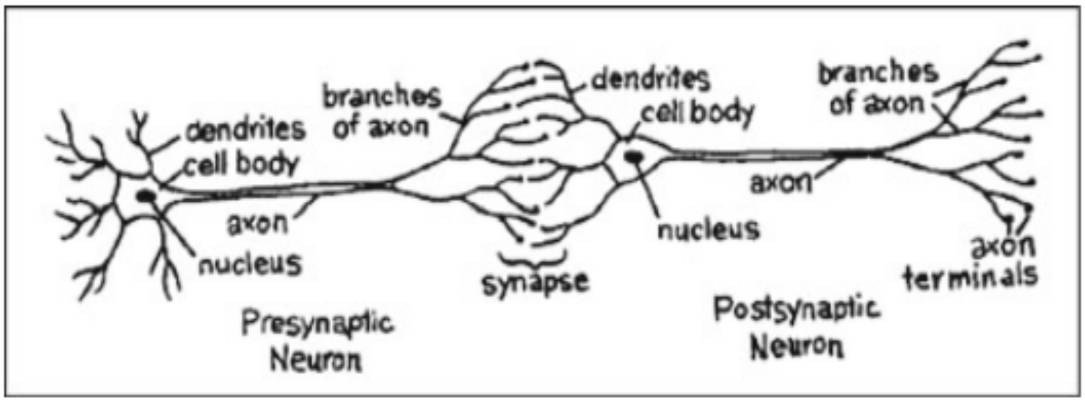
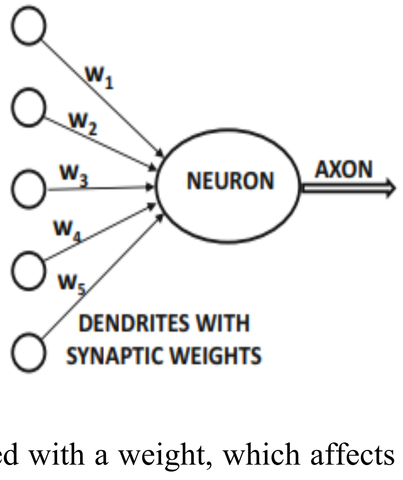
MODULE 1

INTRODUCTION

Artificial neural networks are popular machine learning techniques that simulate the mechanism of learning in biological organisms. The human nervous system contains cells, which are referred to as neurons. The neurons are connected to one another with the use of axons and dendrites, and the connecting regions between axons and dendrites are referred to as synapses. These connections are illustrated in the following figure.



The strengths of synaptic connections often change in response to external stimuli. This change is how learning takes place in living organisms. This biological mechanism is simulated in artificial neural networks, which contain computation units that are referred to as neurons. The computational units are connected to one another through weights, which serve the same role as the strengths of synaptic connections in biological organisms.



Each input to a neuron is scaled with a weight, which affects the function computed at that unit. This architecture is illustrated in the above figure. An artificial neural network computes a function of the inputs by propagating the computed values from the input neurons to the output neuron(s) and using the weights as intermediate parameters.

Learning occurs by changing the weights connecting the neurons. Just as external stimuli are needed for learning in biological organisms, the external stimulus in artificial neural networks

is provided by the training data containing examples of input-output pairs of the function to be learned.
For example, the training data might contain pixel representations of images (input) and their annotated labels (e.g., carrot, banana) as the output. These training data pairs are fed into the neural network by using the input representations to make predictions about the output labels. The training data provides feedback to the correctness of the weights in the neural network depending on how well the predicted output (e.g., probability of carrot) for a particular input matches the annotated output label in the training data. One can view the errors made by the neural network in the computation of a function as a kind of unpleasant feedback in a biological organism, leading to an adjustment in the synaptic strengths. Similarly, the weights between neurons are adjusted in a neural network in response to prediction errors. *The goal of changing the weights is to modify the computed function to make the predictions more correct in future iterations*. Therefore, if the neural network is trained with many different images of bananas, it will eventually be able to properly recognize a banana in an image it has not seen before. This ability to accurately compute functions of unseen inputs by training over a finite set of input-output pairs is referred to as **model generalization.**

Deep learning is a subset of machine learning that uses artificial neural networks to model and understand complex patterns in data. It involves algorithms that learn to perform tasks by ingesting vast amounts of labeled or unlabeled data and gradually improving their performance through iterations.

THE BASIC ARCHITECTURE OF NEURAL NETWORKS

Single-layer and multi-layer neural networks: In the single layer network, a set of inputs is directly mapped to an output by using a generalized variation of a linear function. This simple instantiation of a neural network is also referred to as the perceptron. In multi-layer neural networks, the neurons are arranged in layered fashion, in which the input and output layers are separated by a group of hidden layers. This layer-wise architecture of the neural network is also referred to as a feed-forward network.

- **Single Computational Layer: The Perceptron**

The simplest neural network is referred to as the perceptron. This neural network contains a single input layer and an output node. The basic architecture of perceptron is shown in Fig 1. Consider a situation where each training instance is of the form (X,y), where each X = [x1,...xd] contains d feature variables, and $y \in \{-1, +1\}$ contains the observed value of the binary class variable.

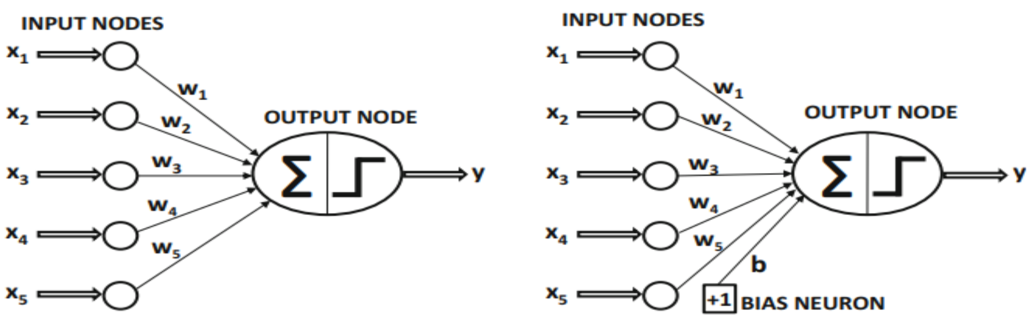


Fig 1: Perceptron with bias and without bias

The input layer contains d nodes that transmit the d features X = [x1 ...xd] with edges of weight W = [w1 ...wd] to an output node. The input layer does not perform any computation in its own right. The linear function $\overline{W} \cdot \overline{X} = \sum_{i=1}^{d} w_i x_i$ is computed at the output node. Subsequently, the sign of this real value is used in order to predict the dependent variable of X.

Therefore, the prediction ŷ is computed as follows:

$$\hat{y} = \text{sign}\{\overline{W} \cdot \overline{X}\} = \text{sign}\{\sum_{j=1}^{d} w_j x_j\}$$

The sign function maps a real value to either +1 or −1, which is appropriate for binary classification. The error of the prediction is therefore E(X) = y − ŷ. In cases where the error value E(X) is nonzero, the weights in the neural network need to be updated in the (negative) direction of the error gradient.

- **Multilayer Neural Networks**

Multi-Layer Perceptron Neural Network is a Neural Network with multiple layers, and all its layers are connected. It uses a Back Propagation algorithm for training the model. This network has three main layers that combine to form a complete Artificial Neural Network. These layers are as follows:

Input Layer

It takes input from the training data set and forwards it to the hidden layer. There are n input nodes in the input layer. The number of input nodes depends on the number of dataset features. Each input vector variable is distributed to each of the nodes of the hidden layer.

