UNIT : 1 Review of Neural Networks

What is Artificial Neural Network?

Artificial Neural Network ANN is an efficient computing system whose central theme is borrowed from the analogy of biological neural networks. ANNs are also named as “artificial neural systems,” or “parallel distributed processing systems,” or “connectionist systems.” ANN acquires a large collection of units that are interconnected in some pattern to allow communication between the units. These units, also referred to as nodes or neurons, are simple processors which operate in parallel.

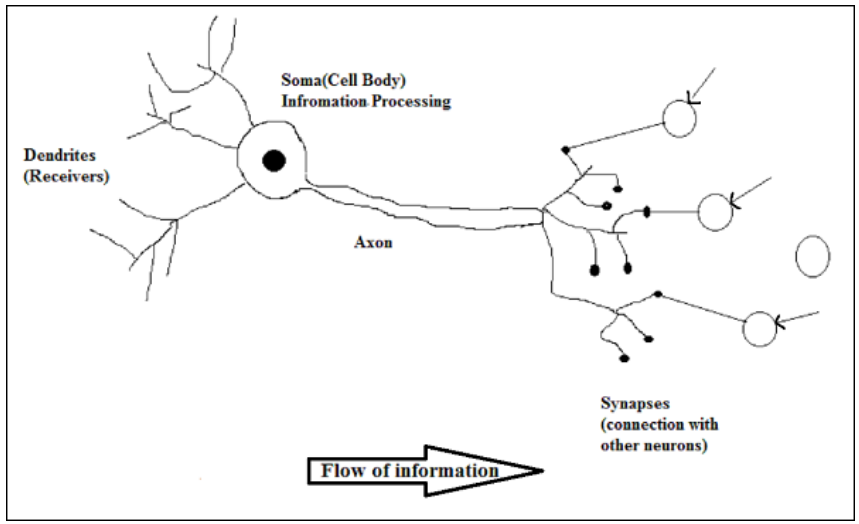
Every neuron is connected with other neuron through a connection link. Each connection link is associated with a weight that has information about the input signal. This is the most useful information for neurons to solve a particular problem because the weight usually excites or inhibits the signal that is being communicated. Each neuron has an internal state, which is called an activation signal. Output signals, which are produced after combining the input signals and activation rule, may be sent to other units.

Ref: https://www.tutorialspoint.com/artificial_neural_network

Biological Neuron

A nerve cell neuron is a special biological cell that processes information. According to an estimation, there are huge number of neurons, approximately 10^{11} with numerous interconnections, approximately 10^{15}.

Schematic Diagram



Working of a Biological Neuron

As shown in the above diagram, a typical neuron consists of the following four parts with the help of which we can explain its working −

- **Dendrites** − They are tree-like branches, responsible for receiving the information from other neurons it is connected to. In other sense, we can say that they are like the ears of neuron.
- **Soma** − It is the cell body of the neuron and is responsible for processing of information, they have received from dendrites.
- **Axon** − It is just like a cable through which neurons send the information.
- **Synapses** − It is the connection between the axon and other neuron dendrites.

ANN versus BNN

Before taking a look at the differences between Artificial Neural Network ANN and Biological Neural Network BNN, let us take a look at the similarities based on the terminology between these two.

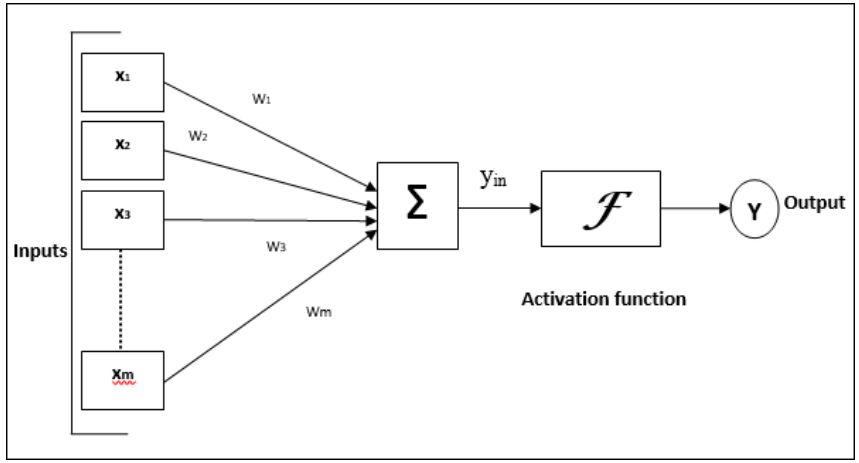
Biological Neural Network BNN	**Artificial Neural Network** ANN
Soma	Node
Dendrites	Input
Synapse	Weights or Interconnections
Axon	Output

The following table shows the comparison between ANN and BNN based on some criteria mentioned.

Criteria	**BNN**	**ANN**
Processing	Massively parallel, slow but superior than ANN	Massively parallel, fast but inferior than BNN
Size	10^{11} neurons and 10^{15} interconnections	10^{2} to 10^{4} nodes mainlydependsonthetypeofapplicationandnetworkdesignermainlydependsonthetypeofapplicationandnetworkdesigner
Learning	They can tolerate ambiguity	Very precise, structured and formatted data is required to tolerate ambiguity
	ty	
Fault tolerance	Performance degrades with even partial damage	It is capable of robust performance, hence has the potential to be fault tolerant
Storage capacity	Stores the information in the synapse	Stores the information in continuous memory locations

Model of Artificial Neural Network

The following diagram represents the general model of ANN followed by its processing.



For the above general model of artificial neural network, the net input can be calculated as follows −

yin=x1.w1+x2.w2+x3.w3…xm.wm

i.e., Net input $y_{in} = \sum_i^m x_i . w_i$

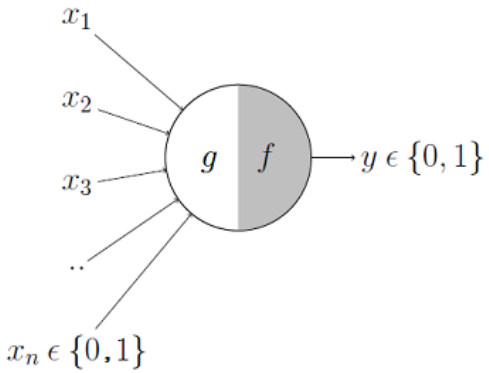
The output can be calculated by applying the activation function over the net input.

Y=F(yin)

Output = function (netinputcalculated)

McCulloch-Pitts Neuron

The first computational model of a neuron was proposed by Warren MuCulloch (neuroscientist) and Walter Pitts (logician) in 1943.



This is where it all began..

