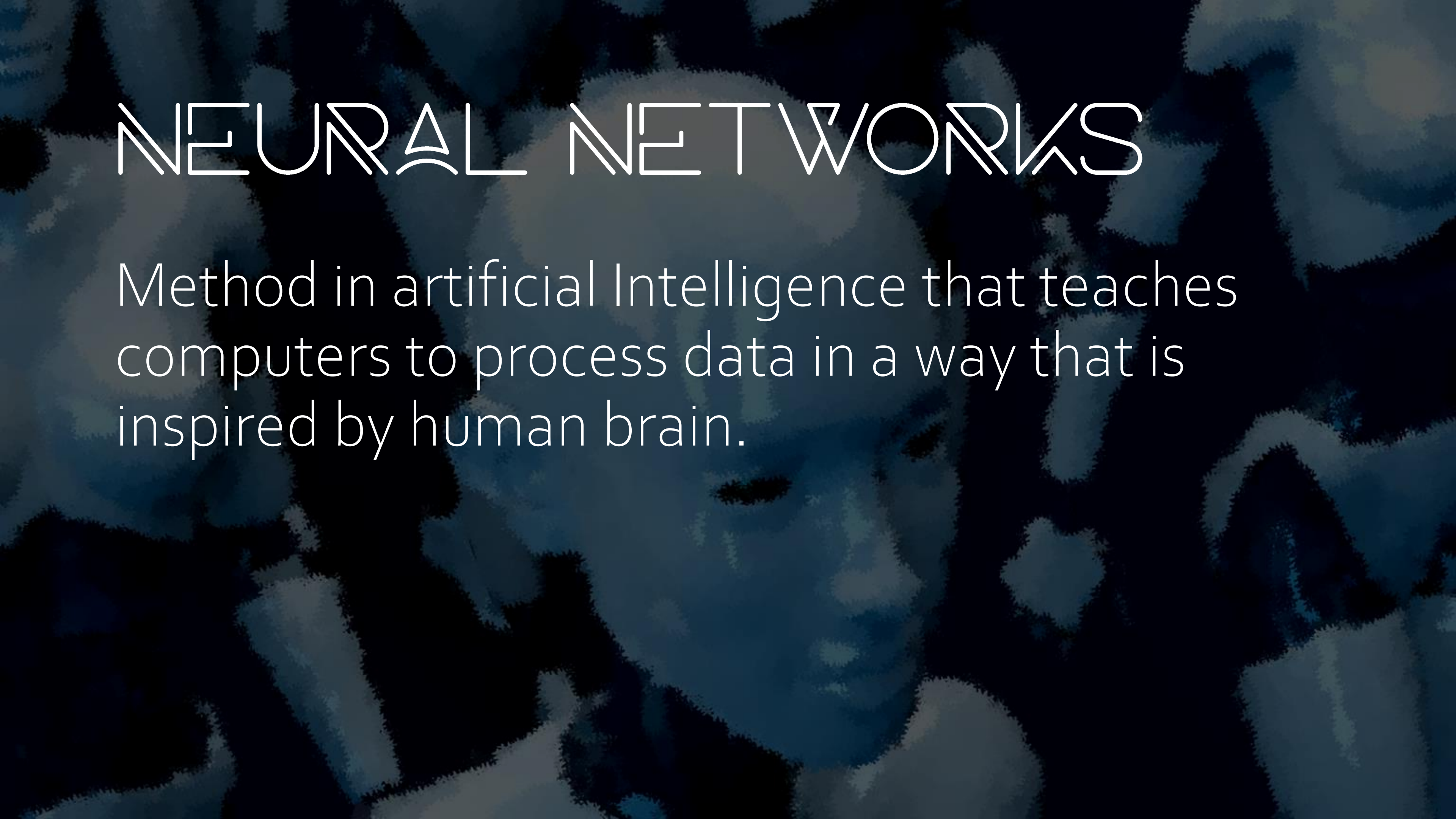
Group Members

- 07. Parth Zarekar
- 10. Krishnaraj Thadesar
- 24. Singh Soubhagya
- 25. Sourab Karad

NEURAL NETWORKS

Method in artificial Intelligence that teaches computers to process data in a way that is inspired by human brain.



COMPONENTS OF NEURAL NETWORK

Input Layer

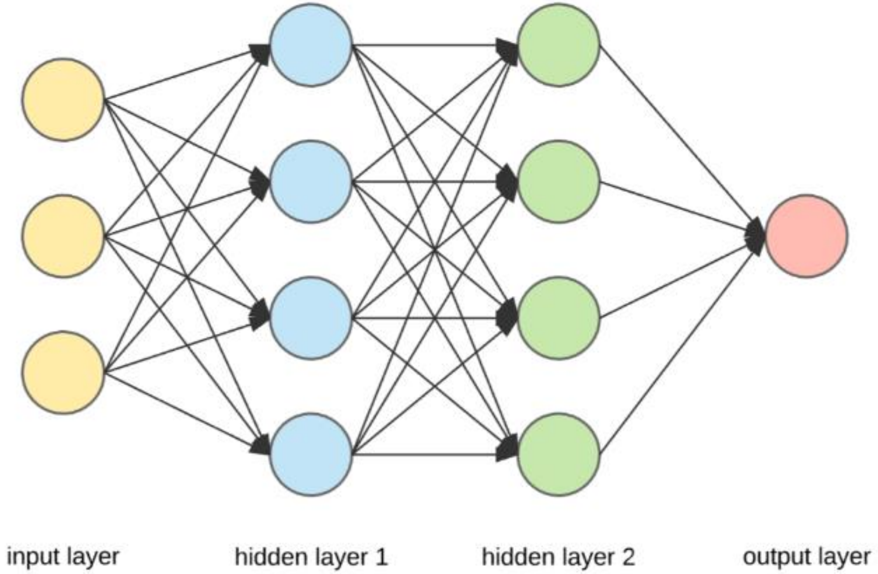
Imagine this as eyes and ears of the network.

Hidden Layers

Hidden layers are where most of the thinking happens

Output Layer

The output Layer produces the final results of the networks calculations



Neurons (Nodes)

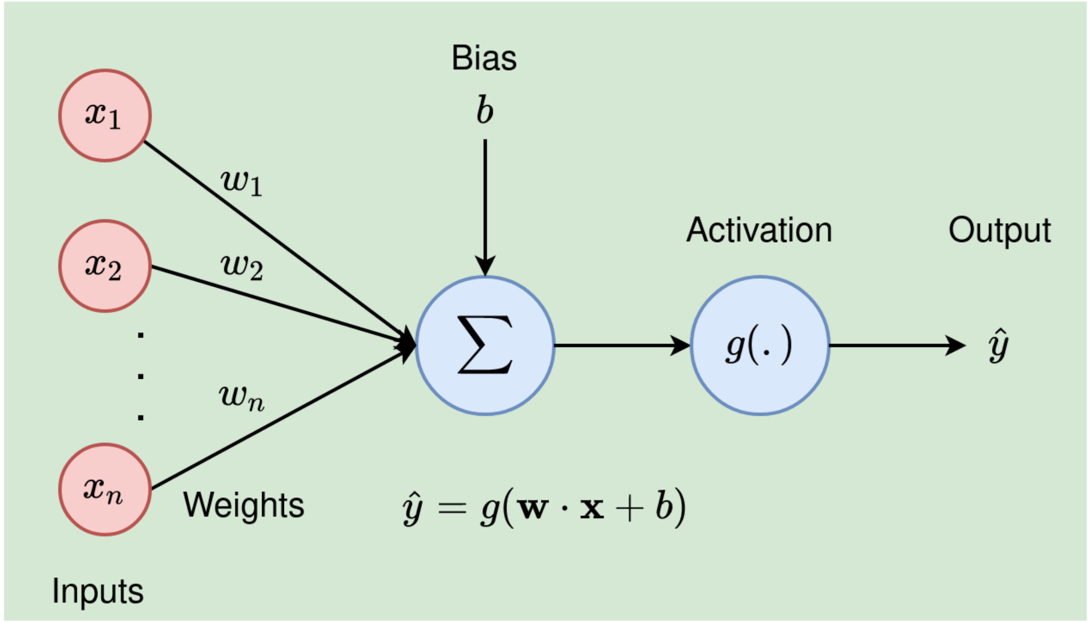
Neurons information from input layer or other neuron and make calculations.

Weights

Neurons assign importance to the information they receive through weights

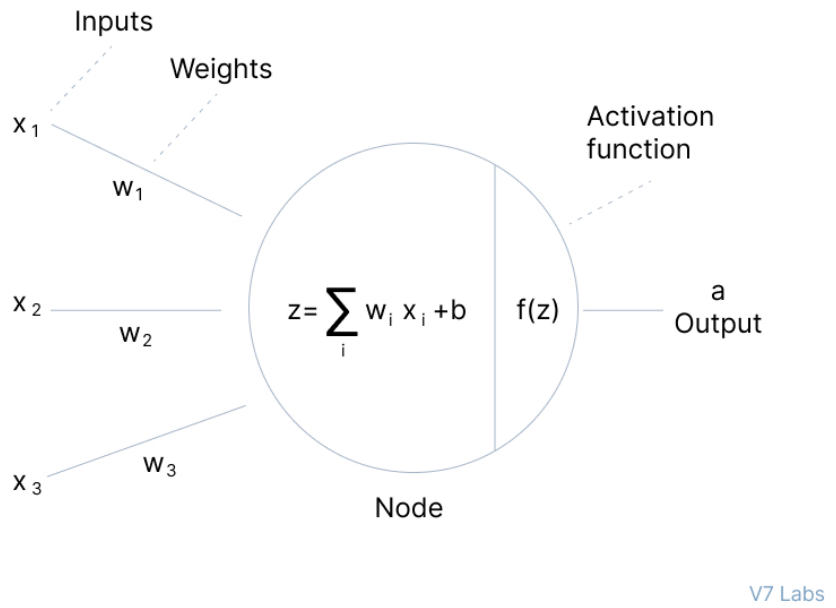
Bias

Biases act like a starting point for each neuron's calculation.



