**Deep Learning**

**Activation function**: It is also known as **Transfer Function**.

Every single moment our brain is trying to segregate the incoming information into the “useful” and “not-so-useful” categories. Activation Function helps the neural network to use important information while suppressing irrelevant data points. **Activation Function helps the neural network to use important information while suppressing irrelevant data points.**

**Why do we need Activation Functions?**

The purpose of an activation function is to provide some sort of non-linear property to the neural network. Without the activation functions, the neural network could only compute linear mappings from inputs to outputs.

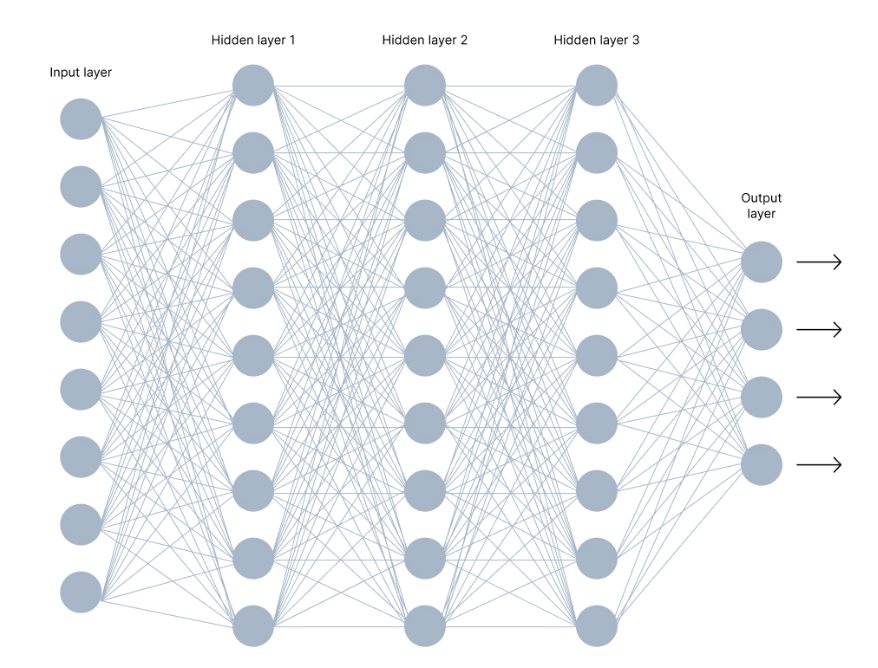
**Why is this so?**

Without the activation functions, the only mathematical operation during the forward propagation would consist of products between an input vector and a weight matrix.

**Since a single product is a linear operation, multiple consecutive products would be nothing more than multiple linear operations repeated in sequence. Multiple, successive linear operations can be considered as a single linear operation.**

To compute really interesting things, neural networks need to be able to approximate non-linear relationships between input vectors and outputs. The more complex the data we are trying to learn something from, the more “non-linear” the mapping from to tends to be.

A neural network that does not have an activation function in the hidden layer would not be able to mathematically realize such complex relationships, and would not be able to solve the tasks we are trying to solve with the network.



In the image above, you can see a neural network made of interconnected neurons. Each of them is characterized by its **weight**, **bias**, and **activation function**.

Here are other elements of this network.

#### **Input Layer**: The input layer takes raw input from the domain. No computation is performed at this layer. Nodes here just pass on the information (features) to the hidden layer.

#### **Hidden Layer**: As the name suggests, the nodes of this layer are not exposed. They provide an abstraction to the neural network. The hidden layer performs all kinds of computation on the features entered through the input layer and transfers the result to the output layer.

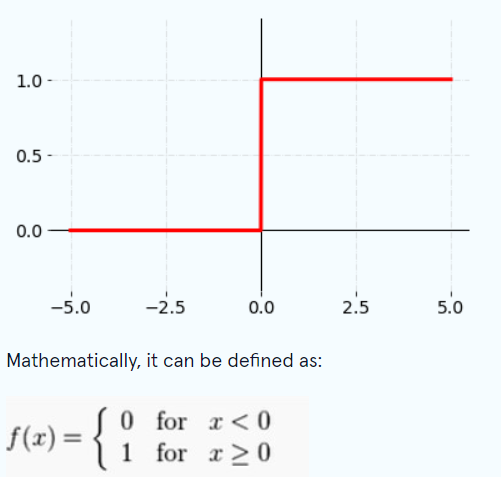
#### **Output Layer**: It’s the final layer of the network that brings the information learned through the hidden layer and delivers the final value as a result.

**Note***:*All hidden layers usually use the same activation function. However, the output layer will typically use a different activation function from the hidden layers. The choice depends on the goal or type of prediction made by the model.

**Feedforward Propagation** - the flow of information occurs in the forward direction. The input is used to calculate some intermediate function in the hidden layer, which is then used to calculate an output.

**Backpropagation** - the weights of the network connections are repeatedly adjusted to minimize the difference between the actual output vector of the net and the desired output vector.

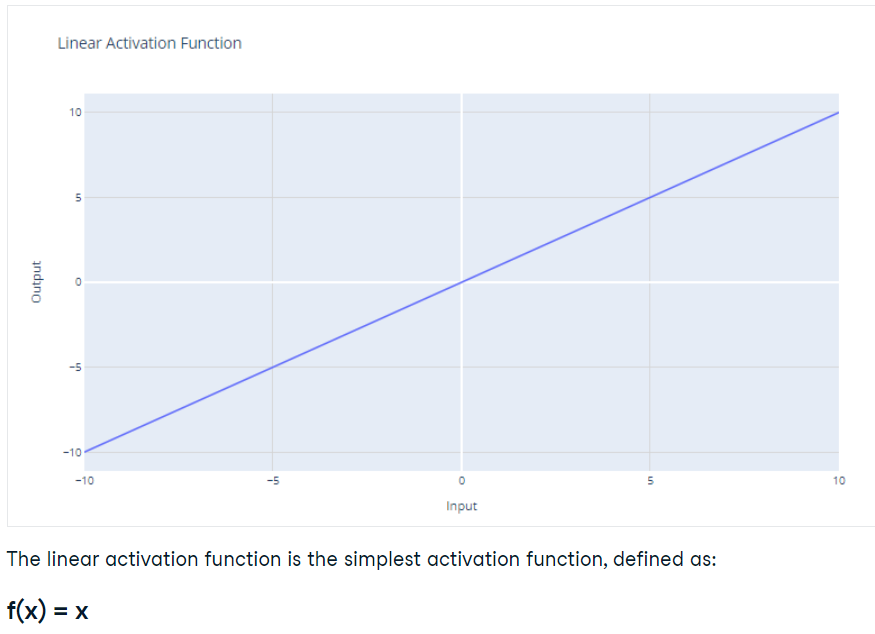
**Three types of activation function:**

**Binary step function**: Depends on a threshold value that decides whether a neuron should be activated or not. The input fed to the activation function is compared to a certain threshold; if the input is greater than it, then the neuron is activated, else it is deactivated, meaning that its output is not passed on to the next hidden layer. 

