



KESHAV MEMORIAL INSTITUTE OF TECHNOLOGY
AN AUTONOMOUS INSTITUTION - ACCREDITED BY NAAC WITH 'A' GRADE
Narayanauguda, Hyderabad.

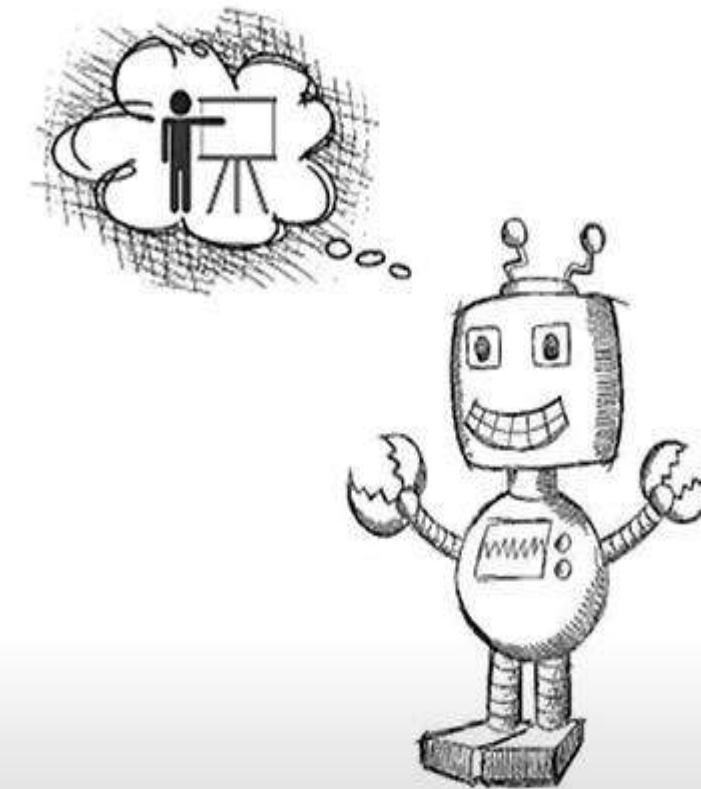
Deep Learning

SESSION-1

BY
ASHA

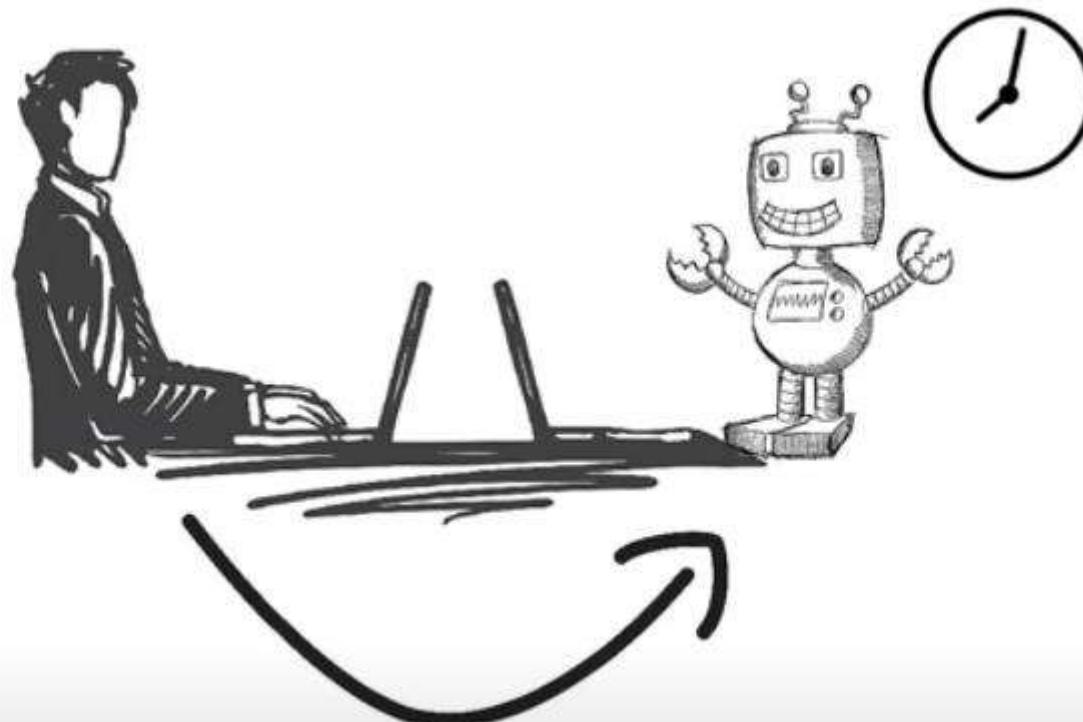


**HUMANS LEARN FROM
PAST EXPERIENCES**



**MACHINES FOLLOW INSTRUCTIONS
GIVEN BY HUMANS**

WHAT IF HUMANS CAN TRAIN THE MACHINES...

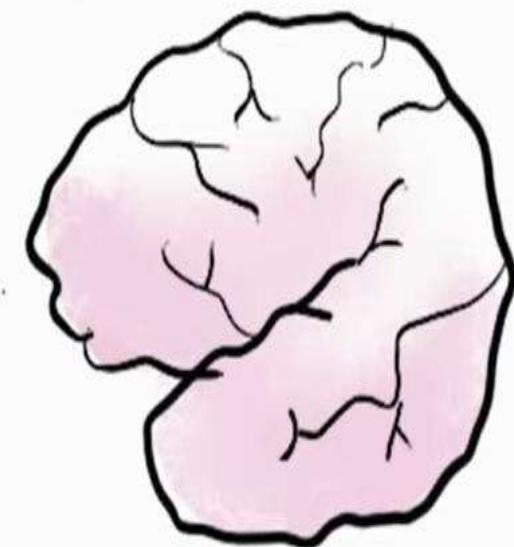


Execute Instructions

```
function withdraw(amount) {  
    if (amount > balance) {  
        fail("Hey you ain't got the cash!")  
    } else {  
        balance = balance - amount  
    }  
}
```



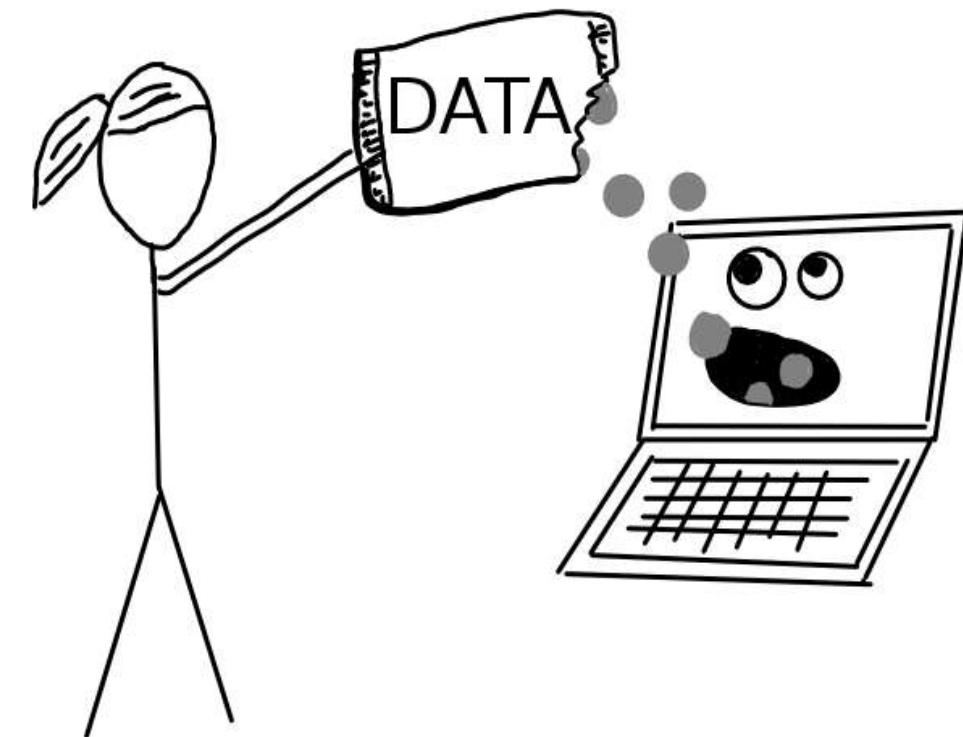
Learn + Think

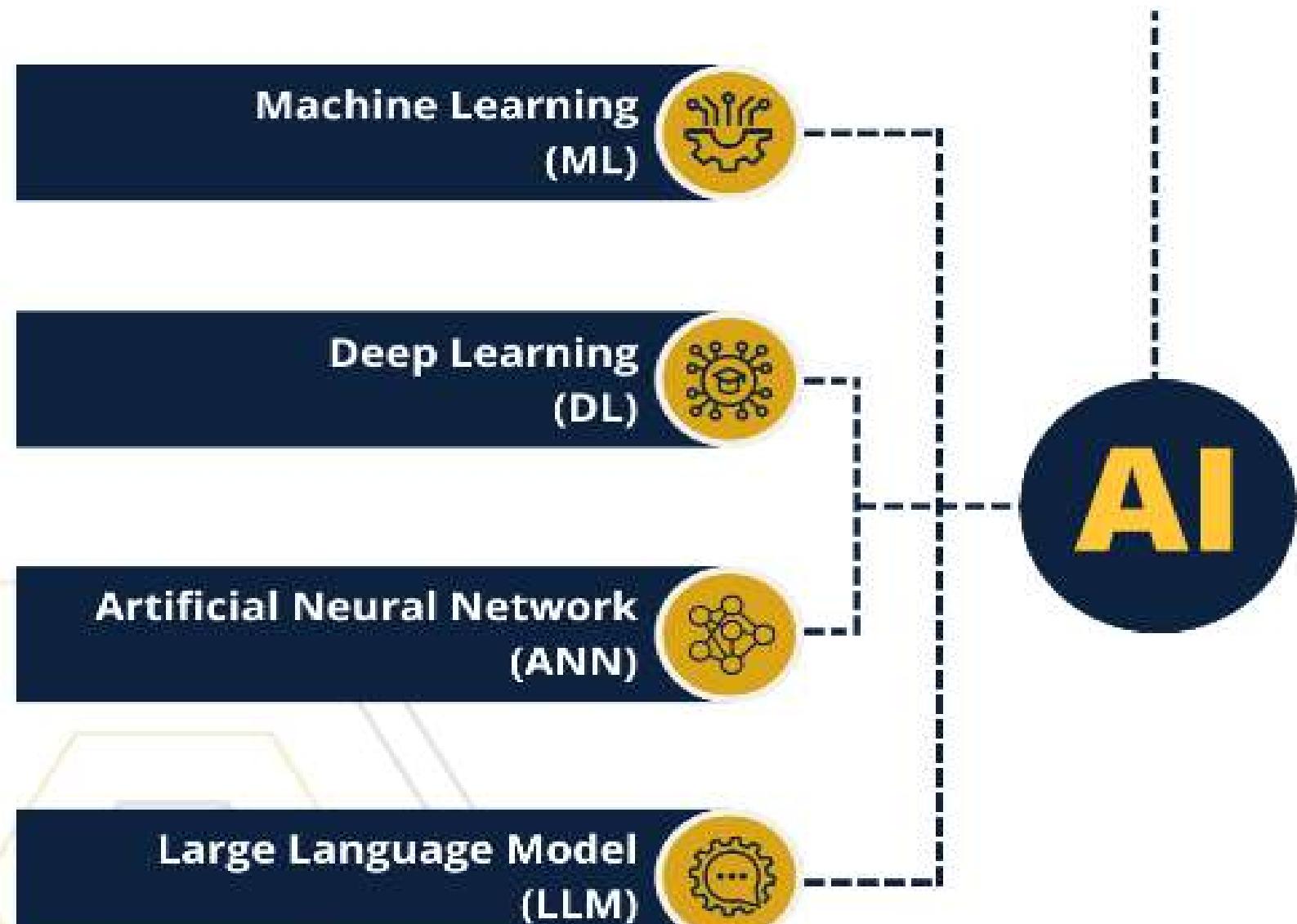


Without Machine Learning



With Machine Learning





ARTIFICIAL INTELLIGENCE

- ARTIFICIAL INTELLIGENCE REFERS TO THE DEVELOPMENT OF COMPUTER SYSTEMS THAT CAN PERFORM TASKS THAT TYPICALLY REQUIRE HUMAN INTELLIGENCE.



Tesla Autopilot Car



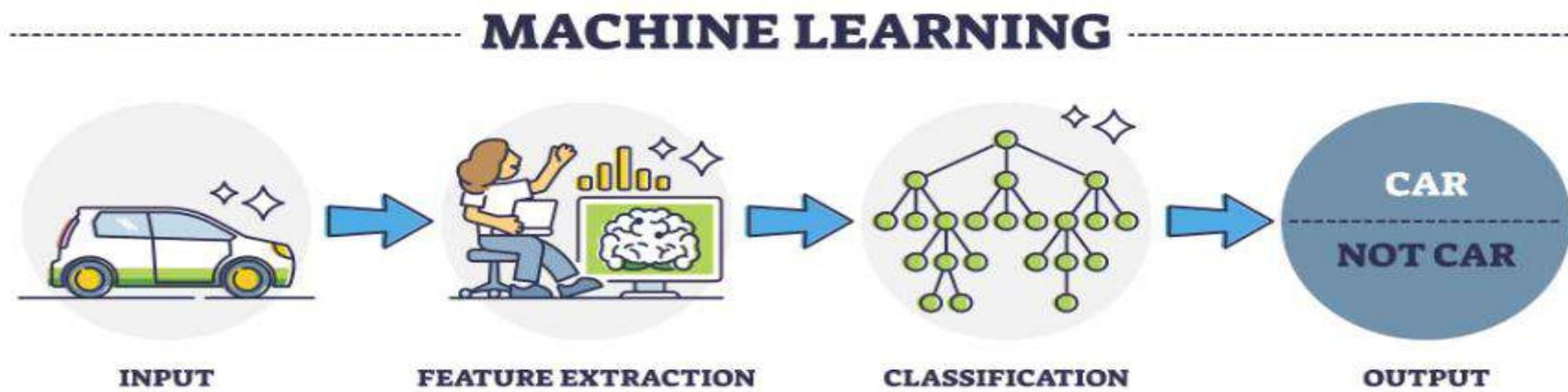
Sophia Humanoid Robot



Amazon Alexa

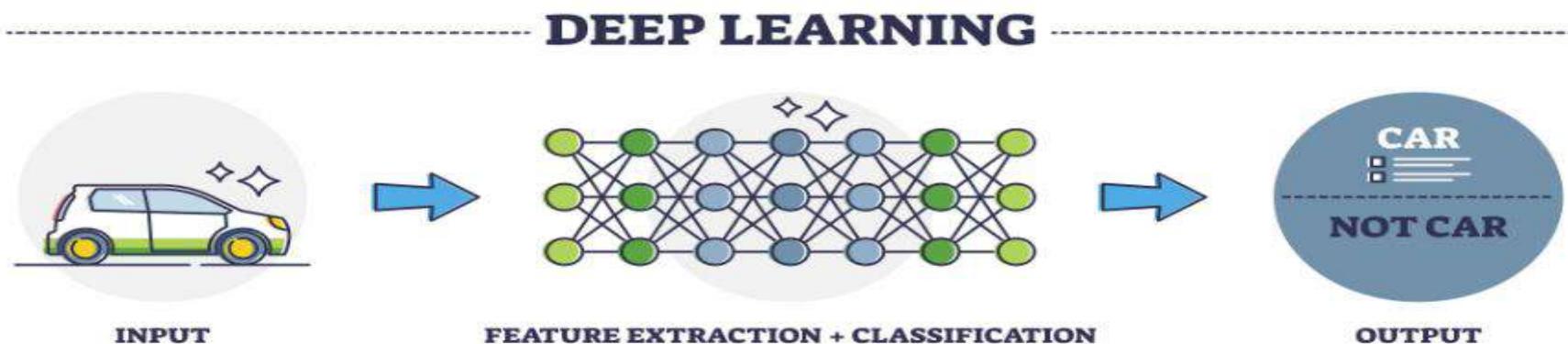
MACHINE LEARNING

- MACHINE LEARNING INVOLVES ALGORITHMS AND STATISTICAL MODELS THAT ENABLE COMPUTERS TO IMPROVE THEIR PERFORMANCE ON A SPECIFIC TASK WITHOUT EXPLICIT PROGRAMMING.
- IT FOCUSES ON PATTERN RECOGNITION AND LEARNING FROM DATA.
- MACHINE LEARNING IS A SUBSET OF ARTIFICIAL INTELLIGENCE.

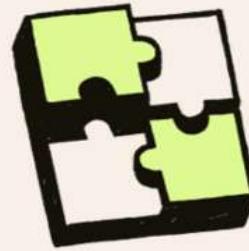


DEEP LEARNING

- DEEP LEARNING IS A SUBSET OF MACHINE LEARNING THAT INVOLVES NEURAL NETWORKS WITH MULTIPLE LAYERS (DEEP NEURAL NETWORKS).
- THESE NETWORKS CAN AUTOMATICALLY LEARN TO EXTRACT FEATURES FROM DATA AND MAKE COMPLEX DECISIONS BASED ON LARGE AMOUNTS OF DATA.



Machine learning vs. deep learning



Machine learning

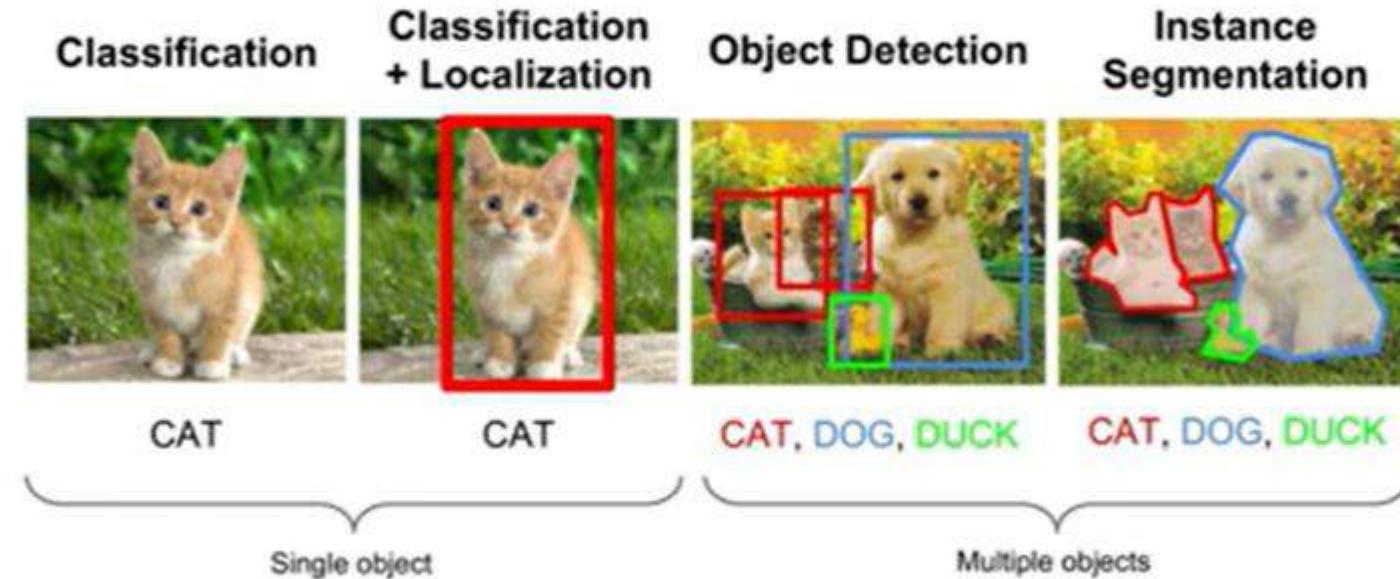
Uses algorithms and learns on its own but may need human intervention to correct errors



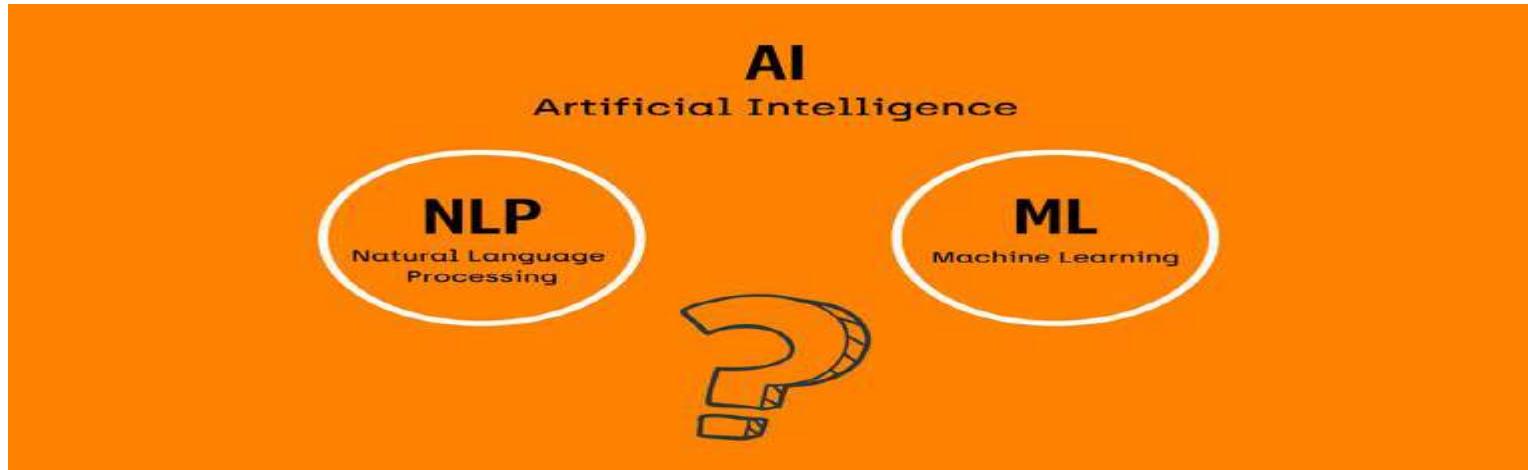
Deep learning

Uses advanced computing, its own neural network, to adapt with little to no human intervention

COMPUTER VISION (CV) IS A FIELD OF ARTIFICIAL INTELLIGENCE THAT ENABLES COMPUTERS TO INTERPRET AND UNDERSTAND THE VISUAL WORLD. USING DIGITAL IMAGES FROM CAMERAS AND VIDEOS AND DEEP LEARNING MODELS, MACHINES CAN ACCURATELY IDENTIFY AND CLASSIFY OBJECTS, AND THEN REACT TO WHAT THEY "SEE."



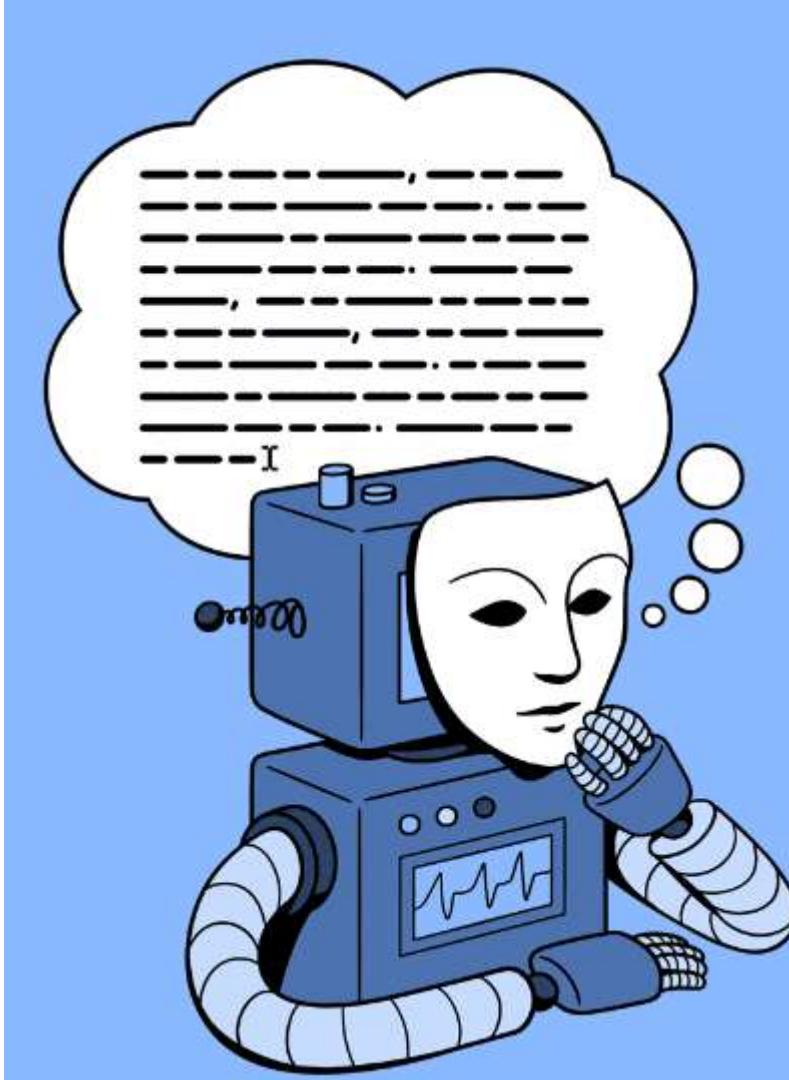
NATURAL LANGUAGE PROCESSING



- NATURAL LANGUAGE PROCESSING (NLP) IS A SUBFIELD OF ARTIFICIAL INTELLIGENCE THAT FOCUSES ON THE INTERACTION BETWEEN COMPUTERS AND HUMANS THROUGH NATURAL LANGUAGE.
- THE ULTIMATE GOAL OF NLP IS TO ENABLE COMPUTERS TO UNDERSTAND, INTERPRET, AND GENERATE HUMAN LANGUAGES IN A VALUABLE WAY.

LARGE LANGUAGE MODEL

- LARGE LANGUAGE MODELS ARE ADVANCED AI MODELS TRAINED ON VAST AMOUNTS OF TEXT DATA, ENABLING THEM TO UNDERSTAND AND GENERATE HUMAN-LIKE LANGUAGE.
- VIRTUAL ASSISTANTS LIKE SIRI OR ALEXA UTILIZE LARGE LANGUAGE MODELS TO UNDERSTAND AND RESPOND TO NATURAL LANGUAGE QUERIES.



Large Language Model (LLM)

[ˈlärj ˈlaŋ-gwij ˈmä-dəl]

A deep learning algorithm that's equipped to summarize, translate, predict, and generate human-sounding text to convey ideas and concepts.

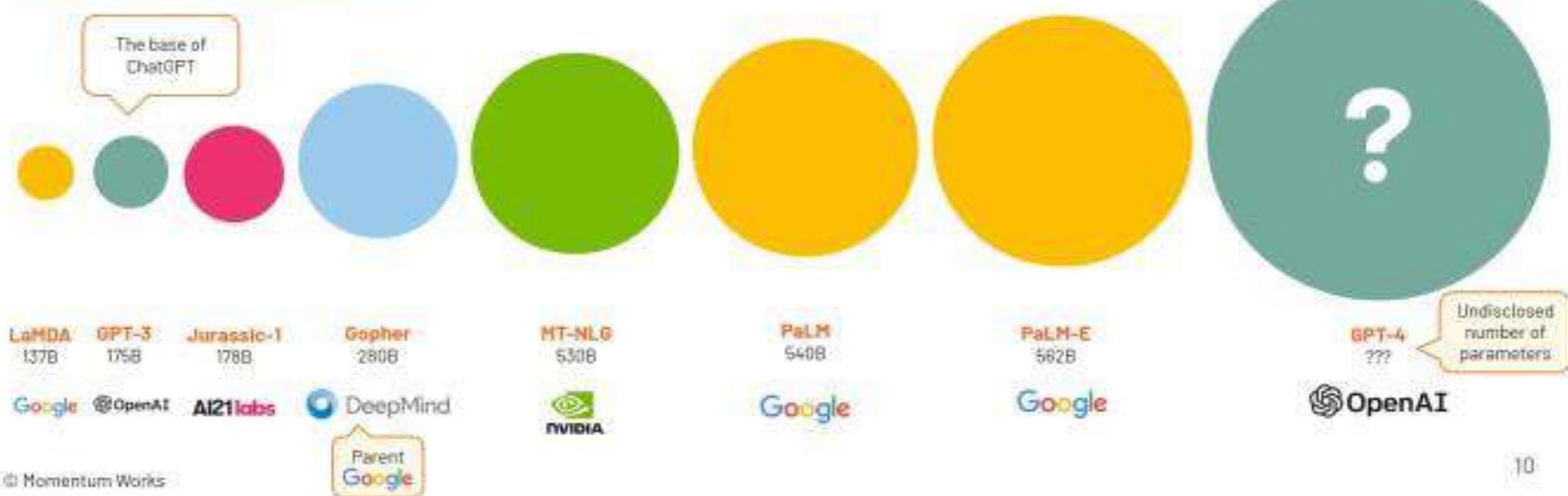


Large Language Models are becoming very large indeed

Small models (<= 100b parameters)



Large models (>100b parameters)



GENERATIVE AI

- GENERATIVE AI REFERS TO ARTIFICIAL INTELLIGENCE SYSTEMS THAT ARE CAPABLE OF CREATING NEW CONTENT SUCH AS TEXT, IMAGES, OR MUSIC.
- THESE SYSTEMS LEARN FROM EXISTING DATA PATTERNS AND GENERATE FRESH, ORIGINAL CONTENT.

GENERATIVE AI

who are you? explain it to a 10 year old kid



Hi there! I'm ChatGPT, and I'm like a super-smart robot that can talk to you and help answer your questions. Imagine I'm a really big library that knows a lot about almost everything, but I can talk and understand what you say. Here's a simple way to understand me:

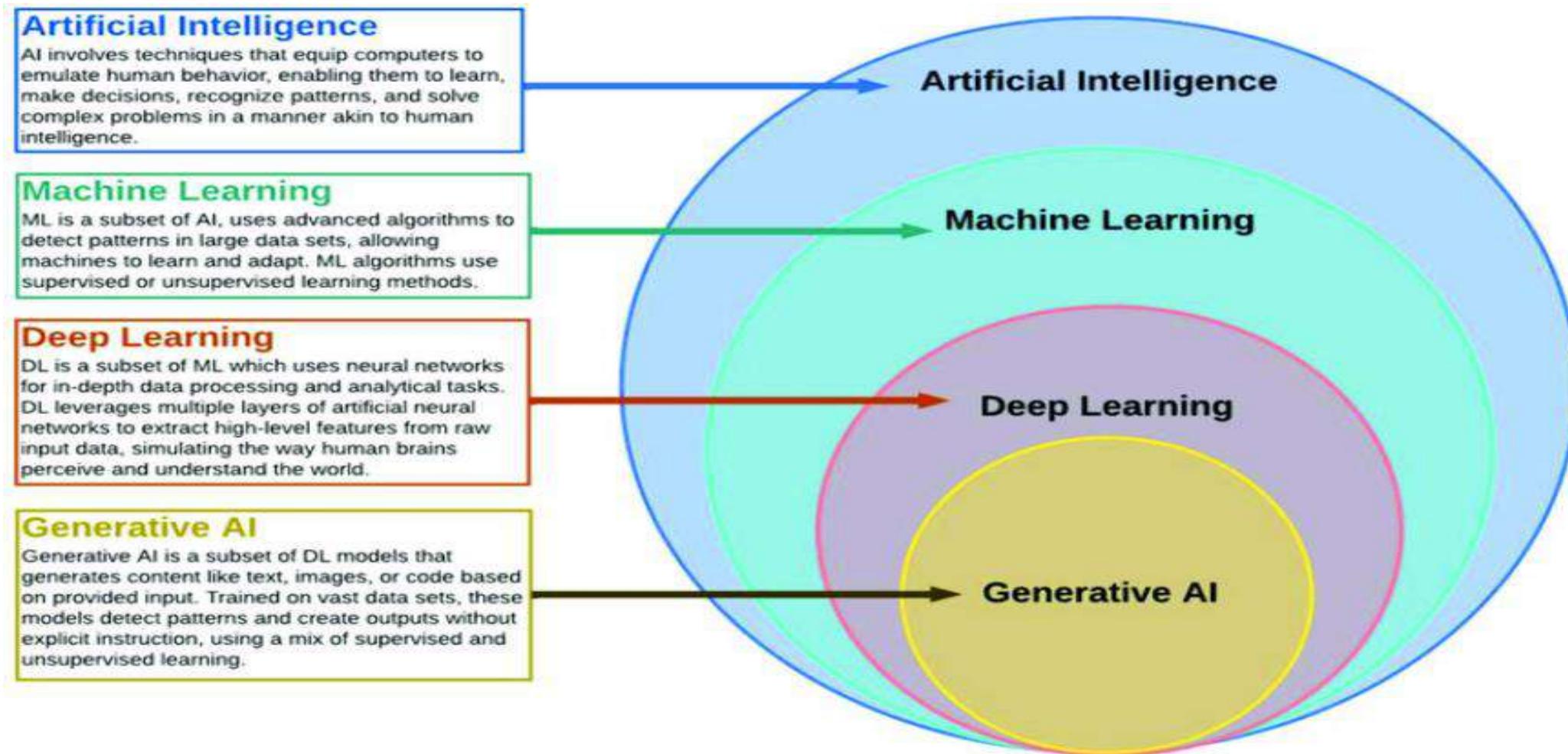
MACHINE LEARNING



In 1959, Arthur Samuel, a computer scientist who pioneered the study of artificial intelligence, described machine learning as "The study that gives computers the ability to learn."

Machine learning is a subset of AI, which enables the machine to automatically learn from data, improve performance from past experiences, and make predictions.

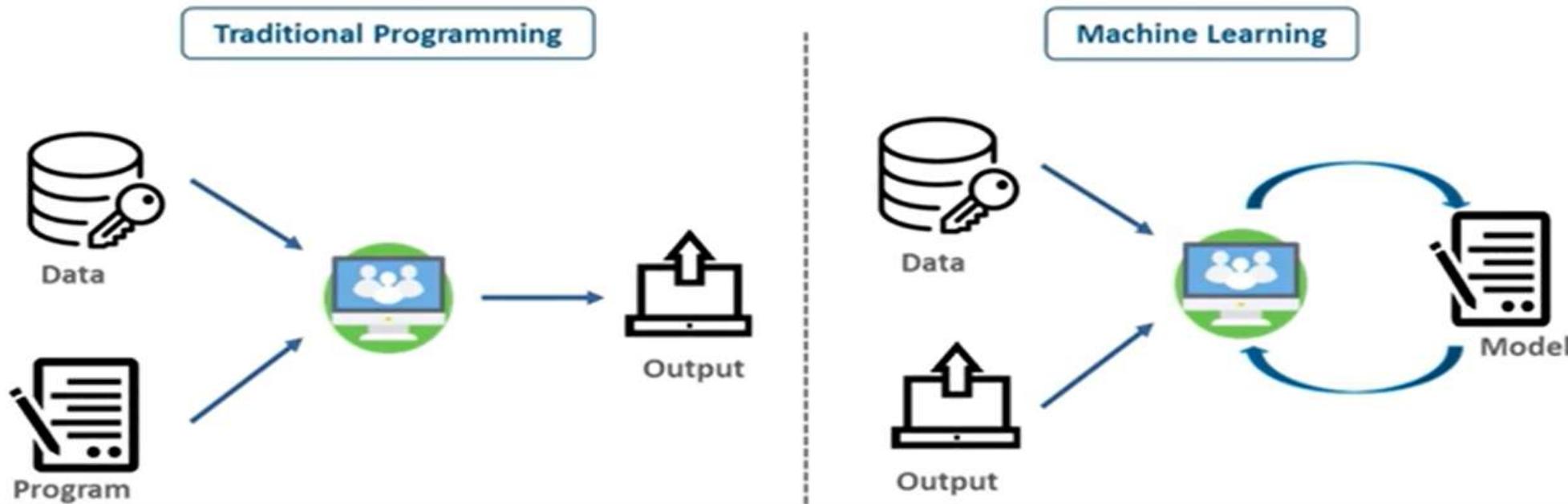




MACHINE LEARNING IS A SUBSET OF ARTIFICIAL INTELLIGENCE THAT AIMS TO MIMIC HOW HUMAN BEINGS LEARN BY USING DATA.

A more technical definition given by Tom M. Mitchell's (1997) : “A computer program is said to learn from experience E with respect to some class of tasks T and performance measure P, if its performance at tasks in T, as measured by P, improves with experience E.”

HOW MACHINE LEARNING WORKS?

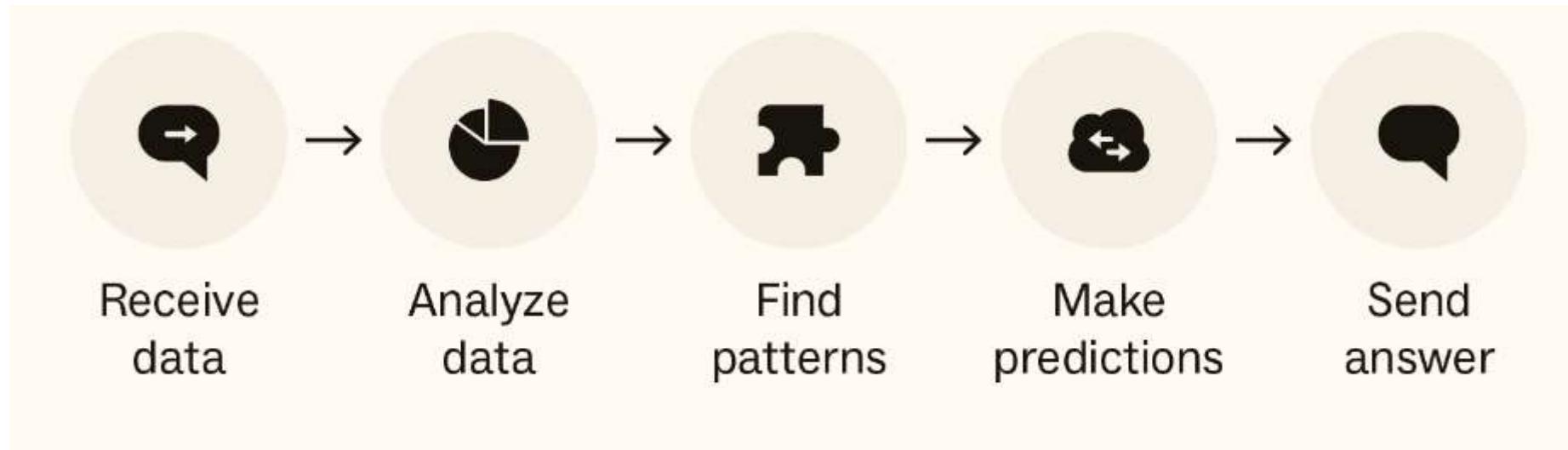


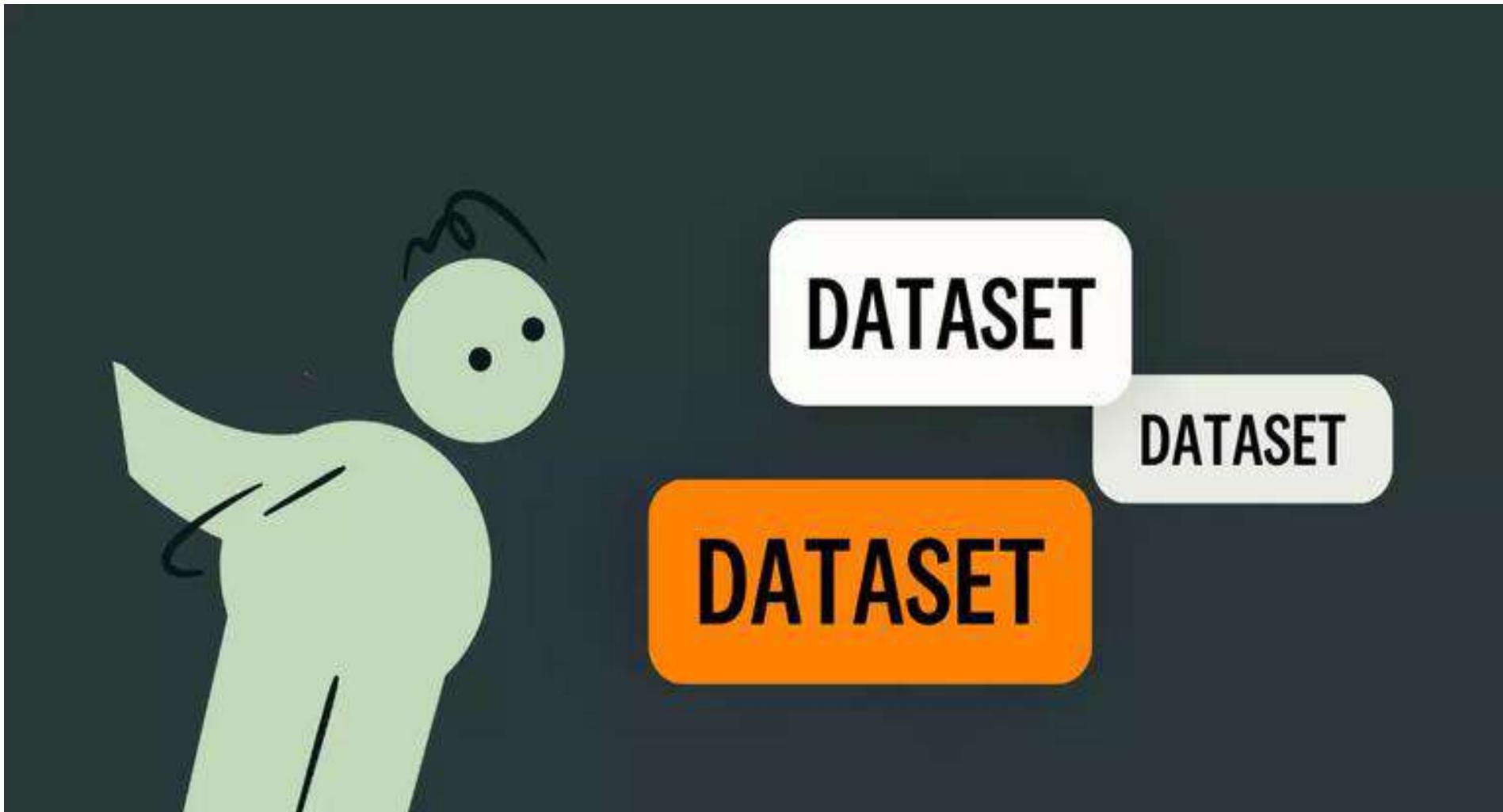
Learn from Data

Find Hidden Insights

Train and Grow

HOW MACHINE LEARNING WORKS?





DATASET

- A dataset is a collection of data in which data is arranged in some order. A dataset can contain any data from a series of an array to a database table.
- A tabular dataset can be understood as a database table or matrix, where each column corresponds to a particular variable, and each row corresponds to the fields of the dataset. The most supported file type for a tabular dataset is "Comma Separated File," or CSV.

NEED OF DATASET

To work with machine learning projects, we need a huge amount of data. Collecting and preparing the dataset is one of the most crucial parts while creating an ML/AI project.

POPULAR SOURCES FOR MACHINE LEARNING DATASETS

- Kaggle Datasets
- UCI Machine Learning Repository
- Datasets via AWS
- Google's Dataset Search Engine
- Microsoft Datasets

DATA PREPROCESSING

DATA PRE-PROCESSING IS A PROCESS OF CLEANING THE RAW DATA I.E. THE DATA IS COLLECTED IN THE REAL WORLD AS MOST OF THE REAL-WORLD DATA IS MESSY, SOME OF THESE TYPES OF DATA ARE:

- 1. MISSING DATA**
- 2. NOISY DATA**
- 3. INCONSISTENT DATA**

WHY IS DATA PREPROCESSING IMPORTANT?

THE MAJORITY OF THE REAL-WORLD DATASETS FOR MACHINE LEARNING ARE HIGHLY SUSCEPTIBLE TO BE MISSING, INCONSISTENT, AND NOISY.

- DATA PROCESSING IS, THEREFORE, IMPORTANT TO IMPROVE THE OVERALL DATA QUALITY.
- DUPLICATE OR MISSING VALUES MAY GIVE AN INCORRECT VIEW OF THE OVERALL STATISTICS OF DATA
- OUTLIERS AND INCONSISTENT DATA POINTS OFTEN TEND TO DISTURB THE MODEL'S OVERALL LEARNING, LEADING TO FALSE PREDICTIONS.

DATA REDUCTION

- THE SIZE OF THE DATASET IN A DATA WAREHOUSE CAN BE TOO LARGE TO BE HANDLED BY DATA ANALYSIS AND DATA MINING ALGORITHMS.
- ONE POSSIBLE SOLUTION IS TO OBTAIN A REDUCED REPRESENTATION OF THE DATASET THAT IS MUCH SMALLER IN VOLUME BUT PRODUCES THE SAME QUALITY OF ANALYTICAL RESULTS.

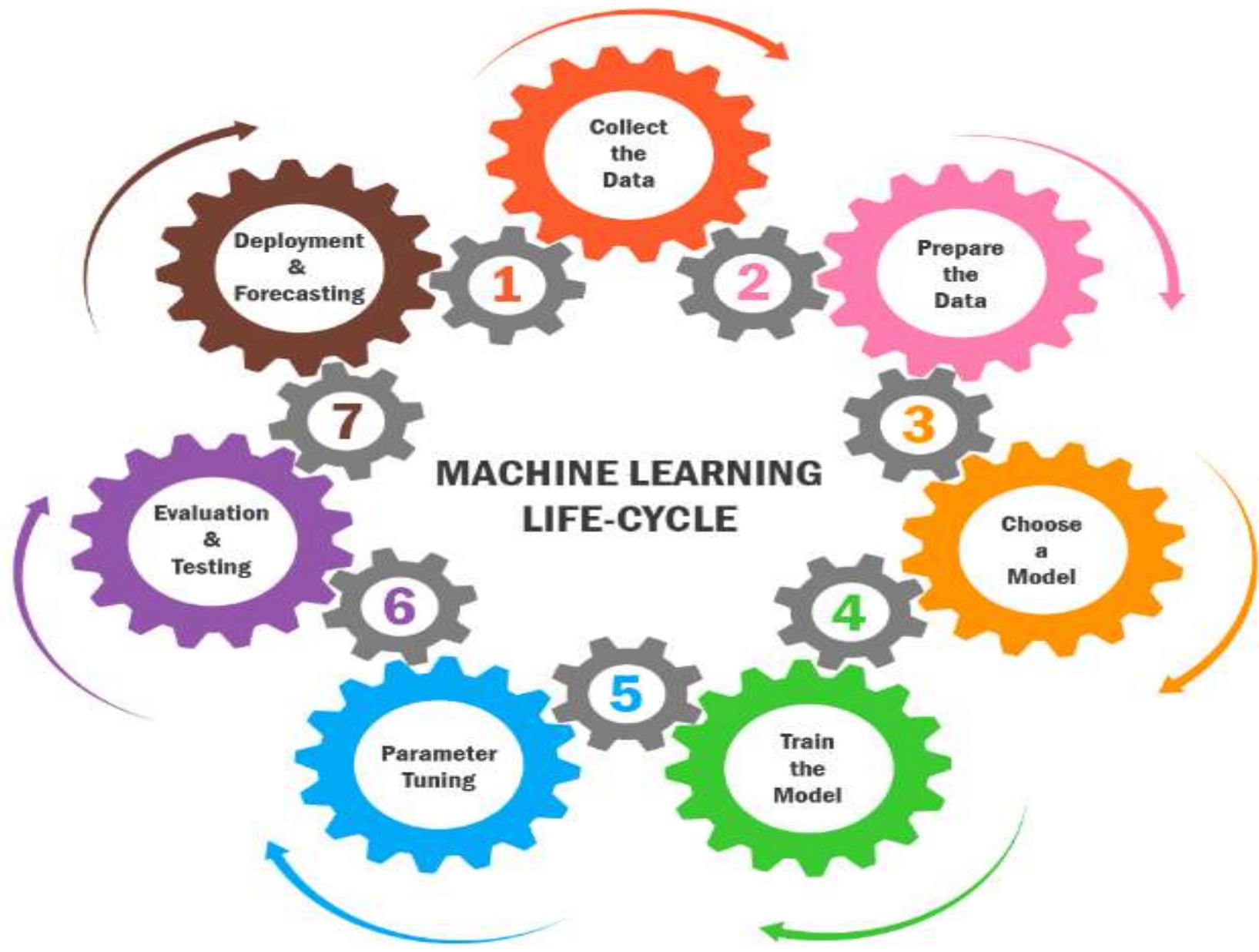
- **HANDLING MISSING VALUES:** TECHNIQUES INCLUDE REMOVING INSTANCES WITH MISSING VALUES, IMPUTING MISSING VALUES WITH THE MEAN, MEDIAN, OR MODE, OR USING ADVANCED TECHNIQUES LIKE KNN IMPUTATION.
- **REMOVING DUPLICATES:** IDENTIFYING AND REMOVING DUPLICATE INSTANCES TO ENSURE THE DATASET IS CLEAN.
- **FEATURE SCALING:**
- **NORMALIZATION:** RESCALING THE FEATURES TO A RANGE OF [0, 1].
- **STANDARDIZATION:** RESCALING THE FEATURES TO HAVE A MEAN OF 0 AND A STANDARD DEVIATION OF 1.

ENCODING CATEGORICAL DATA:

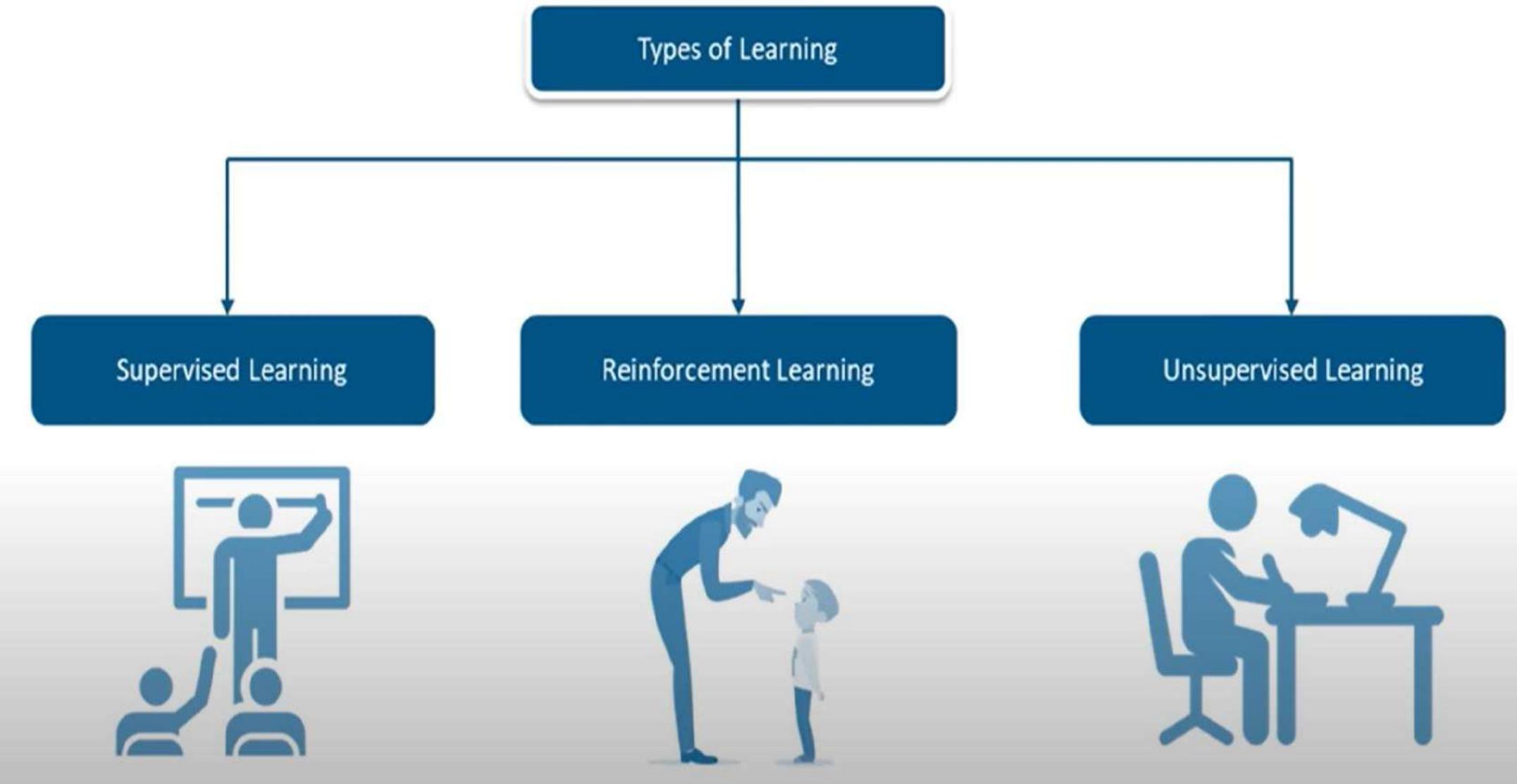
- **ONE-HOT ENCODING:** CONVERTING CATEGORICAL VARIABLES INTO BINARY VECTORS.
- **LABEL ENCODING:** CONVERTING CATEGORICAL VARIABLES INTO INTEGER VALUES.

SPLITTING DATA:

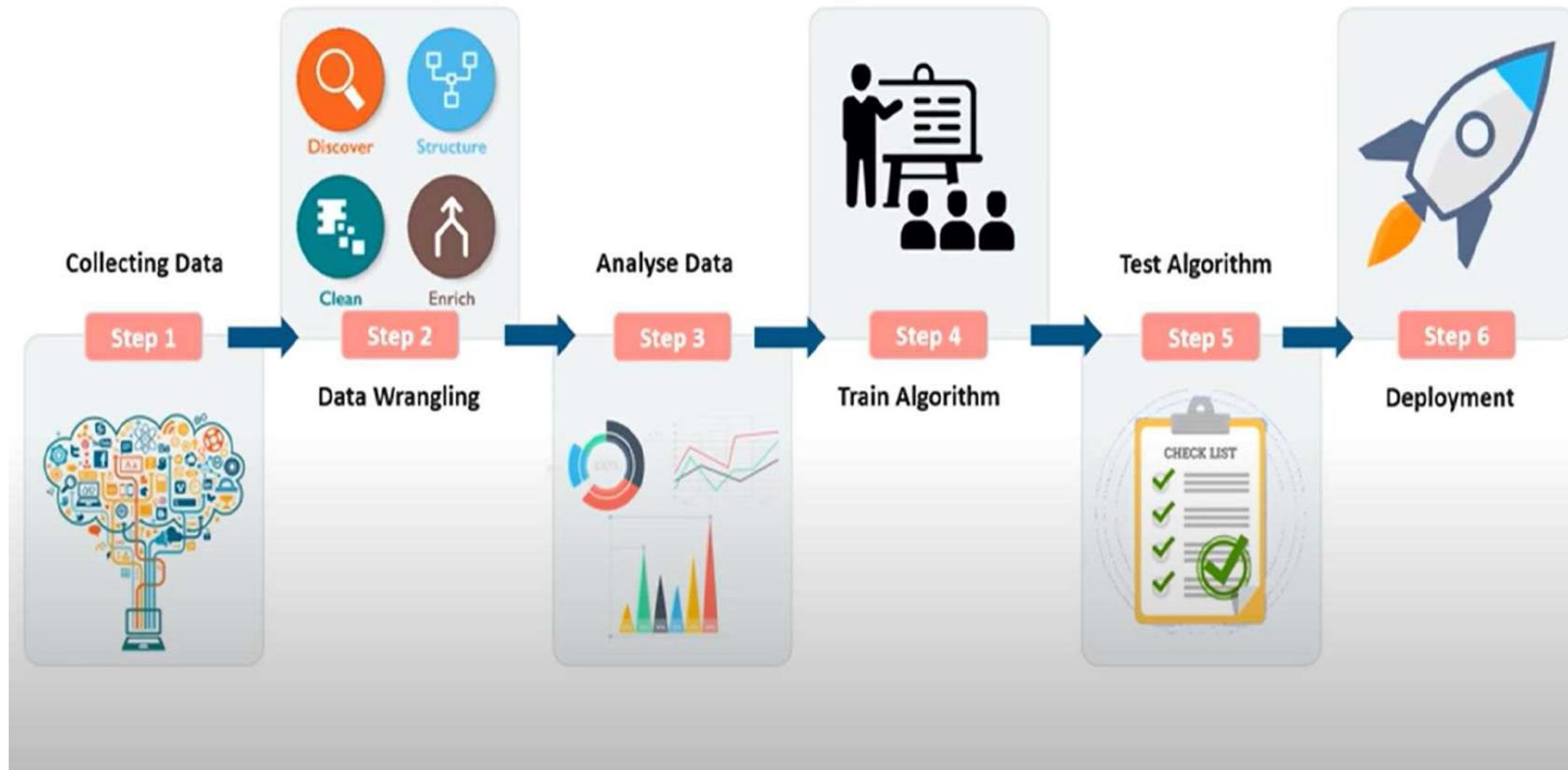
- DIVIDING THE DATASET INTO TRAINING AND TESTING SETS TO EVALUATE THE MODEL'S PERFORMANCE.



MACHINE LEARNING TYPES ?



ALGORITHM DEVELOPMENT STEPS



BASIC TERMINOLOGY

FEATURES AND LABELS:

- FEATURES:** THE INPUT VARIABLES (INDEPENDENT VARIABLES) USED BY THE MODEL TO MAKE PREDICTIONS.
- LABELS:** THE OUTPUT VARIABLE (DEPENDENT VARIABLE) THAT THE MODEL IS TRYING TO PREDICT.

Features					Label
Position	Experience	Skill	Country	City	Salary (\$)
Developer	0	1	USA	New York	103100
Developer	1	1	USA	New York	104900
Developer	2	1	USA	New York	106800
Developer	3	1	USA	New York	108700
Developer	4	1	USA	New York	110400
Developer	5	1	USA	New York	112300
Developer	6	1	USA	New York	114200
Developer	7	1	USA	New York	116100
Developer	8	1	USA	New York	117800
Developer	9	1	USA	New York	119700
Developer	10	1	USA	New York	121600

TRAINING AND TESTING:

- **TRAINING SET:** A SUBSET OF THE DATASET USED TO TRAIN THE MODEL.
- **TESTING SET:** A SUBSET OF THE DATASET USED TO EVALUATE THE MODEL'S PERFORMANCE.

TRAINING SET

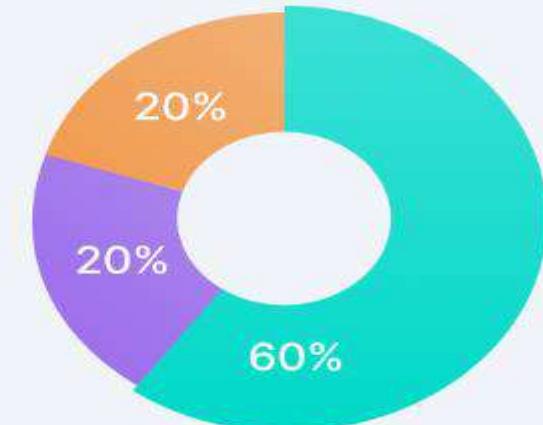
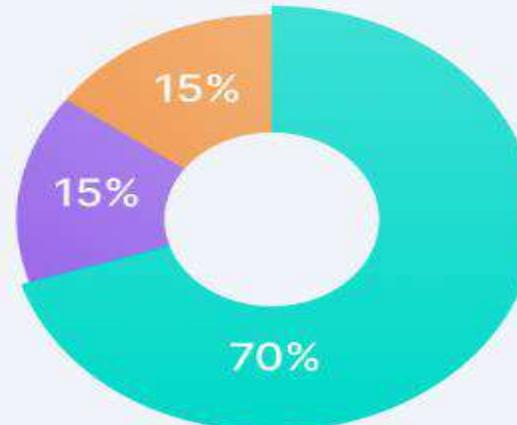
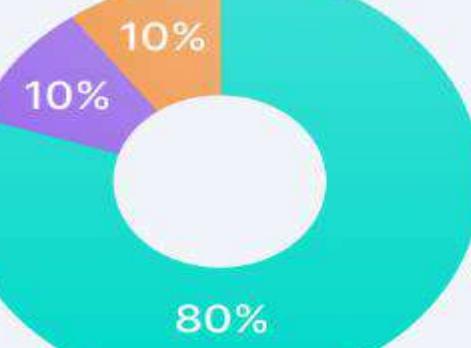
The subset of data used to train a machine learning model

TEST SET

The subset of data used to evaluate the performance of a trained machine learning model on unseen examples, simulating real-world data

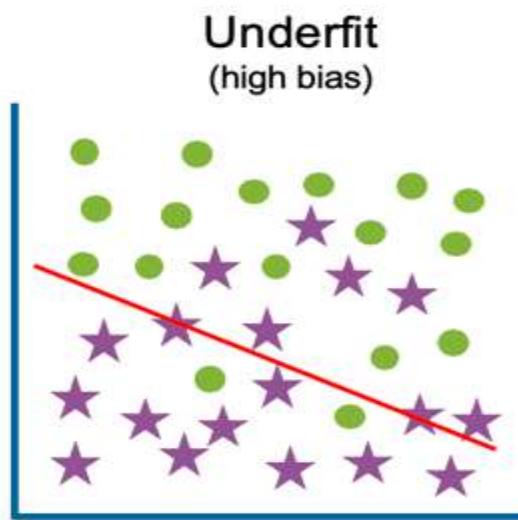
VALIDATION SET

The intermediary subset of data used during the model development process to fine-tune hyperparameters

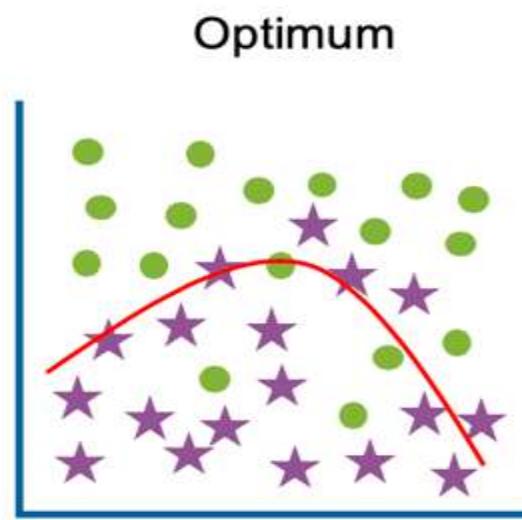


OVERFITTING AND UNDERFITTING:

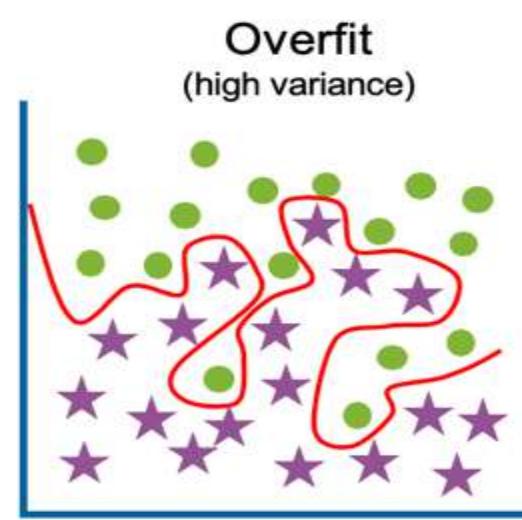
- **OVERFITTING:** WHEN THE MODEL PERFORMS WELL ON THE TRAINING DATA BUT POORLY ON THE TESTING DATA BECAUSE IT HAS LEARNED NOISE AND DETAILS FROM THE TRAINING DATA.
- **UNDERFITTING:** WHEN THE MODEL PERFORMS POORLY ON BOTH THE TRAINING AND TESTING DATA BECAUSE IT IS TOO SIMPLE TO CAPTURE THE UNDERLYING PATTERNS IN THE DATA.



High training error
High test error

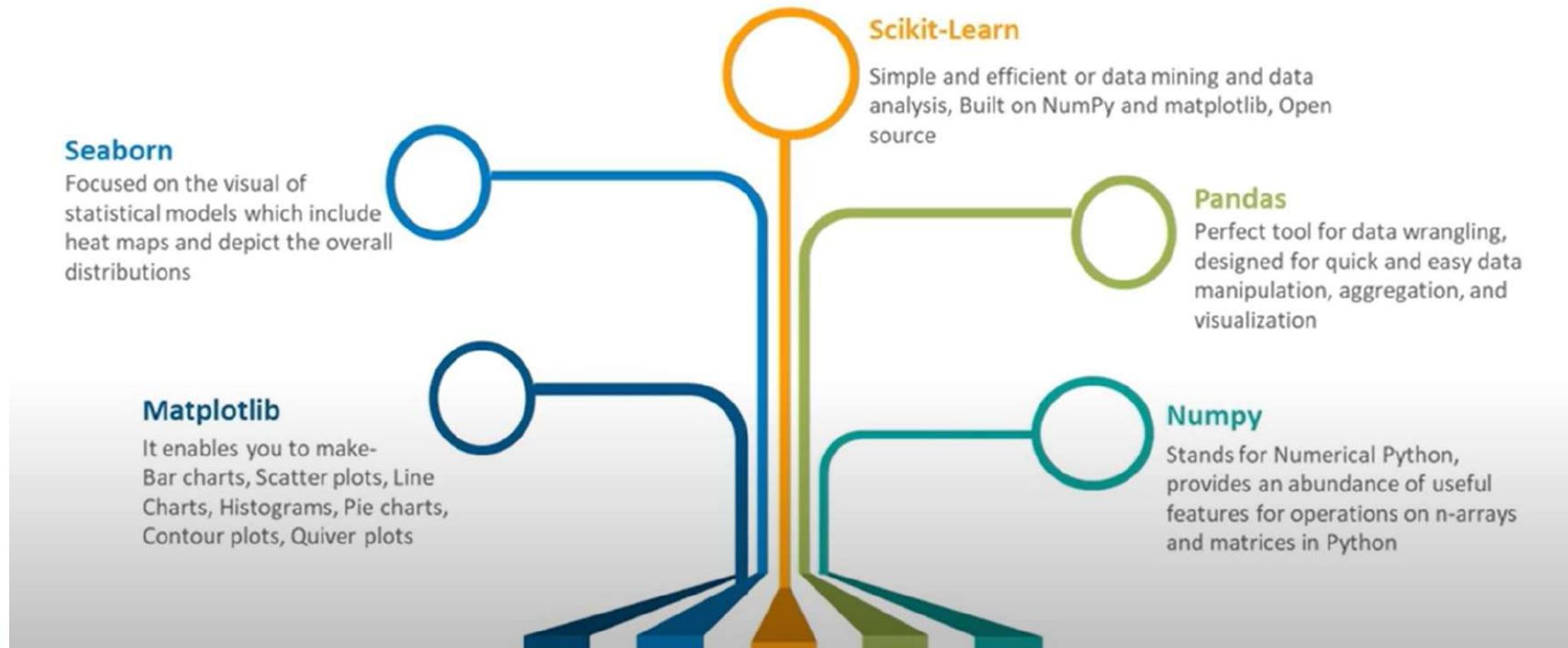


Low training error
Low test error

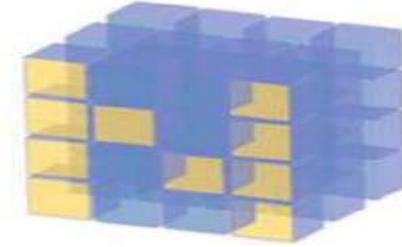


Low training error
High test error

WHAT LIBRARIES DO WE USE FOR MACHINE LEARNING?



NumPy



- NumPy – Numerical python is a very popular python library for array and matrix processing, with the help of a large collection of high-level mathematical functions.
- It is very useful for fundamental scientific computations in Machine Learning.

Pandas



- Pandas-Panel data is a popular Python library for data analysis.
- It is not directly related to Machine Learning but the dataset must be prepared before training for which Pandas are useful as it is developed specifically for data extraction and preparation.
- It provides data structures and wide variety tools for data analysis. It provides many inbuilt methods for filtering, combining and grouping data.

Matplotlib

- Matplotlib is a Python library for data visualization. Like Pandas, it is not directly related to Machine Learning. It is needed when a programmer wants to visualize the patterns in the data.
- A module named pyplot makes it easy for programmers for plotting .

Scikit-learn



- Scikit-learn is one of the most popular ML libraries for classical ML algorithms.
- Scikit-learn supports most of the supervised and unsupervised learning algorithms

TensorFlow



- TensorFlow is a popular open-source library for high performance numerical computation.
- It can train and run deep neural networks that can be used to develop several AI applications. TensorFlow is widely used in the field of deep learning research and application.



THANK YOU



KESHAV MEMORIAL INSTITUTE OF TECHNOLOGY
AN AUTONOMOUS INSTITUTION - ACCREDITED BY NAAC WITH 'A' GRADE
Narayananaguda, Hyderabad.

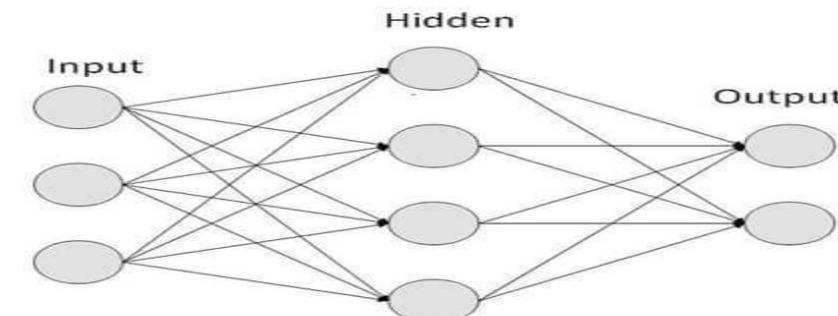
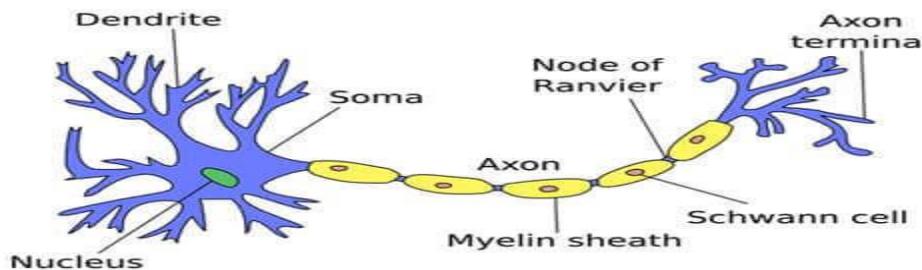
Deep Learning

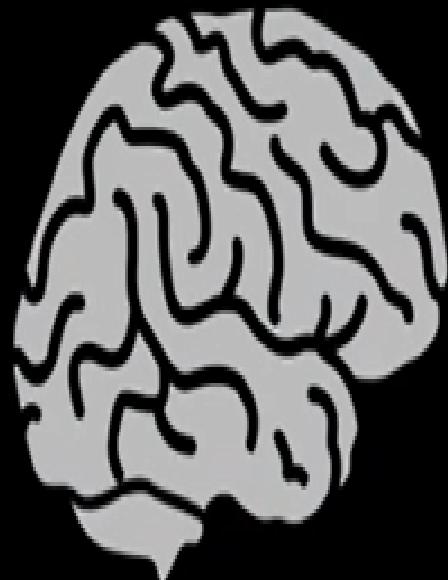
SESSION-2

BY
ASHA

Introduction to Neural Networks

- Neural networks are computational models inspired by the human brain. They consist of interconnected units or nodes called neurons, organized into layers.
- Neural networks are capable of learning from data and making predictions or decisions without being explicitly programmed.

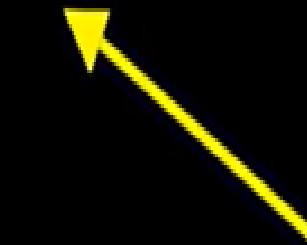




Neural network



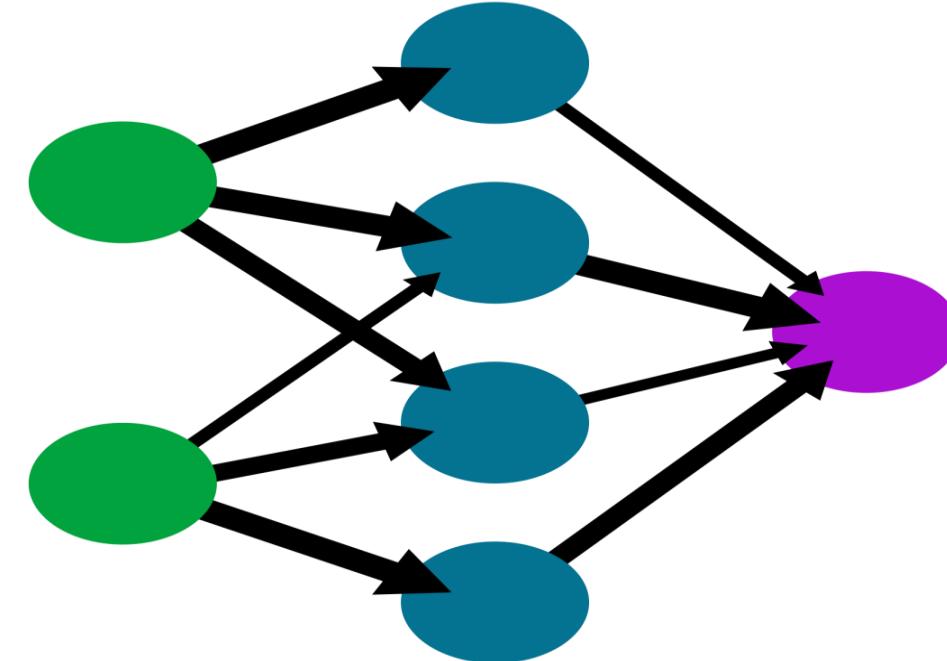
What are
the neurons?



How are
they connected?

A simple neural network

input layer hidden layer output layer



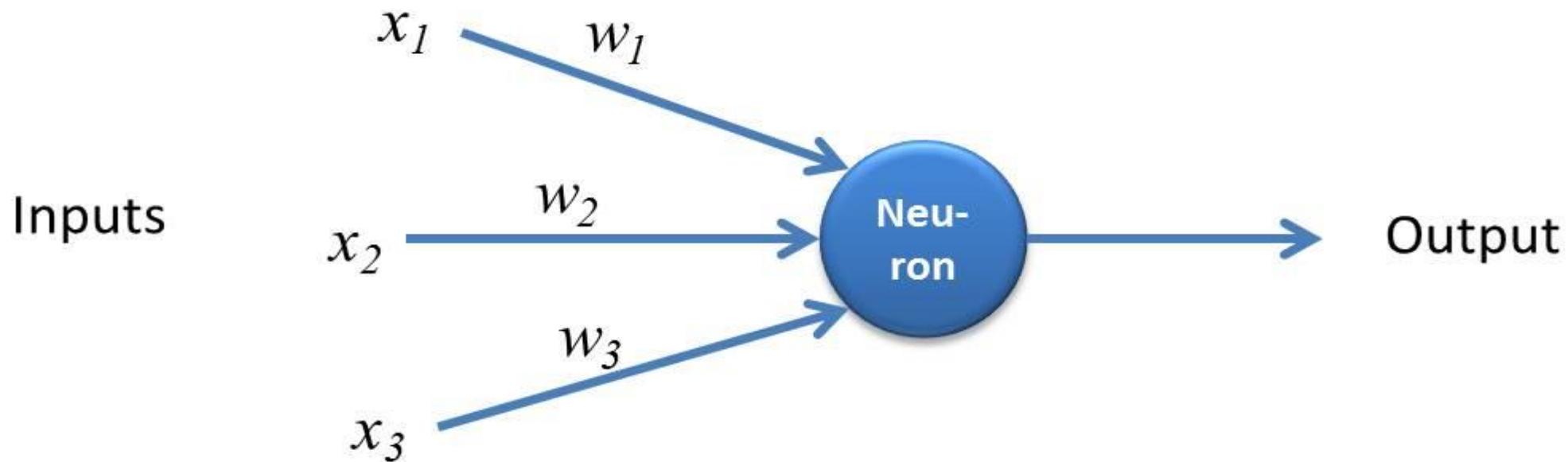
A simple neural network consists of three components :

- Input layer
- Hidden layer
- Output layer

- **Input Layer:** Also known as Input nodes are the inputs/information from the outside world is provided to the model. Input nodes pass the information to the next layer i.e Hidden layer.
- **Hidden Layer:** Hidden layer is the set of neurons where all the computations are performed on the input data. There can be any number of hidden layers in a neural network. The simplest network consists of a single hidden layer.
- **Output layer:** The output layer is the output/conclusions of the model derived from all the computations performed. There can be single or multiple nodes in the output layer. If we have a binary classification problem the output node is 1 but in the case of multi-class classification, the output nodes can be more than 1.

