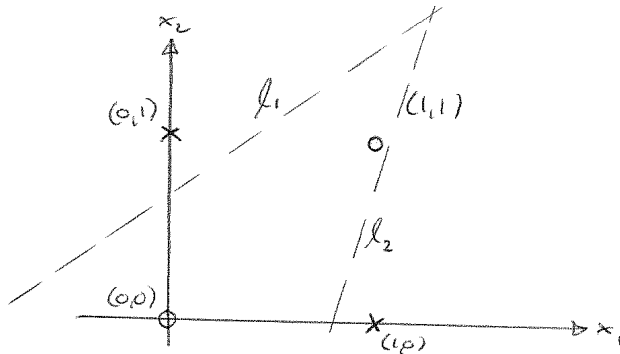


# Adaptive Computation and Machine Learning

## 2. NEURAL NETWORKS

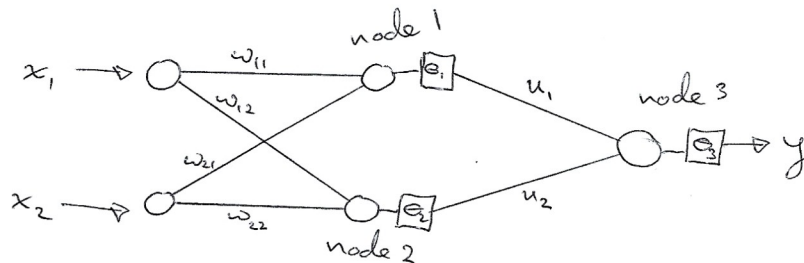
### 2.1. Multi-Layer Perceptrons.

The dataset  $D = \{((0,0),0), ((0,1),1), ((1,0),1), ((1,1),0)\}$  is not linearly separable, i.e., it is impossible to draw a straight line in  $\mathbb{R}^2$  so that all the 1's are on one side of the line and all the 0's are on the other. However, it is possible to separate the dataset using **two** straight lines, as shown below, where a  $\circ$  represents a target of 0 and a  $\times$  represents a target of 1.



If a point lies above line  $\ell_1$  and above line  $\ell_2$  then it is a  $\times$ . If it lies below  $\ell_1$  and above  $\ell_2$  then it is a  $\circ$ ; and if it lies below line  $\ell_1$  and below line  $\ell_2$  then it is a  $\times$ .

A 2-layer perceptron can be used to model the separation using two straight lines as follows:



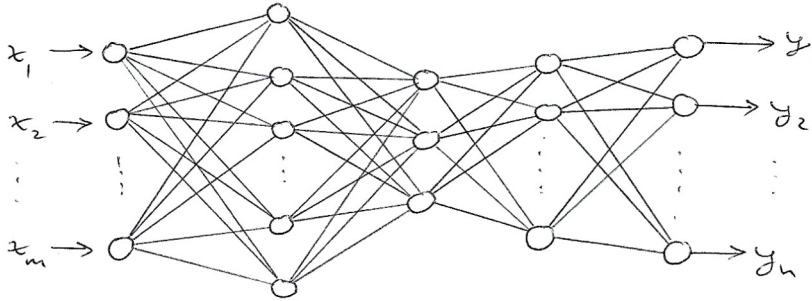
We shall often use the word **node** in place of artificial neuron.

In the above figure, node 1 represents line  $\ell_1$ , node 2 represent line  $\ell_2$  and node 3 decides in which region the point lies, for example, above  $\ell_1$  and above  $\ell_2$ , or below  $\ell_1$  and above  $\ell_2$ , etc.

*EXERCISE:* Try to find suitable values for  $w_{11}, w_{12}, w_{21}, w_{22}, u_1, u_2, \theta_1, \theta_2$  and  $\theta_3$  so that the above 2-layer perceptron correctly outputs the data in  $D$ .

By a **multilayer perceptron** we mean a network with an input layer followed by number of layers of nodes followed by an output layer, as shown below.

Every edge has a real number **weight** assigned to it and every node has its own threshold value.



Training a multilayer perceptron with 2 layers is far more complicated than a simple perceptron as the straight lines need to be ‘moved’ into correct positions and at the same time the weights on the second layer that control the regions defined by the straight lines need to be learned. If there are more layers then it becomes even more complicated.