It may be divided into 2 parts. The first part, ***g*** takes an input, performs an aggregation and based on the aggregated value the second part, ***f*** makes a decision.

Lets suppose that I want to predict my own decision, whether to watch a random football game or not on TV. The inputs are all boolean i.e., {0,1} and my output variable is also boolean {0: Will watch it, 1: Won't watch it}.

- So, ***x_1*** could be *isPremierLeagueOn* (I like Premier League more)
- ***x_2*** could be *isItAFriendlyGame* (I tend to care less about the friendlies)
- ***x_3*** could be *isNotHome* (Can't watch it when I'm running errands. Can I?)
- ***x_4*** could be *isManUnitedPlaying* (I am a big Man United fan. GGMU!) and so on.

These inputs can either be *excitatory* or *inhibitory*. Inhibitory inputs are those that have maximum effect on the decision making irrespective of other inputs i.e., if ***x_3*** is 1 (not home) then my output will always be 0 i.e., the neuron will never fire, so ***x_3*** is an inhibitory input. Excitatory inputs are NOT the ones that will make the neuron fire on their own but they might fire it when combined together. Formally, this is what is going on:

$$g(x_1, x_2, x_3, ..., x_n) = g(\mathbf{x}) = \sum_{i=1}^{n} x_i$$

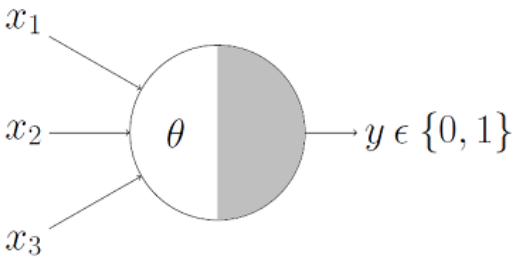
$$\begin{aligned} y = f(g(\mathbf{x})) &= 1 \quad if \quad g(\mathbf{x}) \geq \theta \\ &= 0 \quad if \quad g(\mathbf{x}) < \theta \end{aligned}$$

We can see that ***g*(x)** is just doing a sum of the inputs — a simple aggregation. And ***theta*** here is called thresholding parameter. For example, if I always watch the game when the sum turns out to be 2 or more, the ***theta*** is 2 here. This is called the Thresholding Logic.

Boolean Functions Using M-P Neuron

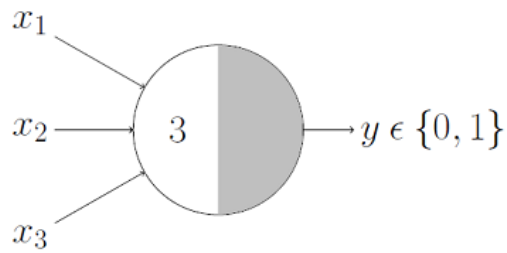
So far we have seen how the M-P neuron works. Now lets look at how this very neuron can be used to represent a few boolean functions. Mind you that our inputs are all boolean and the output is also boolean so essentially, the neuron is just trying to learn a boolean function. A lot of boolean decision problems can be cast into this, based on appropriate input variables— like whether to continue reading this post, whether to watch Friends after reading this post etc. can be represented by the M-P neuron.

M-P Neuron: A Concise Representation



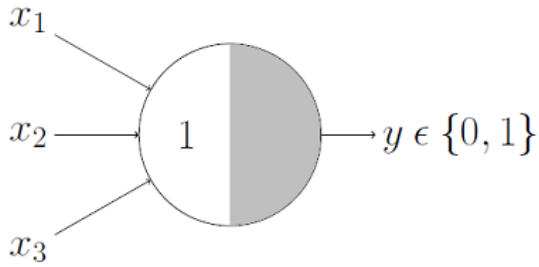
This representation just denotes that, for the boolean inputs ***x_1***, ***x_2*** and ***x_3*** if the ***g*(x)** i.e., **sum ≥ theta**, the neuron will fire otherwise, it won't.

AND Function



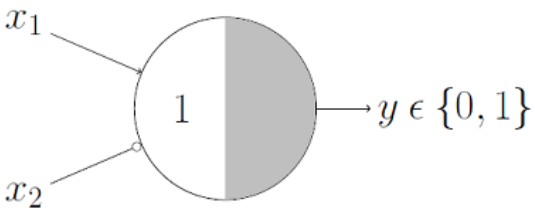
An AND function neuron would only fire when ALL the inputs are ON i.e., ***g*(x)** ≥ 3 here.

OR Function



I believe this is self explanatory as we know that an OR function neuron would fire if ANY of the inputs is ON i.e., ***g*(x)** ≥ 1 here.

A Function With An Inhibitory Input



$x_1\ AND\ !x_2$*

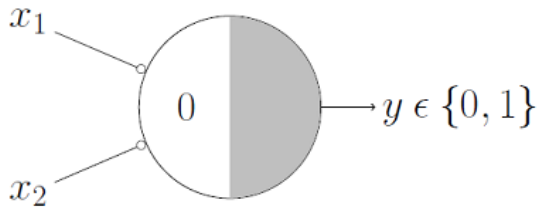
Now this might look like a tricky one but it's really not. Here, we have an inhibitory input i.e., ***x_2*** so whenever ***x_2*** is 1, the output will be 0. Keeping that in mind, we know that ***x_1 AND !x_2*** would output 1 only when ***x_1*** is 1 and ***x_2*** is 0 so it is obvious that the threshold parameter should be 1.

Lets verify that, the ***g*(x)** i.e., ***x_1*** + ***x_2*** would be ≥ 1 in only 3 cases:

Case 1: when ***x_1*** is 1 and ***x_2*** is 0
Case 2: when ***x_1*** is 1 and ***x_2*** is 1
Case 3: when ***x_1*** is 0 and ***x_2*** is 1

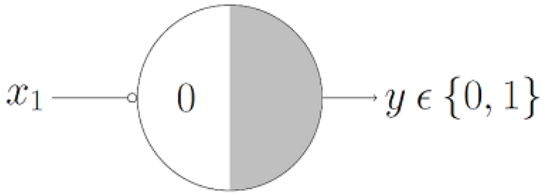
But in both Case 2 and Case 3, we know that the output will be 0 because ***x_2*** is 1 in both of them, thanks to the inhibition. And we also know that ***x_1 AND !x_2*** would output 1 for Case 1 (above) so our thresholding parameter holds good for the given function.

NOR Function



For a NOR neuron to fire, we want ALL the inputs to be 0 so the thresholding parameter should also be 0 and we take them all as inhibitory input.

NOT Function



For a NOT neuron, 1 outputs 0 and 0 outputs 1. So we take the input as an inhibitory input and set the thresholding parameter to 0. It works!

