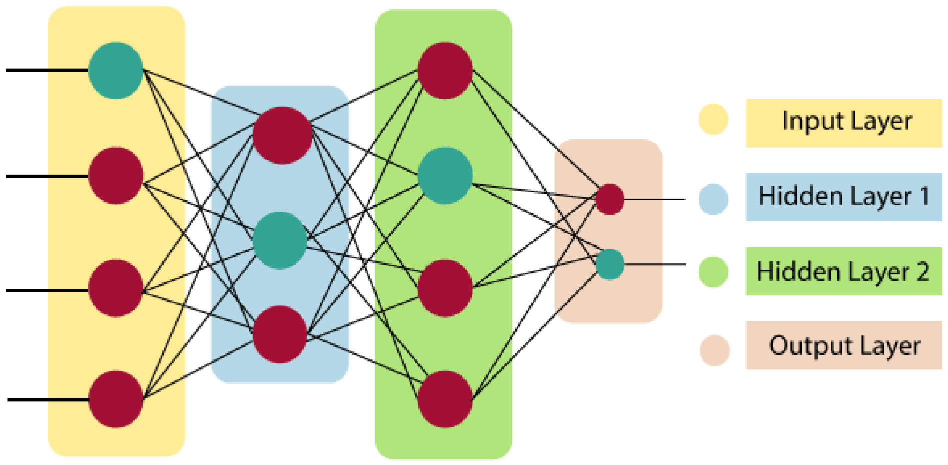
What is Artificial Neural Network?

The term **"Artificial neural network"** refers to a biologically inspired sub-field of artificial intelligence modeled after the brain. An Artificial neural network is usually a computational network based on biological neural networks that construct the structure of the human brain. Similar to a human brain has neurons interconnected to each other, artificial neural networks also have neurons that are linked to each other in various layers of the networks. These neurons are known as nodes. Artificial neural network tutorial covers all the aspects related to the artificial neural network. In this tutorial, we will discuss ANNs, Adaptive resonance theory, Kohonen self-organizing map, Building blocks, unsupervised learning, Genetic algorithm, etc.



Input Layer:

As the name suggests, it accepts inputs in several different formats provided by the programmer.

Hidden Layer:

The hidden layer presents in-between input and output layers. It performs all the calculations to find hidden features and patterns.

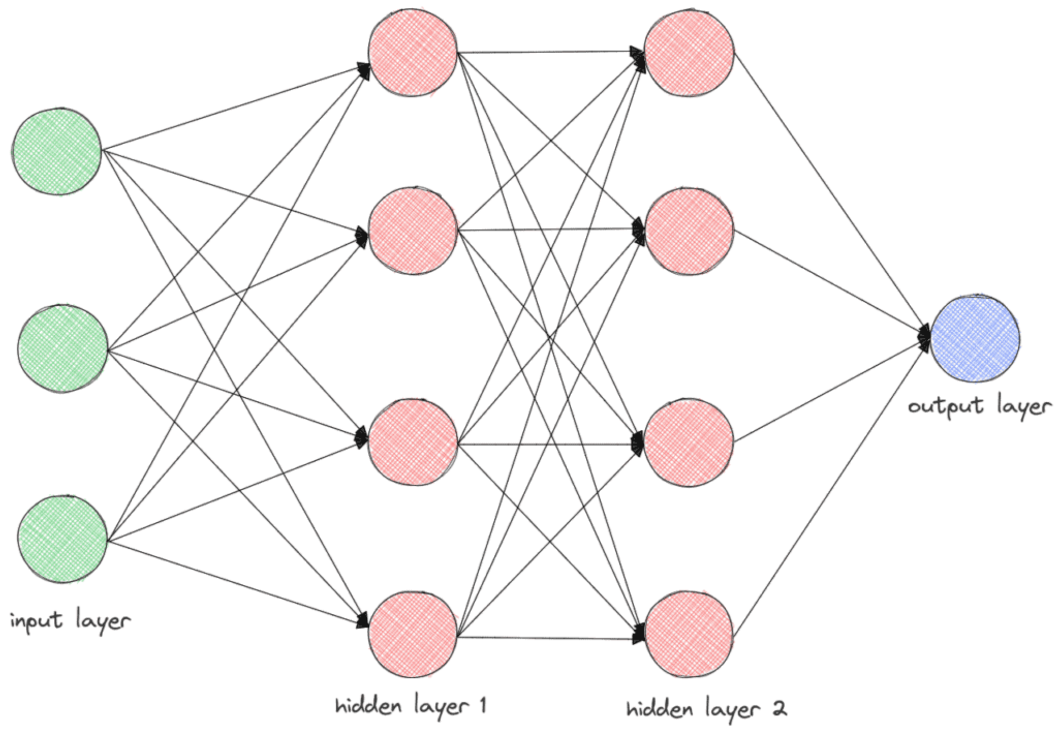
Output Layer:

The input goes through a series of transformations using the hidden layer, which finally results in output that is conveyed using this layer.

The artificial neural network takes input and computes the weighted sum of the inputs and includes a bias. This computation is represented in the form of a transfer function

Hidden Layer

A **Hidden Layer** is a crucial component of an **Artificial Neural Network (ANN)** that lies **between** the **input layer** and the **output layer**. These layers **process** the input data by performing mathematical transformations using **weights, biases, and activation functions**, allowing the network to learn complex patterns and relationships.



Types of Hidden Layer

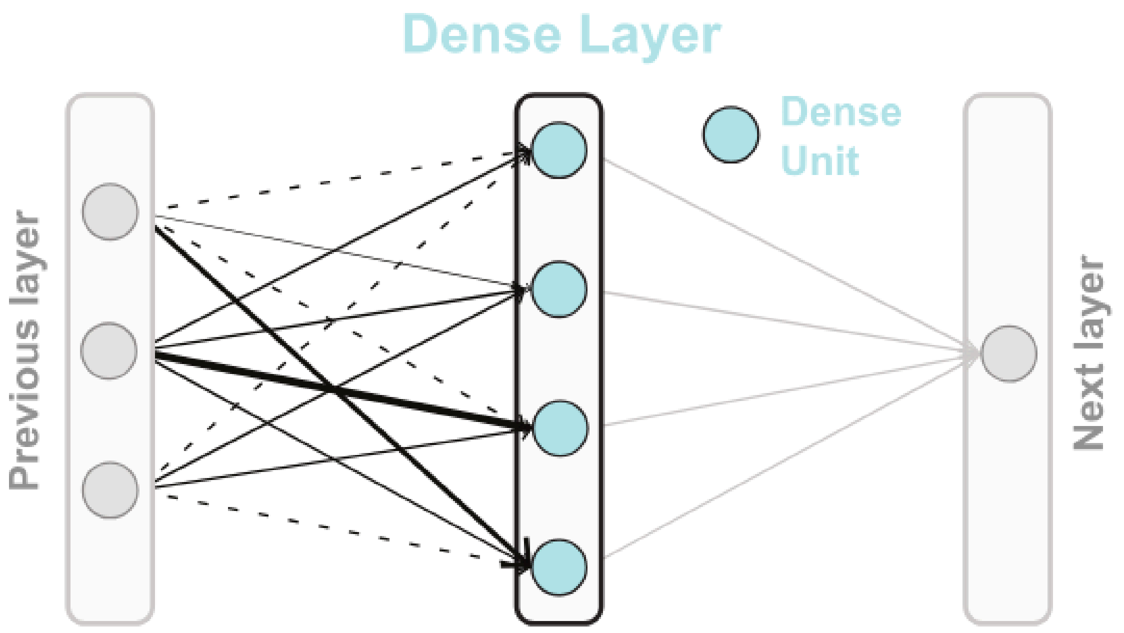
1.Dense Layer (Fully Connected Layer)

A **Dense Layer** is the most common type of hidden layer in an **Artificial Neural Network (ANN)**. In this layer, **every neuron** is connected to **all neurons** in both the **previous** and **next** layers, making it a **fully connected** structure.

This layer performs two key operations:

1. **Weighted Sum of Inputs**: Each neuron computes a weighted sum of all its inputs.
2. **Activation Function**: A non-linear activation function (such as **ReLU, Sigmoid, or Tanh**) is applied to introduce non-linearity, allowing the network to learn complex patterns.

