ASS 2

```
# 1. Import necessary libraries
import tensorflow as tf
from keras.models import Sequential
from keras.datasets import mnist
from keras import layers
import matplotlib.pyplot as plt
import numpy as np
import random

 2. Load MNIST dataset

(x_train, y_train), (x_test, y_test) = mnist.load_data()
len(x_train)
len(x_test)
len(y_train)
len(y_test)
x_train.shape

# 2. Normalize data (convert pixel values 0–255 → 0–1)
x_train = x_train / 255.0
x_test = x_test / 255.0

# 3. Define the feedforward network architecture Define the network architecture using Keras
model = Sequential()
model.add(layers.Flatten(input_shape=(28, 28)))       # Flatten 28x28 → 784
model.add(layers.Dense(128, activation='relu'))       # Hidden layer
model.add(layers.Dense(10, activation='softmax'))     # Output for 10 digits
model.summary()

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

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

# 5.A. Evaluate the model
test_loss,test_acc=model.evaluate(x_test,y_test)
print("Final Test Loss=%.3f" %test_loss)
print("Final Test Accuracy=%.3f" %test_acc)

# 5.B. Predict on a random test image
n = random.randint(0, 9999)

plt.imshow(x_test[n])
plt.show()

plt.imshow(x_test[n], cmap='gray')
plt.title("Test Image")
plt.axis('off')
plt.show()

predictions = model.predict(x_test)
print("Predicted Number:", np.argmax(predictions[n]))

# 6.A Plot training loss and accuracy
plt.figure(figsize=(12, 5))

# Plot training vs validation accuracy
plt.subplot(1, 2, 1)
plt.plot(history.history['accuracy'], label='Train Accuracy')
plt.plot(history.history['val_accuracy'], label='Test Accuracy')
plt.title('Model Accuracy')
plt.xlabel('Epoch')
plt.ylabel('Accuracy')
plt.legend()

# Plot training vs validation loss
plt.subplot(1, 2, 2)
plt.plot(history.history['loss'], label='Train Loss')
plt.plot(history.history['val_loss'], label='Test Loss')
plt.title('Model Loss')
plt.xlabel('Epoch')
plt.ylabel('Loss')
plt.legend()

plt.show()

## 6.A Plot training loss and accuracy
plt.plot(history.history['accuracy'])
plt.plot(history.history['val_accuracy'])
plt.plot(history.history['loss'])
plt.plot(history.history['val_loss'])
plt.title('Training Loss and accuracy')
plt.ylabel('accuracy/Loss')
plt.xlabel('epoch')
plt.legend(['accuracy', 'val_accuracy','loss','val_loss'])
plt.show()
```

