Artifical Neural Networks - Multilayer Perceptron

Ugenteraan Manogaran

E-mail: m.ugenteraan15@gmail.com

March 2019

Contents

1.0 Dataset

Suppose there is a dataset that consists of m number of data points. Each of the data points are an n-dimensional vector.

- Each of the data points can be represented as $\mathbf{x}^{(i)}$. The superscript (i) is used to denote the *i-th* data point. Hence, i $\in 1, ..., m$.
- Since each of the data points are an n-vector,

$$\mathbf{x}^{(i)} = \begin{bmatrix} x_1^{(i)} \\ x_2^{(i)} \\ . \\ . \\ . \\ x_n^{(i)} \end{bmatrix}$$

- The data set can be represented in a single n x m matrix $\mathbf{X}$ as shown below. $\mathbf{X}$ is called a data matrix.

$$\mathbf{X} = \begin{bmatrix} x_1^{(1)} & x_1^{(2)} & . & . & . & x_1^{(m)} \\ x_2^{(1)} & x_2^{(2)} & . & . & . & x_2^{(m)} \\ . & . & . & & & . \\ . & . & & . & & . \\ . & . & & & . & . \\ x_n^{(1)} & x_n^{(2)} & . & . & . & x_n^{(m)} \end{bmatrix}$$

Note : *Each of the data points could have been represented as 1* x *n matrices* $\mathbf{x}^{(i)^{\mathbf{T}}}$ *as well. That would have resulted* $\mathbf{X}$ *to be a m* x *n matrix* $\mathbf{X}^{\mathbf{T}}$ *as shown below.*

$$\mathbf{x}^{(i)^{\mathbf{T}}} = \begin{bmatrix} x_1^{(i)} & x_2^{(i)} & . & . & . & x_n^{(i)} \end{bmatrix}, \quad \mathbf{X}^{\mathbf{T}} = \begin{bmatrix} x_1^{(1)} & x_2^{(1)} & . & . & . & x_n^{(1)} \\ x_1^{(2)} & x_2^{(2)} & . & . & . & x_n^{(2)} \\ . & . & . & & & . \\ . & . & & . & & . \\ . & . & & & . & . \\ x_1^{(m)} & x_2^{(m)} & . & . & . & x_n^{(m)} \end{bmatrix}$$

However, in this note, we will be denoting each of the data points as n-vectors $\mathbf{x}^{(i)}$ and the dataset as the n x m matrix $\mathbf{X}$.

2.0 Problem Statement

Generally, datasets obtained from real-world are non-linear, noisy and the relationship between the dependent and independent variables in each dataset hardly makes any sense to humans. Identifying the appropriate nonlinear functions to model the relationship can be difficult and computationally expensive. The question is, if algorithms such as General Linear Regressions (GLR) and SVM are not good enough to extract patterns and detect trends in such data, what do we do next?

3.0 Artificial Neural Networks

Artificial Neural Networks or simply Neural Networks , are an attempt to model the information processing capabilities of a human brain. Neural Networks are a wide class of flexible nonlinear regression and discriminant models, data reduction models, and nonlinear dynamical systems. Examples of Neural Network types are Perceptrons, Multilayer Perceptron (MLP), Recurrent Neural Network (RNN), Radial Basis Function Neural Network (RBNN) and so on.

3.1 Perceptron

Perceptrons are a type of Artificial Neural Networks that are based on a unit called perceptron. A perceptron takes a vector of real-valued inputs, calculates the linear combination of these inputs, z and a possibly nonlinear function is applied on the linear combination to produce an output $g(z)$. The function applied on z is known as activation function. There are many types of activation functions that can be used. Some common activation functions g are:

- Linear or Identity: $g(x) = x$
- Hyperbolic Tangent: $g(x) = tanh(x) = \frac{(e^x - e^{-x})}{(e^x + e^{-x})}$
- Sigmoid: $g(x) = sigmoid(x) = \frac{1}{1+e^{-x}}$
- Rectified Linear Unit (ReLU): $g(x) = ReLU(x) = \begin{cases} 0 & for\ x < 0 \\ x & for\ x \geq 0 \end{cases}$
- Threshold: $g(x) = \begin{cases} 0\ for\ x < 0 \\ 1\ for\ x \geq 0 \end{cases}$

Let $\mathbf{x} = \{x_1,\ x_2,\ \ldots,\ x_N\}$ be the input to a perceptron. Every element in the vector will be scaled by a real-valued constant. Hence, there'll be N scalars. In the context of Neural Networks, these scalars are known as weights and they can be represented as $\mathbf{w} = \{w_1,\ w_2,\ \ldots,\ w_N\}$. In other words, w_i determines the contribution of x_i to the output. There is also another real-valued constant known as bias w_0 added to the perceptron to manipulate the output of an activation function without the intervention of the input.

