Name: Prathamesh Arvind Jadhav

Unit-2- Deep Neural Networks

Introduction to Neural Networks in Deep Learning

What is a Neural Network?

A **Neural Network** is a computational model inspired by the human brain. It consists of layers of **neurons (nodes)** that process data and learn patterns from it. Neural networks form the backbone of **Deep Learning (DL)** and are used in tasks like image recognition, natural language processing, and autonomous systems.

Structure of a Neural Network

A neural network consists of three main layers:

1. **Input Layer**: Receives raw data (e.g., pixels of an image).
2. **Hidden Layers**: Process data using weighted connections and activation functions.
3. **Output Layer**: Produces the final prediction or classification.

Each **neuron** (node) in a layer is connected to neurons in the next layer through **weights** and **biases**.

Working of a Neural Network

1. Forward Propagation

- The input data is multiplied by weights and passed through an **activation function**.
- This process continues through multiple layers until the final output is generated.

Name: Prathamesh Arvind Jadhav

2. Activation Functions

These introduce non-linearity, allowing the network to learn complex patterns:

- **Sigmoid**: $f(x) = \frac{1}{1+e^{-x}}$ (Used for probabilities)
- **ReLU (Rectified Linear Unit)**: $f(x) = \max(0, x)$ (Speeds up learning)
- **Softmax**: Converts outputs into probabilities (used for classification).

3. Loss Function

- Measures the error between the predicted and actual values.
- Examples: **Mean Squared Error (MSE)** for regression, **Cross-Entropy Loss** for classification.

4. Backpropagation & Optimization

- **Backpropagation** computes gradients to adjust weights and minimize errors.
- **Gradient Descent** is an optimization algorithm used to update weights efficiently.
- Variants include **Stochastic Gradient Descent (SGD)** and **Adam Optimizer**.

Types of Neural Networks

1. **Feedforward Neural Network (FNN)**: Basic type where data flows in one direction.
2. **Convolutional Neural Network (CNN)**: Used in image processing and computer vision.
3. **Recurrent Neural Network (RNN)**: Designed for sequential data like speech and time series.
4. **Long Short-Term Memory (LSTM)**: A special RNN for handling long-term dependencies.
5. **Generative Adversarial Network (GAN)**: Used for generating realistic images, videos, etc.

Name: Prathamesh Arvind Jadhav

Applications of Neural Networks

✔ **Image Recognition** (e.g., Face Detection, Object Classification)
✔ **Speech Recognition** (e.g., Alexa, Google Assistant)
✔ **Natural Language Processing** (e.g., Chatbots, Machine Translation)
✔ **Autonomous Vehicles** (e.g., Self-Driving Cars)
✔ **Healthcare** (e.g., Disease Diagnosis, Drug Discovery)

The Biological Neuron in Deep Learning

Introduction

A **biological neuron** is the fundamental unit of the human brain. It serves as the inspiration for **artificial neural networks (ANNs)** in **Deep Learning (DL)**. Understanding biological neurons helps us grasp how artificial neural networks are structured and function.

Structure of a Biological Neuron

A biological neuron consists of several key components:

1. **Dendrites**
    - Receive input signals from other neurons.
    - Act as input channels, similar to how artificial neurons receive input data.
2. **Cell Body (Soma)**
    - Processes incoming signals and determines if the neuron should "fire" (send an output).
    - Similar to an artificial neuron applying an **activation function**.
3. **Axon**
    - Transmits the output signal to other neurons.
    - Comparable to the output of an artificial neuron passing to the next layer.
4. **Synapses**
    - Connections between neurons where signals are transmitted via neurotransmitters.
    - In artificial neurons, these are represented by **weights** that determine the strength of connections.

Name: Prathamesh Arvind Jadhav

How a Biological Neuron Works

1. **Receives Input**: Dendrites collect electrical impulses from other neurons.
2. **Processes Information**: The soma sums up the inputs and applies a threshold.
3. **Generates Output**: If the summed input exceeds a threshold, the neuron "fires" and sends a signal through the axon.

This process is **mimicked** in artificial neurons, where inputs are weighted, summed, and passed through an activation function.