Perceptron

- **Perceptron** is a simple form of Neural Network and consists of a single layer where all the mathematical computations are performed.

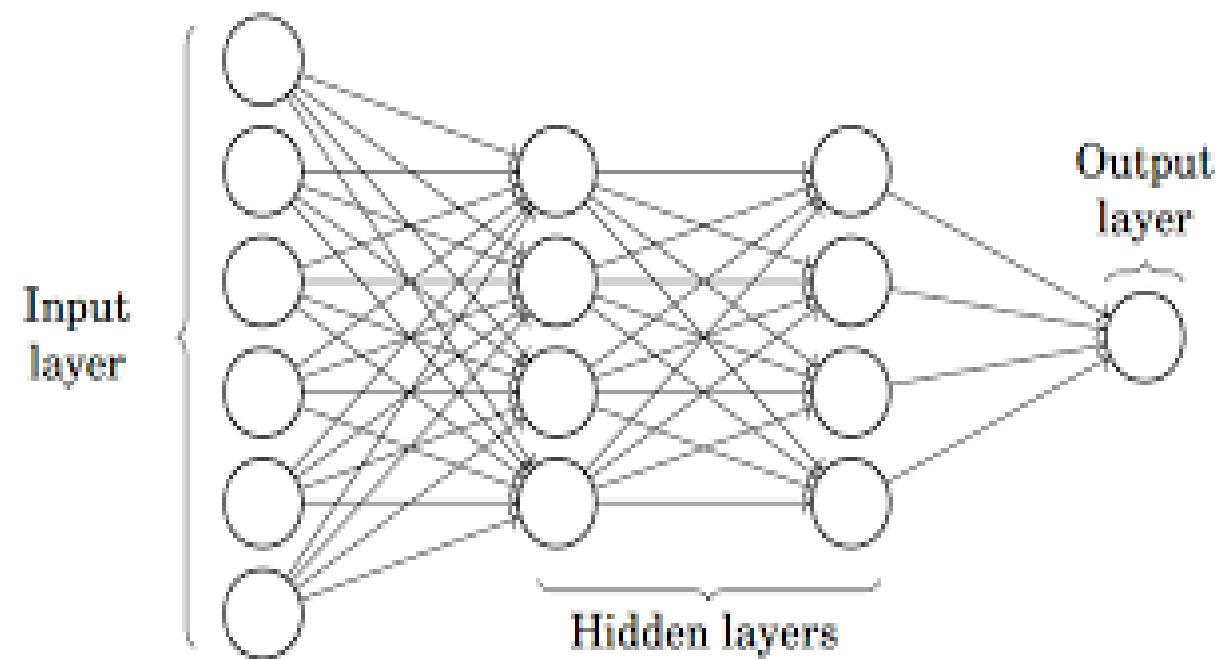


Types of neural networks

The three important types of neural networks in deep learning are:

- Artificial Neural Networks (ANN)
- Convolution Neural Networks (CNN)
- Recurrent Neural Networks (RNN)

- Whereas, **Multilayer Perceptron** also known as **Artificial Neural Networks** consists of more than one perception which is grouped together to form a multiple layer neural network.

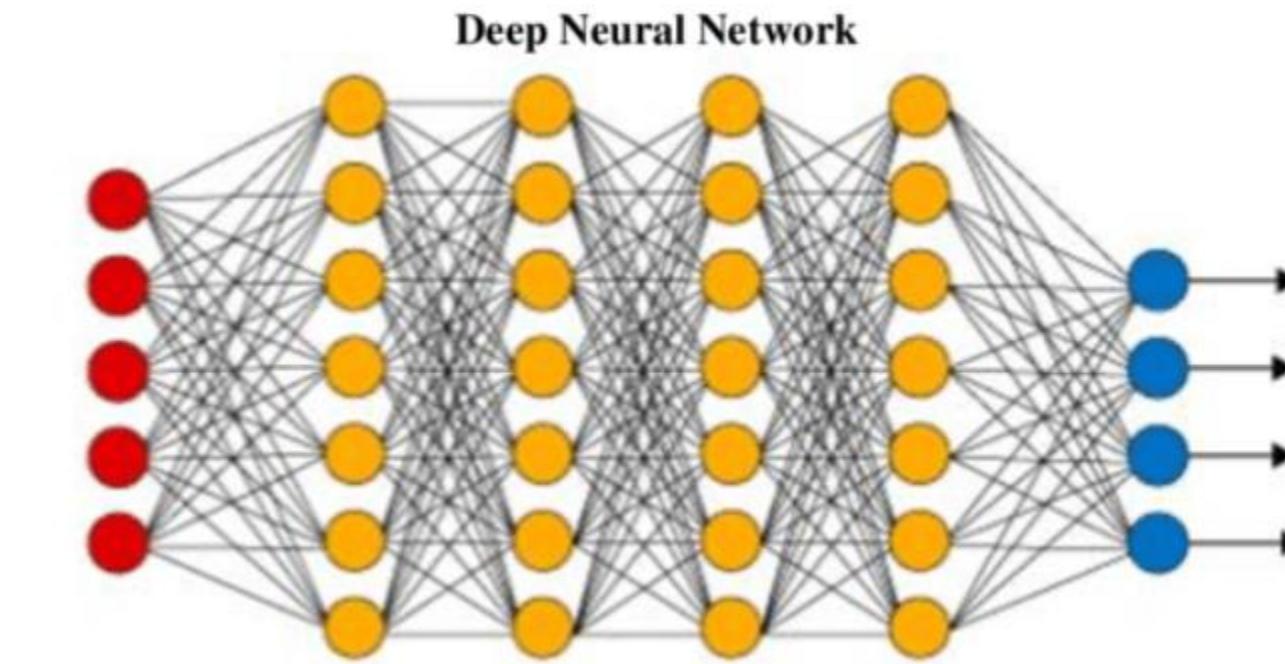


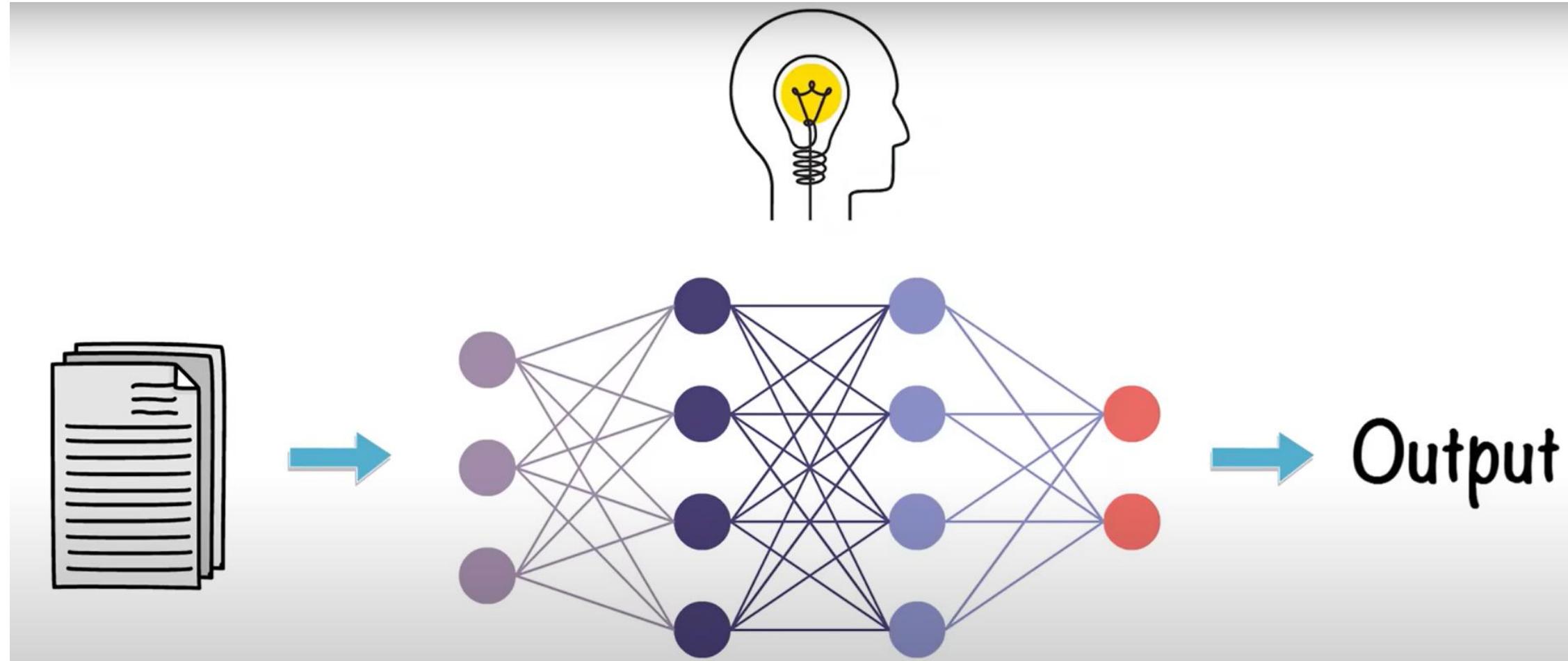
In the image, The Artificial Neural Network consists of four layers interconnected with each other:

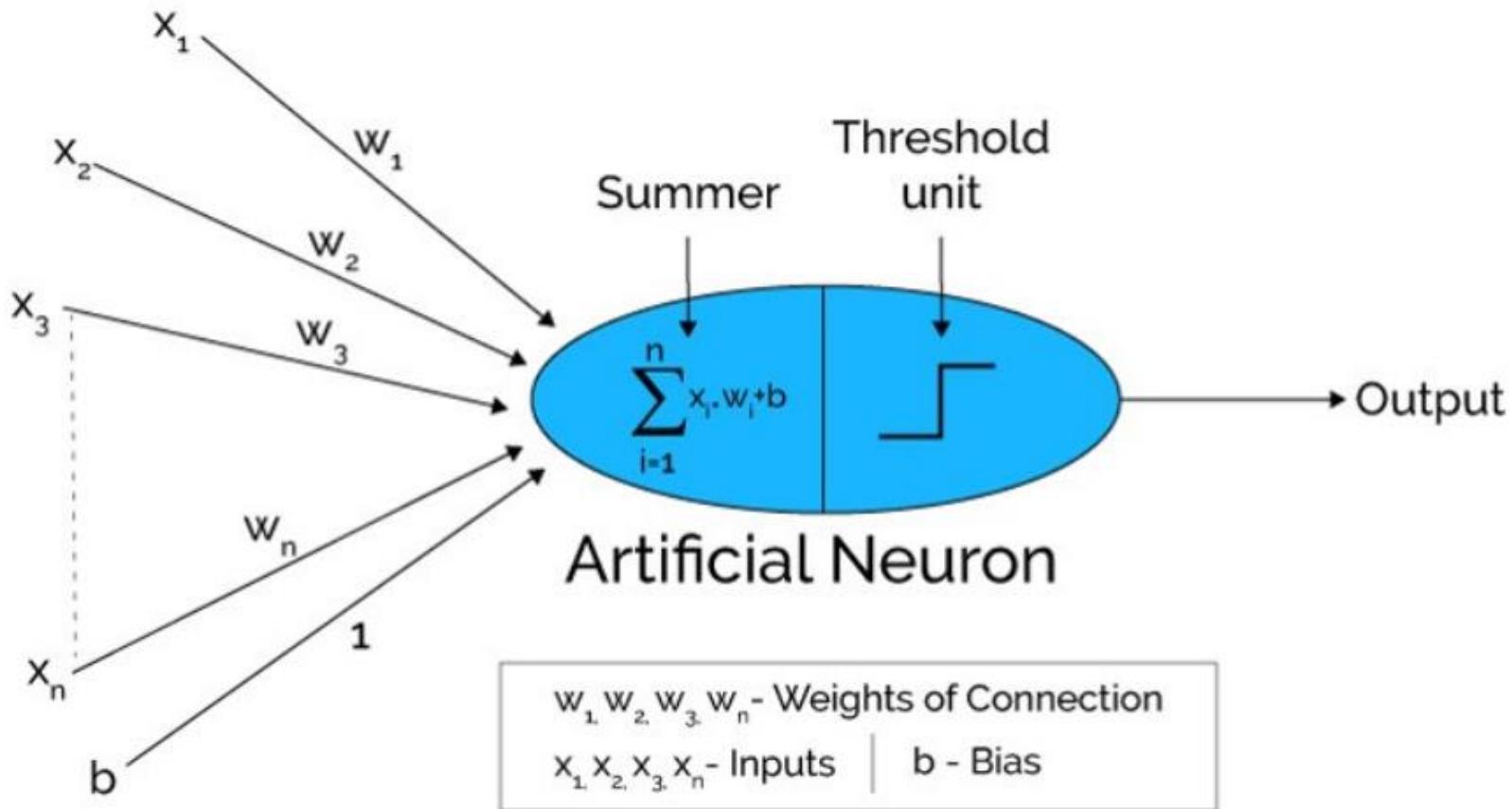
- An input layer, with 6 input nodes
- Hidden Layer 1, with 4 hidden nodes/4 perceptrons
- Hidden layer 2, with 4 hidden nodes
- Output layer with 1 output node

DEEP NEURAL NETWORK

A Deep Neural Network is a simple neural network that consists of multiple hidden layers.



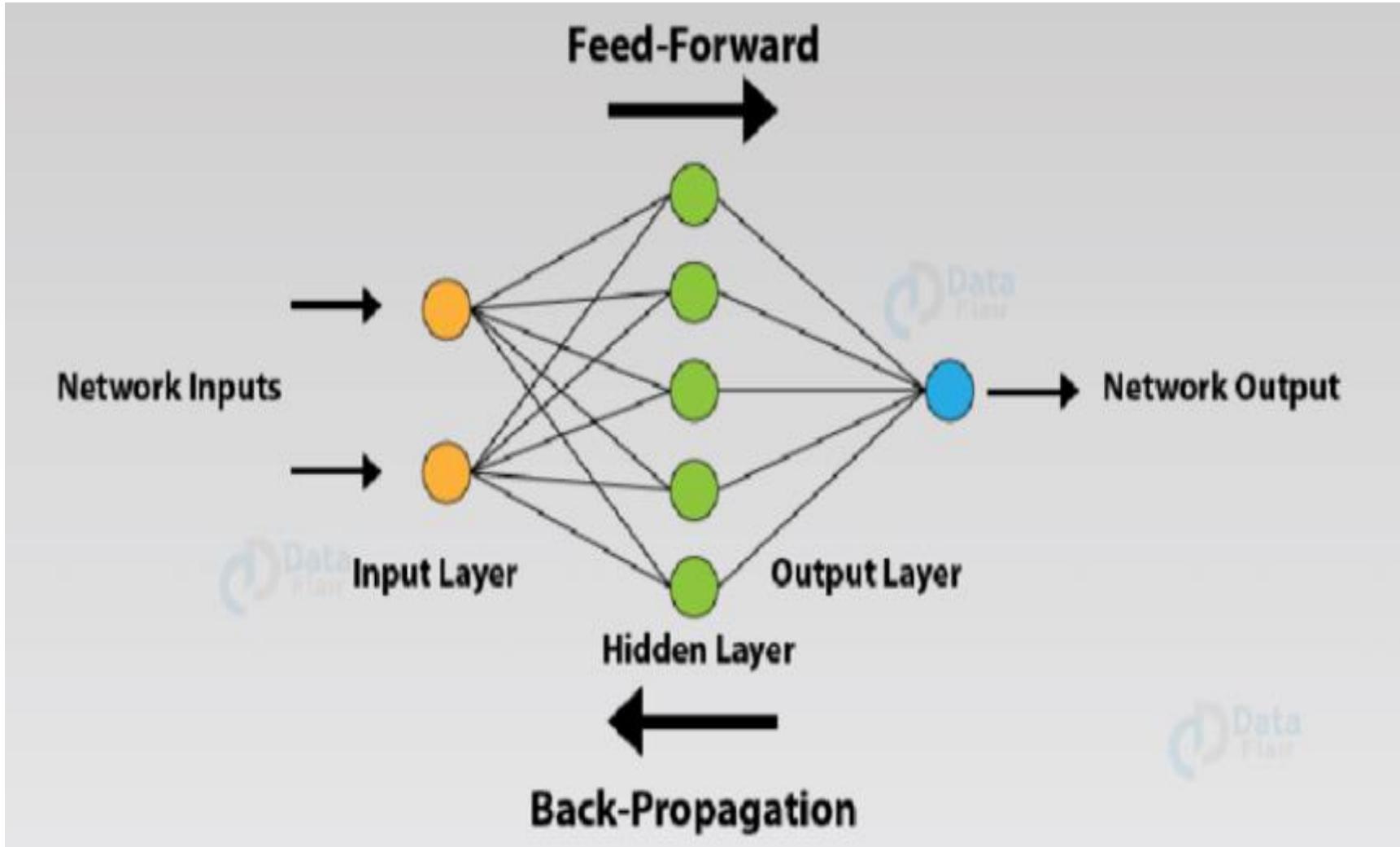




In neural networks, **weights** and **biases** are fundamental components that significantly influence the model's ability to learn patterns and make accurate predictions.

- **Neuron:** The basic unit of a neural network. Each neuron receives inputs, processes them, and produces an output.
- **Layers:** Neural networks are organized into layers:
 - Input Layer:** Receives the input data.
 - Hidden Layers:** Intermediate layers where computation occurs. There can be one or more hidden layers.
 - Output Layer:** Produces the final prediction or classification.
- **Weights and Biases:** Connections between neurons have weights that are adjusted during training. Each neuron also has a bias term to shift the activation function.
- **Activation Function:** Determines whether a neuron should be activated or not.

- Weights and biases together control the outputs of the neurons and the behavior of the network. Through training, these parameters are updated iteratively so the network can better predict the target outputs by minimizing the difference between the actual and predicted values.
- **weights determine the significance of inputs** in a neural network, while **biases help the network adjust its output more flexibly**. Both are key to ensuring the model can accurately learn from and generalize to new data.



- 1. In the first step, Input units are passed i.e data is passed with some weights attached to it to the hidden layer.** We can have any number of hidden layers. In the above image inputs $x_1, x_2, x_3, \dots, x_n$ is passed.
- 2. Each hidden layer consists of neurons.** All the inputs are connected to each neuron.
- 3. After passing on the inputs, all the computation is performed in the hidden layer.**

- Computation performed in hidden layers are done in two steps which are as follows :
- First of all, **all the inputs are multiplied by their weights.** Weight is the gradient or coefficient of each variable. It shows the strength of the particular input. After assigning the weights, a bias variable is added. **Bias** is a constant that helps the model to fit in the best way possible.

$$Z_1 = W_1 * In_1 + W_2 * In_2 + W_3 * In_3 + W_4 * In_4 + W_5 * In_5 + b$$

- W_1, W_2, W_3, W_4, W_5 are the weights assigned to the inputs $In_1, In_2, In_3, In_4, In_5$, and b is the bias.
- Then in the second step, the **activation function is applied to the linear equation Z1.** The activation function is a nonlinear transformation that is applied to the input before sending it to the next layer of neurons. The importance of the activation function is to inculcate nonlinearity in the model.

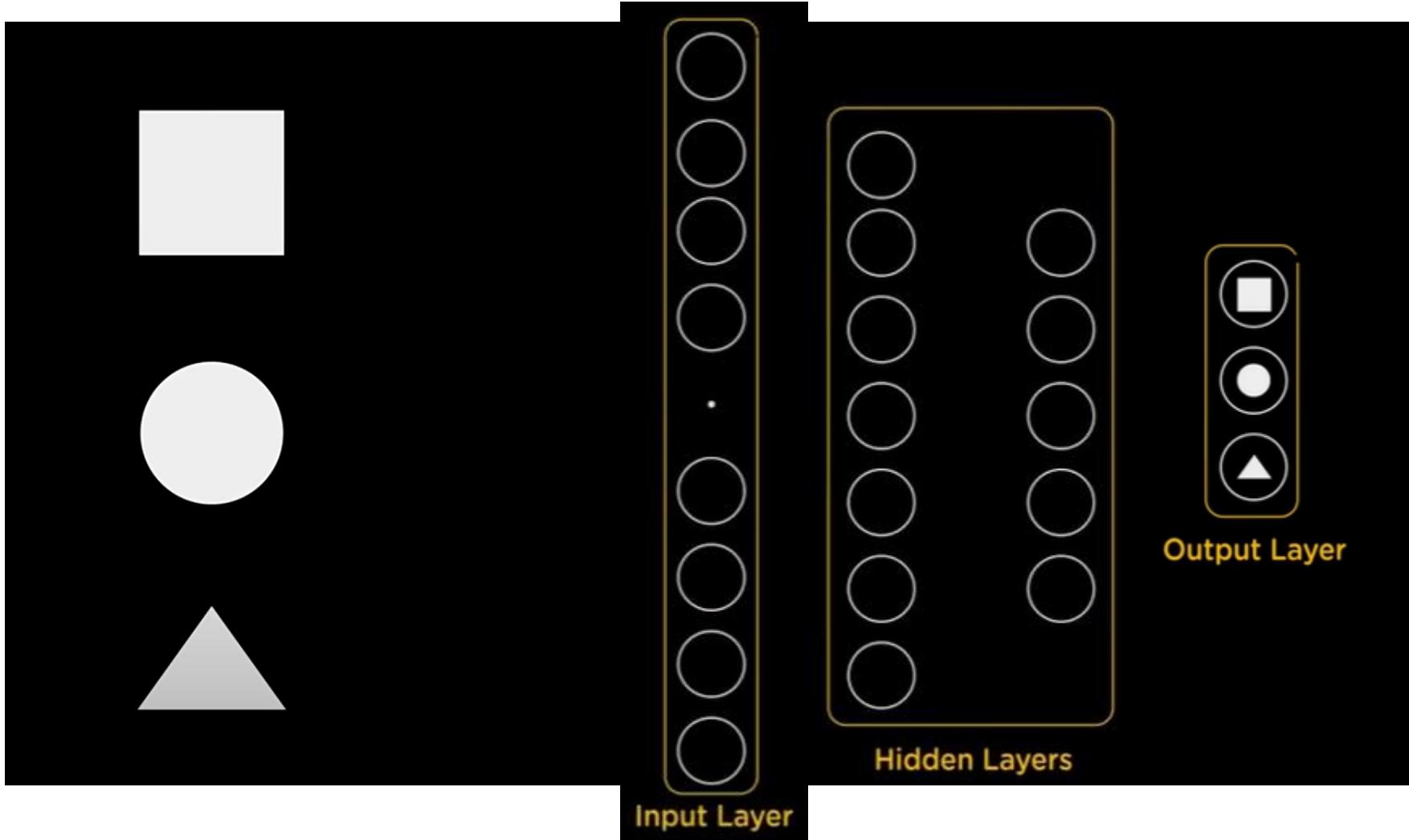
4. The whole process described in point 3 is performed in each hidden layer. After passing through every hidden layer, **we move to the last layer i.e our output layer which gives us the final output.**

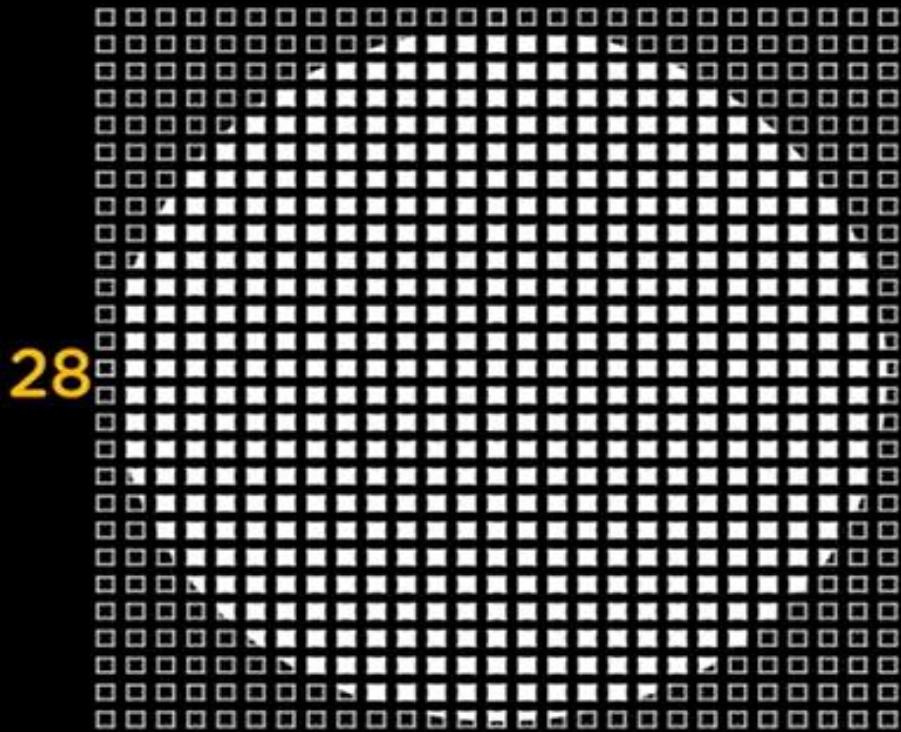
The process explained above is known as Forwarding Propagation.

5. After getting the predictions from the output layer, the **error is calculated i.e the difference between the actual and the predicted output.**

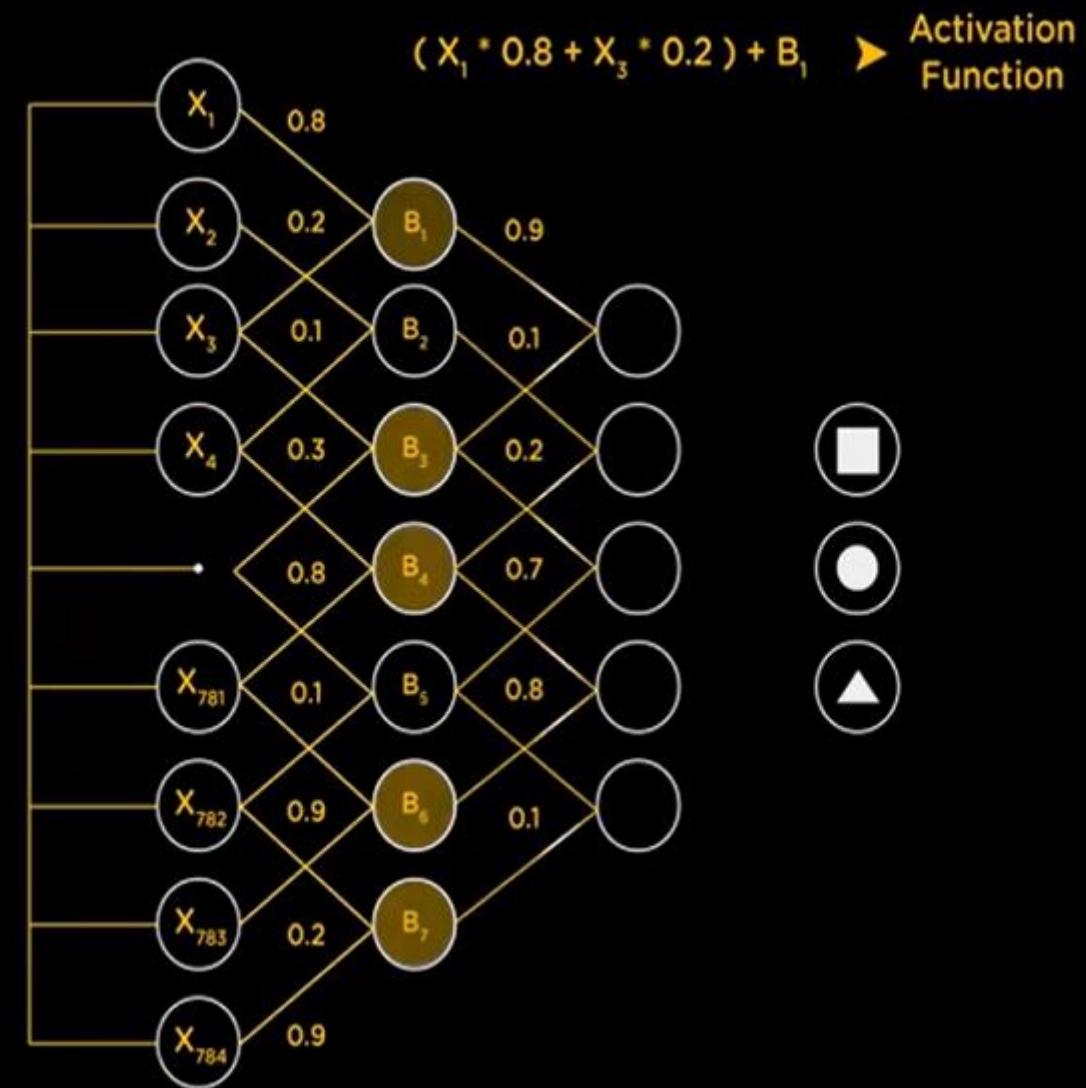
If the error is large, then the steps are taken to minimize the error and for the same purpose, **Back Propagation is performed.**

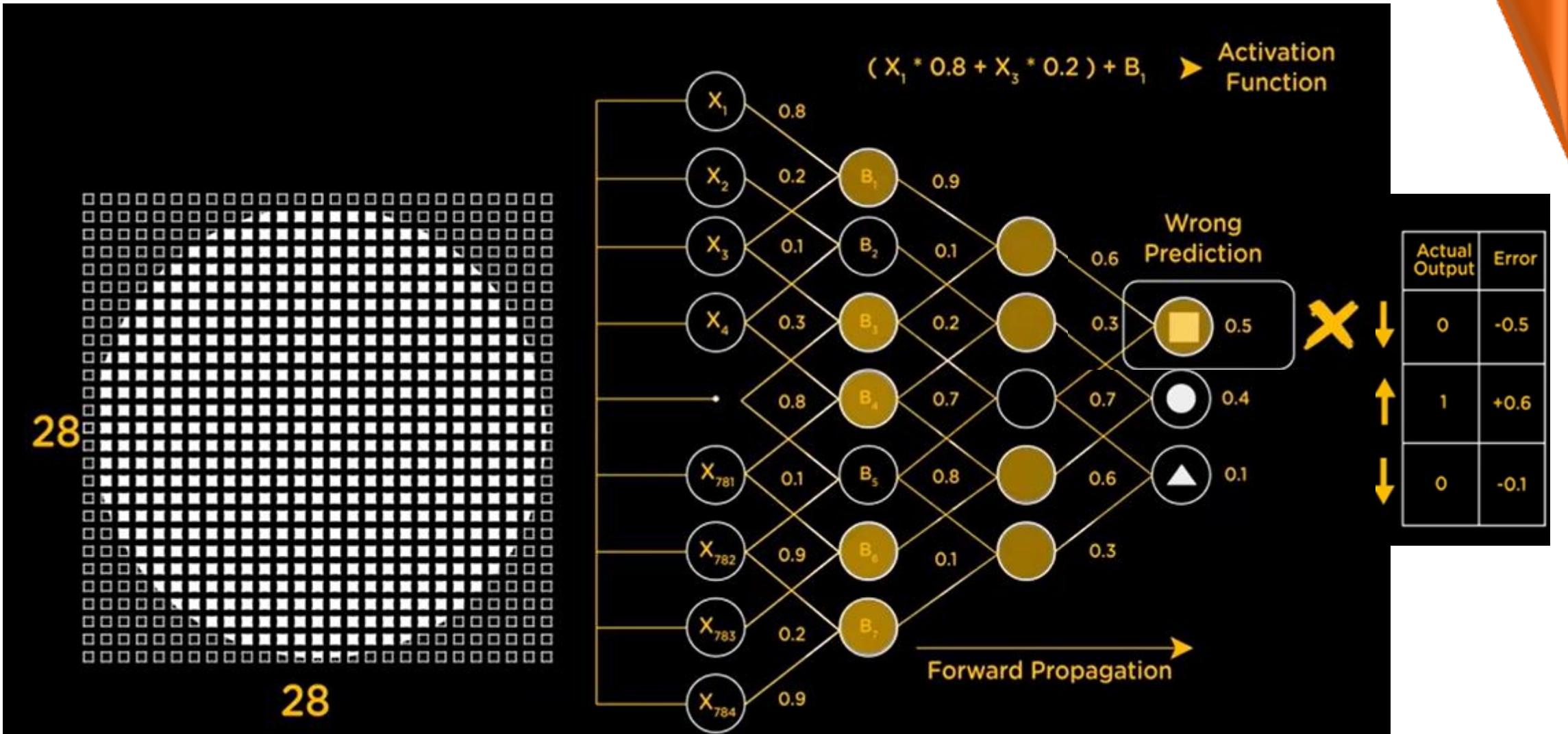
EXAMPLE-1

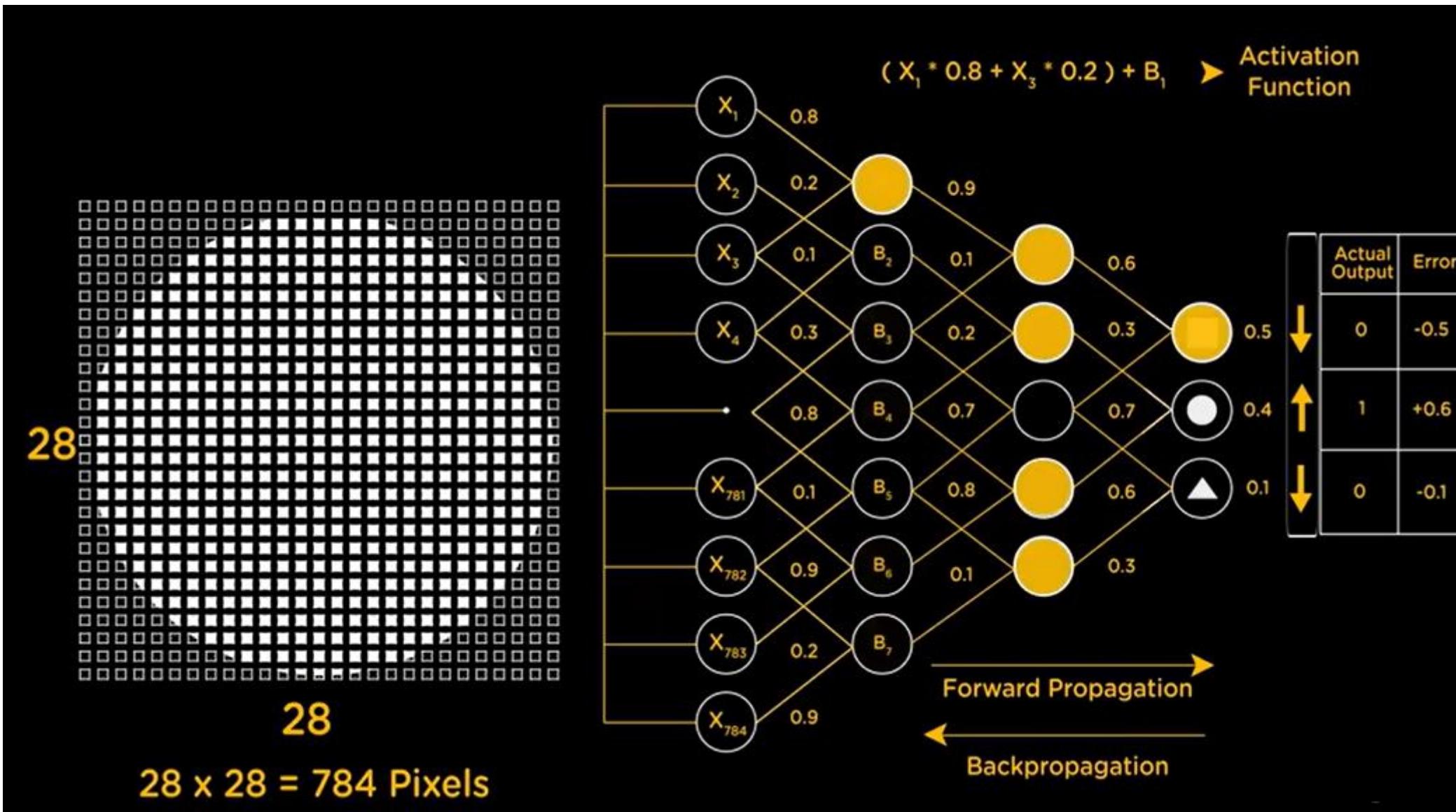


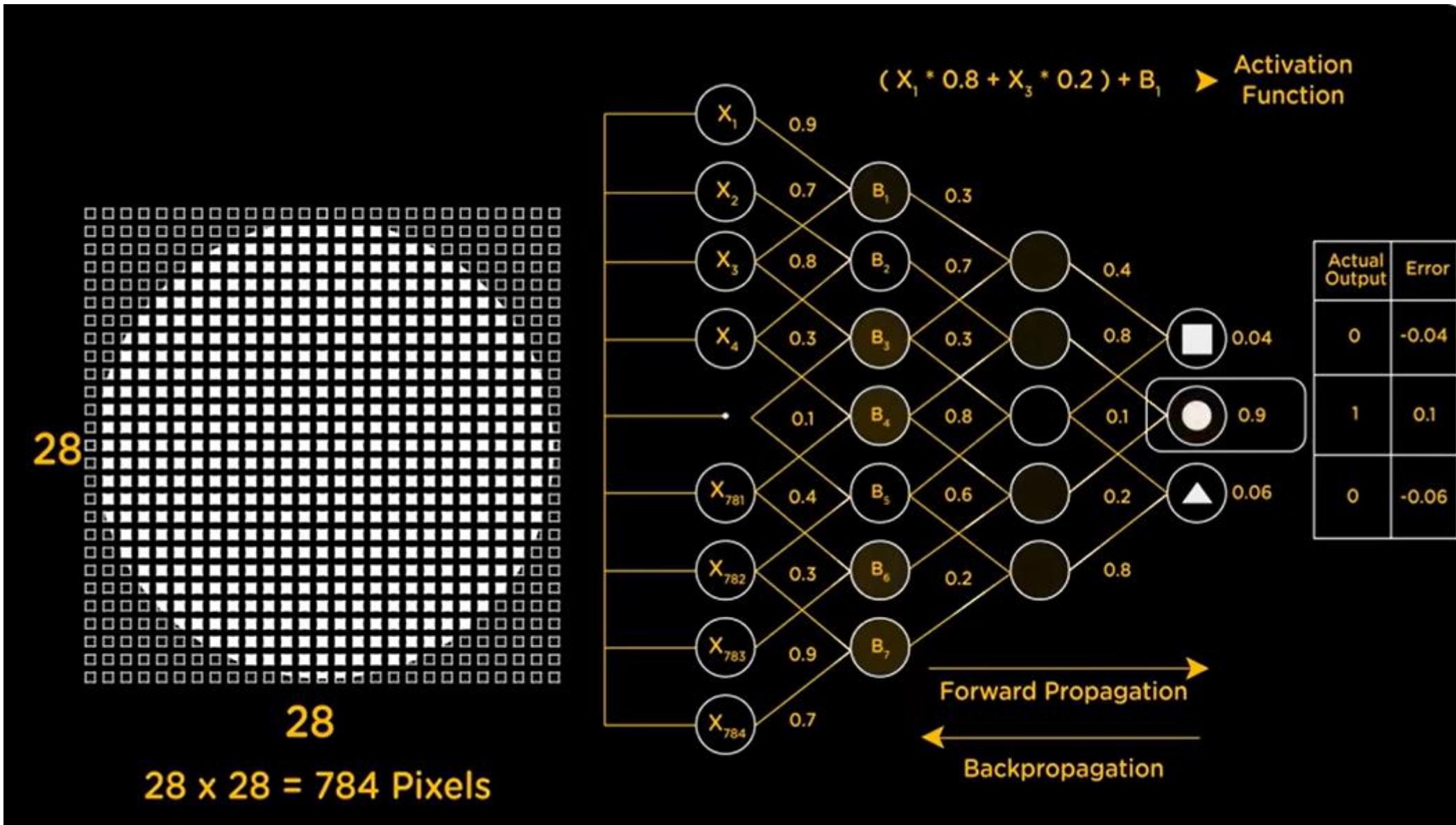


$28 \times 28 = 784$ Pixels



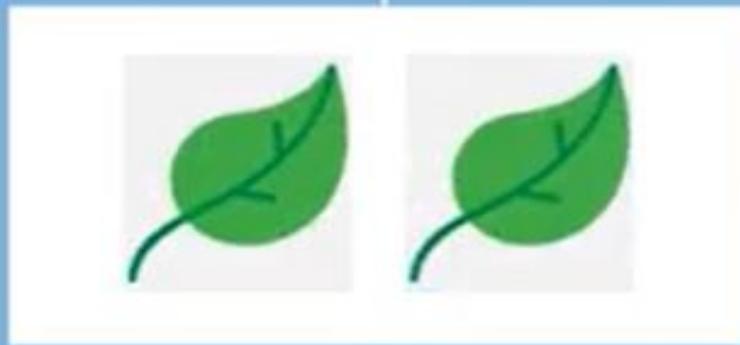




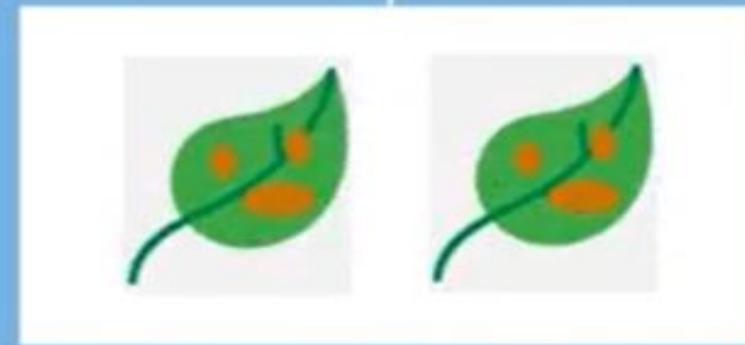


EXAMPLE-2

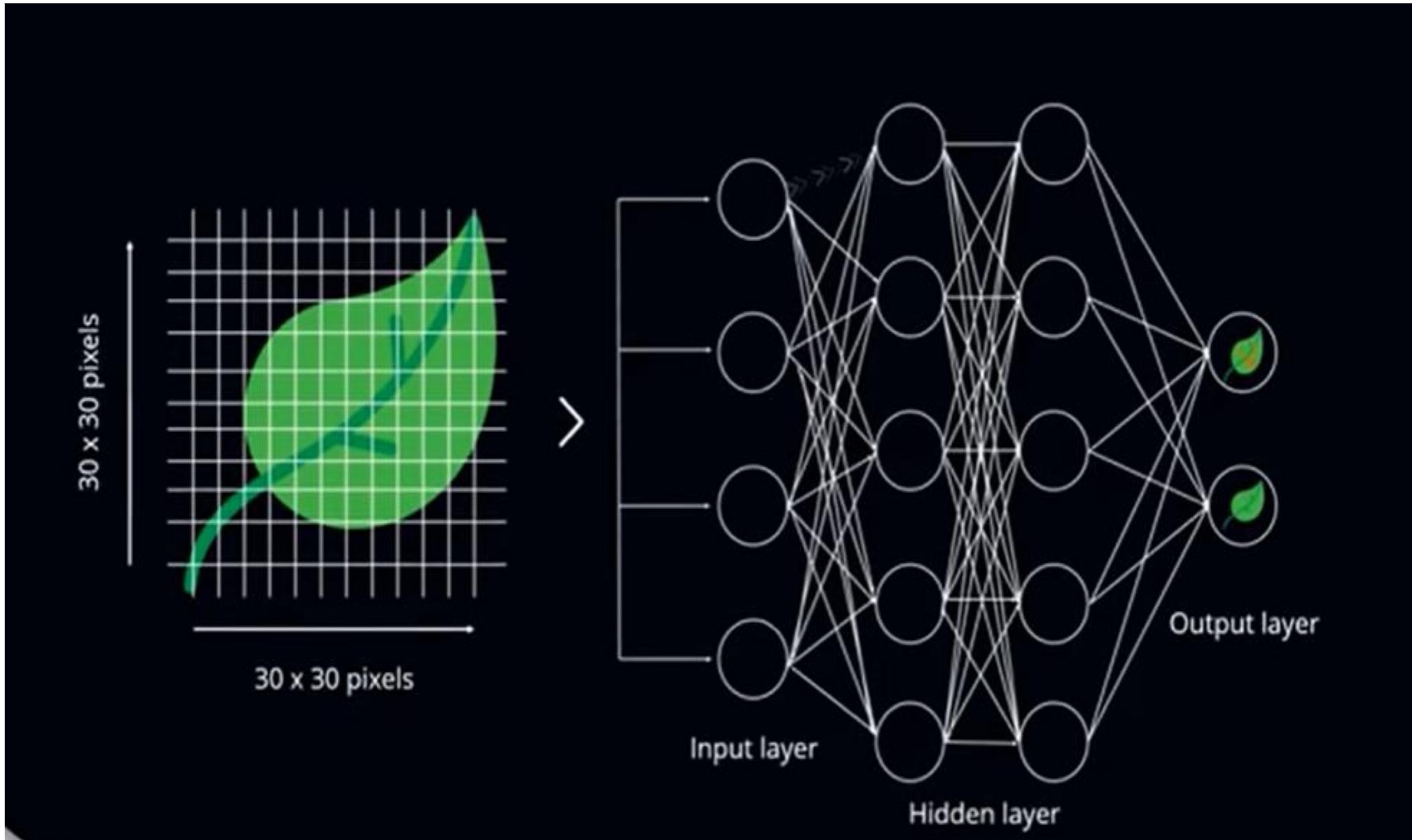
Problem Statement: Classify leaf images as either diseased or non-diseased.

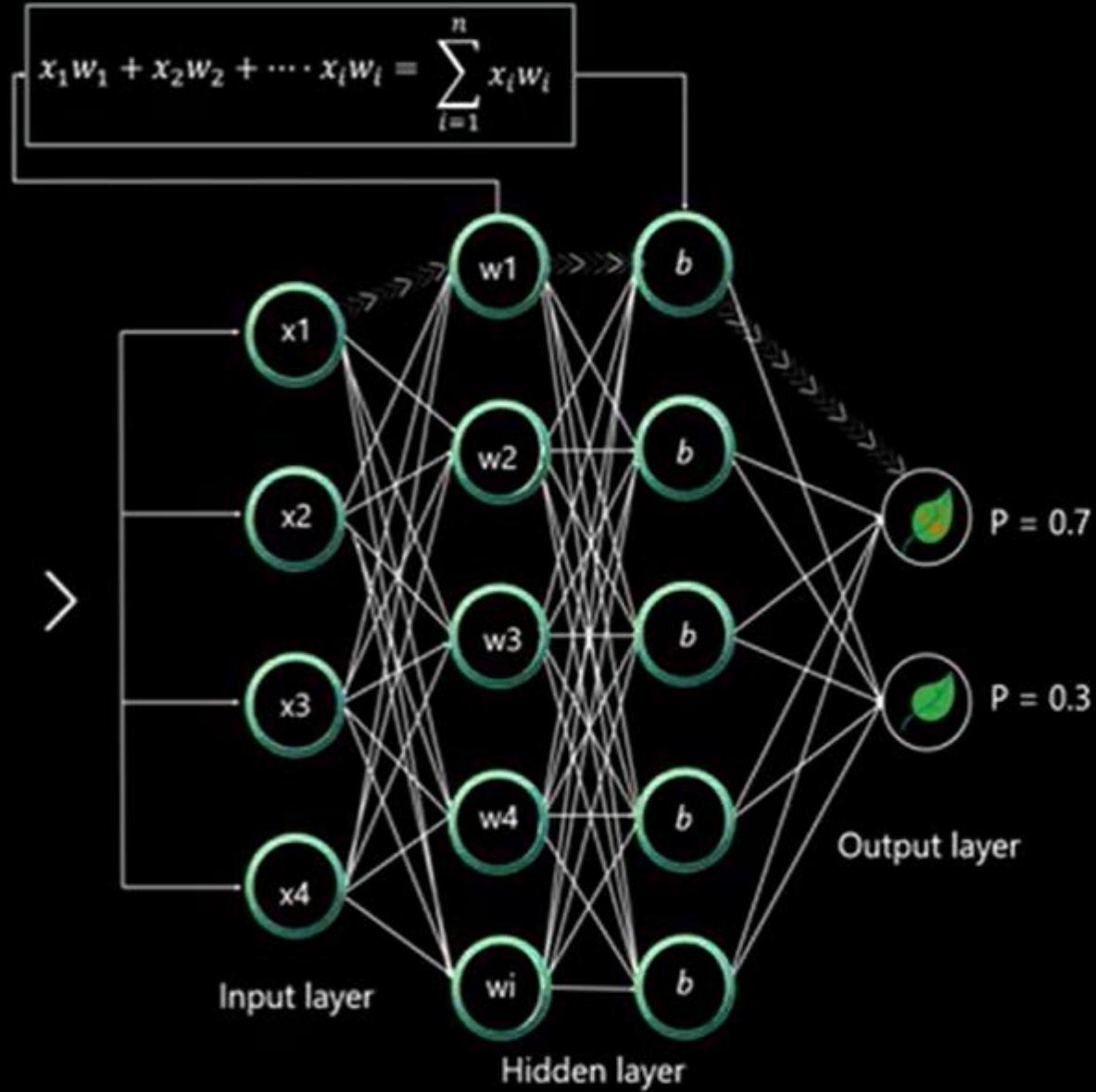
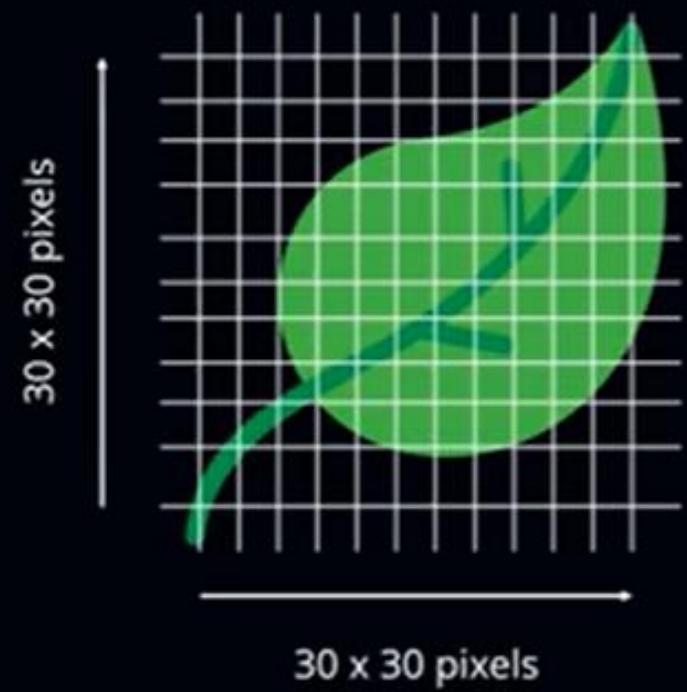


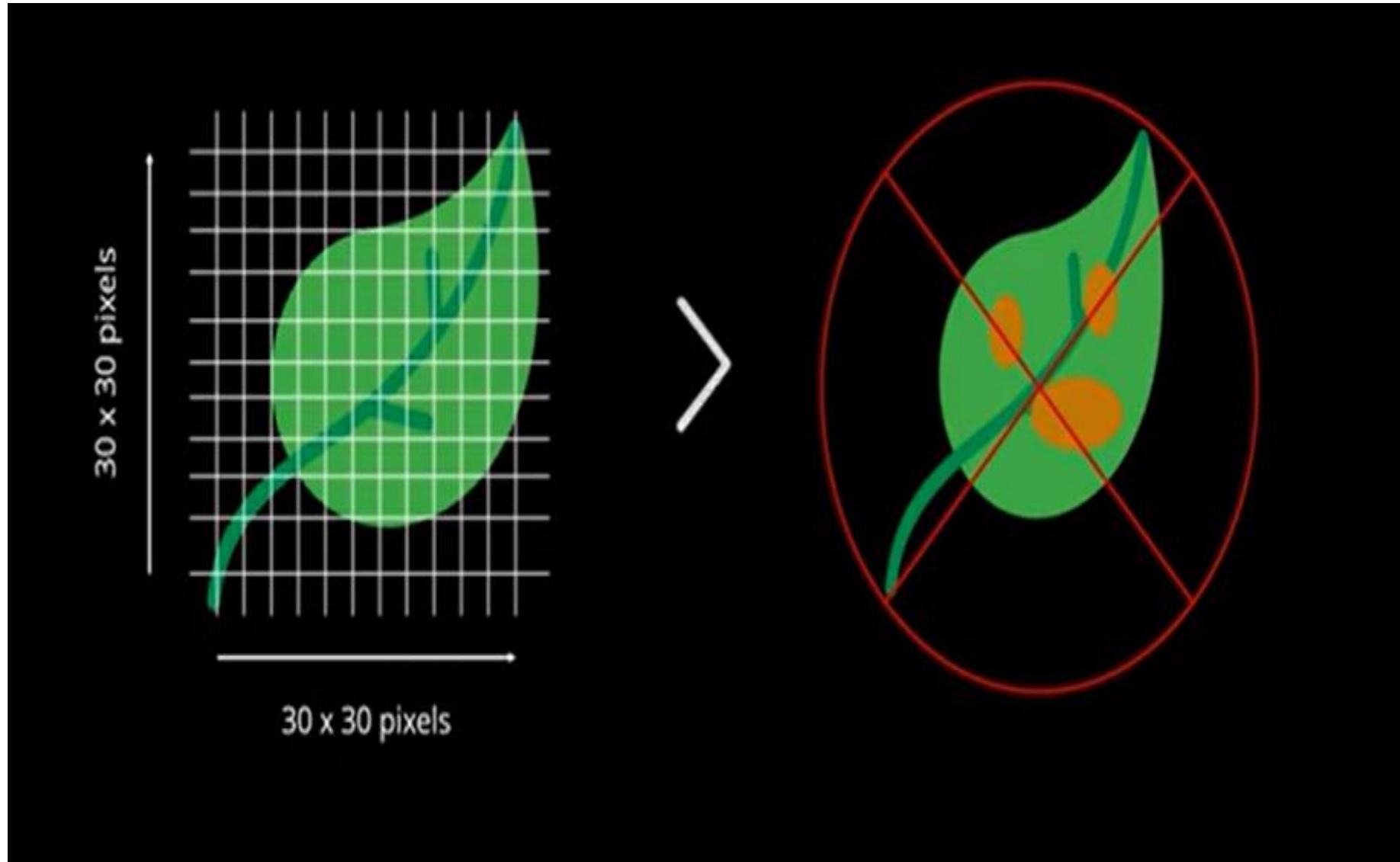
Non-diseased leaves

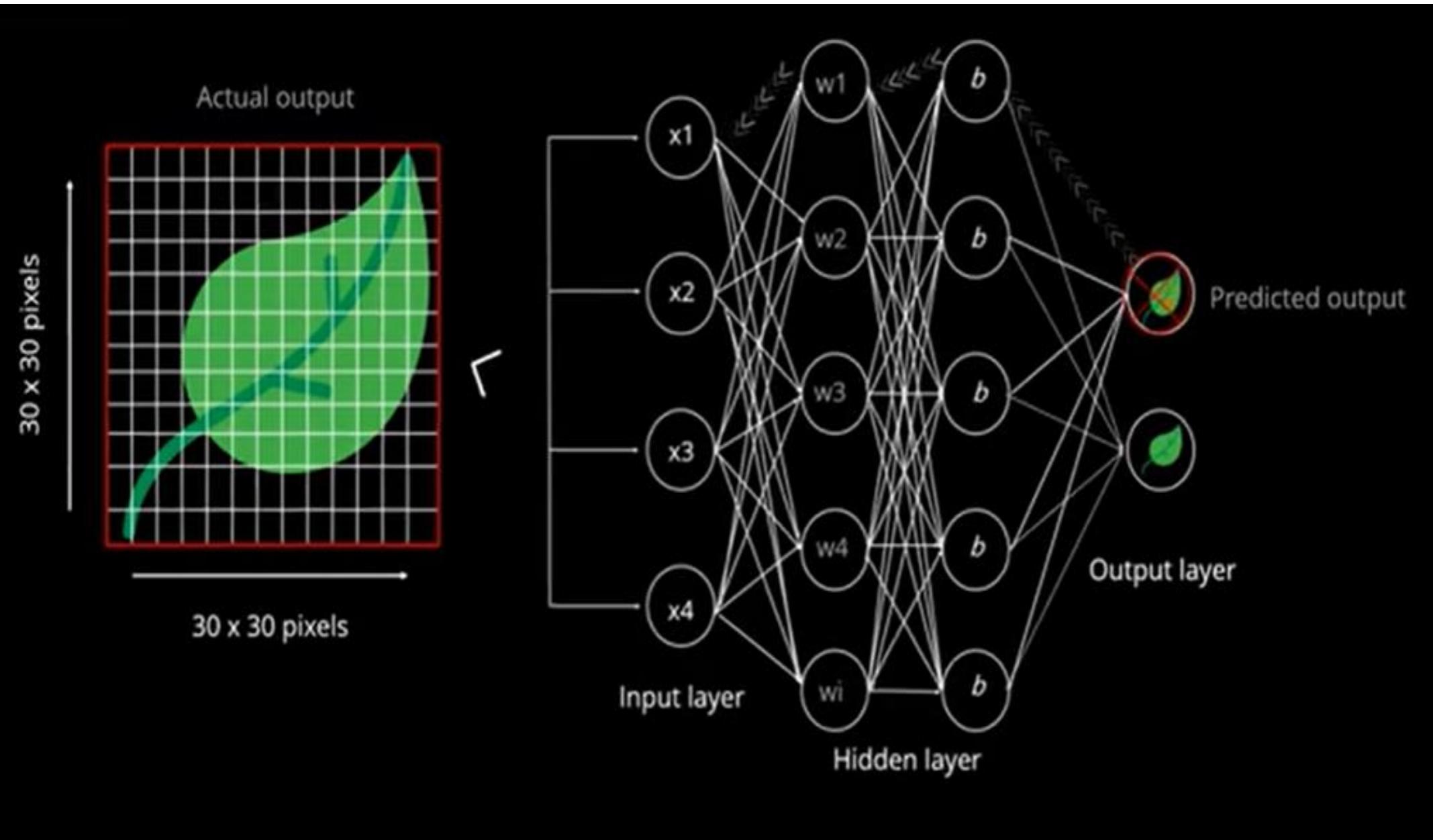


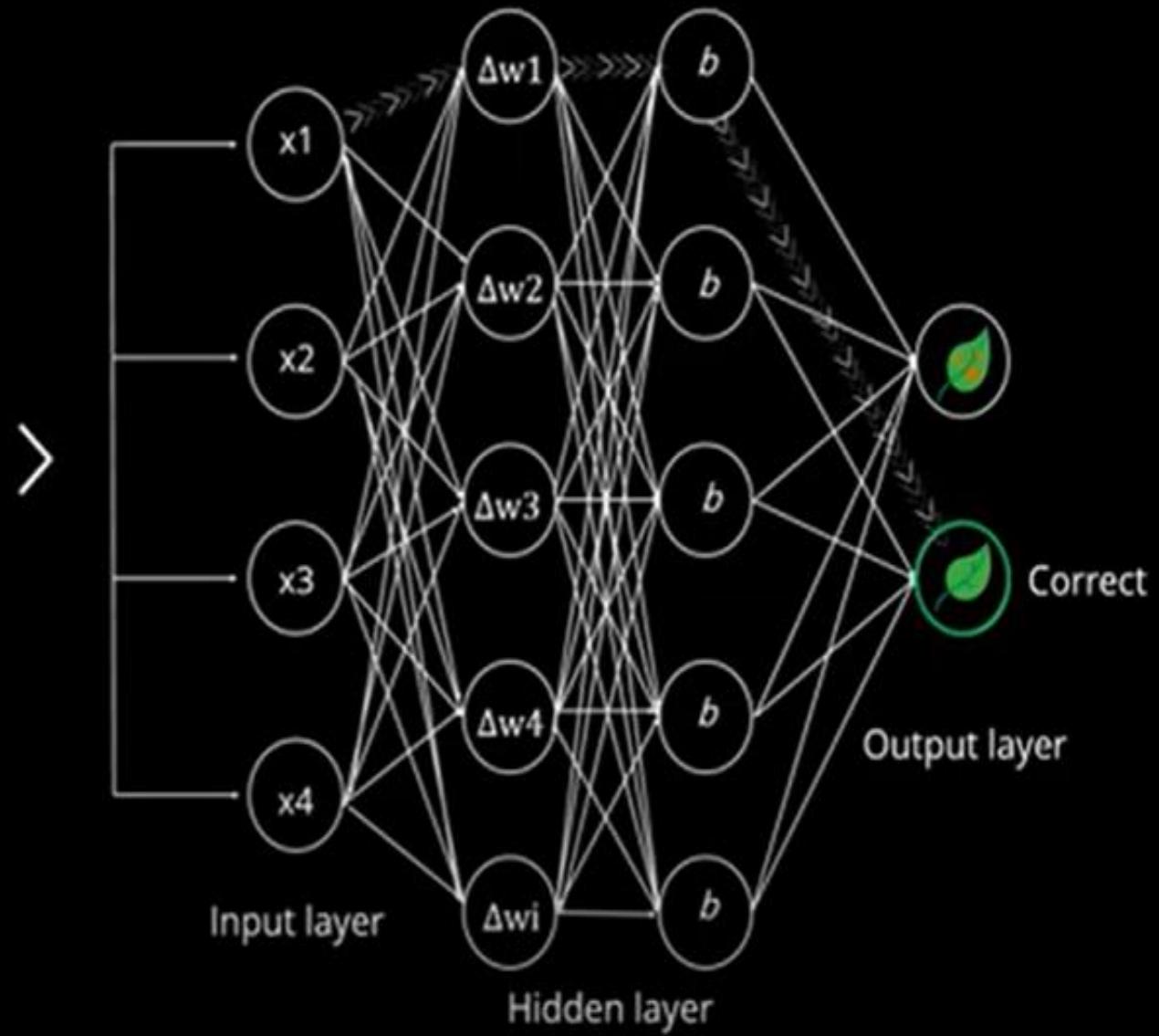
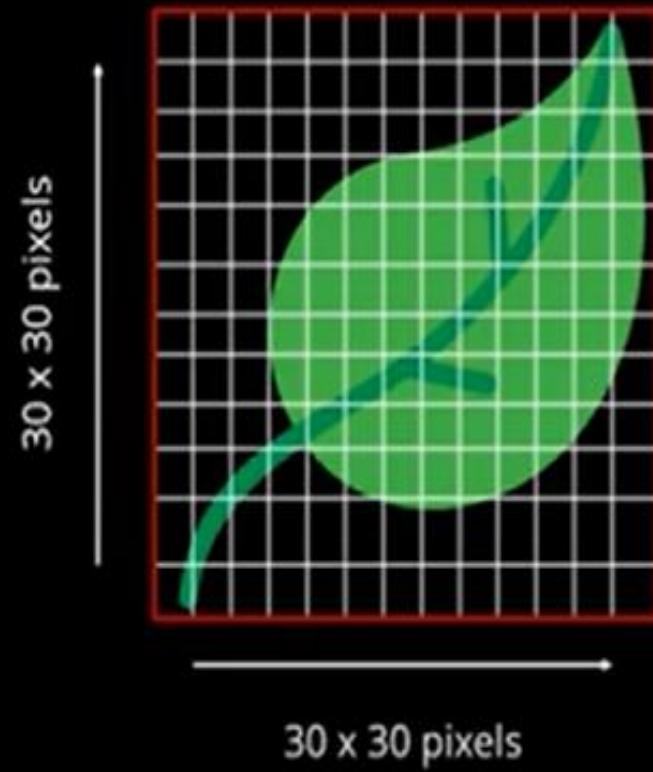
Diseased leaves



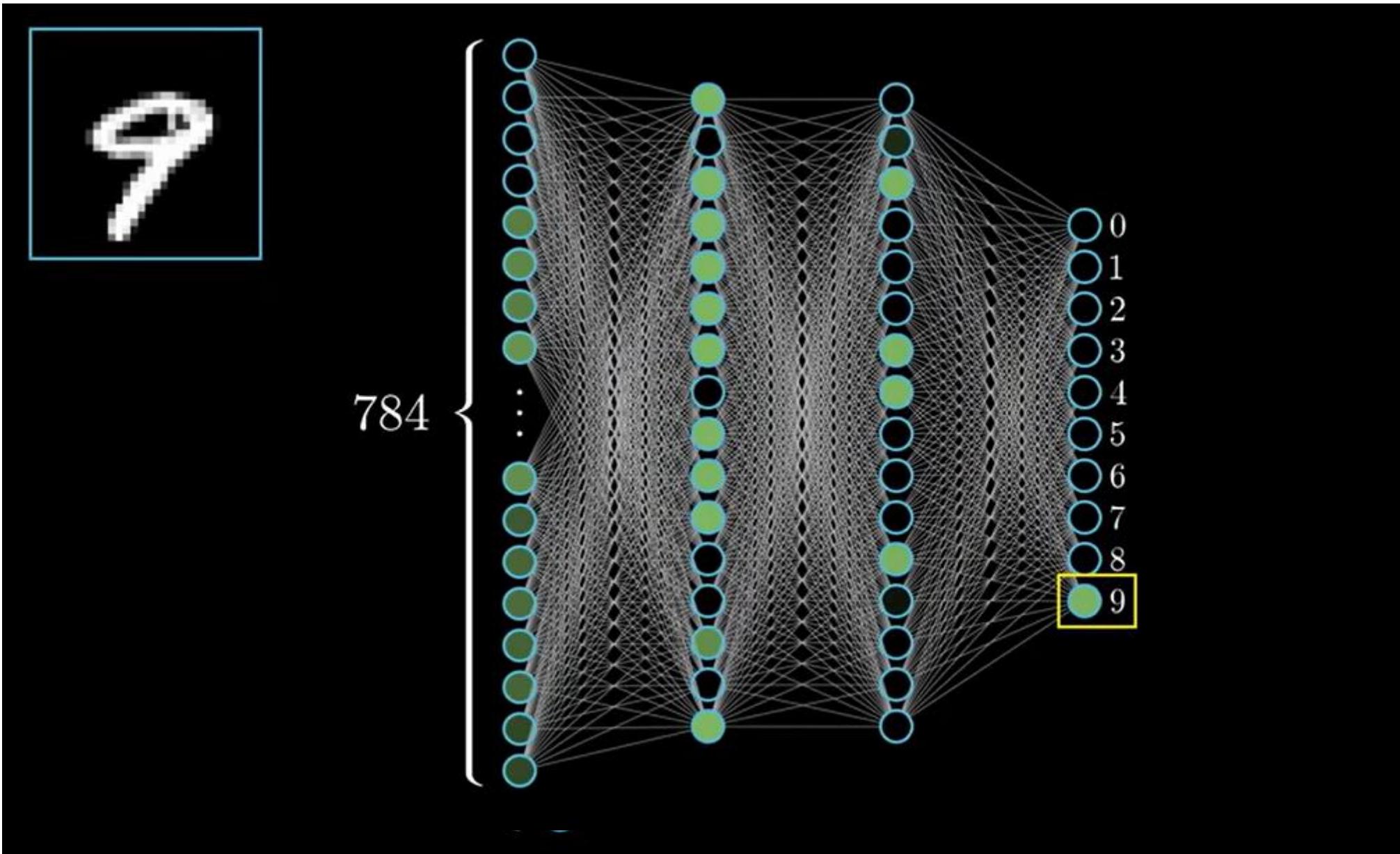






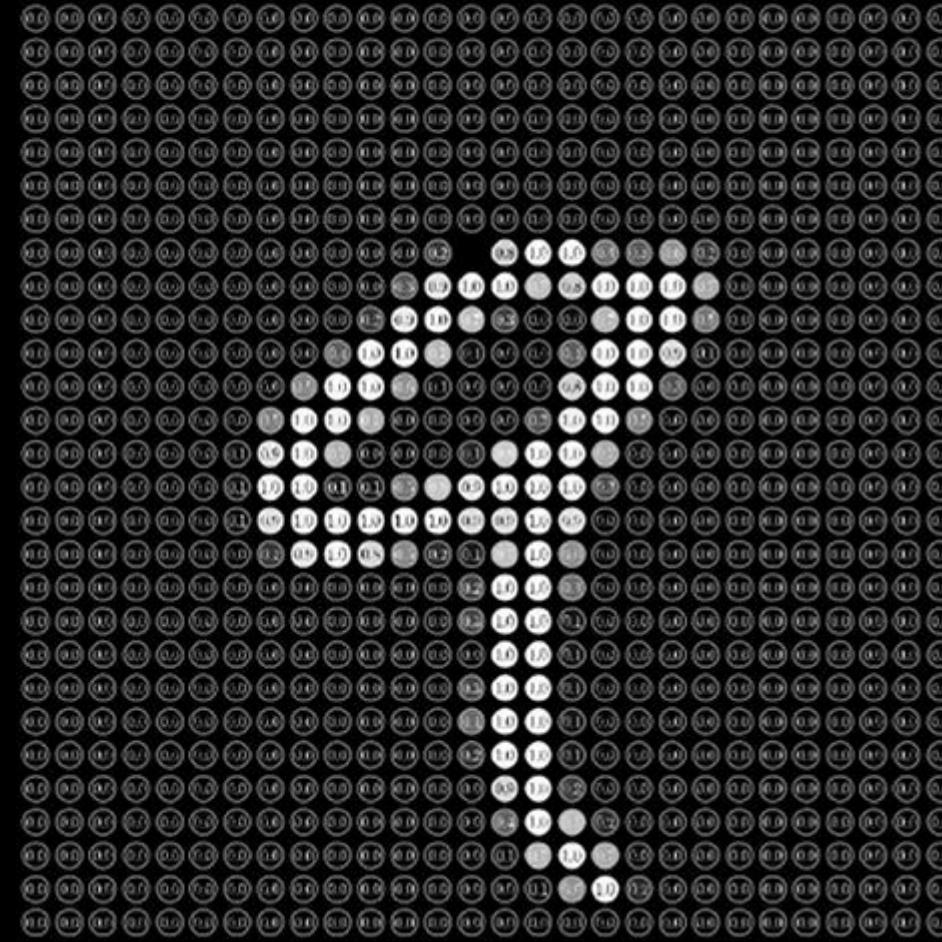


EXAMPLE-3



28

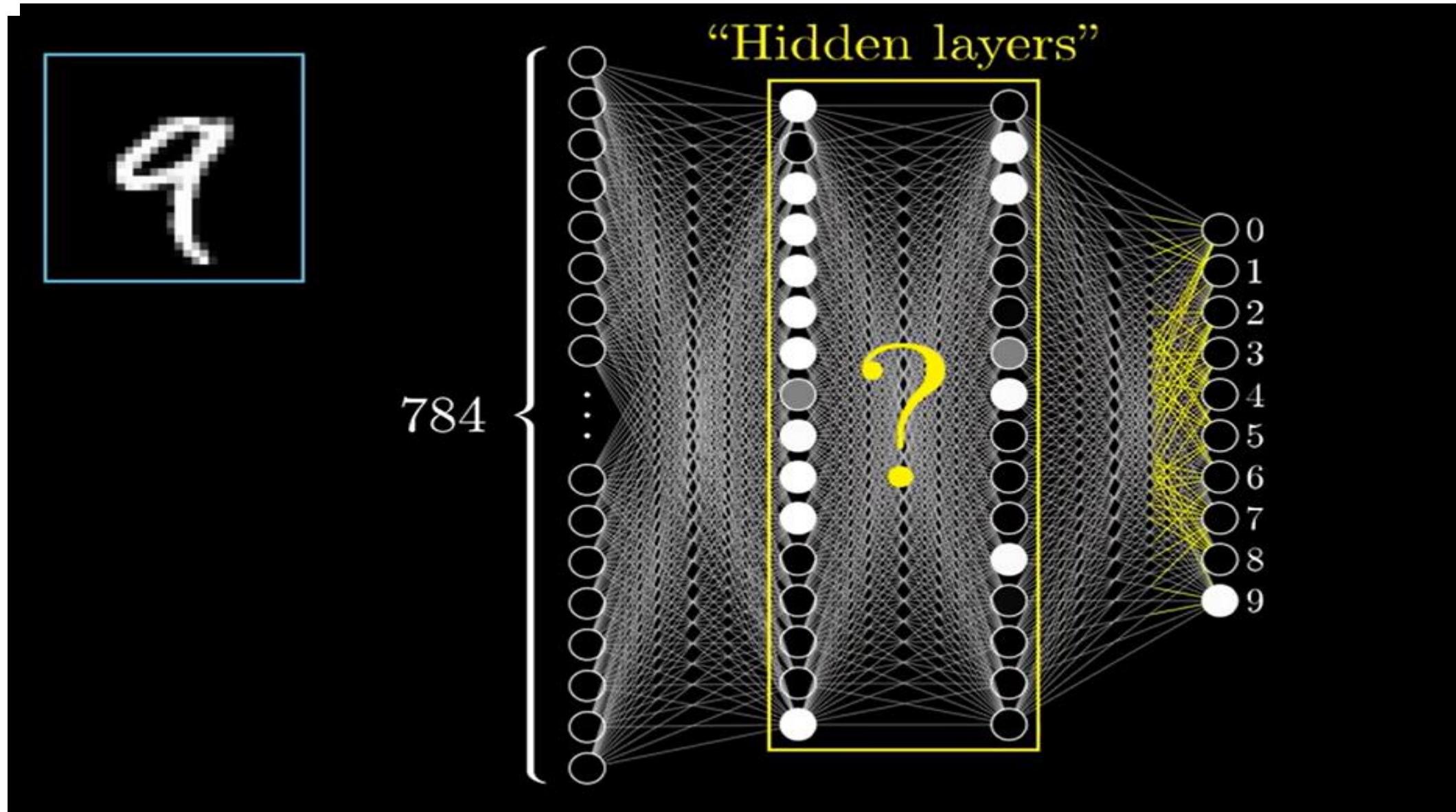
28



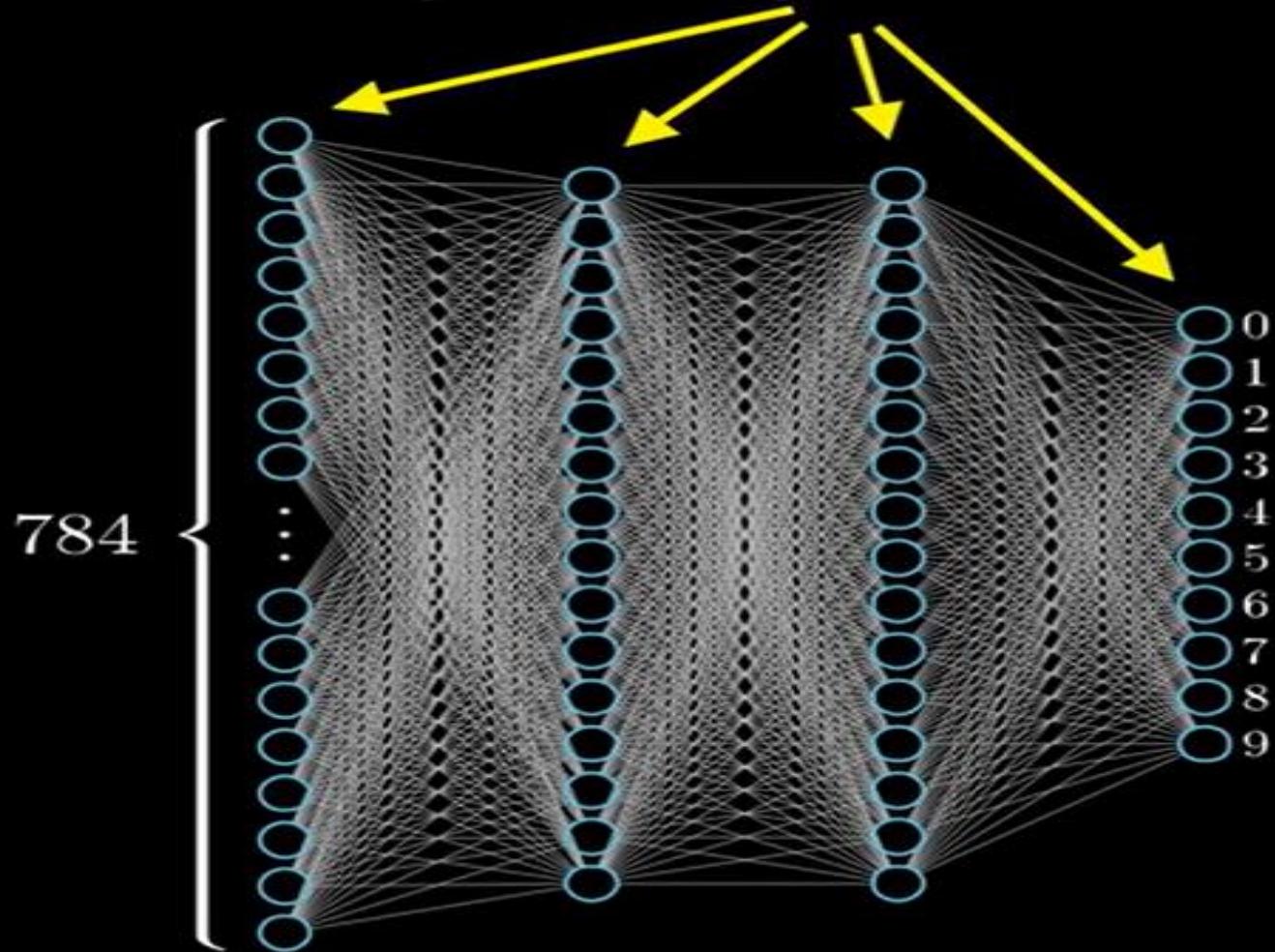
$$28 \times 28 = 784$$

0.82

“Activation”



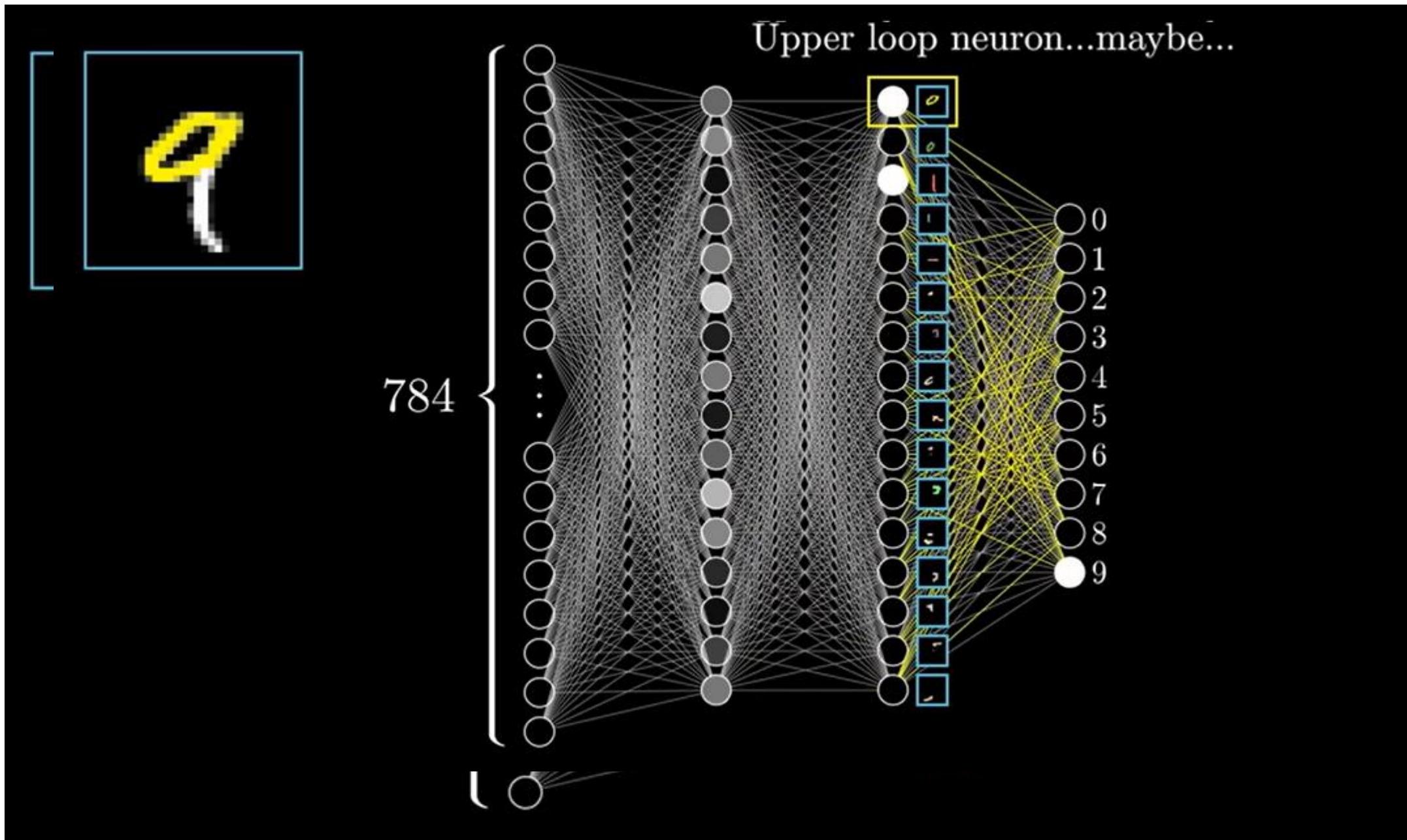
Why the layers?



