Neural Network - A Simple Perceptron Assignment

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

Answer:

Deep Learning is a subset of Machine Learning that focuses on algorithms inspired by the structure and function of the human brain, known as artificial neural networks. In other words, it basically lets computers automatically learn hierarchical representations from large amounts of data-for example, to recognize faces, understand speech, or detect objects-without explicitly having to program them to do so for every single feature.

Evolution of Deep Learning:

1. 1950s-1980s: Early Neural Networks
- The concept began with Perceptron (Frank Rosenblatt, 1958), a simple neural model.
- However, due to limited computational power and a lack of data, the going was slow.

2. 1980s–1990s: Backpropagation and ANN revival
- Introduction of the Backpropagation algorithm by Rumelhart & Hinton (1986) made learning in multi-layer neural networks efficient.
- Still, networks remained shallow due to hardware and data limitations.

3. 2000s–2010s: Rise of Deep Learning
- With powerful GPUs, large datasets, and techniques like ReLU activation, dropout, and batch normalization, deep learning became practical.
- AlexNet achievements in 2012 for image classification marked the modern deep learning era.

4. Present Day:
- Deep learning powers NLP (ChatGPT, BERT), computer vision, autonomous driving, and speech recognition.

Differences Between Traditional Machine Learning and Deep Learning:

1. Feature Extraction:

   Machine Learning: Features are chosen or created manually by a human.

   Deep Learning: Detects features from raw data automatically.

2. Data Requirement:

   Machine Learning: performs well with small to medium datasets.

   Deep Learning: It requires large volumes of data for accuracy.

3. Computation Power:

   Machine Learning: Can run on normal CPUs.

Deep Learning: Requires powerful GPUs and high computational resources.

4. Training Time:

   Machine Learning: Trains faster and requires less time.

   Deep Learning: it takes longer to train because of the complexity in network structures.

5. Performance on Complex Data:

   Machine Learning: Less effective for high-dimensional or unstructured data like images, audio, and text.

   Deep Learning handles unstructured and complex data quite well.

6. Model Interpretability:
   Machine Learning: Easier to interpret and understand the results.
   Deep Learning: A “black box” – harder to interpret.
7. Dependency of hardware:

   Machine Learning: Works fine on basic hardware.

   Deep Learning: Relies heavily on advanced hardware, GPUs/TPUs.

Examples:

- Machine Learning: Linear Regression, Decision Trees, Random Forest.
- Deep Learning: Convolutional Neural Networks (CNN), Recurrent Neural Networks (RNN), Transformers.

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

Answer:

A Perceptron is the simplest form of an artificial neural network, which was introduced by Frank Rosenblatt in 1958.

It's a binary classifier; its purpose is to assign either of two classes to an input, based on a linear function of its inputs.

It forms the building block of modern neural networks.

Basic Architecture of a Perceptron

A single layer perceptron consists of:

- Input layer: Takes input features x_1, x_2, ., xn.
- Weights (w1, w2, ….,wn) – Every input comes with weight association, which shows its importance.
- Summation function: Calculates the weighted sum of inputs:
- Activation function: Applies a step function to decide the output, 0 or 1.

Functioning of a Perceptron

Step-by-step process:

- Input: Provide feature values to the input layer.
- Weighted Sum: Multiply each of the inputs by its corresponding weight and add them, including the bias.
- Activation: Take the step function:.

  if $z \geq 0$

  if $z < 0$

- The perceptron outputs 1 or 0 based on the activation function.

Limitations of a Perceptron

- Can only solve problems that are linearly separable, e.g., works for AND, OR but fails for XOR.
- Limited learning capacity: Cannot model complex relationships.
- Single-layer structure: there are no hidden layers that might extract deeper features.
- Sensitive to input scaling: requires proper normalisation of data.
- Does not handle noisy or overlapping data well.

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

Answer:

Purpose:

Introduction of nonlinearities: Activation functions introduce non-linearity in a neural network. Without them, the network would behave like a simple linear model, no matter how many layers it has.