We are going to use a learning rule based on the gradient descent method, which is an optimisation technique that makes use of the derivative of a function with respect to its variables. Note that a multilayer perceptron with  $m$  input nodes and  $n$  output nodes is a function from  $\mathbb{R}^m$  to  $\mathbb{R}^n$ . The weights and thresholds are the adaptable parameters of this function.

A difficulty for applying gradient descent in multilayer perceptrons is the use of ‘thresholding’ because this has derivative 0 and so the gradient descent method is of no use. For this reason, the threshold function will be replaced with other activation functions, as discussed below.

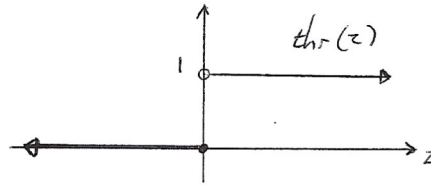
## 2.2. Activation functions.

The threshold function in perceptrons can be rewritten in the following way.

Instead of comparing  $\sum_i x_i w_i$  with a threshold value  $\theta$ , we first compute  $\sum_i x_i w_i - \theta$  and then compare the result with 0. That is, we apply a **threshold** function  $thr$ , defined by:

$$thr(z) = \begin{cases} 1 & \text{if } z > 0 \\ 0 & \text{if } z \leq 0 \end{cases}$$

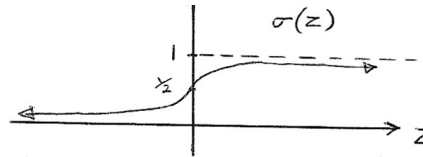
The graph of  $thr$  is shown below; as you can see its derivative is zero everywhere (except at  $z = 0$ , where it is not differentiable):



A way around the zero derivative problem is to replace  $thr$  with the following **sigmoid** function (sometimes called **logistic sigmoid**):

$$\sigma(z) = \frac{1}{1 + e^{-z}}.$$

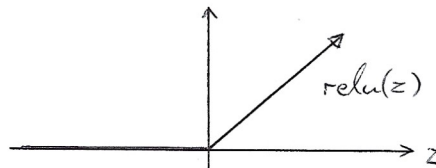
The sigmoid function is a continuous approximation to  $thr$ ; the graph of  $\sigma(z)$  looks like:



Another commonly used function is  $relu$ , or **rectified linear unit**, which is defined as:

$$relu(z) = \begin{cases} z & \text{if } z > 0 \\ 0 & \text{if } z \leq 0 \end{cases}$$

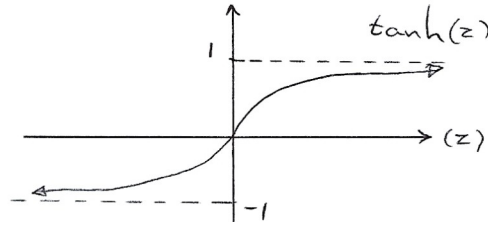
The  $relu$  function is also a variation of the  $thr$  function; the graph of  $relu$  looks like:



A function which is often used if the output is required to be scaled between  $-1$  and  $1$  is the  $tanh$  function, or **hyperbolic tangent**, defined by:

$$tanh(z) = \frac{e^z - e^{-z}}{e^z + e^{-z}}$$

The graph of  $tanh(z)$  looks like:



Sometimes the **linear** function  $lin$  is used, which is just the identity function:  $lin(z) = z$ .

The term **activation function** is used for functions like  $\sigma$ ,  $relu$ ,  $tanh$  and  $lin$ .

We use the symbol  $g$  to represent an arbitrary activation function.

The above activation functions are not the only possible ones. There are variants of the above functions which are often used, as well as completely different types of activation function.

### 2.3. Activation value of a node.

We describe here the structure of nodes, or artificial neurons, which may use any activation function and how to compute the activation value of a node.

Every node has a **bias** value  $b$ , which can be any real number (positive or negative).

The bias replaces the  $-\theta$  term in the expression  $\sum_i x_i w_i - \theta$ .