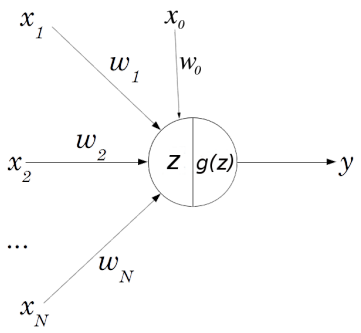

Figure 1: A perceptron

Mathematically, the operations in a perceptron can be represented as follows:

$$y = g(z)$$

where

$$z = w_0 + x_1 w_1 + x_2 w_2 + ... + x_N w_N$$

In order to simplify the notations, $x_0 = 1$ can be added as an input and scale it by w_0. Therefore,

$$z = x_0 w_0 + x_1 w_1 + x_2 w_2 + ... + x_N w_N$$

Figure 1 illustrates a perceptron. Since,

$$\mathbf{x} = \{x_0,\ x_1,\ x_2,\ \ldots,\ x_N\} \quad and \quad \mathbf{w} = \{w_0,\ w_1,\ w_2,\ \ldots,\ w_N\},$$

z can be simplified as

$$z = \mathbf{x} \bullet \mathbf{w}$$

The activation function is selected based on the use case of the perceptron. For example, in order for a perceptron to act as logic gates, we would want the output to be either 1 or 0. Otherwise the output would be 0. Therefore, the threshold function should be used as the activation function for the perceptron.

3.1.1 Perceptron as Logic Gates

Let's suppose we have two boolean inputs, x_1 and x_2. We want a perceptron to imitate the AND gate's functionality. In other words, the output of the perceptron will be 1 if and only if $x_1 = x_2 = 1$. To imitate such functionality, the weights of the perceptron can be set as such :

$$\begin{aligned} w_0 &= -6 \\ w_1 &= 4 \\ w_2 &= 4 \end{aligned}$$

With $z = -6 + 4x_1 + 4x_2$ and threshold as the activation function, Table 1 summarizes the output of such perceptron under different inputs.

x_1	x_2	z	$g(z)$
0	0	-6	0
1	0	-2	0
0	1	-2	0
1	1	2	1

Table 1: Perceptron's Output Based On The Different Boolean Inputs

A perceptron can be viewed as representing a hyperplane decision surface in the N-dimensional space of instances **i.e.** *data points*. For this AND Gate's example, based on the linear equation of z, we would have 3 dimensions to represent x_1, x_2 *and* z. Note that in a 3-dimensional space, the decision surface would be a 2-dimensional (*N-dimensional*) plane. The plane would represent all the possible values of z that results from all the possible pairs of x_1 *and* x_2 based on the values of w_0, w_1 *and* w_2.

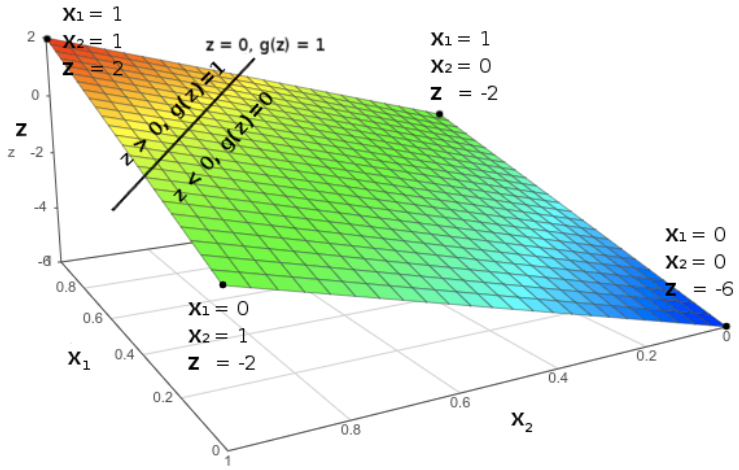

Figure 2: Decision surface and decision boundary for the AND gate perceptron

Figure 2 illustrates the decision surface and the decision boundary for the perceptron that imitates the AND gate's functionality. However, note that the input is either {0,0}, {0,1}, {1,0} or {1,1} only. The activation function acts as the decision boundary in this specific case.

By adjusting the weights appropriately, the perceptron can also imitate the functionality of an OR gate, a NAND gate and a NOR gate. **NOTE** that the weights are **NOT UNIQUE!** For example, the AND gate can also be imitated by setting $w_0 = -5,\ w_1 = 4\ and\ w_2 = 4$.

