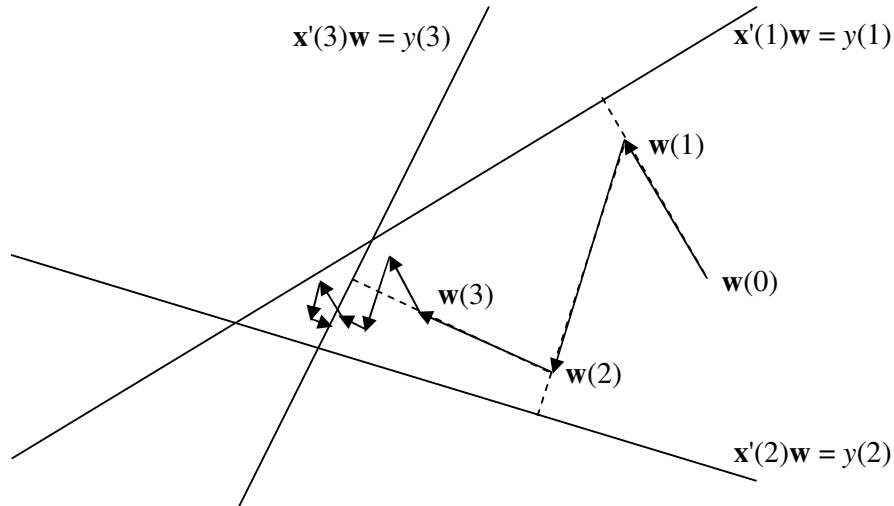
---

<sup>1</sup> P. Wolfe, Convergence conditions for ascent methods. SIAM Review, Vol. 11, pp. 226–235, 1969

**Example (least-squares linear regression continued)**

In the case of linear least-squares (as above) there is an easy geometrical interpretation of row-updating. In this case we have

$\mathbf{w}(k+1) = \mathbf{w}(k) - 2\eta \mathbf{x}(i)(\mathbf{x}'(i)\mathbf{w}(k) - y(i))$ . Thus  $\mathbf{w}(k+1)$  is obtained from  $\mathbf{w}(k)$  by moving directly towards the hyperplane  $\mathbf{x}'(i)\mathbf{w}(k) = y(i)$  (that is, moving along the orthogonal projection of  $\mathbf{w}(k)$  onto the hyperplane). The distance moved is proportional to the distance from the hyperplane. This is illustrated below in an example with  $k = 2$  and  $n = 3$ .

**The backpropagation algorithm**

Given a FFNN we fix the architecture and the activation functions, then attempt to find weights and biases to minimize the SSE  $E = E(\mathbf{w}) = \sum_{p=1}^n \|\hat{\mathbf{y}}(p) - \mathbf{y}(p)\|^2$ .

Backpropagation is the name given to the iterative gradient based algorithm described above, with row-updating applied to the sum of squared errors (SSE) for a feed-forward neural net. It was introduced by Rumelhart, Hinton and Williams in 1986.

Let  $f_j$  be the activation function,  $\theta_j$  the bias at node  $j$  and  $w_{i,j}$  the weight between node  $i$  and node  $j$  (Rumelhart, Hinton and Williams call this  $w_{j,i}$ ). By using extra nodes with fixed values it is possible to regard the biases as weights, so for the purposes of describing the backpropagation algorithm it is sufficient to consider weights only. From the discussion above, we only need to show how to calculate  $\partial E_p(\mathbf{w}) / \partial w_{i,j}$  for each  $i, j$  and  $1 \leq p \leq n$ .

We need some more notation. Let  $\hat{y}_j = \hat{y}_j(\mathbf{x}; \mathbf{w})$  be the value of node  $j$  in the network when the input nodes have values  $\mathbf{x}$  and the weights are  $\mathbf{w}$ . Let  $\hat{x}_j = \sum_{i \rightarrow j} \hat{y}_i w_{i,j}$  be the weighted sum of the inputs into node  $j$  (here  $i \rightarrow j$  means there is an edge from  $i$  to  $j$ ).

