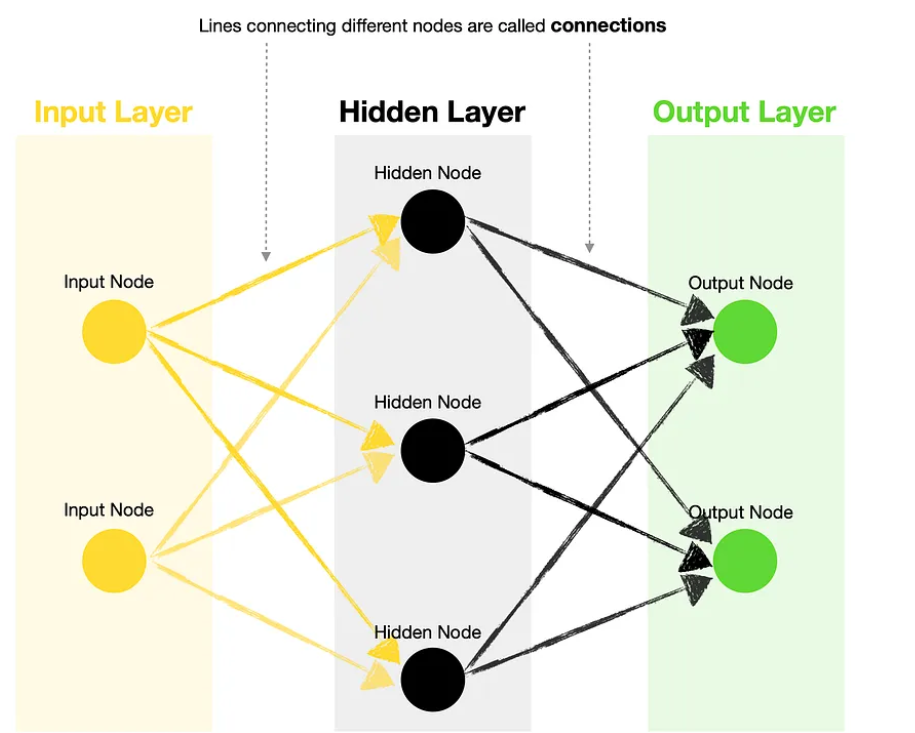
**Neural Networks**

It is essential to remember that Neural Networks are most frequently employed to solve classification and regression problems using labeled training data. Hence, an alternative approach could be to put them under the Supervised branch of Machine Learning.

****

* **Input Layer**— contains one or more input nodes. For example, suppose you want to predict whether it will rain tomorrow and base your decision on two variables, humidity and wind speed. In that case, your first input would be the value for humidity, and the second input would be the value for wind speed.
* **Hidden Layer**— this layer houses hidden nodes, each containing an **activation function** (more on these later). Note that a Neural Network with multiple hidden layers is known as Deep Neural Network.
* **Output Layer**— contains one or more output nodes. Following the same weather prediction example above, you could choose to have only one output node generating a rain probability (where >0.5 means rain tomorrow, and ≤0.5 no rain tomorrow). Alternatively, you could have two output nodes, one for rain and another for no rain. Note, you can use a different **activation function**for output nodes vs. hidden nodes.
* **Connections**— lines joining different nodes are known as connections. These contain **kernels (weights)** and **biases**, the parameters that get optimized during the training of a neural network.

## **Parameters and activation functions**

# Let’s take a closer look at kernels (weights) and biases to understand what they do. For simplicity, we create a basic neural network with one input node, two hidden nodes, and one output node (1–2–1):

# 

* **Kernels (weights)**—used to scale input and hidden node values. Each connection typically holds a different weight.
* **Biases**— used to adjust scaled values before passing them through an activation function.
* **Activation functions**— think of activation functions as standard curves (building blocks) used by the Neural Network to create a custom curve to fit the training data. Passing different input values through the network selects different sections of the standard curve, which are then assembled into a final custom-fit curve.

# There are many activation functions to choose from, with [Softplus](https://www.tensorflow.org/api_docs/python/tf/keras/activations/softplus" \t "_blank), [ReLU](https://www.tensorflow.org/api_docs/python/tf/keras/activations/relu" \t "_blank), and [Sigmoid](https://www.tensorflow.org/api_docs/python/tf/keras/activations/sigmoid) being the most commonly used. Here are the shapes and equations of six frequently used activation functions in Neural Networks:

# 

As we are now familiar with kernels (weights), biases, and activation functions, let’s use the same Neural Network to calculate the probability of rain tomorrow based on today’s humidity.

