Feed-forward Neural Nets: Additional Notes

Steepest Descent
Suppose that, for \( p = 1, \ldots, N \), we have inputs \( x(p) \), outputs \( y(p) \) and model outputs \( \hat{y}(p) = F(x(p); w) \) where \( w \) is a vector of parameters (weights and biases/thresholds in the case of a neural network). We wish to find \( w \) which minimizes the sum of squared errors (SSE) \( E = E(w) = \sum_{p=1}^{N} \| \hat{y}(p) - y(p) \|^2 \).

The steepest descent algorithm is an iterative procedure. Starting with some \( w(0) \) we put
\[
  w(k+1) = w(k) - \eta \nabla E(w(k))
\]
where \( \eta > 0 \) is a user supplied tuning parameter. The algorithm iterates until the \( w(k) \) converge. If \( E(w) \) is smooth enough then the algorithm will converge to a local minimum, provided you start “close enough” to the minimum.

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

Note that putting \( \nabla E(w) = 0 \) we get \( X'Xw = X'y \) which can be easily solved for \( w \) (without having to calculate \( (X'X)^{-1} \)), so the steepest descent method is not recommended in this case. None-the-less, you do see variants of steepest descent used to solve constrained linear least-squares problems. For example, to minimize \( E(w) \) subject to \( w \geq 0 \) you can use
\[
  w(k+1) = \max(w(k) - \eta \nabla E(w(k)), 0)
\]
(here the max is taken componentwise).

Row and block updating A variant of the algorithm (*), called row updating, is given by
\[
  w(k+1) = w(k) - \eta \nabla E_p(w(k))
\]
where \( E_p(w) = \| \hat{y}(p) - y(p) \|^2 \) is the p-th component of \( E(w) \) and \( p = (k \mod N) + 1 \).

More generally if \( I(1), \ldots, I(M) \) form a partition of \( \{1, \ldots, N\} \) then for
\[
  E_{I(q)}(w) = \sum_{p \in I(q)} \| \hat{y}(p) - y(p) \|^2
\]
we can take
\[
  w(k+1) = w(k) - \eta \nabla E_{I(q)}(w(k))
\]
for \( q = (k \mod M) + 1 \). 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 $\nabla E_{t(q)}$ are faster to calculate than $\nabla E$. The disadvantage is that convergence is less certain.

Note that row or block updating can be used for any error of the form

$$E = E(w) = \sum_{p=1}^{N} d(\hat{y}(p), y(p))$$

provided $d(\hat{y}(p), y(p))$ is non-negative and differentiable w.r.t. $w$.

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

$$E = E(w) = \sum_{p=1}^{N} \|\hat{y}(p) - y(p)\|^2.$$

Backpropagation is the name given to the steepest descent algorithm 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 $f$ be the activation function, $\theta_j$ the bias at node $j$ and $w_{ij}$ 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 \( \frac{\partial E_p(w)}{\partial w_{i,j}} \) for each \( i, j \) and \( 1 \leq p \leq N \).

We need some more notation. Let \( \hat{y}_j = \hat{y}_j(x; w) \) be the value of node \( j \) in the network when the input nodes have values \( x \) and the weights are \( w \) (if \( j \) is an output node then this coincides with our previous notation). 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 \( \hat{y}_j = f(\hat{x}_j) \). For input nodes we have \( \hat{y}_j = x_j \).

Applying the chain rule we have
\[
\frac{\partial E_p(w)}{\partial w_{i,j}} = \frac{\partial E_p(w)}{\partial \hat{x}_j} \frac{\partial \hat{x}_j}{\partial w_{i,j}} = \frac{\partial E_p(w)}{\partial \hat{x}_j} \hat{y}_i
\]
\[
\frac{\partial E_p(w)}{\partial \hat{x}_j} = \frac{\partial E_p(w)}{\partial \hat{y}_j} \frac{\partial \hat{y}_j}{\partial \hat{x}_j} = \frac{\partial E_p(w)}{\partial \hat{y}_j} f'(\hat{x}_j)
\]

