Learning algorithms – Back-propagation

0 Notation

The notes will use the following notation:

- i, j, k – Indices of different neurons in the network.
- n – Iteration of the training.
- $d_j(n)$ – Expected output of the j-th neuron in the output layer.
- $y_j(n)$ – Value of the j-th neuron in the output layer.
- $e_j(n)$ – Error of the value of the j-th neuron in the output layer.
- $\mathcal{E}(n)$ – Sum of square errors $e_j(n)$ over all neurons j in the output layer.
- η – Learning rate.

1 Back-propagation Algorithm

The error signal for output neuron j at the iteration n is defined as:

$$e_j(n) := d_j(n) - y_j(n), \tag{1}$$

or as the difference between the expected and the calculated value.

We define *instantaneous sum of squared errors* $\mathcal{E}(n)$ as:

$$\mathcal{E} := \frac{1}{2} \sum_{j \in C} e_j^2(n), \tag{2}$$

where C is the set of all output neurons.

If N is the number of examples in the training set, the *averaged square error* is obtained as:

$$\mathcal{E}_{\text{av}} := \frac{1}{N} \sum_{n=1}^{N} \mathcal{E}(N) \tag{3}$$

Our **objective** is to **minimize $\mathcal{E}_{\text{av}}$ by adjusting the network parameters**. How do we do it?

The basic principle is as follows. In the n-th training step, we choose a training sample randomly from our training data-set and evaluate the full network. The network will give some values $y_j(n)$ as output, which might be close or far from the expected values $d_j(n)$. To improve the accuracy of the network (decrease the errors $e_j(n)$ or the total error $\mathcal{E}(n)$), we calculate for each parameter how much it affects the final result and thereby the total error. Namely, we calculate $\frac{\partial \mathcal{E}(n)}{\partial w}$ for all weights w in all layers of the network. After that, the weights are changed in the favorable direction and magnitude, which is what the gradient $\frac{\partial \mathcal{E}(n)}{\partial w}$ tells us.

The reason the full training data-set is not used in each training step is that it could be too large to handle (e.g. couldn't fit into memory) and because the minimization might be too slow, as each time step we would need to evaluate

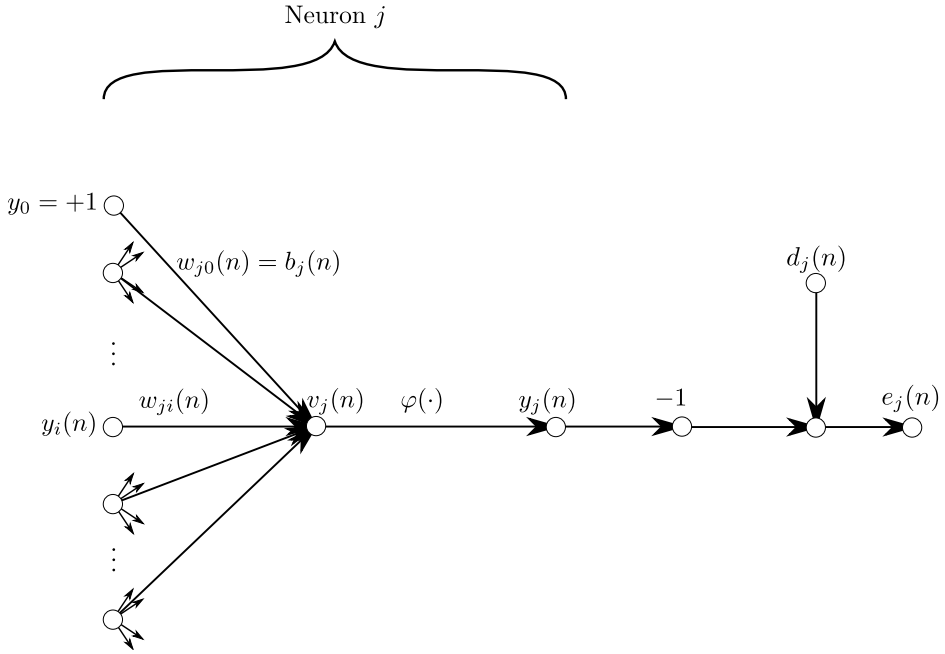

Figure 1: Signal-flow graph highlighting the details of the neuron j.

the network for each sample from the data-set. Therefore, the weight gradients described above (and therefore the final weights themselves) serve only as *estimates* of the correct gradients and weights that would be retrieved if the full data-set was used. In practice, the data-set is split into small batches, an estimate of the gradient is computed from each of the samples in a batch and the weights are updated according to the average gradient of the batch.

