

Question 1: What is Deep Learning? Briefly describe how it evolved and how it differs from traditional machine learning.

Answer:

Deep learning is a subfield of machine learning , that focuses on the algorithms that are inspired by the functioning of the human brain (for example, how the human brain do different tasks like understanding, finding patterns etc), known as artificial neural network. It uses multi-layered neural networks that are used to understand the hierarchical representation of data automatically, enabling machines to recognize complex patterns in unstructured datasets like images, videos, texts etc

Evolution:

1940-1980s : Early neural network models like perceptrons were introduced, but limited computational power and data restricted their performance.

1990s: Traditional Machine Learning techniques like SVM, decision trees became dominant due to their better theoretical understanding and efficiency.

2000-2010s: Advancement in computational power (GPUs), availability of large datasets and improvements in training algorithms (e.g. Backpropagation, ReLU activation etc) led to resurgence of neural networks.

Present: Deep learning achieved state-of-art results in domain like computer vision, natural language processing and speech recognition etc.

Difference between Deep learning and Traditional Machine learning:

Feature Extraction:

Machine Learning Models: Feature extraction is done manually , require domain expertise.

Deep Learning Models: Automatic feature learning.

Data Requirement :

Machine Learning Models: Works well with low to medium datasets.

Deep Learning Models: Requires large amount of data.

Model Complexity:

Machine Learning Models: Simpler models

Deep Learning Models: Complex models , multi-layered networks.

Computational Cost:

Machine Learning Models: Lower computational cost.

Deep learning Models: High computational cost (GPUs, TPUs)

Performance on unstructured data:

Machine Learning Models: Limited performance on unstructured dataset.

Deep Learning Models: Excellent performance on unstructured dataset.

Question 2: Explain the basic architecture and functioning of a Perceptron. What are its limitations?

Answer:

Perceptron is the most simplest form of an artificial neural network that also serves as the basic building block of more complex neural networks. It is a single layered, binary linear classifier used for supervised machine learning.

Basic Architecture of a Perceptron:

1. **Input Layer (x_1, x_2, \dots, x_n)**: These are the feature values provided to the model.
2. **Weights (w_1, w_2, \dots, w_n)**: Each input has an associated weight that describes its importance.
3. **Biases(b)**: A constant added to shift the decision boundary.
4. **Summation Unit**: Computes the weighted sum of inputs.

$$z = \sum_{i=1}^n w_i x_i + b$$

5. **Activation Function**: Typically a step function that produces a binary output.

$$y = \begin{cases} 1, & \text{if } z \geq 0 \\ 0, & \text{otherwise} \end{cases}$$

Functioning of a perceptron:

1. Inputs are multiplied by their respective weights.
2. Weighted inputs are the summed and added to the bias.
3. The activation function determines the final output.
4. During training, weights are updated using the perceptron learning rule.

$$w_i \leftarrow w_i + \eta(y_{\text{actual}} - y_{\text{predicted}})x_i$$

Limitations of perceptron:

1. **Only Linearly Separable Data**:
Cannot solve non-linear problems such as XOR.
2. **Single-Layer Structure**:
Lacks hidden layers, limiting its representational power.

3. **Binary Output Only:**
Produces only 0 or 1 outputs.
4. **No Continuous Output:**
Not suitable for regression tasks.
5. **Sensitive to Noise:**
Performance degrades with noisy or overlapping data.

Question 3: Describe the purpose of activation function in neural networks. Compare Sigmoid, ReLU, and Tanh functions.

Answer:

Purpose of Activation Functions in Neural Networks

Activation functions introduce non-linearity into neural networks, enabling them to learn and model complex patterns beyond simple linear relationships. Without activation functions, a neural network regardless of the number of layers—would behave like a linear model. Activation functions also help control the output range, influence gradient flow during backpropagation, and determine how signals are passed from one layer to the next.

