Deep Learning

Section 01

Overview

Introduction

Deep Learning algorithm attempts to draw similar conclusions as humans by continuously analyzing data with a given logical structure called a neural network.

AI vs ML vs DL

Artificial Intelligence

Artificial Intelligence is the development of computer systems that can perform tasks requiring human-like intelligence, such as learning, reasoning, and problem-solving.

Machine Learning

ML is a subset of AI that focuses on the development of algorithms and models that enable computers to learn from data, make predictions, and improve their performance over time without explicit programming.

Deep Learning

Deep Learning (DL) is a subset of machine learning that utilizes deep neural networks to automatically learn and represent patterns from data.

Type of ML Techniques

Supervised ML

- Classification
  - Multiclass
  - Multilabel
  - Binary
- Regression
  - Forecasting

Unsupervised ML

- Clustering
- Dimensionality Reduction

Semi Supervised

- Reinforcement Learning

Neural Networks

01 ANN

02 CNN

03 RNN

04 LSTM

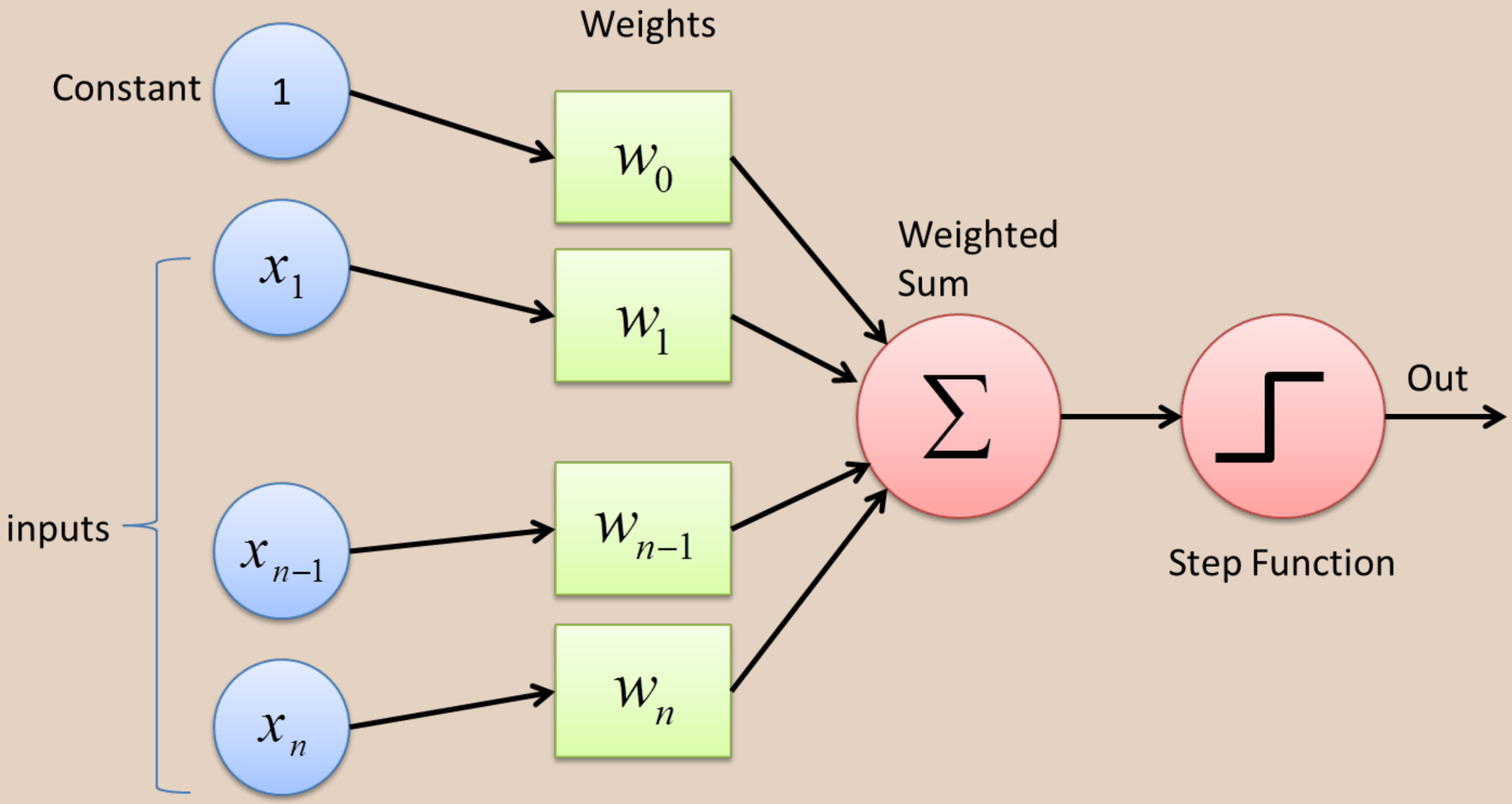
05 Auto encoder, GAN

Perceptron

Perceptrons are the basic building blocks of neural networks. Initially, in the mid-19th century, Mr. Frank Rosenblatt invented the Perceptron for performing certain calculations to detect input data capabilities.

Perceptron is a linear Machine Learning algorithm used for supervised learning for various binary classifiers. This algorithm enables neurons to learn elements and process them one by one during preparation.

Perceptron


Weights
Constant
1
w_0
x_1
w_1
Weighted
Sum
Σ
Out
inputs
x_{n-1}
w_{n-1}
x_n
w_n
Step Function

Perceptron

The net input function is a simple function that computes the dot product of input values. It will be as follows:

$$z = x_1 w_1 + x_2 w_2 + x_{n-1} w_{n-1} + x_n w_n + b$$

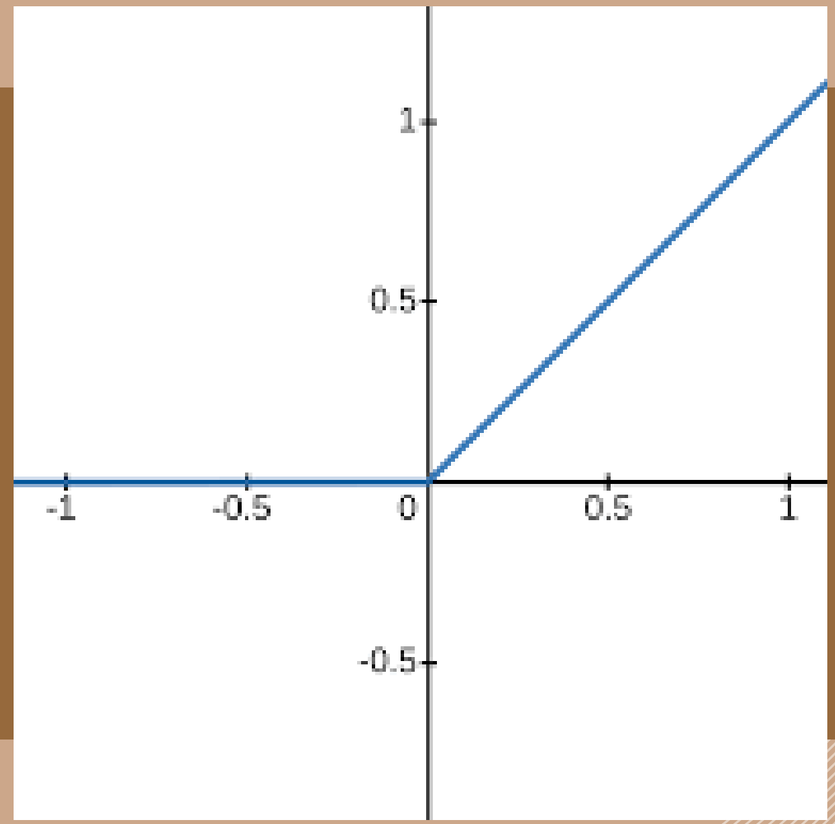
Step Function is used to convert this value of z into some range. So that we can easily predict output value.

Perceptron Problem

- **Limited to linear separability**: Perceptrons struggle with handling data that is not linearly separable, as they can only learn linear decision boundaries.
- **Lack of depth**: Perceptrons are a single layer and cannot learn complex hierarchical representations.

Activation Functions

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



Ideal Activation Function

1. Non Linear
2. Differentiable
3. Computationally Inexpensive
4. Zero Centered (Normalized)
5. Non Saturated

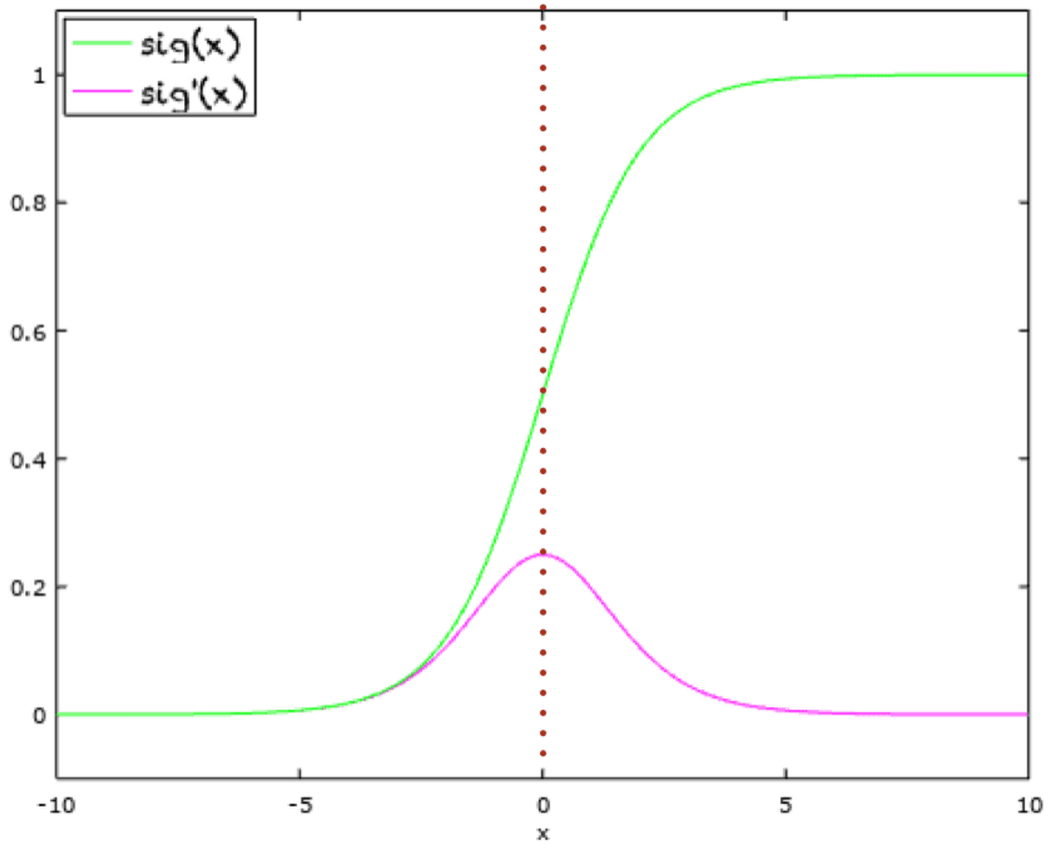
Sigmoid Activation Function

The sigmoid function is a special form of the logistic function and is usually denoted by σ(x) or sig(x). It is given by:

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

The sigmoid function is continuous, nonlinear, differential, and monotonically increasing. Using a non-linear function produces non-linear boundaries and hence, the sigmoid function can be used in neural networks for learning complex decision functions.

Sigmoid Function



Plot of σ(x) and its derivate σ'(x)

Domain: (−∞, +∞)

Range: (0, +1)

σ(0) = 0.5

Other properties

$$\sigma(x) = 1 - \sigma(-x)$$

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

$$\sigma'(x) = \sigma(x)(1 - \sigma(x))$$

Disadvantages

- Saturating Function (Gradient very small, little weight update)
- Non-Zero Centered (Make training slow & unstable)
- Computationally Expensive

