Chapter 11
Neural Networks

11.1 The Problem

Rule-based systems and Bayesian networks cannot effectively solve problems such as image or speech recognition. Artificial neural networks (ANNs), or simply neural networks, are effective in solving complex problems, i.e., in modeling complex nonlinear functions. ANNs model the functioning of the brain's neurons; ANN can be trained to "learn" how to recognize patterns and classify data [1].

11.2 A Practical Example

11.2.1 Example 1

Let us take an example of a dataset that has four instances with two variables, x and y, and two classes (i.e., class "grey" and class "black"), which are drawn in Fig. 11.1. We can notice two groups of instances: those in black and those in grey. But there is no way that one straight line can classify these instances into two classes/categories. If we have two lines like those present in Fig. 11.2, we can correctly classify the instances. So, the function that separates these two classes cannot be linear; we therefore have a nonlinear solution to this classification problem (Fig. 11.2). Every time a linear classification cannot work, we can make use of an artificial neural network (ANN), or more accurately, an ANN with hidden layers.

To make our point clear, we can draw two straight lines to separate the two classes (Fig. 11.3).

Each line is expressed as $y = ax + b$, or to write it slightly differently, $y - ax - b = 0$, which is equivalent to

C. El Morr et al., *Machine Learning for Practical Decision Making*, International Series in Operations Research & Management Science 334,
https://doi.org/10.1007/978-3-031-16990-8_11

Attribute 1	Attribute 2	Class
1.30	2.50	1.00
1.50	2.50	1.00
1.92	9.00	2.00
2.20	8.50	2.00
3.70	2.70	2.00
3.74	2.30	2.00
4.00	5.50	1.00
4.40	5.00	1.00

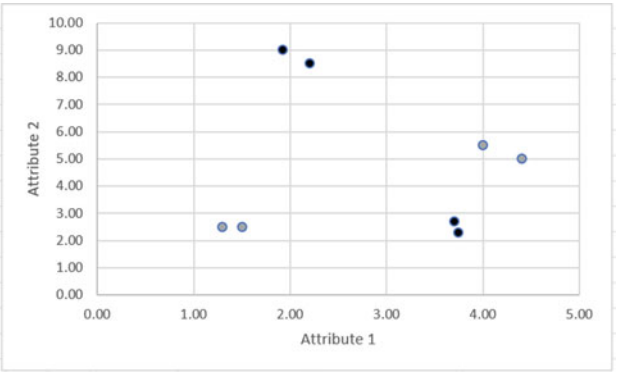

Fig. 11.1 Eight instances belonging to two classes represented by black dots and grey dots

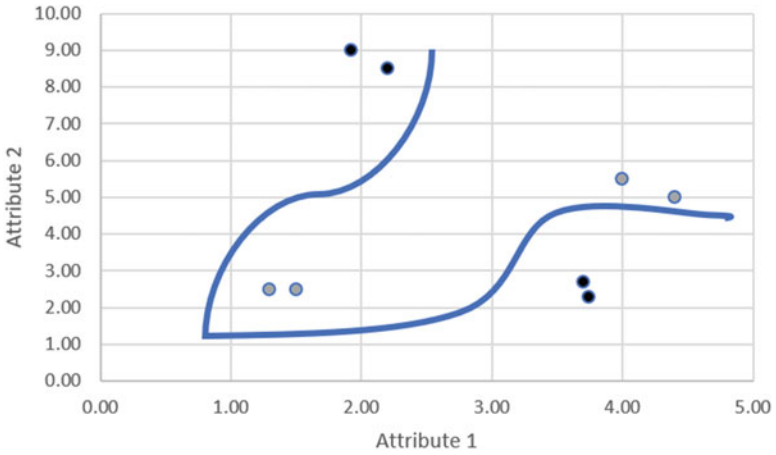

Fig. 11.2 Eight instances belonging to two classes separated by nonlinear function

$$w_2 x_2 + w_1 x_1 + w_0 = 0,$$

where $w_2=1$, $w_1=-a$, and $w_0=-b$.

We will see in the Multilayer Perceptron paragraph how an artificial neural network can solve this problem.

11.3 The Algorithm

A biological neuron can be schematized typically in the following figure (Fig. 11.4).

A brain neuron can be considered an information-processing unit. Neurons communicate through electrical signals. By discharging chemicals known as

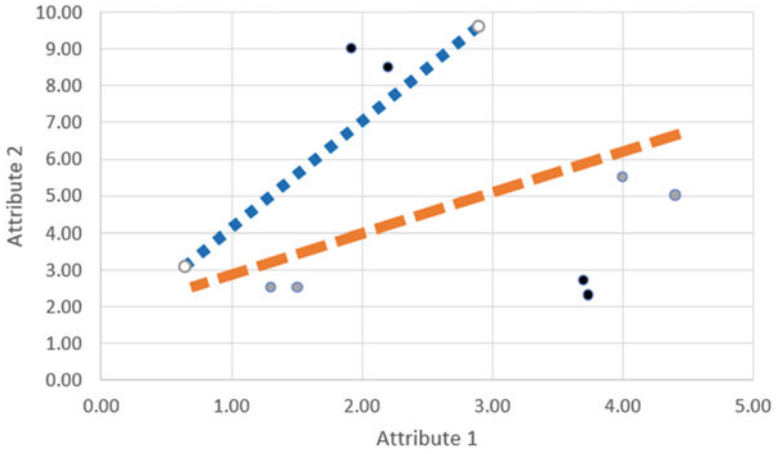

Fig. 11.3 An example of two straight lines drawn in an attempt to separate the two classes

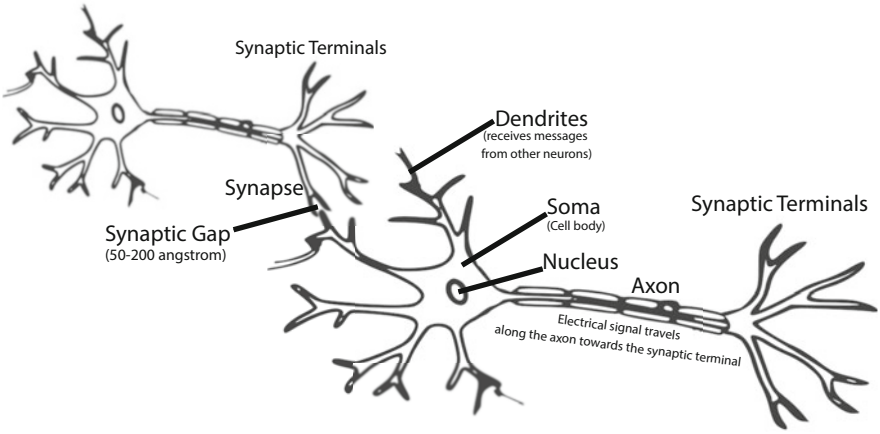

Fig. 11.4 Typical biological neuron

neurotransmitters, the synaptic terminals of one neuron produce a voltage pulse which is communicated to the soma through the dendrites of another neuron. At the soma, the potentials are added, and when the summation rises above a critical threshold, then an electrical signal travels through the axon to the synaptic terminals [2, 3].

Hence, dendrites play the role of input, and the axon, the role of output. As a processing unit, the neuron has many inputs and one output that is connected to many other processing units [3].

Synapses might excite or inhibit the dendrites; exciting a dendrite results in a positive direction of its potential, while inhibiting it results in a negative direction of its potential. Hence, the inputs communicated through the dendrites are "weighted":

some signals are positive (excite), and others are negative (inhibit). At the soma, the weighted inputs are added, and if the sum crosses a threshold, the neuron fires (i.e., gives output). A neuron can fire between 0 and 1500 times per second [2]. The neuron either fires or does not, but what changes is the rate of firing.

11.3.1 The McCulloch–Pitts Neuron

In 1943, Warren McCulloch and Walter Pitts proposed a mathematical model of the neuron known today as the McCulloch–Pitts (M-P) neuron [4, 5] (Fig. 11.5). The inputs (e.g., dendrites) of an M-P neuron are either 0 or 1, and it can be thought of as formed of two parts: the first part sums up all input values, and the second makes a decision about the resulting sum. The decision function f will provide an output of 1 if the sum of the inputs is greater than or equal to a certain threshold θ (pronounced theta) and 0 otherwise.

Let us use an M-P neuron to decide whether to go to the movie theater or not. Suppose that we base our decision on four binary parameters: it is a weekday (x_1), it is after 6:00 p.m. (x_2), it is not during the COVID-19 pandemic (x_3), and the actor is Shah Rukh Khan (x_4). A decision will be made to go to watch the movie if three out of the four conditions are met ($\theta = 3$): $f(g(x)) = 1$ if $g(x) \geq \vartheta$; $f(g(x)) = 0$ *if* $g(x) < \vartheta$.

θ is called the bias; we can think of it as the prior prejudice. For example, for a certain group of people, it might be enough that two of the conditions are met to decide to go to the movie theater; for others, the threshold could be 4 or even 0.

What would be the M-P decision on a Tuesday at 5:00 p.m. during the pandemic if the actor was Shah Rukh Khan?

$$o = g(x) = x_1 + x_2 + x_3 + x_4 = 1 + 0 + 0 + 1 = 2$$

$f(\mathrm{o}) = f(g(x)) = f(2) = 0$; the decision is not to go to the movie theater (i.e., the neuron will not fire).

What would be the M-P decision on a Tuesday at 7:00 p.m. during the pandemic if the actor was Shah Rukh Khan?

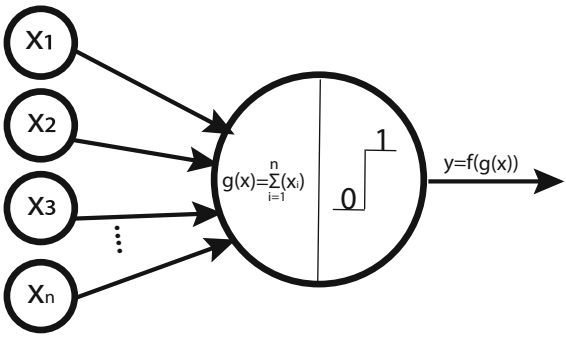

Fig. 11.5 A McCulloch–Pitts neuron

$$g(x) = x_1 + x_2 + x_3 + x_4 = 1 + 1 + 0 + 1 = 3$$

$f(g(x)) = f(3) = 1$; the decision is to go to the movie theater (i.e., the neuron will fire).

The M-P neuron was the first step towards today's neural network; however, it was very restrictive. First, not all our inputs are binary; they can be numerical or categorical. Also, the output we desire is not always binary—we might want to predict a number in the case of regression or predict a class out of multiple existing classes (more than 2) in the case of classification problems.

11.3.2 The Perceptron

To overcome these limitations, the perceptron model was proposed by Frank Rosenblatt in 1958; the model was refined by Minsky and Papertin 1969. Mainly, the perceptron proposed to add adaptive weights to the inputs (Fig. 11.6).

The neuron has many inputs ($x_0 \ldots x_n$) and adaptive weights ($w_0 \ldots w_n$); each input x_i is multiplied by a corresponding weight w_i, and then the results are summed up, mimicking the dendrites-soma-axon behavior. When the summed-up result is higher than a threshold ϑ, the outcome y is set to 1; otherwise, y is set to 0; y is in fact a function of the weighted sum $y = f(g(x)) = f\left(\sum_{i=1}^{n} (w_i \times x_i)\right)$.

If we go back to the same problem above—the decision to go to the movie theater—but we add to it the weight for each input, the weights can be decided based on knowledge about the importance of each input: a highly important input for making the right decision can be assigned a high weight, and inputs that do not play a major role can be assigned lower weights. Finding ways to determine the best weights and θ for a decision problem is the main goal in the next paragraphs.

Suppose that the perceptron is deciding for a group of Shah Rukh Khan diehard fans, hence the weight w_4 could be 10, while the other weights are set as follows: $w_1 = 2$, $w_2 = 3$, $w_3 = -5$. For the sake of this example, let us change x_3 to represent pandemic if it is equal to 1 and no pandemic if it is equal to 0.

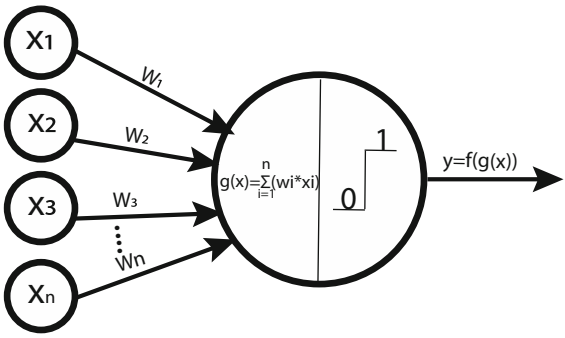

Fig. 11.6 The perceptron

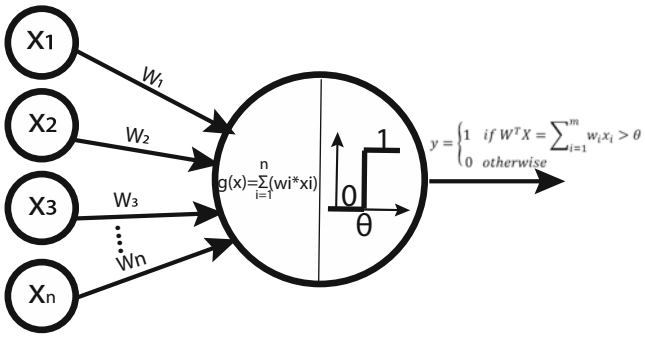

Fig. 11.7 A perceptron with threshold ϑ

What would the perceptron's decision be on a Tuesday at 7:00 p.m. during the pandemic if the actor was Shah Rukh Khan?