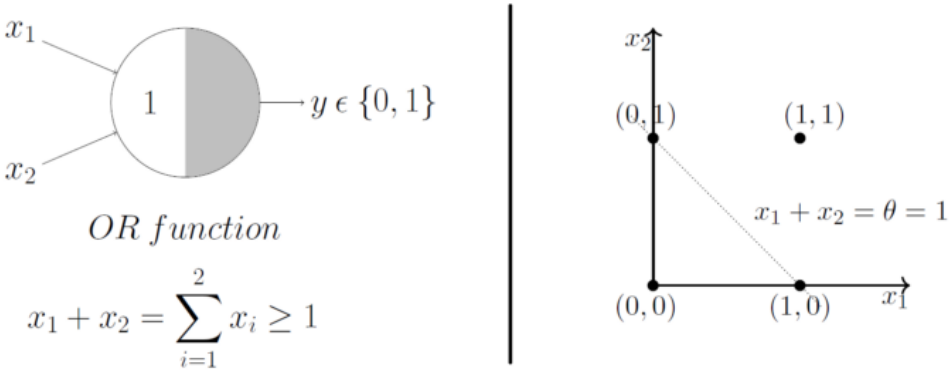
Geometric Interpretation Of M-P Neuron

This is the best part of the post according to me. Lets start with the OR function.

OR Function

We already discussed that the OR function's thresholding parameter ***theta*** is 1, for obvious reasons. The inputs are obviously boolean, so only 4 combinations are possible — (0,0), (0,1), (1,0) and (1,1). Now plotting them on a 2D graph and making use of the OR function's aggregation equation

i.e., ***x_1 + x_2*** ≥ **1** using which we can draw the decision boundary as shown in the graph below. Mind you again, this is not a real number graph.

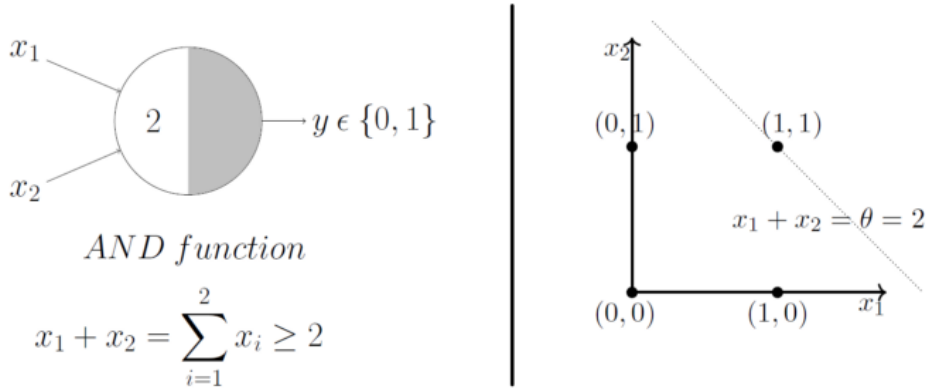


We just used the aggregation equation i.e., ***x_1 + x_2*** =**1** to graphically show that all those inputs whose output when passed through the OR function M-P neuron lie ON or ABOVE that line and all the input points that lie BELOW that line are going to output 0.

Voila!! The M-P neuron just learnt a linear decision boundary! The M-P neuron is splitting the input sets into two classes — positive and negative. Positive ones (which output 1) are those that lie ON or ABOVE the decision boundary and negative ones (which output 0) are those that lie BELOW the decision boundary.

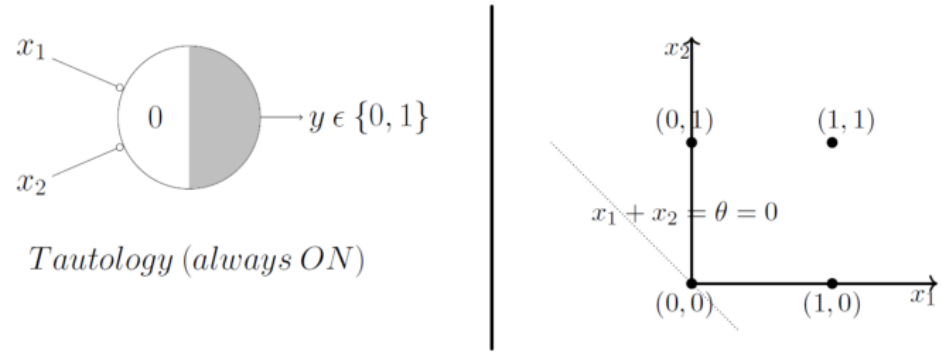
Lets convince ourselves that the M-P unit is doing the same for all the boolean functions by looking at more examples (if it is not already clear from the math).

AND Function

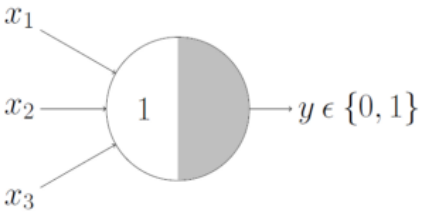


In this case, the decision boundary equation is ***x_1 + x_2*** =**2**. Here, all the input points that lie ON or ABOVE, just (1,1), output 1 when passed through the AND function M-P neuron. It fits! The decision boundary works!

Tautology

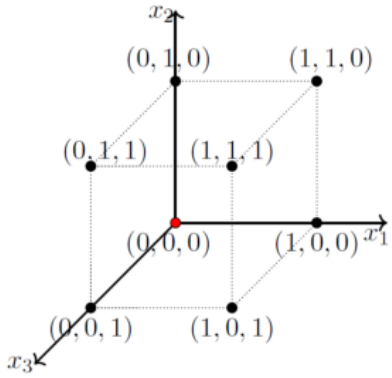


OR Function With 3 Inputs



OR function

$$x_1 + x_2 + x_3 = \sum_{i=1}^{3} x_i \geq 1$$

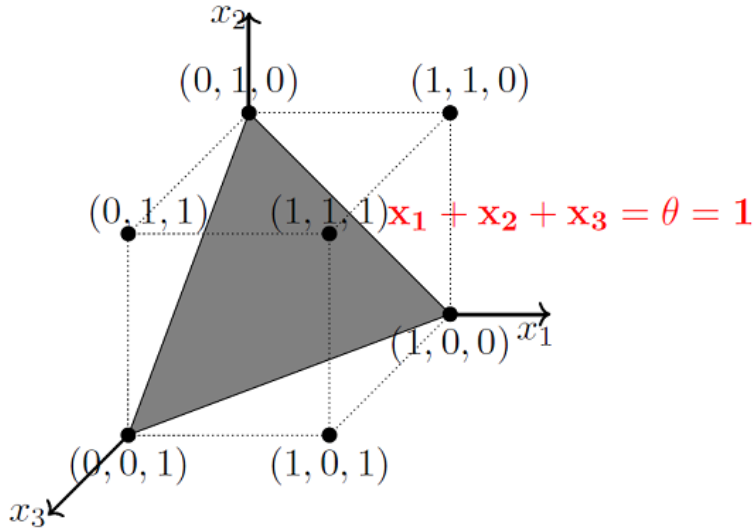


Lets just generalize this by looking at a 3 input OR function M-P unit. In this case, the possible inputs are 8 points — (0,0,0), (0,0,1), (0,1,0), (1,0,0), (1,0,1),… you got the point(s). We can map these on a 3D graph and this time we draw a decision boundary in 3 dimensions.