Comparison: Biological vs. Artificial Neurons

Feature	Biological Neuron	Artificial Neuron
Input	Received through dendrites	Received as numerical values
Processing	Soma integrates signals	Weighted sum of inputs
Activation	Fires if threshold is met	Activation function (e.g., ReLU, Sigmoid)
Output	Signal sent via axon	Output sent to the next layer
Connections	Synapses with chemical signals	Weights adjusting connection strength

---

How Biological Neurons Inspire Deep Learning

- **Parallel Processing**: Like biological neurons, artificial neural networks work with multiple interconnected nodes.
- **Learning through Adjustments**: In the brain, synaptic strength changes based on experience (learning). In ANNs, **weights** are adjusted using techniques like **backpropagation**.
- **Complex Decision-Making**: Just as the brain processes vast amounts of sensory data, deep learning models analyze large datasets to make predictions.

Name: Prathamesh Arvind Jadhav

The Perceptron in Deep Learning

Introduction

The **Perceptron** is the most fundamental unit of a **Neural Network** and serves as the building block for **Deep Learning** models. It was introduced by **Frank Rosenblatt in 1958** as a mathematical model of a **biological neuron**.

Structure of a Perceptron

A perceptron consists of the following components:

1. **Inputs** ($x_1, x_2, \ldots, x_n$)
   - Represents the features of the input data.
   - Example: In a spam detection model, inputs could be words in an email.
2. **Weights** ($w_1, w_2, \ldots, w_n$)
   - Each input has an associated weight that determines its importance.
   - The model learns the optimal weights during training.
3. **Weighted Sum (Net Input,** z**)**
   - The perceptron calculates the weighted sum of inputs:
   $$z = w_1x_1 + w_2x_2 + \ldots + w_nx_n + b$$
   - b is the **bias**, which allows shifting the activation function.
4. **Activation Function**
   - The perceptron uses a step function:
   $$f(z) = \begin{cases} 1, & \text{if } z \geq 0 \\ 0, & \text{otherwise} \end{cases}$$
   - This determines whether the neuron "fires" (outputs 1) or not (outputs 0).
5. **Output** (y)
   - A single binary value (0 or 1) representing the classification result.

Name: Prathamesh Arvind Jadhav

Working of a Perceptron

1. **Receive Input**: Takes numerical input features.
2. **Compute Weighted Sum**: Multiplies each input by its corresponding weight and adds bias.
3. **Apply Activation Function**: Uses a step function to determine the final output.
4. **Output Decision**: If the weighted sum is above the threshold, the perceptron outputs 1; otherwise, it outputs 0.

3. Update Weights using the Perceptron Learning Rule

- If the prediction is incorrect, update weights using:

$$w_i = w_i + \Delta w_i$$

$$\Delta w_i = \eta(y - \hat{y})x_i$$

where:

- η = learning rate (controls how much weights change).
- y = actual output.
- $\hat{y}$ = predicted output.

4. Repeat Until Convergence

- Adjust weights iteratively until the model correctly classifies all training data.

Name: Prathamesh Arvind Jadhav

Example: AND Gate using a Perceptron

x_1	x_2	Output (AND)
0	0	0
0	1	0
1	0	0
1	1	1

- Assign initial weights: $w_1 = 0.5$, $w_2 = 0.5$, and bias $b = -0.7$.
- Compute: $z = w_1 x_1 + w_2 x_2 + b$.
- Apply activation function to determine if output is 0 or 1.

Multilayer Feed-Forward Networks in Deep Learning

Introduction

A **Multilayer Feed-Forward Network (MLFFN)** is an advanced type of neural network where information flows **only in one direction**—from the **input layer** to the **output layer** through one or more **hidden layers**. Unlike a simple **Perceptron**, it can solve complex, non-linear problems.

Structure of a Multilayer Feed-Forward Network

A Multilayer Feed-Forward Network consists of three main types of layers:

1. Input Layer

- Receives raw input data.
- Each neuron represents a feature of the dataset (e.g., pixel values in an image).

2. Hidden Layers (One or more layers)

- Performs intermediate computations.
- Each neuron in a hidden layer applies **weights, biases, and an activation function**.
- Allows the network to learn complex patterns.