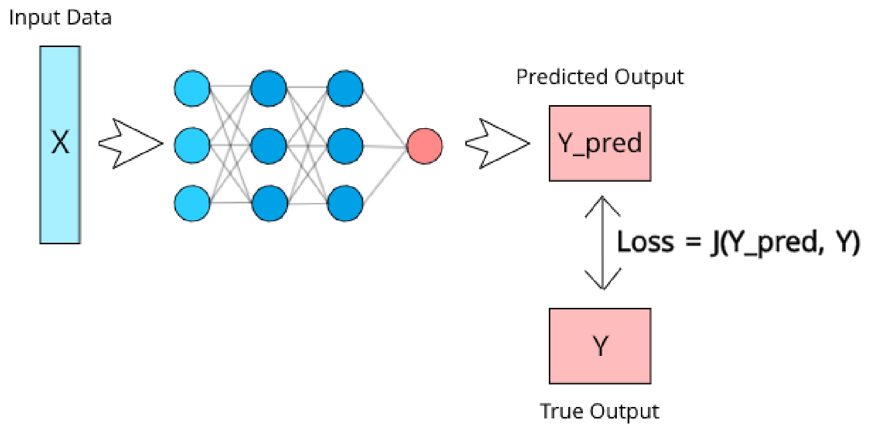
ACTIVATION FUNCTION

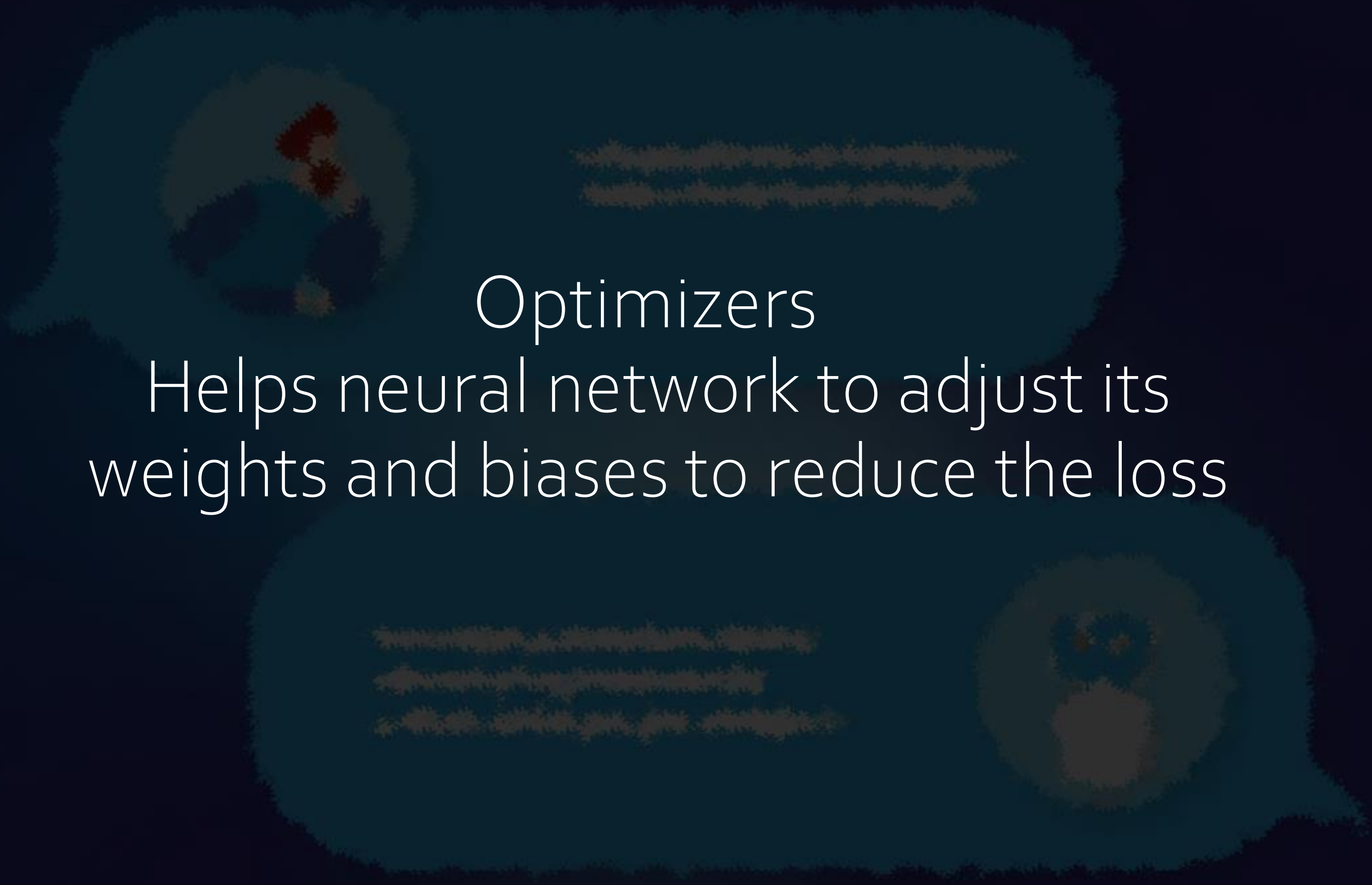
Decides when a neuron should "fire" or activate.



LOSS FUNCTION

It calculates how far networks calculated output is from actual desired output.





Optimizers

Helps neural network to adjust its weights and biases to reduce the loss

BACK PROPAGATION

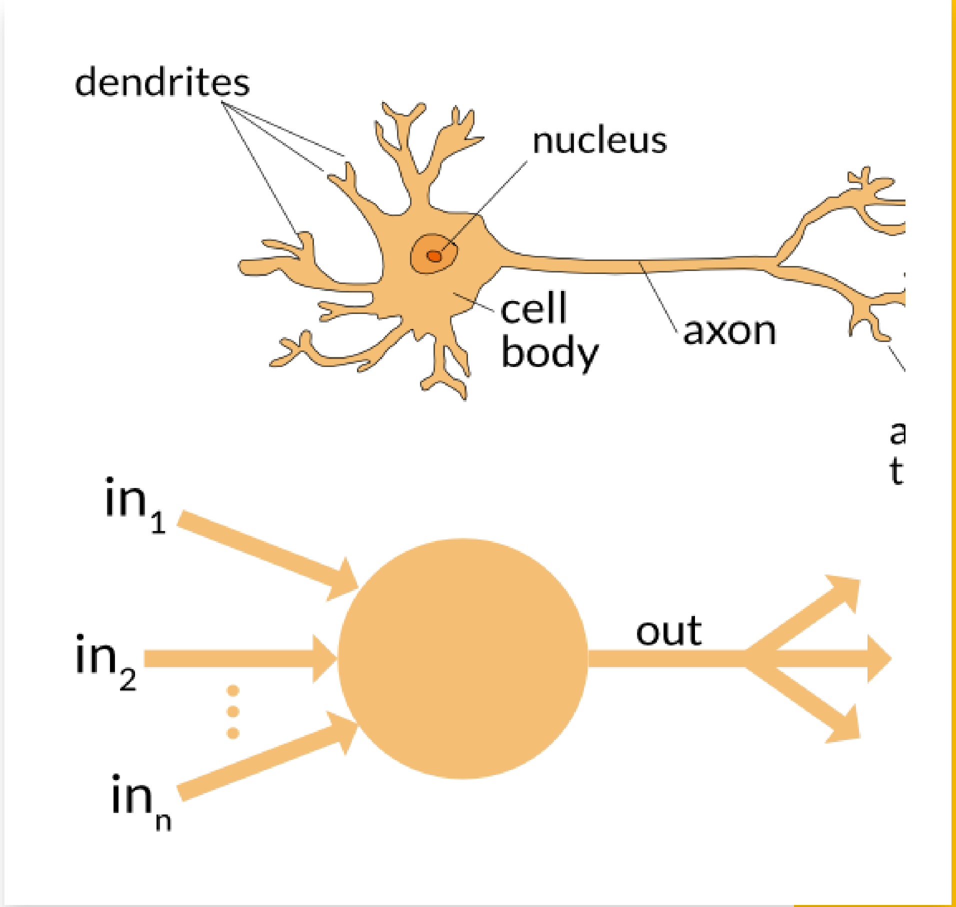
It tells network how it should adjust its internal settings to perform better in future

WHAT IS A PERCEPTRON

A perceptron is a fundamental building block in artificial neural networks and serves as a simple model of a biological neuron. It's a binary linear classifier that can be used for various machine learning tasks, such as binary classification.