**Limitations**: Binary step function is commonly used in primitive neural networks without hidden layers. The binary step function can be used as an activation function while creating a binary classifier. However, the function is not helpful when there are multiple classes to deal with.  
Moreover, the **gradient of the step function is zero** which causes a hindrance in the backpropagation process. That is if you calculate the derivative of f(x) with respect to x, it comes out to be 0.

**Linear Activation Function:**

The linear activation function, also known as "no activation," or "identity function" (multiplied x1.0), is where the activation is proportional to the input.

****

**Limitations:** It’s not possible to use backpropagation as the derivative of the function is a constant and has no relation to the input x.

All layers of the neural network will collapse into one if a linear activation function is used. No matter the number of layers in the neural network, the last layer will still be a linear function of the first layer. So, essentially, a linear activation function turns the neural network into just one layer.

**Non-Linear Activation Function:**

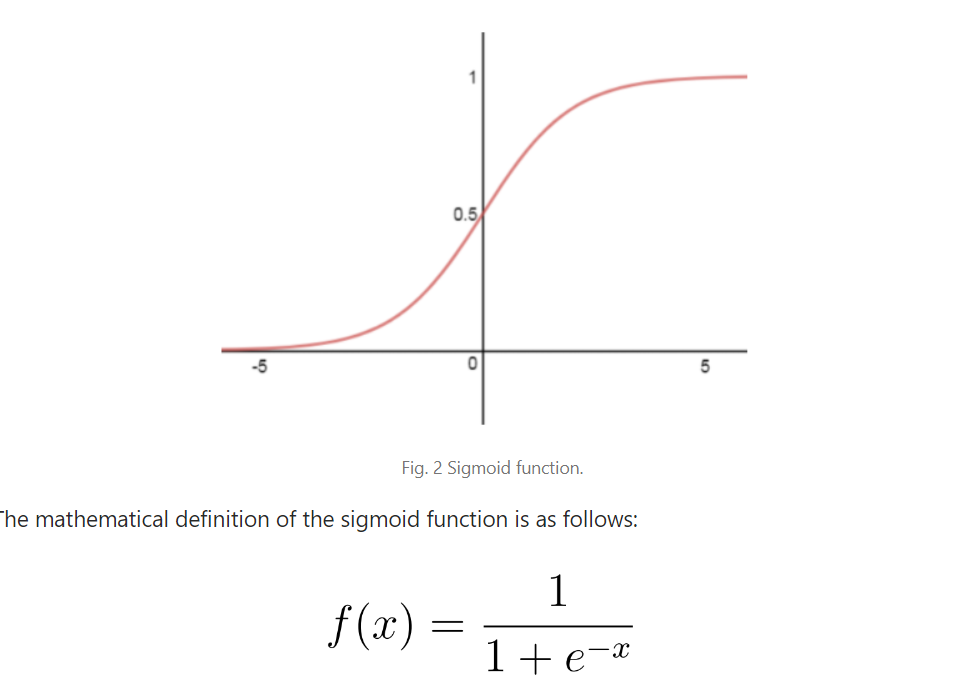
**Derivative or Differential:**Change in y-axis w.r.t. change in x-axis. It is also known as slope. When updating the curve, know in **which direction** and **how much** to change or update the curve depending on the slope. That is why we use differentiation in almost every part of Machine Learning and Deep Learning.

**Monotonic function:** A function which is either entirely non-increasing or non-decreasing.

Non-linear activation functions solve the following limitations of linear activation functions:

* They allow backpropagation because now the derivative function would be related to the input, and it’s possible to go back and understand which weights in the input neurons can provide a better prediction.
* They 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 in a neural network.

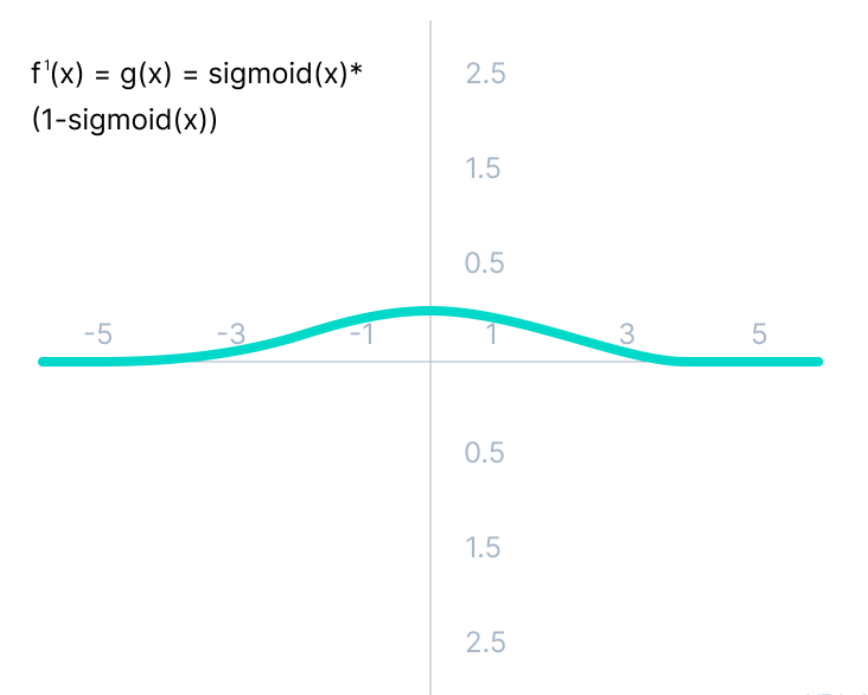
**Sigmoid / Logistic Activation Function:**  The sigmoid function maps the incoming inputs to a range between 0 and 1:



Here’s why the sigmoid/logistic activation function is one of the most widely used functions:

* It is commonly used for models where we have to predict the probability as an output. Since the probability of anything exists only between the range of 0 and 1, sigmoid is the right choice because of its range. The function is **monotonic**but **the function’s derivative is not.**
* The function is differentiable and provides a smooth gradient, i.e., preventing jumps in output values. This is represented by an S-shape of the sigmoid activation function.