They help in the learning of complex relationships and patterns within data.

Main roles of activation functions:

Introduce non-linearity: The network learns from complex data, not just straight-line relationships.

Control neuron output: decide whether a neuron should fire or not.

Help gradient flow: change the way errors are backpropagated during training. Mapping inputs to a specific range: This helps in stabilizing the training and avoiding exploding or vanishing gradients.

Comparison of Activation Functions

1. Sigmoid

- Output range: 0 to 1
- Used mostly for binary classification outputs

- Smooth and easy to interpret as probability
- Disadvantages:
    - Causes **vanishing gradients** (slow learning)
    - Not zero-centered (can slow down training)

2. Tanh

- Output range: -1 to 1
- Zero-centered (better than sigmoid for hidden layers)
- Produces stronger gradients
- Disadvantages:
    - Still faces vanishing gradient problem
    - Slightly slower due to complex computation

3. ReLU (Rectified Linear Unit)

- Output range: 0 to infinity
- Very fast and efficient to compute
- Reduces vanishing gradient issue
- Disadvantages:
    - Can cause **dead neurons** (some neurons stop learning)
    - Output not zero-centered

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

Answer:

1. Loss Function

- The loss function measures the error for a single training example.
- It shows how far the model's prediction is from the actual output for one data point.
- It helps the network learn by updating the weights during training.

Examples:

- Mean Squared Error (MSE): Used for regression problems.
- Binary Cross-Entropy: This is used for binary classification.
- Categorical Cross-Entropy: Used in multi-class classification.

2. Cost Function

- It is the average of all individual losses over the entire training dataset or a batch.
- It gives a general idea of the performance of the model.
- The aim of training is to minimize the cost function.

Examples:

- Average of all MSE values across the dataset.
- Average of all Cross-Entropy losses in a batch.

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

Answer :

Role of Optimizers

1. Optimizers are algorithms used to update the weights of a neural network to minimize the cost (error).
2. They decide how fast and in what direction the model learns during training.
3. The main goal is to find the best set of weights that give the least error.
4. They use the gradients (from backpropagation) to adjust weights efficiently.

Comparison of Optimizers

1. Gradient Descent

- Updates weights opposite to the direction of the gradient.
- Simple and easy to implement.
- Works well for small datasets.

Disadvantages:

- Can be slow to converge.
- May get stuck in local minima.
- Needs a properly chosen learning rate.

2. RMSprop - Root Mean Square Propagation

- Automatically adjusts the learning rate for each parameter.
- It uses moving average of past squared gradients.

Advantages:

- Faster convergence compared to basic Gradient Descent.
- Works well for non-stationary data and recurrent networks.

Disadvantages:

- Sensitive to the tuning of hyperparameters.
- Might not work well for every dataset.

3. Adam (Adaptive Moment Estimation)

- Combines the ideas behind momentum and RMSprop.

- Maintains both the average of past gradients and squared gradients.

Advantages:

- Fast and efficient for large data sets.
- Requires less tuning of the learning rate.
- Works well for deep and complex neural networks.

Disadvantages:

- can sometimes overshoot the minimum.
- Uses slightly more memory.

● **Use NumPy, Matplotlib, and Tensorflow/Keras for implementation.**

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

Answer :

CODE :

```
import numpy as np


# Step 1: Define input (X) and output (y) for AND gate

X = np.array([[0, 0],

              [0, 1],

              [1, 0],

              [1, 1]])


y = np.array([0, 0, 0, 1])  # Expected output
```

```
# Step 2: Initialize weights and bias
weights = np.zeros(2)
bias = 0
learning_rate = 0.1
epochs = 10

# Step 3: Define activation function (Step Function)
def step_function(x):
    return 1 if x >= 0 else 0

# Step 4: Training the Perceptron
for epoch in range(epochs):
    print(f"Epoch {epoch+1}")
    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 rule
        weights += learning_rate * error * X[i]
        bias += learning_rate * error

        print(f"Input: {X[i]}, Predicted: {y_pred}, Error: {error}")

print("\nFinal Weights:", weights)
print("Final Bias:", bias)

# Step 5: Testing the trained Perceptron
print("\nTesting the AND Gate:")
for i in range(len(X)):
    output = step_function(np.dot(X[i], weights) + bias)
```

```
    print(f"Input: {X[i]} → Output: {output}")
```

