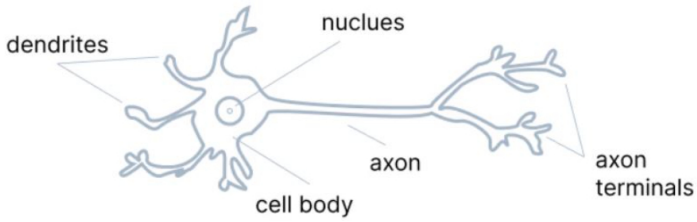
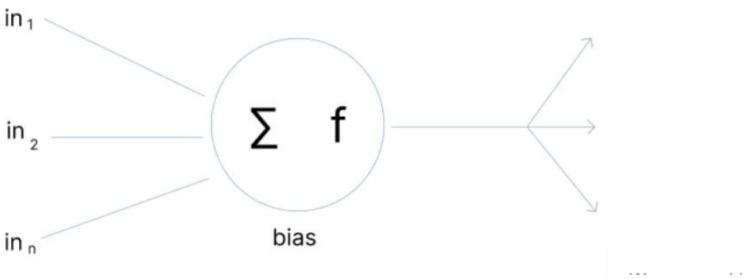
```
[ ] Start coding or generate with AI.
```

What is an Activation Function?

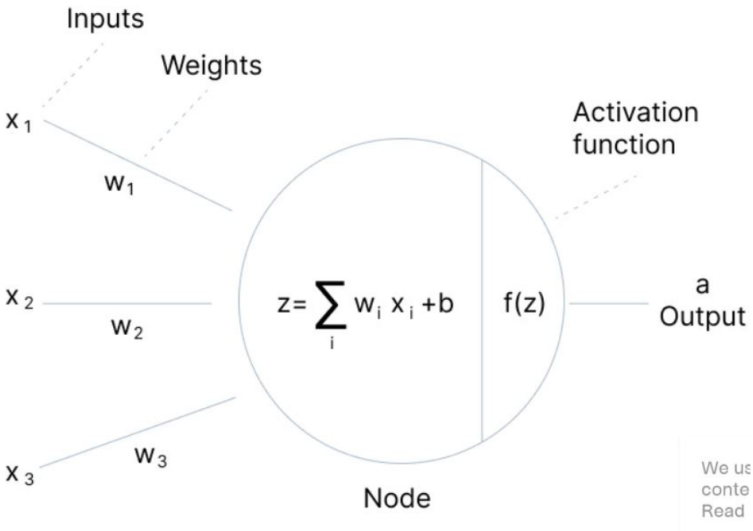
An Activation Function decides whether a neuron should be activated or not. This means that it will decide whether the neuron's input to the network is important or not in the process of prediction using simpler mathematical operations.

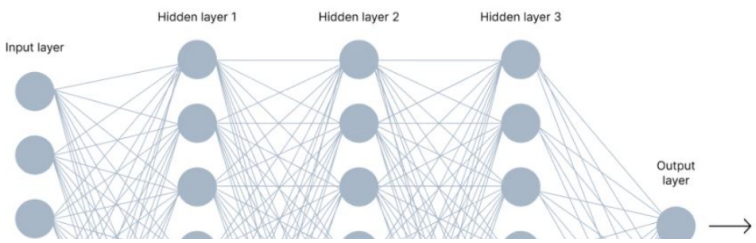


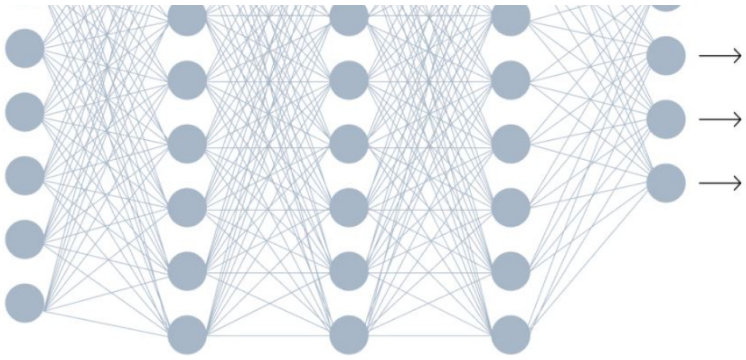


The primary role of the Activation Function is to transform the summed weighted input from the node into an output value to be fed to the next hidden layer or as output.

Depending on the nature and intensity of these input signals, the brain processes them and decides whether the neuron should be activated ("fired") or not.







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.

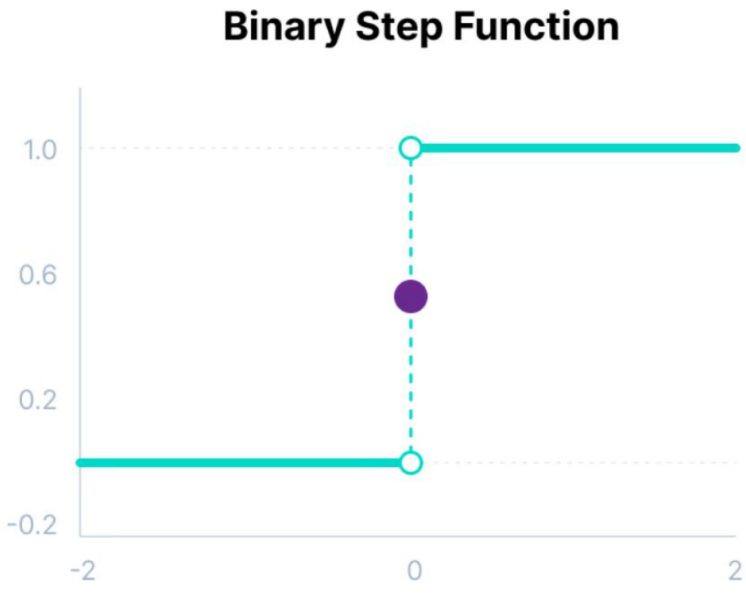
3 Types of Neural Networks Activation Functions

Now, as we've covered the essential concepts, let's go over the most popular neural networks activation functions.