**Limitations:**

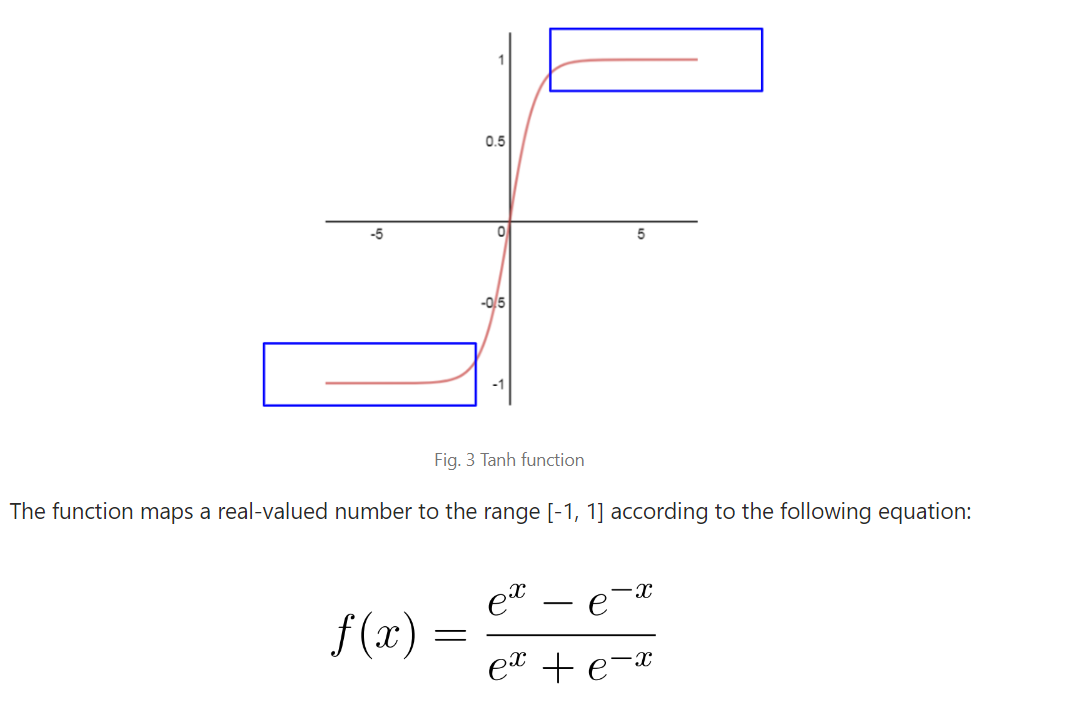
****

As we can see from the above Figure, the gradient values are only significant for the range -3 to 3, and the graph gets much flatter in other regions. It implies that for values greater than 3 or less than -3, the function will have very small gradients. As the gradient value approaches zero, the network ceases to learn and suffers from the **Vanishing gradient problem.**

The output of the **logistic function is not symmetric around zero.** So, the output of all the neurons will be of the same sign. This makes the [training of the neural network](https://www.v7labs.com/training) more difficult and unstable.

**Tanh Function (Hyperbolic Tangent):**

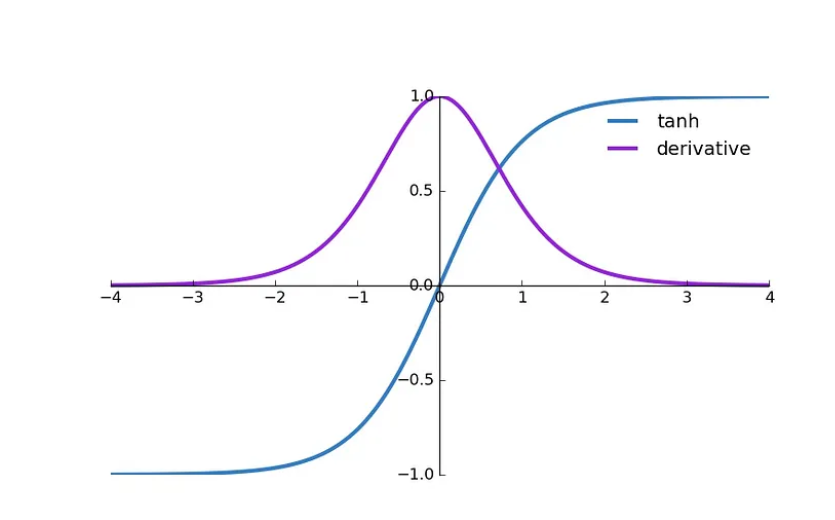
Tanh function is very similar to the sigmoid/logistic activation function and even has the same S-shape with the difference in the output range of -1 to 1.



The advantages of using this activation function are:

* The output of the tanh activation function is zero-centered. Hence, we can easily map the output values as strongly negative, neutral, or strongly positive. The function is **monotonic** while its **derivative is not monotonic**.Both tanh and logistic sigmoid activation functions are **used in feed-forward nets.**
* Usually used in hidden layers of a neural network as its values lie between -1 to 1 therefore, the mean for the hidden layer comes out to be 0 or very close to it. It helps in centering the data and makes learning for the next layer much easier.