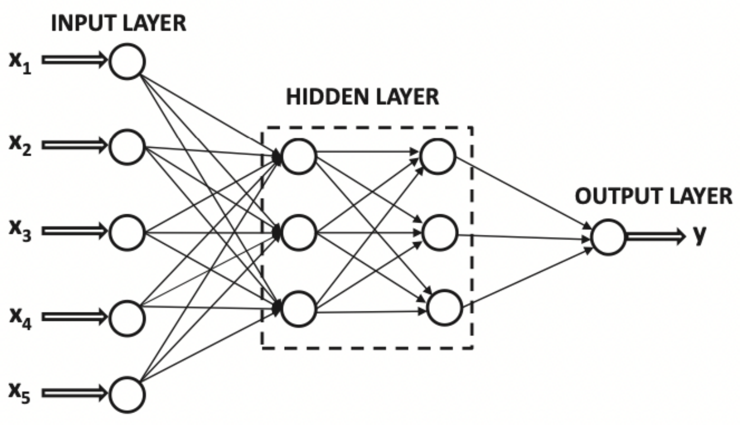


Fig 2: Multi-Layer Perceptron

Hidden Layer

It is the heart of all Artificial neural networks. This layer comprises all computations of the neural network. The edges of the hidden layer have weights multiplied by the node values. This layer uses the activation function. There can be one or two hidden layers in the model.

Several hidden layer nodes should be accurate as few nodes in the hidden layer make the model unable to work efficiently with complex data. More nodes will result in an overfitting problem.

Output Layer

This layer gives the estimated output of the Neural Network. The number of nodes in the output layer depends on the type of problem. For a single targeted variable, use one node. N classification problem, ANN uses N nodes in the output layer.

Working of Multi-Layer Perceptron Neural Network

- The input node represents the feature of the dataset.
- Each input node passes the vector input value to the hidden layer.
- In the hidden layer, each edge has some weight multiplied by the input variable. All the production values from the hidden nodes are summed together. To generate the output
- The activation function is used in the hidden layer to identify the active nodes.
- The output is passed to the output layer.
- Calculate the difference between predicted and actual output at the output layer.
- The model uses backpropagation after calculating the predicted output.

Advantages of MultiLayer Perceptron Neural Network

- MultiLayer Perceptron Neural Networks can easily work with non-linear problems.
- It can handle complex problems while dealing with large datasets.
- Developers use this model to deal with the fitness problem of Neural Networks.
- It has a higher accuracy rate and reduces prediction error by using backpropagation.
- After training the model, the Multilayer Perceptron Neural Network quickly predicts the output.

Disadvantages of MultiLayer Perceptron Neural Network

- This Neural Network consists of large computation, which sometimes increases the overall cost of the model.
- The model will perform well only when it is trained perfectly.
- Due to this model’s tight connections, the number of parameters and node redundancy increases.

TRAINING A NEURAL NETWORK WITH BACKPROPAGATION

Backpropagation is a fundamental concept in training neural networks. It's an algorithm used to calculate the gradients of the loss function with respect to the weights of the network, enabling the optimization of these weights to minimize the error or loss. The steps involved are

Forward Pass: During the forward pass, input data is fed into the neural network. The data propagates through the network layer by layer, transforming weighted sums and activation functions. At each layer, the input is multiplied by weights, summed up, and then passed through an activation function, generating outputs that progressively move toward the final prediction.
Loss Computation: Once the forward pass is complete and the network produces an output, the output is compared to the actual target or ground truth. The difference between the predicted output and the actual target is quantified using a loss function, which measures the network's performance.

Backward Pass (Backpropagation): The goal of backpropagation is to calculate the gradient of the loss function with respect to each weight in the network.

- The algorithm computes the gradient (partial derivative) of the loss function with respect to each weight in the network.
- This involves applying the chain rule of calculus to compute these gradients layer by layer, starting from the output layer and moving backward to the input layer. This process continues backward through the layers until the algorithm reaches the input layer, providing gradients for each weight.

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

If $\hat{y} = f(z)$ where $z = Wx + b$, then:

$$\frac{\partial \hat{y}}{\partial W} = f'(z) \cdot x$$

So:

$$\frac{\partial L}{\partial W} = (\hat{y} - y) \cdot f'(z) \cdot x$$

The error calculation gives us a measure of how far off our predictions are, and the gradients tell us how to change the weights to reduce this error.

Weight Update: Once the gradients of the loss function with respect to the weights are calculated, the network updates its weights using an optimization algorithm like stochastic gradient descent (SGD), Adam, or RMSprop. Weight update rule is as follows

The weight update rule using gradient descent is:

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

where η is the learning rate.

So:

$$W \leftarrow W - \eta \cdot (\hat{y} - y) \cdot f'(z) \cdot x$$

Iteration: The forward pass, backward pass, and weight updates are repeated iteratively for multiple epochs until the network's performance converges or reaches a satisfactory level.

ACTIVATION FUNCTIONS

Sign, Sigmoid, Tanh, ReLU, leaky ReLU, Hard Tanh, Softmax.

An activation function in neural networks is a mathematical operation that determines the output of a node or neuron in the network, typically after it has processed its inputs. It introduces non-linear properties to the network, allowing neural networks to learn and represent more intricate patterns in data, making them capable of solving a wider range of problems.
Popular activation functions include sigmoid, tanh, ReLU (Rectified Linear Unit), and variants like Leaky ReLU and ELU (Exponential Linear Unit). These functions manipulate the input signal and decide whether and to what extent the signal should be transmitted to the next layer in the neural network.
The choice of activation function is a critical part of neural network design. In the case of the perceptron, the choice of the sign activation function is motivated by the fact that a binary class label needs to be predicted. However, it is possible to have other types of situations where different target variables may be predicted. For example, if the target variable to be predicted is real, then it makes sense to use the identity activation function, and the resulting algorithm is the same as least-squares regression. If it is desirable to predict a probability of a binary class, it makes sense to use a sigmoid function for activating the output node, so that the prediction indicates the probability that the observed value, y, of the dependent variable is 1.
The importance of nonlinear activation functions becomes significant when one moves from the single-layered perceptron to the multi-layered. Different types of nonlinear functions such as the sign, sigmoid, or hyperbolic tangents may be used in various layers.
We use the notation Φ to denote the activation function. The value computed before applying the activation function Φ(·) will be referred to as the pre-activation value, whereas the value computed after applying the activation function is referred to as the post-activation value. The output of a neuron is always the post-activation value, although the pre-activation variables are often used in different types of analyses, such as the computations of the backpropagation algorithm.

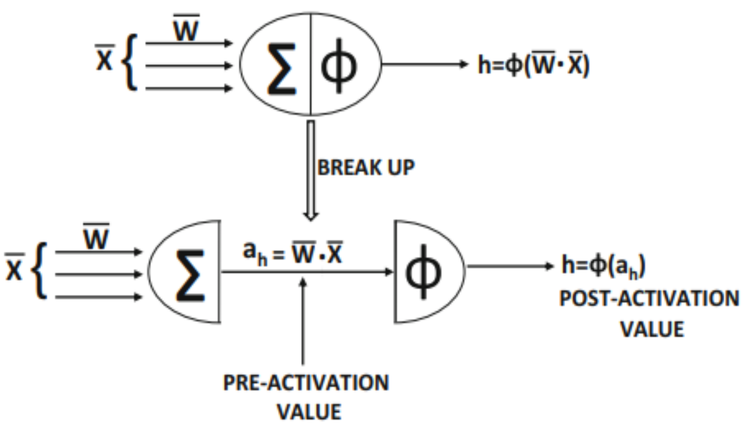


Fig: Pre-activation and Post-activation

- **Identity**

$$\Phi(v) = v$$

The most basic activation function Φ(·) is the identity or linear activation, which provides no nonlinearity: Φ(v) = v The linear activation function is often used in the output node, when the target is a real value.