NOTE : *OR gate can be imitated by setting* $w_0 = -3,\ w_1 = 6\ and\ w_2 = 4$. *NAND gate can be imitated by setting* $w_0 = 6,\ w_1 = -4\ and\ w_2 = -4$. *NOR gate can be imitated by setting* $w_0 = 3,\ w_1 = -6\ and\ w_2 = -4$.

If data points of different classes can be separated with lines, planes or hyperplanes such as the AND gate, NAND gate, OR gate and NOR gate as shown above, then the classification problem is linear and the classes are linearly separable. However, there are cases where the classes are **NOT** linearly separable. A good example would be the XOR gate problem. Our perceptron will find that it is impossible to imitate the XOR gate's functionality. Table 2 illustrates a XOR gate's output for each pair of x_1 *and* x_2 :

x_1	x_2	XOR gate's output
0	0	0
1	0	1
0	1	1
1	1	0

Table 2: XOR gate's output

Figure 3 illustrates the decision surface and the decision boundary of the earlier AND gate's problem. The colour of the data points represents their respective classes. Red represents class 0 (False) and green represents class 1 (True). It can be clearly seen that the data points can be separated based on their respective classes.

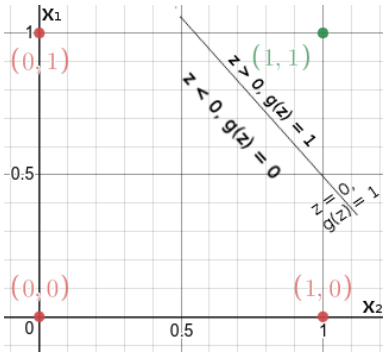

Figure 3: Decision surface of the AND gate perceptron

Figure 4 illustrates the decision surface of a 2-input perceptron on the same data points. However, this time, the classes that each data point belongs to are different. Red represents class 0 (False) and green represents class 1 (True) as before. It can be seen that it is impossible to classify the data points correctly by any straight line (decision boundary). The classes are not linearly separable.

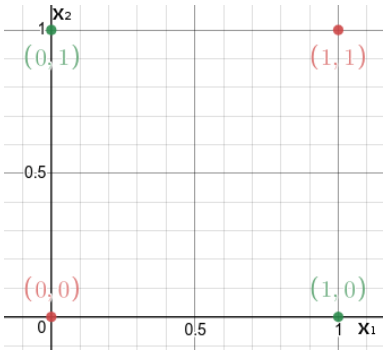

Figure 4: Decision surface of the XOR gate perceptron

Since a XOR gate is made up by performing an AND gate's functionality on the outputs of a OR gate and an NAND gate, a XOR gate's functionality can be imitated by using 3 perceptrons that imitates the functionality of an AND gate, a OR gate and an NAND gate as well. The inputs x_1 *and* x_2 would have to be used as inputs for both the NAND gate's perceptron and the OR gate's perceptron separately. Then the outputs of both the perceptron will be used as inputs for the AND gate's perceptron as illustrated in Figure 5.

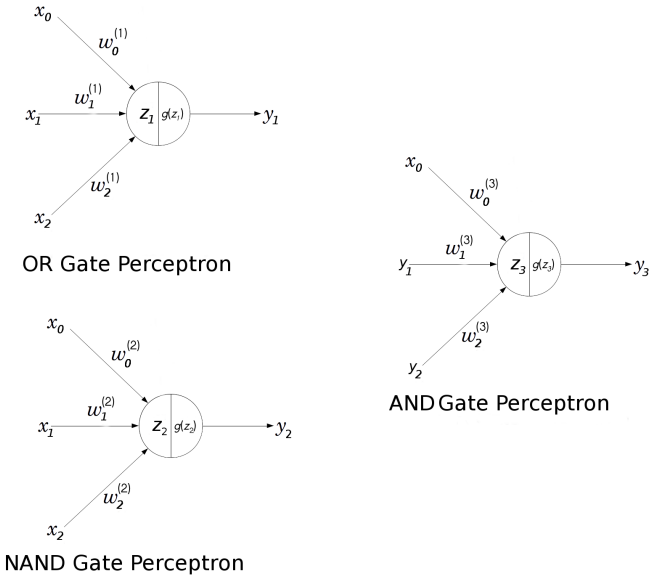

Figure 5: Combination of the perceptrons

Notice that, in figure 5, x_0 is shared among all perceptrons and x_1 *and* x_2 are shared among the NAND and OR gate perceptrons. The outputs of both the NAND and OR gate perceptrons are used as an input to the AND gate perceptron. Therefore, the connections can be summarised as illustrated in Figure 6.