The below illustration shows you a step-by-step process of how FF Neural Network takes an input value and produces the answer (output value).

# 

# As you can see, the above Neural Network tells us that a 50% humidity today implies a 33% probability of rain tomorrow.

## **Loss functions, optimizers, and training**

Training Neural Networks involves a complicated process known as **backpropagation**.

## **Backward Propagation**

This algorithm is called backpropagation because it tries to reduce errors from output to input. It looks for the minimum value of the error function in the weight field using a technique called gradient descent.

Backpropagation is the core of neural network training. It is a method of adjusting the weights of a neural network based on the loss value obtained in the previous epoch. Correctly adjusting the weights allows us to reduce the error rate and increase its generalization, making the model reliable.

Let me briefly introduce you to loss functions and optimizers and summarize what happens when we “train” a Neural Network.

* **Loss**— represents the “size” of error between the **true** values/labels and the **predicted** values/labels. The goal of training a Neural Network is to minimize this loss. The smaller the loss, the closer the match between the true and the predicted data. There are many **loss functions** to choose from, with [BinaryCrossentropy](https://www.tensorflow.org/api_docs/python/tf/keras/losses/BinaryCrossentropy" \t "_blank), [CategoricalCrossentropy](https://www.tensorflow.org/api_docs/python/tf/keras/losses/CategoricalCrossentropy" \t "_blank), and [MeanSquaredError](https://www.tensorflow.org/api_docs/python/tf/keras/losses/MeanSquaredError" \t "_blank) being the most common.

# Types of Loss Functions

In supervised learning, there are two main types of loss functions — these correlate to the 2 major types of neural networks: regression and classification loss functions

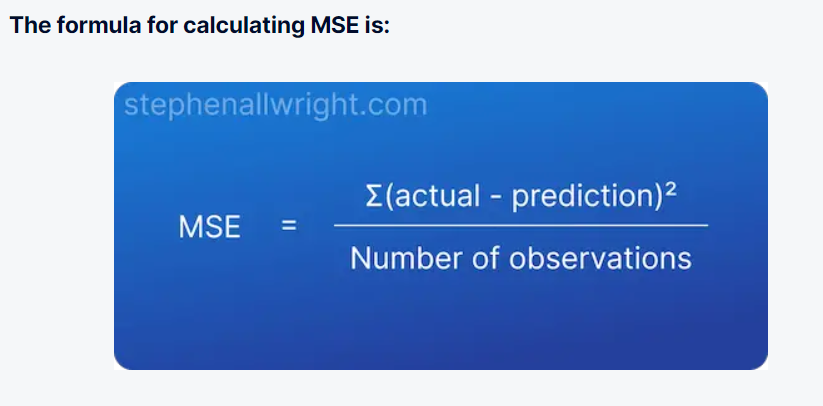
1. Regression Loss Functions — used in regression neural networks; given an input value, the model predicts a corresponding output value (rather than pre-selected labels); Ex. Mean Squared Error, Mean Absolute Error.

# MSE vs. RMSE: Which Metric Should You Use?

# Regression models are used to quantify the relationship between one or more predictor variables and a [response variable](https://www.statology.org/explanatory-response-variables/).

# Whenever we fit a regression model, we want to understand how well the model is able to use the values of the predictor variables to predict the value of the response variable. (target variable).

# Two metrics we often use to quantify how well a model fits a dataset are the mean squared error (MSE) and the root mean squared error (RMSE), which are calculated as follows:



Mean Squared Error (MSE) is the average squared error between actual and predicted values.

**The main draw for using MSE is that it squares the error, which results in large errors being punished or clearly highlighted**.

You can see that the error is not returned on the same scale as the target, therefore making it difficult to interpret its meaning.

### Advantages of using MSE

1. Easy to calculate in Python
2. Simple to understand calculation for end users
3. Designed to punish large errors

### Disadvantages of using MSE

1. Error value not given in terms of the target
2. Difficult to interpret
3. Not comparable across use cases

The closer your MSE value is to 0, the more accurate your model is. However, there is no 'good' value for MSE. It is an absolute value which is unique to each dataset and can only be used to say whether the model has become more or less accurate than a previous run.

### Can MSE be used to compare models?

**MSE cannot be used to compare different models from different datasets as it’s an absolute value that is only relevant to that given dataset**. If you need to compare models across different datasets then it would be best to use percentage metrics such as [MAPE](https://stephenallwright.com/good-mape-score/).

### What is a normal MSE?

**There is no MSE value which is considered ‘normal’ as it’s an absolute error score which is unique to that model and dataset**. For example, a house price prediction model will have much larger MSE values than a model which predicts height, as they are predicting for very different scales.