Binary Step 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.



$$\textit{Binary step}$$

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

Here are some of the limitations of binary step function:

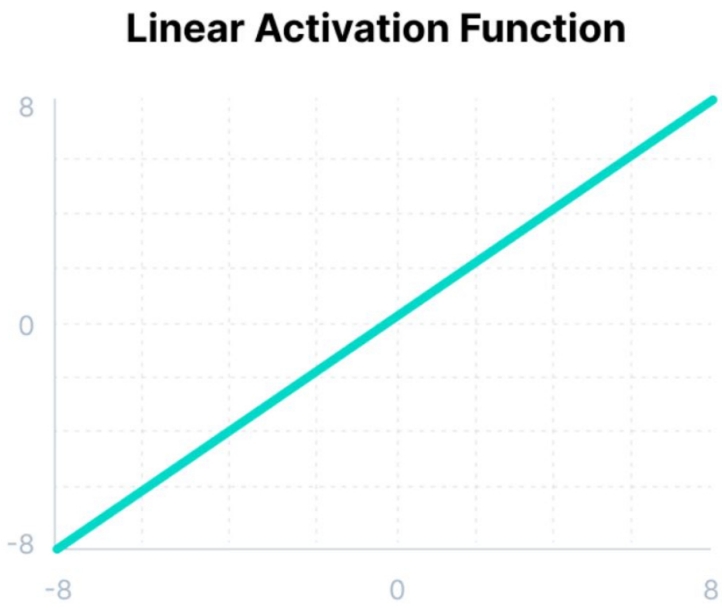
It cannot provide multi-value outputs—for example, it cannot be used for multi-class classification problems.

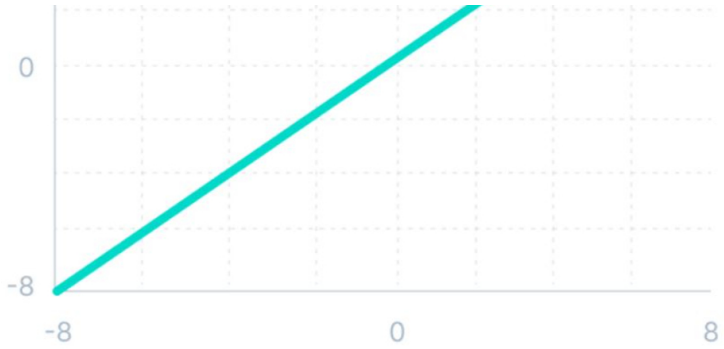
The gradient of the step function is zero, which causes a hindrance in the backpropagation process.

Linear activation function

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

The function doesn't do anything to the weighted sum of the input, it simply spits out the value it was given.





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, as shown below.

Sigmoid / Logistic

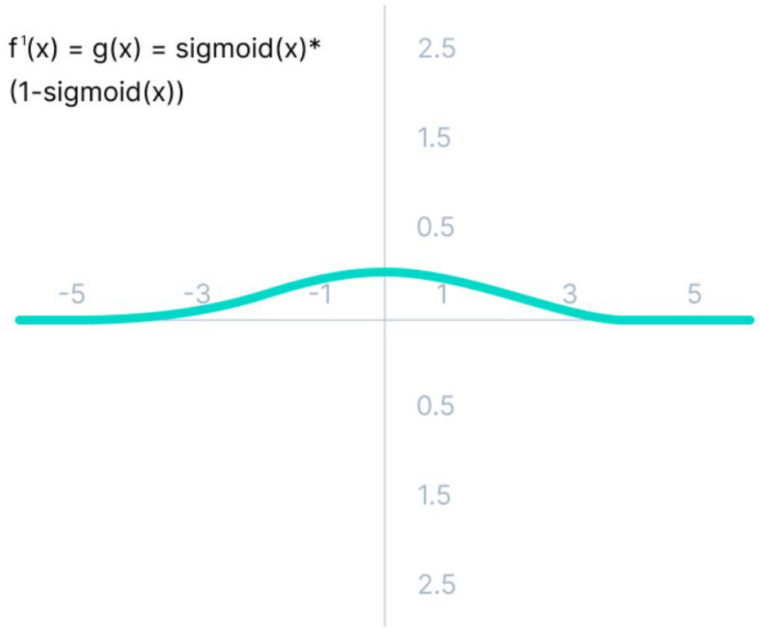
$$f(x) = \frac{1}{1 + e^{-x}}$$

Here's why 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 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 limitations of sigmoid function are discussed below:

- The derivative of the function is f'(x) = sigmoid(x)*(1-sigmoid(x)).



As we can see from the above Figure, the gradient values are only significant for range -3 to 3, and the graph gets much flatter in other regions.