==

Ass3

```
# ============================================================
# a. LOADING AND PREPROCESSING THE IMAGE DATA
# ============================================================
import numpy as np
import tensorflow as tf
import matplotlib.pyplot as plt
import random
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Flatten, Conv2D, Dense, MaxPooling2D
from tensorflow.keras.optimizers import SGD
from tensorflow.keras.datasets import mnist

# Load MNIST dataset
(X_train, y_train), (X_test, y_test) = mnist.load_data()
print("Training data shape:", X_train.shape)
print("Testing data shape:", X_test.shape)

# Normalize pixel values to range [0, 1]
X_train = X_train.astype('float32') / 255.0
X_test = X_test.astype('float32') / 255.0

# Reshape data to include channel dimension
X_train = X_train.reshape((X_train.shape[0], 28, 28, 1))
X_test = X_test.reshape((X_test.shape[0], 28, 28, 1))

# Plot sample images
def plot_digit(image, digit, plt, i):
    plt.subplot(4, 5, i + 1)
    plt.imshow(image.squeeze(), cmap='gray')
    plt.title(f"Digit: {digit}")
    plt.axis('off')

plt.figure(figsize=(12, 8))
for i in range(20):
    plot_digit(X_train[i], y_train[i], plt, i)
plt.show()

# Split training data into train and validation sets
X_train, X_val, y_train, y_val = train_test_split(
    X_train, y_train, test_size=0.2, random_state=42
)

# ============================================================
# b. DEFINING THE MODEL’S ARCHITECTURE
# ============================================================
model = Sequential([
    Conv2D(32, (3, 3), activation="relu", input_shape=(28, 28, 1)),
    MaxPooling2D((2, 2)),
    Flatten(),
    Dense(100, activation="relu"),
    Dense(10, activation="softmax")
])

# Compile model
optimizer = SGD(learning_rate=0.01, momentum=0.9)
model.compile(optimizer=optimizer,
              loss="sparse_categorical_crossentropy",
              metrics=["accuracy"])
model.summary()

# ============================================================
# c. TRAINING THE MODEL
# ============================================================
Model_log = model.fit(
    X_train,
    y_train,
    epochs=10,
    batch_size=15,
    verbose=1,
    validation_data=(X_val, y_val)
)

# ============================================================
# d. ESTIMATING THE MODEL'S PERFORMANCE
# ============================================================
# Evaluate on test set
test_loss, test_acc = model.evaluate(X_test, y_test, verbose=0)
print(f"\nTest Accuracy: {test_acc * 100:.2f}%")

# Predict and visualize results on random test samples
plt.figure(figsize=(12, 8))
for i in range(20):
    image = random.choice(X_test)
    true_label = y_test[random.randint(0, len(y_test) - 1)]
    pred_label = np.argmax(model.predict(image.reshape((1, 28, 28, 1))), axis=-1)[0]
    plot_digit(image, f"Pred: {pred_label}", plt, i)
plt.show()

# Compute accuracy using sklearn
predictions = np.argmax(model.predict(X_test), axis=-1)
print("Accuracy Score (sklearn):", accuracy_score(y_test, predictions))

# Display a random test image
n = random.randint(0, len(X_test) - 1)
plt.imshow(X_test[n].squeeze(), cmap='gray')
plt.title(f"Actual: {y_test[n]}, Predicted: {predictions[n]}")
plt.axis('off')
plt.show()
```

