RAJALAKSHMI ENGINEERING COLLEGE

RAJALAKSHMI NAGAR, THANDALAM - 602 105

AI23531 - DEEP LEARNING

LABORATORY LAB MANUAL

NAME: HARISH KUMAR V

REGISTER NUMBER: 2116-231501057

YEAR / BRANCH / SECTION: III YEAR / AIML / A

SEMESTER: V SEMESTER

ACADEMIC YEAR: 2025-2026

BONAFIDE CERTIFICATE

CERTIFIED THAT THIS LABORATORY RECORD REPORT FOR "**DEEP LEARNING**" IS THE BONAFIDE WORK OF "**HARISH KUMAR V** [231501057]" WHO CARRIED OUT THE PRACTICAL WORK UNDER MY SUPERVISION.

Submitted for the Practical Examination held on ______03/11/2025______

SIGNATURE

Dr RAJU K, AIML,

REC (Autonomous) Thandalam,

Chennai - 602 105

INTERNAL EXAMINER **EXTERNAL EXAMINER**

TABLE OF CONTENTS

REG NO: 231501057 **NAME:** HARISHKUMAR V

YEAR: III YEAR **BRANCH**: AIML **SEC**: A

EX 1 THREE-LAYER NEURAL NETWORK FROM SCRATCH

DATE: 29/072025

Problem Statement:

Design and implement a three-layer neural network from scratch using Python. Train the network using the backpropagation algorithm with appropriate activation and loss functions. Apply the model to recognize handwritten digits using the MNIST dataset.

Objectives:

1. Understand the structure and working of a basic feedforward neural network.
2. Implement forward propagation and backpropagation from scratch.
3. Use the sigmoid activation function and cross-entropy loss.
4. Train the model on the MNIST dataset and evaluate the loss over epochs.
5. Visualize training performance and perform predictions on sample images.

Scope:

This experiment demonstrates the core principles behind neural networks and training them using backpropagation. It is foundational for understanding deep learning and how more complex models (like CNNs or Transformers) build upon this architecture.

Tools and Libraries Used:

1. Python 3.x
2. NumPy
3. Matplotlib
4. TensorFlow (for dataset loading)
5. scikit-learn (OneHotEncoder)

Implementation Steps:

Step 1: Import Necessary Libraries

```
import numpy as np
import matplotlib.pyplot as plt
from tensorflow.keras.datasets import mnist
from sklearn.preprocessing import OneHotEncoder
```

Step 2: Load and Preprocess the Dataset

```
(X_train, y_train), (_, _) = mnist.load_data()
X_train = X_train.reshape(-1, 784) / 255.0
encoder = OneHotEncoder(sparse_output=False)
y_train = encoder.fit_transform(y_train.reshape(-1, 1))
```

Step 3: Initialize the Network

```
input_size, hidden_size, output_size = 784, 64, 10
W1 = np.random.randn(input_size, hidden_size) * 0.01
b1 = np.zeros((1, hidden_size))
W2 = np.random.randn(hidden_size, output_size) * 0.01
b2 = np.zeros((1, output_size))
```

Step 4: Define Activation and Loss Functions

```
sigmoid = lambda x: 1 / (1 + np.exp(-x))
sigmoid_deriv = lambda x: x * (1 - x)
loss_fn = lambda y, y_hat: -np.mean(y * np.log(y_hat + 1e-8))
```

Step 5: Train the Model

```
epochs, lr = 10, 0.1
losses = []
for epoch in range(epochs):
  total_loss = 0
  for i in range(X_train.shape[0]):
    x = X_train[i:i+1]
    y = y_train[i:i+1]

    z1 = x @ W1 + b1
    a1 = sigmoid(z1)
    z2 = a1 @ W2 + b2
    a2 = sigmoid(z2)
    loss = loss_fn(y, a2)
    total_loss += loss

    dz2 = a2 - y
    dW2 = a1.T @ dz2
    db2 = dz2

    dz1 = (dz2 @ W2.T) * sigmoid_deriv(a1)
    dW1 = x.T @ dz1
    db1 = dz1

    W2 -= lr * dW2
    b2 -= lr * db2
    W1 -= lr * dW1
    b1 -= lr * db1
  losses.append(total_loss / X_train.shape[0])
  print(f"Epoch {epoch+1}, Loss: {losses[-1]:.4f}")
```