$g(x) = w_1x_1 + w_2x_2 + w_3x_3 + w_4x_4 = 2x_1 + 3x_2 + -5x_3 + 10x_4 = 10 > \vartheta = 3$, the decision is to go to watch the movie.

Now, we can change the weights for people who are more reasonable and decide that watching a movie with Shah Rukh Khan (or any other actor) is not of a higher value than their and others' lives; we can decide that $w_3 = -100$, which will push the perceptron's decision "do not go to theater" to always fire under a pandemic.

Mathematically, we could look at the inputs $(x_0 \ldots x_n)$ and the weights $(w_0 \ldots w_n)$ as vectors.

$$\text{The input vector } x \text{ is defined as } x = \begin{bmatrix} x_1 \\ x_2 \\ \ldots \\ x_n \end{bmatrix}.$$

And the weights' vector transpose is defined as $w^T = [w_1\ w_2 \ldots w_n]$.

In mathematics, the multiplication of two vectors w^T and x is written w^Tx and is expressed as follows:

$$w^Tx = [w_1\ w_2 \ldots w_n] \begin{bmatrix} x_1 \\ x_2 \\ \ldots \\ x_n \end{bmatrix} = \sum_{i=1}^{n} (w_i \times x_i)$$

The function f that we have used to provide the output y is a function of the total weighted sum (i.e., $g(x)$) and is called the activation function because it allows us to activate the neuron when the value is greater than or equal to ϑ.

The activation function compares the weighted sum to ϑ and decides to activate the neuron if the weighted sum is greater than ϑ (Fig. 11.7).

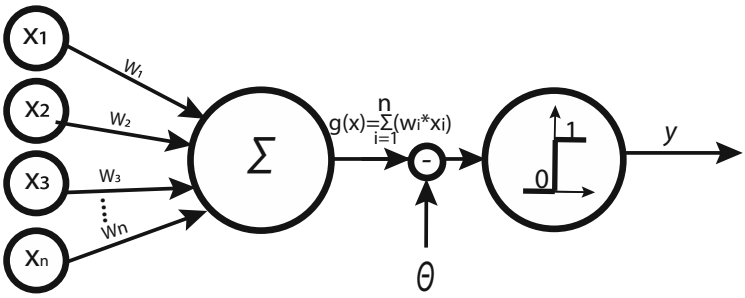

Fig. 11.8 A perceptron with the threshold ϑ subtracted from the weighted sum

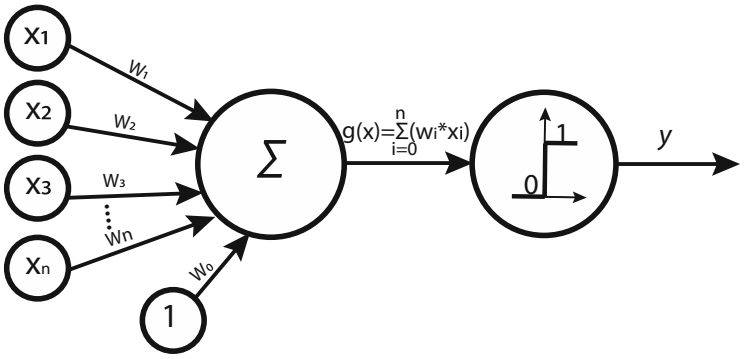

Fig. 11.9 -ϑ as an input weight W_0 for an attribute x_0 of value 1

The same result can be achieved if we subtract the value ϑ from the sum (Fig. 11.8). The result y is the same; however, the activation decision is made based on whether $\sum_{i=0}^{n} (w_i \times x_i) - \theta > 0$ or not.

We can move one step further by treating -ϑ as an extra weight called w_0 multiplied by an attribute x_0 of value 1 (Fig. 11.9).

w_0 is the bias of the model, which we sometimes represent with the letter b, which is familiar in linear functions (i.e., $y = ax + b$).

Hence the following:

$$g(x) = \sum_{i=1}^{n} (w_i \times x_i) + b$$

$$y = f(g(x)) = f\left(\sum_{i=1}^{n} (w_i \times x_i) + b\right)$$

can also be written

$$g(x) = \sum_{i=1}^{n} (w_i \times x_i) + (-\theta) = \sum_{i=\mathbf{1}}^{n} (w_i \times x_i) + (w_0 \times 1) = \sum_{i=\mathbf{0}}^{n} (w_i \times x_i)$$

$$y = f(g(x) = f\left(\sum_{i=\mathbf{0}}^{n} (w_i \times x_i)\right)$$

11.3.3 The Perceptron as a Linear Function

In fact, the perceptron estimates a linear function. Let us take the following example with two input variables, x_1 and x_2, and their corresponding weights, w_1 and w_2. Suppose that we have the following values for the dataset we are trying to model using the perceptron (Table 11.1). That dataset is plotted in Fig. 11.10, where the points corresponding to a zero output are in grey.

Table 11.1 A training dataset

x_1	x_2	y
1	2	1
2	3	1
1	5	1
3	0.5	0
4	1	0
5	2.3	0
0.7	4	1
0.1	4	1
0.2	5	1

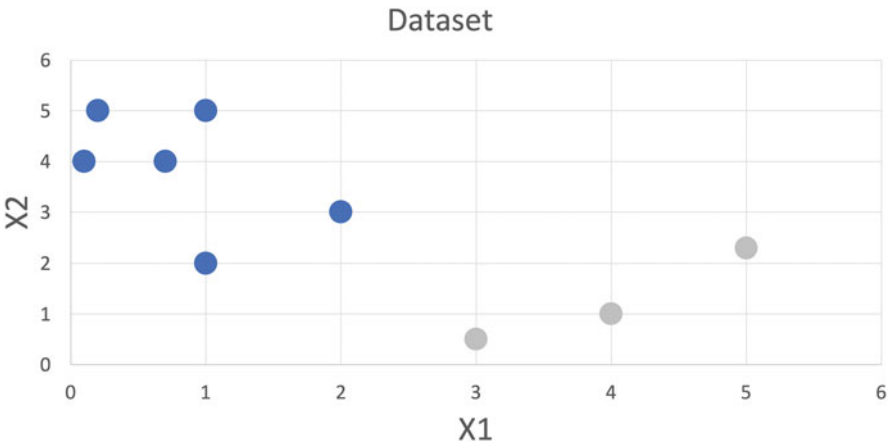

Fig. 11.10 The dataset plotted on a graph

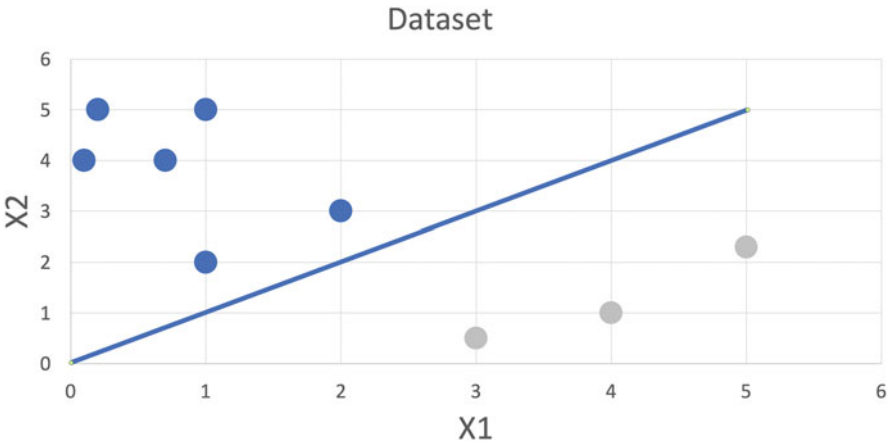

Fig. 11.11 A line ($f(x) = x$) separating the data points that belong to two different categories

It is obvious that we can find a solution to differentiate between the points in grey and the others: we can plot a straight line that separates the two sets of data points. A line such as $y = x$ will do the job (Fig. 11.11).

If we use a perceptron with weights $w_1 = 1$, $w_2 = -1$, and $w_0 = 0$, the perceptron behaves exactly like $f(x) = x$.

Let us start with $f(x) = x$; we can rewrite it as $y = x$.

We can also write it as $x_2 - x_1 = 0$, or $x_2 - x_1\_0 = 0$, or even $w_2x_2 + w_1x_1 + w_0 = 0$, where $w_0 = 0$, $w_1 = -1$, and $w_2 = 1$.

All the points on the line have in common the property $x_2 - x_1\_0 = 0$.

The points on both sides of the lines satisfy either of these two conditions: $x_2 - x_1 - 0 > 0$ or $x_2 - x_1 - 0 < 0$.

We are in a situation of a summation $\sum_{i=0}^{2} (w_i \times x_i)$ and then a decision based on comparison of the resulting sum with a threshold $\theta = 0$. We are in the domain of the perceptron. The perceptron is modeling a straight line, so it is a linear model.

11.3.4 Activation Functions

The activation function f can be different than the one mentioned above; it can, for example, propose that the output be -1 instead of 0; such a function is called bipolar as opposed to unipolar (i.e., output positive or zero). The passage from one output to another (0 to 1 or $-$ 1 to 1) was abrupt in the previous paragraph, but we can use activation functions with a smoother passage; such functions are called soft-limiting (Fig. 11.12).

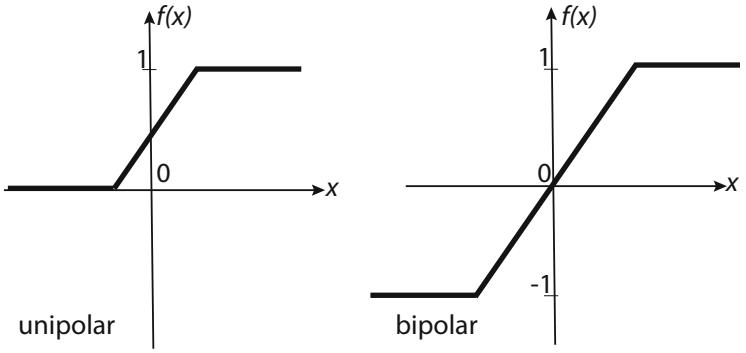

Fig. 11.12 Unipolar and bipolar activation functions

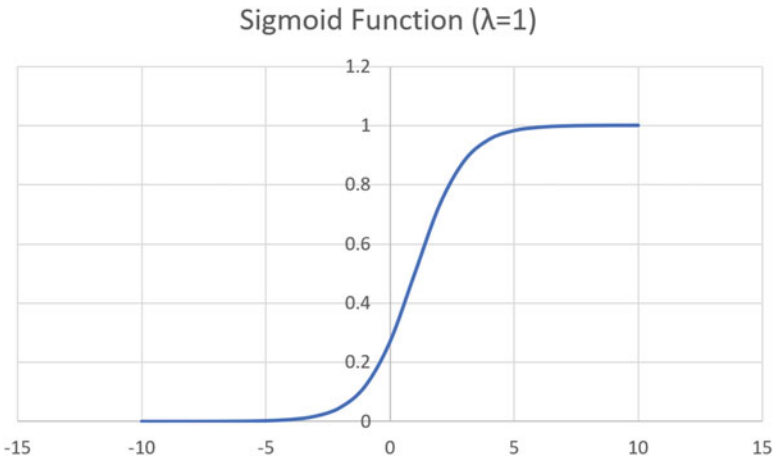

Fig. 11.13 Sigmoid function for $\lambda = 1$

However, for reasons, we will discuss below (i.e., gradient descent), we will need to compute the derivative of the activation function, i.e., it must be differentiable. We have many activation functions to choose from [6].

11.3.4.1 The Sigmoid Function

A function that satisfies the differentiability criterion and that can play the role of a soft-limiting activation function is the sigmoid function, defined as:

$$f(x) = \frac{1}{1 + e^{-\lambda x}}$$

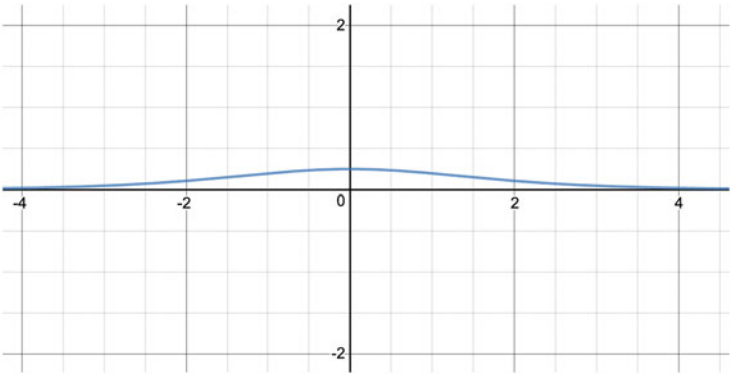

Fig. 11.14 A graph showing the gradient of the sigmoid function

The graph of the sigmoid function is given in Fig. 11.13:where λ determines the steepness of the sigmoid function. We can notice that the outcome of the sigmoid function varies between 0 and 1.