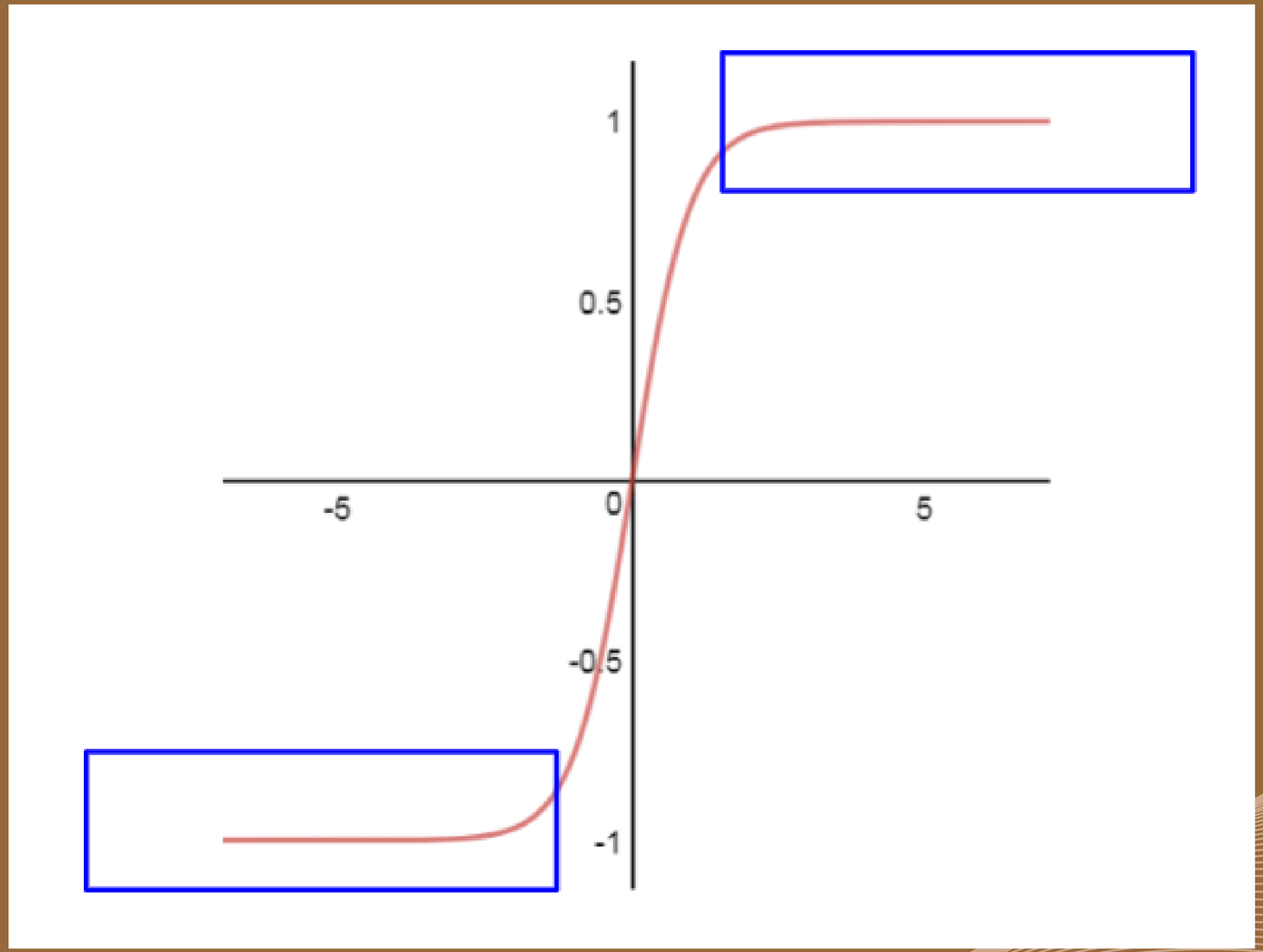
Tanh Activation Function

The tanh function maps a real-valued number to the range [-1, 1] according to the following equation:

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

Unlike the sigmoid function, its outputs are zero-centered. It is differentiable and non-linear. It’s disadvantages include computational expansiveness and saturating properties.