"Is it a line? Is it a plane?"

Yes, it is a PLANE!

The plane that satisfies the decision boundary equation ***x_1 + x_2 + x_3 = 1*** is shown below:



Just by hand coding a thresholding parameter, M-P neuron is able to conveniently represent the boolean functions which are linearly separable.

Linear separability (for boolean functions): There exists a line (plane) such that all inputs which produce a 1 lie on one side of the line (plane) and all inputs which produce a 0 lie on other side of the line (plane).

Limitations Of M-P Neuron

- What about non-boolean (say, real) inputs?
- Do we always need to hand code the threshold?
- Are all inputs equal? What if we want to assign more importance to some inputs?

- What about functions which are not linearly separable? Say XOR function.

Activation Functions

“In artificial neural networks, each neuron forms a weighted sum of its inputs and passes the resulting scalar value through a function referred to as an activation function.”

Why do we need activation functions?

An activation function determines if a neuron should be **activated or not activated**. This implies that it will use some simple mathematical operations to determine if the neuron’s input to the network is relevant or not relevant in the prediction process.

The ability to introduce **non-linearity** to an artificial neural network and generate output from a collection of input values fed to a layer is the purpose of the activation function.

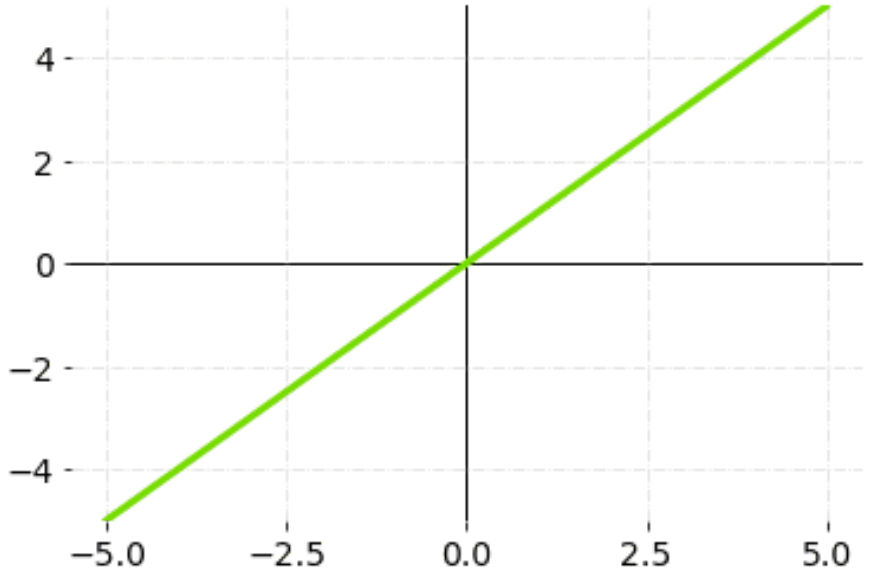
Types of Activation functions

Activation functions can be divided into three types:

1. Linear Activation Function
2. Binary Step Function
3. Non-linear Activation Functions

Linear Activation Function

The linear activation function, often called the **identity activation function**, is proportional to the input. The range of the linear activation function will be (-∞ to ∞). The linear activation function simply adds up the weighted total of the inputs and returns the result.



Linear Activation Function—Graph

Mathematically, it can be represented as:

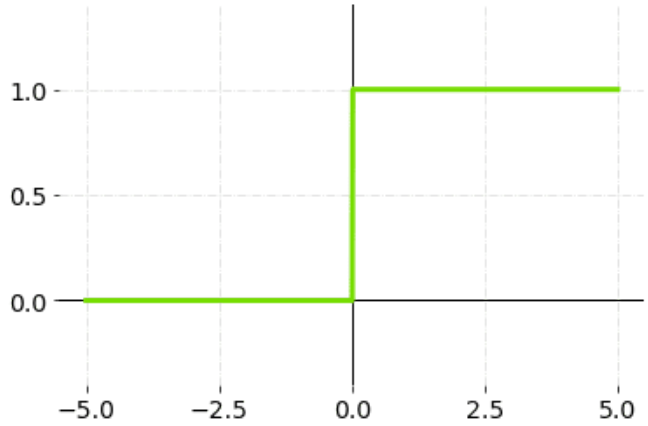
$$f(x) = x$$

Linear Activation Function—Equation

Binary Step Activation Function

A **threshold value** determines whether a neuron should be activated or not activated in a binary step activation function.

The activation function compares the input value to a threshold value. If the input value is greater than the threshold value, the neuron is activated. It's disabled if the input value is less than the threshold value, which means its output isn't sent on to the next or hidden layer.



Binary Step Function—Graph

Mathematically, the binary activation function can be represented as:

$$f(x) = \begin{cases} 0 & for\ x < 0 \\ 1 & for\ x \geqslant 0 \end{cases}$$

Binary Step Activation Function—Equation

Non-linear Activation Functions

The non-linear activation functions are the most-used activation functions. They make it uncomplicated for an artificial neural network model to adapt to a variety of data and to differentiate between the outputs.

