UNIT-2

1.Explain gradient descent in ANN?

Gradient Descent in Artificial Neural Networks (ANN)

Gradient descent is a fundamental optimization algorithm used to train artificial neural networks by minimizing the error (or loss) function. It iteratively adjusts the network's parameters (weights and biases) to reduce the difference between the predicted outputs and the actual target values.

What is Gradient Descent?

Gradient descent is an iterative method for finding the minimum of a function. In the context of ANNs, the function is the loss function (also called cost function), which measures how well the network is performing.

- The **goal** is to find the set of weights and biases that minimize this loss.
- Gradient descent uses the **gradient** (vector of partial derivatives) of the loss function with respect to the parameters to know the direction of steepest increase.
- It updates parameters by moving in the **opposite direction** of the gradient to reduce error.

How Gradient Descent Works in ANN

1. **Initialization:**

   The network starts with initial random weights and biases.

2. **Forward Propagation:**

   Input data is passed through the network to compute the output (prediction).

3. **Loss Calculation:**

   The network's output is compared to the true output using a loss function (e.g., Mean Squared Error, Cross-Entropy), quantifying prediction error.

4. **Backward Propagation (Backpropagation):**

   Calculate the gradient of the loss function with respect to each weight and bias using the chain rule. This tells how much a small change in each parameter affects the loss.

5. **Parameter Update:**

   Each parameter (weight or bias) is updated by subtracting a fraction (learning rate) of its gradient:

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

   Where:

   - θ is a weight or bias parameter
   - η is the learning rate (step size)
   - $\frac{\partial L}{\partial \theta}$ is the gradient of the loss with respect to θ

6. **Repeat:**

   Steps 2-5 are repeated over many iterations (epochs) until the loss converges to a minimum or reaches an acceptable value.

Types of Gradient Descent

- **Batch Gradient Descent:** Uses the entire training dataset to compute the gradient before updating parameters.
- **Stochastic Gradient Descent (SGD):** Uses a single training example at a time, making updates more frequent and noisy but often faster.
- **Mini-batch Gradient Descent:** Uses small subsets ("mini-batches") of the data to balance efficiency and stability.

Example Summary:

Imagine training a neural network to recognize handwritten digits. Initially, predictions are poor (high loss). Gradient descent iteratively tweaks the weights—guided by the gradient of the loss—to gradually reduce the error and improve prediction accuracy.

2.Briefly explain about dropout.

Dropout is a regularization technique used in neural networks to prevent overfitting during training. It works by randomly "dropping out" or deactivating a fraction of neurons (along with their connections) in a layer during each training iteration. This means these neurons are temporarily removed from the network's forward and backward passes.

Key points:

- The dropout rate specifies how many neurons are dropped (e.g., 0.3 means 30% of neurons are randomly turned off during training).
- The remaining active neurons' outputs are scaled up to keep the overall signal magnitude consistent.
- By effectively training many smaller subnetworks, dropout reduces reliance on any single neuron and its co-adaptations, improving the model's ability to generalize.
- During testing/inference, dropout is turned off, and the full network is used with scaled weights.

Example:

If a layer has 6 neurons and a dropout rate of 1/3, then in each training batch, roughly 2 neurons are randomly deactivated, and the other 4 are scaled up accordingly. Over epochs, different subsets are trained which reduces overfitting.

Why it works:

- Introduces noise in the network during training, forcing neurons to develop more robust features.
- Acts as an ensemble of many smaller networks, improving generalization.

In summary:

Dropout randomly disables neurons during training to prevent overfitting by encouraging a network to learn redundant, distributed representations and not rely too heavily on any single neuron.

This explanation is synthesized from reliable sources including GeeksforGeeks, Dremio, and foundational papers. geeksforgeeks +2

3.What is trainable parameter and hyperparameters?

1. Trainable Parameters

Definition:

Trainable parameters are the internal variables of a machine learning model that the model learns and adjusts automatically during the training process to perform its task accurately.

- These parameters are **part of the model** itself and directly influence how the input data is transformed to predictions.
- In *neural networks*, trainable parameters primarily refer to the **weights** and **biases** associated with each neuron or connection.