3. Output Layer

- Produces the final result (classification, regression, etc.).
- The number of neurons in the output layer depends on the task:
  - **Binary classification**: 1 neuron with **Sigmoid** activation.
  - **Multi-class classification**: Multiple neurons with **Softmax** activation.
  - **Regression**: 1 neuron with **Linear** activation.

How a Multilayer Feed-Forward Network Works

1. Forward Propagation

- Input data is passed through the layers sequentially.
- Each neuron computes a **weighted sum of inputs**, applies a **bias**, and then passes the result through an **activation function**.
- The process continues until the output layer produces the final prediction.

Mathematically, for a neuron in a hidden layer:

$$z = w_1x_1 + w_2x_2 + \ldots + w_nx_n + b$$

$$a = f(z)$$

where:

- w_i are weights,
- x_i are inputs,
- b is the bias,
- $f(z)$ is an activation function (e.g., ReLU, Sigmoid).

Name: Prathamesh Arvind Jadhav

2. Activation Functions in Hidden Layers

Activation functions introduce **non-linearity**, allowing the network to learn complex relationships.

Common activation functions include:

- **ReLU (Rectified Linear Unit)**: $f(x) = \max(0, x)$ (most common for deep networks).
- **Sigmoid**: Used for binary classification.
- **Tanh**: Similar to sigmoid but ranges from -1 to 1.

3. Backpropagation & Weight Optimization

- The network makes an initial prediction and calculates **error/loss** using a **loss function** (e.g., **Cross-Entropy** for classification, **Mean Squared Error** for regression).
- The error is propagated **backward** using **Backpropagation** to update weights.
- **Gradient Descent (or Adam optimizer)** adjusts weights to minimize error.

Advantages of Multilayer Feed-Forward Networks

✔ **Solves Non-Linear Problems** (Unlike a simple perceptron).
✔ **Can Handle Large-Scale Data** (Multiple layers extract features automatically).
✔ **Generalization** (Learns patterns beyond training data if well-regularized).

Example: MLFFN for Handwritten Digit Recognition (MNIST Dataset)

1. **Input Layer**: 784 neurons (28×28 pixel grayscale images).
2. **Hidden Layers**: 2 layers (128 neurons each, using ReLU).
3. **Output Layer**: 10 neurons (each representing digits 0-9, using Softmax).
4. **Training Process**:
    - Use **Cross-Entropy Loss**.
    - Optimize using **Adam or SGD**.

Name: Prathamesh Arvind Jadhav

Training Neural Networks in Deep Learning

Introduction

Training a **Neural Network (NN)** means adjusting its weights and biases so that it can accurately predict outputs from given inputs. This is done using an iterative learning process involving **forward propagation, loss computation, backpropagation, and optimization**.

Steps in Training a Neural Network

1. Initialize Weights and Biases

- Weights are assigned random small values (e.g., Gaussian distribution).
- Biases are typically initialized to zero or small values.

2. Forward Propagation

- The input data is passed through the network, layer by layer.
- Each neuron applies the **weighted sum of inputs**, adds bias, and applies an **activation function**:

$$z = W \cdot X + b$$

$$a = f(z)$$

where:

- W = weights
- X = input
- b = bias
- $f(z)$ = activation function (ReLU, Sigmoid, etc.)

- This process continues until the final **output layer** generates a prediction.

Name: Prathamesh Arvind Jadhav

3. Compute Loss (Error Calculation)

- The difference between the **predicted output** and **actual output (ground truth)** is calculated using a **loss function**.
- The choice of **loss function** depends on the task:
  - **Regression → Mean Squared Error (MSE)**

$$L = \frac{1}{N}\sum(y - \hat{y})^2$$

  - **Binary Classification → Binary Cross-Entropy**
  - **Multi-Class Classification → Categorical Cross-Entropy**

4. Backpropagation (Gradient Calculation)

- Backpropagation is used to **calculate gradients of the loss function** with respect to each weight and bias using **chain rule differentiation**.
- The gradients indicate **how much each weight contributes to the error**.