Comparison of Sigmoid, ReLU, and Tanh Activation Functions:

Aspect	Sigmoid	Tanh	ReLU
Mathematical Form	$\sigma(x) = \frac{1}{1+e^{-x}}$	$\tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$	$f(x) = \max(0, x)$
Output Range	(0, 1)	(-1, 1)	[0, ∞)
Zero-Centred	No	Yes	No
Main Advantage	<i>Smooth, probabilistic output</i>	<i>Stronger gradients than sigmoid</i>	<i>Fast computation, avoids vanishing gradient</i>

Main Limitation	<i>Vanishing gradient, slow training</i>	<i>Still suffers from vanishing gradient</i>	<i>Dying ReLU (neurons output zero)</i>
Common Usage	<i>Output layer for binary classification</i>	<i>Hidden layers (older models)</i>	<i>Hidden layers in deep networks</i>

Key Observations

- Sigmoid is useful when outputs need to be interpreted as probabilities but is less effective in deep networks due to vanishing gradients.
- Tanh improves upon sigmoid by being zero-centered, leading to better convergence, though gradient saturation remains an issue.
- ReLU is computationally efficient and widely used in modern deep learning, especially in hidden layers, but may suffer from inactive (dead) neurons.

Question 4: What is the difference between Loss function and Cost function in neural networks? Provide examples.

Answer:

Loss Function vs Cost Function in neural networks:

In neural networks, loss function and cost function are used to measure how well a model's prediction matches to the actual target values. Although, the terms are sometimes used interchangeably, but there are some conceptual differences between these two.

Loss Function

A **loss function** measures the error for a **single training example**.

Purpose:

- Quantifies how wrong the model's prediction is for one data point.
- Guides weight updates during backpropagation.

Examples:

1. Mean Squared Error (per sample):

$$L(y, \hat{y}) = (y - \hat{y})^2$$

2. Binary Cross-Entropy (per sample):

$$L(y, \hat{y}) = -[y \log(\hat{y}) + (1 - y) \log(1 - \hat{y})]$$

Cost Function

A cost function represents the **average (or total) loss over the entire training dataset.**

Purpose:

- Evaluates the overall performance of the model.
- Optimization algorithms aim to minimize the cost function.

Examples:

1. Mean Squared Error Cost:

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

2. Cross-Entropy Cost:

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

In conclusion, the **loss function** evaluates error at the individual sample level, while the **cost function** aggregates these losses to assess and optimize the overall model performance during training.

Question 5: What is the role of optimizers in neural networks? Compare Gradient Descent, Adam, and RMSprop.

Answer:

Role of optimizers in neural network:

Optimizers are used to update the weights and biases of a neural network during training so as to minimize the loss (cost) function. They determine how the gradients computed via backpropagation are applied to the model's parameters. An effective

optimizers ensure faster convergence, stable training and better overall performance.

Comparison of Gradient Descent, RMSprop, and Adam :

Aspect	Gradient Descent	RMSprop	Adam
Basic Idea	Updates parameters using the full dataset gradient	Adapts learning rate using moving average of squared gradients	Combines momentum and RMSprop
Learning Rate	Fixed (manually tuned)	Adaptive	Adaptive
Update Rule	$w = w - \eta \nabla J(w)$	Uses exponential moving average of squared gradients	Uses first and second moment estimates
Convergence Speed	Slow	Faster than Gradient Descent	Very fast
Memory Requirement	Low	Moderate	Moderate
Handles Sparse Gradients	Poorly	Well	Very well
Common Usage	Simple models, theoretical understanding	RNNs, non-stationary problems	Most deep learning applications

Brief explanation of each optimizer:

Gradient Descent (Batch Gradient Descent):

- Uses the entire dataset to compute gradients.
- Stable but computationally expensive and slow for large datasets.
- Sensitive to learning rate choice.

RMSprop (Root Mean Square Propagation):

Adjusts the learning rate for each parameter.

- Reduces oscillations and improves convergence.
- Well-suited for problems with varying gradient magnitudes.

Adam (Adaptive Moment Estimation):

- Maintains both momentum (first moment) and RMSprop-like (second moment) estimates.
- Efficient, robust, and widely used in practice.
- Performs well with noisy and sparse gradients.

Question 6: Write a Python program to implement a single-layer perceptron from scratch using NumPy to solve the logical AND gate.