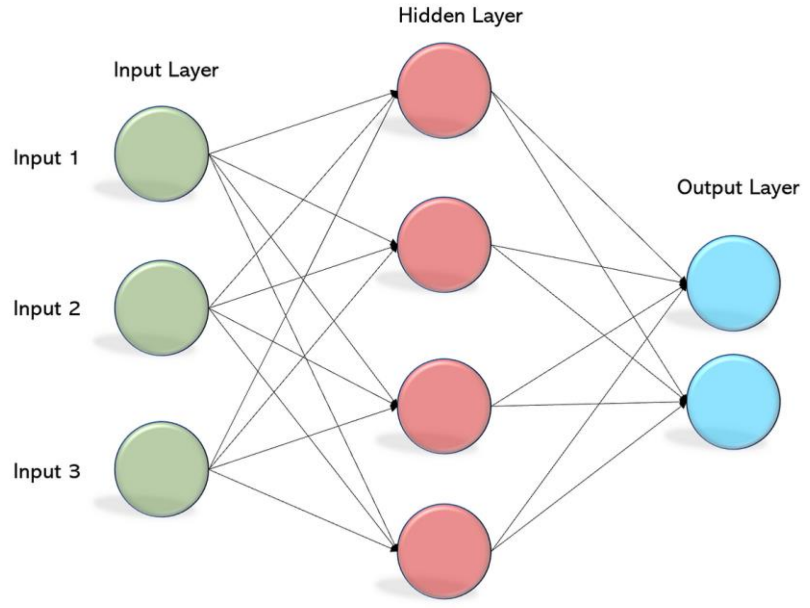
COMPARISON TO A NEURON

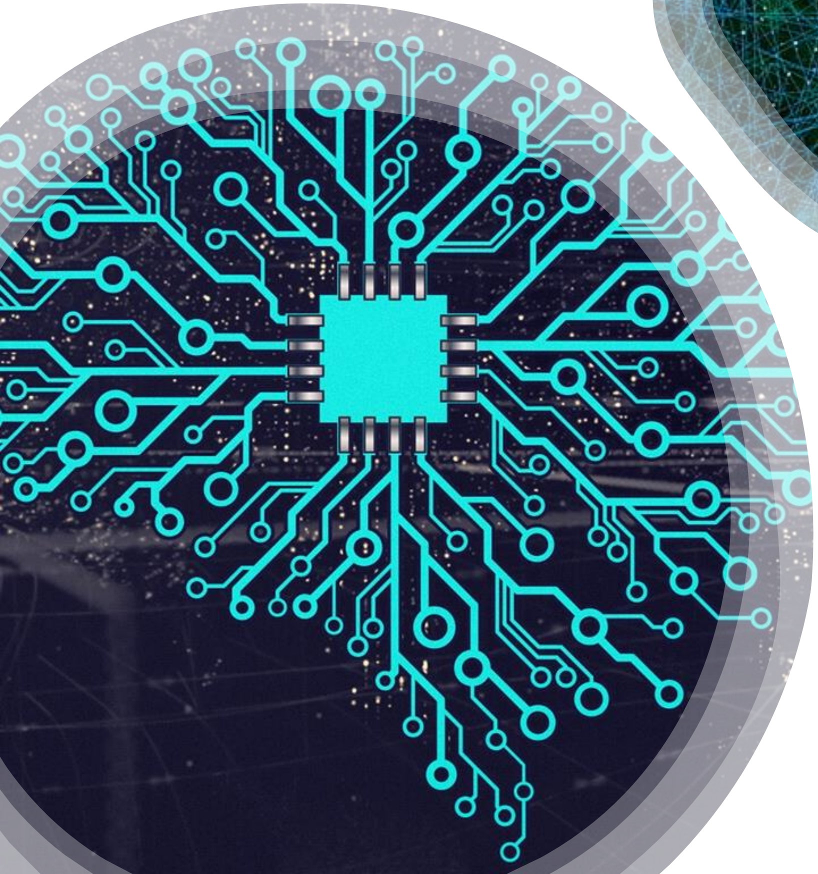
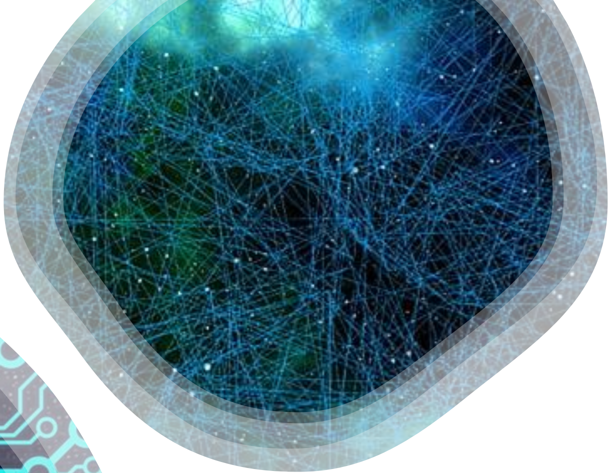
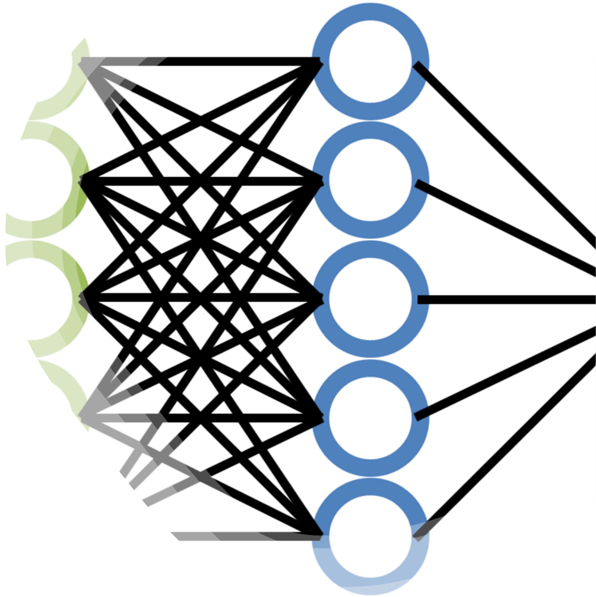
It is a basic unit of an artificial neural network that takes a set of input values, processes them, and produces an output.



MULTILAYER PERCEPTRON

- Multilayer Perceptrons (MLPs), also known as feedforward neural networks or artificial neural networks, are a type of artificial neural network with multiple layers of interconnected neurons.
- These networks consist of an input layer, one or more hidden layers, and an output layer. Multilayer Perceptrons are designed to handle more complex tasks than single-layer perceptrons, as they can capture non-linear patterns in data.



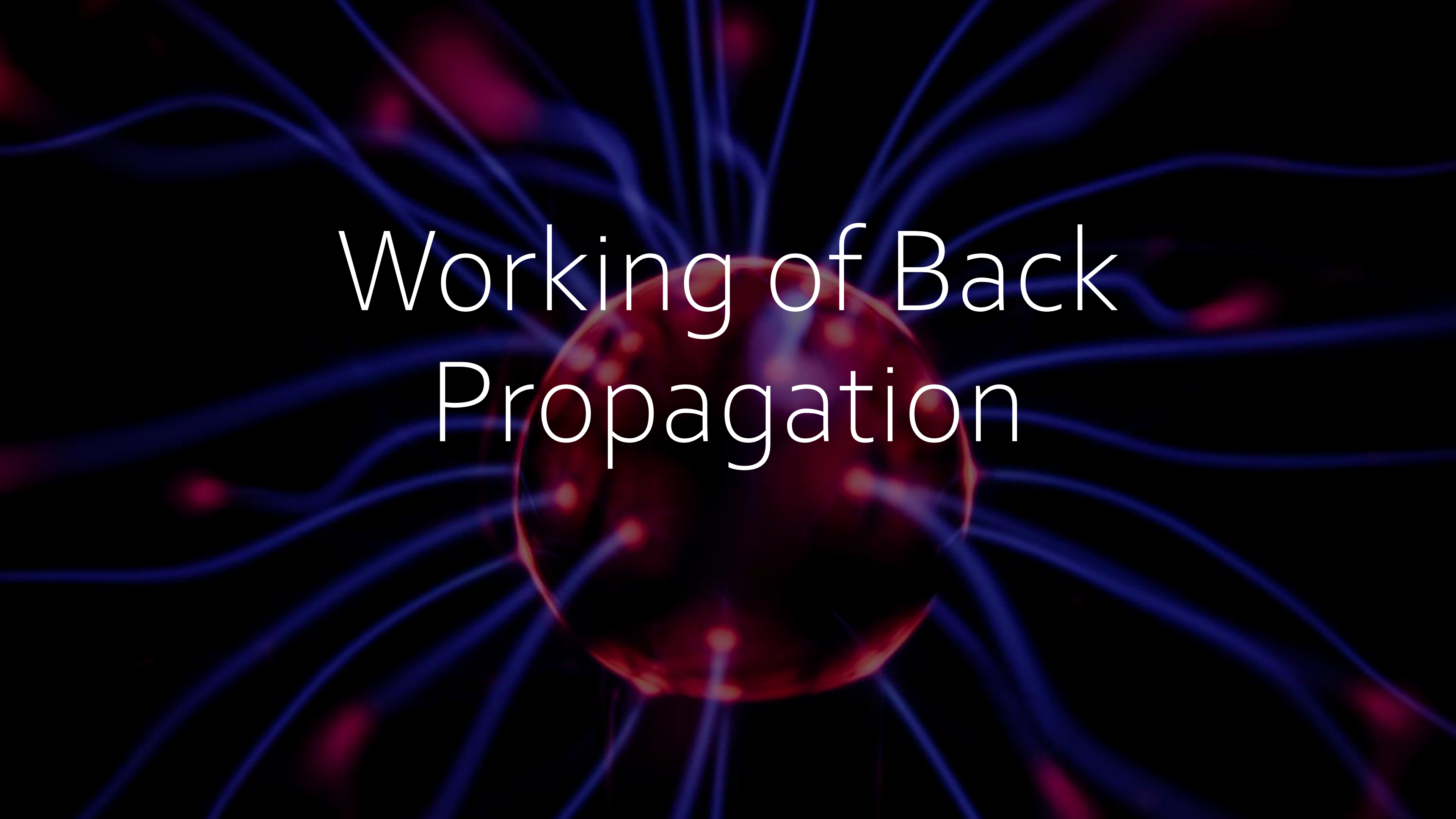
Back Propagation: Unveiling the Inner Workings of Neural Networks