The gradient of the sigmoid is defined as follows:

$$f'(x) = \text{sigmoid}(x) \times (1 - \text{sigmoid}(x))$$

Since the sigmoid function's output is always positive, the gradient of the sigmoid will always be positive, whatever the value of x (Fig. 11.14). In fact, the gradient approaches 0 above +3 and below −3, which indicates that little learning is done above +3 or below −3.

We will overcome this issue if we scale the sigmoid function, and that is the solution proposed by the tanh function.

11.3.4.2 The Tanh Function

The hyperbolic tangent function is defined as follows:

$$f(x) = \tanh\left(\frac{1}{2}\,\lambda x\right)$$

$$\text{for } \lambda = 2, \tanh(x) = \frac{1 - e^{-2x}}{1 + e^{-2x}}$$

The graph of the tanh function is like that of the sigmoid, but it is scaled so that it is symmetric around zero (Fig. 11.15).

The gradient of the tanh function is defined as follows:

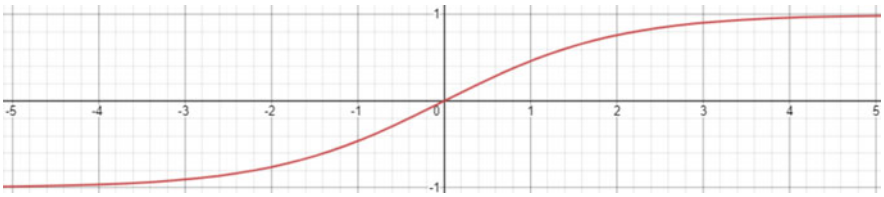

Fig. 11.15 The tanh function for $\lambda = 2$

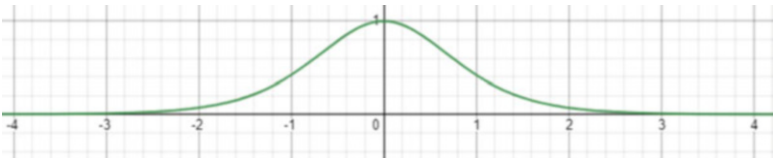

Fig. 11.16 A graph showing the gradient of the tanh function

$$f'(x) = 1 - (\tanh(x))^2$$

We can notice that the graph of the tanh gradient is symmetric around zero, and hence it can be positive or negative (Fig. 11.16)

11.3.4.3 The ReLU Function

The rectified unit function (ReLU) is defined as follows:

$$\text{ReLU}(x) = f(x) = \begin{cases} 0 & \text{if } x < 0 \\ x & \text{otherwise} \end{cases}$$

The ReLU graph is shown in Fig. 11.17.

For all values that are below 0, the activation function will have 0 as an output; hence, ReLU might activate a subset of all the neurons, which makes it more efficient than other activation functions. The gradient of ReLU is a constant (0 or 1) and is defined as

$$f'(x) = \begin{cases} 0 & \text{if } x < 0 \\ 1 & \text{otherwise} \end{cases}$$

Since the gradient might be 0 for some neurons, during backpropagation, some weights and biases will not be updated, and the corresponding neurons might never get activated; we call such neurons "dead neurons." The leaky ReLU activation function addresses this problem.

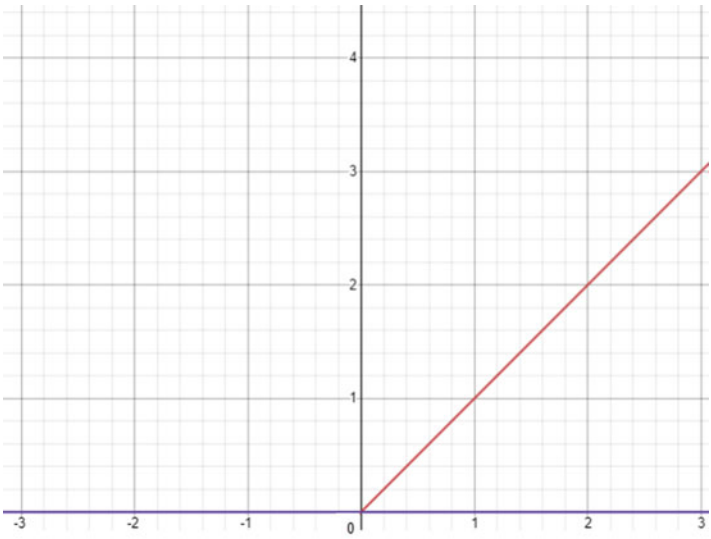

Fig. 11.17 The ReLU function

11.3.4.4 The Leaky ReLU Function

Leaky ReLU is defined as follows:

$$\text{ReLU}(x) = f(x) = \begin{cases} 0.01x & \text{if } x < 0 \\ x & \text{otherwise} \end{cases}$$

The leaky ReLU graph is shown in Fig. 11.18
The leaky ReLU gradient function is defined as follows:

$$\text{ReLU}(x) = f(x) = \begin{cases} 0.01 & \text{if } x < 0 \\ 1 & \text{otherwise} \end{cases}$$

With leaky ReLU, the gradient of the negative inputs will never be 0; hence, there will be no dead neurons.

11.3.4.5 The Parameterized ReLU Function

The parameterized ReLU function adds flexibility for the negative values of x as it introduces the slope as a parameter (instead of the constant slope 0.01). The function is defined as follows:

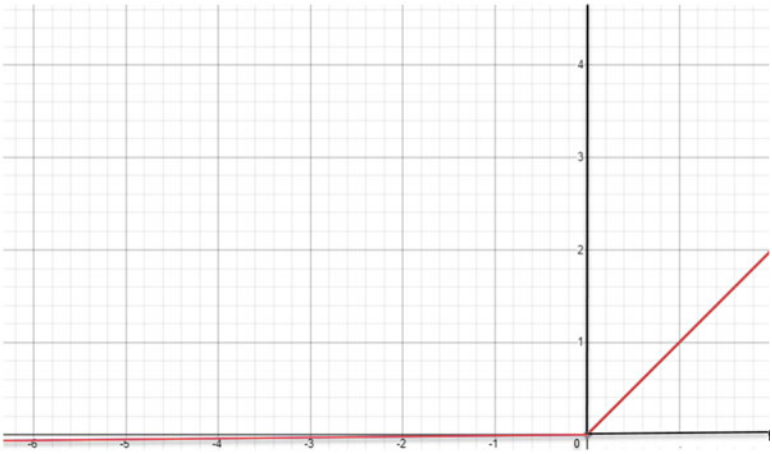

Fig. 11.18 The leaky ReLU function

$$\text{ReLU}(x) = f(x) = \begin{cases} ax & \text{if } x < 0 \\ x & \text{otherwise} \end{cases}$$

The only caveat is that the artificial neural network will also learn the slope a for an optimal convergence.

The parameterized ReLU gradient function is defined as follows:

$$\text{ReLU}(x) = f(x) = \begin{cases} a & \text{if } x < 0 \\ 1 & \text{otherwise} \end{cases}$$

11.3.4.6 The Swish Function

The swish activation function shows better performance than ReLU and is very efficient; it is defined as follows (Fig. 11.19):

$$f(x) = \frac{x}{1 + e^{-x}}$$

11.3.4.7 The SoftMax Function

The SoftMax function turns a *vector* x of k real values x_j, $j = 1$ to k, into a vector of k real values that sum to 1; it is defined as:

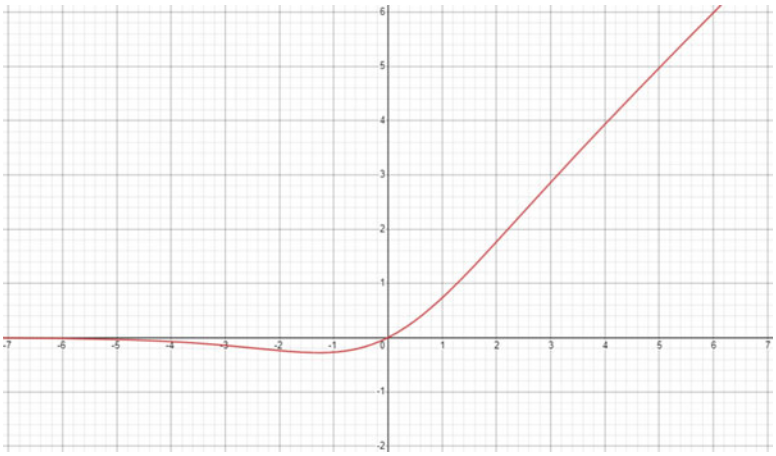

Fig. 11.19 The swish Function

$$\sigma(x_i) = \frac{e^{-x_i}}{\sum_{j=1}^{K} e^{-x_j}}$$

Since the SoftMax function returns values between 0 and 1, we can treat these values as probabilities that an input belongs to a particular class. The SoftMax activation function is very useful for multiclass classification, where the ANN has multiple neurons as output.

11.3.4.8 Which Activation Function to Choose?

There is no formula; however, the following are rules of thumb:

- Sigmoid functions work well in classification problems.
- Sigmoid and tanh functions have one notable drawback: the vanishing gradient.
- The ReLU function is generic and is widely used.
- In the case of dead neurons, use leaky ReLU.
- Use ReLU first; if it does not provide you with a good solution, then you can try other activation functions.
- Use SoftMax for multiclass classification problems.

11.3.5 Training the Perceptron

The question is how to find the right weights for the perceptron. We will do that by gradient descent, which we have seen during regression.

Let us define an error function E for the perceptron. If we take the error function as the mean squared error (MSE), and suppose that we have a training set of N instances (x^i, y^i), then E can be formulated as:

$$E = \frac{1}{2N}\sum_{i=1}^{N}\left(y^i - \widehat{y}^i\right)^2 = \frac{1}{2N}\sum_{i=1}^{N}\left(y^i - f\left(w^T x^i + w_0\right)\right)^2$$

- The value $\frac{1}{2}$ is chosen for convenience in later calculations (i.e., derivatives).
- The training dataset is formed of N instances $\{(x^1, y^1), (x^2, y^2). . .(x^N, y^N)\}$. Each x^i is an input vector with n attributes/features $(x^i{}_1, x^i{}_2, \ldots x^i{}_n)$, and each y^i is an expected output for vector x^i.
- w^T is the transpose of the weight vector $(w_1, w_2, \ldots w_n)$, where w_0 is the bias.
- $\widehat{y}^i$ is the thresholded output computed by the perceptron for the input vector x^i.

Our aim is to find the set of weights that minimizes E.

The error value depends on the values of w_0, which is the bias b, and on all other weights represented by the vector w, so E is a function of both variables. To obtain $E(w, b)$, we replace $f(w^T x^i + b)$ with $w^T x^i + b$. The error function is then expressed as follows:

$$E(w, b) = \frac{1}{2N}\sum_{i=1}^{N}\left(y^i - \left(w^T x^i + b\right)\right)^2$$

When $w^T x^i + b = y^i$ for all x^i, $i = 1$ to N, then $E = 0$; our aim is to find a set of weights that makes E as close as possible to 0. When the perceptron learns how to fit the unthresholded outputs $w^T x^i + b$ to the desired outputs y^i, it is simple to take the same weights, apply them to the input vectors x^i, and then use a threshold function f to obtain perceptron outputs $\widehat{y}^i$ that correctly classify the x^i. For example, suppose the vectors x^i are in two classes, $y^i = 1$ and $y^i = 0$; then if $(w^T x^i + b)$ correctly classifies a vector x^i into one of the two classes (e.g., 1), that means that $w^T x^i + b$ is equal to either 1 or 0; if we apply an activation function $f(w^T x^i + w_0)$ that produces 1 if $w^T x^i + b = 1$ and produces 0 if $w^T x^i + b = 0$, the perceptron's output $f(w^T x^i + w_0)$ will correctly classify x^i.