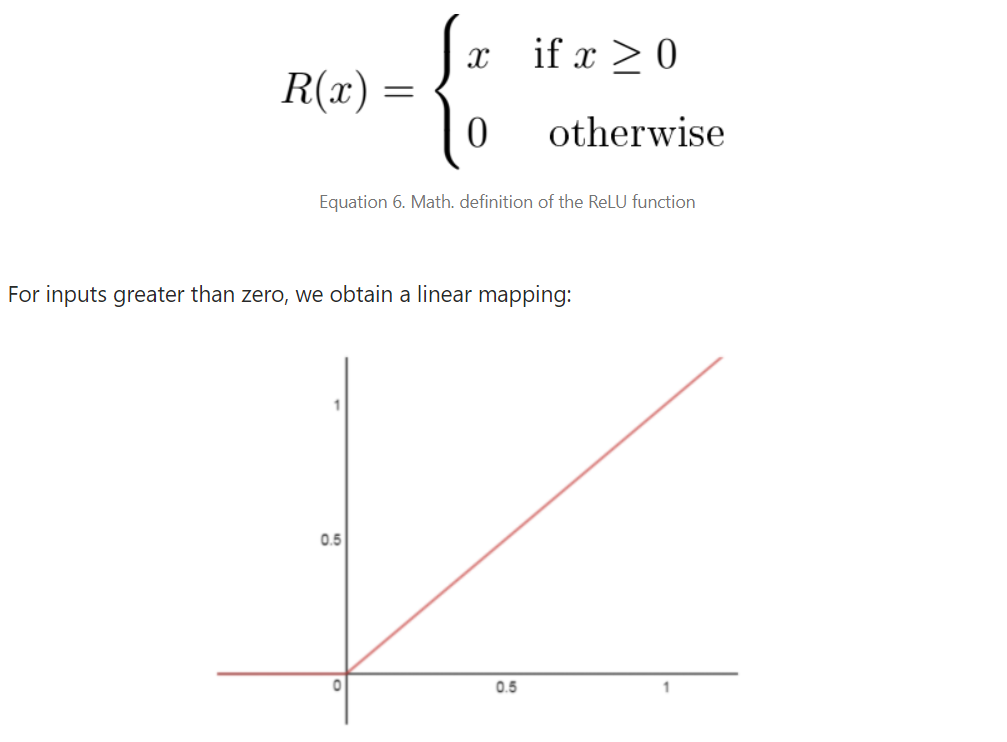
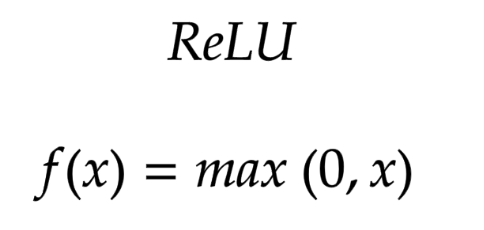
**Limitations:**



Still, it suffers from a Vanishing gradient problem. As you can see— it also faces the problem of vanishing gradients similar to the sigmoid activation function. Plus, the gradient of the tanh function is much steeper as compared to the sigmoid function. Although both sigmoid and tanh face vanishing gradient issues, tanh is zero-centered, and the gradients are not restricted to moving in a certain direction. Therefore, in practice, tanh nonlinearity is always preferred to sigmoid nonlinearity.

**ReLu (Rectified Linear Unit):**

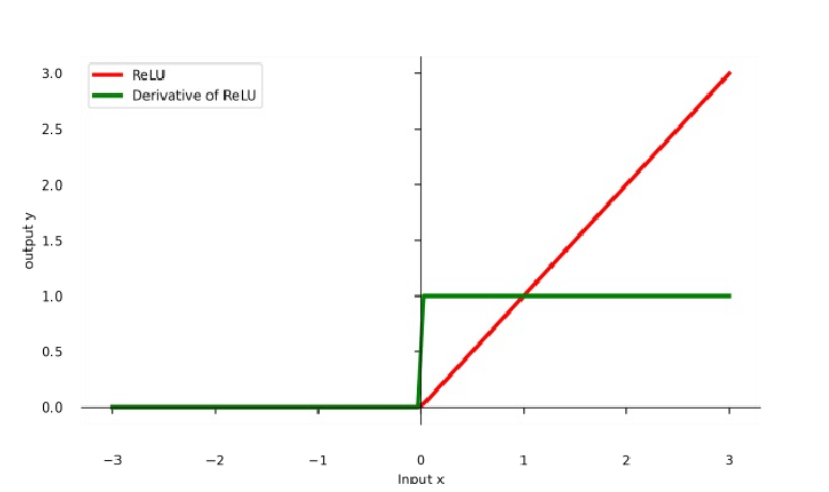
Although it gives the impression of a linear function, ReLU has a derivative function and allows for backpropagation while simultaneously making it computationally efficient. The main catch here is that the **ReLU function does not activate all the neurons at the same time.** The function and its derivative **both are** **monotonic**. The neurons will only be **deactivated** if the output of the linear transformation is **less than 0.**

**** ****

The advantages of using ReLU as an activation function are as follows:

* Since only a certain number of neurons are activated, the ReLU function is far more computationally efficient when compared to the sigmoid and tanh functions.
* ReLU accelerates the convergence of gradient descent towards the global minimum of the [loss function](https://www.v7labs.com/blog/pytorch-loss-functions) due to its linear, **non-saturating** property means a "non-linear function with no limit as x approaches infinity"
* ReLU can introduce sparsity into the network, which means that it can zero out negative inputs and reduce the number of active neurons in the network. This can improve the performance of the network and reduce the amount of computational resources needed, which is particularly important for large networks with many hidden layers.

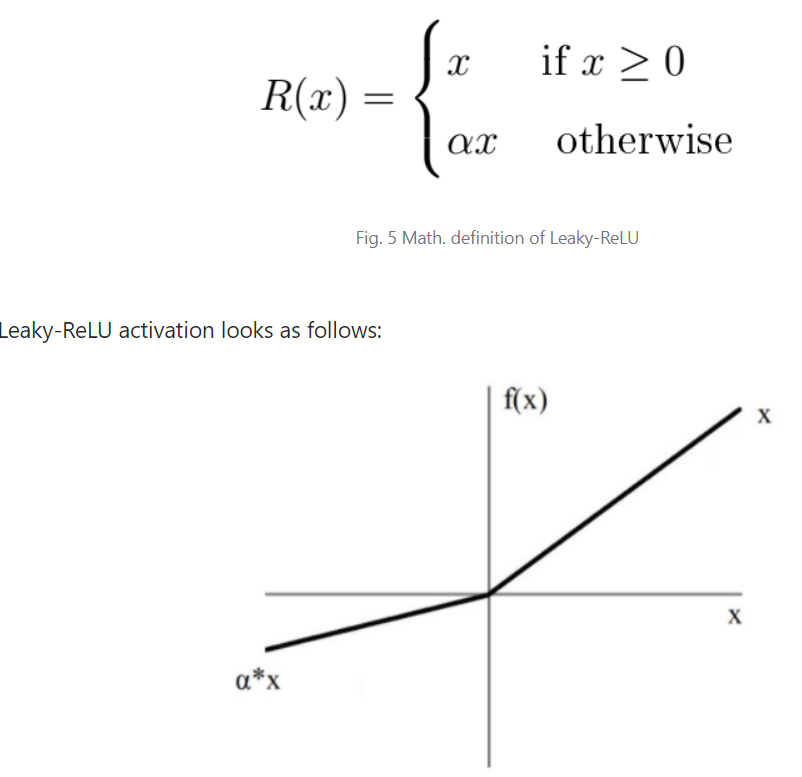
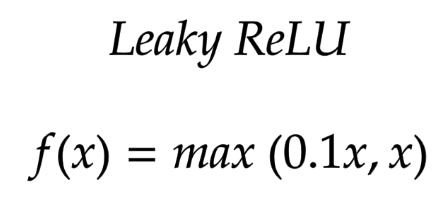
**Limitations:**