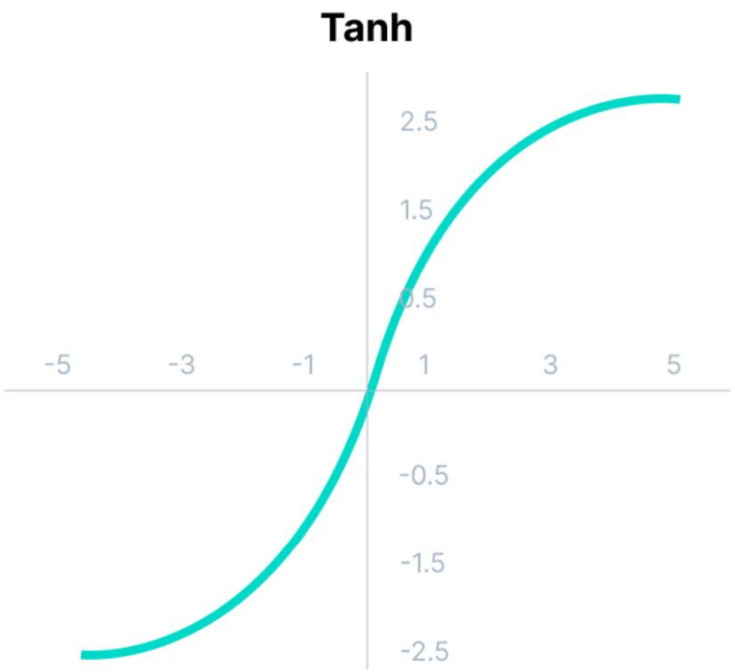
It implies that for values greater than 3 or less than -3, the function will have very small gradients. As the gradient value approaches zero, the network ceases to learn and suffers from the Vanishing gradient problem.

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 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 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.



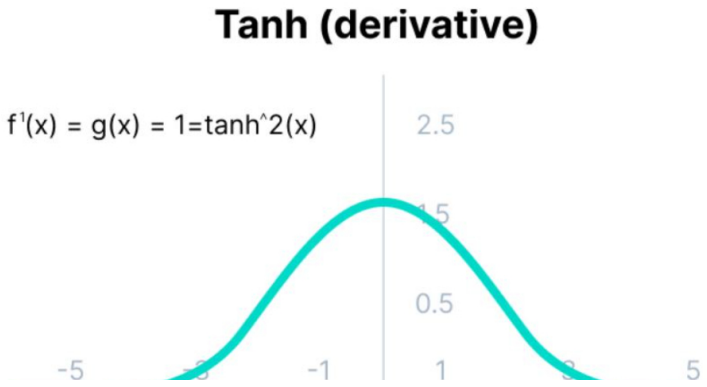
$$Tanh$$

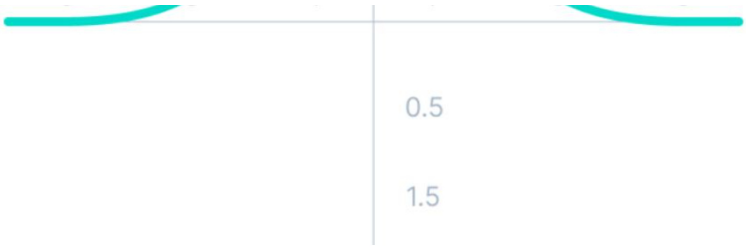
$$f(x) = \frac{\left(e^{x} - e^{-x}\right)}{\left(e^{x} + e^{-x}\right)}$$

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.

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.





Although both sigmoid and tanh face vanishing gradient issue, tanh is zero centered, and the gradients are not restricted to move in a certain direction. Therefore, in practice, tanh nonlinearity is always preferred to sigmoid nonlinearity.

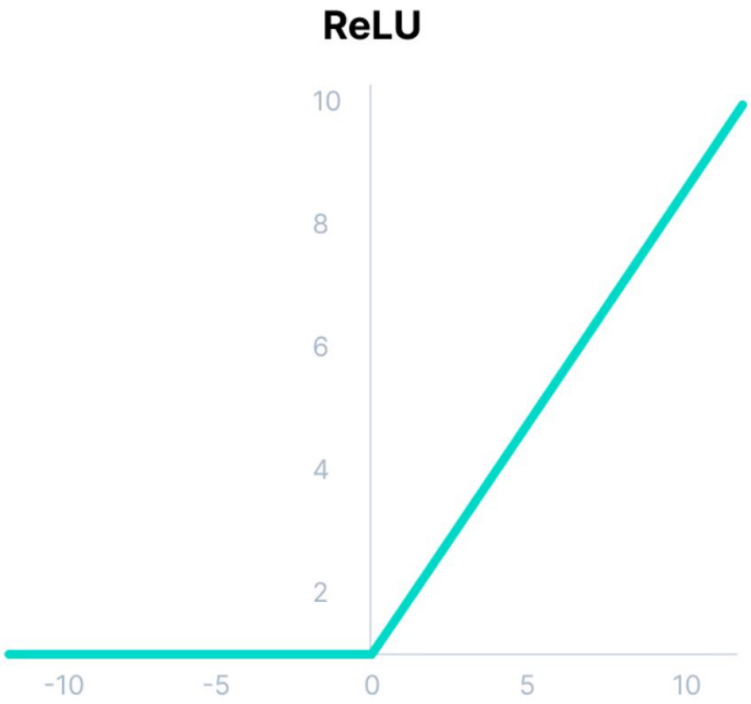
- ReLU Function

ReLU stands for 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.



$$ReLU$$

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

- 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.