### Is lower MSE better?

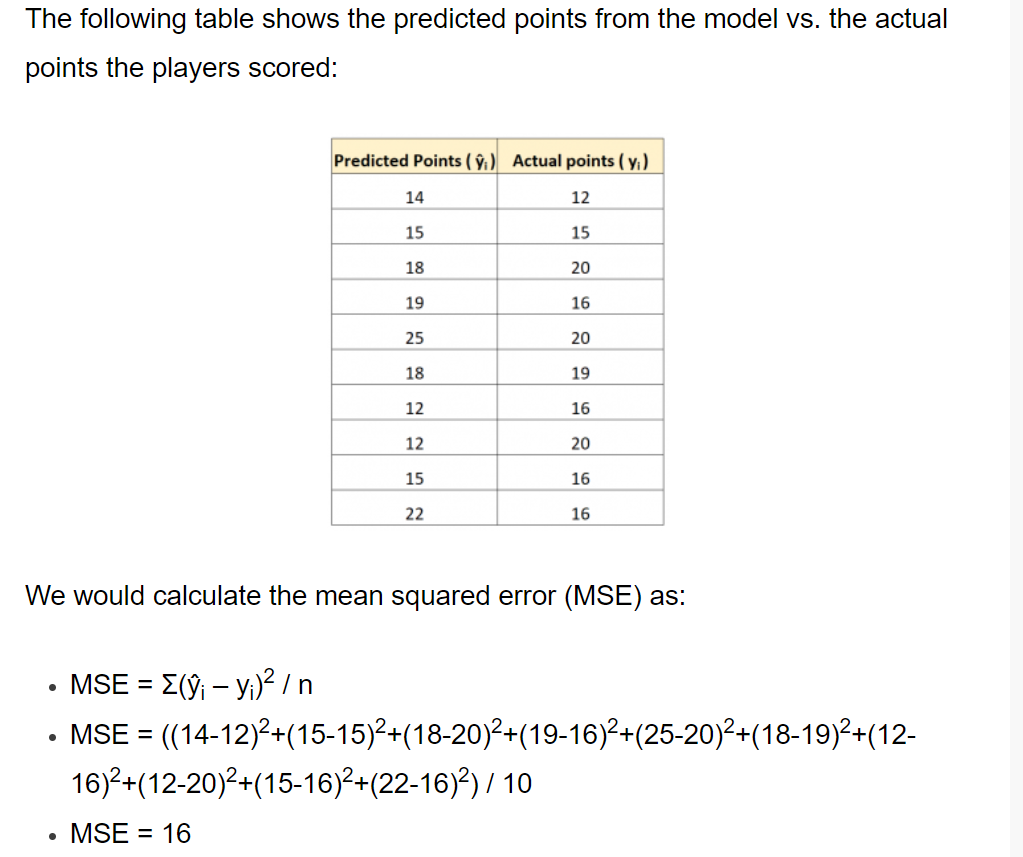
**The lower the MSE value there more accurate the model is**. Lower is of course a relative term, so it’s important to know that MSE values can only be compared to other MSE values calculated for that same dataset, as MSE is an absolute metric unique to each use case.