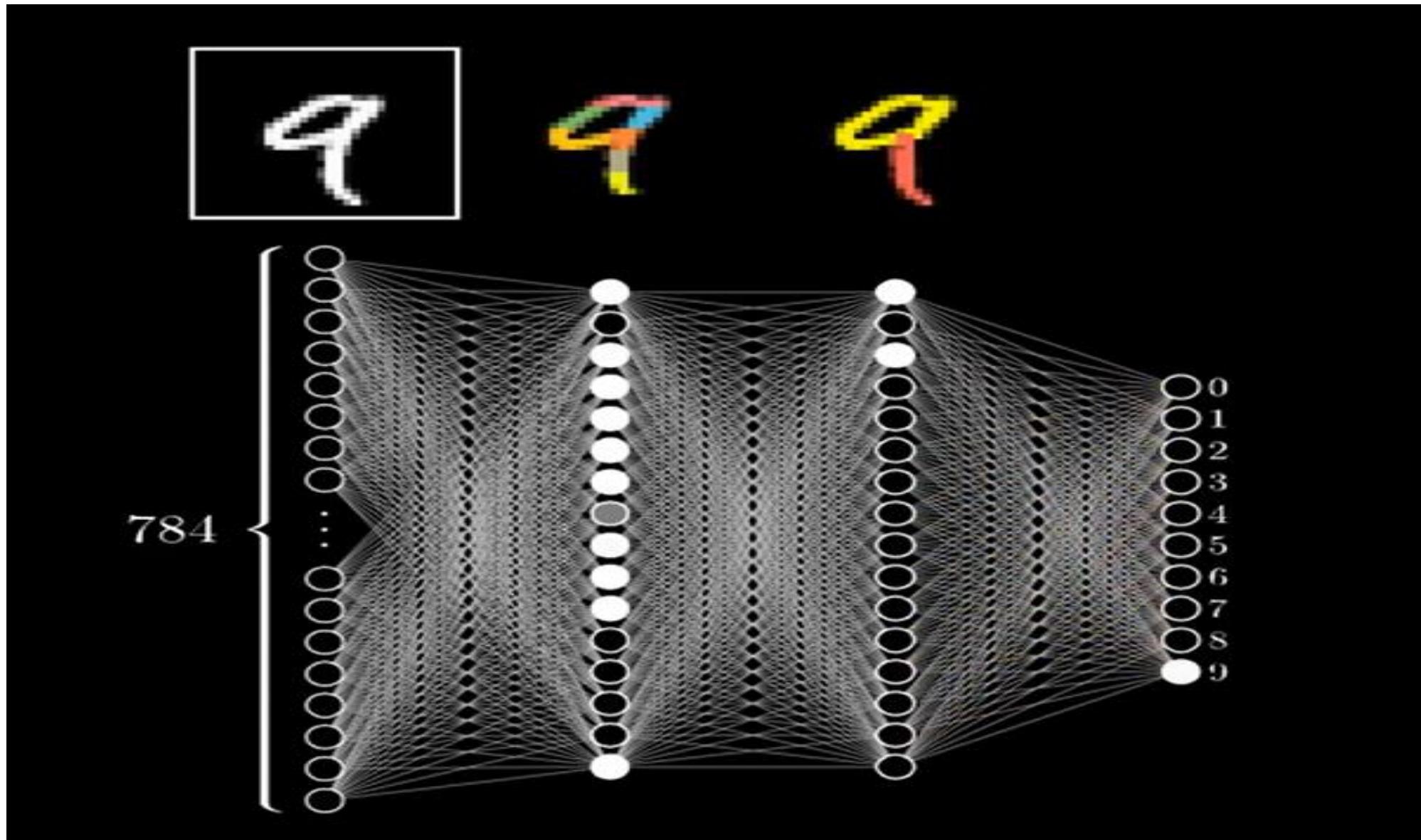
$$q = \textcolor{yellow}{o} + \textcolor{red}{l}$$

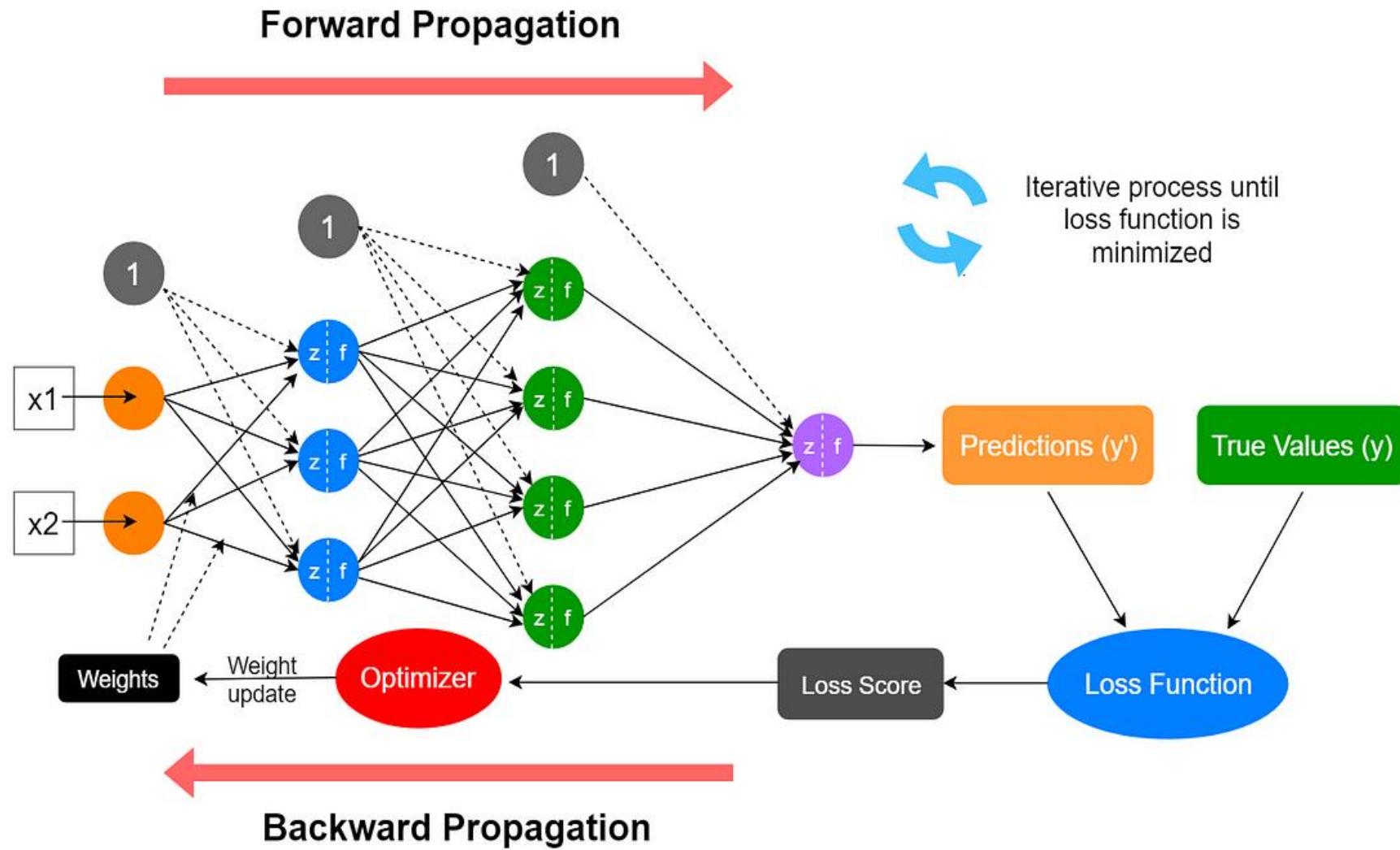
$$g = \textcolor{yellow}{o} + \textcolor{green}{o}$$

$$y = \textcolor{red}{l} + \textcolor{blue}{f} + \textcolor{brown}{r}$$



$$\begin{aligned} \text{Image A} &= \text{Image B} + \text{Image C} + \text{Image D} + \text{Image E} + \text{Image F} \\ \text{Image G} &= \text{Image H} + \text{Image I} + \text{Image J} \end{aligned}$$

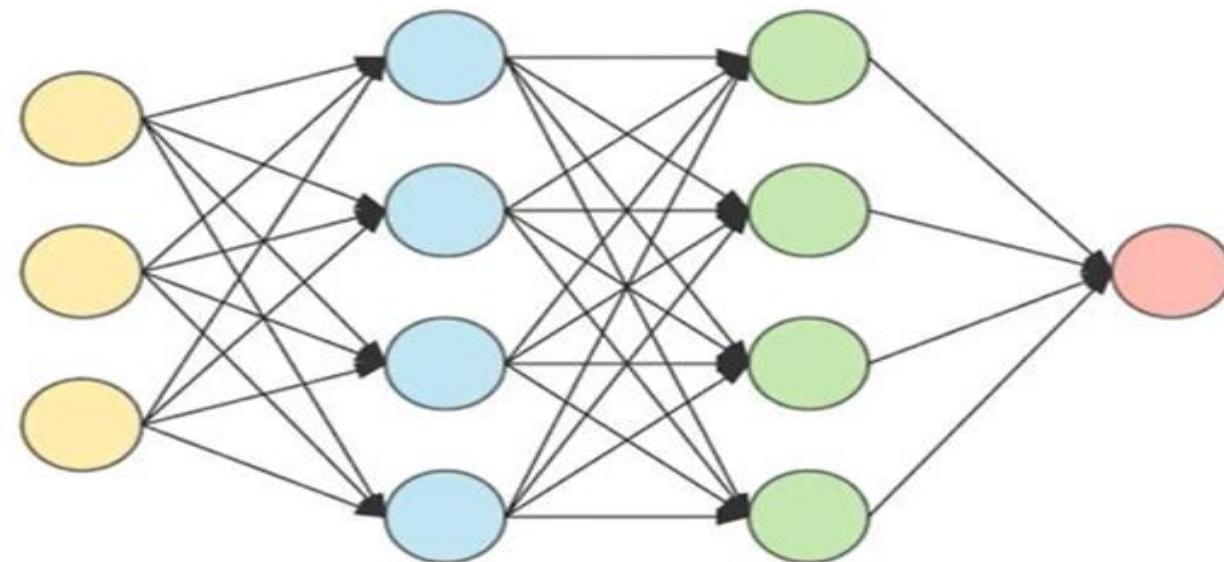




Forward Propagation

- The process of computing the output of a neural network. It involves:
 1. Multiplying inputs by weights.
 2. Adding biases.
 3. Applying the activation function to produce the output.

Activation Functions in Neural Networks



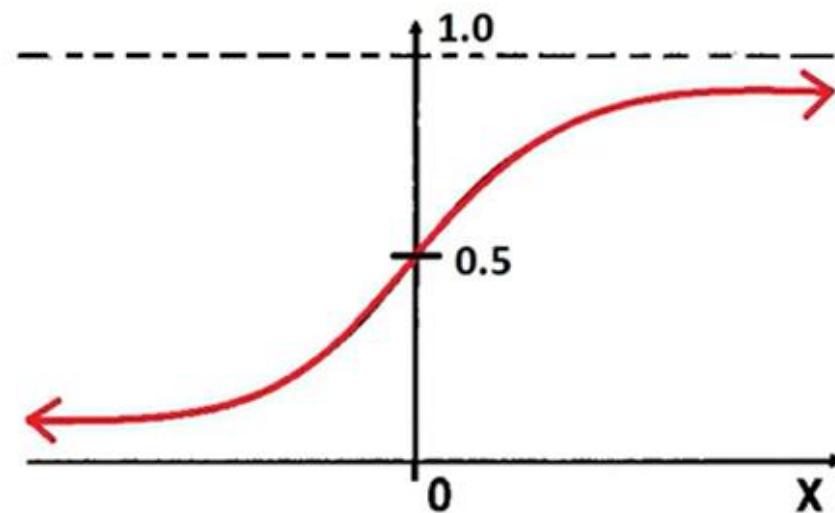
Why do we need Activation functions?

Activation functions are crucial in neural networks because they introduce **non-linearity** into the model, enabling the network to learn and model complex patterns in the data. Without activation functions, a neural network would essentially behave like a linear regression model, limiting its ability to capture the true underlying structure in most real-world problems.

Common Activation Functions

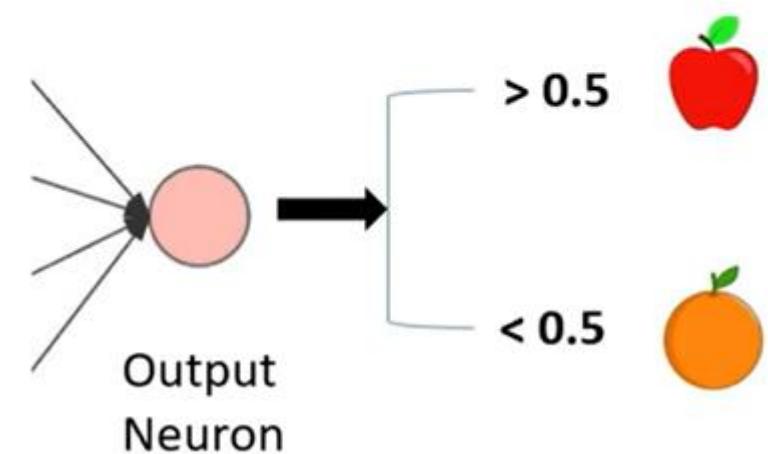
- **Sigmoid:** Maps inputs to the range (0, 1). It's used in binary classification tasks but can suffer from vanishing gradient problems in deep networks.
- **Tanh (Hyperbolic Tangent):** Maps inputs to the range (-1, 1), centering the output, which can sometimes lead to better convergence than sigmoid. However, it also suffers from vanishing gradient problems.
- **ReLU (Rectified Linear Unit):** Introduces non-linearity by outputting the input directly if it's positive and zero otherwise. It's widely used in deep learning due to its simplicity and effectiveness. It also helps with the vanishing gradient problem.
- **Softmax:** Typically used in the output layer for multi-class classification problems. It converts logits (raw output values) into probabilities that sum to 1 across the classes.

Sigmoid Function

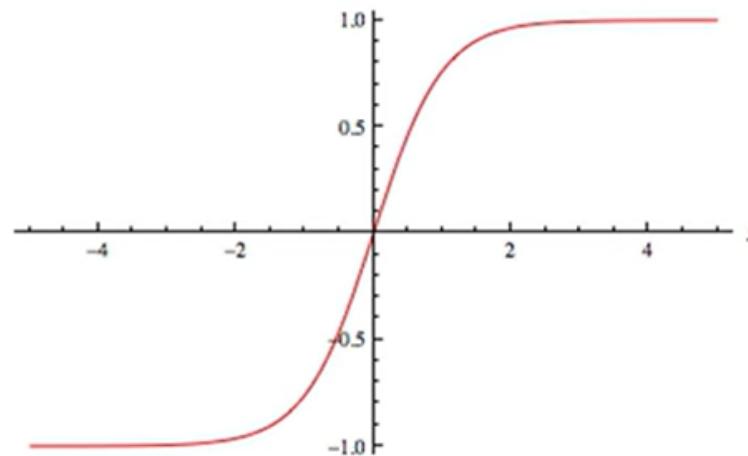


Sigmoid

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



Tanh Function

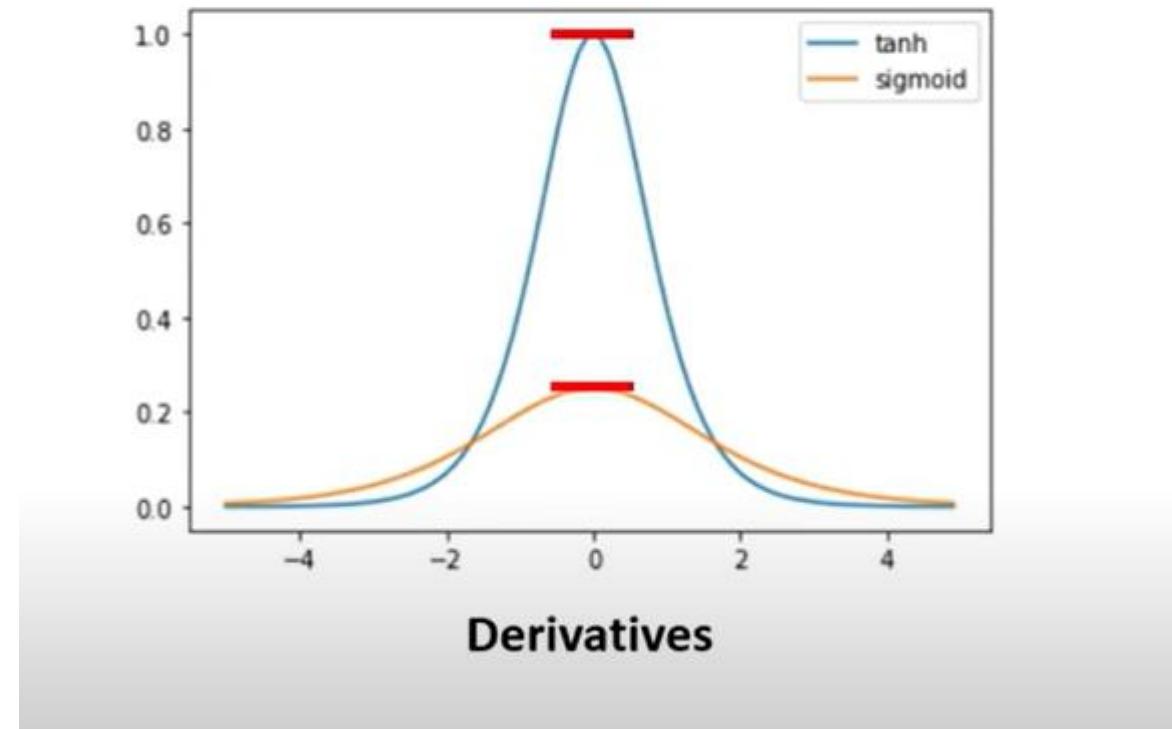


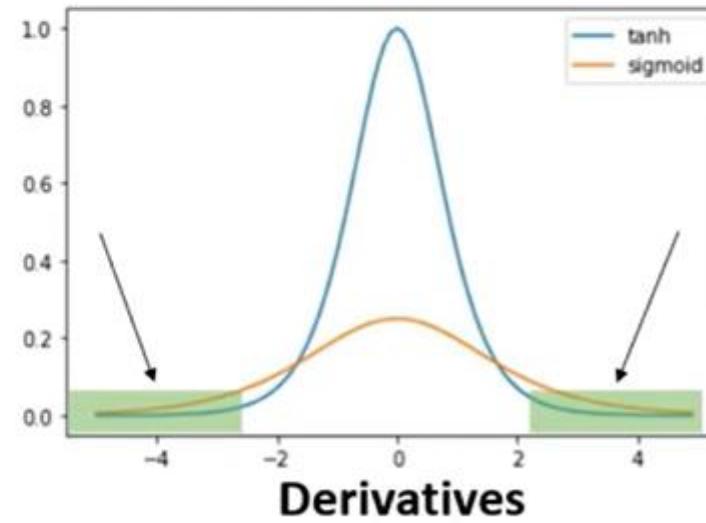
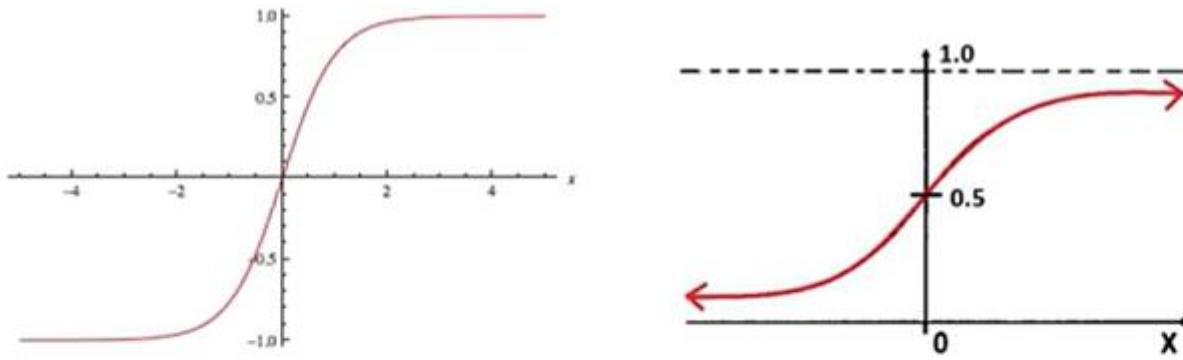
$$\tanh = \frac{e^x - e^{-x}}{e^x + e^{-x}}$$

$$W = W - \alpha * \frac{\partial \text{cost}}{\partial W}$$

We need to take derivative of activation functions

$$\frac{\partial \tanh}{\partial x}$$

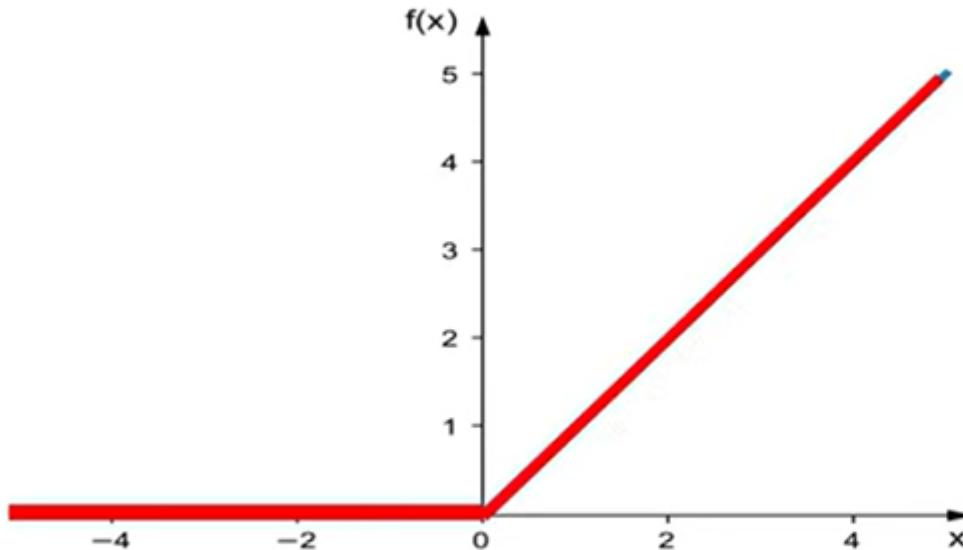




Smaller value of derivative leads to very slow learning

Vanishing Gradient Problem

ReLU (Rectified Linear Unit)



Piece-wise Linear

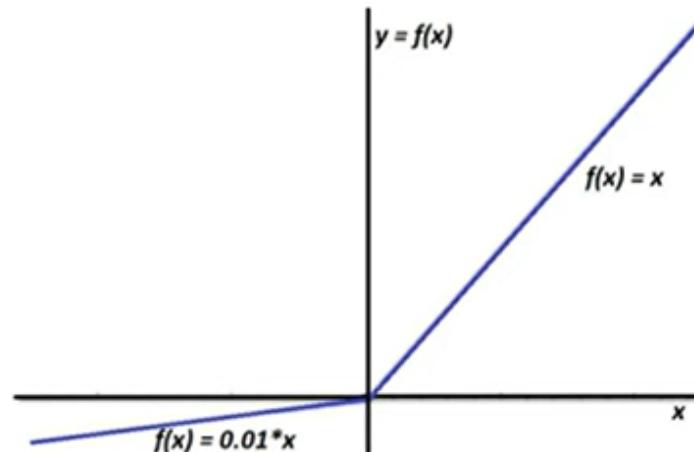
Advantages of both linear and non-linear property

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

Overcome the
Vanishing Gradient
Problem

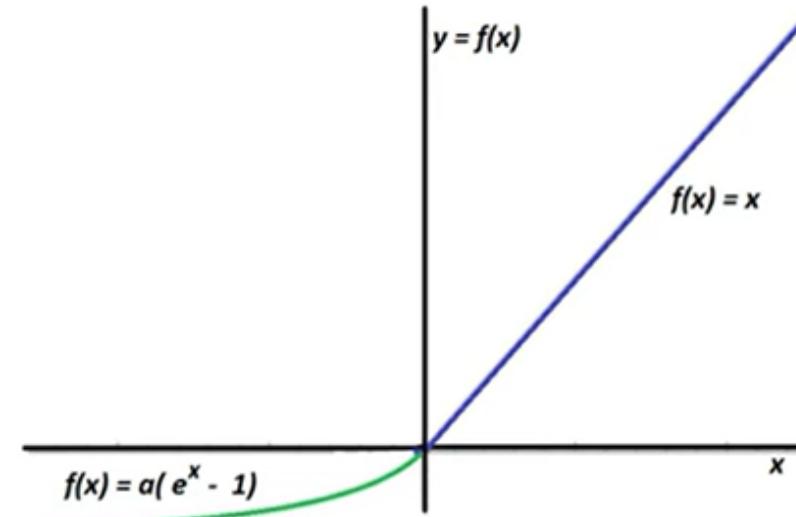
$$\frac{\partial f(x)}{\partial x} = \begin{cases} 1, & x > 0 \\ 0, & x < 0 \end{cases}$$

Variations of ReLU



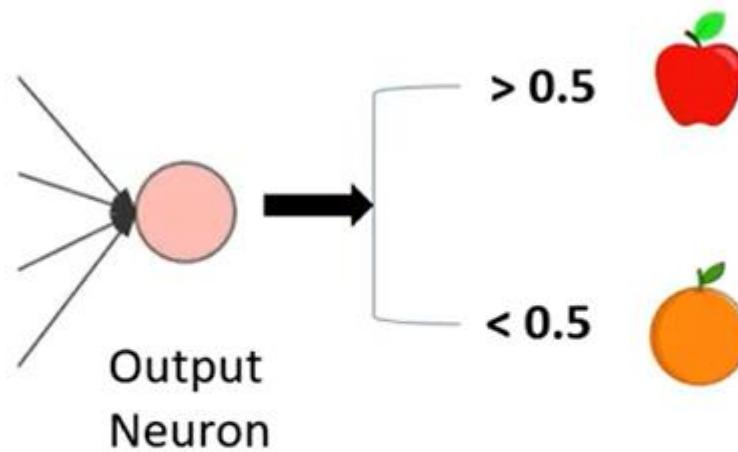
Leaky ReLU

$$f(x) = \max(0.01 * x, x)$$

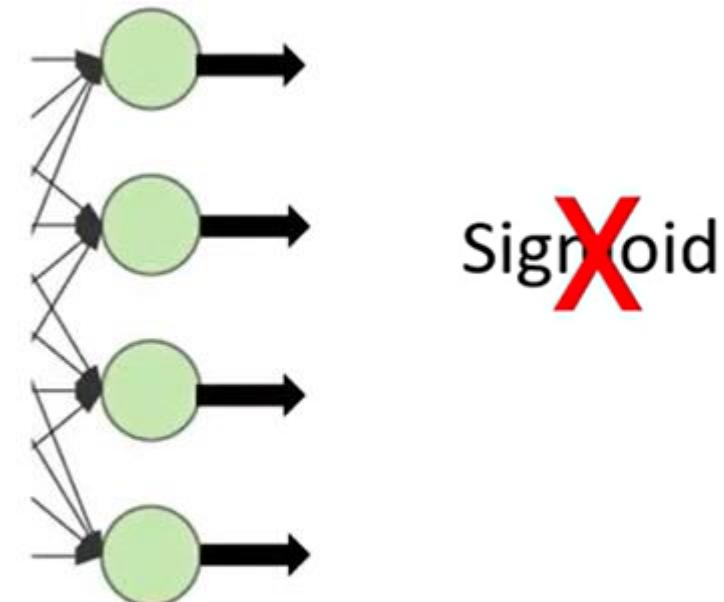


ELU (Exponential Linear Unit)

$$f(x) = \max(\alpha * (e^x - 1), x)$$

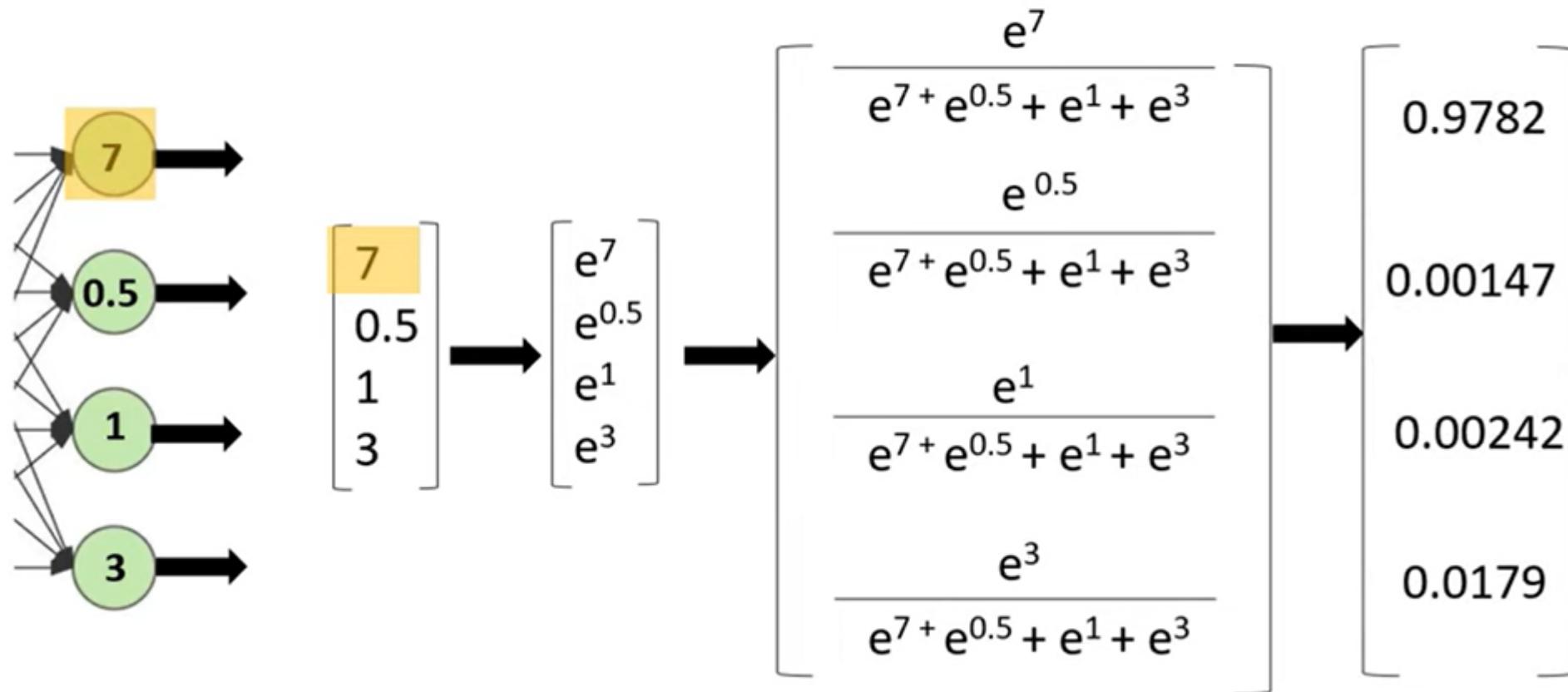


Binary Classification



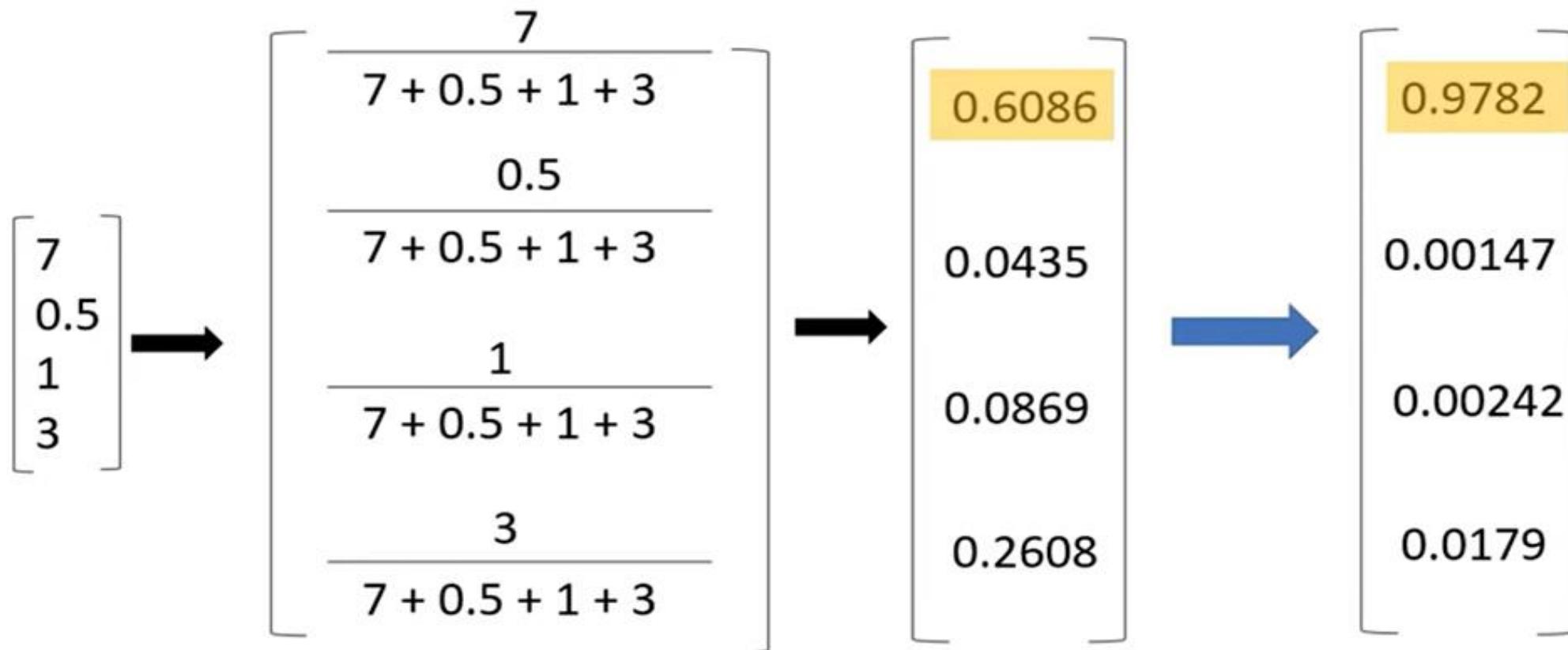
Multi-Class Classification
??

Softmax Function



- So why do we have to pass each value through an exponential before normalizing them? Why can't we just normalize the values themselves? This is because the goal of softmax is to make sure **one value is very high** (close to 1) and all **other values are very low** (close to 0). Softmax uses exponential to make sure this happens. And then we are normalizing because we need probabilities.

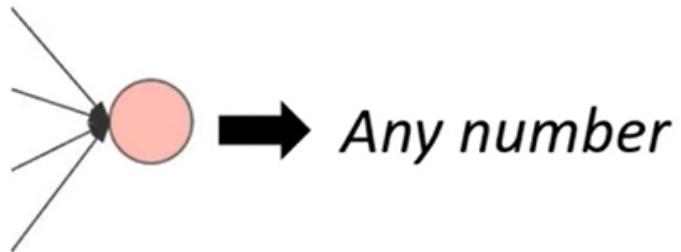
Softmax Function



Softmax Function

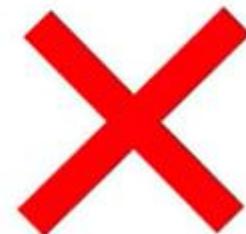
$$\text{Softmax } f_i(x) = \frac{e^{a_i}}{\sum_k e^{a_k}}$$

Linear Regression

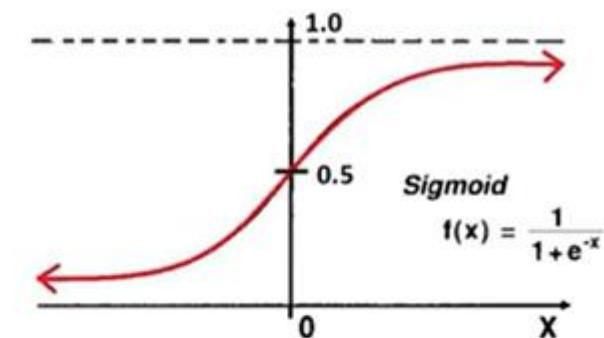
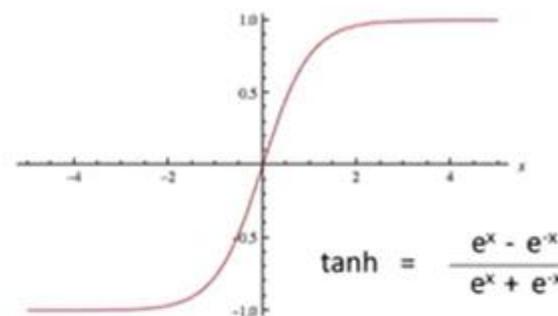
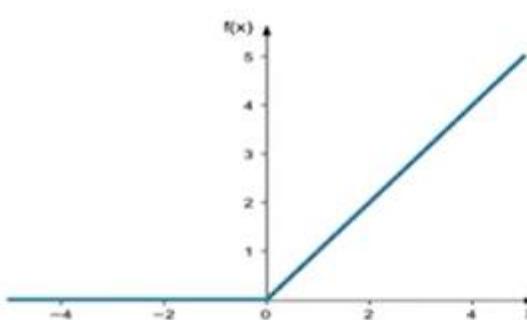


House Price Prediction

*Price of the house can be
anywhere from 0 to \$1 million*

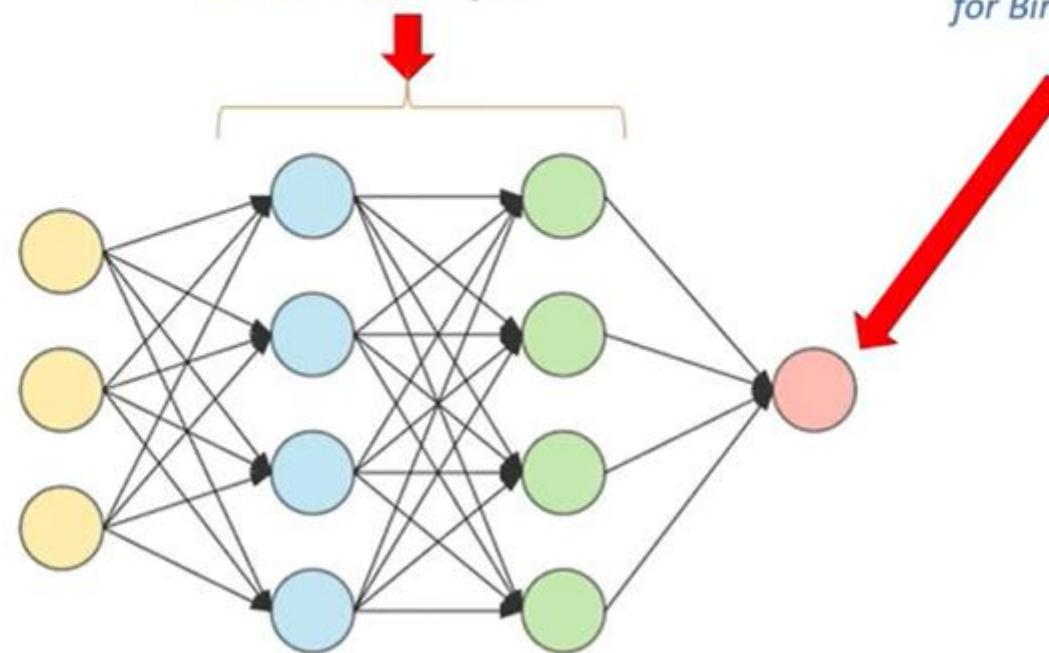


*No Activation
Function in
Output Neuron*

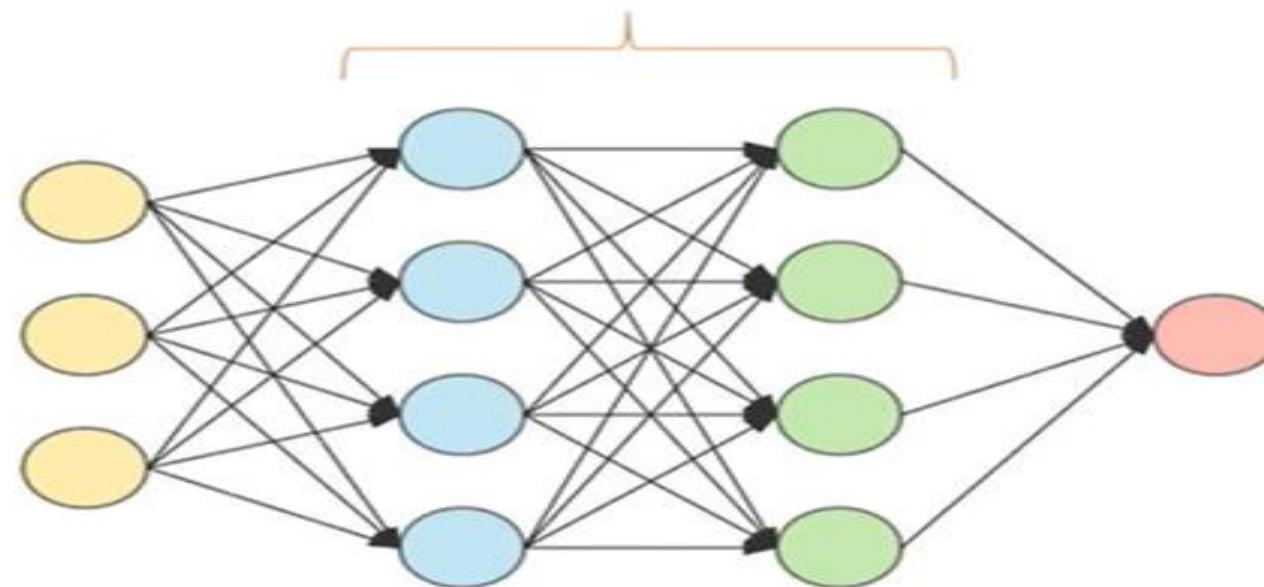
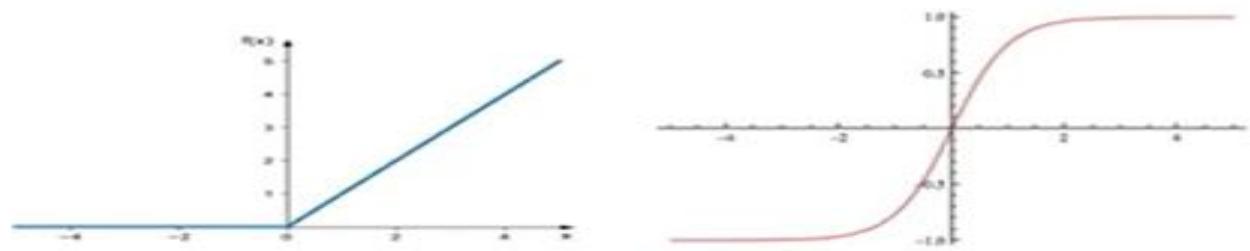


2) Either ReLU or tanh can be used in hidden layers

1) Sigmoid in output Neuron for Binary Classification

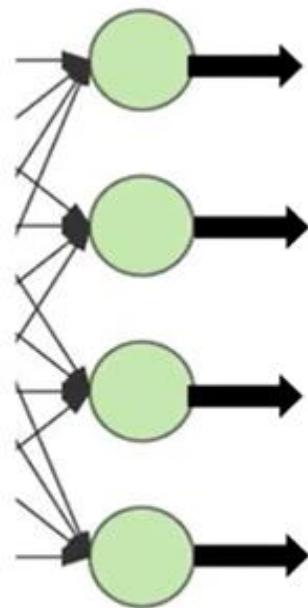


Try using both and see which gives better performance



Multi-Class Classification

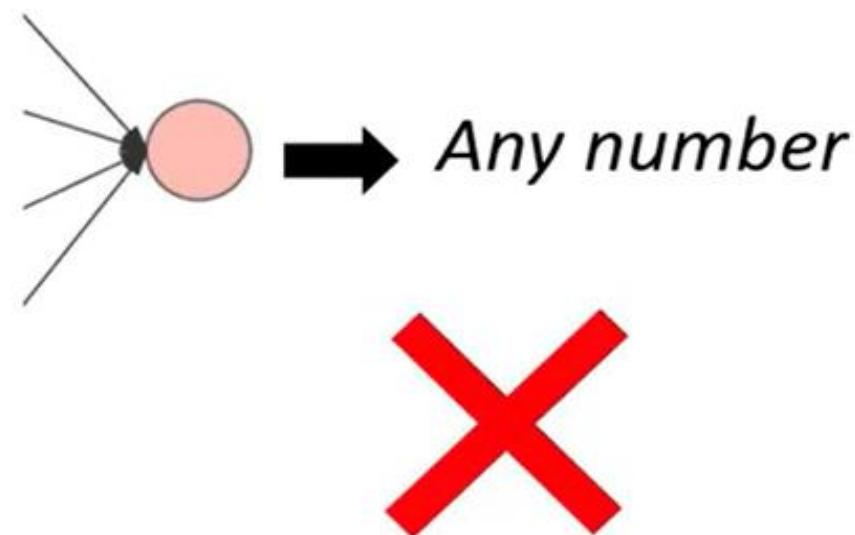
*tanh or ReLU in
hidden layers*



$$\text{Softmax } f_i(x) = \frac{e^{a_i}}{\sum_k e^{a_k}}$$

Softmax in Output layer

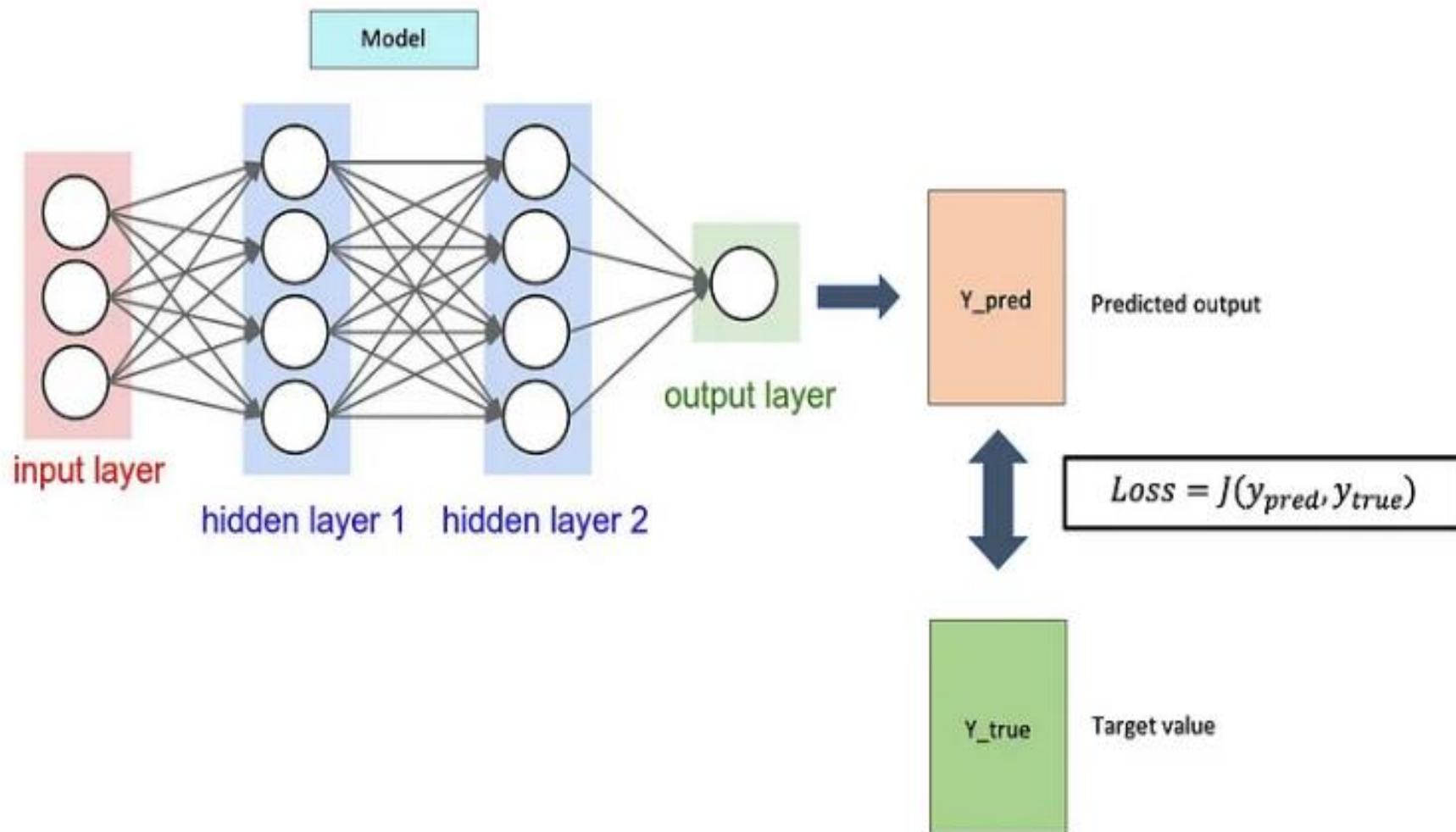
Linear Regression



*No Activation
Function in
Output Neuron*

Loss Function

- A measure of how well the neural network's predictions match the actual results. Common loss functions include:
- **Mean Squared Error (MSE):** For regression tasks.
- **Cross-Entropy Loss:** For classification tasks.



Cost Function

Cost Function & Loss Function - Actual - Predicted

Types of Cost Functions

Regression
Cost Function

Binary
Classification
Function

Multi-Class
Classification
Function

- **Loss function:** Used when we refer to the error for a single training example.

Cost function: Used to refer to an average of the loss functions over an entire training data.

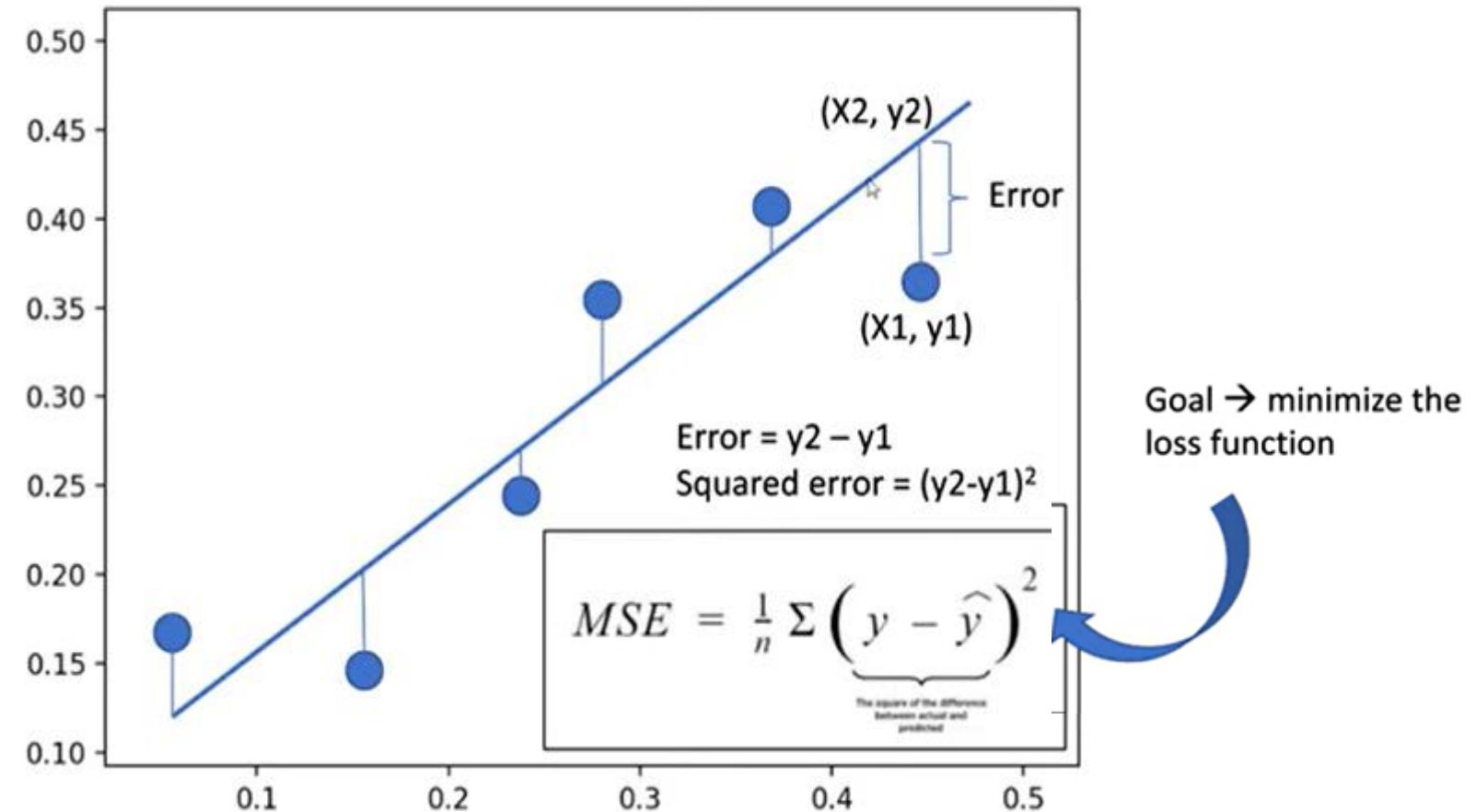
How to select a loss function for your task?

Based on the nature of your task, loss functions are classified as follows:

Regression Loss:

- Mean Square Error
- Mean Absolute Error

Linear Regression

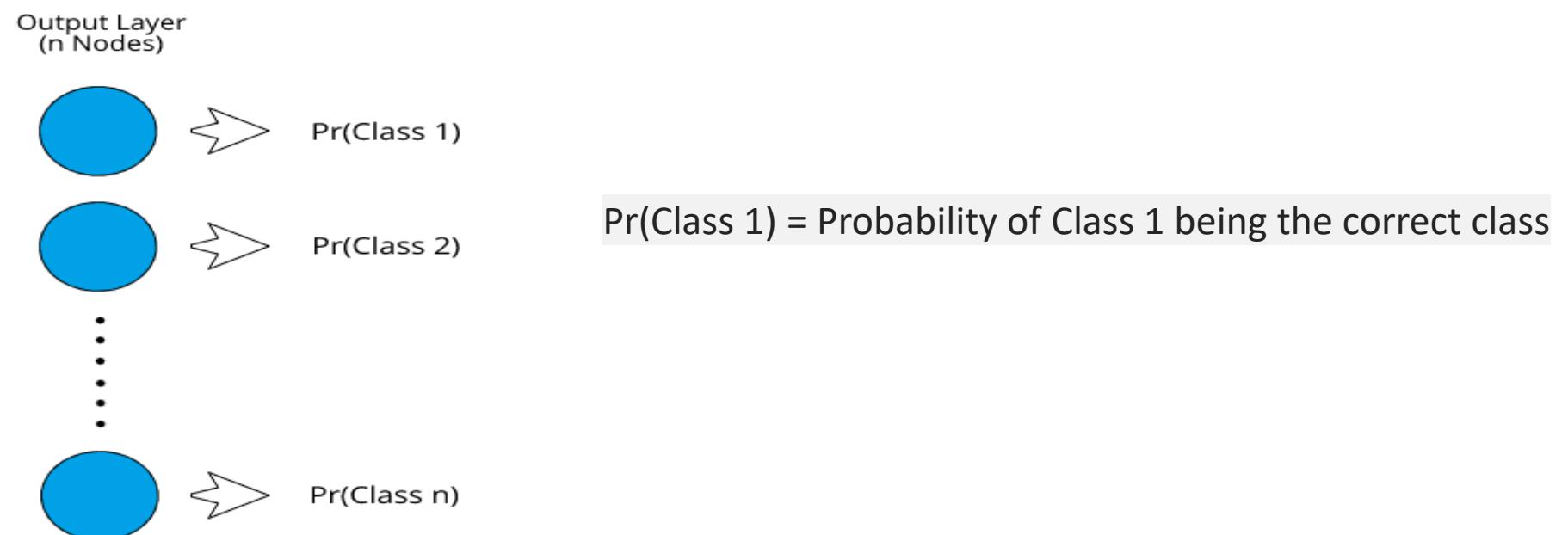


Classification Loss:

- ***Binary Classification:***
- Binary Cross Entropy Loss
- ***Multi-Class Classification:***
- Categorical Cross Entropy Loss

Classification Losses

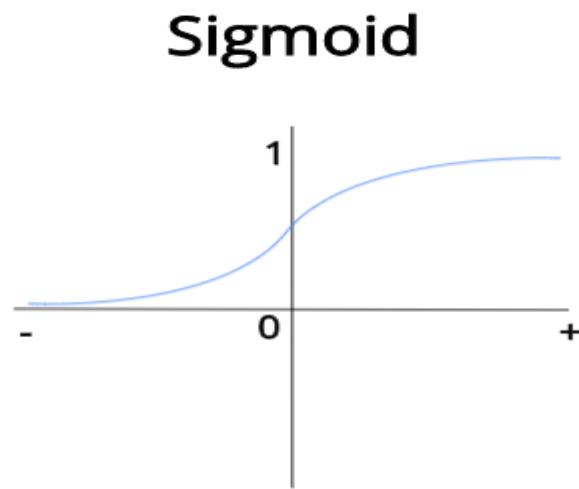
- When a neural network is trying to predict a **discrete** value, we can consider it to be a classification model. This could be a network trying to predict what kind of animal is present in an image, or whether an email is spam or not.



- The number of nodes of the output layer will depend on the number of classes present in the data.
- Each node will represent a single class. The value of each output node essentially represents the **probability** of that class being the correct class.
- Once we get the probabilities of all the different classes, we will consider the class having the highest probability to be the predicted class for that instance

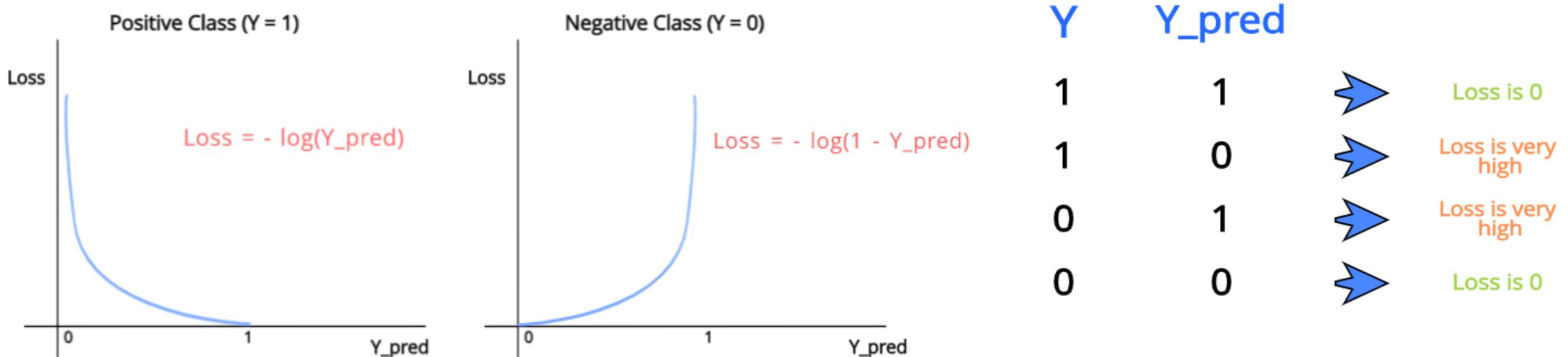
Binary Classification

- In binary classification, there will be only one node in the output layer even though we will be predicting between two classes. In order to get the output in a probability format, we need to apply an activation function. Since probability requires a value in between 0 and 1 we will use the **sigmoid function** which can *squish* any real value to a value between 0 and 1.



If the output is above 0.5 (50% Probability), we will consider it to be falling under the **positive class** and if it is below 0.5 we will consider it to be falling under the **negative class**.

The loss function we use for binary classification is called **binary cross entropy (BCE)**.



As you can see, there are two separate functions, one for each value of Y. When we need to predict the **positive class** ($Y = 1$), we will use $\text{Loss} = -\log(Y_{\text{pred}})$. And when we need to predict the **negative class** ($Y = 0$), we will use $\text{Loss} = -\log(1-Y_{\text{pred}})$.

We can mathematically represent the entire loss function into one equation as follows:

$$\text{Loss} = (Y)(-\log(Y_{pred})) + (1 - Y)(-\log(1 - Y_{pred}))$$

Remains when Y = 1

Removed when Y = 0

Remains when Y = 0

Removed when Y = 1

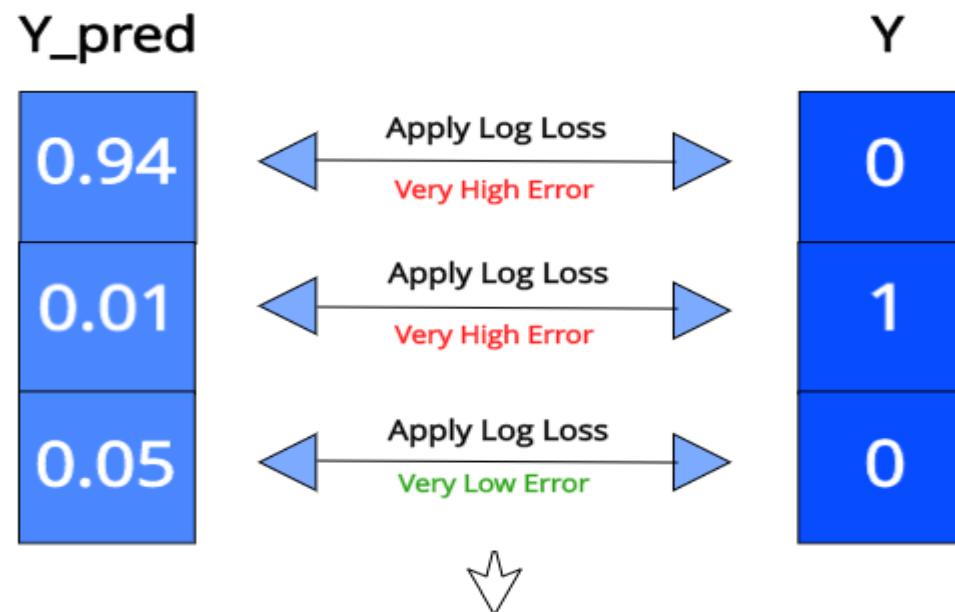
Multiclass Classification

- Multiclass classification is appropriate when we need our model to predict **one** possible class output every time.
- The activation function we use in this case is **softmax**. This function ensures that all the output nodes have values between 0–1 and the **sum of all** output node values **equals to 1 always**. The formula for softmax is as follows

$$\text{Softmax}(y_i) = \frac{e^{y_i}}{\sum_{i=0}^n e^{y_i}}$$



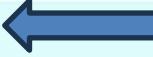
The loss function is essentially the same as that of binary classification. We will just apply **log loss** on each output node with respect to its respective target value and then we will find the sum of this across all output nodes.



This loss is called
as **Categorical Cross Entropy**

Total Loss = Sum of all output nodes Log Loss

Summary

Problem type	Last layer output nodes	Hidden layer activation	Last layer activation	Loss function
Binary classification	1	ReLU (first choice)	Sigmoid	Binary Cross Entropy 
				Weighted Cross Entropy
			Tanh	Hinge Loss
Multi-class, single label classification	Number of classes		SoftMax	Categorical Cross Entropy 
				Sparse Categorical Cross Entropy
				KullBack Leiber Divergence Loss

Task	Error type	Loss function	Note
Regression	Mean-squared error	$\frac{1}{n} \sum_{i=1}^n (y_i - \hat{y}_i)^2$	Easy to learn but sensitive to outliers (MSE, L2 loss)
	Mean absolute error	$\frac{1}{n} \sum_{i=1}^n y_i - \hat{y}_i $	Robust to outliers but not differentiable (MAE, L1 loss)
Classification	Cross entropy = Log loss	$-\frac{1}{n} \sum_{i=1}^n [y_i \log(\hat{y}_i) + (1 - y_i) \log(1 - \hat{y}_i)]$	Quantify the difference between two probability



THANK YOU



KESHAV MEMORIAL INSTITUTE OF TECHNOLOGY
AN AUTONOMOUS INSTITUTION - ACCREDITED BY NAAC WITH 'A' GRADE
Narayananaguda, Hyderabad.

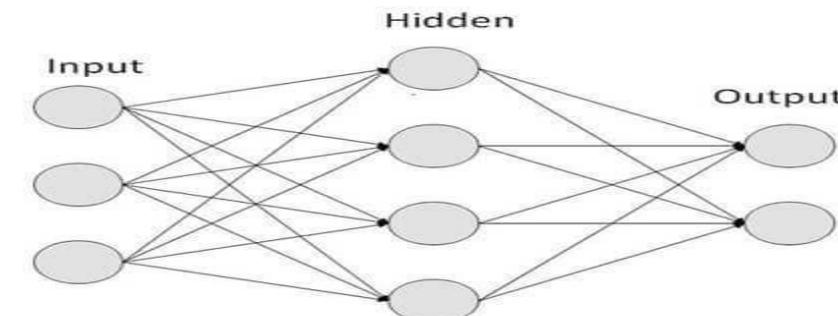
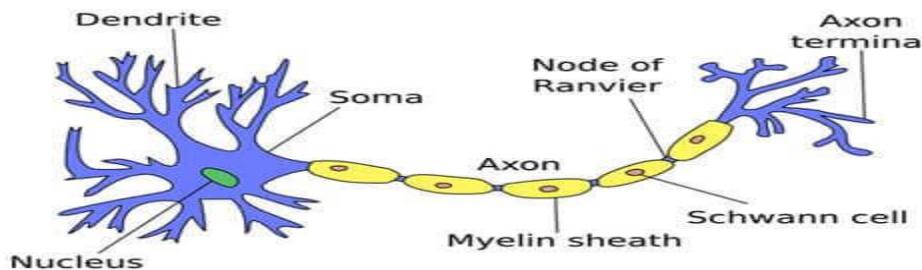
Deep Learning

SESSION-3

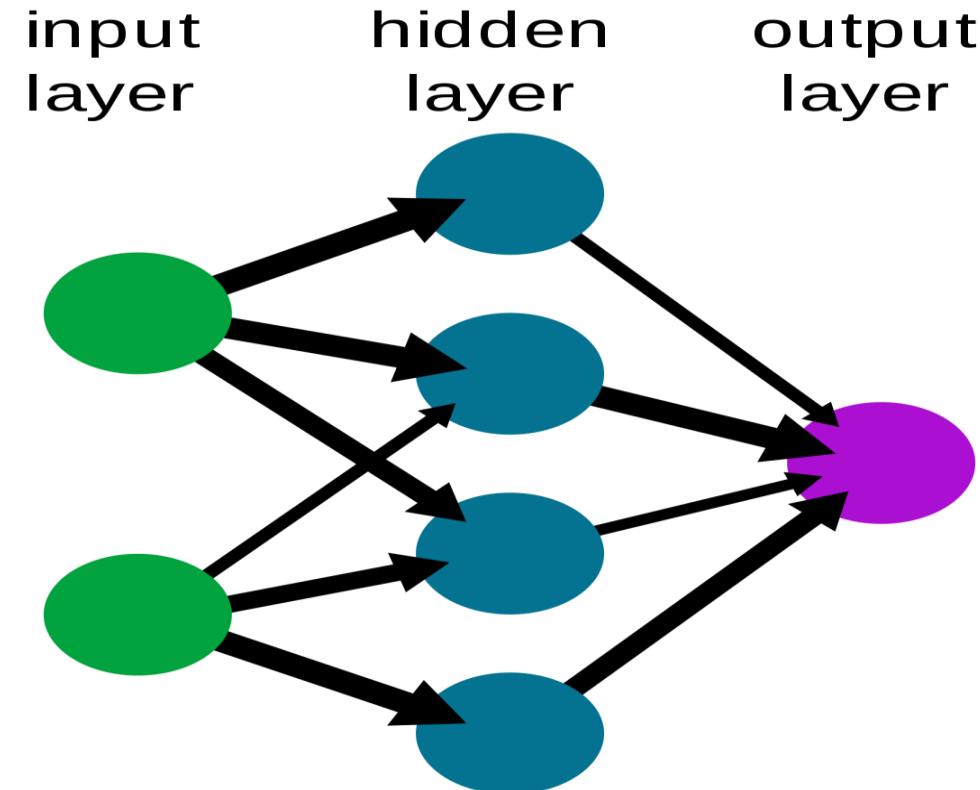
BY
ASHA

Introduction to Neural Networks

- Neural networks are computational models inspired by the human brain. They consist of interconnected units or nodes called neurons, organized into layers.
- Neural networks are capable of learning from data and making predictions or decisions without being explicitly programmed.

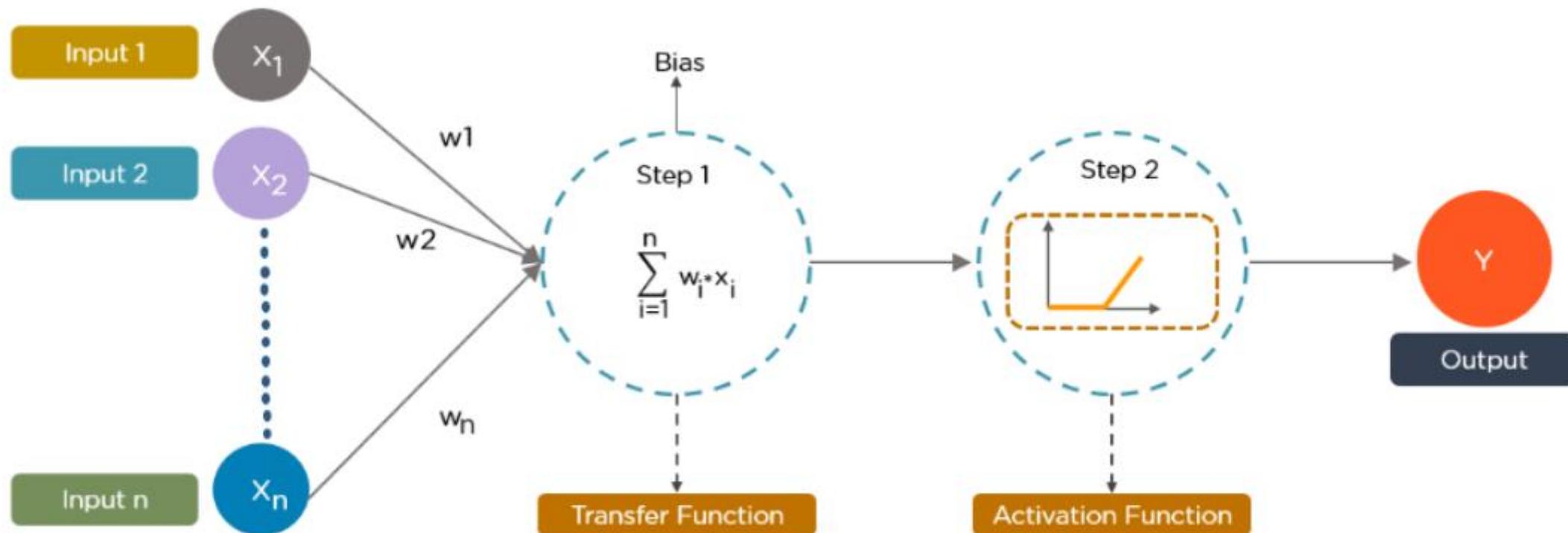


A simple neural network



A simple neural network consists of three components :

- Input layer
- Hidden layer
- Output layer



- 1. In the first step, Input units are passed i.e data is passed with some weights attached to it to the hidden layer.** We can have any number of hidden layers. In the above image inputs $x_1, x_2, x_3, \dots, x_n$ is passed.
- 2. Each hidden layer consists of neurons.** All the inputs are connected to each neuron.
- 3. After passing on the inputs, all the computation is performed in the hidden layer.**

- Computation performed in hidden layers are done in two steps which are as follows :
- First of all, **all the inputs are multiplied by their weights.** Weight is the gradient or coefficient of each variable. It shows the strength of the particular input. After assigning the weights, a bias variable is added. **Bias** is a constant that helps the model to fit in the best way possible.

$$Z_1 = W_1 * In_1 + W_2 * In_2 + W_3 * In_3 + W_4 * In_4 + W_5 * In_5 + b$$

- W_1, W_2, W_3, W_4, W_5 are the weights assigned to the inputs $In_1, In_2, In_3, In_4, In_5$, and b is the bias.
- Then in the second step, the **activation function is applied to the linear equation Z1.** The activation function is a nonlinear transformation that is applied to the input before sending it to the next layer of neurons. The importance of the activation function is to inculcate nonlinearity in the model.