**The Dying ReLU problem:** The negative side of the graph makes the gradient value zero. Due to this reason, during the backpropagation process, the weights and biases for some neurons are not updated. This can create dead neurons that never get activated. All the negative input values become zero immediately, which decreases the model’s ability to fit or train from the data properly.

**Leaky ReLu:**

Leaky ReLU is an improved version of the ReLU function to solve the Dying ReLU problem as it has a small positive slope in the negative area.

When **a is not 0.01** then it is called **Randomized ReLU**.

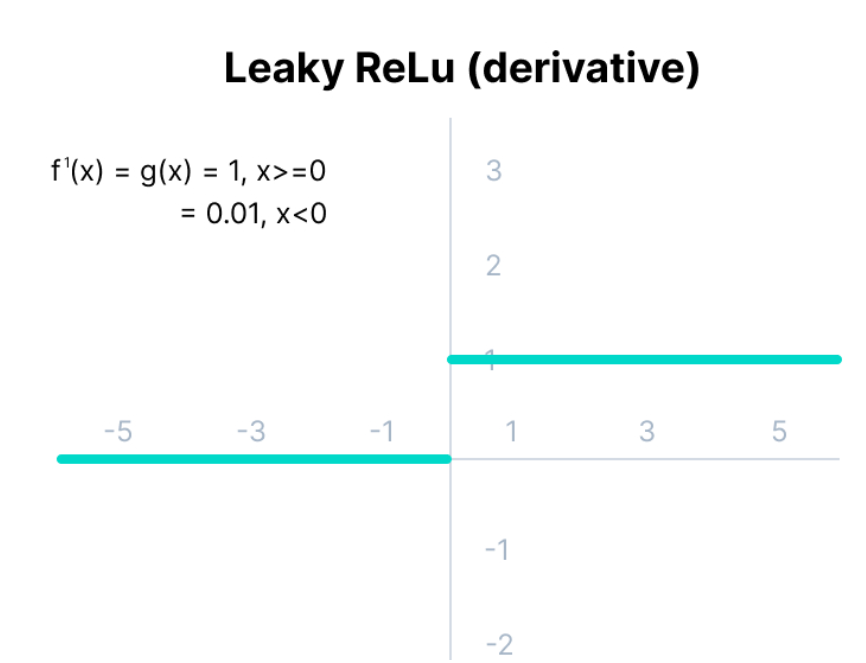
The advantages of Leaky ReLU are the same as that of ReLU, in addition to the fact that it does enable backpropagation, even for negative input values.

By making this minor modification for negative input values, the gradient of the left side of the graph comes out to be a non-zero value. Therefore, we would no longer encounter dead neurons in that region.

Therefore, the **range** of the Leaky ReLU is (-infinity to infinity).

Both Leaky and Randomized ReLU functions are monotonic in nature. Also, their derivatives also monotonic in nature.

Here is the derivative of the Leaky ReLU function.



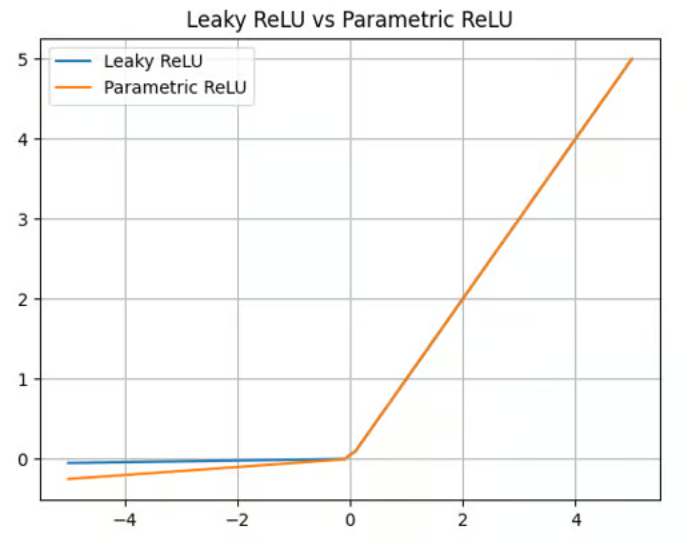
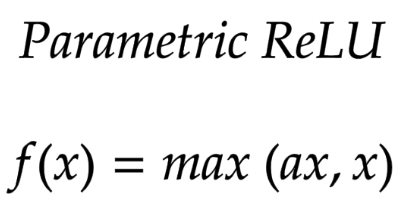
The limitations that this function faces include:

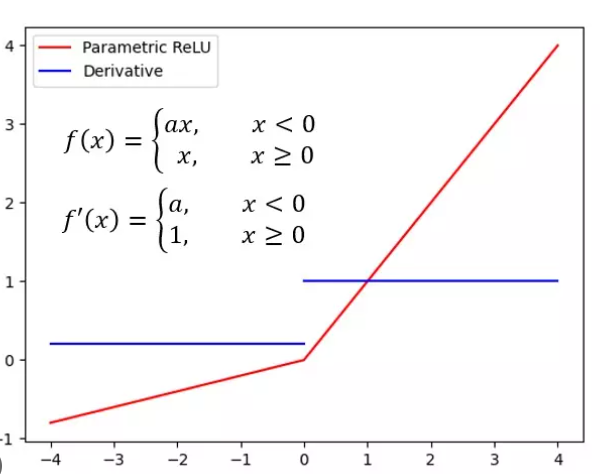
* The predictions may not be consistent for negative input values.
* The gradient for negative values is a small value that makes the learning of model parameters time-consuming.

**Parametric ReLU Function:**

Parametric ReLU is another variant of ReLU that aims to solve the problem of gradient’s becoming zero for the left half of the axis.