==

ass4

```
# ============================================================
# a. IMPORT REQUIRED LIBRARIES
# ============================================================

import pandas as pd
import numpy as np
import tensorflow as tf
import matplotlib.pyplot as plt
import seaborn as sns

from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from sklearn.metrics import confusion_matrix, recall_score, accuracy_score, precision_score

# Constants
RANDOM_SEED = 2021
TEST_PCT = 0.3
LABELS = ["Normal", "Fraud"]
```

```
# ============================================================
# b. UPLOAD / ACCESS THE DATASET
# ============================================================

# Load dataset (ensure creditcard.csv is in your working directory)
dataset = pd.read_csv("creditcard.csv")

print("Any nulls in the dataset:", dataset.isnull().values.any())
print('-------')
print("No. of unique labels:", len(dataset['Class'].unique()))
print("Label values:", dataset.Class.unique())
print('-------')
print("Breakdown of Normal and Fraud Transactions:")
print(pd.value_counts(dataset['Class'], sort=True))

# Visualization of class distribution
count_classes = pd.value_counts(dataset['Class'], sort=True)
count_classes.plot(kind='bar', rot=0)
plt.xticks(range(len(dataset['Class'].unique())), dataset.Class.unique())
plt.title("Frequency by Observation Number")
plt.xlabel("Class")
plt.ylabel("Number of Observations")
plt.show()

# Separate normal and fraud transactions
normal_dataset = dataset[dataset.Class == 0]
fraud_dataset = dataset[dataset.Class == 1]

# Visualize transaction amounts
bins = np.linspace(200, 2500, 100)
plt.hist(normal_dataset.Amount, bins=bins, alpha=1, density=True, label='Normal')
plt.hist(fraud_dataset.Amount, bins=bins, alpha=0.5, density=True, label='Fraud')
plt.legend(loc='upper right')
plt.title("Transaction Amount vs Percentage of Transactions")
plt.xlabel("Transaction Amount (USD)")
plt.ylabel("Percentage of Transactions")
plt.show()

# Scale Time and Amount columns
sc = StandardScaler()
dataset['Time'] = sc.fit_transform(dataset['Time'].values.reshape(-1, 1))
dataset['Amount'] = sc.fit_transform(dataset['Amount'].values.reshape(-1, 1))

# Separate features and labels
raw_data = dataset.values
labels = raw_data[:, -1]
data = raw_data[:, 0:-1]

# Split dataset
train_data, test_data, train_labels, test_labels = train_test_split(
    data, labels, test_size=0.2, random_state=RANDOM_SEED
)

# Normalize data
min_val = tf.reduce_min(train_data)
max_val = tf.reduce_max(train_data)
train_data = (train_data - min_val) / (max_val - min_val)
test_data = (test_data - min_val) / (max_val - min_val)

train_data = tf.cast(train_data, tf.float32)
test_data = tf.cast(test_data, tf.float32)

# Convert labels to boolean
train_labels = train_labels.astype(bool)
test_labels = test_labels.astype(bool)

# Create Normal and Fraud subsets
normal_train_data = train_data[~train_labels]
normal_test_data = test_data[~test_labels]
fraud_train_data = train_data[train_labels]
fraud_test_data = test_data[test_labels]

print("No. of records in Fraud Train Data =", len(fraud_train_data))
print("No. of records in Normal Train Data =", len(normal_train_data))
print("No. of records in Fraud Test Data =", len(fraud_test_data))
print("No. of records in Normal Test Data =", len(normal_test_data))

# ============================================================
# c. ENCODER CONVERTS INPUT INTO LATENT REPRESENTATION
# ============================================================

nb_epoch = 50
batch_size = 64
input_dim = normal_train_data.shape[1]  # Number of features (30)
encoding_dim = 14
hidden_dim1 = int(encoding_dim / 2)
hidden_dim2 = 4
learning_rate = 1e-7

# Input Layer
input_layer = tf.keras.layers.Input(shape=(input_dim,))

# Encoder layers
encoder = tf.keras.layers.Dense(
    encoding_dim, activation="tanh",
    activity_regularizer=tf.keras.regularizers.l2(learning_rate)
)(input_layer)
encoder = tf.keras.layers.Dropout(0.2)(encoder)
encoder = tf.keras.layers.Dense(hidden_dim1, activation='relu')(encoder)
encoder = tf.keras.layers.Dense(hidden_dim2, activation=tf.nn.leaky_relu)(encoder)

# ============================================================
# d. DECODER NETWORK CONVERTS IT BACK TO ORIGINAL INPUT
# ============================================================

decoder = tf.keras.layers.Dense(hidden_dim1, activation='relu')(encoder)
decoder = tf.keras.layers.Dropout(0.2)(decoder)
decoder = tf.keras.layers.Dense(encoding_dim, activation='relu')(decoder)
decoder = tf.keras.layers.Dense(input_dim, activation='tanh')(decoder)

# Combine Encoder + Decoder into Autoencoder
autoencoder = tf.keras.Model(inputs=input_layer, outputs=decoder)
autoencoder.summary()

# ============================================================
# e. COMPILE MODEL WITH OPTIMIZER, LOSS, AND EVALUATION METRICS
# ============================================================

# Define callbacks
cp = tf.keras.callbacks.ModelCheckpoint(
    filepath="autoencoder_fraud.keras", mode='min',
    monitor='val_loss', verbose=2, save_best_only=True
)

early_stop = tf.keras.callbacks.EarlyStopping(
    monitor='val_loss', min_delta=0.0001, patience=10,
    verbose=11, mode='min', restore_best_weights=True
)

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

# Train the model (on normal transactions only)
history = autoencoder.fit(
    normal_train_data, normal_train_data,
    epochs=nb_epoch,
    batch_size=batch_size,
    shuffle=True,
    validation_data=(test_data, test_data),
    verbose=1,
    callbacks=[cp, early_stop]
).history

# Plot training vs validation loss
plt.plot(history['loss'], linewidth=2, label='Train')
plt.plot(history['val_loss'], linewidth=2, label='Test')
plt.legend(loc='upper right')
plt.title('Model Loss')
plt.ylabel('Loss')
plt.xlabel('Epochs')
plt.show()

# ============================================================
# 🔍 ANOMALY DETECTION AND EVALUATION
# ============================================================

# Predict and compute reconstruction errors
test_x_predictions = autoencoder.predict(test_data)
mse = np.mean(np.power(test_data - test_x_predictions, 2), axis=1)
error_df = pd.DataFrame({'Reconstruction_error': mse, 'True_class': test_labels})

# Visualize reconstruction error
threshold_fixed = 50
groups = error_df.groupby('True_class')
fig, ax = plt.subplots()

for name, group in groups:
    ax.plot(group.index, group.Reconstruction_error, marker='o', ms=3.5, linestyle='',
            label="Fraud" if name == 1 else "Normal")

ax.hlines(threshold_fixed, ax.get_xlim()[0], ax.get_xlim()[1], colors="r", zorder=100, label="Threshold")
ax.legend()
plt.title("Reconstruction Error for Normal and Fraud Data")
plt.ylabel("Reconstruction Error")
plt.xlabel("Data Point Index")
plt.show()

# Apply threshold to detect anomalies
threshold_fixed = 52
pred_y = [1 if e > threshold_fixed else 0 for e in error_df.Reconstruction_error.values]
error_df['pred'] = pred_y

# Confusion matrix
conf_matrix = confusion_matrix(error_df.True_class, pred_y)
plt.figure(figsize=(4, 4))
sns.heatmap(conf_matrix, xticklabels=LABELS, yticklabels=LABELS, annot=True, fmt="d")
plt.title("Confusion Matrix")
plt.ylabel("True Class")
plt.xlabel("Predicted Class")
plt.show()

# Evaluation metrics
print("Accuracy :", accuracy_score(error_df['True_class'], error_df['pred']))
print("Recall   :", recall_score(error_df['True_class'], error_df['pred']))
print("Precision:", precision_score(error_df['True_class'], error_df['pred']))
```