- **Sign**

The sign activation can be used to map to binary outputs at prediction time, its non-differentiability prevents its use for creating the loss function at training time.

- **Sigmoid**

The sigmoid activation outputs a value in (0, 1), which is helpful in performing computations that should be interpreted as probabilities. Furthermore, it is also helpful in creating probabilistic outputs and constructing loss functions derived from maximum-likelihood models.

- **Tanh**

The tanh function has a shape similar to that of the sigmoid function, except that it is horizontally re-scaled and vertically translated/re-scaled to [−1, 1]. The tanh function is preferable to the sigmoid when the outputs of the computations are desired to be both positive and negative.

$$\Phi(v) = \text{sign}(v) \text{ (sign function)}$$
$$\Phi(v) = \frac{1}{1+e^{-v}} \text{ (sigmoid function)}$$
$$\Phi(v) = \frac{e^{2v}-1}{e^{2v}+1} \text{ (tanh function)}$$

- **ReLU**

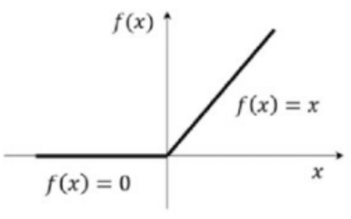
The ReLU and hard tanh activation functions have largely replaced the sigmoid and soft tanh activation functions in modern neural networks because of the ease in training multilayered neural networks with these activation functions.

$$\Phi(v) = \max\{v, 0\} \text{ (Rectified Linear Unit [ReLU])}$$
$$\Phi(v) = \max\{\min[v, 1], -1\} \text{ (hard tanh)}$$

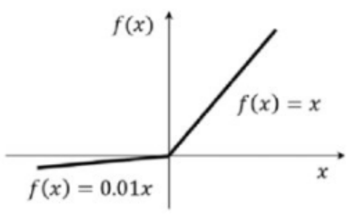
- **Leaky ReLU**

In a typical ReLU function, when the input is negative, the output is zero, which can cause some neurons to stop learning if they consistently output zero. Leaky ReLU resolves this by allowing a small, non-zero gradient for negative inputs. The function is defined as:

$$f(x) = \begin{cases} x & \text{if } x > 0 \\ \alpha x & \text{if } x \leq 0 \end{cases}$$



ReLU activation function



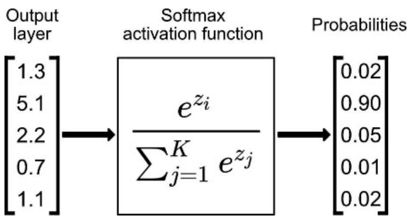
LeakyReLU activation function

- **Softmax**

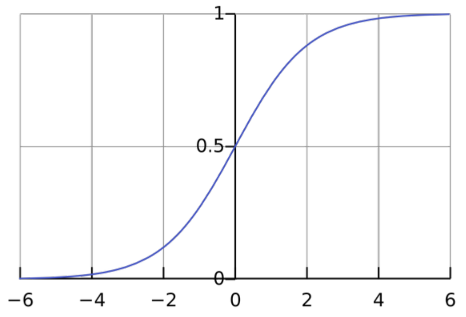
The choice and number of output nodes is also tied to the activation function, which in turn depends on the application at hand. For example, if k-way classification is intended, k output values can be used, with a softmax activation function with respect to outputs v = [v1,...,vk] at the nodes in a given layer.

$$\Phi(\overline{v})_i = \frac{\exp(v_i)}{\sum_{j=1}^{k} \exp(v_j)} \quad \forall i \in \{1, \ldots, k\}$$

The use of softmax with a single hidden layer of linear activations exactly implements a model, which is referred to as multinomial logistic regression



Output layer
1.3
5.1
2.2
0.7
1.1
Softmax activation function
$\frac{e^{z_i}}{\sum_{j=1}^{K} e^{z_j}}$
Probabilities
0.02
0.90
0.05
0.01
0.02



