An Introduction To Artificial Neural Network

<u>Reference(s):</u>

An Introduction to Neural Networks for Beginners*,* Dr. Andy Thomas (Adventures in Machine Learning).

https://www.analyticsvidhya.com/blog/2021/11/a-comprehensive-guide-to-linear-regression-with-pytorch/

Artificial Intelligence: A Guide to Intelligence Systems*,* Michael Negnevitsky, Addison Wesley

Introduction to Neural Networks

- An **artificial neural network** (**ANN**) or **neural network** (**NN**) is a software implementation of the neuronal structure of a human brain.
- The brain contains **neurons** which are kind of like biological switches.
  - These can change their **output state** depending on the strength of their electrical or chemical **input**.
  - The **NN** in a person's brain is a hugely interconnected network of **neurons**, where the **output** of any given neuron may be the **input** to thousands of other neurons (massively parallel structure!).

Introduction to Neural Networks

- **Neural learning** occurs by repeatedly activating certain neural connections over others, reinforcing those connections.
- This makes them more likely to produce the desired outcome given a specified input
  - This learning involves **feedback** – when the **desired outcome** occurs, the neural connections causing that outcome become strengthened.

Introduction to Neural Networks

- **NNs** attempt to simplify and mimic brain behavior.
- They can be trained in a ***supervised*** or ***unsupervised*** learning manner.
- In a **supervised NN**, the network is trained by providing matched **input-output data samples**, with the intention of getting the ANN to provide the **desired output** for a given **input**.

Supervised NN An Example

- Consider an e-mail **spam filter** – the **input training data** could be the count of various words in the body of the email, and the **output training data** would be a classification of whether the e-mail was truly spam or not.
- If many examples of e-mails are passed through the NN allows the network to learn what input data makes it likely that an e-mail is a spam or not.
- This learning takes place by adjusting the **weights** of the NN connections.

Unsupervised NN

- In **unsupervised learning,** the NN tries to "understand" and generate the **output** structure of the provided input data set "on its own" without its output pattern.
  - There is no supervised learning happening at this point

Structure of an Artificial Neuron

- The **neurons** that we are going to see here are not biological but are **Artificial Neuron**s
- The **artificial Neuron**s are extremely simple abstractions of **biological neurons**, realized as elements in *a program* or perhaps as *a circuit* made of silicon.
- Networks of these **artificial neurons** do not have a fraction of the power of the human brain, but they can merely be trained to perform useful functions.

The Structure of an Artificial Neuron

- A *single-input*, *single-output* artificial neuron is shown in **figure 1**.
- The scalar input ***N*** is multiplied by the scalar weight ***W*** to form ***W*N***, one of the terms that are sent to the **summer unit**:
  - Where ***N*** = {n_1, n_2, n_3,…} and ***W*** = {w_1, w_2, w_3,….}
  - The summer output is often referred to as the **net input**. It incorporates a **threshold** (or **bias**), the **weight of the +1 bias element** that goes into a **transfer function (or activation function)** ***f***, which determines and produces the scalar neuron output ***y***.

The Structure of an Artificial Neuron

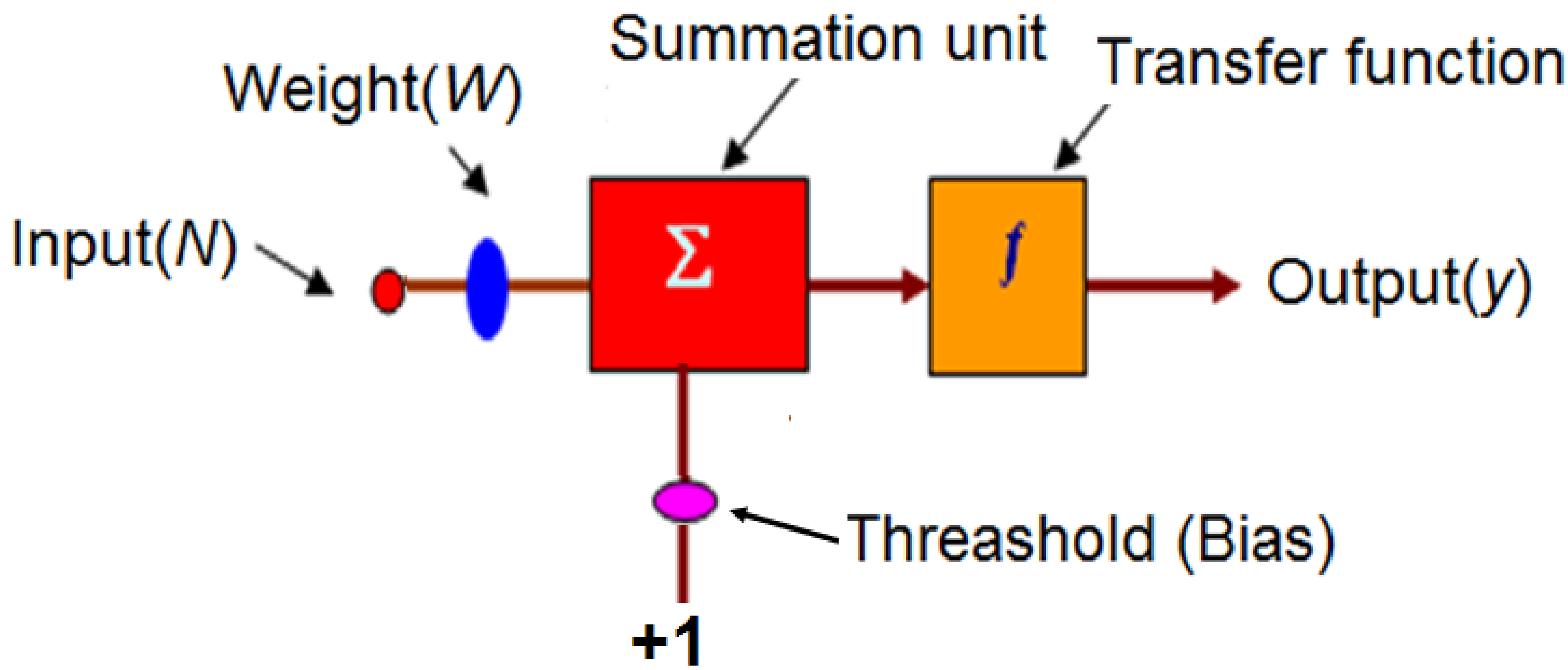


Figure 1: A single input, single output neuron

Neuron as Simple Computing Element

- An artificial **neuron** receives several signals from its **input** links, computes with its **activation function** (**activation level**), and sends the result as an **output** through its **output links** (**figure 2** shows a neuron with ***n*** inputs)
- An activation function simulates the neuron in an ANN.
- The **input** signal can be ***raw data*** or ***outputs*** of other neurons
- The **output** signal can be either a ***final solution*** to a problem or an ***input*** to other neurons.

Neuron as a Simple Computing Element

- The single neuron node which is shown in **figure 2** is called as a **perceptron**.