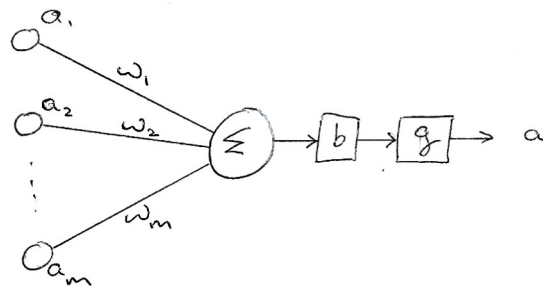
$b$  plays a similar role to  $\theta$ , but we prefer to work with  $+b$  rather than  $-\theta$ .

Every node has an **activation function**, say  $g$ .

Usually,  $g$  is one of the following:  $\sigma$ ,  $relu$ ,  $tanh$  or  $lin$ .

At every node, the **activation value** at that node is computed as follows.

Consider all nodes that have an edge connecting to this node. Let  $a_1, \dots, a_m$  be the activation values at those nodes (or input values if they are input nodes), and let  $w_1, \dots, w_m$  be the weights of the corresponding edges, as in the figure below.



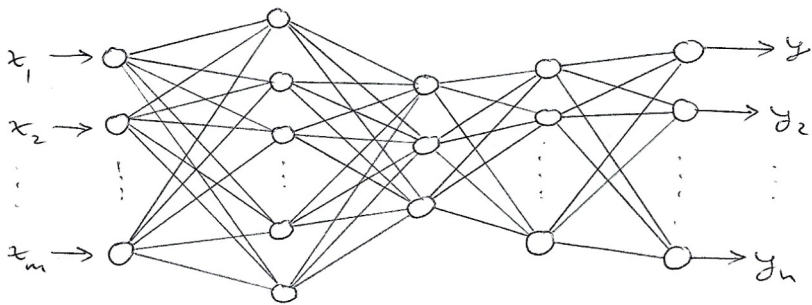
The **activation value**, or output, of the node is calculated as follows

- 1) calculate the following sum:  $\sum a_i w_i$
- 2) add the **bias** value of the node:  $\sum a_i w_i + b$
- 3) apply the **activation function**:  $g(\sum a_i w_i + b)$ .

The activation value of the node is then  $a = g(\sum a_i w_i + b)$ .

## 2.4. Neural Networks.

An (artificial) **neural network** is network consisting of nodes and edges; the nodes are usually organised into layers with edges connecting nodes in consecutive layers, as shown below. The first layer is called the input layer, which is followed by a number of layers of nodes, with the last layer called the output layer.



The middle layers are usually called **hidden layers**.

Every edge has a real number **weight** assigned to it.

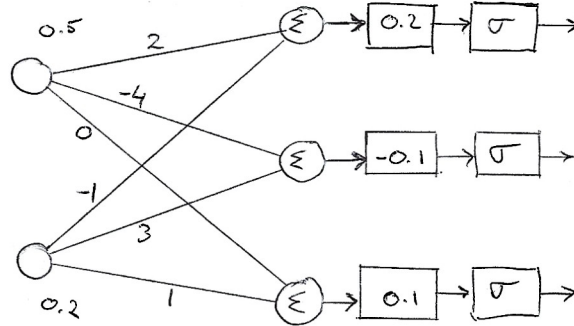
Every node has its own **bias value** and **activation function**. Usually, all the nodes within a given layer use the same activation function; however, different layers can use different activation functions.

In the above figure, every node in a layer has an edge to every node in the next layer – in such a case, the neural network is said to be **fully connected**.

The weights of the edges between one layer and the next are usually represented by a matrix and the bias values of the nodes in a layer are represented by a list.

**Example:** In this example we show how the activation values of a layer of nodes in a neural network are computed. The following figure represents two consecutive layers within a neural

network. The weights of each edge are shown in the figure and the bias values of nodes in the second layer are shown in the square boxes. The activation value at each node is the sigmoid function  $\sigma$ . Suppose that 0.5 and 0.2 are the activation values of the two nodes in the left layer.