Answer:

```
import numpy as np

# AND gate dataset
# Inputs
X = np.array([
    [0, 0],
    [0, 1],
    [1, 0],
    [1, 1]
])

# Outputs
y = np.array([0, 0, 0, 1])

# Initialize weights and bias
weights = np.zeros(X.shape[1])
bias = 0
learning_rate = 0.1
epochs = 10
```

```

# Step activation function
def step_function(x):
    return 1 if x >= 0 else 0

# Training the perceptron
for epoch in range(epochs):
    for i in range(len(X)):
        linear_output = np.dot(X[i], weights) + bias
        y_pred = step_function(linear_output)
        error = y[i] - y_pred

        # Update weights and bias
        weights += learning_rate * error * X[i]
        bias += learning_rate * error

# Testing the perceptron
print("Final weights:", weights)
print("Final bias:", bias)
print("\nAND Gate Predictions:")
for i in range(len(X)):
    output = step_function(np.dot(X[i], weights) + bias)
    print(f"Input: {X[i]} Output: {output}")

```

Output:

```

Final weights: [0.2 0.1]
Final bias: -0.2000000000000004

AND Gate Predictions:
Input: [0 0] Output: 0
Input: [0 1] Output: 0
Input: [1 0] Output: 0
Input: [1 1] Output: 1

```

Question 7: Implement and visualize Sigmoid, ReLU, and Tanh activation functions using Matplotlib.

