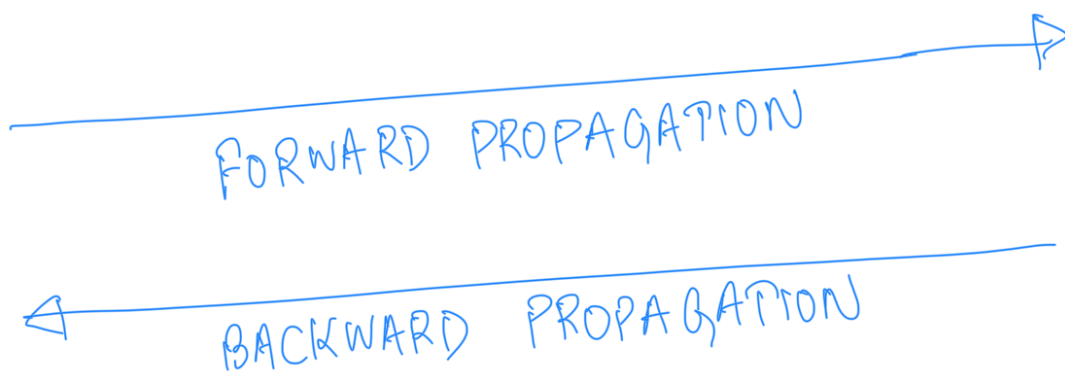
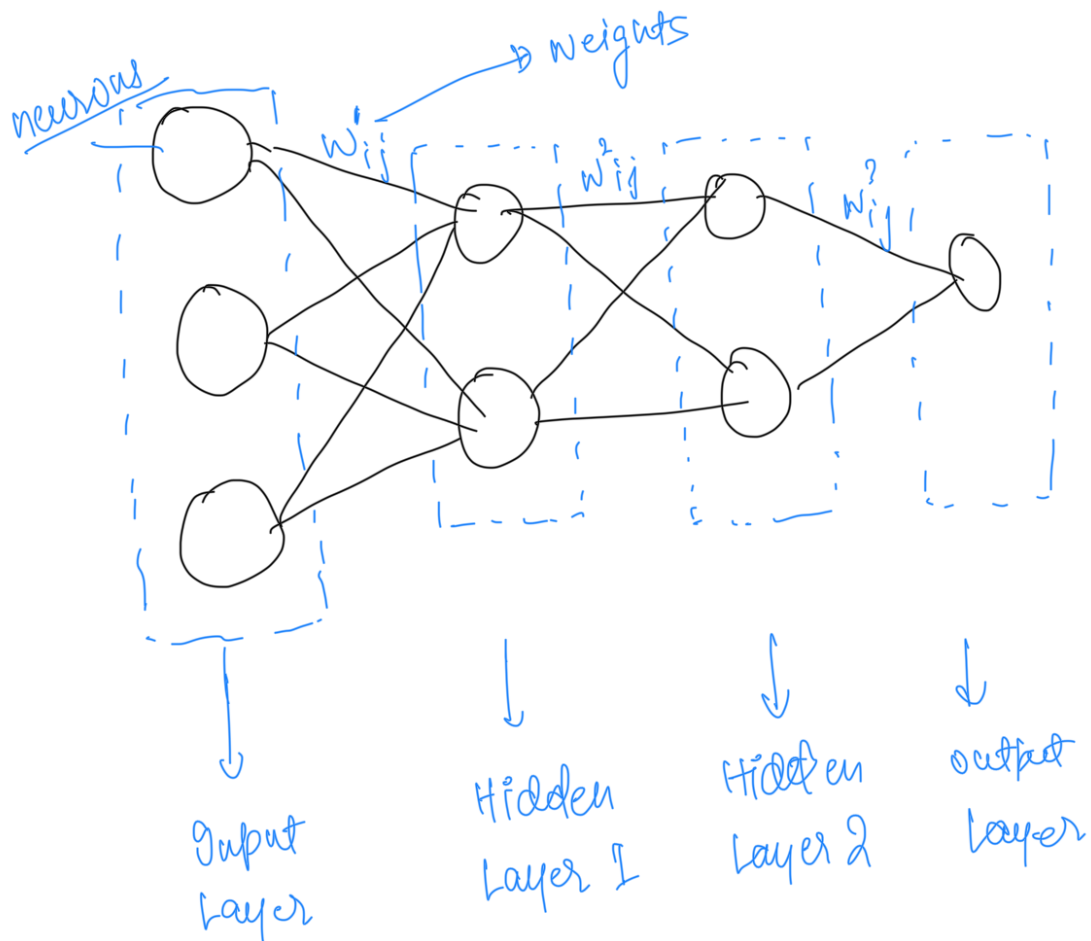