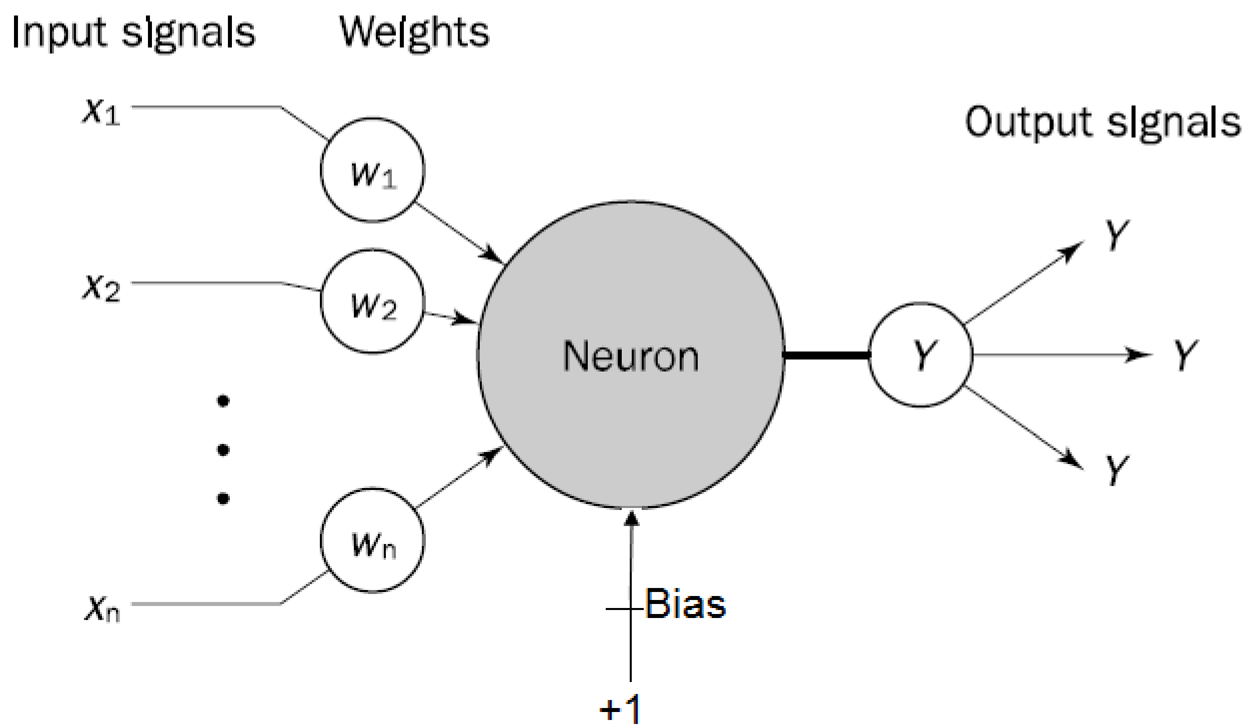


Figure 2. Single neuron with *n* inputs

How Does a Neuron Determine its Output?

- The neuron computes the **weighted sum** of the input signals and compares the result with a **bias** value ***b***.
- *If the **net input** is less than the **b**, then the neuron output is* **–1**
- But *if the **net input** is greater than or equal to the **bias b**, the neuron becomes activated, and its output attains a value* **+1** (McCulloch, 1943)
- From **figure 2**, the **weighted_sum *X*** can be given as:

$$\boldsymbol{X} = \sum_{i=1}^{n} (x_i w_i) \qquad (1)$$

$$\text{where output, } \boldsymbol{Y} = \begin{cases} +1 & \text{if } X \geq b \\ -1 & \text{if } X < b \end{cases}$$

- Where $\boldsymbol{x_i}$ is the value of the input, $\boldsymbol{w_i}$ is the weight of input, and ***n*** is the number of inputs of the neuron.

How Does a Neuron Determine its Output?

- The output of the neuron ***Y*** (from **Figure 2**) is estimated as;

$$\boldsymbol{Y} = \begin{cases} +1 & \text{if } X \geq b \\ -1 & \text{if } X < b \end{cases}$$

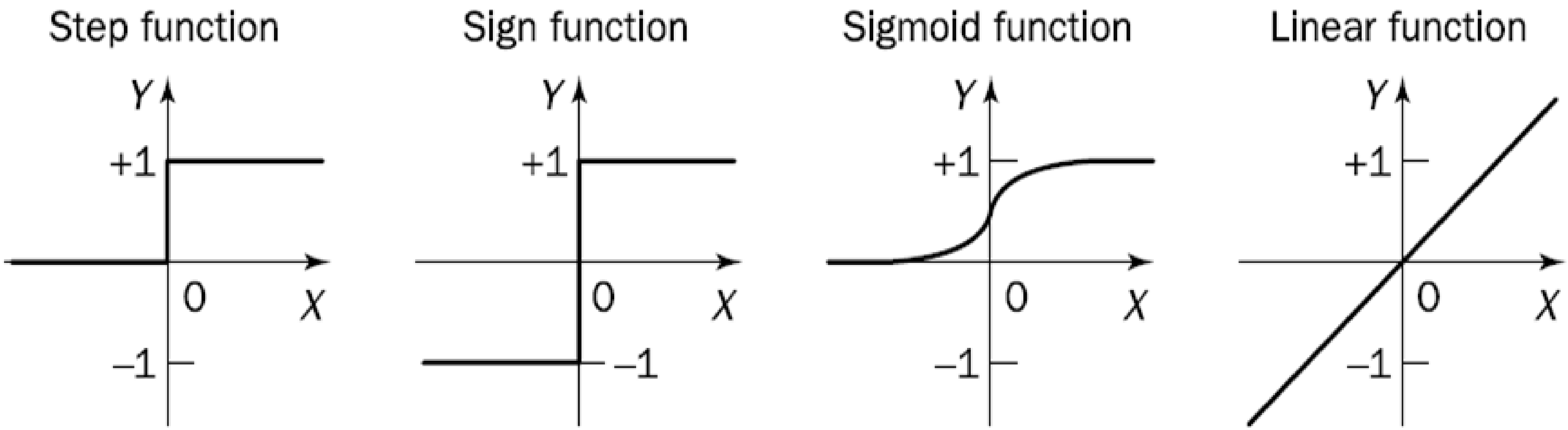
- This type of activation function is called a **sign activation** function
- Thus the **actual output** of the neuron with a **sign activation** function can be represented as:

$$\boldsymbol{Y} = \boldsymbol{sign}[\boldsymbol{X}] \qquad \text{;Where } \boldsymbol{X} = \sum_{i=1}^{n}(x_i w_i) - b \quad \text{or} \quad \sum_{i=1}^{n}(x_i w_i) + b \qquad (2)$$

Then output can be shown as, $\boldsymbol{Y} = \begin{cases} +1 \text{ if } X \geq 0 \\ -1 \text{ if } X < 0 \end{cases}$

- There are many activation functions: *step, sign, linear, sigmoid, etc.*
- **Figure 3** shows the common activation functions of a neuron.

Activation Functions of a Neuron



$$Y^{step} = \begin{cases} 1, \text{ if } X \geq 0 \\ 0, \text{ if } X < 0 \end{cases} \qquad Y^{sign} = \begin{cases} +1, \text{ if } X \geq 0 \\ -1, \text{ if } X < 0 \end{cases} \qquad Y^{sigmoid} = \frac{1}{1 + e^{-X}} \qquad Y^{linear} = X$$

Figure 3: Activation functions of a neuron

Activation Functions of a Neuron

- The ***step*** and ***sign*** activation functions are also called **hard limit** functions, are often used in **decision making neurons** for **classification** application
- The **sigmoid** function ($Y_{sigmoid}$ = $\dfrac{1}{1+e^{-X}}$, where **e** = 2.7183) transforms the **input**, which can have any value between ***plus* and *minus* infinity**, into a reasonable value in the range between **0** to **1**
  - Neurons with this function are used in the **back-propagation** networks
- The **linear function** ($Y_{linear} = X$) provides an *output equal to the neuron weighted input*
  - Neurons with *linear function are often used for linear approximation*

The Sigmoid Function

- The **Sigmoid** activation function: ($Y_{sigmoid} = 1/(1 + e^{-x})$)

```
import matplotlib.pylab as plt
import numpy as np
x = np.arange(-8, 8, 0.1)
f = 1/(1 + np. exp(-x))
plt.plot(x, f)
plt.xlabel('x')
plt.ylabel('f(x)')
plt.show()
```