Answer:

```
import numpy as np
import matplotlib.pyplot as plt

# Input range
x = np.linspace(-10, 10, 100)

# Activation functions
def sigmoid(x):
    return 1 / (1 + np.exp(-x))

def relu(x):
    return np.maximum(0, x)

def tanh(x):
    return np.tanh(x)

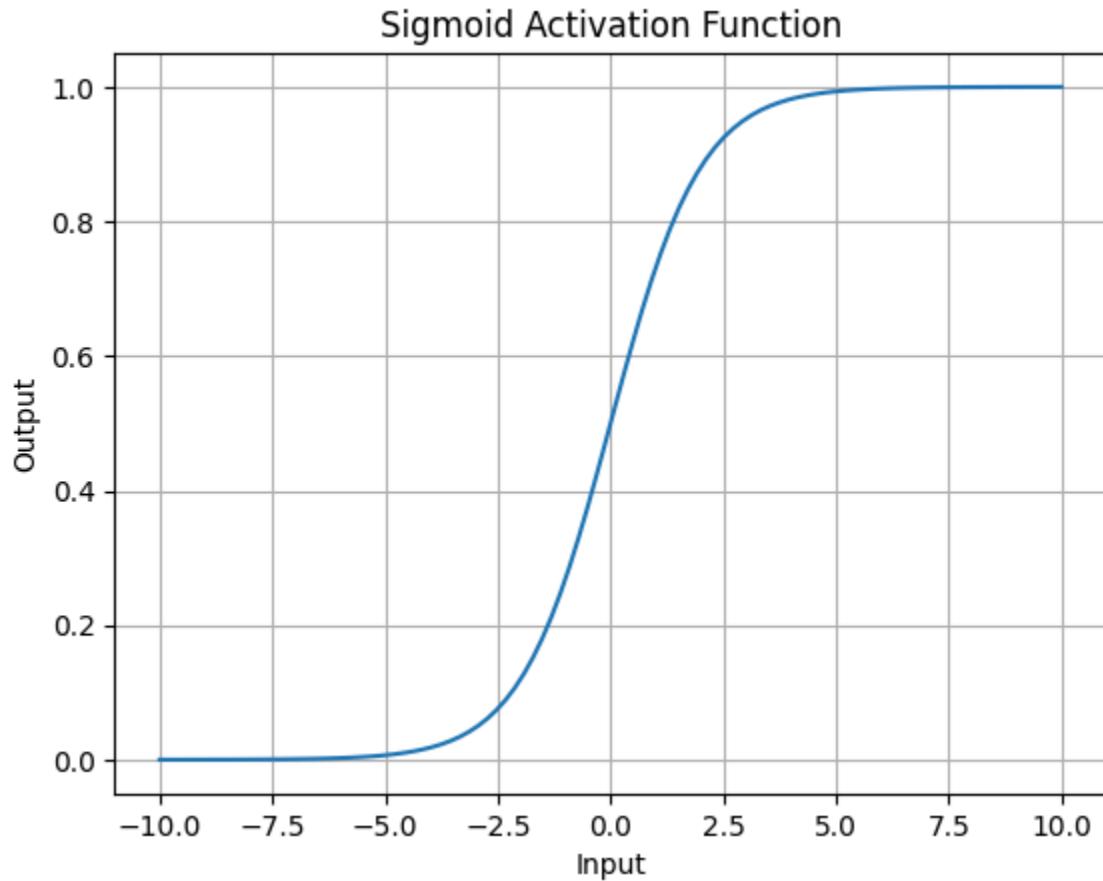
# Plot Sigmoid
plt.figure()
plt.plot(x, sigmoid(x))
plt.title("Sigmoid Activation Function")
plt.xlabel("Input")
plt.ylabel("Output")
