

# Multilayer perceptrons and backpropagation learning

Sebastian Seung

9.641 Lecture 4: September 17, 2002

## 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 did the algorithm take off. This happened because computers had finally become fast enough to train networks to solve 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. I'll generally follow the first convention.

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

$$x^0 \xrightarrow{W^1, b^1} x^1 \xrightarrow{W^2, b^2} \dots \xrightarrow{W^L, b^L} x^L \quad (1)$$

where  $x^l \in R^{n_l}$  for all  $l = 0, \dots, L$  and  $W^l$  is an  $n_l \times n_{l-1}$  matrix for all  $l = 1, \dots, L$ . There are  $L+1$  layers of neurons, and  $L$  layers of synaptic weights. We'd like to change the weights  $W$  and biases  $b$  so that the actual output  $x^L$  becomes closer to the desired output  $d$ .

The backprop algorithm consists of the following steps.

1. Forward pass. The input vector  $x^0$  is transformed into the output vector  $x^L$ , by evaluating the equation

$$x_i^l = f(u_i^l) = f\left(\sum_{j=1}^{n_{l-1}} W_{ij}^l x_j^{l-1} + b_i^l\right)$$

for  $l = 1$  to  $L$ .

2. Error computation. The difference between the desired output  $d$  and actual output  $x^L$  is computed.

$$\delta_i^L = f'(u_i^L)(d_i - x_i^L) \quad (2)$$

3. Backward pass. The error signal at the output units is propagated backwards through the entire network, by evaluating

$$\delta_j^{l-1} = f'(u_j^{l-1}) \sum_{i=1}^{n_l} \delta_i^l W_{ij}^l \quad (3)$$

from  $l = L$  to 1.

4. Learning updates. The synaptic weights and biases are updated using the results of the forward and backward passes,

$$\Delta W_{ij}^l = \eta \delta_i^l x_j^{l-1} \quad (4)$$

$$\Delta b_i^l = \eta \delta_i^l \quad (5)$$

These are evaluated for  $l = 1$  to  $L$ . The order of evaluation doesn't matter.

Later I'll show that this is gradient descent on a cost function, but first let's see an application of backprop.

## 4 The significance of hidden layers

This is a demonstration of backprop training of a multilayer perceptron to perform image classification on the MNIST database. At the beginning of the training, the output neurons are activated equally, more or less. As the training proceeds further, often one output neuron is clearly activated more than the others, and corresponds correctly with the class of the input image.

This two-layer perceptron does image classification in two stages. The input vector  $x^0$  is transformed into another vector  $x^1$ , which in turns is transformed into the output vector  $x^2$ . The ten perceptrons in the output layer do not look at the image directly, but rather at  $x^1$ , which can be regarded as an encoding of the information in the image.

The intermediate layer  $x^1$  is said to consist of hidden neurons. These contrast with the input and output neurons, whose values are set by the data in the training set. The weight vectors of the hidden neurons are depicted as images by the MATLAB simulation. It is difficult to interpret what they mean.

Backprop is an example of management-by-objective. We simply require that the network reduce the cost function. The network is free to choose whatever method will accomplish this. It is not “micromanaged.” In particular, the network has the freedom to choose what representation it uses in the hidden layer

How does one decide how many neurons to put in the hidden layer? This is a problem in model selection, and will be discussed later.

## 5 Backprop as gradient descent

Let’s define the cost function

$$E(W, b) = \frac{1}{2} \sum_{i=1}^{n_L} (d_i - x_i^L)^2 \quad (6)$$

where  $x^L$  is a function of  $W$  and  $b$  arises through the equations of the forward pass. This cost function measures the squared error between the desired and actual output vectors.

We’re going to prove that backprop is gradient descent on this cost function. In other words, the backprop weight updates are equivalent to

$$\Delta W_{ij}^l = -\eta \frac{\partial E}{\partial W_{ij}^l} \quad (7)$$

$$\Delta b_i^l = -\eta \frac{\partial E}{\partial b_i^l} \quad (8)$$

## 6 Properties of the algorithm

The backprop algorithm has a number of interesting features

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

$$x_j^{l-1} \xrightarrow{W_{ij}^l} x_i^l \quad (9)$$

$$\delta_j^{l-1} \xleftarrow{W_{ij}^l} \delta_i^l \quad (10)$$

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, once the forward and backward passes are complete.
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.

## 7 Derivation with the chain rule

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. You'll get to reproduce it in the homework assignment.