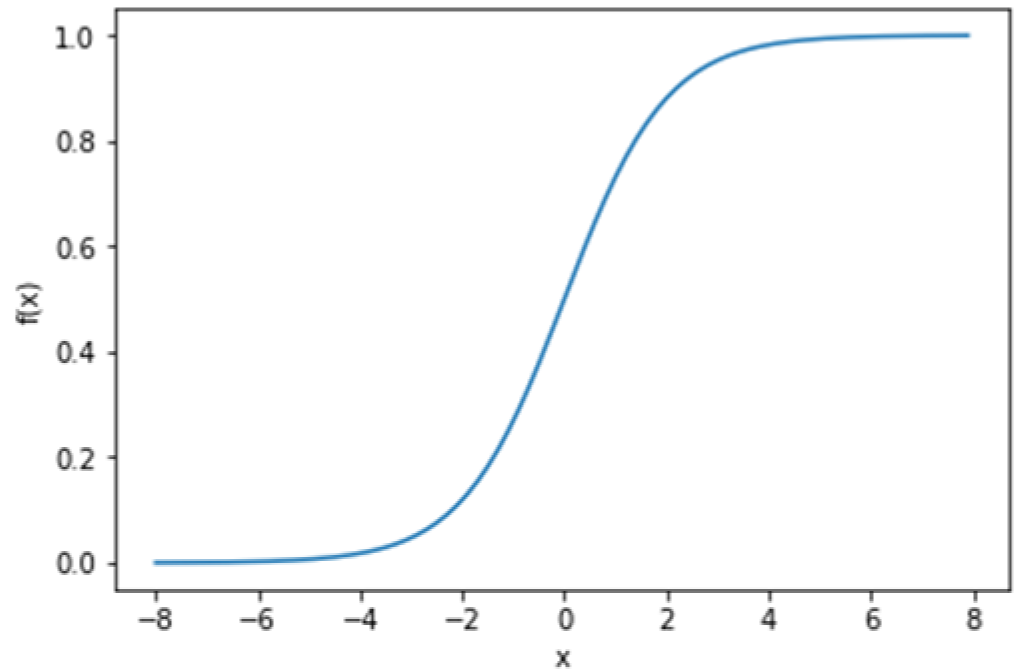


Figure 4 The sigmoid function.

The Sigmoid Function

- **Properties of Sigmoid Function**
  - The sigmoid function returns a **real-valued** output.
  - The first derivative of the sigmoid function will be non-negative or non-positive.
    - **Non-Negative:** If a number is greater than or equal to zero.
    - **Non-Positive:** If a number is less than or equal to Zero.
- **Sigmoid Function Usage**
  - The Sigmoid function used for **binary classification** in the logistic regression model.
  - While creating ANNs, uses the sigmoid function as the testing **activation function**.
  - In statistics, the **sigmoid function** graphs are standard as a **cumulative distribution function**.

Activation Function: tanh

- Another popular NN activation function is the **tanh** (**hyperbolic tangent**) function.
- The **tanh** function is defined as:

$$f(x) = \tanh(x) = \frac{1 - e^{-2x}}{1 + e^{-2x}}$$

- It looks very similar to **sigmoid** function; in fact**, tanh function is a scaled sigmoid function**.
- As **sigmoid function**, this is also a **nonlinear function**, defined in the range of values **(-1, 1)**.
- The **gradient (**or **slope)** is stronger for ***tanh*** than ***sigmoid*** (the derivatives of its *exponential components* are more steep).

Activation Function: tanh

- Deciding between **sigmoid** and **tanh** will depend on gradient strength requirement of an application. Like the **sigmoid**, **tanh** also has the **missing slope** problem. **Figure5** shows the **tanh** activation function:

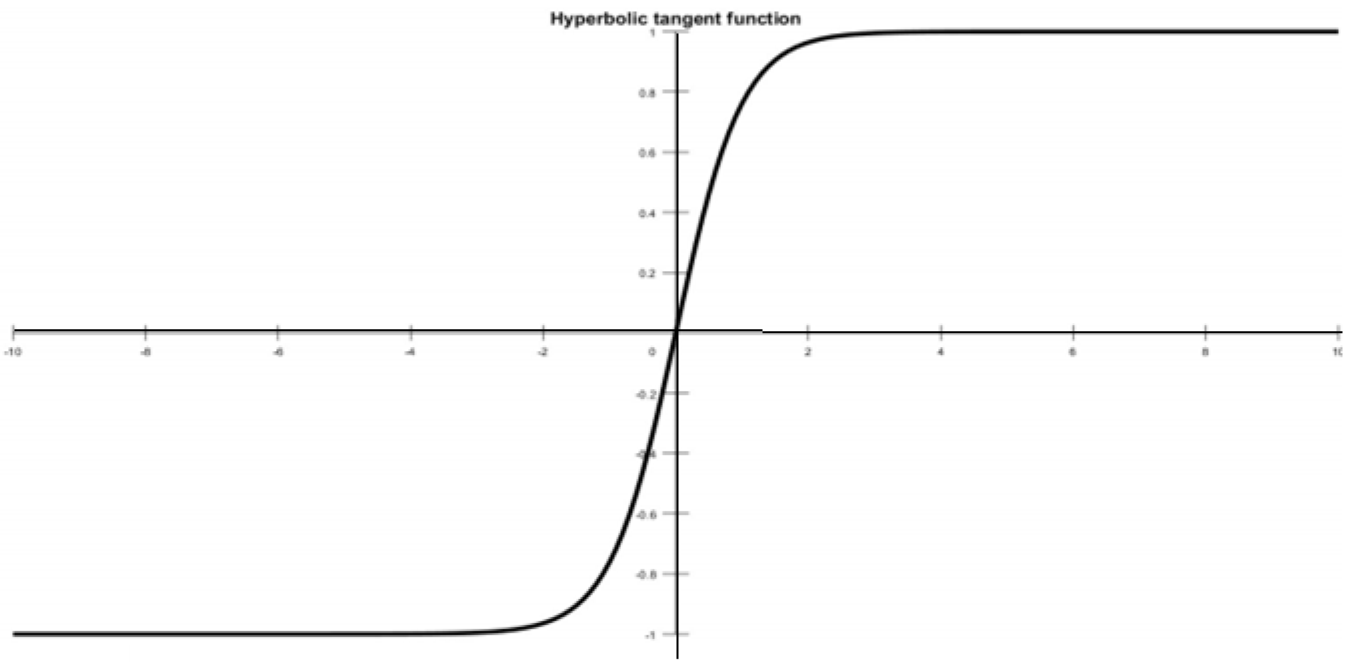


Figure 5

It looks very similar to sigmoid function; in fact, it is a **scaled sigmoid** function.

Activation Function: ReLU

- **Rectified Linear Unit (ReLU)** is the most used activation function for applications based on **CNN** (Convolutional NN).
- It is a simple condition and has advantages over the other functions.
- The function is defined by the following formula:

$$\boldsymbol{f(x) = \begin{cases} 0 & \text{when } (x < 0) \\ x & \text{when } (x >= 0) \end{cases}}$$

- $\boldsymbol{f(x) = \max(x, 0)}$
- The range of output is between **0** and **infinity**.
- **ReLU** finds applications in **computer vision** and **speech recognition**.