The beauty of this approach is that calculation of all derivatives $\frac{\partial \mathcal{E}(n)}{\partial w}$ can be done in the same complexity as the evaluation of the network itself, i.e. calculating $y_j(n)$ and $\frac{\partial \mathcal{E}(n)}{\partial w}$ is only twice as slow as calculating only $y_j(n)$. The process of calculating these derivatives is called *back-propagation*. The name comes from the inverse direction of the data flow. Namely, in the evaluation of the network, the "information" flows from the input side towards the output, while in the back-propagation step the information flows from the output towards the input. This is a simple consequence of the chain rule for the derivatives of the composition of functions. We will derive the procedure in details mathematically.

Consider the output neuron j, as shown in the Figure 1. The results $y_i(n)$ of the previous layer are passed to the neuron j, summed using the matrix $w_{ji}(n)$ to form the results $v_j(n)$. These are passed through the non-linear function φ to get the final output value $y_j(n)$ of the neuron j. Please note that in this notation, j and i refer to different layers. The value $y_j(n)$ is then subtracted from the expected value $d_j(n)$ to calculate the error $e_j(n)$. We are interested in the gradients $\frac{\partial \mathcal{E}(n)}{\partial w_{ji}(n)}$. We use the chain rule to do the calculation. The iteration

(n) is omitted for brevity:

$$\begin{aligned}
\frac{\partial \mathcal{E}}{\partial w_{ji}} &= \frac{\partial \mathcal{E}}{\partial e_j}\frac{\partial e_j}{\partial w_{ji}} \\
&= \frac{\partial \mathcal{E}}{\partial e_j}\frac{\partial e_j}{\partial y_j}\frac{\partial y_j}{\partial w_{ji}} \\
&= \frac{\partial \mathcal{E}}{\partial e_j}\frac{\partial e_j}{\partial y_j}\frac{\partial y_j}{\partial v_j}\frac{\partial v_j}{\partial w_{ji}} \\
&= \left[\frac{\partial}{\partial e_j}\frac{1}{2}e_j^2\right]\left[\frac{\partial}{\partial y_j}(d_j - y_j)\right]\left[\varphi'(v_j)\right]\left[\frac{\partial}{\partial w_{ji}}\sum_{i=0}^{p} w_{ji}y_i\right] \\
&= -e_j\varphi'(v_j)y_i \\
&= -e_j(n)\varphi'(v_j(n))y_i(n)
\end{aligned} \tag{4}$$

If updating the weights from single gradient estimates, the correction $\Delta w_{ji}(n)$ is calculated by:

$$\Delta w_{ji}(n) = -\eta\frac{\partial \mathcal{E}(n)}{\partial w_{ji}(n)}, \tag{5}$$

where η is a constant that determines the rate of learning, called *learning rate*. If the learning rate is too small, the convergence will be too slow, if it's too high, the weights will not converge.

Correction $\Delta w_{ji}(n)$ can be written as:

$$\Delta w_{ji}(n) = \eta\delta_j(n)y_i(n), \tag{6}$$

where $\delta_j(n)$ is the *local gradient* defined as:

$$\begin{aligned}
\delta_j(n) &:= \frac{\partial \mathcal{E}(n)}{\partial v_j(n)} \\
&= e_j(n)\varphi_j'(v_j(n)).
\end{aligned} \tag{7}$$

The previous derivation of weight corrections is done assuming j is an output neuron so that the error signal is directly computed from Eq. (1) as the desired output d_j is given.

Same derivation for a network with hidden layers requires knowledge of how to penalize hidden neurons for their share of responsibility in determining the output. This problem is referred to as the *credit-assignment problem* which is decomposed into two subproblems

1. Temporal credit assignment: involves instants of time when the actions that deserve credit were actually taken;
2. Structural credit assignment: involves assigning credit to the internal structures of actions generated by the system.