4. The whole process described in point 3 is performed in each hidden layer. After passing through every hidden layer, **we move to the last layer i.e our output layer which gives us the final output.**

The process explained above is known as Forwarding Propagation.

5. After getting the predictions from the output layer, the **error is calculated i.e the difference between the actual and the predicted output.**

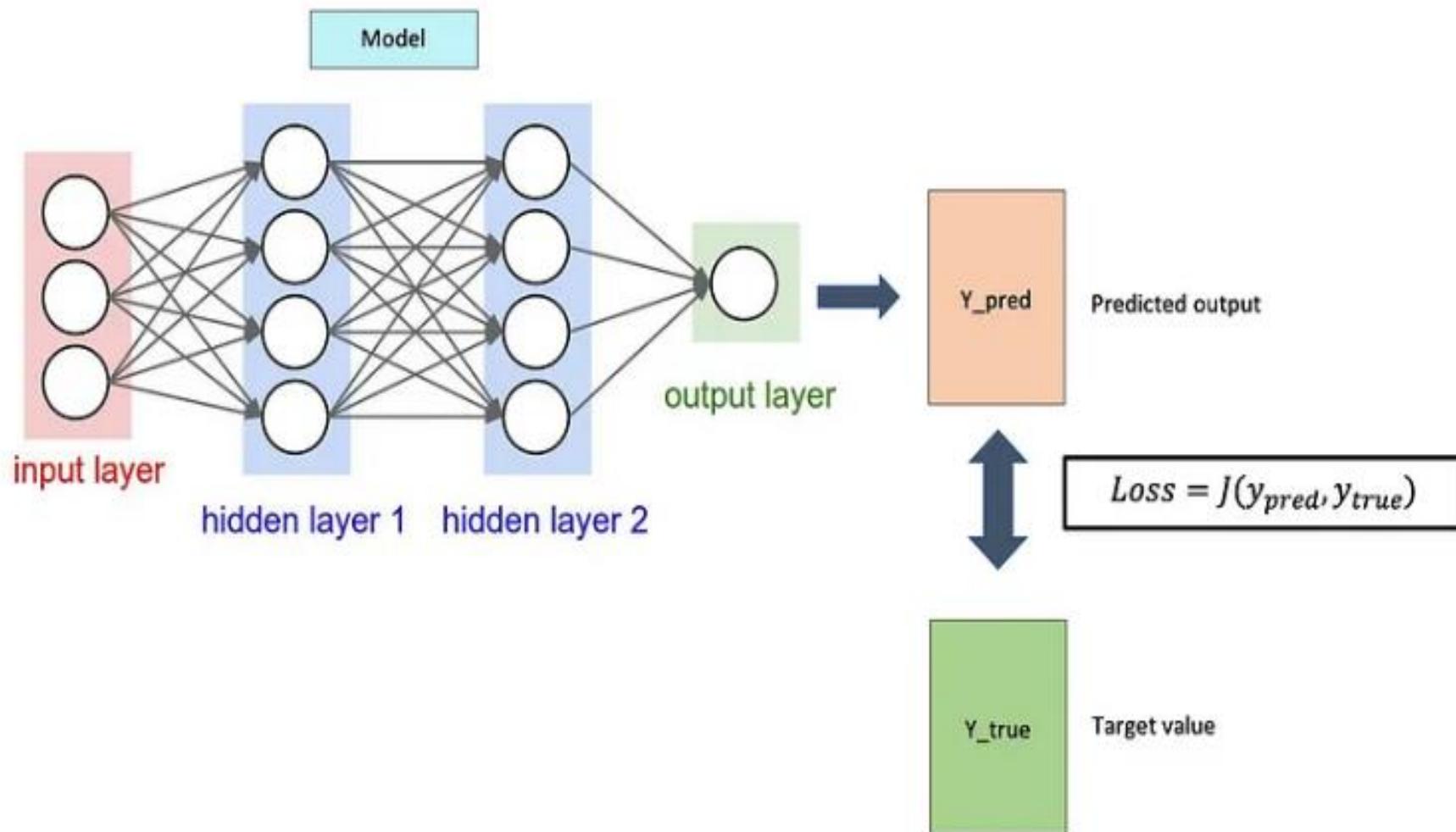
If the error is large, then the steps are taken to minimize the error and for the same purpose, **Back Propagation is performed.**

Forward Propagation

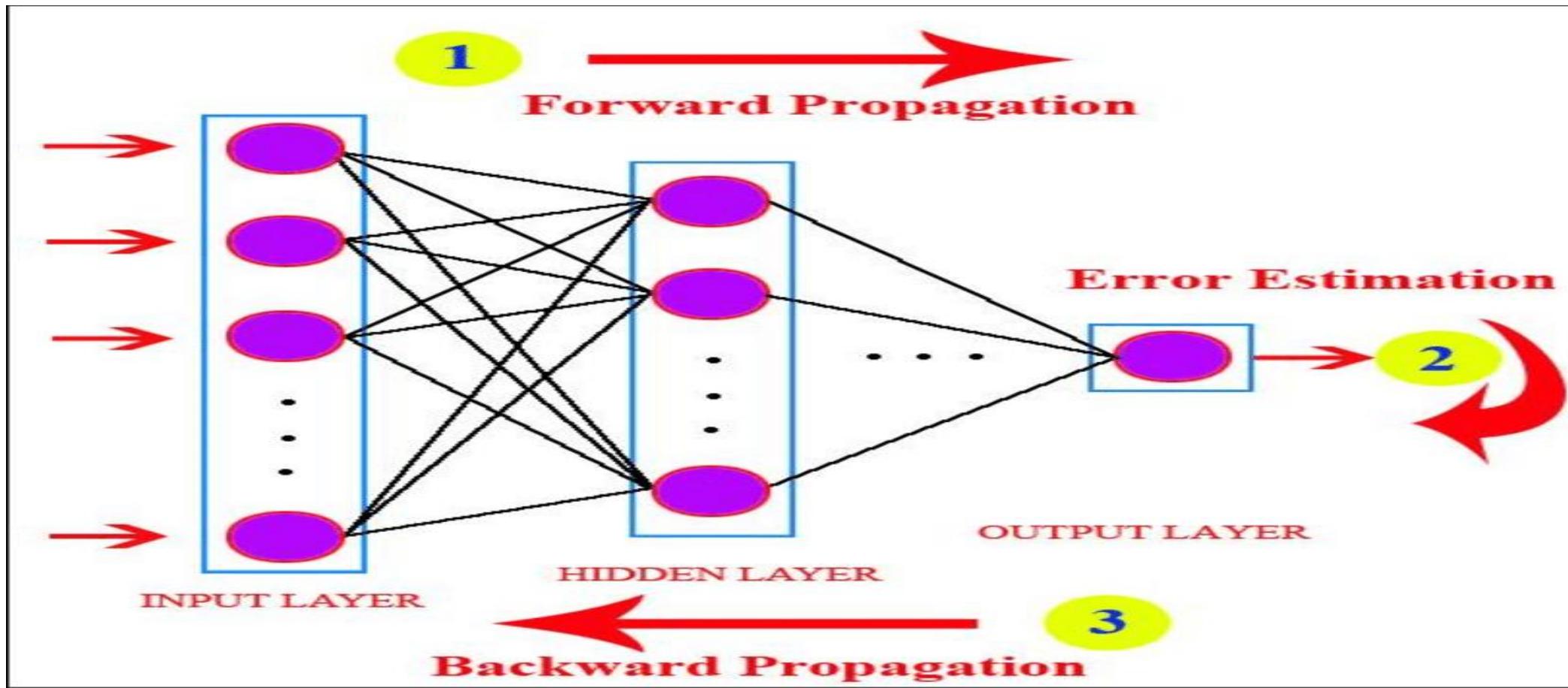
- The process of computing the output of a neural network. It involves:
 1. Multiplying inputs by weights.
 2. Adding biases.
 3. Applying the activation function to produce the output.

Loss Function

- A measure of how well the neural network's predictions match the actual results. Common loss functions include:
- **Mean Squared Error (MSE):** For regression tasks.
- **Cross-Entropy Loss:** For classification tasks.



Task	Error type	Loss function	Note
Regression	Mean-squared error	$\frac{1}{n} \sum_{i=1}^n (y_i - \hat{y}_i)^2$	Easy to learn but sensitive to outliers (MSE, L2 loss)
	Mean absolute error	$\frac{1}{n} \sum_{i=1}^n y_i - \hat{y}_i $	Robust to outliers but not differentiable (MAE, L1 loss)
Classification	Cross entropy = Log loss	$-\frac{1}{n} \sum_{i=1}^n [y_i \log(\hat{y}_i) + (1 - y_i) \log(1 - \hat{y}_i)]$	Quantify the difference between two probability



Backpropagation

- The process of updating weights and biases based on the error of the network's predictions. It involves:
 1. Computing the gradient of the loss function with respect to each weight using the chain rule.
 2. Adjusting weights and biases to minimize the loss function.

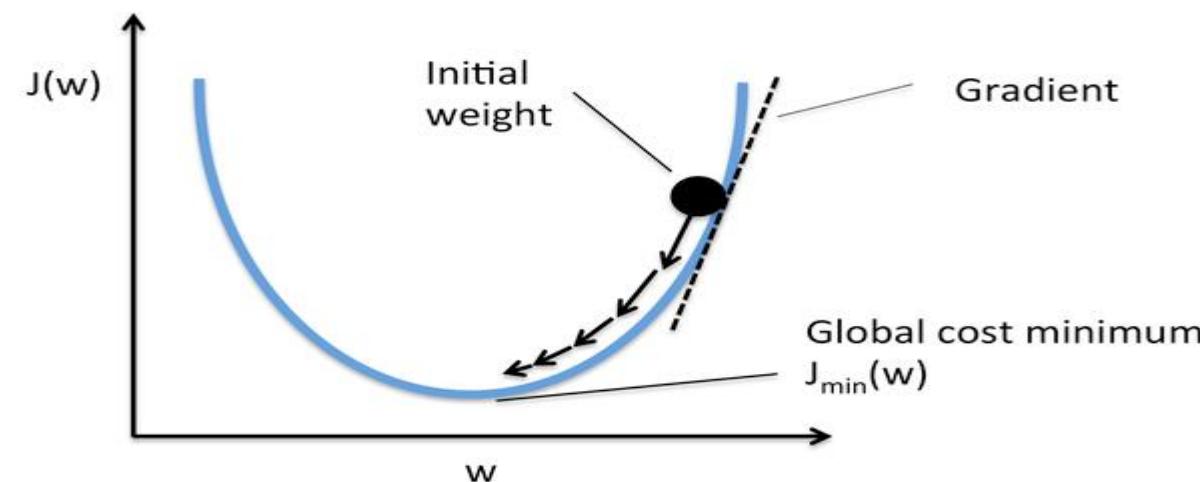
What is Back Propagation and How it works?

- Back Propagation is the process of updating and finding the optimal values of weights or coefficients which helps the model to minimize the error i.e difference between the actual and predicted values.

- How the **weights** are updated and new **weights** are calculated?
- The **weights** are updated with the help of **optimizers**.
Optimizers are the methods/ mathematical formulations to change the attributes of neural networks i.e weights to minimize the error

Back Propagation with Gradient Descent

- Gradient Descent is one of the optimizers which helps in calculating the new weights. Let's understand step by step how Gradient Descent optimizes the cost function.
- In the image below, the curve is our cost function curve and our aim is the minimize the error such that J_{\min} i.e global minima is achieved.



Steps to achieve the global minima:

1. First, **the weights are initialized randomly** i.e random value of the weight, and intercepts are assigned to the model while forward propagation and **the errors are calculated** after all the computation. (As discussed above)
2. Then **the gradient is calculated** i.e derivative of error w.r.t **current weights**

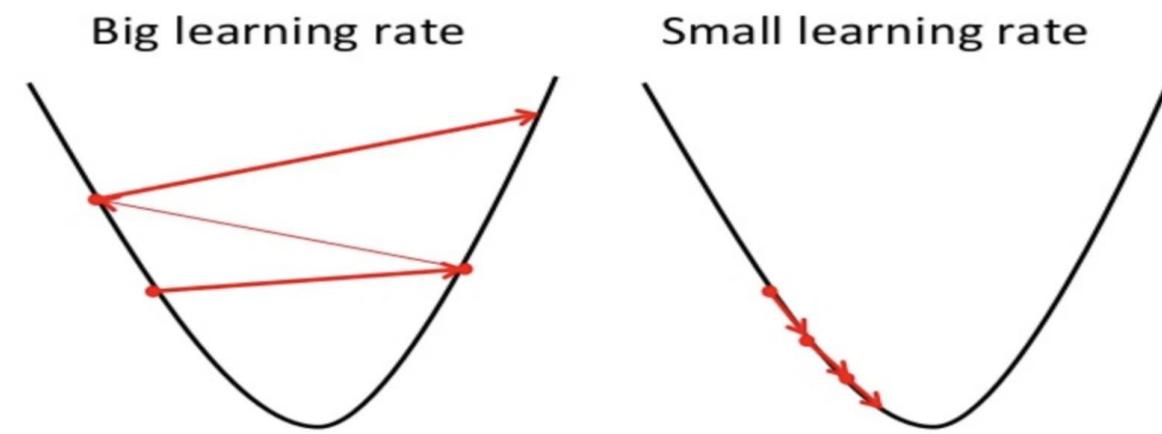
- Then new weights are calculated using the below formula, where **a** is the learning rate which is the parameter also known as step size to control the speed or steps of the backpropagation. It gives additional control on how fast we want to move on the curve to reach global minima.

$*W_x = W_x - a \left(\frac{\partial \text{Error}}{\partial W_x} \right)$

4. This process of calculating the new weights, then errors from the new weights, and then updation of weights **continues till we reach global minima and loss is minimized.**

A point to note here is that the learning rate i.e α in our weight updation equation should be chosen wisely.

Learning rate is the amount of change or step size taken towards reaching global minima. **It should not be very small** as it will take time to converge as well as **it should not be very large** that it doesn't reach global minima at all.



BUILDING ANN FROM SCRATCH

Step 1: Load the Dataset

Load the breast cancer dataset from scikit-learn. The Churn Modeling dataset is commonly used in machine learning to predict customer churn in the banking sector. Churn refers to the loss of customers, which can significantly impact a company's profitability. The dataset typically contains various features about customers that can help predict whether they will leave the bank.

Step 2: Split the Dataset

Split the data into training and testing sets. This helps in evaluating the performance of the ANN on unseen data.

Step 3: Normalize the Features

Normalize the input features to ensure all features contribute equally to the training process. This can be done using methods like standard scaling.

Step 4: Initialize the ANN Parameters

Decide on the architecture of the ANN, including the number of layers and the number of neurons in each layer. Initialize the weights and biases for each layer randomly.

Step 5: Define the Activation Functions

Choose activation functions for the neurons in each layer. Common choices are the sigmoid function for the output layer (binary classification).

Step 6: Forward Propagation

Implement the forward propagation process, where the input data passes through the network layer by layer, applying the weights, biases, and activation functions to produce the final output.

Step 7: Compute the Loss

Calculate the loss using an appropriate loss function. For binary classification, the binary cross-entropy loss is typically used.

Step 8: Backward Propagation

Implement backward propagation to compute the gradients of the loss function with respect to the weights and biases. This involves calculating the gradient of the loss function from the output layer back to the input layer.

Step 9: Update the Weights and Biases

Use an optimization algorithm, such as gradient descent, to update the weights and biases using the gradients computed during backward propagation. This step aims to minimize the loss function.

Step 10: Train the ANN

Iterate through the training dataset for a specified number of epochs. In each epoch, perform forward propagation, compute the loss, perform backward propagation, and update the weights and biases.

Step 11: Evaluate the ANN

After training, evaluate the performance of the ANN on the testing set. This involves using the trained model to make predictions on the test data and comparing these predictions to the true labels to calculate metrics like accuracy.

Step 12: Make Predictions

Use the trained ANN to make predictions on new or unseen data by performing forward propagation with the learned weights and biases.



THANK YOU



KESHAV MEMORIAL INSTITUTE OF TECHNOLOGY
AN AUTONOMOUS INSTITUTION - ACCREDITED BY NAAC WITH 'A' GRADE
Narayanaguda, Hyderabad.

Deep Learning

28-10-2024

BY
ASHA

Building ANN from Scratch

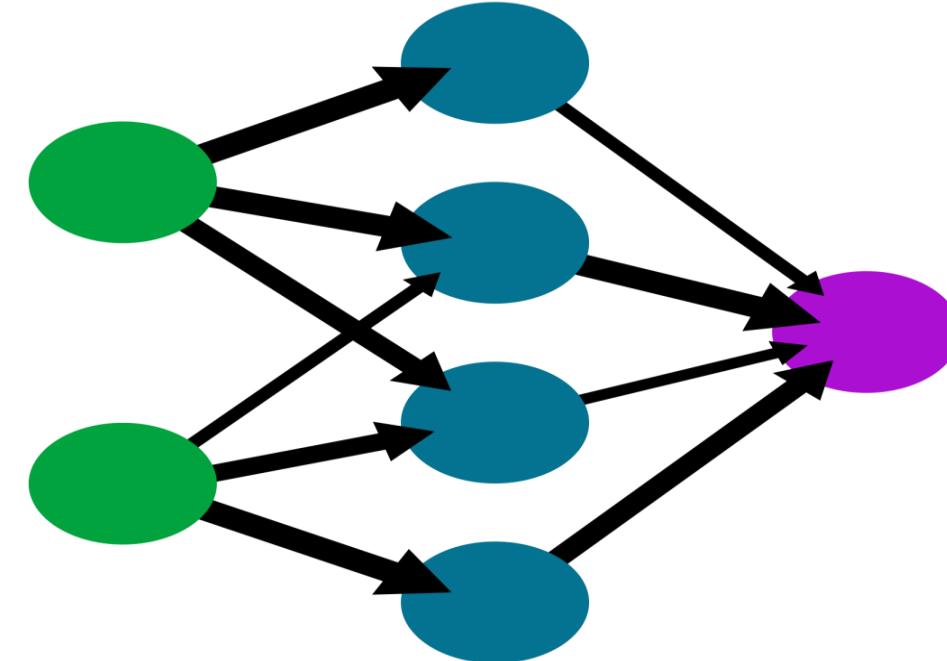
Neural Networks, Activation Functions, Loss Functions, and Optimizers

What are Neural Networks?

- At the heart of AI systems, **neural networks** are computational models inspired by the structure and function of the human brain. They consist of interconnected nodes, called **neurons**, that are organized in layers. These networks are particularly powerful for tasks that require pattern recognition, such as image classification, language processing, and time series prediction.

A simple neural network

input layer hidden layer output layer



A simple neural network consists of three components :

- Input layer
- Hidden layer
- Output layer

- **Input layer:** The layer where raw data (features) is fed into the network.
- **Hidden layers:** Intermediate layers where the network processes and transforms the input data through weights and biases.
- **Output layer:** The final layer, producing the model's prediction.

Why Neural Networks?

Traditional machine learning algorithms may struggle with complex, non-linear problems. Neural networks, however, excel at:

- Learning from data and improving their performance.
- Handling a variety of tasks, from regression to classification, across multiple domains (vision, speech, text, etc.).
- Detecting intricate patterns and relationships that traditional models can miss, particularly in unstructured data like images or audio.

Why Do We Use Neural Networks?

We use neural networks because of their ability to:

- **Learn complex, non-linear relationships:** Unlike traditional models like linear regression, neural networks can model highly complex relationships between input and output data.
- **Generalize well to unseen data:** With proper training, neural networks can learn to generalize from the training set to new, unseen data, improving predictive performance.
- **Versatility across domains:** Neural networks can be applied to a wide range of tasks including classification, regression, and generation of new content in domains like computer vision, natural language processing, and more.
- **Automatic feature extraction:** Deep neural networks, particularly convolutional neural networks (CNNs), can automatically extract important features from raw data without the need for manual feature engineering.

How Do We Build an Artificial Neural Network (ANN)?

Step 1: Initialize the Network Architecture

A basic ANN consists of three types of layers:

Input Layer: Receives the input features (e.g., pixel values of an image, words in a text, etc.).

Hidden Layers: Consist of neurons that apply a transformation to the input data using weights and biases.

Output Layer: Produces the final prediction, typically a classification or regression output.

Each connection between neurons has a **weight**, and each neuron has a **bias**.

Step 2: Apply an Activation Function

Each neuron takes a weighted sum of its inputs, applies a bias, and passes the result through an **activation function**. The role of activation functions is to introduce **non-linearity**, allowing the network to model complex data patterns.

Step 4: Define a Loss Function

The **loss function** quantifies how far off the network's predictions are from the actual values. Common loss functions include:

Mean Squared Error (MSE) for regression tasks.

Cross-Entropy Loss for classification tasks. The goal is to minimize this loss by adjusting the network's weights and biases.

Step 3: Forward Propagation – Generating Predictions

Once the architecture is set, the data is fed forward through the network:

Each neuron in the hidden layers calculates the weighted sum of its inputs, adds the bias, and applies the activation function.

The final layer outputs a prediction, often using activation functions like **Softmax** for multi-class classification or **Sigmoid** for binary classification.

Step 5: Backpropagation – Learning from Errors

After generating predictions, the network needs to learn from its mistakes.

Backpropagation is the process of calculating the **gradient** of the loss function with respect to each weight in the network, using the chain rule.

This gradient indicates how each weight should be adjusted to reduce the overall loss.

Step 6: Use an Optimizer to Update Weights

The **optimizer** is responsible for adjusting the weights based on the gradients from backpropagation. Popular optimizers include:

Stochastic Gradient Descent (SGD): Updates weights using small, random batches of data.

Adam: Combines the advantages of SGD and other adaptive algorithms, allowing for faster and more efficient convergence.

Optimizers are key to ensuring that the model converges to an optimal solution efficiently.

Step 7: Train the Network

The training process consists of feeding the model with batches of data (called **epochs**), performing forward and backward passes, and updating weights to reduce the loss function. After several iterations, the model learns to make accurate predictions.



KESHAV MEMORIAL INSTITUTE OF TECHNOLOGY
AN AUTONOMOUS INSTITUTION - ACCREDITED BY NAAC WITH 'A' GRADE
Narayanauguda, Hyderabad.

BUILD ANN FROM SCRATCH

Exercise-2

By
Asha

1. Data Loading and Preprocessing:

We load the Boston Housing dataset, scale the features, and split the data into training and testing sets.

2. Forward Pass:

Compute the linear combination of inputs and weights for the hidden layer (Z_1).

Apply the sigmoid activation function to get the activations of the hidden layer (A_1).

Compute the linear combination for the output layer (Z_2).

3. Loss Calculation:

Compute the Mean Squared Error (MSE) loss between the predicted and actual target values.

4. Backpropagation:

Compute the gradients of the loss with respect to the weights and biases of the output layer.

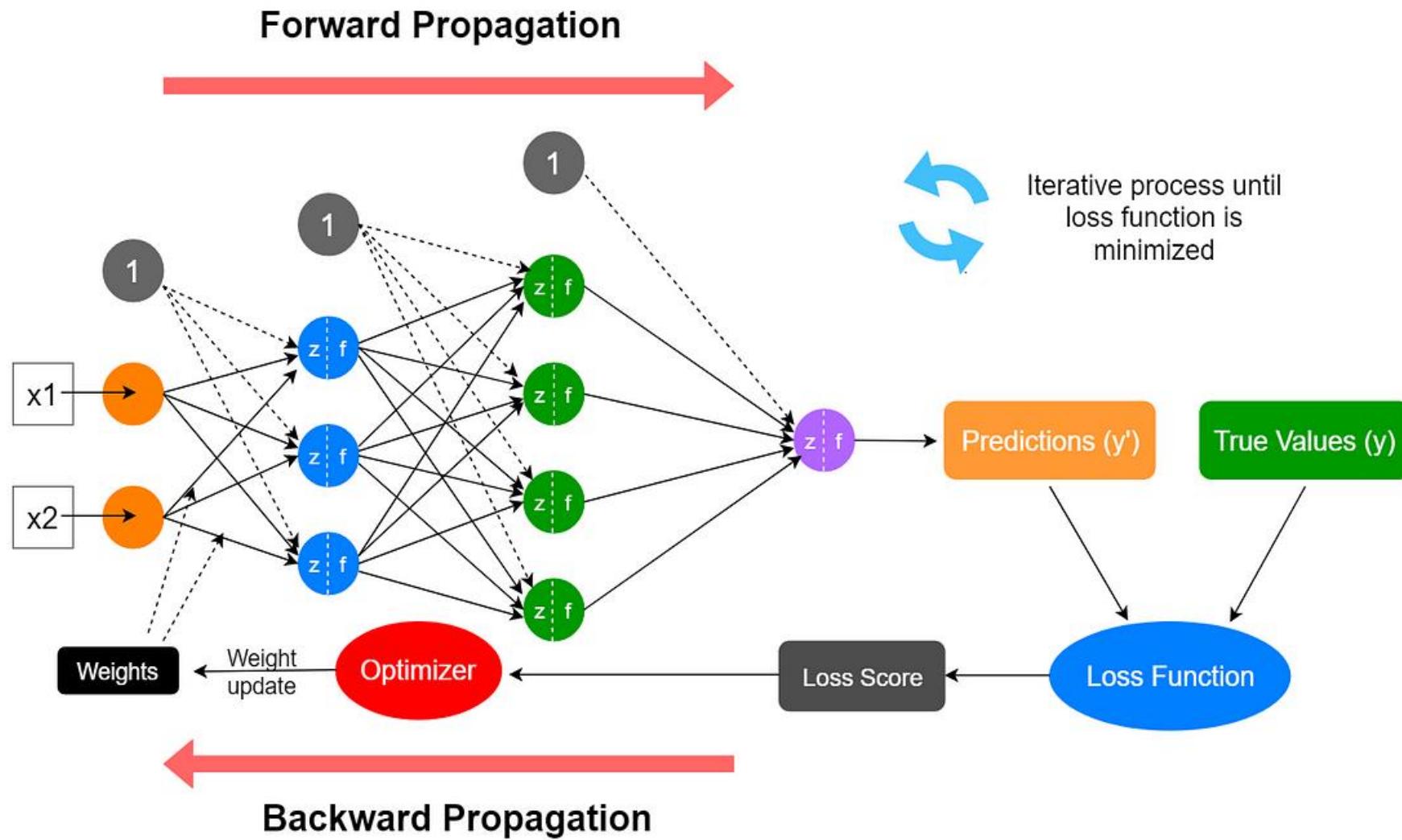
Compute the gradients for the hidden layer weights and biases using the chain rule and sigmoid derivative.

Update the weights and biases using gradient descent.

5. Training and Testing:

Train the ANN on the training data and print the training loss every 100 epochs.

Evaluate the model on the test data and print the test loss.



You are tasked with creating a simple artificial neural network (ANN) from scratch to find price of the house using boston data using the Scikit-learn boston dataset. Write a Python script that includes the following steps:

A. Data Loading and preprocessing

1. Import Necessary libraries
2. # Load the Boston Housing dataset and Split the dependent and independent variables(X,y)
3. # Standardize the features
4. # Split the dataset into training and testing sets

B. Neural Network Implementation

1. # Define the architecture
2. # Initialize weights and biases
3. # Define Activation function (ReLU)
4. # Define Derivative of ReLU
5. # Mean squared error loss
6. # Forward propagation
7. # Backward propagation
 # Compute the gradients
8. # Update parameters

C. Neural Network Training

1. # Training loop
2. # Predictions

D. Evaluate the model

1. Loading and Preprocessing the Data

Load the Boston housing dataset and preprocess it by splitting it into training and testing sets and scaling the features. Implement the necessary code for these steps.

Mathematical Concepts:

- **Splitting Data:** Training set (80%), Testing set (20%)
- **Standardizing Features:** $X_{\text{scaled}} = \frac{X - \mu}{\sigma}$

2. ReLU Activation Function

Implement the ReLU (Rectified Linear Unit) activation function and its derivative. Explain how these functions transform the input values mathematically.

Mathematical Formulas:

- ReLU Activation: $A = \max(0, Z)$
- ReLU Derivative: $d\text{ReLU} = \begin{cases} 1 & \text{if } Z > 0 \\ 0 & \text{otherwise} \end{cases}$

3. Mean Squared Error Loss Function

Define the Mean Squared Error (MSE) loss function. Calculate the loss between the true and predicted values.

Mathematical Formula:

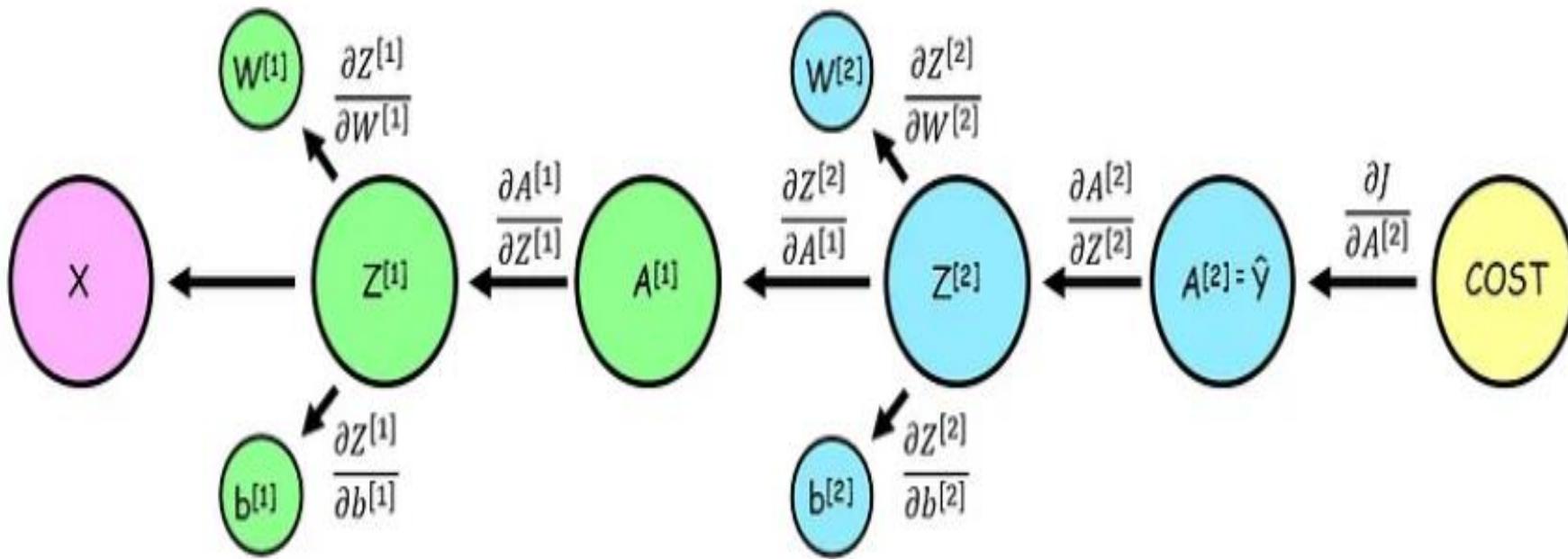
$$\text{MSE} = \frac{1}{m} \sum_{i=1}^m (y_i - \hat{y}_i)^2$$

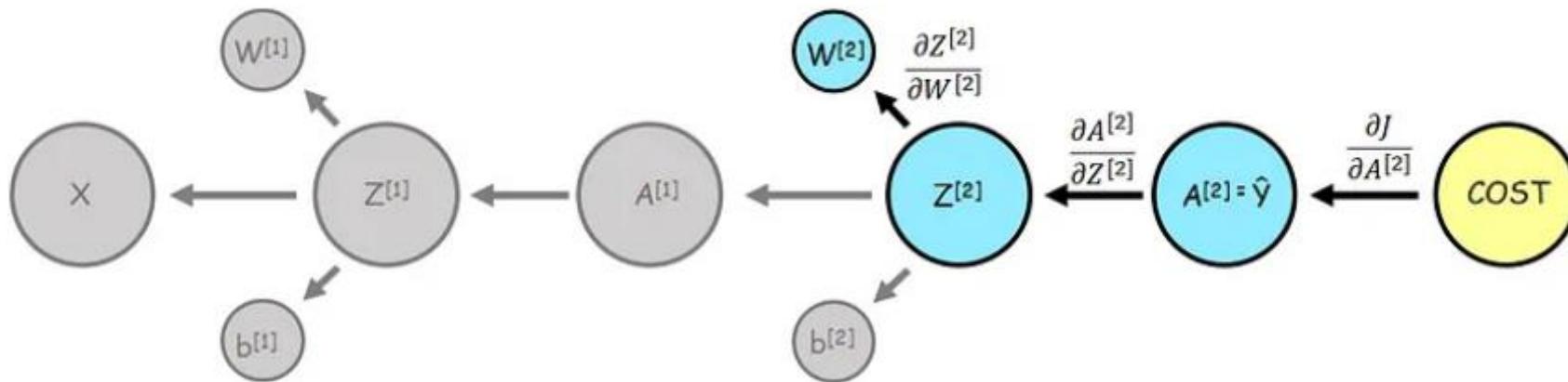
4. Forward Propagation

Implement the forward propagation step of a neural network with one hidden layer. Describe the formulas for the linear transformations and activation functions used.

Mathematical Formulas:

- First Layer: $Z_1 = XW_1 + b_1$
- ReLU Activation: $A_1 = \text{ReLU}(Z_1)$
- Second Layer: $Z_2 = A_1W_2 + b_2$





From the diagram above we can clearly see that the change in the cost J with respect to $W[2]$ is:

$$\frac{\partial J}{\partial W^{[2]}} = \frac{\partial J}{\partial A^{[2]}} \frac{\partial A^{[2]}}{\partial Z^{[2]}} \frac{\partial Z^{[2]}}{\partial W^{[2]}}$$

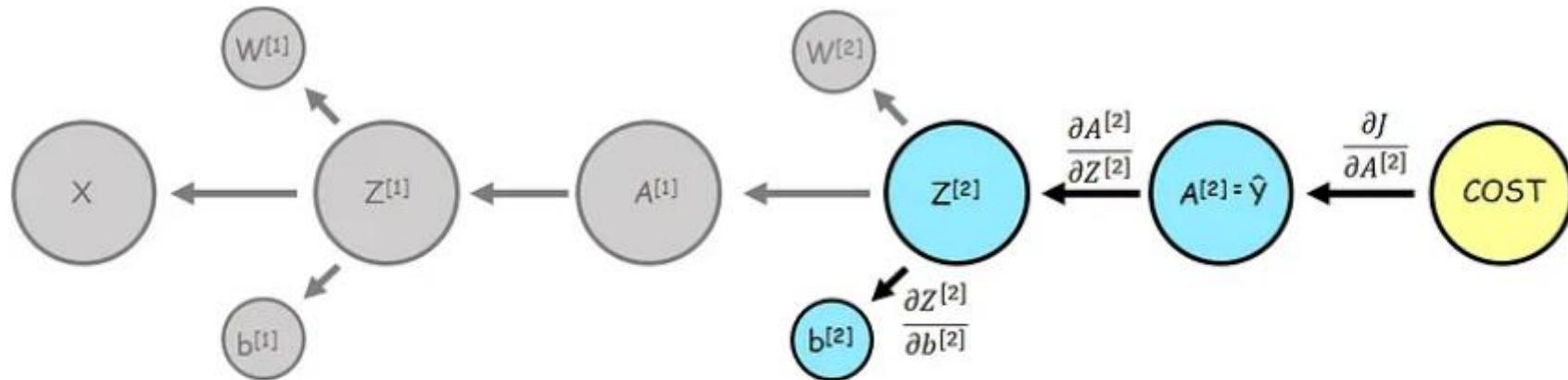
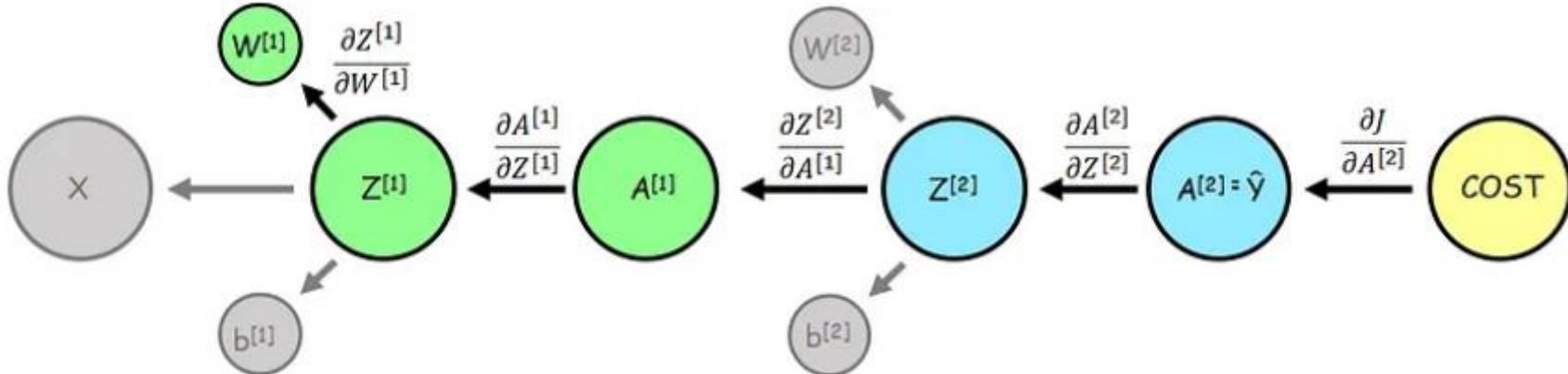


Figure 6. Gradient of the cost J with respect to the bias of layer two $b^{[2]}$ | Image by Author

$$\frac{\partial J}{\partial b^{[2]}} = \frac{\partial J}{\partial A^{[2]}} \frac{\partial A^{[2]}}{\partial Z^{[2]}} \frac{\partial Z^{[2]}}{\partial b^{[2]}}$$

Figure 7. $W^{[1]}$ gradient | Image by Author

The first two parts of the gradient were previously calculated for layer 2. The partial derivative of $Z^{[2]}$ with respect to $A^{[1]}$, is $W^{[2]}$:

Original function:

$$Z^{[2]} = W^{[2]}A^{[1]} + b^{[2]}$$

Partial derivative:

$$\left| \quad \frac{\partial Z^{[2]}}{\partial A^{[1]}} = W^{[2]} \right.$$

And if we follow the $b[1]$ gradient:

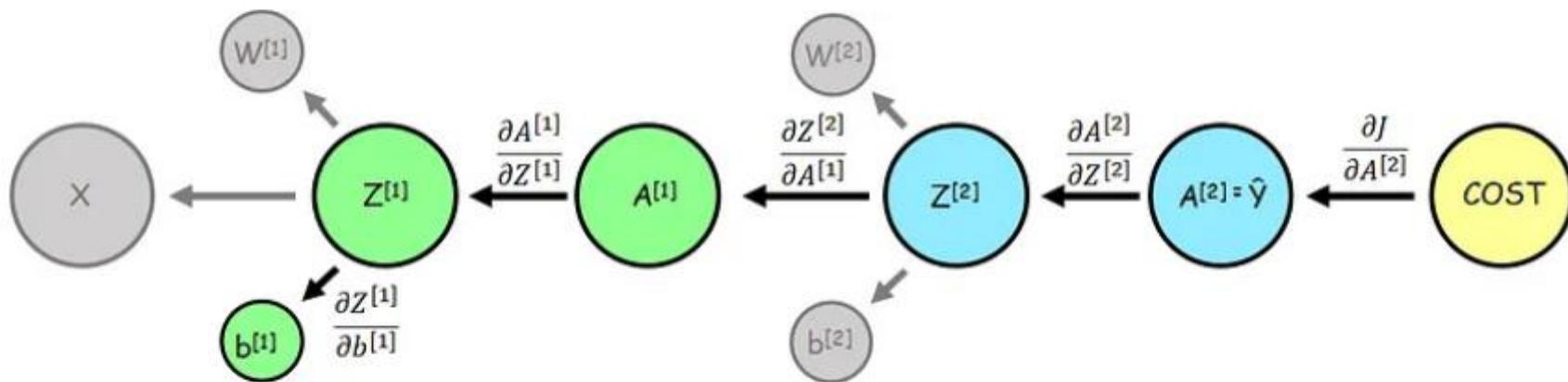


Figure 8. $b[1]$ gradient | Image by Author

5. Backward Propagation

Implement the backward propagation step to compute gradients for updating weights and biases.
Derive the gradients with respect to each parameter.

Mathematical Formulas:

- Gradient of Loss with Respect to Z_2 : $dZ_2 = \frac{\partial \text{Loss}}{\partial Z_2} = Z_2 - y$

- Gradients for W_2 and b_2 :

$$dW_2 = \frac{1}{m} A_1^T dZ_2$$

$$db_2 = \frac{1}{m} \sum_{i=1}^m dZ_2[i]$$

- Gradient for A_1 :

$$dA_1 = dZ_2 W_2^T$$

- Gradient of ReLU Activation: $dZ_1 = dA_1 \times \text{dReLU}(Z_1)$

- Gradients for W_1 and b_1 :

$$dW_1 = \frac{1}{m} X^T dZ_1$$

$$db_1 = \frac{1}{m} \sum_{i=1}^m dZ_1[i]$$

Summary of Derivatives

- $\delta_2 = \frac{\partial \mathcal{L}}{\partial Z2} = \hat{Y} - Y$
- $dW2 = \frac{1}{m} \delta_2 \cdot A1^T$
- $db2 = \frac{1}{m} \sum_{i=1}^m \delta_2$
- $\delta_1 = \frac{\partial \mathcal{L}}{\partial Z1} = (W2^T \cdot \delta_2) \cdot \sigma'(Z1)$
- $dW1 = \frac{1}{m} \delta_1 \cdot X^T$
- $db1 = \frac{1}{m} \sum_{i=1}^m \delta_1$

6. Parameter Update

Update the weights and biases of the neural network using the gradients computed during backpropagation. Apply gradient descent to update the parameters.

Mathematical Formulas:

- **Weight Update:** $W = W - \alpha \cdot dW$
- **Bias Update:** $b = b - \alpha \cdot db$

Where α is the learning rate.

7. Training Loop

Implement the training loop for the neural network. Iterate through the training data, perform forward and backward propagation, and update the parameters for each epoch.

Mathematical Formulas:

- **Loss Calculation:** $\text{Loss} = \frac{1}{m} \sum_{i=1}^m (y_i - \hat{y}_i)^2$
- **Parameter Update:** $W = W - \alpha \cdot dW$
 $b = b - \alpha \cdot db$

8. Testing the Model

Evaluate the performance of the trained neural network on the test set. Calculate and display the test loss to assess how well the model generalizes to new data.

Mathematical Formula:

- **Test Loss Calculation:** $\text{Test Loss} = \frac{1}{m_{\text{test}}} \sum_{i=1}^{m_{\text{test}}} (y_i - \hat{y}_i)^2$

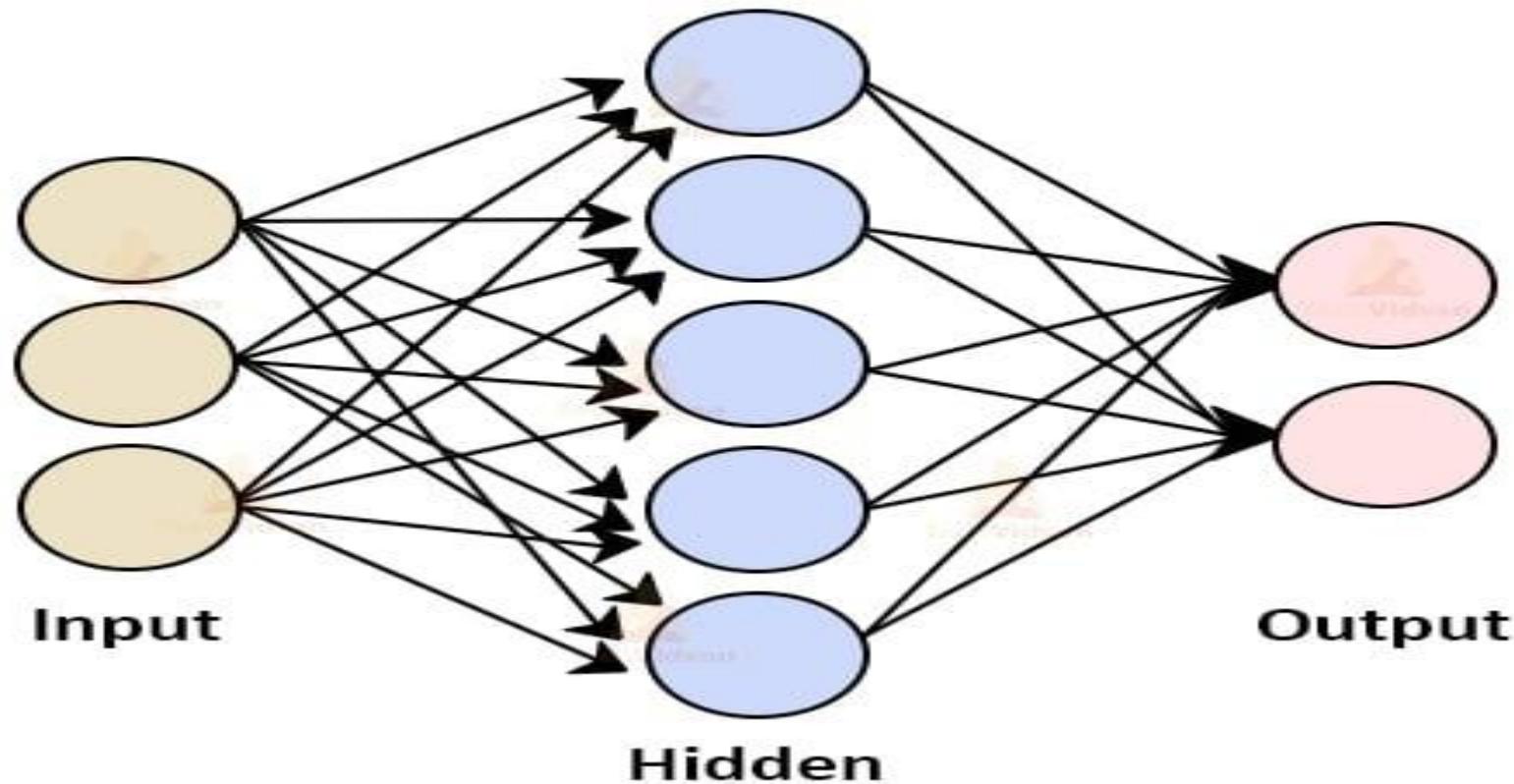


KESHAV MEMORIAL INSTITUTE OF TECHNOLOGY
AN AUTONOMOUS INSTITUTION - ACCREDITED BY NAAC WITH 'A' GRADE
Narayanauguda, Hyderabad.

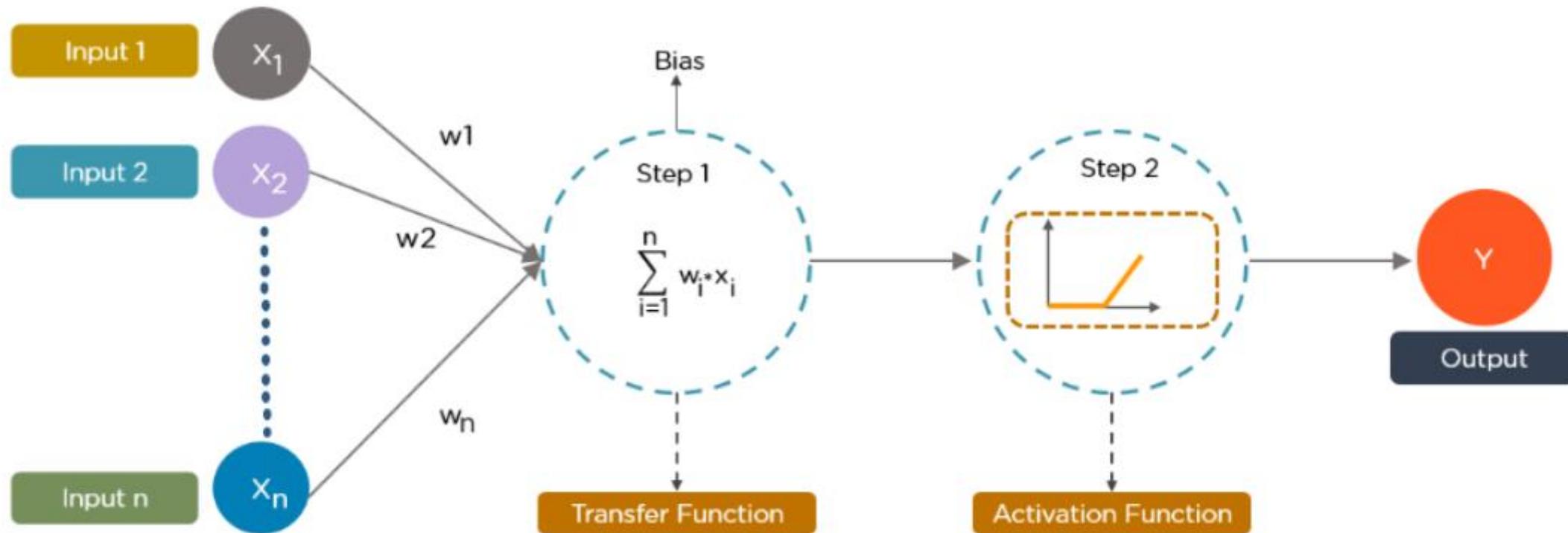
BUILD ANN FROM SCRATCH

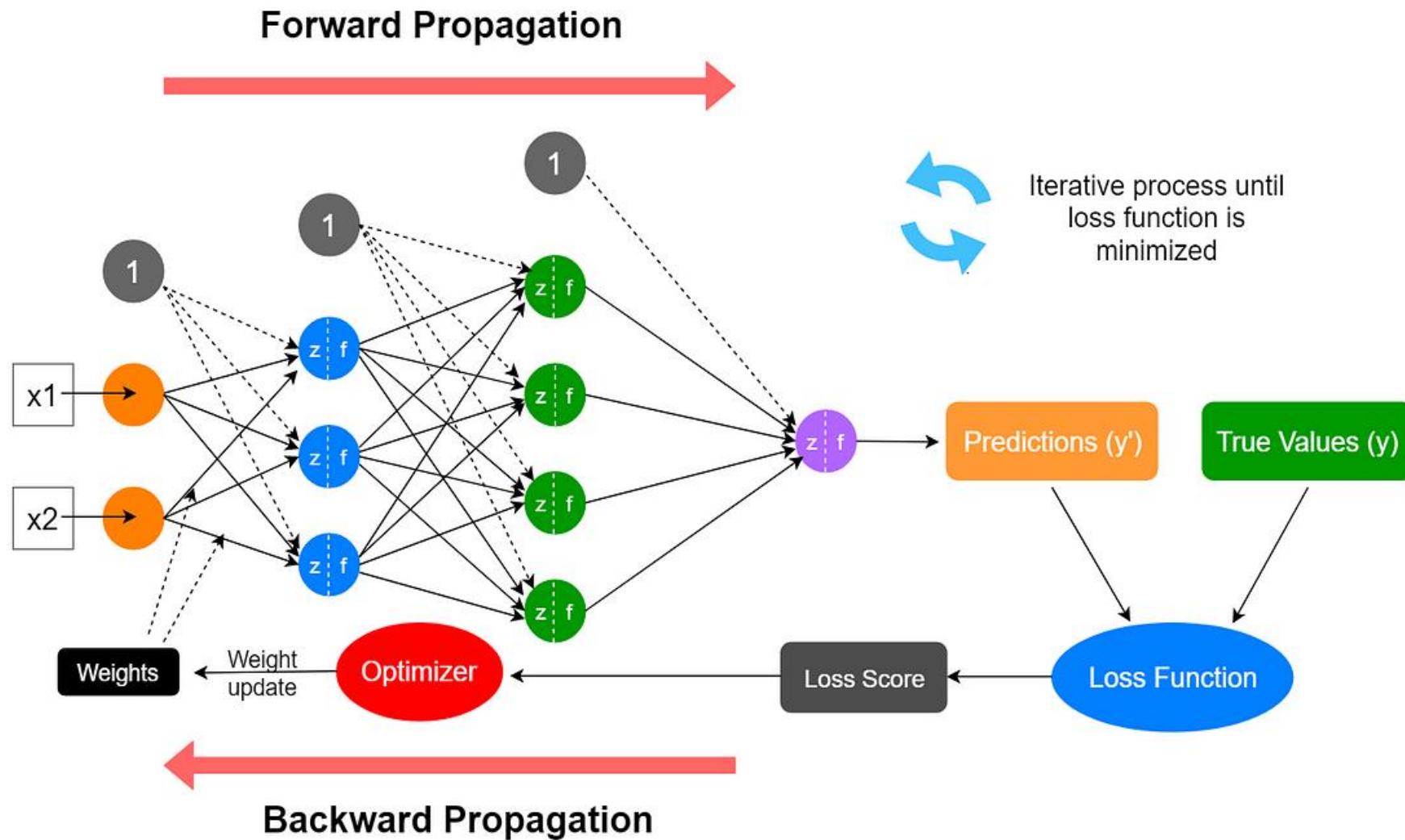
By
Asha

Architecture of Artificial Neural Network



WORK FLOW





BUILDING ANN FROM SCRATCH

Step 1: Load the Dataset

Import necessary libraries and Load the dataset insurance.csv.

Step 2: Separate the features (X) and the target variable (y) and Split the Dataset

Separate the features (X) and the target variable (y) and Split the data into training and testing sets. This helps in evaluating the performance of the ANN on unseen data.

Step 3: Normalize the Features

Normalize the input features to ensure all features contribute equally to the training process. This can be done using methods like standard scaling.

Step 4: Initialize the ANN Parameters

Decide on the architecture of the ANN, including the number of layers and the number of neurons in each layer. Initialize the weights and biases for each layer randomly.

Step 5: Define the Activation Functions

Choose activation functions for the neurons in each layer.

Step 6: Forward Propagation

Implement the forward propagation process, where the input data passes through the network layer by layer, applying the weights, biases, and activation functions to produce the final output.

Step 7: Compute the Loss

Calculate the loss using an appropriate loss function. For binary classification, the binary cross-entropy loss is typically used.

Step 8: Backward Propagation

Implement backward propagation to compute the gradients of the loss function with respect to the weights and biases. This involves calculating the gradient of the loss function from the output layer back to the input layer.

Step 9: Update the Weights and Biases

Use an optimization algorithm, such as gradient descent, to update the weights and biases using the gradients computed during backward propagation. This step aims to minimize the loss function.

Step 10: Train the ANN

Iterate through the training dataset for a specified number of epochs. In each epoch, perform forward propagation, compute the loss, perform backward propagation, and update the weights and biases.

Step 11: Evaluate the ANN

After training, evaluate the performance of the ANN on the testing set. This involves using the trained model to make predictions on the test data and comparing these predictions to the true labels to calculate metrics like accuracy.

Step 12: Make Predictions

Use the trained ANN to make predictions on new or unseen data by performing forward propagation with the learned weights and biases.

Step 1: Load the Dataset

Import necessary libraries and Load the dataset insurance.csv.

```
import numpy as np  
import pandas as pd  
from sklearn.model_selection import train_test_split  
from sklearn.preprocessing import StandardScaler
```

```
# Load the dataset  
data = pd.read_csv('insurance.csv')
```

Step 2: Separate the features (X) and the target variable (y) and Split the Dataset

Separate the features (X) and the target variable (y) and Split the data into training and testing sets. This helps in evaluating the performance of the ANN on unseen data.

```
# Separate features and target
```

```
X = data.drop('expenses', axis=1).values
```

```
y = data['expenses'].values.reshape(-1, 1)
```

```
# Split the data into training and testing sets
```

```
X_train, X_test, y_train, y_test = train_test_split(X_scaled, y, test_size=0.2,  
random_state=42)
```

Step 3: Normalize the Features

Normalize the input features to ensure all features contribute equally to the training process. This can be done using methods like standard scaling.

```
# Standardize the features  
scaler = StandardScaler()  
X_scaled = scaler.fit_transform(X)
```

Step 4: Initialize the ANN Parameters

Decide on the architecture of the ANN, including the number of layers and the number of neurons in each layer. Initialize the weights and biases for each layer randomly.

```
def initialize_parameters(input_size, hidden_size, output_size):
    np.random.seed(42)
    W1 = np.random.randn(input_size, hidden_size) * 0.01
    b1 = np.zeros((1, hidden_size))
    W2 = np.random.randn(hidden_size, output_size) * 0.01
    b2 = np.zeros((1, output_size))
    return W1, b1, W2, b2
```

Step 5: Define the Activation Functions

Choose activation functions for the neurons in each layer.

```
def relu(z):  
    return np.maximum(0, z)
```

```
def relu_derivative(z):  
    return np.where(z > 0, 1, 0)
```

```
def identity(z):  
    return z
```

Step 6: Forward Propagation

Implement the forward propagation process, where the input data passes through the network layer by layer, applying the weights, biases, and activation functions to produce the final output.

```
def forward_propagation(X, W1, b1, W2, b2):
    Z1 = np.dot(X, W1) + b1
    A1 = relu(Z1)
    Z2 = np.dot(A1, W2) + b2
    A2 = identity(Z2)
    return Z1, A1, Z2, A2
```

Step 8: Backward Propagation

Implement backward propagation to compute the gradients of the loss function with respect to the weights and biases. This involves calculating the gradient of the loss function from the output layer back to the input layer.

```
def backward_propagation(X, y, Z1, A1, A2, W2):
    m = X.shape[0]

    dZ2 = A2 - y
    dW2 = (1/m) * np.dot(A1.T, dZ2)
    db2 = (1/m) * np.sum(dZ2, axis=0, keepdims=True)

    dA1 = np.dot(dZ2, W2.T)
    dZ1 = dA1 * relu_derivative(Z1)
    dW1 = (1/m) * np.dot(X.T, dZ1)
    db1 = (1/m) * np.sum(dZ1, axis=0, keepdims=True)

    return dW1, db1, dW2, db2
```

23rd September 2024

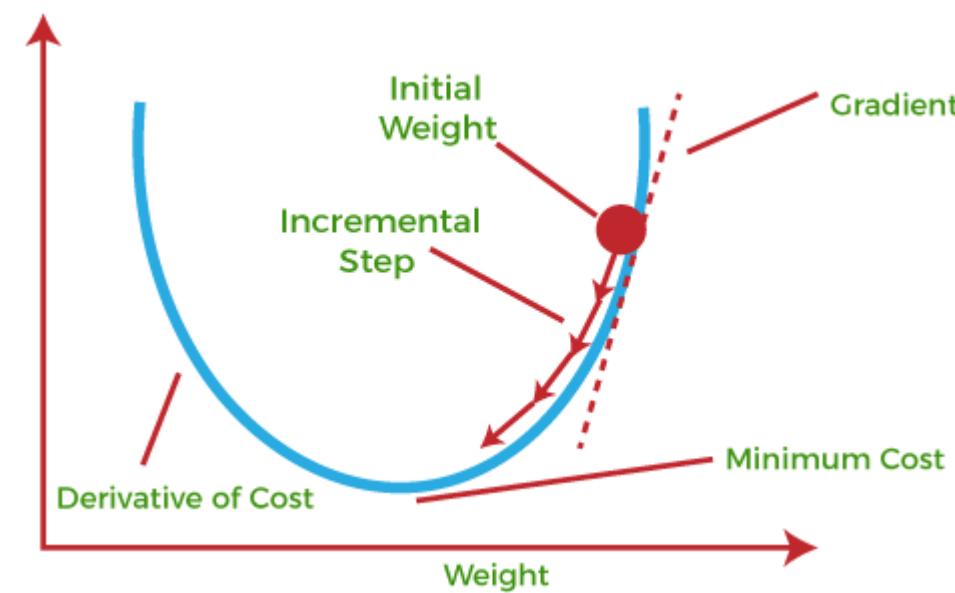
Step 9: Update the Weights and Biases

Use an optimization algorithm, such as gradient descent, to update the weights and biases using the gradients computed during backward propagation. This step aims to minimize the loss function.

Gradient Descent is known as one of the most commonly used optimization algorithms to train Neural Networks.

In mathematical terminology, Optimization algorithm refers to the task of minimizing/maximizing an objective function $f(x)$ parameterized by x .

Similarly, in machine learning, optimization is the task of minimizing the cost function parameterized by the model's parameters.



- Gradient Descent (GD) works by iteratively adjusting the weights or parameters of the model in the direction of the negative gradient of the cost function until the minimum of the cost function is reached.
- The learning happens during the backpropagation while training the neural network-based model.
- Gradient Descent is used to optimize the weight and biases based on the cost function. The cost function evaluates the difference between the actual and predicted outputs.

How do Gradients Work in Backpropagation?

In a neural network, the training process involves multiple layers of neurons. The process of computing the gradients for all the layers is called **backpropagation**. Here's how it works:

- **Forward Pass:** Inputs are passed through the network, layer by layer, and predictions are made.
- **Loss Calculation:** The loss (or error) is calculated by comparing the predicted output to the actual target values.
- **Backpropagation:** Using the chain rule from calculus, we calculate the gradients of the loss with respect to the network's parameters (weights and biases) by moving backward through the network, layer by layer.
 - The key here is that gradients for one layer depend on the gradients from the next layer, which is where the **chain rule** comes in.

Gradient Descent Algorithm

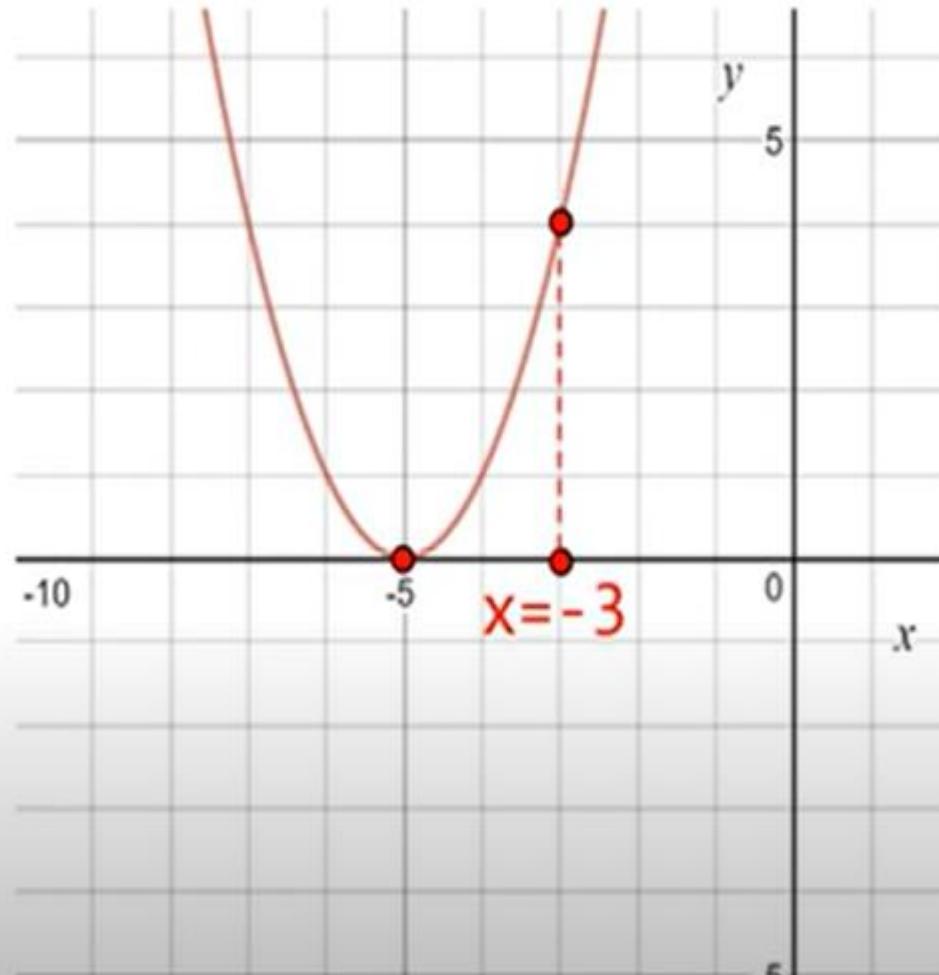
Once gradients are computed, they are used in **gradient descent** to update the parameters and reduce the loss. This process works as follows:

- 1. Initialize:** Start with random weights and biases.
- 2. Compute Gradients:** For each parameter (weight, bias), compute how much the loss changes when that parameter changes (this is the gradient).
- 3. Update Parameters:** Update each parameter by moving in the direction of the negative gradient.

$$\theta = \theta - \eta \frac{\partial L}{\partial \theta}$$

where θ is the parameter (weights or biases), $\partial L / \partial \theta$ is the gradient of the loss with respect to θ and η is the learning rate (step size).

Example

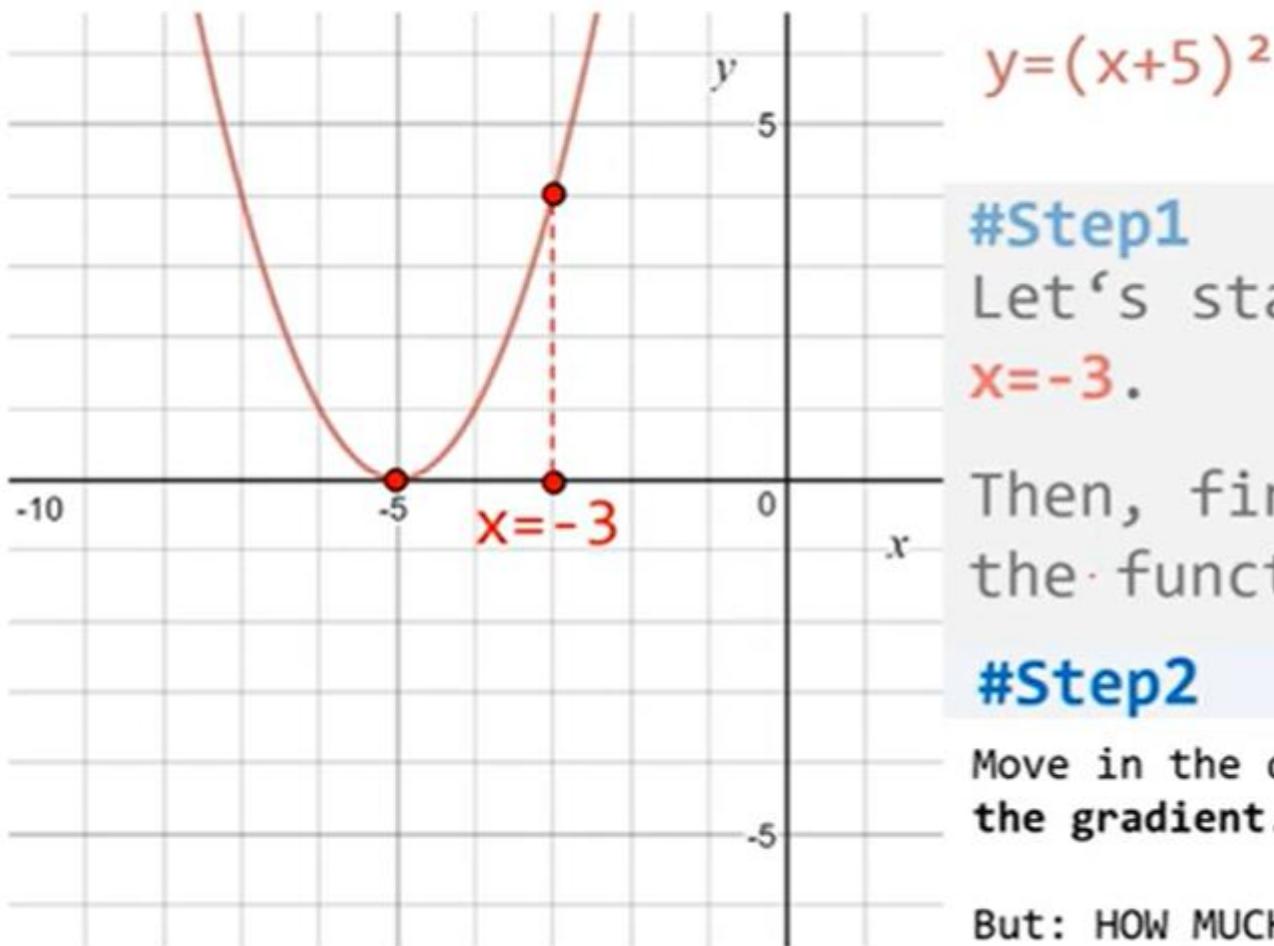


$y = 2(x+5)^2$

#Step1

Let's start from random point
 $x = -3$.

Then, find the gradient of
the function, $dy/dx = 2 \times (x+5)$.



$$y = (x+5)^2$$

#Step1

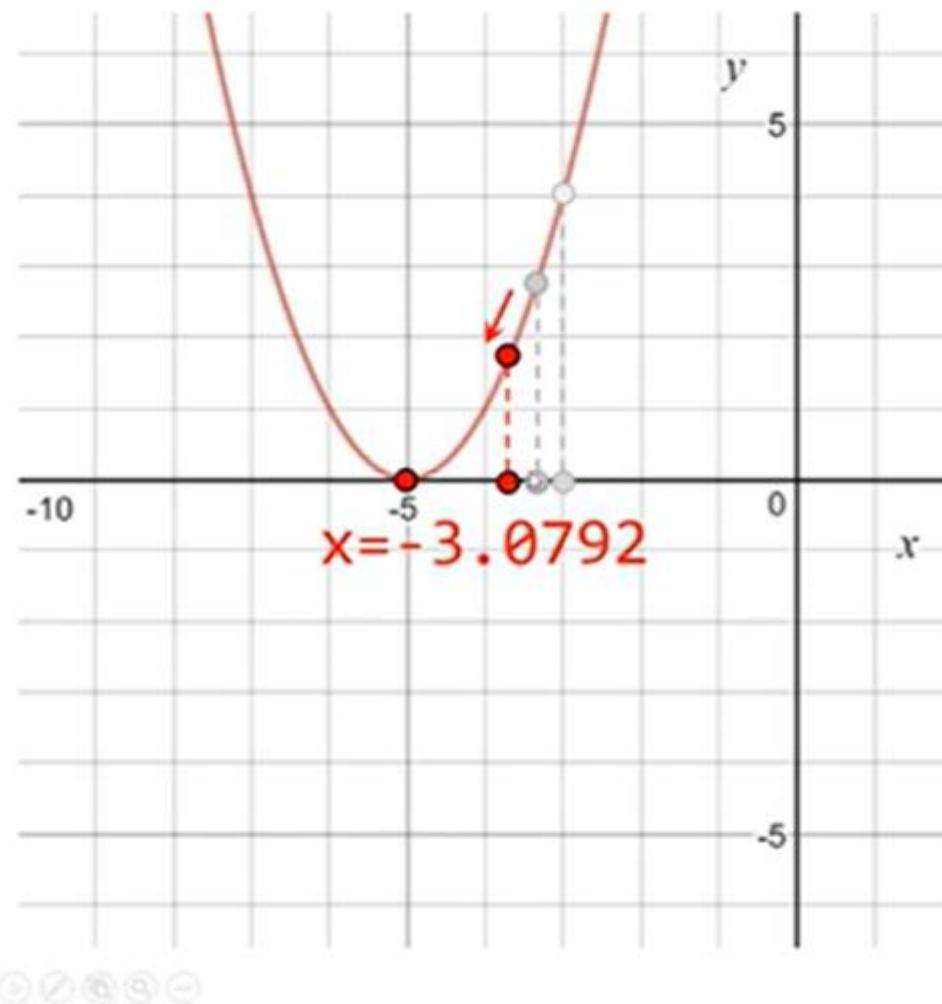
Let's start from random point
 $x = -3$.

Then, find the gradient of
the function, $dy/dx = 2x(x+5)$.

#Step2

Move in the direction of the negative of
the gradient.

But: HOW MUCH to move? For that, we define
a learning rate: $\text{learning_rate} = 0.01$



$$y = (x+5)^2$$

#Step3

Perform 2 iterations of gradient descent.

Initialize Parameters

$$x_0 = -3 \quad dy/dx = 2(x+5)$$

learning_rate = 0.01

Iteration 1

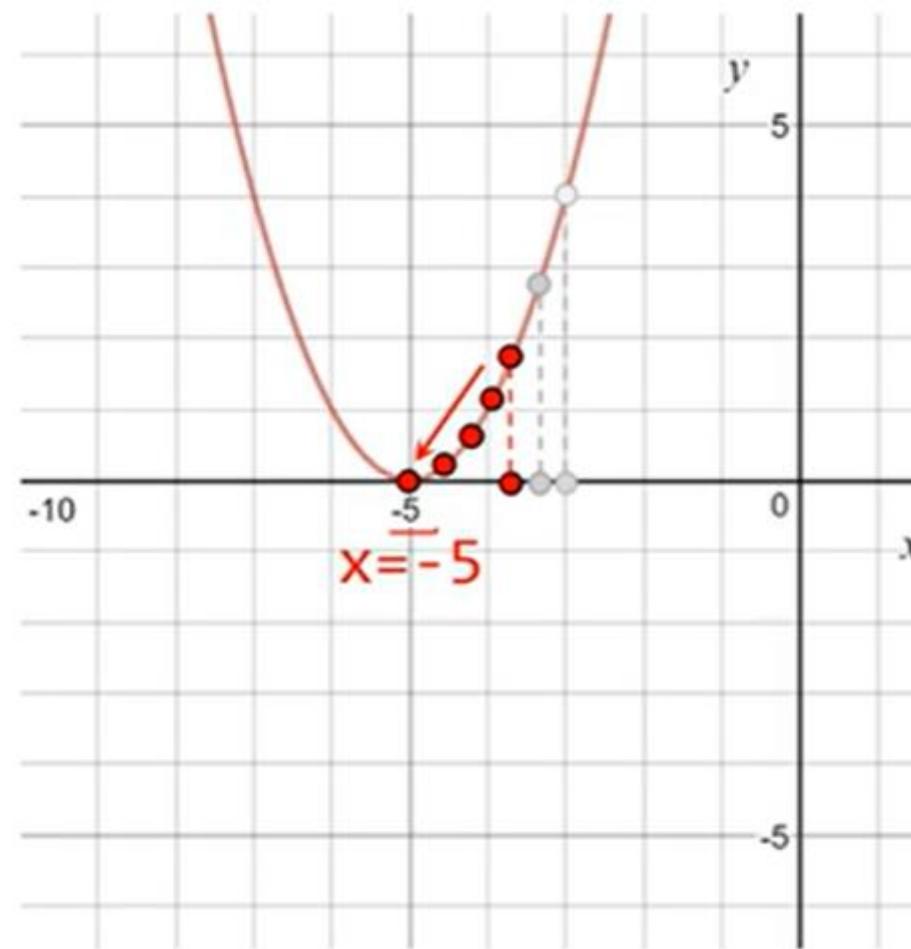
$$x_1 = x_0 - (\text{Learning rate}) \times (dy/dx)$$

$$x_1 = (-3) - (0.01) \times (2 \times ((-3)+5)) = -3.04$$

Iteration 2

$$x_2 = x_1 - (\text{Learning rate}) \times (dy/dx)$$

$$x_2 = (-3.04) - (0.01) \times (2 \times ((-3.04)+5)) = -3.0792$$

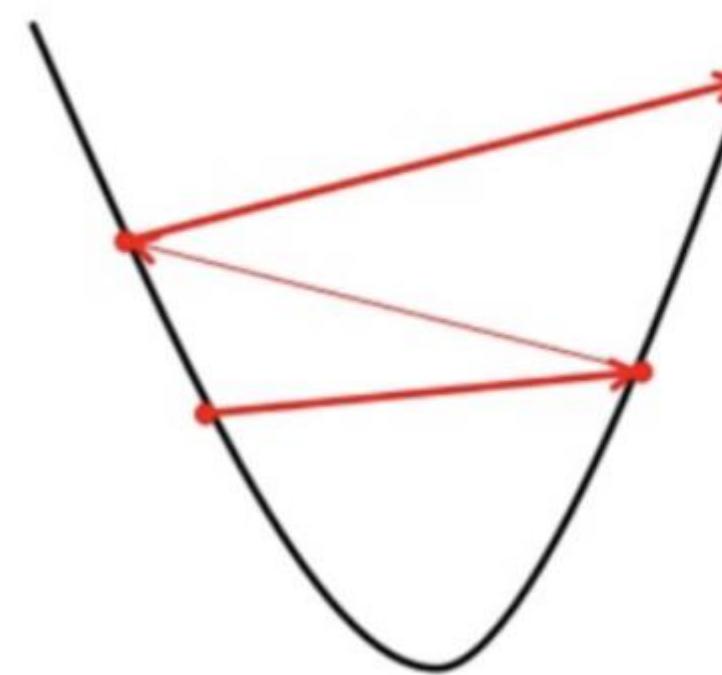


$$x = -5$$

Learning rate (also referred to as step size or the alpha)

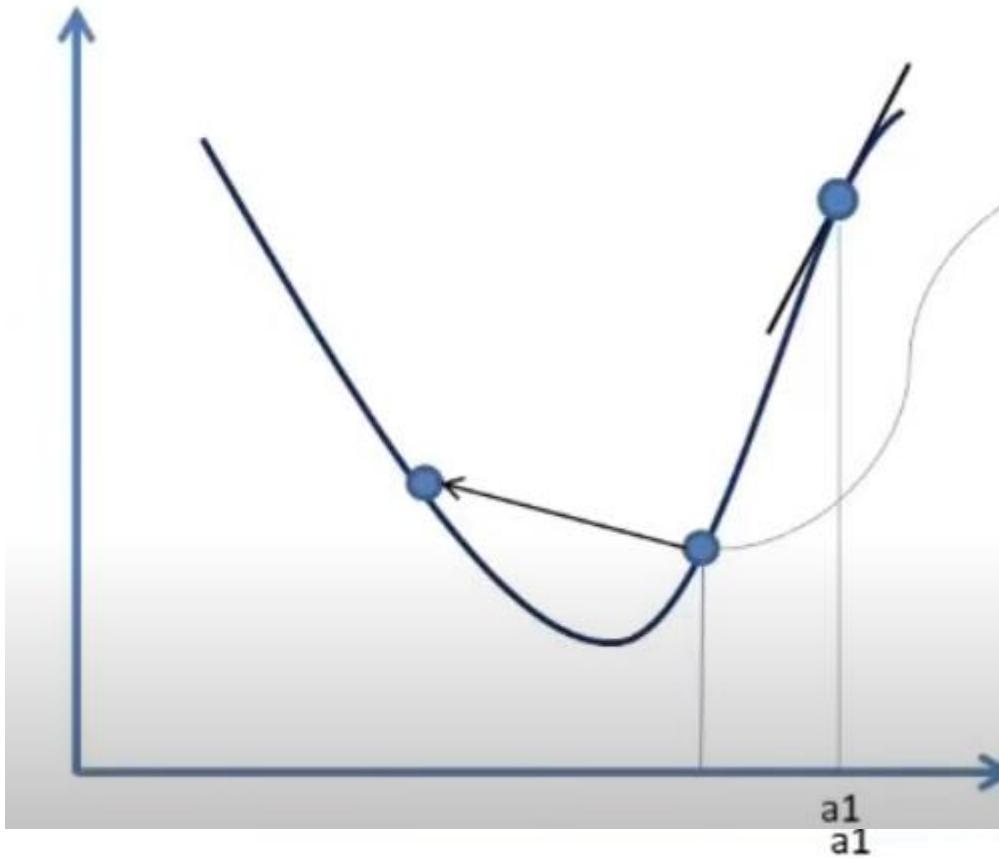
The **learning rate** controls how much to adjust the model's parameters (such as weights and biases) with respect to the gradients in each iteration of the training process.