How Trainable Parameters Work:

- When you start training a model, these parameters are initialized with random or default values.
- During training, optimization algorithms like **gradient descent** update these parameters to minimize the error between the predicted output and the actual output.
- The values of these parameters evolve gradually to better represent patterns in the training data.

Examples:

- **Weights:** Each connection from one neuron to the next has a weight determining the strength and importance of that connection.
- **Biases:** Each neuron has a bias term added to the weighted sum before applying an activation function, allowing the model to fit the data better.

2. Hyperparameters

Definition:

Hyperparameters are external configurations or settings of the machine learning model and training algorithm that you set **before** the training process begins. They are **not learned** by the model from data but are crucial for controlling the training procedure and model behavior.

Examples of Common Hyperparameters:

- **Learning Rate:** The size of the step the optimizer takes during each update of trainable parameters.
- **Batch Size:** Number of training examples processed before the model's parameters are updated.
- **Number of Epochs:** How many times the entire training dataset is passed through the model.
- **Network Architecture:** Number of layers, number of neurons per layer.
- **Dropout Rate:** Fraction of neurons randomly deactivated during training to prevent overfitting.
- **Activation Functions:** Functions applied at neurons to introduce non-linearity (e.g., ReLU, sigmoid).
- **Optimizer Type:** Algorithm used to update parameters (e.g., SGD, Adam).

Why Hyperparameters Matter:

- They influence *how* and *how well* the model trains.
- Proper tuning can significantly improve model accuracy and convergence speed.
- Poor choice can lead to slow training, convergence to bad solutions, or over/underfitting.

4.What is the role of the encoder and decoder in an autoencoder architecture?

In an autoencoder architecture, the roles of the encoder and decoder are complementary and integral to the process of learning efficient data representations:

Encoder

- The **encoder** compresses the input data into a smaller, dense representation called the **latent space** or **bottleneck**.
- Its purpose is to extract and encode the most important features or patterns from the input while reducing dimensionality.
- The encoder consists of layers (usually neural network layers) that progressively reduce the input dimensions, transforming the high-dimensional input into a compact feature vector.
- This compressed representation captures the key underlying structure of the data.

Decoder

- The **decoder** takes the compact latent representation produced by the encoder and reconstructs it back into the original data format.
- Its role is to decode or transform the latent space vector into an output that approximates the original input as closely as possible.
- The decoder consists of layers that progressively expand the compressed representation back to the original input size.
- The reconstruction quality is used to train the autoencoder, minimizing the difference between input and output (reconstruction error).

Summary

- The **encoder** learns to compress the input into a lower-dimensional encoding.
- The **decoder** learns to reconstruct the input from this encoding.
- Together, they enable the autoencoder to learn efficient, meaningful representations of data without supervision.

This architecture is widely used in tasks such as dimensionality reduction, anomaly detection, denoising, and unsupervised feature learning.

5.Explain multilayer perceptron with architecture and computations?

1. What is a Multilayer Perceptron?

A **Multilayer Perceptron (MLP)** is a **feedforward artificial neural network** made of multiple layers of neurons:

- **Input Layer** – takes the raw data.
- **One or more Hidden Layers** – perform transformations and learn patterns.
- **Output Layer** – produces the final prediction.

It's called *multilayer* because it has at least one hidden layer between the input and output.

2. Architecture of MLP

Components

1. **Input Layer**
   - One neuron per input feature.
   - Simply passes the input values to the next layer — no actual computation here.
2. **Hidden Layer(s)**
   - One or more layers of neurons between input and output.
   - Each neuron connects to *all* outputs of the previous layer (fully connected/dense).
   - Performs:
     - **Weighted sum** of inputs
     - **Adds bias**
     - **Applies activation function** (e.g., ReLU, sigmoid, tanh) to introduce non-linearity.
3. **Output Layer**
   - Takes input from the final hidden layer and computes final predictions.
   - Activation depends on task:
     - Sigmoid → binary classification
     - Softmax → multi-class classification
     - Linear → regression

3. Computations in MLP (Forward Pass)

Let's define:

- **Input vector:** $\mathbf{x} = [x_1, x_2, \ldots, x_n]$
- **Weights:** $W^{(l)}$ = weight matrix for layer l
- **Biases:** $\mathbf{b}^{(l)}$ for layer l
- **Activation function:** $\sigma(\cdot)$