The activation values at the three nodes in the second layer are obtained as follows:

(Node 1 is the top node in the figure and node 3 is the bottom node.)

node 1:  $(0.5)(2) + (0.2)(-1) = 0.8 \rightarrow$  (add bias)  $0.8 + 0.2 = 1 \rightarrow \sigma(1) = 0.731$

node 2:  $(0.5)(-4) + (0.2)(3) = -1.4 \rightarrow$  (add bias)  $-1.4 + (-0.1) = -1.5 \rightarrow \sigma(-1.5) = 0.182$

node 3:  $(0.5)(0) + (0.2)(1) = 0.2 \rightarrow$  (add bias)  $0.2 + 0.1 = 0.3 \rightarrow \sigma(0.3) = 0.574$

Thus, the activation values at the three nodes are given by (0.731, 0.182, 0.574).

Next, we show how to compute the same activation values if we represent the weights by a matrix. Let

$$W = \begin{bmatrix} 2 & -4 & 0 \\ -1 & 3 & 1 \end{bmatrix}.$$

Then  $W$  contains all the weights of the edges between the two layers in the figure above. Using the notation  $W = [w_{ij}]$ , the entry  $w_{ij}$  is the weight of the edge between node  $i$  in the left layer and node  $j$  in the right layer.

We can write the bias values as  $\mathbf{b} = (0.2, -0.1, 0.1)$ .

If we write the activation values of the nodes in the left layer as  $\mathbf{a} = (0.5, 0.2)$ , then the activation values at the right layer are computed as follows:

$$\mathbf{a}W = (0.5, 0.2) \begin{bmatrix} 2 & -4 & 0 \\ -1 & 3 & 1 \end{bmatrix} = (0.8, -1.4, 0.2).$$

Next, add the bias:

$$\mathbf{a}W + \mathbf{b} = (0.8, -1.4, 0.2) + (0.2, -0.1, 0.1) = (1, -1.5, 0.3).$$

Lastly, apply the activation function:

$$\sigma(\mathbf{a}W + \mathbf{b}) = \sigma(1, -1.5, 0.3) = (\sigma(1), \sigma(-1.5), \sigma(0.3)) = (0.731, 0.182, 0.574).$$

## 2.5. Feedforward Step.

Given a neural network and a list of input values to the network, the corresponding list of output values can be computed. This computation is called the **feedforward step**, or **forward propagation**. The computation proceeds layer-by-layer.

The input layer is denoted as layer 0 and the list of input values is denoted by  $\mathbf{a}_0$ .

Let  $W_1$  be the matrix of edge weights between the input nodes and the first hidden layer, i.e., layer 1, let  $\mathbf{b}_1$  be the list of bias values of the nodes in layer 1, and let  $g_1$  be the activation function at layer 1. Then the activation values at layer 1 are computed as follows.

First, multiply  $\mathbf{a}_0$  with the matrix of weights  $W_1$ , then add the bias values  $\mathbf{b}_1$ , and then apply the activation function  $g_1$  to obtain the activation values at layer 1, which is denoted by  $\mathbf{a}_1$ :

$$\mathbf{a}_0 \longrightarrow \mathbf{a}_0 W_1 \longrightarrow \mathbf{a}_0 W_1 + \mathbf{b}_1 \longrightarrow g_1(\mathbf{a}_0 W_1 + \mathbf{b}_1) = \mathbf{a}_1$$

The values in  $\mathbf{a}_1$  are the input into the next layer and the pattern repeats with the next layer's weights, biases and activation function. The general pattern is as follows.

Let  $W_k$  be the matrix of weights between layer  $k-1$  and layer  $k$ , let  $\mathbf{b}_k$  be the list of bias values at layer  $k$ , and  $g_k$  the activation function at layer  $k$ . If  $\mathbf{a}_{k-1}$  are the activation values at layer  $k-1$ , then the activation values at layer  $k$ , that is  $\mathbf{a}_k$ , are calculated as follows:

$$\mathbf{a}_{k-1} \longrightarrow \mathbf{a}_{k-1} W_k \longrightarrow \mathbf{a}_{k-1} W_k + \mathbf{b}_k \longrightarrow g_k(\mathbf{a}_{k-1} W_k + \mathbf{b}_k) = \mathbf{a}_k$$

The list of activation values at the final layer is the output of the network.