Step 6: Visualize Training Loss

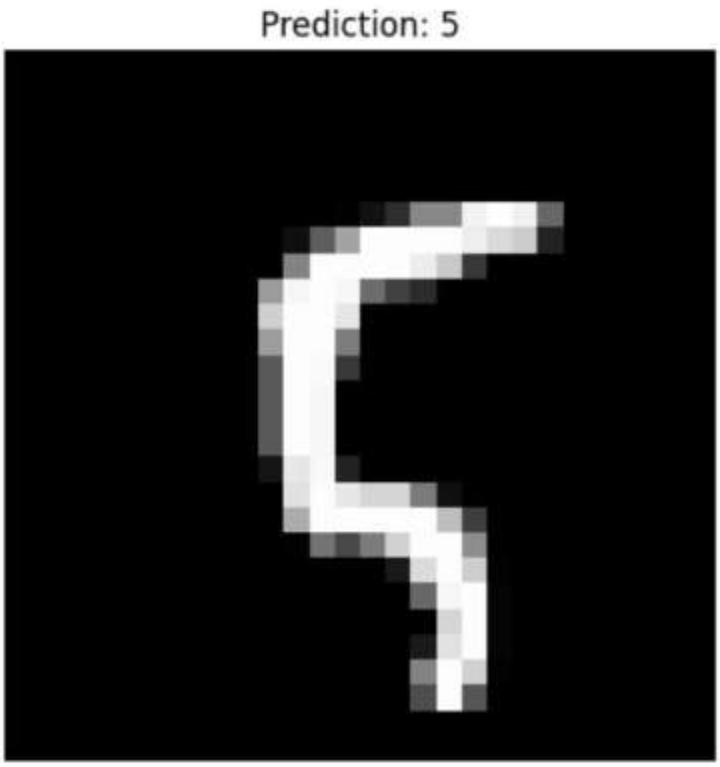
```
plt.plot(losses)
```

```
plt.title("Training Loss")
plt.xlabel("Epoch")
plt.ylabel("Loss")
plt.grid(True)
plt.show()
```

Step 7: Predict a Sample Digit

```
def predict(img):
    img = img.reshape(1, 784) / 255.0
    a1 = sigmoid(img @ W1 + b1)
    a2 = sigmoid(a1 @ W2 + b2)
    return np.argmax(a2)

idx = 100
plt.imshow(X_train[idx].reshape(28, 28), cmap='gray')
plt.title(f"Prediction: {predict(X_train[idx])}")
plt.axis('off')
plt.show()
```

Output:



EX 2 MULTI-LAYER PERCEPTRON (MLP) FOR CLASSIFICATION

DATE: 31/07/2025

Problem Statement:

Develop a Multi-Layer Perceptron (MLP) for a simple classification task. Experiment with different numbers of hidden layers and activation functions, and evaluate the model's performance using accuracy and loss.

Suggested Dataset: Iris Dataset

Objectives:

1. Understand the structure and purpose of MLPs for classification.
2. Experiment with various hidden layer configurations and activation functions.
3. Train the MLP using the Iris dataset and evaluate its accuracy and loss.
4. Visualize training progress and use the trained model for prediction.

Scope:

This experiment provides insights into how neural networks with multiple layers (MLPs) perform on structured classification tasks. It demonstrates model tuning using different architectures and activation functions, an essential concept in designing effective deep learning models.

Tools and Libraries Used:

1. Python 3.x
2. TensorFlow / Keras
3. scikit-learn (for data preprocessing and dataset loading)
4. Matplotlib (for visualization)

Implementation Steps:

Step 1: Import Necessary Libraries

```
import matplotlib.pyplot as plt
import numpy as np
from sklearn.datasets import load_iris
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler, OneHotEncoder
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense
```

Step 2: Load and Preprocess Data

```
iris = load_iris()
X = iris.data
y = iris.target.reshape(-1, 1)
```

```
classes = iris.target_names

encoder = OneHotEncoder(sparse_output=False)
y_encoded = encoder.fit_transform(y)

scaler = StandardScaler()
X_scaled = scaler.fit_transform(X)

X_train, X_test, y_train, y_test = train_test_split(X_scaled, y_encoded, test_size=0.2,
random_state=42)
```

Step 3: Define MLP Model Creation Function

```
def create_mlp(input_dim, output_dim, hidden_layers, activation='relu'):
    model = Sequential()
    model.add(Dense(hidden_layers[0], input_dim=input_dim, activation=activation))
    for units in hidden_layers[1:]:
        model.add(Dense(units, activation=activation))
    model.add(Dense(output_dim, activation='softmax'))
    model.compile(optimizer='adam', loss='categorical_crossentropy', metrics=['accuracy'])
    return model
```

Step 4: Train and Evaluate MLP with Various Configurations

```
hidden_layer_configs = [[8], [16, 8], [32, 16, 8]]
activations = ['relu', 'tanh', 'sigmoid']

for hidden_layers in hidden_layer_configs:
    for activation in activations:
        print(f"\nTesting MLP with hidden_layers={hidden_layers}, activation={activation}")
        model = create_mlp(input_dim=4, output_dim=3, hidden_layers=hidden_layers,
activation=activation)
        history = model.fit(X_train, y_train, epochs=50, batch_size=5, verbose=0,
validation_split=0.1)
        test_loss, test_acc = model.evaluate(X_test, y_test, verbose=0)
        print(f"Test Accuracy: {test_acc:.4f}, Test Loss: {test_loss:.4f}")
```