Tanh Function



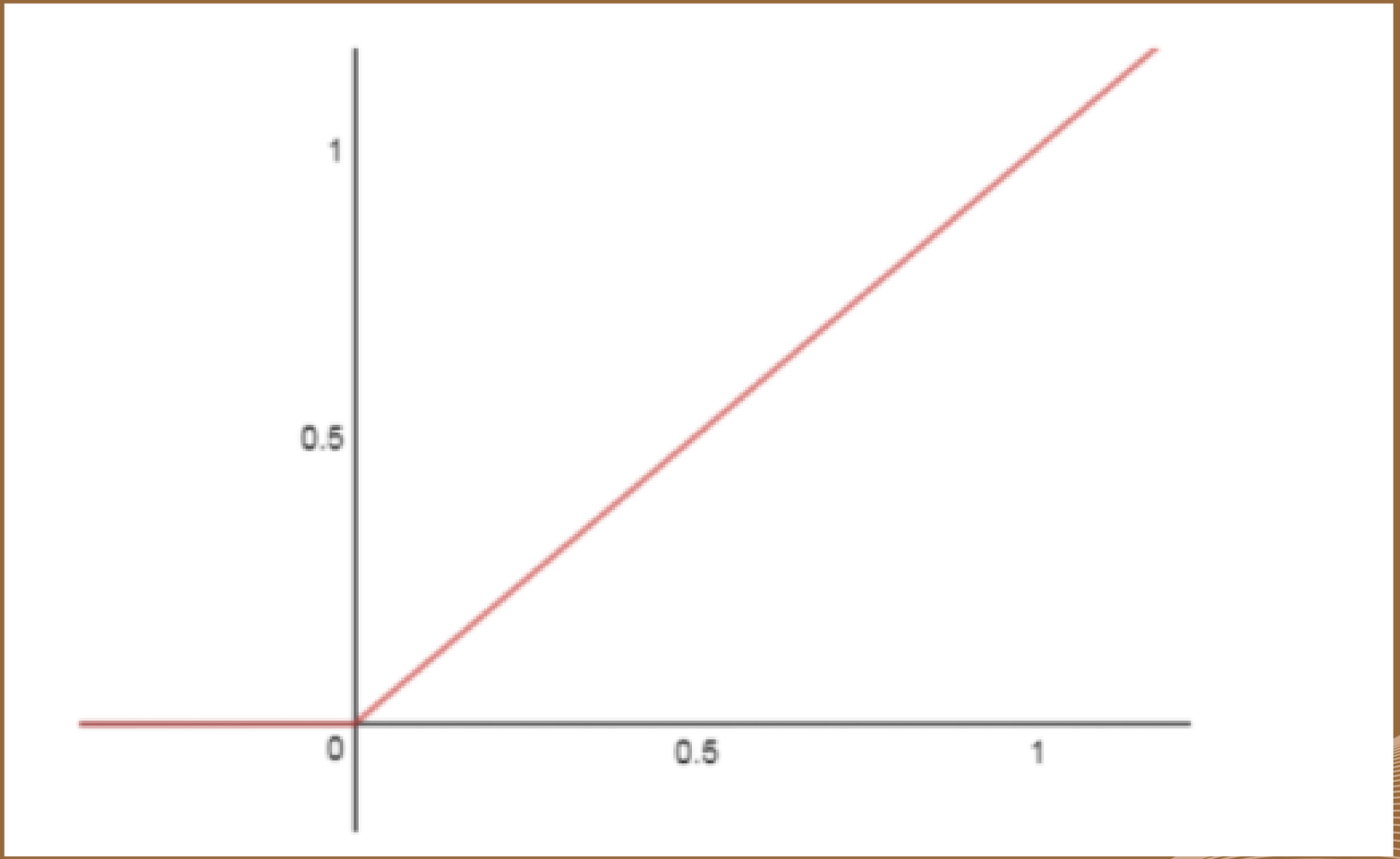
ReLU Activation Function

The activation is linear for input values greater than zero

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

Its advantages include non-linearity and non-saturating in positive regions. Its disadvantages are non-differentiability at zero and not zero centered.

ReLU Function



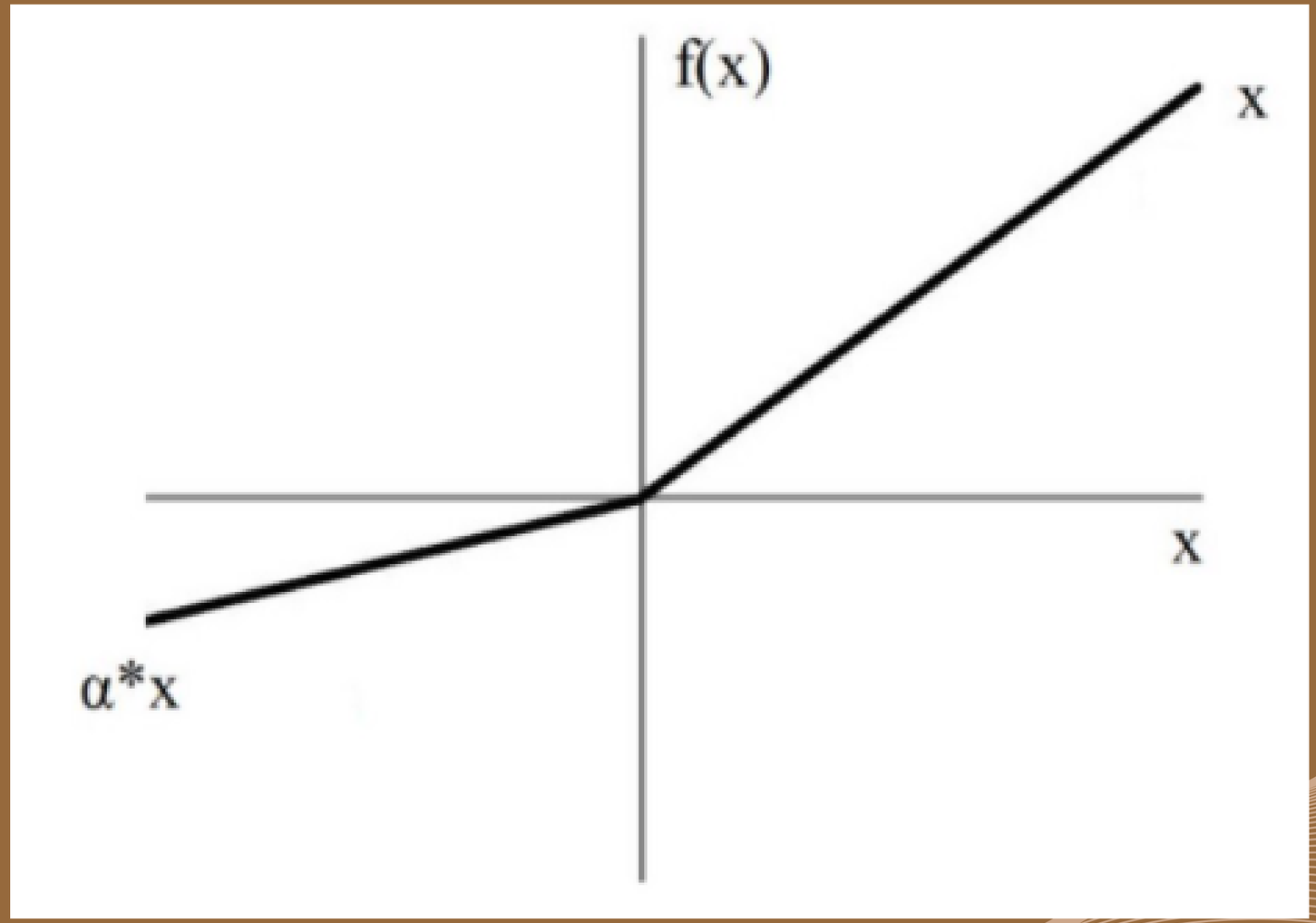
Leaky ReLU Activation Function

Leaky ReLU, is a type of activation function based on a ReLU, but it has a small slope for negative values instead of a flat slope. The slope coefficient is determined before training,

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

Its advantages include non-saturation, no ReLU dying problem, and close to zero centred.