Rumelhart, Hinton and Williams use the notation  $net_j$  for  $\hat{x}_j$ ). Thus  $\hat{y}_j = f_j(\hat{x}_j)$ . For input nodes we have  $\hat{y}_j = \hat{x}_j = x_j$ .

Applying the chain rule we have

$$\frac{\partial E_p(\mathbf{w})}{\partial w_{i,j}} = \frac{\partial E_p(\mathbf{w})}{\partial \hat{x}_j} \frac{\partial \hat{x}_j}{\partial w_{i,j}} = \frac{\partial E_p(\mathbf{w})}{\partial \hat{x}_j} \hat{y}_i$$

$$\frac{\partial E_p(\mathbf{w})}{\partial \hat{x}_j} = \frac{\partial E_p(\mathbf{w})}{\partial \hat{y}_j} \frac{\partial \hat{y}_j}{\partial \hat{x}_j} = \frac{\partial E_p(\mathbf{w})}{\partial \hat{y}_j} f_j'(\hat{x}_j)$$

If  $j$  is an output node then since  $E_p(\mathbf{w}) = \sum_{h \in \text{output layer}} (\hat{y}_h(p) - y_h(p))^2$  we have

$$\frac{\partial E_p(\mathbf{w})}{\partial \hat{y}_j} = 2(\hat{y}_j(p) - y_j(p))$$

$$\frac{\partial E_p(\mathbf{w})}{\partial \hat{x}_j} = 2f_j'(\hat{x}_j)(\hat{y}_j(p) - y_j(p))$$

If  $j$  is not an output node then

$$\frac{\partial E_p(\mathbf{w})}{\partial \hat{y}_j} = \sum_{k:j \rightarrow k} \frac{\partial E_p(\mathbf{w})}{\partial \hat{x}_k} \frac{\partial \hat{x}_k}{\partial \hat{y}_j} = \sum_{k:j \rightarrow k} \frac{\partial E_p(\mathbf{w})}{\partial \hat{x}_k} w_{j,k}$$

$$\frac{\partial E_p(\mathbf{w})}{\partial \hat{x}_j} = f_j'(\hat{x}_j) \frac{\partial E_p(\mathbf{w})}{\partial \hat{y}_j} = f_j'(\hat{x}_j) \sum_{k:j \rightarrow k} \frac{\partial E_p(\mathbf{w})}{\partial \hat{x}_k} w_{j,k}$$

These last equations allow us to calculate  $\partial E_p(\mathbf{w})/\partial \hat{x}_j$  recursively: firstly for nodes  $j$  in the output layer, then for nodes in the last hidden layer, then the second last hidden layer, and so on. In fact this will work for any NN which has no cycles.

Write  $\delta_j(p)$  for  $\partial E_p(\mathbf{w})/\partial \hat{x}_j$  then the backpropagation algorithm is, for each weight  $w_{i,j}$ ,

$$w_{i,j}(k+1) = w_{i,j}(k) - \eta \sum_p \hat{y}_i(p) \delta_j(p) \quad (\dagger)$$

### Remarks

1) The backpropagation algorithm requires the existence of  $f_j'$ , so clearly it can not be used with the threshold activation function.

2) We can have  $E_p(\mathbf{w}) = d(\hat{\mathbf{y}}(p), \mathbf{y}(p))$  provided  $d(\hat{\mathbf{y}}(p), \mathbf{y}(p))$  is non-negative and differentiable w.r.t.  $\mathbf{w}$ . In particular the backpropagation algorithm can be used to maximize the log likelihood, though methods that make use of second derivative information typically perform better.

3) If observations  $(\mathbf{x}(p), \mathbf{y}(p))$  continually arrive over time then rather than cycling through a finite set of observations,  $p = 1, \dots, n$ , we can just use each observation when it arrives to update  $\mathbf{w}(k)$ , then discard it. This is analogous to row updating, but with an infinite number of rows.