Activation Function: ReLU

- **Figure6** shows a **ReLU** activation function:

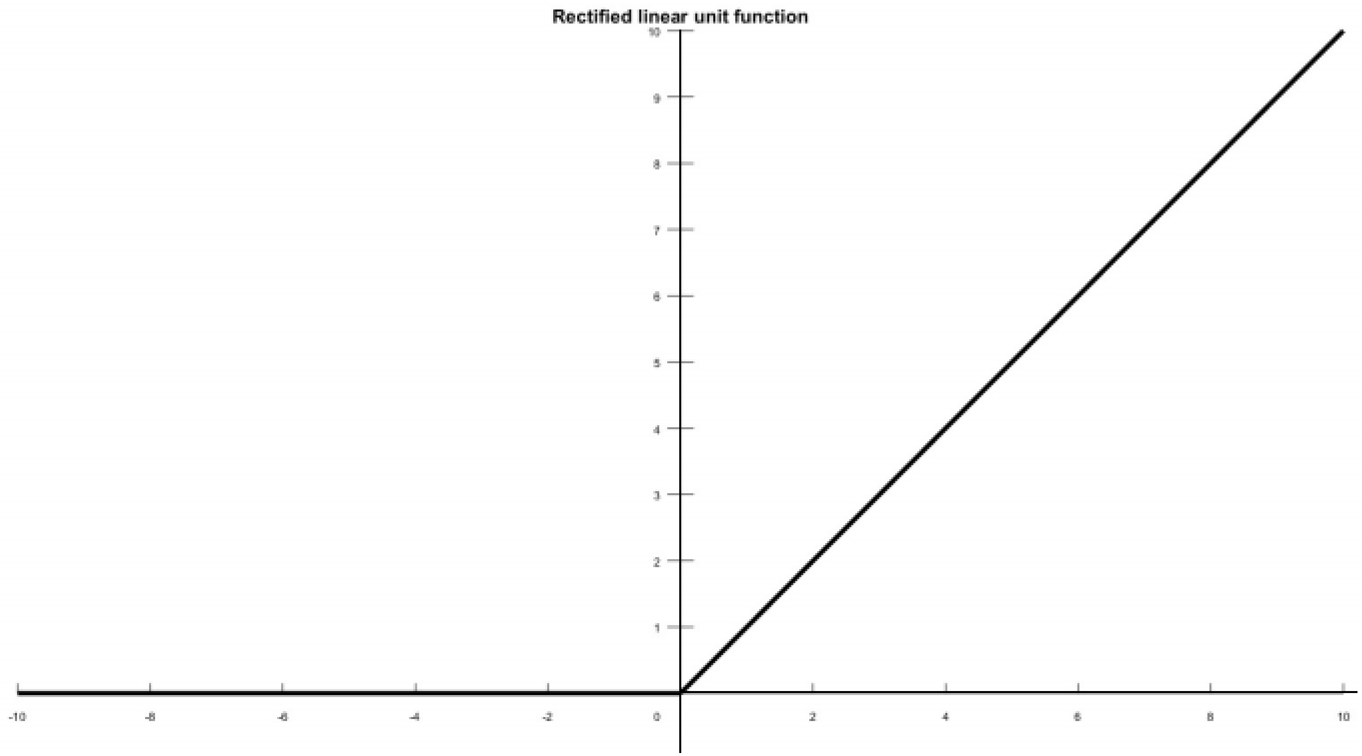


Figure 6

Which activation functions to use?

- The **sigmoid** is the most used activation function, but it suffers from the following setbacks:
  - **Since it uses a logistic model, the computations are time-consuming and complex.**
  - **It causes gradients to vanish, and no signals pass through the neurons at some point.**
  - **It is slow in convergence.**
  - **It is not zero-centered.**

Effects of Adjusting Weights

- Let’s take a neuron node with only **one input** and **one output** (without a ***bias*** value) which is shown in **Figure 7**:

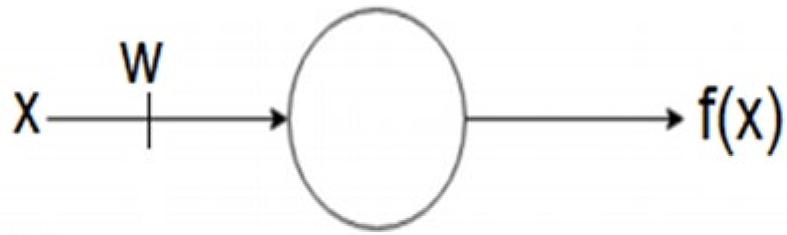


Figure 7: A single neuron node.

- The neuron’s activation function, in this case, is the **sigmoid function**. What does changing **weight *w*** do in this simple network?
  – **Figure 8** shows that <u>changing the **weight** changes the slope of the output of the sigmoid activation function</u>, which is helpful if we want to model different strengths of relationships between the **input** and **output** variables.

Effects of Adjusting Weights

```
import matplotlib.pylab as plt
import numpy as np
x = np.arange(-8, 8, 0.1)
w1 = 0.5
w2 = 1.0
w3 = 2.0
l1 = 'w1 = 0.5'
l2 = 'w2 = 1.0'
l3 = 'w3 = 2.0`
for w,l in [(w1, l1), (w2,l2), (w3,l3)]:
    f = 1/(1+np.exp(-x*w))
    plt.plot(x, f, label = l)
plt.xlabel('x')
plt.ylabel('y = f(x)')
plt.legend(loc = 2)
plt.show()
```

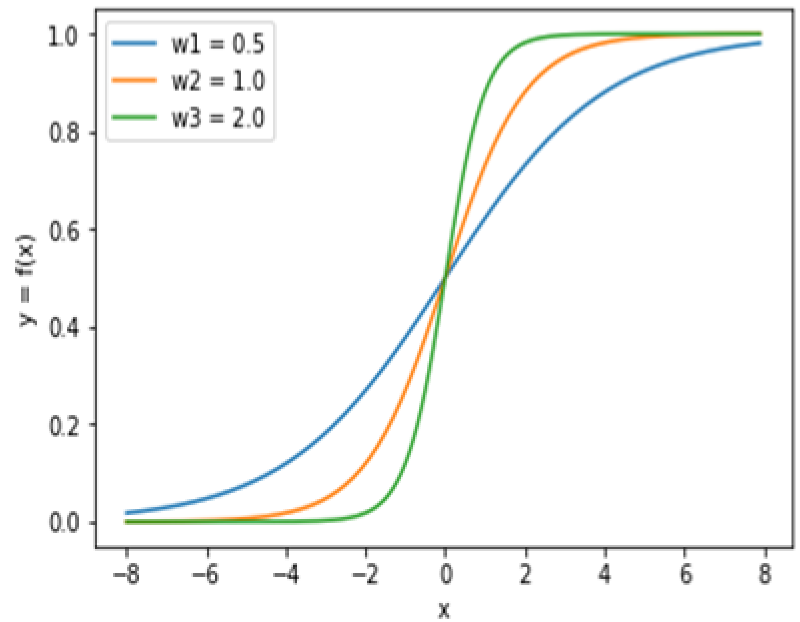


Figure 8: Effect of adjusting weights

Effects of Adjusting Bias

- The **weights** are real-valued numbers multiplied by the inputs and then summed up in the NN node.
  - In this case, the ***w*** has been increased to simulate a more defined "turn on" function.
- So, in other words, the **weighted input** to a neuron node with three inputs ($\boldsymbol{x_1}$, $\boldsymbol{x_2}$, and $\boldsymbol{x_3}$) and their respective weights ($\boldsymbol{w_1}$, $\boldsymbol{w_2}$, and $\boldsymbol{w_3}$) can be: $\boldsymbol{x_1 w_1 + x_2 w_2 + x_3 w_3 + b}$
  - where the ***b*** is the weight of the **+1** bias element of the neuron node, and
  - the inclusion of this **bias weight** enhances the flexibility of the neuron node.
  - The effect of **bias** adjustment is shown in **Figure 9**.

