Artificial Neural Network

A Multilayer Perceptron (MLP) Feed Forward Neural Network is a fundamental type of artificial neural network. It is characterized by the unidirectional flow of information, meaning data moves from the input layer, through one or more hidden layers, to the output layer without forming cycles or feedback loops. Each layer consists of interconnected nodes (neurons), and connections between neurons in adjacent layers have associated weights. These networks are called "feed-forward" because the computations proceed strictly from input to output. MLPs are renowned for their ability to learn complex, non-linear relationships within data.

Here's a detailed breakdown of key concepts related to MLP Feed Forward Neural Networks:

1. Forward and Backward Passes:

- Forward Pass (Forward Propagation): This is the process where the network takes input data and processes it through its layers to produce an output (a prediction or classification).
    - Working:
        1. The input data is fed into the input layer.
        2. For each neuron in a subsequent layer, a weighted sum of the outputs from the neurons in the previous layer is calculated. A bias term is typically added to this sum.
        3. This weighted sum is then passed through an activation function (discussed below) to produce the neuron's output.
        4. This process repeats layer by layer until the output layer is reached, which produces the final network output.
    - Essentially, information flows "forwards" through the network, with each layer transforming the data based on its weights, biases, and activation functions. Intermediate values (activations of hidden layers) are calculated and stored as they are needed for the backward pass.
- Backward Pass (Backpropagation of Error): After the forward pass produces an output, this output is compared to the actual target value (in supervised learning) using a loss function (like Cross-Entropy). The backward pass is the process of

propagating the calculated error back through the network to update the weights and biases in a way that minimizes this error.

- Working:
    1. The error (or loss) is calculated at the output layer.
    2. The algorithm then calculates the gradient of the loss function with respect to the weights and biases of the output layer. This tells us how much a small change in each weight/bias would affect the error.
    3. These gradients are then propagated backward to the previous hidden layer. The chain rule of calculus is used to calculate the gradients of the loss function with respect to the weights, biases, and activations of this layer.
    4. This process continues backward, layer by layer, until the input layer is reached.
    5. The calculated gradients are then used by an optimization algorithm (like stochastic gradient descent) to update the weights and biases throughout the network. The goal is to adjust them in the direction that reduces the overall error.
- The forward and backward passes are iteratively performed during the training process. The forward pass computes the output and the error, and the backward pass uses this error to adjust the network's parameters.

2. Nonlinearity: Activation Functions:

- Activation functions are a critical component of neural networks, introducing non-linearity into the model. Without them, an MLP, regardless of how many layers it has, would behave like a simple linear model, severely limiting its ability to learn complex patterns.
- Working: Each neuron in a hidden or output layer applies an activation function to its weighted sum of inputs. This transformed output then becomes the input for the neurons in the next layer.
- Common Activation Functions:
    - Sigmoid (Logistic):
        - Formula: $f(x)=(1+e-x)1$

        - Output Range: (0,1)
        - Use: Historically common, especially in the output layer for binary classification problems (outputting probabilities).
        - Drawbacks: Can suffer from the vanishing gradient problem (discussed later), and its outputs are not zero-centered.
    - Tanh (Hyperbolic Tangent):
        - Formula: $f(x)=(ex+e−x)(ex−e−x)$
        - Output Range: $(−1,1)$
        - Use: Similar S-shape to sigmoid but zero-centered, which can sometimes lead to faster convergence.
        - Drawbacks: Also prone to vanishing gradients.
    - ReLU (Rectified Linear Unit):
        - Formula: $f(x)=max(0,x)$
        - Output Range: $[0,∞)$
        - Use: Very popular in hidden layers due to its computational efficiency and ability to mitigate the vanishing gradient problem for positive inputs.
        - Drawbacks: Can suffer from the "dying ReLU" problem, where neurons can become inactive and only output zero if their input is consistently negative.
    - Leaky ReLU:
        - Formula: $f(x)=max(ax,x)$ (where 'a' is a small constant, e.g., 0.01)
        - Output Range: $(−∞,∞)$
        - Use: An attempt to fix the dying ReLU problem by allowing a small, non-zero gradient when the unit is not active.
    - Parametric ReLU (PReLU): Similar to Leaky ReLU, but 'a' is a learnable parameter.
    - ELU (Exponential Linear Unit):
        - Formula: $f(x)=x$ if $x>0$, and $a(ex−1)$ if $x≤0$

