Feed Forward Neural Networks
and how to train them

Contents

1 Feed Forward Neural Network

Recall the basic model for regression and classification tasks :

$$y(w, x) = f\left(\sum_i w_i \phi_i(x)\right) \tag{1}$$

where f is called the activation function.
With neural networks, our goal is to extend this model to basis function that can be parametrizable. Each basis function is itself a non-linear function of a linear combination of the inputs, where the combination weights are adaptable parameters.

1.1 The basic feed-forward neural network

Such considerations lead to the most basic neural network model. Let us define the following :

$$\forall j \in \{1, \dots, M\}, \quad a_j = w_{j0}^{(1)} + \sum_{i=1}^{D} w_{ji}^{(1)} x_i \tag{2}$$

The scalars $(a_j)_{j\in\{1,\dots,M\}}$ are called activations units. They are transformed using a *differentiable*, usually non-linear activation function :

$$\forall j \in \{1, \dots, M\}, \quad z_j = h(a_j) \tag{3}$$

The activation function $h(.)$ is usually chosen to a sigmoid function such as the logistic sigmoid or the $\tanh(.)$ function.

In the context of neural networks, the $(z_j)_j$ correspond to the output of the initial basis functions are called the *output of the hidden units*. They are therefore linearly mixed to produce the output unit activations :

$$\forall k \in \{1, \dots, K\}, \quad a_k = w_{k0}^{(2)} + \sum_{j=1}^{M} w_{kj}^{(2)} z_j \tag{4}$$

The network's output is then computed b applying an output activation function :

$$\forall k \in \{1, \dots, K\}, \quad y_k = h(a_k) \tag{5}$$

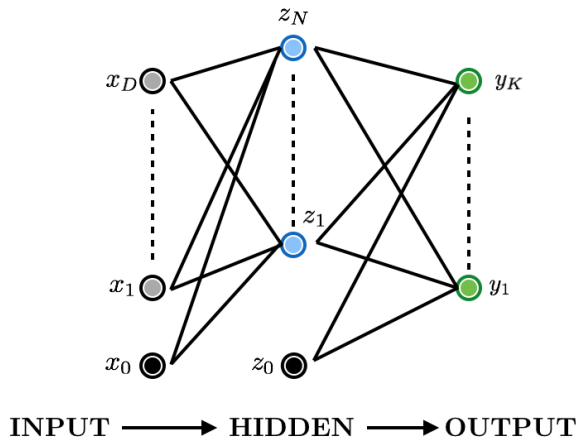

Figure 1: Topology of a neural network with one hidden layer

The process we just described to compute both the hidden layer activations, the hidden layer output, the output unit activations and the networks output is called **feedforward propagation**. It is the basis of feed-forward net, which topology can be summed un by figure (1).

Of course, one can decide to stack as many hidden layer as he wants. The resulting neural net is therefore called a *deep* neural net, performing *deep learning*.

1.2 Activation function

Activation functions for the hidden and output layers highly depend on the task that the neural net is performing. For standard regression problems, linear activation or rectified linear activation functions are preferred. For binary classification, the logistic function is the most common activation function :

$$\sigma(a) = \frac{1}{1 + e^{-a}} \tag{6}$$

For multi-class problems, the softmax function is the most reasonable choice :

$$\text{softmax}(z) = \left(\frac{e^{z_j}}{\sum_k e^{z_k}} \right)_j \tag{7}$$

1.3 The network's output

Assuming that the output activation function is a logistic sigmoid, the output takes the form of a non-linear function of the inputs :

$$\forall k \in \{1, \dots, K\}, \quad y_k = \sigma \left(\sum_{j=1}^{M} w_{kj}^{(2)} h \left(\sum_{i=1}^{D} w_{ji}^{(1)} + w_{j0}^{(1)} \right) + w_{k0}^{(2)} \right) \tag{8}$$

which can be simplified by adding a dimension to the input to absorb the bias into a mock dimension :

$$\forall k \in \{1, \dots, K\}, \quad y_k = \sigma \left(\sum_{j=0}^{M} w_{kj}^{(2)} h \left(\sum_{i=0}^{D} w_{ji}^{(1)} x_i \right) \right) \tag{9}$$

The approximation abilities of feed-forward networks have been widely studied. Feed-forward artificial neural networks are said to be *general approximators*. Indeed, it was shown that a two-layer network with linear output can uniformly approximate any continuous functions on a compact input domain to arbitrary accuracy, given that the hidden layer has a sufficient large number of units. The remaining questions is how to tune the parameters (i.e the weights) in order to achieve any given accuracy for a function approximation ?