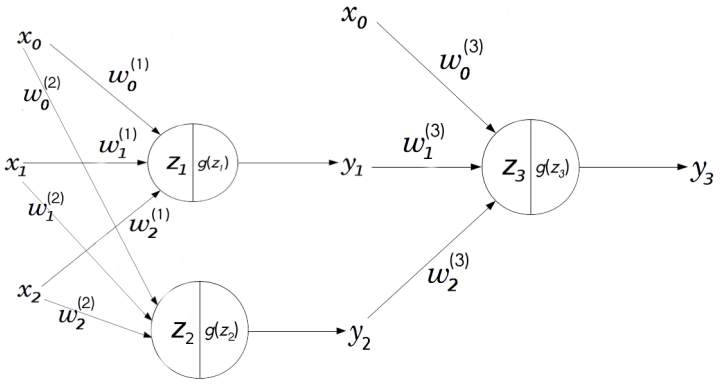

Figure 6: Combination of the perceptrons

By substituting the appropriate values for the weights of each perceptron and $x_0 = 1$ as shown below, the perceptrons above can be illustrated as in figure 7.

$$\begin{aligned} w_0^{(1)} &= -3,\ w_1^{(1)} = 6,\ w_2^{(1)} = 4 \\ w_0^{(2)} &= 6,\ w_1^{(2)} = -4,\ w_2^{(2)} = -4 \\ w_0^{(3)} &= -6,\ w_1^{(3)} = 4,\ w_2^{(3)} = 4 \end{aligned}$$

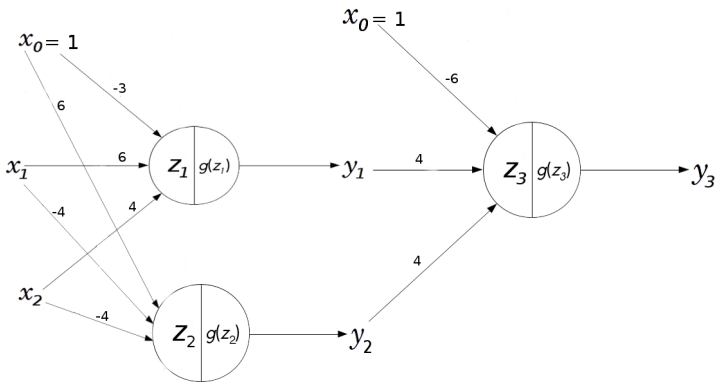

Figure 7: Values of the weights

With this information, the output, y_3 can be decided for each pair of inputs as shown in Table 3.

x_1	x_2	z_1	z_2	y_1	y_2	z_3	y_3
0	0	-3	6	0	1	-2	0
1	0	3	2	1	1	2	1
0	1	1	2	1	1	2	1
1	1	7	-2	1	0	-2	0

Table 3: XOR gate's output

The flow of the inputs from one layer to another can be viewed as a transformation between spaces. In the example shown, OR gate perceptron and NAND gate perceptron together has decision boundaries as illustrated in Figure 8.

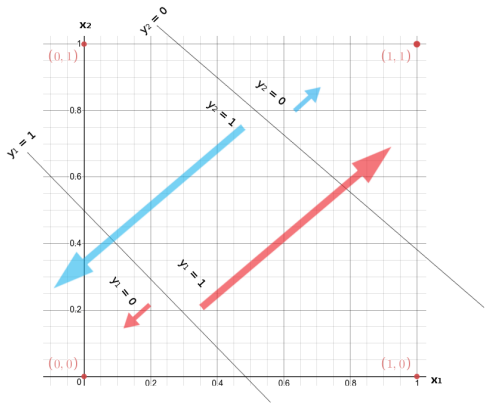

Figure 8: Decision Surface and Decision Boundary for NAND gate and OR gate perceptrons.

The value of y_1 and y_2 will then be used by the following AND gate perceptron to produce y_3. Figure 9 shows the decision surface and decision boundary for the AND gate perceptron.

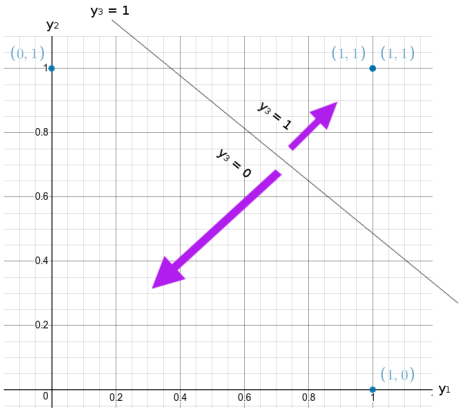

Figure 9: Decision Surface and Decision Boundary for the final AND gate perceptrons.

Stacking perceptrons such that the output of one or more perceptrons becomes the input to a different set of perceptrons is known as Multilayer Perceptron (MLP) or feedforward neural network.