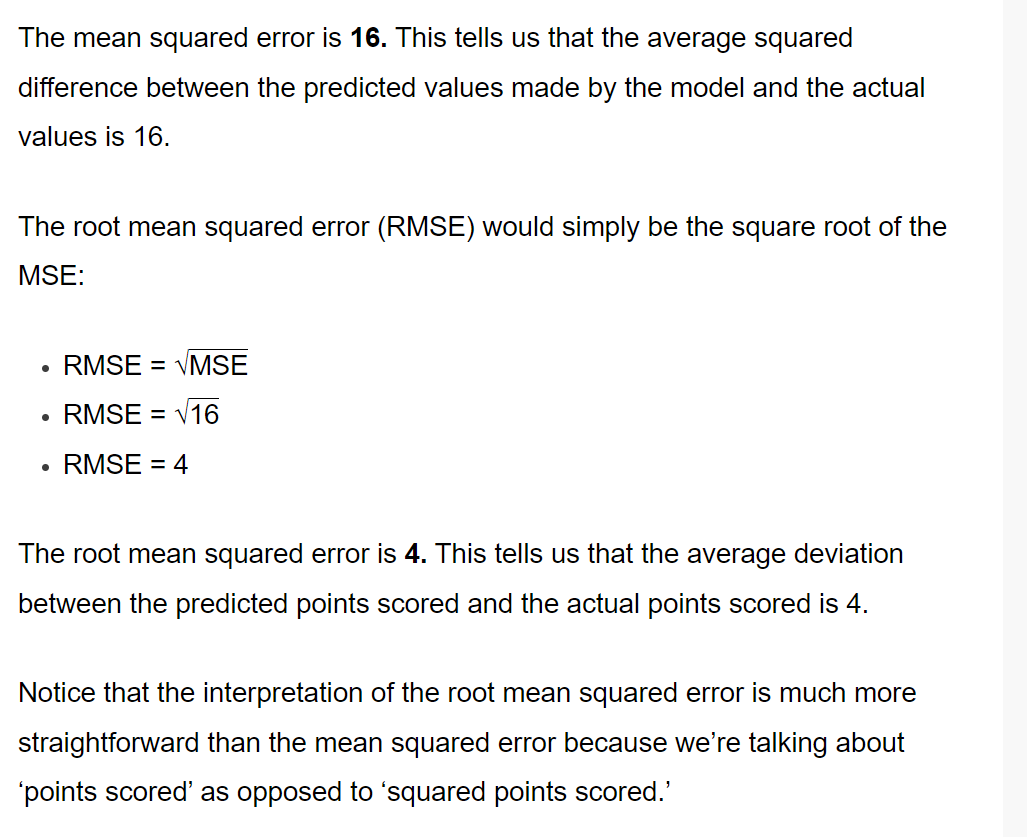
### Can MSE be greater than 1?

**MSE is a metric which ranges from 0 to infinity, and can therefore be greater than 1.**

### RMSE vs. MSE: Which Metric Should You Use?

**When assessing how well a model fits a dataset, we use the RMSE more often because it is measured in the same units as the response variable, target variable.**

****

****

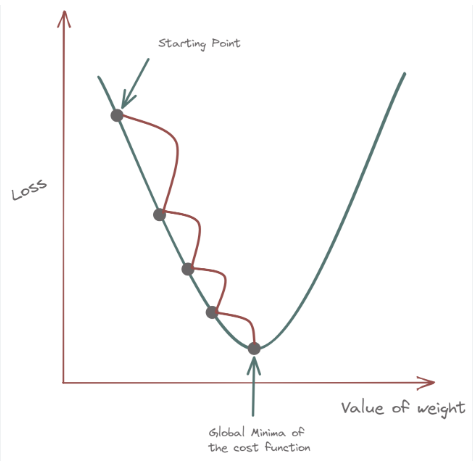
Conversely, the MSE is measured in squared units of the response variable.

1. Classification Loss Functions — used in classification neural networks; given an input, the neural network produces a vector of probabilities of the input belonging to various pre-set categories — can then select the category with the highest probability of belonging; Ex. Binary Cross-Entropy, Categorical Cross-Entropy

# 

* **Optimizers**— are the **algorithms used in backpropagation**. The goal of an optimizer is to find the optimum set of kernels (weights) and biases to minimize the loss. Optimizers typically use a gradient descent approach, which allows them to iteratively find the “best” possible configuration of weights and biases. The most commonly used ones are [SGD](https://www.tensorflow.org/api_docs/python/tf/keras/optimizers/SGD), [ADAM](https://www.tensorflow.org/api_docs/python/tf/keras/optimizers/Adam), and [RMSProp](https://www.tensorflow.org/api_docs/python/tf/keras/optimizers/RMSprop).

**Gradient descent is an iterative optimization algorithm used in machine learning to minimize a loss function. The loss function describes how well the model will perform given the current set of parameters (weights and biases), and gradient descent is used to find the best set of parameters.**

****

* In general, the wide selection of activation functions combined with the ability to add as many hidden nodes as we wish (provided we have sufficient computational power) means that Neural Networks can create a curve of any shape to fit the data.
* However, having this extreme flexibility may sometimes lead to overfitting the data. Hence, we must always ensure that we validate the model on the test/validation set before using it to make predictions.

## **Summarizing what we have learned**

Feed Forward Neural Networks take one or multiple input values and apply transformations using kernels (weights) and biases before passing results through activation functions. In the end, we get an output (prediction), which is a result of this complex set of transformations optimized through training.

We train Neural Networks by fitting a custom curve through the training data, guided by loss minimization and achieved through parameter (kernels and biases) optimization,Gradient descent algorithm.

**Each hidden layer has its own biases and each line has its own weight. As you can see all inputs and hidden layers are fully connected:**

# 

# 

# Every neuron is a unique bias, and every connection was its own weight.

# 

# binary\_crossentropy: Used as a loss function for binary classification model. The binary\_crossentropy function computes the cross-entropy loss between true labels and predicted labels. categorical\_crossentropy: Used as a loss function for multi-class classification model where there are two or more output labels.