**Example:** Consider a neural network with two input nodes, one hidden layer with 3 nodes and 2 output nodes, that uses the sigmoid activation function  $\sigma$  at each node (so  $g_1 = \sigma$  and  $g_2 = \sigma$ ). Let the weights between the input layer and the hidden layer (layer 1) be given by  $W_1$  and let the bias values at layer 1 be given by  $\mathbf{b}_1$ :

$$W_1 = \begin{bmatrix} 2 & 1 & -1 \\ -2 & -4 & 1 \end{bmatrix}, \quad \mathbf{b}_1 = (1, -1, 2).$$

Let the weights between the hidden layer and the output layer (layer 2) be given by  $W_2$  and the bias values at the output layer (layer 2) be given by  $\mathbf{b}_2$ :

$$W_2 = \begin{bmatrix} -2 & 3 \\ 3 & -1 \\ 5 & 0 \end{bmatrix}, \quad \mathbf{b}_2 = (0, -1).$$

Suppose the input  $(2, 1)$  is fed into the network.

The output of the network is calculated as follows.

Set  $\mathbf{a}_0 = (2, 1)$  and calculate:

$$\mathbf{a}_0 W_1 = (2, 1) \begin{bmatrix} 2 & 1 & -1 \\ -2 & -4 & 1 \end{bmatrix} = (2, -2, -1).$$

Then add the bias values in  $\mathbf{b}_1$ :

$$(2, -2, -1) + (1, -1, 2) = (3, -3, 1).$$

Then apply the activation function  $g_1$  to get:

$$g_1(3, -3, 1) = \sigma(3, -3, 1) = (\sigma(3), \sigma(-3), \sigma(1)) = (0.953, 0.047, 0.731).$$

Thus,  $\mathbf{a}_1 = (0.953, 0.047, 0.731)$  is the list of activation values at layer 1, which forms the input into layer 2. To get the activation values at layer 2, first compute:

$$\mathbf{a}_1 W_2 = (0.953, 0.047, 0.731) \begin{bmatrix} -2 & 3 \\ 3 & -1 \\ 5 & 0 \end{bmatrix} = (1.89, 2.812).$$

Then add the bias values in  $\mathbf{b}_2$ :

$$(1.89, 2.812) + (0, -1) = (1.89, 1.812).$$

Then apply the activation function  $g_2$  to get:

$$g_2(1.89, 1.812) = \sigma(1.89, 1.812) = (\sigma(1.89), \sigma(1.812)) = (0.869, 0.860).$$

Thus, if  $(2, 1)$  is input into the network, then the output obtained is  $(0.869, 0.860)$ .



## EXERCISES

- (1) Consider a network with 4 input nodes, one hidden layer with 4 nodes and 2 output nodes. The weights between layer 0 and layer 1, and the biases at layer 1 are given by:

$$W_1 = \begin{bmatrix} 4 & -5 & 0 & 1 \\ -3 & 6 & -1 & 2 \\ 0 & 1 & 1 & -2 \\ 2 & 0 & -3 & 4 \end{bmatrix}, \quad \mathbf{b}_1 = (1, -2, 0, -1).$$

The weights between layer 1 and layer 2, and the biases at layer 2 are given by:

$$W_2 = \begin{bmatrix} -2 & 1 \\ -1 & 0 \\ 1 & -3 \\ 5 & -1 \end{bmatrix}, \quad \mathbf{b}_2 = (-2, 2).$$

- (a) Using the sigmoid activation function  $\sigma$  at each node, calculate the output from the network for the following inputs:

- (i)  $(1, 0, 2, -3)$
- (ii)  $(-2, 3, 1, 0)$
- (iii)  $(1, 0, 1, 1)$

- (b) Using the activation function *relu* at each node, calculate the output from the network with same inputs as above.

- (c) Using *relu* at the hidden layer and  $\sigma$  at the output layer, calculate the output from the network with same inputs as above.