plt.grid(True)
plt.show()

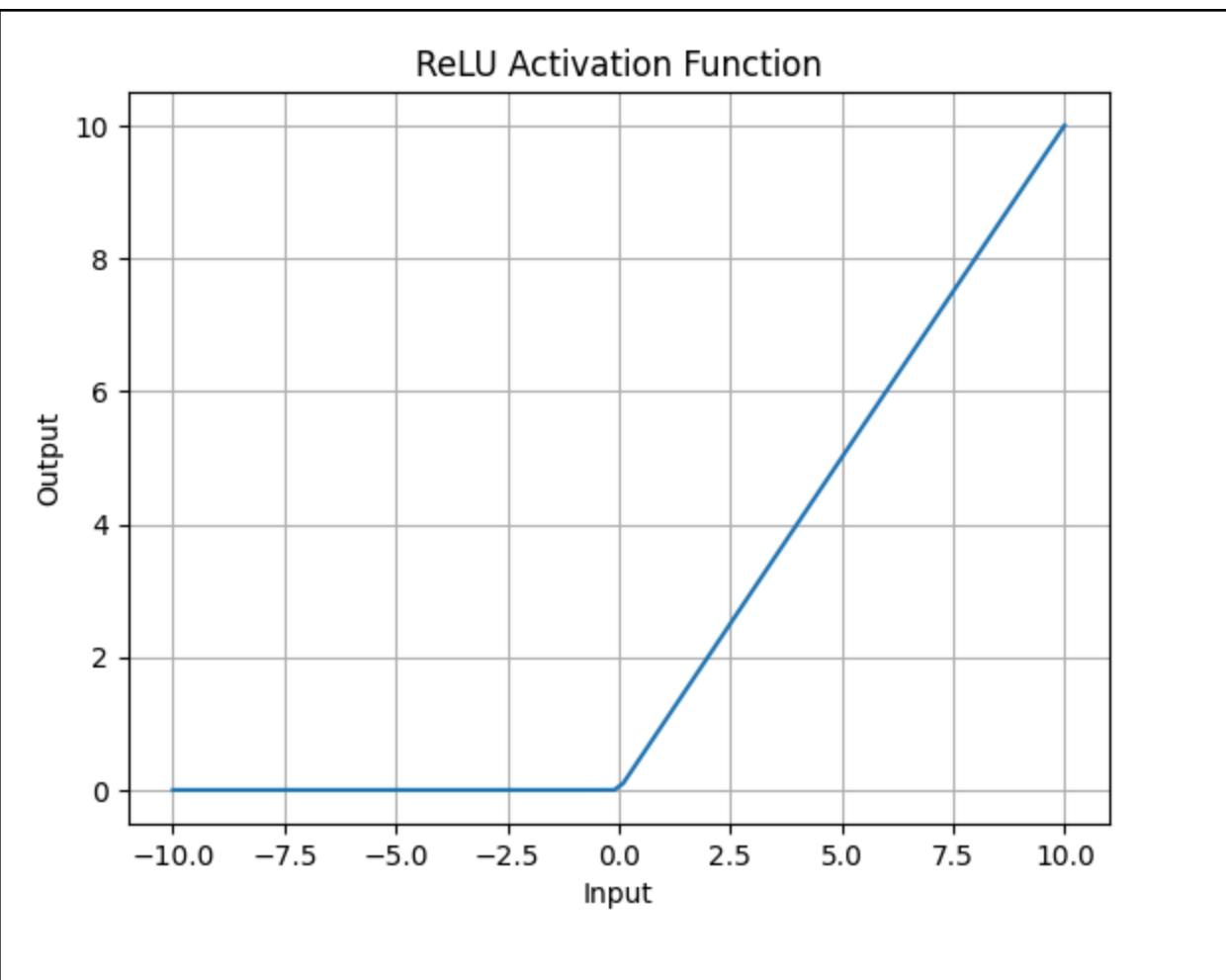
# Plot ReLU
plt.figure()
plt.plot(x, relu(x))
plt.title("ReLU Activation Function")
plt.xlabel("Input")
plt.ylabel("Output")
plt.grid(True)
plt.show()

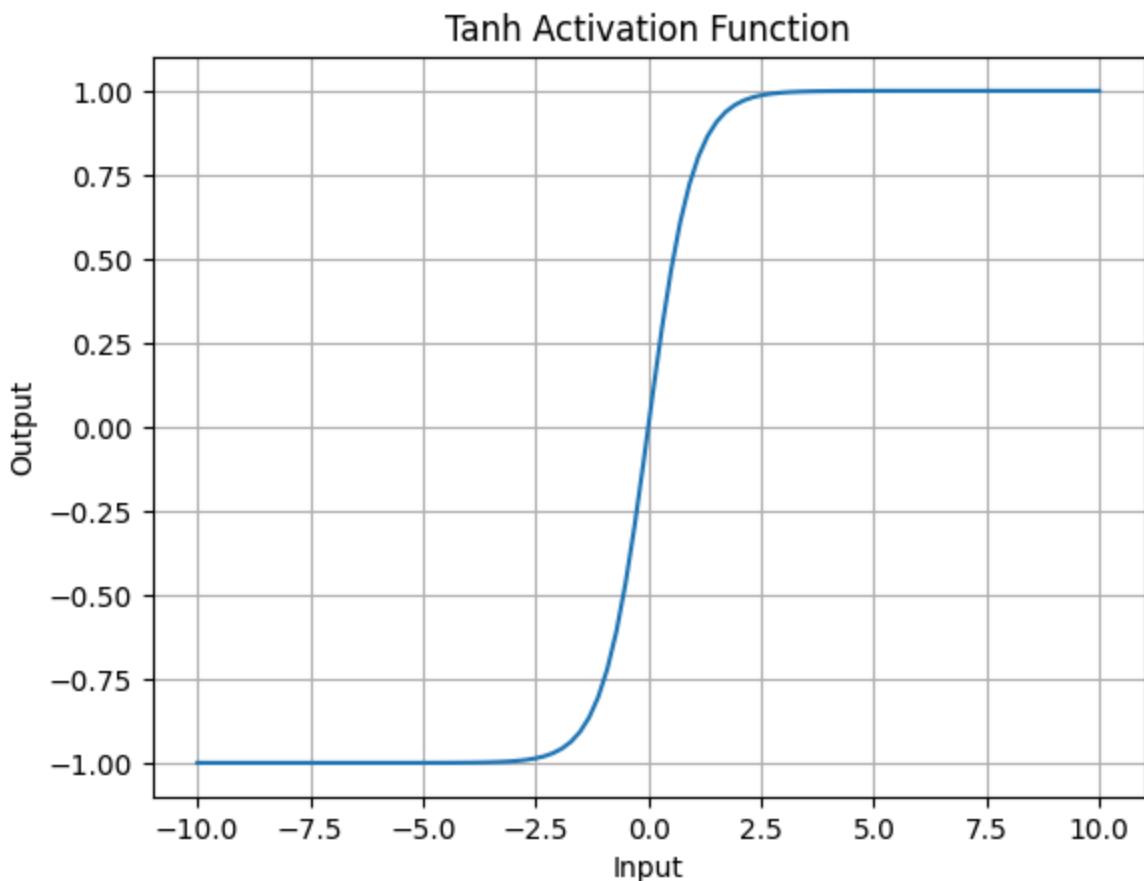
# Plot Tanh
plt.figure()
plt.plot(x, tanh(x))
plt.title("Tanh Activation Function")
plt.xlabel("Input")
```

```
plt.ylabel("Output")
plt.grid(True)
plt.show()
```