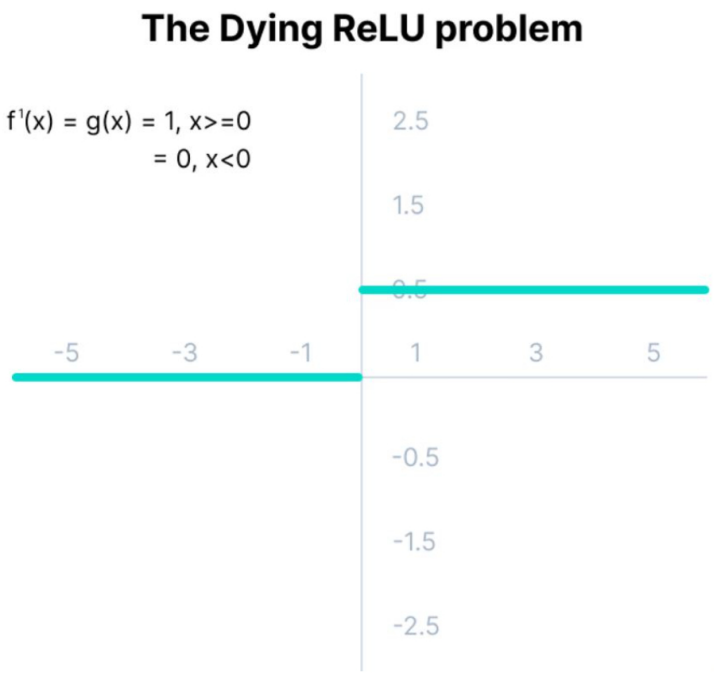
ReLU accelerates the convergence of gradient descent towards the global minimum of the loss

ReLU accelerates the convergence of gradient descent towards the global minimum of the loss function due to its linear, non-saturating property.

The limitations faced by this function are:

The Dying ReLU problem, which I explained below.

Double-click (or enter) to edit



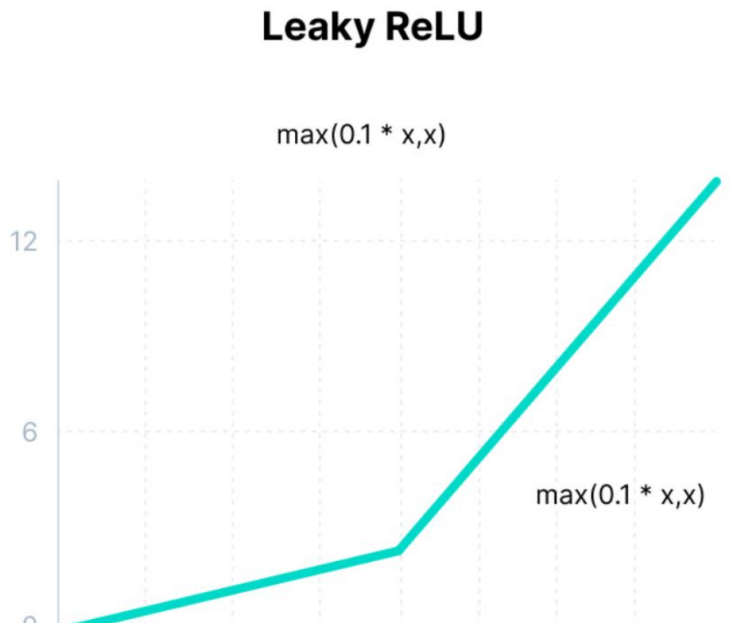
The negative side of the graph makes the gradient value zero. Due to this reason, during the backpropagation process, the weights and biases for some neurons are not updated. This can create dead neurons which never get activated.

All the negative input values become zero immediately, which decreases the model's ability to fit or train from the data properly.

Note: For building the most reliable ML models, split your data into train, validation, and test sets.

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.



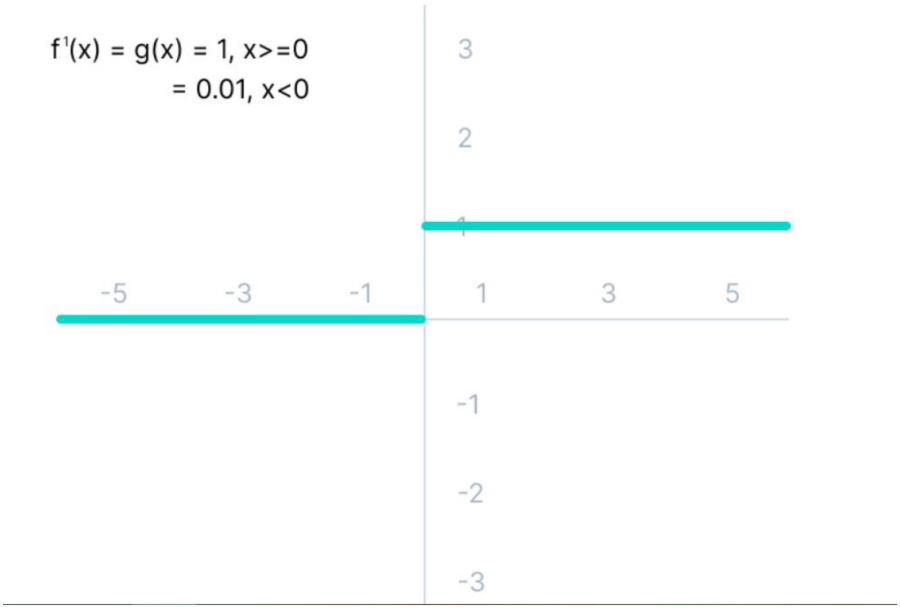
Leaky ReLU

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

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.

Here is the derivative of the Leaky ReLU function.