==

Ass5

```
# ==============================================================
# BE(IT)/2019 Pattern/LP-IV | Chit #4
# Continuous Bag of Words (CBOW) Model Implementation
# ==============================================================

# a. Import required libraries
import numpy as np
import re
import tensorflow as tf
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, Embedding, Lambda
from tensorflow.keras.preprocessing.text import Tokenizer
import matplotlib.pyplot as plt
import seaborn as sns

# b. Data Preparation
data = """Deep learning (also known as deep structured learning) is part of a broader family of machine learning methods based on artificial
neural networks with representation learning. Learning can be supervised, semi-supervised or unsupervised. Deep-learning architectures such as
deep neural networks, deep belief networks, deep reinforcement learning, recurrent neural networks, convolutional neural networks and
Transformers have been applied to fields including computer vision, speech recognition, natural language processing, machine translation,
bioinformatics, drug design, medical image analysis, climate science, material inspection and board game programs, where they have produced
results comparable to and in some cases surpassing human expert performance."""
```

```
# Split into sentences
sentences = data.split('.')

# Clean sentences
clean_sent = []
for sentence in sentences:
    if sentence.strip() == "":
        continue
    sentence = re.sub('[^A-Za-z0-9]+', ' ', sentence)
    sentence = re.sub(r'(?:^| )\w (?:$| )', ' ', sentence).strip()
    sentence = sentence.lower()
    clean_sent.append(sentence)

# c. Tokenization
tokenizer = Tokenizer()
tokenizer.fit_on_texts(clean_sent)
sequences = tokenizer.texts_to_sequences(clean_sent)
word_to_index = tokenizer.word_index
index_to_word = tokenizer.index_word
vocab_size = len(word_to_index) + 1

# d. Generate training data (context-target pairs)
context_size = 2
contexts, targets = [], []

for sequence in sequences:
    for i in range(context_size, len(sequence) - context_size):
        target = sequence[i]
        context = [sequence[i - 2], sequence[i - 1], sequence[i + 1], sequence[i + 2]]
        contexts.append(context)
        targets.append(target)

X = np.array(contexts)
Y = np.array(targets)

print(f"✅ Vocabulary Size: {vocab_size}")
print(f"✅ Total Training Samples: {len(X)}")

# Display first few training samples
print("\nSample Context-Target Pairs:\n")
for i in range(5):
    ctx_words = [index_to_word[j] for j in X[i]]
    tgt_word = index_to_word[Y[i]]
    print(f"{ctx_words}  -->  {tgt_word}")

# e. Build the CBOW Model
emb_size = 10
model = Sequential([
    Embedding(input_dim=vocab_size, output_dim=emb_size, input_length=2 * context_size),
    Lambda(lambda x: tf.reduce_mean(x, axis=1)),
    Dense(256, activation='relu'),
    Dense(512, activation='relu'),
    Dense(vocab_size, activation='softmax')
])

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

# f. Train the Model
history = model.fit(X, Y, epochs=80, verbose=1)

# g. Visualize training loss and accuracy
plt.figure(figsize=(8, 5))
sns.lineplot(data=history.history)
plt.title("Training Progress (Loss & Accuracy)")
plt.xlabel("Epochs")
plt.ylabel("Values")
plt.grid(True)
plt.show()

# h. Test the model on new sentences
test_sentences = [
    "known as structured learning",
    "transformers have applied to",
    "where they produced results",
    "cases surpassing expert performance",
    "convolutional neural where"
]

print("\n--- CBOW Predictions ---\n")
for sent in test_sentences:
    test_words = sent.lower().split()
    x_test = [word_to_index.get(w, 0) for w in test_words]  # handle missing words
    x_test = np.array([x_test])

    pred = model.predict(x_test, verbose=0)
    pred_word = index_to_word.get(np.argmax(pred), "unknown")

    print(f"Context: {test_words}")
    print(f"Predicted Center Word: {pred_word}\n")
```
—---