Effects of Adjusting Bias

```
import matplotlib.pylab as plt
import numpy as np
x = np.arange(-8, 8, 0.1)
w = 5.0
b1 = -8.0
b2 = 0.0
b3 = 8.0
l1 = 'b1 = -8.0'
l2 = 'b2 = 0.0'
l3 = 'b3 = 8.0'
for b, l in [(b1, l1), (b2, l2), (b3, l3)]:
    f = 1/(1 + np.exp(-(x*w) + b))
    plt.plot(x, f, label = l)
plt.xlabel('x')
plt.ylabel('y = f(x)')
plt.legend(loc = 2)
plt.show()
```

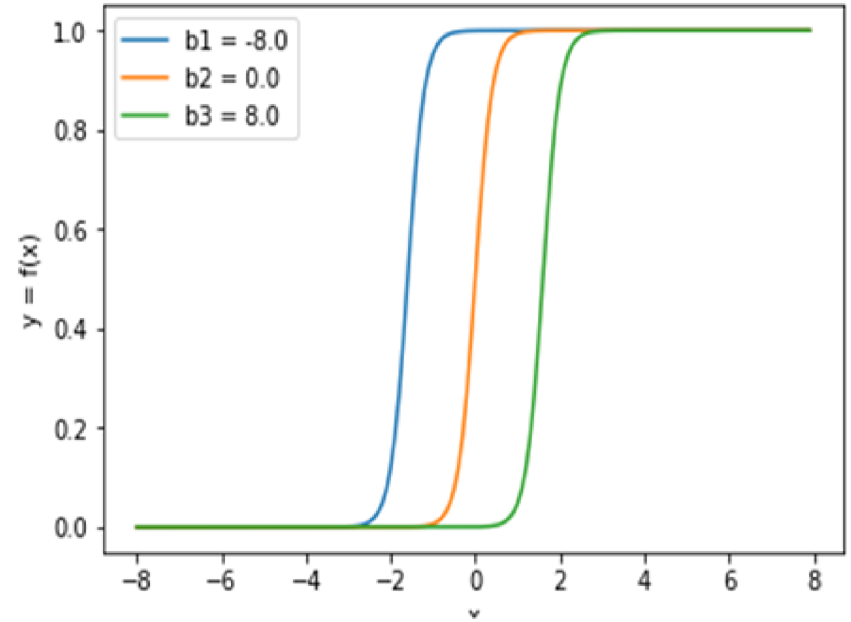


Figure 9: Effects of bias adjustments

Effects of Adjusting Bias

- **Figure 9** shows that by varying the **bias weight *b***, you can change the **output** when the neuron node activates.
- Therefore, by adding a **bias** term, you can make a neuron node simulate a generic **if** function;
  - i.e**. if (*x* > *z*) then 1 else 0.**
  - Without a **bias** term, you cannot vary the ***z*** of the **if statement**; it will always be stuck around 0.
- This is very useful if you are trying to simulate conditional relationships.

Gradient Descent Optimization

- **Gradient descent** is an iterative method to find the ***minimum*** of a function (*gradient descent will be very clear from the **perceptron learning** section*).
- **Gradient descent** starts with an initial set of ***weights***; in each step, it ***decreases*** each ***weight*** in proportion to its partial derivative:

$$w_i := w_i + \eta \times \frac{\partial Error_E(\overline{w})}{\partial w_i}$$

  - where ***η****, the gradient descent step size, is called the **learning rate.***
  - **The learning rate**, as well as the features and the data, is given as input to the learning algorithm.
  - The partial derivative specifies ***how much a small change in the weight would change the error***.

Gradient Descent Optimization

- Consider minimizing the **sum-of-squares error**. The error is the sum of all the examples. The partial derivative of a sum is the sum of the partial.
- For each example ***e***, *let* **δ = *val*(*e*, *Y*) − *pval*^{w}(*e*, *Y*) = error.**
- Thus, each example ***e*** updates each weight w_i:

$$w_i \quad := \quad w_i + \eta \times \delta \times val(e, X_i)$$

  - Algorithm, ***LinearLearner* (*X, Y, E, η*)**, for learning a **linear function for minimizing the sum-of-square error (SOSE)**.

Gradient Descent Optimization

- **Gradient descent** is an iterative method ***to find the minimum of a function***.
- **Gradient descent** for ***minimizing error*** starts with an initial set of **weights**; in each step***, it decreases each* weight** in proportion to its **partial derivative** (*a **derivative** shows the sensitivity of change of a function's output with respect to its input.*):

$$\boxed{w_i := w_i - \eta * \frac{\partial}{\partial w_i} error(Es, \overline{w})}$$

where η, the gradient descent step size, is called the **learning rate**. The learning rate, as well as the features and the data, is given as input to the learning algorithm. The partial derivative specifies how much a small change in the weight would change the error.

Gradient Descent Optimization

- Neural learning keeps changing the **weights** until there is the greatest ***error reduction*** by an amount known as the **learning rate** (α).
  - **Learning rate** is a scalar parameter used to set the rate of adjustments <u>to reduce the errors faster</u>.
  - It adjusts **weights** and **biases** in a backpropagation learning process.
- <u>**The higher the learning rate, the faster the algorithm will reduce the errors and the faster the training process**</u>.

The Perceptron

- The **perceptron** is the simplest form of neural net with **one neuron**.
- It consists of single **neuron** with adjustable *weights* and a **hard limiter** output function (depends on the application)
- A **perceptron** with two input is shown in **Figure 10:**

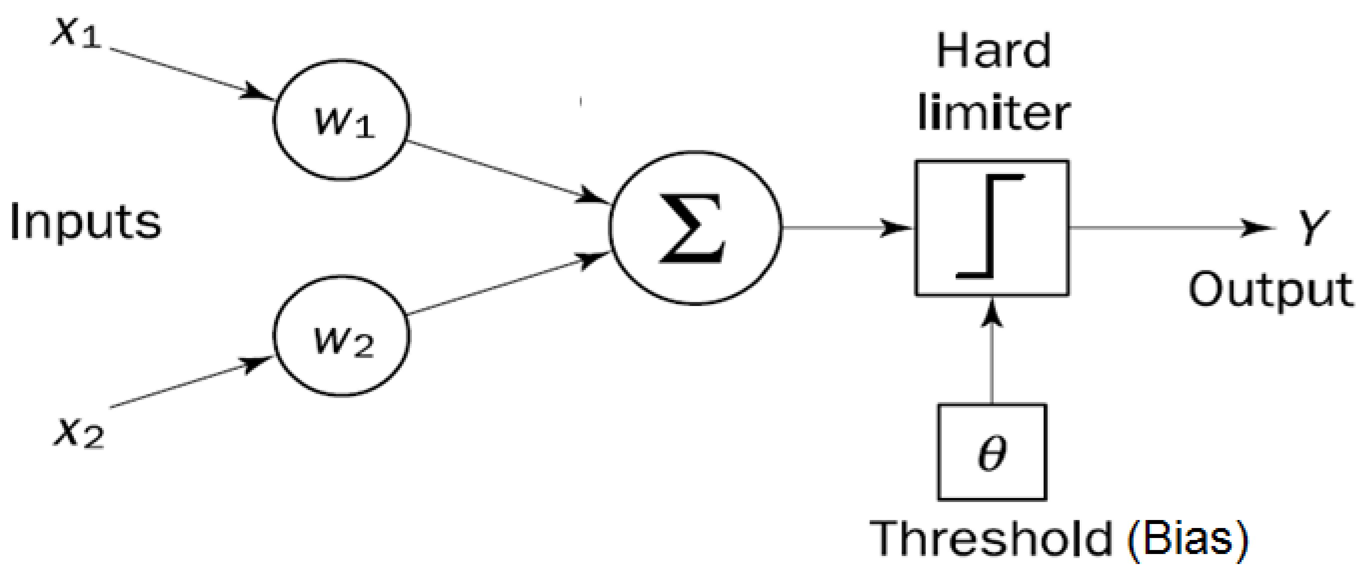
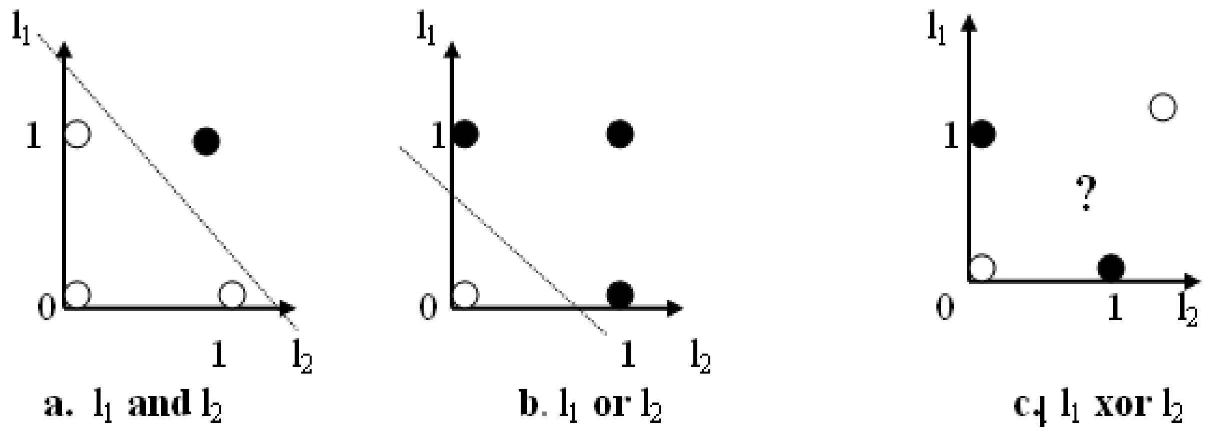