- (2) Consider a network with 2 input nodes, 3 nodes in the first hidden layer, 3 nodes in the second hidden layer and 2 output nodes. The weights between the respective layers are given by the following matrices:

$$W_1 = \begin{bmatrix} 1 & -1 & 0 \\ -2 & 4 & -1 \end{bmatrix} \quad W_2 = \begin{bmatrix} 2 & 0 & -1 \\ 1 & -2 & -3 \\ 5 & 0 & 0 \end{bmatrix} \quad W_3 = \begin{bmatrix} 0 & 2 \\ -1 & -1 \\ 4 & -3 \end{bmatrix}$$

The bias values at the two hidden layers and output layer are given by:

$$\mathbf{b}_1 = (0, -3, 2), \quad \mathbf{b}_2 = (3, 3, -1), \quad \mathbf{b}_3 = (0, 1).$$

(a) Using the *relu* activation function at each node, calculate the output from the network for the following inputs:

(i)  $(1, -1)$

(ii)  $(3, -2)$

(b) Using the sigmoid activation function  $\sigma$  at each node, calculate the output from the network with same inputs as above.

## 2.6. Structuring the output.

Consider a dataset with inputs and corresponding targets as in the tables below:

$$\begin{array}{ccccc}
 x_1 & x_2 & \cdots & x_m & t \\
 \left[ \begin{array}{cccc} 1 & 2 & \cdots & -3 \\ -3 & -2 & \cdots & 5 \\ 0 & 1 & \cdots & -7 \\ \vdots & \vdots & \cdots & \vdots \end{array} \right] & & & & \left[ \begin{array}{c} 0 \\ 1 \\ 0 \\ \vdots \end{array} \right]
 \end{array}$$

Each input is an  $m$ -tuple  $(x_1, \dots, x_m) \in \mathbb{R}^m$  and each target value is a classification of the input into either 0 or 1, i.e.,  $t \in \{0, 1\}$ .

A possible structure of a neural network for such a dataset would have  $m$  input nodes, then some hidden layers, followed by an output layer with 1 output node.

The number of hidden layers and the number of nodes in each hidden layer, as well as the activation functions at each hidden layer, would be chosen by the user.

For the output node, a sigmoid activation function could be used, which means that the output for a given input is a value between 0 and 1.

The output then needs to be interpreted as either class 0 or class 1, which can be done as follows:

if  $0 \leq y \leq \frac{1}{2}$ , then the classification is 0,  
 if  $\frac{1}{2} < y \leq 1$ , then the classification is 1.

Another way to structure the output is to have two output nodes, where the first output node corresponds to a target of 0, and the second node corresponds to a target of 1. To do that, the

target table must be changed as follows:

$$\begin{array}{cccc} x_1 & x_2 & \cdots & x_m \\ \left[ \begin{array}{cccc} 1 & 2 & \cdots & -3 \\ -3 & -2 & \cdots & 5 \\ 0 & 1 & \cdots & -7 \\ \vdots & \vdots & \cdots & \vdots \end{array} \right] & & & \begin{array}{cc} t_1 & t_2 \\ \left[ \begin{array}{cc} 1 & 0 \\ 0 & 1 \\ 1 & 0 \\ \vdots & \vdots \end{array} \right] \end{array}
 \end{array}$$

The original 0's were changed to (1, 0) and the original 1's were change to (0, 1).

If the sigmoid activation function is used at the two output nodes, then the output of the two nodes is a pair of values, say  $(y_1, y_2)$ , in the range 0 to 1. The node which fires more strongly determines the classification:

- if  $y_1 \geq y_2$ , i.e.,  $y_1$  fires more strongly than  $y_2$ , then the classification is 0;
- if  $y_1 < y_2$ , i.e.,  $y_2$  fires more strongly than  $y_1$ , then the classification is 1.

For the next example, consider a dataset with inputs and corresponding targets as in the tables below:

$$\begin{array}{cccc} x_1 & x_2 & \cdots & x_m \\ \left[ \begin{array}{cccc} -3 & 4 & \cdots & -1 \\ 2 & -1.5 & \cdots & 6.7 \\ 0.5 & -1 & \cdots & -3 \\ \vdots & \vdots & \cdots & \vdots \end{array} \right] & & & \begin{array}{c} t \\ \left[ \begin{array}{c} a \\ b \\ c \\ \vdots \end{array} \right] \end{array}
 \end{array}$$

Each input is an  $m$ -tuple  $(x_1, \dots, x_m) \in \mathbb{R}^m$  and each target value is a classification of the input into either  $a$ ,  $b$ , or  $c$ , i.e.,  $t \in \{a, b, c\}$ .