Leaky ReLU Function

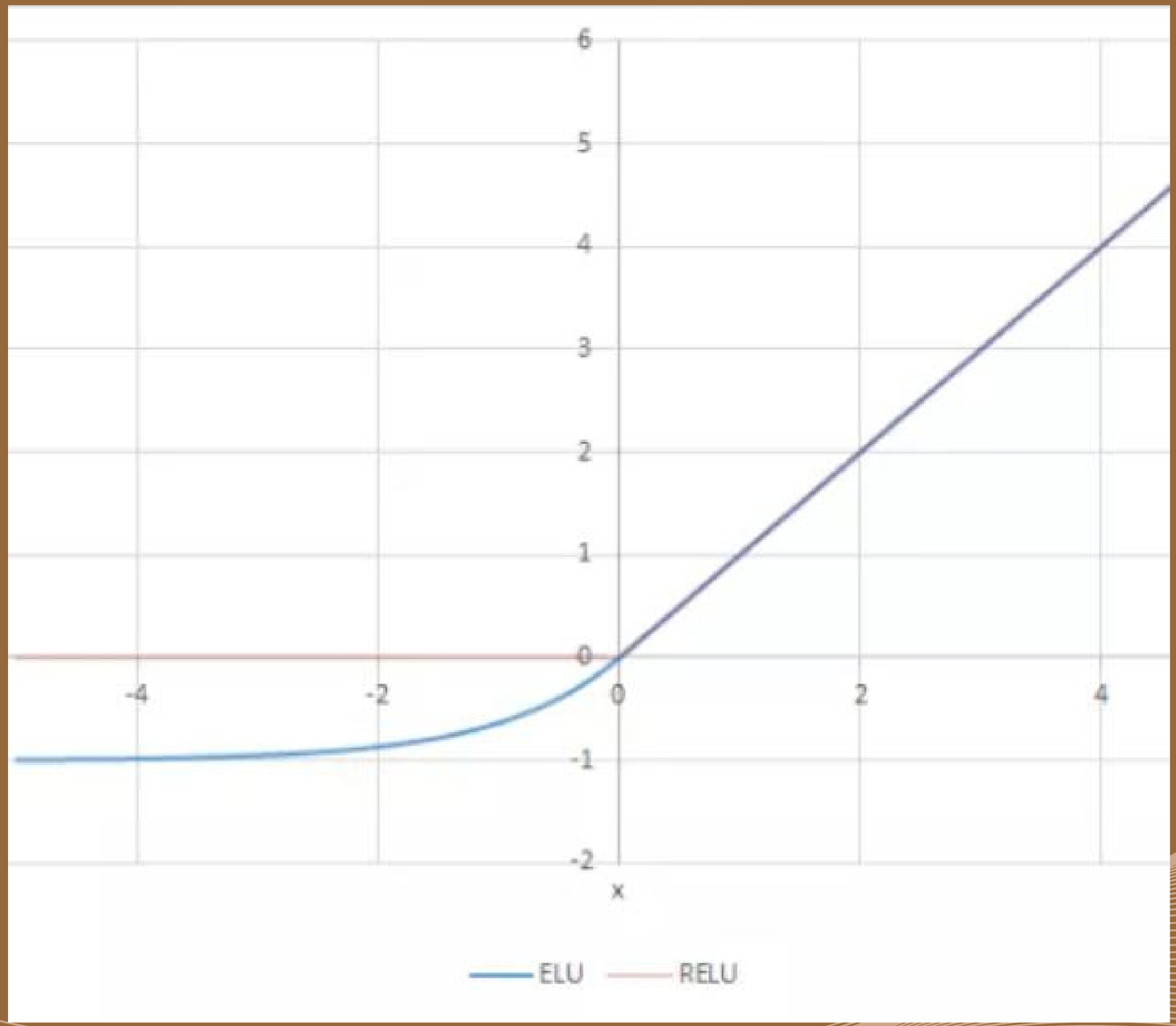


ELU Activation Function

ELU is a function that tends to converge cost to zero faster and produce more accurate results. Different to other activation functions, ELU has an extra alpha constant which should be positive number.
ELU is very similiar to RELU except negative inputs. They are both in identity function form for non-negative inputs. On the other hand, ELU becomes smooth slowly until its output equal to -α whereas RELU sharply smoothes.

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

ELU Function



Softmax Activation Function

The softmax activation function transforms the raw outputs of the neural network into a vector of probabilities, essentially a probability distribution over the input classes. Consider a multiclass classification problem with N classes.

$$f(x) = \frac{e^{zi}}{\Sigma e^{j}}$$

Model Type

Sequential Model

A Sequential model is appropriate for a plain stack of layers where each layer has exactly one input tensor and one output tensor.

Functional Model

Functional models are more flexible and powerful, allowing for more complex models with multiple inputs or outputs, shared layers, and non-linear flows.

Core Layers in NN

01

Input Layer: it accepts inputs in several different formats provided by the programmer.

02

Hidden Layer: It performs all the calculations to find hidden features and patterns.