This function provides the slope of the negative part of the function as an argument *a*. By performing backpropagation, the most appropriate value of *a* is learnt.



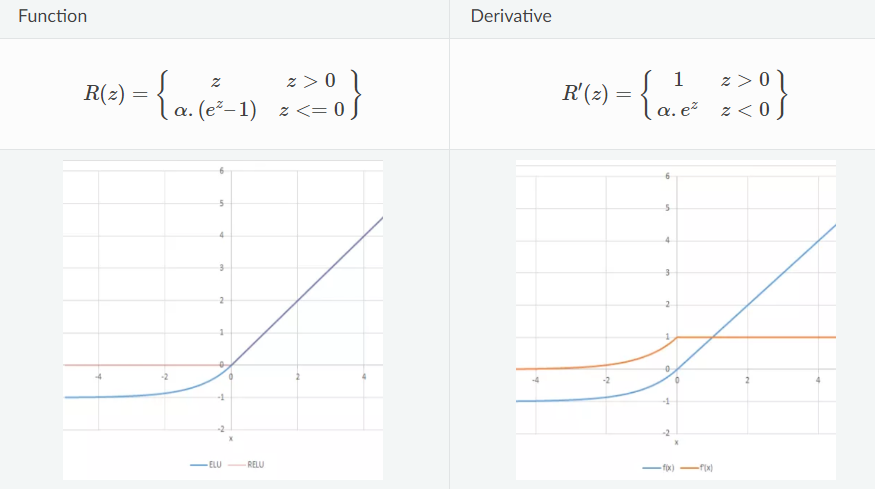
Where "a" is the slope parameter for negative values.

The parameterized ReLU function is used when the leaky ReLU function still fails at solving the problem of dead neurons, and the relevant information is not successfully passed to the next layer.

**This function’s limitation** is that it may perform **differently for different problems depending upon the value of slope parameter a**.

**Exponential Linear Units (ELUs) Function:**

ELU uses a **log** curve to define the negative values unlike the leaky ReLU and Parametric ReLU functions with a straight line.



**Advantages:**

ELU becomes smooth slowly until its output is equal to -α whereas RELU sharply smoothes. Avoids dead ReLU problem by introducing a Log curve for negative values of input. It helps the network nudge weights and biases in the right direction.

**Limitations**:

* It increases the computational time because of the exponential operation included
* No learning of the ‘a’ value takes place
* **Exploding gradient problem.**

## Softmax Activation Function:

## 

The output of the sigmoid function was in the range of 0 to 1, which can be thought of as probability.

But—  
This function faces certain problems.

Let’s suppose we have five output values of 0.8, 0.9, 0.7, 0.8, and 0.6, respectively. How can we move forward with it?

The answer is: We can’t.

The above values don’t make sense as the sum of all the classes/output probabilities should be equal to 1.

You see, the Softmax function is described as a combination of multiple sigmoids.

It calculates the relative probabilities. Similar to the sigmoid/logistic activation function, the SoftMax function returns the probability of each class.

It is most commonly used as an activation function for the last layer of the neural network in the case of multi-class classification.

Let’s go over a simple example together.

Assume that you have three classes, meaning that there would be three neurons in the output layer. Now, suppose that your output from the neurons is [1.8, 0.9, 0.68].

Applying the softmax function over these values to give a probabilistic view will result in the following outcome: [0.58, 0.23, 0.19].

The function returns 1 for the largest probability index while it returns 0 for the other two array indexes. Here, giving full weight to index 0 and no weight to index 1 and index 2. So the output would be the class corresponding to the 1st neuron(index 0) out of three.

You can see now how the softmax activation function makes things easy for multi-class classification problems.

[Why Won’t Normalization by the Sum Suffice?](https://www.pinecone.io/learn/softmax-activation/)

OTHER ACTIVATION FUNCTION

SWISH

GELU

SELU

ARGMAX

<https://www.v7labs.com/blog/neural-networks-activation-functions>

**Exploding Gradient Problems: (**[**here**](https://machinelearningmastery.com/exploding-gradients-in-neural-networks/)**)**

[**https://machinelearningmastery.com/exploding-gradients-in-neural-networks/**](https://machinelearningmastery.com/exploding-gradients-in-neural-networks/)

**How to choose the right Activation Function?**

You need to match your activation function for your output layer based on the type of prediction problem that you are solving—specifically, the type of predicted variable.

Here’s what you should keep in mind.

As a rule of thumb, you can begin with using the ReLU activation function and then move over to other activation functions if ReLU doesn’t provide optimum results.

And here are a few other guidelines to help you out.

1. The ReLU activation function should **only be used** in the hidden layers.
2. Sigmoid/Logistic and Tanh functions **should not be used** in hidden layers as they make the model more susceptible to problems during training (due to vanishing gradients).
3. Swish function is used in neural networks having a depth greater than 40 layers.

Finally, a few rules for choosing the activation function for your output layer based on the type of prediction problem that you are solving:

1. Regression - Linear Activation Function
2. Binary Classification—Sigmoid/Logistic Activation Function
3. Multiclass Classification—Softmax
4. Multilabel Classification—Sigmoid

The activation function used in hidden layers is typically chosen based on the type of neural network architecture.

1. Convolutional Neural Network (CNN): ReLU activation function.
2. [Recurrent Neural Network](https://www.v7labs.com/blog/recurrent-neural-networks-guide): Tanh and/or Sigmoid activation function.

**Optimizers**:

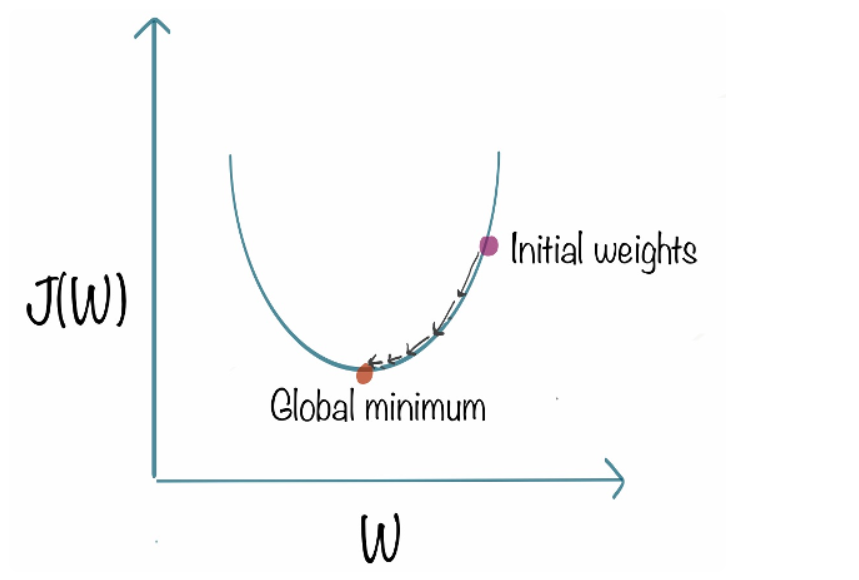
**What is optimizers?**

An optimizer is a function or an algorithm that adjusts the attributes of the neural network, such as weights and learning rates. Its primary role is to minimize the model’s error or loss function, enhancing performance.