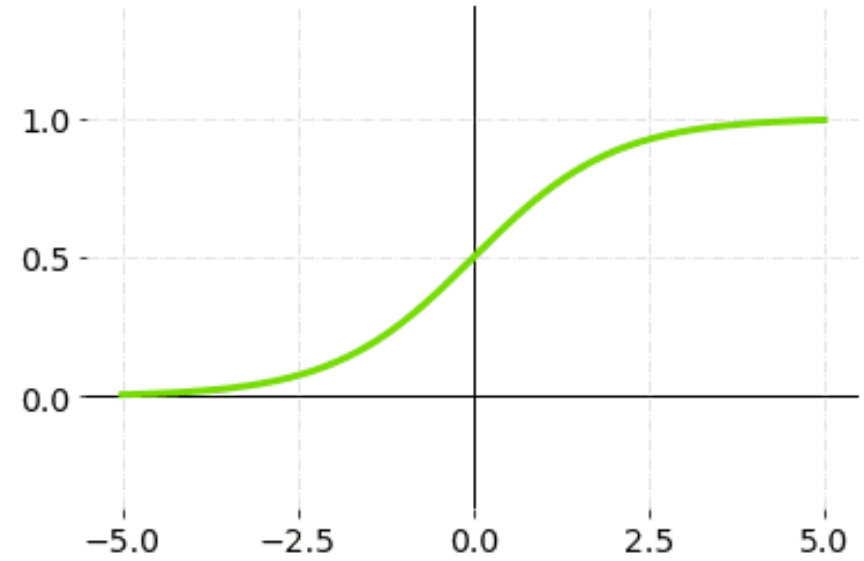
Non-linear activation functions allow the stacking of multiple layers of neurons, as the output would now be a non-linear combination of input passed through multiple layers. Any output can be represented as a functional computation output in a neural network.

These activation functions are mainly divided basis on their range and curves. The remainder of this article will outline the major non-linear activiation functions used in neural networks.

1. Sigmoid

Sigmoid accepts a number as input and returns a number between 0 and 1. It's simple to use and has all the desirable qualities of activation functions: nonlinearity, continuous differentiation, monotonicity, and a set output range.

This is mainly used in **binary classification problems.** This sigmoid function gives the probability of an existence of a particular class.



Sigmoid Activation Function—Graph

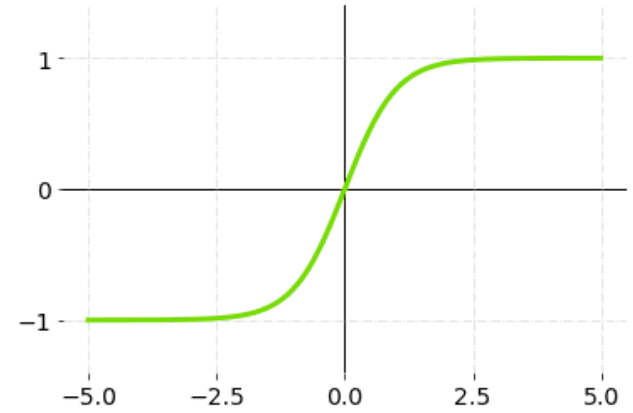
Mathematically, it can be represented as:

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

Sigmoid Activation Function—Equation

2.TanH (Hyperbolic Tangent)

TanH compress a real-valued number to the range **[-1, 1]**. It's non-linear, But it's different from Sigmoid,and its output is **zero-centered**. The main advantage of this is that the negative inputs will be mapped strongly to the negative and zero inputs will be mapped to almost zero in the graph of TanH.



TanH Activation Function—Graph

Mathematically, TanH function can be represented as:

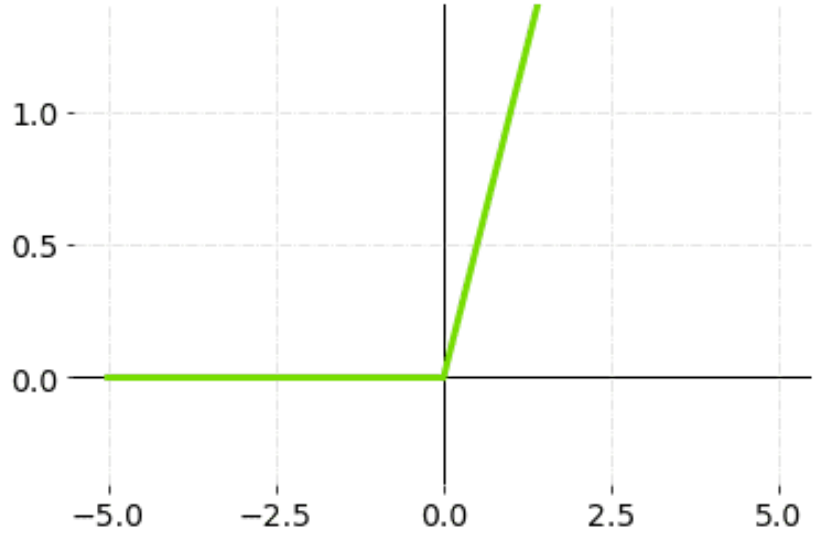
$$f(x) = \frac{\left(e^{x} - e^{-x}\right)}{\left(e^{x} + e^{-x}\right)}$$

TanH Activation Function—Equation

3.ReLU (Rectified Linear Unit)

ReLU stands for Rectified Linear Unit and is one of the most commonly used activation function in the applications. It's solved the problem of vanishing gradient because the maximum value of the gradient of ReLU function is one. It

also solved the problem of saturating neuron, since the slope is never zero for ReLU function. The range of ReLU is between **0 and infinity.**



ReLU Activation Function—Graph

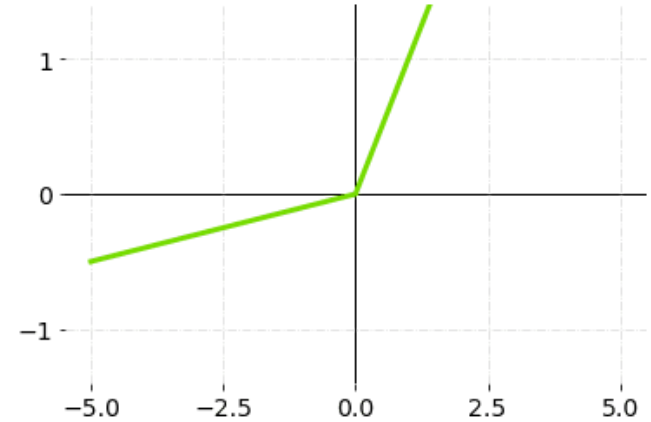
Mathematically, it can be represented as:

$$f(x) = max\ (0, x)$$

ReLU Activation Function—Equation

4.Leaky ReLU

Leaky ReLU is an upgraded version of the ReLU activation function to solve the dying ReLU problem, as it has a small positive slope in the negative area. But, the consistency of the benefit across tasks is presently ambiguous.