Role: Learns high-level representations from input data.
Function: Performs a weighted sum of inputs followed by activation.
Flexibility: Works with various activation functions and optimizers to adapt to different learning tasks.
Trainable Parameters: Contains a large number of parameters (weights and biases), making it computationally expensive but powerful for learning.

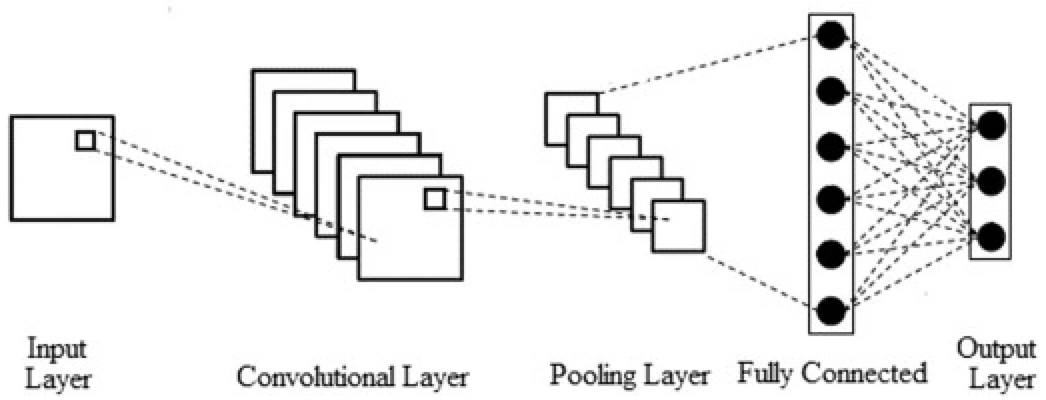


Example:

In a **Natural Language Processing (NLP) model**, a **Dense Layer** is often used in the **final classification step** of a **sentiment analysis model**. After extracting text features using an **Embedding Layer** and **Recurrent Layers (like LSTMs)**, a Dense Layer with a **Softmax activation** predicts sentiment categories (e.g., **positive, neutral, negative**).

2. Convolutional Layer

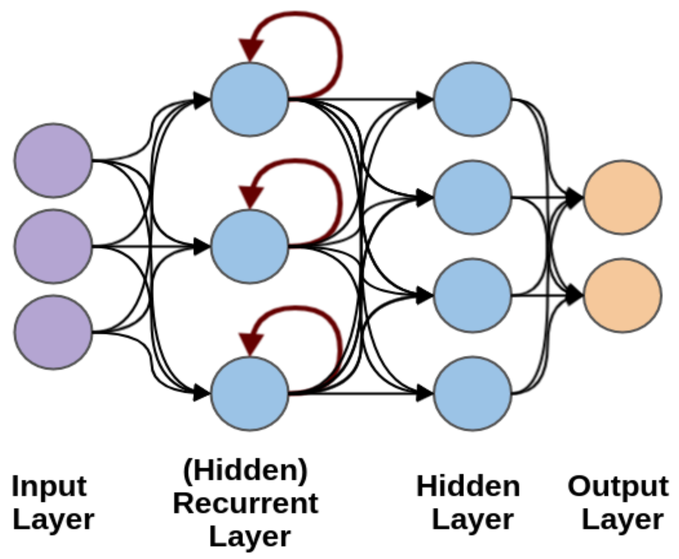
Convolutional Neural Networks (CNNs) are a specialized type of neural network that excels in tasks like image classification, object detection, and segmentation. They are similar to regular Neural Networks in that they consist of neurons with learnable weights and biases. Each neuron processes input through a dot product, followed by an optional non-linearity. Despite the differences in structure, the underlying goal remains the same: transforming raw input into differentiable outputs. For CNNs, this involves transforming raw image pixels into class scores or features.



Function: These layers apply convolution using filters to scan the input and create feature maps. This helps in detecting patterns like edges and textures.
Role: Extracts spatial features from images by identifying patterns in data.
Example: In **object detection**, convolutional layers detect edges and shapes, and in **facial recognition**, they learn features like eyes and facial contours.

3. Recurrent Layer (RNN, LSTM, GRU)

Recurrent layers, like LSTM (Long Short-Term Memory) and GRU (Gated Recurrent Unit), are used in Recurrent Neural Networks (RNNs) to handle sequential data such as time series or text. They have feedback loops that allow information to persist across time steps, making them ideal for tasks with context and temporal dependencies.



Role: Processes sequential data with temporal dependencies.

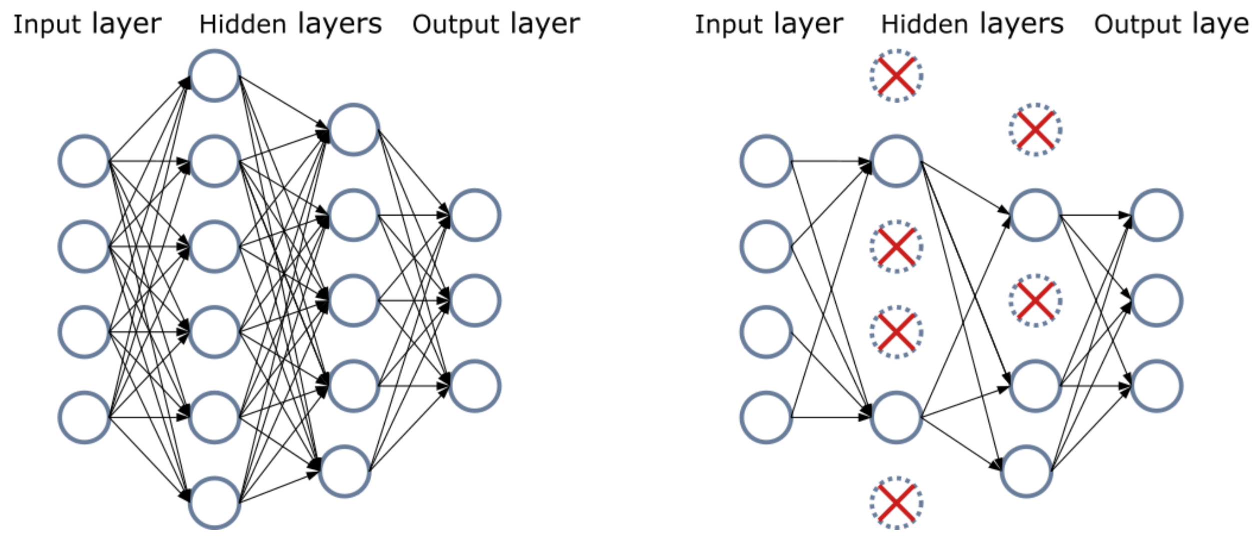
Function: Maintains state across time steps.

Example: Used in language modeling and time series prediction

4. Dropout Layer

Dropout layers are a **regularization technique** used to **prevent overfitting** in neural networks. During training, a fraction of neurons are **randomly dropped** (set to zero) at each training step, which forces the network to learn **more robust features** and generalize better. The neurons that remain active are chosen with a probability **p**, and during training, the network essentially learns with a subset of neurons, making it harder for the model to memorize the data.

This approach helps the model avoid overfitting by preventing it from becoming too reliant on specific neurons or connections. As a result, the network is encouraged to learn distributed, more **generalizable features** that perform better on unseen data.

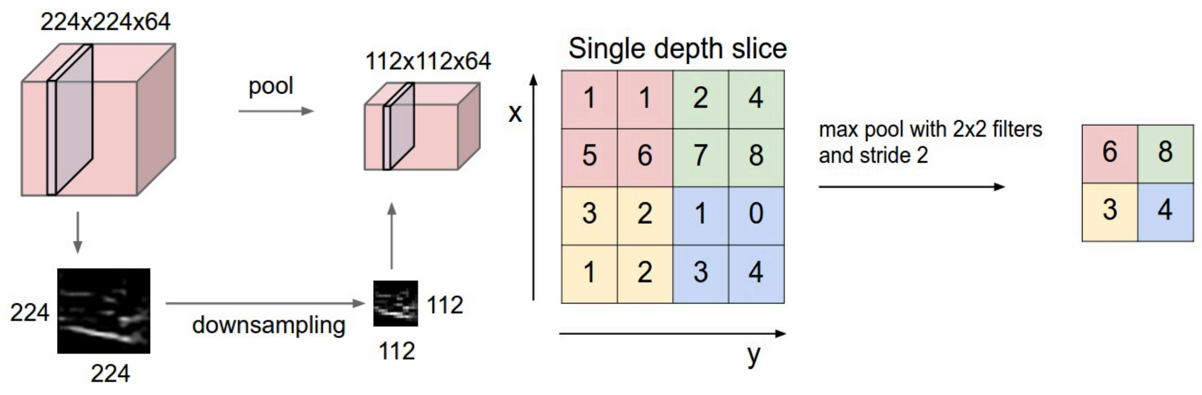


Role: **Prevents overfitting** by reducing model complexity.
Function: **Randomly drops neurons** during training to make the model less reliant on specific neurons.
Example: Often used in **deep learning models** (e.g., CNNs, RNNs) to improve **generalization** on unseen data.

