Universitat
Rovira i Virgili

Departament d'Enginyeria Informàtica i Matemàtiques

Neural and Evolutionary Computation

Sergio Gómez

Supervised learning using Back-Propagation

A successful implementation of Back-Propagation (BP) requires a set of techniques that must be taken into account before proceeding with the BP algorithm itself, so let us summarize the whole procedure.

Preprocessing

Data needs *preprocessing* before starting any other analysis: outliers' detection, treatment of missing data, transformation of categorical data into an appropriate numeric representation, etc. In graded activities A1 and A2, the data comes almost ready for the supervised learning phase, but there is one step missing for A1: the scaling of the data. If you look at the data for A1, each of the five columns has a completely different range of variation: height in [591, 630], fall in [56.7, 96.21], net fall in [55.14, 95.935], flow in [3.0, 9.0], and power in [1675.16, 7261.37]. Since we do not want to give a priori more importance to some of the input variables w.r.t. the others, we should scale all of them to the same range of variation. This can be done by a separate linear scaling of each input variable from its [min, max] range to [0.0, 1.0] (or [-1.0, 1.0]), or by standardization (subtracting the mean value and dividing by the standard deviation). This is not needed in A2, since the two input variables lay in the same range [-1.0, 1.0].

The scaling of the output variables has an additional requirement: since the output of a sigmoid lays in the range (0.0, 1.0), the desired output values must strictly fall within these limits. For predictions tasks (e.g. A1), where the output variable takes values in a certain [min, max] range, a convenient choice is its linear scaling to a range like [0.1, 0.9]. Otherwise, for classification problems (e.g. A2), it is better to represent the two classes as 0 and 1, respectively, due to the interpretation of the unconstrained minimum squared error as the conditional probabilities of being in each of the classes $P(class|input)$. Thus, no scaling of the output variable is needed for A2.

The equation for the linear scaling of a variable x in range $[x_{\text{min}}, x_{\text{max}}]$ to a new variable s in range $[s_{\text{min}}, s_{\text{max}}]$ is

$$s = s_{\text{min}} + \frac{s_{\text{max}} - s_{\text{min}}}{x_{\text{max}} - x_{\text{min}}}(x - x_{\text{min}})\,, \tag{1}$$

and you recover the original variables (descaling) using the equivalent equation:

$$x = x_{\min} + \frac{x_{\max} - x_{\min}}{s_{\max} - s_{\min}} (s - s_{\min}) \,. \tag{2}$$

Standardization requires the calculation of the mean value $\langle x \rangle$ and the standard deviation σ_x of the values x, and defining the new variables s as

$$s = \frac{x - \langle x \rangle}{\sigma_x} \,. \tag{3}$$

This relationship is inverted using

$$x = \langle x \rangle + s \, \sigma_x \,. \tag{4}$$

Summarizing, you must first scale (or standardize) separately each input variable of your data, scale your desired output variables, and use these scaled patterns to train the neural network. Then, when the training is finished, you can make predictions over new patterns in three steps: scale the pattern, introduce it into the neural network, and descale the obtained output from the network.

Cross-validation: training, validation and test datasets

In both A1 and A2, the test data is clearly specified, so you must not use it during the training of the network. Supposing you have 1000 training patterns, a 4-fold cross-validation would divide (randomly) this set in four subsets (S_1, S_2, S_3 and S_4), each one containing 250 patterns. Then, cross-validation consists in repeating the training process four times, fixing the rest of the parameters (architecture, learning rate, momentum, number of epochs, etc.):

- Training with $S_2 \cup S_3 \cup S_4$, validation with $S1 \longrightarrow$ validation error E_1
- Training with $S_3 \cup S_4 \cup S_1$, validation with $S2 \longrightarrow$ validation error E_2
- Training with $S_4 \cup S_1 \cup S_2$, validation with $S3 \longrightarrow$ validation error E_3
- Training with $S_1 \cup S_2 \cup S_3$, validation with $S4 \longrightarrow$ validation error E_4

The global expected prediction error E is obtained as $(E_1 + E_2 + E_3 + E_4)/4$. You can repeat this process for different values of the parameters (e.g. modifying the architecture, learning rate, etc.), thus choosing the parameters which yield the lowest expected prediction error. Once you have found the best set of parameters, you fix them and perform a final training using the complete training set ($S_1 \cup S_2 \cup S_3 \cup S_4$). Finally, you use your final neural network to predict over the test set, and evaluate the test error as described in A1 and A2.

Online Back-Propagation algorithm

The main objective of BP consists in minimizing the quadratic error $E[\boldsymbol{o}]$ over the training set. Given a training set $\{(\boldsymbol{x}^\mu, \boldsymbol{z}^\mu) \in \mathbb{R}^n \times \mathbb{R}^m, \mu = 1, \dots, p\}$, this quadratic error reads

$$E[\boldsymbol{o}] \equiv \frac{1}{2} \sum_{\mu=1}^{p} \sum_{i=1}^{m} \left(o_i(\boldsymbol{x}^\mu) - z_i^\mu \right)^2 . \tag{5}$$

where $\boldsymbol{o}(\boldsymbol{x})$ represents the neural network as a function that assigns a value in $\mathbb{R}^m$ for each input $\boldsymbol{x} \in \mathbb{R}^n$. This function depends on the architecture of the neural network, whose information is fully encoded in its weights $\{\omega_{ij}^{(\ell)}\}$ and thresholds $\{\theta_i^{(\ell)}\}$.

