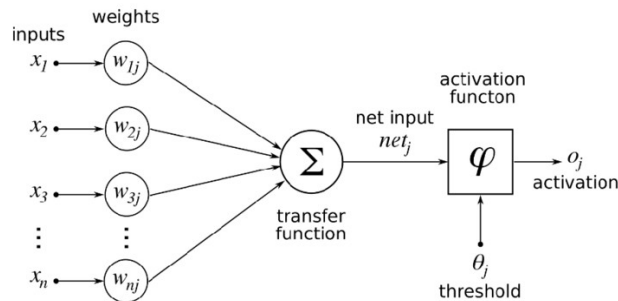


Unit 1

1. Explain the structure and working of an artificial neuron

A. The structure of artificial neural networks is inspired by biological neurons. A biological neuron has a cell body or soma to process the impulses, dendrites to receive them, and an axon that transfers them to other neurons. Similarly the input nodes of artificial neural networks receive input signals, the hidden layer nodes compute these input signals, and the output layer nodes compute the final output by processing the hidden layer's results using activation functions.



a. Inputs (x_1, x_2, \dots, x_n):

These represent the signals or data that the neuron receives. Each input corresponds to a feature of the data (for example, in an image recognition task, inputs could represent pixel values).

b. Weights (w_1, w_2, \dots, w_n):

Each input is associated with a weight. Weights are the parameters that determine the importance of each input. The weights are initially set randomly and are adjusted during training.

c. Bias (b):

The bias is an additional parameter that allows the neuron to shift its activation function, helping the model to make more accurate predictions. It can be thought of as an extra input with a fixed value of 1, helping the model to learn more complex patterns.

d. Summation (Σ):

The neuron calculates the weighted sum of the inputs.

e. Activation Function (f):

The summed value (called the net input) is passed through an activation function. The purpose of this function is to introduce non-linearity to the model, allowing it to learn more complex patterns.

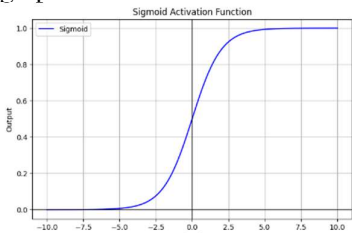
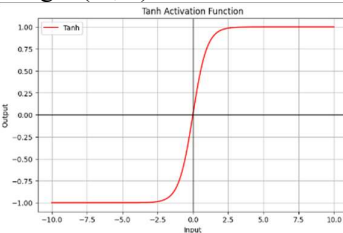
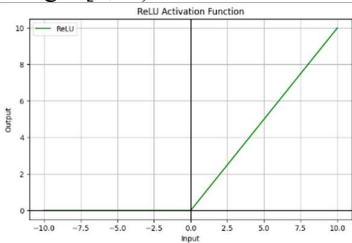
Working of an Artificial Neuron

The working of an artificial neuron can be broken down into these steps:

1. **Input Data:** The neuron receives inputs (features or signals) from the previous layer or directly from the dataset.
2. **Weighted Sum:** Each input is multiplied by its corresponding weight, and the results are summed together with the bias.
3. **Activation Function:** The summation is passed through an activation function, which determines whether the neuron should be activated or not. This function introduces the non-linearity, which is crucial for the model's ability to learn complex patterns.
4. **Output:** The result after the activation function is the output of the neuron, which can be sent to the next layer (in a multi-layer network) or as the final prediction in the case of the output layer.

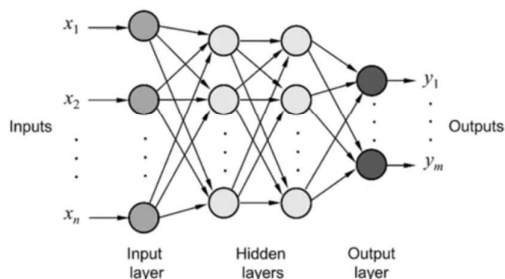
2. Compare different activation functions (ReLU, Sigmoid, Tanh) with their advantages and disadvantages.

A.