3.2 Multilayer Perceptron (MLP)

The inputs are usually regarded as the input layer and the perceptron(s) that produces output(s) is regarded as the output layer. The layers of perceptron in between the input and output layer is known as the hidden layers. The most basic MLP is sometimes known as the 3-layer neural network. It consists of an input layer, a hidden layer and an output layer as the previous XOR gate example. Figure 10 below shows the XOR gate MLP. Note that the only thing that has changed is the representation of the inputs and the bias term.

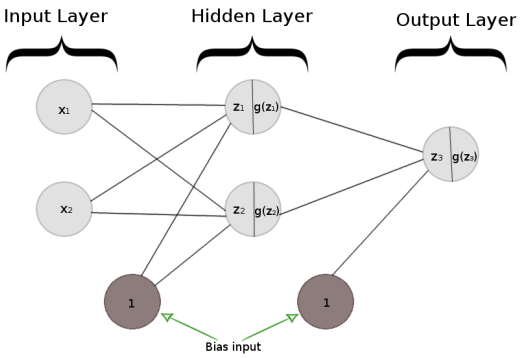

Figure 10: XOR Gate MLP.

However, an MLP in general, is highly flexible. This means that the number of layers and perceptrons in a layer can vary greatly. An example of an MLP with 3 hidden layers with 9 perceptrons in each layer is shown in figure 11 below.

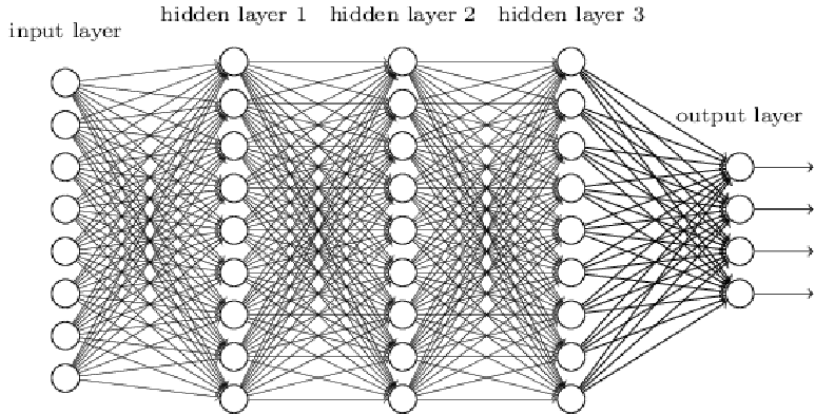

Figure 11: Example of an MLP.

The goal of an MLP is to approximate some function f^*. It defines a mapping $\boldsymbol{y} = f(\boldsymbol{x};\boldsymbol{w})$ and learns the value of the parameters $\mathbf{w}$ that result in the best function approximation. Basically, an MLP is a composition of many different functions and regarded as a universal approximator as proved in *universal approximation theorem*. The overall length of the composed functions gives the depth of the model, hence the term "deep learning".

An MLP algorithm can be used for either a regression problem or a classification problem. In order to enable the MLP to represent nonlinear functions of $\boldsymbol{x}$, a nonlinear activation function is applied to the outputs of each perceptron as mentioned in the previous subsection. MLP is generally used when the knowledge about the relationship between the dependent and independent variables in a dataset is little. Now, the question is, how MLP learns the value of the parameter $\boldsymbol{w}$? The process of learning the parameter $\boldsymbol{w}$ is known as backpropagation which is discussed in the next section. MLP requires a label for each input data to enable it to "learn". Thus, MLP falls under the category of supervised machine learning algorithm.

3.3 Backpropagation

Backpropagation is a method to calculate the gradient of the loss/error of the network with respect to a weight or bias variable. The difference between the actual output and the calculated output is what known as error or loss. The error in an MLP is minimized by choosing the appropriate values for $\boldsymbol{w}$. The loss of an MLP is calculated using some common loss functions (*a.k.a. objective functions*) such as mean-squared error function or cross-entropy loss function (*a.k.a. log loss*).

Why do we have to calculate the values of the weights iteratively? Unlike linear models such as linear regression, the loss function of MLP is non-convex due to the non-linearity. Figure 12 below shows the difference between a convex and non-convex function. Finding the best solution **i.e. finding $\boldsymbol{w}$**, that produces the least error in an algorithm is called the optimization. In a convex

optimization problem, there exist only one optimal solution which is globally optimal or there might be no feasible solution at all. However, in a non-convex optimization problem, there are multiple locally optimal solutions. Hence, the gradient descent method is used to determine whether if a solution is a globally optimal solution by "descending" the gradient of the loss function that was calculated via backpropagation.

