**Javier F. Troncoso**

## Back-propagation algorithm

The Back-propagation algorithm is used to train neural networks and determine model parameters. For simplification, we will consider a simple neural network with 1 hidden layer with 5 nodes, but we will see that the generalization is straightforward. This neural network is described by the diagram shown in Fig. 1.

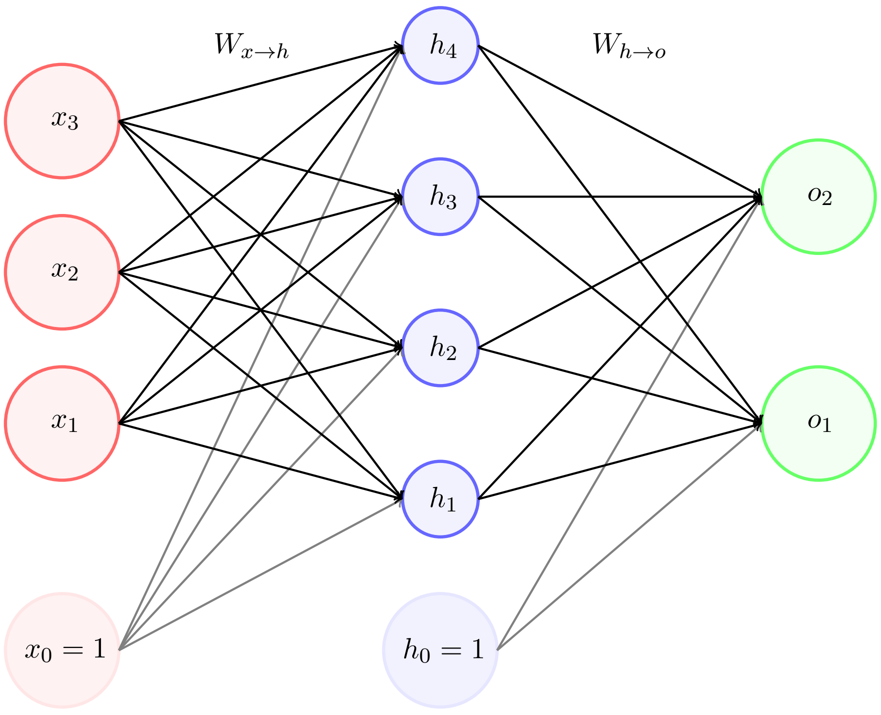

Figure 1: Neural network formed by one input layer with 3 nodes $(x_1, x_2, x_3)$, one hidden layer with 4 nodes $(h_1, h_2, h_3, h_4)$ and an output layer with 2 nodes $(o_1, o_2)$. $x_0$ and $h_0$ are biases.

Let's define the variables in use:

- $x_i$ with $i = 0, 1, 2, 3$, where $x_0 = 0$ corresponds to the bias. $x_1, x_2$ and $x_3$ stand for the input features. Dimension of the vector $\vec{x} \equiv (x_0, x_1, x_2, x_3)^\intercal$: $[3 + 1, 1] = [4, 1]$.
- $h_i$ with $i = 0, 1, 2, 3, 4$, where $h_0 = 0$ corresponds to the bias. $h_1, h_2, h_3$ and $h_4$ stand for the

neurons in the hidden layers. Dimension of the vector $\vec{h} \equiv (h_0, h_1, h_2, h_3, h_4)^\intercal$: $[4+1, 1] = [5, 1]$.

- $o_i$ with $i = 1, 2$ stand for the output features. Dimension of the vector $\vec{o} \equiv (o_1, o_2)^\intercal$: $[2, 1]$.
- $t_i$ with $i = 1, 2$ stand for the target values - desired outputs. Dimension: $[2, 1]$.
- $W_{x\to h}$ is the matrix to calculate the output in $h_i$. Dimension: $[4, 4]$.
- $W_{h\to o}$ is the matrix to calculate the output. Dimension: $[2, 5]$.

## Forward pass

In a simple Neural Network, the final output is given by the application of a non-linear function to a linear combination of the elements in the previous layer. We will consider that the function applied to the elements of the hidden layer is $F_h$ and the function applied to the output is $F_o$. Common functions are the sigmoid function or the hyperbolic tangent function, which return small values and hence it's more difficult to obtain high weight parameters. Thus, the outputs are given by the following expressions:

$$\vec{h} = F_h(W_{x\to h}\vec{x}), \tag{1}$$

$$\vec{o} = F_o(W_{h\to o}\vec{h}). \tag{2}$$

Note that the dimensionality is mantained in these expressions, since $[4(+1), 1] = [4, 4] \cdot [4, 1]$ in Eq. 1 and $[2, 1] = [2, 5] \cdot [5, 1]$ in Eq. 2, where $(+1)$ means that the bias is added a posteriori.