For a hidden neuron j, the weight corrections need to be determined recursively in terms of the error signals from output neurons as illustrated in Fig. 2 where the hidden neuron j is connected to output neurons k. The local gradient is redefined as

$$\begin{aligned}
\delta_j(n) &= -\frac{\partial \mathcal{E}(n)}{\partial y_j(n)}\frac{\partial y_j(n)}{\partial v_j(n)} \\
&= -\frac{\partial \mathcal{E}(n)}{\partial y_j(n)}\varphi_j'(v_j(n))
\end{aligned} \tag{8}$$

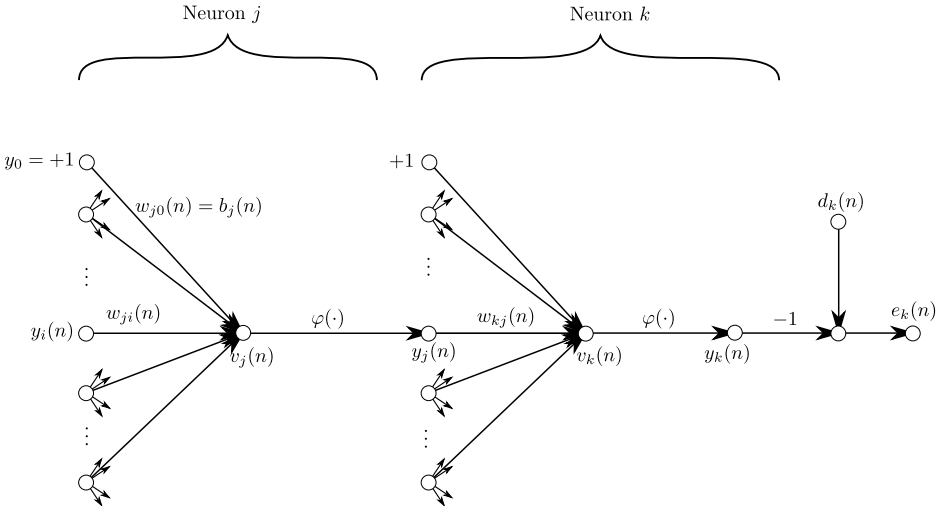

Figure 2: Connections between hidden j and output neurons k.

and the error is computed from output neurons

$$\mathcal{E}(n) = \frac{1}{2} \sum_{k \in C} e_k^2(n). \tag{9}$$

Differentiating $\mathcal{E}(n)$ with respect to $y_j(n)$, we get

$$\frac{\partial \mathcal{E}}{\partial y_j} = \sum_k e_k \frac{\partial e_k}{\partial y_j} = \sum_k e_k \frac{\partial e_k}{\partial v_k} \frac{\partial v_k}{\partial y_j} \tag{10}$$

where the error signal for output neurons is known directly from the desired output $d_k(n)$

$$e_k(n) = d_k(n) - y_k(n) = d_k(n) - \varphi_k(v_k(n)) \tag{11}$$

$$\frac{\partial e_k}{\partial v_k} = -\varphi_k'(v_k(n)) \tag{12}$$

with

$$v_k(n) = \sum_{j=0}^{q} w_{kj}(n) y_j(n), \tag{13}$$

where q is the total number of inputs applied to neuron k. Differentiating Eq. (13) with respect to $y_j(n)$ yields

$$\frac{\partial v_k(n)}{\partial y_j(n)} = w_{kj}(n). \tag{14}$$

Substitution of Eq. (12) and Eq. (14) in Eq. (10) yields the desired partial derivative of the instantaneous sum of squared errors

$$\begin{aligned} \frac{\partial \mathcal{E}(n)}{\partial y_j(n)} &= -\sum_k e_k(n) \varphi_k'(v_k(n)) w_{kj}(n) \\ &= -\sum_k \delta_k(n) w_{kj}(n), \end{aligned} \tag{15}$$

where the definition of the local gradient $\delta_k(n)$ from Eq. (7) has been used in the second line with index k substituted for j (neuron k is an output node). By substituting Eq. (15) into Eq. (8) we get the local gradient for hidden neuron j as follows

$$\delta_j(n) = \varphi_j'(v_j(n)) \sum_k \underbrace{\delta_k(n)}_{\text{I}} \underbrace{w_{kj}(n)}_{\text{II}}. \tag{16}$$