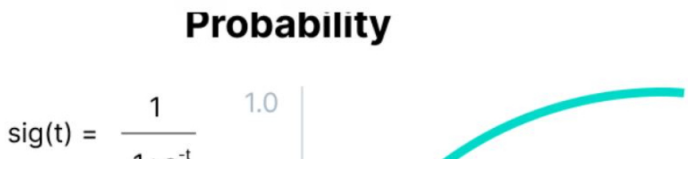
The limitations that this function faces include:

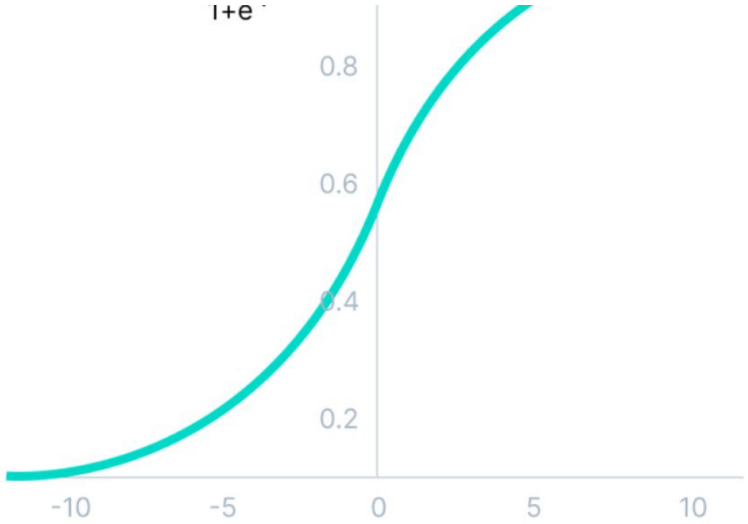
The predictions may not be consistent for negative input values.

The gradient for negative values is a small value that makes the learning of model parameters time-consuming.

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.





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.

We can 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.

Mathematically it can be represented as:

Softmax

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

How to choose the right Activation Function?

We 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 rule of thumb, we 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.

ReLU activation function should only be used in the hidden layers.

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).

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

Regression - Linear Activation Function

Binary Classification—Sigmoid/Logistic Activation Function

Multiclass Classification—Softmax

Multilabel Classification—Sigmoid

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

Convolutional Neural Network (CNN): ReLU activation function.

Recurrent Neural Network: Tanh and/or Sigmoid activation function.

Activation Functions are used to introduce non-linearity in the network.

A neural network will almost always have the same activation function in all hidden layers. This activation function should be differentiable so that the parameters of the network are learned in backpropagation.

ReLU is the most commonly used activation function for hidden layers.

While selecting an activation function, you must consider the problems it might face: vanishing and exploding gradients.

Regarding the output layer, we must always consider the expected value range of the predictions. If it can be any numeric value (as in case of the regression problem) you can use the linear activation function or ReLU.

Use Softmax or Sigmoid function for the classification problems.

implemenation of Activation Fucntions

```
[1] import math

    def sigmoid(x):
      return 1 / (1 + math.exp(-x))
```

```
[ ] sigmoid(100)
```

```
[ ] sigmoid(1)
```

```
[2] sigmoid(-56)
```

```
4.780892883885469e-25
```

```
[ ] sigmoid(0.5)
```

tanh

```
[ ] def tanh(x):
      return (math.exp(x) - math.exp(-x)) / (math.exp(x) + math.exp(-x))
```

```
[ ] tanh(-56)
```

```
[ ] tanh(50)
```

```
[ ] tanh(1)
```

ReLU

```
[ ] def relu(x):
        return max(0,x)
```

```
[ ] relu(-100)
```

```
[ ] relu(8)
```

Leaky ReLU

```
[ ] def leaky_relu(x):
        return max(0.1*x,x)
```

```
[ ] leaky_relu(-100)
```

```
[ ] leaky_relu(8)
```

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.

Weights and biases are crucial for the generalization of neural networks. Generalization refers to a network's ability to make accurate predictions on new, unseen data. By adjusting weights and biases during training, the network learns to capture meaningful patterns in the training data without fitting noise. This allows the network to generalize its knowledge and make accurate predictions on diverse datasets beyond the training set.

Backward Propagation is the process of moving from right (output layer) to left (input layer). Forward propagation is the way data moves from left (input layer) to right (output layer) in the neural network.

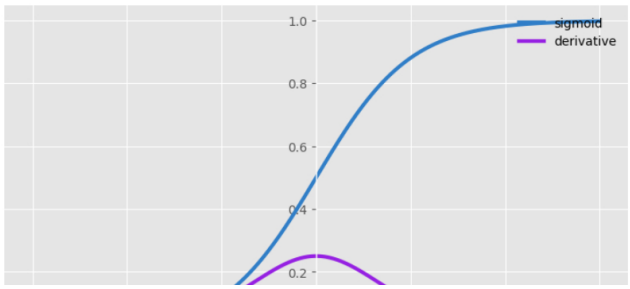
A neural network can be understood by a collection of connected input/output nodes.

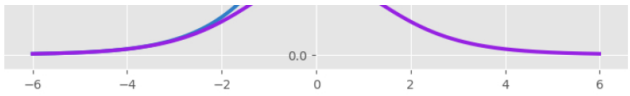
Sigmoid or Logistic Activation Function

The Sigmoid Function curve looks like a S-shape.

The main reason why we use sigmoid function is because it exists between (0 to 1). Therefore, it is especially 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.

```
[ ] import matplotlib.pyplot as plt
    import numpy as np
    def sigmoid(x):
        s=1/(1+np.exp(-x))
        ds=s*(1-s)
        return s,ds
    x=np.arange(-6,6,0.01)
    sigmoid(x)
    fig, ax = plt.subplots(figsize=(9, 5))
    ax.spines['left'].set_position('center')
    ax.spines['right'].set_color('none')
    ax.spines['top'].set_color('none')
    ax.xaxis.set_ticks_position('bottom')
    ax.yaxis.set_ticks_position('left')
    ax.plot(x,sigmoid(x)[0], color="#307EC7", linewidth=3, label="sigmoid")
    ax.plot(x,sigmoid(x)[1], color="#9621E2", linewidth=3, label="derivative")
    ax.legend(loc="upper right", frameon=False)
    fig.show()
```





Observations:

The sigmoid function has values between 0 to 1.