Mathematically:

$$\frac{\partial L}{\partial W} = \frac{\partial L}{\partial a} \cdot \frac{\partial a}{\partial z} \cdot \frac{\partial z}{\partial W}$$

where:

- L = Loss function
- W = Weight
- z = Weighted sum

Name: Prathamesh Arvind Jadhav

5. Update Weights (Optimization Step)

- Weights are updated using an **optimization algorithm** such as **Gradient Descent** to minimize the error:

$$W = W - \eta \cdot \frac{\partial L}{\partial W}$$

where:

- η (learning rate) controls how much weights are updated.

Common Optimizers

- **Stochastic Gradient Descent (SGD):** Updates weights using a small batch of data.
- **Adam Optimizer:** Combines momentum and adaptive learning rates (commonly used).
- **RMSprop:** Adjusts learning rate based on recent gradient magnitudes.

6. Repeat Until Convergence

- Steps **2-5** are repeated for **multiple epochs** (full passes over the dataset) until the loss function **converges to a minimum**.

Hyperparameters Affecting Training

- **Learning Rate (η)**: Determines step size for weight updates.
- **Batch Size**: Number of training examples processed at once.
- **Number of Epochs**: How many times the entire dataset is processed.
- **Regularization (L1/L2, Dropout)**: Prevents overfitting.

Final Output: A Trained Neural Network

- After training, the network **generalizes** patterns from the dataset.
- It can make **predictions** on **new, unseen data**.

Name: Prathamesh Arvind Jadhav

Forward Propagation and Backpropagation in Deep Learning

1. Forward Propagation

Forward Propagation is the process of **passing inputs through the neural network** to compute the output (prediction). It occurs in the following steps:

Steps of Forward Propagation

1. **Input Layer**:
    - The input data is fed into the neural network.

2. **Weighted Sum Calculation** (for each neuron):
    - Each neuron computes the weighted sum of its inputs plus a bias term:

$$z = W \cdot X + b$$

where:

- W = Weights
- X = Inputs
- b = Bias

3. **Activation Function Application**:
    - The result passes through an **activation function** to introduce non-linearity:

$$a = f(z)$$

    - Common activation functions:
        - **ReLU**: $\max(0, z)$
        - **Sigmoid**: $\frac{1}{1+e^{-z}}$
        - **Tanh**: $\frac{e^{z}-e^{-z}}{e^{z}+e^{-z}}$

4. **Propagation Through Hidden Layers**:
    - The output from one layer becomes the input for the next.

5. **Output Layer**:
    - The final layer produces the network's prediction.
    - Uses an appropriate activation function (**Softmax** for classification, **Linear** for regression).

Name: Prathamesh Arvind Jadhav

Example

If we have an input $X = [x_1, x_2]$, the forward propagation process for one neuron in a hidden layer looks like this:

$$z = w_1 x_1 + w_2 x_2 + b$$

$$a = ReLU(z) = \max(0, z)$$

Once the network generates an output, the next step is to compute the **loss** and **optimize the weights**, which leads to **Backpropagation**.

2. Backpropagation

Backpropagation is the process of **adjusting weights and biases** in the network to reduce prediction error. It calculates how much each weight contributed to the error and updates them accordingly.

Steps of Backpropagation

1. **Compute the Loss (Error Calculation)**
    - The error is measured using a **loss function**:
        - **Mean Squared Error (MSE)** for regression:

$$L = \frac{1}{N} \sum (y - \hat{y})^2$$

        - **Cross-Entropy Loss** for classification:

$$L = -\sum y \log(\hat{y})$$

    - Here, y is the actual output, and $\hat{y}$ is the predicted output.

Name: Prathamesh Arvind Jadhav

2. **Calculate Gradients (Derivative of Loss with Respect to Weights)**
   - Using the **chain rule of calculus**, we compute how much each weight contributed t
   - The gradient of loss with respect to weight W is:

$$\frac{\partial L}{\partial W} = \frac{\partial L}{\partial \hat{y}} \cdot \frac{\partial \hat{y}}{\partial z} \cdot \frac{\partial z}{\partial W}$$

3. **Update Weights (Gradient Descent or Optimizers)**
   - We adjust the weights in the direction that reduces the loss:

$$W = W - \eta \cdot \frac{\partial L}{\partial W}$$

     where η **(learning rate)** controls the step size.
   - Common optimizers: **SGD**, **Adam**, **RMSprop**.
4. **Repeat for All Layers (Backpropagation Through Hidden Layers)**
   - The error is propagated **backward**, layer by layer, until all weights are updated.