Step 5: Plot Accuracy and Loss Curves

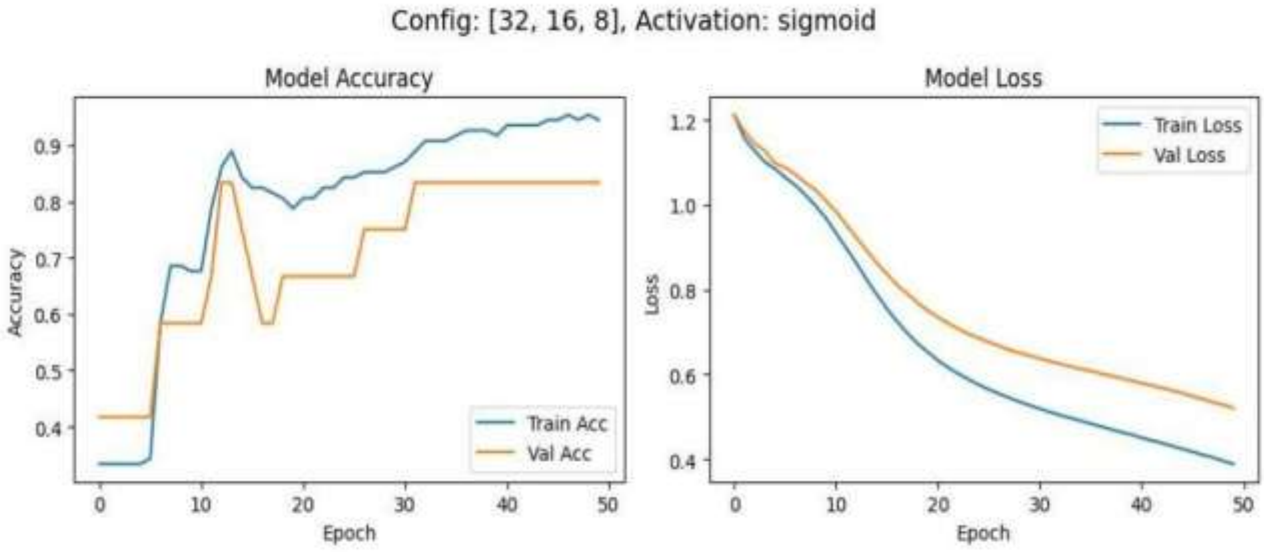
```
        # Plot accuracy and loss
        plt.figure(figsize=(10, 4))
        plt.suptitle(f"Config: {hidden_layers}, Activation: {activation}", fontsize=14)

        plt.subplot(1, 2, 1)
        plt.plot(history.history['accuracy'], label='Train Acc')
        plt.plot(history.history['val_accuracy'], label='Val Acc')
        plt.xlabel('Epoch')
        plt.ylabel('Accuracy')
```

```
plt.title('Model Accuracy')
plt.legend()

plt.subplot(1, 2, 2)
plt.plot(history.history['loss'], label='Train Loss')
plt.plot(history.history['val_loss'], label='Val Loss')
plt.xlabel('Epoch')
plt.ylabel('Loss')
plt.title('Model Loss')
plt.legend()

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

Step 6: Predict on New Input

```
sepal_length = float(input("Sepal length (cm): "))
sepal_width = float(input("Sepal width (cm): "))
petal_length = float(input("Petal length (cm): "))
petal_width = float(input("Petal width (cm): "))

user_input = np.array([[sepal_length, sepal_width, petal_length, petal_width]])
user_input_scaled = scaler.transform(user_input)

# Predict using the last trained model
prediction = model.predict(user_input_scaled)
predicted_class_index = np.argmax(prediction)
predicted_class_name = classes[predicted_class_index]

print(f"\n★ Predicted Iris Species: {predicted_class_name}")
```

Output:



EX 3 COMPARISON OF SGD WITH MOMENTUM VS ADAM OPTIMIZER

DATE: 28/08/2025

PROBLEM STATEMENT

Implement a training algorithm using Stochastic Gradient Descent (SGD) with momentum and compare it with the Adam optimizer. Train both models on a dataset and compare their convergence rates and performance.

Suggested Dataset: CIFAR-10

Objectives:

1. Understand the principles of optimization algorithms in deep learning.
2. Implement and train models using SGD with momentum and Adam.
3. Analyze and compare the learning behavior and convergence patterns of the two optimizers.
4. Visualize loss and accuracy across epochs for both optimization methods.

Scope:

This experiment gives students a comparative understanding of two widely used optimization strategies: SGD with momentum and Adam. Using a basic MLP and the CIFAR-10 dataset, students will learn the impact of optimizer choice on model convergence and final accuracy.

Tools and Libraries Used:

1. Python 3.x
2. PyTorch
3. Matplotlib
4. torchvision (for CIFAR-10 dataset)

Implementation Steps:

Step 1: Import Necessary Libraries

```
import torch
import torch.nn as nn
import torch.optim as optim
import torchvision
import torchvision.transforms as transforms
import matplotlib.pyplot as plt
```

Step 2: Set Device and Load Dataset

```
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")

transform = transforms.Compose([
    transforms.ToTensor(),
    transforms.Normalize((0.5,), (0.5,))
```

```
])
trainset = torchvision.datasets.CIFAR10(root='./data', train=True, transform=transform,
download=True)
trainloader = torch.utils.data.DataLoader(trainset, batch_size=128, shuffle=True)
```

Step 3: Define MLP Model