## Backward pass

The model parameters $W$ are calculated using the Back-propagation algorithm together with the gradient descent method. These model parameters are obtained such that the total error defined as

$$E = \frac{1}{2}\sum_i (t_i - o_i)^2 \tag{3}$$

is minimized. According to the gradient descent method, the model parameters will evolve in the direction of their negative gradient until convergence:

$$W^{i+1} = W^i - \alpha \frac{\partial E}{\partial W}, \tag{4}$$

where the subindices $i+1$ and $i$ correspond to the model parameter after and before the iteration, and $\alpha$ is the learning rate, which determines how fast the convergence is reached. Its value has to be small enough to reach a local minimum. Since the iteration method depends on the derivatives of the total energy, we will see that these terms can be easily obtained by moving backwards. Thus, applying the chain rule, we have that:

$$\frac{\partial E}{\partial W_{h\to o}} = \frac{\partial E}{\partial \vec{o}}\frac{\partial \vec{o}}{\partial W_{h\to o}}. \tag{5}$$

$$\frac{\partial E}{\partial o_1} = \frac{\partial\left[\frac{1}{2}\sum_i (t_i - o_i)^2\right]}{\partial o_1} = -(t_1 - o_1) = o_1 - t_1, \tag{6}$$

$$\frac{\partial E}{\partial o_2} = \frac{\partial\left[\frac{1}{2}\sum_i (t_i - o_i)^2\right]}{\partial o_2} = o_2 - t_2, \tag{7}$$

$$\frac{\partial \vec{o}}{\partial W_{h\to o}} = \frac{\partial\left[F_o(W_{h\to o}\vec{h})\right]}{\partial W_{h\to o}} = F_o'(W_{h\to o}\vec{h})\vec{h}. \tag{8}$$

$$\boxed{\frac{\partial E}{\partial W_{h\to o}} = \left[\begin{pmatrix} o_1 - t_1 \\ o_2 - t_2 \end{pmatrix} \circ F_o'(W_{h\to o}\vec{h})\right]\begin{pmatrix} h_1 & h_2 & h_3 & h_4 \end{pmatrix} = \left[(\vec{o} - \vec{t}) \circ F_o'(W_{h\to o}\vec{h})\right]\vec{h}^\intercal \equiv \delta_o \vec{h}^\intercal} \tag{9}$$

where $\circ$ is the Hadamard product [1] and $\delta_o = \left[(o - t) \circ F_o'(W_{h\to o}\vec{h})\right]$.

Similarly:

$$\frac{\partial E}{\partial W_{x\to h}} = \frac{\partial E}{\partial \vec{h}}\frac{\partial \vec{h}}{\partial W_{x\to h}}. \tag{11}$$

$$\frac{\partial E_t}{\partial \vec{h}} = \frac{\partial\left[\frac{1}{2}\sum_i (t_i - o_i)^2\right]}{\partial \vec{h}} = \frac{\partial\left[\frac{1}{2}\sum_i (t_i - o_i)^2\right]}{\partial \vec{o}}\frac{\partial \vec{o}}{\partial \vec{h}} = (\vec{o} - \vec{t})\frac{\partial \vec{o}}{\partial \vec{h}}, \tag{12}$$

$$\frac{\partial \vec{o}}{\partial \vec{h}} = \frac{\partial\left[F_o\left(W_{h\to o}\vec{h}\right)\right]}{\partial \vec{h}} = W_{h\to o}^\intercal F_o'\left(W_{h\to o}\vec{h},\right) \tag{13}$$

$$\frac{\partial \vec{h}}{\partial W_{x\to h}} = \frac{\partial\left[F_h(W_{x\to h}\vec{x})\right]}{\partial W_{\to h}} = F_h'(W_{x\to h}\vec{x})\vec{x}. \tag{14}$$

$$\boxed{\frac{\partial E_t}{\partial W_{xh}} = W_{h\to o}^\intercal\left[(\vec{o} - \vec{t}) \circ F_o'(W_{h\to o}\vec{h})\right] \circ F_h'(W_{x\to h}\vec{x})\vec{x}^\intercal = \left[W_{h\to o}^\intercal \delta_o \circ F_h'(W_{x\to h}\vec{x})\right]\vec{x}^\intercal \equiv \delta_h \vec{x}^\intercal} \tag{15}$$

where $\delta_h = \left[W_{h\to o}^\intercal \delta_o \circ F_h'(W_{x\to h}\vec{x})\right]$.