Sigmoid	Tanh	ReLU
Formula(Mathematical definition): $\sigma(x) = \frac{1}{1 + e^{-x}}$	$\tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$	$ReLU(x) = \max(0, x)$
Range: (0, 1)	Range: (-1, 1)	Range: [0, ∞)
graph 		
Advantages: <ul style="list-style-type: none"> Differentiable and smooth, making it suitable for backpropagation. Ideal for binary classification tasks (output between 0 and 1). Works well for shallow networks. 	Advantages: <ul style="list-style-type: none"> Zero-centered output, helping faster convergence during training. Better than sigmoid in hidden layers as it avoids issues caused by the range (0, 1). 	Advantages: <ul style="list-style-type: none"> Gradient is constant for positive inputs, enabling faster learning. Computationally efficient and promotes sparsity in the network.
Disadvantages: <ul style="list-style-type: none"> For large inputs, gradients become very small, slowing down learning. Non-zero-centered output can slow down optimization. 	Disadvantages: <ul style="list-style-type: none"> Like sigmoid, tanh suffers from vanishing gradients for large inputs. 	Disadvantages: <ul style="list-style-type: none"> Neurons can "die" (output zero permanently) if they get stuck in negative input regions. Unbounded output can cause exploding gradients in some cases.

3. Derive the backpropagation algorithm for a multilayer feedforward neural network

• BACKPROPAGATION algorithm



BACKPROPAGATION(*training_examples*, η , n_{in} , n_{out} , n_{hidden})

Each training example is a pair of the form (\vec{x}, \vec{t}) , where \vec{x} is the vector of network input values, and \vec{t} is the vector of target network output values.

η is the learning rate (e.g., .05). n_{in} is the number of network inputs, n_{hidden} the number of units in the hidden layer, and n_{out} the number of output units.

The input from unit i into unit j is denoted x_{ji} , and the weight from unit i to unit j is denoted w_{ji} .

- Create a feed-forward network with n_{in} inputs, n_{hidden} hidden units, and n_{out} output units.
- Initialize all network weights to small random numbers (e.g., between -.05 and .05).
- Until the termination condition is met, Do
 - For each (\vec{x}, \vec{t}) in *training_examples*, Do

Propagate the input forward through the network:

1. Input the instance \vec{x} to the network and compute the output o_u of every unit u in the network.

Propagate the errors backward through the network:

2. For each network output unit k , calculate its error term δ_k

$$\delta_k \leftarrow o_k(1 - o_k)(t_k - o_k) \quad (T4.3)$$

3. For each hidden unit h , calculate its error term δ_h

$$\delta_h \leftarrow o_h(1 - o_h) \sum_{k \in \text{outputs}} w_{kh} \delta_k \quad (T4.4)$$

4. Update each network weight w_{ji}

$$w_{ji} \leftarrow w_{ji} + \Delta w_{ji}$$

where

$$\Delta w_{ji} = \eta \delta_j x_{ji} \quad (T4.5)$$

A.

4. What is perceptron? Explain the working of the perceptron.

A. Perceptron is a type of neural network that performs binary classification that maps input features to an output decision, usually classifying data into one of two categories, such as 0 or 1. Perceptron consists of a single layer of input nodes that are fully connected to a layer of output nodes.

Working of a perceptron.

The weight is assigned to each input node of a perceptron, indicating the importance of that input in determining the output. The Perceptron's output is calculated as a weighted sum of the inputs, which is then passed through an activation function to decide whether the Perceptron will fire.

The weighted sum is computed as:

$$z = w_1x_1 + w_2x_2 + \dots + w_nx_n = X^T W$$

The step function compares this weighted sum to a threshold. If the input is larger than the threshold value, the output is 1; otherwise, it's 0. This is the most common activation function used in Perceptrons are represented by the Heaviside step function:

$$h(z) = \begin{cases} 0 & \text{if } z < \text{Threshold} \\ 1 & \text{if } z \geq \text{Threshold} \end{cases}$$

During training, the Perceptron's weights are adjusted to minimize the difference between the predicted output and the actual output. This is achieved using supervised learning algorithms like the delta rule or the Perceptron learning rule.

$$w_{i,j} = w_{i,j} + \eta(y_j - \hat{y}_j)x_i$$

The weight update formula is:

Where:

- $w_{i,j}$ is the weight between the i^{th} input and j^{th} output neuron,
 - x_i is the i^{th} input value,
 - y_j is the actual value, and \hat{y}_j is the predicted value,
 - η is the learning rate, controlling how much the weights are adjusted.
-