24. Saubhagya Singh

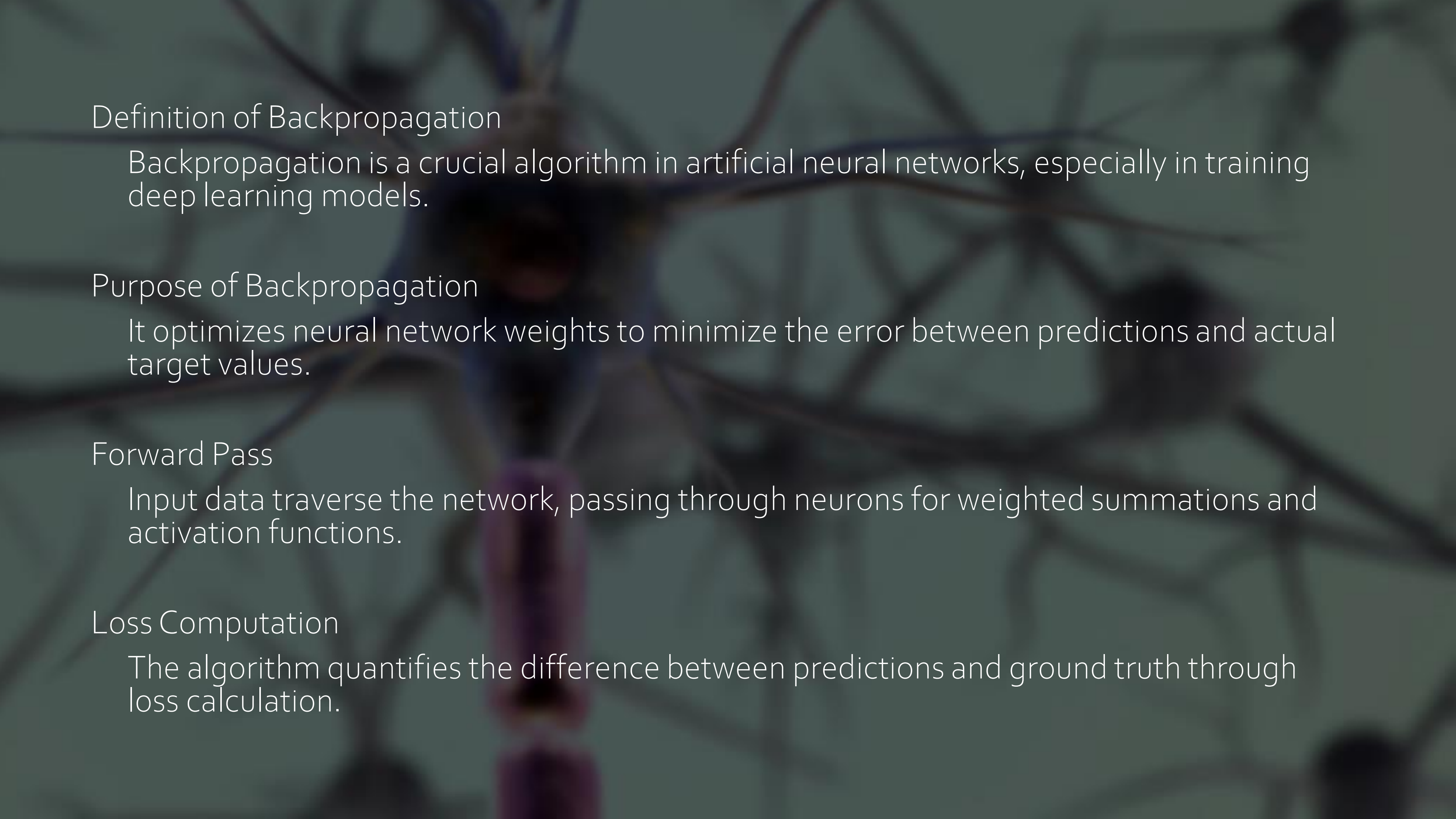
UNDERSTANDING BACK PROPAGATION

Back propagation is a key algorithm for training neural networks. It enables the network to learn from labeled training data and adjust its internal parameters to minimize the error between predicted and actual outputs.

The algorithm works by propagating the error backwards through the network, updating the weights and biases of each neuron based on their contribution to the overall error. This iterative process continues until the network achieves satisfactory performance.



Working of Back Propagation



Definition of Backpropagation

Backpropagation is a crucial algorithm in artificial neural networks, especially in training deep learning models.

Purpose of Backpropagation

It optimizes neural network weights to minimize the error between predictions and actual target values.

Forward Pass

Input data traverse the network, passing through neurons for weighted summations and activation functions.

Loss Computation

The algorithm quantifies the difference between predictions and ground truth through loss calculation.

Error Propagation

Backpropagation propagates error from output to input layer, computing gradients of loss with respect to each neuron's weights.

Guidance for Weight Adjustments

These gradients serve as guidance for adjusting neuron weights to minimize the loss function.

Gradient Descent Optimization

Weight adjustments are made using gradient descent algorithms to iteratively minimize the loss.

Iterative Process

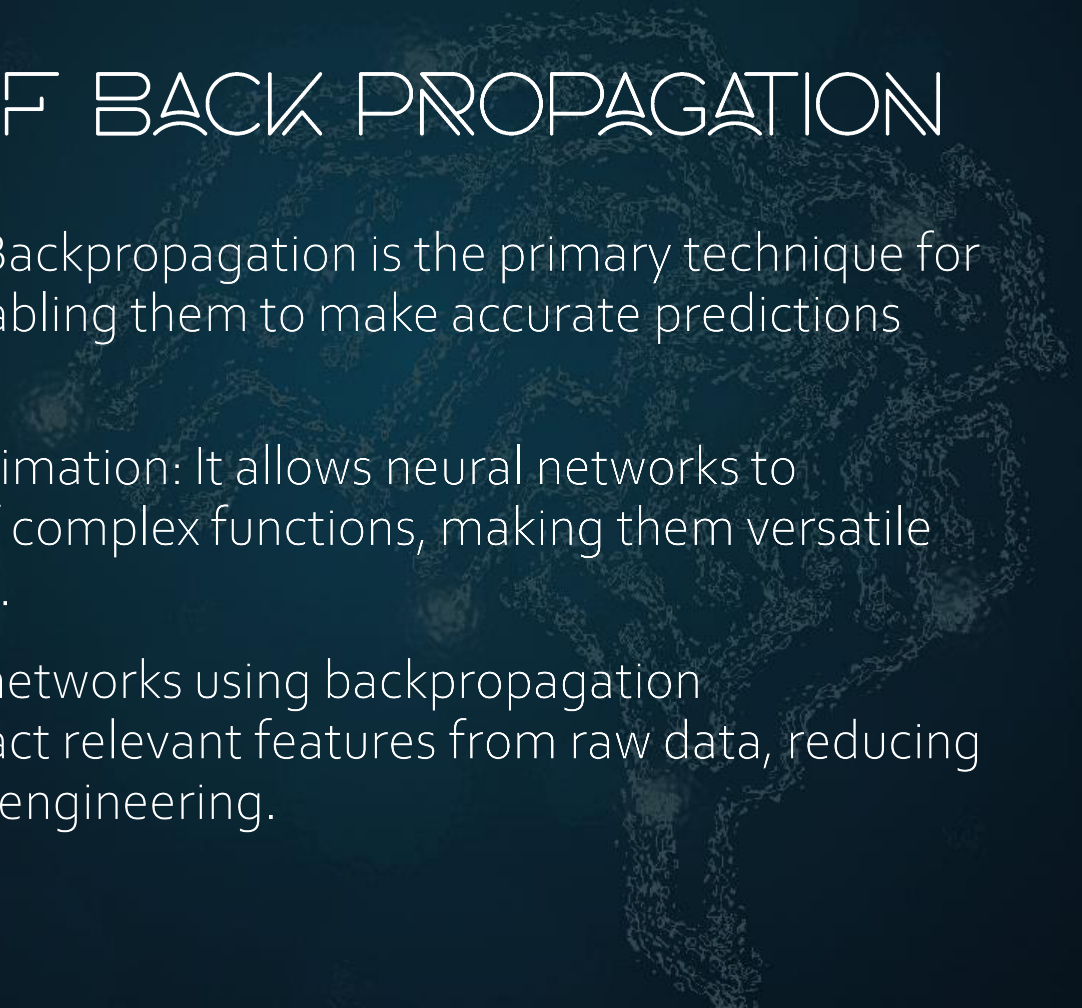
This process continues until the model's predictions closely align with the target values, enabling accurate predictions.