OUTPUT :

```
Epoch 1
Input: [0 0], Predicted: 1, Error: -1
Input: [0 1], Predicted: 0, Error: 0
Input: [1 0], Predicted: 0, Error: 0
Input: [1 1], Predicted: 0, Error: 1
Epoch 2
Input: [0 0], Predicted: 1, Error: -1
Input: [0 1], Predicted: 1, Error: -1
Input: [1 0], Predicted: 0, Error: 0
Input: [1 1], Predicted: 0, Error: 1
Epoch 3
Input: [0 0], Predicted: 0, Error: 0
Input: [0 1], Predicted: 1, Error: -1
Input: [1 0], Predicted: 1, Error: -1
Input: [1 1], Predicted: 0, Error: 1
Epoch 4
Input: [0 0], Predicted: 0, Error: 0
Input: [0 1], Predicted: 0, Error: 0
Input: [1 0], Predicted: 0, Error: 0
Input: [1 1], Predicted: 1, Error: 0
Epoch 5
Input: [0 0], Predicted: 0, Error: 0
Input: [0 1], Predicted: 0, Error: 0
Input: [1 0], Predicted: 0, Error: 0
Input: [1 1], Predicted: 1, Error: 0
Epoch 6
Input: [0 0], Predicted: 0, Error: 0
Input: [0 1], Predicted: 0, Error: 0
```

Input: [1 0], Predicted: 0, Error: 0

Input: [1 1], Predicted: 1, Error: 0

Epoch 7

Input: [0 0], Predicted: 0, Error: 0

Input: [0 1], Predicted: 0, Error: 0

Input: [1 0], Predicted: 0, Error: 0

Input: [1 1], Predicted: 1, Error: 0

Epoch 8

Input: [0 0], Predicted: 0, Error: 0

Input: [0 1], Predicted: 0, Error: 0

Input: [1 0], Predicted: 0, Error: 0

Input: [1 1], Predicted: 1, Error: 0

Epoch 9

Input: [0 0], Predicted: 0, Error: 0

Input: [0 1], Predicted: 0, Error: 0

Input: [1 0], Predicted: 0, Error: 0

Input: [1 1], Predicted: 1, Error: 0

Epoch 10

Input: [0 0], Predicted: 0, Error: 0

Input: [0 1], Predicted: 0, Error: 0

Input: [1 0], Predicted: 0, Error: 0

Input: [1 1], Predicted: 1, Error: 0

Final Weights: [0.2 0.1]

Final Bias: -0.20000000000000004

Testing the AND Gate:

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.