The output is not zero-centered.

Sigmoids saturate and kill gradients.

Type of Function	Pros	Cons
Linear	– Provides a range of activations, not binary. – Can connect multiple neurons and make decisions based on max activation. – Constant gradient for stable descent. – Changes are constant for error correction.	– Limited modeling capacity due to linearity.
Sigmoid	– Nonlinear, allowing complex combinations. – Produces analog activation, not binary.	– Suffers from the "vanishing gradients" problem, making it slow to learn.
Tanh	– Stronger gradient compared to sigmoid. – Addresses the vanishing gradient issue to some extent.	– Still prone to vanishing gradient problems.
ReLU	– Solves the vanishing gradient problem. – Computationally efficient.	– Can lead to "Dead Neurons" due to fragile gradients. Should be used only in hidden layers.
Leaky ReLU	– Mitigates the "dying ReLU" problem with a small negative slope.	– Lacks complexity for some classification tasks.
ELU	– Can produce negative outputs for x>0.	– May lead to unbounded activations, resulting in a wide output range.

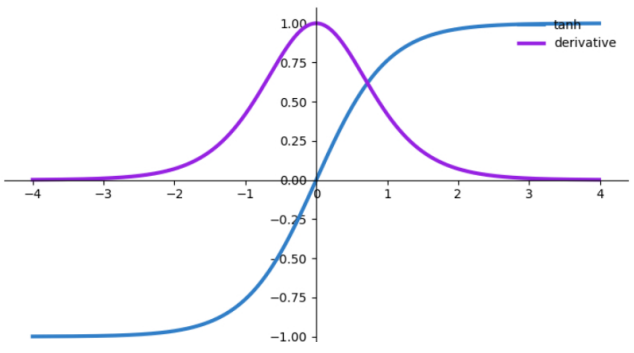
At the top and bottom level of sigmoid functions, the curve changes slowly, the derivative curve above shows that the slope or gradient it is zero.

Tanh Activation Function

Tanh or hyperbolic tangent Activation Function

tanh is also like logistic sigmoid but better. The range of the tanh function is from (-1 to 1). tanh is also sigmoidal (s - shaped).

```
[ ] import matplotlib.pyplot as plt
    import numpy as np
    def tanh(x):
        t=(np.exp(x)-np.exp(-x))/(np.exp(x)+np.exp(-x))
        dt=1-t**2
        return t,dt
    z=np.arange(-4,4,0.01)
    tanh(z)[0].size,tanh(z)[1].size
    fig, ax = plt.subplots(figsize=(9, 5))
    ax.spines['left'].set_position('center')
    ax.spines['bottom'].set_position('center')
    ax.spines['right'].set_color('none')
    ax.spines['top'].set_color('none')
    ax.xaxis.set_ticks_position('bottom')
    ax.yaxis.set_ticks_position('left')
    ax.plot(z,tanh(z)[0], color="#307EC7", linewidth=3, label="tanh")
    ax.plot(z,tanh(z)[1], color="#9621E2", linewidth=3, label="derivative")
    ax.legend(loc="upper right", frameon=False)
    fig.show()
```



Observations:

Its output is zero-centered because its range is between -1 to 1. i.e. -1 < output < 1.

Optimization is easier in this method hence in practice it is always preferred over the Sigmoid function.

Double-click (or enter) to edit

Pros and Cons of Activation Functions

Type of Function	Pros	Cons
Linear	– Provides a range of activations, not binary. – Can connect multiple neurons and make decisions based on max activation. – Constant gradient for stable descent. – Changes are constant for error correction.	– Limited modeling capacity due to linearity.
Sigmoid	– Nonlinear, allowing complex combinations. – Produces analog activation, not binary. – Stronger gradient compared to sigmoid.	– Suffers from the "vanishing gradients" problem, making it slow to learn.
Tanh	– Addresses the vanishing gradient issue to some extent.	– Still prone to vanishing gradient problems.
ReLU	– Solves the vanishing gradient problem. – Computationally efficient.	– Can lead to "Dead Neurons" due to fragile gradients. Should be used only in hidden layers.
Leaky ReLU	– Mitigates the "dying ReLU" problem with a small negative slope.	– Lacks complexity for some classification tasks.
ELU	– Can produce negative outputs for x>0.	– May lead to unbounded activations, resulting in a wide output range.

Double-click (or enter) to edit

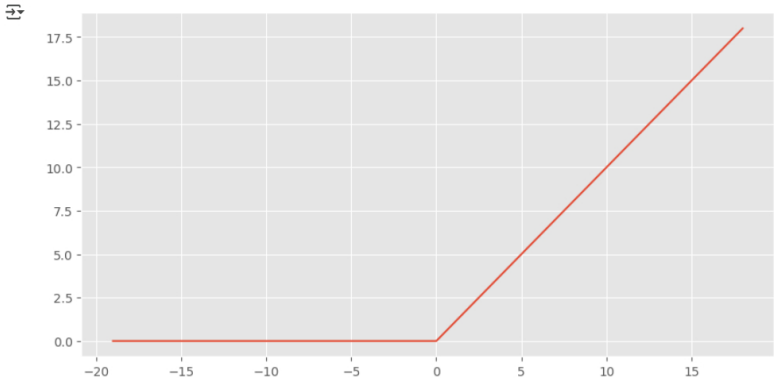
What is ReLU Activation Function?