Figure10: A perceptron with two input.

Linear Separability in Perceptrons

- **Figure 11** and **Figure 12** show three different Boolean functions of two inputs, the AND, OR, and XOR functions

Figure 11



- Each function is represented as a 2D plot, based on the values of the two inputs (**black dots** indicate **1**, and **white dots** indicate **0**)
- A perceptron can represent a function only if there is some line that separates all the **white dots** from the **black dots** called a **linearly separable** function.
- Thus, a perceptron can represent **AND**, and **OR**, but **not XOR !**

Linear Separability in Perceptrons

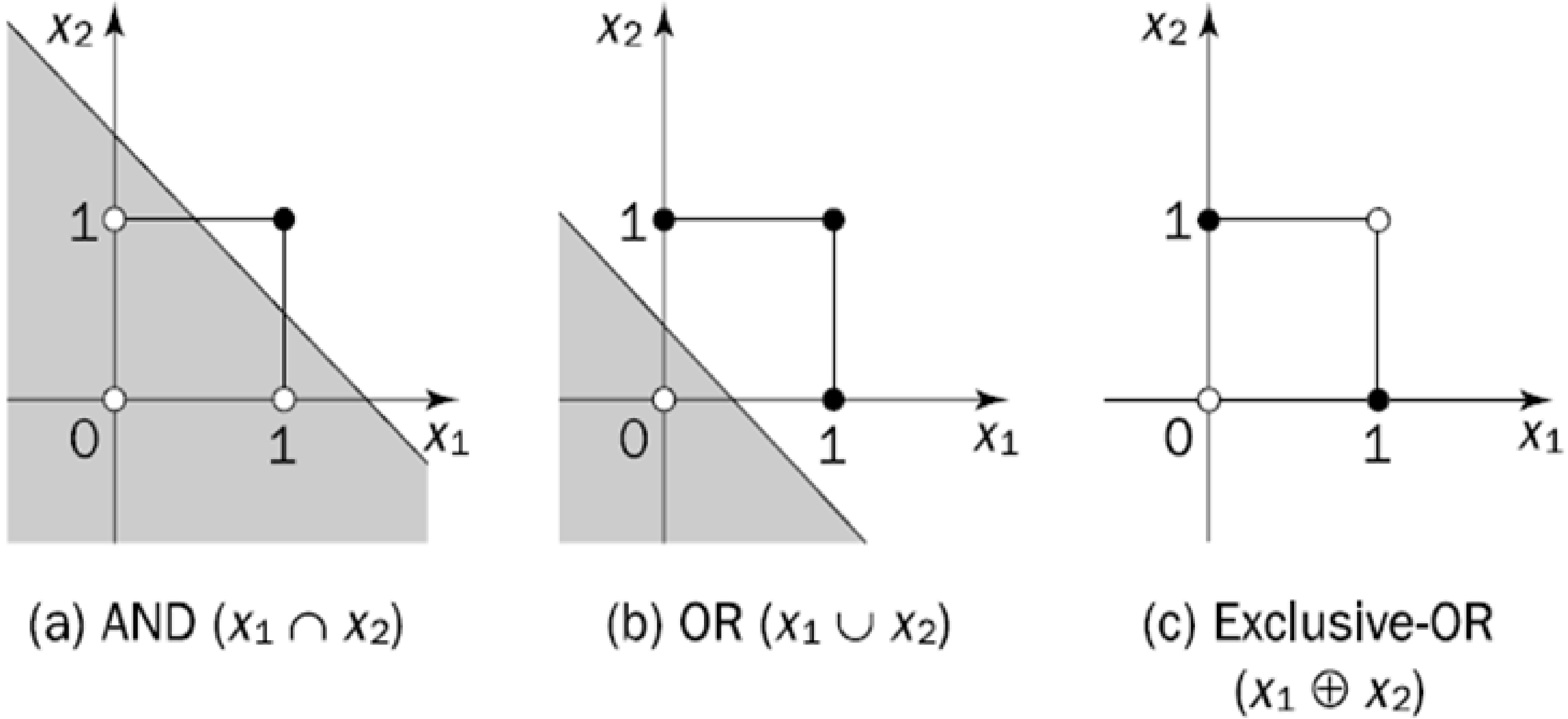


Figure 12: Two-dimensional plots of basic logical operations

The Perceptron Learning Rule

- Rosenblatt first proposed the perceptron learning rule in 1960
- Using this rule, we can derive the **perceptron training algorithm** for classification tasks
- This is done by <u>*making small adjustments in the* **weights** *to reduce the difference between the* ***perceptron's actual (calculated/predicted) and desired (given/expected) outputs***</u>.
  - The **initial weights** are randomly assigned in the range [-0.5, 0.5] and then updated to obtain the output consistent with the training examples

The Perceptron Learning Rule

- If at iteration ***p***, *the **calculated/predicted** output is* ***Y(p)*** and the ***desired*** *output is* $\boldsymbol{Y_d(p)}$, then the **error *e*** at iteration ***p*** is given by

$$\boldsymbol{e(p) = Y_d(p) - Y(p)}, \text{ where } \boldsymbol{p} = 1, 2, 3, \ldots \quad (3)$$

- If the error ***e*(*p*)** is **positive**, we need to **increase *Y*(*p*)**
- If the error is **negative**, we need to **decrease *Y*(*p*)**
- Taking into the account that each perceptron input contributes $\boldsymbol{x_i(p) \times w_i(p)}$ with a **base** (threashold)θ to the total input ***X*(*p*)**
- Based on this concept, the **perceptron learning rule** is established:

$$\boldsymbol{w_i(p+1) = w_i(p) + \propto \times\, x_i(p) \times e(p)} \qquad (4)$$

Where $\propto$ is the ***learning rate***, a positive constant **less than the unity**

Steps Behind Perceptron Learning Process

- <u>Step1</u>:**Initialization**
  - Set initial weights w_1, w_2,…., w_n and base θ to random numbers in the range [-0.5, 0.5]
- <u>Step2</u>: **Activation**
  - Activate the perceptron by applying inputs $x_1(p)$, $x_2(p)$,…$x_n(p)$ and **desired** output $Y_d(p)$. Calculate the **actual output** $Y(p)$ at $p = 1$:

$$Y(p) = \text{step}\left[\sum_{i=1}^{n} x_i(p) * w_i(p) - \theta \right] \qquad (5)$$

  - Where n is the number of the perceptron inputs and **step** is the **activation function**