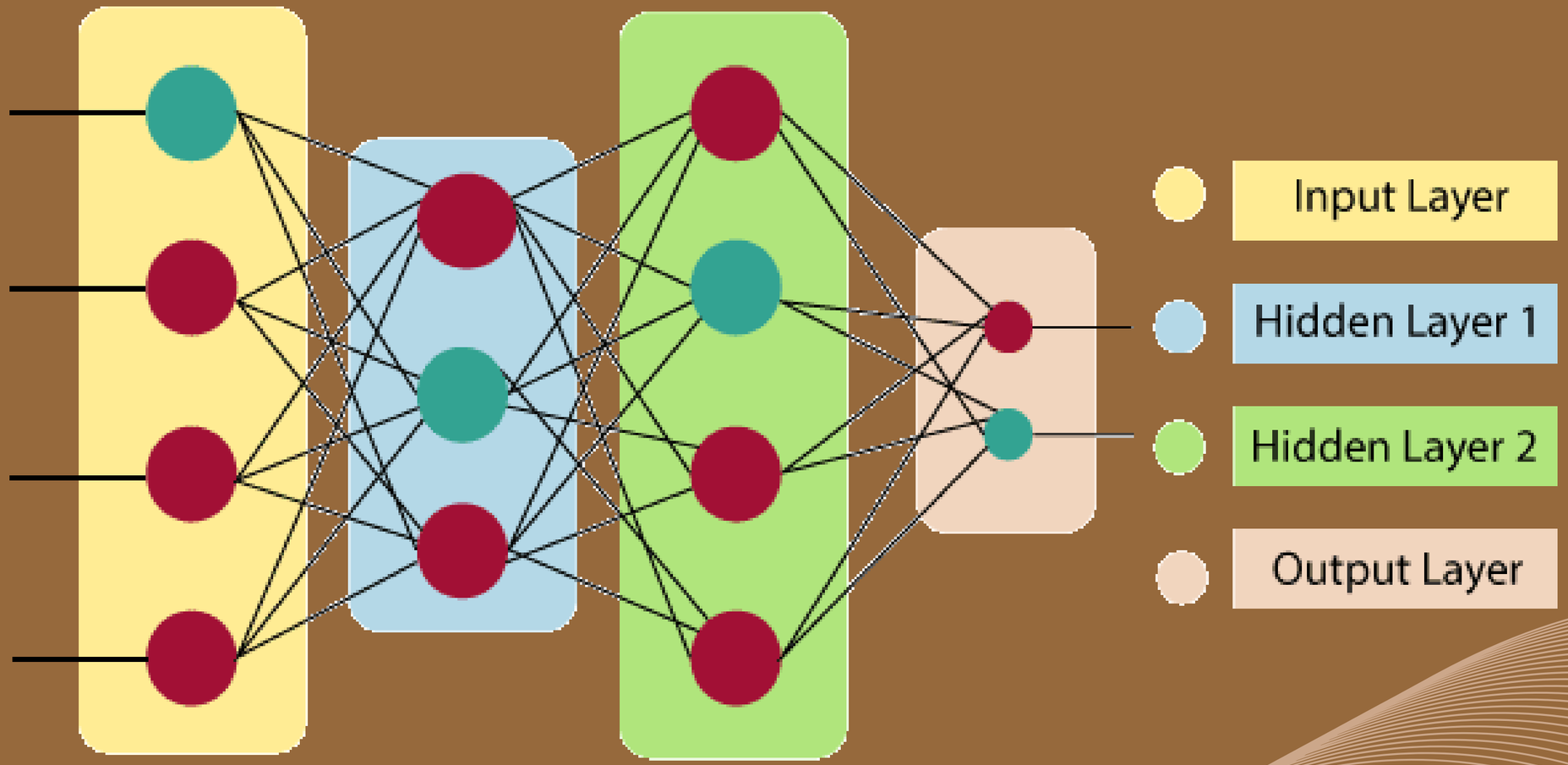
03

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

Artificial Neural Network (ANN)

An Artificial Neural Network (ANN) is a computational model inspired by the human brain’s neural structure. It consists of interconnected nodes (neurons) organized into layers. Information flows through these nodes, and the network adjusts the connection strengths (weights) during training to learn from data, enabling it to recognize patterns, make predictions, and solve various tasks in machine learning and artificial intelligence.

ANN



Advantages

- Storing data on the entire network
- Capability to work with incomplete knowledege
- Having Fault tolerence
- Can generalize

Disadvantages

- Unrecognized behaviour of network
- Hardware dependence
- Loss important features like spatial arrangement of pixels
- Overfitting

Forward Propagation

Forward propagation in a simple artificial neural network (ANN) involves the process of computing the outputs of each layer starting from the input layer and moving forward through the hidden layers to the output layer. Let's break down the process step by step for the given ANN architecture:

Input Layer

Denote the input layer as

$$X = [x_1, x_2, x_3, x_4]$$

Each input node is connected to every node in the first hidden layer.

1st Hidden Layer

Denote the weights connecting the input layer to the first hidden layer as

$$W^1 = [w^1 ij]$$

where w1 is weight from i-th input node to j-th node in first hidden layer.

compute the activation of each node in the first hidden layer using the weighted sum and an activation function f, typically a sigmoid or ReLU function:

$$z_j^1 = \sum_{i=1}^{4} w_{ij}^1 \cdot x_i + b_j^1$$

2nd Hidden Layer

Denote the weights connecting the first hidden layer to the second hidden layer as

$$W^2 = [w_{ij}^2]$$

where w2 is weight from i-th node of first hidden layer to j-th node in second hidden layer.

Compute the activation of each node in the second hidden layer similarly to the first hidden layer:

$$z_j^2 = \sum_{i=1}^{3} w_{ij}^2 \cdot a_i^1 + b_j^2$$

Output Layer

Denote the weights connecting the second hidden layer to the output layer as

$$W^3 = [w_{ij}^3]$$

where w3 is weight from i-th node of second hidden layer to j-th node in output layer.

Compute the activation of each node in the output layer using a suitable activation function

$$z_j^3 = \sum_{i=1}^{4} w_{ij}^3 \cdot a_i^2 + b_j^3$$

Back Propagation

Backpropagation in a neural network involves the process of updating the weights and biases by propagating the error backwards from the output layer to the input layer. Here's how backpropagation works in the given neural network with 4 input nodes, 2 hidden layers (3 nodes in the first hidden layer and 4 nodes in the second hidden layer), and 2 output nodes:

Output Layer Error

Compute the error at the output layer for each node using the difference between the predicted output and the actual target values:

$$\delta_j^3 = (a_j^3 - y_j) \cdot f'(z_j^3)$$

Where deltaj is error at node j in the output layer, aj is the predicted value and yj is the actual target value.

Backpropagate Error to Second Hidden Layer:

- Compute the error at each node in the second hidden layer using the errors from the output layer:

$$\delta_j^2 = \sum_{k=1}^{2} \delta_k^3 \cdot w_{kj}^3 \cdot f'(z_j^2)$$

Backpropagate Error to First Hidden Layer:

- Compute the error at each node in the first hidden layer using the errors from the second hidden layer:

$$\delta_j^1 = \sum_{k=1}^{3} \delta_k^2 \cdot w_{kj}^2 \cdot f'(z_j^1)$$

Update Weights and Biases:

Update the weights and biases using the computed errors and the learning rate

$$\Delta w_{ij}^1 = -\eta \cdot \delta_j^1 \cdot a_i^0$$

$$\Delta b_j^1 = -\eta \cdot \delta_j^1$$

$$\Delta w_{ij}^2 = -\eta \cdot \delta_j^2 \cdot a_i^1$$

$$\Delta b_j^2 = -\eta \cdot \delta_j^2$$

Ways to increase accuracy of NN

1. Increase number of epochs
2. Use ReLU in hidden layers
3. Increase number of nodes in hidden layers
4. Increase number of hidden layers

Gradient Descent

Gradient descent is a fundamental optimization algorithm used in machine learning and mathematical optimization to minimize a function by iteratively moving in the direction of the steepest descent of the function's gradient. It is widely used in training machine learning models, particularly in tasks like linear regression, logistic regression, neural network training, and more.

Gradient Descent

Gradient descent is an iterative optimization algorithm used to minimize a cost function J(θ) for parameters θ.

It works by computing the gradient of the cost function at the current parameter values and updating the parameters in the direction that minimizes the cost.
The update rule for gradient descent is typically expressed as:

$$\theta := \theta - \alpha \cdot \nabla J(\theta)$$

where α is the learning rate and $\nabla J(\theta)$ is the gradient of the cost function with respect to θ

Batch Gradient Descent

- In batch gradient descent, the entire training dataset is used to compute the gradient of the cost function at each iteration.
- It calculates the average gradient of the entire dataset and updates the parameters accordingly.
- Batch gradient descent is computationally expensive for large datasets but guarantees convergence to the global minimum for convex cost functions.

Stochastic Gradient Descent (SGD):

- In stochastic gradient descent, only one randomly selected data point (or a small subset called mini-batch) is used to compute the gradient at each iteration.
- It updates the parameters more frequently and is computationally less expensive than batch gradient descent, especially for large datasets.
- However, it introduces more noise due to the random selection of data points, which can lead to more oscillations during training.