ass5B

```
# ✅ Continuous Bag of Words (CBOW) Model Implementation

# a. Data Preparation
import matplotlib.pyplot as plt
import numpy as np
import re
%matplotlib inline

sentences = """We are about to study the idea of a computational process.
Computational processes are abstract beings that inhabit computers.
As they evolve, processes manipulate other abstract things called data.
The evolution of a process is directed by a pattern of rules
called a program. People create programs to direct processes. In effect,
we conjure the spirits of the computer with our spells."""

# Clean the data
sentences = re.sub('[^A-Za-z0-9]+', ' ', sentences)
sentences = re.sub(r'(?:^| )\w(?:$| )', ' ', sentences).strip()
sentences = sentences.lower()

# Create vocabulary
words = sentences.split()
vocab = set(words)
vocab_size = len(vocab)
embed_dim = 10
context_size = 2

word_to_ix = {word: i for i, word in enumerate(vocab)}
ix_to_word = {i: word for i, word in enumerate(vocab)}


# b. Generate Training Data
data = []
for i in range(2, len(words) - 2):
    context = [words[i - 2], words[i - 1], words[i + 1], words[i + 2]]
    target = words[i]
    data.append((context, target))
print("Sample training data:", data[:5])


# c. Train Model
embeddings = np.random.random_sample((vocab_size, embed_dim))
theta = np.random.uniform(-1, 1, (2 * context_size * embed_dim, vocab_size))

def linear(m, theta):
    return m.dot(theta)

def log_softmax(x):
    e_x = np.exp(x - np.max(x))
    return np.log(e_x / e_x.sum())

def NLLLoss(logs, targets):
    out = logs[range(len(targets)), targets]
    return -out.sum() / len(out)

def log_softmax_crossentropy_with_logits(logits, target):
    out = np.zeros_like(logits)
    out[np.arange(len(logits)), target] = 1
    softmax = np.exp(logits) / np.exp(logits).sum(axis=-1, keepdims=True)
    return (-out + softmax) / logits.shape[0]

def forward(context_idxs, theta):
    m = embeddings[context_idxs].reshape(1, -1)
    n = linear(m, theta)
    o = log_softmax(n)
    return m, n, o

def backward(preds, theta, target_idxs):
    m, n, o = preds
    dlog = log_softmax_crossentropy_with_logits(n, target_idxs)
    dw = m.T.dot(dlog)
    return dw

def optimize(theta, grad, lr=0.03):
    theta -= grad * lr
    return theta

# Training loop
epoch_losses = {}
for epoch in range(80):
    losses = []
    for context, target in data:
        context_idxs = np.array([word_to_ix[w] for w in context])
        preds = forward(context_idxs, theta)
        target_idxs = np.array([word_to_ix[target]])
        loss = NLLLoss(preds[-1], target_idxs)
        losses.append(loss)
        grad = backward(preds, theta, target_idxs)
        theta = optimize(theta, grad, lr=0.03)
    epoch_losses[epoch] = losses

# Plot training loss
ix = np.arange(0, 80)
plt.figure(figsize=(6,4))
plt.title('Epoch vs Loss', fontsize=14)
plt.plot(ix, [epoch_losses[i][0] for i in ix])
plt.xlabel('Epochs')
plt.ylabel('Loss')
plt.show()


# d. Output
def predict(words):
    context_idxs = np.array([word_to_ix[w] for w in words])
    preds = forward(context_idxs, theta)
    word = ix_to_word[np.argmax(preds[-1])]
    return word

def accuracy():
    wrong = 0
    for context, target in data:
        if predict(context) != target:
            wrong += 1
    return (1 - (wrong / len(data)))

print("Predicted word:", predict(['we', 'are', 'to', 'study']))
print("Model Accuracy:", accuracy())
```