2 Network training

We consider a dataset $\mathcal{X} = \{x_1, \dots, x_n\} \in \left(\mathbb{R}^D\right)^N$, with a corresponding set of target vectors $\mathcal{T} = \{t_1, \dots, t_N\}$.

2.1 Error function

As with other machine learning algorithm, we are going to train our network by minimizing an error function that represent how well the network approximate the underlying function sampled at the dataset's points.

2.1.1 Regression task

For regression tasks, a commonly used error function is the *sum of square* error function :

$$E(w, \mathcal{X}, \mathcal{T}) = \sum_{n=1}^{N} \|y(x_n, w) - t_n\|^2 \tag{10}$$

As with many different model, this error function can be motivated by a probabilistic approach. Let us assume that the real-valued target are Gaussian distributed around an x-dependent mean. Therefore $\forall n \in \{1, \dots, N\}$:

$$p(t_n \,|\, \mathcal{X}, w) = t_n \overset{d}{\sim} \mathcal{N}\left(t_n \,\middle|\, y(x_n, w), \beta^{-1}\right) \tag{11}$$

where β is known as the distribution's *precision*. Therefore assuming independence between the drawn samples, we have that :

$$p(\mathcal{T} \,|\, \mathcal{X}, w) = \prod_{n=1}^{N} t_n \overset{d}{\sim} \mathcal{N}\left(t_n \,\middle|\, y(x_n, w), \beta^{-1}\right) \tag{12}$$

and

$$-\log\left[p(\mathcal{T} \,|\, \mathcal{X}, w)\right] = cste + \frac{\beta}{2} \sum_{n=1}^{N} \{y(x_n, w) - t_n\}^2 \tag{13}$$

Therefore a *maximum-likelihood* solution, which will try to maximize the likelihood of the dataset, is given by minimizing the sum-of-square error function. This approach is adapted to regression problem, but can lead to slow learning in case of sigmoid activation function. For binary classification tasks (where the common approach is logistic regression), there exists a more natural error function, which is not the victim of slow learning far from correctness.

2.1.2 Binary classification task

We now consider the case of binary classification. Let's consider a logistic sigmoid output activation function. Therefore, the network output can be interpreted as the condition probability $p\left(\mathcal{C}_1|x\right)$. The conditional distribution of the targets given the inputs is therefore a Bernouilli distribution of the form :

$$p(t \,|\, x, w) = y(x, w)^t \{1 - y(w, x)\}^{1-t} \tag{14}$$

The derivation of the error function is performed as in the regression task. We obtain the cross-entropy function :

$$E(w, \mathcal{X}, \mathcal{T}) = -\sum_{n=1}^{N} \left[t_n \log y_n + (1 - t_n) \log\left(1 - y_n\right)\right] \tag{15}$$

The real benefit of this error function is that it avoid slow learning that can appear with the sum-of-square error function (computes the derivatives with respect to w when there is misclassification ..).

The same reflexion with the standard multi-class problem (Multinouilli distribution) leads to the following error function :

$$E(w, \mathcal{X}, \mathcal{T}) = -\sum_{n=1}^{N} \sum_{k=1}^{K} t_{nk} \log y_k(x_n, w) \tag{16}$$

2.2 Network optimization

Of course, for complex neural network, analytical solving of the optimization problem :

$$\max_{w} E(w, \mathcal{X}, \mathcal{T}) \tag{17}$$

is untractable. We therefore need to use sequential optimization algorithms (steepest descent, Newton or quasi-Newtons methods).