Steps Behind Perceptron Learning Process

- <u>Step3</u>: **Weight training**
  - Update the weights of the perceptron

    $$\boldsymbol{w_i(p+1) = w_i(p) + \Delta w_i(p)} \qquad (6)$$

  - Where $\boldsymbol{\Delta w_i(p)}$ is the weight correction at iteration $\boldsymbol{p}$. The weight correction is computed by the **delta rule**:

    $$\boldsymbol{\Delta w_i(p) = \propto \times x_i(p) \times e(p)} \qquad (7)$$

- <u>Step4</u>: **Iteration**
  - Increase iteration $\boldsymbol{p}$ by one, go back to step 2 and repeat the process until **convergence** (all the $\boldsymbol{y(p)}$ focus with $\boldsymbol{y_d(p)}$ without error)

Train Perceptron for AND and OR functions

- The truth tables for operations AND, OR and XOR are shown in **Table 1**. The perceptron must be trained to classify the input patterns

Table 1 Truth tables for the basic logical operations

Input variables		AND	OR	Exclusive-OR
x_1	x_2	$x_1 \cap x_2$	$x_1 \cup x_2$	$x_1 \oplus x_2$
0	0	0	0	0
0	1	0	1	1
1	0	0	1	1
1	1	1	1	0

- The training process for AND function is shown in **Table 2**

Table 2 Example of perceptron learning: the logical operation AND

Epoch	x_1	x_2	Y_d	w_1	w_2	Y	e	w_1	w_2
1	0	0	0	0.3	−0.1	0	0	0.3	−0.1
	0	1	0	0.3	−0.1	0	0	0.3	−0.1
	1	0	0	0.3	−0.1	1	−1	0.2	−0.1
	1	1	1	0.2	−0.1	0	1	0.3	0.0
2	0	0	0	0.3	0.0	0	0	0.3	0.0
	0	1	0	0.3	0.0	0	0	0.3	0.0
	1	0	0	0.3	0.0	1	−1	0.2	0.0
	1	1	1	0.2	0.0	1	0	0.2	0.0
3	0	0	0	0.2	0.0	0	0	0.2	0.0
	0	1	0	0.2	0.0	0	0	0.2	0.0
	1	0	0	0.2	0.0	1	−1	0.1	0.0
	1	1	1	0.1	0.0	0	1	0.2	0.1
4	0	0	0	0.2	0.1	0	0	0.2	0.1
	0	1	0	0.2	0.1	0	0	0.2	0.1
	1	0	0	0.2	0.1	1	−1	0.1	0.1
	1	1	1	0.1	0.1	1	0	0.1	0.1
5	0	0	0	0.1	0.1	0	0	0.1	0.1
	0	1	0	0.1	0.1	0	0	0.1	0.1
	1	0	0	0.1	0.1	0	0	0.1	0.1
	1	1	1	0.1	0.1	1	0	0.1	0.1

Threshold: $\theta = 0.2$; learning rate: $\alpha = 0.1$.

Why Can a Perceptron Learn Only Linearly Separable Functions?

- The fact that a perceptron can learn only **linearly separable functions** based on the following equation:

$$X = \sum_{i=1}^{n} x_i w_i$$

$$Y = \begin{cases} +1 & \text{if } X \geqslant \theta \\ -1 & \text{if X } < \theta \end{cases}$$

- The perceptron output *Y* is 1 only if the total weighted input *X* is greater than or equal to the threshold, θ. This means that the entire input space is divided in two along a boundary defined by $\boldsymbol{X = \theta}$
- A separating line for the operation AND is defined by the equation $\boldsymbol{x_1 w_1 + x_2 w_2 = \theta}$

Why Can a Perceptron Learn Only Linearly Separable Functions?

- If we substitute values for weights $\boldsymbol{w_1}$ and $\boldsymbol{w_2}$ and threshold θ given in **Table 2**, we obtain one of the possible separating lines as (see **5th** iteration):

  $0.1x_1 + 0.1x_2 = 0.2$ or $\boldsymbol{x_1 + x_2 = 2}$

- Thus, the region below the boundary line, where the output is **0**, is given by $\boldsymbol{x_1 + x_2 - 2 < 0}$,
- And the region above this line, where the output is **1**, is given by

  $\boldsymbol{x_1 + x_2 - 2 \geq 0}$

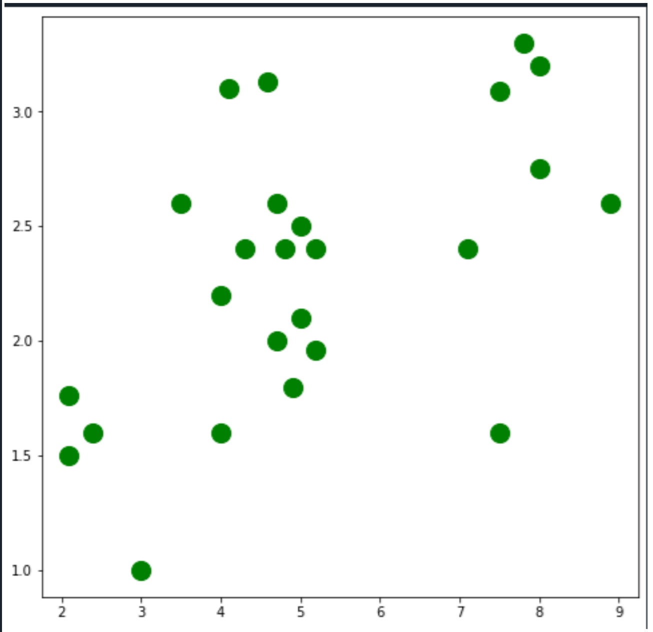
- So a perceptron can learn only **linear separable functions** and there are not many such functions!

Linear Regression with Perceptron

https://www.analyticsvidhya.com/blog/2021/11/a-comprehensive-guide-to-linear-regression-with-pytorch/

```
import numpy as np
import matplotlib.pyplot as plt
import torch
from torch.autograd import Variable
#Dataset is created using NumPy arrays.
x_train = np.array ([[4.7], [2.4], [7.5], [7.1], [4.3],
                     [7.8], [8.9], [5.2], [4.59], [2.1],
                     [8], [5], [7.5], [5], [4],
                     [8], [5.2], [4.9], [3], [4.7],
                     [4], [4.8], [3.5], [2.1], [4.1]],
                    dtype = np.float32)

y_train = np.array ([[2.6], [1.6], [3.09], [2.4], [2.4],
                     [3.3], [2.6], [1.96], [3.13], [1.76],
                     [3.2], [2.1], [1.6], [2.5], [2.2],
                     [2.75], [2.4], [1.8], [1], [2],
                     [1.6], [2.4], [2.6], [1.5], [3.1]],
                    dtype = np.float32)
#Visualizing the data.
plt.figure(figsize=(8,8))
plt.scatter(x_train, y_train, c='green', s=200, label='Original data')
plt.show()
```



Linear Regression with Perceptron

https://www.analyticsvidhya.com/blog/2021/11/a-comprehensive-guide-to-linear-regression-with-pytorch/

```python
import numpy as np
import matplotlib.pyplot as plt
import torch
from torch.autograd import Variable
#Dataset is created using NumPy arrays.
x_train = np.array ([[4.7], [2.4], [7.5], [7.1], [4.3],
                     [7.8], [8.9], [5.2], [4.59], [2.1],
                     [8], [5], [7.5], [5], [4],
                     [8], [5.2], [4.9], [3], [4.7],
                     [4], [4.8], [3.5], [2.1], [4.1]],
                    dtype = np.float32)
y_train = np.array ([[2.6], [1.6], [3.09], [2.4], [2.4],
                     [3.3], [2.6], [1.96], [3.13], [1.76],
                     [3.2], [2.1], [1.6], [2.5], [2.2],
                     [2.75], [2.4], [1.8], [1], [2],
                     [1.6], [2.4], [2.6], [1.5], [3.1]],
                    dtype = np.float32)
#Visualizing the data.
plt.figure(figsize=(8,8))
plt.scatter(x_train, y_train, c='green', s=200, label='Original data')
plt.show()
X_train = torch.from_numpy(x_train) #Convert numpy arrays intp Pytorch arrays
Y_train = torch.from_numpy(y_train)  #Convert numpy arrays intp Pytorch arrays
print('requires_grad for X_train: ', X_train.requires_grad)
print('requires_grad for Y_train: ', Y_train.requires_grad)
```