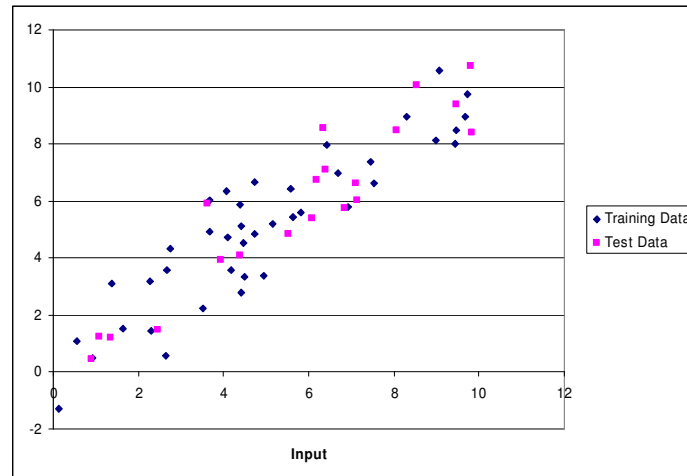
4) The convergence of the algorithm depends on  $\mathbf{w}(0)$  and  $\eta$ . There is no way of knowing before-hand the best choice for either of these. Typically we choose  $\mathbf{w}(0)$  randomly, and repeat the algorithm a few times to obtain a good fit.

## Overfitting

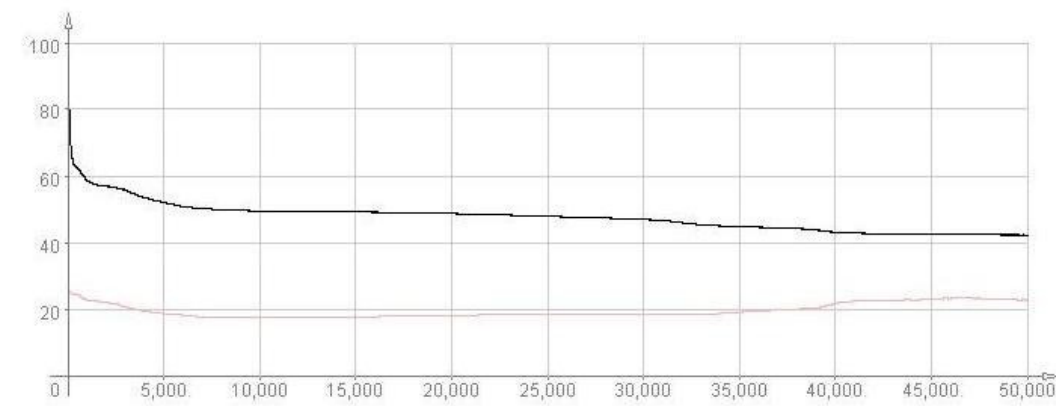
As they can have a large number of parameters, NNs are prone to overfitting. Consider the following example: we have 60 data points generated by the model

$$Y(i) = X(i) + E(i)$$

Where the  $X(i)$  are i.i.d.  $U(0,10)$  and the  $E(i)$  are i.i.d.  $N(0,1)$  random variables. We split the data into a training set of size 40 and a test set of size 20. A plot of the data points is given below.



We fit a FFNN to the training data using the backpropagation algorithm<sup>2</sup>. It has one input node; a single hidden layer with 15 nodes and sigmoid activation functions; and a single output node with a linear activation function. The sum of squared errors on the training and test data sets is given below, for the first 50,000 cycles of the training algorithm. The SSE on the training set is the darker line.



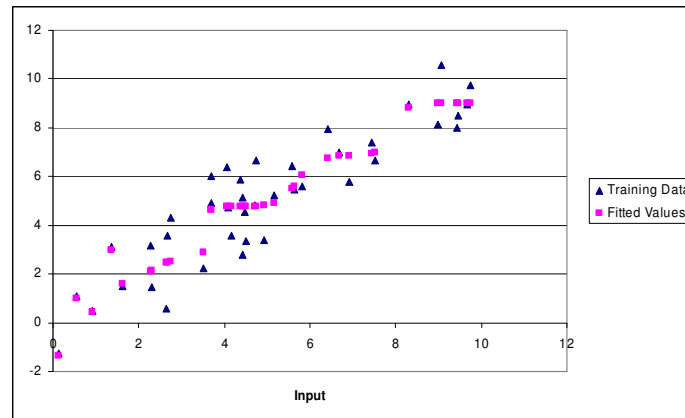
<sup>2</sup> The software used was JavaNNS