```
class MLP(nn.Module):
    def __init__(self):
        super().__init__()
        self.flatten = nn.Flatten()
        self.net = nn.Sequential(
            nn.Linear(3*32*32, 256),
            nn.ReLU(),
            nn.Linear(256, 10)
        )

    def forward(self, x):
        x = self.flatten(x)
        return self.net(x)
```

Step 4: Define Training Function

```
def train(model, optimizer, epochs=10):
    model.to(device)
    loss_fn = nn.CrossEntropyLoss()
    losses = []
    accuracies = []

    for epoch in range(epochs):
        total_loss = 0
        correct = 0
        total = 0
        model.train()

        for imgs, labels in trainloader:
            imgs, labels = imgs.to(device), labels.to(device)

            outputs = model(imgs)
            loss = loss_fn(outputs, labels)

            optimizer.zero_grad()
            loss.backward()
            optimizer.step()

            total_loss += loss.item()
            _, predicted = outputs.max(1)
            total += labels.size(0)
            correct += (predicted == labels).sum().item()
```

```
        avg_loss = total_loss / len(trainloader)
        accuracy = 100.0 * correct / total

        losses.append(avg_loss)
        accuracies.append(accuracy)

        print(f"Epoch {epoch+1}: Loss = {avg_loss:.4f}, Accuracy = {accuracy:.2f}%")

    return losses, accuracies
```

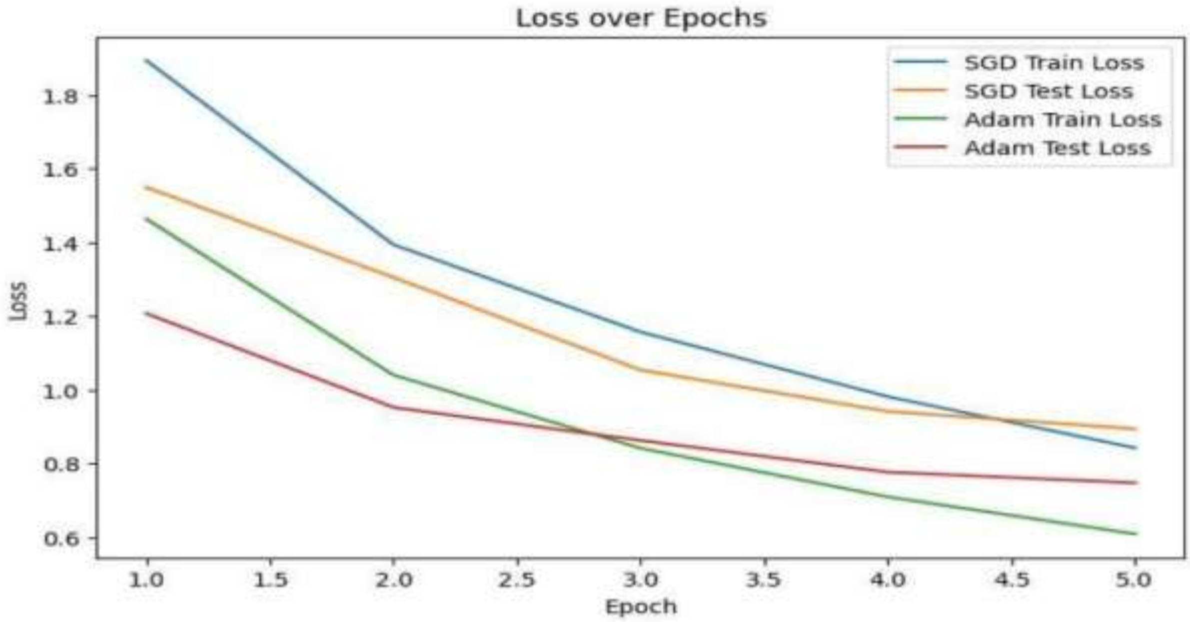
Step 5: Train with SGD + Momentum and with Adam

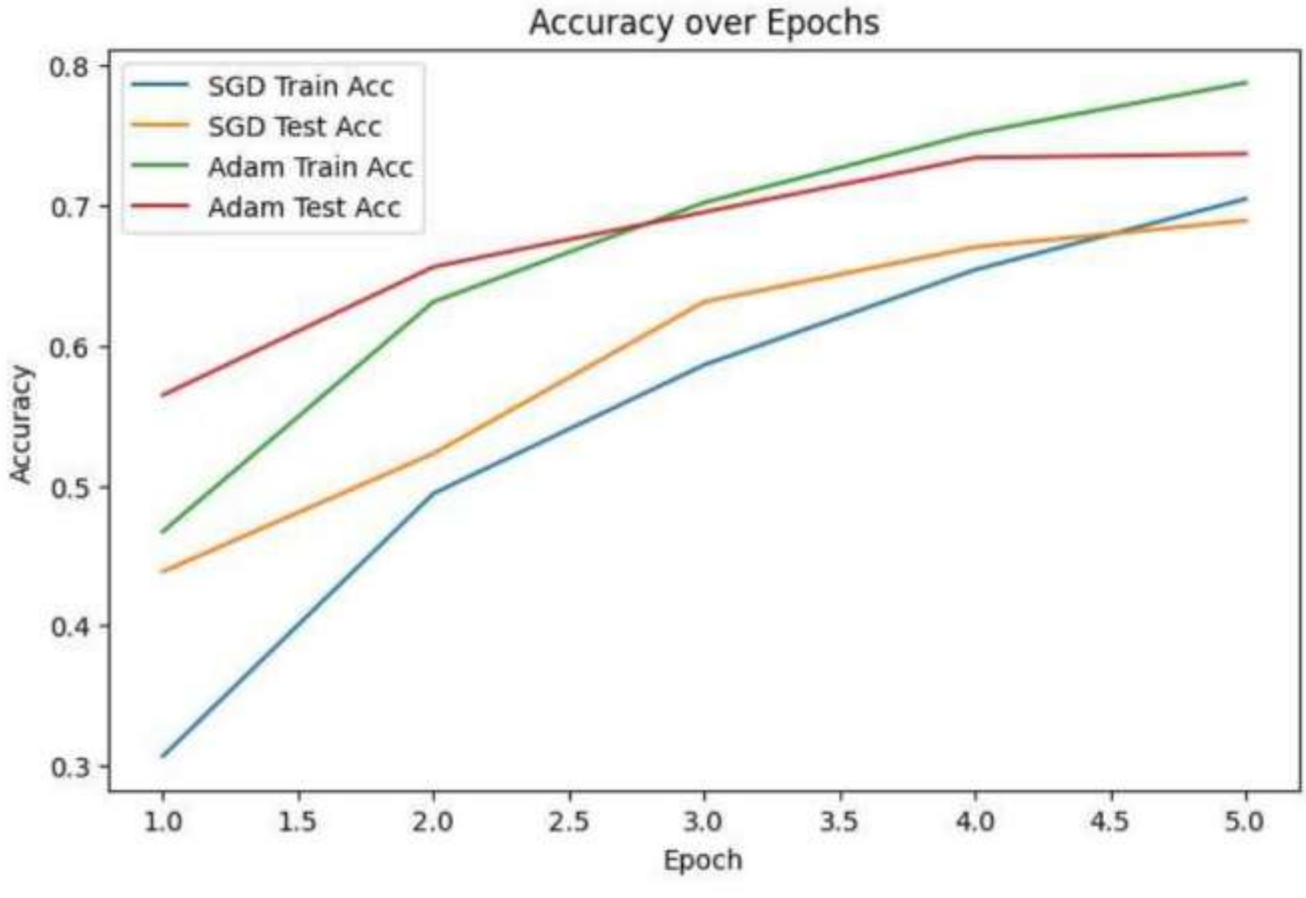
```
model_sgd = MLP()
sgd = optim.SGD(model_sgd.parameters(), lr=0.01, momentum=0.9)
losses_sgd, acc_sgd = train(model_sgd, sgd)
model_adam = MLP()
adam = optim.Adam(model_adam.parameters(), lr=0.001)
losses_adam, acc_adam = train(model_adam, adam)
```

Step 6: Visualize Loss and Accuracy Comparison