# For example, If we are building a model which predicts all the clothing articles a person is wearing, we can use a multi-label classification model since there can be more than one possible option at once.

# In multi-label classification, we have several labels that are the outputs for a given prediction. When making predictions, a given input may belong to more than one label. For example, when predicting a given movie category, it may belong to horror, romance, adventure, action, or all simultaneously.

# Multiclass Classification is where each data sample is assigned one and only one label from more than two classes. Multi-label Classification is a type of supervised machine learning algorithm that can be used to assign zero or more labels to each data sample.

# Loss functions you can use for regression:

# <https://medium.com/@mlblogging.k/14-loss-functions-you-can-use-for-regression-b24db8dff987>

# Deep Feed Forward Neural Networks and the Advantage of ReLU Activation Function

# The difference between Feed Forward (FF) and Deep Feed Forward (DFF) Neural Networks

## Structure

The structure of a DFF is very similar to that of an FF. The major difference between them is the number of hidden layers.

# 

## What is the point of depth?

Typically you will find that deep NNs perform better than shallow ones. However, it is not always necessary to use a deep network. The choice will largely depend on the task you have at hand.

If you are working with many inputs, such as image data, then using a Deep Feed Forward (DFF) or a Convolutional Neural Network (CNN) would likely yield better results than a simple Feed Forward network.

However, suppose your task is to do some basic classification with a limited number of inputs. In that case, you may be better off using a simple FF network or even a tree-based algorithm such as [XGBoost](https://towardsdatascience.com/xgboost-extreme-gradient-boosting-how-to-improve-on-regular-gradient-boosting-5c6acf66c70a" \t "_blank), [Random Forest](https://towardsdatascience.com/random-forest-models-why-are-they-better-than-single-decision-trees-70494c29ccd1), or a single [Decision tree](https://towardsdatascience.com/cart-classification-and-regression-trees-for-clean-but-powerful-models-cc89e60b7a85).

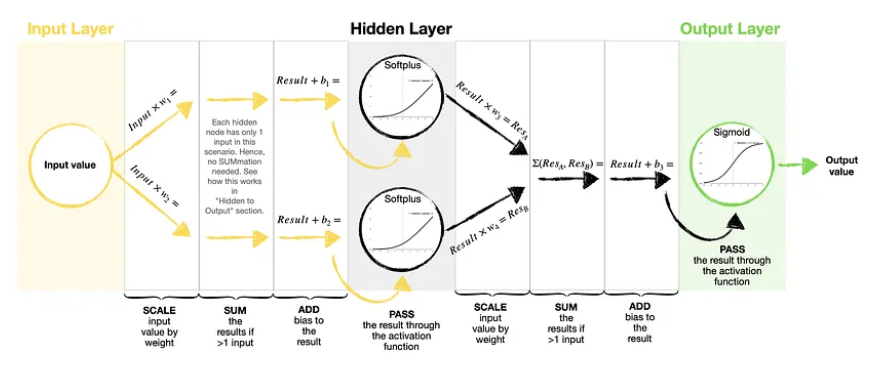
So, going back to the point of depth, the simple answer is that deeper networks tend to deliver better performance on more complex tasks. There are multiple hypotheses on why deep networks perform better, ranging from efficiency to improved ability to learn more abstract representations.

# What is the purpose of activation functions, and why did ReLU become a go-to for Deep Neural Networks?

## Challenges presented by deep networks

A Neural Network architecture contains **activation functions** inside the hidden nodes and output nodes. In short, the activation function takes the input value entering the node, performs a transformation, and then passes the result onwards to the next set of neurons.

Here is a simple illustration of a Feed Forward Neural Network that shows what activation functions look like (in this case, softplus and sigmoid) and how they transform data within the Neural Network:

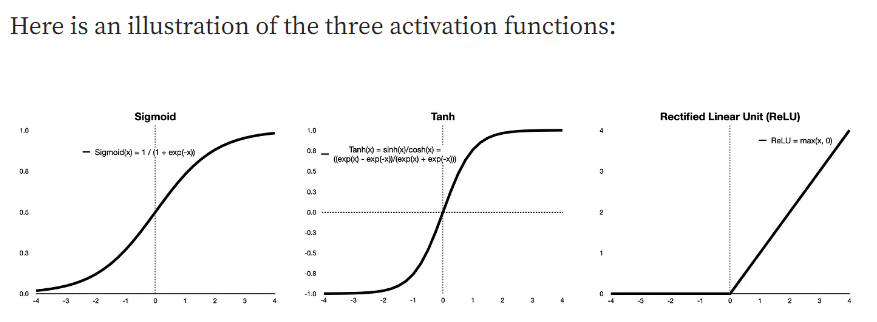


Traditionally, the two most widely used nonlinear activation functions were the [sigmoid](https://www.tensorflow.org/api_docs/python/tf/keras/activations/sigmoid) and the [hyperbolic tangent (tanh)](https://www.tensorflow.org/api_docs/python/tf/keras/activations/tanh). However, using these activation functions with Deep Neural Networks presented a problem of [**vanishing gradient**](https://en.wikipedia.org/wiki/Vanishing_gradient_problem).

The error is backpropagated through the network during the training process and is used to update the weights. Unfortunately, sigmoid and tanh activation functions tend to **saturate**.

It means that large negative and large positive values are transformed to 0 and 1 by sigmoid and -1 and 1 by tanh. The saturation often happens regardless of whether the inputs provided to the node contain useful information or not.

As the functions saturate, the derivate becomes close to zero. Hence, there is **essentially no gradient left** to propagate back through the network, making it challenging for the learning algorithm to continue adapting the weights.



## **Rectified Linear Unit (ReLU) activation function**

ReLU has been introduced as a solution to a vanishing gradient problem and quickly became a default option for most Deep Feed Forward (DFF) and Convolutional Neural Networks (CNN).

It has a very simple function that sets all negative values to 0 while returning the same value for all positive inputs.

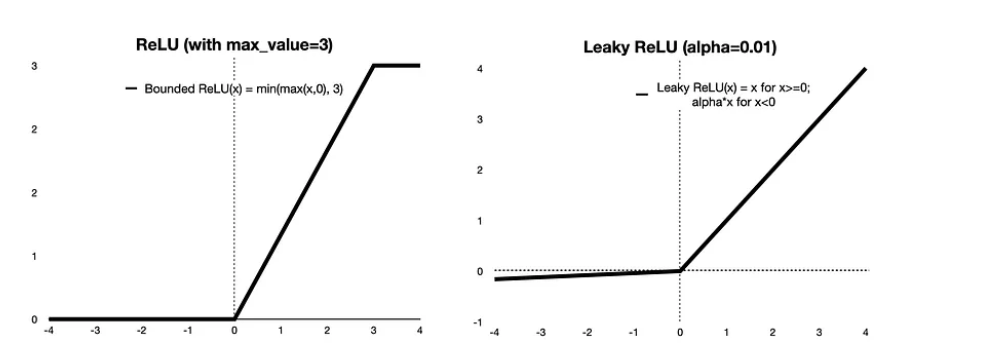
**ReLU(x) = max(x,0)**

ReLU is, of course, a nonlinear function. However, it is very close to being linear, enabling it to preserve many of the properties that make linear models easy to optimize with gradient-based methods. At the same time, it tends to generalize well too.

[**Tensorflow implementation of ReLU**](https://www.tensorflow.org/api_docs/python/tf/keras/activations/relu)allows you to set a few parameters to tune ReLU to your liking. E.g.:

* You can set the saturation threshold by specifying max\_value or;
* You can turn ReLU into a Leaky ReLU by setting an alpha parameter, where alpha governs the slope for values lower than 0 (or another chosen threshold).

Here is an illustration of ReLU with max\_value set to 3 and Leaky ReLU with alpha at 0.01.



You may choose to use ReLU with a max cap if you run into a problem of **exploding gradient** while training your model. As you may have guessed, the exploding gradient is the opposite problem to the vanishing gradient, and it results from having large weights.

Although, typically, you can avoid the issue of exploding gradient if you use He weight initialization with a standard ReLU. He initialization (see: [HeNormal](https://www.tensorflow.org/api_docs/python/tf/keras/initializers/HeNormal" \t "_blank), [HeUniform](https://www.tensorflow.org/api_docs/python/tf/keras/initializers/HeUniform" \t "_blank)) ensures that the initial weights are small enough to minimize the risk of exploding gradient.

Meanwhile, a Leaky ReLU can be beneficial when you want to avoid having “dead” neurons, which result from negative inputs being set to 0 by a standard ReLU.