5. Pooling Layer

It is common to periodically insert a **Pooling Layer** between successive **Convolutional Layers** in a ConvNet architecture.
The function of the pooling layer is to progressively reduce the **spatial size** of the feature maps, which helps in reducing the number of parameters and computation, thereby controlling overfitting.
The **Pooling Layer** operates independently on every **depth slice** of the input and resizes it spatially using operations like **Max Pooling** or **Average Pooling**.

The most common form of pooling is **Max Pooling**, where a **2×2 filter** with a **stride of 2** is applied to downsample every depth slice in the input by a factor of **2** along both width and height, effectively discarding **75%** of the activations.
Every **Max operation** takes the **maximum value** from a small **2×2 region** in a depth slice, retaining only the most important features while reducing computational complexity.

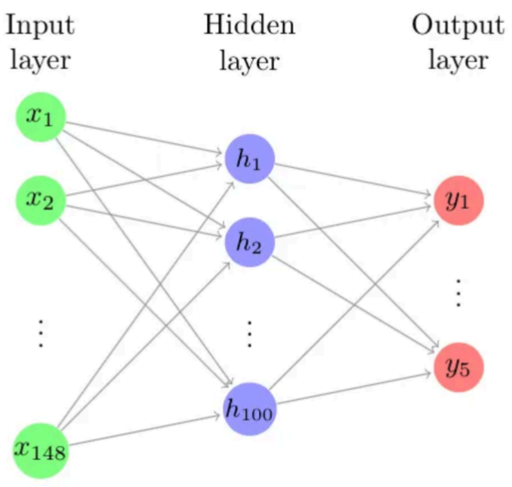


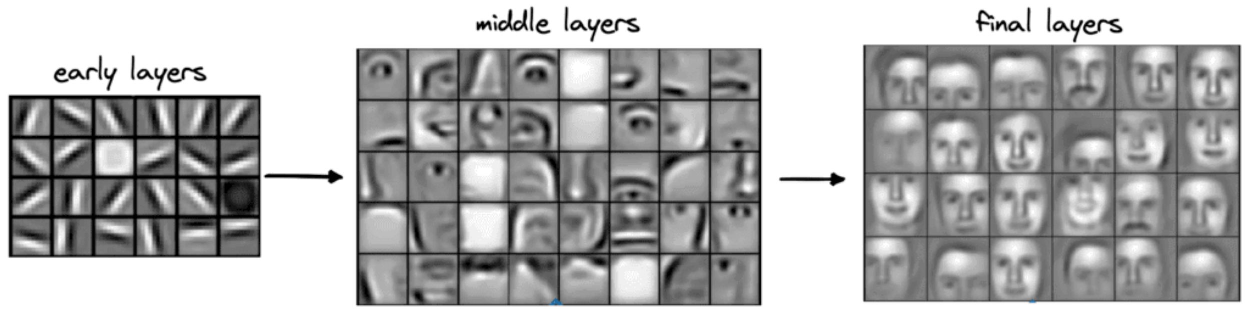
Pooling layer downsamples the volume spatially, independently in each depth slice of the input volume. **Left**: In this example, the input volume of size [224x224x64] is pooled with filter size 2, stride 2 into output volume of size [112x112x64]. Notice that the volume depth is preserved. **Right:** The most common downsampling operation is max, giving rise to max pooling, here shown with a stride of 2. That is, each max is taken over 4 numbers (little 2x2 square).

6. Batch Normalization Layer

A **Batch Normalization (BN) Layer** normalizes the output of the previous activation layer by **subtracting the batch mean** and **dividing by the batch standard deviation**. This ensures that the activations of each layer have a **mean of zero** and a **standard deviation of one**. By doing so, it helps in **accelerating the training process** and **improving the overall performance** of the neural network. Batch normalization also reduces the risk of **vanishing/exploding gradients** and allows the network to use **higher learning rates**, which further speeds up training.

The BN layer can be applied to **both convolutional** and **fully connected layers**, and it is typically followed by a **scaling and shifting step**, where learnable parameters (gamma and beta) are introduced to restore the representational power of the model after normalization



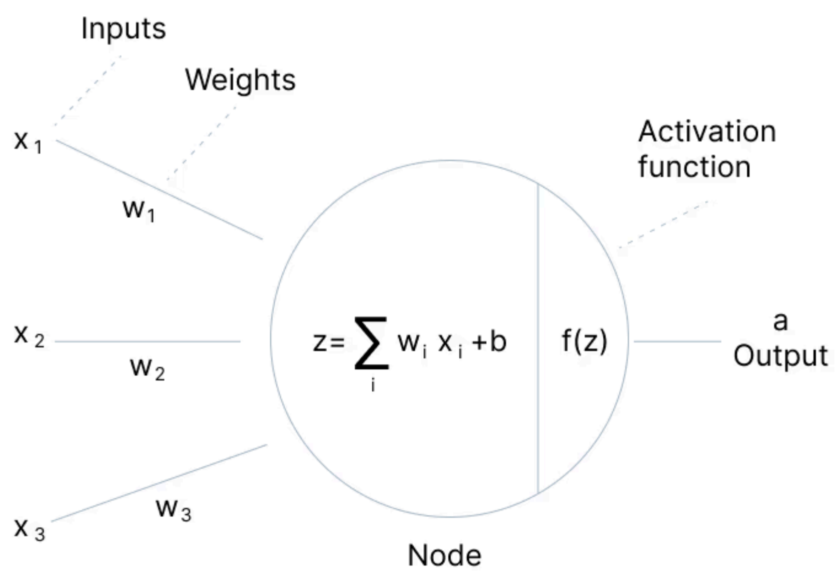


Activation Function

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 role of the Activation Function is to derive output from a set of input values fed to a node (or a layer).

What exactly is a node? Well, if we compare the neural network to our brain, a node is a replica of a neuron that receives a set of input signals—external stimuli. In Deep Learning this is also the role of the Activation Function—that's why it's often referred to as a Transfer Function in Artificial Neural Network.

The primary role of the Activation Function is to transform the summed weighted input from the node into an output value to be fed to the next hidden layer or as output.

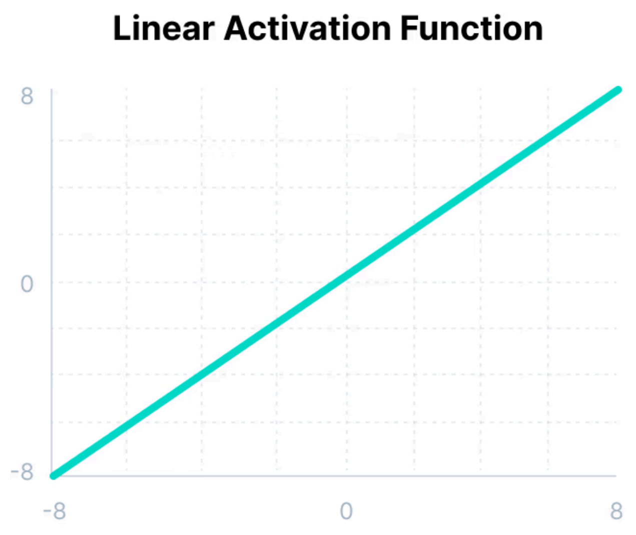


Different types of Activation function

Linear Activation Function

Linear Activation Function resembles straight line define by y=x. No matter how many layers the neural network contains, if they all use linear activation functions, the output is a linear combination of the input.
The range of the output spans from
$(-\infty \text{ to } +\infty)$ $(-\infty \text{ to } +\infty)$
Linear activation function is used at just one place i.e. output layer.
Using linear activation across all layers makes the network's ability to learn complex patterns limited.