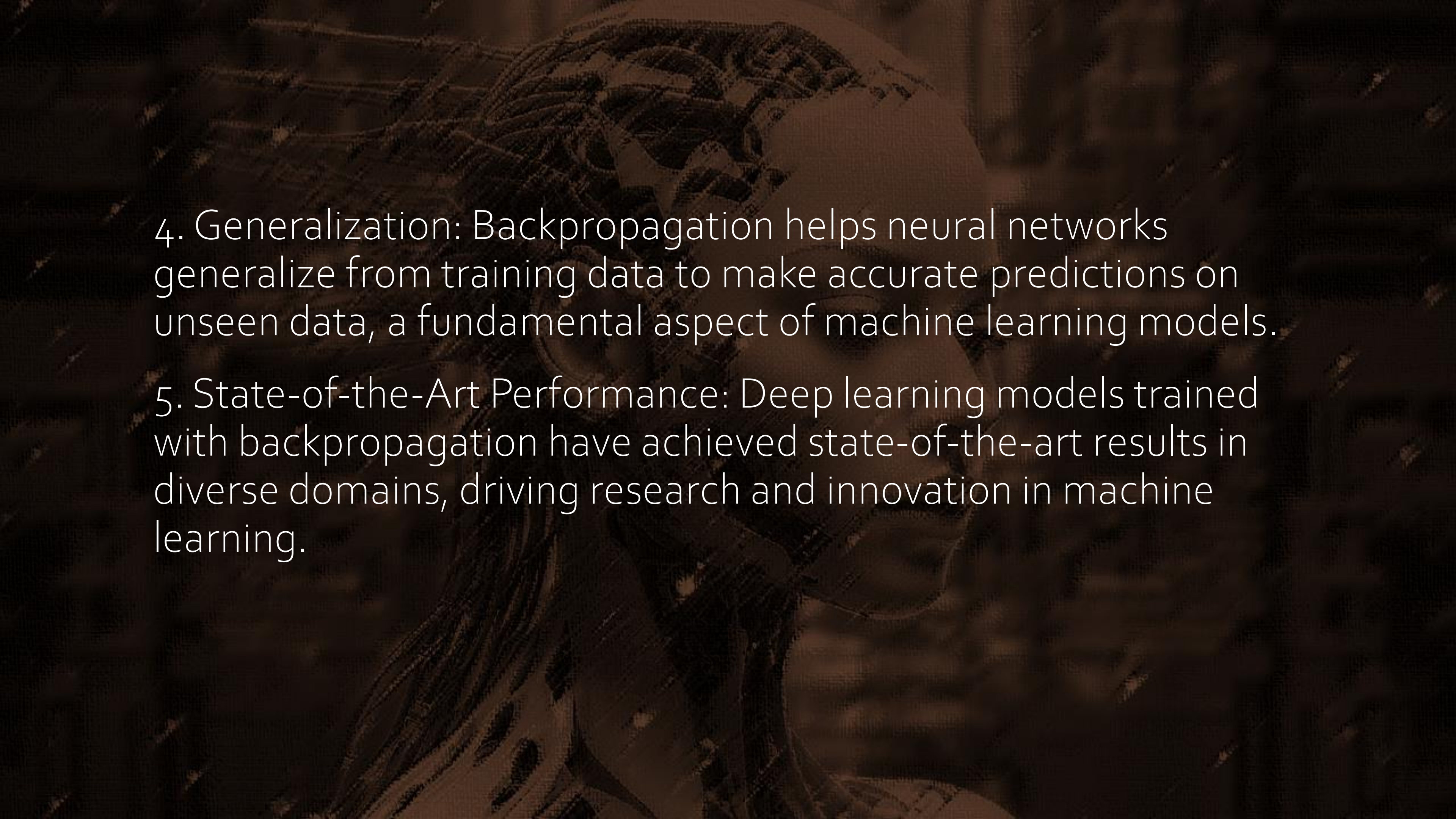
IMPORTANCE OF BACK PROPAGATION

1. Neural Network Training: Backpropagation is the primary technique for training neural networks, enabling them to make accurate predictions and solve complex tasks.

2. Universal Function Approximation: It allows neural networks to approximate a wide range of complex functions, making them versatile tools for various applications.

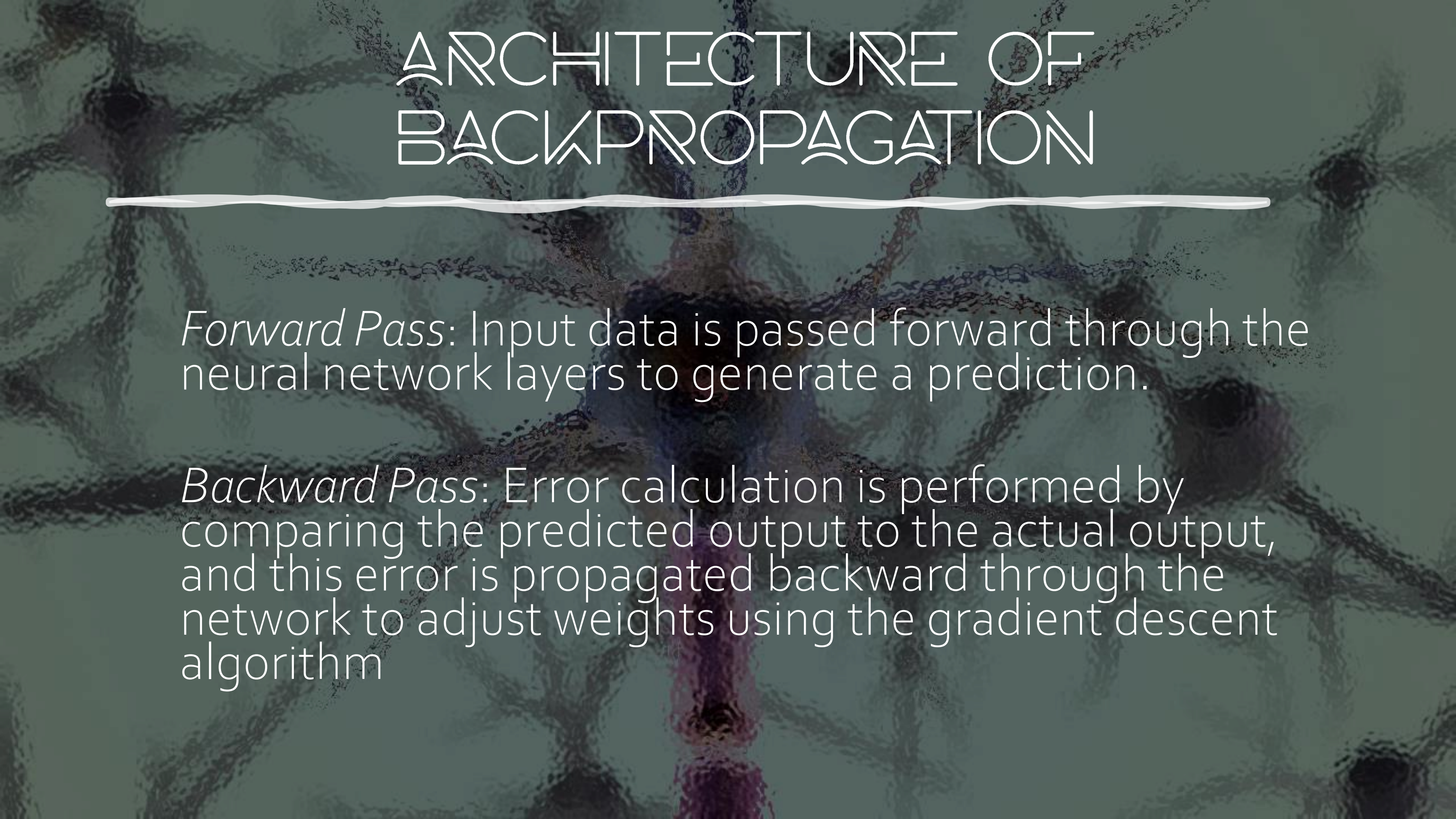
3. Feature Learning: Neural networks using backpropagation automatically learn and extract relevant features from raw data, reducing the need for manual feature engineering.





4. Generalization: Backpropagation helps neural networks generalize from training data to make accurate predictions on unseen data, a fundamental aspect of machine learning models.

5. State-of-the-Art Performance: Deep learning models trained with backpropagation have achieved state-of-the-art results in diverse domains, driving research and innovation in machine learning.



ARCHITECTURE OF BACKPROPAGATION

Forward Pass: Input data is passed forward through the neural network layers to generate a prediction.

Backward Pass: Error calculation is performed by comparing the predicted output to the actual output, and this error is propagated backward through the network to adjust weights using the gradient descent algorithm

GRADIENT DESCENT

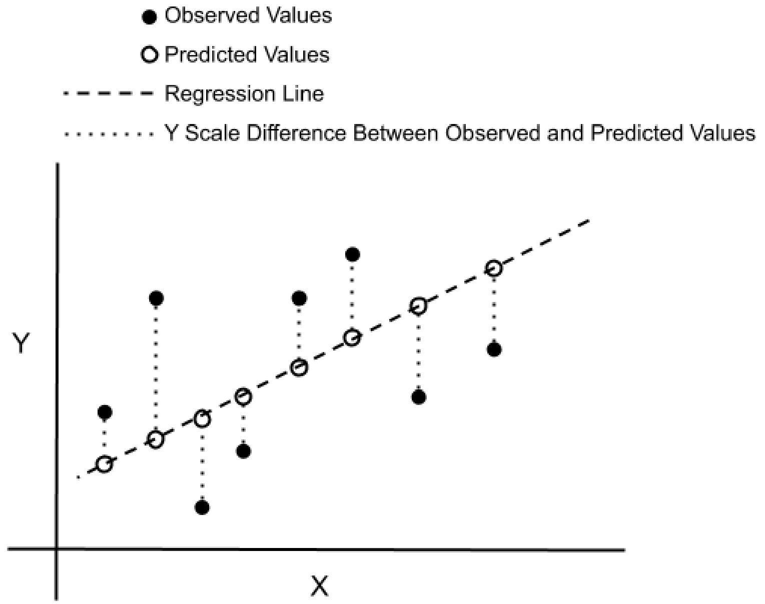
Optimization technique used in backpropagation to minimize the error by adjusting weights. It calculates the gradient of the cost function with respect to each weight.

ACTIVATION FUNCTIONS

Functions used at each neuron to introduce non-linearity. Common activation functions include ReLU, sigmoid, and tanh.

BACKPROPAGATION KEY COMPONENTS

Error calculation, chain rule application, weight updates using calculated gradients, learning rate adjustment.



Cost Functions

$$\text{MSE} = \boxed{\frac{1}{n}\sum_{i=1}^{n}}\boxed{(Y_i - \hat{Y}_i)}\boxed{^2}$$

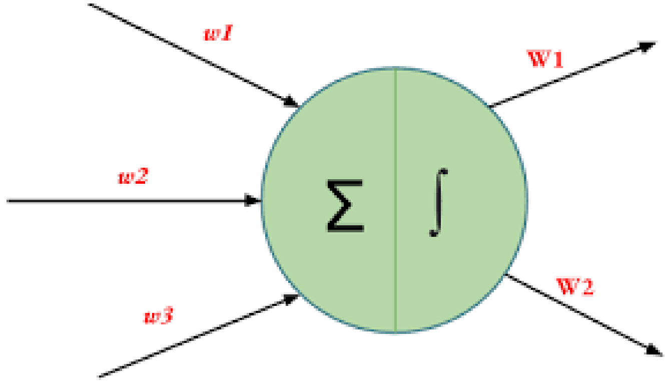
- Mean Squared Error (MSE): Measures the average of the squared differences between predictions and actual values.
- Cross-Entropy Loss: Commonly used in classification problems. It measures the performance of a classification model whose output is a probability value between 0 and 1

ROLE OF COST FUNCTIONS

Cost functions help quantify the error between predicted and actual values. The goal is to minimize this error during training by adjusting network weights.