Artificial neural networks

It's a computational model inspired by the structure and function of the human brain, used for tasks like pattern recognition, classification, regression, and more.

Some important terminology in Artificial Neural Networks (ANNs) includes:

1. **Neuron:** A basic computational unit that receives input signals, applies weights to them, and produces an output signal.
2. **Activation Function:** A function that determines the output of a neuron, typically introducing non-linearity into the network eg sigmoid, tanh, ReLU, etc., used to introduce non-linearity into the network.
3. **Layer:** A collection of neurons that receive the same inputs and produce outputs that are fed to the neurons in the next layer.
4. **Input Layer:** The layer of neurons that receives input data from the external environment.
5. **Hidden Layer:** Layers of neurons between the input and output layers that perform computations.
6. **Output Layer:** The final layer of neurons that produces the output of the network.
7. **Weights and Biases:** Parameters that adjust the strength of connections between neurons.
8. **Forward Propagation:** The process of passing input data through the network to generate an output.
9. **Backpropagation:** An algorithm used to train the network by adjusting the weights and biases based on the error between predicted and actual outputs.
10. **Loss Function:** A function that measures the difference between predicted and actual outputs, used to guide the training process.
11. **Gradient Descent:** An optimization algorithm used to minimize the loss function by adjusting the weights and biases in the direction of the steepest descent.
12. **Learning Rate:** A hyperparameter that determines the size of the step taken during gradient descent.
13. **Epoch:** One complete pass through the entire training dataset during the training process.
14. **Batch Size:** The number of samples used in one iteration of training.



1 epoch \rightarrow 1 complete pass (LFP + LBP)

In artificial neural networks (ANNs), there are primarily three types of connections:

1. Feedforward Connections: These connections transmit signals from one layer to the next without forming cycles. Information flows in one direction, typically from the input layer through one or more hidden layers to the output layer. Feedforward connections are prevalent in most neural network architectures, such as multilayer perceptrons (MLPs) and

convolutional neural networks (CNNs).

2. **Recurrent Connections:** Unlike feedforward connections, recurrent connections allow feedback loops within the network. Neurons in a recurrent neural network (RNN) can receive input not only from the previous layer but also from their own previous states or from other neurons in the same layer. Recurrent connections enable ANNs to process sequential data and handle tasks like time series prediction, natural language processing, and speech recognition.

3. **Lateral Connections:** Lateral connections refer to connections between neurons within the same layer. These connections can facilitate information exchange between neighboring neurons and are commonly found in networks designed for tasks like image segmentation, where contextual information from neighboring pixels is useful.

Types of activation functions:

1. Sigmoid Function (Logistic Function):

- Outputs values between 0 and 1.
- Smooth gradient, but prone to vanishing gradient problem.
- Used in the output layer for binary classification problems.

$$\left[f(x) = \frac{1}{1 + e^{-x}} \right]$$

2. Hyperbolic Tangent (Tanh) Function:

- Outputs values between -1 and 1, centered around 0.
- Similar to sigmoid but with a range from -1 to 1.
- Also prone to vanishing gradient problem.

$$\left[f(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}} \right]$$

3. Rectified Linear Unit (ReLU):

- Outputs the input if it is positive, otherwise outputs zero.
- Faster convergence than sigmoid and tanh.
- Can suffer from the "dying ReLU" problem where neurons get stuck in the negative region and cease to update.