So, we will be interested in minimizing the error function for the output $\widehat{y}^i$:

$$E(w, b) = \frac{1}{2N}\sum_{i=1}^{N}\left(y^i - \left(w^T x^i + b\right)\right)^2$$

As was the case with the regression, we will use gradient descent to minimize the error function until convergence is reached.

$$\frac{\partial E}{\partial w_j} = \frac{\partial \left(\frac{1}{2N} \sum_{i=1}^{N} \left(y^i - \sum_{j=1}^{n} \left(w_j x_j^i + w_0 \right) \right)^2 \right)}{\partial w_j}$$

The error e_i made for the ith sample can be expressed as follows:

$$e^i = y^i - \sum_{j=1}^{n} \left(w_j x_j^i + w_0 \right)$$

Hence

$$\frac{\partial E}{\partial w_j} = \frac{\partial \left(\frac{1}{2N} \sum_{i=1}^{N} (e^i)^2 \right)}{\partial w_j}$$

$$\frac{\partial E}{\partial w_j} = \frac{1}{2N} \frac{\partial \left(\sum_{i=1}^{N} (e^i)^2 \right)}{\partial w_j}$$

$$\frac{\partial E}{\partial w_j} = \frac{1}{2N} \sum_{i=1}^{N} \frac{\partial (e^i)^2}{\partial w_j}$$

$$\frac{\partial E}{\partial w_j} = \frac{1}{2N} \sum_{i=1}^{N} 2e^i \frac{\partial (e^i)}{\partial w_j}$$

$$\frac{\partial E}{\partial w_j} = \frac{1}{N} \sum_{i=1}^{N} e^i \frac{\partial (e^i)}{\partial w_j}$$

$$\frac{\partial E}{\partial w_j} = \frac{1}{N} \sum_{i=1}^{N} e^i \frac{\partial \left(y^i - \sum_{j=1}^{n} \left(w_j x_j^i + w_0 \right) \right)}{\partial w_j}$$

$$\frac{\partial E}{\partial w_j} = \frac{1}{N} \sum_{i=1}^{N} \left(e^i \left(-x_j^i \right) \right)$$

$$\frac{\partial E}{\partial w_j} = -\frac{1}{N} \sum_{i=1}^{N} e^i x_j^i = -\frac{1}{N} \sum_{i=1}^{N} \left(\left(\left(y^i - \sum_{j=1}^{n} \left(w_j x_j^i + w_0 \right) \right) x_j^i \right) \right)$$

Now, we will compute $\frac{\partial E}{\partial b}$, which is $\frac{\partial E}{\partial w_0}$, where w_0 is the weight for $x^i{}_0$ and $x_0^i = 1$.

$$\frac{\partial E}{\partial b} = \frac{\partial E}{\partial w_0} = -\frac{1}{N}\sum_{i=1}^{N}\left(\left(\left(y^i - \sum_{j=1}^{n}\left(w_j x_j^i + w_0\right)\right)x_0^i\right)\right) =$$

$$-\frac{1}{N}\sum_{i=1}^{N}\left(y^i - \sum_{j=1}^{n}\left(w_j x_j^i + w_0\right)\right)$$

We can start at the first iteration at a random value for the weights w and the bias b; then, we adjust these values at the next iteration based on the following formula, where k is the current iteration:

$$w_{(k+1)} = w_{(k)} - \alpha\frac{\partial E}{\partial w_{(k)}}$$

$$b_{(k+1)} = b_{(k)} - \alpha\frac{\partial E}{\partial b_{(k)}}$$

where α is the learning rate (e.g., 0.01) and $\frac{\partial E}{\partial w_{(k)}}$ and $\frac{\partial E}{\partial b_{(k)}}$ are the gradient of E with respect to w and b, at iteration k, respectively.

The updates of the perceptron parameters w and b are calculated as the difference (represented by delta Δ) between their values in the next iteration k and in the current one [7]:

$$w_{(k+1)} = w_{(k)} - \alpha\frac{\partial E}{\partial w_{(k)}}$$

$$w_{(k+1)} = w_{(k)} + \alpha\frac{1}{N}\sum_{i=1}^{N} e^i x_j^i$$

$$\boldsymbol{w_{(k+1)} = w_{(k)} + \frac{\alpha}{N}\sum_{i=1}^{N}\left(\left(y^i - \sum_{j=1}^{n}\left(w_{j(k)}x_j^i + w_{0(k)}\right)\right)x_j^i\right)}$$

$$b_{(k+1)} = b_{(k)} - \alpha\frac{\partial E}{\partial b_{(k)}}$$

$$b_{(k+1)} = b_{(k)} + \alpha\frac{1}{N}\sum_{i=1}^{N} e^i$$

$$\boldsymbol{b_{(k+1)} = b_{(k)} + \frac{\alpha}{N}\sum_{i=1}^{N}\left(y^i - \sum_{j=1}^{n}\left(w_{j(k)}x_j^i + w_{0(k)}\right)\right)},$$

which is equivalent to writing $w_{0(k+1)} = w_{0(k)} + \alpha\frac{1}{N}\sum_{i=1}^{N}\left(y^i - \sum_{j=1}^{n}\left(w_{j(k)}x_j^i + w_{0(k)}\right)\right)$.

To train the perceptron, we can proceed by:

1. Forward calculation: Calculating $w^T x_i + b$ for all x_i. Such a run into the N instances of the training set is called an epoch.
2. Updating the weights and the bias $w_{(k+1)}$ and $w_{0(k+1)}$.
3. Repeating steps 1 and 2 until $E(w, b)$ converges.
4. Using the last calculated weights and bias to predict the output y for any new input x.

As we can guess, the perceptron is a linear model and cannot solve a nonlinear problem.

In practice, using all the available instances to make a single update of the weights might be extremely slow, so instead, we sample a random smaller batch of the training dataset to compute every update. This method is called the minibatch **stochastic gradient descent**.

11.3.6 Perceptron Limitations: XOR Modeling

The exclusive-or (XOR) function is a function with two input variables, x_1 and x_2, that has an output of 1 if either x_1 or x_2 is 1; otherwise, it is 0. The truth table is shown in Table 11.2, and the corresponding plot is in Fig. 11.20.

To model the XOR function, we need to find a line that separates outputs 1 (black dots in Fig. 11.20) from outputs 0 (grey dots in Fig. 11.20). However, we cannot find a linear function that separates those outputs; we can see an example failing to represent XOR in Fig. 11.20.

The perceptron is a linear classifier and hence cannot find a model to classify correctly an XOR function. We can, however, extend the perceptron by adding more layers so that it becomes a multilayer perceptron (MLP), which will enable it to model nonlinear complex functions and virtually any function.

11.3.7 Multilayer Perceptron (MLP)

A multilayer perceptron (MLP) has one or more hidden layers. Figure 11.21 shows an example of two hidden layers, one input layer and one output layer. It is important to note that the perceptron principles function on the hidden layer and the output

Table 11.2 XOR function truth table

x_1	x_2	XOR
0	0	0
0	1	1
1	0	1
1	1	0

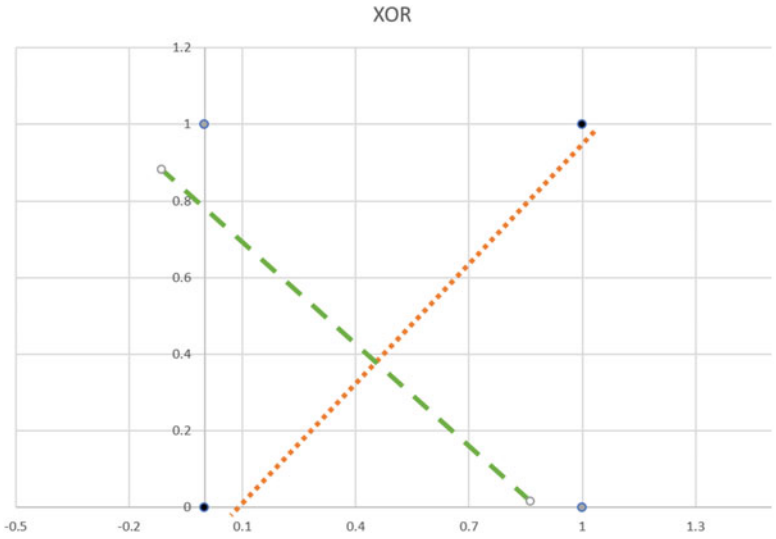

Fig. 11.20 The XOR function with two discriminant lines

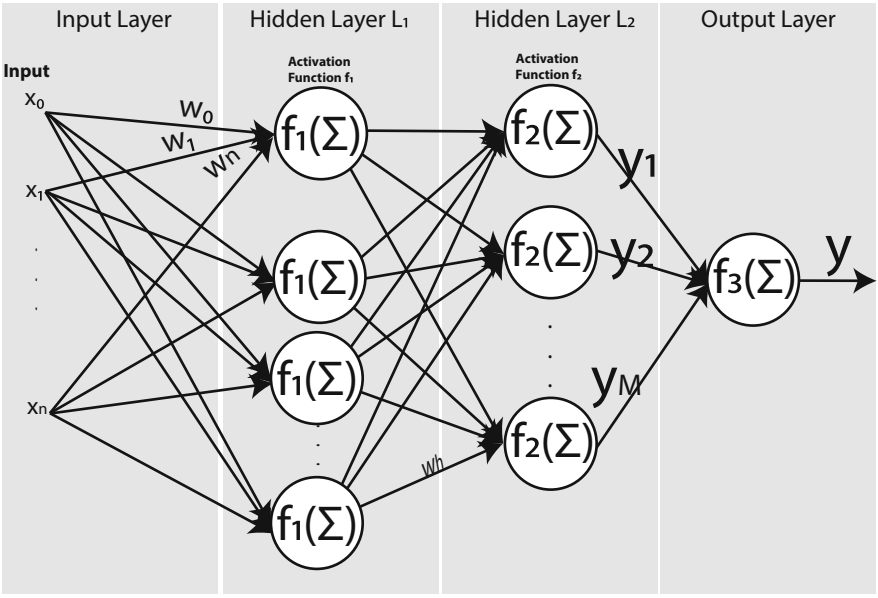

Fig. 11.21 An MLP with two hidden layers and one output

layer, where the hidden layer is formed of the data instances from the training dataset but no perceptron is involved in it; that is why many authors do not count the input layer as part of the total number of layers; however, some authors do.

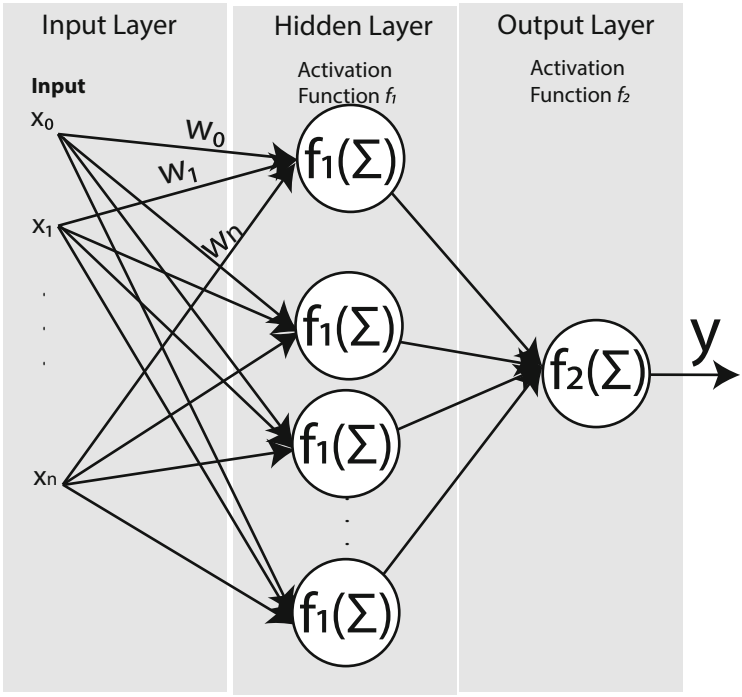

Fig. 11.22 An MLP with one hidden layer and one output y

The inputs in Fig. 11.21 and their corresponding weights are fed into the first hidden layer, composed of several neurons, which in turn generate their own outputs using an activation function f_1 and feed them with their own weights to the second hidden layer, which in turn applies an activation function f_2 and feeds its outputs to the output layer; the latter applies an activation function f_3 and generates the final MLP output.

The activation functions in each layer (i.e., f_1, f_2, and f_3) can be the same or different. If the activation function of the output layer is a linear function, the MLP generates a regression model, while if it is nonlinear, then the model is nonlinear (i.e., if the function is logistic, then the model is logistic regression or binary classification) [8]. Figure 11.22 shows an MLP with one hidden layer and one output, while Fig. 11.23 shows an MLP with one hidden layer and three outputs.

Within an MLP, we have different weights and different biases (i.e., constant b) for each layer; hence, expressing the learning problem becomes more elaborate, but it follows the same principle as in the case of one perceptron.

What do hidden layers do exactly? We will continue the practical example to answer this question.