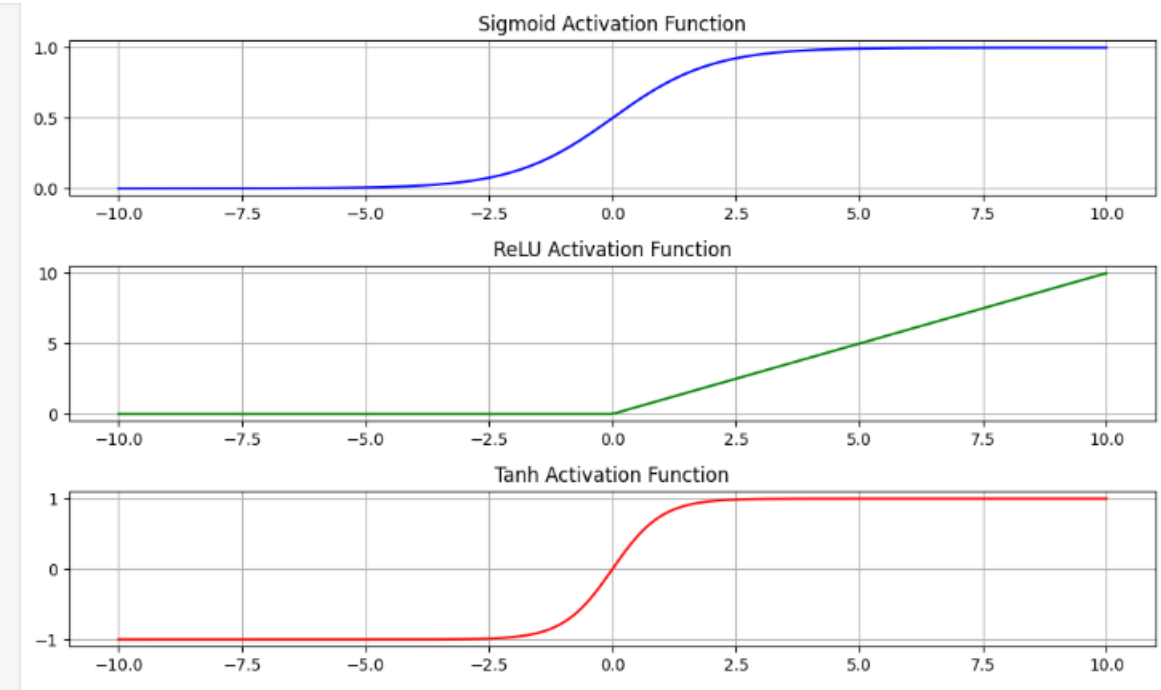
Answer :

CODE :

```
import numpy as np

import matplotlib.pyplot as plt


# Step 1: Create a range of input values

x = np.linspace(-10, 10, 200)


# Step 2: Define 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)


# Step 3: Compute outputs for each activation

y_sigmoid = sigmoid(x)

y_relu = relu(x)

y_tanh = tanh(x)


# Step 4: Plot all activation functions

plt.figure(figsize=(10, 6))


# Plot Sigmoid

plt.subplot(3, 1, 1)

plt.plot(x, y_sigmoid, color='blue')
```

```
plt.title('Sigmoid Activation Function')
plt.grid(True)

# Plot ReLU
plt.subplot(3, 1, 2)
plt.plot(x, y_relu, color='green')
plt.title('ReLU Activation Function')
plt.grid(True)

# Plot Tanh
plt.subplot(3, 1, 3)
plt.plot(x, y_tanh, color='red')
plt.title('Tanh Activation Function')
plt.grid(True)
plt.tight_layout()
plt.show()
```

OUPUT:



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

Answer :

CODE :

```
# Import required libraries
import tensorflow as tf
from tensorflow import keras
from tensorflow.keras import layers

# Step 1: Load MNIST dataset
(x_train, y_train), (x_test, y_test) = keras.datasets.mnist.load_data()

# Step 2: Preprocess data
x_train = x_train.reshape((x_train.shape[0], 28 * 28)).astype("float32") / 255
x_test = x_test.reshape((x_test.shape[0], 28 * 28)).astype("float32") / 255

# Step 3: Build the neural network model
model = keras.Sequential([
    layers.Dense(128, activation='relu', input_shape=(784,)),  # Hidden layer 1
    layers.Dense(64, activation='relu'),                       # Hidden layer 2
    layers.Dense(10, activation='softmax')                     # Output layer (10 classes)
])

# Step 4: Compile the model
model.compile(optimizer='adam',
              loss='sparse_categorical_crossentropy',
              metrics=['accuracy'])

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

```
# Step 6: Print the training accuracy

train_accuracy = history.history['accuracy'][-1]

print(f"\n✅ Training Accuracy: {train_accuracy * 100:.2f}%")
```

OUTPUT :

```
Epoch 1/5
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 7s 3ms/step - accuracy: 0.9296 - loss: 0.2401
Epoch 2/5
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 5s 3ms/step - accuracy: 0.9694 - loss: 0.1005
Epoch 3/5
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 6s 3ms/step - accuracy: 0.9784 - loss: 0.0693
Epoch 4/5
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 6s 3ms/step - accuracy: 0.9831 - loss: 0.0533
Epoch 5/5
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 5s 3ms/step - accuracy: 0.9861 - loss: 0.0431