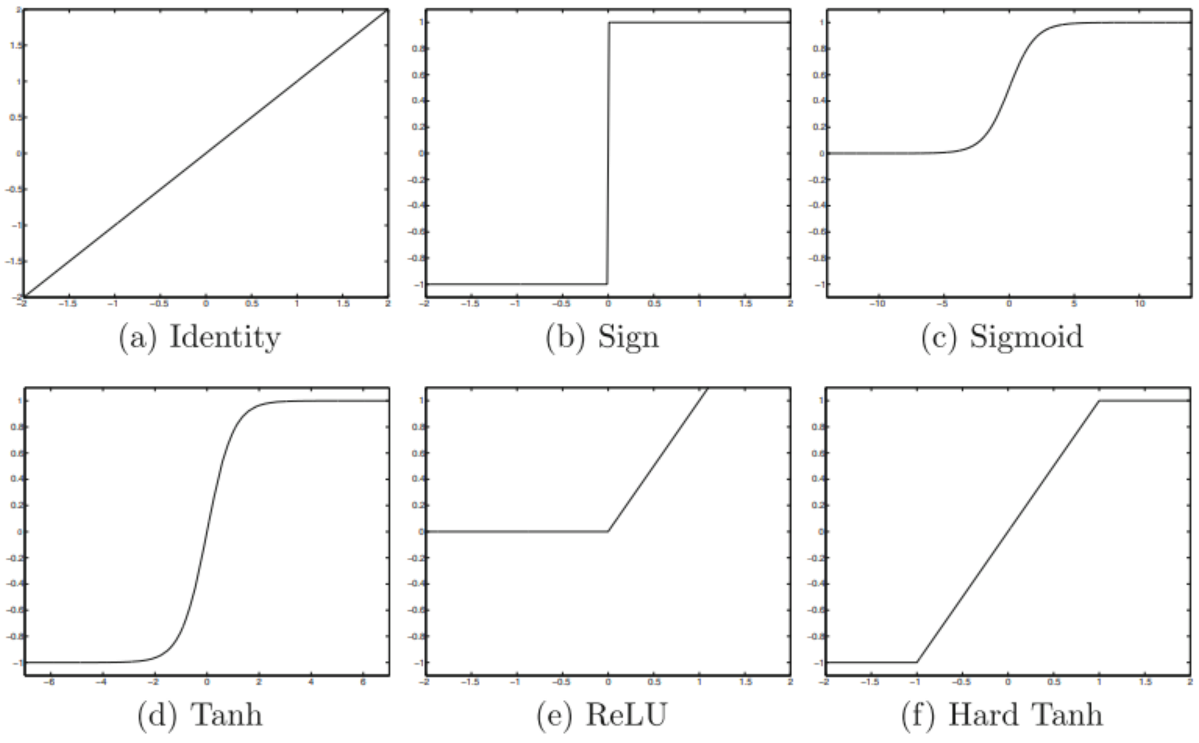


Fig 3: Various Activation function

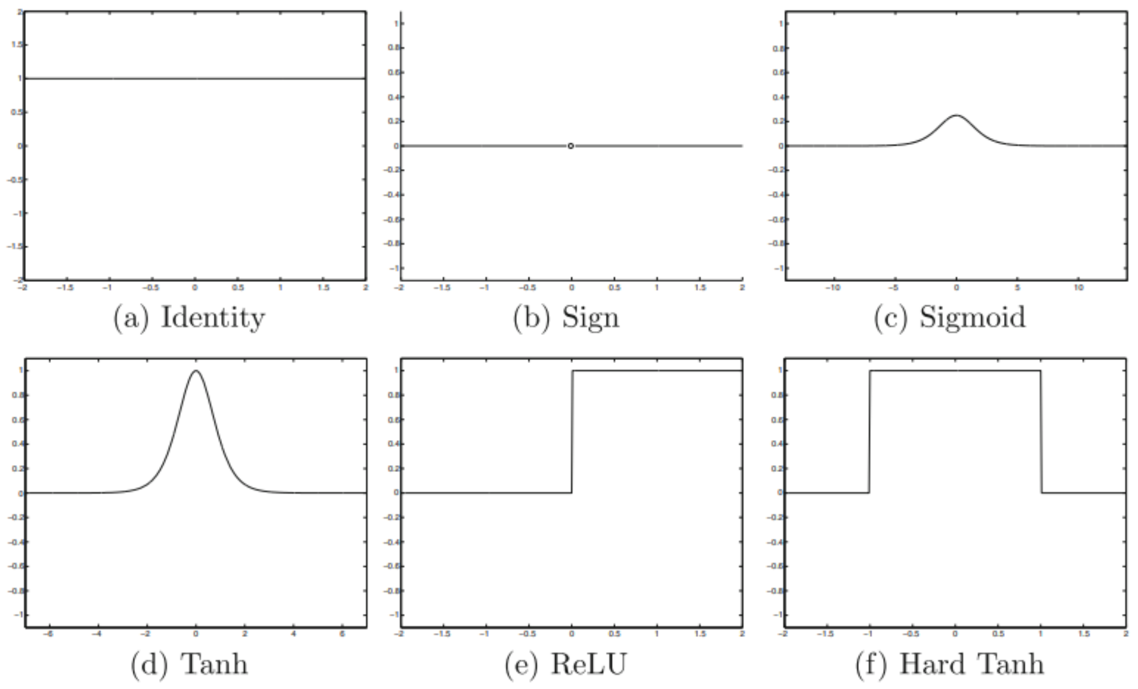


Fig 4: Derivatives of various Activation function

LOSS FUNCTION

Loss functions, also known as cost functions or objective functions, are used in neural nets to measure how well a model's predictions match the actual data. The choice of loss function depends on the type of problem being solved (e.g., regression, classification) and the specific requirements of the task.
During the training process of a machine learning model, the loss function is used to compute the error for each prediction compared to the actual target. The optimization algorithm (e.g., gradient descent) then uses this error to update the model's parameters in a way that minimizes the loss function. The process is iterative, with the goal of finding parameter values that result in the lowest possible loss.

Here are some common loss functions used in machine learning**.**

REGRESSION LOSS FUNCTIONS

1. **Mean Squared Error (MSE):**

MSE is primarily used for regression problems where the goal is to predict continuous values. MSE is a convex function, meaning it has a single global minimum. This property ensures that optimization algorithms like gradient descent can converge to the optimal solution.

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

2. **Mean Absolute Error (MAE):**

MAE is less sensitive to outliers because it uses the absolute value of the errors instead of squaring them(as in MSE). This makes MAE a better choice when dealing with data that contains outliers. MAE is not differentiable at zero, which can make optimization more challenging for certain algorithms.

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

CLASSIFICATION LOSS FUNCTIONS

3. **Binary Cross-Entropy Loss (Log Loss):**

Binary Cross-Entropy (BCE) Loss, also known as Log Loss, is a loss function used for binary classification tasks. It measures the performance of a classification model whose output is a probability value between 0 and 1.

BCE is commonly used as the loss function for binary classification in neural networks, often in conjunction with a sigmoid activation function in the output layer.

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

4. **Categorical Cross-Entropy Loss:**

Categorical Cross-Entropy (CCE) Loss, also known as Softmax Loss or Multi-Class Cross-Entropy Loss, is used for multi-class classification tasks. It measures the performance of a classification model whose output is a probability distribution over multiple classes.
CCE is used in tasks where an instance can belong to one of multiple classes, such as image classification (e.g., identifying different objects in images), text classification, and more. CCE is commonly used often in conjunction with a softmax activation function in the output layer.

$$\text{CCE} = -\frac{1}{n}\sum_{i=1}^{n}\sum_{c=1}^{C} y_{ic}\log(\hat{y}_{ic})$$

5. **Hinge Loss:**

Hinge Loss, also known as large margin loss, is primarily used for training classifiers such as Support Vector Machines (SVMs). Hinge loss introduces a margin of 1. If the predicted score for the correct class is greater than or equal to 1, the loss is zero. Otherwise, the loss increases linearly as the score moves away from 1. Hinge loss is not designed for probabilistic outputs as well as multi-class classification.

$$L(y, \hat{y}) = \max(0, 1 - y\hat{y})$$

6. **Kullback-Leibler Divergence (KL Divergence):**

KL Divergence Loss is a powerful tool in machine learning for comparing probability distributions. It is especially useful in models where the output is a probability distribution, such as in variational autoencoders and other generative models.

$$\mathrm{KL}(P \parallel Q) = \sum_i P(i) \log \frac{P(i)}{Q(i)}$$

PRACTICAL ISSUES IN NEURAL NETWORK TRAINING

There are considerable challenges remain with respect to actually training neural networks to provide high level of performance. These challenges are primarily related to several practical problems associated with training, the most important one of which is overfitting and other issues are The Vanishing and Exploding Gradient Problems, Difficulties in Convergence, Local and Spurious Optima, and Computational Challenges

1. **OVERFITTING**

The problem of overfitting refers to the fact that fitting a model to a particular training data set does not guarantee that it will provide good prediction performance on unseen test data, even if the model predicts the targets on the training data perfectly. In other words, there is always a gap between the training and test data performance, which is particularly large when the models are complex and the data set is small.
Increasing the number of training instances improves the generalization power of the model, whereas increasing the complexity of the model reduces its generalization power. At the same time, when a lot of training data is available, simple model is unlikely to capture complex relationships between the features and target.

A good rule of thumb is that the total number of training data points should be at least 2 to 3 times larger than the number of parameters in the neural network, although the precise number of data instances depends on the specific model at hand. In general, models with a larger number of parameters are said to have high capacity, and they require a larger amount of data in order to gain generalization power to unseen test data. Thus, neural networks require careful design to minimize the harmful effects of overfitting, even when a large amount of data is available.
Some methods used to mitigate the impact of overfitting are

Simplify the Model: Use a less complex model with fewer parameters.

Regularization: involves adding a penalty term to the loss function to prevent overfitting, typically using the squared value of parameters weighted by a regularization parameter λ

- L1 Regularization (Lasso)**:** Adds a penalty equal to the absolute value of the magnitude of coefficients.
- L2 Regularization (Ridge)**:** Adds a penalty equal to the square of the magnitude of coefficients.

Pruning: For decision trees, reduce the size of the tree by removing sections that provide little power.