        - Use: Aims to combine the benefits of ReLU with outputs closer to zero-mean, potentially alleviating vanishing gradients and speeding up learning.
    - Softmax:
        - Formula: $f(x_i)=\sum_j e^{x_j}e^{x_i}$ (applied to a vector of inputs x)
        - Output Range: Each output is between (0,1), and the sum of all outputs is 1.
        - Use: Typically used in the output layer for multi-class classification problems, as it converts a vector of raw scores (logits) into a probability distribution over the classes.
    - Linear (Identity):
        - Formula: $f(x)=x$
        - Output Range: $(-\infty,\infty)$
        - Use: Sometimes used in the output layer for regression problems where the target value is continuous and not bounded.

3. Cross-Entropy:

- Cross-entropy is a widely used loss function (also known as log loss) in classification tasks. It measures the dissimilarity between the predicted probability distribution (output by the network, often after a softmax activation) and the true probability distribution (the actual labels).
- Working:
    - The goal during training is to minimize the cross-entropy loss. A lower cross-entropy value indicates that the model's predicted probabilities are closer to the actual labels.
    - Binary Cross-Entropy: Used for binary classification problems (two classes).
        - Formula: $L=-(y\log(p)+(1-y)\log(1-p))$
            - y: actual label (0 or 1)
            - p: predicted probability for class 1
    - Categorical Cross-Entropy: Used for multi-class classification problems (more than two classes).

        - Formula: $L = -\sum_{i=1}^{C} y_i \log(p_i)$
            - C: number of classes
            - yi: 1 if the sample belongs to class i, 0 otherwise (one-hot encoded true label)
            - pi: predicted probability for class i
- The cross-entropy loss penalizes predictions that are confident but wrong more heavily. For a perfect model, the cross-entropy loss would be 0.

4. Computational Graph and Backpropagation:

- Computational Graph: A computational graph is a way of representing any mathematical expression or computation as a directed graph. In the context of neural networks:
    - Nodes: Represent operations (e.g., addition, multiplication, activation functions) or variables (e.g., inputs, weights, biases).
    - Edges: Represent the flow of data (tensors) between operations.
    - Working: The entire forward pass of a neural network, from input to the final loss calculation, can be visualized as a large computational graph. This graph clearly shows the dependencies between different operations.
- Backpropagation (on a Computational Graph): Backpropagation is an efficient algorithm for computing gradients in a computational graph by applying the chain rule of calculus.
    - Working:
        1. Forward Pass: Compute the value of each node in the graph from inputs to the final output (the loss).
        2. Backward Pass: Starting from the final output node (the loss), calculate the gradient of this node with respect to itself (which is 1).
        3. Then, move backward through the graph. For each node, calculate the gradient of the final output (loss) with respect to that node's inputs by using the chain rule and the gradients already computed for the nodes that follow it.
        4. This process continues until the gradients of the loss with respect to all the network's parameters (weights and biases) are computed.

- Deep learning frameworks like TensorFlow and PyTorch automatically build these computational graphs and perform backpropagation, making it easier to define and train complex models. Understanding the computational graph helps in visualizing how gradients flow and how updates are made.

5. Vanishing and Exploding Gradients:

These are common problems encountered during the training of deep neural networks (networks with many layers) that can hinder the learning process. They relate to the magnitude of the gradients being backpropagated.

- Vanishing Gradients:
  - Description: This occurs when the gradients (the signals used to update the network's weights) become extremely small as they are propagated backward from the output layer to the earlier layers.
  - Causes:
    - Activation Functions: Sigmoid and tanh functions have derivatives that are small for large positive or negative inputs (their outputs saturate). When many such layers are stacked, the gradients can diminish exponentially.
    - Deep Architectures: In very deep networks, the repeated multiplication of small numbers during backpropagation can cause gradients to shrink to near zero.
    - Weight Initialization: Poor weight initialization can also contribute.
  - Impact: The weights in the earlier layers learn very slowly or not at all, as their updates become negligible. This prevents the network from learning complex features from the input data effectively.
  - Solutions:
    - Use activation functions like ReLU and its variants (Leaky ReLU, ELU), which have less of a saturating effect.
    - Proper weight initialization techniques (e.g., He initialization for ReLU, Xavier/Glorot initialization for tanh/sigmoid).
    - Batch Normalization.
    - Residual connections (as in ResNets).

