Artificial neural networks

Artificial neural networks (ANNs) are imitations of biological neural networks, like our brains. They can be very adept at approximating nonlinear functions, even when only few training data is available. How they do this, and how they can be trained, will be examined in this chapter.

1 The structure of a neural network

1.1 The neuron

The basic building block of an ANN is a **neuron**. A neuron has several inputs x_i. Each of these inputs is multiplied by a weight w_i and then added up. Often, a bias b is added as well. The result is the neuron's **activation** z. So,

$$z = \sum_{i=1}^{p} w_i x_i = \mathbf{w}^T \mathbf{x} \qquad \text{or} \qquad z = \sum_{i=1}^{p} w_i x_i + b = \begin{bmatrix} \mathbf{w}^T & b \end{bmatrix} \begin{bmatrix} \mathbf{x} \\ 1 \end{bmatrix}. \tag{1.1}$$

From the above equation, it can be seen that adding a bias b works the same as adding an additional input with weight b and value 1. So, we will simply use $z = \mathbf{w}^T \mathbf{x}$ in the remainer of this chapter.

Once the neuron's activation z has been obtained, it is fed into the **activation function** $\sigma(z)$. This function returns a value on the interval $[-1, 1]$ (or alternatively sometimes on the interval $[0, 1]$). Which activation function is used depends on the ANN designer's choice. Common activation functions are the **threshold function** and the **sigmoidal function**, respectively defined as

$$\sigma(z) = \begin{cases} 0 & \text{for } z < 0 \\ 1 & \text{for } z \geq 0 \end{cases} \qquad \text{and} \qquad \sigma(z) = \frac{1}{1 + \exp(-sz)}. \tag{1.2}$$

The parameter s determines how 'steep' the sigmoid function is. If $s \to \infty$, then the threshold function is again obtained. But often $s = 1$ is simply chosen. The output of the activation is then the output of the neuron.

1.2 Neural network architecture

An artificial neural network consists of interconnected neurons. (That is, the output of one neuron is fed as input to the next neuron.) The neurons are usually assembled in layers. In a **feedforward network**, the neurons of every layer are connected to the next layers. On the other hand, in **recurrent networks**, neurons are also connected to previous layers as some sort of 'feedback mechanism'. We will mainly consider feedforward networks though, because they are relatively simple.

ANNs always have an **input layer** (at the start) and an **output layer** (at the end). Often, there are also **hidden layers** in between. Most of the times, only one hidden layer is used. The reason is that multiple hidden layers make the neural network computationally quite complex. Also, one hidden layer is already capable of approximating any continuous function. That is, as long as there is a sufficient number of hidden neurons in it.

Choosing the right number of hidden neurons in the hidden layer is very important, but also very difficult. If you use too few neurons, then the neural network can't approximate the desired output function well enough. If, however, you use too many neurons, then overtraining can occur: the system only works on the few test samples that have been provided, but is useless for any other input. Generally, the number of hidden neurons primarily depends on the number of training samples (more training samples implies that more neurons can be used) and the complexity of the output function (more complex output functions often require more neurons).

1.3 Finding the output of a neural network

Let's suppose that we have an ANN with 1 hidden layer. Given an input $\mathbf{x_i}$, how do we find the output $\mathbf{y_i}$ of this network?.

Well, we start with the input $\mathbf{x_i}$. The input layer doesn't really do anything with this input. It only passes it on to every neuron of the hidden layer. The activation of one hidden neuron j can then be found using $z_j = \mathbf{w_j^h}^T\mathbf{x}$. But we usually have multiple (N) input samples $\mathbf{x_i}$ and multiple (p) hidden neurons j. So, we can define

$$\mathbf{X} = \begin{bmatrix} \mathbf{x_1}^T \\ \mathbf{x_2}^T \\ \vdots \\ \mathbf{x_N}^T \end{bmatrix}, \qquad \mathbf{W^h} = \begin{bmatrix} \mathbf{w_1^h}^T \\ \mathbf{w_2^h}^T \\ \vdots \\ \mathbf{w_p^h}^T \end{bmatrix} \quad \text{and} \quad \mathbf{Z} = \begin{bmatrix} z_{11} & z_{12} & \cdots & z_{1p} \\ z_{21} & z_{22} & \cdots & z_{2p} \\ \vdots & \vdots & \ddots & \vdots \\ z_{N1} & z_{N2} & \cdots & z_{Np} \end{bmatrix}. \tag{1.3}$$

The element z_{ij} of $\mathbf{Z}$ thus denotes the activation of hidden neuron j to input sample $\mathbf{x_i}$. Now we can simply find the hidden layer activation $\mathbf{Z}$, the hidden layer output $\mathbf{V}$ and the system output $\mathbf{Y}$ using

$$\mathbf{Z} = \mathbf{X}\mathbf{W^h}, \qquad \mathbf{V} = \sigma(\mathbf{Z}) \quad \text{and} \quad \mathbf{Y} = \sigma(\mathbf{V}\mathbf{W^o}). \tag{1.4}$$

Here, we have $\mathbf{Y} = \begin{bmatrix} \mathbf{y_1} & \mathbf{y_2} & \cdots & \mathbf{y_N} \end{bmatrix}^T$. Also, the output layer weights $\mathbf{W^o}$ are defined similarly as the hidden layer weights $\mathbf{W^h}$.