The minimization of Eq. 5 cannot be done by just taking the derivatives with respect to the weights and thresholds and equating them to zero, since it is impossible to solve the resulting system of equations. Thus, the best option is the application of the method of *steepest descent*, which consists in finding the direction of fastest decrease of the quadratic error, make a small movement in that direction, and repeat these two steps until the error no longer improves. The direction of fastest decrease is given by the minus gradient, $-\boldsymbol{\nabla} E$, which can efficiently be computed for multilayer networks, using BP.

The pseudo-code for the online (or incremental) BP algorithm is as follows:

```
Scale input and/or output patterns, if needed
Initialize all weights and thresholds randomly
For epoch = 1 To num_epochs
    For pat = 1 To num_training_patterns
        Choose a random pattern ($\boldsymbol{x}^\mu$, $\boldsymbol{z}^\mu$) of the training set
        Feed−forward propagation of pattern $\boldsymbol{x}^\mu$ to obtain the output $\boldsymbol{o}(\boldsymbol{x}^\mu)$
        Back−propagate the error for this pattern
        Update the weights and thresholds
    End For
    Feed−forward all training patterns and calculate their prediction quadratic error
    Feed−forward all validation patterns and calculate their prediction quadratic error
End For
# Optional: Plot the evolution of the training and validation errors
Feed−forward all test patterns
Descale the predictions of test patterns, and evaluate them
```

Listing 1: Online BP algorithm

Let us now explain in detail how to calculate each of these steps.

Feed-forward propagation

The feed-forward propagation of an input pattern $\boldsymbol{x}$ consists in introducing this pattern in the input layer of the network,

$$\boldsymbol{\xi}^{(1)} = \boldsymbol{x}\,, \tag{6}$$

and then calculate (and store) the activations of the neurons in the second layer, then the third layer, and so on, until you reach the output layer. Supposing a multilayer neural network with $n_1, n_2, \ldots, n_L$ units in the respective L layers, the equations for the calculation of these activations $\xi_i^{(\ell)}$ are:

$$\xi_i^{(\ell)} = g(h_i^{(\ell)})\,,\ i = 1, \ldots, n_\ell\,,\ \ell = 2, \ldots, L\,, \tag{7}$$

where the fields $h_i^{(\ell)}$ are given by

$$h_i^{(\ell)} = \sum_{j=1}^{n_{\ell-1}} \omega_{ij}^{(\ell)} \xi_j^{(\ell-1)} - \theta_i^{(\ell)}\,. \tag{8}$$

The output $\boldsymbol{o}(\boldsymbol{x})$ is just given by the activation of the units in the output layer:

$$\boldsymbol{o}(\boldsymbol{x}) = \boldsymbol{\xi}^{(L)}\,. \tag{9}$$

Note that the activations $\boldsymbol{\xi}^{(\ell)}$ at layer ℓ depend (through the fields) on the activations $\boldsymbol{\xi}^{(\ell-1)}$ at layer $\ell - 1$, thus they have to be computed in the right order: $\ell = 2, 3, \ldots, L$.

The most commonly used activation function g is the sigmoid,

$$g(h) = \frac{1}{1 + e^{-h}}\,, \tag{10}$$

but other options are also possible.

Error back-propagation

Once we have found the prediction $\boldsymbol{o}(\boldsymbol{x})$ corresponding to our input pattern $\boldsymbol{x}$, we can compare it to the desired output $\boldsymbol{z}$, and calculate the contribution of this pair to the minus gradient. The best option consists in the previous computation of a set of auxiliary variables $\Delta_i^{(\ell)}$, one for each unit in the network (except for units in the input layer). First, we compute their values in the output layer,

$$\Delta_i^{(L)} = g'(h_i^{(L)})\ (o_i(\boldsymbol{x}) - z_i)\ , \tag{11}$$

and then we back-propagate them to the rest of the network using

$$\Delta_j^{(\ell-1)} = g'(h_j^{(\ell-1)}) \sum_{i=1}^{n_\ell} \Delta_i^{(\ell)} \omega_{ij}^{(\ell)}\,. \tag{12}$$

In this case, we must start with the last layer, and then proceed with the rest of the layers in reversed order, i.e. $\ell = L, L-1, L-2, \ldots, 2$, hence the name of *back-propagation*. Note that, if the prediction coincides with the desired output, all the $\Delta_i^{(\ell)} = 0$, thus no modification of weights and thresholds is necessary.