The factor $\varphi_j'(v_j(n))$ involved in the computation of the local gradient depends solely on the *activation function* associated with hidden neuron j. The factor given by the summation depends on two terms:

I: This term requires knowledge of the error signals $e_k(n)$ for all neurons that are in the layer to the immediate *right* of hidden neuron j and are directly connected to neuron j as shown in Fig. 2;

II: This term consists of the synaptic weights that are associated with these direct connections.

In summary, the relations derived for the back-propagation algorithm are as follows:

1. The correction $\Delta w_{ji}(n)$ applied to the synaptic weight connecting neuron i to neuron j is defined by the delta-rule

$$\underbrace{\Delta w_{ji}(n)}_{\substack{\text{Weight} \\ \text{correction}}} = \underbrace{\eta}_{\substack{\text{Learning-rate} \\ \text{parameter}}} \underbrace{\delta_j(n)}_{\substack{\text{Local} \\ \text{gradient}}} \underbrace{y_i(n)}_{\substack{\text{Input signal} \\ \text{of neuron } j}}$$

2. The local gradient $\delta_j(n)$ depends on whether neuron j is an output node or a hidden node:

   i If neuron j is an *output* node, $\delta_j(n)$ is computed by the product of the derivative $\varphi_j'(v_j(n))$ and the error signal $e_j(n)$, both terms are associated with neuron j.

   ii If neuron j is a *hidden* node, $\delta_j(n)$ is determined by the derivative $\varphi_j'(v_j(n))$ and the weighted sum of the local gradients computed for the neurons in the next hidden layer (to the right) or output layer that are connected to neuron j.

1.1 Application of the back-propagation algorithm

Two distinct passes may be considered for the application of the back-propagation algorithm

1. Forward pass to output
2. Backward pass to input

Forward pass

The synaptic weights of the network remain unchanged during the forward pass. Only function signals are computed on a neuron-by-neuron basis. The function signal at the output of neuron j is given by

$$y_j(n) = \varphi(v_j(n)), \tag{17}$$

where $v_j(n)$ is the net internal activity level of neuron j, given by

$$v_j(n) = \sum_{i=0}^{p} w_{ji}(n) y_i(n) \tag{18}$$

with p the total number of inputs applied to neuron j, $w_{ji}(n)$ the synaptic weight connecting neuron i to neuron j and $y_i(n)$ is the input of neuron j or output of neuron i, respectively. For the first hidden layer, the input is determined by the terminal network input, whereas the output of neuron j in the output layer corresponds to the j-th terminal output of the network. The error signal $e_j(n)$ is then obtained with Eq. (1), where $y_j(n)$ corresponds to the network output of neuron j. Therefore, the forward phase propagates from the network input to the network output (left to right in Fig. 2).

Backward pass

The second phase propagates in the opposite direction. The error signals $e_j(n)$ are passed as inputs to the output layer. The local gradients δ are then computed for each neuron recursively layer by layer. The recursive computation allows the synaptic weights to change, where the correction is given by the delta rule. The local gradient for a neuron j in the output layer is given by the product of the error signal $e_j(n)$ and the derivative of its nonlinearity $\varphi_j'(v_j(n))$. We then continue by using the delta rule to find a correction for the synaptic weights and apply Eq. (16) to compute the local gradient for the next hidden layer.