Unit 2

1. Compare various optimization techniques such as Adam, RMSProp, and AdaGrad.

Aspect	AdaGrad	RMSProp	Adam
Learning Rate Adaptation	Adapts learning rate for each parameter based on past gradients.	Adapts learning rate for each parameter but with an exponentially decaying average.	Combines momentum and RMSProp-like updates with bias corrections.
Update Rule	$\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{G_t + \epsilon}} \cdot g_t$	$\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{E[g^2]_t + \epsilon}} \cdot g_t$	$\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{m_t + \epsilon}} \cdot \hat{v}_t$
Memory of Past Gradients	Accumulates all past squared gradients.	Uses an exponentially decaying average of past gradients.	Uses both exponential moving averages of gradients and squared gradients.
Bias Correction	No bias correction.	No bias correction.	Yes, includes bias correction for first and second moments.
Advantages	<ul style="list-style-type: none"> - Good for sparse data. - Well-suited for simple problems with frequent updates. 	<ul style="list-style-type: none"> - Handles non-stationary objectives. - Can work well in practice for many tasks. 	<ul style="list-style-type: none"> - Combines the advantages of both momentum and RMSProp. - More robust in practice, often yields better performance.
Disadvantages	<ul style="list-style-type: none"> - Learning rate decays too quickly for long training. - Poor performance with long-term training. 	<ul style="list-style-type: none"> - Hyperparameter tuning required. - Can get stuck in local minima for complex problems. 	<ul style="list-style-type: none"> - Slightly more computationally expensive. - Can still suffer from issues like vanishing gradients.
Use Cases	<ul style="list-style-type: none"> - Sparse data. - Simple models with quick convergence. 	<ul style="list-style-type: none"> - Non-stationary or noisy data. - Recurrent neural networks (RNNs). 	<ul style="list-style-type: none"> - A wide range of tasks, including deep learning, NLP, and reinforcement learning.
Computational Complexity	Moderate (requires storing gradients for each parameter).	Moderate (requires storing squared gradients for each parameter) ↓	High (requires storing both first and second moment estimates).

2. What are the key steps in designing a deep learning model.

A. Key steps involved in designing a deep learning model are as follows:

1. Data Pre-processing

- Data pre-processing is crucial as it ensures your data is in a format suitable for deep learning algorithms. This includes cleaning the data and converting categorical variables into numerical values because deep learning models can only work with numerical data.

2. Splitting the Dataset

- In this step we divide dataset into two parts: training and testing sets. The training set is used to teach the model, while the testing set is used to evaluate the model's performance.

3. Transforming the Data

- Data transformation helps make computations more efficient. Feature scaling is a common practice where you standardize the features so they have a mean of 0 and a standard deviation of 1.

4. Building the Artificial Neural Network

- This is the important step in designing the model. One can create a deep learning model by initializing a Sequential model and adding layers.
 - Input Layer: Receives the input features from your dataset.

- Hidden Layers: Perform computations and feature extraction.
- Output Layer: Provides the final prediction or classification output.
- You may adjust the number of hidden layers and neurons in each layer depending on the complexity of the problem you are solving.

5. Running Predictions on the Test Set

- Once your model is trained, you can use it to make predictions on the testing dataset. This step checks how well the model generalizes to unseen data.

6. Evaluating with a Confusion Matrix

- After making predictions, evaluate the model's performance by using a confusion matrix. This matrix provides a detailed breakdown of the model's predictions, showing true positives, true negatives, false positives, and false negatives.
- It is an important tool to assess classification performance, especially for imbalanced datasets.

7. Improving Model Accuracy

- K-fold Cross-validation helps improve model accuracy and reduces the variance of the results. The dataset is divided into K subsets (often 10), and the model is trained on K-1 folds and tested on the remaining fold. This process repeats, and the final accuracy is the average of all iterations, giving a more reliable estimate of the model's performance.

8. Hyperparameter Tuning

- To fine-tune the model and enhance performance, Grid Search can be used. This technique systematically searches through a range of hyperparameters (e.g., number of layers, learning rate) to find the optimal combination that yields the best accuracy.

3. Discuss about the evolution and historical trends in deep learning.

A.

4. A neural network is trained using gradient descent with the following settings: Initial weight $w_0 = 0.6w$, Gradient $g = -0.5$ Learning rates: $\alpha_1 = 0.01$ and $\alpha_2 = 0.1$ Using the weight update rule: $W_{\text{new}} = W_{\text{old}} - \alpha g$. Find the updated weights for both learning rates and discuss which one would lead to faster convergence and why.

A. Given:

$W_{\text{old}} = 0.6$ (the initial weight),

$g = -0.5$ (the gradient),

$\alpha_1 = 0.01$ (the learning rate for the first case),

$\alpha_2 = 0.1$ (the learning rate for the second case).