================================

===

ass6(maam cust)

```
# (a) Import libraries & load pre-trained CNN model
import tensorflow as tf
from tensorflow.keras.applications import VGG16
from tensorflow.keras.models import Model
from tensorflow.keras.layers import Dense, Flatten, Dropout
from tensorflow.keras.preprocessing.image import ImageDataGenerator
from tensorflow.keras.optimizers import Adam
import matplotlib.pyplot as plt
import numpy as np

# Load pre-trained VGG16 (without top layers) C:\Users\Safa Quadri\Downloads
weights_path = "vgg16_weights_tf_dim_ordering_tf_kernels_notop.h5"
base_model = VGG16(weights=weights_path, include_top=False, input_shape=(32, 32, 3))

print("✅ Pre-trained VGG16 model loaded successfully.")

#  (b) Freeze parameters (lower convolutional layers) ======
for layer in base_model.layers:
    layer.trainable = False

print("✅ All base model layers frozen.")

# ====== (c) Add custom classifier layers (trainable part) ======
x = Flatten()(base_model.output)
x = Dense(256, activation='relu')(x)
x = Dropout(0.3)(x)
x = Dense(256, activation='relu')(x)
x = Dropout(0.3)(x)
predictions = Dense(10, activation='softmax')(x)

# Combine base model + classifier
model = Model(inputs=base_model.input, outputs=predictions)

print("✅ Custom classifier added successfully.")

# ====== (d) Load and preprocess dataset (CIFAR-10) ======
train_dir = "C:\Users\Safa Quadri\Downloads\archive.zip\cifar10\cifar10\train"
test_dir = "C:\Users\Safa Quadri\Downloads\archive.zip\cifar10\cifar10\test"

# Data generators for training and testing
train_datagen = ImageDataGenerator(rescale=1.0 / 255)
test_datagen = ImageDataGenerator(rescale=1.0 / 255)

train_batch_size = 5000
test_batch_size = 1000

train_generator = train_datagen.flow_from_directory(
    train_dir,
    target_size=(32, 32),
    batch_size=train_batch_size,
    class_mode='categorical'
)

test_generator = test_datagen.flow_from_directory(
    test_dir,
    target_size=(32, 32),
    batch_size=test_batch_size,
    class_mode='categorical'
)

# Extract data from generator
x_train, y_train = train_generator[0]
x_test, y_test = test_generator[0]

print("✅ Training and testing data loaded.")
print("Training samples:", len(x_train))
print("Testing samples:", len(x_test))

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

print("\n🚀 Training started...")
history = model.fit(x_train, y_train, batch_size=64, epochs=10, validation_data=(x_test, y_test))
print("✅ Training completed.")

# ====== (e) Fine-tuning (optional improvement) ======
# Unfreeze last few convolutional layers for fine-tuning
for layer in base_model.layers[-4:]:
    layer.trainable = True
```

```
# Recompile with lower learning rate
model.compile(optimizer=Adam(learning_rate=1e-5), loss='categorical_crossentropy', metrics=['accuracy'])

print("\n🔧 Fine-tuning last layers...")
model.fit(x_train, y_train, batch_size=64, epochs=5, validation_data=(x_test, y_test))
print("✅ Fine-tuning completed.")

# ====== Evaluate model performance ======
loss, accuracy = model.evaluate(x_test, y_test)
print(f"\n🎯 Test Accuracy: {accuracy * 100:.2f}%")

# ====== Prediction & Visualization ======
predicted_value = model.predict(x_test)
labels = list(test_generator.class_indices.keys())

n = 890  # index of image to visualize
plt.imshow(x_test[n])
plt.title(f"Predicted: {labels[np.argmax(predicted_value[n])]} | Actual: {labels[np.argmax(y_test[n])]}")
plt.axis("off")
plt.show()

print("✅ Visualization complete.")
```
----