Output:







Question 8: Use Keras to build and train a simple multilayer neural network on the MNIST digits dataset. Print the training accuracy.

Answer:

```

import tensorflow as tf
from tensorflow.keras.datasets import mnist
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, Flatten
from tensorflow.keras.utils import to_categorical

# Load MNIST dataset
(x_train, y_train), (x_test, y_test) = mnist.load_data()

# Normalize the data
x_train = x_train / 255.0
x_test = x_test / 255.0

```

```
# One-hot encode the labels
y_train = to_categorical(y_train, 10)
y_test = to_categorical(y_test, 10)

# Build the multilayer neural network
model = Sequential([
    Flatten(input_shape=(28, 28)),
    Dense(128, activation='relu'),
    Dense(64, activation='relu'),
    Dense(10, activation='softmax')
])

# Compile the model
model.compile(
    optimizer='adam',
    loss='categorical_crossentropy',
    metrics=['accuracy']
)

# Train the model
history = model.fit(
    x_train,
    y_train,
    epochs=5,
    batch_size=32,
    verbose=1
)

# Print training accuracy
training_accuracy = history.history['accuracy'][-1]
print("Training Accuracy:", training_accuracy)
```

Output:

```
Downloading data from https://storage.googleapis.com/tensorflow/tf-keras-datasets/mnist.npz
11490434/11490434          16s 1us/step
c:\Users\HP\AppData\Local\Programs\Python\Python313\Lib\site-packages\keras\src\layers\reshape.py:17: DeprecationWarning: This class is deprecated. It is recommended to use tf.keras.layers.Reshape instead.
  super().__init__(**kwargs)
Epoch 1/5
1875/1875          37s 16ms/step - accuracy: 0.9295 - loss: 0.2423
Epoch 2/5
1875/1875          26s 14ms/step - accuracy: 0.9694 - loss: 0.1008
Epoch 3/5
1875/1875          48s 17ms/step - accuracy: 0.9776 - loss: 0.0705
Epoch 4/5
1875/1875          30s 16ms/step - accuracy: 0.9827 - loss: 0.0545
Epoch 5/5
1875/1875          44s 17ms/step - accuracy: 0.9861 - loss: 0.0426
Training Accuracy: 0.9861166477203369
```

Question 9: Visualize the loss and accuracy curves for a neural network model trained on the Fashion MNIST dataset. Interpret the training behavior.