Big learning rate



Small learning rate





We reached here in a step
from a_1 by setting α to 100.

Can you guess what can potentially
happen in the next iteration?

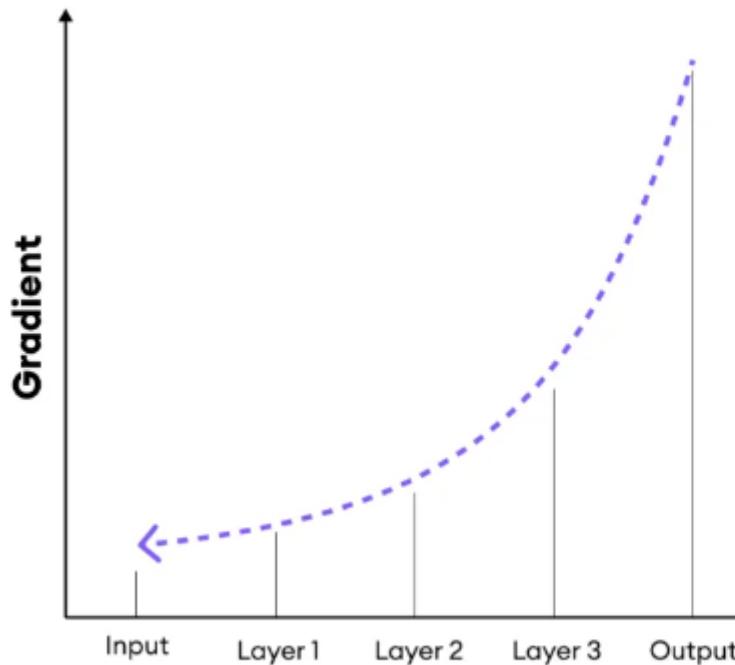
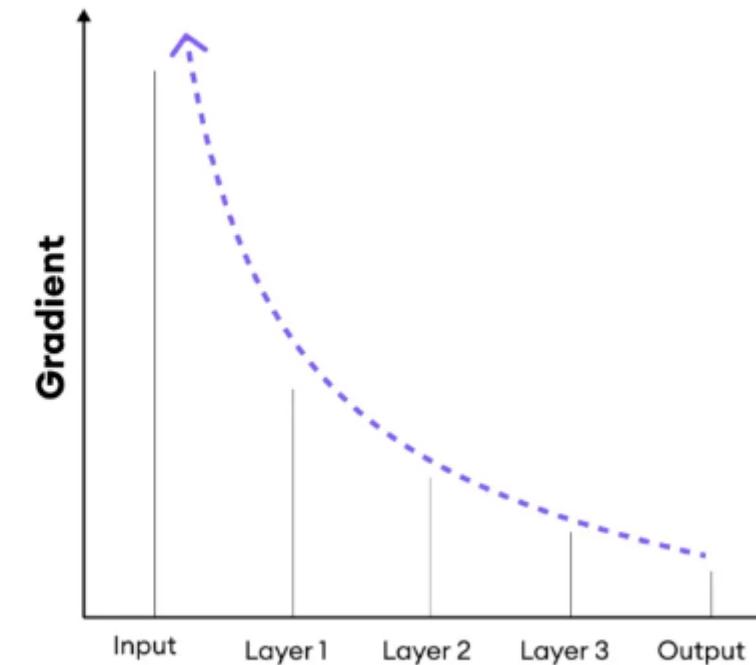
It will **fail to converge to minimum**
and will keep on **oscillating around
minimum** but can never converge
even in a billion iteration.

Solution??

Keep α small , its value is kept around **0.01** so that it neither makes gradient descent too slow nor does it fail to converge.

Challenges with Gradients

- **Vanishing Gradients:** In deep networks, the gradient can become very small in the early layers (close to zero), making learning slow or ineffective. This is called the vanishing gradient problem.
- **Exploding Gradients:** Gradients can also become excessively large, causing the model's parameters to diverge (get too large), which is the exploding gradient problem.

Vanishing Gradient**Exploding Gradient**

Step 10: Train the ANN

Iterate through the training dataset for a specified number of epochs. In each epoch, perform forward propagation, compute the loss, perform backward propagation, and update the weights and biases.

An **epoch** in machine learning refers to one complete pass of the entire training dataset through the model. During an epoch, the model processes all the input data once, updating its weights based on the error at each step.

The **training loop** in machine learning refers to the repeated process of passing data through the model, updating the model's parameters, and improving its performance over time. Here's a basic breakdown of a training loop:

Steps in a Training Loop:

1. Initialize Parameters: Before the loop starts, you initialize the model's parameters (weights and biases) randomly or with some small values.

2. Loop for Each Epoch: The model processes the entire training dataset for a certain number of epochs (iterations). Each epoch involves:

1. **Forward Propagation:** The input data is passed through the model, producing predictions.
2. **Loss/Cost Calculation:** The error (cost or loss) between the predicted output and the actual output is computed using a cost function.
3. **Backward Propagation:** The gradients (derivatives) of the loss function with respect to the model's parameters are calculated, showing how much to adjust each parameter.
4. **Parameter Update:** Using the gradients and a learning rate, the model's parameters (weights and biases) are updated using gradient descent.

3. Track Progress: Optionally, after a certain number of epochs, you can monitor the training progress by printing the cost, accuracy, or other performance metrics.

4. Repeat: The loop continues for a predefined number of epochs, or until the model achieves satisfactory performance.

Key Concepts in the Training Loop:

- **Epochs:** The number of times the model processes the entire dataset.
- **Batch:** Often, data is divided into smaller batches to process efficiently (Batch Gradient Descent).
- **Learning Rate:** Controls the size of the steps during the parameter update.

The training loop continues until the model reaches the desired performance or until the maximum number of epochs is reached.

Step 11: Evaluate the ANN

After training, evaluate the performance of the ANN on the testing set. This involves using the trained model to make predictions on the test data and comparing these predictions to the true labels to calculate metrics like accuracy.

```
def predict(X, W1, b1, W2, b2):  
    # Your code here  
    return predictions
```

The predict function effectively uses the trained neural network to compute and return predictions for new input data by leveraging the forward propagation process.

Step 12: Make Predictions

Use the trained ANN to make predictions on new or unseen data by performing forward propagation with the learned weights and biases.

1. Train the Model: The neural network learns from training data by adjusting its parameters based on the inputs and corresponding outputs over several iterations (epochs).

2. Make Predictions: The trained model is used to predict outputs for a new test dataset that it hasn't seen before.

3. Calculate Mean Squared Error (MSE): The predictions are compared to the actual outcomes from the test dataset to measure the prediction error using MSE.

4. Print MSE: The MSE value is displayed, indicating the model's accuracy—lower values mean better performance.

- **High Cost and MSE:** If Both the cost after the first epoch and the test MSE are very high, suggesting that the model is not learning well. This could be due to various reasons, such as:
 - An inappropriate learning rate (too high or too low).
 - Insufficient training.
 - Poor data quality or features.
 - Model architecture issues



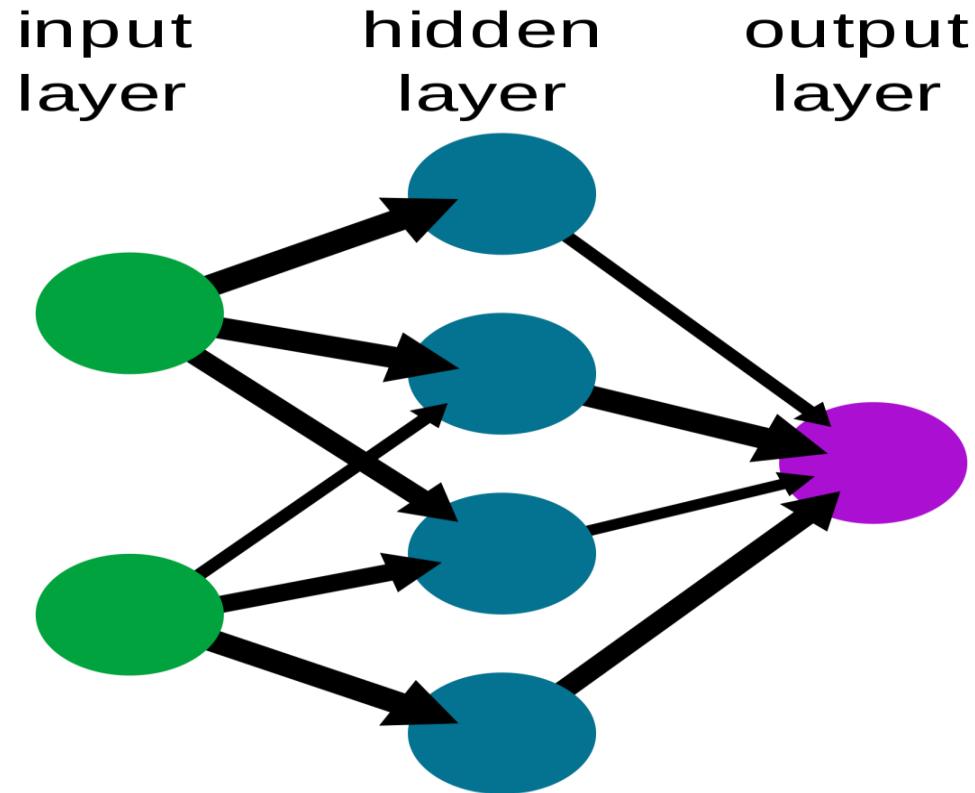
KESHAV MEMORIAL INSTITUTE OF TECHNOLOGY
AN AUTONOMOUS INSTITUTION - ACCREDITED BY NAAC WITH 'A' GRADE
Narayanauguda, Hyderabad.

Deep Learning

ANN-Binary classification

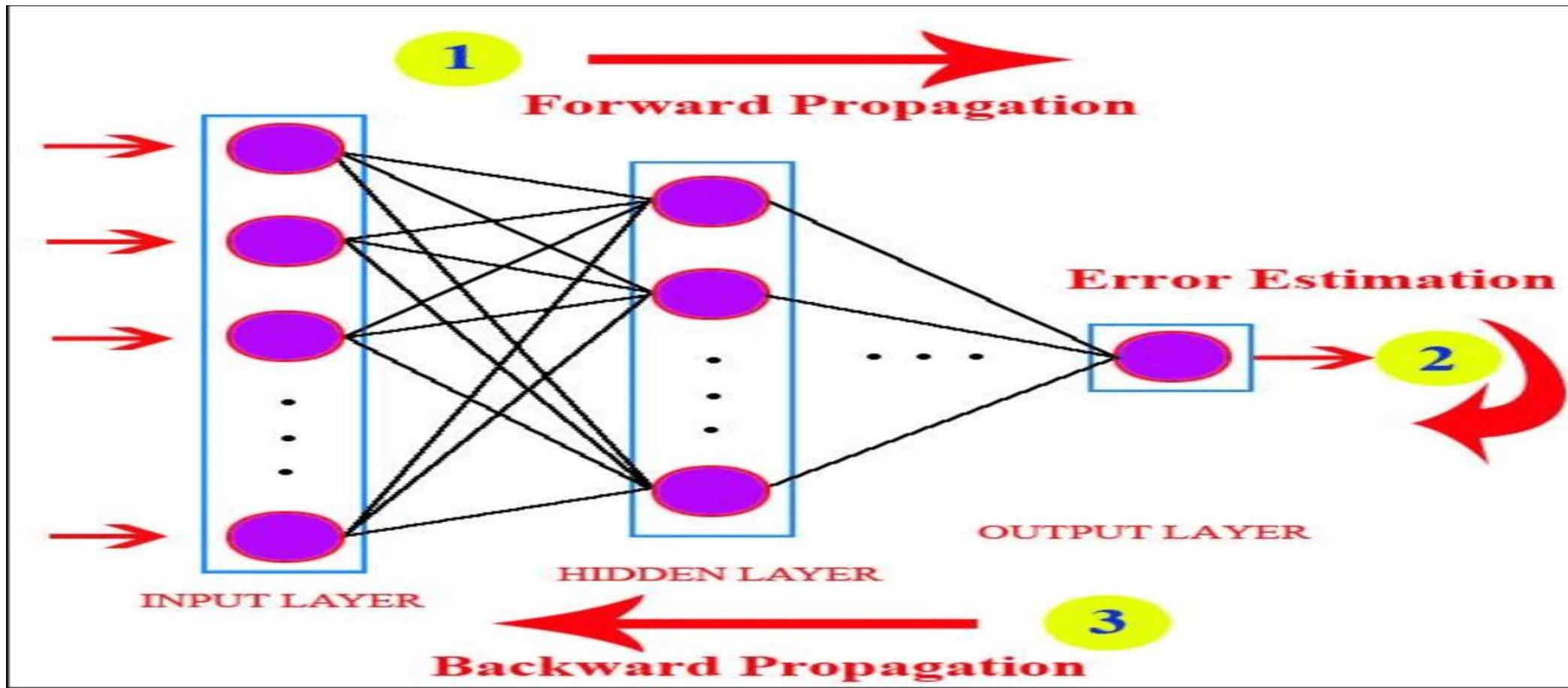
BY
ASHA

A simple neural network



A simple neural network consists of three components :

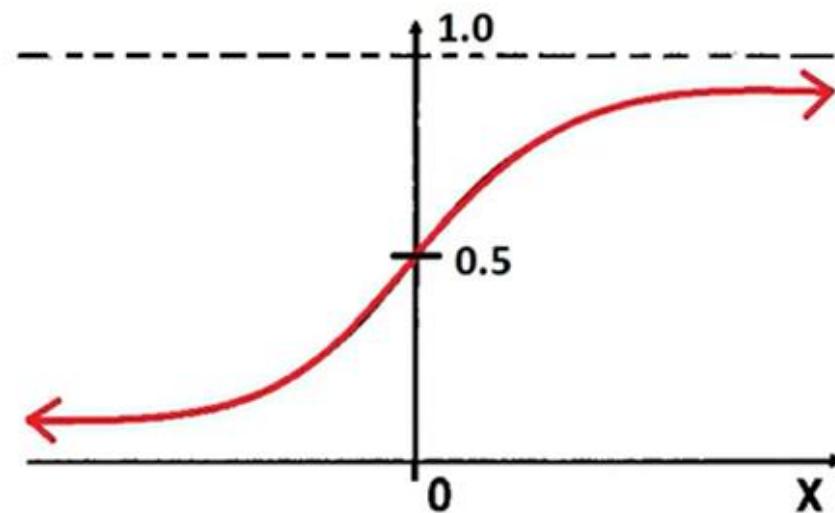
- Input layer
- Hidden layer
- Output layer



Common Activation Functions

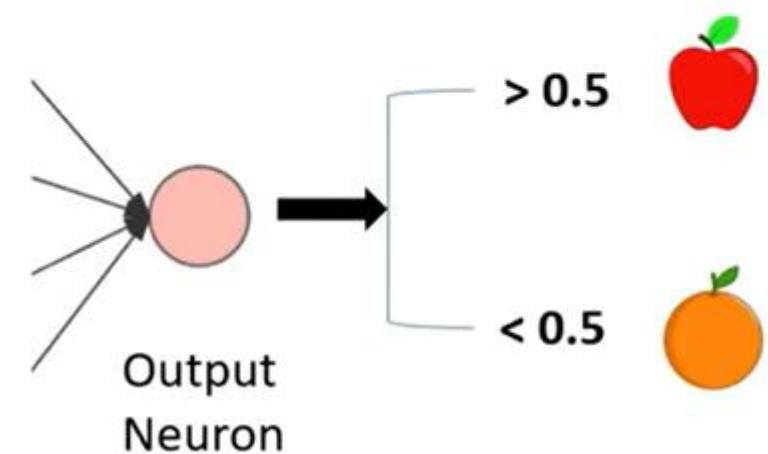
- **Sigmoid:** Maps inputs to the range (0, 1). It's used in binary classification tasks but can suffer from vanishing gradient problems in deep networks.
- **Tanh (Hyperbolic Tangent):** Maps inputs to the range (-1, 1), centering the output, which can sometimes lead to better convergence than sigmoid. However, it also suffers from vanishing gradient problems.
- **ReLU (Rectified Linear Unit):** Introduces non-linearity by outputting the input directly if it's positive and zero otherwise. It's widely used in deep learning due to its simplicity and effectiveness. It also helps with the vanishing gradient problem.
- **Softmax:** Typically used in the output layer for multi-class classification problems. It converts logits (raw output values) into probabilities that sum to 1 across the classes.

Sigmoid Function



Sigmoid

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



Loss Function

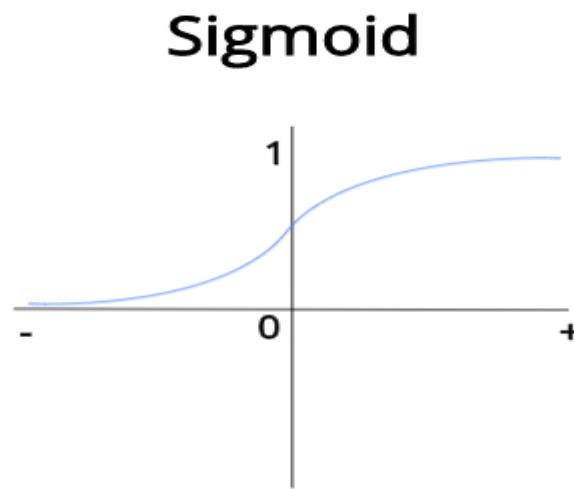
- A measure of how well the neural network's predictions match the actual results. Common loss functions include:
- **Mean Squared Error (MSE):** For regression tasks.
- **Cross-Entropy Loss:** For classification tasks.

Classification Loss:

- ***Binary Classification:***
- Binary Cross Entropy Loss
- ***Multi-Class Classification:***
- Categorical Cross Entropy Loss

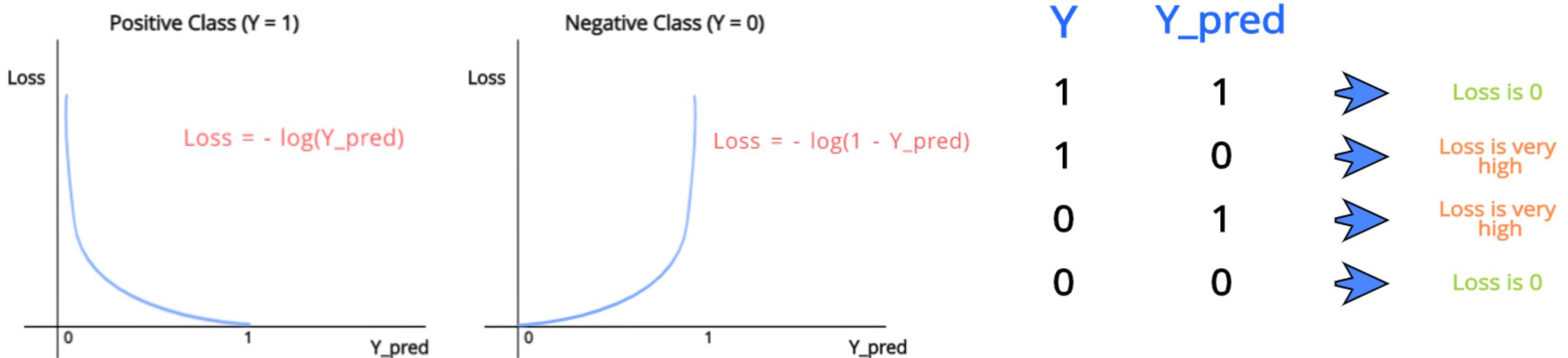
Binary Classification

- In binary classification, there will be only one node in the output layer even though we will be predicting between two classes. In order to get the output in a probability format, we need to apply an activation function. Since probability requires a value in between 0 and 1 we will use the **sigmoid function** which can *squish* any real value to a value between 0 and 1.



If the output is above 0.5 (50% Probability), we will consider it to be falling under the **positive class** and if it is below 0.5 we will consider it to be falling under the **negative class**.

The loss function we use for binary classification is called **binary cross entropy (BCE)**.



As you can see, there are two separate functions, one for each value of Y . When we need to predict the **positive class** ($Y = 1$), we will use $\text{Loss} = -\log(Y_{\text{pred}})$. And when we need to predict the **negative class** ($Y = 0$), we will use $\text{Loss} = -\log(1-Y_{\text{pred}})$.

We can mathematically represent the entire loss function into one equation as follows:

$$\text{Loss} = (Y)(-\log(Y_{pred})) + (1 - Y)(-\log(1 - Y_{pred}))$$

Remains when $Y = 1$

Removed when $Y = 0$

Remains when $Y = 0$

Removed when $Y = 1$

BUILDING ANN FROM SCRATCH

Step 1: Load the Dataset

Load the breast cancer dataset from scikit-learn. The Churn Modeling dataset is commonly used in machine learning to predict customer churn in the banking sector. Churn refers to the loss of customers, which can significantly impact a company's profitability. The dataset typically contains various features about customers that can help predict whether they will leave the bank.

Step 2: Split the Dataset

Split the data into training and testing sets. This helps in evaluating the performance of the ANN on unseen data.

Step 3: Normalize the Features

Normalize the input features to ensure all features contribute equally to the training process. This can be done using methods like standard scaling.

Step 4: Initialize the ANN Parameters

Decide on the architecture of the ANN, including the number of layers and the number of neurons in each layer. Initialize the weights and biases for each layer randomly.

Step 5: Define the Activation Functions

Choose activation functions for the neurons in each layer. Common choices are the sigmoid function for the output layer (binary classification).

Step 6: Forward Propagation

Implement the forward propagation process, where the input data passes through the network layer by layer, applying the weights, biases, and activation functions to produce the final output.

Step 7: Compute the Loss

Calculate the loss using an appropriate loss function. For binary classification, the binary cross-entropy loss is typically used.

Step 8: Backward Propagation

Implement backward propagation to compute the gradients of the loss function with respect to the weights and biases. This involves calculating the gradient of the loss function from the output layer back to the input layer.

Step 9: Update the Weights and Biases

Use an optimization algorithm, such as gradient descent, to update the weights and biases using the gradients computed during backward propagation. This step aims to minimize the loss function.

Step 10: Train the ANN

Iterate through the training dataset for a specified number of epochs. In each epoch, perform forward propagation, compute the loss, perform backward propagation, and update the weights and biases.

Step 11: Evaluate the ANN

After training, evaluate the performance of the ANN on the testing set. This involves using the trained model to make predictions on the test data and comparing these predictions to the true labels to calculate metrics like accuracy.

Step 12: Make Predictions

Use the trained ANN to make predictions on new or unseen data by performing forward propagation with the learned weights and biases.

Overview of the Churn Modeling Dataset

A typical **Churn Modeling** dataset for a bank might include the following columns (features):

- 1.CustomerID:** Unique identifier for each customer.
- 2.Surname:** The surname of the customer (often excluded for modeling).
- 3.CreditScore:** The credit score of the customer.
- 4.Geography:** Country of residence (e.g., France, Spain, Germany).
- 5.Gender:** Gender of the customer (e.g., Male, Female).
- 6.Age:** Age of the customer.
- 7.Tenure:** Number of years the customer has been with the bank.
- 8.Balance:** Account balance.
- 9.NumberOfProducts:** Number of products the customer has with the bank.
- 10.HasCrCard:** Whether the customer has a credit card (1 = Yes, 0 = No).
- 11.IsActiveMember:** Whether the customer is an active member (1 = Yes, 0 = No).
- 12.EstimatedSalary:** The estimated salary of the customer.
- 13.Exited:** The target variable (1 if the customer has left the bank, 0 otherwise).

Sample Dataset

Here is a small sample of what the Churn Modeling dataset might look like:

CustomerID	Surname	CreditScore	Geography	Gender	Age	Tenure	Balance	NumberOfProducts	HasCrCard	IsActiveMember	EstimatedSalary	Exited
15634602	Hargrave	619	France	Female	42	2	0	1	1	1	101348.88	1
15647311	Hill	608	France	Male	41	1	83807.86	1	1	1	112542.58	0
15619304	Onofre	502	Spain	Male	42	8	159660.80	3	1	1	113931.57	0
15664772	Araujo	699	Spain	Female	39	3	0	2	1	1	115641.75	0
15640423	Hughes	764	Germany	Male	30	1	0	2	1	0	108439.61	1



THANK YOU



KESHAV MEMORIAL INSTITUTE OF TECHNOLOGY
AN AUTONOMOUS INSTITUTION - ACCREDITED BY NAAC WITH 'A' GRADE
Narayanauguda, Hyderabad.

BUILD ANN FROM SCRATCH

Regression

Exercise-1

09-09-2024

1. Data Loading and Preprocessing:

We load the Boston Housing dataset, scale the features, and split the data into training and testing sets.

2. Forwardpropogation:

Compute the linear combination of inputs and weights for the hidden layer (Z_1).

Apply the sigmoid activation function to get the activations of the hidden layer (A_1).

Compute the linear combination for the output layer (Z_2).

3. Loss Calculation:

Compute the Mean Squared Error (MSE) loss between the predicted and actual target values.

4. Backpropagation:

Compute the gradients of the loss with respect to the weights and biases of the output layer.

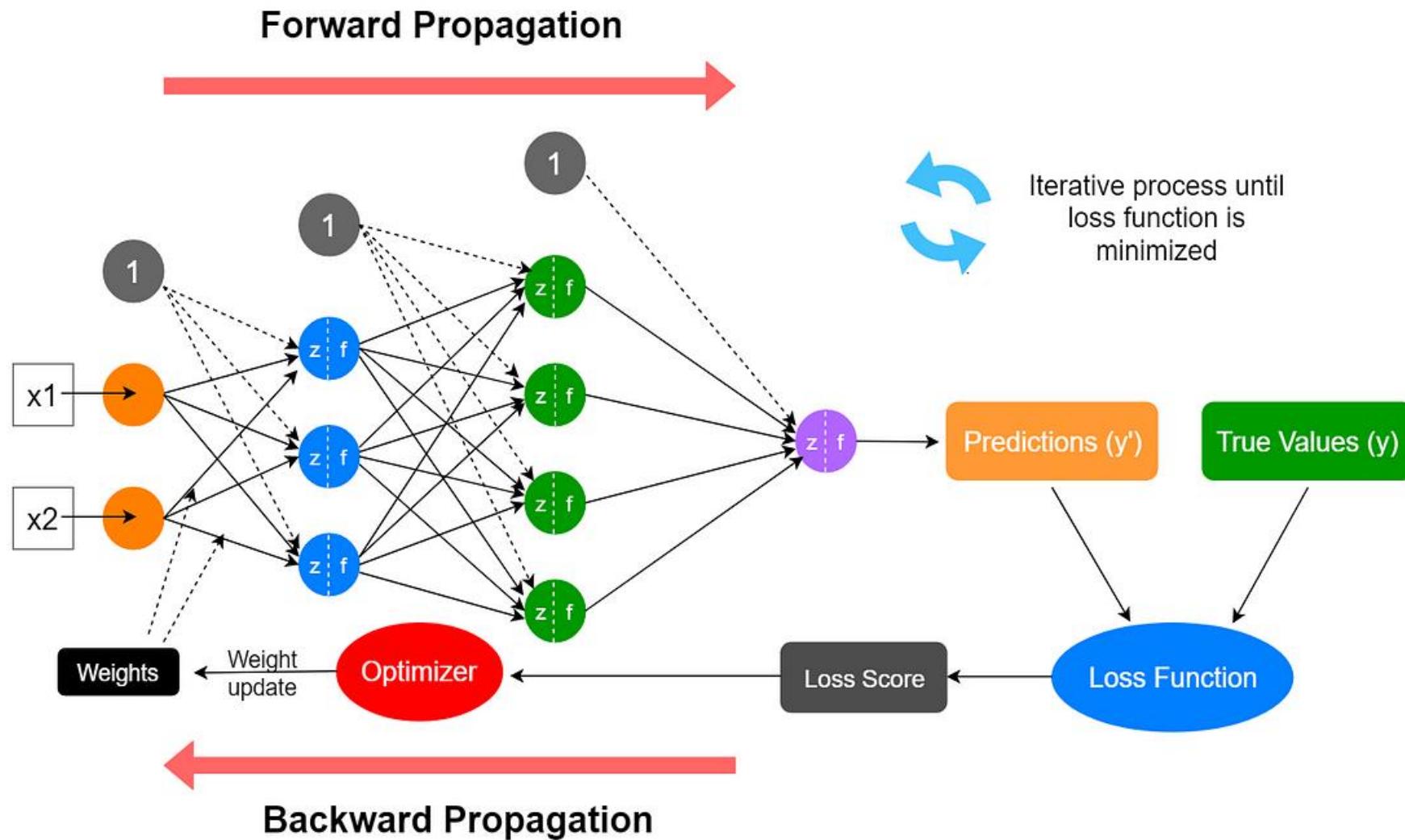
Compute the gradients for the hidden layer weights and biases using the chain rule and sigmoid derivative.

Update the weights and biases using gradient descent.

5. Training and Testing:

Train the ANN on the training data and print the training loss every 100 epochs.

Evaluate the model on the test data and print the test loss.



You are tasked with creating a simple artificial neural network (ANN) from scratch to find price of the house using boston data using the Scikit-learn boston dataset. Write a Python script that includes the following steps:

A. Data Loading and preprocessing

- 1.#Import Necessary libraries
- 2.# Load the Boston Housing dataset and Split the data into dependent and independent variables(X,y)
- 3.# Standardize the features
- 4.# Split the dataset into training and testing sets

B. Neural Network Implementation

- 1.# Define the architecture
- 2.# Initialize weights and biases
- 3.# Define Activation function (ReLU)
- 4.# Define Derivative of ReLU
- 5 # Forward propagation
- 6 # Mean squared error loss
- 7# Backward propagation
- ## Compute the gradients
- 8.# Update parameters

C . Evaluate the model

1. Loading and Preprocessing the Data

Load the Boston housing dataset and preprocess it by splitting it into training and testing sets and scaling the features. Implement the necessary code for these steps.

Mathematical Concepts:

- **Splitting Data:** Training set (80%), Testing set (20%)
- **Standardizing Features:** $X_{\text{scaled}} = \frac{X - \mu}{\sigma}$

2. ReLU Activation Function

Implement the ReLU (Rectified Linear Unit) activation function and its derivative. Explain how these functions transform the input values mathematically.

Mathematical Formulas:

- ReLU Activation: $A = \max(0, Z)$
- ReLU Derivative: $d\text{ReLU} = \begin{cases} 1 & \text{if } Z > 0 \\ 0 & \text{otherwise} \end{cases}$

4. Forward Propagation

Implement the forward propagation step of a neural network with one hidden layer. Describe the formulas for the linear transformations and activation functions used.

Mathematical Formulas:

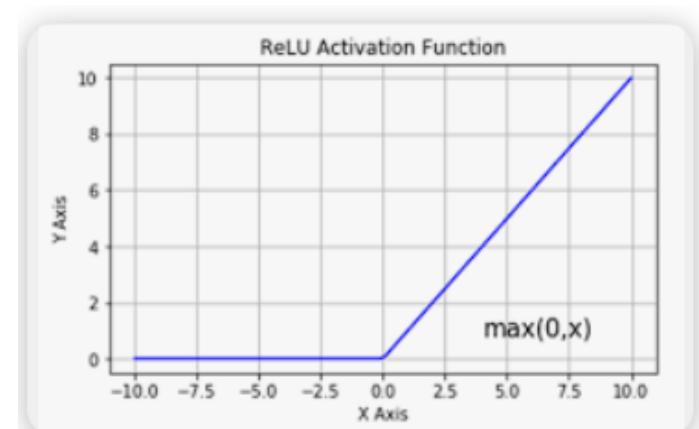
- First Layer: $Z_1 = XW_1 + b_1$
- ReLU Activation: $A_1 = \text{ReLU}(Z_1)$
- Second Layer: $Z_2 = A_1W_2 + b_2$

Forward Propagation

1. Forward Step for Hidden Layer:

$$Z1 = W1 \cdot X + b1$$

$$A1 = \text{ReLU}(Z1) = \max(0, Z1)$$



2. Forward Step for Output Layer:

$$Z2 = W2 \cdot A1 + b2$$

$$\hat{Y} = Z2 \quad (\text{since it's a regression problem})$$

Why You Might Skip the Identity Function:

If you don't use the identity function, you're essentially doing the same thing — passing the value `z` directly. So, the need for it comes down to maintaining a **clear, structured design and potential future modifications**. It doesn't have an immediate impact on the model's performance but helps for code readability and maintaining consistency in activation layers.

Key Points:

- **Technically**, you don't need the identity function because passing the raw value without it has the same effect.
- **Practically**, it's useful for keeping a consistent and readable model structure, especially when designing deep models or when other layers always apply some non-linear transformations.

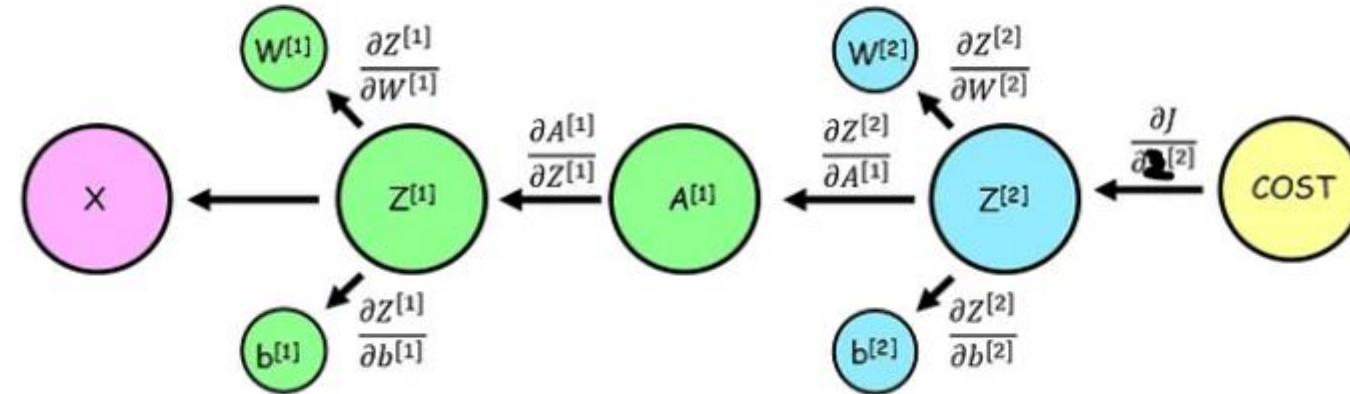
Loss Function

Mean Squared Error (MSE) loss:

$$\mathcal{L} = \frac{1}{m} \sum_{i=1}^m (\hat{Y}^{(i)} - Y^{(i)})^2$$

where:

- $\hat{Y}^{(i)}$ is the predicted value for the i -th sample.
- $Y^{(i)}$ is the true value for the i -th sample.
- m is the number of samples.

**Parameters:**

- X : Input data
- y : True labels, shape
- Z_1 : Linear output of the hidden layer before activation
- A_1 : Activation output from the hidden layer
- Z_2 : Linear output of the output layer before activation
- A_2 : Final output (predicted values)

Backward Propagation

1. Compute the Derivative of the Loss with Respect to \hat{Y}

For MSE:

$$\frac{\partial \mathcal{L}}{\partial \hat{Y}} = \frac{2}{m}(\hat{Y} - Y)$$

Let's denote this as:

$$\delta_2 = \frac{2}{m}(\hat{Y} - Y)$$

$$= \hat{Y}^{(i)} - Y^{(i)}$$

In practice, when implementing backpropagation, the factor 2/m is sometimes omitted for simplicity.

Simplified Expression

In practice, especially in the context of implementing gradient descent, the factor $\frac{2}{m}$ is sometimes omitted or factored into the learning rate. This results in the simplified gradient expression used for updating the weights:

$$\frac{\partial \mathcal{L}}{\partial \hat{Y}^{(i)}} = \hat{Y}^{(i)} - Y^{(i)}$$

5. Backward Propagation

Implement the backward propagation step to compute gradients for updating weights and biases.
Derive the gradients with respect to each parameter.

Mathematical Formulas:

- Gradient of Loss with Respect to Z_2 : $dZ_2 = \frac{\partial \text{Loss}}{\partial Z_2} = Z_2 - y$

- Gradients for W_2 and b_2 :

$$dW_2 = \frac{1}{m} A_1^T dZ_2$$

$$db_2 = \frac{1}{m} \sum_{i=1}^m dZ_2[i]$$

- Gradient for A_1 :

$$dA_1 = dZ_2 W_2^T$$

- Gradient of ReLU Activation: $dZ_1 = dA_1 \times \text{dReLU}(Z_1)$

- Gradients for W_1 and b_1 :

$$dW_1 = \frac{1}{m} X^T dZ_1$$

$$db_1 = \frac{1}{m} \sum_{i=1}^m dZ_1[i]$$

1. Forward Pass:

- Compute the values Z_1 , A_1 , and Z_2 .
- Calculate the MSE loss.

2. Backward Pass:

- Gradient with respect to Z_2 : $\frac{\partial \text{Loss}}{\partial Z_2} = Z_2 - Y$
- Gradient with respect to W_2 : $\frac{\partial \text{Loss}}{\partial W_2} = (Z_2 - Y)A_1^T$
- Gradient with respect to b_2 : $\frac{\partial \text{Loss}}{\partial b_2} = \sum(Z_2 - Y)$
- Gradient with respect to A_1 : $\frac{\partial \text{Loss}}{\partial A_1} = W_2^T(Z_2 - Y)$
- Gradient with respect to Z_1 : $\frac{\partial \text{Loss}}{\partial Z_1} = (W_2^T(Z_2 - Y)) \odot \text{ReLU}'(Z_1)$
- Gradient with respect to W_1 : $\frac{\partial \text{Loss}}{\partial W_1} = ((W_2^T(Z_2 - Y)) \odot \text{ReLU}'(Z_1))X^T$
- Gradient with respect to b_1 : $\frac{\partial \text{Loss}}{\partial b_1} = ((W_2^T(Z_2 - Y)) \odot \text{ReLU}'(Z_1)) \cdot \mathbf{1}$

Step 1: Compute dZ2 (Output layer error)

python

$$dZ2 = A2 - y$$

- This computes the error in the output layer (i.e., the difference between the predicted output A2 and the true label y).

Step 2: Compute gradients for W2 and b2

•**dW2:** The gradient of the cost function with respect to the weights of the output layer.

This computes how much the weights from the hidden layer to the output layer should be adjusted, based on the hidden layer activations A1 and the error dZ2.

db2: The gradient of the cost function with respect to the bias of the output layer.

In neural networks, biases are scalar values added to the neurons of a layer. During backpropagation, we calculate the gradient of the loss with respect to the biases to update them. Here, db2 is the gradient of the bias in the second layer (the output layer in this case).

This computes how much the bias term for the output layer should be adjusted based on the error dZ2.

Step 3: Compute dA1 (Error propagated to the hidden layer)

- This propagates the error back from the output layer to the hidden layer, using the weight matrix W2. This tells us how much each hidden unit contributed to the output error.

Step 4: Compute dZ1 (Hidden layer error)

- This calculates the error in the hidden layer. Since we use a ReLU activation function in the hidden layer, we apply the **derivative of the ReLU function** to the propagated error $dA1$. This ensures that the error is only propagated through neurons that are active (i.e., where $Z1 > 0$).

Step 5: Compute gradients for W1 and b1

- **dW1:** The gradient of the cost function with respect to the weights of the hidden layer.

This computes how much the weights from the input layer to the hidden layer should be adjusted, based on the input X and the hidden layer error $dZ1$.

- **db1:** The gradient of the cost function with respect to the bias of the hidden layer.

This computes how much the bias term for the hidden layer should be adjusted, based on the error $dZ1$.

1. Gradient of Loss with Respect to Z_2 :

$$dZ_2 = A_2 - y = Z_2 - y$$

Since $A_2 = Z_2$, this remains as the original difference between the output and the actual labels.

2. Gradients for W_2 and b_2 :

$$dW_2 = \frac{1}{m} A_1^T dZ_2$$

$$db_2 = \frac{1}{m} \sum_{i=1}^m dZ_2[i]$$

These calculate the gradients for the weights and biases in the output layer, based on the error at the output.

3. Gradient for A_1 :

$$dA_1 = dZ_2 W_2^T$$

This propagates the error back to the hidden layer.

4. Gradient of ReLU Activation:

$$dZ_1 = dA_1 \times dReLU(Z_1)$$

This applies the chain rule to propagate the error through the ReLU activation of the hidden layer.

5. Gradients for W_1 and b_1 :

$$dW_1 = \frac{1}{m} X^T dZ_1$$

$$db_1 = \frac{1}{m} \sum_{i=1}^m dZ_1[i]$$

These update the weights and biases in the hidden layer based on the backpropagated error.

6. Parameter Update

Update the weights and biases of the neural network using the gradients computed during backpropagation. Apply gradient descent to update the parameters.

Mathematical Formulas:

- **Weight Update:** $W = W - \alpha \cdot dW$
- **Bias Update:** $b = b - \alpha \cdot db$

Where α is the learning rate.

- **Gradient Descent:** Each weight and bias is updated by subtracting the gradient (which indicates the direction and magnitude of change) scaled by the learning rate.
 - $W = W - \alpha \cdot dW$
 - $b = b - \alpha \cdot db$
 - Where α is the learning rate and dW, db are the computed gradients from backpropagation.
- **Learning Rate:** Controls how big the steps in updating the parameters are. A smaller learning rate means slower but more precise updates, while a larger rate means faster but possibly more erratic updates.
- **Return Updated Parameters:** The updated W_1, b_1, W_2 , and b_2 are returned to be used in the next iteration.

1. No Learning (No Optimization):

- If you skip updating the parameters (weights W and biases b) with their gradients, the parameters remain the same throughout the training process.
- As a result, the network doesn't adjust itself to minimize the cost function. The output will remain almost the same, and the model will fail to approximate or classify the data correctly.

Why Updating Parameters is Crucial:

In machine learning, especially in neural networks:

- **Weights and biases** represent the learned knowledge of the model.
- **Backpropagation** computes how much each parameter should change based on the current prediction error.
- **Gradient descent** (or other optimization techniques) uses this information to update the parameters in a way that minimizes the error.

Without updating parameters, the model would never "learn" anything from the data and remain static, which defeats the purpose of training a model.

8. Testing the Model

Evaluate the performance of the trained neural network on the test set. Calculate and display the test loss to assess how well the model generalizes to new data.

Mathematical Formula:

- **Test Loss Calculation:** $\text{Test Loss} = \frac{1}{m_{\text{test}}} \sum_{i=1}^{m_{\text{test}}} (y_i - \hat{y}_i)^2$



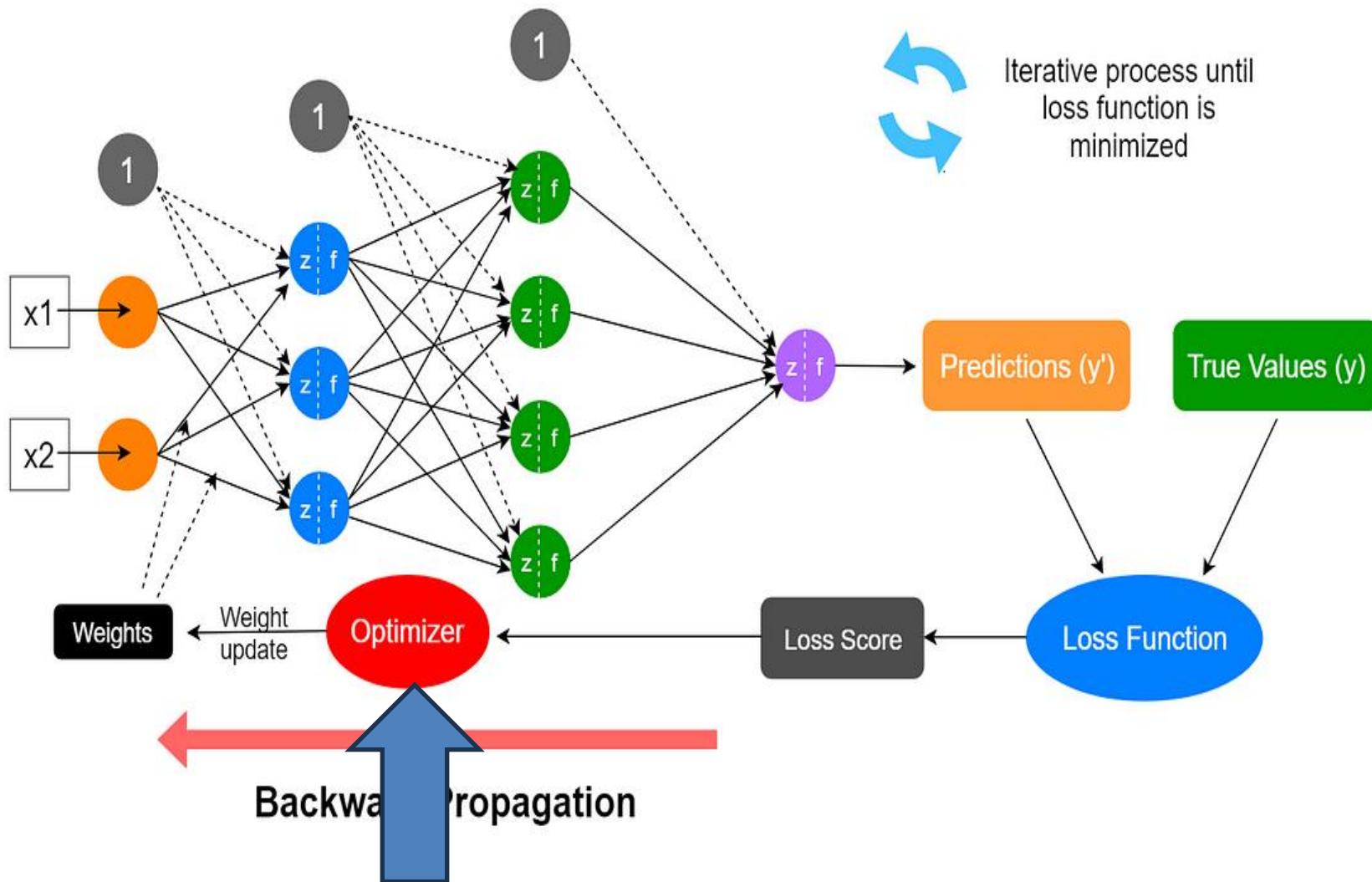
KESHAV MEMORIAL INSTITUTE OF TECHNOLOGY
AN AUTONOMOUS INSTITUTION - ACCREDITED BY NAAC WITH 'A' GRADE
Narayanauguda, Hyderabad.

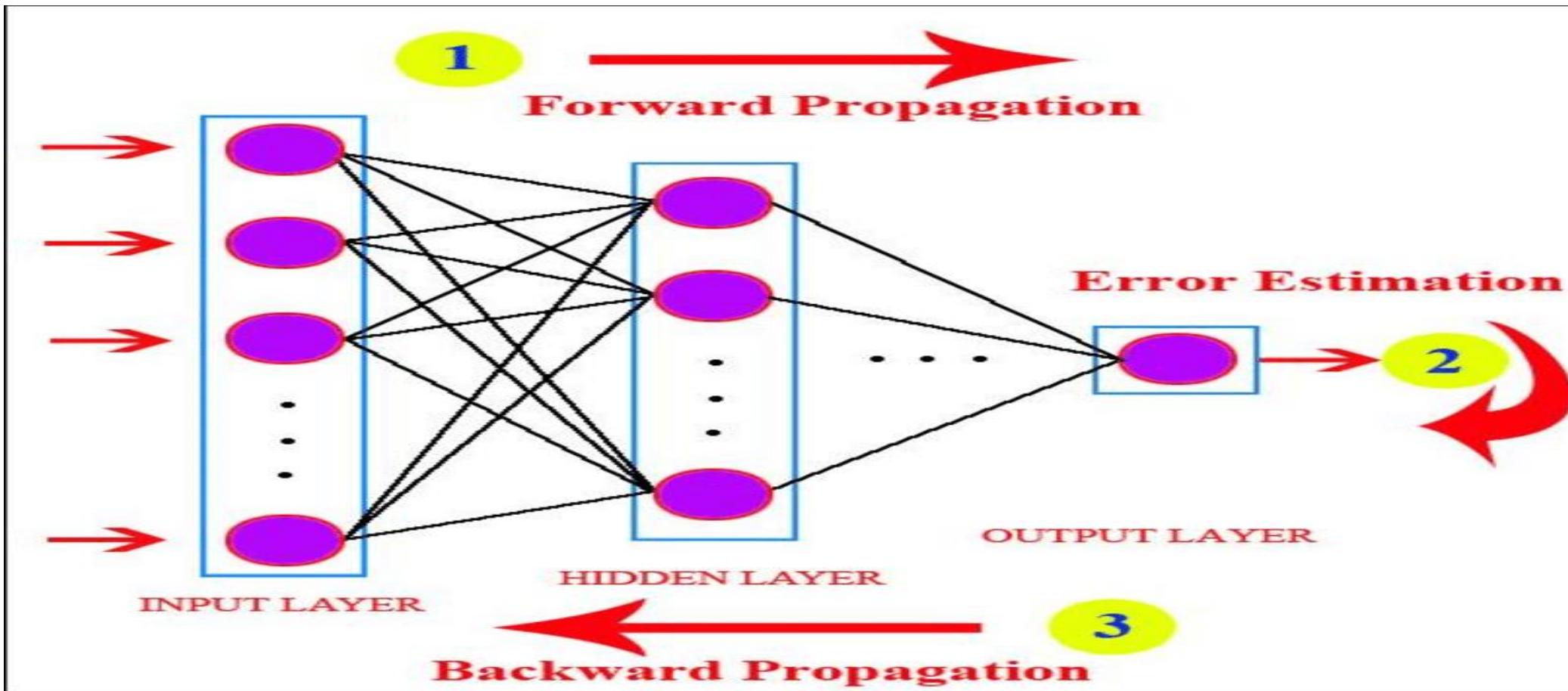
Deep Learning Optimizers

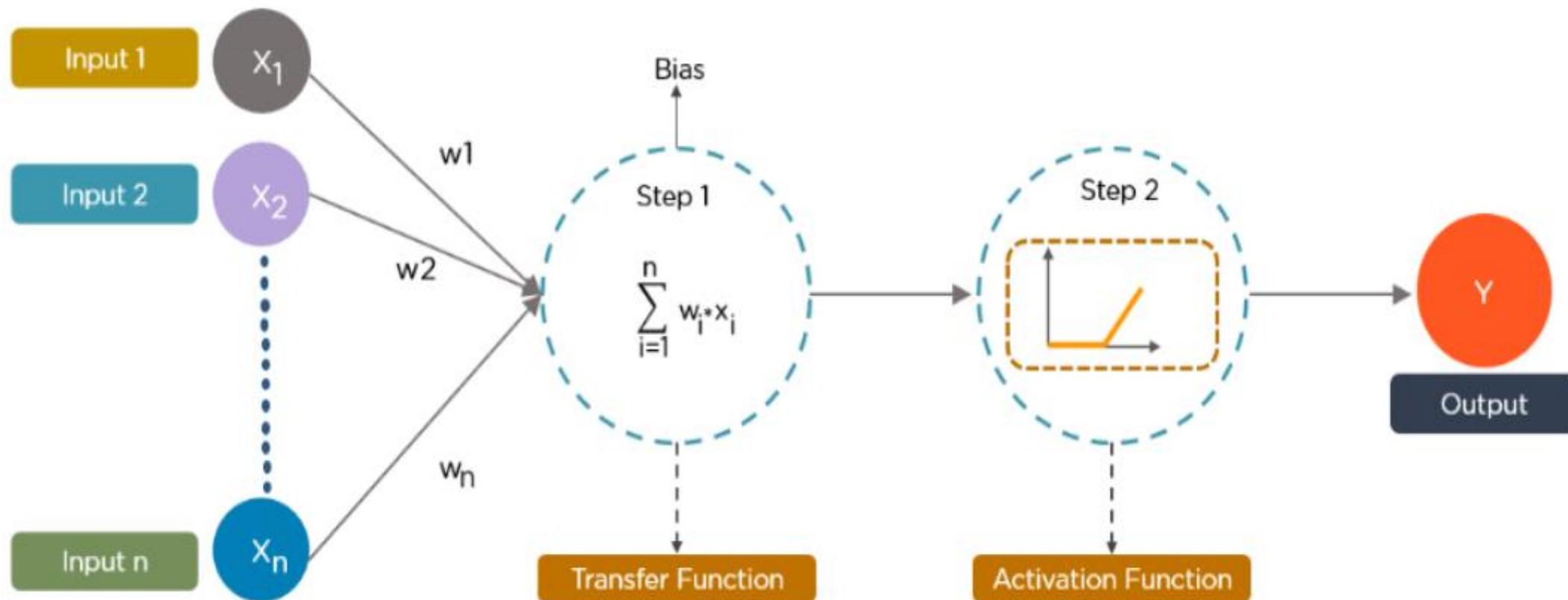
28-09-2024

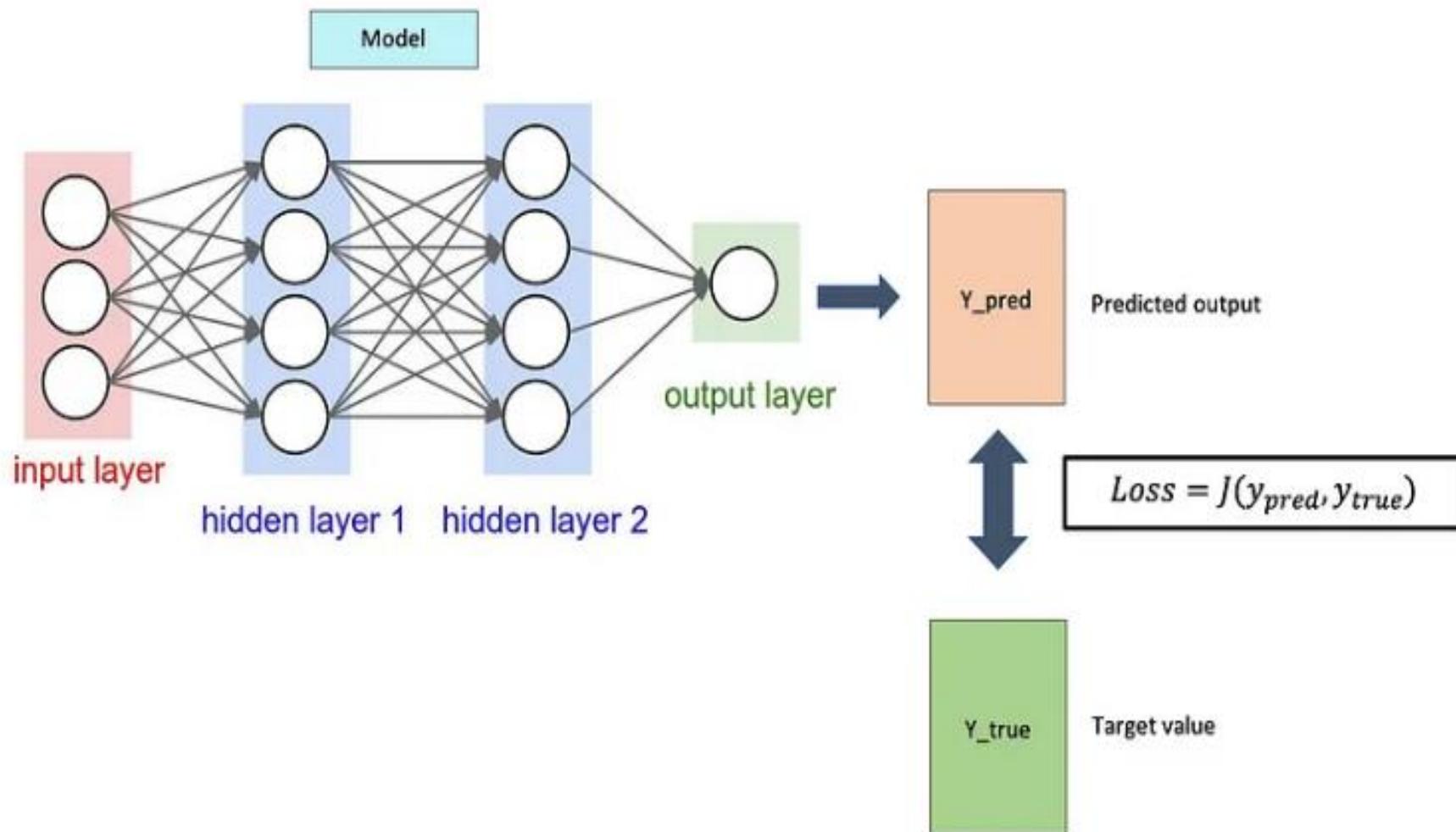
**BY
ASHA**

Forward Propagation



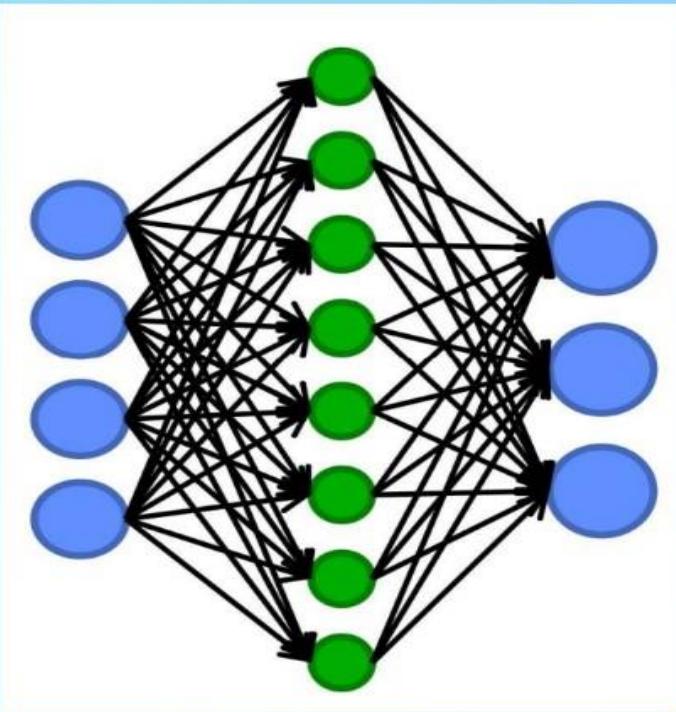






Backpropagation

Learning algorithms use backpropagation to:



Compute a gradient descent with respect to weights.

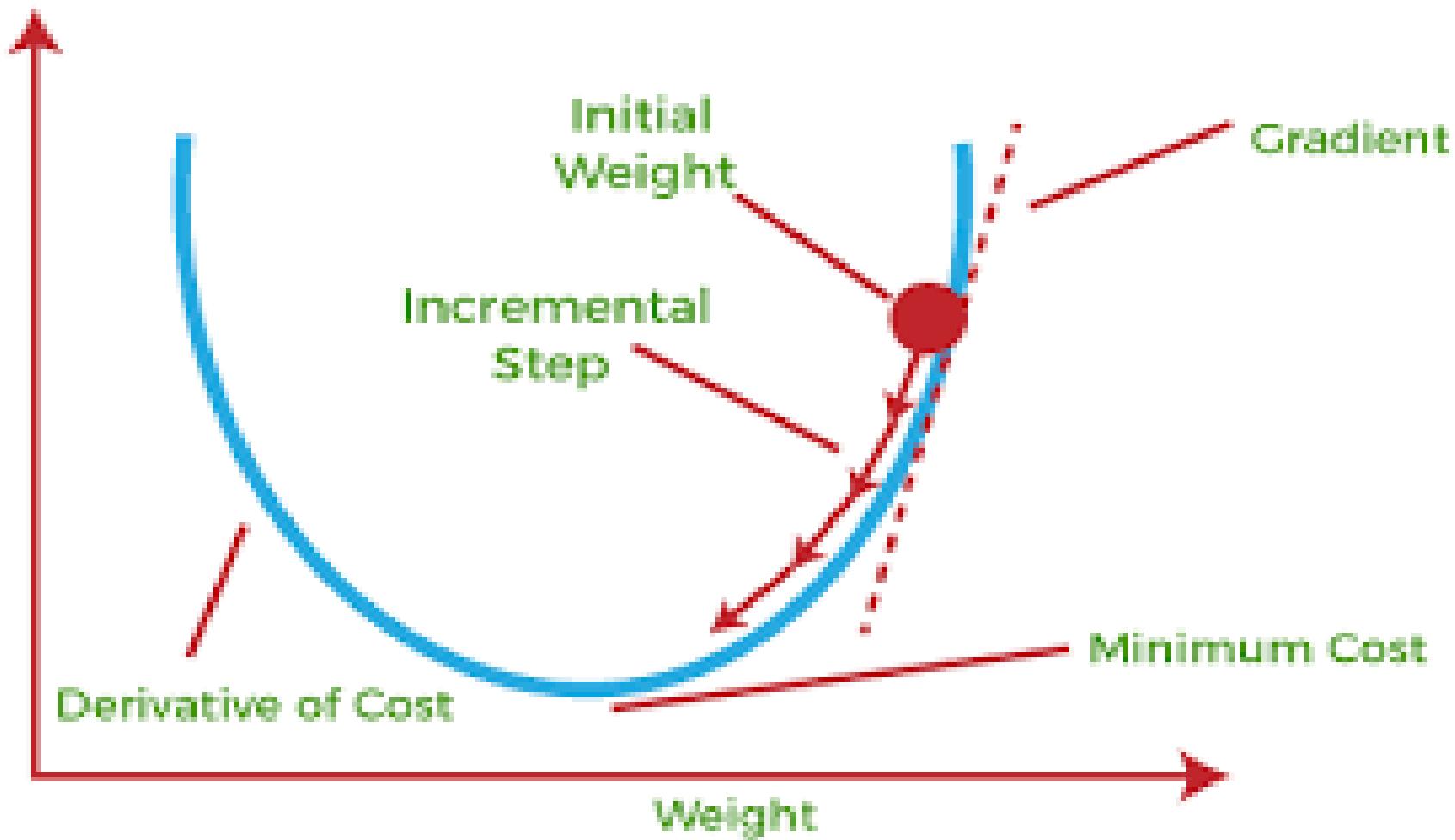


Comparing outputs to desired system outputs.

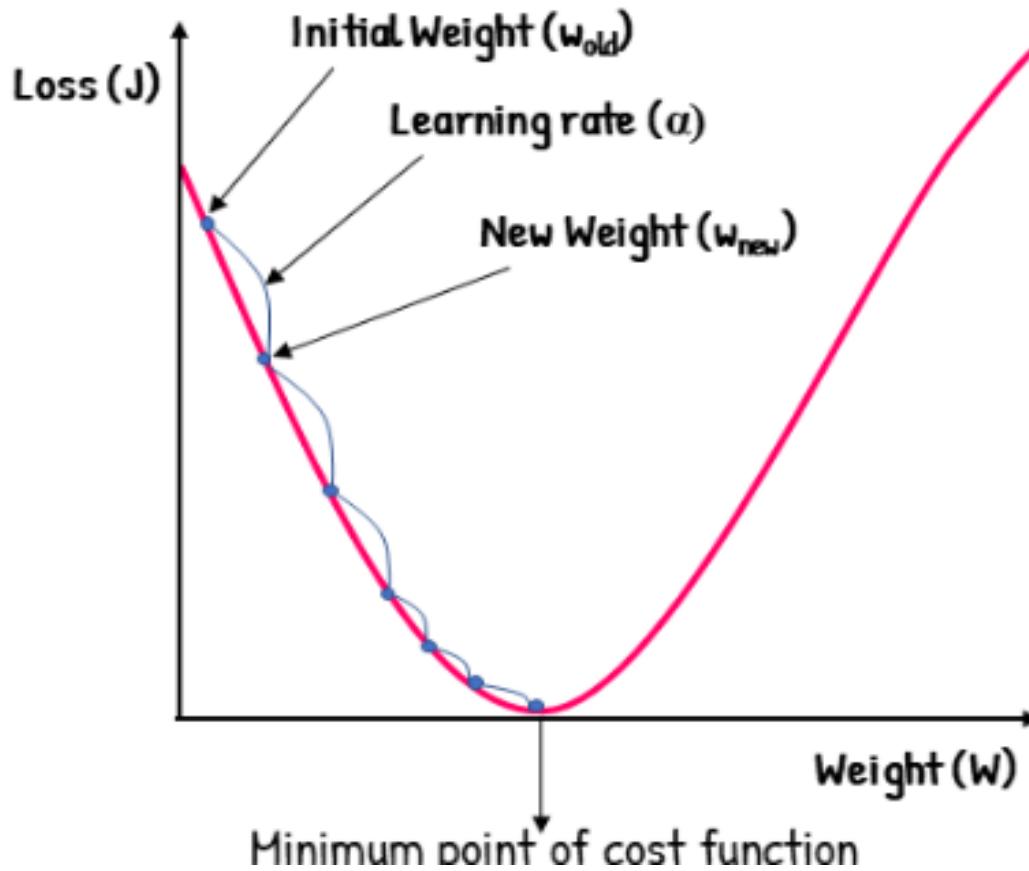


Adjust connection weights to narrow the difference between the two.

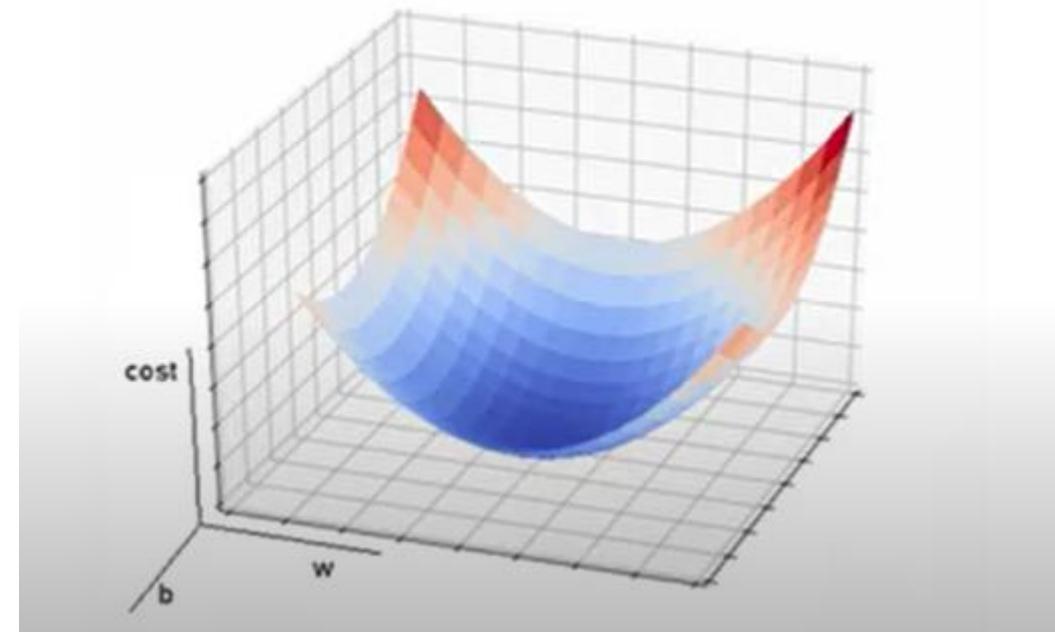
- **Gradient Descent** is the most common optimization algorithm in *machine learning* and *deep learning*.
- It is a first-order optimization algorithm. This means it only takes into account the first derivative when performing the updates on the parameters.
- On each iteration, we update the parameters in the opposite direction of the gradient of the cost function w.r.t the parameters where the gradient gives the direction of the steepest ascent.
- The size of the step we take on each iteration to reach the local minimum is determined by the learning rate α .
- Therefore, we follow the direction of the slope downhill until we reach a local minimum.

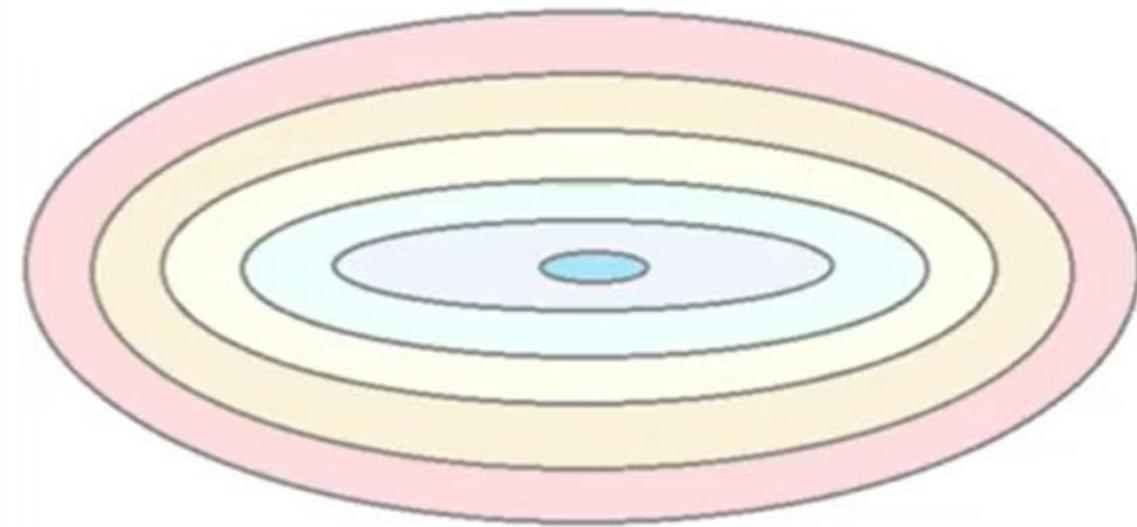
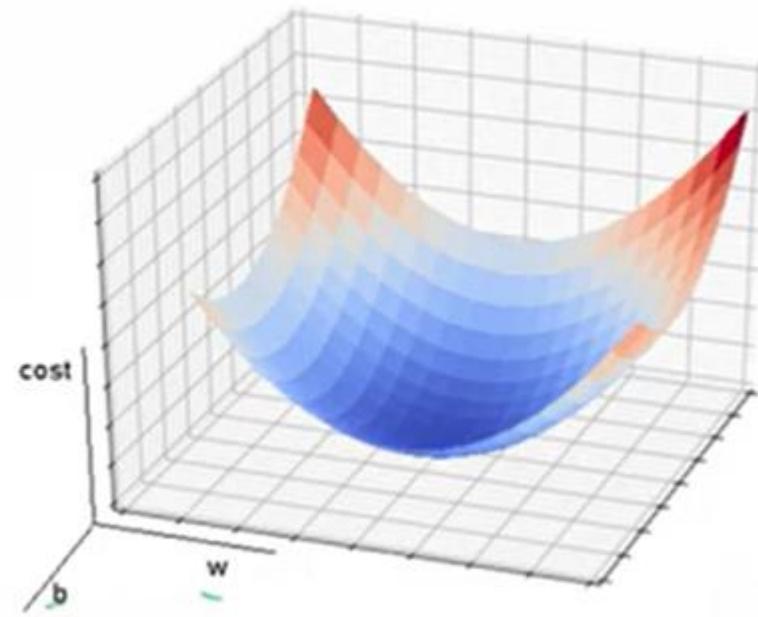


Gradient Descent



$$w_{\text{new}} = w_{\text{old}} - \alpha \frac{\delta J}{\delta w}$$



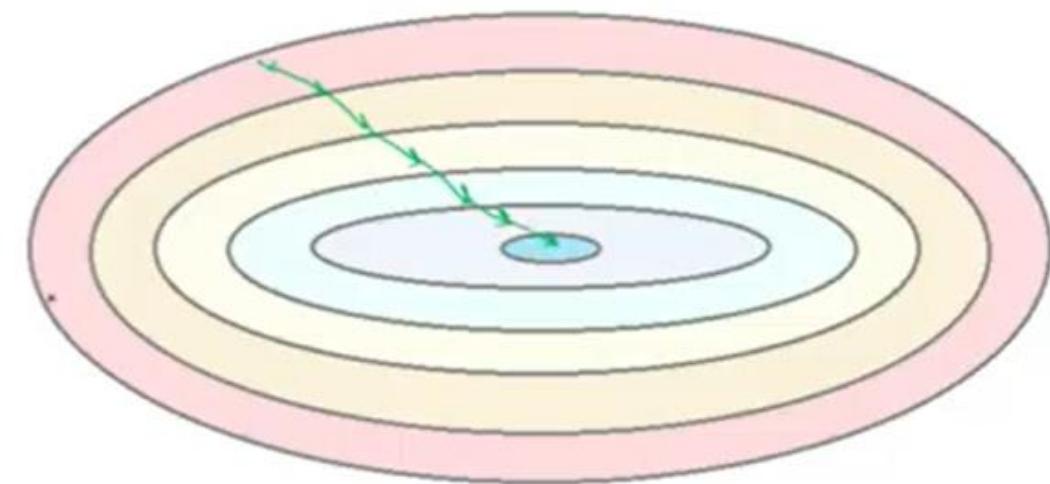
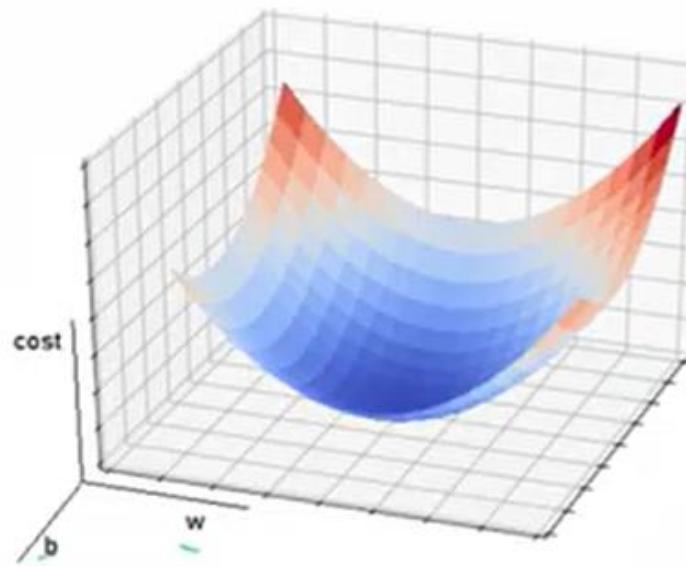


Types of GRADIENT DESCENT

- Batch Gradient Descent
- Stochastic Gradient Descent
- Mini-Batch Gradient Descent

Batch Gradient Descent

- Batch Gradient Descent is a type of Gradient Descent which calculates the error for each example within the training dataset, but only after all training examples have been evaluated does the model get updated. This can be computationally expensive and hence can be slow on very large datasets.