Also Weight update rule is given by $W_{\text{new}} = W_{\text{old}} - \alpha * g$

Now,

For $\alpha_1 = 0.01$

$$\begin{aligned} W_{\text{new}} &= W_{\text{old}} - \alpha_1 * g \\ W_{\text{new}} &= 0.6 - (0.01 * -0.5) \\ W_{\text{new}} &= 0.6 + 0.005 \\ W_{\text{new}} &= 0.605 \end{aligned}$$

For $\alpha_2 = 0.1$

$$\begin{aligned} W_{\text{new}} &= W_{\text{old}} - \alpha_2 * g \\ W_{\text{new}} &= 0.6 - (0.1 * -0.5) \\ W_{\text{new}} &= 0.6 + 0.05 \\ W_{\text{new}} &= 0.65 \end{aligned}$$

Learning rate $\alpha_1=0.01$: With a smaller learning rate, the update to the weight is small, leading to slower progress towards the optimal weight. In the early stages of training, this results in more stable but slow convergence.

Learning rate $\alpha_2=0.1$: With a larger learning rate, the update to the weight is larger, allowing the neural network to make quicker adjustments to the weights and thus potentially converge faster. However, if the learning rate is too large, the updates may overshoot the optimal value and lead to instability or oscillations.

$\alpha_2=0.1$ will converge faster, but it carries a higher risk of instability if the learning rate is too large.

$\alpha_1=0.01$ will converge more slowly but may be more stable in the long run

Unit 3

1) Compare different Deep CNN architectures (LeNet, AlexNet, VGG, PlacesNet,).

A.

Architecture	LeNet	AlexNet	VGG	PlacesNet
Creator	Yann LeCun et al.	Alex Krizhevsky, Ilya Sutskever, Geoffrey Hinton	Visual Geometry Group, University of Oxford	MIT and Harvard University
Year	1998	2012	2014	2016
Primary Purpose	Handwritten digit classification (MNIST)	Image classification (ImageNet)	Image classification (ImageNet)	Scene recognition (Places dataset)
Architecture Depth	7 layers	8 layers (5 convolutional + 3 fully connected)	16/19 layers (all convolutional + fully connected)	Similar to VGG, but optimized for scene recognition
Input Size	32x32 grayscale	224x224 RGB	224x224 RGB	224x224 RGB
Convolutional Layers	2 convolutional layers	5 convolutional layers	13 convolutional layers	Similar to VGG, optimized for scene features
Fully Connected Layers	2 fully connected layers	3 fully connected layers	3 fully connected layers	Fully connected layers similar to VGG
Pooling	Average pooling	Max pooling	Max pooling	Max pooling
Activation Function	Sigmoid/Tanh	ReLU	ReLU	ReLU
Regularization	None	Dropout, LRN	None	Dropout, batch normalization
Key Features	Shallow network, simple structure	Deep CNN, GPU optimization, ReLU, Dropout	Deep network with 3x3 filters, consistent design	Optimized for scene recognition, high performance in large-scale datasets
Limitations	Small scale, lacks depth for complex tasks	Large, computationally expensive, outdated techniques like LRN	Large number of parameters, computationally expensive	Designed for scenes, not as versatile for general image classification
Notable Use Case	Digit classification (MNIST)	ImageNet classification, object detection	ImageNet classification	Scene classification, Places365 dataset

2) Discuss about pooling layers in CNNs. How does max pooling differ from average pooling?

A. Pooling layer is used in CNNs to reduce the spatial dimensions (width and height) of the input feature maps while retaining the most important information. It involves sliding a two-dimensional filter over each channel of a feature map and summarizing the features within the region covered by the filter.

Purpose of pooling layers

- ➔ Dimensionality Reduction:
- ➔ Translation Invariance
- ➔ Preventing Overfitting
- ➔ Extracting Dominant Features

The primary difference between max pooling and average pooling is Max pooling selects the maximum value within each region as the output, while average pooling calculates the average value.

-> Max pooling helps retain important features like edges, corners, and textures. Max pooling also tends to be more robust in practice, as it is more likely to retain critical features even when the image undergoes slight translations or distortions.

-> Average pooling provides a smoother, more generalized representation of the input features. It is less likely to highlight outliers or extreme values.

-> Max Pooling is more commonly used in most CNN architectures, especially in image classification tasks. On the other hand Average Pooling is less commonly used