ReLU stands for rectified linear activation unit and is considered one of the few milestones in the deep learning revolution. It is simple yet really better than its predecessor activation functions such as sigmoid or tanh.

```
[ ] Start coding or generate with AI.
```

```
[ ] def ReLU(x):
      if x>0:
        return x
      else:
        return 0
```

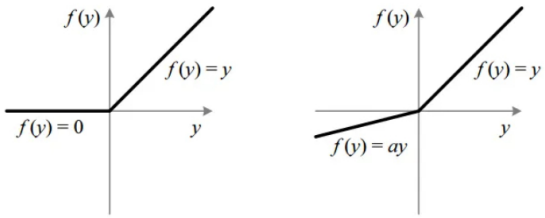
```
[ ] from matplotlib import pyplot
    pyplot.style.use('ggplot')
    pyplot.figure(figsize=(10,5))
    # define a series of inputs
    input_series = [x for x in range(-19, 19)]
    # calculate outputs for our inputs
    output_series = [ReLU(x) for x in input_series]
    # line plot of raw inputs to rectified outputs
    pyplot.plot(input_series, output_series)
    pyplot.show()
```



Leaky ReLU activation function

Leaky ReLU function is an improved version of the ReLU activation function. As for the ReLU activation function, the gradient is 0 for all the values of inputs that are less than zero, which would deactivate the neurons in that region and may cause dying ReLU problem.

Leaky ReLU is defined to address this problem. Instead of defining the ReLU activation function as 0 for negative values of inputs(x), we define it as an extremely small linear component of x. Here is the formula for this activation function



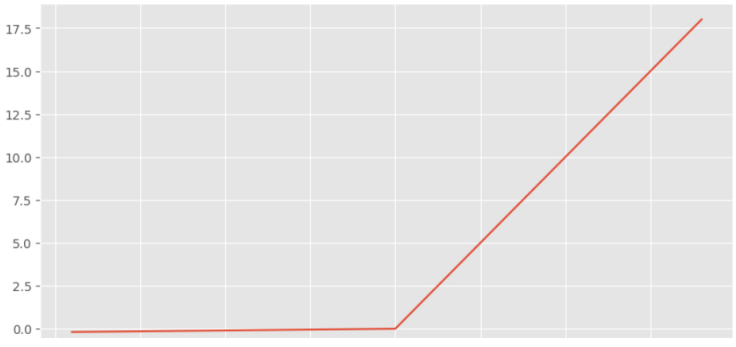
Double-click (or enter) to edit

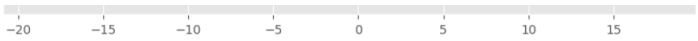
f(x)=max(0.01*x , x).

This function returns x if it receives any positive input, but for any negative value of x, it returns a really small value which is 0.01 times x. Thus it gives an output for negative values as well. By making this small modification, the gradient of the left side of the graph comes out to be a non zero value. Hence we would no longer encounter dead neurons in that region.

Now like we did for ReLU activation function, we will give values to Leaky ReLU activation functions and plot them.

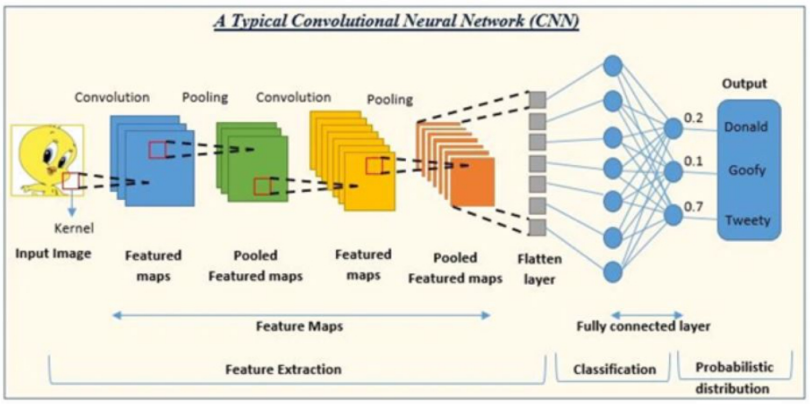
```
[ ] from matplotlib import pyplot
    pyplot.style.use('ggplot')
    pyplot.figure(figsize=(10,5))

    # rectified linear function
    def Leaky_ReLU(x):
      if x>0:
        return x
      else:
        return 0.01*x

    # define a series of inputs
    input_series = [x for x in range(-19, 19)]
    # calculate outputs for our inputs
    output_series = [Leaky_ReLU(x) for x in input_series]
    # line plot of raw inputs to rectified outputs
    pyplot.plot(input_series, output_series)
    pyplot.show()
```





If you look carefully at the plot, you would find that the negative values are not zero and there is a slight slope to the line on the left side of the plot.

CNN architecture



Key Components of CNN Architecture

1. Input Layer: The input layer accepts raw pixel values from an image, typically represented as a three-dimensional tensor (height, width, depth). For instance, a color image with dimensions 100×100 pixels would have a shape of

100×100×3

100×100×3 (where 3 represents the RGB color channels).

2. Convolutional Layers: These layers are the core building blocks of CNNs. They apply convolution operations using filters (also known as kernels) that slide over the input image to detect features such as edges, textures, and shapes.

Each filter produces a feature map that highlights specific patterns in the input data. The convolution operation involves computing the dot product between the filter and a local region of the input image.