In Eqs. (11) and (12) we need the derivative g' of the activation function g. For the sigmoidal activation function in Eq. (10), we may use the following property:

$$g'(h) = g(h)\ (1 - g(h))\ . \tag{13}$$

Update of weights and thresholds

Using the activation $\xi_i^{(\ell)}$ and the $\Delta_i^{(\ell)}$ variables of the units, we can now proceed to calculate the modification of all the weights and thresholds that must be applied in order to decrease the error between the desired output $\boldsymbol{z}$ of our input pattern $\boldsymbol{x}$, and the predicted output from the neural network $\boldsymbol{o}(\boldsymbol{x})$. The result is:

$$\left\{ \begin{array}{rcl} \delta\omega_{ij}^{(\ell)} & = & -\eta \Delta_i^{(\ell)}\, \xi_j^{(\ell-1)} + \alpha\, \delta\omega_{ij}^{(\ell)}{}_{(\text{prev})}\,, \\ \\ \delta\theta_i^{(\ell)} & = & \eta \Delta_i^{(\ell)} + \alpha\, \delta\theta_i^{(\ell)}{}_{(\text{prev})}\,, \end{array} \right. \tag{14}$$

where $\delta\omega_{ij}^{(\ell)}$ and $\delta\theta_i^{(\ell)}$ represent the changes to be applied to the corresponding weights and thresholds, and $\delta\omega_{ij}^{(\ell)}{}_{(\text{prev})}$ and $\theta_i^{(\ell)}{}_{(\text{prev})}$ are the changes we applied in the previous step. Two important parameters of BP appear in Eq. (14): the *learning rate* (η), and the *momentum* (α). The learning rate defines the length of the movement in the direction of the minus gradient, and thus it must be a relatively small constant. Large values of the learning rate are useful for a fast learning in the initial epochs of the BP, but are not adequate for the fine-tuning of predictions in the last epochs. On the contrary, a tiny learning rate may produce the opposite effects, very slow learning but high precision. The momentum term introduces a kind of inertia in the training, which is very necessary for online BP, in which consecutive training patterns may point to very different directions in the weights and thresholds space. With the momentum term activated, the training is more smooth and robust to the randomness in the selection of the training patterns. Unfortunately, the optimal values of the learning rate and momentum parameters depend on the training set and the architecture of the multilayer neural network, thus you should try different values and decide upon the obtained results. As a general guide, the learning rate could be between 0.2 and 0.01, and the momentum between 0.0 (no momentum at all) and 0.9.

Finally, we update all the weights and thresholds:

$$\left\{ \begin{array}{rcl} \omega_{ij}^{(\ell)} & \longrightarrow & \omega_{ij}^{(\ell)} + \delta\omega_{ij}^{(\ell)}\,, \\ \\ \theta_i^{(\ell)} & \longrightarrow & \theta_i^{(\ell)} + \delta\theta_i^{(\ell)}\,. \end{array} \right. \tag{15}$$

Partial batched Back-Propagation

In the online version of BP explained so far, we do not calculate the minus gradient for the whole quadratic error in Eq. 5, but only the contribution from just one pattern: the randomly selected pattern in line 5 of Listing 1. This seems strange, since what we want to optimize is the total quadratic error, which is the sum over all patterns in the training set. The correct procedure, known as *batched* BP, should have been to calculate the contributions from all the patterns, sum them all, and afterwards update the weights and thresholds. However, this turns out to be a very slow training of the network, being online BP much more efficient. An intermediate and more effective option consists in choosing a small subset of patterns $\mathcal{P}$ (e.g. five random patterns), make their feed-forward propagation, and substitute Eq. (14) by

$$\left\{ \begin{array}{rcl} \delta\omega_{ij}^{(\ell)} & = & -\eta \sum_{\mu\in\mathcal{P}} \Delta_i^{(\ell)\mu}\, \xi_j^{(\ell-1)\mu} + \alpha\, \delta\omega_{ij}^{(\ell)}{}_{\text{(prev)}}\,, \\ \\ \delta\theta_i^{(\ell)} & = & \eta \sum_{\mu\in\mathcal{P}} \Delta_i^{(\ell)\mu} + \alpha\, \delta\theta_i^{(\ell)}{}_{\text{(prev)}}\,. \end{array} \right. \tag{16}$$

In this way, we combine in one algorithm the speed of online BP and the consistency of a full batched BP.

Evolution of the quadratic error during training

It is convenient to plot the evolution of the training and validation errors during the training, since they provide useful information. A plot of the train quadratic error with respect to the epoch should show a tendency to go down, indicating the network is really learning the training patterns. The decrease in error is not monotonous, showing random oscillations, but the overall tendency to reduce the error should be clearly evident; otherwise, it would be indicating a problem with the training. The same plot for the validation quadratic error should also go down, but it can go up in the last epochs if the training is too large, due to overtraining. Therefore, these plots are useful to check if the training is performing correctly, the speed of the training to achieve low values of the quadratic error, and to decide how many epochs to select for the training, avoiding overtraining.