The main advantage:

- It has straight trajectory towards the minimum and it is guaranteed to converge in theory to the global minimum.

The main disadvantages:

- Even though we can use vectorized implementation, it may still be slow to go over all examples especially when we have large datasets.
- Each step of learning happens after going over all examples where some examples may be redundant and don't contribute much to the update.

Stochastic Gradient Descent

- Stochastic Gradient Descent (SGD) is a type of Gradient Descent where the step size is typically much larger, leading to a lot more randomness in the descent down the hill. This randomness can help the algorithm jump out of local minima, finding the global minimum. SGD performs a parameter update for each training example, which is less computationally expensive than Batch Gradient Descent.

Mini-batch Gradient Descent

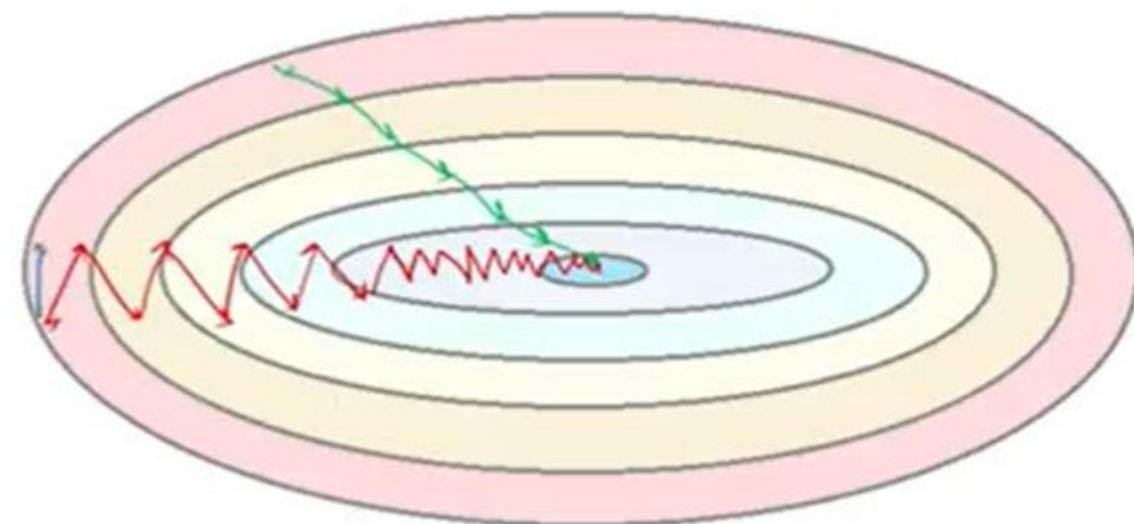
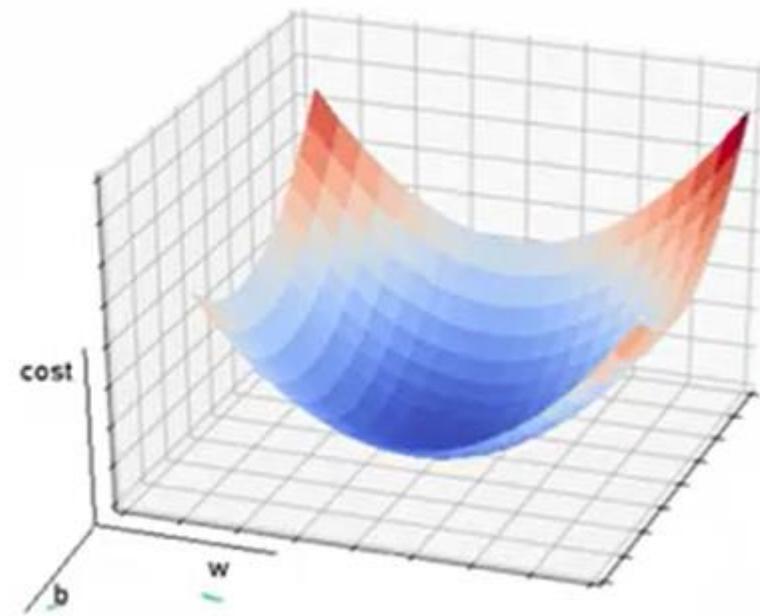
- Mini Batch Gradient Descent is a variation of the gradient descent algorithm that splits the training dataset into small batches that are used to calculate model error and update model coefficients. It combines the advantages of Batch Gradient Descent and Stochastic Gradient Descent by performing an update for every batch of n training examples.

The main advantages:

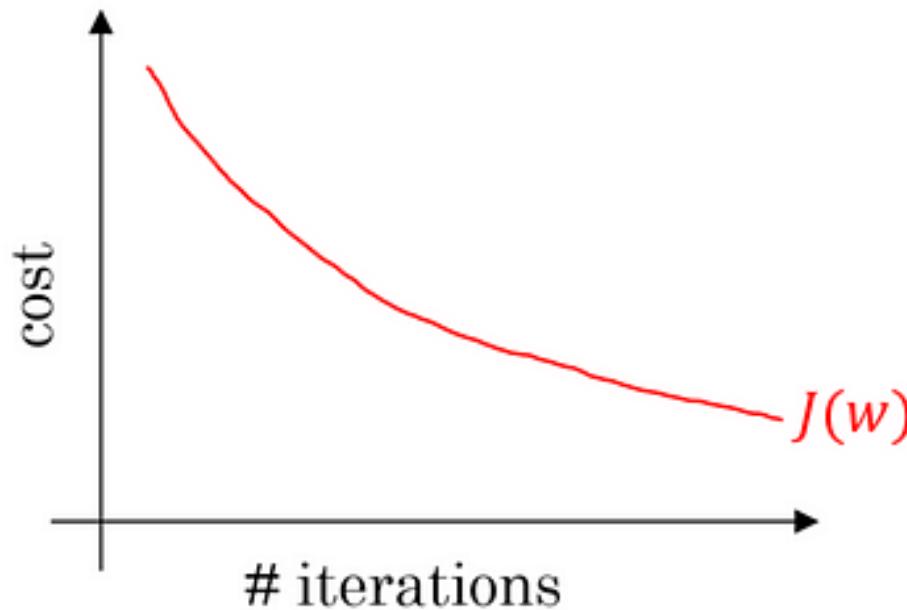
- Faster than Batch version because it goes through a lot less examples than Batch (all examples).

The main disadvantages:

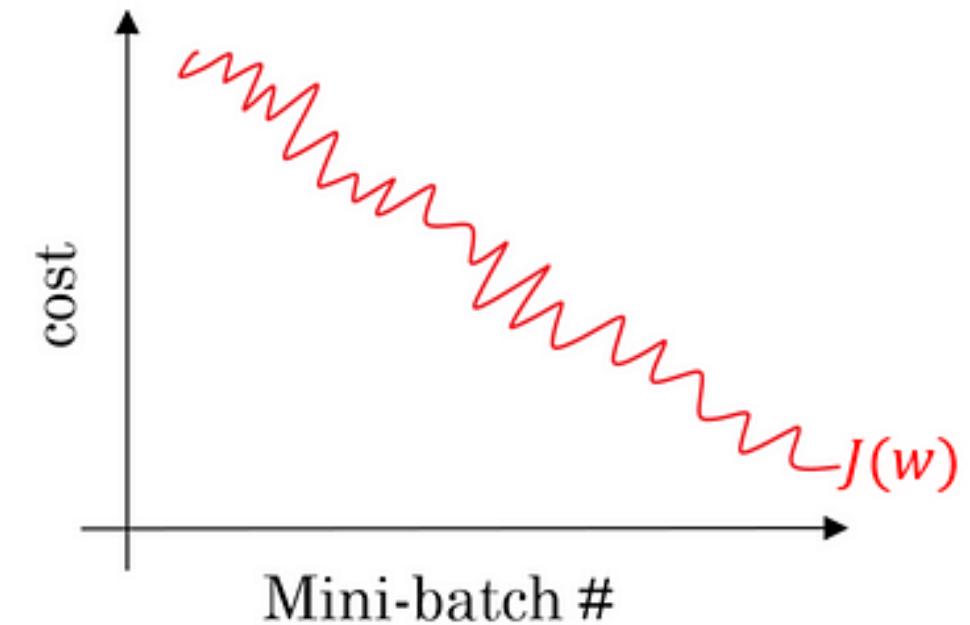
- It won't converge. On each iteration, the learning step may go back and forth due to the noise. Therefore, it wanders around the minimum region but never converges.
- Due to the noise, the learning steps have more oscillations (see figure 4) and requires adding learning-decay to decrease the learning rate as we become closer to the minimum.



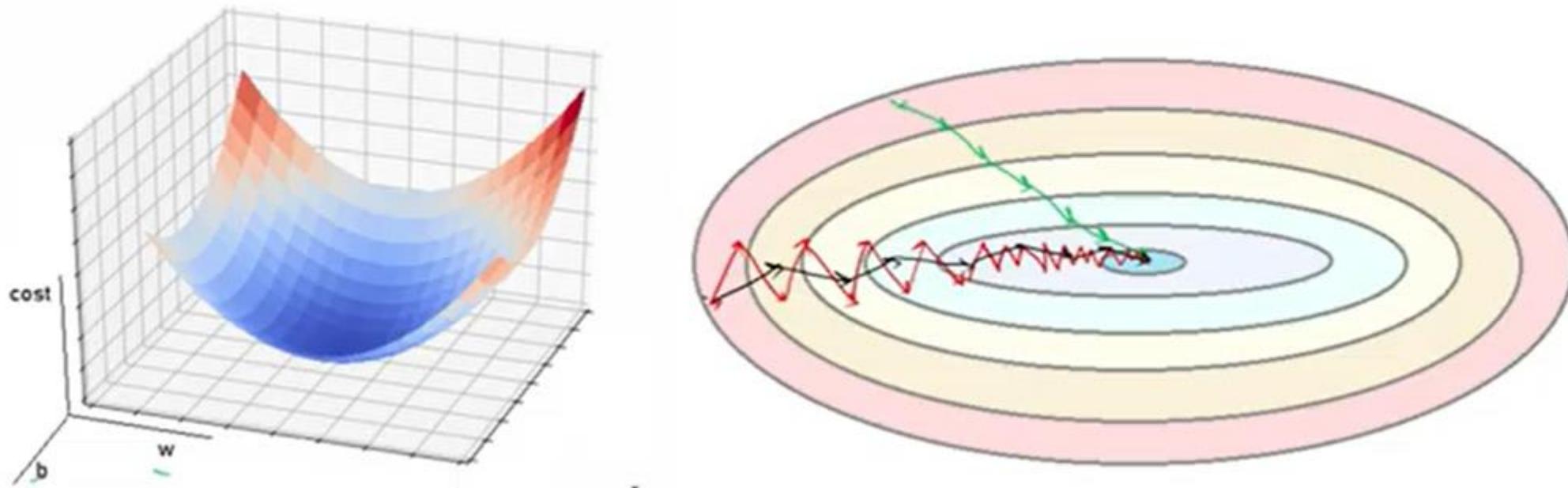
Batch gradient descent

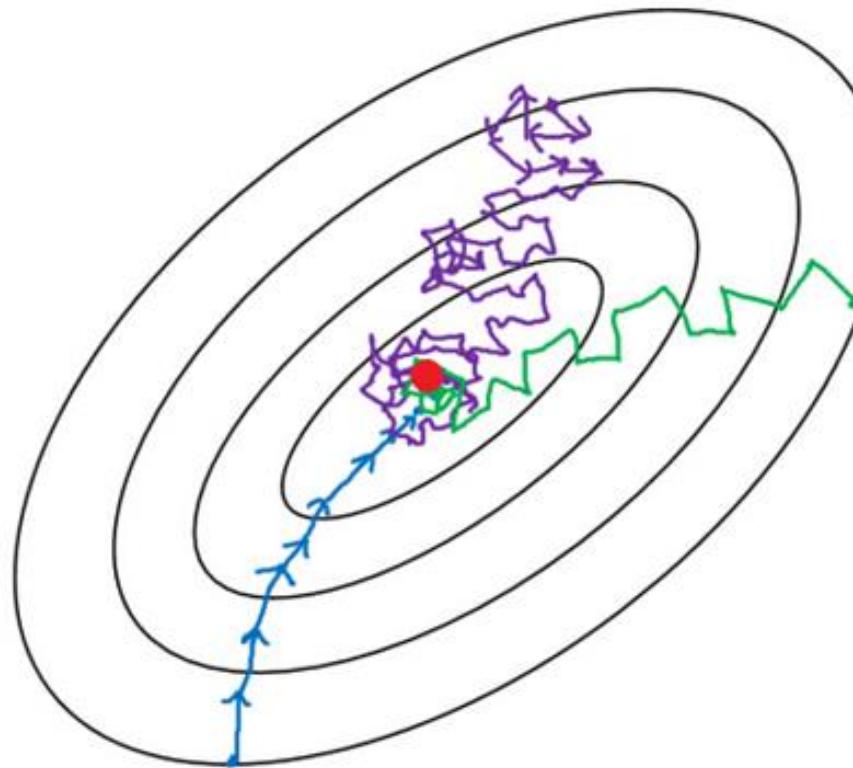


Mini-batch gradient descent



Optimization : is a technique which speed up the training/learning of a model in Deep Learning, while implementing mini-batch gradient descent or stochastic gradient descent

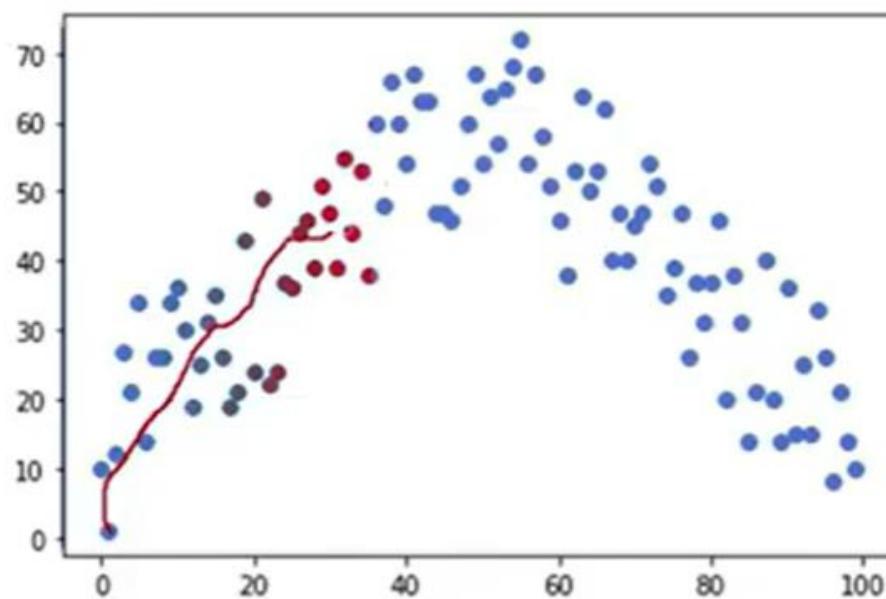




- Batch gradient descent
- Mini-batch gradient Descent
- Stochastic gradient descent

- The choice of Gradient Descent type influences the speed and quality of the optimization of a Neural Network.
- Batch Gradient Descent, while computationally expensive, provides a stable and steady descent towards the minimum.
- Stochastic Gradient Descent is faster and has the ability to jump out of local minima, but it also has a higher variance in the optimization path.
- Mini Batch Gradient Descent offers a balance between the two, providing a blend of stability and speed.

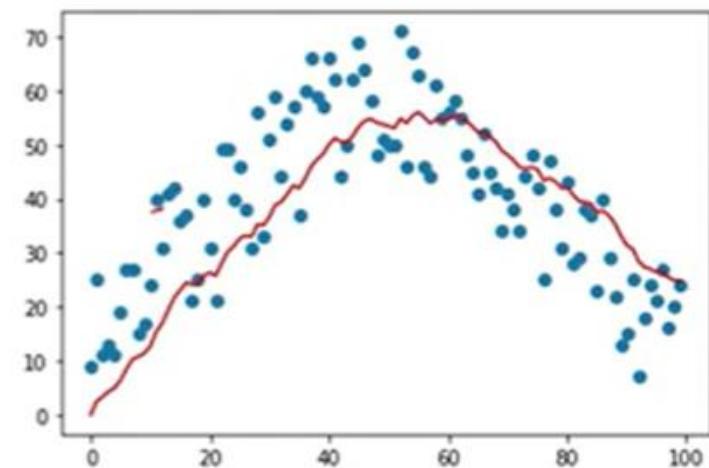
What is Exponentially Weighted Moving Average?



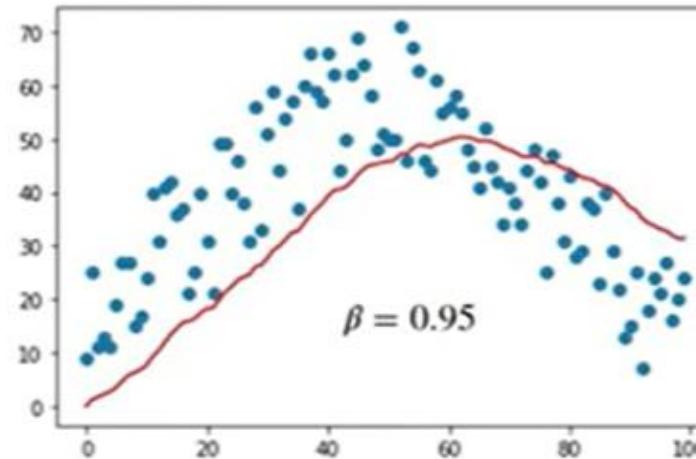
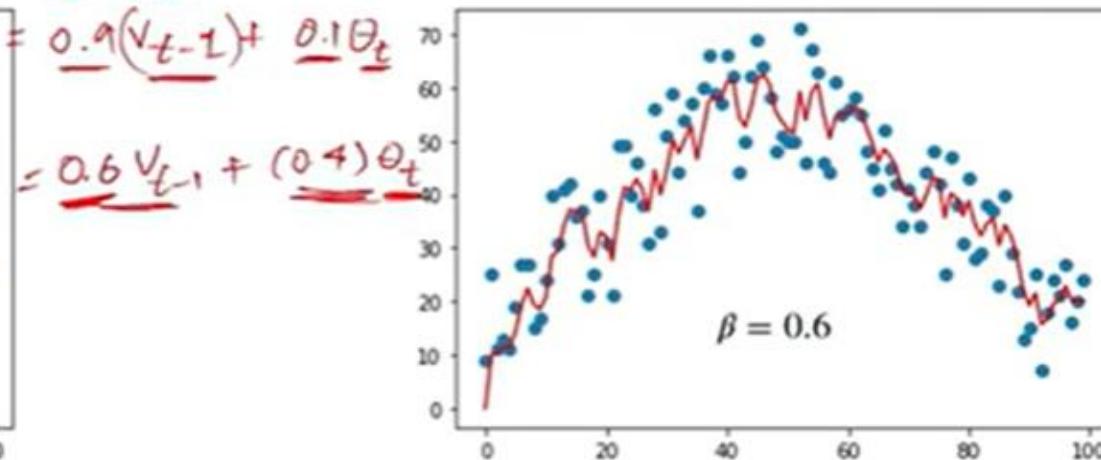
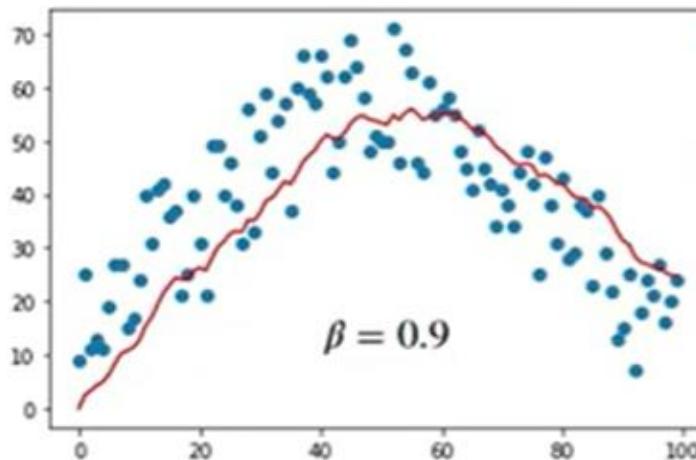
$$V_t = \beta * V_{t-1} + (1 - \beta) * \theta_t$$

$$\underline{V_t} = \beta^{\checkmark} \underline{V_{t-1}} + (1 - \beta) * \underline{\theta_t}$$

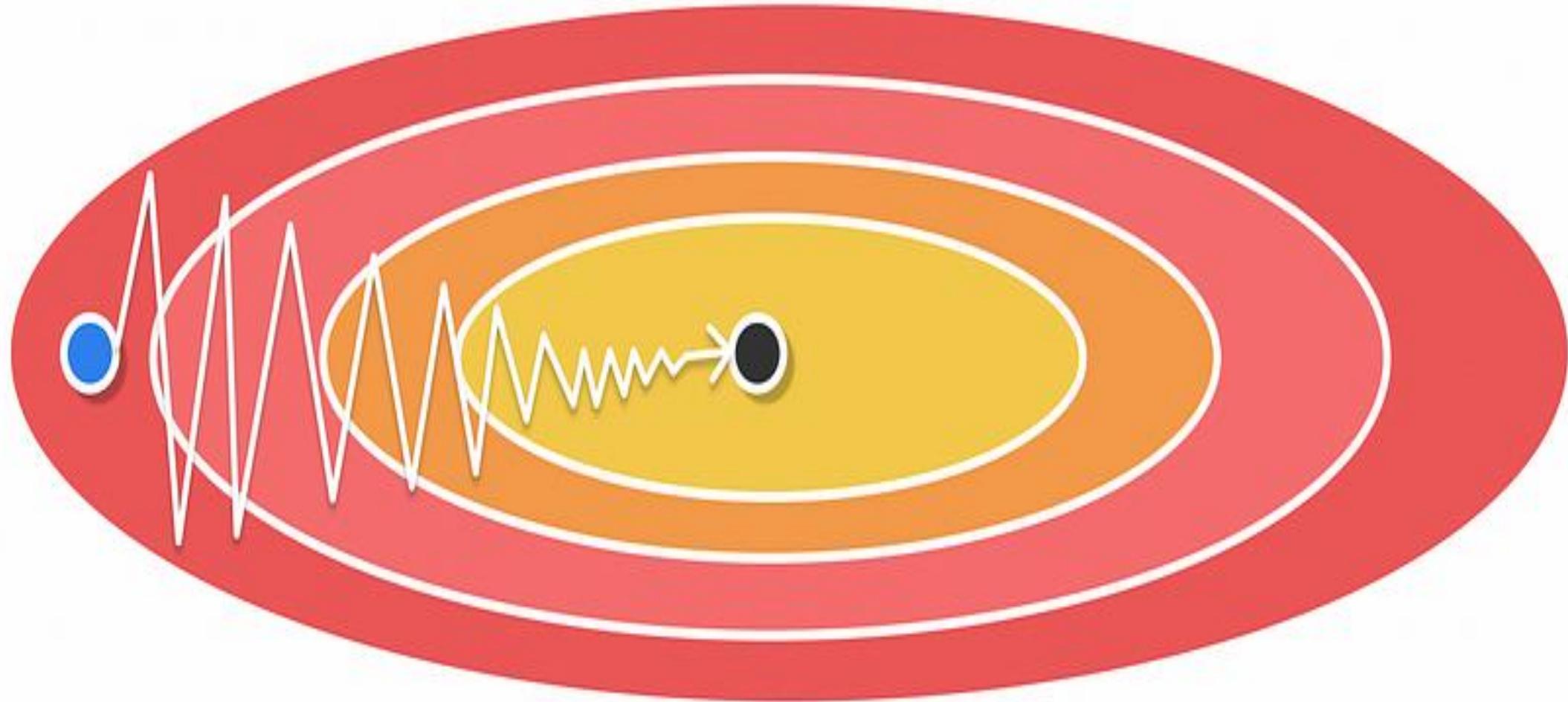
$$\begin{aligned}V_0 &= \underline{\theta} \\V_1 &= \beta * \underline{V_0} + (1 - \beta) * \underline{\theta_1} \\V_2 &= \beta * \underline{V_1} + (1 - \beta) * \underline{\theta_2} \\V_3 &= \beta * \underline{V_2} + (1 - \beta) * \underline{\theta_3} \\V_4 &= \beta * \underline{V_3} + (1 - \beta) * \underline{\theta_4} \\V_5 &= \beta * \underline{V_4} + (1 - \beta) * \underline{\theta_5} \\V_6 &= \beta * \underline{V_5} + (1 - \beta) * \underline{\theta_6} \\V_7 &= \beta * \underline{V_6} + (1 - \beta) * \underline{\theta_7} \\V_8 &= \beta * \underline{V_7} + (1 - \beta) * \underline{\theta_8} \\V_9 &= \beta * \underline{V_8} + (1 - \beta) * \underline{\theta_9}\end{aligned}$$



$$V_t = \beta * V_{t-1} + (1 - \beta) * \theta_t$$



$\boxed{\beta = 0.9}$



The starting point is depicted in blue and the local minimum is shown in black.

Momentum

- Based on the example above, it would be desirable to make a loss function performing larger steps in t This way, the convergence Momentum.

Gradient Descent

$$W = W - \alpha * \frac{\partial \text{cost}}{\partial W}$$

$$B = B - \alpha * \frac{\partial \text{cost}}{\partial B}$$

| smaller steps in the vertical.

; effect is exactly achieved by

$$v_t = \beta v_{t-1} + (1 - \beta) dw_t$$

$$w_t = w_{t-1} - \alpha v_t$$

Gradient descent



the gradient computed on the current iteration does not prevent gradient descent from oscillating in the vertical direction

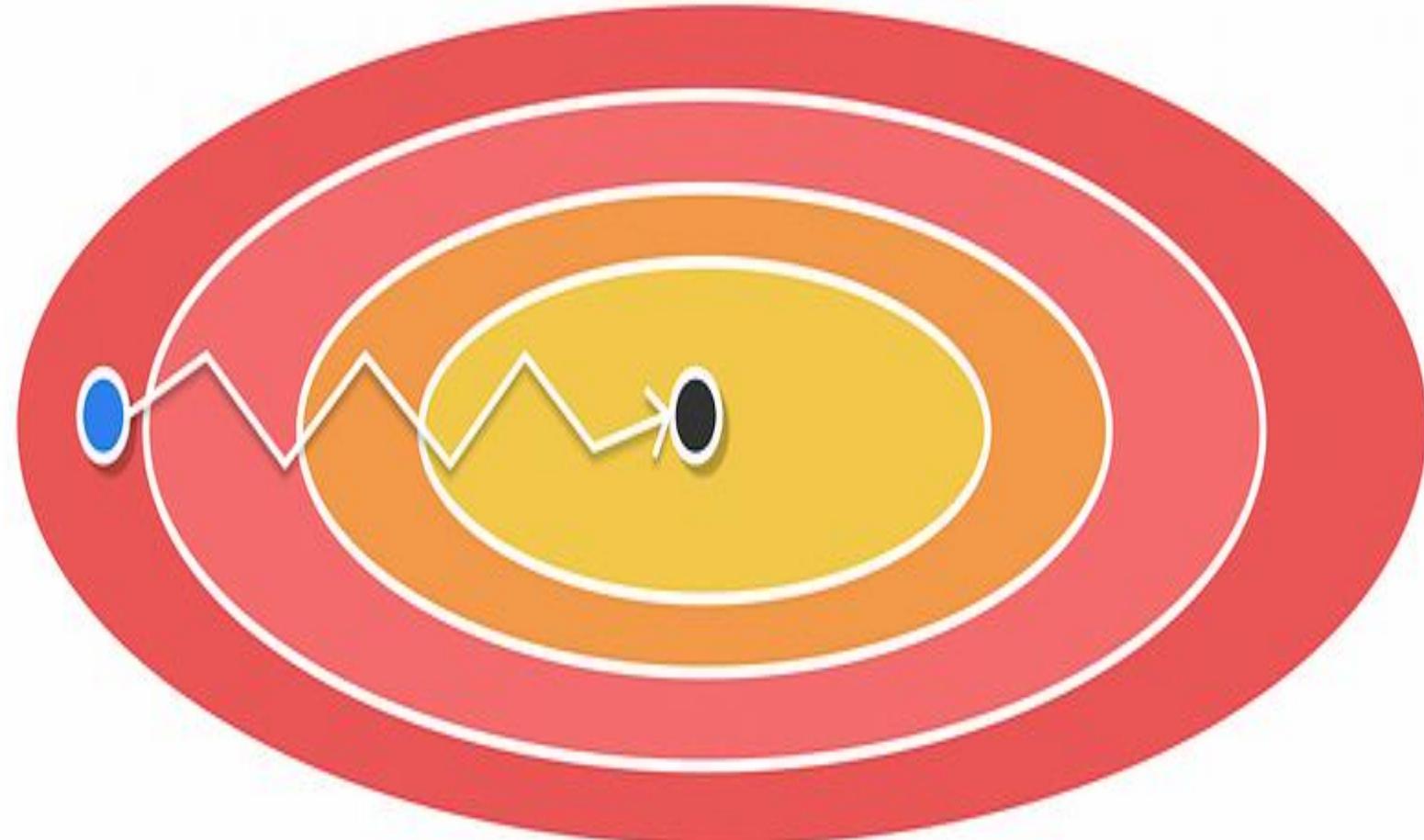
Momentum



the average vector of the horizontal component is aligned towards the minimum

the average vector of the vertical component is close to 0

- Optimization with Momentum



- In practice, Momentum usually converges much faster than gradient descent. With Momentum, there are also fewer risks in using larger learning rates, thus accelerating the training process.
- In Momentum, it is recommended to choose β close to 0.9.

AdaGrad (Adaptive Gradient Algorithm)

- AdaGrad is another optimizer with the motivation to adapt the learning rate to computed gradient values. There might occur situations when during training, one component of the weight vector has very large gradient values while another one has extremely small. **This happens especially in cases when an infrequent model parameter appears to have a low influence on predictions.**

- AdaGrad deals with the aforementioned problem **by independently adapting the learning rate for each weight component.** If gradients corresponding to a certain weight vector component are large, then the respective learning rate will be small. Inversely, for smaller gradients, the learning rate will be bigger. This way, Adagrad deals with vanishing and exploding gradient problems.

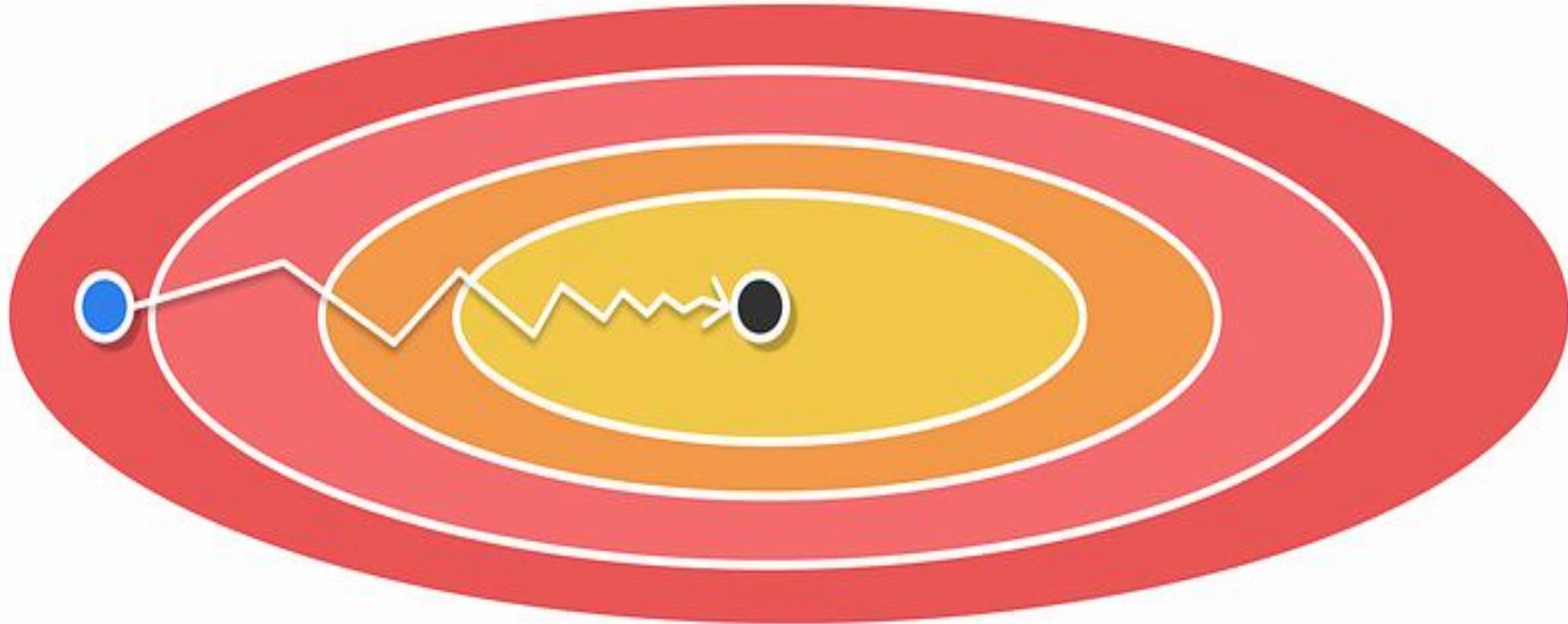
$$v_t = v_{t-1} + dw_t^2$$

$$w_t = w_{t-1} - \frac{\alpha}{\sqrt{v_t + \epsilon}} dw_t$$

- The greatest advantage of AdaGrad is that there is no longer a need to manually adjust the learning rate as it adapts itself during training.
- Nevertheless, there is a negative side of AdaGrad: **the learning rate constantly decays with the increase of iterations** (the learning rate is always divided by a positive cumulative number). Therefore, the algorithm tends to converge slowly during the last iterations where it becomes very low.



Optimization with AdaGrad



RMSProp (Root Mean Square Propagation)

- RMSProp was elaborated as an improvement over AdaGrad which tackles the issue of learning rate decay. Similarly to AdaGrad, RMSProp uses a pair of equations for which the weight update is absolutely the same.

$$v_t = \beta v_{t-1} + (1 - \beta) dw_t^2$$

$$w_t = w_{t-1} - \frac{a}{\sqrt{v_t + \epsilon}} dw_t$$

Adam (Adaptive Moment Estimation)

- For the moment, Adam is the most famous optimization algorithm in deep learning. At a high level, Adam combines Momentum and RMSProp algorithms. To achieve it, it simply keeps track of the exponentially moving averages for computed gradients and squared gradients respectively.

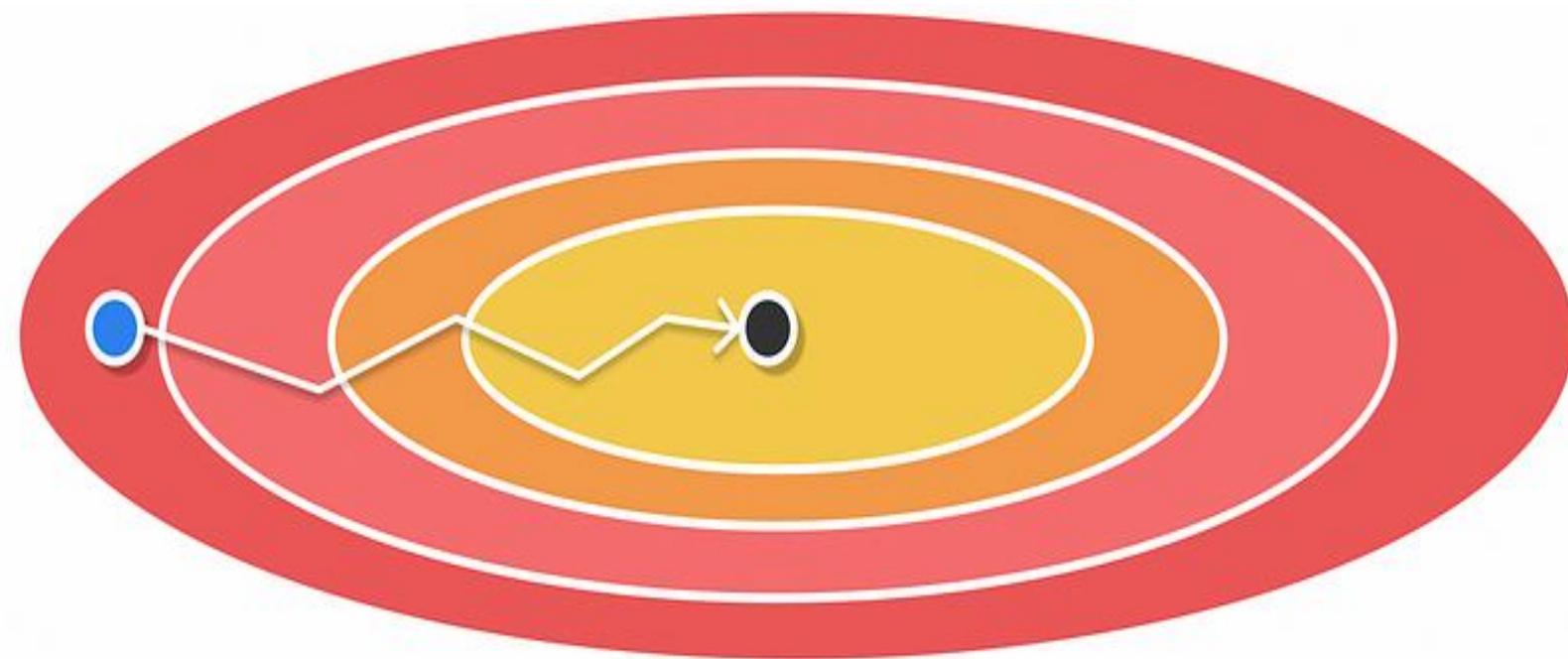
$$v_t = \beta_1 v_{t-1} + (1 - \beta_1) dw_t \longrightarrow \hat{v}_t = \frac{v_t}{1 - \beta_1^t}$$

$$s_t = \beta_2 s_{t-1} + (1 - \beta_2) dw_t^2 \longrightarrow \hat{s}_t = \frac{s_t}{1 - \beta_2^t}$$

$$w_t = w_{t-1} - \frac{\alpha \hat{v}_t}{\sqrt{\hat{s}_t} + \epsilon} dw_t$$

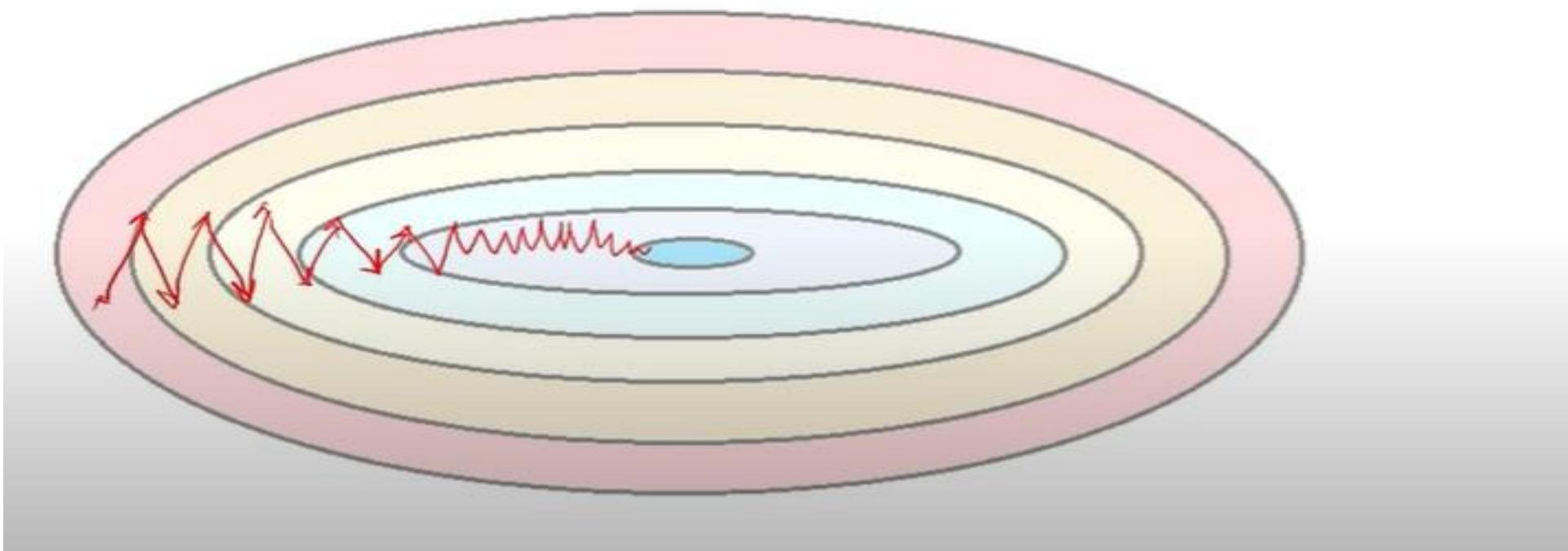
- Furthermore, it is possible to use bias correction for moving averages for a more precise approximation of gradient trend during the first several iterations. The experiments show that Adam adapts well to almost any type of neural network architecture taking the advantages of both Momentum and RMSProp.

Optimization with Adam



Adam

RMSprop
Momentum



Momentum

$$V_{dw} = \beta \cdot V_{dw\text{prev}} + (1-\beta) \cdot dW$$

$$V_{dB} = \beta \cdot V_{dB\text{prev}} + (1-\beta) \cdot dB$$

$$w = w - \alpha \cdot V_{dw}$$

$$\beta = B - \alpha \cdot V_{dB}$$

RMS prop

$$S_{dw} = \beta \cdot S_{dw\text{prev}} + (1-\beta) \cdot (dW)^2$$

$$S_{dB} = \beta \cdot S_{dB\text{prev}} + (1-\beta) \cdot (dB)^2$$

$$w = w - \alpha \cdot (dW / \sqrt{S_{dw} + \epsilon})$$

$$B = B - \alpha \cdot (dB / \sqrt{S_{dB} + \epsilon})$$

Momentum

$$V_{dw} = \beta_1 V_{dw\text{prev}} + (1-\beta_1) dW$$

$$V_{dB} = \beta_1 V_{dB\text{prev}} + (1-\beta_1) dB$$

$$w = w - \alpha \cdot \underline{V_{dw}}$$

$$B = B - \alpha \cdot \underline{V_{dB}}$$

RMS prop

$$S_{dw} = \beta_2 S_{dw\text{prev}} + (1-\beta_2) (dW)^2$$

$$S_{dB} = \beta_2 S_{dB\text{prev}} + (1-\beta_2) (dB)^2$$

$$w = w - \alpha \cdot (dW / \sqrt{S_{dw} + \epsilon})$$

$$B = B - \alpha \cdot (dB / \sqrt{S_{dB} + \epsilon})$$

* Adam (Adam moment estimation)

$$\left\{ \begin{array}{l} w = w - \alpha \cdot \frac{V_{dw}}{\sqrt{S_{dw} + \epsilon}} \\ B = B - \alpha \cdot \frac{V_{dB}}{\sqrt{S_{dB} + \epsilon}} \end{array} \right.$$

$$\beta_1 = 0.9$$

$$\beta_2 = 0.999$$

$$\epsilon = 10^{-8}$$

```
def initialize_adam(n_x, n_h, n_y):  
    # Your code here  
    return vW1, vb1, vW2, vb2, sW1, sb1, sW2, sb2
```

Expected Output:

First Moment Vectors (Velocity Terms):

1. vW1: A NumPy array of shape (n_h, n_x) initialized to zeros, representing the velocity term for the weight matrix connecting the input layer to the hidden layer.
2. vb1: A NumPy array of shape $(n_h, 1)$ initialized to zeros, representing the velocity term for the bias vector of the hidden layer.
3. vW2: A NumPy array of shape (n_y, n_h) initialized to zeros, representing the velocity term for the weight matrix connecting the hidden layer to the output layer.
4. vb2: A NumPy array of shape $(n_y, 1)$ initialized to zeros, representing the velocity term for the bias vector of the output layer.

Second Moment Vectors (Squared Gradient Terms):

1. sW1: A NumPy array of shape (n_h, n_x) initialized to zeros, representing the squared gradient term for the weight matrix connecting the input layer to the hidden layer.
2. sb1: A NumPy array of shape $(n_h, 1)$ initialized to zeros, representing the squared gradient term for the bias vector of the hidden layer.
3. sW2: A NumPy array of shape (n_y, n_h) initialized to zeros, representing the squared gradient term for the weight matrix connecting the hidden layer to the output layer.
4. sb2: A NumPy array of shape $(n_y, 1)$ initialized to zeros, representing the squared gradient term for the bias vector of the output layer.

```
def update_parameters_with_adam(W1, b1, W2, b2, dW1,  
db1, dW2, db2, vW1, vb1, vW2, vb2, sW1, sb1, sW2, sb2,  
t, learning_rate=0.001, beta1=0.9, beta2=0.999, epsilon=1e-  
8):  
    # Your code here  
    return W1, b1, W2, b2, vW1, vb1, vW2, vb2, sW1, sb1,  
    sW2, sb2
```

Parameters:

W1: A NumPy array of shape (n_h, n_x) representing the weights of the layer connecting the input to the hidden layer.

b1: A NumPy array of shape $(n_h, 1)$ representing the biases of the hidden layer.

W2: A NumPy array of shape (n_y, n_h) representing the weights of the layer connecting the hidden layer to the output layer.

b2: A NumPy array of shape $(n_y, 1)$ representing the biases of the output layer.

dW1: A NumPy array of shape (n_h, n_x) representing the gradient of the weights of the layer connecting the input to the hidden layer.

db1: A NumPy array of shape $(n_h, 1)$ representing the gradient of the biases of the hidden layer.

dW2: A NumPy array of shape (n_y, n_h) representing the gradient of the weights of the layer connecting the hidden layer to the output layer.

db2: A NumPy array of shape $(n_y, 1)$ representing the gradient of the biases of the output layer.

vW1: A NumPy array of shape (n_h, n_x) representing the moving average of the gradients for W1.

vb1: A NumPy array of shape ($n_h, 1$) representing the moving average of the gradients for b1.

vW2: A NumPy array of shape (n_y, n_h) representing the moving average of the gradients for W2.

vb2: A NumPy array of shape ($n_y, 1$) representing the moving average of the gradients for b2.

sW1: A NumPy array of shape (n_h, n_x) representing the moving average of the squared gradients for W1.

sb1: A NumPy array of shape ($n_h, 1$) representing the moving average of the squared gradients for b1.

sW2: A NumPy array of shape (n_y, n_h) representing the moving average of the squared gradients for W2.

sb2: A NumPy array of shape ($n_y, 1$) representing the moving average of the squared gradients for b2.

t: An integer representing the current time step (iteration) of the optimization process.

learning_rate: A float representing the learning rate for the optimization (default is 0.001).

beta1: A float representing the exponential decay rate for the first moment estimates (default is 0.9).

beta2: A float representing the exponential decay rate for the second moment estimates (default is 0.999).

epsilon: A small constant to prevent division by zero (default is 1e-8).

Conclusion

- We have looked at different optimization algorithms in neural networks. Considered as a combination of Momentum and RMSProp, Adam is the most superior of them which robustly adapts to large datasets and deep networks. Moreover, it has a straightforward implementation and little memory requirements making it a preferable choice in the majority of situations.



KESHAV MEMORIAL INSTITUTE OF TECHNOLOGY
AN AUTONOMOUS INSTITUTION - ACCREDITED BY NAAC WITH 'A' GRADE
Narayanaguda, Hyderabad.

Deep Learning

28-10-2024

BY
ASHA

Building ANN from Scratch

Neural Networks, Activation Functions, Loss Functions, and Optimizers

How Do We Build an Artificial Neural Network (ANN)?

Step 1: Initialize the Network Architecture

A basic ANN consists of three types of layers:

Input Layer: Receives the input features (e.g., pixel values of an image, words in a text, etc.).

Hidden Layers: Consist of neurons that apply a transformation to the input data using weights and biases.

Output Layer: Produces the final prediction, typically a classification or regression output.

Each connection between neurons has a **weight**, and each neuron has a **bias**.

Step 2: Apply an Activation Function

Each neuron takes a weighted sum of its inputs, applies a bias, and passes the result through an **activation function**. The role of activation functions is to introduce **non-linearity**, allowing the network to model complex data patterns.

Step 3: Forward Propagation – Generating Predictions

Once the architecture is set, the data is fed forward through the network:

Each neuron in the hidden layers calculates the weighted sum of its inputs, adds the bias, and applies the activation function.

The final layer outputs a prediction, often using activation functions like **Softmax** for multi-class classification or **Sigmoid** for binary classification.

Step 4: Define a Loss Function

The **loss function** quantifies how far off the network's predictions are from the actual values. Common loss functions include:

Mean Squared Error (MSE) for regression tasks.

Cross-Entropy Loss for classification tasks. The goal is to minimize this loss by adjusting the network's weights and biases.

Step 5: Backpropagation – Learning from Errors

After generating predictions, the network needs to learn from its mistakes.

Backpropagation is the process of calculating the **gradient** of the loss function with respect to each weight in the network, using the chain rule.

This gradient indicates how each weight should be adjusted to reduce the overall loss.

Step 6: Use an Optimizer to Update Weights

The **optimizer** is responsible for adjusting the weights based on the gradients from backpropagation. Popular optimizers include:

Stochastic Gradient Descent (SGD): Updates weights using small, random batches of data.

Adam: Combines the advantages of SGD and other adaptive algorithms, allowing for faster and more efficient convergence.

Optimizers are key to ensuring that the model converges to an optimal solution efficiently.

Step 7: Train the Network

The training process consists of feeding the model with batches of data (called **epochs**), performing forward and backward passes, and updating weights to reduce the loss function. After several iterations, the model learns to make accurate predictions.

How the Core Components Affect Performance

- **Activation Functions:** The right activation function helps capture non-linearities in data and ensures that gradients flow properly through the network during backpropagation.
- **Loss Functions:** Selecting an appropriate loss function helps the model understand how to measure errors. For classification problems, **Cross-Entropy Loss** is effective, while for regression tasks, **Mean Squared Error** is commonly used.
- **Optimizers:** Optimizers like **Adam** adjust weights efficiently, improving convergence speed and ensuring the model finds a good local or global minimum in the loss function landscape. The choice of optimizer influences how quickly and how well the model learns.

1. Activation Functions: Shaping the Output

Activation functions introduce non-linearity into the model, allowing it to learn complex patterns. Here are some commonly used ones:

- **Sigmoid:** Maps inputs between 0 and 1, used often in binary classification tasks. However, it suffers from **vanishing gradients** for large inputs.
- **Tanh:** Outputs values between -1 and 1, making it zero-centered. It's an improvement over sigmoid but still faces the vanishing gradient issue for deep networks.
- **ReLU (Rectified Linear Unit):** Outputs the input directly if it's positive; otherwise, it returns 0. It solves the vanishing gradient problem but may suffer from the **dying ReLU** issue (neurons stop updating for negative inputs).
- **Softmax:** Converts outputs into probabilities, commonly used in the final layer for multi-class classification problems.

2. Loss Functions: Quantifying Error

- Loss functions measure how far the predicted outputs are from the actual values, guiding the learning process:
- **Mean Squared Error (MSE):** Used in regression tasks, it penalizes larger errors more heavily due to squaring.
- **Binary Cross-Entropy:** Common in binary classification, this loss function measures the performance of the model when predicting between two classes.
- **Categorical Cross-Entropy:** Extends binary cross-entropy for multi-class classification tasks.

3. Optimizers: Fine-Tuning the Learning Process

Optimizers adjust the model's weights based on the gradient of the loss function. They control how the model learns during training:

- **Gradient Descent:** The basic approach where weights are updated in the direction that minimizes the loss. Variants include:
 - **Batch Gradient Descent:** Uses the entire dataset for each update. Slow for large datasets.
 - **Stochastic Gradient Descent (SGD):** Updates weights for each training sample, making it faster but noisier.
 - **Mini-Batch Gradient Descent:** A balance between batch and stochastic gradient descent, where updates are made on mini-batches.
- **Adam (Adaptive Moment Estimation):** Combines the benefits of both RMSprop and momentum, adjusting the learning rate based on first and second moments of the gradient. It's one of the most widely used optimizers for deep learning.

Exercise 1: Regression on Insurance Dataset (Basic ANN with Gradient Descent)

Objective:

We focused on building a simple neural network for regression tasks. Using an insurance dataset, the task was to predict continuous outputs such as insurance charges based on input features like age, BMI, and region.

Key Concepts Covered

1. Basic Structure of an ANN:

- 1. Input Layer, Hidden Layer, Output Layer:** We designed a network with these layers to model the data.
- 2. Activation Function:** We applied the ReLU activation function for hidden layers and a linear activation function for the output layer, suited for regression problems.

2. Forward Propagation:

- We used forward propagation to pass inputs through the layers and predict outputs based on learned weights.
- The Mean Squared Error (MSE)** loss function was applied to measure the difference between predicted and actual values.

3. Backward Propagation & Gradient Descent:

- Using gradient descent, we adjusted the weights to minimize the error by computing gradients during backward propagation.
- The importance of the learning rate was highlighted in controlling the speed of training and convergence.

Outcomes

- A solid understanding of ANN architecture and how data flows through it.
- Explored the role of forward and backward propagation in learning.
- Worked with loss functions and gradient descent to train the model.

Exercise 2: Classification on Churn Modeling (Binary Classification using Cross-Entropy and Gradient Descent)

Objective:

The focus was on applying ANNs to a binary classification task, predicting customer churn. We developed a model to determine if a customer would leave the service.

Key Concepts Covered

1. Binary Classification:

1. We transitioned from regression to classification, implementing a **sigmoid activation** in the output layer to output probabilities for binary outcomes.

2. Loss Function:

1. We used **Binary Cross-Entropy** as the loss function to measure the difference between the predicted probabilities and the actual class labels.

3. Gradient Descent in Classification:

1. We utilized gradient descent to adjust weights based on binary cross-entropy loss and sigmoid activation.

Takeaways

- Developed an understanding of how ANNs handle binary classification tasks.
- Worked with binary cross-entropy for classification loss.
- Learned to evaluate classification models using accuracy and other metrics.

Exercise 3: Regression on Insurance Dataset with Different Optimizers

Objective:

We revisited the insurance dataset to demonstrate how different optimizers influence model training. By changing the optimization algorithm, we examined the effects on convergence speed and accuracy.

Key Concepts Covered

1. Optimization Algorithms:

1. We explored advanced optimizers like **Adam**, **RMSprop**, and **Momentum**, comparing them to simple gradient descent. Each optimizer's impact on learning efficiency and speed was observed.

2. Challenges with Gradient Descent:

1. We addressed common issues like slow convergence or local minima, seeing how advanced optimizers like Adam and RMSprop help overcome these challenges.

3. Implementation of Adam Optimizer:

1. Adam optimizers was tested, and their effect on training and convergence rate was observed.

Takeaways

- Gained insight into how different optimizers impact the training process of neural networks.
- Compared gradient descent with more advanced optimizers to understand their practical implications.

Exercise 4: Multilayer ANN for Classification on Iris Dataset

Objective:

We will extend the neural network by adding multiple hidden layers to solve a multi-class classification problem using the Iris dataset. The goal is to classify different species of iris flowers based on input features like sepal length, sepal width, petal length, and petal width.

1. Multilayer Perceptron (MLP) for Classification:

1. We will build a deeper neural network by adding multiple hidden layers. This will allow the model to capture more complex patterns in the data for multi-class classification.

2. Softmax Activation for Multi-Class Classification:

1. In the output layer, we will apply the **softmax activation function**. This will help the network output probabilities for each class (setosa, versicolor, virginica), making it suitable for multi-class classification.

3. Loss Function - Categorical Cross-Entropy:

1. We will use **categorical cross-entropy** as the loss function, which is optimal for multi-class problems like this one.

Steps in the Code for Building a Multilayer ANN from Scratch for the Iris Dataset

Step 1: Import Libraries

Description: This step imports the necessary libraries and modules for data manipulation, dataset loading, splitting, standardization, and accuracy measurement.

```
import numpy as np  
  
from sklearn.datasets import load_iris  
  
from sklearn.model_selection import train_test_split  
  
from sklearn.preprocessing import StandardScaler  
  
from sklearn.metrics import accuracy_score
```

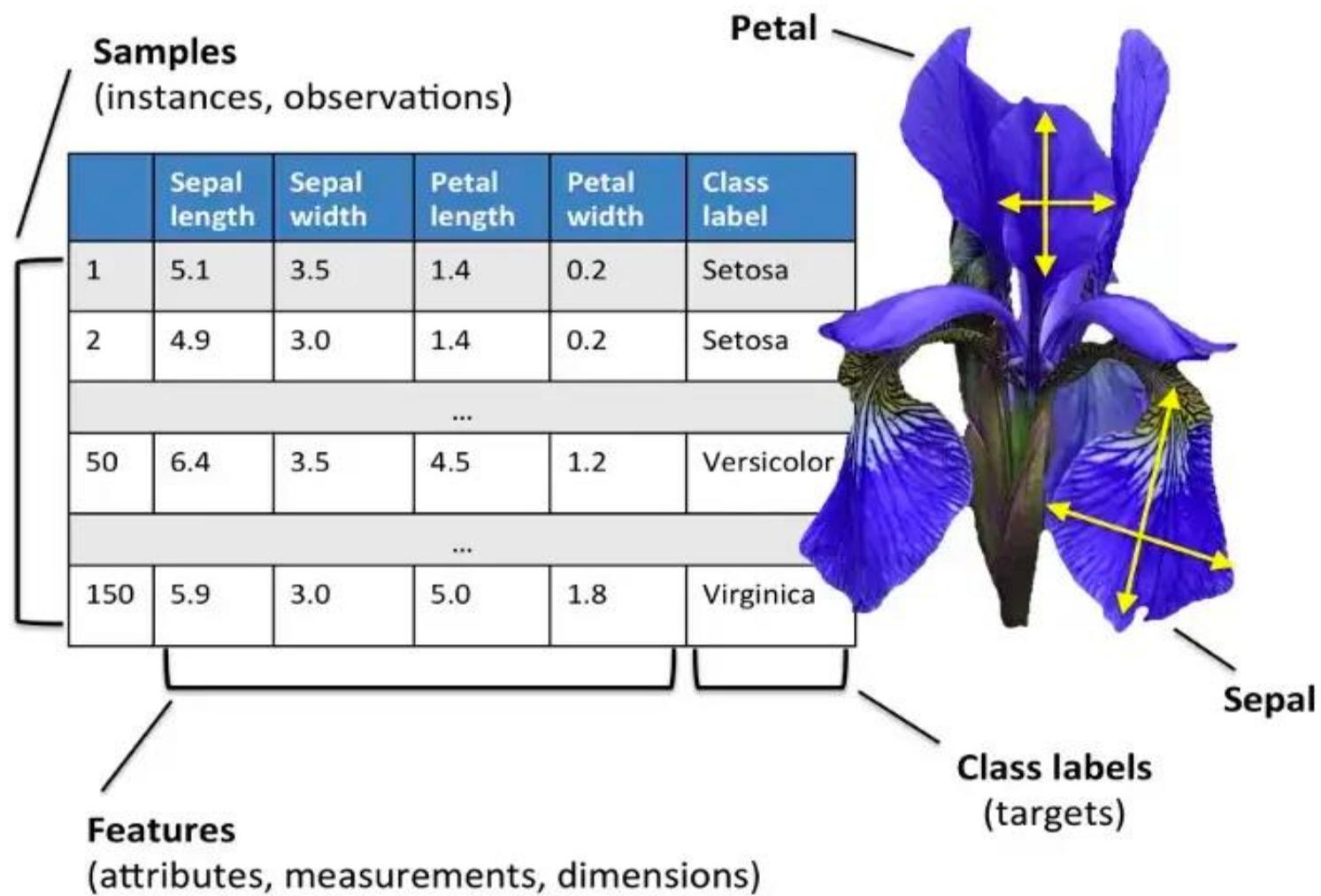
Step 2: Load the Iris Dataset

Description: load the Iris dataset and extracts the features and target labels:

- X: Contains the feature data (sepal length, sepal width, petal length, petal width).
- y: Contains the target labels (species of the iris).

```
data = load_iris()  
x = data.data  
y = data.target
```

iris setosa**iris versicolor****iris virginica**



Step3 : Convert Labels to One-Hot Encoding

Description: This step converts the target variable into a one-hot encoded format:

- **Function Definition:** The `one_hot_encoding` function creates a one-hot encoded representation of the target labels using NumPy's `eye` function, which generates a 2D identity matrix. For example, if a sample belongs to class 1, it will be represented as [0, 1, 0].
- `num_classes`: Specifies the number of unique classes (3 for the Iris dataset).
- `y_one_hot`: The resulting one-hot encoded labels, which are used for multi-class classification.

One-Hot Encoding: An Overview

One-hot encoding is a technique used in machine learning and data processing to convert categorical variables into a numerical format.

Example: One-Hot Encoding on a Sample Dataset

Sample Dataset

Consider a simple dataset with a feature "Color":

ID	Color
1	Red
2	Green
3	Blue
4	Green
5	Red

The transformed dataset after one-hot encoding would look like this:

ID	Color	Color_Red	Color_Green	Color_Blue
1	Red	1	0	0
2	Green	0	1	0
3	Blue	0	0	1
4	Green	0	1	0
5	Red	1	0	0

```
# Convert Labels to one-hot encoding
def one_hot_encoding(y, num_classes):
    return np.eye(num_classes)[y]

y_one_hot = one_hot_encoding(y, num_classes=3)
```

- **y**: The array of target labels (class indices).
- **num_classes**: The total number of unique classes present in the target variable.
- **np.eye(num_classes)**: This function from the NumPy library creates an identity matrix of size `num_classes x num_classes`. An identity matrix has ones on the diagonal and zeros elsewhere.

For instance, if num_classes is 3, the output will look like this:

```
[[1. 0. 0.]  
 [0. 1. 0.]  
 [0. 0. 1.]]
```

- **[y]:** This indexing operation selects rows from the identity matrix. Each entry in the target array y acts as an index to the identity matrix, effectively selecting the corresponding one-hot encoded vector for each class label.
- For example, if y is [0, 1, 2, 1], the function will map:
 - 0 → [1, 0, 0] (Setosa)
 - 1 → [0, 1, 0] (Versicolor)
 - 2 → [0, 0, 1] (Virginica)
 - 1 → [0, 1, 0] (Versicolor)
- **Return Value:** The function returns a 2D array where each row corresponds to the one-hot encoded vector for the respective class label in y.

- Here, we call the `one_hot_encoding` function with `y` and `num_classes` set to 3 (since there are three classes in the Iris dataset).
- The result is stored in `y_one_hot`, which will be a 2D array with shape `(n_samples, n_classes)` where `n_samples` is the number of samples in the dataset and `n_classes` is the number of unique classes (3 in this case).

Step 4 : Split the Dataset and Standardize Features

Description:

- **Split Dataset:** This part divides the dataset into training and testing sets:

- `train_test_split`: A utility function that randomly splits the data.
- `X_train, y_train`: The training data and corresponding one-hot encoded labels.
- `X_test, y_test`: The testing data and corresponding labels.
- `test_size=0.2`: Indicates that 20% of the data will be reserved for testing.
- `random_state=42`: Ensures that the random splitting is reproducible.

- **Standardize Features:** This part standardizes the feature values:

- `StandardScaler`: Used to standardize the features to have a mean of 0 and a standard deviation of 1.
- `fit_transform()`: Computes the mean and standard deviation from the training data and applies the transformation.
- `transform()`: Standardizes the testing data using the same statistics from the training data.

```
# Split the dataset into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(X, y_onehot, test_size=0.2, random_state=42)

# Standardize the features
scaler = StandardScaler()
X_train = scaler.fit_transform(X_train)
X_test = scaler.transform(X_test)
```

Step 5: Initialize Weights and Biases

Description: This step initializes weights and biases for each layer:

- Weights are initialized using a normal distribution to introduce randomness.
- Biases are initialized to zeros, which is a common practice.

```
def initialize_parameters(input_size, hidden_size1, hidden_size2, output_size):
    np.random.seed(42) # For reproducibility

    # Weights and biases for input to first hidden layer
    weights_input_hidden1 = np.random.randn(input_size, hidden_size1) * 0.01
    bias_hidden1 = np.zeros((1, hidden_size1))

    # Weights and biases for first hidden layer to second hidden layer
    weights_hidden1_hidden2 = np.random.randn(hidden_size1, hidden_size2) * 0.01
    bias_hidden2 = np.zeros((1, hidden_size2))

    # Weights and biases for second hidden layer to output layer
    weights_hidden2_output = np.random.randn(hidden_size2, output_size) * 0.01
    bias_output = np.zeros((1, output_size))

    return {
        "weights_input_hidden1": weights_input_hidden1,
        "bias_hidden1": bias_hidden1,
        "weights_hidden1_hidden2": weights_hidden1_hidden2,
        "bias_hidden2": bias_hidden2,
        "weights_hidden2_output": weights_hidden2_output,
        "bias_output": bias_output
    }
```

Step 6: Define Activation Functions and compute loss function (CCE)

Description: This step defines the activation functions used in the network:

- **Sigmoid:** Used for the hidden layers, it introduces non-linearity.
- **Softmax:** Used for the output layer, it converts logits into probabilities for multi-class classification.

```
def sigmoid(z):
    return 1 / (1 + np.exp(-z))

def sigmoid_derivative(z):
    s = sigmoid(z)
    return s * (1 - s)

def softmax(z):
    exp_z = np.exp(z - np.max(z, axis=1, keepdims=True))
    return exp_z / np.sum(exp_z, axis=1, keepdims=True)
```

$$\text{CCE} = -\frac{1}{N} \sum_{j=1}^N \sum_{i=1}^K y_{ij} \cdot \log(\hat{y}_{ij} + \epsilon)$$

where:

- N is the number of samples in the batch.
- K is the number of classes.
- y_{ij} is the true label for class i of sample j (either 0 or 1 in a one-hot encoded format).
- \hat{y}_{ij} is the predicted probability for class i of sample j .
- ϵ is a small constant (e.g., 1×10^{-8}) added to avoid taking the logarithm of zero.

```
def compute_loss(y_true, y_pred):
    """Calculate the cross-entropy loss."""
    loss = -np.mean(np.sum(y_true * np.log(y_pred + 1e-8), axis=1))
    return loss
```

step7: Forward Propagation

Define a function to perform forward propagation:

Compute the hidden layer activation by applying the sigmoid function. Compute the output layer activation using the softmax function.

Implement the function forward():

```
(X, weights_input_hidden1, bias_hidden1, weights_hidden1_hidden2,  
bias_hidden2, weights_hidden2_output, bias_output)
```

```
def forward(X, weights_input_hidden1, bias_hidden1, weights_hidden1_hidden2, bias_hidden2, weights_hidden2_output, bias_output):
    # First hidden layer
    z_hidden1 = np.dot(X, weights_input_hidden1) + bias_hidden1
    a_hidden1 = sigmoid(z_hidden1)

    # Second hidden layer
    z_hidden2 = np.dot(a_hidden1, weights_hidden1_hidden2) + bias_hidden2
    a_hidden2 = sigmoid(z_hidden2)

    # Output layer
    z_output = np.dot(a_hidden2, weights_hidden2_output) + bias_output
    a_output = softmax(z_output)

    return a_hidden1, a_hidden2, a_output, z_hidden1, z_hidden2, z_output
```

step8: Backward Propagation

Define a function to perform backward propagation:

Compute gradients of the loss function with respect to weights and biases using the chain rule. Update gradients for the weights and biases of both layers.

Implement the function backward():

(X, y, a_hidden1, a_hidden2, a_output, z_hidden1, z_hidden2,
weights_hidden1_hidden2, weights_hidden2_output)

```
def backward(x, y, a_hidden1, a_hidden2, a_output, z_hidden1, z_hidden2, weights_hidden1_hidden2, weights_hidden2_output):
    m = y.shape[0]

    # Output layer gradients
    dz_output = a_output - y
    dw_hidden2_output = np.dot(a_hidden2.T, dz_output) / m
    db_output = np.sum(dz_output, axis=0, keepdims=True) / m

    # Second hidden layer gradients
    dz_hidden2 = np.dot(dz_output, weights_hidden2_output.T) * sigmoid_derivative(z_hidden2)
    dw_hidden1_hidden2 = np.dot(a_hidden1.T, dz_hidden2) / m
    db_hidden2 = np.sum(dz_hidden2, axis=0, keepdims=True) / m

    # First hidden layer gradients
    dz_hidden1 = np.dot(dz_hidden2, weights_hidden1_hidden2.T) * sigmoid_derivative(z_hidden1)
    dw_input_hidden1 = np.dot(x.T, dz_hidden1) / m
    db_hidden1 = np.sum(dz_hidden1, axis=0, keepdims=True) / m

return dw_input_hidden1, db_hidden1, dw_hidden1_hidden2, db_hidden2, dw_hidden2_output, db_output
```

step9: Train the Neural Network

Define a function to train the network:

Use the forward and backward propagation functions to update weights and biases over multiple epochs. Print the loss value at regular intervals (e.g., every 100 epochs) to monitor training progress. Implement the function train():

Description: This step begins the training loop for the network:

- **Forward Pass:** Calculate activations for each layer using the weights and biases.
- **Compute Loss:** Calculate the loss using the categorical cross-entropy function.
- **Backward Pass:** Calculate gradients to update weights and biases (not fully shown here).

```
def train(X_train, y_train, input_size, hidden_size1, hidden_size2, output_size, learning_rate=0.01, epochs=1000):
    # Initialize parameters with two hidden layers
    parameters = initialize_parameters(input_size, hidden_size1, hidden_size2, output_size)

    for epoch in range(epochs):
        # Forward pass
        a_hidden1, a_hidden2, a_output, z_hidden1, z_hidden2, z_output = forward(
            X_train,
            parameters["weights_input_hidden1"], parameters["bias_hidden1"],
            parameters["weights_hidden1_hidden2"], parameters["bias_hidden2"],
            parameters["weights_hidden2_output"], parameters["bias_output"]
        )

        # Compute Loss
        loss = compute_loss(y_train, a_output)

        # Backward pass
        dw_input_hidden1, db_hidden1, dw_hidden1_hidden2, db_hidden2, dw_hidden2_output, db_output = backward(
            X_train, y_train, a_hidden1, a_hidden2, a_output, z_hidden1, z_hidden2,
            parameters["weights_hidden1_hidden2"], parameters["weights_hidden2_output"]
        )

        # Update weights and biases
        parameters["weights_input_hidden1"] -= learning_rate * dw_input_hidden1
        parameters["bias_hidden1"] -= learning_rate * db_hidden1
        parameters["weights_hidden1_hidden2"] -= learning_rate * dw_hidden1_hidden2
        parameters["bias_hidden2"] -= learning_rate * db_hidden2
        parameters["weights_hidden2_output"] -= learning_rate * dw_hidden2_output
        parameters["bias_output"] -= learning_rate * db_output

        # Print loss every 100 epochs
        if epoch % 100 == 0:
            print(f'Epoch {epoch}, Loss: {loss}')

    return parameters
```

step10: make Predictions and Evaluate the Model

Define a function to make predictions:

Use the trained network to compute probabilities for the test set and predict the class with the highest probability. Define a function to evaluate the model:

Calculate the accuracy of the predictions by comparing them with the true labels of the test set.

Implement the functions predict() and evaluation code:

```
def predict(X, parameters):
    """Make predictions using the trained weights."""
    a_hidden1, a_hidden2, a_output, _, _, _ = forward(
        X,
        parameters["weights_input_hidden1"], parameters["bias_hidden1"],
        parameters["weights_hidden1_hidden2"], parameters["bias_hidden2"],
        parameters["weights_hidden2_output"], parameters["bias_output"]
    )
    return np.argmax(a_output, axis=1)

# Train the neural network with two hidden Layers
input_size = X_train.shape[1]
hidden_size1 = 10 # Size of the first hidden layer
hidden_size2 = 10 # Size of the second hidden layer
output_size = y_train.shape[1]
learning_rate = 0.01
epochs = 1000

# Train the model
parameters = train(X_train, y_train, input_size, hidden_size1, hidden_size2, output_size, learning_rate, epochs)

# Make predictions and evaluate
y_pred = predict(X_test, parameters)
y_true = np.argmax(y_test, axis=1)
accuracy = accuracy_score(y_true, y_pred)
print(f'Accuracy: {accuracy * 100:.2f}%')
```



KESHAV MEMORIAL INSTITUTE OF TECHNOLOGY
AN AUTONOMOUS INSTITUTION - ACCREDITED BY NAAC WITH 'A' GRADE
Narayanaguda, Hyderabad.

Deep Learning

04-11-2024

**By
Asha**

Building ANN from Scratch

Multilayer and Multiclass ANN on Iris dataset

Exercise 4: Multilayer ANN for Classification on Iris Dataset

Objective:

We will extend the neural network by adding multiple hidden layers to solve a multi-class classification problem using the Iris dataset. The goal is to classify different species of iris flowers based on input features like sepal length, sepal width, petal length, and petal width.

1. Multilayer Perceptron (MLP) for Classification:

1. We will build a deeper neural network by adding multiple hidden layers. This will allow the model to capture more complex patterns in the data for multi-class classification.

2. Softmax Activation for Multi-Class Classification:

1. In the output layer, we will apply the **softmax activation function**. This will help the network output probabilities for each class (setosa, versicolor, virginica), making it suitable for multi-class classification.

3. Loss Function - Categorical Cross-Entropy:

1. We will use **categorical cross-entropy** as the loss function, which is optimal for multi-class problems like this one.