Answer:

```
import tensorflow as tf
from tensorflow.keras.datasets import fashion_mnist
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, Flatten
import matplotlib.pyplot as plt

# Load Fashion MNIST dataset
(x_train, y_train), (x_test, y_test) = fashion_mnist.load_data()

# Normalize pixel values
x_train = x_train / 255.0
x_test = x_test / 255.0

# Build the neural network
model = Sequential([
    Flatten(input_shape=(28, 28)),
    Dense(128, activation='relu'),
    Dense(64, activation='relu'),
    Dense(10, activation='softmax')
])
```

```

# Compile the model
model.compile(
    optimizer='adam',
    loss='sparse_categorical_crossentropy',
    metrics=['accuracy']
)

# Train the model
history = model.fit(
    x_train,
    y_train,
    epochs=10,
    validation_data=(x_test, y_test),
    verbose=1
)

# Plot Training and Validation Accuracy
plt.figure()
plt.plot(history.history['accuracy'])
plt.plot(history.history['val_accuracy'])
plt.title("Model Accuracy")
plt.xlabel("Epoch")
plt.ylabel("Accuracy")
plt.legend(["Training Accuracy", "Validation Accuracy"])
plt.show()

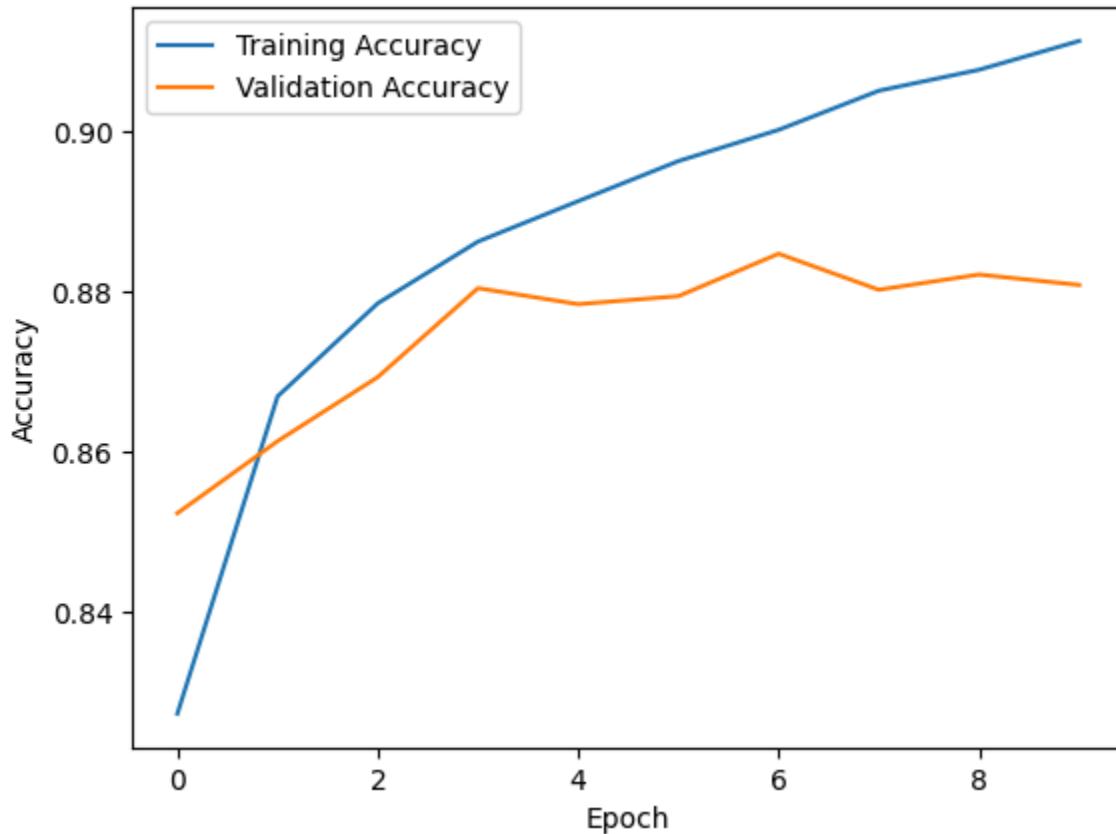
# Plot Training and Validation Loss
plt.figure()
plt.plot(history.history['loss'])
plt.plot(history.history['val_loss'])
plt.title("Model Loss")
plt.xlabel("Epoch")
plt.ylabel("Loss")
plt.legend(["Training Loss", "Validation Loss"])
plt.show()

```

Output:

```
Downloading data from https://storage.googleapis.com/tensorflow/tf-keras-datasets/train-labels-idx1-ubyte.gz
29515/29515          0s 3us/step
Downloading data from https://storage.googleapis.com/tensorflow/tf-keras-datasets/train-images-idx3-ubyte.gz
26421880/26421880    16s 1us/step
Downloading data from https://storage.googleapis.com/tensorflow/tf-keras-datasets/t10k-labels-idx1-ubyte.gz
5148/5148             0s 1us/step
Downloading data from https://storage.googleapis.com/tensorflow/tf-keras-datasets/t10k-images-idx3-ubyte.gz
4422102/4422102      3s 1us/step
Epoch 1/10
1875/1875            53s 26ms/step - accuracy: 0.8273 - loss: 0.4860 - val_accuracy: 0.8523 - val_loss: 0.4123
Epoch 2/10
1875/1875            44s 24ms/step - accuracy: 0.8669 - loss: 0.3656 - val_accuracy: 0.8613 - val_loss: 0.3754
Epoch 3/10
1875/1875            105s 36ms/step - accuracy: 0.8785 - loss: 0.3292 - val_accuracy: 0.8693 - val_loss: 0.3622
Epoch 4/10
1875/1875            74s 39ms/step - accuracy: 0.8862 - loss: 0.3067 - val_accuracy: 0.8804 - val_loss: 0.3400
Epoch 5/10
1875/1875            45s 19ms/step - accuracy: 0.8913 - loss: 0.2901 - val_accuracy: 0.8784 - val_loss: 0.3425
Epoch 6/10
1875/1875            30s 16ms/step - accuracy: 0.8963 - loss: 0.2761 - val_accuracy: 0.8794 - val_loss: 0.3454
Epoch 7/10
1875/1875            86s 40ms/step - accuracy: 0.9002 - loss: 0.2653 - val_accuracy: 0.8847 - val_loss: 0.3253
Epoch 8/10
1875/1875            82s 40ms/step - accuracy: 0.9050 - loss: 0.2524 - val_accuracy: 0.8802 - val_loss: 0.3557
Epoch 9/10
1875/1875            50s 22ms/step - accuracy: 0.9077 - loss: 0.2449 - val_accuracy: 0.8821 - val_loss: 0.3349
Epoch 10/10
1875/1875            38s 20ms/step - accuracy: 0.9113 - loss: 0.2337 - val_accuracy: 0.8808 - val_loss: 0.3498
```