Step-by-Step:

For the **first hidden layer:**

1. **Weighted sum:**

$$\mathbf{z}^{(1)} = W^{(1)}\mathbf{x} + \mathbf{b}^{(1)}$$

2. **Activation:**

$$\mathbf{a}^{(1)} = \sigma(\mathbf{z}^{(1)})$$

For the **second hidden layer:**

$$\mathbf{z}^{(2)} = W^{(2)}\mathbf{a}^{(1)} + \mathbf{b}^{(2)}$$

$$\mathbf{a}^{(2)} = \sigma(\mathbf{z}^{(2)})$$

Output Layer:

$$\mathbf{z}^{(L)} = W^{(L)}\mathbf{a}^{(L-1)} + \mathbf{b}^{(L)}$$

$$\hat{\mathbf{y}} = f(\mathbf{z}^{(L)}) \quad \text{(output activation)}$$

Numeric Example (Single Hidden Layer)

Assume:

- Inputs: x_1, x_2
- **Hidden layer:** 2 neurons (h_1, h_2)
- **Output layer:** 1 neuron

Calculations:

1. **Hidden Neuron 1:**

$$z_1 = w_{11}x_1 + w_{12}x_2 + b_1$$

$$h_1 = \sigma(z_1)$$

2. **Hidden Neuron 2:**

$$z_2 = w_{21}x_1 + w_{22}x_2 + b_2$$

$$h_2 = \sigma(z_2)$$

3. **Output Neuron:**

$$z_o = w_{o1}h_1 + w_{o2}h_2 + b_o$$

$$y = \text{sigmoid}(z_o)$$

---

4. Key Points to Remember

- All neurons between layers are **fully connected**.
- **Weights** and **biases** are **trainable parameters** — learned during training via backpropagation and gradient descent.
- **Activation functions** allow the network to learn **non-linear relationships**.
- MLPs can approximate any continuous function if big enough (*Universal Approximation Theorem*).

Summary Table

Layer	Function
Input Layer	Receives raw features and passes them to hidden layer(s)
Hidden Layer	Weighted sum → Bias → Activation → Pass to next layer
Output Layer	Computes and outputs final prediction

6.Tell about (a) Stochastic Autoencoder?
(b) Contractive Autoencoder?

(a) Stochastic Autoencoder

A **Stochastic Autoencoder** is a variant of the autoencoder where randomness is incorporated in the encoding and decoding processes. Instead of the encoder deterministically mapping input data to a single point in latent space, it models the encoding as a *probability distribution*. The decoder then samples from this distribution to reconstruct the input.

- **Key idea:** The encoder outputs parameters of a distribution (e.g., mean and variance of a Gaussian), and the latent vector is randomly sampled from this distribution.
- **Purpose:** This stochasticity allows the model to represent uncertainty and generate diverse outputs. It promotes better generalization and can be used in generative modeling.
- **Example:** Variational Autoencoders (VAEs) are a popular type of stochastic autoencoders that learn to approximate the data distribution and generate new samples.
- **Training:** Often trained by minimizing a combination of reconstruction loss (like mean squared error) and a regularization term (e.g., KL divergence) to encourage the latent space to follow a desired prior.

Thus, stochastic autoencoders explicitly incorporate randomness, helping the model learn richer latent representations and perform tasks like data generation and uncertainty estimation.

(b) Contractive Autoencoder

A **Contractive Autoencoder (CAE)** is a regularized autoencoder designed to learn robust and invariant feature representations by encouraging the encoding function to be insensitive to small perturbations of the input.

- **Key idea:** In addition to minimizing the usual reconstruction loss (difference between input and output), CAEs add a regularization term that penalizes the sensitivity of the encoder output with respect to small changes in the input.
- **How:** This penalty is implemented by adding the Frobenius norm of the Jacobian matrix of the encoder activations with respect to input to the loss function. This encourages the encoder to produce representations that "contract" around the data points.
- **Purpose:** By learning representations that are not overly sensitive to minor input noise or variations, CAEs create more stable features that enhance generalization and robustness.
- **Applications:** Feature extraction, dimensionality reduction, denoising, and tasks requiring invariant representations.
- **Advantages:** Leads to better noise robustness and smoother learned manifolds compared to basic autoencoders.