Linear activation functions are useful for specific tasks but must be combined with non-linear functions to enhance the neural network's learning and predictive capabilities.



The linear activation function, also known as the identity function or "no activation", is where the activation is directly proportional to the input. Mathematically, it is represented as $f(x) = x$

However, a linear activation function has two major problems :

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 Functions

The linear activation function shown above is simply a linear regression model.

Because of its limited power, this does not allow the model to create complex mappings between the network's inputs and outputs.

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

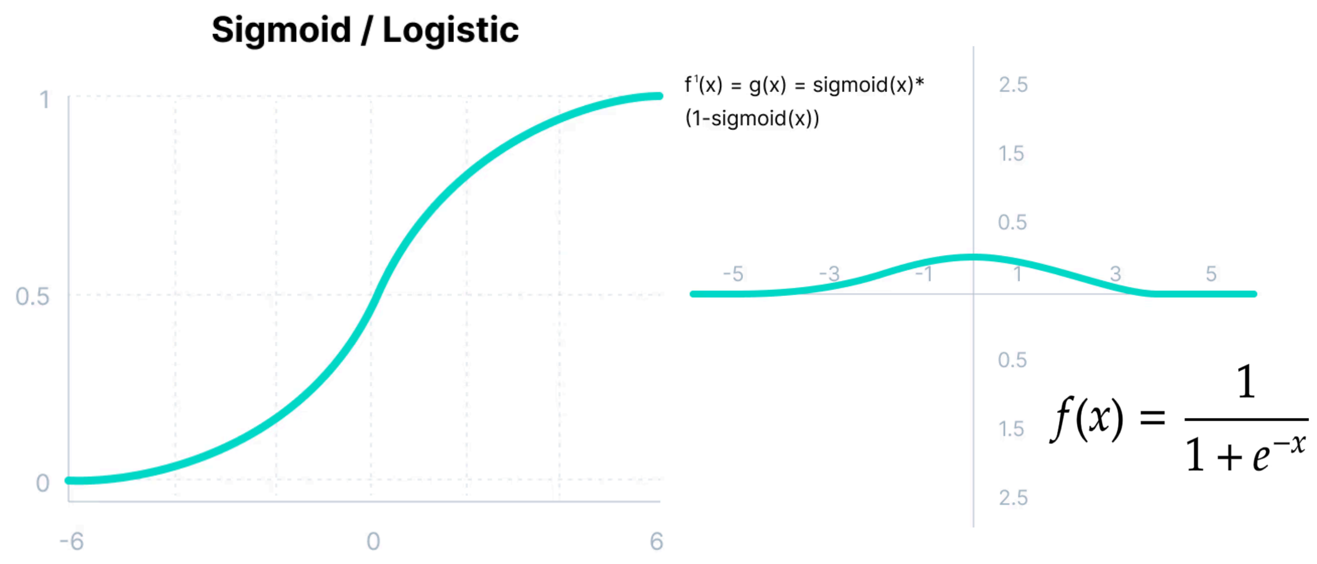
They allow backpropagation because now the derivative function would be related to the input, and it's possible to go back and understand which weights in the input neurons can provide a better prediction.

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

This function takes any real value as input and outputs values in the range of 0 to 1.

The larger the input (more positive), the closer the output value will be to 1.0, whereas the smaller the input (more negative), the closer the output will be to 0.0,



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

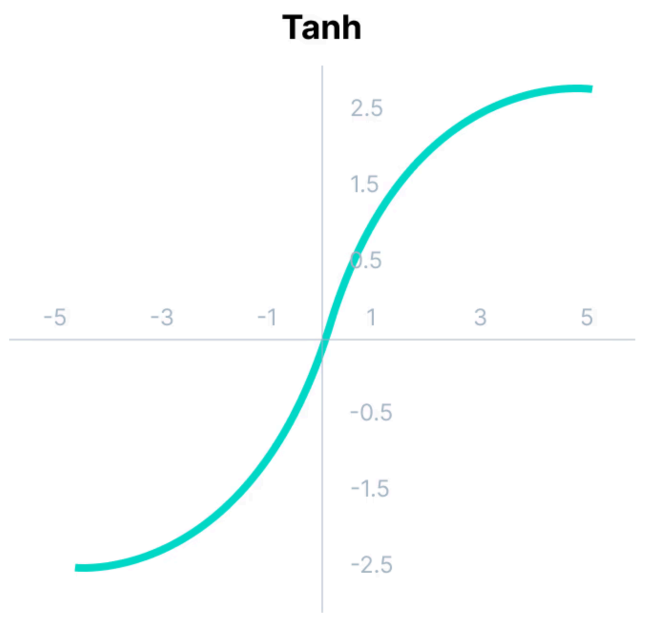
★ It is commonly used for models where we have to predict the probability as an output. Since probability of anything exists only between the **range of 0 and 1**, sigmoid is the right choice because of its range.

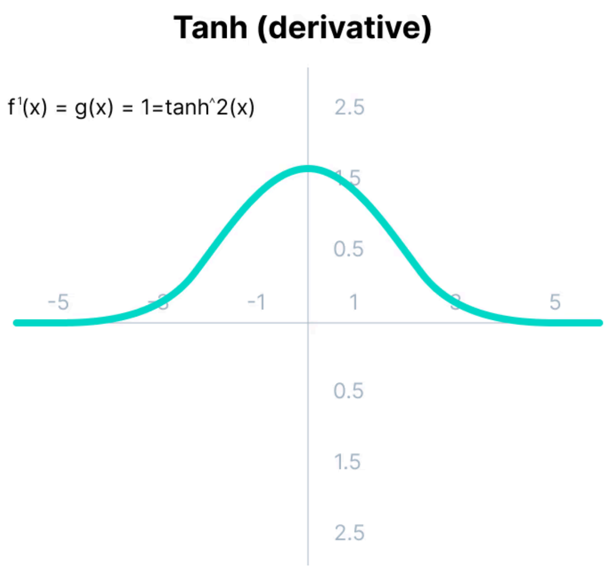
★ The function is differentiable and provides a smooth gradient, i.e., preventing jumps in output values. This is represented by an **S-shape** of the sigmoid activation function.

★ The sigmoid function's gradient is significant only between -3 and 3, becoming nearly zero beyond this range. This leads to the vanishing gradient problem, hindering learning. Additionally, its output is not symmetric around zero.

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 output range of -1 to 1. In Tanh, the larger the input (more positive), the closer the output value will be to 1.0, whereas the smaller the input (more negative), the closer the output will be to -1.0.