If \( j \) is an output node then since \( E_p(w) = \sum_{h \text{output layer}} (\hat{y}_h(p) - y_h(p))^2 \) we have
\[
\frac{\partial E_p(w)}{\partial \hat{y}_j} = 2(\hat{y}_j(p) - y_j(p))
\]
\[
\frac{\partial E_p(w)}{\partial \hat{x}_j} = 2 f'(\hat{x}_j)(\hat{y}_j(p) - y_j(p))
\]

If \( j \) is not an output node then
\[
\frac{\partial E_p(w)}{\partial \hat{y}_j} = \sum_{k:j \rightarrow k} \frac{\partial E_p(w)}{\partial \hat{x}_k} \frac{\partial \hat{x}_k}{\partial \hat{y}_j} = \sum_{k:j \rightarrow k} \frac{\partial E_p(w)}{\partial \hat{x}_k} w_{j,k}
\]
\[
\frac{\partial E_p(w)}{\partial \hat{x}_j} = 2 f'(\hat{x}_j) \frac{\partial E_p(w)}{\partial \hat{y}_j} = 2 f'(\hat{x}_j) \sum_{k:j \rightarrow k} \frac{\partial E_p(w)}{\partial \hat{x}_k} w_{j,k}
\]

These last equations allow us to calculate \( \frac{\partial E_p(w)}{\partial \hat{x}_j} \) 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.

Remarks 1) The backpropogation algorithm requires the existence of \( f' \), so clearly it can not be used with the stepwise (also called threshold) activation function.
2) The activation function \( f \) can be allowed to depend on the node \( j \).
3) We can have \( E_p(w) = d(\hat{y}(p), y(p)) \) provided \( d(\hat{y}(p), y(p)) \) is non-negative and differentiable w.r.t. \( w \). In particular the backpropogation algorithm can be used to minimize the minus log likelihood.
4) If observations \((x(p), y(p))\) continually arrive over time then rather than cycling through a finite set of observations, \(p = 1, \ldots, N\), we can just use each observation when it arrives to update \(w(k)\), then discard it.

5) The final set of weights obtained by the algorithm depends on \(w(0)\), \(\eta\) and how many iterations are performed. There is no way of knowing before-hand the best choice for any of these.

**Overfitting Example**

We have 60 data points generated by the model

\[ \hat{Y}(i) = X(i) + E(i) \]

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 backpropogation algorithm\(^1\). 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.

----

\(^1\) 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 $y = f(x) = \mathbb{E}(Y \mid X = x) = x$. For this model we have

$$\mathbb{E} \sum_{i=1}^{N} (f(X(i)) - Y(i))^2 = \sum_{i=1}^{N} \mathbb{E}(i)^2 = N,$$

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\(^2\). 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.

\(^2\) The fitted values were obtained rather laboriously using the Updating function of JavaNNS
To avoid overfitting we can try to stop training before overfitting occurs, by looking at the SSE on the test data set. This is difficult in practice however, because the SSE on the test 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). 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.

A better approach, which is much more time consuming, is to look for a model which is simpler and thus less prone to overfitting. For a FNN, a simpler model is one with fewer edges and/or nodes in the hidden layers. We fitted a FFNN as above, but with only 5 nodes in the hidden layer, and obtained 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.

**Exercises**

1. Show how you can replace the bias/threshold at a node by an extra node with fixed value and suitable edge weight.
2. Find weights so that the neural net below reproduces the XOR function:

   \[
   \begin{array}{c|c|c}
   x_1 & x_2 & y \\
   \hline
   0 & 0 & 0 \\
   0 & 1 & 1 \\
   1 & 0 & 1 \\
   1 & 1 & 0 \\
   \end{array}
   \]

   Use stepwise/threshold activation functions for each (non-input) node.

3. Find weights so that the neural net below “best” reproduces the function \( y = a + bx \) for \( x \) in the range \([0, 1]\). Use stepwise/threshold activation functions for the hidden layer and a linear activation function for the output node.