What is the meaning of the quantity  $\delta_i^l$ ? It is the sensitivity of the cost function to changes in the bias of neuron  $i$  in layer  $l$ .

$$\delta_i^l = -\frac{dE}{db_i^l} \quad (11)$$

In gradient learning for a single-layer perceptron, the weight update is the product of presynaptic activity, and an error term that is proportional to the difference between the desired and actual outputs. In gradient learning for a multilayer perceptron, no desired output for the hidden neurons is available. But the backpropagated error serves as a proxy. What replaces the error term is the sensitivity of the cost function to input to the postsynaptic neuron.

## 8 Derivation with Lagrange multipliers

Another method is to use Lagrange multipliers. This method is closely related to dynamic programming and optical control.

We'll need to define the function  $\phi(y) = f'(f^{-1}(y))$ . This is the slope of  $f$ , considered as a function of the neural output. Equivalently, we can write  $\phi(f(x)) =$

$f'(x)$ , or  $\phi(y) = 1/f^{-1'}(y)$ . This can be illustrated with the logistic function  $f(x) = 1/(1 + \exp(-x))$ . The inverse of this function is  $f^{-1}(y) = \log y - \log(1 - y)$ . Therefore  $\phi(y) = y(1 - y)$ .

The equations of the forward pass can be written as

$$f^{-1}(x_i^l) = \sum_{j=1}^{n_{l-1}} W_{ij}^l x_j^{l-1} + b_i^l \quad (12)$$

Now define a Hamiltonian

$$H(x, \delta, W, b) = \frac{1}{2} \sum_{i=1}^{n_L} (d_i - x_i^L)^2 + \sum_{l=1}^L \sum_{i=1}^{n_{l-1}} \delta_i^l \left[ f^{-1}(x_i^l) - \sum_j W_{ij}^l x_j^{l-1} - b_i^l \right] \quad (13)$$

Suppose that the Hamiltonian is stationary with respect to  $x$  and  $y$ , meaning that the derivatives with respect to  $x$  and  $y$  vanish. Then

$$\frac{\partial H}{\partial \delta_i^l} = f^{-1}(x_i^l) - \sum_j W_{ij}^l x_j^{l-1} - b_i^l$$

vanishes. This implies that the equations of the forward pass are satisfied, and furthermore that  $H = E$ .

$$E(W, b) = \text{stat}_{x,y} H(x, y, W, b)$$

Therefore, minimizing  $H$  with respect to  $W$  and  $b$  at a stationary point with respect to  $x$  and  $y$  is equivalent to minimizing  $E$  with respect to  $W$  and  $b$  subject to the constraint that the  $x_i^l$  satisfy the equations of the forward pass.

The other requirement for a stationary point is that the partial derivatives

$$\frac{\partial H}{\partial x_i^L} = -(d_i - x_i^L) + \frac{\delta_i^L}{\phi(x_i^L)} \quad (14)$$

$$\frac{\partial H}{\partial x_j^{l-1}} = \frac{\delta_j^{l-1}}{\phi(x_j^{l-1})} - \sum_{i=1}^{n_l} \delta_i^l W_{ij}^l \quad (15)$$

must also vanish. Setting these to zero yields the error computation and the backward pass. Therefore, the backward pass is a way of solving the equation  $\partial H / \partial x = 0$ .

We can quantify the change in  $E$  by looking at the change in  $H$

$$dE = \frac{\partial H}{\partial x} dx + \frac{\partial H}{\partial y} dy + \frac{\partial H}{\partial W} dW + \frac{\partial H}{\partial b} db$$

In this expression we assume that the changes  $dx$  and  $dy$  are slaved to  $dW$  and  $db$ , since both  $x$  and  $y$  are both implicitly functions of  $W$  and  $b$ . By our cunning, we have defined  $H$  so that the first two terms vanish, and we are left with

$$dE = \frac{\partial H}{\partial W} dW + \frac{\partial H}{\partial b} db$$

Therefore we can compute the gradients of  $E$  as

$$\frac{\partial E}{\partial W_{ij}^l} = \frac{\partial H}{\partial W_{ij}^l} = -\delta_i^l x_j^{l-1} \quad (16)$$

$$\frac{\partial E}{\partial b_i^l} = \frac{\partial H}{\partial b_i^l} = -\delta_i^l \quad (17)$$

Only the explicit dependence of  $H$  on  $W$  and  $b$  matters in the gradient computation. The implicit dependence doesn't matter because we are at a stationary point.