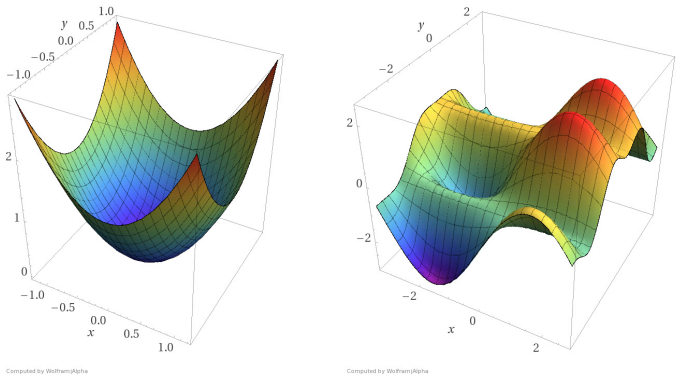

Figure 12: Convex (left) vs Non-Convex (right) function. Source: [2]

3.4 Multilayer Perceptron Training

In order to mathematically represent different MLP architectures consistently, we will be using the standard notations introduced by Stanford in [1] from now on. For simplicity sake, we'll use a simple MLP architecture (1 hidden layer) to demonstrate how it works. The purpose of the MLP is to perform a classification task. Therefore, softmax function is used on the output layer to represent the probability of a class and ReLU will be used as the activation function.

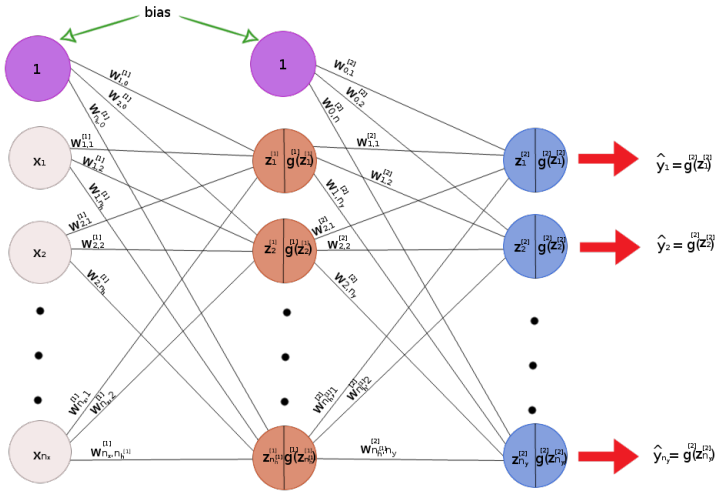

Figure 13: MLP standard notation

From figure 13 above, we can see that :

$$\text{Input to the network : } \begin{bmatrix} x_0 \\ x_1 \\ . \\ . \\ . \\ x_{n_x} \end{bmatrix} \text{ where } x_0 = 1$$

$$\text{Weight matrix of the first hidden layer : } \begin{bmatrix} w^{[1]}_{1,0} & w^{[1]}_{1,1} & . & . & . & w^{[1]}_{1,n_x} \\ w^{[1]}_{2,0} & w^{[1]}_{2,1} & . & . & . & w^{[1]}_{2,n_x} \\ . & . & . & & & . \\ . & . & & . & & . \\ . & . & & & . & . \\ w^{[1]}_{n^{[1]}_h,0} & w^{[1]}_{n^{[1]}_h,1} & . & . & . & w^{[1]}_{n^{[1]}_h,n_x} \end{bmatrix}$$

$$\text{Output of the first hidden layer : } \begin{bmatrix} g^{[1]}(z^{[1]}_1) \\ g^{[1]}(z^{[1]}_2) \\ . \\ . \\ . \\ g^{[1]}(z^{[1]}_{n^{[1]}_h}) \end{bmatrix} = \begin{bmatrix} a^{[1]}_0 \\ a^{[1]}_1 \\ a^{[1]}_2 \\ . \\ . \\ . \\ a^{[1]}_{n^{[1]}_h} \end{bmatrix} \text{ where } a^{[1]}_0 = 1 \text{ is added.}$$

$$\text{Weight matrix of the output layer : } \begin{bmatrix} w_{1,0}^{[2]} & w_{1,1}^{[2]} & . & . & . & w_{1,n_h^{[1]}}^{[2]} \\ w_{2,0}^{[2]} & w_{2,1}^{[2]} & . & . & . & w_{2,n_h^{[1]}}^{[2]} \\ . & . & . & & & . \\ . & . & & . & & . \\ . & . & & & . & . \\ w_{n_y,0}^{[2]} & w_{n_y,1}^{[2]} & . & . & . & w_{n_y,n_h^{[1]}}^{[2]} \end{bmatrix}$$

$$\text{Output of the first hidden layer : } \begin{bmatrix} g^{[2]}(z_1^{[2]}) \\ g^{[2]}(z_2^{[2]}) \\ . \\ . \\ . \\ g^{[2]}(z_{n_y}^{[2]}) \end{bmatrix} = \begin{bmatrix} \hat{y}_1 \\ \hat{y}_2 \\ . \\ . \\ . \\ \hat{y}_{n_y} \end{bmatrix}$$