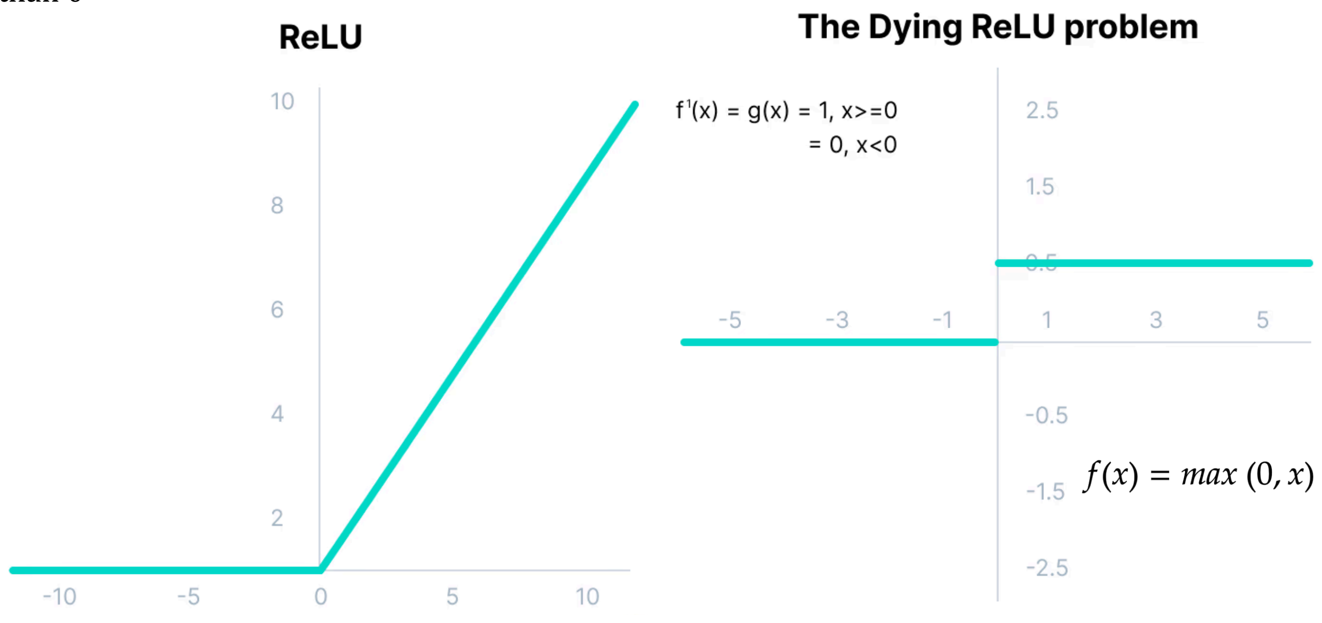


★ The output of the tanh activation function is Zero centered; hence we can easily map the output values as strongly negative, neutral, or strongly positive.
★ Usually used in hidden layers of a neural network as its values lie between -1 to; therefore, the mean for the hidden layer comes out to be 0 or very close to it. It helps in centering the data and makes learning for the next layer much easier.
★ 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.

ReLU Function (Rectified Linear Unit)

Although it gives an 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 neurons will only be deactivated if the output of the linear transformation is less than 0



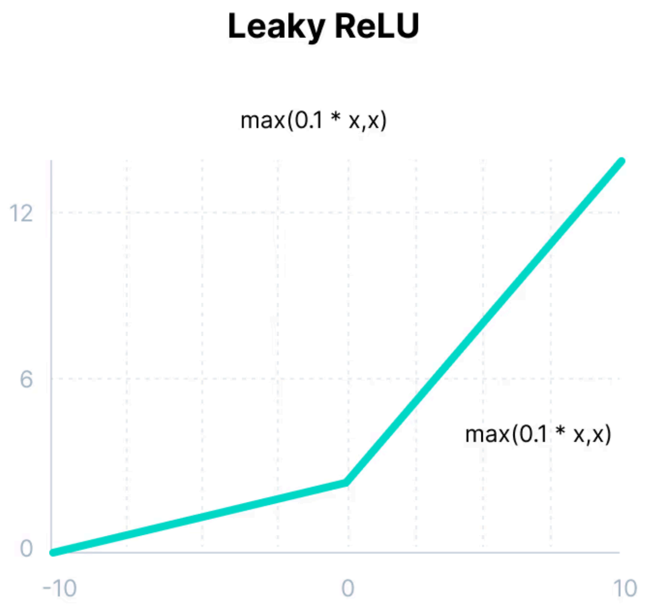
The advantages of using ReLU as an activation function:

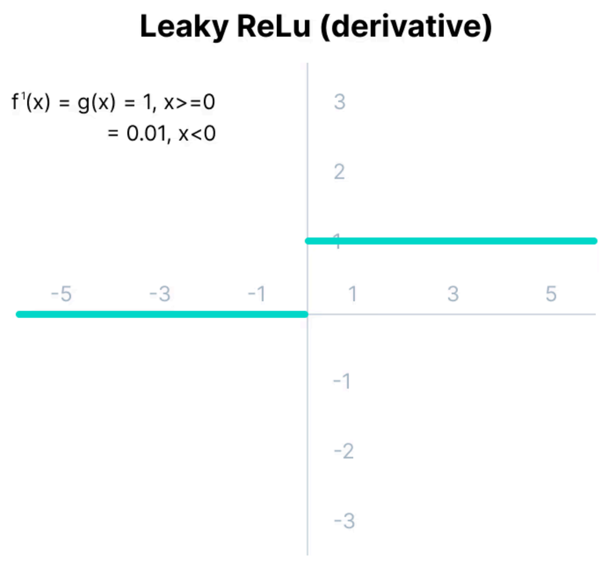
★ Since only a certain number of neurons are activated, the ReLU function is far more computationally efficient when compared to the sigmoid and tanh functions.
★ ReLU accelerates the convergence of gradient descent towards the global minimum of the loss function due to its linear, non-saturating property.

★ 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 which 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 Function

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





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

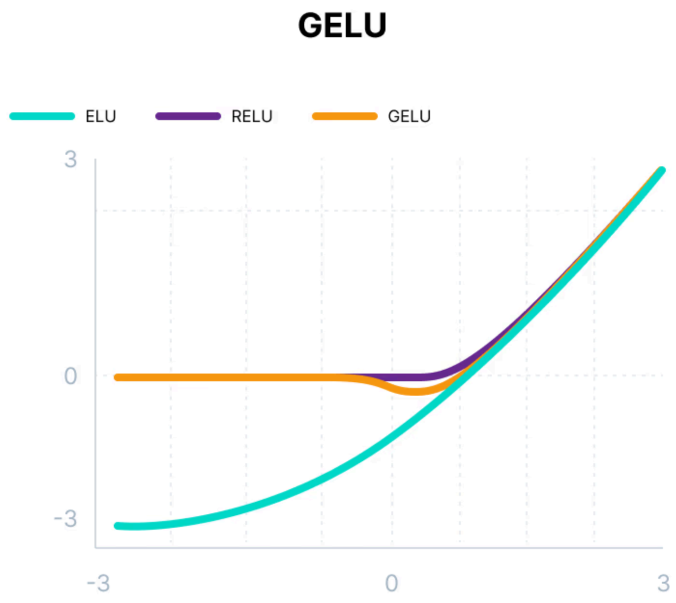
★ The advantages of Leaky ReLU are same as that of ReLU, in addition to the fact that it does enable backpropagation, even for negative input values.
★ By making this minor modification for negative input values, the gradient of the left side of the graph comes out to be a non-zero value. Therefore, we would no longer encounter dead neurons in that region.

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.

Gaussian Error Linear Unit (GELU)

➢ The Gaussian Error Linear Unit (GELU) activation function is compatible with BERT, ROBERTa, ALBERT, and other top NLP models. This activation function is motivated by combining properties from dropout, zoneout, and ReLUs.

- ➢ GELU combines properties of dropout, zoneout, and ReLU.
- ➢ It stochastically multiplies the input by zero or one, based on the input value.
- ➢ Uses a Bernoulli distribution with the cumulative distribution function of a standard normal distribution (Φ(x)).
- ➢ This approach is effective in models like BERT, ROBERTa, and ALBERT.
- ➢ Neuron inputs tend to follow a normal distribution, especially with Batch Normalization.



$$f(x) = xP(X \leq x) = x\Phi(x)$$
$$= 0.5x\left(1 + tanh\left[\sqrt{2/\pi}\left(x + 0.044715x^3\right)\right]\right)$$

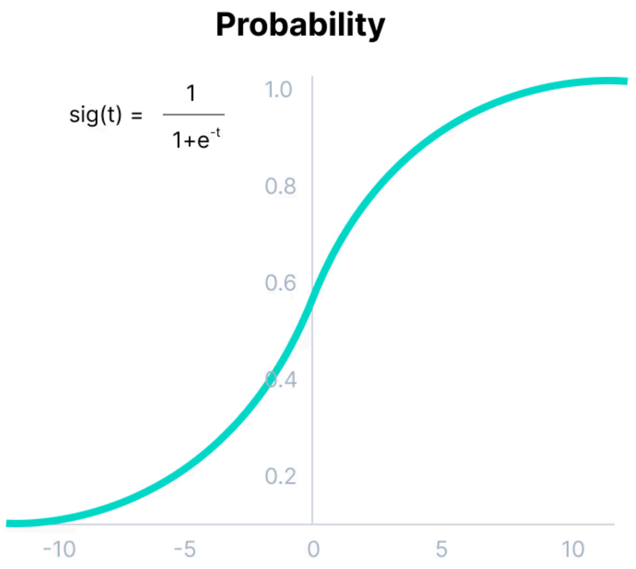
GELU nonlinearity is better than ReLU and ELU activations and finds performance improvements across all tasks in domains of computer vision, natural language processing, and speech recognition.

Softmax Function

Before exploring the ins and outs of the Softmax activation function, we should focus on its building block—the sigmoid/logistic activation function that works on calculating probability values.