We have seen in Fig. 11.3 that we need two lines to separate the two given classes. We know that a perceptron models a linear function; needing n lines to separate two classes is equivalent to say that we need n perceptrons. In our example,

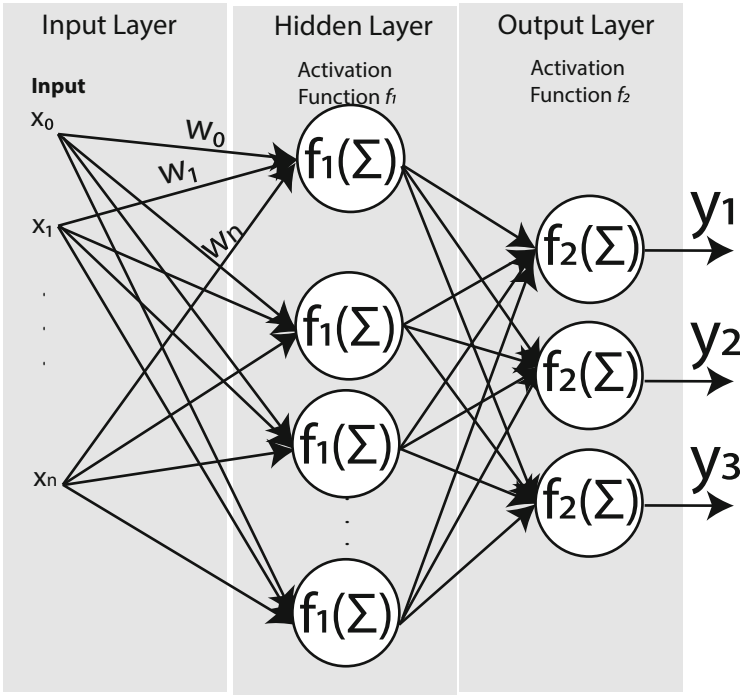

Fig. 11.23 An MLP with one hidden layer and three outputs y_1, y_2, and y_3

we will need two perceptrons in one hidden layer, where each hidden perceptron (i.e., neuron) produces one line. Since we need to join the two lines in order to have one model that separates the two classes, then we will need to join the two neurons' outputs into one neuron (the output neuron). The result is shown in Fig. 11.24.

This example is to illustrate the benefit of hidden neurons; in complex real-life problems, we cannot just guess the number of hidden neurons and the number of hidden layers required to create the model.

11.3.8 MLP Algorithm Overview

The MLP follows the algorithm below:

Initialize the input layer
Initialize the weights' vectors and bias vectors for all layers
PHASE 1: forward computation

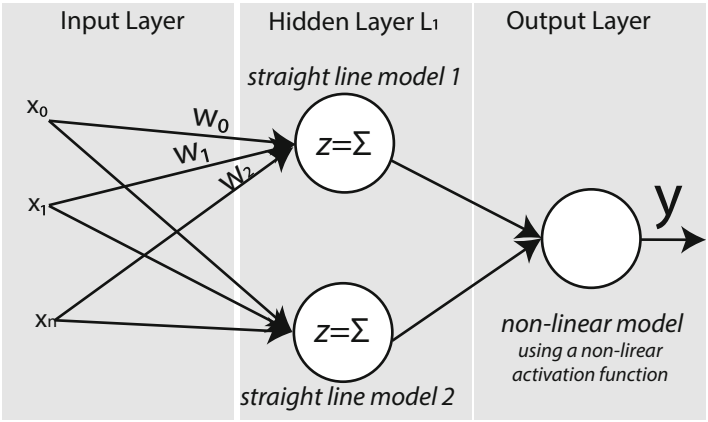

Fig. 11.24 A multilayer perceptron with a two-neuron hidden layer to model a nonlinear classifier solving the problem in Fig. 11.3

For *each layer l* from layer 2 to the output layer L (layer 1 being the MLP inputs)

For *each neuron i* in layer *l*

Compute the sum based on layer *l*'s weights, bias, and the *previous* layer outputs

$$z_i^{(l)} = \sum_{j=1}^{N^{(l-1)}} w_{ij}^{(l)} \times a_j^{(l-1)} + b^{(l)}$$

Compute the output based on the previous sum

$$a_i^{(l)} = f^{(l)}\left(z_i^{(l)}\right)$$

End For

End For

Learning the model entails iteratively calculating the gradient for a cost function such as the mean squared error (MSE) until the minimum is found (the algorithm converges).

The example here is for MSE.

$$\text{Use } E(X) = \frac{1}{2N}\sum_{i=1}^{N}(\hat{y}_i - y_i)^2$$

Starting with the last layer and working backward, compute for every neuron

$$\frac{\partial E(X)}{\partial w_{ji}^{l}} \text{ and } \frac{\partial E(X)}{\partial b_{i}^{l}}$$

PHASE 2: Backward propagation

We start from the output layer and update the weights and biases of each layer backward: layer L first, then L-1, and we continue until the weights of layer 0 are updated.

Weights' update:

$$w_{i+1}^{l} = w_i - \alpha \frac{\partial E(X)}{\partial w_{ji}^{l}}$$

$$b_{i+1}^{l} = b_i - \alpha \frac{\partial E(X)}{\partial b_{i}^{l}}$$

Then we repeat the two phases until convergence, i.e., the cost is less than a certain threshold.

11.3.9 Backpropagation

The problem we are facing is to find a method to minimize the error the MLP can produce by minimizing the error function that estimates the difference between the final output of the MLP and the expected outcome.

Backpropagation is a technique that allows us to achieve such aim; it performs a gradient descent by working backward from the output layer to the input layer, calculating in each layer the gradient of the error function with respect to the neural network's weights. The gradients of the last layer of weights are computed first and then used in computation of the gradient for the previous layer; the process continues until we reach the first layer of weights [9].

The mathematical notation is complex if we want to take a fully connected neural network, so we will start with an example and move towards the fully connected situation.

We will use the following denotations:

- E denotes our error (i.e., cost) function
- L denotes the number of layers
- N_l denotes the number of neurons in layer l
- $w_{ij}^{(l)}$ denotes the weight for neuron i in layer l in relation to the incoming neuron j in layer l-1
- $b_i^{(l)}$ denotes the bias for neuron i in layer l
- $z_i^{(l)}$ denotes the product sum plus bias for neuron i in layer l: $z_i^{(l)} =$

$\sum_{j=1}^{N^{(L-1)}} w_{ij} a_j^{(l-1)} + b_j^{(l-1)}$

- σ denotes a nonlinear activation function in layer l
- $a_i^{(l)}$ denotes the output at a neuron i in layer l: $a_i^{(l)} = \sigma\left(z_i^{(l)}\right)$
- $a^{(l)}$ denotes the output vector for layer l: $a^{(l)} = \left\{a_1^{(l)}, a_2^{(l)}, \ldots a_{N_l}^{(l)}\right\}$;
- $w_i^{(l)}$ denotes the weight vector for neuron i in layer l; $w_i^{(l)} = \left\{ w_1^{(l)}, w_2^{(l)}, \ldots w_{N_l}^{(l)} \right\}$
- $w_{ij}^{(l)}$ denotes the weight vector connecting the neuron i in layer l to neuron j in layer *l-1*

11.3.9.1 Simple 1–1–1 Network

Let us take an example of a three-layer neural network ($L = 3$) with an input layer with 1 neuron, a hidden layer with 1 neuron, and an output layer with 1 neuron (Fig. 11.25).

There are only three layers: layer L (output), layer L-1 (hidden), and layer L-2 (input). We have one neuron in each layer, so we will not use the subscript i; for example, instead of a_i^l we will use a^l, and the same applies for all other notations.

$$a^{(L)} = \sigma\left(z^{(L)}\right)$$

$$z^{(L)} = w^{(L-1)} a^{(L-1)} + b^{(L-1)}$$

$$z^{(L-1)} = w^{(L-2)} a^{(L-2)} + b^{(L-2)}$$

We need to compute the gradient (partial derivative) of the error function E (or cost function C) with respect to the weights and the biases. That is, we would like to know how our cost function would change if we changed the weights and biases of the network.

Starting in the last layer, we then investigate how this gradient propagates backward through the network.

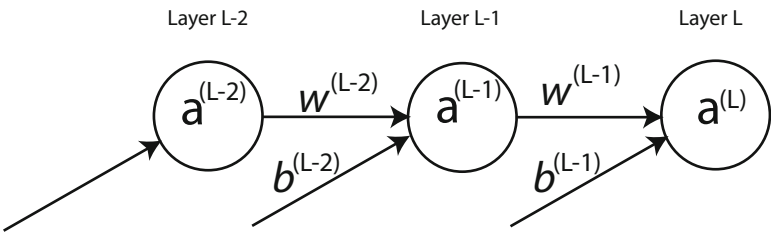

Fig. 11.25 A three-layer neural network formed; each layer is 1 neuron

11.3.9.1.1 Computation with Respect to Layer *L*-1

Having one node per layer will help us understand the computational work and its implications. We will start with the layer L and calculate the gradient of our error function E with respect to the weights and biases of neurons in the previous layer $L-1$, $\frac{\partial E}{\partial w^{(L-1)}}$. Figure 11.26 clarifies the relationship between the cost function and the weights and biases.

Let us consider the mean squared error as an error function. Since we have only one neuron in the output:

$$E = \frac{1}{2}\left(a^{(L)} - y\right)^2$$

$$\left(\text{the } \frac{1}{2} \text{ is for convenience}\right)$$

Using the chain rule, we can write:

$$\frac{\partial E}{\partial w^{(L-1)}} = \frac{\partial E}{\partial a^{(L)}} \frac{\partial a^{(L)}}{\partial z^{(L)}} \frac{\partial z^{(L)}}{\partial w^{(L-1)}}$$

$$\frac{\partial E}{\partial a^{(L)}} = \frac{1}{2} \frac{\partial \left(a^{(L)} - y\right)^2}{\partial a^{(L)}} = \frac{1}{2} 2\left(a^{(L)} - y\right) = \left(a^{(L)} - y\right)$$

$$\frac{\partial a^{(L)}}{\partial z^{(L)}} = \frac{\partial \sigma\left(z^{(L)}\right)}{\partial z^{(L)}} = \sigma'\left(z^{(L)}\right)$$

$$\frac{\partial z^{(L)}}{\partial w^{(L-1)}} = \frac{\partial \left(w^{(L-1)} a^{(L-1)} + b^{(L-1)}\right)}{\partial w^{(L-1)}} = a^{(L-1)}$$

Hence, we can solve $\frac{\partial E}{\partial w^{(L-1)}}$:

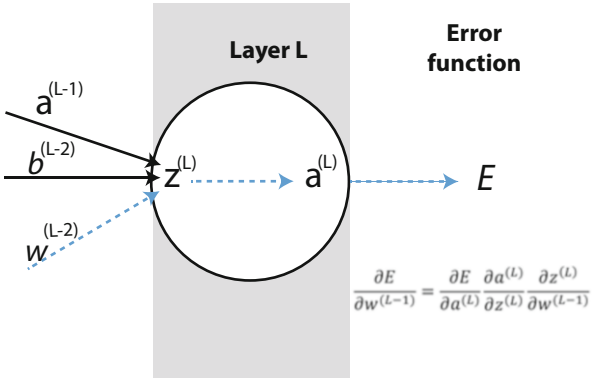

Fig. 11.26 The cost function E's relationship with the weights and biases passes through a chain from E to a, from a to z, and from z to the weights and biases

$$\frac{\partial \boldsymbol{E}}{\partial \boldsymbol{w}^{(\boldsymbol{L}-\mathbf{1})}} = \left(\boldsymbol{a}^{(\boldsymbol{L})} - \boldsymbol{y}\right)\boldsymbol{\sigma}'\left(\boldsymbol{z}^{(\boldsymbol{L})}\right)\boldsymbol{a}^{(\boldsymbol{L}-\mathbf{1})}$$

Just note that the sigmoid is used as the nonlinear activation function; then, $\sigma = \frac{1}{1+e^{(-z)}}$, and its derivative is $\sigma' = \frac{e^{(-z)}}{(1+e^{(-z)})^2}$. Also note that we would like to see how the cost function changes with the change of the weight $w^{(L-2)}$; we will see that in a moment. First, let us see how the cost function changes with the change of the bias $b^{(L-1)}$. We will use the chain rule $\frac{\partial E}{\partial b^{(L-1)}}$, which can be written as follows:

$$\frac{\partial E}{\partial b^{(L-1)}} = \frac{\partial E}{\partial a^{(L)}}\frac{\partial a^{(L)}}{\partial z^{(L)}}\frac{\partial z^{(L)}}{\partial b^{(L-1)}}$$

$$\frac{\partial E}{\partial a^{(L)}} = \left(a^{(L)} - y\right)$$

$$\frac{\partial a^{(L)}}{\partial z^{(L)}} = \sigma'\left(z^{(L)}\right)$$

$$\frac{\partial z^{(L)}}{\partial b^{(L-1)}} = \frac{\partial\left(w^{(L-1)}a^{(L-1)} + b^{(L-1)}\right)}{\partial w^{(L-1)}} = 1$$

Therefore,

$$\frac{\partial \boldsymbol{E}}{\partial \boldsymbol{b}^{(\boldsymbol{L}-\mathbf{1})}} = \left(\boldsymbol{a}^{(\boldsymbol{L})} - \boldsymbol{y}\right)\boldsymbol{\sigma}'\left(\boldsymbol{z}^{(\boldsymbol{L})}\right)$$

So, based on the weights and biases' initial values, we have used a training instance to compute the predicted output $a^{(L)}$, then computed the gradient of the cost (i.e., error) with respect to the weights and biases, as we have just seen. We can use those gradients in the following equations to update the weights and biases before going forward with another training round (time $t + 1$):

$$w^{(L-1)}(t+1) = w^{(L)}(t) - \alpha\frac{\partial \boldsymbol{E}}{\partial \boldsymbol{w}^{(\boldsymbol{L}-\mathbf{1})}}$$

$$b^{(L-1)}(t+1) = b^{(L)}(t) - \alpha\frac{\partial \boldsymbol{E}}{\partial \boldsymbol{b}^{(\boldsymbol{L}-\mathbf{1})}}$$

where α is the training rate, t denotes a round of training.