2 Training neural networks

Before ANNs work, they need to be trained. That is, their weights (and biases) need to be set such that certain inputs give certain outputs. So, let's suppose that we have a set of inputs $\mathbf{X}$ with **desired outputs D**. How do we find the right weights? Several techniques exist. One of the simplest is the **backpropagation technique**. Let's examine it.

2.1 Backpropagation – the output layer

First, we randomly initialize the neural network. We then take an input $\mathbf{x}$ and find the resulting output $\mathbf{y}$. We compare this with the desired output $\mathbf{d}$ and calculate the error $\mathbf{e} = \mathbf{d} - \mathbf{y}$. Our goal now is to minimize the cost function

$$J = \frac{1}{2}\sum_l e_l^2. \tag{2.1}$$

First, let's focus on minimizing the contribution of the output layer. For simplicity, we assume that the output layer has no activation function. So, we simply have $y_l = \sum_j w_{jl}^o v_j$. We now adjust the weights of the output layer using the update rule

$$w_{jl}^o(n+1) = w_{jl}^o(n) - \alpha(n)\frac{\partial J}{\partial w_{jl}^o}. \tag{2.2}$$

Here, $\alpha(n)$ is the **learning rate**. To find the **Jacobian** $\partial J/\partial w_{jl}^o$, we can use the chain rule. So,

$$\frac{\partial J}{\partial w_{jl}^o} = \frac{\partial J}{\partial e_l}\frac{\partial e_l}{\partial y_l}\frac{\partial y_l}{\partial w_{jl}^o}. \tag{2.3}$$

These three partial derivatives are all relatively easy to find. We have

$$\frac{\partial J}{\partial e_l} = e_l, \qquad \frac{\partial e_l}{\partial y_l} = -1 \qquad \text{and} \qquad \frac{\partial y_l}{\partial w_{jl}^o} = v_j. \tag{2.4}$$

Thus, the update rule for the output layer weights becomes

$$w_{jl}^o(n+1) = w_{jl}^o(n) + \alpha(n) v_j e_l. \tag{2.5}$$

2.2 Backpropagation – the hidden layer

A similar principle is applied when adjusting the weights for the hidden layer. But now the Jacobian is given by

$$\frac{\partial J}{\partial w_{ij}^h} = \frac{\partial J}{\partial v_j} \frac{\partial v_j}{\partial z_j} \frac{\partial z_j}{\partial w_{ij}^h}. \tag{2.6}$$

Finding the partial derivatives now is a bit more difficult. But, after some computation, we can find that

$$\frac{\partial J}{\partial v_j} = \sum_l \left(\frac{\partial J}{\partial e_l} \frac{\partial e_l}{\partial y_l} \frac{\partial y_l}{\partial v_j} \right) = \sum_l -e_l w_{jl}^o, \qquad \frac{\partial v_j}{\partial z_j} = \sigma_j'(z_j) \qquad \text{and} \qquad \frac{\partial z_j}{\partial w_{ij}^h} = x_i. \tag{2.7}$$

These data result in the update law for hidden neuron weights, being

$$w_{ij}^h(n+1) = w_{ij}^h(n) + \alpha(n) x_i \sigma_j'(z_j) \sum_l e_l w_{jl}^o. \tag{2.8}$$

By using this equation, the weights of the hidden layer are adjusted.

When using the backpropagation technique, you usually use a set of N test samples $(\mathbf{x_i}, \mathbf{d_i})$ to adjust the weights. The presentation of the whole training set to the system is called an **epoch**. Multiple epochs are necessary before the backpropagation algorithm converges to a minimum. However, since backpropagation is a gradient descent method, it is quite likely that the resulting minimum is a local minimum. This is a significant downside of the backpropagation method.

2.3 The radial basis function network

An other type of neural network is the **radial basis function network** (RBFN). This network has a hidden layer. However, the neurons in this hidden layer don't have weights. Also, there is no real activation function. Instead, a **radial basis function** (RBF) $\phi_i(r)$ is used. A common RBF is the **Gaussian function**

$$\phi_i(r) = \exp\left(-\frac{r^2}{\rho_i^2} \right), \qquad \text{where } r = ||\mathbf{x} - \mathbf{c_i}||. \tag{2.9}$$

The output layer does have weights, but it does not have an activation function. The output of an output node is thus given by

$$y_j = \sum_{i=1}^{n} w_{ij} \phi_i \left(||\mathbf{x} - \mathbf{c_i}|| \right). \tag{2.10}$$

If we put the outputs of the RBFs in a row vector $\mathbf{V} = [\phi_1(r) \ \ldots \ \phi_n(r)]$, then the output equation reduces to $\mathbf{y} = \mathbf{V}\mathbf{w}$. This is a linear equation. So, if we have a set of known inputs $\mathbf{x}$ with desired outputs $\mathbf{d}$, then we can use the least-squares theorem to find $\mathbf{w}$. To do this, we simply have to find $\mathbf{V}$ and apply

$$\mathbf{w} = \left(\mathbf{V}^T \mathbf{V} \right)^{-1} \mathbf{V}^T \mathbf{d}. \tag{2.11}$$

In this way, the weights of the network can be trained. However, training the values of $\mathbf{c_i}$ and ρ_i is not possible in this way, since the output $\mathbf{y}$ doesn't linearly depend on these parameters. Instead, more complicated nonlinear optimization methods need to be applied.