ass6(gg)

```
# a. Import libraries
import tensorflow as tf
from tensorflow.keras.applications import MobileNetV2
from tensorflow.keras.models import Model
from tensorflow.keras.layers import Dense, GlobalAveragePooling2D
from tensorflow.keras.optimizers import Adam
import matplotlib.pyplot as plt
import numpy as np

# b. Load CIFAR-10 dataset
(x_train, y_train), (x_test, y_test) = tf.keras.datasets.cifar10.load_data()
x_train, x_test = x_train / 255.0, x_test / 255.0
y_train = tf.keras.utils.to_categorical(y_train, 10)
y_test  = tf.keras.utils.to_categorical(y_test, 10)

# Simulate generator-like info display
print("Train samples:", len(x_train))
print("Test samples:", len(x_test))

# Class labels (same as generator’s class_indices)
labels = ['airplane','automobile','bird','cat','deer',
          'dog','frog','horse','ship','truck']

# c. Load pre-trained CNN (MobileNetV2)
base_model = MobileNetV2(weights='imagenet', include_top=False, input_shape=(32, 32, 3))

# Freeze lower layers
for layer in base_model.layers:
    layer.trainable = False

# d. Add custom classifier
x = GlobalAveragePooling2D()(base_model.output)
x = Dense(128, activation='relu')(x)
output = Dense(10, activation='softmax')(x)
model = Model(inputs=base_model.input, outputs=output)

# e. Train classifier layers
model.compile(optimizer=Adam(), loss='categorical_crossentropy', metrics=['accuracy'])
model.fit(x_train, y_train, epochs=3, batch_size=128, validation_split=0.1, verbose=1)

# f. Fine-tune entire model (simple one-line version)
base_model.trainable = True
model.fit(x_train, y_train, epochs=1, batch_size=128, verbose=1)

# g. Evaluate
test_loss, test_acc = model.evaluate(x_test, y_test, verbose=0)
print(f"\n✅ Test Accuracy: {test_acc:.4f}")
```

```
# h. Prediction and visualization
predicted_value = model.predict(x_test)
n = 100
plt.imshow(x_test[n])
plt.title(f"Predicted: {labels[np.argmax(predicted_value[n])]} | Actual: {labels[np.argmax(y_test[n])]}")
plt.axis('off')
plt.show()
```

—---

ASS6(accMaam)

```
import tensorflow as tf
from tensorflow import keras
from tensorflow.keras.applications import VGG16
from tensorflow.keras.models import Model
from tensorflow.keras.layers import Dense, Flatten
from tensorflow.keras.optimizers import Adam
from tensorflow.keras.preprocessing.image import ImageDataGenerator
import numpy as np

train_dir = "C:\\Users\\hp\\Downloads\\cifar-10-img\\cifar-10-img\\train"
test_dir = "C:\\Users\\hp\\Downloads\\cifar-10-img\\cifar-10-img\\test"

train_datagen = ImageDataGenerator(
    rescale=1.0 / 255,
)

test_datagen = ImageDataGenerator(
    rescale=1.0 / 255,
)

train_batch_size = 5000
train_generator = train_datagen.flow_from_directory(
    train_dir,
    target_size=(32, 32),
    batch_size=train_batch_size,
    class_mode='categorical'
)

test_batch_size = 1000
test_generator = test_datagen.flow_from_directory(
    test_dir,
    target_size=(32, 32),
    batch_size=test_batch_size,
    class_mode='categorical'
)
x_train, y_train =  train_generator[0]
x_test, y_test = test_generator[0]

print(len(x_train))
print(len(x_test))
weights_path = "C:\\Users\\hp\\Downloads\\vgg16_weights_tf_dim_ordering_tf_kernels_notop.h5"
base_model = VGG16(weights=weights_path, include_top=False, input_shape=(32, 32, 3))
for layer in base_model.layers:
   layer.trainable = False
x = Flatten()(base_model.output)
x = Dense(256, activation='relu')(x)
x = tf.keras.layers.Dropout(0.3)(x)
x = Dense(256, activation='relu')(x)
x = tf.keras.layers.Dropout(0.3)(x)
predictions = Dense(10, activation='softmax')(x)

model = Model(inputs=base_model.input, outputs=predictions)

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

model.fit(x_train, y_train, batch_size=64, epochs=10, validation_data=(x_test, y_test))

import matplotlib.pyplot as plt
predicted_value = model.predict(x_test)
labels = list(test_generator.class_indices.keys())
n = 890
plt.imshow(x_test[n])
```

```
print("Preditcted: ",labels[np.argmax(predicted_value[n])])
print("Actual: ", labels[np.argmax(y_test[n])])
```