Mini-Batch Gradient Descent:

- Mini-batch gradient descent is a compromise between batch gradient descent and stochastic gradient descent.
- It uses a small random subset of the training data (mini-batch) to compute the gradient and update the parameters.
- This approach combines the benefits of both batch and stochastic gradient descent, offering a good balance between computational efficiency and convergence speed.

1 Convergence and Learning Rate:

- The learning rate (α) in gradient descent controls the step size of parameter updates. Choosing an appropriate learning rate is crucial for the algorithm's convergence.
- If the learning rate is too small, the algorithm may take a long time to converge.
- If the learning rate is too large, the algorithm may oscillate or diverge, failing to converge to the minimum.

2 Stopping Criteria:

Gradient descent iteratively updates parameters until a stopping criterion is met. Common stopping criteria include reaching a maximum number of iterations, achieving a certain threshold of improvement in the cost function, or detecting convergence based on the change in parameters.

Vanishing Gradient Problem

Vanishing gradient problem is a phenomenon that occurs during the training of deep neural networks, where the gradients that are used to update the network become extremely small or "vanish" as they are backpropogated from the output layers to the earlier layers.

Usage of Sigmoid or tanh in hidden layer could be the reason

How to recognize it?

1. If the loss is not reducing
2. Plot weight graphs

How to handle it?

1. Reduce complexity of network
2. Use ReLu Activation function
3. Proper weight initialization
4. Batch Normalization
5. Use Residual Network

Early Stopping

Early stopping is a technique used in machine learning to prevent overfitting by monitoring the performance of a model during training and stopping the training process when the model's performance on a validation set starts to degrade. The goal of early stopping is to find the optimal point where the model generalizes well to unseen data without overfitting to the training data.

Normalization

Normalization is a preprocessing technique used in machine learning to scale and normalize data, ensuring that features are on a similar scale and have a similar distribution. This helps improve the performance and convergence of machine learning algorithms, particularly gradient-based optimization methods like gradient descent

Input Normalization

Input normalization, also known as feature scaling, involves scaling the input features to a similar range. This is typically done to prevent features with larger magnitudes from dominating the learning process. Common methods for input normalization include

- Min-Max scaling
- Standardization (z-score normalization).

Activation Normalization

Activation normalization, also known as output normalization, involves normalizing the outputs of activation functions within each layer of a neural network. This helps stabilize the learning process by keeping the activations within a certain range and preventing saturation of activation functions. Common methods for activation normalization include batch normalization and layer normalization.

Batch Normalization

Batch normalization normalizes the activations of each mini-batch during training. It computes the mean and standard deviation of activations within each mini-batch and applies a normalization transformation using learned parameters (γ and β). The formula for batch normalization is:

$$\hat{X} = \frac{X - \mu_{\text{batch}}}{\sqrt{\sigma^2_{\text{batch}} + \epsilon}}$$

$$Y = \gamma \hat{X} + \beta$$

Layer Normalization

Layer normalization normalizes the activations within each layer across all units (nodes) rather than within mini-batches. It computes the mean and standard deviation across the features of each data sample. The formula for layer normalization is:

$$\mu_{\text{layer}} = \frac{1}{m}\sum_{i=1}^{m} X_i$$

$$\sigma^2_{\text{layer}} = \frac{1}{m}\sum_{i=1}^{m}(X_i - \mu_{\text{layer}})^2$$

$$\hat{X} = \frac{X - \mu_{\text{layer}}}{\sqrt{\sigma^2_{\text{layer}} + \epsilon}}$$

$$Y = \gamma\hat{X} + \beta$$

Batch Normalization vs. Layer Normalization:

Batch normalization is applied per mini-batch and is effective for deep networks with mini-batch training. It introduces additional parameters (γ and β) to learn scale and shift.

Layer normalization is applied per data sample and is useful for recurrent neural networks (RNNs) or models where batch sizes vary. It does not introduce additional parameters for scale and shift.

Dropout

Dropout is a regularization technique used in deep learning and neural networks to prevent overfitting and improve the generalization of the model. It works by randomly deactivating (dropping out) a fraction of neurons during training, forcing the network to learn more robust and generalized features. Dropout is particularly effective in preventing co-adaptation of neurons, where neurons rely too heavily on each other and can lead to overfitting.

How does Dropout work?

During each training iteration, dropout randomly selects a fraction of neurons in the network and sets their outputs to zero.

This process effectively removes the selected neurons from the network for that iteration, forcing the model to rely on different combinations of neurons for each input, which reduces overfitting.

- Dropout is typically applied after the activation function in each layer during training.
- Common dropout rates range from 0.2 to 0.5, meaning 20% to 50% of neurons are dropped during each training iteration.

$$y_i = \begin{cases} 0 & \text{with probability } p \\ \frac{x_i}{1-p} & \text{otherwise} \end{cases}$$

Weight Initialization Technique

Weight initialization is a crucial aspect of training neural networks effectively. It involves setting the initial values of the weights in the network before training begins. Proper weight initialization can significantly impact the convergence speed, stability, and performance of neural networks.

Zero Initialization

Initializing all weights to zero is generally not recommended because it leads to symmetry breaking issues. When all weights are the same, all neurons in a layer will compute the same output, causing the network to fail to learn meaningful features.

When all weights and biases are initialized to zero, the output of each neuron and the gradients during backpropagation will also be zero. This situation leads to what is known as the "vanishing gradients" problem, where gradients become extremely small or zero, preventing meaningful updates to the weights and hindering the training process.

Random Initialization

Random initialization involves setting the weights to random values drawn from a specified distribution. This helps break symmetry and ensures that neurons in the network start with different initial values, promoting diverse feature learning.

Common distributions used for random initialization include uniform distribution and normal distribution.
If large values(between 0 to 1). It will lead to activation function saturation that leads to slow training and in the worst case vanishing gradient can be caused.

Xavier Initialization (Glorot Initialization):

Xavier initialization is designed to address the problem of vanishing or exploding gradients by setting the initial weights based on the number of input and output neurons in each layer.

Xavier initialization is suitable for activation functions like tanh and sigmoid.

He Initialization (Kaiming Initialization):

He initialization is similar to Xavier initialization but is specifically designed for activation functions that benefit from a mean of 0 and a variance of 2 in the activations, such as ReLU (Rectified Linear Unit).