1.2 Activation function

The computation of the local gradients for neuron j requires the derivative $\varphi_j'(\cdot)$ of the activation function φ_j. An activation function that is commonly used in multilayer perceptrons is the *sigmoidal nonlinearity* given by

$$\begin{aligned} y_j(n) &= \varphi_j(v_j(n)) \\ &= \frac{1}{1 + e^{-v_j(n)}} \qquad -\infty < v_j(n) < \infty, \end{aligned} \tag{19}$$

where $v_j(n)$ is the net internal activity level of neuron j. With this choice, the output lies within $0 \leq y_j \leq 1$. The derivative $\varphi_j'(v_j(n))$ is found by

$$\begin{aligned} \frac{y_j(n)}{v_j(n)} &= \varphi_j'(v_j(n)) \\ &= \frac{e^{-v_j(n)}}{[1 + e^{-v_j(n)}]^2} \\ &= \frac{e^{-v_j(n)}}{1 + e^{-v_j(n)}} \frac{1}{1 + e^{-v_j(n)}} = [1 - y_j(n)] y_j(n). \end{aligned} \tag{20}$$

Note that the derivative $\varphi_j'(v_j(n))$ obeys its maximum for $y_j(n) = 0.5$ and its minimum value zero for $y_j(n) = 0$ or $y_j(n) = 1$. Thus, for the sigmoidal activation function, the synaptic weights are changed the most for neurons with values $y_j(n)$ close to 0.5.

1.3 Training modes

The learning in the back-propagation algorithm is achieved by presenting a set of prescribed training examples to the neural network. One complete presentation of a training set with N patterns is called an *epoch*. The learning process is maintained epoch-by-epoch until the synaptic weights stabilize and the average squared error $\mathcal{E}_{\text{av}}$ converges to a minimum value. It is good practice to randomize the order of the training samples presented to the network from one epoch to the next. There are two basic ways to perform the weight updates in back-propagation learning:

Pattern mode: In this mode, the weight update is performed after the presentation of a training sample. For each training set, the forward and backward passes are performed as described above. Let $\Delta w_{ji}(n)$ denote the correction to the synaptic weight w_{ji} after the presentation of training pattern n. The net weight change averaged over all patterns is given by

$$\begin{aligned} \Delta \hat{w}_{ji} &= \frac{1}{N} \sum_{n=1}^{N} \Delta w_{ji}(n) \\ &= -\frac{\eta}{N} \sum_{n=1}^{N} \frac{\partial \mathcal{E}(n)}{\partial w_{ji}(n)} \\ &= -\frac{\eta}{N} \sum_{n=1}^{N} e_j(n) \frac{\partial e_j(n)}{\partial w_{ji}(n)}, \end{aligned} \tag{21}$$

where Eq. (2) and (5) have been used.

Mini-batch mode: In mini-batch mode, the weights are updated after the presentation of a subset (batch) of B training samples. The limiting case of the mini-batch mode is batch learning, where each weight update is performed according to the average gradient of the entire dataset (i.e. $B = N$ the batch has the same size as the data set).

For this mode the cost function is defined by the average squared error:

$$\mathcal{E}_{\text{av}} = \frac{1}{2B} \sum_{n=1}^{B} \sum_{j \in C} e_j^2(n). \tag{22}$$

The correction for the synaptic weight Δw_{ji} is again defined by the delta rule

$$\begin{aligned} \Delta w_{ji} &= -\eta \frac{\partial \mathcal{E}_{\text{av}}}{\partial w_{ji}} \\ &= -\frac{\eta}{B} \sum_{n=1}^{B} e_j(n) \frac{\partial e_j(n)}{\partial w_{ji}}. \end{aligned} \tag{23}$$

The weight adjustment is then applied only after the entire mini-batch has been presented to the network.

The batch mode provides a more accurate estimate of the gradient vector, especially when we expect the training samples to be noisy, but it requires more computation per each weight update. The correction $\Delta\hat{w}_{ji}$ for the pattern mode is an estimation to Δw_{ji} of the batch mode.

Batch learning (estimating gradients from the whole dataset), is generally avoided as it tends to converge and be stuck (in weight space) in local minima. Provided that the order of the training patterns are randomized per epoch, pattern mode learning is the least likely to be stuck in a local minimum. However, the update might never converge to a fixed point unless the learning rate is extremely small. The mini-batch mode can be considered a compromise between these two modes as it provides accurate gradients from noisy data sets and can escape local minima.

Mini-batch mode is currently the standard way to train neural networks. In addition to the stated benefits, this mode has the computational benefit of being efficiently parallelizable, allowing faster training of large network models.