```
plt.figure(figsize=(12, 5))

plt.subplot(1, 2, 1)
plt.plot(losses_sgd, label="SGD + Momentum")
plt.plot(losses_adam, label="Adam")
plt.xlabel("Epoch")
plt.ylabel("Loss")
plt.title("Loss Comparison on CIFAR-10 (MLP)")
plt.legend()

plt.subplot(1, 2, 2)
plt.plot(acc_sgd, label="SGD + Momentum")
plt.plot(acc_adam, label="Adam")
plt.xlabel("Epoch")
plt.ylabel("Accuracy (%)")
plt.title("Accuracy Comparison on CIFAR-10 (MLP)")
plt.legend()
plt.tight_layout()
plt.show()
```

Output:


Loss over Epochs
SGD Train Loss
SGD Test Loss
Adam Train Loss
Adam Test Loss
Loss
Epoch
1.8
1.6
1.4
1.2
1.0
0.8
0.6
1.0
1.5
2.0
2.5
3.0
3.5
4.0
4.5
5.0


Accuracy over Epochs
SGD Train Acc
SGD Test Acc
Adam Train Acc
Adam Test Acc
Accuracy
Epoch
0.8
0.7
0.6
0.5
0.4
0.3
1.0
1.5
2.0
2.5
3.0
3.5
4.0
4.5
5.0

EX 4 IMPLEMENTATION OF A CONVOLUTIONAL NEURAL NETWORK (CNN)

DATE: 11/09/2025

Problem Statement:

Implement a Convolutional Neural Network (CNN) from scratch using PyTorch to classify images. Train the network using a dataset of labeled images and evaluate its performance. Additionally, visualize the learned filters in the convolution layers.

Suggested Dataset: CIFAR-10

Objectives:

1. Understand the architecture and functionality of Convolutional Neural Networks (CNNs).
2. Implement CNN layers including convolution, pooling, and fully connected layers.
3. Train the model on the CIFAR-10 dataset and evaluate its performance.
4. Visualize the learned filters to interpret feature extraction at early layers.

Scope:

CNNs are powerful architectures for image classification tasks. This experiment helps students grasp key CNN concepts such as spatial feature learning, hierarchical representation, and how filters learn patterns in images. Visualizing filters bridges the gap between model architecture and interpretability.

Tools and Libraries Used:

1. Python 3.x
2. PyTorch
3. torchvision
4. Matplotlib

Implementation Steps:

Step 1: Load and Preprocess CIFAR-10 Dataset