## WHAT IS A RELU ACTIVATION FUNCTION?

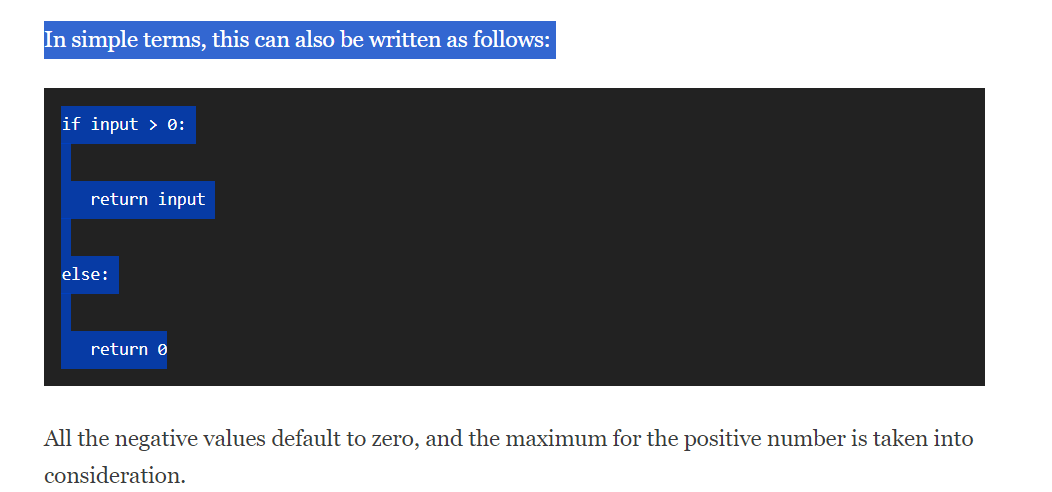
A rectified linear unit (ReLU) is an activation function that introduces the property of non-linearity to a deep learning model and solves the vanishing gradients issue. "It interprets the positive part of its argument. It is one of the most popular activation functions in deep learning.

Sigmoid and tanh were monotonous, differentiable and previously more popular activation functions. However, these functions suffer saturation over time, and this leads to problems occurring with [vanishing gradients](https://builtin.com/data-science/recurrent-neural-networks-and-lstm). An alternative and the most popular activation function to overcome this issue is theRectified Linear Unit (ReLU).

**In this article, we’ll only look at the rectified linear unit (ReLU) because it’s still the most used activation function by default for performing a majority of deep learning tasks. Its variants are typically used for specific purposes in which they might have a slight edge over the ReLU.**

 Researchers tended to use differentiable functions like sigmoid and tanh. However, it’s now determined that ReLU is the best activation function for [deep learning](https://builtin.com/learn/deep-learning).

In simple terms, this can also be written as follows:



# An Activation Function decides whether a neuron should be activated or not. This means that it will decide whether the neuron's input to the network is important or not in the process of prediction using simpler mathematical operations.

# The main job of an activation function is to introduce non-linearity in a neural network. One way to look at this is that without a non-linear activation function, a neural network will behave just like a single-layer perceptron; it does not matter how many layers it has.

**The ReLU function is another non-linear activation function that has gained popularity in the deep learning domain. ReLU stands for Rectified Linear Unit. The main advantage of using the ReLU function over other activation functions is that it does not activate all the neurons at the same time.  
We’ll derive the network to find its optimum. This is called Gradient Descent.**

Just like for a function, the derivative enables us to know for which values of x the network optimizes its performance.

Only here, we don’t have only one x but a multitude !

Indeed, in our Neural Network, we have several functions.

In fact, each layer of a Deep Learning model is a function to optimize.

So when we want to optimize our model, we have to optimize each of its layers.

**This is why we call it Deep Learning. The calculation of the derivatives is tremendous which makes it very long or… very deep !**

This is the strength of Deep Learning. This learning complexity makes it able to solve complex tasks.

**The purpose of non-linearity is to mark a gap between each layer.**

Let me explain.

The functions found in the layers of a Neural Network are linear.

The problem is that a succession of linear functions can be easily summarized to a single linear function (the mathematical demonstration can be found on the internet).

In other words, if we had a succession of 100 linear layers, we could simplify them mathematically to only one linear layer.

Immediately, we lose in complexity !

In fact, with only linearity, we lose the main asset of Deep Learning : its depth, the number of layers that follow one another.

So to keep this complexity, we will mark a step between each layer.

This step is the activation function and it is this step that brings the non-linearity.

**This non-linearity creates a boundary between each of the layers so that they cannot be simplified.**

In other words, non-linearity helps us to maintain the complexity of a neural network.

Thus, a neural network is composed of layers and activation functions that allow, thanks to their non-linearity, to create artificial boundaries between the layers.