11.3.9.1.2 Computation with Respect to Layer *L*-2

We will now proceed further up the network and calculate the gradient of our error function E with respect to the weights and biases of neurons in the previous layer

L-2. This is a measurement of how much E changes with respect to changes in weights and biases at level L-2.

But before we can know that, we will need $\frac{\partial E}{\partial a^{(L-1)}}$, so let us compute that derivative.

$$\frac{\partial E}{\partial a^{(L-1)}} = \frac{\partial E}{\partial a^{(L)}} \frac{\partial a^{(L)}}{\partial z^{(L)}} \frac{\partial z^{(L)}}{\partial a^{(L-1)}}$$

$$\frac{\partial E}{\partial a^{(L)}} = \left(a^{(L)} - y\right)$$

$$\frac{\partial a^{(L)}}{\partial z^{(L)}} = \sigma'\left(z^{(L)}\right)$$

$$\frac{\partial z^{(L)}}{\partial a^{(L-1)}} = w^{(L-1)}$$

Hence,

$$\frac{\partial E}{\partial a^{(L-1)}} = \left(a^{(L)} - y\right)\sigma'\left(z^{(L)}\right)w^{(L-1)}$$

Now that we have found the gradient of E with respect to $a^{(L-1)}$, we can proceed with our investigation:

$$\frac{\partial E}{\partial w^{(L-2)}} = \frac{\partial E}{\partial a^{(L-1)}} \frac{\partial a^{(L-1)}}{\partial z^{(L-1)}} \frac{\partial z^{(L-1)}}{\partial w^{(L-2)}}$$

$$\frac{\partial a^{(L-1)}}{\partial z^{(L-1)}} = \frac{\partial \sigma\left(z^{(L-1)}\right)}{\partial z^{(L-1)}} = \sigma'\left(z^{(L-1)}\right)$$

$$\frac{\partial z^{(L-1)}}{\partial w^{(L-2)}} = \frac{\partial\left(w^{(L-2)}a^{(L-2)} + b^{(L-2)}\right)}{\partial w^{(L-2)}} = a^{(L-2)}$$

$$\frac{\partial E}{\partial w^{(L-2)}} = \frac{\partial E}{\partial a^{(L-1)}}\boldsymbol{\sigma'}\left(\boldsymbol{z^{(L-1)}}\right)\boldsymbol{a^{(L-2)}}$$

$$\frac{\partial \boldsymbol{E}}{\partial \boldsymbol{w^{(L-2)}}} = \left(\boldsymbol{a^{(L)}} - \boldsymbol{y}\right)\boldsymbol{\sigma'}\left(\boldsymbol{z^{(L)}}\right)\boldsymbol{w^{(L-1)}}\boldsymbol{\sigma'}\left(\boldsymbol{z^{(L-1)}}\right)\boldsymbol{a^{(L-2)}}$$

Similarly, $\frac{\partial E}{\partial b^{(L-1)}}$ can be computed as follows:

$$\frac{\partial E}{\partial b^{(L-2)}} = \frac{\partial E}{\partial a^{(L-1)}} \frac{\partial a^{(L-1)}}{\partial z^{(L-1)}} \frac{\partial z^{(L-1)}}{\partial b^{(L-2)}}$$

$$\frac{\partial a^{(L-1)}}{\partial z^{(L-1)}} = \sigma'\left(z^{(L-1)}\right)$$

$$\frac{\partial z^{(L-1)}}{\partial b^{(L-2)}} = \frac{\partial\left(w^{(L-2)}a^{(L-2)} + b^{(L-2)}\right)}{\partial w^{(L-2)}} = 1$$

$$\frac{\partial \boldsymbol{E}}{\partial \boldsymbol{b}^{(\boldsymbol{L-2})}} = \frac{\partial E}{\partial a^{(L-1)}}\boldsymbol{\sigma}'\left(\boldsymbol{z}^{(\boldsymbol{L-1})}\right)$$

$$\frac{\partial \boldsymbol{E}}{\partial \boldsymbol{b}^{(\boldsymbol{L-2})}} = \left(\boldsymbol{a}^{(\boldsymbol{L})} - \boldsymbol{y}\right)\boldsymbol{\sigma}'\left(\boldsymbol{z}^{(\boldsymbol{L})}\right)\boldsymbol{w}^{(\boldsymbol{L-1})}\boldsymbol{\sigma}'\left(\boldsymbol{z}^{(\boldsymbol{L-1})}\right)$$

We can also update the weights in level L-2 using the usual formula:

$$w^{(L-2)}(t+1) = w^{(L)}(t) - \alpha\frac{\partial \boldsymbol{E}}{\partial \boldsymbol{w}^{(\boldsymbol{L-2})}}$$

$$b^{(L-2)}(t+1) = b^{(L)}(t) - \alpha\frac{\partial \boldsymbol{E}}{\partial \boldsymbol{b}^{(\boldsymbol{L-2})}}$$

11.3.9.2 Fully Connected Neural Network

There are a few adjustments that we have to consider when we have a fully connected neural network.

The mean squared error function E is still a function of the weights vector and the bias b but is now expressed as an average:

$$E(w, b) = \frac{1}{2N}\sum_{i=1}^{N}(a_i - y_i)^2$$

where N is the number of instances in the training set.

11.3.9.2.1 Computation with Respect to Layer *L*-1

$$\frac{\partial \boldsymbol{E}}{\partial \boldsymbol{w}_{\boldsymbol{ij}}^{(\boldsymbol{L-1})}} = \frac{\partial \boldsymbol{E}}{\partial \boldsymbol{a}_{\boldsymbol{i}}^{(\boldsymbol{L})}}\frac{\partial \boldsymbol{a}_{\boldsymbol{i}}^{(\boldsymbol{L})}}{\partial \boldsymbol{z}_{\boldsymbol{i}}^{(\boldsymbol{L})}}\frac{\partial \boldsymbol{z}_{\boldsymbol{i}}^{(\boldsymbol{L})}}{\partial \boldsymbol{w}_{\boldsymbol{ij}}^{(\boldsymbol{L-1})}}$$

$$\frac{\partial \boldsymbol{E}}{\partial \boldsymbol{w}_{\boldsymbol{ij}}^{(\boldsymbol{L-1})}} = \left(\boldsymbol{a}_{\boldsymbol{i}}^{(\boldsymbol{L})} - \boldsymbol{y}_{\boldsymbol{i}}\right)\boldsymbol{\sigma}'\left(\boldsymbol{z}_{\boldsymbol{i}}^{(\boldsymbol{L})}\right)\boldsymbol{a}_{\boldsymbol{i}}^{(\boldsymbol{L-1})}$$

$$\frac{\partial E}{\partial a_j^{(L-1)}} = \sum_{i=1}^{N^{(l-1)}} \frac{\partial E}{\partial a_i^{(L)}} \frac{\partial a_i^{(L)}}{\partial z_i^{(L)}} \frac{\partial z_i^{(L)}}{\partial a_j^{(L-1)}}$$

The sum is added, as the activation from every neuron a_j from layer L-1 will affect all the activations of the neurons in layer L, which will affect the cost of the neural network.

11.3.9.2.2 Computation with Respect to Layer *L*-2

$$\frac{\partial \boldsymbol{E}}{\partial \boldsymbol{w_{ij}}^{(\boldsymbol{L-2})}} = \frac{\partial \boldsymbol{E}}{\partial \boldsymbol{a_i}^{(\boldsymbol{L-1})}} \frac{\partial \boldsymbol{a_i}^{(\boldsymbol{L-1})}}{\partial \boldsymbol{z_i}^{(\boldsymbol{L-1})}} \frac{\partial \boldsymbol{z_i}^{(\boldsymbol{L-1})}}{\partial \boldsymbol{w_{ij}}^{(\boldsymbol{L-2})}}$$

We can see clearly from the above that the error in a layer *l* depends on the error in the next layer *l* + *1*; hence, the errors propagate backward from the last to the first layer. Once we compute the error at the output layer and once the partial derivatives for all the neurons are known, the weights can be updated. The process is repeated until convergence.

Note that strictly speaking, the term "backpropagation" refers not to the learning process but to the method used to compute the gradient [10].

11.3.10 Backpropagation Algorithm

The backpropagation algorithm runs in four steps:

1. Forward phase: Proceeding from the input layer to the output layer, for each input-output pair in the training dataset, calculate the predicted output and save the result for each neuron.
2. Backward phase: Proceeding from the output layer to the input layer, calculate and save the resulting gradients.
3. Combine the individual gradients to obtain the total gradient.
4. Update the weights using α and total gradient.
5. Repeat until the minimum cost is reached.

11.4 Final Notes: Advantages, Disadvantages, and Best Practices

Neural networks are considered one of most prominent ML models given its ability to deal with different type of outputs, including discrete, real-value, vectors, images, and many others. Those models can learn and model complex, nonlinear and highly volatile data. Their architecture allows those models to be robust to any noises during the training period. Even with long training period, neural networks can generate interesting results. Note that neural networks can also be used for anomaly detection (even if we are dealing with unlabeled data); in this case, the learning results can be used to give fast second opinion with good accuracy in any used application.

Like an ML model, neural networks need parallel processing power, which makes it hardware dependence in a way. Although it gives promising results, the latter are unexplainable in many cases in terms of why and how we reached such decisions which might affect the trust in such models. In terms of its technical structure, there is no well-defined rule on how to design such architecture (number of hidden layers, number of hidden nodes, error thresholds for best training time and optimal results); it is more of trial-and-error process

With this in mind, we tend to depend on best practices to try and optimize the neural networks results. Some of key practices include the following:

- Always check the size of the training data; if it is not enough, it is important to increase.
- If the model overfits, you can either use simpler network (a smaller number of hidden layers/nodes), use dropout layers, increase data samples, or remove some features (execute preprocessing of data again).
- If the mode underfits, you can add more features (using feature engineering techniques).
- Starting with large batch size can reduce the training time in some cases.
- If the model suffers from vanishing gradient problem, using lower learning rate might allow the model to converge.
- Normalizing the inputs in every layer might help the stability and performance of the model.

11.5 Key Terms

1. Artificial neural networks (ANN)
2. McCulloch–Pitts (M-P) neuron
3. Perceptron
4. Linear function
5. Linear model

6. Bipolar activation functions
7. Unipolar activation functions
8. Sigmoid function
9. Hyperbolic tangent function
10. Tanh function
11. Rectified unit function
12. ReLU function
13. Leaky ReLU function
14. Parameterized ReLU function
15. Swish function
16. SoftMax function
17. Training the perceptron
18. Gradient descent
19. Stochastic gradient descent
20. XOR
21. Exclusive OR
22. Multilayer perceptron
23. MLP
24. Backpropagation
25. Chain rule
26. Fully connected neural network

11.6 Test Your Understanding

1. Can we identify a perceptron as a linear classifier or a nonlinear one?
2. What type of problems does a perceptron solve?
3. Why should the activation function of a multilayer perceptron be nonlinear?
4. What is the aim of backpropagation?
5. Explain backpropagation in simple words for a specialist.
6. The hyperbolic tangent function overcomes a problem we find in the sigmoid functions. What is it?
7. Why does ReLU perform better than tanh and sigmoid functions?
8. Explain the "dead" neuron problem and how to overcome it.
9. What kind of issues does a leaky ReLU overcome in comparison with a ReLU?
10. SoftMax is very useful to solve a specific kind of problem; what is it?