Using the usual statistical convention, we use lower case  $x$  and  $y$  to refer to observed values of the random variables  $X$  and  $Y$ . A good model for  $y$  as a function of  $x$  is  $y = f(x) = E(Y | X = x) = x$ . For this model we have

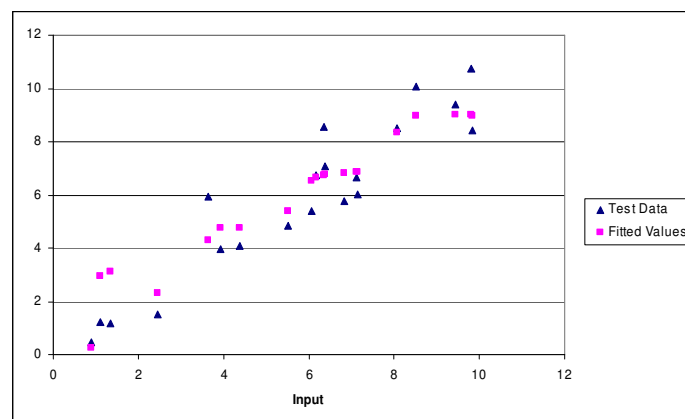
$$E \sum_{i=1}^N (f(X(i)) - Y(i))^2 = \sum_{i=1}^N EE(i)^2 = N, \text{ thus a SSE of around 40 for the training set}$$

and around 20 for the test set is reasonably good. None-the-less, we see that as training of the FFNN progresses, the error on the test set starts to get worse. This is indicative of *overfitting*.

To see how the model has overfitted the training data, we consider the fitted value of each of the input patterns<sup>3</sup>. Below we plot the original output and fitted values against the input for the training set. We see that for values of  $x$  near 0, the FFNN has fitted the observed  $y$  closely, and in so doing has fitted the noise as well as the underlying mean.



If we plot the observed and fitted responses for the test data, we see that near 0 the FFNN does not perform well.



Three different approaches are used to avoid overfitting by neural networks: *early stopping*, *pruning* and *regularisation*.

<sup>3</sup> The fitted values were obtained rather laboriously using the Updating function of JavaNNS

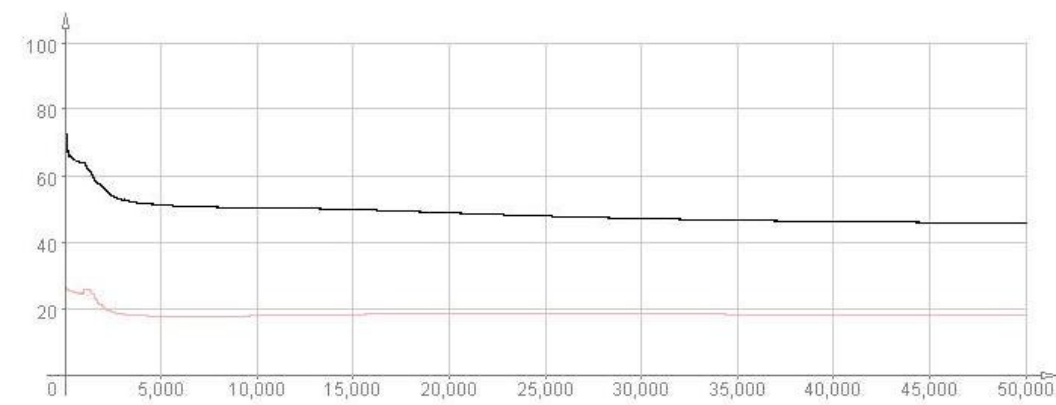
### Early stopping

The idea behind early stopping is that the model is underfitted when you choose  $\mathbf{w}(0)$ , will eventually become overfitted, but will be just-right somewhere in between. The decision of which  $\mathbf{w}(k)$  to choose is made by looking at the error on some validation data set, separate to the training set. The error measure used on the validation set can be  $E(\mathbf{w}(k))$ , but could also be the misclassification error, for example. You can think of the model as a stiff piece of string: it starts reasonably straight then with each iteration of the fitting algorithm you mould it more closely to the training set and you want to stop moulding it once you have captured the main features of the data but before you start picking up the small fluctuations due to errors.