Steps in the Code for Building a Multilayer ANN from Scratch for the Iris Dataset

Step 1: Import Libraries

Description: This step imports the necessary libraries and modules for data manipulation, dataset loading, splitting, standardization, and accuracy measurement.

```
import numpy as np  
  
from sklearn.datasets import load_iris  
  
from sklearn.model_selection import train_test_split  
  
from sklearn.preprocessing import StandardScaler  
  
from sklearn.metrics import accuracy_score
```

Step 2: Load the Iris Dataset

Description: load the Iris dataset and extracts the features and target labels:

- X: Contains the feature data (sepal length, sepal width, petal length, petal width).
- y: Contains the target labels (species of the iris).

```
data = load_iris()  
x = data.data  
y = data.target
```

iris setosa

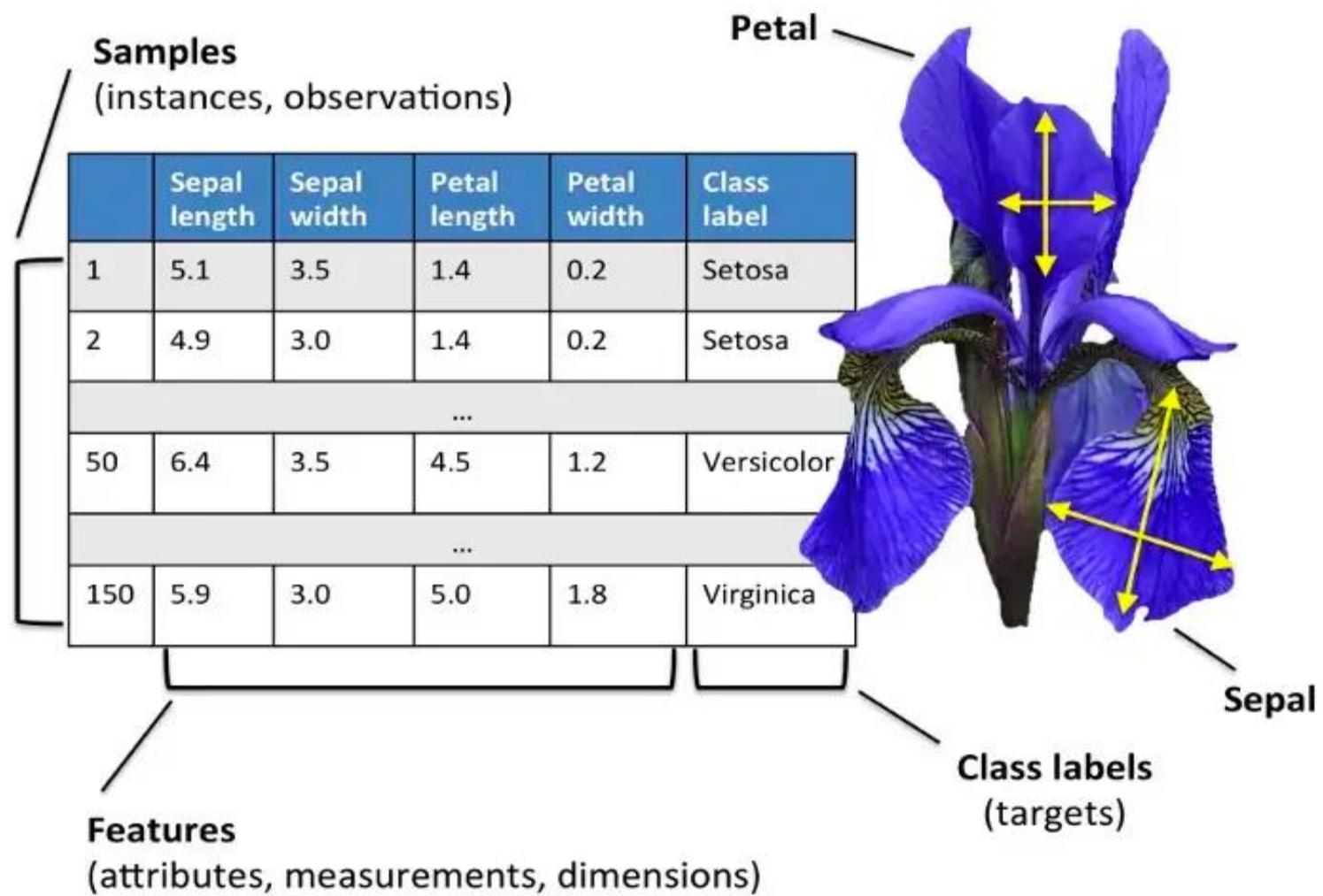
petal sepal

iris versicolor

petal sepal

iris virginica

petal sepal



Step3 : Convert Labels to One-Hot Encoding

Description: This step converts the target variable into a one-hot encoded format:

- **Function Definition:** The `one_hot_encoding` function creates a one-hot encoded representation of the target labels using NumPy's `eye` function, which generates a 2D identity matrix. For example, if a sample belongs to class 1, it will be represented as [0, 1, 0].
- `num_classes`: Specifies the number of unique classes (3 for the Iris dataset).
- `y_one_hot`: The resulting one-hot encoded labels, which are used for multi-class classification.

One-Hot Encoding: An Overview

One-hot encoding is a technique used in machine learning and data processing to convert categorical variables into a numerical format.

Example: One-Hot Encoding on a Sample Dataset

Sample Dataset

Consider a simple dataset with a feature "Color":

ID	Color
1	Red
2	Green
3	Blue
4	Green
5	Red

The transformed dataset after one-hot encoding would look like this:

ID	Color	Color_Red	Color_Green	Color_Blue
1	Red	1	0	0
2	Green	0	1	0
3	Blue	0	0	1
4	Green	0	1	0
5	Red	1	0	0

```
# Convert Labels to one-hot encoding
def one_hot_encoding(y, num_classes):
    return np.eye(num_classes)[y]

y_one_hot = one_hot_encoding(y, num_classes=3)
```

- **y**: The array of target labels (class indices).
- **num_classes**: The total number of unique classes present in the target variable.
- **np.eye(num_classes)**: This function from the NumPy library creates an identity matrix of size `num_classes x num_classes`. An identity matrix has ones on the diagonal and zeros elsewhere.

For instance, if num_classes is 3, the output will look like this:

```
[[1. 0. 0.]  
 [0. 1. 0.]  
 [0. 0. 1.]]
```

- **[y]:** This indexing operation selects rows from the identity matrix. Each entry in the target array y acts as an index to the identity matrix, effectively selecting the corresponding one-hot encoded vector for each class label.
- For example, if y is [0, 1, 2, 1], the function will map:
 - 0 → [1, 0, 0] (Setosa)
 - 1 → [0, 1, 0] (Versicolor)
 - 2 → [0, 0, 1] (Virginica)
 - 1 → [0, 1, 0] (Versicolor)
- **Return Value:** The function returns a 2D array where each row corresponds to the one-hot encoded vector for the respective class label in y.

- Here, we call the `one_hot_encoding` function with `y` and `num_classes` set to 3 (since there are three classes in the Iris dataset).
- The result is stored in `y_one_hot`, which will be a 2D array with shape `(n_samples, n_classes)` where `n_samples` is the number of samples in the dataset and `n_classes` is the number of unique classes (3 in this case).

Step 4 : Split the Dataset and Standardize Features

Description:

- **Split Dataset:** This part divides the dataset into training and testing sets:

- `train_test_split`: A utility function that randomly splits the data.
- `X_train, y_train`: The training data and corresponding one-hot encoded labels.
- `X_test, y_test`: The testing data and corresponding labels.
- `test_size=0.2`: Indicates that 20% of the data will be reserved for testing.
- `random_state=42`: Ensures that the random splitting is reproducible.

- **Standardize Features:** This part standardizes the feature values:

- `StandardScaler`: Used to standardize the features to have a mean of 0 and a standard deviation of 1.
- `fit_transform()`: Computes the mean and standard deviation from the training data and applies the transformation.
- `transform()`: Standardizes the testing data using the same statistics from the training data.

```
# Split the dataset into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(X, y_onehot, test_size=0.2, random_state=42)

# Standardize the features
scaler = StandardScaler()
X_train = scaler.fit_transform(X_train)
X_test = scaler.transform(X_test)
```

Step 5: Initialize Weights and Biases

Description: This step initializes weights and biases for each layer:

- Weights are initialized using a normal distribution to introduce randomness.
- Biases are initialized to zeros, which is a common practice.

```
def initialize_parameters(input_size, hidden_size1, hidden_size2, output_size):
    np.random.seed(42) # For reproducibility

    # Weights and biases for input to first hidden layer
    weights_input_hidden1 = np.random.randn(input_size, hidden_size1) * 0.01
    bias_hidden1 = np.zeros((1, hidden_size1))

    # Weights and biases for first hidden layer to second hidden layer
    weights_hidden1_hidden2 = np.random.randn(hidden_size1, hidden_size2) * 0.01
    bias_hidden2 = np.zeros((1, hidden_size2))

    # Weights and biases for second hidden layer to output layer
    weights_hidden2_output = np.random.randn(hidden_size2, output_size) * 0.01
    bias_output = np.zeros((1, output_size))

    return {
        "weights_input_hidden1": weights_input_hidden1,
        "bias_hidden1": bias_hidden1,
        "weights_hidden1_hidden2": weights_hidden1_hidden2,
        "bias_hidden2": bias_hidden2,
        "weights_hidden2_output": weights_hidden2_output,
        "bias_output": bias_output
    }
```

Step 6: Define Activation Functions and compute loss function (CCE)

Description: This step defines the activation functions used in the network:

- **Sigmoid:** Used for the hidden layers, it introduces non-linearity.
- **Softmax:** Used for the output layer, it converts logits into probabilities for multi-class classification.

```
def sigmoid(z):
    return 1 / (1 + np.exp(-z))

def sigmoid_derivative(z):
    s = sigmoid(z)
    return s * (1 - s)

def softmax(z):
    exp_z = np.exp(z - np.max(z, axis=1, keepdims=True))
    return exp_z / np.sum(exp_z, axis=1, keepdims=True)
```

$$\text{CCE} = -\frac{1}{N} \sum_{j=1}^N \sum_{i=1}^K y_{ij} \cdot \log(\hat{y}_{ij} + \epsilon)$$

where:

- N is the number of samples in the batch.
- K is the number of classes.
- y_{ij} is the true label for class i of sample j (either 0 or 1 in a one-hot encoded format).
- \hat{y}_{ij} is the predicted probability for class i of sample j .
- ϵ is a small constant (e.g., 1×10^{-8}) added to avoid taking the logarithm of zero.

```
def compute_loss(y_true, y_pred):  
    """Calculate the cross-entropy loss."""  
    loss = -np.mean(np.sum(y_true * np.log(y_pred + 1e-8), axis=1))  
    return loss
```

- `y_true` is multiplied element-wise with the logarithm of `y_pred`. The expression `np.log(y_pred + 1e-8)` takes the log of each predicted probability in `y_pred`.
- The small constant `1e-8` is added to `y_pred` to avoid taking the logarithm of zero, which is undefined and would cause computational errors.

`np.sum(..., axis=1):`

- This sums the values across each row (i.e., across the classes) for every sample in the batch.

`np.mean(...):`

- `np.mean` calculates the average of the cross-entropy values across all samples in the batch.

The negative sign at the beginning ensures the result is positive, as cross-entropy is defined to be a positive measure.

step7: Forward Propagation

Define a function to perform forward propagation:

Compute the hidden layer activation by applying the sigmoid function. Compute the output layer activation using the softmax function.

Implement the function forward():

```
(X, weights_input_hidden1, bias_hidden1, weights_hidden1_hidden2,  
bias_hidden2, weights_hidden2_output, bias_output)
```

```
def forward(x, weights_input_hidden1, bias_hidden1, weights_hidden1_hidden2, bias_hidden2, weights_hidden2_output, bias_output):
    # First hidden layer
    z_hidden1 = np.dot(x, weights_input_hidden1) + bias_hidden1
    a_hidden1 = sigmoid(z_hidden1)

    # Second hidden layer
    z_hidden2 = np.dot(a_hidden1, weights_hidden1_hidden2) + bias_hidden2
    a_hidden2 = sigmoid(z_hidden2)

    # Output layer
    z_output = np.dot(a_hidden2, weights_hidden2_output) + bias_output
    a_output = softmax(z_output)

    return a_hidden1, a_hidden2, a_output, z_hidden1, z_hidden2, z_output
```

step8: Backward Propagation

Define a function to perform backward propagation:

Compute gradients of the loss function with respect to weights and biases using the chain rule. Update gradients for the weights and biases of both layers.

Implement the function backward():

(X, y, a_hidden1, a_hidden2, a_output, z_hidden1, z_hidden2,
weights_hidden1_hidden2, weights_hidden2_output)

```
def backward(x, y, a_hidden1, a_hidden2, a_output, z_hidden1, z_hidden2, weights_hidden1_hidden2, weights_hidden2_output):
    m = y.shape[0]

    # Output layer gradients
    dz_output = a_output - y
    dw_hidden2_output = np.dot(a_hidden2.T, dz_output) / m
    db_output = np.sum(dz_output, axis=0, keepdims=True) / m

    # Second hidden layer gradients
    dz_hidden2 = np.dot(dz_output, weights_hidden2_output.T) * sigmoid_derivative(z_hidden2)
    dw_hidden1_hidden2 = np.dot(a_hidden1.T, dz_hidden2) / m
    db_hidden2 = np.sum(dz_hidden2, axis=0, keepdims=True) / m

    # First hidden layer gradients
    dz_hidden1 = np.dot(dz_hidden2, weights_hidden1_hidden2.T) * sigmoid_derivative(z_hidden1)
    dw_input_hidden1 = np.dot(x.T, dz_hidden1) / m
    db_hidden1 = np.sum(dz_hidden1, axis=0, keepdims=True) / m

return dw_input_hidden1, db_hidden1, dw_hidden1_hidden2, db_hidden2, dw_hidden2_output, db_output
```

step9: Train the Neural Network

Define a function to train the network:

Use the forward and backward propagation functions to update weights and biases over multiple epochs. Print the loss value at regular intervals (e.g., every 100 epochs) to monitor training progress. Implement the function train():

Description: This step begins the training loop for the network:

- **Forward Pass:** Calculate activations for each layer using the weights and biases.
- **Compute Loss:** Calculate the loss using the categorical cross-entropy function.
- **Backward Pass:** Calculate gradients to update weights and biases (not fully shown here).

```
def train(X_train, y_train, input_size, hidden_size1, hidden_size2, output_size, learning_rate=0.01, epochs=1000):
    # Initialize parameters with two hidden layers
    parameters = initialize_parameters(input_size, hidden_size1, hidden_size2, output_size)

    for epoch in range(epochs):
        # Forward pass
        a_hidden1, a_hidden2, a_output, z_hidden1, z_hidden2, z_output = forward(
            X_train,
            parameters["weights_input_hidden1"], parameters["bias_hidden1"],
            parameters["weights_hidden1_hidden2"], parameters["bias_hidden2"],
            parameters["weights_hidden2_output"], parameters["bias_output"]
        )

        # Compute Loss
        loss = compute_loss(y_train, a_output)

        # Backward pass
        dw_input_hidden1, db_hidden1, dw_hidden1_hidden2, db_hidden2, dw_hidden2_output, db_output = backward(
            X_train, y_train, a_hidden1, a_hidden2, a_output, z_hidden1, z_hidden2,
            parameters["weights_hidden1_hidden2"], parameters["weights_hidden2_output"]
        )

        # Update weights and biases
        parameters["weights_input_hidden1"] -= learning_rate * dw_input_hidden1
        parameters["bias_hidden1"] -= learning_rate * db_hidden1
        parameters["weights_hidden1_hidden2"] -= learning_rate * dw_hidden1_hidden2
        parameters["bias_hidden2"] -= learning_rate * db_hidden2
        parameters["weights_hidden2_output"] -= learning_rate * dw_hidden2_output
        parameters["bias_output"] -= learning_rate * db_output

        # Print loss every 100 epochs
        if epoch % 100 == 0:
            print(f'Epoch {epoch}, Loss: {loss}')

    return parameters
```

step10: make Predictions and Evaluate the Model

Define a function to make predictions:

Use the trained network to compute probabilities for the test set and predict the class with the highest probability. Define a function to evaluate the model:

Calculate the accuracy of the predictions by comparing them with the true labels of the test set.

Implement the functions predict() and evaluation code:

```
def predict(X, parameters):
    """Make predictions using the trained weights."""
    a_hidden1, a_hidden2, a_output, _, _, _ = forward(
        X,
        parameters["weights_input_hidden1"], parameters["bias_hidden1"],
        parameters["weights_hidden1_hidden2"], parameters["bias_hidden2"],
        parameters["weights_hidden2_output"], parameters["bias_output"]
    )
    return np.argmax(a_output, axis=1)

# Train the neural network with two hidden Layers
input_size = X_train.shape[1]
hidden_size1 = 10 # Size of the first hidden layer
hidden_size2 = 10 # Size of the second hidden layer
output_size = y_train.shape[1]
learning_rate = 0.01
epochs = 1000

# Train the model
parameters = train(X_train, y_train, input_size, hidden_size1, hidden_size2, output_size, learning_rate, epochs)

# Make predictions and evaluate
y_pred = predict(X_test, parameters)
y_true = np.argmax(y_test, axis=1)
accuracy = accuracy_score(y_true, y_pred)
print(f'Accuracy: {accuracy * 100:.2f}%')
```



KESHAV MEMORIAL INSTITUTE OF TECHNOLOGY
AN AUTONOMOUS INSTITUTION - ACCREDITED BY NAAC WITH 'A' GRADE
Narayanauguda, Hyderabad.

Deep Learning

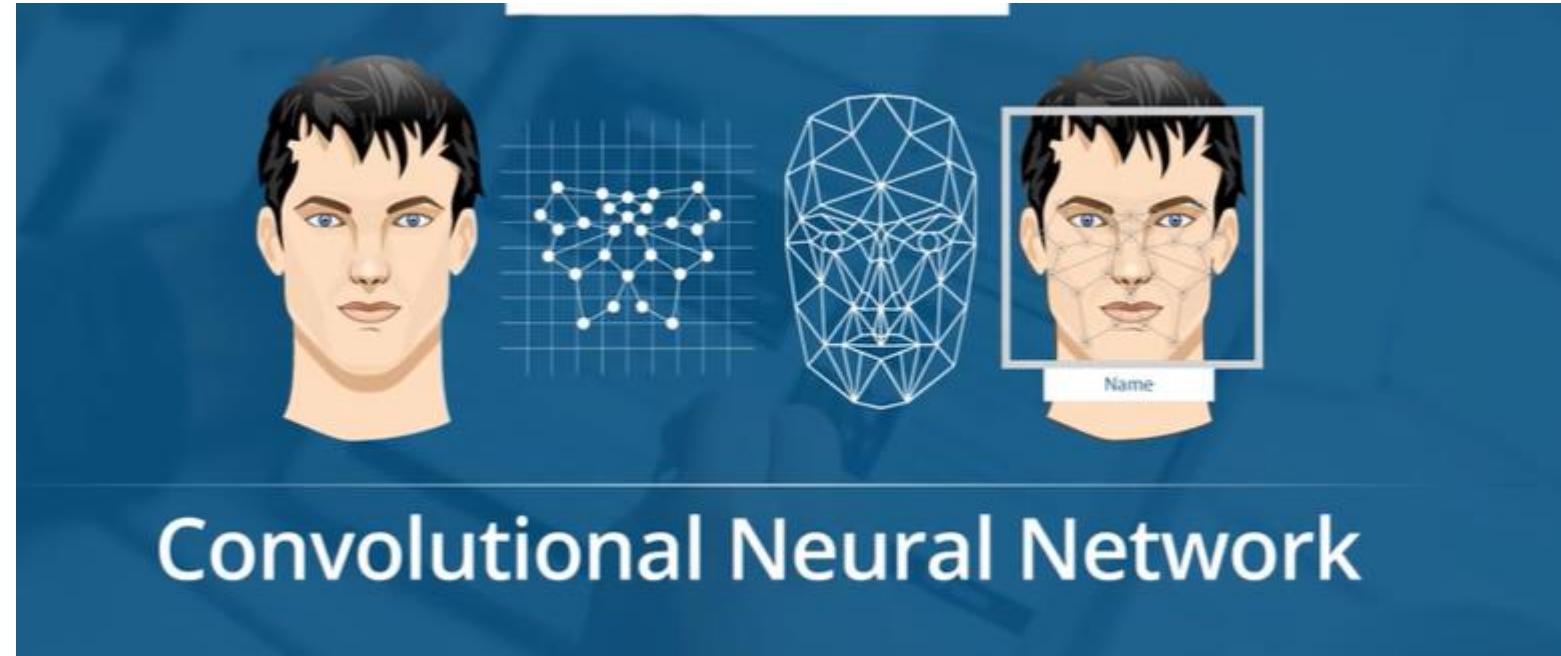
Introduction to CNN

SESSION1

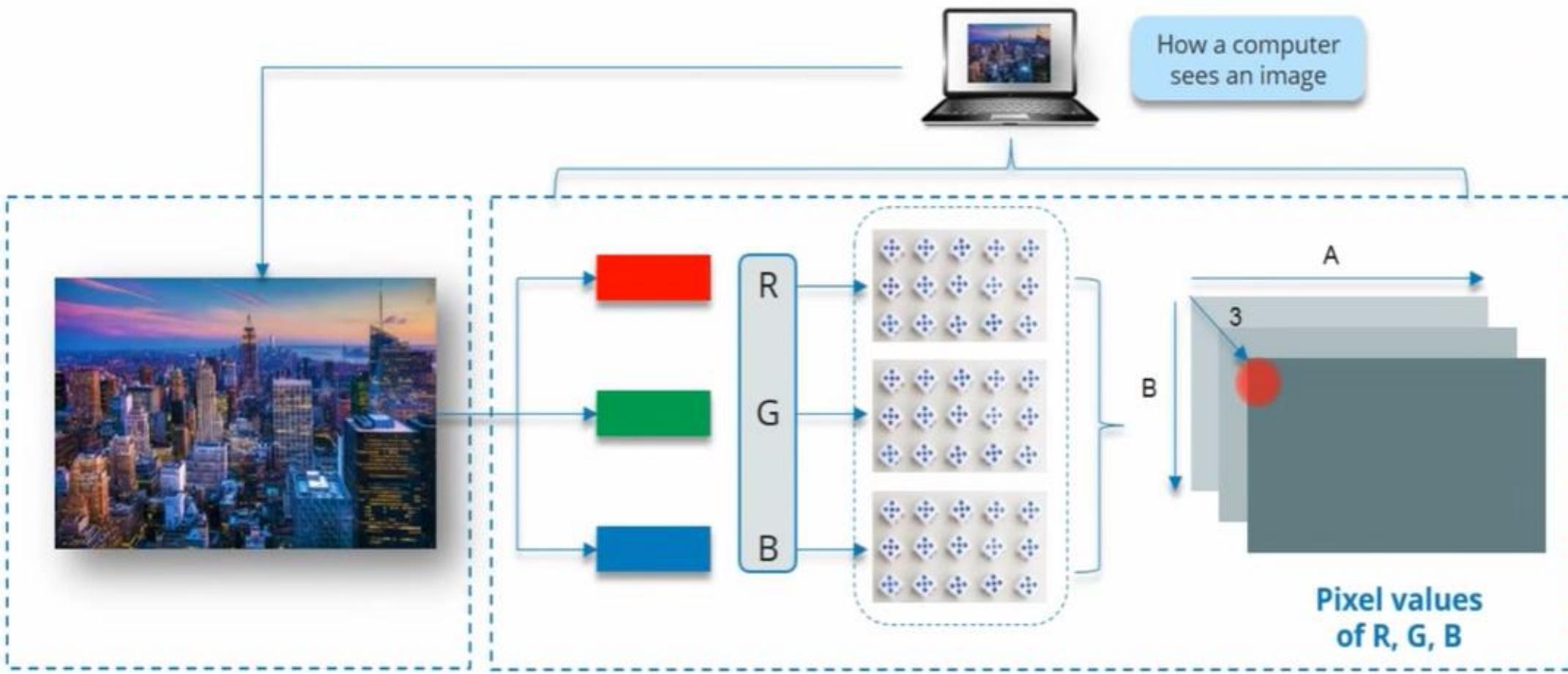
18-11-2024

BY
ASHA

INTRODUCTION TO CNN

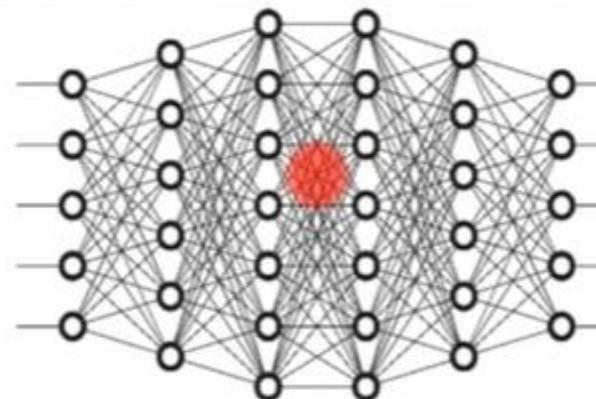


How a Computer Reads an Image



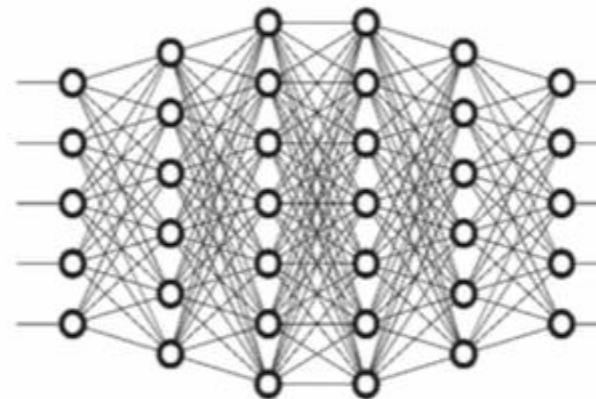
Why Not Fully Connected Networks

Image with
28 x 28 x 3
pixels



*Number of weights in
the first hidden layer
will be 2352*

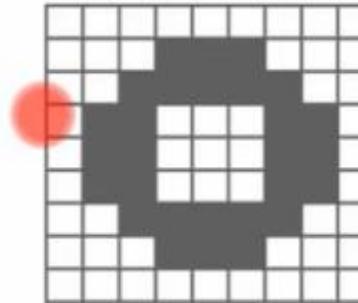
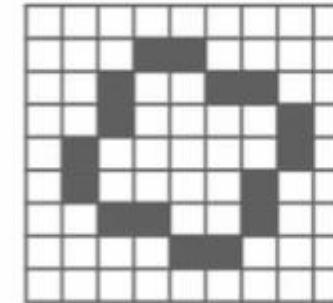
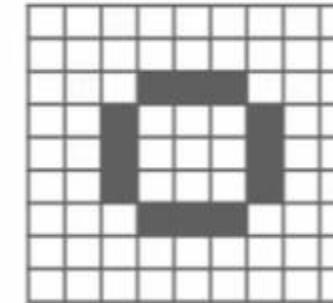
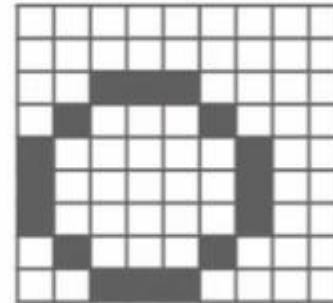
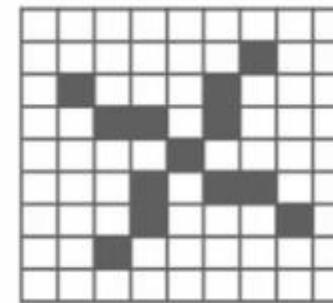
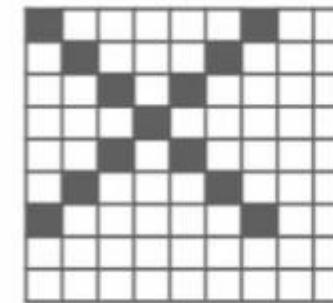
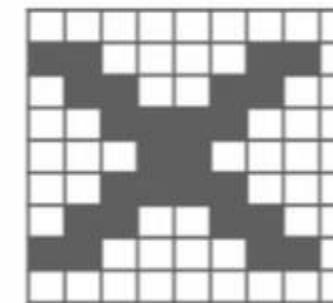
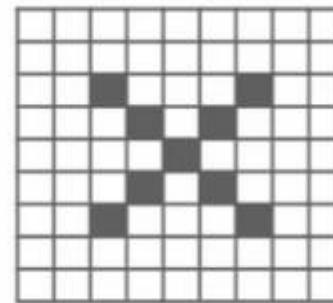
Image with
200 x 200 x 3
pixels



*Number of weights in
the first hidden layer
will be 120,000*

Trickier Case

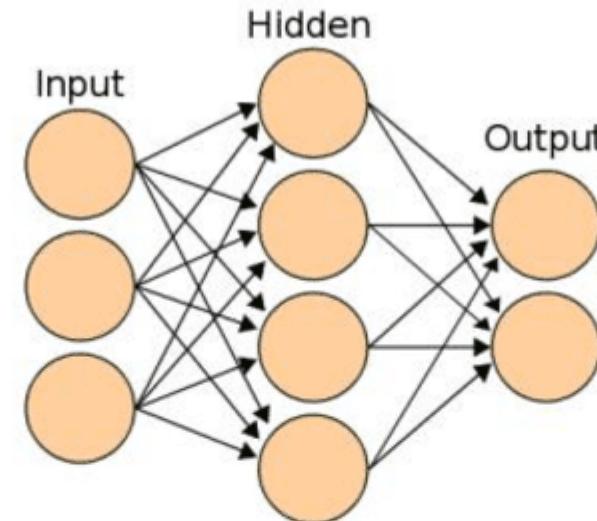
Here, we will have some problems, because X and O images won't always have the same images. There can be certain deformations. Consider the diagrams shown below:



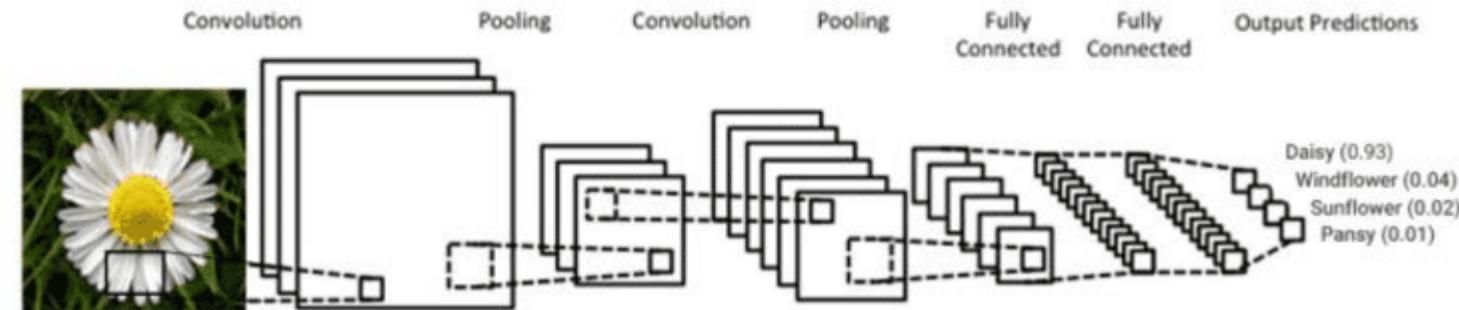
- A **convolutional neural network (CNN)**, is a network architecture for deep learning which learns directly from data.
- CNNs are particularly useful for finding patterns in images to recognize objects.
- They can also be quite effective for classifying non-image data such as audio, time series, and signal data.

Convolutional Neural Networks vs. Artificial Neural Networks

Artificial Neural Network (ANN)



Convolutional Neural Network (CNN)



Aspect	ANN (Artificial Neural Network)	CNN (Convolutional Neural Network)
Architecture	Fully connected layers where each neuron connects to all neurons in the next layer.	Incorporates convolutional layers, pooling layers, and fully connected layers.
Input Data	Works well with structured data (e.g., tabular data) or low-dimensional data.	Designed for spatial data, particularly images, videos, or data with a grid-like topology.
Feature Extraction	Relies on manual feature engineering.	Automatically extracts hierarchical features using convolutional layers.
Parameter Sharing	Does not share weights across neurons, leading to a high number of parameters.	Shares weights in convolutional layers, reducing the number of parameters and improving efficiency.
Use Cases	<ul style="list-style-type: none">- Predictive modeling- Financial forecasting- Tabular data analysis	<ul style="list-style-type: none">- Image classification- Object detection- Facial recognition- Natural language processing
Performance	Performs well on small to medium-sized datasets with simple relationships.	Excels on complex datasets, especially large-scale ones with spatial or hierarchical patterns.
Complexity	Relatively simple structure; easier to train on small datasets.	More complex; requires more computational resources and often larger datasets for effective training.

-1	1	1	1	-1
-1	1	-1	1	-1
-1	1	1	1	-1
-1	-1	-1	1	-1
-1	-1	-1	1	-1
-1	-1	1	-1	-1
-1	1	-1	-1	-1

-1	1	1	1	-1
-1	1	-1	1	-1
-1	1	1	1	-1
-1	-1	-1	1	-1
-1	-1	-1	1	-1
-1	-1	1	-1	-1
-1	1	-1	-1	-1

Location shifted

1	1	1	-1	-1
1	-1	1	-1	-1
1	1	1	-1	-1
-1	-1	1	-1	-1
-1	-1	1	-1	-1
-1	1	-1	-1	-1
1	-1	-1	-1	-1



1	1	1	-1	-1
1	-1	1	-1	-1
1	1	1	-1	-1
-1	-1	1	-1	-1
-1	-1	1	-1	-1
-1	1	-1	-1	-1
1	-1	-1	-1	-1



-1	1	1	1	-1
-1	1	-1	1	-1
-1	1	1	1	-1
-1	-1	-1	1	-1
-1	-1	-1	1	-1
-1	-1	1	-1	-1
-1	1	-1	-1	-1

To handle **variety** in digits we can use simple artificial neural network (ANN)



g

g

g

g



0	0	0	0	0	0	0
0	87	240	210	24	0	0
0	13	0	101	195	0	0
0	35	167	99	210	0	0
0	145	230	240	201	189	140
0	0	102	67	17	13	0
0	0	0	0	0	0	0

7 by 7 grid

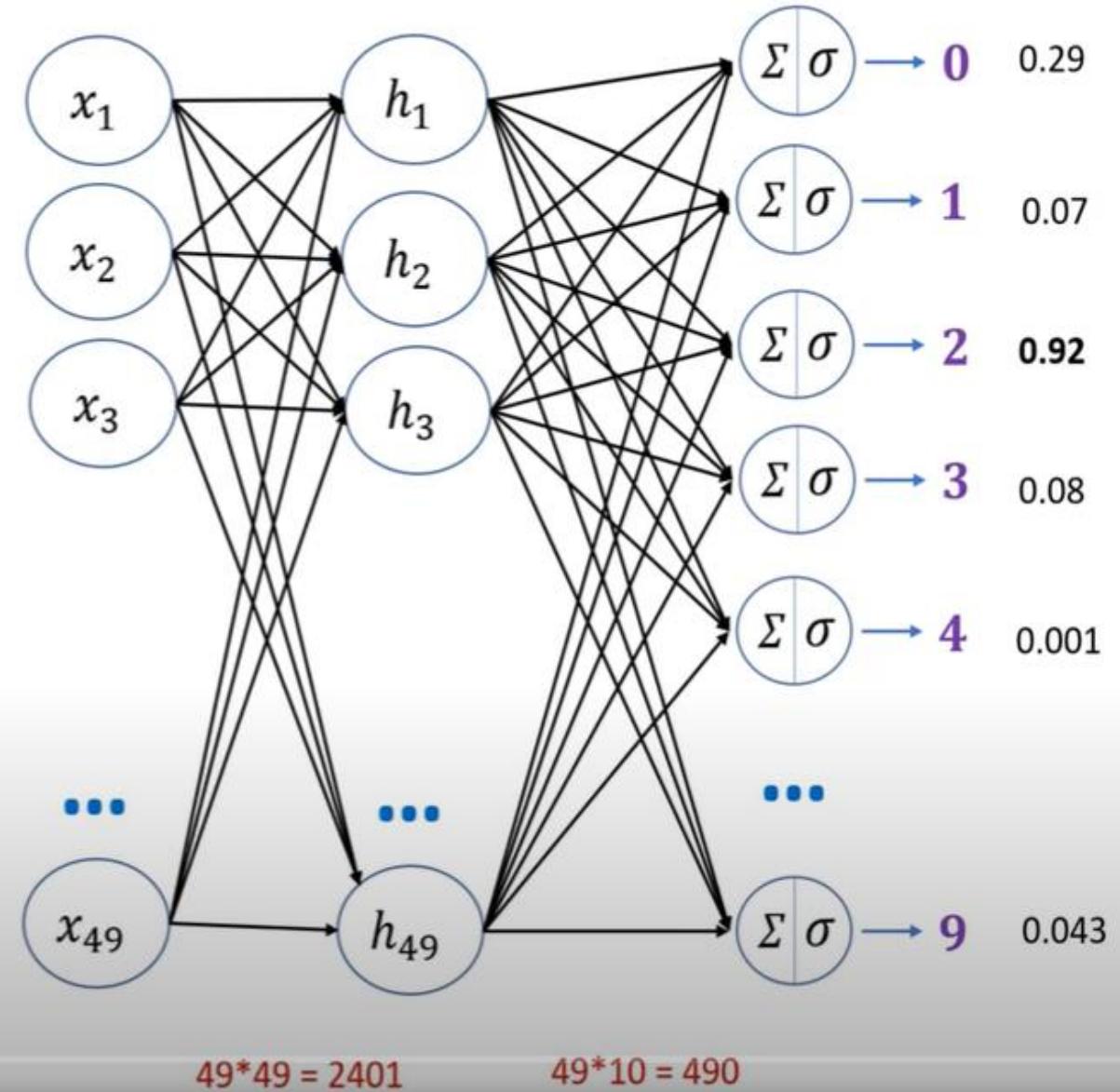




Image size = 1920 x 1080 X 3

First layer neurons = 1920 x 1080 X 3 ~ 6 million



Hidden layer neurons = Let's say you keep it ~ 4 million

Weights between input and hidden layer = $6 \text{ mil} * 4 \text{ mil}$
= 24 million

Disadvantages of using ANN for image classification

1. Too much computation
2. Treats local pixels same as pixels far apart
3. Sensitive to location of an object in an image



Koala's **eye?** = Y



Koala's **nose?** = Y



Koala's **ears?** = Y





Koala's **eye?** = Y



Koala's **nose?** = Y



Koala's **ears?** = Y



Koala's **head?** = Y





Koala's **eye?** = Y



Koala's **nose?** = Y



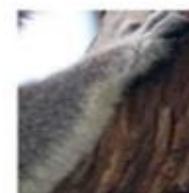
Koala's **ears?** = Y



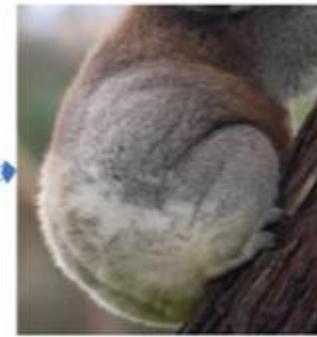
Koala's **head?** = Y



Koala's **hands?** = Y

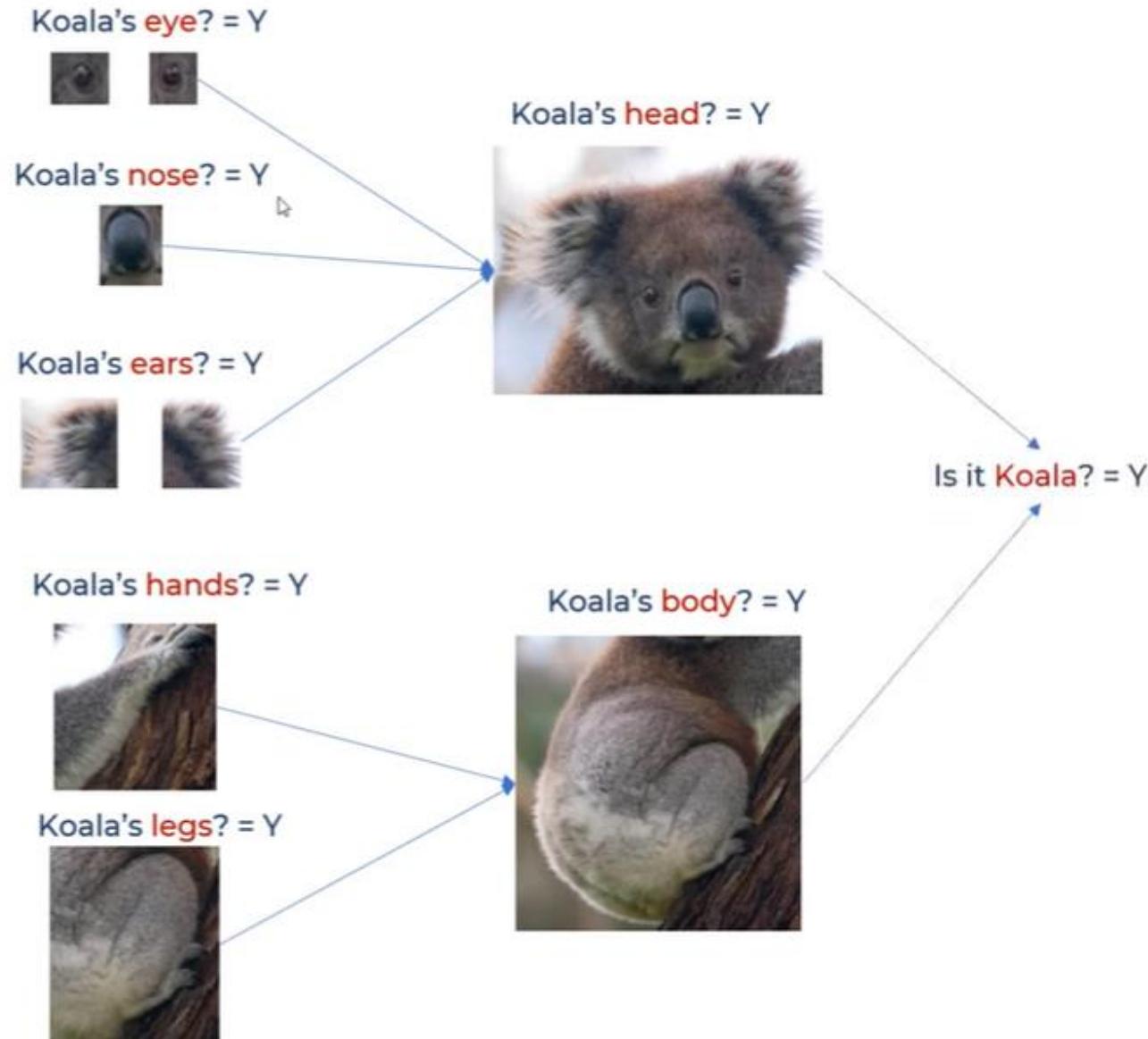


Koala's **body?** = Y

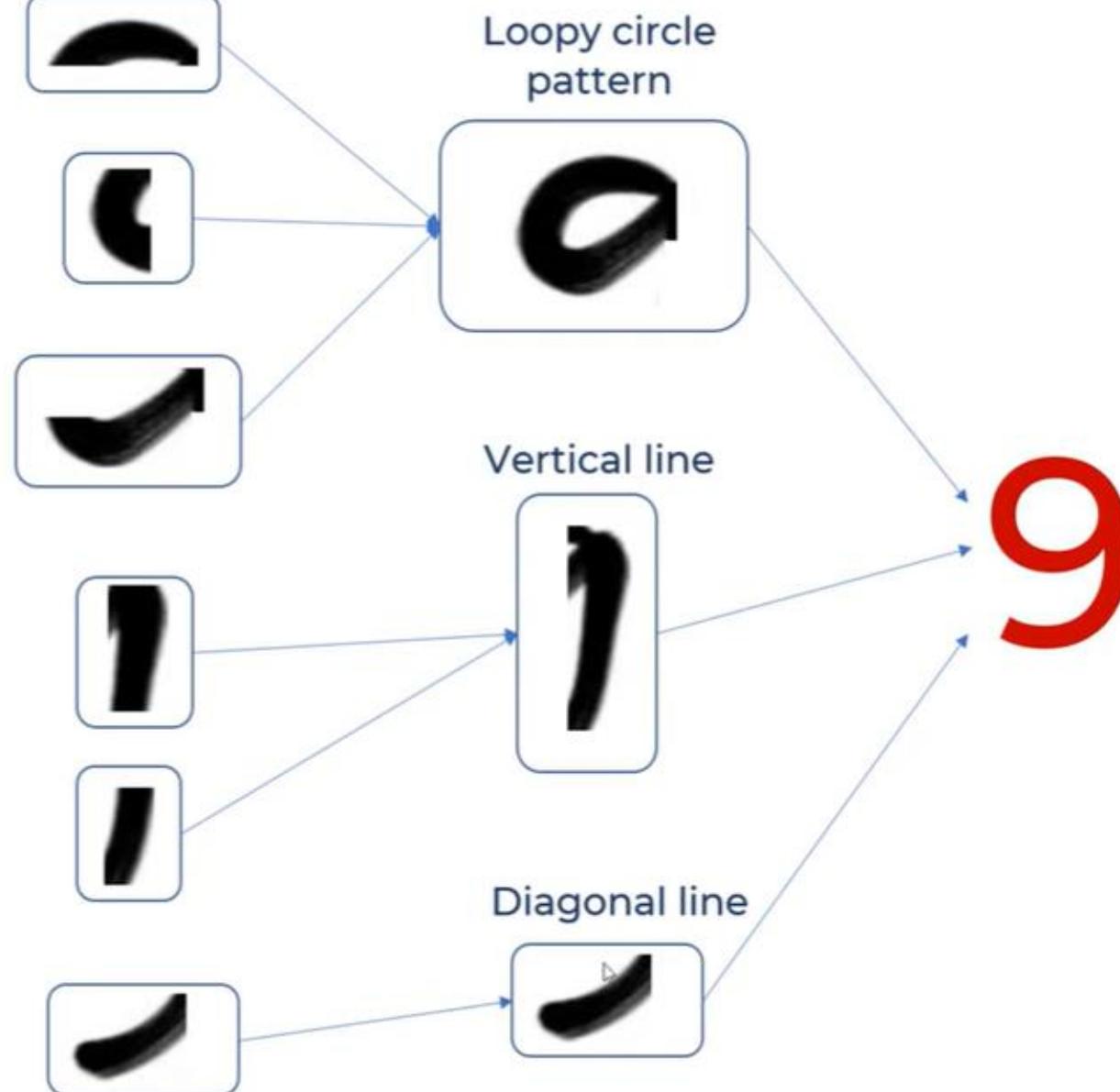


Koala's **legs?** = Y

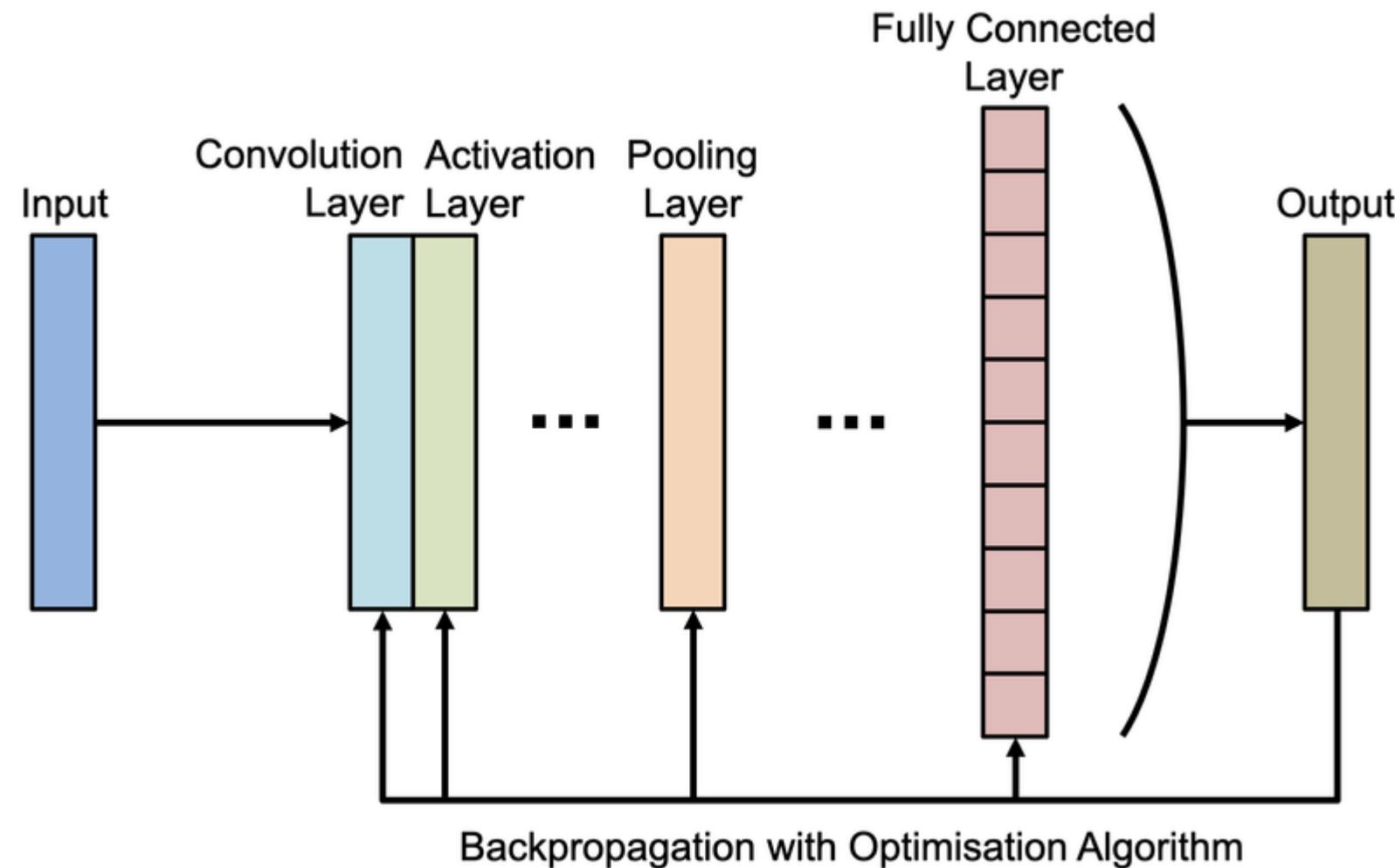


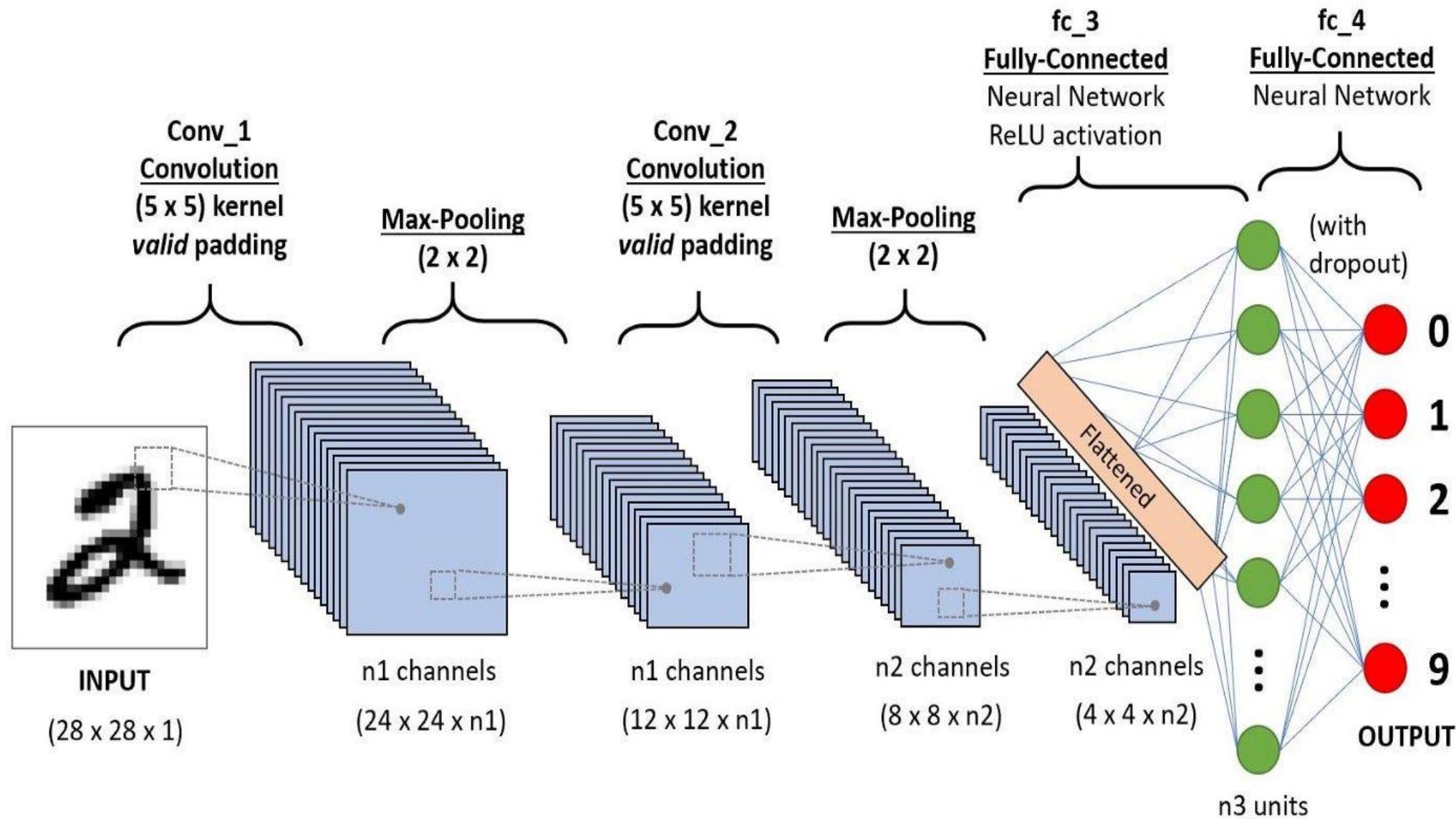


A large, bold, black handwritten digit 'g' is positioned on the left side of the image.



The standard architecture of a CNN is composed of a series of layers, each serving a specific purpose.



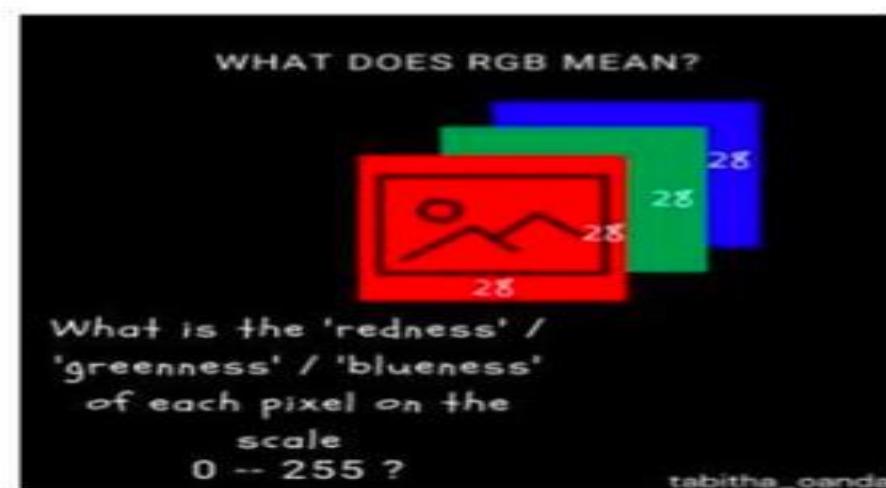
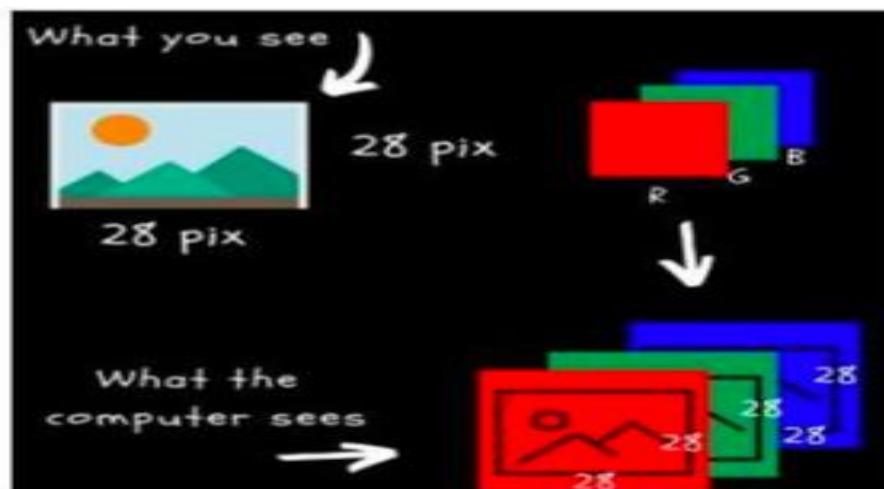


Basic Layers in a CNN Architecture

1. Input Layer
2. Convolutional Layer
3. Activation Layer (e.g., ReLU)
4. Pooling Layer (e.g., Max Pooling)
5. Fully Connected (Dense) Layer
6. Output Layer

1. Input Layer:

1. This layer holds the raw pixel values of the image, typically in the form of a 3D matrix (height x width x channels).
2. For instance, an RGB image with a size of 28x28 pixels would have an input shape of $28 \times 28 \times 3$, where 3 represents the Red, Green, and Blue color channels.



2. Convolutional Layer:

1. The core layer in CNNs, this layer applies a convolutional filters (or kernels) over the input image to detect patterns.
2. Each filter slides (convolves) over the image and computes a "dot product" between the filter and a small section of the input, generating a feature map.
3. Each filter is designed to detect specific patterns like edges, textures, or shapes.

Source layer

5	2	6		8	2	0	1	2
4	3	4		5	1	9	6	3
3	9	2		4	7	7	6	9
1	3	4		6	8	2	2	1
8	4	6	2	3	1	8	8	
5	8	9	0	1	0	2	3	
9	2	6	6	3	6	2	1	
9	8	8	2	6	3	4	5	

Convolutional kernel

-1	0	1
2	1	2
1	-2	0

Destination layer

$$\begin{aligned} & (-1 \times 5) + (0 \times 2) + (1 \times 6) + \\ & (2 \times 4) + (1 \times 3) + (2 \times 4) + \\ & (1 \times 3) + (-2 \times 9) + (0 \times 2) = 5 \end{aligned}$$

To apply the convolution:

- Overlay the Kernel on the Image: Start from the top-left corner of the image and place the kernel so that its center aligns with the current image pixel.
- Element-wise Multiplication: Multiply each element of the kernel with the corresponding element of the image it covers.
- Summation: Sum up all the products obtained from the element-wise multiplication. This sum forms a single pixel in the output feature map.
- Continue the Process: Slide the kernel over to the next pixel and repeat the process across the entire image.

Step-1

1	0	-2	1
-1	0	1	2
0	2	1	0
1	0	0	1



0	1
-1	2



1		

Step-2

1	0	-2	1
-1	0	1	2
0	2	1	0
1	0	0	1



0	1
-1	2



1	0	

Step-3

1	0	-2	1
-1	0	1	2
0	2	1	0
1	0	0	1



0	1
-1	2



1	0	4

Step-4

1	0	-2	1
-1	0	1	2
0	2	1	0
1	0	0	1



0	1
-1	2



1	0	4
4		

Step-5

1	0	-2	1
-1	0	1	2
0	2	1	0
1	0	0	1



0	1
-1	2



1	0	4
4	1	

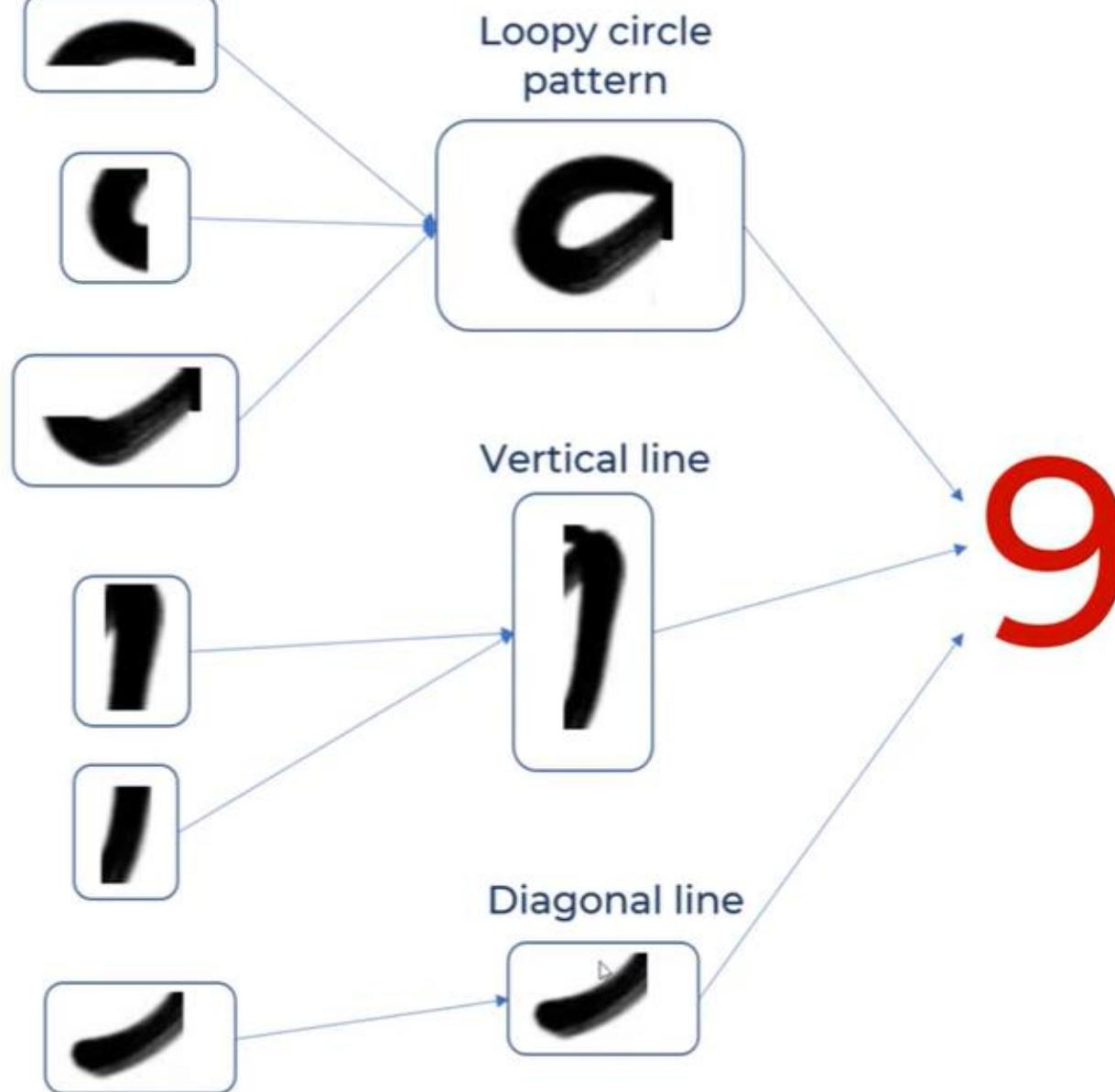
1. Hyperparameters:

1. Filter Size (Kernel Size): Usually 3x3 or 5x5.

2. Stride: Determines the step size as the filter slides over the input.

3. Padding: Controls the spatial dimensions of the output (adding "zero-padding" can preserve input size).

2. Output: The output of a convolutional layer is a set of **feature maps**, each representing the presence of a specific feature across the spatial dimensions.



-1	1	1	1	-1
-1	1	-1	1	-1
-1	1	1	1	-1
-1	-1	-1	1	-1
-1	-1	-1	1	-1
-1	-1	1	-1	-1
-1	1	-1	-1	-1

-1	1	1	1	-1
-1	1	-1	1	-1
-1	1	1	1	-1
-1	-1	-1	1	-1
-1	-1	-1	1	-1
-1	-1	1	-1	-1
-1	1	-1	-1	-1

Loopy pattern
filter

-1	1	1	1	-1
-1	1	-1	1	-1
-1	1	1	1	-1
-1	-1	-1	1	-1
-1	-1	-1	1	-1
-1	-1	1	-1	-1
-1	1	-1	-1	-1

Vertical line
filter

-1	1	1	1	-1
-1	1	-1	1	-1
-1	1	1	1	-1
-1	-1	-1	1	-1
-1	-1	-1	1	-1
-1	-1	1	-1	-1
-1	1	-1	-1	-1

Vertical line
filter

-1	1	1	1	-1
-1	1	-1	1	-1
-1	1	1	1	-1
-1	-1	-1	1	-1
-1	-1	-1	1	-1
-1	-1	1	-1	-1
-1	1	-1	-1	-1

Diagonal line
filter

-1	1	1	1	-1
-1	1	-1	1	-1
-1	1	1	1	-1
-1	-1	-1	1	-1
-1	-1	-1	1	-1
-1	-1	1	-1	-1
-1	1	-1	-1	-1

1	1	1
1	-1	1
1	1	1

$$-1+1+1-1-1-1+1+1 = -1 \rightarrow -1/9 = -0.11$$

-1	1	1	1	-1
-1	1	-1	1	-1
-1	1	1	1	-1
-1	-1	-1	1	-1
-1	-1	-1	1	-1
-1	-1	1	-1	-1
-1	1	-1	-1	-1

*

1	1	1
1	-1	1
1	1	1

-0.11		

-1	1	1	1	-1
-1	1	-1	1	-1
-1	1	1	1	-1
-1	-1	-1	1	-1
-1	-1	-1	1	-1
-1	-1	1	-1	-1
-1	1	-1	-1	-1

*

1	1	1
1	-1	1
1	1	1

-0.11	1	-0.11
-0.55	0.11	-0.33
-0.33	0.33	-0.33
-0.22	-0.11	-0.22
-0.33	-0.33	-0.33

Feature Map

9

6

Loopy pattern
detector

1	1	1
1	-1	1
1	1	1

=

		1

8

Loopy pattern detector

1	1	1
1	-1	1
1	1	1

=

		1
		1

96

Loopy pattern
detector

1	1	1
1	-1	1
1	1	1

=

		1		
			1	
				1

9

Loopy pattern
detector

$$* \begin{array}{|c|c|c|} \hline 1 & 1 & 1 \\ \hline 1 & -1 & 1 \\ \hline 1 & 1 & 1 \\ \hline \end{array} = \begin{array}{|c|c|c|} \hline & & 1 \\ \hline & & \\ \hline & & \\ \hline \end{array}$$

g

Vertical line detector

-1	1	-1
-1	1	-1
-1	1	-1

=

		1

g

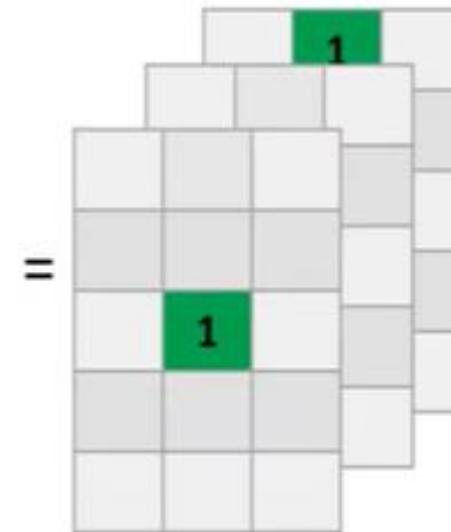
Diagonal line detector

*

Filters

	1	1	1
-1	-1	-1	1
-1	1	-1	1
-1	1	-1	1
-1	1	-1	1

Feature Maps



Filters are nothing
but the feature
detectors



$$\text{eye detector} * \begin{array}{|c|c|c|}\hline & & \\ \hline \end{array} = \boxed{\begin{array}{|c|c|}\hline 1 & 1 \\ \hline \end{array}}$$

Location invariant: It can detect eyes in any location of the image



$$\text{eye detector} * \begin{array}{|c|c|c|}\hline & & \\ \hline \end{array} = \boxed{\begin{array}{|c|c|}\hline 1 & 1 \\ \hline \end{array}}$$



$$\text{eye detector} * \begin{array}{|c|c|c|}\hline & & \\ \hline & & \\ \hline & & \\ \hline \end{array} = \boxed{\begin{array}{|c|c|}\hline 0 & 1 \\ \hline \end{array}}$$

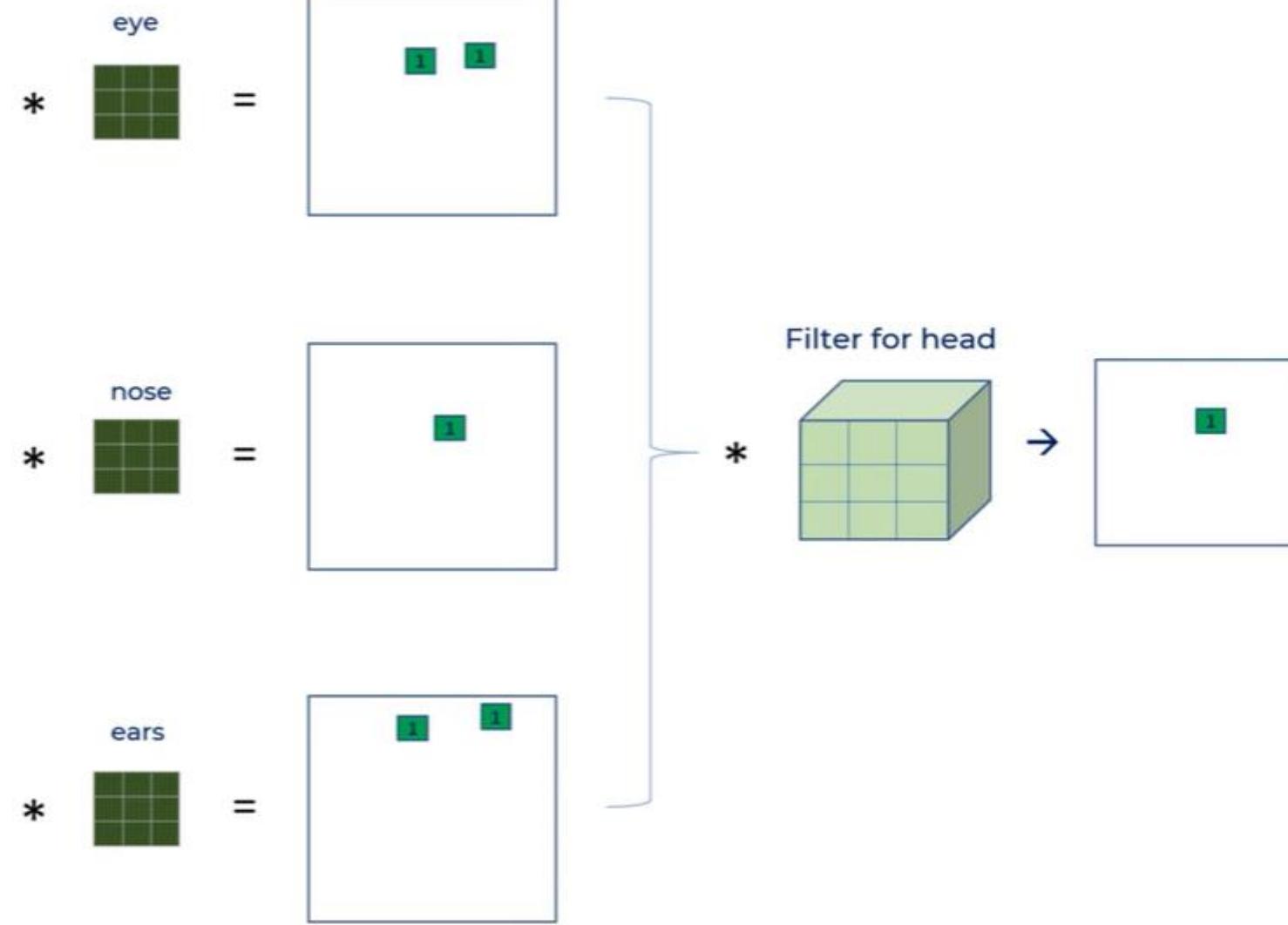
Location invariant: It can detect eyes in any location of the image



$$\text{eye detector} * \begin{array}{|c|c|c|}\hline & & \\ \hline & & \\ \hline & & \\ \hline \end{array} = \boxed{\begin{array}{|c|c|}\hline 1 & 1 \\ \hline \end{array}}$$

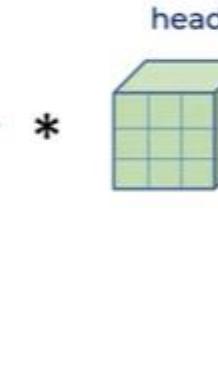


$$\text{eye detector} * \begin{array}{|c|c|c|}\hline & & \\ \hline & & \\ \hline & & \\ \hline \end{array} = \boxed{\begin{array}{|c|c|c|}\hline 1 & 1 & 1 \\ \hline 1 & 1 & 1 \\ \hline \end{array}}$$





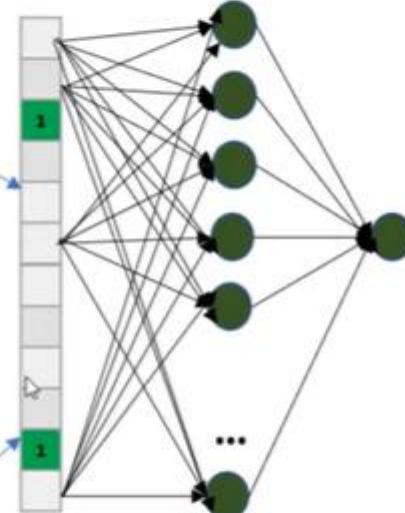
$$\begin{aligned} \text{eye} * \begin{matrix} \text{grid} \\ \text{mask} \end{matrix} &= \boxed{\text{eye}} \\ \text{nose} * \begin{matrix} \text{grid} \\ \text{mask} \end{matrix} &= \boxed{\text{nose}} \\ \text{ears} * \begin{matrix} \text{grid} \\ \text{mask} \end{matrix} &= \boxed{\text{ears}} \end{aligned}$$



\rightarrow



flatten



Is this
Koala?

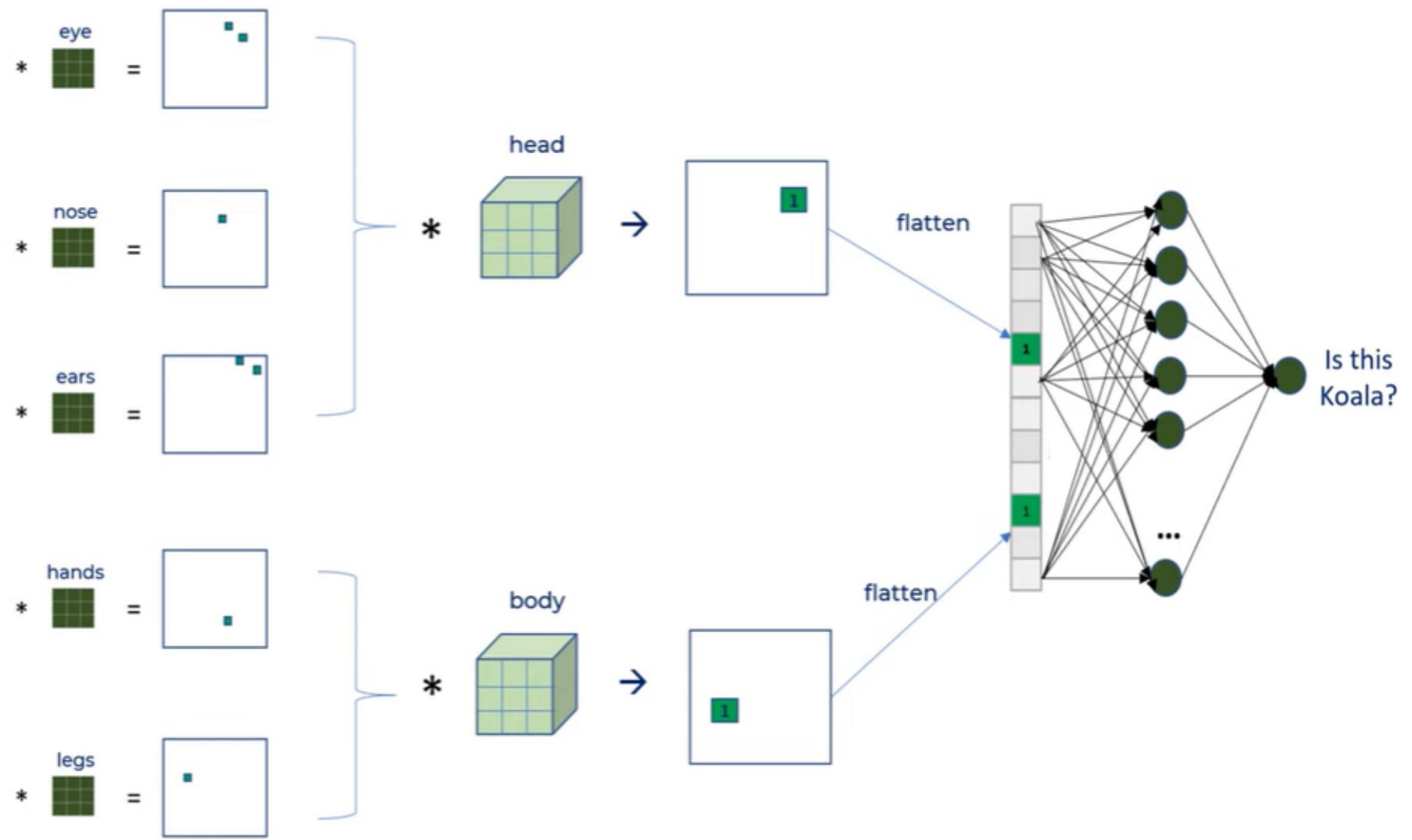


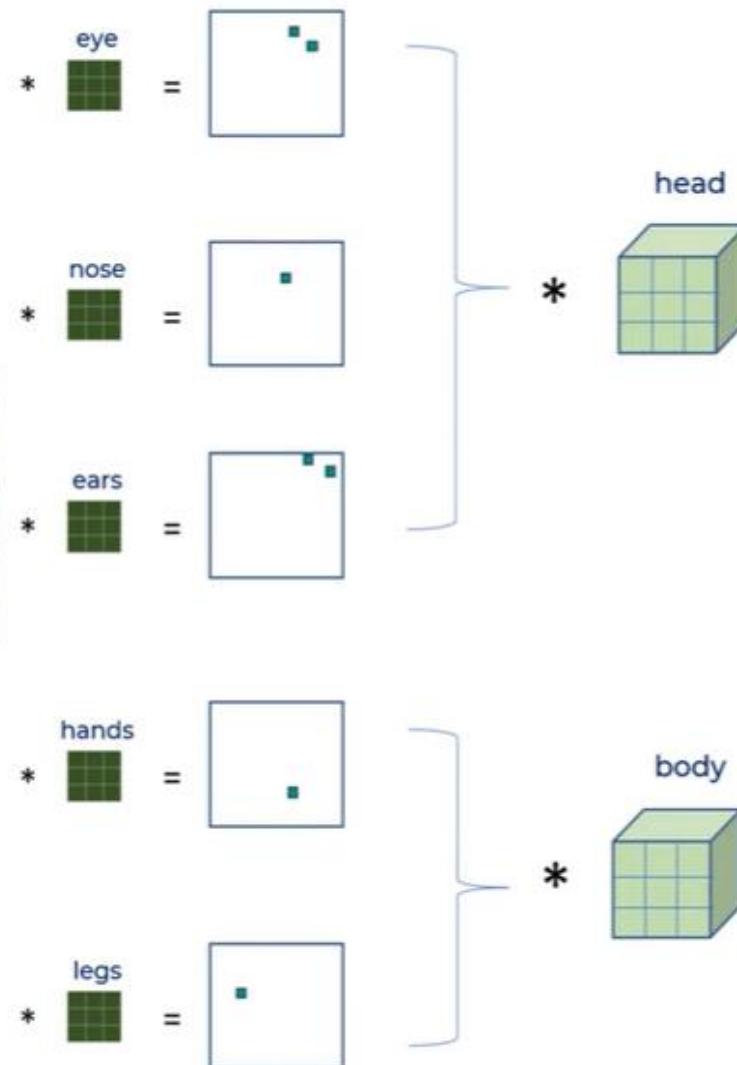
$$\begin{aligned} \text{hands} * \begin{matrix} \text{grid} \\ \text{mask} \end{matrix} &= \boxed{\text{hands}} \\ \text{legs} * \begin{matrix} \text{grid} \\ \text{mask} \end{matrix} &= \boxed{\text{legs}} \end{aligned}$$

\rightarrow



flatten

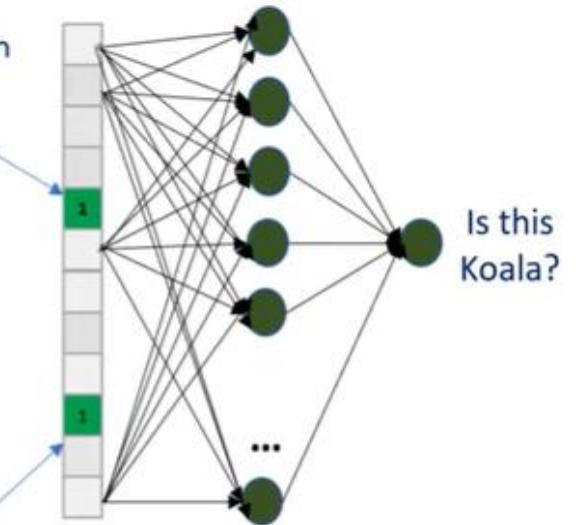




→



flatten



Feature Extraction

Classification

3. Activation Layer (e.g. ReLU):

1. Following the convolutional layer, an activation function is typically applied element-wise to add non-linearity to the network.
2. The **Rectified Linear Unit (ReLU)** is the most common activation function in CNNs. It replaces all negative values with zero, keeping only the positive activations.
3. This layer helps the CNN learn more complex patterns by allowing non-linear combinations of features.