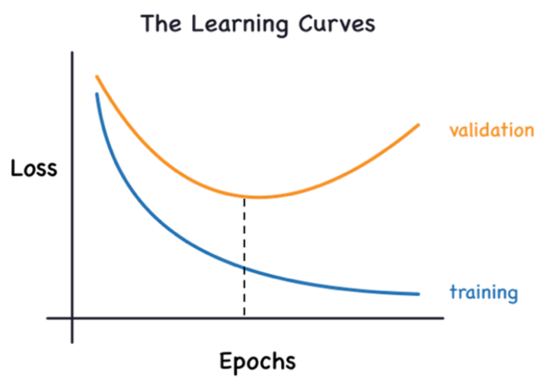
Early Stopping: Stop training when the performance on a validation set starts to deteriorate.

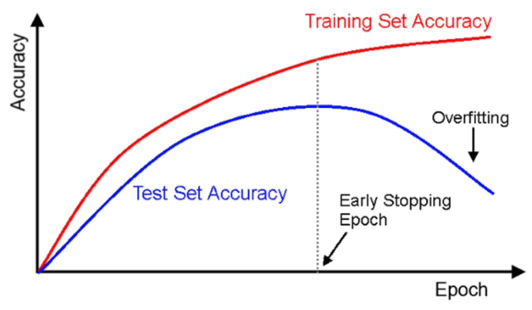
Cross-Validation: Use cross-validation techniques to ensure the model generalizes well.

Ensemble methods like **Bootstrap Aggregating (Bagging)**: Train multiple models on different subsets of the training data and combine their outputs.

Dropout: In neural networks, randomly drop neurons during training to prevent co-adaptation.

Data Augmentation: Increase the size of the training data by augmenting existing data (e.g., rotating, flipping images).





Generalization is the model's ability to make accurate predictions on new, unseen data that has the same characteristics as the training set. However, if your model is not able to generalize well, you are likely to face overfitting or underfitting problems.

2. THE VANISHING AND EXPLODING GRADIENT PROBLEMS

Vanishing gradient problem:

In deep neural networks, especially those with many layers, gradients can become very small as they are propagated backward through the network during training. This problem occurs because the derivatives of activation functions, particularly in sigmoid and tanh, can be very small in certain regions of their input domain. When these small derivatives are multiplied during backpropagation, the gradients can shrink exponentially, leading to very slow learning or no learning at all for the early layers of the network.

- When using activation functions like sigmoid or tanh, which squash input values to a small range (e.g., between 0 and 1 for sigmoid), the derivatives of these functions are also very small for large input values.
- The gradients are multiplied together (chain rule) as they are backpropagated through each layer. If the gradients are small (less than 1), their product becomes smaller and smaller as they are propagated backward.

As a result of the vanishing gradients, weights in the early layers of the network update very slowly, which makes training deep networks very difficult and time-consuming.

Solutions to Vanishing Gradient Problem:

- Use activation functions like ReLU, Leaky ReLU and its variants
- Initialization Techniques like Xavier/Glorot Initialization, He Initialization
- Batch Normalization
- Residual Networks

Exploding gradient problem:

The exploding gradient problem is the counterpart to the vanishing gradient problem. It occurs when gradients grow exponentially during backpropagation, causing very large updates to the model weights. This can make the training process unstable and prevent the model from converging.

Solutions to Exploding Gradient Problem

- Gradient Clipping: Limits the value of gradients during backpropagation to prevent them from becoming too large.
- Weight Regularization: L2 Regularization
- Proper Weight Initialization:
    - He Initialization: Suitable for ReLU and its variants.
    - Xavier/Glorot Initialization: Suitable for sigmoid and tanh.
- Batch Normalization:
- Residual Networks (ResNets):
- Learning Rate Adjustment:

Vanishing and exploding gradient problems, arise due to unstable updates in earlier layers caused by the chain rule during backpropagation. When local derivatives along a path have expected values less than 1, the gradient diminishes exponentially, while values greater than 1 lead to a gradient explosion. Even with expected values exactly at 1, instability persists based on their actual distribution. This inherent instability complicates the training process of deep networks. Solutions like ReLU activations, ensuring a derivative of 1 for positive values, help mitigate the vanishing gradient problem, unlike sigmoid activations prone to causing it.

3. DIFFICULTIES IN CONVERGENCE

Sufficiently fast convergence of the optimization process is difficult to achieve with very deep networks, as depth leads to increased resistance to the training process in terms of letting the gradients smoothly flow through the network. This problem is somewhat related to the vanishing gradient problem, but has its own unique characteristics. Gating networks and residual networks can solve this issue.

4. LOCAL AND SPURIOUS OPTIMA

Neural networks have many ups and downs in their path to finding the best solution. When there are lots of options to consider, it's smart to choose starting points carefully. Pretraining, a method like a warm-up exercise, helps by training smaller parts first. It ignores unnecessary details initially and helps prevent focusing too much on one type of result during training. This method aims for better overall performance by finding better starting points for the real test. In neural networks, the twists and turns during the training journey are often more challenging than getting stuck in one specific spot(local minima).

5. COMPUTATIONAL CHALLENGES

Training neural networks, especially in text and image processing, often takes weeks, posing a significant challenge. Advances in hardware, like GPUs, have notably accelerated these processes. The emphasis on leveraging modern hardware capabilities has driven a lot of the recent gains in deep learning. Interestingly, most computational load occurs during training rather than prediction, which is typically time-critical. Techniques have emerged to compress trained networks for mobile or space-constrained settings

6. UNDERFITTING

Underfitting occurs in machine learning when a model is too simple to capture the underlying patterns in the training data. This results in poor performance on both the training data and new, unseen data.

Causes of Underfitting

1. Overly Simple Model: The model lacks complexity, such as using a linear model for non-linear data.
2. Insufficient Training Time: The model has not been trained for enough epochs.
3. Low Number of Features: The model lacks important features that capture the underlying patterns.

Techniques to Mitigate Underfitting

- Increase Model Complexity:

Use a more complex model (e.g., moving from linear regression to polynomial regression).Add more layers or units in neural networks.

- Feature Engineering:

Create new features that better capture the underlying patterns. Use feature selection techniques to identify and include important features.

- Train Longer:

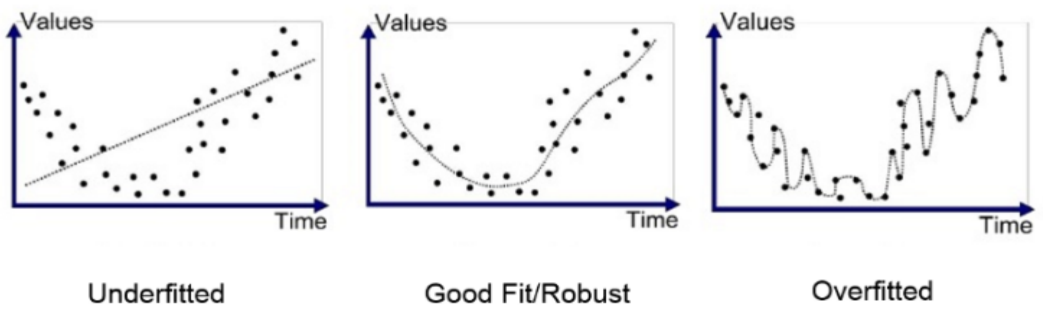
Increase the number of training epochs to allow the model more time to learn.

- Use Advanced Algorithms:

Switch to more sophisticated algorithms that can capture complex patterns (e.g., decision trees, ensemble methods)

- Eliminate noise from data

Another cause of underfitting is the existence of outliers and incorrect values in the dataset. Data cleaning techniques can help deal with this problem.