* Epoch – The number of times the algorithm runs on the whole training dataset.
* Sample – A single row of a dataset.
* Batch – It denotes the number of samples to be taken for updating the model parameters.
* Learning rate – It is a parameter that provides the model a scale of how much model weights should be updated.
* Cost Function/Loss Function – A cost function is used to calculate the cost, which is the difference between the predicted value and the actual value.
* Weights/ Bias – The learnable parameters in a model that control the signal between two neurons.

**What is Gradient?**

In simple terms, consider you are holding a ball resting at the top of a bowl. When you lose the ball, it goes along the steepest direction and eventually settles at the bottom of the bowl. A Gradient provides the ball in the steepest direction to reach the local minimum which is the bottom of the bowl.



--alpha is the step size that represents how far to move against each gradient with each iteration.

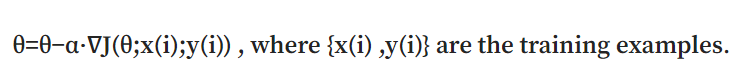
**Advantages**: Easy computation. Easy to implement. Easy to understand.

**Limitations:**

* It is expensive to calculate the gradients if the **size of the data is huge**.  Gradient descent works well for convex functions, but **it doesn’t know how far to travel along the gradient for nonconvex functions**. Sensitive to the choice of learning rate.
* A convex function has an increasing first derivative, making it appear to bend upwards. Contrarily, a concave function has a decreasing first derivative making it bend downwards.

**SGD (Stochastic gradient descent):**

It tries to update the model’s parameters more frequently. if the dataset contains 1000 rows SGD will update the model parameters 1000 times in one cycle of dataset instead of one time as in Gradient Descent.



**Advantages**:

* Frequent updates of model parameters hence, converges in less time.
* Requires less memory as no need to store values of loss functions. May get new minima.

**Disadvantages**:

* High variance in model parameters.
* May shoot even after achieving global minima.
* To get the same convergence as gradient descent needs to slowly reduce the value of the learning rate.

**Mini-Batch gradient descent:**

To tackle the challenges of large datasets. In stochastic gradient descent, instead of processing the entire dataset during each iteration, we randomly select batches of data. This implies that only a few samples from the dataset are considered at a time, allowing for more efficient and computationally feasible optimization in deep learning models.



Since we are not using the whole dataset but the batches of it for each iteration, the path taken by the algorithm is full of noise as compared to the gradient descent algorithm. Thus, SGD uses a higher number of iterations to reach the local minima. Due to an increase in the number of iterations, the overall computation time increases. But even after increasing the number of iterations, the computation cost is still less than that of the gradient descent optimizer. So, the conclusion is if the data is enormous and computational time is an essential factor, stochastic gradient descent should be preferred over batch gradient descent algorithm.

**Advantages**:

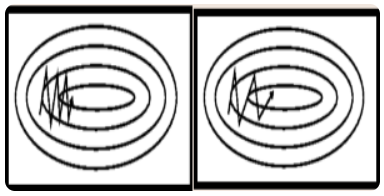
* Frequently updates the model parameters and also has less variance.
* Requires a medium amount of memory.

**Limitations:**

* Choosing an optimum value of the learning rate. If the learning rate is too small then gradient descent may take ages to converge.
* Have a constant learning rate for all the parameters. There may be some parameters that we may not want to change at the same rate.
* May get trapped at local minima.
* Increases in the number of iterations.

**Stochastic gradient descent with momentum:**

As discussed in the earlier section, you have learned that stochastic gradient descent takes a much more noisy path than the gradient descent algorithm when addressing optimizers in deep learning. Due to this, it requires a more significant number of iterations to reach the optimal minimum, and hence, computation time is very slow. To overcome the problem, we use stochastic gradient descent with a momentum algorithm.

Now, the weights are updated by **θ=θ−V(t).**

The momentum term **γ** is usually set to 0.9 or a similar value.

In the above image, the left part shows the convergence graph of the stochastic gradient descent algorithm. At the same time, the right side shows SGD with momentum. From the image, you can compare the path chosen by both algorithms and realize that using momentum helps reach convergence in less time. You might be thinking of using a large momentum and learning rate to make the process even faster. But remember that while increasing the momentum, the possibility of passing the optimal minimum also increases. This might result in poor accuracy and even more oscillations.

Momentum was invented to reduce high variance in SGD and soften the convergence. It accelerates the convergence towards the relevant direction and reduces the fluctuation to the irrelevant direction. One more hyperparameter is used in this method known as momentum symbolized by ‘**γ**’.

**Advantages**:

1. Reduces the oscillations and high variance of the parameters.
2. Converges faster than gradient descent.
3. We can avoid local minima.

**Disadvantages**:

1. One more hyper-parameter is added which needs to be selected manually and accurately.
2. while increasing the momentum, the possibility of passing the optimal minimum also increases.

## Adagrad (Adaptive Gradient Descent):

## The adaptive gradient descent algorithm is slightly different from other gradient descent algorithms. This is because it uses different learning rates for each iteration. The change in learning rate depends upon the difference in the parameters during training. The more the parameters get changed, the more minor the learning rate changes. This modification is highly beneficial because real-world datasets contain sparse as well as dense features. So it is unfair to have the same value of learning rate for all the features. The Adagrad algorithm uses the below formula to update the weights. Here the alpha(t) denotes the different learning rates at each iteration, n is a constant, and E is a small positive to avoid division by 0.