- ➢ Sigmoid outputs values between 0 and 1, representing probabilities, but doesn't ensure the sum of all class probabilities equals 1.
- ➢ Softmax is a combination of multiple sigmoids, calculating relative probabilities for each class.
- ➢ It ensures the sum of all probabilities equals 1.

➢ Softmax is commonly used as the activation function in the final layer of a neural network for multi-class classification.



$$\text{softmax}(z_i) = \frac{\exp(z_i)}{\sum_j \exp(z_j)}$$

Optimizers

In deep learning, optimizers are crucial as algorithms that dynamically fine-tune a model’s parameters throughout the training process, aiming to minimize a predefined loss function. These specialized algorithms facilitate the learning process of neural networks by iteratively refining the weights and biases based on the feedback received from the data. Well-known optimizers in deep learning encompass Stochastic Gradient Descent (SGD), Adam, and RMSprop, each equipped with distinct update rules, learning rates, and momentum strategies, all geared towards the overarching goal of discovering and converging upon optimal model parameters, thereby enhancing overall performance.

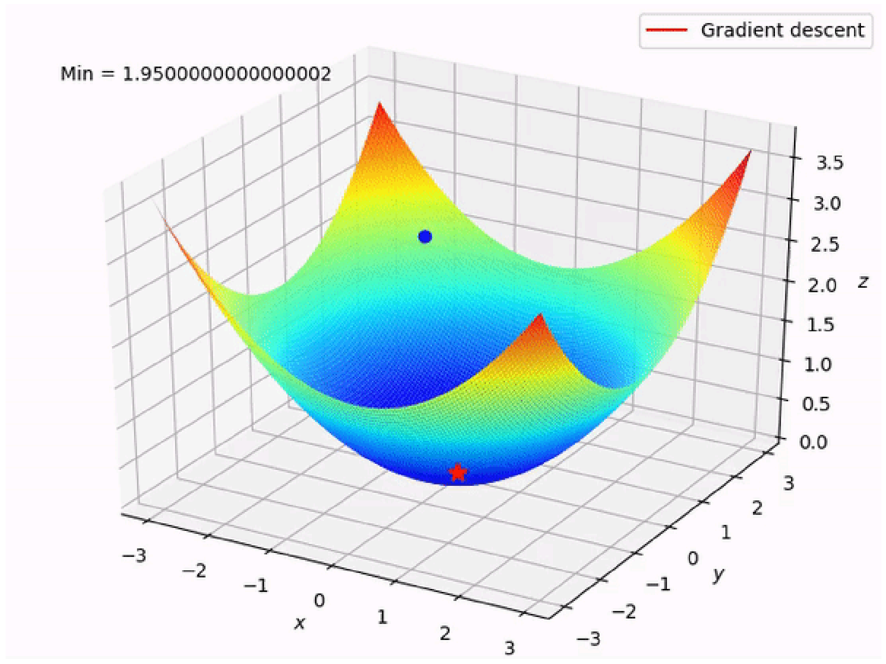
Types of Optimizer

- Gradient Descent
- Stochastic Gradient Descent
- Adagrad
- Adadelta
- RMSprop
- Adam

Gradient Descent :

➔ Gradient Descent is an optimization algorithm used to find the local minimum of a differentiable function.It minimizes a cost function by adjusting the parameters (coefficients) of the function.

- ➔ It is widely used in neural networks, particularly for convex optimization problems.
- ➔ The algorithm updates the weights by moving in the direction of the negative gradient (downhill) to minimize the cost function.
- ➔ It aims to find the best-suited parameter values that correspond to the global minima.
- ➔ Weights are adjusted based on the slope: a negative slope increases the weight, while a positive slope reduces it.



Finding the global minima for a convex optimization problem using Gradient Descent

Stochastic Gradient Descent

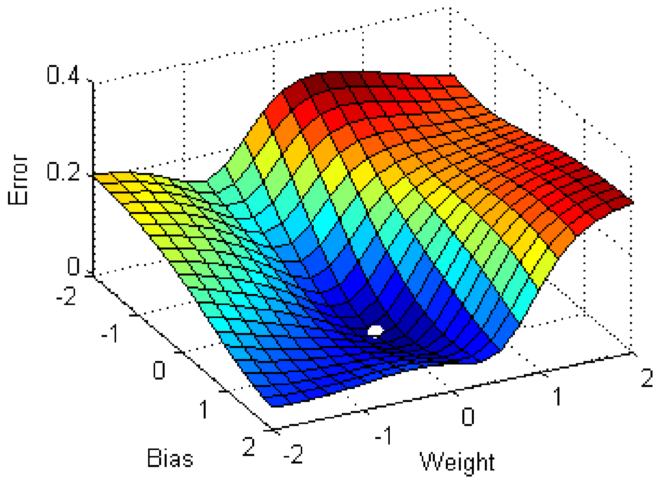
- ➢ Stochastic Gradient Descent (SGD) is a variant of Gradient Descent designed to handle non-convex optimization problems.

- ➢ It addresses the issue of local minima by updating weights one at a time, instead of processing in batches.

- ➢ This approach is faster and minimizes the cost function after each iteration **(epoch).**

- ➢ Frequent updates with high variance allow the gradient to jump to potential global minima, but can cause fluctuations in the cost function.

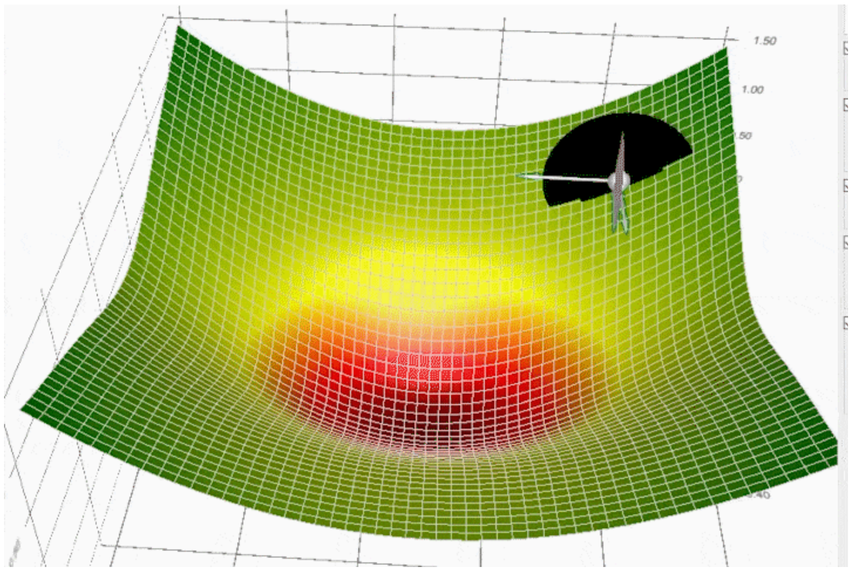
- ➢ Choosing a small learning rate can lead to slow convergence, while a large learning rate can cause poor convergence, with the cost function fluctuating or diverging.



Converging at the global minima using SGD for non-convex data

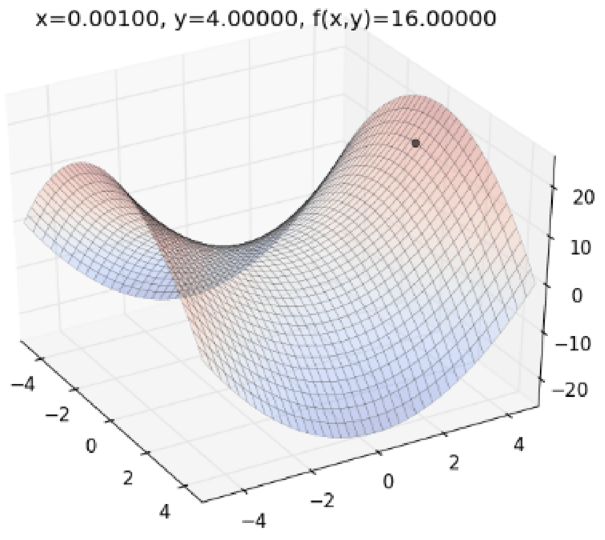
Adagrad (Adaptive Gradient**)**

- ➢ Adagrad is an adaptive gradient optimization algorithm where the learning rate varies for each parameter.
- ➢ Unlike Stochastic Gradient Descent (SGD), it uses a different learning rate for each iteration (epoch).
- ➢ It performs smaller updates for weights corresponding to high-frequency features and larger updates for low-frequency features, improving performance and accuracy.
- ➢ Adagrad is especially effective for sparse data.
- ➢ At each iteration, the learning rate (**α**) is recalculated, and as the number of iterations increases, the learning rate decreases.