```
import torchvision.transforms as transforms
import torchvision

transform = transforms.Compose([
    transforms.ToTensor(),
    transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5))
])

trainset = torchvision.datasets.CIFAR10(root='./data', train=True, download=True,
transform=transform)
```

```
trainloader = torch.utils.data.DataLoader(trainset, batch_size=64, shuffle=True)

testset = torchvision.datasets.CIFAR10(root='./data', train=False, download=True,
transform=transform)
testloader = torch.utils.data.DataLoader(testset, batch_size=64, shuffle=False)

classes = trainset.classes
```

Step 2: Define CNN Architecture

```
import torch.nn as nn

class SimpleCNN(nn.Module):
    def __init__(self):
        super(SimpleCNN, self).__init__()
        self.conv1 = nn.Conv2d(3, 16, 3, padding=1)  # Output: 16x32x32
        self.pool = nn.MaxPool2d(2, 2)               # Output: 16x16x16
        self.conv2 = nn.Conv2d(16, 32, 3, padding=1)  # Output: 32x16x16 → 32x8x8 after
pooling
        self.fc1 = nn.Linear(32 * 8 * 8, 128)
        self.fc2 = nn.Linear(128, 10)
        self.relu = nn.ReLU()

    def forward(self, x):
        x = self.pool(self.relu(self.conv1(x)))
        x = self.pool(self.relu(self.conv2(x)))
        x = x.view(-1, 32 * 8 * 8)
        x = self.relu(self.fc1(x))
        x = self.fc2(x)
        return x
```

Step 3: Train the CNN

```
import torch
import torch.optim as optim

device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
model = SimpleCNN().to(device)

lossfn = nn.CrossEntropyLoss()
optimizer = optim.Adam(model.parameters(), lr=0.001)

for epoch in range(10):
    running_loss = 0.0
    for inputs, labels in trainloader:
        inputs, labels = inputs.to(device), labels.to(device)

        optimizer.zero_grad()
        outputs = model(inputs)
        loss = lossfn(outputs, labels)
```

```
        loss.backward()
        optimizer.step()

        running_loss += loss.item()
    print(f"Epoch {epoch+1}, Loss: {running_loss / len(trainloader):.4f}")
```

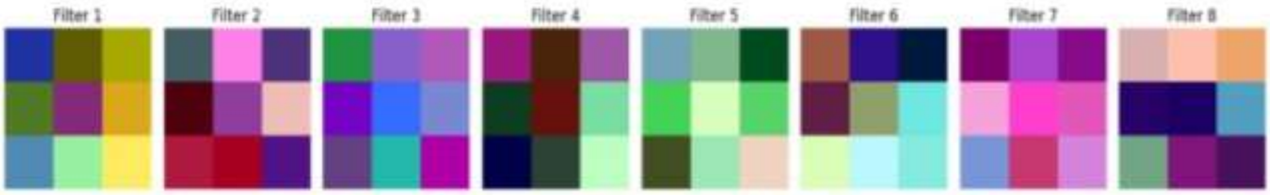
Step 4: Evaluate Model Performance

```
correct, total = 0, 0
with torch.no_grad():
    for images, labels in testloader:
        images, labels = images.to(device), labels.to(device)
        outputs = model(images)
        _, predicted = outputs.max(1)
        total += labels.size(0)
        correct += (predicted == labels).sum().item()

print(f"Accuracy on test data: {100 * correct / total:.2f}%")
```

Step 5: Visualize Learned Filters

```
import matplotlib.pyplot as plt

def visualize_filters(layer, n_filters=8):
    filters = layer.weight.data.clone().cpu()
    fig, axs = plt.subplots(1, n_filters, figsize=(15, 4))
    for i in range(n_filters):
        f = filters[i]
        f = (f - f.min()) / (f.max() - f.min())  # Normalize for display
        axs[i].imshow(f.permute(1, 2, 0))
        axs[i].axis('off')
        axs[i].set_title(f'Filter {i+1}')
    plt.tight_layout()
    plt.show()

visualize_filters(model.conv1)
```

Output:



EX 5 COMPARATIVE ANALYSIS OF VGG, RESNET, AND GOOGLENET ON IMAGE CLASSIFICATION

DATE: 18/09/2025

Problem Statement:

Implement and compare the performance of three popular CNN architectures: VGG, ResNet, and GoogLeNet for image classification. Use a labeled dataset to train each model and evaluate their convergence and accuracy.

Suggested Dataset: Dogs vs. Cats dataset

Objectives:

1. Understand the architectural differences between VGG, ResNet, and GoogLeNet.
2. Train all three models on the same dataset using transfer learning.
3. Analyze and compare performance using validation accuracy.
4. Apply the trained models to predict classes for custom images.

Scope:

This experiment demonstrates the power of transfer learning using pre-trained CNN models. Students explore how architectural changes affect accuracy and generalization. Comparing models under identical training settings aids in choosing the right model for real-world applications.

Tools and Libraries Used:

1. Python 3.x
2. PyTorch
3. torchvision
4. Matplotlib
5. PIL (Python Imaging Library)

Implementation Steps:

Step 1: Import Necessary Libraries