Early stopping is difficult in practice, because the error on the validation data set can go up and down many times as the fitting algorithm progresses, so you can never know if you have stopped too early. Moreover, the point at which the error turns around depends heavily on the initial weights (which are typically chosen randomly).

### Pruning

As for decision trees and logistic regression models, we can avoid overfitting by looking for a model which is simpler and thus only picks up features of the data, not noise. For a FFNN, a simpler model is one with fewer edges and/or nodes in the hidden layers. Consider again the example above. If we fit a FFNN as before, but with only 5 nodes in the hidden layer, we obtain the following errors on the training and test data sets.



We see that the SSE on the training data set is very close to that of the more complex model, but more importantly the SSE on the test data set does not creep up again once it has reached its minimum. In this case the FFNN is complex enough to capture the main features of the data, but not complex enough to model the noise as well.

This is an example of the *principle of parsimony* in action. The principle of parsimony states that good models are parsimonious, in that they are just rich enough to capture the important features of the data, but no more.

Techniques for pruning neural nets have been developed, see for example [2, 3], however in practice regularisation is a more straight-forward and just as effective approach

### Regularisation

Pruning a NN does not make it any easier to interpret. The idea behind regularisation is not to set certain weights to zero, but just to encourage them to be small. That is, instead of minimising  $E(\mathbf{w})$  we minimise

$$\tilde{E}(\mathbf{w}) = E(\mathbf{w}) + \alpha\Omega(\mathbf{w})$$

where  $\Omega(\mathbf{w})$  measures the complexity of the network, and  $\alpha$  is a user supplied weight.  $\Omega(\mathbf{w})$  is called a penalty or smoothing term, and the larger  $\alpha$ , the “smoother” the fitted model will be. One popular way of measuring network complexity is

$$\Omega(\mathbf{w}) = \frac{1}{2} \sum_i w_i^2$$

Using this  $\Omega$  the backpropagation algorithm (†) becomes

$$w_{i,j}(k+1) = w_{i,j}(k) - \eta \sum_p \hat{y}_i(p) \delta_j(p) - \alpha \eta w_{i,j}(k)$$

For this reason the parameter  $\alpha$  is sometimes called a weight decay parameter.

### Practical considerations

- 1) NNs perform better when the inputs range over similar scales. That is, rescaling the inputs can improve performance.
- 2) The performance of a NN can be improved by reducing the dimensionality of the input vector, for example by principle components.
- 3) FFNN can be improved using skip connections, that is, edges directly linking the input nodes to the output nodes.

### Other types of network

We briefly mention two other types of NN.

#### Hopfield nets

Hopfield nets are complete graphs with symmetric weights  $w_{ij} = w_{ji}$ . They can be trained as classifiers using *Hebbian learning*. They are not much used in data-mining.

#### Kohonen Self Organising Maps (SOM)

SOMs are used for clustering. In NN terminology this is called *unsupervised learning*. The Kohonen SOM is a non-hierarchical method which has similarities to the k-means algorithm. SOMs are a valid technique for clustering.

### References

- [1] D.E. Rumelhart, G.E. Hinton & R.J. Williams. *Learning internal representations by error propagation*. In D.E. Rumelhart & J.L. McClelland Eds, Parallel Distributed Processing, MIT Press, 1986.
- [2] S. Fahlman & C. Lebiere. *The cascade-correlation learning algorithm*. Tech. Rep. CMU-CS-90-100, Computer Science Dept., Carnegie Mellon Uni., 1990.
- [3] Y. Le Cun, J.S. Denker & S.A. Solla. *Optimal brain damage*. In D. Touretzky Ed., Advances in Neural Information Processing Systems 2, San Mateo CA. Morgan Kaufmann, 1990.
- [4] B.D. Ripley. *Pattern Recognition and Neural Networks*. CUP, 1996