-1	1	1	1	-1
-1	1	-1	1	-1
-1	1	1	1	-1
-1	-1	-1	1	-1
-1	-1	-1	1	-1
-1	-1	1	-1	-1
-1	1	-1	-1	-1

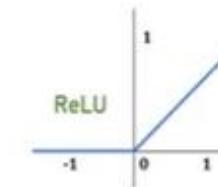
*

Loopy pattern
filter

1	1	1
1	-1	1
1	1	1



-0.11	1	-0.11
-0.55	0.11	-0.33
-0.33	0.33	-0.33
-0.22	-0.11	-0.22
-0.33	-0.33	-0.33



0	1	0
0	0.11	0
0	0.33	0
0	0	0
0	0	0



4. Pooling Layer (e.g., Max Pooling):

1. Pooling layers reduce the spatial dimensions (height and width) of the feature maps, which decreases the number of parameters and computational load.
2. **Max Pooling** is the most common pooling method, where the maximum value within a sliding window (e.g., 2x2) is taken to represent that region.
3. Pooling also adds a degree of translation invariance, making it easier for the network to recognize objects in different positions within the image.

4. Example:

1. For a 2x2 max pooling layer applied to a section like $[1 \ 3 \ 2 \ 4] \begin{bmatrix} 1 & 3 \\ 2 & 4 \end{bmatrix} [1 \ 2 \ 3 \ 4]$, the result is 4.



$$\text{eye} * \begin{array}{|c|c|}\hline \text{green} & \text{green} \\ \hline \end{array} = \boxed{\text{1920} \times 1080}$$



$$\text{nose} * \begin{array}{|c|c|}\hline \text{green} & \text{green} \\ \hline \end{array} = \boxed{\text{1920} \times 1080}$$



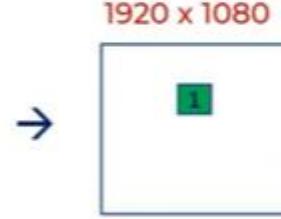
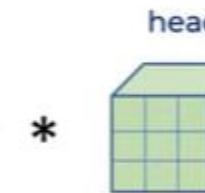
$$\text{ears} * \begin{array}{|c|c|}\hline \text{green} & \text{green} \\ \hline \end{array} = \boxed{\text{1920} \times 1080}$$



$$\text{hands} * \begin{array}{|c|c|}\hline \text{green} & \text{green} \\ \hline \end{array} = \boxed{\text{1920} \times 1080}$$

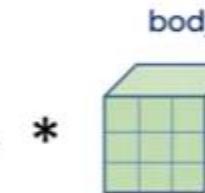


$$\text{legs} * \begin{array}{|c|c|}\hline \text{green} & \text{green} \\ \hline \end{array} = \boxed{\text{1920} \times 1080}$$



flatten

flatten



Is this Koala?

Pooling layer is used to
reduce the size



5	1	3	4
8	2	9	2
1	3	0	1
2	2	2	0

8	9
3	2

2 by 2 filter with stride = 2

0	1	0
0	0.11	0
0	0.33	0
0	0	0
0	0	0

1	

2 by 2 filter with stride = 1

-1	1	1	1	-1
-1	1	-1	1	-1
-1	1	1	1	-1
-1	-1	-1	1	-1
-1	-1	-1	1	-1
-1	-1	1	-1	-1
-1	1	-1	-1	-1

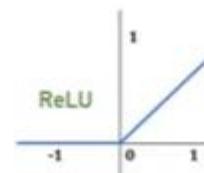
Loopy pattern
filter

1	1	1
1	-1	1
1	1	1

*



-0.11	1	-0.11
-0.55	0.11	-0.33
-0.33	0.33	-0.33
-0.22	-0.11	-0.22
-0.33	-0.33	-0.33



0	1	0
0	0.11	0
0	0.33	0
0	0	0
0	0	0

Max
pooling



1	1
0.33	0.33
0.33	0.33
0	0

Shifted 9 at
different position

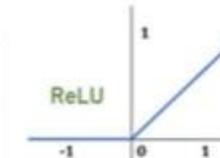
1	1	1	-1	-1
1	-1	1	-1	-1
1	1	1	-1	-1
-1	-1	1	-1	-1
-1	-1	1	-1	-1
-1	1	-1	-1	-1
1	-1	-1	-1	-1

Loopy pattern
filter

1	1	1
1	-1	1
1	1	1

* →

1	-0.11	-0.11
0.11	-0.33	0.33
0.33	-0.33	-0.33
-0.11	-0.55	-0.33
-0.55	-0.33	-0.55



→

1	0	0
0.11	0	0.33
0.33	0	0
0	0	0
0	0	0

Max
pooling
→

1	0.33
0.33	0.33
0.33	0
0	0

There is average pooling also...

5	1	3	4
8	2	9	2
1	3	0	1
2	2	2	0

4	4.5
2	0.75

Benefits of pooling

Reduces dimensions & computation

Reduce overfitting as there are less parameters

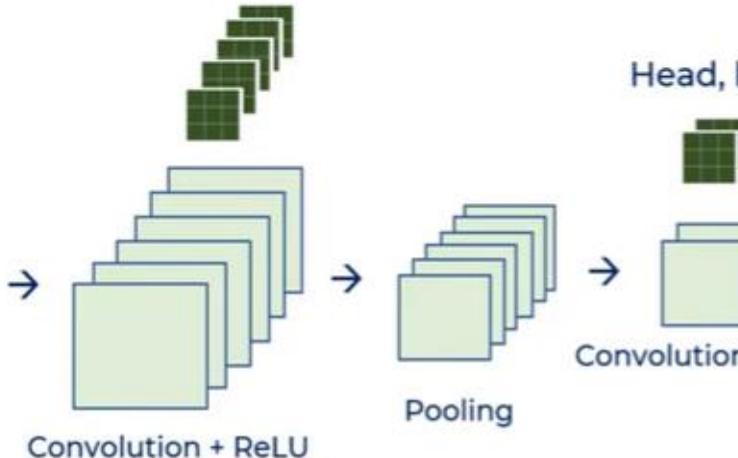
Model is tolerant towards variations, distortions

5. Fully Connected (Dense) Layer:

1. After several convolutional and pooling layers, the feature maps are flattened into a 1D vector and fed into one or more fully connected (dense) layers.
2. These layers learn complex patterns by combining all features learned in previous layers, enabling the network to make final predictions.
3. The fully connected layers treat each feature equally and produce the final output based on the learned weights.



Eye, nose, ears etc



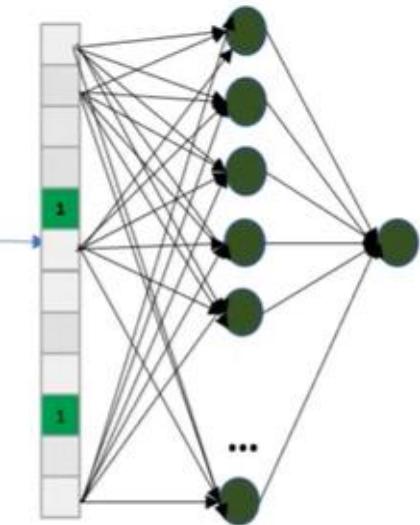
Head, body

...

Convolution + ReLU

Pooling

flatten



Is this
Koala?

4

Feature Extraction

Classification

6. Output Layer:

1. The output layer depends on the task at hand:

1. Classification: A softmax activation function is often used in the output layer for multi-class classification. The softmax outputs a probability distribution over classes, with each value representing the probability of the input belonging to a specific class.

2. Regression: For regression tasks, the output layer usually has a single neuron without activation (or sometimes with linear activation) to predict a continuous value.

Convolution

- Connections sparsity reduces overfitting
- Conv + Pooling gives location invariant feature detection
- Parameter sharing

ReLU

- Introduces nonlinearity
- Speeds up training, faster to compute

Pooling

- Reduces dimensions and computation
- Reduces overfitting
- Makes the model tolerant towards small distortion and variations



KESHAV MEMORIAL INSTITUTE OF TECHNOLOGY
AN AUTONOMOUS INSTITUTION - ACCREDITED BY NAAC WITH 'A' GRADE
Narayananaguda, Hyderabad.

Deep Learning

CNN
EXERCISE-1
18-11-2024

BY
ASHA

EXERCISE1: 3* 3 convolution on 5* 5 image with and without padding

In this task, you will apply a 3x3 convolution on a 5x5 image matrix using a given filter. This will involve performing the convolution operation with and without padding to understand how padding affects the output dimensions.

Follow the steps below to complete the task:

1. Import necessary libraries (Numpy)
2. Define the 5x5 image matrix
3. Define the 3x3 filter (kernel) matrix
4. Convolution Operation
5. Convolution Operation Without padding
6. Convolution Operation With padding of 1

2. Define the 5x5 image matrix

The 5×5 image matrix is a numerical representation of an image where each value corresponds to the intensity of a pixel.

$$\text{image} = \begin{bmatrix} 1 & 2 & 3 & 0 & 1 \\ 4 & 5 & 6 & 1 & 2 \\ 7 & 8 & 9 & 2 & 3 \\ 1 & 2 & 3 & 0 & 1 \\ 4 & 5 & 6 & 1 & 2 \end{bmatrix}$$

```
# Define the 5x5 image matrix
image = np.array([
    [1, 2, 3, 0, 1],
    [4, 5, 6, 1, 2],
    [7, 8, 9, 2, 3],
    [1, 2, 3, 0, 1],
    [4, 5, 6, 1, 2]
])
```

3. Define the 3x3 filter (kernel) matrix

The 3x3 filter (kernel) matrix is used for feature extraction in an image. It works by performing a convolution operation, where it slides over an image and calculates a weighted sum of pixel intensities at each position.

$$\text{filter_kernel} = \begin{bmatrix} 1 & 0 & -1 \\ 1 & 0 & -1 \\ 1 & 0 & -1 \end{bmatrix}$$

```
# Define the 3x3 filter (kernel) matrix
filter_kernel = np.array([
    [1, 0, -1],
    [1, 0, -1],
    [1, 0, -1]
])
```

4. Convolution Operation

Write a Python function named convolve to perform a 2D convolution operation on a grayscale image. The function should:

Take three arguments:

image: A 2D NumPy array representing a grayscale image.

kernel: A 2D NumPy array representing the convolution filter (kernel).

padding: An integer specifying the number of zero-padded rows/columns to add around the image (default is 0).

Return:

A 2D NumPy array representing the convolved output.

Function Signature:

```
def convolve(image, kernel, padding=0):  
    #your code here  
    return output
```

Convolution function steps

1. Add padding to the image if needed
2. Dimensions of the image and kernel
3. Calculate output dimensions
4. Initialize the output matrix
5. Perform convolution
6. return output

The function should implement a **convolution operation** on a 2D image matrix using a given kernel (filter).

1. Input Parameters

1.image: A 2D array representing the input image.

- Example: A 5×5 $\times 5$ matrix of pixel values.

2.kernel: A smaller 2D array representing the filter to be applied.

- Example: A 3×3 $\times 3$ matrix for edge detection.

3.padding: Specifies how many layers of zero-padding to add around the image.

- Default is 0 (no padding).

```
# Convolution function
def convolve(image, kernel, padding=0):
```

2. Steps in the Function

Step 1: Add Padding

- **Purpose:** Padding adds a border around the original image, allowing the kernel to process edge pixels.

- **np.pad:**

- Adds padding layers of zeros around the image.
- The mode='constant' ensures zeros are added.

```
if padding > 0:  
    image = np.pad(image, ((padding, padding), (padding, padding)), mode='constant')
```

Step 2: Extract Dimensions

- Extract the height and width of the image and kernel to calculate the output dimensions.

```
# Dimensions of the image and kernel  
image_height, image_width = image.shape  
kernel_height, kernel_width = kernel.shape
```

Step 3: Calculate Output Dimensions

- **Formula**

$$\text{Output Height} = \text{Image Height} - \text{Kernel Height} + 1$$

$$\text{Output Width} = \text{Image Width} - \text{Kernel Width} + 1$$

- **Why?**: The kernel only fits within certain positions in the image, reducing the output size.
- **With Padding**: If padding is applied, it compensates for the size reduction.

```
# Calculate output dimensions
output_height = image_height - kernel_height + 1
output_width = image_width - kernel_width + 1
```

Step 4: Initialize the Output Matrix

- Creates a zero matrix with the calculated output dimensions.
- This matrix will store the result of the convolution operation.

```
# Initialize the output matrix  
output = np.zeros((output_height, output_width), dtype=int)
```

Step 5: Perform Convolution

1. Loop Over the Output Matrix:

- i and j iterate over each position in the output matrix.

2. Extract a Region:

- `region = image[i:i+kernel_height, j:j+kernel_width]`: Extracts a submatrix of the image the same size as the kernel.

3. Element-Wise Multiplication:

- `region * kernel`: Multiplies each element of the kernel with the corresponding element of the region.

4. Sum the Result:

- `np.sum(region * kernel)`: Sums all the products to produce a single value for the current position.

```
# Perform convolution
for i in range(output_height):
    for j in range(output_width):
        region = image[i:i+kernel_height, j:j+kernel_width]
        output[i, j] = np.sum(region * kernel)
```

```
# Convolution function
def convolve(image, kernel, padding=0):
    # Add padding to the image if needed
    if padding > 0:
        image = np.pad(image, ((padding, padding), (padding, padding)), mode='constant')
    #
    # Dimensions of the image and kernel
    image_height, image_width = image.shape
    kernel_height, kernel_width = kernel.shape

    # Calculate output dimensions
    output_height = image_height - kernel_height + 1
    output_width = image_width - kernel_width + 1

    # Initialize the output matrix
    output = np.zeros((output_height, output_width), dtype=int)

    # Perform convolution
    for i in range(output_height):
        for j in range(output_width):
            region = image[i:i+kernel_height, j:j+kernel_width]
            output[i, j] = np.sum(region * kernel)

    return output
```

5.Convolution Operation Without padding

compute the **convolution** of the given 5×5 image with the 3×3 filter (kernel) **without padding**.

```
# Without padding
output_without_padding = convolve(image, filter_kernel, padding=0)
print("Output without padding:")
print(output_without_padding)
```

6.Convolution Operation With padding of 1

```
# With padding of 1
output_with_padding = convolve(image, filter_kernel, padding=1)
print("\nOutput with padding:")
print(output_with_padding)
```

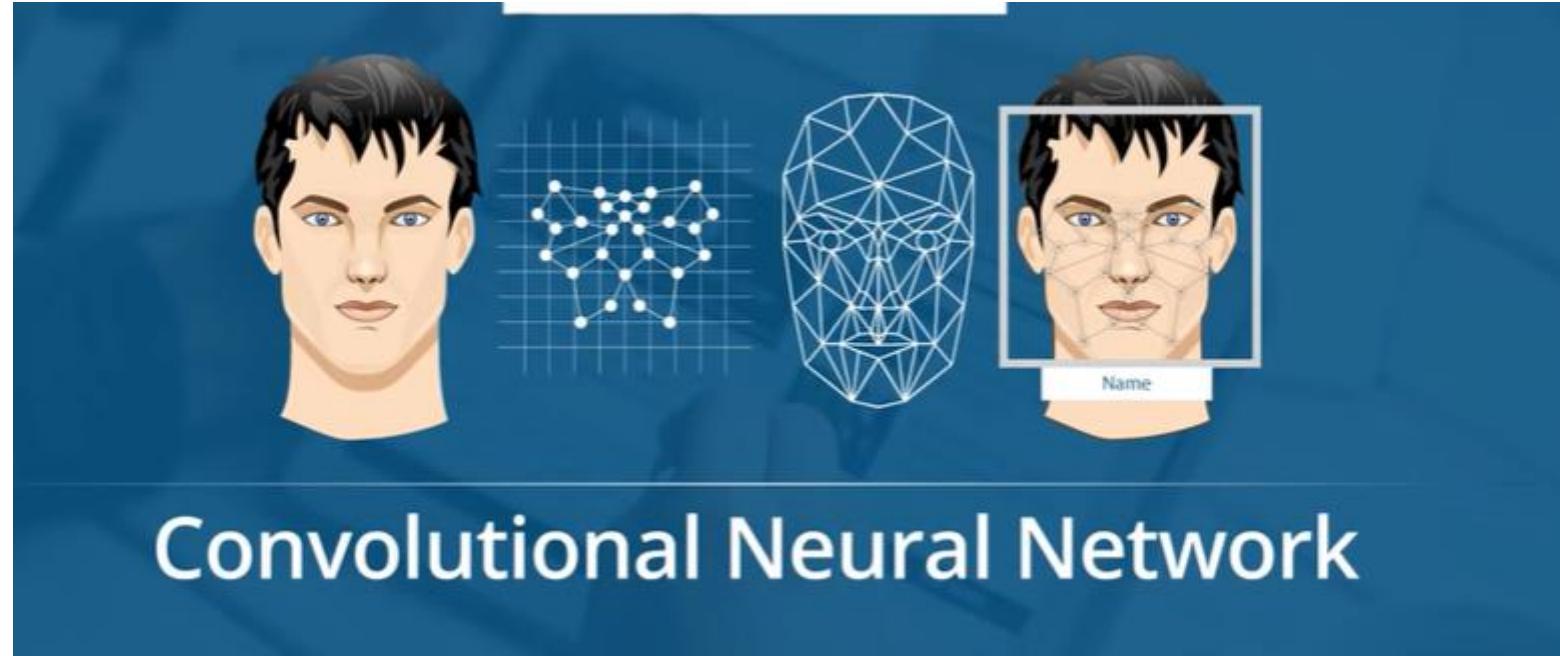


KESHAV MEMORIAL INSTITUTE OF TECHNOLOGY
AN AUTONOMOUS INSTITUTION - ACCREDITED BY NAAC WITH 'A' GRADE
Narayanauguda, Hyderabad.

Deep Learning

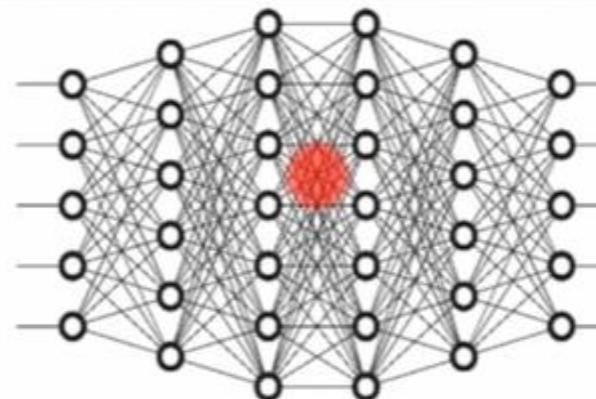
**CNN
SESSION2
20-11-2024**

**BY
ASHA**



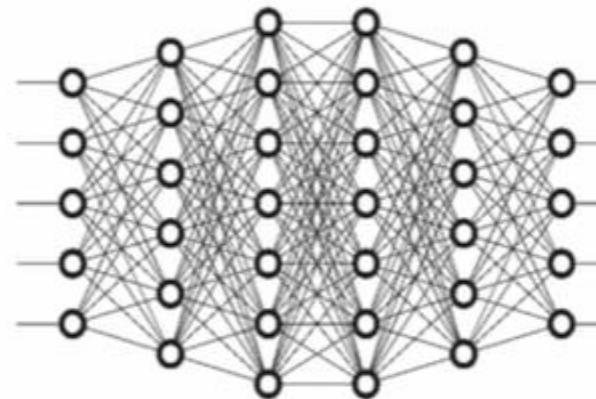
Why Not Fully Connected Networks

Image with
 $28 \times 28 \times 3$
pixels



*Number of weights in
the first hidden layer
will be 2352*

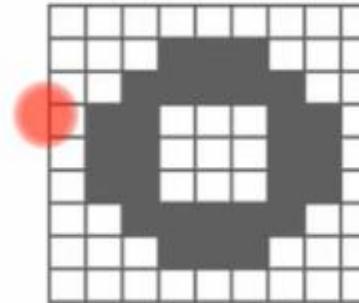
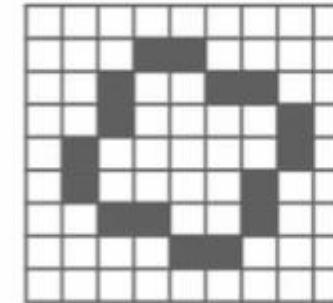
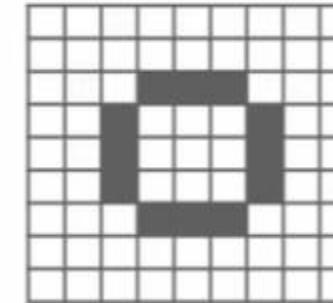
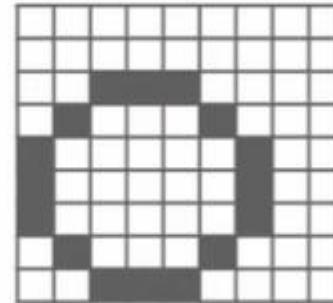
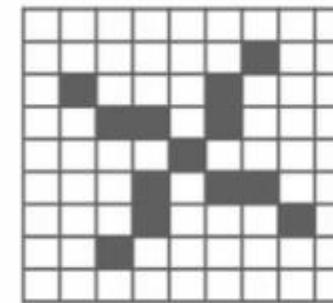
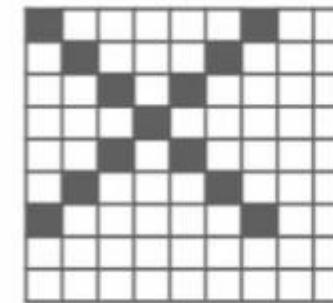
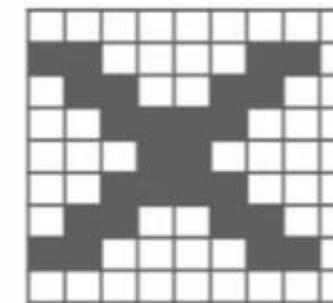
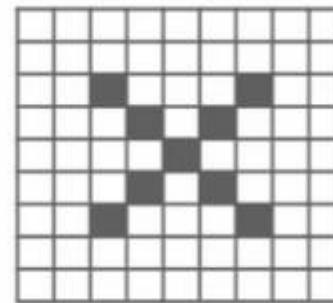
Image with
 $200 \times 200 \times 3$
pixels



*Number of weights in
the first hidden layer
will be 120,000*

Trickier Case

Here, we will have some problems, because X and O images won't always have the same images. There can be certain deformations. Consider the diagrams shown below:



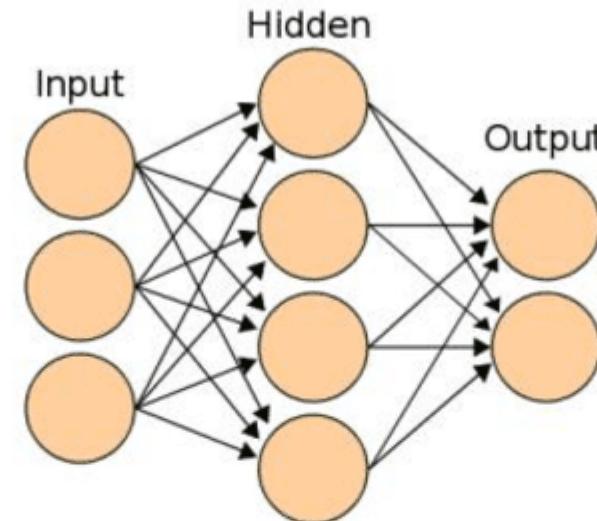
Disadvantages of using ANN for image classification

1. Too much computation
2. Treats local pixels same as pixels far apart
3. Sensitive to location of an object in an image

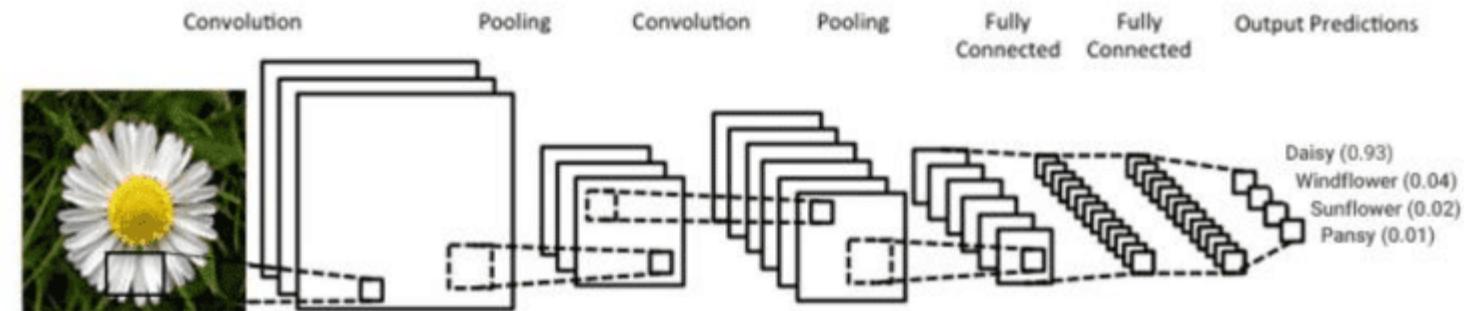
- A **convolutional neural network (CNN)**, is a network architecture for deep learning which learns directly from data.
- CNNs are particularly useful for finding patterns in images to recognize objects.
- They can also be quite effective for classifying non-image data such as audio, time series, and signal data.

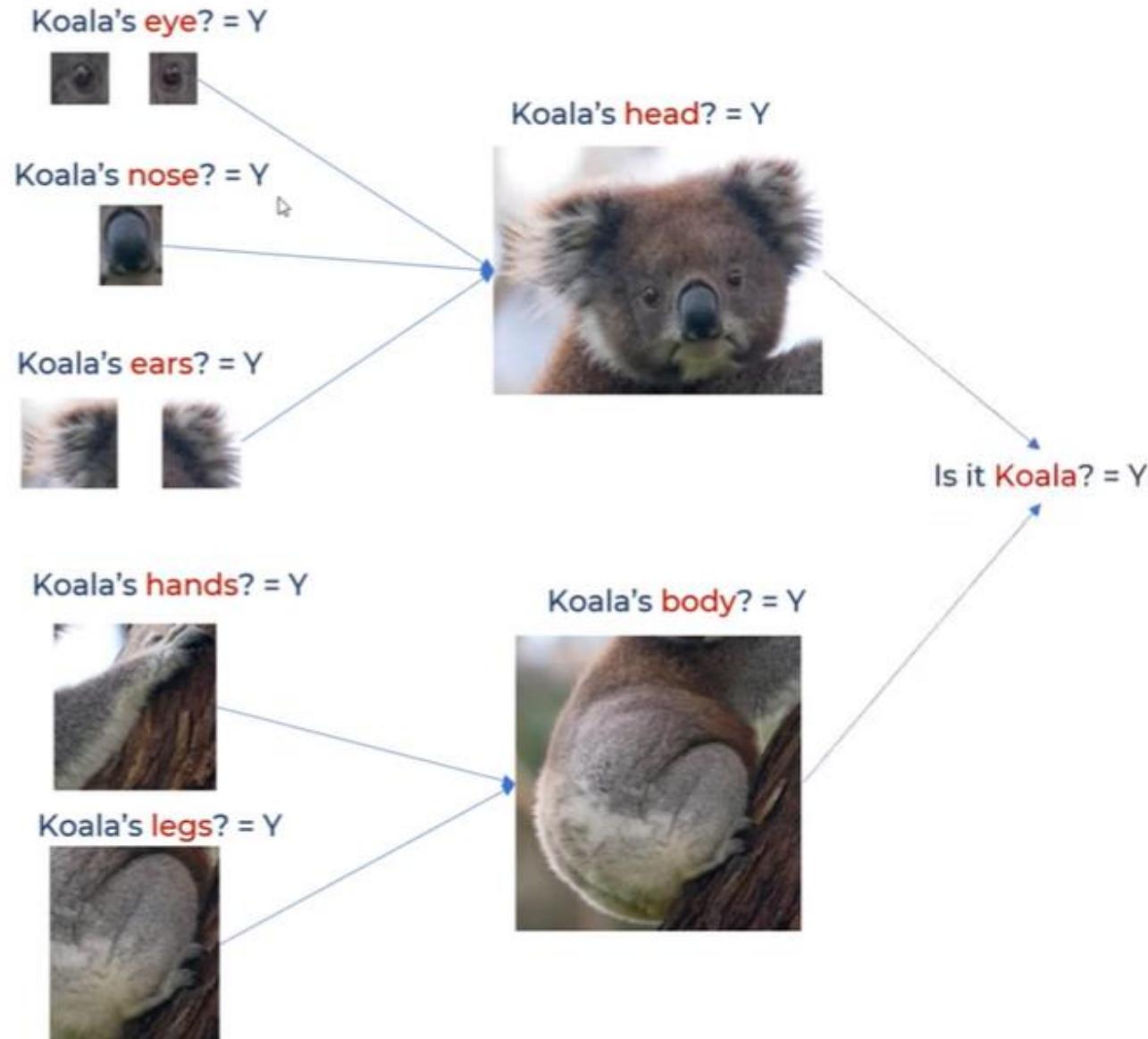
Convolutional Neural Networks vs. Artificial Neural Networks

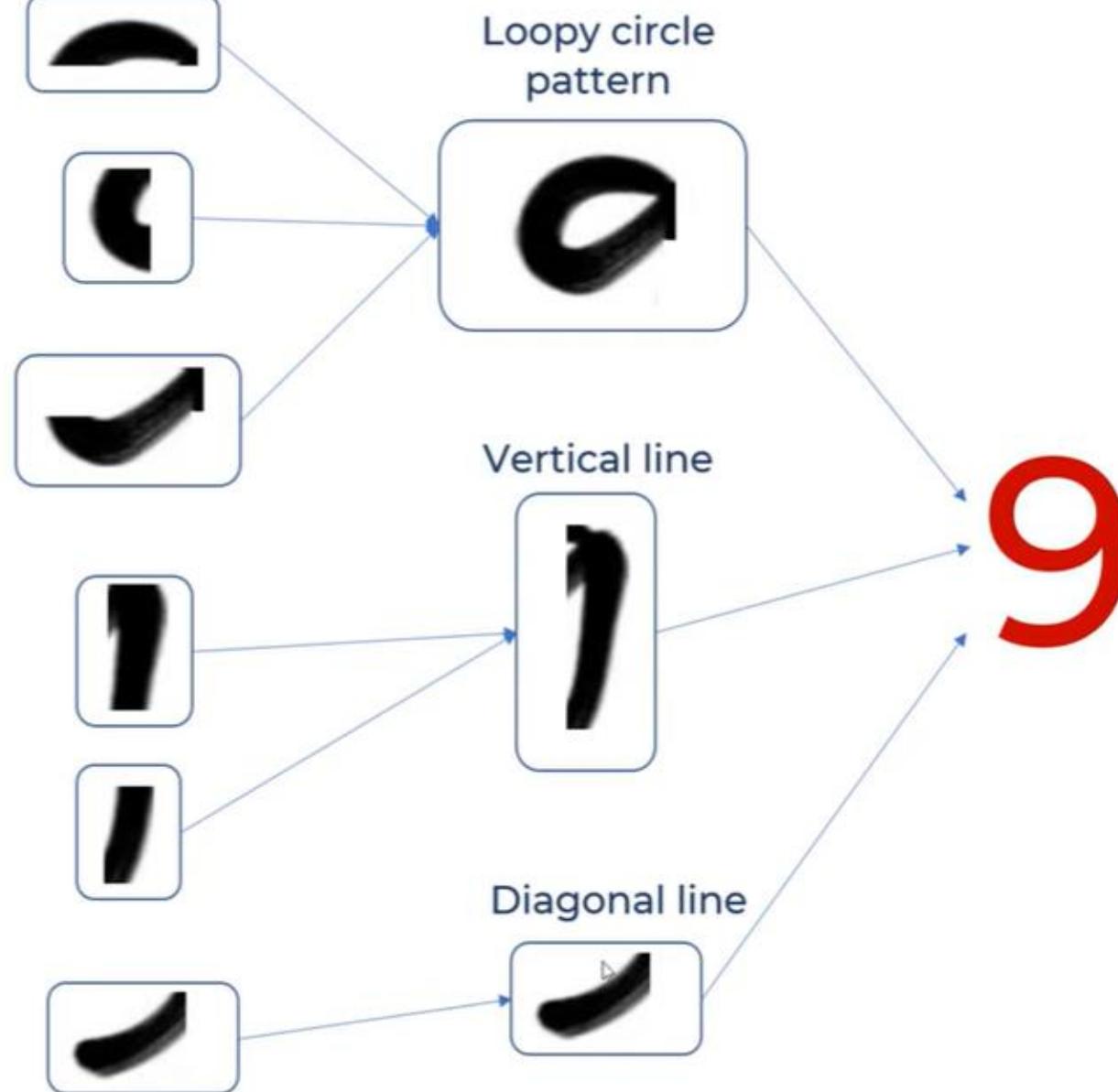
Artificial Neural Network (ANN)



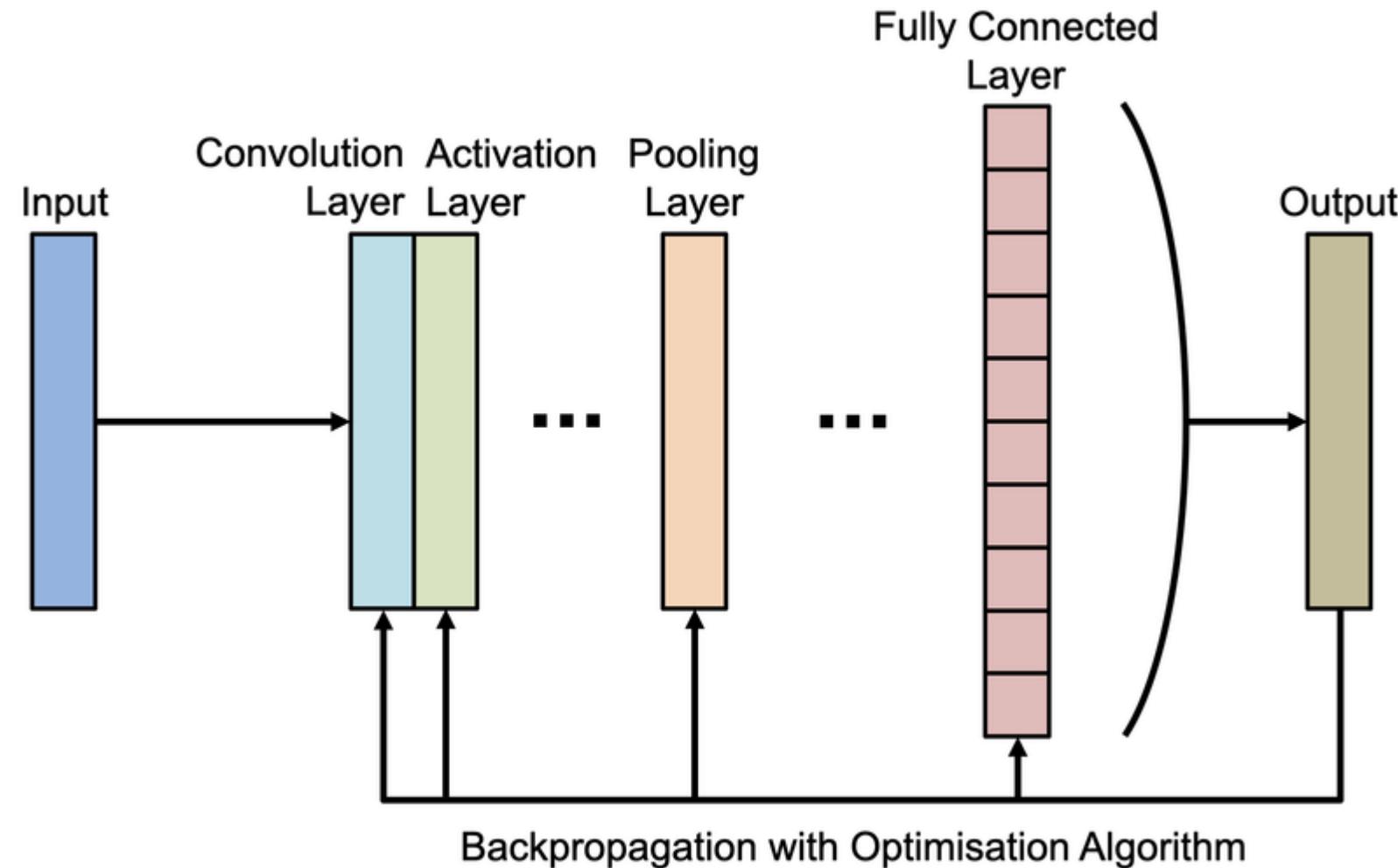
Convolutional Neural Network (CNN)

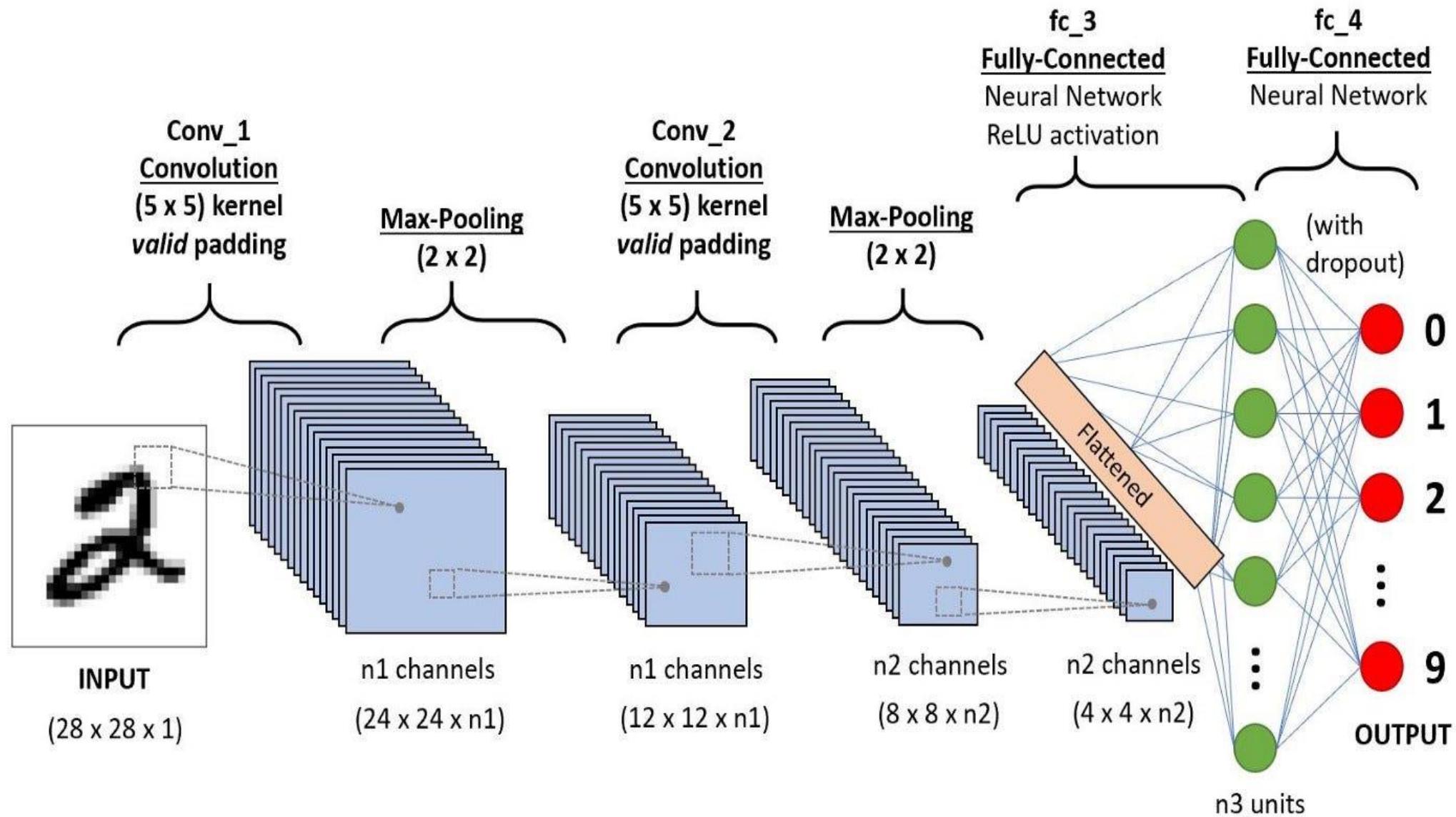






The standard architecture of a CNN is composed of a series of layers, each serving a specific purpose.



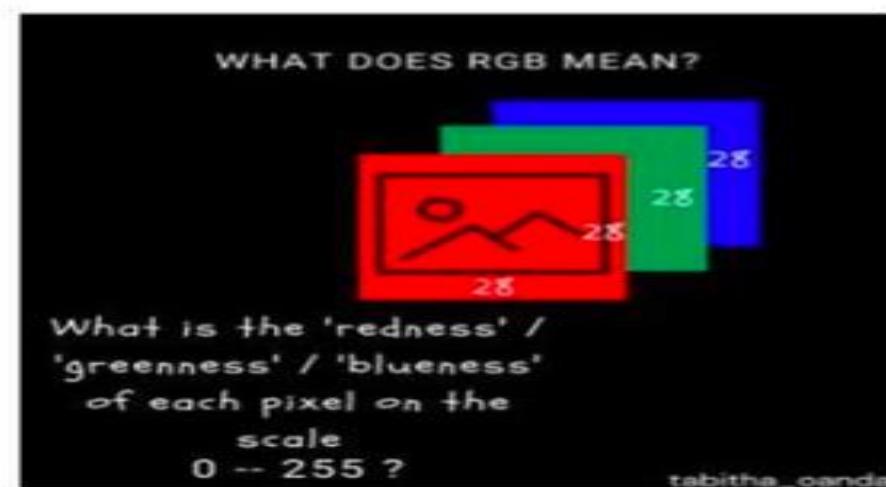
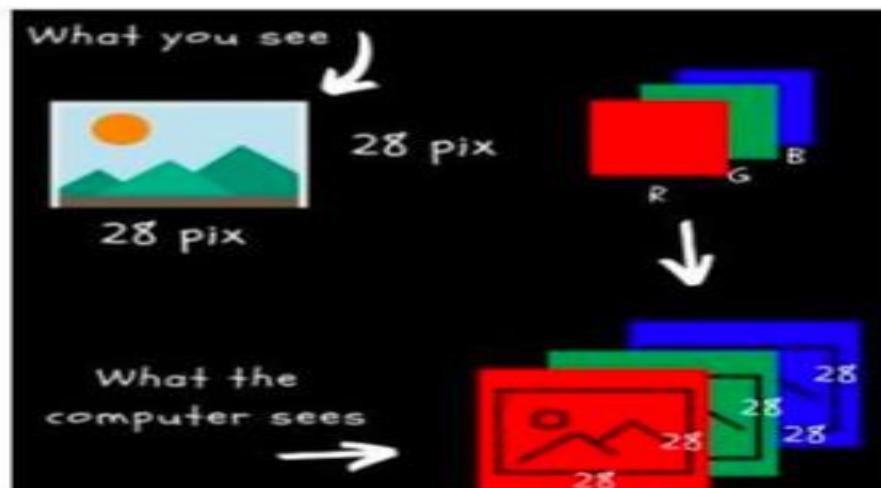


Basic Layers in a CNN Architecture

1. Input Layer
2. Convolutional Layer
3. Activation Layer (e.g., ReLU)
4. Pooling Layer (e.g., Max Pooling)
5. Fully Connected (Dense) Layer
6. Output Layer

1. Input Layer:

1. This layer holds the raw pixel values of the image, typically in the form of a 3D matrix (height x width x channels).
2. For instance, an RGB image with a size of 28x28 pixels would have an input shape of $28 \times 28 \times 3$, where 3 represents the Red, Green, and Blue color channels.



2. Convolutional Layer:

1. The core layer in CNNs, this layer applies a convolutional filters (or kernels) over the input image to detect patterns.
2. Each filter slides (convolves) over the image and computes a "dot product" between the filter and a small section of the input, generating a feature map.
3. Each filter is designed to detect specific patterns like edges, textures, or shapes.

Source layer

5	2	6		8	2	0	1	2
4	3	4		5	1	9	6	3
3	9	2		4	7	7	6	9
1	3	4		6	8	2	2	1
8	4	6	2	3	1	8	8	
5	8	9	0	1	0	2	3	
9	2	6	6	3	6	2	1	
9	8	8	2	6	3	4	5	

Convolutional kernel

-1	0	1
2	1	2
1	-2	0

Destination layer

$$\begin{aligned} & (-1 \times 5) + (0 \times 2) + (1 \times 6) + \\ & (2 \times 4) + (1 \times 3) + (2 \times 4) + \\ & (1 \times 3) + (-2 \times 9) + (0 \times 2) = 5 \end{aligned}$$

To apply the convolution:

- Overlay the Kernel on the Image: Start from the top-left corner of the image and place the kernel so that its center aligns with the current image pixel.
- Element-wise Multiplication: Multiply each element of the kernel with the corresponding element of the image it covers.
- Summation: Sum up all the products obtained from the element-wise multiplication. This sum forms a single pixel in the output feature map.
- Continue the Process: Slide the kernel over to the next pixel and repeat the process across the entire image.

Step-1

1	0	-2	1
-1	0	1	2
0	2	1	0
1	0	0	1



0	1
-1	2



1		

Step-2

1	0	-2	1
-1	0	1	2
0	2	1	0
1	0	0	1



0	1
-1	2



1	0	

Step-3

1	0	-2	1
-1	0	1	2
0	2	1	0
1	0	0	1



0	1
-1	2



1	0	4

Step-4

1	0	-2	1
-1	0	1	2
0	2	1	0
1	0	0	1



0	1
-1	2



1	0	4
4		

Step-5

1	0	-2	1
-1	0	1	2
0	2	1	0
1	0	0	1



0	1
-1	2



1	0	4
4	1	

1. Hyperparameters:

1. Filter Size (Kernel Size): Usually 3x3 or 5x5.

2. Stride: Determines the step size as the filter slides over the input.

3. Padding: Controls the spatial dimensions of the output (adding "zero-padding" can preserve input size).

2. Output: The output of a convolutional layer is a set of **feature maps**, each representing the presence of a specific feature across the spatial dimensions.

Filters are nothing
but the feature
detectors



$$\text{eye detector} * \begin{array}{|c|c|c|}\hline & & \\ \hline & & \\ \hline & & \\ \hline \end{array} = \boxed{\begin{array}{|c|c|}\hline 2 & 1 \\ \hline \end{array}}$$

Location invariant: It can detect eyes in any location of the image



$$\text{eye detector} * \begin{array}{|c|c|c|}\hline & & \\ \hline & & \\ \hline & & \\ \hline \end{array} = \boxed{\begin{array}{|c|c|}\hline 1 & 1 \\ \hline \end{array}}$$



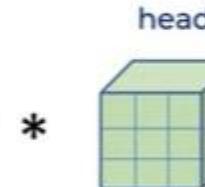
$$\text{eye detector} * \begin{array}{|c|c|c|}\hline & & \\ \hline & & \\ \hline & & \\ \hline \end{array} = \boxed{\begin{array}{|c|c|c|c|}\hline 1 & & 2 & 2 \\ \hline & & 2 & 2 \\ \hline & & 1 & 1 \\ \hline \end{array}}$$



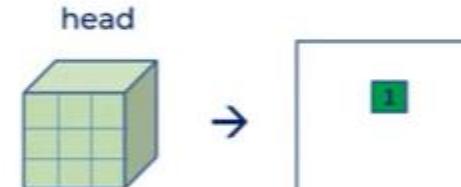
$$\begin{array}{l} \text{eye} \\ * \quad \begin{matrix} \text{grid} \end{matrix} = \boxed{\text{eye}} \end{array}$$

$$\begin{array}{l} \text{nose} \\ * \quad \begin{matrix} \text{grid} \end{matrix} = \boxed{\text{nose}} \end{array}$$

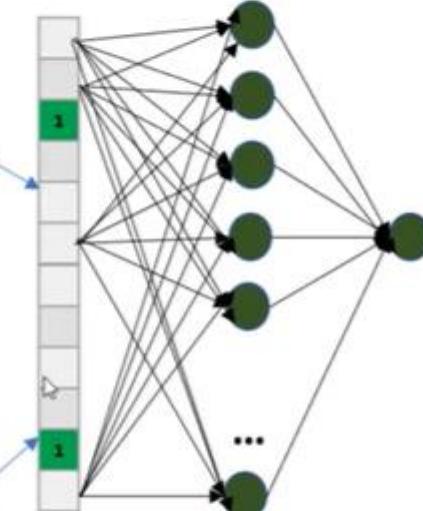
$$\begin{array}{l} \text{ears} \\ * \quad \begin{matrix} \text{grid} \end{matrix} = \boxed{\text{ears}} \end{array}$$



* →



flatten



Is this
Koala?



$$\begin{array}{l} \text{hands} \\ * \quad \begin{matrix} \text{grid} \end{matrix} = \boxed{\text{hands}} \end{array}$$

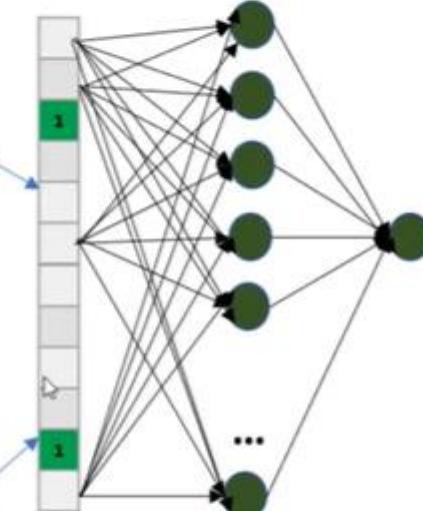
$$\begin{array}{l} \text{legs} \\ * \quad \begin{matrix} \text{grid} \end{matrix} = \boxed{\text{legs}} \end{array}$$

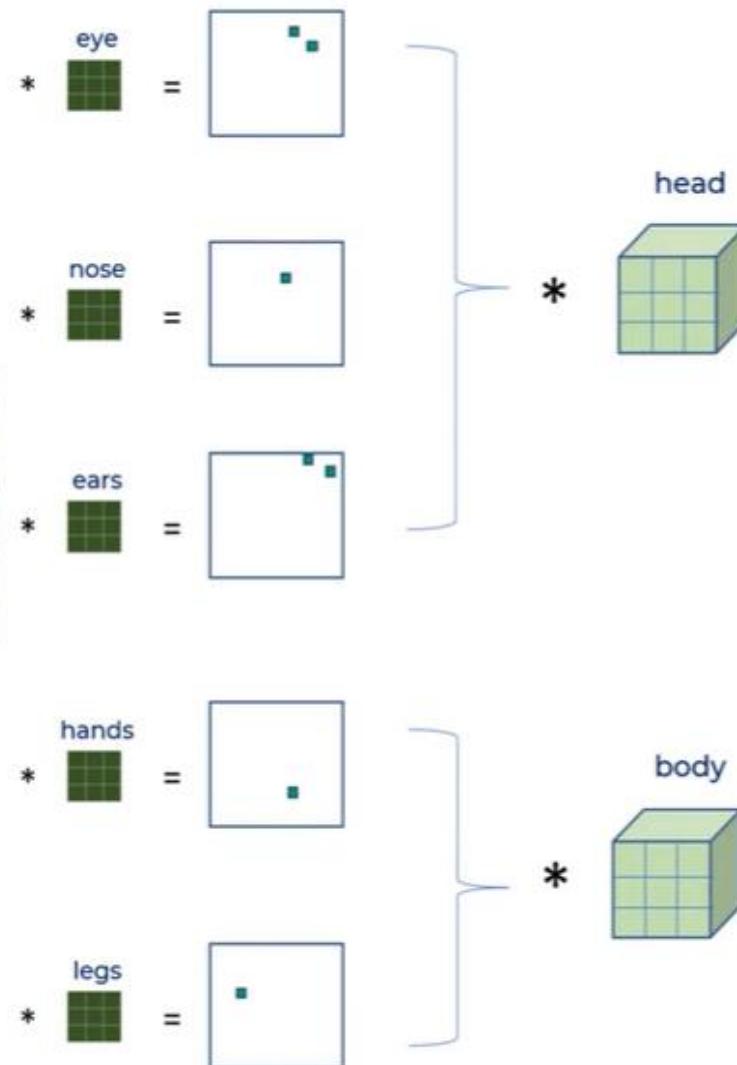


* →



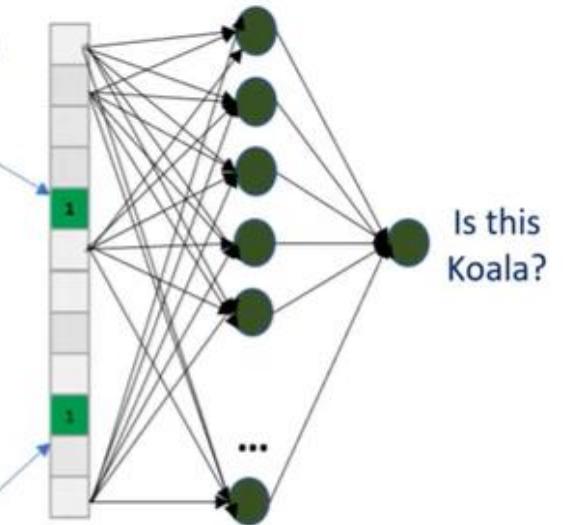
flatten





flatten

flatten



Feature Extraction

Classification

3. Activation Layer (e.g. ReLU):

1. Following the convolutional layer, an activation function is typically applied element-wise to add non-linearity to the network.
2. The **Rectified Linear Unit (ReLU)** is the most common activation function in CNNs. It replaces all negative values with zero, keeping only the positive activations.
3. This layer helps the CNN learn more complex patterns by allowing non-linear combinations of features.

-1	1	1	1	-1
-1	1	-1	1	-1
-1	1	1	1	-1
-1	-1	-1	1	-1
-1	-1	-1	1	-1
-1	-1	1	-1	-1
-1	1	-1	-1	-1

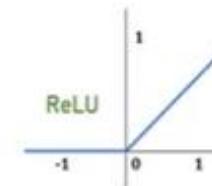
*

Loopy pattern
filter

1	1	1
1	-1	1
1	1	1



-0.11	1	-0.11
-0.55	0.11	-0.33
-0.33	0.33	-0.33
-0.22	-0.11	-0.22
-0.33	-0.33	-0.33



0	1	0
0	0.11	0
0	0.33	0
0	0	0
0	0	0



4. Pooling Layer (e.g., Max Pooling):

1. Pooling layers reduce the spatial dimensions (height and width) of the feature maps, which decreases the number of parameters and computational load.
2. **Max Pooling** is the most common pooling method, where the maximum value within a sliding window (e.g., 2x2) is taken to represent that region.
3. Pooling also adds a degree of translation invariance, making it easier for the network to recognize objects in different positions within the image.

4. Example:

1. For a 2x2 max pooling layer applied to a section like $[1 \ 3 \ 2 \ 4] \begin{bmatrix} 1 & 3 \\ 2 & 4 \end{bmatrix} [1 \ 2 \ 3 \ 4]$, the result is 4.



$$\text{eye} * \begin{array}{|c|c|}\hline \text{green} & \text{green} \\ \hline \end{array} = \boxed{\text{1920} \times 1080}$$



$$\text{nose} * \begin{array}{|c|c|}\hline \text{green} & \text{green} \\ \hline \end{array} = \boxed{\text{1920} \times 1080}$$



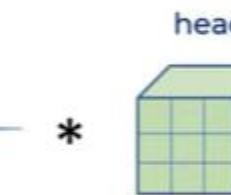
$$\text{ears} * \begin{array}{|c|c|}\hline \text{green} & \text{green} \\ \hline \end{array} = \boxed{\text{1920} \times 1080}$$



$$\text{hands} * \begin{array}{|c|c|}\hline \text{green} & \text{green} \\ \hline \end{array} = \boxed{\text{1920} \times 1080}$$



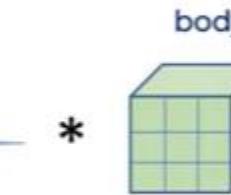
$$\text{legs} * \begin{array}{|c|c|}\hline \text{green} & \text{green} \\ \hline \end{array} = \boxed{\text{1920} \times 1080}$$



flatten



Is this Koala?



flatten



Pooling layer is used to
reduce the size



Shifted 9 at
different position

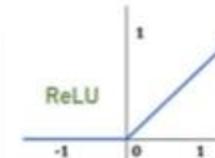
1	1	1	-1	-1
1	-1	1	-1	-1
1	1	1	-1	-1
-1	-1	1	-1	-1
-1	-1	1	-1	-1
-1	1	-1	-1	-1
1	-1	-1	-1	-1

Loopy pattern
filter

1	1	1
1	-1	1
1	1	1

* →

1	-0.11	-0.11
0.11	-0.33	0.33
0.33	-0.33	-0.33
-0.11	-0.55	-0.33
-0.55	-0.33	-0.55



→

1	0	0
0.11	0	0.33
0.33	0	0
0	0	0
0	0	0

Max
pooling
→

1	0.33
0.33	0.33
0.33	0
0	0

Benefits of pooling

Reduces dimensions & computation

Reduce overfitting as there are less parameters

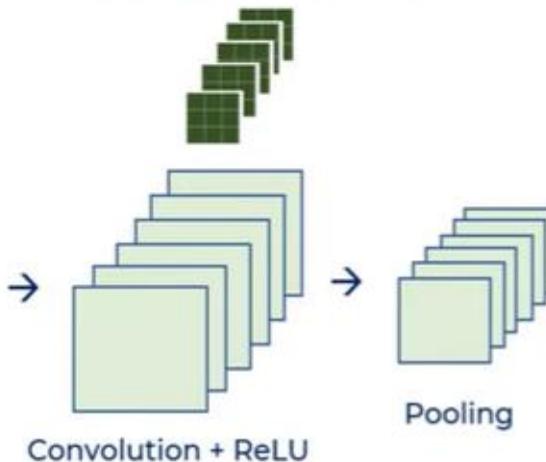
Model is tolerant towards variations, distortions

5. Fully Connected (Dense) Layer:

1. After several convolutional and pooling layers, the feature maps are flattened into a 1D vector and fed into one or more fully connected (dense) layers.
2. These layers learn complex patterns by combining all features learned in previous layers, enabling the network to make final predictions.
3. The fully connected layers treat each feature equally and produce the final output based on the learned weights.



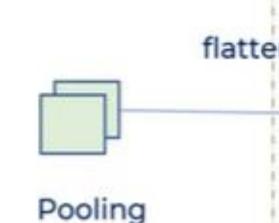
Eye, nose, ears etc



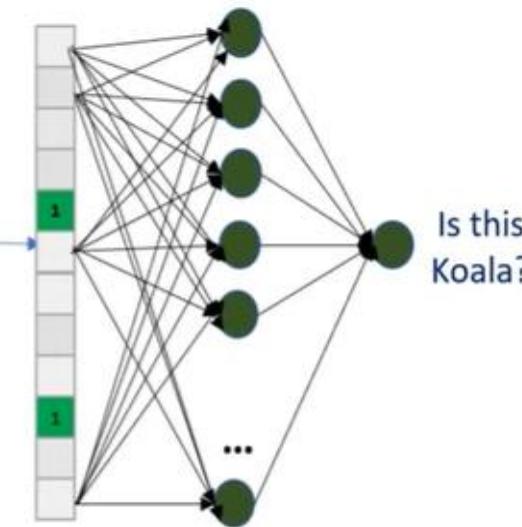
Head, body

Pooling

Convolution + ReLU



flatten



Feature Extraction

Classification

6. Output Layer:

1. The output layer depends on the task at hand:

1. Classification: A softmax activation function is often used in the output layer for multi-class classification. The softmax outputs a probability distribution over classes, with each value representing the probability of the input belonging to a specific class.

2. Regression: For regression tasks, the output layer usually has a single neuron without activation (or sometimes with linear activation) to predict a continuous value.



KESHAV MEMORIAL INSTITUTE OF TECHNOLOGY
AN AUTONOMOUS INSTITUTION - ACCREDITED BY NAAC WITH 'A' GRADE
Narayananaguda, Hyderabad.

Deep Learning

CNN
EXERCISE-1
18-11-2024

BY
ASHA

EXERCISE 2: 3* 3 convolution on 5* 5 image with and without padding

1.Import necessary libraries

```
from PIL import Image  
import numpy as np  
import matplotlib.pyplot as plt
```

2.Load the cat image and perform image Preprocessing

```
# Load the cat image and convert it to grayscale
image_path = 'img.jpeg' # replace with the path to your cat image
new_image = Image.open(image_path).convert('L') # Convert to grayscale
# Convert the image to a numpy array
image_array = np.array(new_image)
```

3. Display the original image

```
# Display the original image
plt.imshow(image_array, cmap='gray')
plt.title("Original Image")
plt.axis('off')
plt.show()
```

Original Image



4.Define the 3x3 filter (kernel) matrix

```
]: # Define the 3x3 filter (kernel) matrix  
filter_kernel = np.array([  
    [1, 0, -1],  
    [1, 0, -1],  
    [1, 0, -1]  
)
```

5. Define Convolution Function

```
# Convolution function
def convolve(image, kernel, padding=0):
    # Add padding if needed
    if padding > 0:
        image = np.pad(image, ((padding, padding), (padding, padding)), mode='constant')

    # Dimensions of the image and kernel
    image_height, image_width = image.shape
    kernel_height, kernel_width = kernel.shape

    # Calculate output dimensions
    output_height = image_height - kernel_height + 1
    output_width = image_width - kernel_width + 1

    # Initialize the output matrix
    output = np.zeros((output_height, output_width))

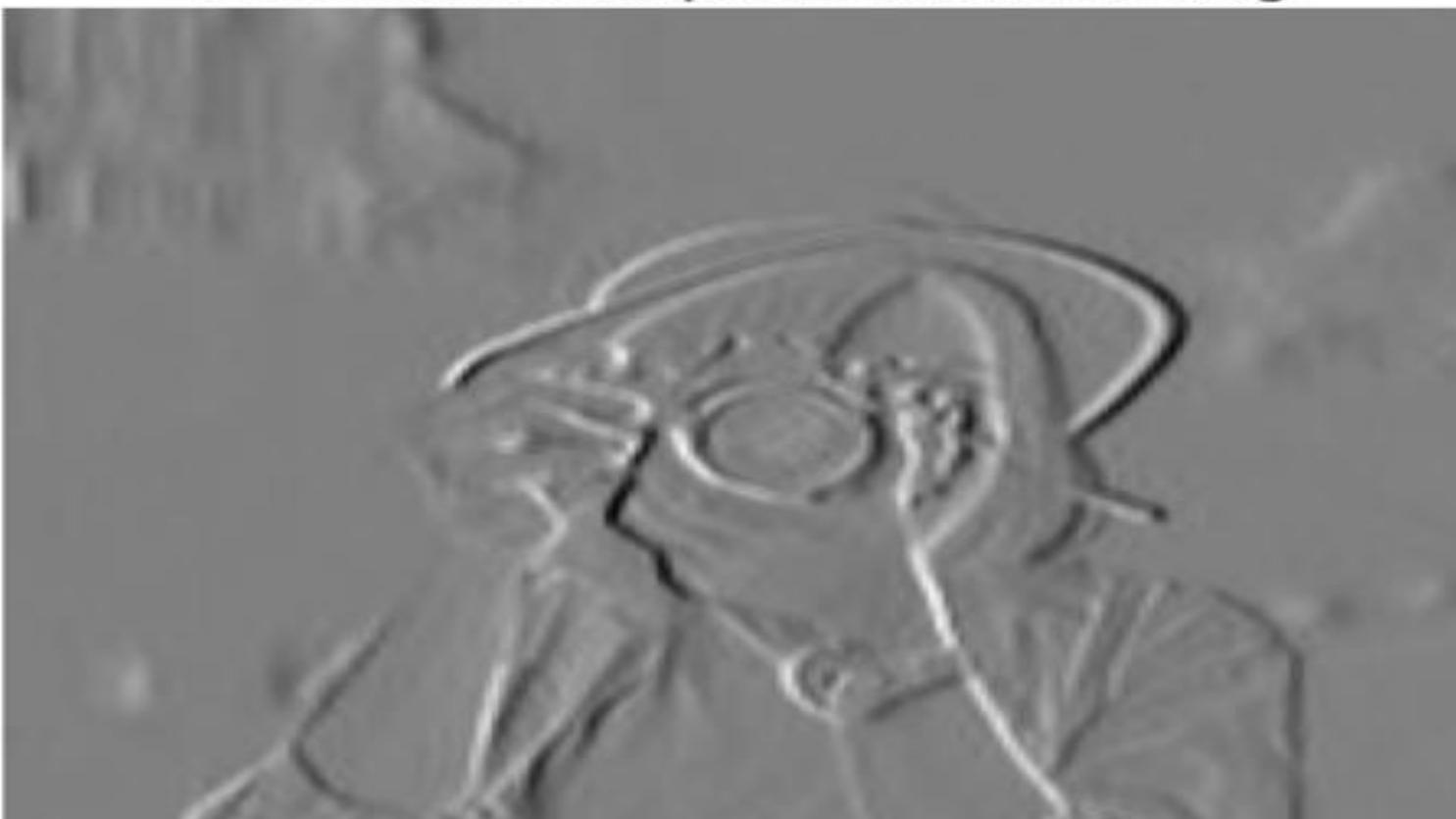
    # Perform convolution
    for i in range(output_height):
        for j in range(output_width):
            region = image[i:i+kernel_height, j:j+kernel_width]
            output[i, j] = np.sum(region * kernel)

    return output
```

6.Convolution Operation Without padding

```
# Convolve without padding
output_without_padding = convolve(image_array, filter_kernel, padding=0)
plt.imshow(output_without_padding, cmap='gray')
plt.title("Convolution Output without Padding")
plt.axis('off')
plt.show()
```

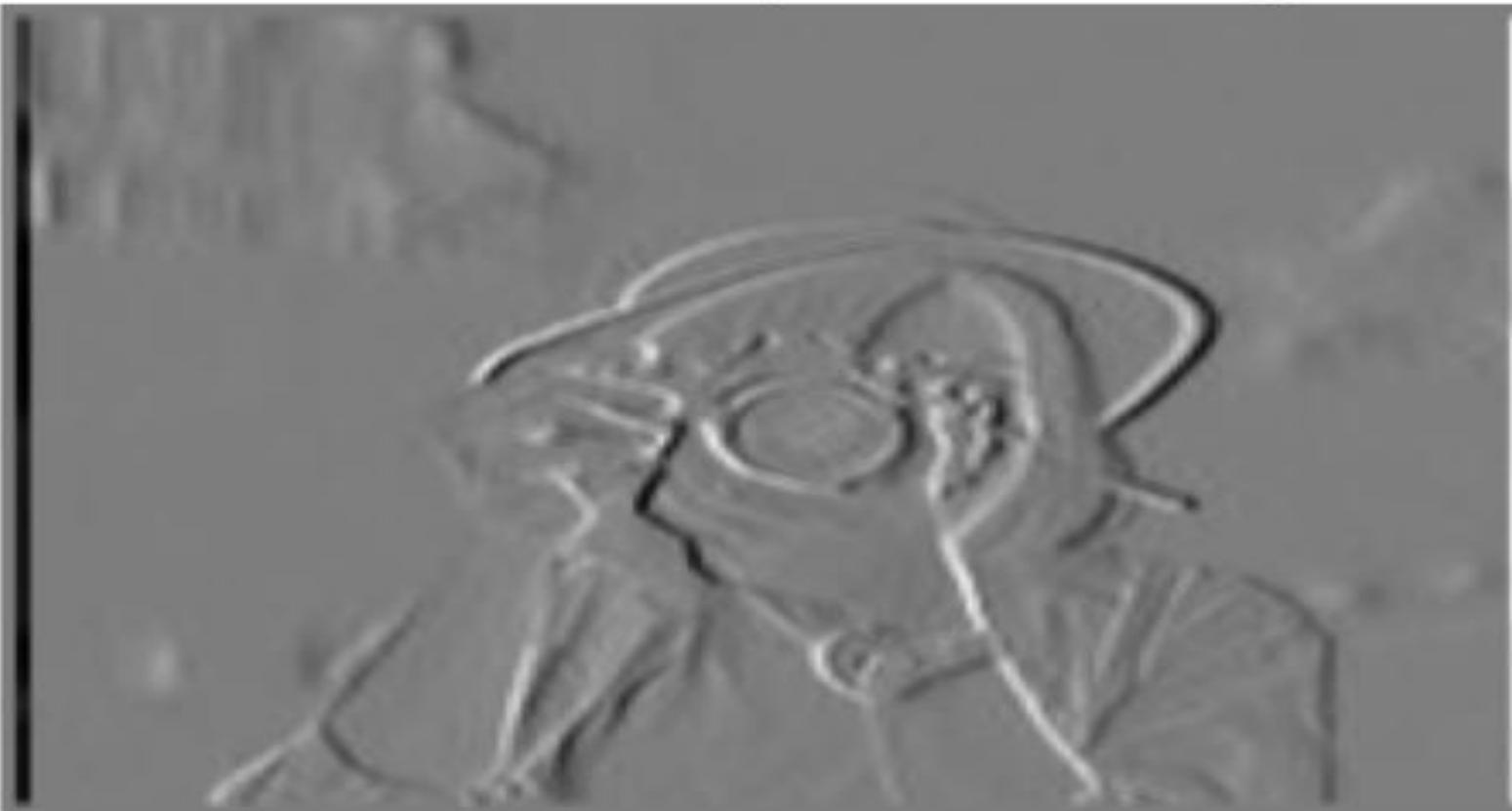
Convolution Output without Padding



7.Convolution Operation With padding of 4

```
# Convolve with padding of 1
output_with_padding = convolve(image_array, filter_kernel, padding=4)
plt.imshow(output_with_padding, cmap='gray')
plt.title("Convolution Output with Padding")
plt.axis('off')
plt.show()
```

Convolution Output with Padding



EXERCISE 2: MAX AND AVERAGE POOLING ON THE FEATURE MAP

POOLING

1.Import Necessary Libraries

```
!pip install opencv-python-headless
```

```
Collecting opencv-python-headless
  Using cached opencv_python_headless-4.10.0.84-cp37-abi3-manylinux_2_17_x86_64.manylinux2014_x86_64.whl (49.9 MB)
Requirement already satisfied: numpy>=1.17.0 in /opt/conda/lib/python3.10/site-packages (from opencv-python-headless) (1.26.3)
Installing collected packages: opencv-python-headless
Successfully installed opencv-python-headless-4.10.0.84
```

```
import numpy as np
import cv2
import matplotlib.pyplot as plt
```

2. Take image as input and Preprocess it

```
|: # Save or provide a path to a small test image for demonstration  
test_image_path = "cat_image.png"
```

```
|: # Load the image and convert it to grayscale  
def process_image(image_path):  
    image = cv2.imread(image_path, cv2.IMREAD_GRAYSCALE)  
    if image is None:  
        raise ValueError("Invalid image path or unable to load image.")  
    return image
```

3. Implement Max Pooling

```
]: def max_pooling(matrix, pool_size=(2, 2), stride=2):
    pooled_matrix = []
    for i in range(0, matrix.shape[0], stride):
        row = []
        for j in range(0, matrix.shape[1], stride):
            row.append(np.max(matrix[i:i+pool_size[0]], j:j+pool_size[1])))
        pooled_matrix.append(row)
    return np.array(pooled_matrix)
```

4.Implement Average Pooling

```
: def average_pooling(matrix, pool_size=(2, 2), stride=2):
    pooled_matrix = []
    for i in range(0, matrix.shape[0], stride):
        row = []
        for j in range(0, matrix.shape[1], stride):
            row.append(np.mean(matrix[i:i+pool_size[0], j:j+pool_size[1]])))
        pooled_matrix.append(row)
    return np.array(pooled_matrix)
```

5.Define Function to display Original Image and Pooled images

```
def plot_results(original, max_pooled, avg_pooled):

    # Display the original and pooled images
    plt.figure(figsize=(12, 4))

    plt.subplot(1, 3, 1)
    plt.title("Original Image")
    plt.imshow(original, cmap='gray')
    plt.axis('off')

    plt.subplot(1, 3, 2)
    plt.title("Max Pooled Image")
    plt.imshow(max_pooled, cmap='gray')
    plt.axis('off')

    plt.subplot(1, 3, 3)
    plt.title("Average Pooled Image")
    plt.imshow(avg_pooled, cmap='gray')
    plt.axis('off')

    plt.tight_layout()
    plt.show()
```

5. Main function to perform pooling on an image

```
: # Main function to perform pooling on an image
def apply_pooling(image_path, pool_size=(2, 2), stride=4):
    image = process_image(image_path)
    max_pooled = max_pooling(image, pool_size, stride)
    avg_pooled = average_pooling(image, pool_size, stride)
    plot_results(image, max_pooled, avg_pooled)

: test_image_path = "cat_image.png"
apply_pooling(test_image_path)
```



Original Image



Max Pooled Image



Average Pooled Image