        - Gradient clipping (though more common for exploding gradients, can sometimes help).

- Exploding Gradients:
    - Description: This is the opposite problem, where the gradients become excessively large as they are propagated backward.
    - Causes:
        - Large Weight Values: If the weights in the network are initialized to be too large.
        - Deep Architectures: Repeated multiplication of large numbers during backpropagation can cause gradients to grow exponentially.
    - Impact: Leads to very large updates to the weights, causing the optimization process to become unstable. The loss function might oscillate wildly or even diverge (become NaN - Not a Number), preventing the model from converging to a good solution.
    - Solutions:
        - Gradient Clipping: A common technique where if the norm (magnitude) of the gradients exceeds a predefined threshold, they are scaled down to be within that threshold.
        - Proper weight initialization.
        - Batch Normalization.
        - Using activation functions that are less prone to causing explosions (though this is less of an activation function issue compared to vanishing gradients).
        - Smaller learning rates.

6. Overfitting, Underfitting, Dropout Regularization:

These terms describe how well a model learns from the training data and generalizes to new, unseen data.

- Underfitting:

    - Description: Occurs when a model is too simple to capture the underlying patterns in the training data. It performs poorly on both the training data and new,[1] unseen data (test data).
    - Characteristics: High bias (the model makes strong assumptions that don't fit the data) and often low variance (the model's predictions don't change much with different training sets, but they are consistently wrong).
    - Causes:
        - Model is not complex enough (e.g., too few layers or neurons).
        - Insufficient training (not enough epochs).
        - Features used are not informative enough.
    - Solutions:
        - Increase model complexity (more layers/neurons).
        - Train for longer.
        - Feature engineering (add more relevant features).
        - Reduce regularization.
- Overfitting:
    - Description: Occurs when a model learns the training data too well, including its noise and random fluctuations. As a result, it performs[2] exceptionally well on the training data but poorly on new, unseen data (test data). The model has "memorized" the training data instead of "learning" the general patterns.
    - Characteristics: Low bias (the model fits the training data very closely) and high variance (the model's predictions can change drastically with different training sets). Training error is low, but testing error is significantly higher.
    - Causes:
        - Model is too complex for the amount of training data (e.g., too many layers/neurons).
        - Insufficient training data.
        - Training for too many epochs.
        - High dimensionality of data ("curse of dimensionality").

- Solutions:
    - Get more training data.
    - Simplify the model (fewer layers/neurons).
    - Regularization techniques (like L1, L2, and Dropout).
    - Early stopping (stop training when performance on a validation set starts to degrade).
    - Data augmentation.
    - Feature selection (pruning less important features).
    - Ensembling methods.

- Dropout Regularization:
    - Description: Dropout is a powerful and widely used regularization technique specifically for neural networks to combat overfitting.
    - Working:

1. During each training iteration (forward/backward pass), a certain fraction of neurons (e.g., 20% to 50%) in a layer are randomly "dropped out" or deactivated. This means their outputs are set to zero for that iteration, and they do not participate in the forward or backward pass.

2. The choice of which neurons to drop is random for each training sample and each iteration.

3. This prevents neurons from co-adapting too much, meaning they become less reliant on the presence of specific other neurons. Each neuron has to learn features that are useful on their own or in conjunction with different random subsets of other neurons.

4. Effectively, dropout trains a large ensemble of thinner networks with shared weights. At test time, dropout is typically turned off, and the full network is used. To compensate for the fact that more neurons are active during testing than during training, the outputs of the active neurons are often scaled down by the dropout probability (or, equivalently, the weights are scaled down after training).

    - Benefits:
        - Reduces overfitting by making the network more robust and less sensitive to the specific weights of individual neurons.

        - Encourages the network to learn more distributed and redundant representations.
        - Can be seen as a form of model averaging.
    - Implementation Tips:
        - A common dropout rate is between 0.2 and 0.5.
        - It can be applied to input layers and hidden layers.
        - Often used with larger networks that might otherwise overfit.