Key Differences Between Forward and Backpropagation

Feature	Forward Propagation	Backpropagation
Purpose	Compute predictions	Adjust weights to minimize error
Direction	Input → Output	Output → Input
Mathematical Operations	Matrix multiplications, activations	Gradients, chain rule differentiation
Uses Loss Function?	No	Yes
Involves Weight Updates?	No	Yes

Name: Prathamesh Arvind Jadhav

Activation Functions in Deep Learning

Activation functions introduce **non-linearity** in neural networks, allowing them to learn complex patterns. Different activation functions are used depending on the problem type (classification, regression, etc.).

1.Linear Activation Function

Formula:

$$f(x) = x$$

Graph: A straight line passing through the origin.

Pros:

 Simple and easy to implement.
 Used in the output layer for regression problems.

Cons:

 Cannot capture non-linearity in data.
 Causes deep networks to behave like a single-layer model.

Use Case:

- Used in **regression tasks** (output layer).

---

2. Sigmoid Activation Function

Formula:

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

Graph: S-shaped curve (between 0 and 1).

Pros:

 Used for **binary classification** (output is between 0 and 1).
 Good for probability estimation.

Cons:

 Vanishing Gradient Problem (gradients become very small in deep networks, making training slow).
 Not zero-centered (can cause weight updates to be inefficient).

Use Case:

- Used in the **output layer of binary classification models**.

---

3. Tanh (Hyperbolic Tangent) Activation Function

Formula:

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

Graph: S-shaped curve (between -1 and 1).

Pros:

 Zero-centered output (better weight updates compared to Sigmoid).
 Stronger gradient than Sigmoid.

Cons:

 Still suffers from the **Vanishing Gradient Problem** for very large or small values of xxx.

Use Case:

- Preferred over **Sigmoid** in hidden layers of deep networks.
- Used in **RNNs (Recurrent Neural Networks)**.

Name: Prathamesh Arvind Jadhav

4. Hard Tanh (Hard Hyperbolic Tangent) Activation Function

Formula:

$$f(x) = \begin{cases} -1, & x < -1 \\ x, & -1 \leq x \leq 1 \\ 1, & x > 1 \end{cases}$$

Graph: Similar to Tanh, but with hard clipping at -1 and 1.

Pros:

 Faster to compute than regular Tanh.
 Helps in **reducing computational complexity**.

Cons:

 Hard clipping may lead to loss of gradient for large values of xxx.

Use Case:

- Used in **low-power AI applications** where computational efficiency is crucial.

---

5. Softmax Activation Function

Formula:

$$f(x_i) = \frac{e^{x_i}}{\sum_j e^{x_j}}$$

where x_i represents the logits (raw predictions), and the denominator is the sum of exponentials of all logits.

Graph: Converts input values into probabilities (sum = 1).

Pros:

 Converts outputs into probability distributions (each output is between 0 and 1).
 Helps in multi-class classification tasks.

Cons:

 Can be computationally expensive (due to exponentiation).
 Sensitive to **outliers** in logits (high values dominate).

Use Case:

- Used in the **output layer of multi-class classification models**.

---

6. Rectified Linear Unit (ReLU) Activation Function

Formula:

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

Graph: Passes positive values unchanged, sets negative values to zero.

Pros:

 Solves Vanishing Gradient Problem (no exponential operations).
 Computationally efficient and works well in deep networks.

Cons:

 Dying ReLU Problem (if many neurons output 0, they stop learning).
 Not zero-centered.

Use Case:

- **Most commonly used activation function** in **hidden layers of deep neural networks**.