## 

## One downside of the AdaGrad optimizer is that it decreases the learning rate aggressively and monotonically. There might be a point when the learning rate becomes extremely small. This is because the squared gradients in the denominator keep accumulating, and thus the denominator part keeps on increasing. Due to small learning rates, the model eventually becomes unable to acquire more knowledge, and hence the accuracy of the model is compromised.

**Advantages**:

1. Learning rate changes for each training parameter.
2. Don’t need to manually tune the learning rate.
3. Able to train on sparse data.

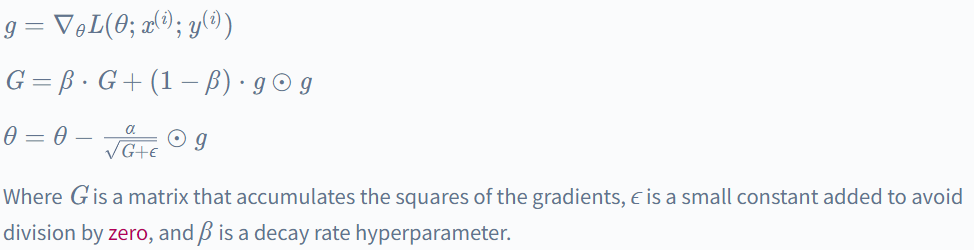
**Disadvantages**:

1. Computationally expensive as a need to calculate the second order derivative.
2. The learning rate is always decreasing resulting in slow training.

## RMS Prop (Root Mean Square):

RMS prop is one of the popular optimizers among deep learning enthusiasts. This is maybe because it hasn’t been published but is still very well-known in the community. RMS prop is ideally an extension of the work RPPROP. It resolves the problem of varying gradients. The problem with the gradients is that some of them were small while others may be huge. So, defining a single learning rate might not be the best idea. RPPROP uses the gradient sign, adapting the step size individually for each weight. In this algorithm, the two gradients are first compared for signs. If they have the same sign, we’re going in the right direction, increasing the step size by a small fraction. If they have opposite signs, we must decrease the step size. Then we limit the step size and can now go for the weight update.

**The problem with RPPROP is that it doesn’t work well with large datasets and when we want to perform mini-batch updates. So, achieving the robustness of RPPROP and the efficiency of mini-batches simultaneously was the main motivation behind the rise of RMS prop. RMS prop is an advancement in AdaGrad optimizer as it reduces the monotonically decreasing learning rate. It uses an exponentially decaying average of the squares of the gradients rather than the sum. This helps to reduce the monotonic learning rate decay of Adagrad and improve convergence.**

****

## In simpler terms, if there exists a parameter due to which the cost function oscillates a lot, we want to penalize the update of this parameter. Suppose you built a model to classify a variety of fishes. The model relies on the factor ‘color’ mainly to differentiate between the fishes. Due to this, it makes a lot of errors. What RMS Prop does is, penalize the parameter ‘color’ so that it can rely on other features too. This prevents the algorithm from adapting too quickly to changes in the parameter ‘color’ compared to other parameters. This algorithm has several benefits as compared to earlier versions of gradient descent algorithms. The algorithm converges quickly and requires lesser tuning than gradient descent algorithms and their variants.

## Pros:

## It can work well with sparse data.

## Automatically adjusts learning rates based on parameter updates.

## Can converge faster than Adagrad.

## Cons:

## It can still converge too slowly for some problems.

## Requires tuning of the decay rate hyperparameter.

## AdaDelta:

## AdaDelta is an optimization algorithm similar to RMSProp but does not require a hyperparameter learning rate. Instead, it uses an exponentially decaying average of the gradients and the squares of the gradients to determine the updated scale.

## AdaDelta can be seen as a more robust version of the AdaGrad optimizer. It is based upon adaptive learning and is designed to deal with significant drawbacks of AdaGrad and RMS prop optimizer. The main problem with the above two optimizers is that the initial learning rate must be defined manually. One other problem is the decaying learning rate which becomes infinitesimally small at some point. Due to this, a certain number of iterations later, the model can no longer learn new knowledge.

## 

Pros:

* Can work well with sparse data.
* Automatically adjusts learning rates based on parameter updates.

Cons:

* Can converge too slowly for some problems.
* Can stop learning altogether if the learning rates become too small.

**Adam (Adaptive Moment Estimation):**

its ability to adaptively adjust the learning rate for each network weight individually. Unlike SGD, which maintains a single learning rate throughout training, Adam optimizer dynamically computes individual learning rates based on the past gradients and their second moments.

The creators of Adam optimizer incorporated the beneficial features of other optimization algorithms such as AdaGrad and RMSProp. Similar to RMSProp, Adam optimizer considers the second moment of the gradients, but unlike RMSProp, it calculates the uncentered variance of the gradients (without subtracting the mean).

By incorporating both the first moment (mean) and second moment (uncentered variance) of the gradients, Adam optimizer achieves an adaptive learning rate that can efficiently navigate the optimization landscape during training. This adaptivity helps in faster convergence and improved performance of the neural network.

In summary, Adam optimizer is an optimization algorithm that extends SGD by dynamically adjusting learning rates based on individual weights. It combines the features of AdaGrad and RMSProp to provide efficient and adaptive updates to the network weights during deep learning training.

why shouldn’t you use Adam in every application? And what was the need to learn about other algorithms in depth? This is because even Adam has some downsides. It tends **to focus on faster computation time**, whereas algorithms like **stochastic gradient descent focus on data points.** That’s why algorithms like SGD generalize the data in a better manner at the cost of low computation speed. So, the optimization algorithms can be picked accordingly depending on the requirements and the type of data.

## 

Pros:

* Can converge faster than other optimization algorithms.
* Can work well with noisy data.

Cons:

* It may require more tuning of hyperparameters than other algorithms.
* May perform better on some types of problems.

## 

The optimizer uses an optimization algorithm to search for the parameters that minimize the loss function. The optimization algorithm uses the gradients of the loss function to the model parameters to determine the direction in which we should adjust the parameters.

The **gradients** are computed using backpropagation, which involves applying the chain rule to compute the gradients of the loss function to each of the model parameters.

The **optimization algorithm** then adjusts the model parameters to minimize the loss function. This process is repeated until the loss function reaches a minimum or the optimizer reaches the maximum number of allowed iterations.

In general, the choice of which optimization algorithm to use will depend on the specific characteristics of the problem, such as the dataset's size and the model's complexity.

It is important to consider each algorithm's pros and cons carefully and tune any relevant hyperparameters to achieve the best possible performance.

**ANN Artificial Neural Network:**