HYPER PARAMETERS AND VALIDATION SETS

HYPER-PARAMETERS

Hyper-parameters in deep learning are the configuration settings that determine the architecture and behaviour of a neural network model. Properly tuning these hyperparameters is crucial for training a neural network effectively. Some of the most important hyperparameters in deep learning:

1**. Learning Rate (α):** The learning rate determines the step size at which the model's weights are updated during training. A higher learning rate may result in faster convergence but risk overshooting the optimal weights, while a lower learning rate may lead to slower convergence or getting stuck in local minima.

2**. Number of Epochs**: An epoch is one complete pass through the entire training dataset. An epoch involves using every sample in the training dataset exactly once to update the model's weights. The number of epochs defines how many times the model will see the entire dataset during training. Too few epochs may result in underfitting, while too many may lead to overfitting.

3. **Batch Size**: The batch size determines the number of data samples used in each forward and backward pass during training. Smaller batch sizes lead to more frequent weight updates but may require more training time. Larger batch sizes can speed up training but might require more memory. During an epoch, the dataset is often divided into smaller batches, which are fed to the model sequentially. This is especially useful for large datasets that cannot be loaded into memory at once.

4. **Network Architecture:** This includes the number of layers, the type of layers (e.g., convolutional, recurrent, dense), and the number of neurons or units in each layer. Choosing the right architecture for your problem is critical.

5. **Activation Functions:** Activation functions introduce non-linearity into the model. Common choices include ReLU (Rectified Linear Unit), Sigmoid, and Tanh. The choice of activation function depends on the nature of the problem.
6. **Dropout Rate:** Dropout is a regularization technique used to prevent overfitting in neural networks by randomly dropping units (neurons) along with their connections during training. This helps the model generalize better by reducing reliance on specific neurons and encouraging the network to learn more robust features.
7. **Weight Initialization:** Weight initialization is a crucial step in training neural networks, as it sets the starting values for the weights of the connections between neurons. Proper weight initialization can significantly impact the convergence speed and performance of a neural network model. Common initialization methods include random initialization, Xavier/Glorot initialization, and He initialization.
8. **Optimizer:** An optimizer in machine learning is an algorithm or method used to adjust the weights and biases of a neural network during training to minimize the loss function. Optimizers play a crucial role in the training process, as they dictate how the model learns from the data. The choice of optimization algorithm, such as Adam, SGD (Stochastic Gradient Descent), RMSprop, etc., affects how the model's weights are updated during training.
9**. Loss Function:** The loss function measures the error between the predicted output and the actual target values. Common loss functions include mean squared error (MSE), categorical cross-entropy, and binary cross-entropy, depending on the problem type.
10**. Regularization Techniques:** Techniques like L1 and L2 regularization (weight decay), as well as batch normalization and early stopping, can be used to prevent overfitting.
11**. Learning Rate Schedule:** Instead of a fixed learning rate, you can use schedules like learning rate decay or adaptive learning rates to fine-tune the learning process as training progresses.
12. **Momentum:** Momentum is a hyperparameter for optimizers like SGD with momentum. It determines the effect of past gradients on the current update step.
13. **Mini-batch Selection Strategy**: In some cases, how you sample mini-batches from your dataset (randomly, by class, etc.) can impact training.
14. **Data Augmentation:** For image data, data augmentation techniques like rotation, scaling, and cropping can be considered as hyperparameters.
15**. Early Stopping Criteria:** This determines when to stop training to prevent overfitting. It involves monitoring a validation metric, like validation loss, and stopping when it starts to degrade.
Tuning these hyperparameters often involves experimentation and using techniques like grid search, random search, or more advanced optimization algorithms like Bayesian optimization. The goal is to find the best combination of hyperparameters that results in a well-performing and well-generalized model for your specific task

Validation Sets

We split the training data into two disjoint subsets. One of these subsets is used to learn the parameters. The other subset is our validation set, used to evaluate the performance of a model during the training process, ensuring that it generalizes well to new, unseen data.

The subset of data used to learn the parameters is still typically called the training set. The subset of data used to guide the selection of hyperparameters is called the validation set. Typically, one uses about 80% of the training data for training and 20% for validation.

Purpose of Validation Data

1. Model Evaluation: Helps assess how well the model is performing and whether it is learning the right patterns.
2. Hyperparameter Tuning: Assists in finding the best hyperparameters (like learning rate, regularization parameters, etc.) that improve the model's performance.
3. Model Selection: Aids in choosing the best model among a set of models by comparing their performances on validation data

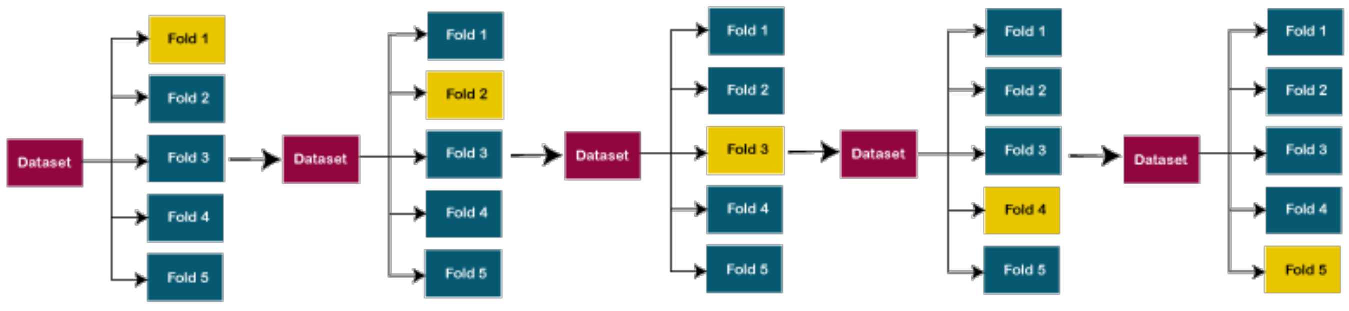
Examples in Practice

- Neural Networks: Validation data helps decide the number of layers, neurons, learning rate, etc.
- Decision Trees: Validation data is used to determine the depth of the tree or the minimum number of samples required at a leaf node.
- Support Vector Machines (SVMs): Validation data helps choose the kernel type and regularization parameter.

Cross-Validation

Cross-validation is a technique for assessing how a model will perform on an independent dataset by partitioning the available data into multiple subsets (or "folds") and training and testing the model on these folds.
Cross validation involves dividing the available data into multiple folds or subsets, using one of these folds as a validation set, and training the model on the remaining folds. This process is repeated multiple times, each time using a different fold as the validation set. Finally, the results from each validation step are averaged to produce a more robust estimate of the model's performance.

The most common of these is **the k-fold cross-validation** procedure. In K-Fold Cross Validation, we split the dataset into k number of subsets (known as folds) then we perform training on the all the subsets but leave one(k-1) subset for the evaluation of the trained model. In this method, we iterate k times with a different subset reserved for testing purpose each time.



ESTIMATORS -BIAS AND VARIANCE

Bias represents the error due to overly simplistic assumptions in the learning algorithm. High bias can cause the model to underfit the data, leading to poor performance on both training and unseen data.

When a model is too simplistic, it fails to capture the underlying structure of the data. This results in underfitting, where the model performs poorly on both the training data and unseen test data. The high bias prevents the model from representing the true relationship**.**

Example: For example, a linear regression model assumes that there is a linear relationship between the inputs and the output. If the actual relationship is more complex, this assumption leads to systematic errors.