[1] Hadamard product between matrices $A$ and $A$ with the same dimensions:

$$(A \circ B)_{i,j} = A_{i,j}B_{i,j} \tag{10}$$

Then, summarizing, the model parameters are solved iteratively until convergence following the rule:

$$W^{i+1}_{v_j \to v_{j+1}} = W^{i}_{v_j \to v_{j+1}} - \alpha \frac{\partial E}{\partial W_{v_j \to v_{j+1}}}, \tag{16}$$

where $v_j$ is the layer number $j$ and $j$ runs from 1 (input layer) to 2 (last hidden layer) and:

$$\frac{\partial E}{\partial W_{v_j \to v_{j+1}}} = \delta_{j+1} \vec{v}_j. \tag{17}$$

If $\vec{v}_{j+1}$ is the output layer, then:

$$\delta_{j+1} = \delta_o = \left[ (\vec{o} - \vec{t}) \circ F'_o(W_{v_j \to v_{j+1}} v_j) \right]. \tag{18}$$

Otherwise,

$$\delta_{j+1} = \delta_h = \left[ W^{\intercal}_{v_{j+1} \to v_{j+2}} \delta_{\delta j+2} \circ F'_h(W_{v_j \to v_{j+1}} v_j) \right]. \tag{19}$$

The use of biases requires the reduction in the dimensionality of matrices or vectors to be coherent with the way they were introduced.

## Numerical Optimization Techniques

### Gradient descent optimization

The simplest approach is to choose the weight update in to comprise a small step in the direction of the negative gradient, so that:

$$W^{i+1} = W^{i} - \alpha \frac{\partial E}{\partial W}, \tag{20}$$

where the parameter $\alpha > 0$ is known as the learning rate. After each such update, the gradient is re-evaluated for the new weight vector and the process is repeated. Note that the error function is defined with respect to a training set, and so each step requires the entire training set to be processed. Techniques that use the whole data set at once are called batch methods. At each step the weight vector is moved in the direction of the greatest rate of decrease of the error function, and so this approach is known as gradient descent or steepest descent.

For batch optimization, there are more efficient methods, such as conjugate gradients and quasi-Newton methods, which are much more robust and much faster than simple gradient descent (Gill et al., 1981; Fletcher, 1987; Nocedal and Wright, 1999). Unlike the gradient descent method, these algorithms have the property that the error function always decreases at each iteration unless the weight vector has arrived at a local or global minimum. The basic back-propagation algorithm adjusts the weights in the steepest descent direction (negative of the gradient). This is the direction in which the performance decreases most rapidly. It turns out that, although the function decreases most rapidly along with the negative of the gradient, this does not necessarily produce the fastest convergence. The conjugate gradient method can be regarded as something intermediate between the gradient descent method and Newton's method [2]. In conjugate gradient algorithms, a search is performed along with conjugate directions, which produces generally faster convergence than steepest descent directions. In order to find a sufficiently good minimum, it may be necessary to run a gradient-based algorithm multiple times, each time using a different randomly chosen starting point, and comparing the resulting performance on an independent validation set. There is, however, an on-line version of gradient descent that has proved useful in practice for training neural networks on large data sets (Le Cun et al., 1989). Error functions based on the maximum likelihood for a set of independent observations can be an interesting alternative.

## Backpropagation with momentum

When the minimum of the error function for a given learning task lies in a narrow valley, the gradient direction can lead to wide oscillations of the search process. The best strategy, in this case, is to orient the search towards the centre of the valley, but the form of the error function is such that the gradient does not point in this direction. A simple solution is to use a momentum term:

$$W^{i+1} = W^i - \alpha \frac{\partial E}{\partial W} + \beta \Delta W, \tag{22}$$

where the parameter $\beta$ is known as the momentum rate.

## Quasi-Newton method

In the Quasi-Newton method, the model parameters are updated according to the following expression:

$$W^{i+1} = W^i + \eta \frac{\nabla W}{[H^{i+1}]}, \tag{23}$$

where $H^{i+1}$ is the Hessian matrix. The main difficulty with this approach is that finding the solution to this system at every iteration is very tedious.

## Lavenberg-Marquardt backpropagation algorithm

This algorithm was designed to approach second-order training speed without having to compute the Hessian matrix:

$$H = J^T J, \tag{24}$$

$$g = J^T e, \tag{25}$$

[2]Newton's method is based on Taylor's approximation to first order. If the function $f(x)$ is a polynomial, the algorithm can be applied iteratively as follows:

$$xi + 1 = x^i - \frac{f(x)}{f'(x)}, \tag{21}$$

where $J$ is a Jacobian matrix, which contains first order derivatives of the network errors with respect to the weights and biases, and $e$ is a vector of network errors. The Jacobian matrix can be computed through a standard backpropagation technique that is much less complex than computing the Hessian matrix. This algorithm uses this approximation to the Hessian matrix in the following a Newton-like update connection weights:

$$W^{i+1} = W^{i} + [J^{T}J + \beta I]^{-1} J^{\intercal} e. \tag{26}$$

When the scalar $\beta$ is zero, this is just Newton‘s method, using the approximate Hessian matrix. When $\beta$ is large, this becomes the gradient descent with a small step size.

**Javier F. Troncoso**
javierfdeztroncoso@gmail.com