1 Some history

In the 1980s, the field of neural networks became fashionable again, after being out of favor during the 1970s. One reason for the renewed excitement was the paper by Rumelhart, Hinton, and McClelland, which made the backpropagation algorithm famous. The algorithm had actually been discovered and rediscovered several times before, as early as in the 1960s. But only in the 1980s was the time right for the algorithm, perhaps because computers had finally become fast enough to implement it for interesting problems.

2 Multilayer perceptrons

Although the backpropagation algorithm can be used very generally to train neural networks, it is most famous for applications to layered feedforward networks, or multilayer perceptrons.

We saw earlier that simple perceptrons are very limited in their representational capabilities. For example, they cannot represent the XOR function. However, it is easy to see that XOR can be represented by a multilayer perceptron. This is because the XOR can be written in terms of the basic functions AND, OR, and NOT, all of which can be represented by a simple perceptron.

In between the input layer and the output layer are the hidden layers of the network. We will consider multilayer perceptrons with $L$ layers of synaptic connections and $L + 1$ layers of neurons. This is sometimes called an $L$-layer network, and sometimes an $L + 1$-layer network.

A network with a single layer can approximate any function, if the hidden layer is large enough. This has been proved by a number of people, generally using the Stone-Weierstrass theorem. So multilayer perceptrons are representationally powerful.

3 The algorithm

Let’s diagram the network as

$$ x^0 \xrightarrow{W^1, b^1} x^1 \xrightarrow{W^2, b^2} \ldots x^L $$

(1)
where \( x^l \in \mathbb{R}^{n_l} \) for all \( l = 0, \ldots, L \) and \( W^l \) is an \( n_l \times n_{l-1} \) matrix for all \( l = 1, \ldots, L \). There are \( L + 1 \) layers of neurons, and \( L \) layers of synaptic weights.

We are provided with pairs of inputs \( x^0 \) and desired outputs \( d \), and would like to change the weights \( W \) so that the actual output \( x^L \) becomes closer to \( d \).

1. Forward pass

\[
x^l_i = f(\hat{x}^l_i) = f \left( \sum_{j=1}^{n_{l-1}} W^l_{ij} x^{l-1}_j + b^l_i \right)
\]

2. Error computation

\[
\delta^L_i = f'(\hat{x}^L_i)(d_i - x^L_i)
\]

3. Backward pass (error propagation)

\[
\delta^{l-1}_j = f'(\hat{x}^{l-1}_j) \sum_{i=1}^{n_l} \delta^l_i W^l_{ij}
\]

4. Learning updates

\[
\Delta W^l_{ij} = \eta \delta^l_i x^{l-1}_j \\
\Delta b^l_i = \eta \delta^l_i
\]

The backprop algorithm has a number of interesting features

1. The forward and backward passes use the same weights, but in the opposite direction

\[
x^{l-1}_j \xrightarrow{W^l_{ij}} x^l_i \quad \delta^{l-1}_j \xleftarrow{W^l_{ij}} \delta^l_i
\]

2. The update for a synapse depends on variables at the neurons to which it is attached. In other words, the learning rules are local.

3. As we will see later, the backprop algorithm is gradient descent on the squared error cost function between the desired and actual outputs. In general, it takes \( \mathcal{O}(N) \) operations to compute the value of the cost function, where \( N \) be the number of synaptic weights. Naively, it should take \( \mathcal{O}(N^2) \) operations to compute the \( N \) components of the gradient. In fact, the backprop algorithm finds the gradient in \( \mathcal{O}(N) \) steps, which is much shorter.
4 Derivation of the backprop algorithm

Here we will show that the backprop algorithm is gradient descent on the cost function

\[ E = \frac{1}{2} \sum_{i=1}^{n_L} (d_i - x^{L-1}_i)^2 \]  

We need to compute the gradient of \( E \) with respect to \( W \) and \( b \). The technical difficulty is that the dependence on \( W \) and \( b \) is implicit, buried inside \( x^L \). The standard way of dealing with this difficulty is to apply the chain rule to the equations of the forward pass, which describe the dependence of the output layer \( x^L \) on the input \( x^0 \). This derivation can be found in all the textbooks, so I will not repeat it here.

A more powerful method is to use Lagrange multipliers. For convenience, let’s change variables from \( x^l_i \) to \( \hat{x}^l_i = f^{-1}(x^l_i) \). Then the equations of the forward pass are

\[ \hat{x}^l_i = \sum_{j=1}^{n_{l-1}} W^l_{ij} f(\hat{x}^{l-1}_j) + b^l_i \]  

Now define a Lagrangian

\[ \mathcal{L}(\hat{x}^l_i, \delta^l_i) = \frac{1}{2} \sum_{i=1}^{n_L} (d_i - f(\hat{x}^L_i))^2 + \sum_{l=1}^{L} \left( \sum_{i=1}^{n_{l-1}} \delta^l_i \left[ \hat{x}^l_i - \sum_{j} W^l_{ij} f(\hat{x}^{l-1}_j) - b^l_i \right] \right) \]  

Suppose that we are at a stationary point of the Lagrangian. Then all of its partial derivatives must vanish. In particular,

\[ \frac{\partial \mathcal{L}}{\partial \delta^l_i} = \hat{x}^l_i - \sum_{j} W^l_{ij} f(\hat{x}^{l-1}_j) - b^l_i \]

must vanish, which means that the equations of the forward pass are satisfied, and furthermore that \( \mathcal{L} = E \). Therefore, minimizing the value of \( \mathcal{L} \) at a stationary point is equivalent to minimizing \( E \) subject to the constraint that the \( x^l_i \) satisfy the equations of the forward pass.

To be at a stationary point of the Lagrangian, the two other partial derivatives

\[ \frac{\partial \mathcal{L}}{\partial x^{l-1}_j} = -f'(\hat{x}^l_j)[d_i - f(\hat{x}^L_i)] + \delta^l_i \]  

\[ \frac{\partial \mathcal{L}}{\partial x^{l-1}_j} = \delta^l_i - f'(\hat{x}^{l-1}_j) \sum_{i=1}^{n_l} \delta^l_i W^l_{ij} \]

must also vanish. Setting these to zero yields the error computation and the backward pass. Therefore, the backprop algorithm sets \( x \) and \( y \) at a stationary point of \( \mathcal{L} \).

Therefore we can compute the gradients of \( E \) as

\[ \frac{\partial E}{\partial W^l_{ij}} = \frac{\partial \mathcal{L}}{\partial W^l_{ij}} = -\delta^l_i f(\hat{x}^{l-1}_j) \]  

\[ \frac{\partial E}{\partial b^l_i} = \frac{\partial \mathcal{L}}{\partial b^l_i} = -\delta^l_i \]
Only the explicit dependence of $\mathcal{L}$ on $W$ and $b$ matters in the gradient computation. The implicit dependence doesn’t matter because we are at a stationary point.