He initialization helps prevent the issue of dying ReLU neurons during training.

Optimizers

Optimizers play a crucial role in training neural networks by updating the model parameters (weights and biases) during the optimization process. They determine how the model adjusts its parameters in response to the gradients computed during backpropagation. Here are some common optimizers along with their formulas, benefits, and potential drawbacks:

Adam (Adaptive Moment Estimation):

Adam is an optimization algorithm that can be used instead of the classical stochastic gradient descent procedure to update network weights iterative based in training data. Adam was presented by Diederik Kingma from OpenAI and Jimmy Ba from the University of Toronto in their 2015 ICLR paper (poster) titled “Adam: A Method for Stochastic Optimization“. I will quote liberally from their paper in this post, unless stated otherwise.

Benefits

- Adaptive learning rate for each parameter.
- Momentum-like behavior helps accelerate convergence.

Drawbacks

- Requires tuning of hyperparameters (β1, β2, ϵ).
- Memory-intensive due to maintaining moments for each parameter.

RMSProp

RMSProp is designed to accelerate the optimization process, e.g. decrease the number of function evaluations required to reach the optima, or to improve the capability of the optimization algorithm, e.g. result in a better final result.

Benefits

- Adaptive learning rate based on the magnitude of gradients.
- Helps mitigate the vanishing/exploding gradients problem.

Drawbacks

- Requires tuning of hyperparameters (β, ϵ).
- Can be sensitive to the choice of learning rate.

SGD with momentum

SGD with Momentum is an extension of the standard SGD algorithm that incorporates a momentum term to accelerate convergence and improve stability during training.

$$v_t = \beta \cdot v_{t-1} + (1-\beta) \cdot \nabla J(\theta_{t-1})$$
$$\theta_t = \theta_{t-1} - \alpha \cdot v_t$$

Benefits

- Accelerated Convergence
- Improved Stability
- Escape Local Minima

Drawbacks

- Parameter Tuning
- Memory Usage

Tricks to improve NN

1 If **Vanishing Gradient** exists:

Change Activation Function
Weight Initialization

2 If **Overfitting:**

Reduce complexity
Increase Data
Dropout Layers
Regularization
Early Stopping

Tricks to improve NN

3 **Normalization**

Normalizing Input
Batch Normalization
Normalizing Activation

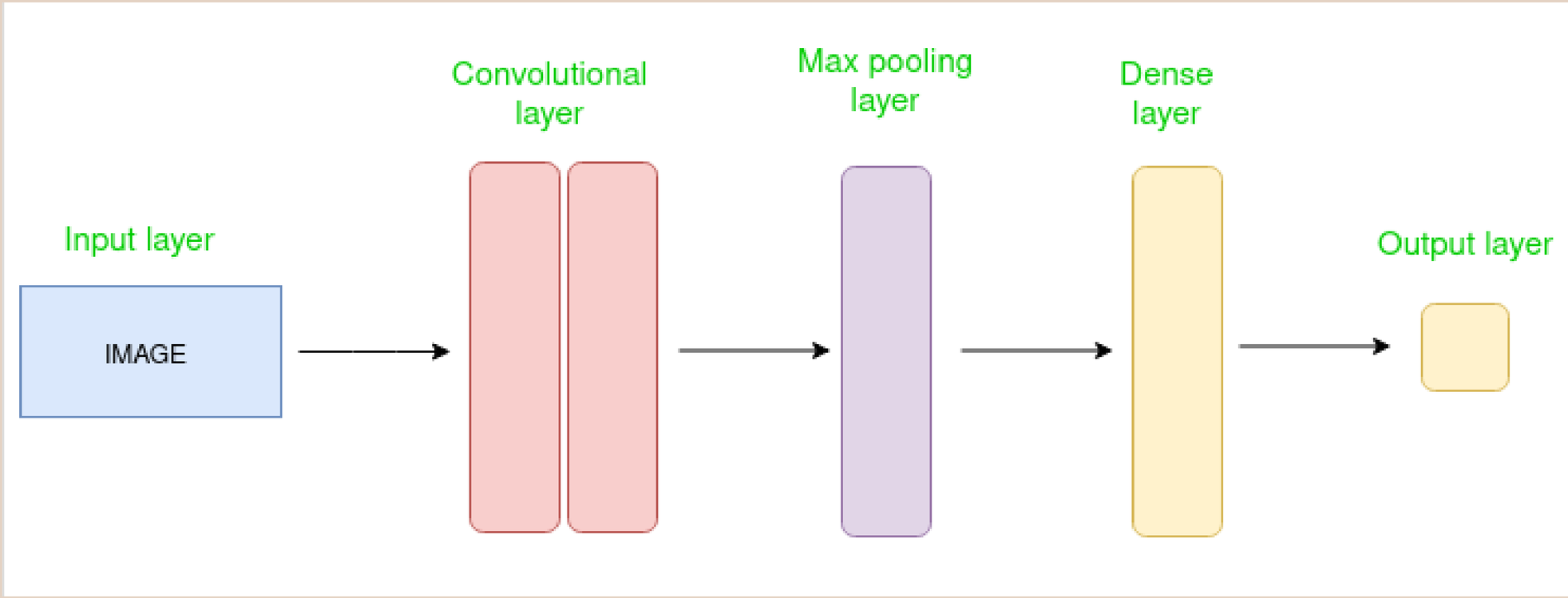
4 Gradient Checking & Clipping

5 Optimizers

Convolution Neural Network

Convolutional Neural Networks (CNNs) are a class of deep neural networks primarily used for image analysis and processing. They have revolutionized computer vision tasks such as image classification, object detection, and image segmentation.

Architecture



CNN Layers

- **Convolutional Layers:** These layers apply convolution operations to extract features from the input image. Each convolutional layer consists of multiple filters (also called kernels) that slide over the input image, performing element-wise multiplication and summation to produce feature maps.
- **Pooling Layers**: Pooling layers (e.g., Max Pooling or Average Pooling) reduce the spatial dimensions of the feature maps while retaining important information. They help in reducing computational complexity and controlling overfitting.
- **Fully Connected Layers (Dense Layers)**: After several convolutional and pooling layers, the extracted features are flattened and passed through one or more fully connected layers. These layers perform classification or regression tasks based on the learned features.

References

1 https://artemoppermann.com/activation-functions-in-deep-learning-sigmoid-tanh-relu/

2 https://n9.cl/0xczb

3 https://www.geeksforgeeks.org

4 machinelearningmastery.com


Thank
You