Q1: What is Perceptron?
Ans: A perceptron is a simple linear binary classifier that models a single artificial neuron. It computes a weighted sum of inputs, adds a bias, and passes it through a step function. Though powerful for linearly separable tasks, it's limited and cannot solve non-linear problems like XOR — which is why deeper networks were developed.

Q2: Activation Function used in Perceptron?
Ans: In a classic perceptron, the **step function** is used as the activation function, which outputs either 0 or 1 depending on the weighted sum of the inputs. However, since it's non-differentiable, it cannot be used with gradient-based learning methods. In modern neural networks, differentiable activation functions like **ReLU**, **sigmoid**, or **tanh** are used instead.

Q3: XOR problem in Perceptron?
Ans: The XOR problem refers to the inability of a single-layer perceptron to model the XOR logic gate, because the XOR function is not linearly separable. A perceptron creates a linear decision boundary, but XOR requires a non-linear one. This problem was historically important because it showed the limitations of early neural networks and motivated the development of multi-layer networks, which can solve XOR using hidden layers and non-linear activations.

Q4: Sigmoid Activation Function
Ans: The sigmoid activation function maps input values to a range between 0 and 1 using the formula $\sigma(z) = \frac{1}{1+e^{-z}}.$. It is commonly used in the output layer of binary classification models to represent probabilities. While elegant and smooth, it suffers from vanishing gradients and is rarely used in hidden layers of deep networks today — ReLU is usually preferred there.

Q5: What is chain rule of derivative?
Ans: The chain rule in deep learning is a fundamental concept from calculus that we use during **backpropagation** to compute gradients. Since a neural network is essentially a composition of functions — where each layer applies a transformation — we apply the chain rule to calculate how the **loss changes with respect to the weights** in earlier layers.

Specifically, the output of one layer becomes the input to the next. So, to compute the gradient of the loss with respect to any weight, we use the chain rule to **multiply the derivatives layer by layer, from the output layer backward to the input layer**. This lets us efficiently update all the weights using gradient descent.

In simple terms: the chain rule allows us to track how a small change in a weight affects the final output and loss, even when that weight is deep inside the network.