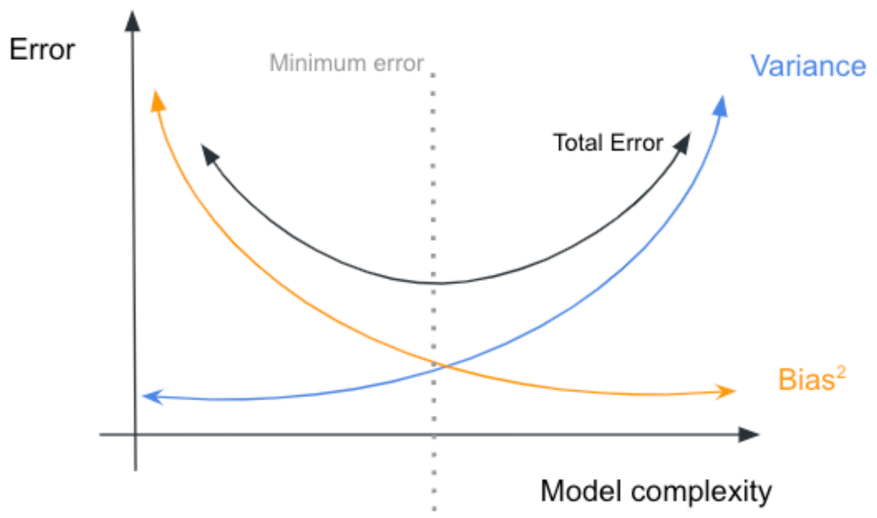
Variance, on the other hand, reflects the model's sensitivity to small fluctuations in the training data. High variance can lead to overfitting, where the model captures noise in the training data and performs poorly on new, unseen data.

The goal is to find the right level of complexity in a model to minimize both bias and variance, achieving good generalization to new data. Balancing these factors is essential for building models that perform well on a variety of datasets.

The relationship between bias and variance is tightly linked to the machine learning concepts of capacity, underfitting and overfitting. In the case where generalization error is measured by the MSE, increasing capacity tends to increase variance and decrease bias.

If the model is too simple then it may be on high bias and low variance condition and thus is error-prone (underfit). If algorithms fit too complex then it may be on high variance and low bias. In the latter condition, the new entries will not perform well(overfit). There is something between both of these conditions, known as a Trade-off or Bias Variance Trade-off. A model can't be more complex and less complex at the same time.
We try to optimize the value of the total error for the model by using the Bias-Variance Trade-off

Total Error = Bias^2 + Variance + Minimum Error



INTRODUCTION TO DEEP LEARNING

DEEP FEED FORWARD NETWORK

A deep feedforward network, also known as a deep feedforward neural network, is a type of artificial neural network where information moves in one direction: forward, from the input nodes through the hidden layers to the output nodes. These networks are often referred to as "deep" because they consist of multiple hidden layers, enabling them to learn hierarchical representations of data.

Deep feedforward networks are characterized by their layered architecture, consisting of an input layer, multiple hidden layers, and an output layer. Each layer contains nodes (or neurons) interconnected to nodes in the subsequent layer. Information flows through the network during training and inference, with computations performed layer by layer through a series of weighted transformations and activation functions.

Steps involved in neural network training

Data Collection and Preprocessing
Model Selection and Architecture Design
Initialization
Forward Propagation
Loss Computation
Backpropagation
Optimization
Validation and Tuning
Testing

****************************** END *************************************

QUESTION 1

Consider the case of the XOR function in which the two points $\{(0, 0),(1, 1)\}$ belong to one class, and the other two points $\{(1, 0),(0, 1)\}$ belong to the other class. Design a multilayer perceptron for this binary classification problem.

Solution

To design a multilayer perceptron (MLP) for the XOR function binary classification problem, you'll need an architecture that can learn to separate the two classes. The XOR problem is not linearly separable, so a single-layer perceptron won't be able to solve it. Instead, a simple two-layer MLP can be used. Here's an example architecture**:**

Input Layer: 2 input neurons (for the two coordinates of each point)
Hidden Layer: 2 neurons with a sigmoid activation function
Output Layer: 1 neuron with a sigmoid activation function
Here's the step-by-step design:
Input Layer: The input layer will have 2 neurons, each corresponding to one coordinate of the input points.
Hidden Layer: The hidden layer will have 2 neurons. These neurons will apply an activation function, like the sigmoid function, to their weighted inputs. The purpose of the hidden layer is to introduce non-linearity into the model, allowing it to learn complex patterns.

Output Layer: The output layer will have 1 neuron, which will produce the final classification output. Since this is a binary classification problem, a sigmoid activation function is used here as well to squash the output between 0 and 1.
The weights and biases in each layer will be learned during training using a suitable optimization algorithm, such as gradient descent. You can initialize the weights randomly and then iteratively update them based on the error between the predicted outputs and the actual targets.
Keep in mind that while this architecture can solve the XOR problem, it might not be the most efficient or optimal solution. More complex problems might require deeper architectures or more advanced techniques like convolutional neural networks (CNNs) or recurrent neural networks (RNNs).

Input to Hidden Layer Weights:

Neuron 1 in Hidden Layer:

Weight from Input Neuron 1: 0.1
Weight from Input Neuron 2: 0.2

Neuron 2 in Hidden Layer:

Weight from Input Neuron 1: -0.3
Weight from Input Neuron 2: 0.4

Hidden to Output Layer Weights:

Output Neuron:
Weight from Hidden Neuron 1: 0.5
Weight from Hidden Neuron 2: -0.6

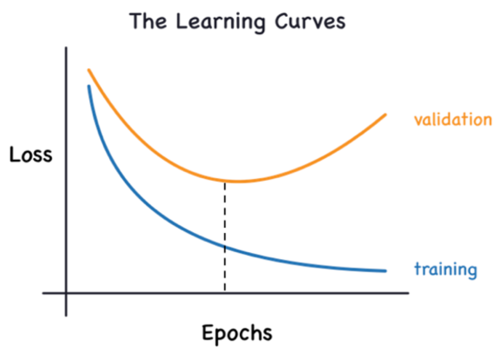
These are just example values and you can initialize the weights randomly within a suitable range. During training, the network will adjust these weights to learn the correct representations for the XOR problem. Keep in mind that the success of the training process also depends on the learning rate, the choice of activation functions, and the optimization algorithm used.

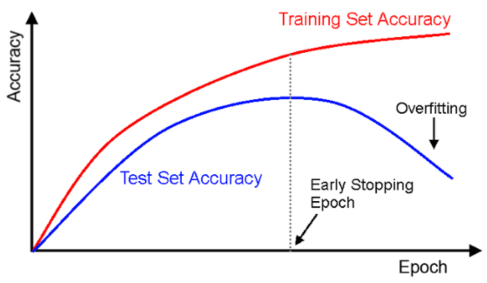
Underfitting Phase:
Initially, both the training and validation error are relatively high. The model is not capturing the underlying patterns in the data well. The model is too simple and cannot represent the complexity of the data.

Learning/Improvement Phase:
As the training continues, the training error decreases. The model is learning to capture the patterns in the training data. The validation error also decreases as the model generalizes better to unseen data. The gap between the training and validation error starts to decrease.

Optimal Phase:
The training and validation errors reach a point where they are both relatively low and close to each other. The model generalizes well to both the training and validation data.