X_train

```
tensor([[4.7000],
        [2.4000],
        [7.5000],
        [7.1000],
        [4.3000],
        [7.8000],
        [8.9000],
        [5.2000],
        [4.5900],
        [2.1000],
        [8.0000],
        [5.0000],
        [7.5000],
        [5.0000],
        [4.0000],
        [8.0000],
        [5.2000],
        [4.9000],
        [3.0000],
        [4.7000],
        [4.0000],
        [4.8000],
        [3.5000],
        [2.1000],
        [4.1000]])
```

Y_train

```
tensor([[2.6000],
        [1.6000],
        [3.0900],
        [2.4000],
        [2.4000],
        [3.3000],
        [2.6000],
        [1.9600],
        [3.1300],
        [1.7600],
        [3.2000],
        [2.1000],
        [1.6000],
        [2.5000],
        [2.2000],
        [2.7500],
        [2.4000],
        [1.8000],
        [1.0000],
        [2.0000],
        [1.6000],
        [2.4000],
        [2.6000],
        [1.5000],
        [3.1000]])
```

Linear Regression with Perceptron

https://www.analyticsvidhya.com/blog/2021/11/a-comprehensive-guide-to-linear-regression-with-pytorch/

```
#Parameters of the model are W1 and b1, which is weight and bias respectively.
#Let us have a look at the parameters that are defined.
input_size = 1
hidden_size = 1
output_size = 1
learning_rate = 0.001
w1 = torch.rand(input_size,
                hidden_size,
                requires_grad=True)

b1 = torch.rand(hidden_size,
                output_size,
                requires_grad=True)
```

```
for iter in range(1, 5000):
    y_pred = X_train.mm(w1).clamp(min=0).add(b1)
    loss = (y_pred - Y_train).pow(2).sum()
    if iter % 100 ==0:
        print(iter, loss.item())
    loss.backward()
    with torch.no_grad():
        w1 -= learning_rate * w1.grad
        b1 -= learning_rate * b1.grad
        w1.grad.zero_()
        b1.grad.zero_()
#Let us check the optimized value for W1 and b1:
```

```
2700 loss =  6.119896411895752
2800 loss =  6.119896411895752
2900 loss =  6.119896411895752
3000 loss =  6.119896411895752
3100 loss =  6.119896411895752
3200 loss =  6.119896411895752
3300 loss =  6.119896411895752
3400 loss =  6.119896411895752
3500 loss =  6.119896411895752
3600 loss =  6.119896411895752
3700 loss =  6.119896411895752
3800 loss =  6.119896411895752
3900 loss =  6.119896411895752
4000 loss =  6.119896411895752
4100 loss =  6.119896411895752
4200 loss =  6.119896411895752
4300 loss =  6.119896411895752
4400 loss =  6.119896411895752
4500 loss =  6.119896411895752
4600 loss =  6.119896411895752
4700 loss =  6.119896411895752
4800 loss =  6.119896411895752
4900 loss =  6.119896411895752
```

Linear Regression with Perceptron

https://www.analyticsvidhya.com/blog/2021/11/a-comprehensive-guide-to-linear-regression-with-pytorch/

```
#Let us check the optimized value for W1 and b1:
print ('w1: ', w1)
print ('b1: ', b1)
#Getting the prediction values using the weights in the linear equation.
getX = float(input("Enter x_tain value: "))
predicted_in_tensor = getX * w1 + b1
print ("The predicted output = ", Variable(predicted_in_tensor) )
```

```
w1:  tensor([[0.1751]], requires_grad=True)
b1:  tensor([[1.4045]], requires_grad=True)
Enter x_tain value: 2.4
The predicted output =  tensor([[1.8247]])
```

Perceptron Exercises

1. Create two input AND functions and two input OR functions from perceptron (assume suitable weights and threshold value)
2. Train a perceptron for getting 2-input OR function. Assume w1 =0.3, w2 = -0.1, θ = 0.2 and $\propto$ = 0.1
3. Consider a perceptron with two real-valued inputs and an output unit with a *sigmoid activation* function. All the initial weights and the bias (threshold) equal 0.5. Assume that the output should be 1 for the input x1 = 0.7 and x2 = -0.6. Show how the delta rule supports the training of the neuron (assume α = 0.1)
4. Predict the value of **BMI** (Body Mass Index) from the ***Height*** and ***Weight*** of a person using a **perceptron**. Use the ***bmi.csv*** dataset for training (ignore its '***gender***' column). **Don't use any Python perceptron library**; use the perceptron learning steps from this lecture slide (use the *sigmoid activation* function and modify your dataset accordingly).

Structure of a Multilayer ANN

- The structure of an **Multilayer ANN** is shown in **Figure 13:**

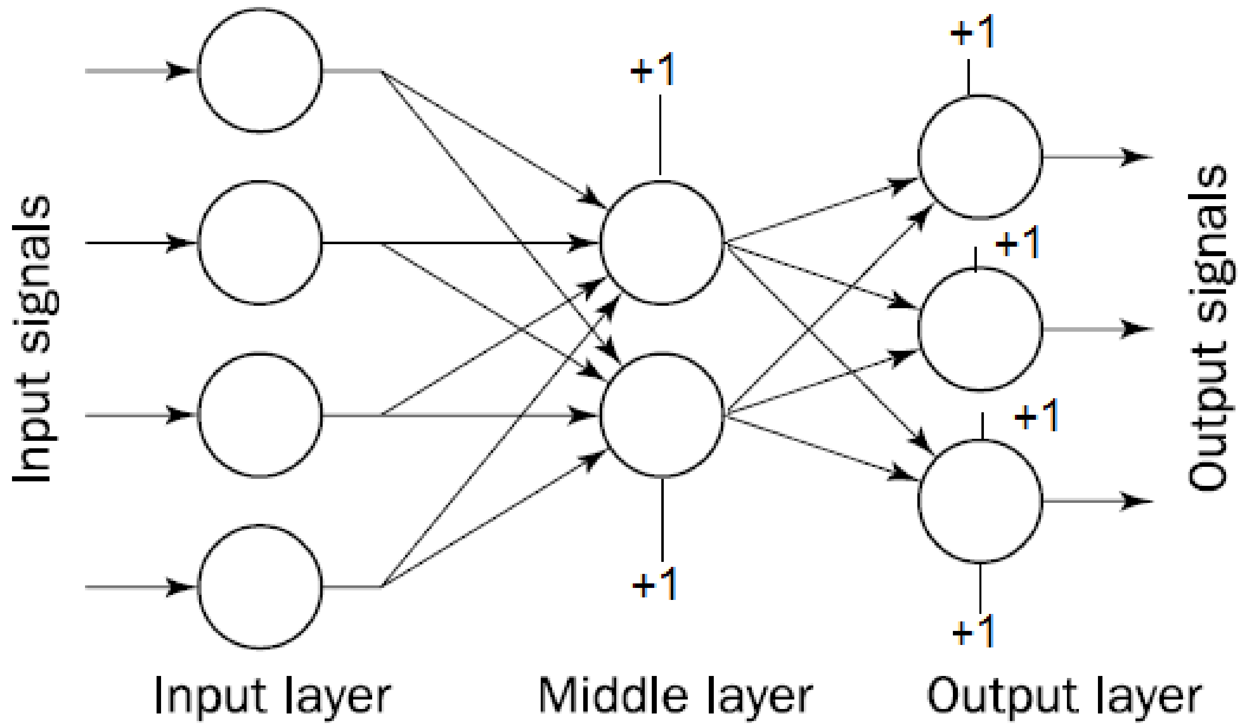


Figure 13: The structure of an ANN