Leaky ReLU Activation Function—Graph

Mathematically, it can be represented as,

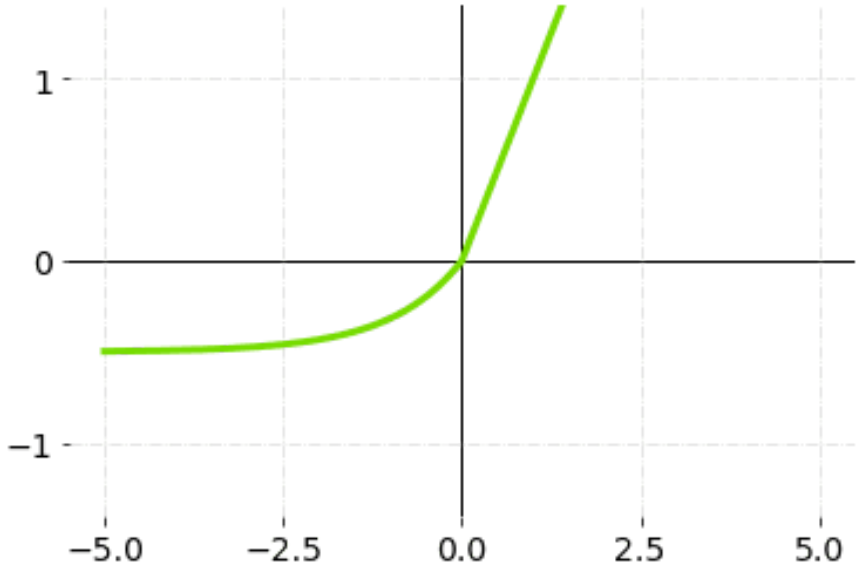
$$f(x) = max\ (0.1x, x)$$

Leaky ReLU Activation Function—Equation

5.ELU (Exponential Linear Units)

ELU is also one of the variations of ReLU which also solves the dead ReLU problem. ELU, just like leaky ReLU also considers negative values by introducing a new alpha parameter and multiplying it will another equation.

ELU is slightly more computationally expensive than leaky ReLU, and it's very similar to ReLU except negative inputs. They are both in identity function shape for positive inputs.



ELU Activation Function-Graph

Mathematically, it can be represented as:

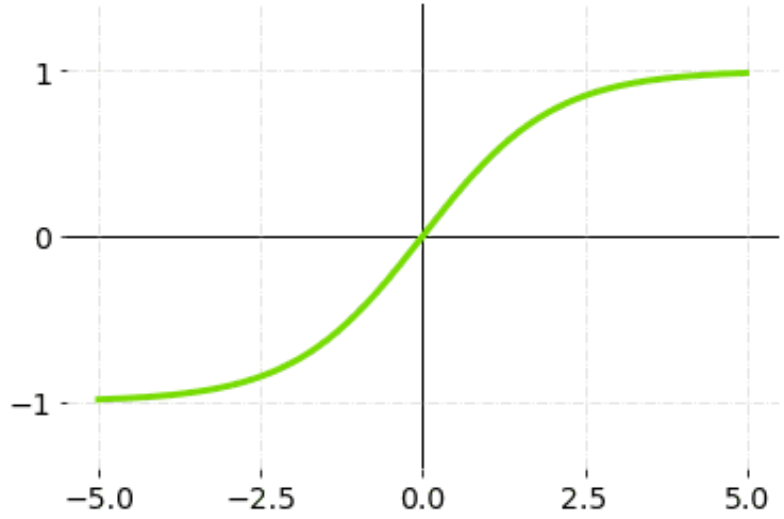
$$\begin{cases} x & for\ x \geqslant 0 \\ \alpha\left(e^{x}-1\right) & for\ x < 0 \end{cases}$$

ELU Activation Function—Equation

6. Softmax

A combination of many sigmoids is referred to as the Softmax function. It determines relative probability. Similar to the sigmoid activation function, the Softmax function returns the probability of each class/labels. **In multi-class classification, softmax activation function is most commonly used for the last layer of the neural network.**

The softmax function gives the probability of the current class with respect to others. This means that it also considers the possibility of other classes too.



Softmax Activation Function—Graph

Mathematically, it can be represented as:

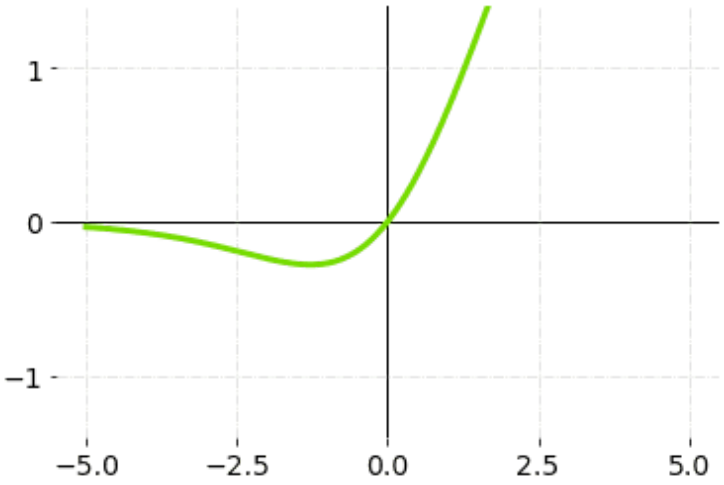
$$\text{Softmax}(x_i) = \frac{\exp(x_i)}{\sum_j \exp(x_j)}$$

Softmax Activation Function—Equation

7. Swish

Swish allows for the propagation of a few numbers of negative weights, whereas ReLU sets all non-positive weights to zero. This is a crucial property that determines the success of non-monotonic smooth activation functions, such as Swish's, in progressively deep neural networks.

It's a self-gated activation function created by Google researchers.



Swish Activation Function—Graph

Mathematically, it can be represented as:

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

Swish Activation Function—Equation

Important Considerations

While choosing the proper activation function, the following problems and issues must be considered:

Vanishing gradient is a common problem encountered during neural network training. Like a sigmoid activation function, some activation functions have a small output range (0 to 1). So a huge change in the input of the sigmoid activation function will create a small modification in the output. Therefore, the derivative also becomes small. These activation functions are only used for shallow networks with only a few layers. When these activation functions are applied to a multi-layer network, the gradient may become too small for expected training.

Exploding gradients are situations in which massive incorrect gradients build during training, resulting in huge updates to neural network model weights. When there are exploding gradients, an unstable network might form, and training cannot be completed. Due to exploding gradients, the weights' values can potentially grow to the point where they overflow, resulting in loss in **NaN** values.