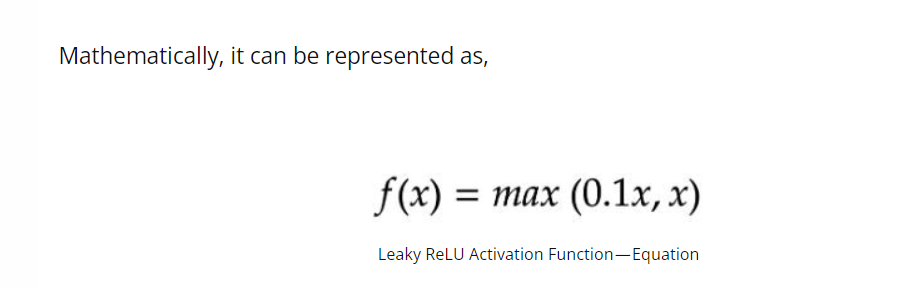
Finally, non-linearity is the icing on the cake of the Neural Network. It is what allows it to function.

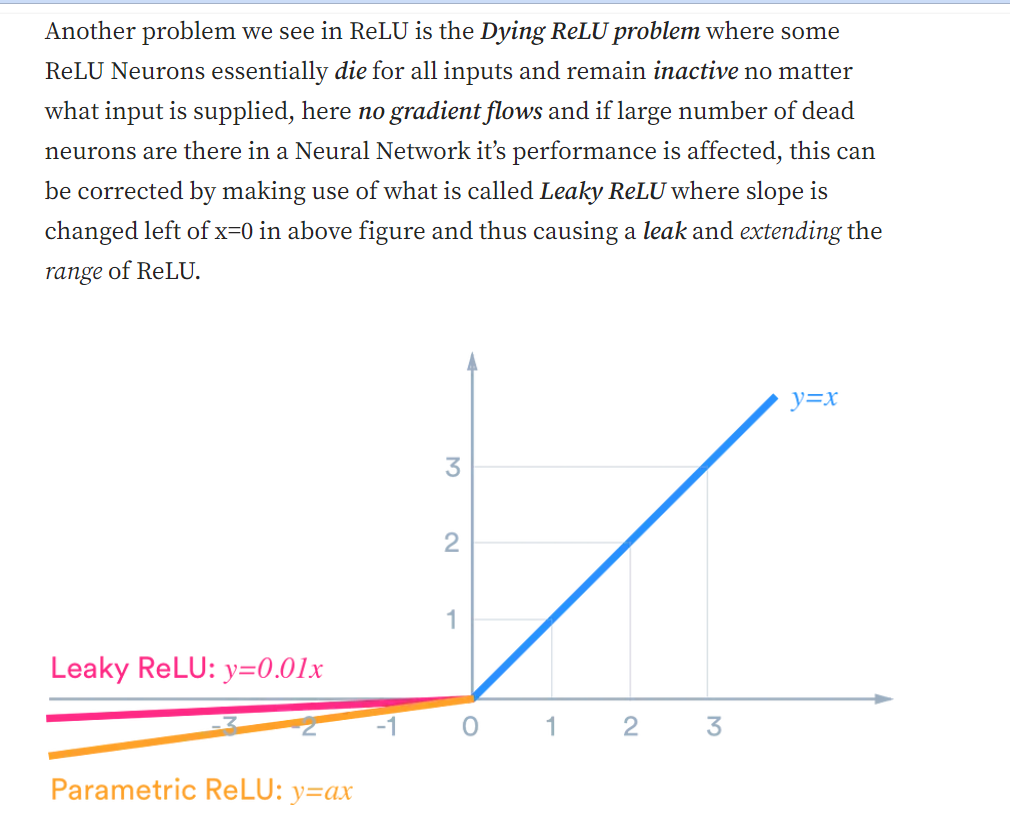
**Without non-linearity, there wouldn’t be any complexity. We would have a single layer network. We would then no longer be talking about Deep Learning but only about Machine Learning.**

Therefore in Deep Learning, we need to mark the separation between each layer to reveal their purpose !

# Activation Functions are mathematical equations that are attached at the neurons in the network. It determines the output of a neural network. Without an activation function the network will behave like a Simple Linear Regression model.

* In other words, For activations in the region (x<0) of ReLu, the gradient will be 0 because of which the weights will not get adjusted during descent. That means, those neurons, which go into that state will stop responding to variations in input (simply because the gradient is 0, nothing changes.) This is called the **dying ReLu problem**.

****

****

* One of its limitations is that it should only be used within hidden layers of an artificial neural network model.
* The advantages of Leaky ReLU are the same as that of ReLU, in addition to the fact that it does enable back propagation, even for negative input values.