The process of calculating the output is called forward propagation. The output is calculated as such:

- The input is first transformed by a set of weights in the first hidden layer as follows :

$$\begin{bmatrix} w_{1,0}^{[1]} & w_{1,1}^{[1]} & . & . & . & w_{1,n_x}^{[1]} \\ w_{2,0}^{[1]} & w_{2,1}^{[1]} & . & . & . & w_{2,n_x}^{[1]} \\ . & . & . & & & . \\ . & . & & . & & . \\ . & . & & & . & . \\ w_{n_h^{[1]},0}^{[1]} & w_{n_h^{[1]},1}^{[1]} & . & . & . & w_{n_h^{[1]},n_x}^{[1]} \end{bmatrix} \begin{bmatrix} x_0 \\ x_1 \\ . \\ . \\ . \\ x_{n_x} \end{bmatrix} = \begin{bmatrix} w_{1,0}^{[1]} + x_1 w_{1,1}^{[1]} + & . & . & . & + x_{n_x} w_{1,n_x}^{[1]} \\ w_{2,0}^{[1]} + x_1 w_{2,1}^{[1]} + & . & . & . & + x_{n_x} w_{2,n_x}^{[1]} \\ . & & . & & . \\ . & & & . & . \\ . & & & & . \\ w_{n_h^{[1]},0}^{[1]} + x_1 w_{n_h^{[1]},1}^{[1]} & . & . & . & x_{n_x} w_{n_h^{[1]},n_x}^{[1]} \end{bmatrix} = \begin{bmatrix} z_1^{[1]} \\ z_2^{[1]} \\ . \\ . \\ . \\ z_{n_h^{[1]}}^{[1]} \end{bmatrix}$$

- A nonlinear activation function, $g^{[1]}$, is then applied on the transformed input to get the result/output from the first hidden layer as follows :

$$\begin{bmatrix} g^{[1]}(z_1^{[1]}) \\ g^{[1]}(z_2^{[1]}) \\ . \\ . \\ . \\ g^{[1]}(z_{n_h^{[1]}}^{[1]}) \end{bmatrix} = \begin{bmatrix} a_0^{[1]} \\ a_1^{[1]} \\ a_2^{[1]} \\ . \\ . \\ . \\ a_{n_h^{[1]}}^{[1]} \end{bmatrix}$$

NOTE : $a_0^{[1]} = 1$ *is added before the next process.*

- The output of the first hidden layer is then used as the input for the output layer. The input is again transformed by another set of weights as follows :

$$\begin{bmatrix} w^{[2]}_{1,0} & w^{[2]}_{1,1} & . & . & . & w^{[2]}_{1,n^{[1]}_h} \\ w^{[2]}_{2,0} & w^{[2]}_{2,1} & . & . & . & w^{[2]}_{2,n^{[1]}_h} \\ . & . & . & & & . \\ . & . & & . & & . \\ . & . & & & . & . \\ w^{[2]}_{n_y,0} & w^{[2]}_{n_y,1} & . & . & . & w^{[2]}_{n_y,n^{[1]}_h} \end{bmatrix} \begin{bmatrix} a^{[1]}_0 \\ a^{[1]}_1 \\ . \\ . \\ . \\ a^{[1]}_{n^{[1]}_h} \end{bmatrix} = \begin{bmatrix} w^{[2]}_{1,0} + a^{[1]}_1 w^{[2]}_{1,1} + & . & . & . & +a^{[1]}_{n^{[1]}_h} w^{[2]}_{1,n^{[1]}_h} \\ w^{[2]}_{2,0} + a^{[1]}_1 w^{[2]}_{2,1} + & . & . & . & +a^{[1]}_{n^{[1]}_h} w^{[2]}_{2,n^{[1]}_h} \\ . & & . & & . \\ . & & & . & . \\ . & & & & . \\ w^{[2]}_{n_y,0} + a^{[1]}_1 w^{[2]}_{n_y,1} + & . & . & . & +a^{[1]}_{n^{[1]}_h} w^{[2]}_{n_y,n^{[1]}_h} \end{bmatrix} = \begin{bmatrix} z^{[2]}_1 \\ z^{[2]}_2 \\ . \\ . \\ . \\ z^{[2]}_{n_y} \end{bmatrix}$$