WEIGHT INITIALIZATION

1. Random Initialization: Weights are initialized randomly to break symmetry among neurons.
2. Random Normal: The weights are initialized from values in a normal distribution.
3. Random Uniform: The weights are initialized from values in a uniform distribution.



$$w_i \sim U\left[-\sqrt{\frac{\sigma}{fan\_in + fan\_out}}, \sqrt{\frac{\sigma}{fan\_in + fan\_out}}\right]$$

XAVIER GLOROT INITIALIZATION

- Sets initial weights based on the number of input and output neurons of the layer.
- Xavier/Glorot Initialization often termed as Xavier Uniform Initialization, is suitable for layers where the activation function used is Sigmoid.
- In Xavier/Glorot weight initialization, the weights are assigned from values of a uniform distribution as follows:

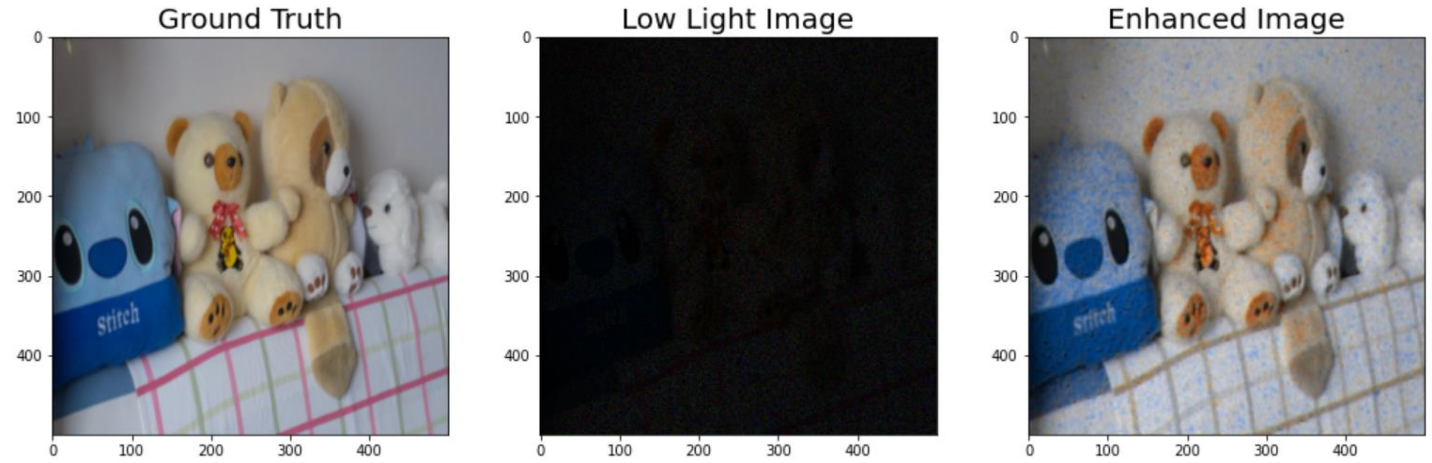
$$w_i \sim U[-\sqrt{\frac{6}{fan\_in}}, \sqrt{\frac{6}{fan\_out}}]$$

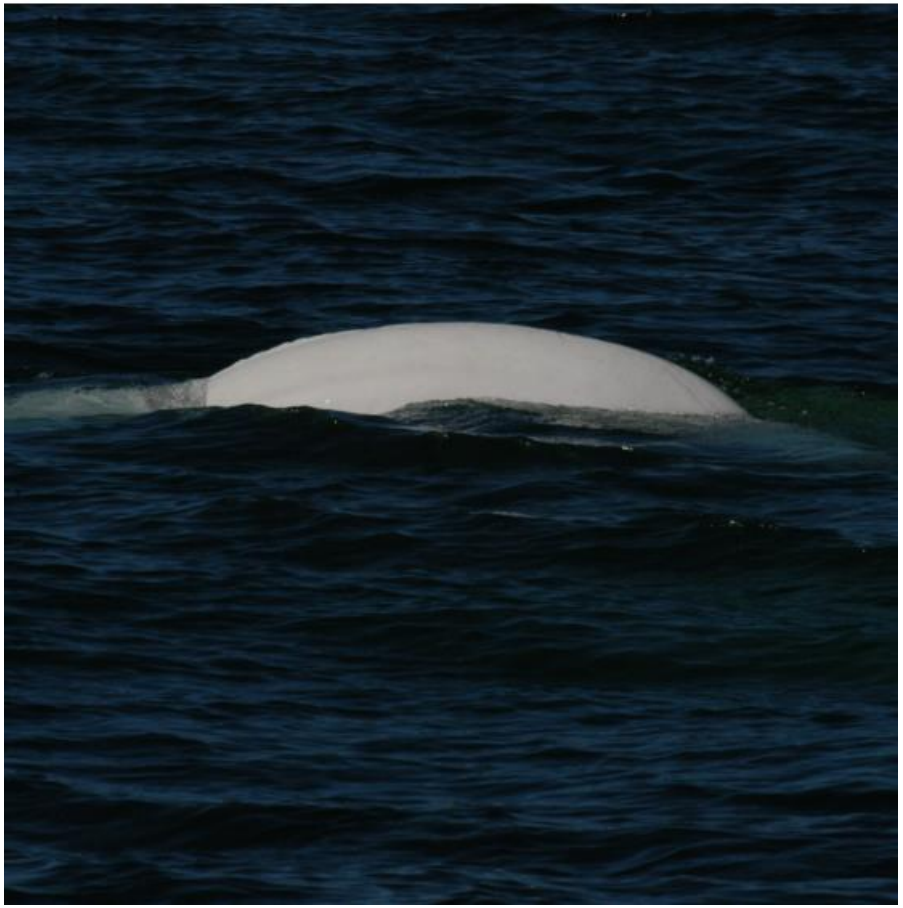
HE INITIALIZATION

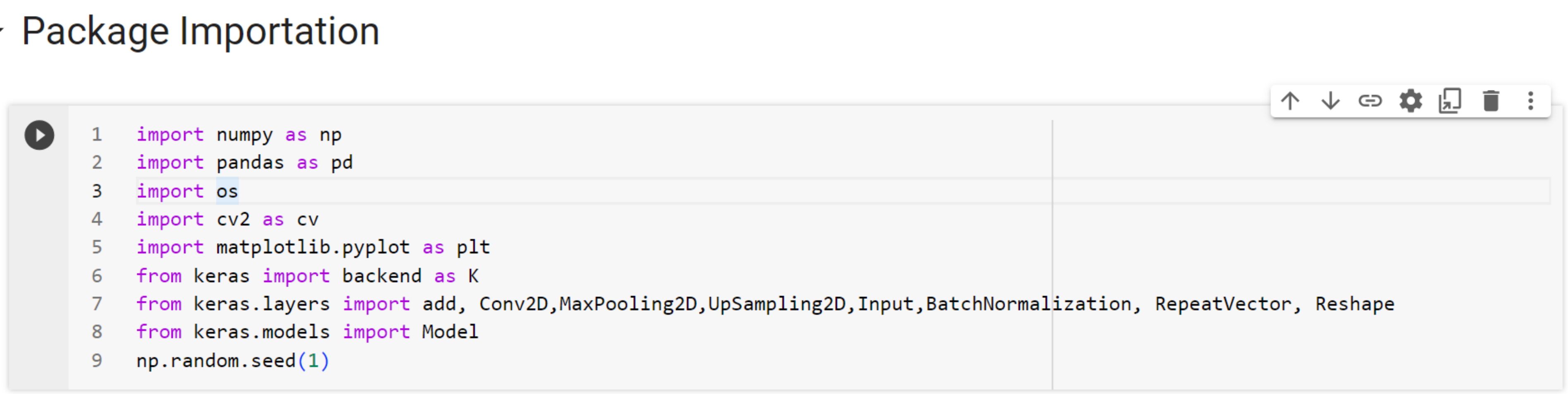
- Similar to Xavier, but taking into account the ReLU (rectified linear) activation function.
- In the Uniform weight initialization, the weights are assigned from values of a uniform distribution as follo
- He Uniform Initialization is suitable for layers where ReLU activation function is used

IMPORTANCE OF WEIGHT INITIALIZATION

Proper weight initialization helps in convergence speed, prevents vanishing/exploding gradients, and aids in better training of neural networks.

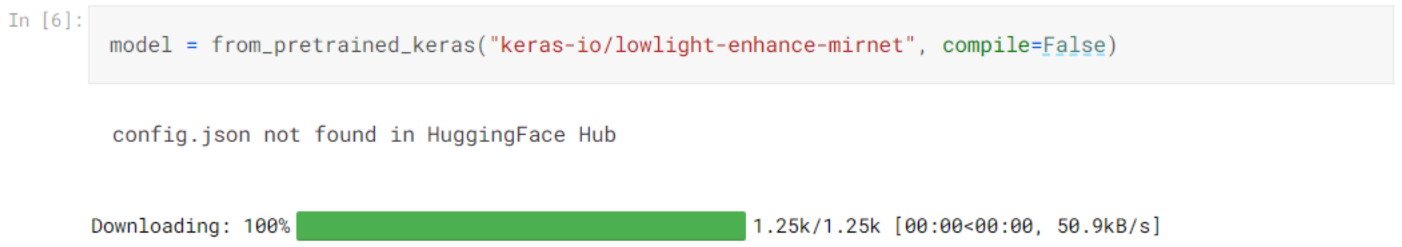

Ground Truth
Low Light Image
Enhanced Image
0
100
200
300
400





Package Importation

```python
import numpy as np
import pandas as pd
import os
import cv2 as cv
import matplotlib.pyplot as plt
from keras import backend as K
from keras.layers import add, Conv2D,MaxPooling2D,UpSampling2D,Input,BatchNormalization, RepeatVector, Reshape
from keras.models import Model
np.random.seed(1)
```



```
In [6]:
model = from_pretrained_keras("keras-io/lowlight-enhance-mirnet", compile=False)
```

```
config.json not found in HuggingFace Hub

Downloading: 100% 1.25k/1.25k [00:00<00:00, 50.9kB/s]
```