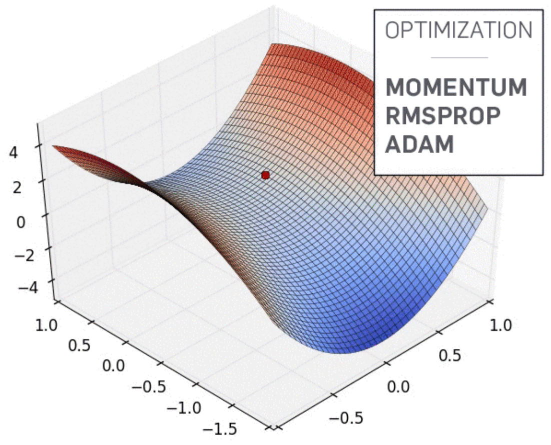
Adadelta

- ➢ **Adadelta** is an extension of the Adagrad optimizer, addressing its issue of aggressively reducing the learning rate.
- ➢ Instead of using past squared gradients, Adadelta uses a weighted average of all past squared gradients, which prevents the learning rate from becoming infinitesimally small.
- ➢ The weight update formula is similar to Adagrad, but the learning rate at each iteration is recalculated using the weighted average.
- ➢ A restricting term ($\gamma = 0.95$) is used to avoid the Vanishing Gradient problem and helps stabilize learning.



RMSprop

- ➢ RMSprop and Adadelta were developed to solve Adagrad's issue of diminishing learning rates.
- ➢ Both use an Exponential Weighted Average to calculate the learning rate at each iteration.
- ➢ RMSprop, proposed by Geoffrey Hinton, adapts the learning rate by dividing it by the exponentially weighted average of squared gradients.
- ➢ It is commonly recommended to set γ (gamma) to 0.95, as it has shown good results in most cases.



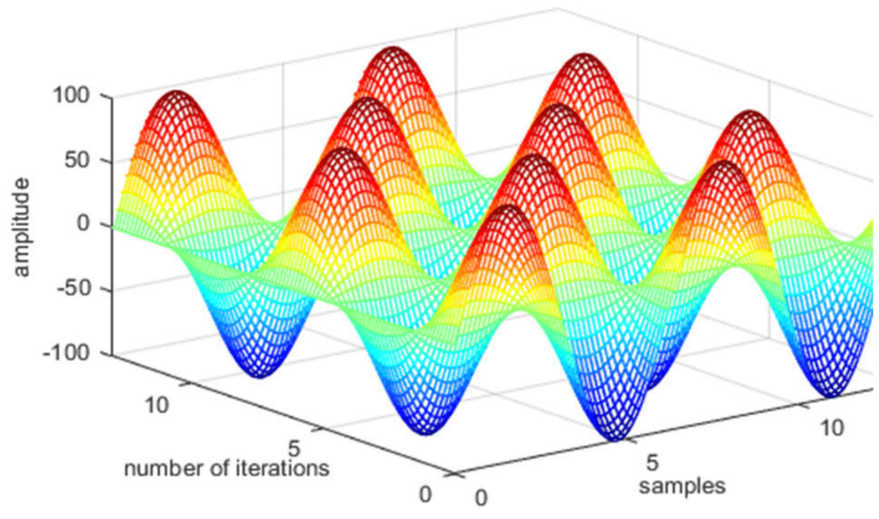
Adam

➢ Adam (Adaptive Moment Estimation) is an optimization algorithm that computes adaptive learning rates for each parameter at every iteration.
➢ It combines Gradient Descent with Momentum and RMSprop to determine parameter updates.
➢ Adam is especially effective for non-convex optimization problems, making it one of the most widely used optimizers.

It has several advantages, such as:

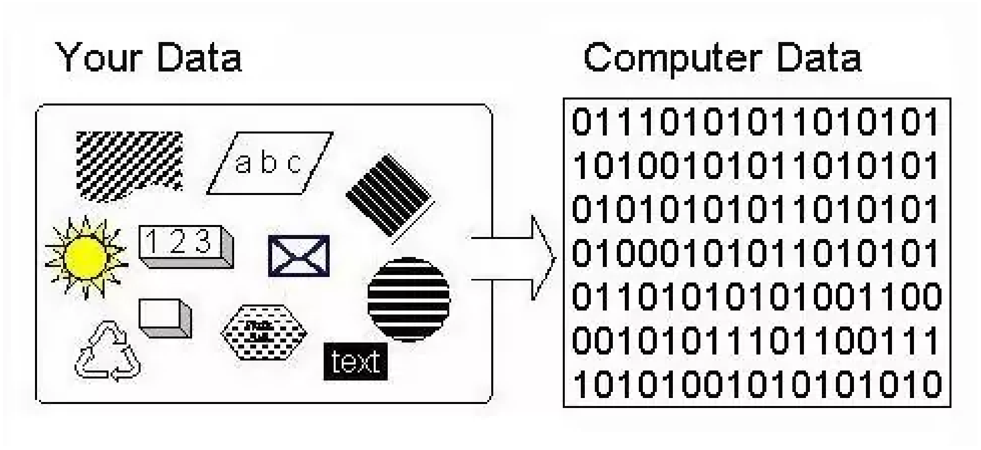
- Low memory requirements.
- Suitable for non-stationary objectives.
- Works well with large data and parameters.
- Efficient computation.

Adam uses adaptive learning rates and stores an exponentially weighted average of the past squared gradients to update the weights.



The Encoder

A structured dataset typically includes a mix of numerical and categorical variables. Machine learning algorithms can only process numerical data, not text. This is where categorical encoding comes into play.In **scikit-learn**, encoding refers to the process of converting categorical variables into a format that can be used by machine learning algorithms, typically by transformingcategorical data into numerical values. There are different types of encoders in scikit-learn to handle categorical data



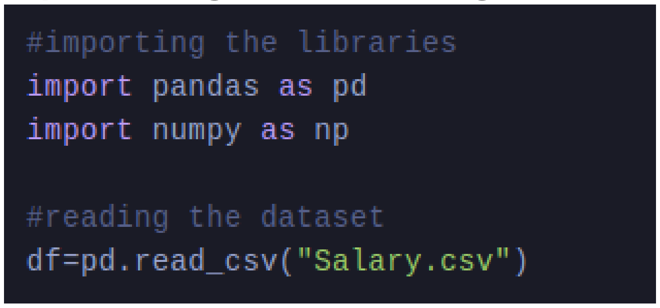
Different types of Encoder in Sci-Kit Learn

- **One-Hot Encoding: Great for many categories, creates new binary features (1 for the category, 0 for others).**
- **Label Encoding: Simple, assigns a number to each category, but assumes order matters (which might not be true).**
- **Ordinal Encoding: Similar to label encoding, but only use it if categories have a natural order (like low, medium, high).**

Label Encoding

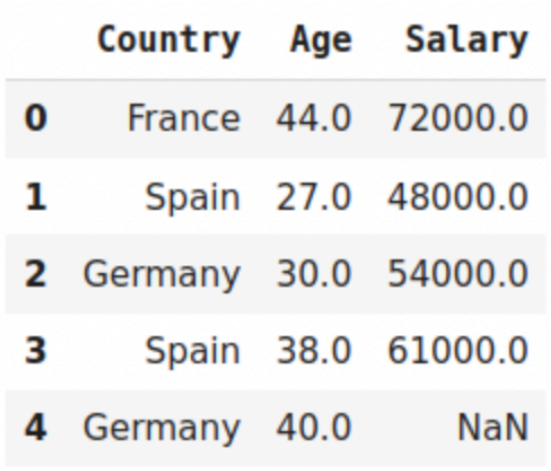
Label Encoding is a common technique for converting categorical variables into numerical values. Each unique category value is assigned a unique integer based on alphabetical or numerical ordering.