- To obtain the probability of an input belonging to a particular class, softmax function, $g^{[2]}$, is applied on the transformed input as follows:

$$\begin{bmatrix} g^{[2]}(z^{[2]}_1) \\ g^{[2]}(z^{[2]}_2) \\ . \\ . \\ . \\ g^{[2]}(z^{[2]}_{n_y}) \end{bmatrix} = \begin{bmatrix} \hat{y}_1 \\ \hat{y}_2 \\ . \\ . \\ . \\ \hat{y}_{n_y} \end{bmatrix}$$

The process of finding and updating the weights such that the output of the network approximates the given label as close as possible is called the training process. First, in order to train the network, we need an objective. The objective is to minimize the difference between the calculated output and the label, $\boldsymbol{y}$, *i.e. error/loss* for each training data. Note that, in this example, we've only used one training data. The loss function that is normally used for classification problems is the cross-entropy loss function with softmax activation function on the output layer. The cross-entropy loss function is defined as

$$J_{CE}(\hat{\boldsymbol{y}}, \boldsymbol{y}) = -\sum_{i=1}^{n} \boldsymbol{y}^{(i)} \; log \hat{\boldsymbol{y}}^{(i)} \tag{1}$$

Since we only have one training data in our example, the (i) and the notations related to (i) can be dropped. Therefore,

$$J_{CE}(\hat{\boldsymbol{y}}, \boldsymbol{y}) = -\boldsymbol{y} \; log \hat{\boldsymbol{y}} \tag{2}$$

In order to minimize the loss, we have to find the derivative of the loss with respect to each weight variable and update the weight variable accordingly in order to minimize the loss. The derivative of the loss function with respect to the output layer's weights are

$$\frac{\partial J}{\partial \mathbf{W}^{[2]}} \; = \; \mathbf{a}^{[1]} \bullet (\hat{\boldsymbol{y}} \; - \; \boldsymbol{y})$$

However, to differentiate the loss function with respect to the hidden layer's weights is not as straightforward. The fact that a slight change in one of the the hidden layer's weight affects each and every output layer's node hints that calculating the gradients of the hidden layer weights require the sum of the weights in the output layer and the outputs of the output layer as well. Therefore, the derivative of the loss function with respect to the hidden layer's weights are "

$$\frac{\partial J}{\partial \mathbf{W}^{[1]}} \;=\; (\mathbf{W}^{[2]^{\mathbf{T}}}(\hat{\boldsymbol{y}} \;-\; \boldsymbol{y})) \bullet \boldsymbol{x}$$

NOTE : *Since*

$$\frac{\mathrm{d}}{\mathrm{d}x} ReLU(x) = \begin{cases} 0 \; for \; x \;\leq\; 0 \\ 1 \; for \; x \;>\; 0 \end{cases}$$

if $\mathbf{a}_j^{[1]} \;\leq\; 0$, then $(\mathbf{W}^{[2]^{\mathbf{T}}}(\hat{\boldsymbol{y}} \;-\; \boldsymbol{y}))_j \;=\; 0$

This process of calculating the gradient of the weights of the hidden layer through the gradients of the weights in the output layer is known as back-propagation. Finally, once the gradients of the weights have been calculated, the current weight values is updated by

$$\mathbf{W}^{[2]} \;:=\; \mathbf{W}^{[2]} \;-\; \alpha \frac{\partial J}{\partial \mathbf{W}^{[2]}}$$
$$\mathbf{W}^{[1]} \;:=\; \mathbf{W}^{[1]} \;-\; \alpha \frac{\partial J}{\partial \mathbf{W}^{[1]}}$$

where α is a constant used to determine the size of the change. It is normally a positive value in the range of $0 < \alpha < 1$. This is known as gradient descent. This whole process is repeated many times until the loss of the network is relatively low.

References

[1] *Standard Deep Learning Notation (2019). Retrieved from https://cs230.stanford.edu/files/Notation.pdf Backup link : link*

[2] *Zadeh, R. (2019). The hard thing about deep learning. Retrieved from https://www.oreilly.com/ideas/the-hard-thing-about-deep-learning*

[3] *Hornik, K. (1991). Approximation capabilities of multilayer feedforward networks. Neural networks, 4(2), 251-257.*

[4] *Goodfellow, I., Bengio, Y., Courville, A. (2016). Deep learning. Cambridge (EE.UU.): MIT Press.*

[5] *Mitchell, T. (2017). Machine learning. New York: McGraw Hill.*

[6] *Sarle, W. S. (1994). Neural networks and statistical models.*

[7] *Sze, V., Chen, Y. H., Yang, T. J., Emer, J. S. (2017). Efficient processing of deep neural networks: A tutorial and survey. Proceedings of the IEEE, 105(12), 2295-2329.*