Final Takeaways

- All hidden layers generally use the same activation functions. **ReLU** activation function **should only** be used in the hidden layer for better results.
- Sigmoid and TanH activation functions **should not be utilized** in hidden layers due to the vanishing gradient, since they make the model more susceptible to problems during training.
- Swish function is used in artificial neural networks having a depth **more than 40 layers.**
- Regression problems should use linear activation functions
- Binary classification problems should use the sigmoid activation function
- Multiclass classification problems shold use the softmax activation function

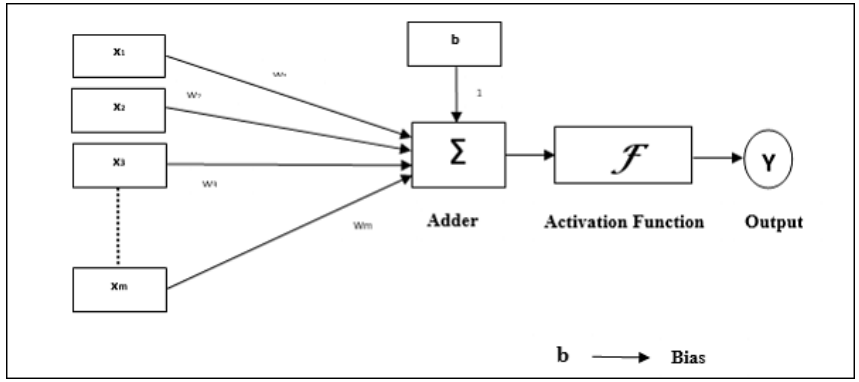
Neural network architecture and their usable activation functions,

- Convolutional Neural Network (CNN): ReLU activation function
- Recurrent Neural Network (RNN): TanH or sigmoid activation functions

Perceptron

Developed by Frank Rosenblatt by using McCulloch and Pitts model, perceptron is the basic operational unit of artificial neural networks. It employs supervised learning rule and is able to classify the data into two classes.

Operational characteristics of the perceptron: It consists of a single neuron with an arbitrary number of inputs along with adjustable weights, but the output of the neuron is 1 or 0 depending upon the threshold. It also consists of a bias whose weight is always 1. Following figure gives a schematic representation of the perceptron.



Perceptron thus has the following three basic elements −

- **Links** − It would have a set of connection links, which carries a weight including a bias always having weight 1.
- **Adder** − It adds the input after they are multiplied with their respective weights.
- **Activation function** − It limits the output of neuron. The most basic activation function is a Heaviside step function that has two possible outputs. This function returns 1, if the input is positive, and 0 for any negative input.

Training Algorithm

Perceptron network can be trained for single output unit as well as multiple output units.

Training Algorithm for Single Output Unit

Step 1 − Initialize the following to start the training −

- Weights
- Bias
- Learning rate α

For easy calculation and simplicity, weights and bias must be set equal to 0 and the learning rate must be set equal to 1.

Step 2 − Continue step 3-8 when the stopping condition is not true.

Step 3 − Continue step 4-6 for every training vector **x**.

Step 4 − Activate each input unit as follows −

$$x_i = s_i\ (i = 1\ to\ n)$$

Step 5 – Now obtain the net input with the following relation –

$$y_{in} = b + \sum_{i}^{n} x_i.\ w_i$$

Here **'b'** is bias and **'n'** is the total number of input neurons.

Step 6 – Apply the following activation function to obtain the final output.

$$f(y_{in}) = \begin{cases} 1 & if\ y_{in} > \theta \\ 0 & if\ -\theta \leqslant y_{in} \leqslant \theta \\ -1 & if\ y_{in} < -\theta \end{cases}$$

Step 7 – Adjust the weight and bias as follows –

Case 1 – if **y ≠ t** then,

$$w_i(new) = w_i(old) + \alpha\, t x_i$$

$$b(new) = b(old) + \alpha t$$

Case 2 – if **y = t** then,

$$w_i(new) = w_i(old)$$

$$b(new) = b(old)$$

Here **'y'** is the actual output and **'t'** is the desired/target output.

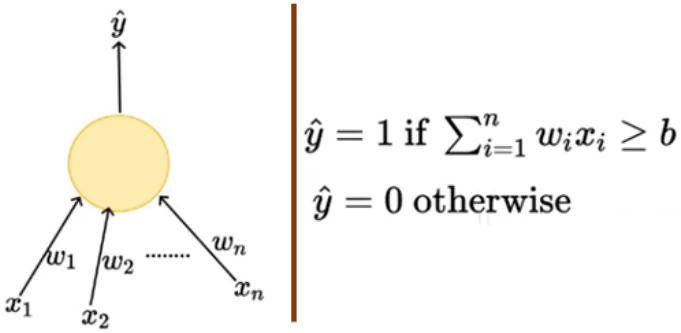
Step 8 – Test for the stopping condition, which would happen when there is no change in weight.

Sigmoid Neuron — Building Block of Deep Neural Networks

Why Sigmoid Neuron

Before we go into the working of a sigmoid neuron, let's talk about the perceptron model and its limitations in brief.

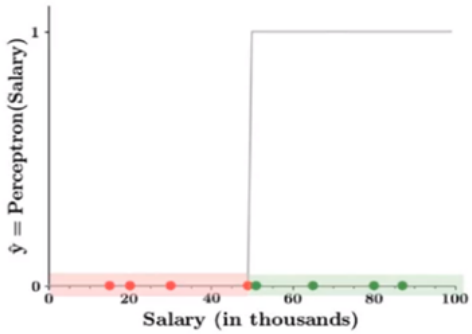
Perceptron model takes several real-valued inputs and gives a single binary output. In the perceptron model, every input `xi` has weight `wi` associated with it. The weights indicate the importance of the input in the decision-making process. The model output is decided by a threshold $\mathbf{W_0}$ if the weighted sum of the inputs is greater than threshold $\mathbf{W_0}$ output will be 1 else output will be 0. In other words, the model will fire if the weighted sum is greater than the threshold.



Perceptron (Left) & Mathematical Representation (Right)

From the mathematical representation, we might say that the thresholding logic used by the perceptron is very harsh. Let's see the harsh thresholding logic with an example. Consider the decision making process of a person, whether he/she would like to purchase a car or not based on only one input `x1`—Salary and by setting the threshold **b**(W_0) = -10 and the weight W_1 = 0.2. The output from the perceptron model will look like in the figure shown below.