Implementing Label Encoding



```
#importing the libraries
import pandas as pd
import numpy as np

#reading the dataset
df=pd.read_csv("Salary.csv")
```



	Country	Age	Salary
0	France	44.0	72000.0
1	Spain	27.0	48000.0
2	Germany	30.0	54000.0
3	Spain	38.0	61000.0
4	Germany	40.0	NaN

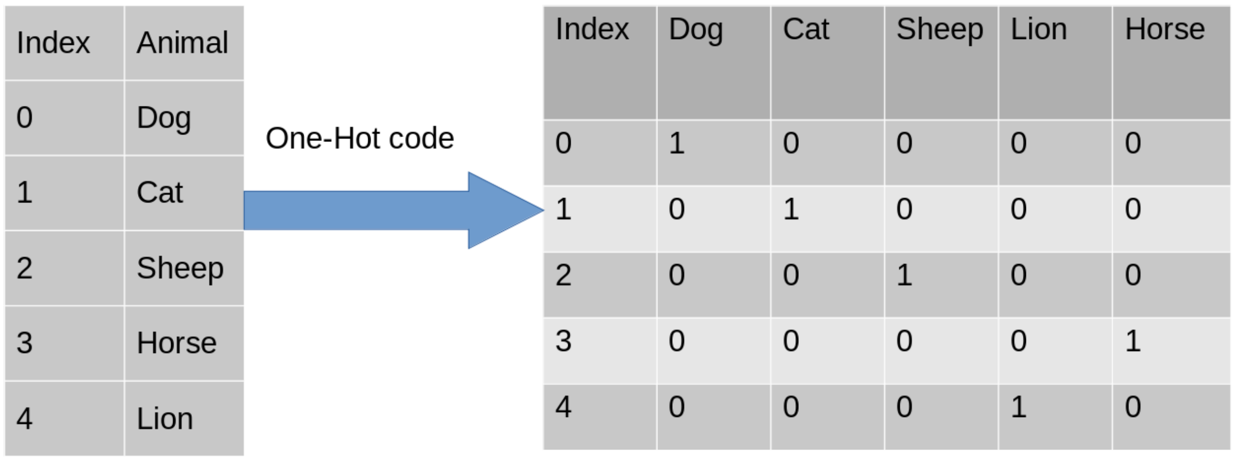
Challenges with Label Encoding:
Label encoding can introduce an arbitrary order in categorical data, which may mislead machine learning models. For example, encoding countries (France, Germany, Spain) assigns integers (e.g., 0, 1, 2), creating an ordinal relationship (France < Germany < Spain), even though no inherent order exists. This can cause the model to incorrectly interpret categories as having a meaningful sequence.

Understanding the differences between One-Hot Encoding and Label Encoding and using them correctly in tools like Pandas and Scikit-learn ensures efficient and accurate conversion of c**ategorical data, while avoiding misinterpretations.**

One-Hot Encoding (OHE)

One-Hot Encoding is another popular technique for treating categorical variables. It simply creates additional features based on the number of unique values in the categorical feature. Every unique value in the category will be added as a feature. One-Hot Encoding is the process of creating dummy variables.

Implementing One-Hot Encoding using Scikit-Learn



Index	Animal
0	Dog
1	Cat
2	Sheep
3	Horse
4	Lion

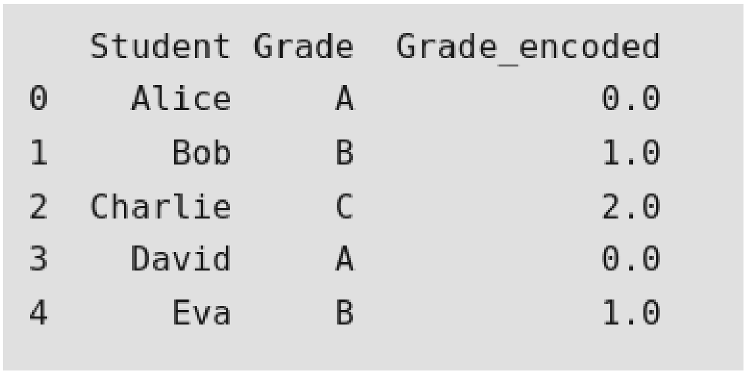
Index	Dog	Cat	Sheep	Lion	Horse
0	1	0	0	0	0
1	0	1	0	0	0
2	0	0	1	0	0
3	0	0	0	0	1
4	0	0	0	1	0

As you can see, three new features are added because the Country column contains three unique values – France, Spain, and Germany. This method avoids the problem of ranking inherent in one hot encoding and label encoding, as each category is represented by a separate binary vector.

OrdinalEncoder

Ordinal Encoding is similar to **Label Encoding** but is specifically designed for **ordinal data**, where the categories have a meaningful order. It preserves the order of categories, making it ideal for data like grades (A, B, C) or levels (Low, Medium, High), where the sequence matters. This encoding helps machine learning algorithms, which function best with numerical data, process ordered categorical variables efficiently.

Ordinal encoding converts categorical data into numeric values by assigning a unique integer to each category. It is ideal for variables with inherent order, like **size (small < medium < large)**. However, it is also applied to unordered categories since models like **decision trees** can still learn from arbitrary assignments. A key advantage over **one-hot encoding** is that it keeps the feature space compact, avoiding unnecessary dimensionality expansion.

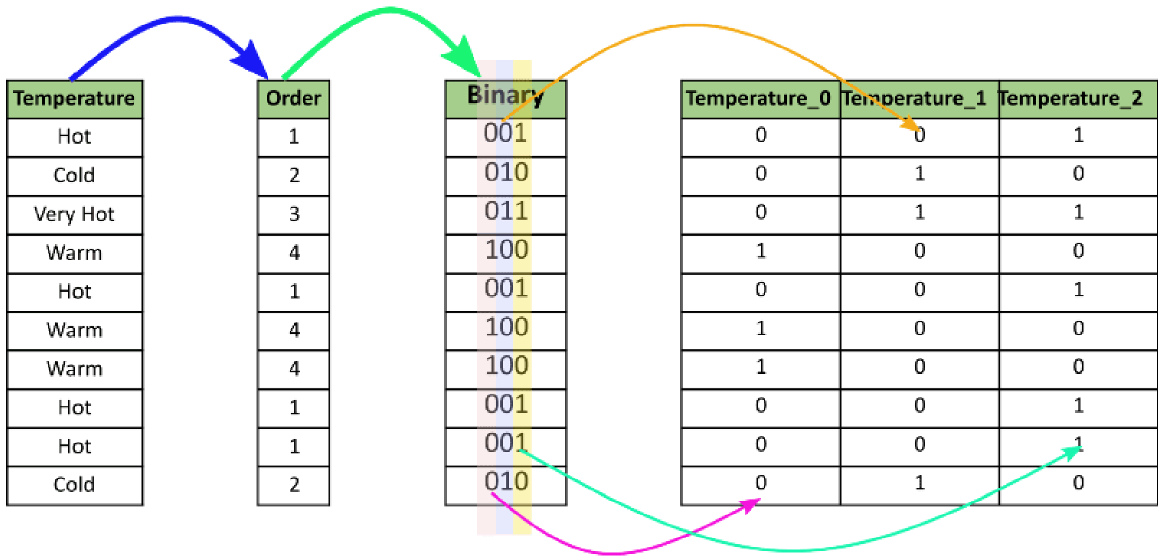


```
   Student Grade  Grade_encoded
0    Alice     A            0.0
1      Bob     B            1.0
2  Charlie     C            2.0
3    David     A            0.0
4      Eva     B            1.0
```

CategoricalEncoder

The **CategoricalEncoder** was a class in earlier versions of **scikit-learn** used for encoding categorical data into numerical representations. It supported both **one-hot encoding** and **ordinal encoding** for categorical variables, allowing users to transform categorical features into a format that machine learning algorithms could process.

However, **CategoricalEncoder** has been **deprecated** in **scikit-learn** due to its overlap with other more specific and effective encoders like **OneHotEncoder** and **OrdinalEncoder**, which provide better handling and functionality for encoding categorical features.



Temperature	Order	Binary	Temperature_0	Temperature_1	Temperature_2
Hot	1	001	0	0	1
Cold	2	010	0	1	0
Very Hot	3	011	0	1	1
Warm	4	100	1	0	0
Hot	1	001	0	0	1
Warm	4	100	1	0	0
Warm	4	100	1	0	0
Hot	1	001	0	0	1
Hot	1	001	0	0	1
Cold	2	010	0	1	0