11.7 Read More

1. Cao, J., Qian, S., Zhang, H., Fang, Q., & Xu, C. (2021). Global Relation-Aware Attention Network for Image-Text Retrieval Proceedings of the 2021 International Conference on Multimedia Retrieval, Taipei, Taiwan. https://doi.org/10.1145/3460426.3463615

2. Chatterjee, B., & Sen, S. (2021). Energy-Efficient Deep Neural Networks with Mixed-Signal Neurons and Dense-Local and Sparse-Global Connectivity Proceedings of the 26th Asia and South Pacific Design Automation Conference, Tokyo, Japan. https://doi.org/10.1145/3394885.3431614
3. Collins, J., Sun, S., Guo, C., Podgorsak, A., Rudin, S., & Bednarek, D. R. (2021). Estimation of Patient Eye-Lens Dose During Neuro-Interventional Procedures using a Dense Neural Network (DNN). Proc SPIE Int Soc Opt Eng, 11,595. https://doi.org/10.1117/12.2580723
4. Grasemann, U., Peñaloza, C., Dekhtyar, M., Miikkulainen, R., & Kiran, S. (2021). Predicting language treatment response in bilingual aphasia using neural network-based patient models. Sci Rep, 11(1), 10,497. https://doi.org/10.1038/s41598-021-89443-6
5. Hasan, N. (2021). A Hybrid Method of Covid-19 Patient Detection from Modified CT-Scan/Chest-X-Ray Images Combining Deep Convolutional Neural Network And Two- Dimensional Empirical Mode Decomposition. Comput Methods Programs Biomed Update, 1, 100,022. https://doi.org/10.1016/j.cmpbup.2021.100022
6. Kimura, Y., Kadoya, N., Oku, Y., Kajikawa, T., Tomori, S., & Jingu, K. (2021). Error detection model developed using a multi-task convolutional neural network in patient-specific quality assurance for volumetric-modulated arc therapy. Med Phys. https://doi.org/10.1002/mp.15031
7. Maharaj, S., Qian, T., Ohiba, Z., & Hayes, W. (2021). Common Neighbors Extension of the Sticky Model for PPI Networks Evaluated by Global and Local Graphlet Similarity. IEEE/ACM Trans. Comput. Biol. Bioinformatics, 18(1), 16–26. https://doi.org/10.1109/tcbb.2020.3017374
8. Minagawa, A., Koga, H., Sano, T., Matsunaga, K., Teshima, Y., Hamada, A., Houjou, Y., & Okuyama, R. (2021). Dermoscopic diagnostic performance of Japanese dermatologists for skin tumors differs by patient origin: A deep learning convolutional neural network closes the gap. J Dermatol, 48(2), 232–236. https://doi.org/10.1111/1346-8138.15640
9. Pan, Q., Zhang, L., Jia, M., Pan, J., Gong, Q., Lu, Y., Zhang, Z., Ge, H., & Fang, L. (2021). An interpretable 1D convolutional neural network for detecting patient-ventilator asynchrony in mechanical ventilation. Comput Methods Programs Biomed, 204, 106,057. https://doi.org/10.1016/j.cmpb.2021.106057
10. Shihada, B., Elbatt, T., Eltawil, A., Mansour, M., Sabir, E., Rekhis, S., & Sharafeddine, S. (2021). Networking research for the Arab world: from regional initiatives to potential global impact. Commun. ACM, 64(4), 114–119. https://doi.org/10.1145/3447748
11. Shorfuzzaman, M., Masud, M., Alhumyani, H., Anand, D., & Singh, A. (2021). Artificial Neural Network-Based Deep Learning Model for COVID-19 Patient Detection Using X-Ray Chest Images. J Healthc Eng, 2021, 5,513,679. https://doi.org/10.1155/2021/5513679
12. Sridhara, S., Wirz, F., Ruiter, J. d., Schutijser, C., Legner, M., & Perrig, A. (2021). Global Distributed Secure Mapping of Network Addresses Proceedings of the ACM SIGCOMM 2021 Workshop on Technologies, Applications,

and Uses of a Responsible Internet, Virtual Event, USA. https://doi.org/10.1145/3472951.3473503
13. Valizadeh, A., Jafarzadeh Ghoushchi, S., Ranjbarzadeh, R., & Pourasad, Y. (2021). Presentation of a Segmentation Method for a Diabetic Retinopathy Patient's Fundus Region Detection Using a Convolutional Neural Network. Comput Intell Neurosci, 2021, 7,714,351. https://doi.org/10.1155/2021/7714351
14. Xiao, Y., Wang, X., Li, Q., Fan, R., Chen, R., Shao, Y., Chen, Y., Gao, Y., Liu, A., Chen, L., & Liu, S. (2021). A cascade and heterogeneous neural network for CT pulmonary nodule detection and its evaluation on both phantom and patient data. Comput Med Imaging Graph, 90, 101,889. https://doi.org/10.1016/j.compmedimag.2021.101889
15. Zhong, Y. W., Jiang, Y., Dong, S., Wu, W. J., Wang, L. X., Zhang, J., & Huang, M. W. (2021). Tumor radiomics signature for artificial neural network-assisted detection of neck metastasis in patient with tongue cancer. J Neuroradiol. https://doi.org/10.1016/j.neurad.2021.07.006
16. Zhu, Y., Xie, R., Zhuang, F., Ge, K., Sun, Y., Zhang, X., Lin, L., & Cao, J. (2021). Learning to Warm Up Cold Item Embeddings for Cold-start Recommendation with Meta Scaling and Shifting Networks. In Proceedings of the 44th International ACM SIGIR Conference on Research and Development in Information Retrieval (pp. 1167–1176). Association for Computing Machinery. https://doi.org/10.1145/3404835.3462843

11.8 Lab

11.8.1 Working Example in Python

The diabetes dataset that is used in this lab can be downloaded from the following link: https://www.kaggle.com/datasets/uciml/pima-indians-diabetes-database

This is a binary classification problem. This dataset contains the following information:

- Pregnancies: number of pregnancies
- Glucose: plasma glucose concentration
- Blood Pressure: diastolic blood pressure measurement
- SkinThikness: triceps skinfold thickness (mm)
- Insulin: 2-hour serum insulin
- BMI: body mass index (BMI)
- DiabetesPedigreeFunction: diabetes pedigree function
- Age: the person's age
- Outcome: tested positive for diabetes or not (1 or 0)

11.8.1.1 Load Diabetes for Pima Indians Dataset

Before loading the Pima Indians dataset, it is important to note that we need to install the Keras, TensorFlow, and SciKeras libraries using the pip install command to create the sequential neural network model.

For visualizing neural network, the Graphviz library is used. Graphviz Python can be downloaded from the following link: https://www.graphviz.org/download/

After downloading Graphviz, the path in the system environment variables needs to be edited to include:

C:\Program Files\Graph viz.\bin

C:\Program Files\Graphviz\bin\dot.exe

We start by importing the required libraries and loading the dataset and display a bar chart for the outcomes a well as pair plots for the features (Fig. 11.27). The displayed graphs are partially shown in Fig. 11.28.

11.8.1.2 Visualize Data

We explore the data visually (Fig. 11.28).

11.8.1.3 Split Dataset into Training and Testing Datasets

The next task is to choose features and target (the "Outcome"). The next step is to split the dataset into training and testing and standardize both (Fig. 11.29).

```
# Imports Required Libraries
import numpy as np
import pandas as pd # data processing
import matplotlib.pyplot as plt
import seaborn as sns
import ann_visualizer

#import warnings
#warnings.filterwarnings('ignore')
from sklearn.metrics import average_precision_score
from sklearn.metrics import precision_recall_curve
from sklearn.metrics import confusion_matrix, roc_curve,accuracy_score,roc_auc_score,classification_report,auc
from sklearn.metrics import f1_score
%matplotlib inline
from keras.models import Sequential
from keras.layers import Dense
from sklearn.model_selection import train_test_split

import numpy as np

# Load Pima Indians Diabetes dataset
df = pd.read_csv('diabetes.csv')
```

```
#Data Visualisation
sns.displot(df['Outcome'],rug=True)
plt.show()

p=sns.pairplot(df, hue = 'Outcome')
```

Fig. 11.27 Load Pima Indians diabetes dataset

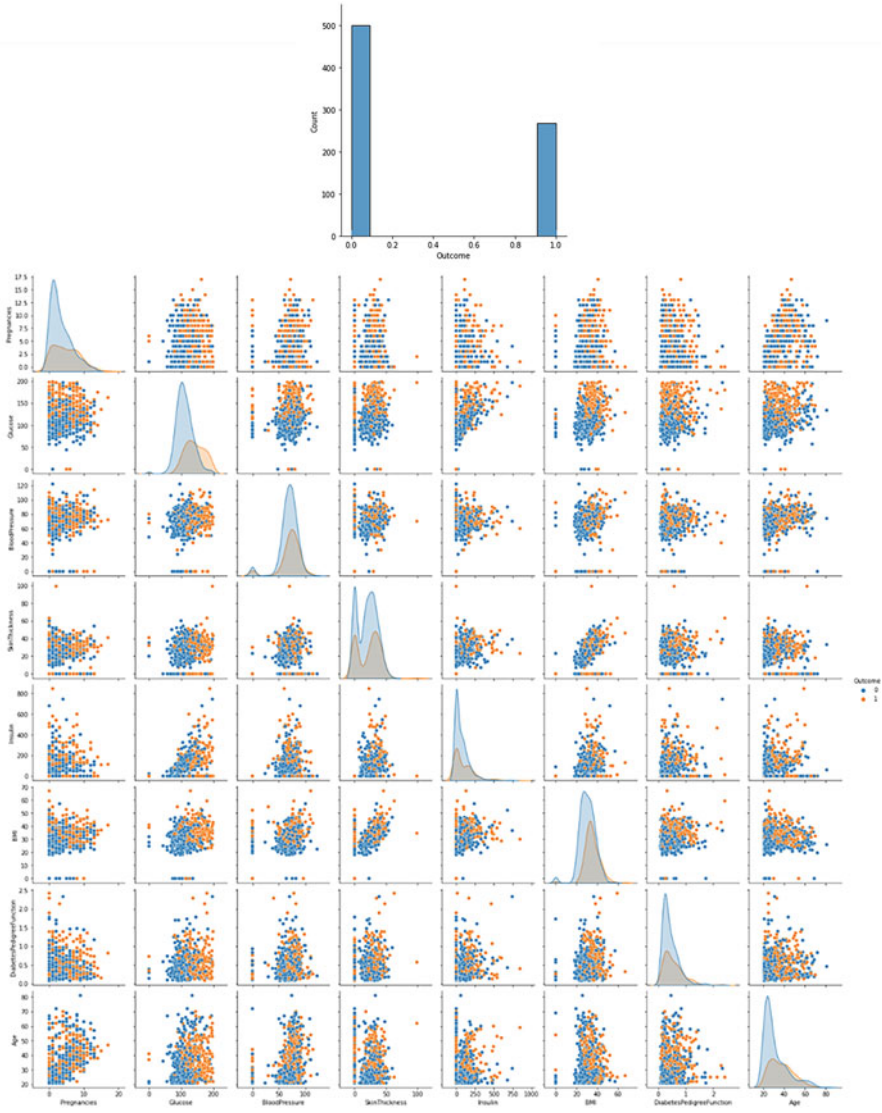

Fig. 11.28 Visualizing diabetic vs. nondiabetic Pima Indians

11.8.1.4 Create Neural Network Model

The next task is to create the sequential neural network model using the Keras library. As expected the input layer has eight nodes to accommodate the 8 features. WE have chosen to add two hidden layers are added, one with 10 nodes and the other with eight nodes (Fig. 11.30).

We can display the NN structure using the graphviz library (Fig. 11.31).

```
# prepate the feature vector x and the outcome vector y
X = df.drop(['Outcome'], axis = 1)
Y = df['Outcome'].values

#split the data into training and testing datasets
(x_train, x_test, y_train, y_test) = train_test_split(X, Y, test_size=0.3, random_state=40)
x_train


# Prepare the scaler
from sklearn.preprocessing import StandardScaler
sc = StandardScaler()

# fit the scaler on the training data only (i.e., the parameters of the standardization should use the training data only)
#  Note : the testing dataset should never be used for fit the scaler
sc.fit(x_train)

x_train = sc.transform(x_train) #Scale the traning dataset with the Scaler fitted on the training parameters
x_test = sc.transform(x_test) #Scale the testing dataset with the Scaler fitted on the training parameters
```

Fig. 11.29 Splitting and scaling Pima Indians diabetes dataset

```
# create the model
modl = Sequential()

#add the input layer with 8 nodes and ReLU activation
modl.add(Dense(10, input_dim=8, activation='relu'))

#add one middle layer with 8 nodes and ReLU activation
modl.add(Dense(8, activation='relu'))

#add the input layer with 1 node anda sigmoid function
modl.add(Dense(1,  activation='sigmoid'))

# compile the model
modl.compile(loss='binary_crossentropy', optimizer='adam', metrics=['accuracy'])

# fit the model
# NOTE: verbose is set to 1 by default, try removing it or setting it to 1 and see the result
modl.fit(X_train, Y_train, epochs=150, batch_size=10, verbose=0)

# use the NN model to predict using the training dataset
y_train_pred = modl.predict(X_train)
```

```
17/17 [==============================] - 0s 821us/step
```

Fig. 11.30 Creating sequential neural network model

11.8.1.5 Optimize Neural Network Model Using Hyperparameter

For model optimization, we use the grid search cross-validation approach (Fig. 11.32). The hyperparameters used for the grid search are the batch size and the number of epochs. We conclude that the model has fair performance (AUC = 72% and accuracy 75%) and can be used on an unseen dataset.