Salary (in thousands)	Can buy a car?
80	1
20	0
65	1
15	0
30	0
49	0
51	1
87	1

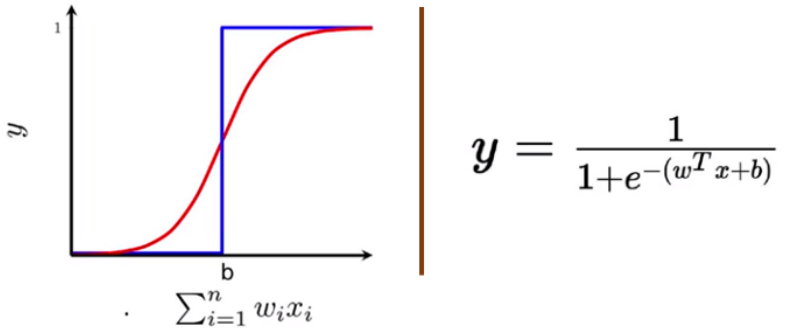


Data (Left) & Graphical Representation of Output(Right)

Red points indicates that a person would not buy a car and green points indicate that person would like to buy a car. Isn't it a bit odd that a person with 50.1K will buy a car but someone with a 49.9K will not buy a car? The small change in the input to a perceptron can sometimes cause the output to completely flip, say from 0 to 1. This behavior is not a characteristic of the specific problem we choose or the specific weight and the threshold we choose. It is a characteristic of the perceptron neuron itself which behaves like a step function. We can overcome this problem by introducing a new type of artificial neuron called a *sigmoid* neuron.

Sigmoid Neuron

Introducing sigmoid neurons where the output function is much smoother than the step function. In the sigmoid neuron, a small change in the input only causes a small change in the output as opposed to the stepped output. There are many functions with the characteristic of an "**S**" shaped curve known as sigmoid functions. The most commonly used function is the logistic function.



Sigmoid Neuron Representation (logistic function)

We no longer see a sharp transition at the threshold **b**. The output from the sigmoid neuron is not 0 or 1. Instead, it is a real value between 0–1 which can be interpreted as a probability.

The inputs to the sigmoid neuron can be real numbers unlike the boolean inputs in MP Neuron and the output will also be a real number between 0–1. In the sigmoid neuron, we are trying to regress the relationship between **X** and **Y** in terms of probability. Even though the output is between 0–1, we can still use the sigmoid function for binary classification tasks by choosing some threshold.

Gradient Descent Algorithm

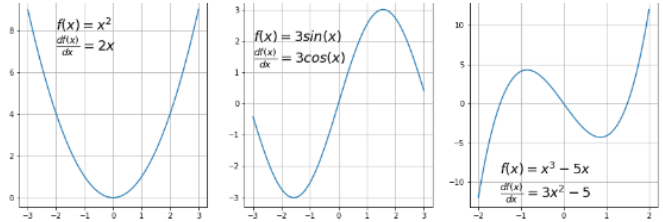
Gradient descent (GD) is an iterative first-order optimisation algorithm used to find a local minimum/maximum of a given function. This method is commonly used in *machine learning* (ML) and *deep learning*(DL) to minimise a cost/loss function (e.g. in a linear regression). Due to its importance and ease of implementation, this algorithm is usually taught at the beginning of almost all machine learning courses.

Function requirements

Gradient descent algorithm does not work for all functions. There are two specific requirements. A function has to be:

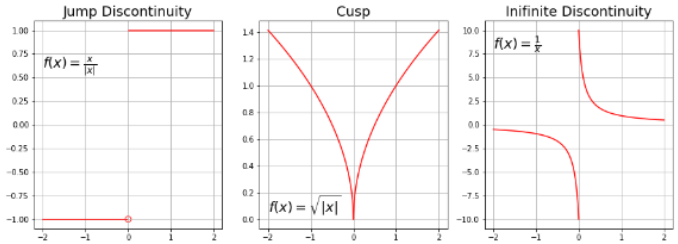
- **differentiable**
- **convex**

First, what does it mean it has to be **differentiable**? If a function is differentiable it has a derivative for each point in its domain — not all functions meet these criteria. First, let's see some examples of functions meeting this criterion:



Examples of differentiable functions; Image by author

Typical non-differentiable functions have a step a cusp or a discontinuity:



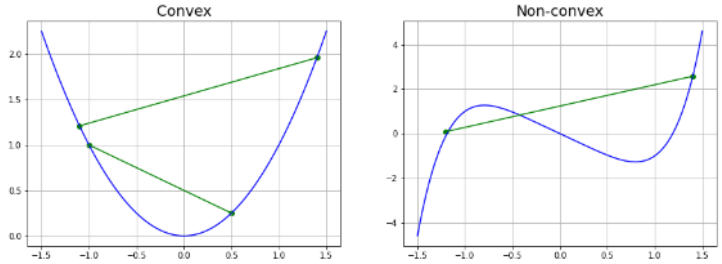
Next requirement — **function has to be convex**. For a univariate function, this means that the line segment connecting two function's points lays on or above its curve (it does not cross it). If it does it means that it has a local minimum which is not a global one.

Mathematically, for two points x , x laying on the function's curve this condition is expressed as:

$$f(\lambda x_1 + (1-\lambda)x_2) \leq \lambda f(x_1) + (1-\lambda)f(x_2)$$

where λ denotes a point's location on a section line and its value has to be between 0 (left point) and 1 (right point), e.g. λ=0.5 means a location in the middle.

Below there are two functions with exemplary section lines.



Exemplary convex and non-convex functions; Image by author

Another way to check mathematically if a univariate function is convex is to calculate the second derivative and check if its value is always bigger than 0.

$$\frac{d^2 f(x)}{dx^2} > 0$$

Let's investigate a simple quadratic function given by:

$$f(x) = x^2 - x + 3$$

Its first and second derivative are:

$$\frac{df(x)}{dx} = 2x - 1, \quad \frac{d^2 f(x)}{dx^2} = 2$$

Because the second derivative is always bigger than 0, our function is strictly convex.

Gradient Descent Algorithm

Gradient Descent Algorithm iteratively calculates the next point using gradient at the current position, scales it (by a learning rate) and subtracts obtained value from the current position (makes a step). It subtracts the value because we want to minimise the function (to maximise it would be adding). This process can be written as:

$$p_{n+1} = p_n - \eta \nabla f(p_n)$$

There's an important parameter **η** which scales the gradient and thus controls the step size. In machine learning, it is called **learning rate** and have a strong influence on performance.

- The smaller learning rate the longer GD converges, or may reach maximum iteration before reaching the optimum point
- If learning rate is too big the algorithm may not converge to the optimal point (jump around) or even to diverge completely.

In summary, Gradient Descent method's steps are:

1. choose a starting point (initialisation)
2. calculate gradient at this point
3. make a scaled step in the opposite direction to the gradient (objective: minimise)
4. repeat points 2 and 3 until one of the criteria is met:

- maximum number of iterations reached
- step size is smaller than the tolerance (due to scaling or a small gradient).

- This function takes 5 parameters:
- 1. **starting poin**t - in our case, we define it manually but in practice, it is often a random initialisation
- 2. **gradient functio**n - has to be specified before-hand
- 3. **learning rate** - scaling factor for step sizes
- 4. maximum number of iterations
- 5. tolerance to conditionally stop the algorithm (in this case a default value is 0.01)