Numerical Example for MLP Feed Forward Neural Network

Setup:

- Input layer: 2 neurons (features)
- One hidden layer: 2 neurons
- Output layer: 1 neuron (binary classification)
- Activation: Sigmoid for hidden and output layers (to keep it simple)
- Loss: Binary cross-entropy

1. Forward Pass Example

Given:

- Input vector:

$$x = \begin{bmatrix} 0.6 \\ 0.9 \end{bmatrix}$$

- Weights and biases for hidden layer (layer 1):

$$W^{(1)} = \begin{bmatrix} 0.1 & 0.4 \\ 0.3 & 0.7 \end{bmatrix}, \quad b^{(1)} = \begin{bmatrix} 0.1 \\ 0.2 \end{bmatrix}$$

- Weights and bias for output layer (layer 2):

$$W^{(2)} = \begin{bmatrix} 0.5 & 0.2 \end{bmatrix}, \quad b^{(2)} = \begin{bmatrix} 0.3 \end{bmatrix}$$

Step 1: Compute pre-activation of hidden layer

$$z^{(1)} = W^{(1)}x + b^{(1)} = \begin{bmatrix} 0.1 & 0.4 \\ 0.3 & 0.7 \end{bmatrix} \begin{bmatrix} 0.6 \\ 0.9 \end{bmatrix} + \begin{bmatrix} 0.1 \\ 0.2 \end{bmatrix}$$

Calculate:

$$z_1^{(1)} = 0.1 \times 0.6 + 0.4 \times 0.9 + 0.1 = 0.06 + 0.36 + 0.1 = 0.52$$

$$z_2^{(1)} = 0.3 \times 0.6 + 0.7 \times 0.9 + 0.2 = 0.18 + 0.63 + 0.2 = 1.01$$

So,

$$z^{(1)} = \begin{bmatrix} 0.52 \\ 1.01 \end{bmatrix}$$

Step 2: Apply sigmoid activation on hidden layer

$$a^{(1)} = \sigma(z^{(1)}) = \begin{bmatrix} \sigma(0.52) \\ \sigma(1.01) \end{bmatrix}$$

Recall sigmoid:

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

Calculate each:

$$\sigma(0.52) = \frac{1}{1 + e^{-0.52}} \approx \frac{1}{1 + 0.594} = 0.627$$

$$\sigma(1.01) = \frac{1}{1 + e^{-1.01}} \approx \frac{1}{1 + 0.364} = 0.733$$

$$a^{(1)} \approx \begin{bmatrix} 0.627 \\ 0.733 \end{bmatrix}$$

Step 3: Compute pre-activation of output layer

$$z^{(2)} = W^{(2)}a^{(1)} + b^{(2)} = \begin{bmatrix}0.5 & 0.2\end{bmatrix}\begin{bmatrix}0.627 \\ 0.733\end{bmatrix} + 0.3$$

Calculate:

$$z^{(2)} = 0.5 \times 0.627 + 0.2 \times 0.733 + 0.3 = 0.3135 + 0.1466 + 0.3 = 0.7601$$

Step 4: Apply sigmoid activation on output layer

$$a^{(2)} = \sigma(z^{(2)}) = \frac{1}{1 + e^{-0.7601}} \approx \frac{1}{1 + 0.467} = 0.681$$

Output (prediction):

$$\hat{y} = 0.681$$

2. Loss Calculation (Binary Cross-Entropy)

Assume true label $y = 1$.

$$L = -\left(y \log \hat{y} + (1 - y) \log(1 - \hat{y})\right) = -(1 \times \log 0.681 + 0 \times \log(1 - 0.681)) = -\log 0.681$$

Calculate:

$$L = -(-0.384) = 0.384$$

3. Backward Pass (Backpropagation)

Step 1: Compute output layer error term

$$\delta^{(2)} = \frac{\partial L}{\partial z^{(2)}} = (a^{(2)} - y) \cdot \sigma'(z^{(2)})$$