Comparison Table of Activation Functions

Activation Function	Output Range	Used For	Pros	Cons
Linear	$(-\infty,\infty)$	Regression	Simple, interpretable	No non-linearity, cannot model complex patterns
Sigmoid	(0,1)	Binary Classification	Good for probabilities	Vanishing Gradient Problem, not zero-centered
Tanh	$(-1,1)$	Hidden Layers	Zero-centered, better than Sigmoid	Vanishing Gradient Problem
Hard Tanh	$(-1,1)$	Low-Power AI	Faster than Tanh	Hard clipping can affect learning
Softmax	(0,1)	Multi-Class Classification	Converts outputs to probabilities	Sensitive to large input values
ReLU	$[0,\infty)$	Deep Learning Hidden Layers	Fast, avoids Vanishing Gradient	Dying ReLU problem

Loss Functions in Deep Learning

A **loss function** measures how far the predicted output of a neural network is from the actual target value. The goal of training a deep learning model is to **minimize the loss** by adjusting the model's weights through backpropagation.

---

1. Loss Function Notation

Let's define key notations used in loss functions:

- y = True (actual) value
- y^ = Predicted value by the model
- N = Number of samples
- L(y,y^) = Loss function measuring the difference between y and y^

Loss functions can be classified into different categories based on the type of task.

---

2. Loss Functions for Regression

Used when the output is a **continuous value** (e.g., predicting house prices, stock prices).

(a) Mean Squared Error (MSE)

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

Pros:

- Penalizes large errors heavily (good for stable predictions).

Cons:

- Sensitive to **outliers** (since errors are squared).

(b) Mean Absolute Error (MAE)

$$MAE = \frac{1}{N}\sum_{i=1}^{N}|y_i - \hat{y}_i|$$

Pros:

- More **robust to outliers** than MSE.

Cons:

- MAE is not differentiable at 0, making optimization difficult.

Name: Prathamesh Arvind Jadhav

(c) Huber Loss

$$L(y, \hat{y}) = \begin{cases} \frac{1}{2}(y - \hat{y})^2, & |y - \hat{y}| \leq \delta \\ \delta(|y - \hat{y}| - \frac{1}{2}\delta), & |y - \hat{y}| > \delta \end{cases}$$

✅ **Pros:**

- Combines benefits of **MSE (small errors)** and **MAE (large errors)**.
- More robust to outliers.

(d) Log-Cosh Loss

$$L(y, \hat{y}) = \sum \log(\cosh(\hat{y} - y))$$

✅ **Pros:**

- Similar to Huber but smoother.
- Less sensitive to outliers than MSE.

3. Loss Functions for Classification

Used when the task is **predicting a category or class** (e.g., dog vs. cat classification).

(a) Binary Cross-Entropy (for Binary Classification)

$$L = -\frac{1}{N} \sum_{i=1}^{N} [y_i \log(\hat{y}_i) + (1 - y_i) \log(1 - \hat{y}_i)]$$

✅ **Pros:**

- Works well for **binary classification** tasks (e.g., spam detection).

(b) Categorical Cross-Entropy (for Multi-Class Classification)

$$L = -\sum_{i=1}^{N}\sum_{j=1}^{C} y_{ij}\log(\hat{y}_{ij})$$

where C is the number of classes.

✅ **Pros:**

- Used when each example belongs to **one** of multiple categories.
- Works with **Softmax activation** in the output layer.

(c) Sparse Categorical Cross-Entropy

Used when labels are integers instead of one-hot encoded vectors.

✅ **Pros:**

- More memory-efficient for multi-class problems.

(d) Kullback-Leibler Divergence (KL Divergence)

$$L = \sum y \log\left(\frac{y}{\hat{y}}\right)$$

✅ **Pros:**

- Used in **probabilistic models** where we compare two probability distributions.

4. Loss Functions for Reconstruction (Autoencoders, GANs)

Used when the goal is to **reconstruct** the original input (e.g., image denoising, generative models).

(a) Mean Squared Error (MSE)

Used in **autoencoders** for image and data reconstruction.

(b) Binary Cross-Entropy

Used when reconstructing **binary** data (e.g., black-and-white images).

(c) Wasserstein Loss (for GANs)

Used in **Wasserstein GANs (WGANs)** to measure the difference between real and generated distributions.

Pros:

- Solves mode collapse in **Generative Adversarial Networks (GANs)**.