In [12]:

```
image = keras.preprocessing.image.img_to_array(low_light_img)
```

In [13]:

```
image.shape
```

Out[13]:

```
(256, 256, 3)
```

In [14]:

```
image = image.astype('float32') / 255.0
```

In [15]:

```
image.shape
```

Out[15]:

```
(256, 256, 3)
```

In [18]:
```
output = model.predict(image) # model inference to enhance the low light pics
```

```
2022-04-22 18:23:00.545896: I tensorflow/compiler/mlir/mlir_graph_optimization_pass.cc:185] None
of the MLIR Optimization Passes are enabled (registered 2)
```

In [19]:
```
output_image = output[0] * 255.0
```

In [20]:
```
output_image.shape
```

Out[20]:
```
(256, 256, 3)
```

In [21]:
```
output_image = output_image.clip(0,255)
```

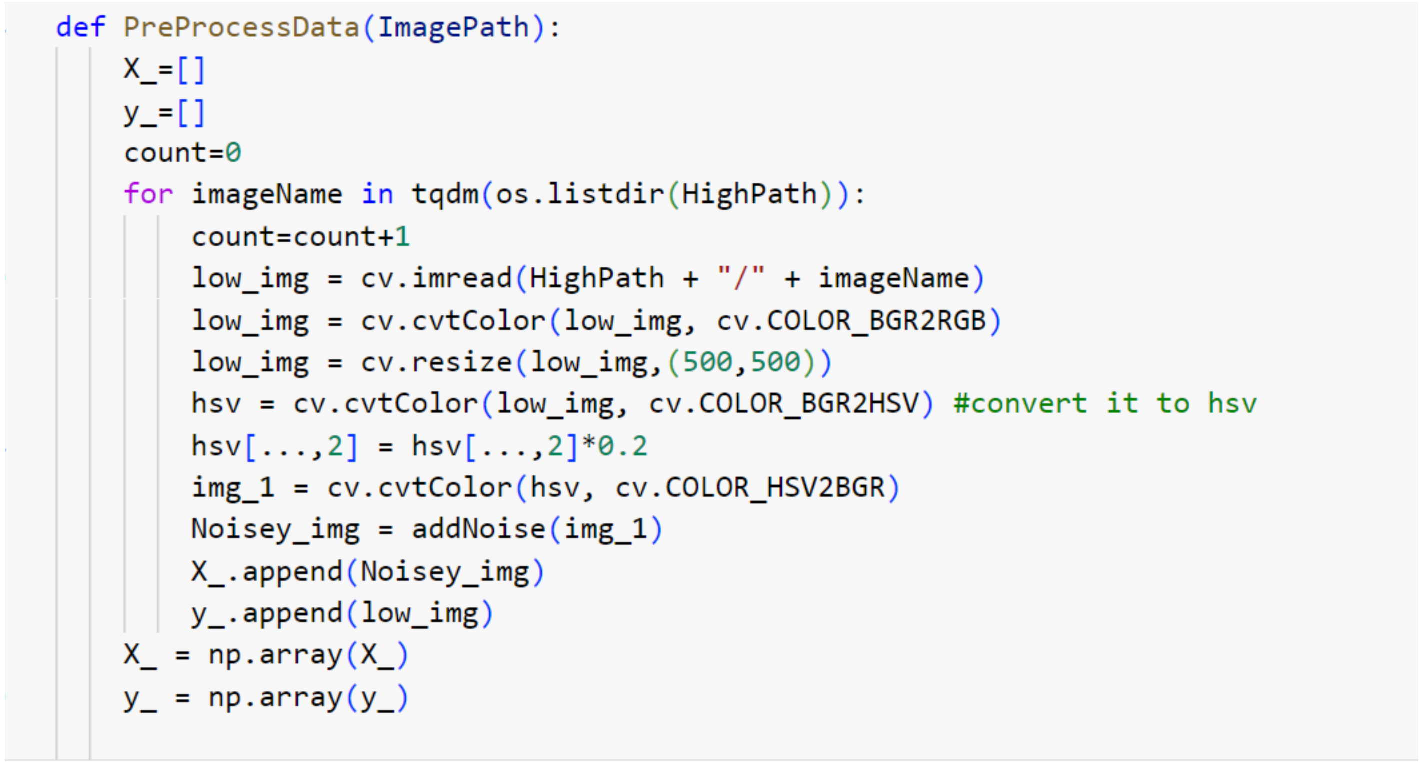
```
In [25]: output_image
```

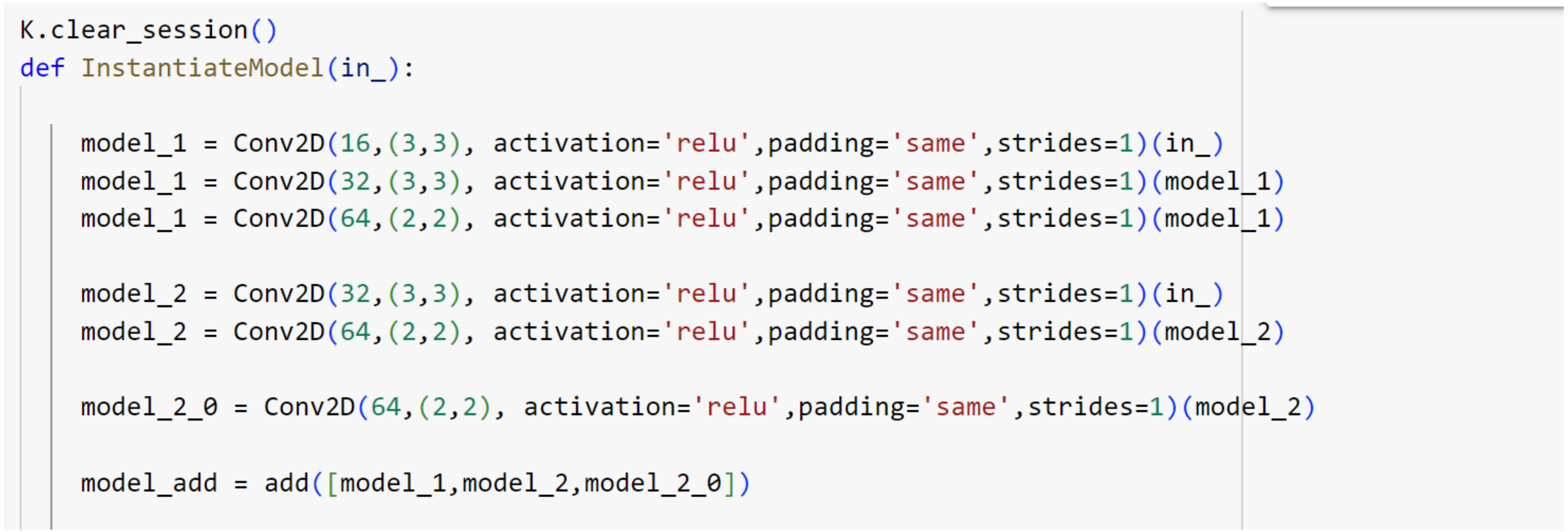
```
Out[25]:
array([[[ 75.927765,  90.44433 , 100.36248 ],
        [ 98.26136 , 116.37938 , 143.26581 ],
        [112.696045, 151.7278  , 177.84164 ],
        ...,
        [ 89.94365 , 114.33969 , 135.48743 ],
        [ 91.46919 , 107.76184 , 123.99914 ],
        [ 84.028366, 100.88614 , 123.06505 ]],

       [[ 96.2401  ,  87.456924, 106.73138 ],
        [ 96.12944 ,  95.80934 , 124.13488 ],
        [ 73.13187 , 101.668564, 136.89339 ],
        ...,
        [ 83.40862 , 126.74953 , 133.8411  ],
        [ 83.54708 , 122.20536 , 133.4005  ]
```

Final Image

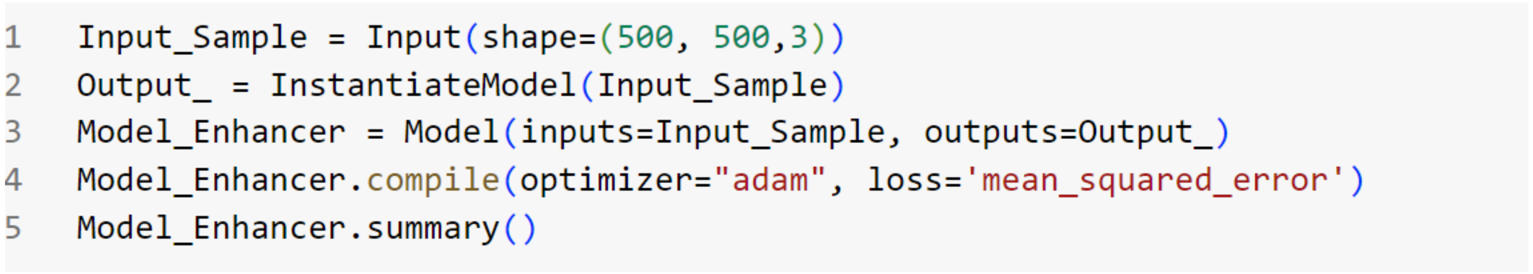
In [28]:
```
Image.fromarray(output_image.astype('uint8'),'RGB')
```

Out[28]:

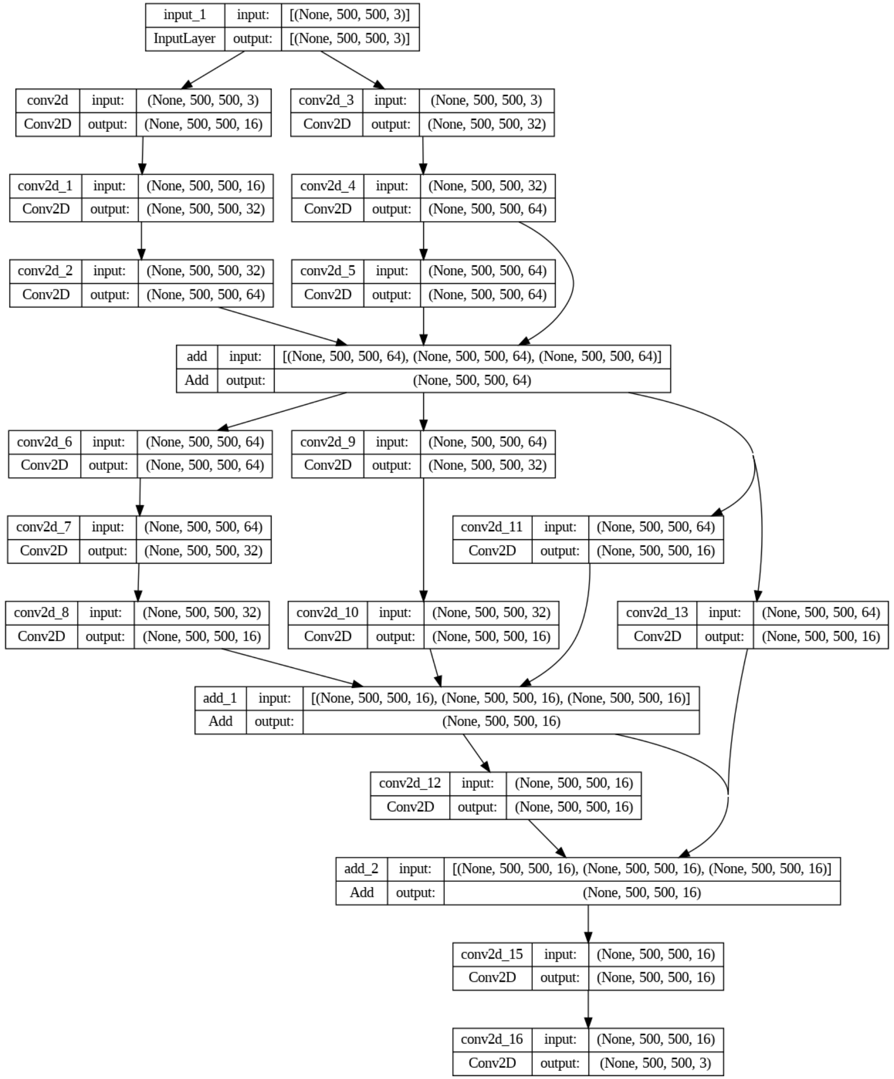

```
def PreProcessData(ImagePath):
    X_=[]
    y_=[]
    count=0
    for imageName in tqdm(os.listdir(HighPath)):
        count=count+1
        low_img = cv.imread(HighPath + "/" + imageName)
        low_img = cv.cvtColor(low_img, cv.COLOR_BGR2RGB)
        low_img = cv.resize(low_img,(500,500))
        hsv = cv.cvtColor(low_img, cv.COLOR_BGR2HSV) #convert it to hsv
        hsv[...,2] = hsv[...,2]*0.2
        img_1 = cv.cvtColor(hsv, cv.COLOR_HSV2BGR)
        Noisey_img = addNoise(img_1)
        X_.append(Noisey_img)
        y_.append(low_img)
    X_ = np.array(X_)
    y_ = np.array(y_)
```

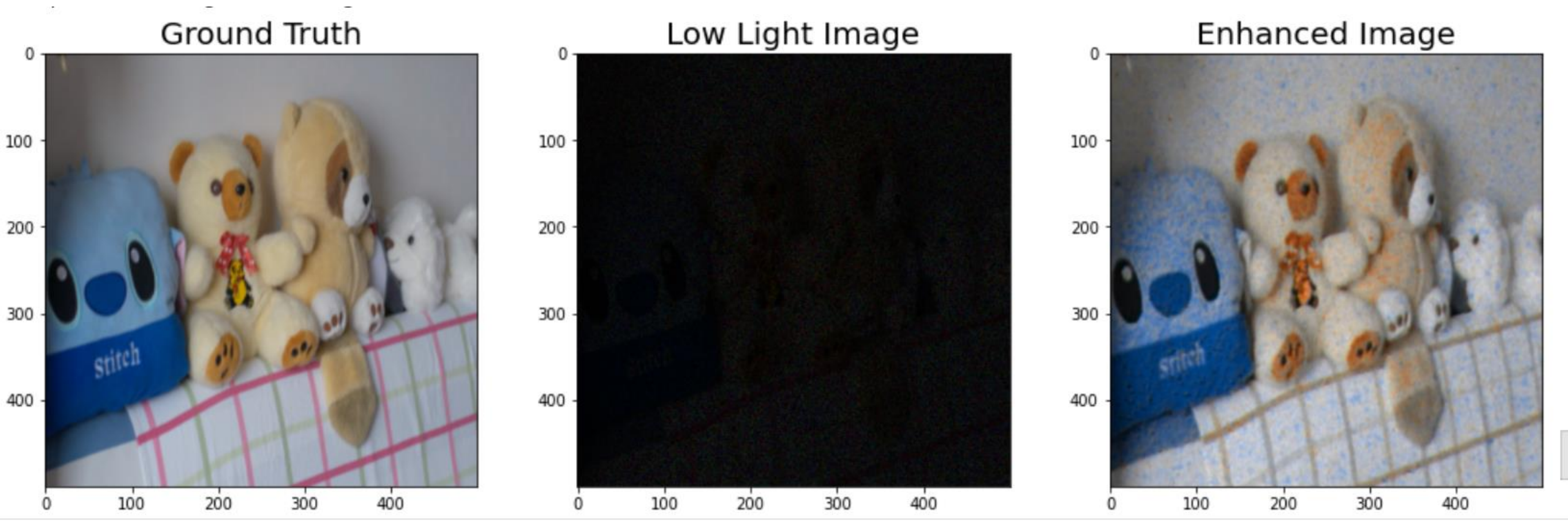


```python
K.clear_session()
def InstantiateModel(in_):

        model_1 = Conv2D(16,(3,3), activation='relu',padding='same',strides=1)(in_)
        model_1 = Conv2D(32,(3,3), activation='relu',padding='same',strides=1)(model_1)
        model_1 = Conv2D(64,(2,2), activation='relu',padding='same',strides=1)(model_1)

        model_2 = Conv2D(32,(3,3), activation='relu',padding='same',strides=1)(in_)
        model_2 = Conv2D(64,(2,2), activation='relu',padding='same',strides=1)(model_2)

        model_2_0 = Conv2D(64,(2,2), activation='relu',padding='same',strides=1)(model_2)

        model_add = add([model_1,model_2,model_2_0])
```



```python
Input_Sample = Input(shape=(500, 500,3))
Output_ = InstantiateModel(Input_Sample)
Model_Enhancer = Model(inputs=Input_Sample, outputs=Output_)
Model_Enhancer.compile(optimizer="adam", loss='mean_squared_error')
Model_Enhancer.summary()
```


input_1 InputLayer input: [(None, 500, 500, 3)] output: [(None, 500, 500, 3)]
conv2d Conv2D input: (None, 500, 500, 3) output: (None, 500, 500, 16)
conv2d_1 Conv2D input: (None, 500, 500, 16) output: (None, 500, 500, 32)
conv2d_2 Conv2D input: (None, 500, 500, 32) output: (None, 500, 500, 64)
conv2d_3 Conv2D input: (None, 500, 500, 3) output: (None, 500, 500, 32)
conv2d_4 Conv2D input: (None, 500, 500, 32) output: (None, 500, 500, 64)
conv2d_5 Conv2D input: (None, 500, 500, 64) output: (None, 500, 500, 64)
add Add input: [(None, 500, 500, 64), (None, 500, 500, 64), (None, 500, 500, 64)] output: (None, 500, 500, 64)
conv2d_6 Conv2D input: (None, 500, 500, 64) output: (None, 500, 500, 64)
conv2d_7 Conv2D input: (None, 500, 500, 64) output: (None, 500, 500, 32)
conv2d_8 Conv2D input: (None, 500, 500, 32) output: (None, 500, 500, 16)
conv2d_9 Conv2D input: (None, 500, 500, 64) output: (None, 500, 500, 32)
conv2d_10 Conv2D input: (None, 500, 500, 32) output: (None, 500, 500, 16)
conv2d_11 Conv2D input: (None, 500, 500, 64) output: (None, 500, 500, 16)
conv2d_13 Conv2D input: (None, 500, 500, 64) output: (None, 500, 500, 16)
add_1 Add input: [(None, 500, 500, 16), (None, 500, 500, 16), (None, 500, 500, 16)] output: (None, 500, 500, 16)
conv2d_12 Conv2D input: (None, 500, 500, 16) output: (None, 500, 500, 16)
add_2 Add input: [(None, 500, 500, 16), (None, 500, 500, 16), (None, 500, 500, 16)] output: (None, 500, 500, 16)
conv2d_15 Conv2D input: (None, 500, 500, 16) output: (None, 500, 500, 16)
conv2d_16 Conv2D input: (None, 500, 500, 16) output: (None, 500, 500, 3)


Ground Truth
Low Light Image
Enhanced Image
0
100
200
300
400
0
100
200
300
400

REFERENCES

- Machinelearningmastery.com
- https://www.youtube.com/live/GK5ATUhdH74?si=umyaNz3ZY-VpjK_6
- https://youtu.be/tMjdQLylyGI?si=VWnXSudpp5r4Wbu5
- https://www.geeksforgeeks.org/weight-initialization-techniques-for-deep-neural-networks/
- (4466) Building a Low-Light Enhancement model using Convolutional Neural Networks (CNN) | Tutorial – YouTube
- Low Light Enhancement - Using CNNs - Shared.ipynb - Colaboratory (google.com)
- Low Light Image Enhancement with Keras MIRNet | Kaggle
- https://chat.openai.com