3. Activation Function: After convolution, an activation function is applied to introduce non-linearity into the model. The Rectified Linear Unit (ReLU) is commonly used due to its effectiveness in accelerating convergence during training.

4. Pooling Layers: Pooling layers reduce the spatial dimensions of feature maps while retaining essential information. This downsampling process helps decrease computational load and mitigate overfitting.

Common pooling techniques include Max Pooling (selecting the maximum value from a region) and Average Pooling (calculating the average value).

5. Fully Connected Layers: Towards the end of the network, fully connected layers are used to combine features learned from previous layers and make final predictions. Each neuron in these layers is connected to every neuron in the preceding layer.

6. Output Layer: The output layer generates predictions based on the learned features. For classification tasks, this layer typically employs a softmax activation function to produce probabilities for each class.

How CNNs Work

The operation of CNNs can be summarized in several key steps:

1. Convolution Operation: Filters slide over the input image to produce feature maps that capture various aspects of the image.

2. Activation: Non-linear activation functions like ReLU are applied to introduce complexity into the model.

3. Pooling: Spatial dimensions are reduced through pooling layers, which help retain important

features while simplifying computations.

4. Hierarchical Feature Learning: As data passes through multiple layers, CNNs learn increasingly complex features—from simple edges in early layers to intricate shapes and objects in deeper layers.

5. Classification: The final fully connected layer processes these features to classify or predict outcomes based on learned patterns.

Advantages of Convolutional Neural Networks

1. Automatic Feature Extraction: Unlike traditional machine learning methods that require manual feature engineering, CNNs automatically learn relevant features from raw data.

2. Translation Invariance: CNNs can recognize objects regardless of their position in an image due to their hierarchical structure and pooling operations.

3. Reduced Computational Complexity: By using shared weights in convolutional layers, CNNs significantly reduce the number of parameters compared to fully connected networks.

4. Scalability: CNN architectures can be scaled up with additional layers and filters to improve performance on complex tasks.

Challenges and Limitations

Despite their advantages, CNNs face several challenges:

1. Data Requirements: Training CNNs requires large labeled datasets to achieve high accuracy; insufficient data can lead to overfitting.

2. Computational Resources: Training deep CNNs can be computationally intensive and may require specialized hardware such as GPUs.

3. Interpretability: Understanding how CNNs make decisions can be challenging due to their complex architectures, leading to concerns about transparency in critical applications like healthcare or autonomous vehicles.

```
#Applications of Convolutional Neural Networks
##CNNs have found applications across various
domains:

##1. Image Classification: Used extensively in
identifying objects within images for applications
ranging from social media tagging to medical
imaging diagnostics.

##2. Object Detection: Techniques like YOLO (You
Only Look Once) utilize CNNs for real-time object
detection in videos and images.

##3. Facial Recognition: Employed in security
systems for identifying individuals based on
facial features.

##4. Autonomous Vehicles: Used for recognizing
road signs, pedestrians, and other vehicles in
real-time for navigation purposes.

##5. Medical Imaging Analysis: Applied in
diagnosing diseases by analyzing X-rays, MRIs, and
CT scans with high accuracy.
```

Applications of Convolutional Neural Networks

CNNs have found applications across various domains:

1. Image Classification: Used extensively in identifying objects within images for applications ranging from social media tagging to medical imaging diagnostics.

2. Object Detection: Techniques like YOLO (You Only Look Once) utilize CNNs for real-time object detection in videos and images.

3. Facial Recognition: Employed in security systems for identifying individuals based on facial features.

4. Autonomous Vehicles: Used for recognizing road signs, pedestrians, and other vehicles in real-time for navigation purposes.

5. Medical Imaging Analysis: Applied in diagnosing diseases by analyzing X-rays, MRIs, and CT scans with high accuracy.

```
[ ] import tensorflow as tf
    from tensorflow.keras.models import Sequential
    from tensorflow.keras.layers import Conv2D, MaxPooling2D, Flatten, Dense, Dropout
    from tensorflow.keras.preprocessing.image import ImageDataGenerator
    import matplotlib.pyplot as plt

    # Define Image Size and Directories
```

```
IMG_SIZE = (128, 128)
BATCH_SIZE = 32

# Data Augmentation and Preprocessing
datagen = ImageDataGenerator(rescale=1./255, validation_split=0.2)

train_data = datagen.flow_from_directory(
    'dataset_path',  # Change to actual path
    target_size=IMG_SIZE,
    batch_size=BATCH_SIZE,
    class_mode='binary',
    subset='training'
)

val_data = datagen.flow_from_directory(
    'dataset_path',
    target_size=IMG_SIZE,
    batch_size=BATCH_SIZE,
    class_mode='binary',
    subset='validation'
)

# Build CNN Model
model = Sequential([
    Conv2D(32, (3, 3), activation='relu', input_shape=(128, 128, 3)),
    MaxPooling2D(2, 2),

    Conv2D(64, (3, 3), activation='relu'),
    MaxPooling2D(2, 2),

    Conv2D(128, (3, 3), activation='relu'),
    MaxPooling2D(2, 2),

    Flatten(),
    Dense(128, activation='relu'),
    Dropout(0.5),
    Dense(1, activation='sigmoid')  # Binary Classification
])

# Compile the Model
model.compile(optimizer='adam', loss='binary_crossentropy', metrics=['accuracy'])

# Train the Model
history = model.fit(train_data, validation_data=val_data, epochs=10)

# Plot Training Results
plt.plot(history.history['accuracy'], label='Train Accuracy')
plt.plot(history.history['val_accuracy'], label='Validation Accuracy')
plt.xlabel('Epochs')
plt.ylabel('Accuracy')
plt.legend()
plt.show()
```