✅ Training Accuracy: 98.61%
```

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

Anwer :

CODE :

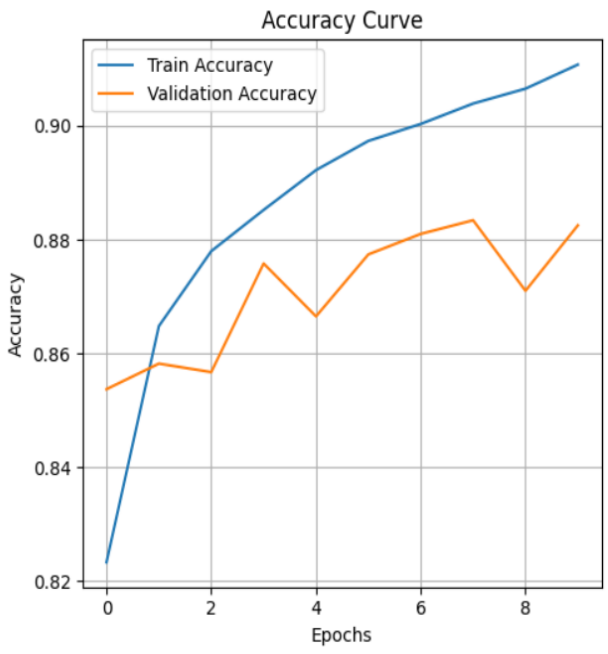
```
import tensorflow as tf

from tensorflow import keras

from tensorflow.keras import layers

import matplotlib.pyplot as plt


# Step 1: Load Fashion MNIST dataset

(x_train, y_train), (x_test, y_test) = keras.datasets.fashion_mnist.load_data()


# Step 2: Preprocess data

x_train = x_train.reshape((x_train.shape[0], 28 * 28)).astype("float32") / 255

x_test = x_test.reshape((x_test.shape[0], 28 * 28)).astype("float32") / 255


# Step 3: Build a simple neural network model
```

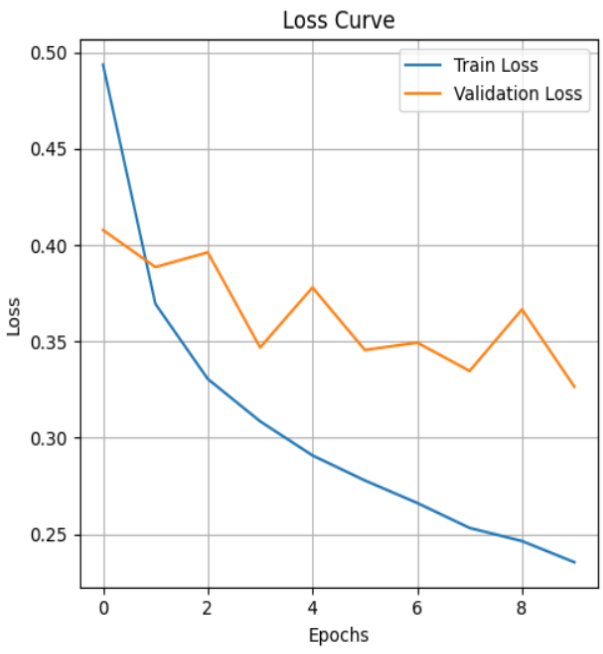
```
model = keras.Sequential([
    layers.Dense(128, activation='relu', input_shape=(784,)),
    layers.Dense(64, activation='relu'),
    layers.Dense(10, activation='softmax')
])

# Step 4: Compile the model
model.compile(optimizer='adam',
              loss='sparse_categorical_crossentropy',
              metrics=['accuracy'])

# Step 5: Train the model
history = model.fit(x_train, y_train, epochs=10, batch_size=32,
                    validation_data=(x_test, y_test), verbose=1)