Overfitting Phase:
The training error continues to decrease, possibly approaching almost zero. However, the validation error starts increasing or stabilizing, indicating that the model is starting to memorize noise in the training data. The gap between the training and validation error starts widening significantly. Here's a rough sketch of what the learning curves might look like:





As you can see, the training error continues to decrease while the validation error starts to increase or stabilize. This indicates that the model is starting to fit the training data too closely and is not generalizing well to new data, which is a characteristic of overfitting.

QUESTION 2

Specify the advantage of ReLU over Sigmoid.

1. Mitigates the Vanishing Gradient Problem

- ReLU: The gradient of ReLU is 1 for positive inputs, and 0 for negative inputs. For positive values, the gradient does not vanish, which helps in maintaining the magnitude of gradients during backpropagation. This avoids the issue where gradients become very small as they are propagated backward through many layers, which can slow down or halt learning in deep networks.
- Sigmoid: The gradient of the sigmoid function can be very small for large positive or negative values of input. This can lead to the vanishing gradient problem, where gradients become extremely small during backpropagation, causing slow or stalled training for early layers.

2. Faster Training

- ReLU: The computation involved in ReLU is simple and efficient, as it only requires a threshold operation. This makes it faster to compute compared to sigmoid, which involves exponential calculations.

$$\text{ReLU}(x) = \max(0,x).$$

- Sigmoid: Computing the sigmoid function involves an exponential function. This is more computationally expensive than the ReLU operation

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

3. Sparsity

- ReLU: It introduces sparsity in the activations because it outputs zero for all negative inputs. This sparsity can be beneficial for neural network performance, as it can lead to more efficient computations and a more compact representation.
- Sigmoid: The sigmoid function does not introduce sparsity, as it always outputs values between 0 and 1. All neurons are activated to some degree, which can lead to less efficient computation in practice.

4. Improved Convergence

- ReLU: Because of its properties, ReLU can often lead to faster convergence during training. The ability to maintain gradients effectively helps in speeding up the learning process and achieving better performance in less time.
- Sigmoid: Due to the vanishing gradient problem and computational overhead, networks using sigmoid activation functions may converge more slowly and require more epochs to train effectively.

These advantages make ReLU a preferred choice in many deep learning architectures, especially in convolutional and fully connected neural networks. However, it's also important to consider that ReLU has its own drawbacks (e.g., dead neurons), and variants like Leaky ReLU, Parametric ReLU, and ELU can address some of these issues.

QUESTION 3

Derive weight updating rule in Gradient Descent when the loss function is Mean Squared Error (MSE) and Cross-Entropy (CE)

To derive the weight updating rules for gradient descent when using Mean Squared Error (MSE) and Cross-Entropy loss functions, we need to perform the following steps:

1. **Compute the Gradient of the Error Function with Respect to Weights**:
   - For both MSE and Cross-Entropy, we need to compute how the loss function changes with respect to the weights.
2. **Update the Weights**:
   - We use the gradient to update the weights in the direction that minimizes the loss.

Mean Squared Error Loss Function:

The MSE loss function for a single training example is defined as:
$L = \frac{1}{2}(y - \hat{y})^2$

where:

- y is the true label.
- $\hat{y}$ is the predicted output from the network.

For multiple training examples, the average MSE is:
$L = \frac{1}{N}\sum_{i=1}^{N}\frac{1}{2}(y_i - \hat{y}_i)^2$

where N is the number of examples.

Gradient Computation:

1. **Forward Pass**:
    - Suppose the predicted output $\hat{y}$ is given by:
      $\hat{y} = f(Wx + b)$
      where W is the weight matrix, x is the input vector, b is the bias, and f is the activation function.
2. **Gradient of MSE with Respect to Output**:
    - The derivative of the MSE loss function with respect to $\hat{y}$ is:
      $\frac{\partial L}{\partial \hat{y}} = \hat{y} - y$
3. **Gradient of MSE with Respect to Weights**:
    - To find the gradient with respect to the weights W, we use the chain rule:
      $\frac{\partial L}{\partial W} = \frac{\partial L}{\partial \hat{y}} \cdot \frac{\partial \hat{y}}{\partial W}$
    - If $\hat{y} = f(z)$ where $z = Wx + b$, then:
      $\frac{\partial \hat{y}}{\partial W} = f'(z) \cdot x$
    - So:
      $\frac{\partial L}{\partial W} = (\hat{y} - y) \cdot f'(z) \cdot x$

Weight Update Rule:

The weight update rule using gradient descent is:
$W \leftarrow W - \eta \cdot \frac{\partial L}{\partial W}$
where η is the learning rate.

So:
$W \leftarrow W - \eta \cdot (\hat{y} - y) \cdot f'(z) \cdot x$

Cross Entropy Loss Function:

For binary classification, the Cross-Entropy loss function is:
$$L = -[y \log(\hat{y}) + (1 - y) \log(1 - \hat{y})]$$

where:

- y is the true label (0 or 1).
- $\hat{y}$ is the predicted probability of the positive class.

For multiple training examples, the average Cross-Entropy is:
$$L = -\frac{1}{N} \sum_{i=1}^{N} [y_i \log(\hat{y}_i) + (1 - y_i) \log(1 - \hat{y}_i)]$$

1. **Forward Pass**:

   - The predicted output $\hat{y}$ is given by:
     $\hat{y} = \sigma(Wx + b)$
     where σ is the sigmoid function.

2. **Gradient of Cross-Entropy with Respect to Output**:

   - The derivative of the Cross-Entropy loss function with respect to $\hat{y}$ is:
     $\frac{\partial L}{\partial \hat{y}} = \frac{\hat{y} - y}{\hat{y}(1 - \hat{y})}$
   - However, when using the sigmoid activation function, this simplifies to:
     $\frac{\partial L}{\partial \hat{y}} = \hat{y} - y$

3. **Gradient of Cross-Entropy with Respect to Weights**:

- Using the chain rule:
  $$\frac{\partial L}{\partial W} = \frac{\partial L}{\partial \hat{y}} \cdot \frac{\partial \hat{y}}{\partial W}$$
- With $\hat{y} = \sigma(z)$ and $\frac{\partial \hat{y}}{\partial W} = \sigma'(z) \cdot x$, where $\sigma'(z) = \hat{y}(1 - \hat{y})$:
  $$\frac{\partial L}{\partial W} = (\hat{y} - y) \cdot \sigma'(z) \cdot x$$
- Since $\sigma'(z) = \hat{y}(1 - \hat{y})$, we get:
  $$\frac{\partial L}{\partial W} = (\hat{y} - y) \cdot x$$

The term $\hat{y}(1 - \hat{y})$ is not explicitly used in the final gradient computation due to the specific properties of the sigmoid activation function and cross-entropy loss function. The combined gradient simplifies naturally, making the implementation more efficient and less error-prone by avoiding unnecessary calculations.

Weight Update Rule:

The weight update rule using gradient descent is:
$$W \leftarrow W - \eta \cdot \frac{\partial L}{\partial W}$$
where η is the learning rate.

So:
$$W \leftarrow W - \eta \cdot (\hat{y} - y) \cdot x$$

Summary

- **Mean Squared Error (MSE)**:
  $$W \leftarrow W - \eta \cdot (\hat{y} - y) \cdot f'(z) \cdot x$$

- **Cross-Entropy Loss**:
  $$W \leftarrow W - \eta \cdot (\hat{y} - y) \cdot x$$

In practice, Cross-Entropy loss is often preferred for classification tasks because it generally leads to faster convergence and better performance compared to MSE. This is particularly important for problems where the output is a probability distribution.