Model Accuracy



Question 10: You are working on a project for a bank that wants to automatically detect fraudulent transactions. The dataset is large, imbalanced, and contains structured features like transaction amount, merchant ID, and customer location. The goal is to classify each transaction as fraudulent or legitimate. Explain your real-time data science workflow:

- How would you design a deep learning model (perceptron or multilayer NN)?
- Which activation function and loss function would you use, and why?
- How would you train and evaluate the model, considering class imbalance?
- Which optimizer would be suitable, and how would you prevent overfitting?

Answer:

To address the problem of real-time fraud detection in a banking environment with a large, imbalanced, structured dataset, the following end-to-end data science workflow would be appropriate.

1. Model Design (Perceptron vs Multilayer Neural Network)

A single-layer perceptron would be insufficient because fraud patterns are complex and non-linear. Therefore, I would design a Multilayer Neural Network (MLP).

Proposed architecture:

- Input layer: Structured features (transaction amount, merchant ID embeddings, location encoding, time features, etc.)
- Hidden layers: 2–3 fully connected (Dense) layers with decreasing neurons (e.g., 128 → 64 → 32)
- Output layer: Single neuron for binary classification (fraud / legitimate)

This architecture can learn non-linear feature interactions, which are critical in fraud detection.

2. Activation Function and Loss Function

Activation Functions

- Hidden layers: ReLU
 - Efficient computation
 - Prevents vanishing gradients
 - Performs well on structured tabular data
- Output layer: Sigmoid
 - Produces probabilities between 0 and 1
 - Suitable for binary classification

Loss Function

- Binary Cross-Entropy Loss

$$L = -[y \log(\hat{y}) + (1-y) \log(1-\hat{y})]$$
- This loss penalizes incorrect fraud probability predictions and aligns well with sigmoid output.

3. Training and Evaluation (Handling Class Imbalance)

Handling Class Imbalance

Since fraudulent transactions are rare:

- Use class weights to penalize misclassification of fraud more heavily
- Alternatively, apply oversampling (SMOTE) or undersampling
- Use threshold tuning instead of default 0.5

Evaluation Metrics

Accuracy is misleading in imbalanced data. Instead, use:

- Precision (avoid false fraud alerts)
- Recall (catch maximum fraud cases)

- F1-score
- ROC-AUC / PR-AUC

Training Strategy

- Split data into train / validation / test
- Use mini-batch training
- Monitor validation metrics during training

4. Optimizer and Overfitting Prevention

Optimizer

- Adam optimizer
 - Adaptive learning rates
 - Fast convergence
 - Handles noisy and sparse gradients well

Overfitting Prevention Techniques

- Dropout layers to reduce co-adaptation
- L2 regularization on weights
- Early stopping based on validation loss
- Feature normalization and proper encoding
- Cross-validation for robustness

Real-Time Deployment Considerations

- Use a lightweight MLP for low latency

- Calibrate probabilities for decision thresholds
- Monitor data drift and retrain periodically
- Integrate with rule-based systems for high-risk transactions

Final Summary

A multilayer neural network with ReLU activations, sigmoid output, binary cross-entropy loss, Adam optimizer, and imbalance-aware training strategies provides a robust and scalable solution for real-time fraud detection while maintaining high recall and controlled false positives.