```
import torch
import torch.nn as nn
import torch.optim as optim
from torchvision import datasets, models, transforms
from torch.utils.data import DataLoader, random_split, Subset
import matplotlib.pyplot as plt
from PIL import Image
import os
```

Step 2: Configure Device and Define Labels

```
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
print("Using device:", device)

class_names = ['airplane', 'automobile', 'bird', 'cat', 'deer',
           'dog', 'frog', 'horse', 'ship', 'truck']
```

Step 3: Preprocess Data and Load CIFAR-10

```
transform = transforms.Compose([
  transforms.Resize((224, 224)),
  transforms.ToTensor(),
  transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5))
])

dataset = datasets.CIFAR10(root='./data', train=True, download=True,
transform=transform)
test_dataset = datasets.CIFAR10(root='./data', train=False, download=True,
transform=transform)

dataset = Subset(dataset, range(500))
train_size = int(0.8 * len(dataset))
val_size = len(dataset) - train_size
train_dataset, val_dataset = random_split(dataset, [train_size, val_size])

train_loader = DataLoader(train_dataset, batch_size=64, shuffle=True)
val_loader = DataLoader(val_dataset, batch_size=64, shuffle=False)
```

Step 4: Define Training and Evaluation Function

```
def train_and_evaluate(model, name, num_epochs=5):
  model = model.to(device)
  criterion = nn.CrossEntropyLoss()
  optimizer = optim.Adam(model.parameters(), lr=1e-4)

  train_accs, val_accs = [], []

  for epoch in range(num_epochs):
    model.train()
    correct, total = 0, 0
    for inputs, labels in train_loader:
      inputs, labels = inputs.to(device), labels.to(device)

      optimizer.zero_grad()
      outputs = model(inputs)
      loss = criterion(outputs, labels)
      loss.backward()
      optimizer.step()
```

```
            _, predicted = torch.max(outputs, 1)
            correct += (predicted == labels).sum().item()
            total += labels.size(0)
        train_accs.append(100 * correct / total)

        model.eval()
        correct, total = 0, 0
        with torch.no_grad():
            for inputs, labels in val_loader:
                inputs, labels = inputs.to(device), labels.to(device)
                outputs = model(inputs)
                _, predicted = outputs.max(1)
                correct += (predicted == labels).sum().item()
                total += labels.size(0)
        val_accs.append(100 * correct / total)

        print(f"{name} Epoch {epoch+1}/{num_epochs} - Train Acc: {train_accs[-1]:.2f}%, Val
Acc: {val_accs[-1]:.2f}%")

    return model, train_accs, val_accs
```

Step 5: Select Pretrained Models and Replace Final Layers

```
def get_model(name):
    if name == "vgg":
        model = models.vgg16(pretrained=True)
        model.classifier[6] = nn.Linear(4096, 10)
    elif name == "resnet":
        model = models.resnet18(pretrained=True)
        model.fc = nn.Linear(model.fc.in_features, 10)
    elif name == "googlenet":
        model = models.googlenet(pretrained=True, aux_logits=True)
        model.fc = nn.Linear(model.fc.in_features, 10)
    else:
        raise ValueError("Unknown model")
    return model
```

Step 6: Train All Models and Collect Results