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

4. Leaky ReLU:

- Similar to ReLU but allows a small, non-zero gradient when the input is negative, preventing neurons from dying.

$$f(x) = \begin{cases} x, & \text{if } x \geq 0 \\ \alpha x, & \text{else} \end{cases}$$

5. Softmax function:

The softmax function is a popular activation function used in neural networks, particularly in the output layer for multi-class classification tasks. The output of the softmax function is a vector where each element represents the probability of the corresponding class.

$$\left[f(x) = \frac{e^{x_1}}{\sum_{j=1}^n e^{x_j}} \right]$$

6. Exponential Linear Unit (ELU):

- Smooth for all values of x , avoiding the "dying ReLU" problem.
- Can result in slower training due to the exponential function.

$$f(x) = \begin{cases} x & , \text{ if } x > 0 \\ \alpha(e^x - 1) & , \text{ otherwise} \end{cases}$$

These are some important activation functions which are frequently used

Loss functions:

Regression

1. Mean Squared Error (MSE):

- Calculates the average squared difference between the predicted values () and the actual targets ().
- Sensitive to outliers due to squaring.

$$MSE = \frac{1}{n} \sum_{i=1}^n (y_i - y_p)^2$$

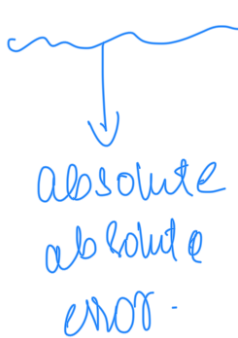
where, y_i = actual value

y_p = predicted value

2. Mean Absolute Error (MAE):

Mean Absolute Error (MAE) is a metric used to measure the average absolute difference between the actual values and the values predicted by a model. It provides a measure of how close the predictions are to the actual values, regardless of the direction of the difference. It can handle outliers.

$$MAE = \frac{1}{n} \sum_{i=1}^n |y_i - y_p|$$



absolute
absolute
error.

where,

$y_i \rightarrow$ actual value

$y_p \rightarrow$ predicted value

3. Huber Loss:

Huber loss, also known as the Huber function or Huber penalty, is a loss function used in robust regression. It combines the advantages of Mean Absolute Error (MAE) and Mean Squared Error (MSE) by being less sensitive to outliers than MSE while still maintaining

differentiability at zero like MAE.

$$\text{Huber loss} = \begin{cases} \frac{1}{2} (y_i - y_p)^2 & , \text{ if } |y_i - y_p| < \delta \\ \frac{1}{2} |y_i - y_p| & , \text{ else} \end{cases}$$

" δ " is a hyperparameter for Huber loss which decides whether MSE or MAE would prevail.

Classification:

Classification losses are used in machine learning for tasks where the goal is to predict a categorical outcome, such as class labels. Here are some common classification loss functions:

1. Binary Cross-Entropy Loss (Log Loss):

- Used for binary classification problems.

$$\text{Loss} = y_i \log y_p - (1 - y_i) \log (1 - y_p)$$

where, y_i = actual class (0/1)
 y_p = predicted probability of any class

2. Categorical Cross-Entropy Loss:

- Used for multi-class classification problems.

$$\text{Loss} = \sum_{k=1}^n y_i \log y_p$$

n = total no. of classes

3. Hinge Loss (SVM Loss):

- Commonly used for binary classification with Support Vector Machines (SVMs).

$$\text{Loss} = \max(0, 1 - y_i y_p)$$

y_i = true label

y_p = predicted label

Types of Optimisers:

In artificial neural networks (ANNs), optimizers are algorithms used to minimize the loss

function during training by updating the weights and biases of the network. Here are some common types of optimizers used in ANNs:

1. Gradient Descent:

- The basic optimization algorithm where the model parameters are updated in the opposite direction of the gradient of the loss function with respect to those parameters.
- Variants include:
 - Stochastic Gradient Descent (SGD): Updates parameters using a single training example at a time.
 - Mini-Batch Gradient Descent: Updates parameters using a small batch of training examples at a time.
 - Batch Gradient Descent: Updates parameters using the entire training dataset at once.