In this case, if the output layer has a single node with the sigmoid activation function then, to interpret the output value  $y$ , some sort of rules are required, such as:

- if  $0 \leq y \leq \frac{1}{3}$ , then classify as  $a$ ,
- if  $\frac{1}{3} < y \leq \frac{2}{3}$ , then classify as  $b$ ,
- if  $\frac{2}{3} < y \leq 1$ , then classify as  $c$ .

This method can create dependencies between the different classification classes.

A better method is to have 3 output nodes in the output layer and to change the target values as follows:

each target  $a$  is replaced by target  $(1, 0, 0)$

each target  $b$  is replaced by target  $(0, 1, 0)$

each target  $c$  is replaced by target  $(0, 0, 1)$ .

Thus, the dataset looks like:

$$\begin{array}{cccc} x_1 & x_2 & \cdots & x_m \\ \left[ \begin{array}{cccc} -3 & 4 & \cdots & -1 \\ 2 & -1.5 & \cdots & 6.7 \\ 0.5 & -1 & \cdots & -3 \\ \vdots & \vdots & \cdots & \vdots \end{array} \right] & & \begin{array}{ccc} t_1 & t_2 & t_3 \\ \left[ \begin{array}{ccc} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \\ \vdots & \vdots & \vdots \end{array} \right] \end{array} \end{array}$$

To interpret the output of the network, consider the three output nodes as corresponding to the three classifications. The node that fires the strongest, i.e., the node with the maximum output value, determines the classification. For example,  $(0.9, 0.2, 0.1)$  gives a classification of  $a$ ,  $(0.6, 0.7, 0.1)$  gives a classification of  $b$ , while  $(0.3, 0.4, 0.9)$  gives a classification of  $c$ . Some extra rules may be needed for special cases such as breaking a tie, as in  $(0.5, 0.5, 0.1)$ .

Creating targets in the above way is called a **one-hot encoding** of the data.

It can be applied regardless of how many output classifications there are; an output node is added for each possible classification.

## 2.7. Regression with Neural Networks.

Suppose our dataset is of the following type:

$$\begin{array}{cccc} x_1 & x_2 & \cdots & x_m \\ \left[ \begin{array}{cccc} -1 & 3 & \cdots & -4 \\ 5 & -1 & \cdots & 6 \\ 1 & 0 & \cdots & -7 \\ \vdots & \vdots & \cdots & \vdots \end{array} \right] & & \begin{array}{c} t \\ \left[ \begin{array}{c} 3.6 \\ 2.7 \\ -4.1 \\ \vdots \end{array} \right] \end{array} \end{array}$$

In this case, the targets are not classification values but real numbers, and the goal is not to separate data into classes but to find a network that acts as a function approximator to the above data. This is called a **regression** problem, or a curve-fitting problem.

One way to deal with real target values is to use a single output node with the linear activation function *lin*. Then the output can take on any real number because the weights leading to the output node can be any real numbers. If the dataset has two or more real-valued targets, then that many output nodes can be used, each with linear activation function.

(If the outputs are all in the range  $[0, 1]$ , then the sigmoid activation function could also be used; if they are in the range  $[-1, 1]$ , then a *tanh* activation function could be used.)

Lastly, we mention the following theoretical result which provides information on what types of functions a neural network can approximate.

### **Universal Approximation Theorem**

*Any continuous real-valued function on a closed and bounded subset of  $\mathbb{R}^m$  can be approximated to any desired nonzero error by a neural network with one hidden layer that uses a sigmoid (or similar) activation function at the hidden nodes and a linear activation function at the output node.*

The above result states that basically any continuous function, as long as it's defined on a closed and bounded region of  $\mathbb{R}^m$ , can be approximated with arbitrarily small error by a neural network. In addition, the network requires only one hidden layer. However, note that the number of nodes in the hidden layer is not specified and it can be huge. The use of additional hidden layers reduces the total number of nodes required but, as we shall see, the more layers there are, the more difficult it is to train the network.