7.What is Regularization? Discuss L1, L2 regularization.

Regularization is a **technique in machine learning** used to prevent **overfitting** by adding a penalty term to the model's loss function.
The idea is to discourage the model from learning overly complex patterns that fit the training data too closely but fail to generalize to new data.

Why Regularization is Needed

- In overfitting, the model has **low training error** but **high test error**.
- This happens when the model learns noise or irrelevant patterns.
- Regularization helps by **shrinking** or **constraining** the learned parameters.

General Idea

For a model with parameters w, the original loss function (e.g., Mean Squared Error):

$$J(w) = \frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2$$

Regularization adds a penalty term:

$$J_{\text{reg}}(w) = \frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2 + \lambda \cdot \text{Penalty}(w)$$

Here:

- λ = regularization strength (hyperparameter).
- Larger λ = more penalty, simpler model.

1. L1 Regularization (Lasso Regression)

Penalty term:

$$\lambda \sum_{j=1}^{p} |w_j|$$

- Encourages **sparsity** in parameters (many weights become exactly zero).
- Good for **feature selection** (automatically removes irrelevant features).
- Loss function with L1:

$$J_{\text{L1}}(w) = \text{Loss} + \lambda \sum_{j=1}^{p} |w_j|$$

Effect on weights:

- Shrinks some coefficients to exactly zero.
- Produces a simpler model with fewer features.

2. L2 Regularization (Ridge Regression)

Penalty term:

$$\lambda \sum_{j=1}^{p} w_j^2$$

- Encourages **small** weights but rarely makes them exactly zero.
- Good for models where **all features may be relevant**.
- Loss function with L2:

$$J_{\text{L2}}(w) = \text{Loss} + \lambda \sum_{j=1}^{p} w_j^2$$

Effect on weights:

- Shrinks weights towards zero, but not exactly zero.
- Helps avoid large coefficients that can cause instability.

8.Differentiate batch, stochastic and minibatch gradient descent.

1. Batch Gradient Descent (BGD)

- **Definition:** Uses the **entire training dataset** to compute the gradient before each update.
- **Process:**
  1. Compute gradients using **all samples**.
  2. Update weights once per epoch.
- **Advantages:**
  - Stable convergence.
  - More accurate gradient estimate.
- **Disadvantages:**
  - Slow for large datasets (must load all data into memory).
  - Not suitable for online learning.
- **Update frequency: Once per epoch.**

2. Stochastic Gradient Descent (SGD)

- **Definition:** Uses **only one training sample** at a time to update the gradient.
- **Process:**
  1. Pick a random sample.
  2. Compute gradient and update weights **immediately**.
- **Advantages:**
  - Can handle large datasets (updates on the fly).
  - Faster initial learning.
- **Disadvantages:**
  - High variance in updates (loss fluctuates).
  - May overshoot minima.
- **Update frequency: Once per sample.**

3. Mini-batch Gradient Descent

- **Definition:** Uses a **small subset** (batch) of the dataset for each update.
- **Process:**
  1. Split data into small batches (e.g., 32, 64 samples).
  2. Compute gradient per batch and update weights.
- **Advantages:**
  - Faster than BGD.
  - Smoother convergence than SGD.
  - Works well with GPU parallelization.
- **Disadvantages:**
  - Requires tuning batch size.
- **Update frequency: Multiple times per epoch.**

Comparison Table

Feature	Batch GD	Stochastic GD	Mini-batch GD
Data used per update	Entire dataset	1 sample	Small batch
Memory requirement	High	Low	Medium
Update speed	Slowest	Fastest	Balanced
Convergence stability	Very stable	Noisy	Stable
Suitable for large data	No	Yes	Yes
Parallelization	Poor	Poor	Excellent

Example Update Rule

For weight w and learning rate η:

- **Batch GD:**

$$w := w - \eta \cdot \nabla_w J(w; \text{all data})$$

- **SGD:**

$$w := w - \eta \cdot \nabla_w J(w; x_i, y_i)$$

- **Mini-batch GD:**

$$w := w - \eta \cdot \nabla_w J(w; \text{batch data})$$