But derivative of sigmoid output wrt input simplifies this gradient to:

$$\delta^{(2)} = a^{(2)} - y = 0.681 - 1 = -0.319$$

Step 2: Gradients for output weights and bias

$$\frac{\partial L}{\partial W^{(2)}} = \delta^{(2)}(a^{(1)})^T = -0.319 \times \begin{bmatrix}0.627\\0.733\end{bmatrix}^T = \begin{bmatrix}-0.200\\-0.234\end{bmatrix}$$

$$\frac{\partial L}{\partial b^{(2)}} = \delta^{(2)} = -0.319$$

Step 3: Compute error term for hidden layer

Recall:

$$\delta^{(1)} = (W^{(2)})^T\delta^{(2)} \odot \sigma'(z^{(1)})$$

- $W^{(2)T} = \begin{bmatrix}0.5\\0.2\end{bmatrix}$
- $\delta^{(2)} = -0.319$

Calculate:

$$W^{(2)T}\delta^{(2)} = \begin{bmatrix}0.5\\0.2\end{bmatrix} \times (-0.319) = \begin{bmatrix}-0.1595\\-0.0638\end{bmatrix}$$

Derivative of sigmoid:

$$\sigma'(x) = \sigma(x)(1-\sigma(x))$$

For $z_1^{(1)} = 0.52$, $a_1^{(1)} = 0.627$:

$$\sigma'(0.52) = 0.627 \times (1-0.627) = 0.627 \times 0.373 = 0.234$$

For $z_2^{(1)} = 1.01$, $a_2^{(1)} = 0.733$:

$$\sigma'(1.01) = 0.733 \times (1-0.733) = 0.733 \times 0.267 = 0.196$$

Element-wise multiply:

$$\delta^{(1)} = \begin{bmatrix}-0.1595\\-0.0638\end{bmatrix} \odot \begin{bmatrix}0.234\\0.196\end{bmatrix} = \begin{bmatrix}-0.0373\\-0.0125\end{bmatrix}$$

Step 4: Gradients for hidden layer weights and biases

$$\frac{\partial L}{\partial W^{(1)}} = \delta^{(1)}(x)^T = \begin{bmatrix} -0.0373 \\ -0.0125 \end{bmatrix} \times \begin{bmatrix} 0.6 & 0.9 \end{bmatrix} = \begin{bmatrix} -0.0224 & -0.0335 \\ -0.0075 & -0.0113 \end{bmatrix}$$

$$\frac{\partial L}{\partial b^{(1)}} = \delta^{(1)} = \begin{bmatrix} -0.0373 \\ -0.0125 \end{bmatrix}$$

4. Weight Updates (Gradient Descent)

Assume learning rate $\eta = 0.1$.

Update weights:

$$W^{(2)}_{\text{new}} = W^{(2)} - \eta \frac{\partial L}{\partial W^{(2)}} = \begin{bmatrix} 0.5 & 0.2 \end{bmatrix} - 0.1 \times \begin{bmatrix} -0.200 & -0.234 \end{bmatrix} = \begin{bmatrix} 0.52 & 0.2234 \end{bmatrix}$$

Update biases:

$$b^{(2)}_{\text{new}} = b^{(2)} - \eta \delta^{(2)} = 0.3 - 0.1 \times (-0.319) = 0.3319$$

Similarly for hidden layer:

$$W^{(1)}_{\text{new}} = W^{(1)} - \eta \frac{\partial L}{\partial W^{(1)}} = \begin{bmatrix} 0.1 & 0.4 \\ 0.3 & 0.7 \end{bmatrix} - 0.1 \times \begin{bmatrix} -0.0224 & -0.0335 \\ -0.0075 & -0.0113 \end{bmatrix} = \begin{bmatrix} 0.1022 & 0.4033 \\ 0.3007 & 0.7011 \end{bmatrix}$$

$$b^{(1)}_{\text{new}} = b^{(1)} - \eta \delta^{(1)} = \begin{bmatrix} 0.1 \\ 0.2 \end{bmatrix} - 0.1 \times \begin{bmatrix} -0.0373 \\ -0.0125 \end{bmatrix} = \begin{bmatrix} 0.1037 \\ 0.2013 \end{bmatrix}$$

Summary

- **Forward pass**: Compute activations layer by layer using weights, biases, and activation functions.
- **Loss**: Compare output with true label using cross-entropy.
- **Backward pass**: Calculate gradients of loss w.r.t weights and biases layer by layer, starting from output.
- **Update weights**: Apply gradients to improve the model.

A **complete numerical example** of a **3-layer neural network**

(input layer → hidden layer → output layer) with:

- Input layer: 3 neurons (3 input features)
- Hidden layer: 3 neurons
- Output layer: 1 neuron

And you want to see the full **feedforward calculation** from input to output.