2. Adaptive Gradient Algorithms:

- These algorithms adapt the learning rate during training based on past gradients.
- Variants include:
 - Adagrad: Adapts the learning rate based on the historical gradient for each parameter.
 - RMSprop: A variation of Adagrad that normalizes the historical gradient by dividing it by a moving average of the squared gradient.
 - Adadelta: An extension of RMSprop that maintains a moving average of both the squared gradient and the squared parameter updates.
 - Adam: Combines the advantages of Adagrad and RMSprop by using both the moving average of gradients and squared gradients, along with bias correction.

3. Momentum-Based Algorithms:

- These algorithms use a momentum term to accelerate optimization in directions with consistent gradients and dampen oscillations in other directions.
- Variants include:
 - Momentum: Introduces a momentum term to the gradient descent update to dampen oscillations.

These optimisers have different characteristics and performance depending on the dataset, model architecture, and training objectives.

Weights changing formula:

The weight changing formula, also known as the weight update rule, is a mathematical formula used in training artificial neural networks to adjust the weights and biases of the network's connections during the optimization process. The most common weight changing formula is based on gradient descent, which aims to minimize the loss function by iteratively updating the network's parameters in the direction that reduces the loss.

For a simple feedforward neural network, the weight changing formula using gradient descent typically involves the following steps:

1. Calculate the gradient of the loss function with respect to each weight and bias in the network using backpropagation.
2. Update each weight and bias using the gradients and a learning rate parameter.

The general formula for updating the weights and biases using gradient descent is:

$$W_{\text{new}} = W_{\text{old}} - \eta \frac{\partial L}{\partial W_{\text{old}}}$$

Where,

W_{new} = updated weights

W_{old} = old weight

η = learning rate

$\frac{\partial L}{\partial W_{\text{old}}}$ = loss gradient

Problems related with weight updation :

Several issues can arise during the weight updating process in neural networks, affecting the training process and the model's performance. Here are some common problems related to weight updating:

1. Vanishing and Exploding Gradients:

- In neural networks, gradients can become extremely small (vanishing gradients) or large (exploding gradients) during backpropagation, especially in networks with many layers.
- Vanishing gradients can cause slow or stalled learning, while exploding gradients can lead to unstable training.

2. Local Minima and Plateaus:

- Gradient-based optimization methods can get stuck in local minima or plateaus, where the gradient is close to zero and the optimization process stagnates.
- This problem can prevent the model from finding the global minimum of the loss function.

3. Learning Rate Selection:

- Choosing an appropriate learning rate is crucial for effective weight updating. A learning rate that is too high can lead to unstable training, while a learning rate that is too low can result in slow convergence or getting stuck in local minima.

4. Gradient Descent Variants:

- Different variants of gradient descent (e.g., momentum, adaptive learning rate methods) have their own hyperparameters and potential issues, such as sensitivity to initialization and difficulty in tuning.

5. Saddle Points:

- Saddle points are points in the parameter space where the gradient is zero but the point is not an optimum.
- Optimization algorithms may struggle to escape saddle points, leading to slow convergence.

Addressing these issues often involves a combination of techniques such as careful initialization of weights, using adaptive learning rates, employing regularization methods, and experimenting with different optimization algorithms. Additionally, advancements in optimization algorithms and network architectures continue to address and mitigate these challenges in neural network training.

Ways to prevent overfitting:

Preventing overfitting in artificial neural networks (ANNs) is crucial for building models that generalize well to unseen data. Here are several techniques commonly used to prevent overfitting in ANNs:

1. Cross-Validation:

- Split the dataset into multiple subsets for training, validation, and testing.
- Use the validation set to monitor the model's performance during training and adjust hyperparameters accordingly.

2. Regularization:

- L1 and L2 Regularization: Add penalties to the loss function based on the magnitudes of

the weights (L1 for absolute values, L2 for squared values) to discourage large weights.

- Dropout: Randomly deactivate neurons during training to prevent co-adaptation of neurons and improve generalization.

3. Regularizing Activations:

- Apply regularization directly to the activations of the neurons, such as through activation functions like ReLU with parameterized rectification or the use of activation functions like Swish.

By combining these techniques and adjusting their parameters appropriately, it's possible to build ANNs that generalize well to unseen data and avoid overfitting.