# Step 6: Visualize Loss and Accuracy
plt.figure(figsize=(10, 5))

# Plot training & validation accuracy
plt.subplot(1, 2, 1)
plt.plot(history.history['accuracy'], label='Train Accuracy')
plt.plot(history.history['val_accuracy'], label='Validation Accuracy')
plt.title('Accuracy Curve')
plt.xlabel('Epochs')
plt.ylabel('Accuracy')
plt.legend()
plt.grid(True)

# Plot training & validation loss
plt.subplot(1, 2, 2)
plt.plot(history.history['loss'], label='Train Loss')
```

```
plt.plot(history.history['val_loss'], label='Validation Loss')
plt.title('Loss Curve')
plt.xlabel('Epochs')
plt.ylabel('Loss')
plt.legend()
plt.grid(True)

plt.tight_layout()
plt.show()
```

OUTPUT :

```
Downloading data from https://storage.googleapis.com/tensorflow/tf-keras-datasets/train-labels-idx1-ubyte.gz
29515/29515 ━━━━━━━━━━━━━━━━━━━━ 0s 1us/step
Downloading data from https://storage.googleapis.com/tensorflow/tf-keras-datasets/train-images-idx3-ubyte.gz
26421880/26421880 ━━━━━━━━━━━━━━━━━━━━ 5s 0us/step
Downloading data from https://storage.googleapis.com/tensorflow/tf-keras-datasets/t10k-labels-idx1-ubyte.gz
5148/5148 ━━━━━━━━━━━━━━━━━━━━ 0s 0us/step
Downloading data from https://storage.googleapis.com/tensorflow/tf-keras-datasets/t10k-images-idx3-ubyte.gz
4422102/4422102 ━━━━━━━━━━━━━━━━━━━━ 1s 0us/step
Epoch 1/10
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 8s 4ms/step - accuracy: 0.8233 - loss: 0.4936 - val_accuracy: 0.8537 - val_loss: 0.4078
Epoch 2/10
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 6s 3ms/step - accuracy: 0.8648 - loss: 0.3695 - val_accuracy: 0.8582 - val_loss: 0.3886
Epoch 3/10
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 5s 3ms/step - accuracy: 0.8780 - loss: 0.3306 - val_accuracy: 0.8567 - val_loss: 0.3962
Epoch 4/10
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 6s 3ms/step - accuracy: 0.8852 - loss: 0.3086 - val_accuracy: 0.8758 - val_loss: 0.3469
Epoch 5/10
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 6s 3ms/step - accuracy: 0.8922 - loss: 0.2909 - val_accuracy: 0.8665 - val_loss: 0.3780
Epoch 6/10
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 5s 3ms/step - accuracy: 0.8974 - loss: 0.2779 - val_accuracy: 0.8774 - val_loss: 0.3456
Epoch 7/10
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 6s 3ms/step - accuracy: 0.9003 - loss: 0.2662 - val_accuracy: 0.8810 - val_loss: 0.3494
Epoch 8/10
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 7s 3ms/step - accuracy: 0.9039 - loss: 0.2533 - val_accuracy: 0.8834 - val_loss: 0.3347
Epoch 9/10
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 5s 3ms/step - accuracy: 0.9065 - loss: 0.2465 - val_accuracy: 0.8710 - val_loss: 0.3667
Epoch 10/10
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 6s 3ms/step - accuracy: 0.9108 - loss: 0.2355 - val_accuracy: 0.8825 - val_loss: 0.3265
```





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 :

1. Model design: perceptron or multilayer NN?

A small to medium-sized multilayer feedforward NN (MLP/dense network) is chosen instead of a single perceptron because, for fraud patterns, interactions and nonlinearity are important.

Recommended Input Pipeline

- Numerical features: normalized (StandardScaler or robust scaler).
- Categorical features (merchant ID, device id, country, channel): Use learned embeddings (dense vectors) - best for high cardinality fields, outperforms one-hot encoding.
- Time features: Cyclic encoding (hour/day-of-week) + engineered drift/seasonality features.
- Behavioral / Aggregates: rolling count, sum, mean, time window 1h/24h/7d, velocity features.

Example Architecture - Simple, Effective

- Input →numeric batchnorm → embeddings concatenated → Dense(128, ReLU) → Dropout(0.3) → Dense(64, ReLU) → Dropout(0.2) → Dense(32, ReLU) → Dense(1, Sigmoid)
- Optionally add residual connections or a small wide branch that uses raw features (wide & deep)