11.8.2 Working Example in Weka

Download the Boston housing dataset from the following website:
https://www.kaggle.com/prasadperera/the-boston-housing-dataset

```
#Visualise Neural Network Using Keras library
from ann_visualizer.visualize import ann_viz
import graphviz

# generate a NN graph and save it in  a file in Graphviz format with  extension .gv
ann_viz(modl,filename='nn_model.gv',title='Neural Network')

#read the .gv file (i.e., the NN graph) into the variablle gfile
gfile = graphviz.Source.from_file('nn_model.gv')

#display the model image (in gfile) on the screen
gfile
```

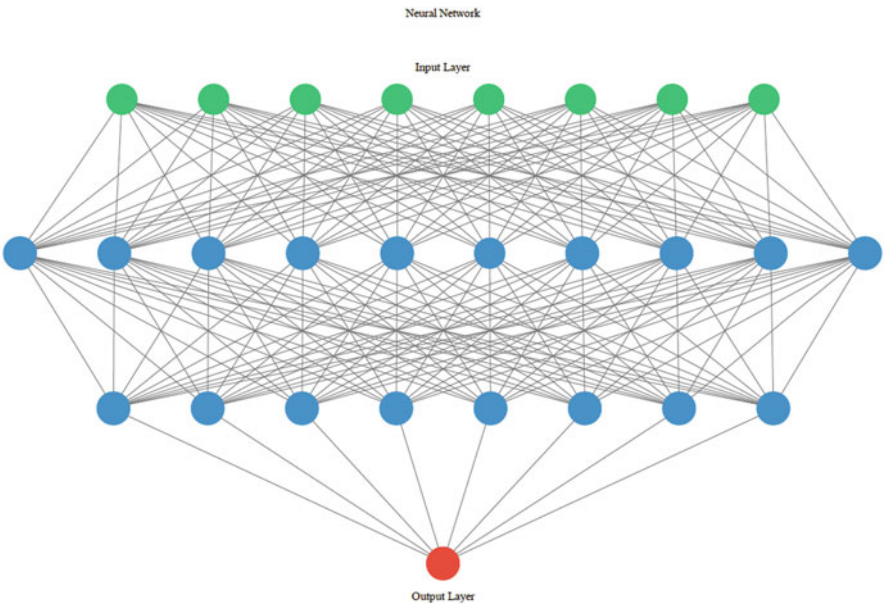

Fig. 11.31 Displaying a sequential neural network model

Open the file in Weka, go to the Classify tab, and choose the Multilayer Perceptron algorithm from Functions (Fig. 11.33).

Click on the function and notice the parameters for the algorithm (Fig. 11.34).

One of the most important parameters is the number of hidden layers; it is set to automatic by default (i.e., the letter *a* denotes automatic), but it can be set to any number you want. The learning rate can be changed; the default is 0.3. Click on GUI and make it True, then click OK, then click Start to run the algorithm. The result is shown in Fig. 11.35, and its graphical representation is shown in Fig. 11.36.

```
#Optimize the model: finetune the hyperparameter
from scikeras.wrappers import KerasClassifier, KerasRegressor
from sklearn.model_selection import GridSearchCV
optimal = KerasClassifier(model=mod1)
params={'batch_size':[100, 20, 50, 25, 32],
        'epochs':[100, 150]
       }

optimalmodel=GridSearchCV(estimator=optimal, param_grid=params, cv=3)
optimalmodel.fit(x_train, y_train, verbose=0)


optimalmodel_pred = optimalmodel.predict(x_test)
optimalmodel_auc_roc = accuracy_score(y_test, optimalmodel_pred.round())*100
print("Best parameters: ", optimalmodel.best_params_)
print("Best accuracy score: ", optimalmodel.best_score_ *100)
print("Optimal AUC on the testing dataset:%. ", optimalmodel_auc_roc)


#Create and print the Confusion Matrix for the model using testing dataset
print('Confusion Matrix for The Model Using Testing dataset:')
confusionmatrix= confusion_matrix(y_test, optimalmodel_pred.round())
sns.heatmap(confusionmatrix,annot=True,cmap="Blues",fmt="d",cbar=False)
```

```
6/6 [==============================] - 0s 3ms/step
INFO:tensorflow:Assets written to: C:\Users\elmorr\AppData\Local\Temp\tmpu3e82ps7\assets
INFO:tensorflow:Assets written to: C:\Users\elmorr\AppData\Local\Temp\tmpfhqtqmqd\assets
10/10 [==============================] - 0s 2ms/step
Best parameters:  {'batch_size': 25, 'epochs': 100}
Best accuracy score:  75.60521415270017
Optimal AUC on the testing dataset:%.  73.59307359307358
Confusion Matrix for The Model Using Testing dataset:
[6]: <AxesSubplot:>
```

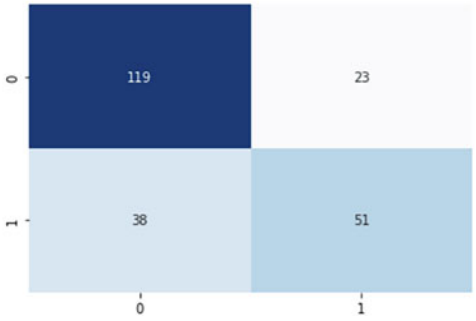

Fig. 11.32 Optimize the neural network model using grid search and its performance

11.8.3 Do it Yourself

11.8.3.1 Diabetes Revisited

How can you enhance the results of the Neural Network above? Hint: think of changing the number of hidden layers, and the number of nodes in each. We have used above standardization but neural network expects values between 0 and 1, would normalization allow the NN to perform better?

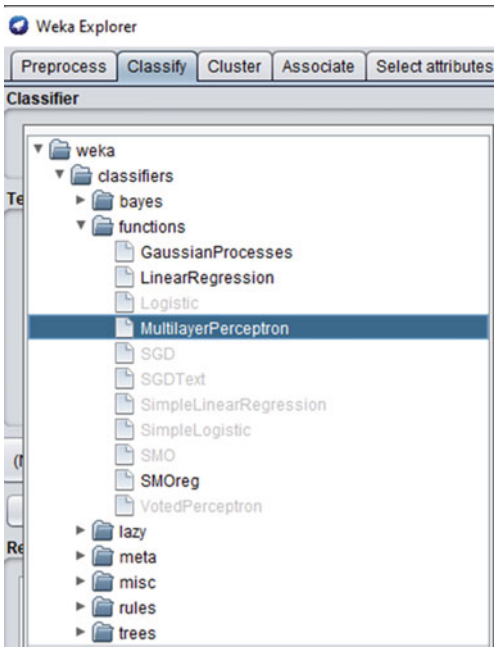

Fig. 11.33 Weka multilayer perceptron algorithm

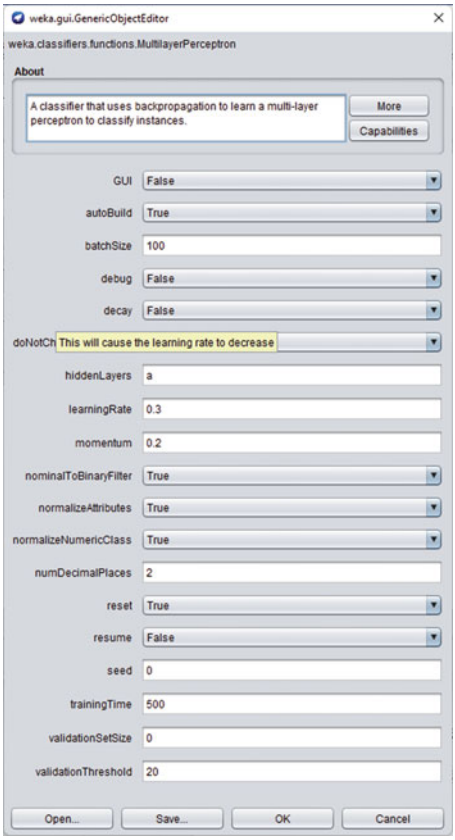

Fig. 11.34 Multilayer perceptron parameters window in Weka

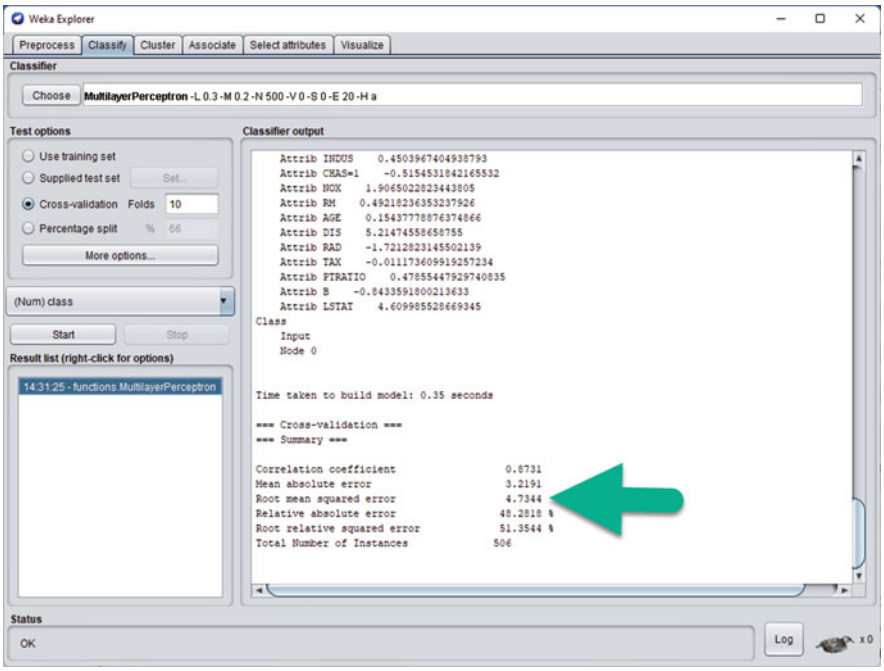

Fig. 11.35 MLP results in Weka; we can notice RMSE = 4.73

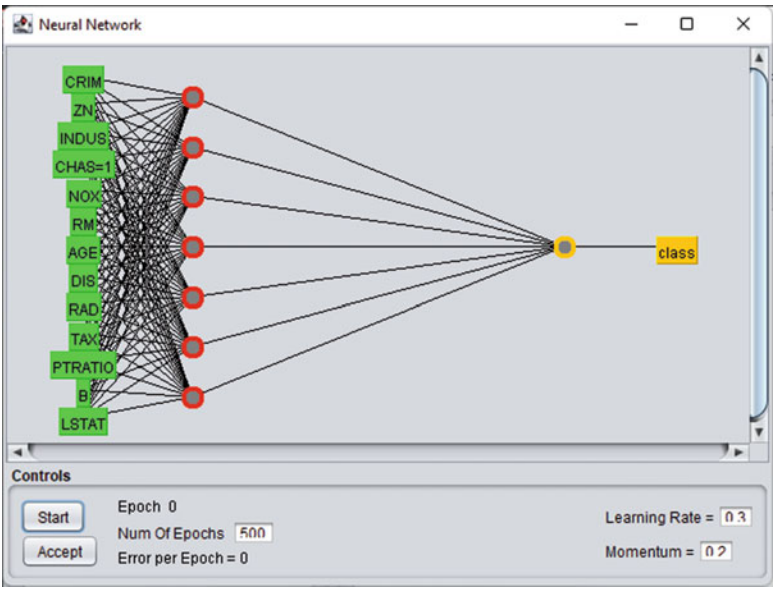

Fig. 11.36 The neural network's graphical representation

11.8.3.2 Choose your Own Problem

Pick a problem of your own and apply NN. Discuss the results with another person. Compare your result with someone else who used NN to solve the same problem. Note the differences in the results.

11.8.4 Do More Yourself

Solve each of the following predictive problems using neural networks.

1. Boston house prices:
   You can load the Boston house prices data file (as well as many other datasets) from within python by writing: boston = dataset.load_boston()
2. Predicting stock prices using neural networks.
   Download the dataset from https://www.kaggle.com/datasets/paultimothymooney/stock-market-data
3. Handwritten digit recognition.
   Download the dataset from http://yann.lecun.com/exdb/mnist/

References

1. Z.H. Zhou, *Introduction, Ensemble Methods: Foundations and Algorithms* ((Chapman & Hall/CRC Machine Learning & Pattern Recognition Series: CRC Press, 2012)
2. M. Gopal, *Applied Machine Learning* (McGraw-Hill Education, 2018)
3. T. Trappenberg, *Fundamentals of Machine Learning* (OUP, Oxford, 2019)
4. W.S. McCulloch, W. Pitts, A logical calculus of the ideas immanent in nervous activity. Bull. Math. Biophys. **5**(4), 115–133 (1 Dec 1943). https://doi.org/10.1007/BF02478259
5. G. Palm, Warren McCulloch and Walter Pitts: A logical calculus of the ideas immanent in nervous activity, in *Brain Theory*, (Springer, Berlin, Heidelberg, 1986), pp. 229–230
6. D. Gupta, *Fundamentals of deep learning – activation functions and when to use them?* https://www.analyticsvidhya.com/blog/2020/01/fundamentals-deep-learning-activation-functions-when-to-use-them/#. Accessed 7 Sep 2021
7. Brilliant.org, *Backpropagation*. Brilliant.org. https://brilliant.org/wiki/backpropagation/. Accessed 25 Aug 2021
8. A. Burkov, *The Hundred-Page Machine Learning Book* (Andriy Burkov, 2019)
9. S. Kostadinov, Understanding Backpropagation Algorithm: Learn the nuts and bolts of a neural network's most important ingredient. https://towardsdatascience.com/understanding-backpropagation-algorithm-7bb3aa2f95fd. Accessed 7 Sep 2021
10. I. Goodfellow, Y. Bengio, A. Courville, *Deep Learning* (MIT Press, 2016)