Setup

Input vector:

$$x = \begin{bmatrix} 0.5 \\ 0.1 \\ 0.4 \end{bmatrix}$$

Weights and biases for hidden layer (layer 1):

Weights $W^{(1)}$ is a 3x3 matrix (3 neurons × 3 inputs):

$$W^{(1)} = \begin{bmatrix} 0.2 & 0.4 & 0.1 \\ 0.5 & 0.3 & 0.6 \\ 0.1 & 0.7 & 0.2 \end{bmatrix}$$

Bias $b^{(1)}$ is a vector of length 3 (one bias per neuron):

$$b^{(1)} = \begin{bmatrix} 0.1 \\ 0.2 \\ 0.3 \end{bmatrix}$$

Weights and bias for output layer (layer 2):

Weights $W^{(2)}$ is a 1x3 vector (output neuron connects to 3 hidden neurons):

$$W^{(2)} = \begin{bmatrix} 0.3 & 0.7 & 0.5 \end{bmatrix}$$

Bias $b^{(2)}$ is a scalar (one bias for output neuron):

$$b^{(2)} = 0.4$$

Step 1: Calculate hidden layer pre-activation $z^{(1)}$

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

Calculate $W^{(1)} \times x$:

$$\begin{bmatrix} 0.2 & 0.4 & 0.1 \\ 0.5 & 0.3 & 0.6 \\ 0.1 & 0.7 & 0.2 \end{bmatrix} \times \begin{bmatrix} 0.5 \\ 0.1 \\ 0.4 \end{bmatrix} = \begin{bmatrix} (0.2 \times 0.5) + (0.4 \times 0.1) + (0.1 \times 0.4) \\ (0.5 \times 0.5) + (0.3 \times 0.1) + (0.6 \times 0.4) \\ (0.1 \times 0.5) + (0.7 \times 0.1) + (0.2 \times 0.4) \end{bmatrix}$$

Calculate each element:

- First neuron:
  $0.1 + 0.04 + 0.04 = 0.18$
- Second neuron:
  $0.25 + 0.03 + 0.24 = 0.52$
- Third neuron:
  $0.05 + 0.07 + 0.08 = 0.20$

Add biases $b^{(1)}$:

$$z^{(1)} = \begin{bmatrix} 0.18 + 0.1 \\ 0.52 + 0.2 \\ 0.20 + 0.3 \end{bmatrix} = \begin{bmatrix} 0.28 \\ 0.72 \\ 0.50 \end{bmatrix}$$

Step 2: Apply sigmoid activation function to get hidden layer output $a^{(1)}$

Recall sigmoid:

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

Calculate for each element of $z^{(1)}$:

- For 0.28:

$$\sigma(0.28) = \frac{1}{1 + e^{-0.28}} \approx \frac{1}{1 + 0.756} = 0.569$$

- For 0.72:

$$\sigma(0.72) = \frac{1}{1 + e^{-}} \approx \frac{1}{1 + 0.487} = 0.673$$

So,

$$a^{(1)} = \begin{bmatrix} 0.569 \\ 0.673 \\ 0.622 \end{bmatrix}$$

Step 3: Calculate output layer pre-activation $z^{(2)}$

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

Calculate $W^{(2)} \times a^{(1)}$:

$$(0.3 \times 0.569) + (0.7 \times 0.673) + (0.5 \times 0.622) = 0.1707 + 0.4711 + 0.311 = 0.9528$$

Add bias:

$$z^{(2)} = 0.9528 + 0.4 = 1.3528$$

Step 4: Apply sigmoid activation function to get final output $a^{(2)}$

$$a^{(2)} = \sigma(z^{(2)}) = \frac{1}{1 + e^{-1.3528}} \approx \frac{1}{1 + 0.258} = 0.795$$

Final answer:

- The output of the network for the input x=[0.5,0.1,0.4]x = [0.5, 0.1, 0.4]x=[0.5,0.1,0.4] is approximately **0.795** after feedforward through 3 neurons in hidden and 1 output neuron.

How to get a binary answer?

- Usually, we choose a **threshold** — most commonly **0.5**.
- If output ≥ 0.5 → predict **1**
- If output < 0.5 → predict **0**