Why this shape: compact capacity to learn nonlinear interactions, embeddings capture categorical similarity, dropout+batchnorm stabilize training.

2. Activation function and loss function (and why)

- Hidden layers: ReLU (or variants like LeakyReLU)

  Why: simple, fast, reduces vanishing gradients, good empirical performance.

- Output layer: Sigmoid (one neuron) for the probability of fraud.

  Why: binary classification, outputs interpretable probability for thresholding.

- Loss function (recommendations):

  Binary Cross-Entropy (BCE) with class weights - baseline.

  Use weight inversely proportional to class frequency or based on business cost.

  Focal Loss as stronger alternative: recommended when the positive class is extremely rare, and many easy negatives dominate.

  Why: downweights easy negatives, focuses learning on hard positives/false negatives.

- Practical: first try BCE with class_weight, then switch to focal loss if model fails to focus on the minority class, tuning gamma ≈ 1–3.

3. Training and evaluation considering class imbalance

Data handling

Train/validation split: Time-based split - train on older windows, validate on more recent to respect temporal drift.

Class balancing strategies (try in order):

- Class weights in loss function (simple and often effective).
- Oversampling positives in training batches (SMOTE for tabular can help but be careful with time/feature leakage).
- Undersampling negatives with an ensemble of the different undersamples to maintain information.
- Hybrid: Weighted loss + mild oversampling.

Batching: make sure each mini-batch contains some positives, stratified batching, to stabilize gradients.

Evaluation metrics: use many, prioritize appropriate ones

Primary metrics for imbalanced fraud:

- Precision, Recall (Sensitivity) — especially recall (catch fraud) with the precision cost.
- Precision–Recall AUC (AUC-PR) — more informative than ROC-AUC in cases where positives are rare.
- ROC-AUC still useful for overall separability.

Business-aware metrics:

- The cost-sensitive metric is: expected monetary loss saved = TP * fraud_value - FP * customer_cost or operational cost.
- Detection latency and false positive rate per 1000 transactions.

Calibration & thresholding:

- Calibration probabilities if decision thresholds are critical: Platt scaling/isotonic.
- Sweep thresholds on validation to pick point that maximizes expected business utility - not just F1.

Cross-validation: use time-series cross-validation / rolling windows instead of random k-fold.

Monitoring during training

- Tracking per-class loss, precision@k, recall@k, AUC-PR, and confusion matrix on the validation set.
- Log model checkpoints based on either validation AUC-PR or business metric.

4) Optimizer choice and overfitting prevention

Optimizer

- Adam (or AdamW) as default.

Why: Adaptive learning rates speed up training; robust to noisy gradients; widely used.

Start with lr=1e-3, try 1e-4 if unstable; use weight decay (AdamW) to regularize.

- Use a learning rate scheduler to improve convergence like ReduceLROnPlateau or cosine annealing.

Overfitting prevention

- Regularization:

L2 weight decay: recommended, e.g., 1e-4 – 1e-3

Dropout on dense layers: 0.2–0.4.

Batch normalization to stabilize distributions.

- Early stopping on the validation AUC-PR with patience, such as 5–10 epochs.
- Feature selection/pruning: remove noisy or unstable features, reduce embedding dimensions for rare categories.
- Ensembling: train several models using different seeds/undersamples and average the predictions; this reduces variance.
- Data augmentation tabular: cautious synthetic minority generation or simulation of plausible fraud patterns.
- Monitor concept drift: keep drift detectors on feature distributions and label distribution, trigger retraining when drift crosses threshold.
- Model complexity control: Prefer smaller models that generalize; only enlarge when clear underfitting.