Comparison of Loss Functions

Loss Function	Task	Pros	Cons
MSE	Regression	Penalizes large errors	Sensitive to outliers
MAE	Regression	Robust to outliers	Harder to optimize
Huber	Regression	Balanced, robust	Requires tuning δ\deltaδ
Binary Cross-Entropy	Binary Classification	Works well with probabilities	Not ideal for multi-class problems
Categorical Cross-Entropy	Multi-Class Classification	Works with Softmax	Requires one-hot labels
Sparse Categorical Cross-Entropy	Multi-Class Classification	More memory-efficient	Requires integer labels
KL Divergence	Probability Distribution	Measures divergence	Requires careful interpretation
Wasserstein Loss	GANs	Solves mode collapse	Computationally intensive

Hyperparameters in Deep Learning

Hyperparameters are **tunable parameters** that control how a neural network is trained. Unlike model parameters (like weights and biases), hyperparameters are **set before training** and significantly impact the model's performance.

Name: Prathamesh Arvind Jadhav

1. Learning Rate (α)

Definition:

The learning rate **controls how much** the model updates its weights in response to the error.

Mathematical Representation:

If w_t is the weight at step t, and $\nabla L(w_t)$ is the gradient of the loss function, then:

$$w_{t+1} = w_t - \alpha \nabla L(w_t)$$

where α is the learning rate.

Effects:

- **High Learning Rate**: Faster training but may overshoot the optimal solution.
- **Low Learning Rate**: More stable convergence but training is slow.

Choosing the Right Learning Rate:

- **Too high** → Model may never converge (oscillates).
- **Too low** → Model converges too slowly or gets stuck in local minima.

 Solution: Use **Learning Rate Scheduling** (reduce α\alphaα over time).

---

2. Regularization

Definition:

Regularization helps **prevent overfitting** by penalizing large weight values.

Name: Prathamesh Arvind Jadhav

Types of Regularization:

(a) L1 Regularization (Lasso)

$$L_{L1} = \lambda \sum |w|$$

✅ Encourages **sparse** weights (some weights become zero).

❌ Can lead to loss of important features.

(b) L2 Regularization (Ridge)

$$L_{L2} = \lambda \sum w^2$$

✅ **Prevents overfitting** by keeping weights small.

❌ Doesn't produce sparse weights like L1.

🔹 **Solution:** Use **Elastic Net** (combination of L1 and L2).

(c) Dropout Regularization

- Randomly **drops** neurons during training to prevent co-adaptation.
- **Dropout Rate (p):** Percentage of neurons dropped.
  - ✅ **Reduces overfitting** in deep networks.

3. Momentum

Definition:

Momentum helps accelerate **gradient descent** by smoothing updates using previous gradients.

Mathematical Representation:

$$v_t = \beta v_{t-1} + (1 - \beta)\nabla L(w_t)$$

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

where:

- v_t is the velocity (previous gradients),
- β (0.9 commonly used) is the momentum term.

Effects:

Helps avoid local minima and sharp oscillations.
Improves training speed and stability.

 Optimization Algorithms Using Momentum:

- **SGD with Momentum**: Standard gradient descent with momentum.
- **Adam Optimizer**: Uses momentum with adaptive learning rates.

---

4. Sparsity

Definition:

Sparsity ensures that only **a few neurons are active** at any time, reducing overfitting and improving interpretability.

Name: Prathamesh Arvind Jadhav

Methods to Induce Sparsity:

(a) L1 Regularization (Lasso)

- Encourages **many weights to become zero**, making the network sparse.

(b) Sparse Autoencoders

- Add a **sparsity constraint** to hidden layers by forcing most neurons to have low activation values.

(c) Batch Normalization

- Helps keep activations balanced and can indirectly promote sparsity.

Effects of Sparsity:

Efficient computation (fewer active neurons).
Better generalization (reduces overfitting).

Comparison of Hyperparameters

Hyperparameter	Purpose	Effect on Model
Learning Rate	Controls step size of weight updates	Too high → unstable; Too low → slow convergence
Regularization	Prevents overfitting by penalizing large weights	L1 → Sparse model; L2 → Smooth weights
Momentum	Speeds up training and avoids local minima	Reduces oscillations in gradient descent
Sparsity	Reduces the number of active neurons	Improves interpretability and efficiency

Name: Prathamesh Arvind Jadhav