The offline optimization can be quite costly for larget dataset. Indeed, if one decide to run a finite difference gradient optimization, he must run $2 * W^2$ feed-forward operations !.) Fortunately, there exist an on-line version of gradient descent that is very helpful for large datasets. Since :

$$E(w) = \sum_{n=1}^{N} E_n(w) \tag{18}$$

one can follow the update rule given by :

$$w^{(\tau+1)} = w^{(\tau)} - \eta \nabla E_n(w^{(\tau)}) \tag{19}$$

This update rule is known as the stochastic gradient descent. It is repeated by cycling through the date (usually with replacement). One can also decide to pack datapoints into small group and then perform optimization on this small part of the data, which is called *mini-batch* gradient descent. Such methods display better handling of data redundancy and can escape from local minima the batch method would get trapped in (a local minima for the whole dataset is not mandatorily a local minima for one datapoint).

3 The backpropagation algorithm

Now that we now how to train our network using variational informations, we need to compute such variations (gradient, Jacobian, Hessian). If all of those can of course be computed via finite difference, those methods are numerically needy. We hereinafter describe the *back-propagation* algorithm, which introduce an efficient way to compute gradient informations.

We consider a neural network having arbitrary feed-forward topology, arbitrary differentiable activations functions.

For the sake of the example, we consider an error function that can be written as a sum of terms errors, one for each datapoint :

$$E(w, x) = \sum_{n} E_n(w) \tag{20}$$

We'll therefore focus on computing $\nabla E_n(w)$.

Let us first consider the simplest linear model

$$y_k = \sum_{i} w_{ki} x_i \tag{21}$$

along with the error function :

$$E_n(w, \mathcal{X}, \mathcal{T}) = \frac{1}{2} \sum_{n} \{y_n - t_n\}^2 \tag{22}$$

Therefore we'll have that :

$$\frac{\partial E_n}{\partial w_{ji}} = (y_{nj} - t_{nj}) x_{ni} \tag{23}$$

and a similar formule arises for a sigmoidal activation function.

Let us now consider computing $\frac{\partial E_n}{\partial w_{ji}}$ for a link from the hidden layer to the output activation layer. By applying the cain rule :

$$\begin{aligned}\frac{\partial E_n}{\partial w_{ji}} &= \frac{\partial E_n}{\partial a_j}\frac{\partial a_j}{\partial w_{ji}} \\ &= \delta_j \frac{\partial a_j}{\partial w_{ji}} \text{ with } \delta_j = \frac{\partial E_n}{\partial a_j} \\ &= \delta_j z_i\end{aligned} \tag{24}$$

As we saw before, we have $\delta_k = y_k - t_k$ for a canonical output activation function.
To evaluate δ's for all units, we make use of the chain rule again :

$$\delta_j = \sum_k \frac{\partial E_n}{\partial a_k}\frac{\partial a_k}{\partial a_j} \tag{25}$$

for all k unit to which j sends a connection. Therefore we obtain the **backprop formula** :

Backprop formula

$$\delta_j = h'(a_j)\left(\sum_k \delta_k w_{kj}\right) \tag{26}$$

The full backpropagation algorithm is given below :

The Error Backpropagation Algorithm

1. Apply a datapoint x_n to the network and forward propagate the input, to compute the activations of the hidden and output units.
2. Evaluate the δ_k's for the output layer unit.
3. Backpropagate the δ_k's using the backprop formula :

$$\delta_j = h'(a_j)\left(\sum_k \delta_k w_{kj}\right) \tag{27}$$

4. Compute $\frac{\partial E}{\partial w_{ji}}$ using :

$$\frac{\partial E}{\partial w_{ji}} = \delta_j h(a_i) \tag{28}$$

For batch-methods, the total error derivatives is given by summing over the dataset :

$$\nabla E(w) = \sum_n \nabla E_n(w) \tag{29}$$

The efficiency of the backprop algorithm is $O(W)$ against $O(W^2)$ for finite differences, hence justifying its use especially for large neural networks.