```
results = {}
trained_models = {}

for model_name in ["vgg", "resnet", "googlenet"]:
    print(f"\n◆ Training {model_name.upper()} on CIFAR-10...")
    model = get_model(model_name)
    trained_model, train_acc, val_acc = train_and_evaluate(model, model_name.upper(),
num_epochs=5)
    results[model_name] = (train_acc, val_acc)
    trained_models[model_name] = trained_model
```

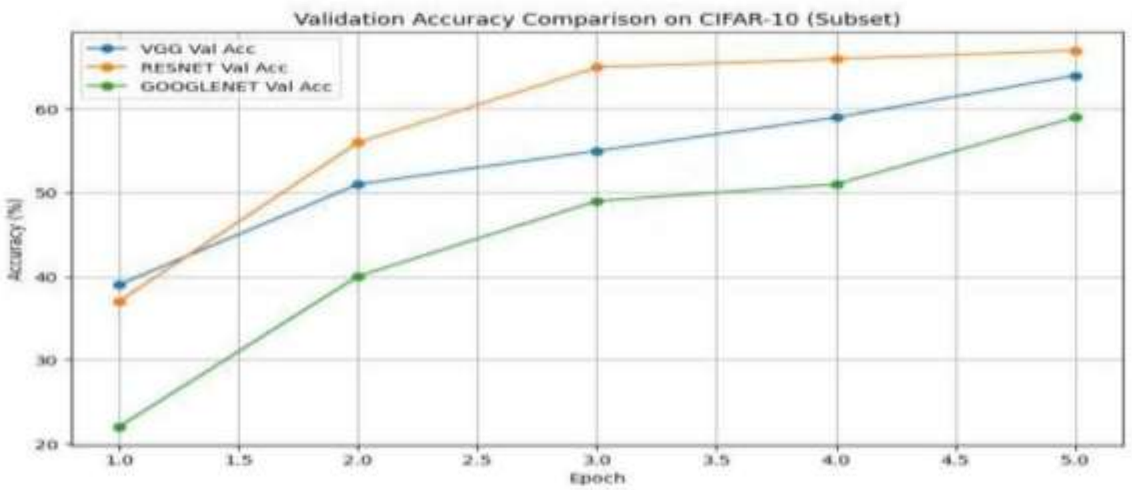
Step 7: Plot Accuracy Comparison

```
plt.figure(figsize=(10, 6))
for name, (train_acc, val_acc) in results.items():
    plt.plot(val_acc, label=f'{name.upper()} Val Acc')
plt.title('Validation Accuracy Comparison on CIFAR-10')
plt.xlabel('Epoch')
plt.ylabel('Accuracy (%)')
plt.legend()
plt.grid(True)
plt.show()
```

Step 8: Predict on Custom Image

```
def predict_image(image_path, models_dict):
    image = Image.open(image_path).convert('RGB')
    image = transform(image).unsqueeze(0).to(device)

    print(f"\n🖼 Prediction results for image: {image_path}")
    for model_name, model in models_dict.items():
        model.eval()
        with torch.no_grad():
            outputs = model(image)
            _, predicted = outputs.max(1)
            pred_class = class_names[predicted.item()]
            print(f"{model_name.upper():<10} => {pred_class}")

custom_image_path = "download.jpeg"
if os.path.exists(custom_image_path):
    predict_image(custom_image_path, trained_models)
else:
    print(f"\n! Image not found: {custom_image_path}. Please add an image to test.")
```

Output:



EX 6 **BIDIRECTIONAL RNN VS FEEDFORWARD NN FOR TIME-SERIES PREDICTION**

DATE: 25/09/2025

Problem Statement:

Implement a Bidirectional Recurrent Neural Network (BiRNN) to predict sequences in time-series data. Train the model and compare its performance with a traditional Feedforward Neural Network (FFNN) for sequence-based tasks.

Suggested Dataset: Airline Passenger Dataset

Objectives:

1. Understand the application of RNNs and FFNNs for time-series forecasting.
2. Train a bidirectional RNN model to capture sequential dependencies.
3. Compare prediction performance with a feedforward neural network.
4. Visualize and evaluate predictions using metrics such as Mean Squared Error (MSE).

Scope:

Recurrent Neural Networks are well-suited for tasks involving sequential data. This experiment demonstrates the power of BiRNNs in modeling time dependencies and compares them with simpler feedforward architectures, providing insight into the role of model memory in sequence modeling.

Tools and Libraries Used:

1. Python 3.x
2. pandas
3. numpy
4. matplotlib
5. scikit-learn
6. PyTorch

Implementation Steps:

Step 1: Import Necessary Libraries

```
import pandas as pd
import numpy as np
from sklearn.preprocessing import MinMaxScaler
```

Step 2: Data Preparation

```
url='https://raw.githubusercontent.com/jbrownlee/Datasets/refs/heads/master/monthly
-airline-passengers.csv'
```

```
df = pd.read_csv(url, usecols=[1])
data = df.values.astype('float32')

scaler = MinMaxScaler(feature_range=(0, 1))
data_scaled = scaler.fit_transform(data)
```

Step 3: Create Sequences for Time-Series Prediction

```
SEQ_LENGTH = 10
X = np.array([data_scaled[i:i+SEQ_LENGTH] for i in range(len(data_scaled) - SEQ_LENGTH)])
y = np.array([data_scaled[i + SEQ_LENGTH] for i in range(len(data_scaled) - SEQ_LENGTH)])

train_size = int(len(X) * 0.8)
X_train, X_test = X[:train_size], X[train_size:]
y_train, y_test = y[:train_size], y[train_size:]
```

Step 4: Prepare PyTorch Datasets and Loaders

```
import torch
from torch.utils.data import TensorDataset, DataLoader

X_train_tensor = torch.tensor(X_train)
y_train_tensor = torch.tensor(y_train)
X_test_tensor  = torch.tensor(X_test)
y_test_tensor  = torch.tensor(y_test)

train_loader = DataLoader(TensorDataset(X_train_tensor, y_train_tensor), batch_size=16, shuffle=True)
test_loader  = DataLoader(TensorDataset(X_test_tensor, y_test_tensor), batch_size=1)
```

Step 5: Define BiRNN and Feedforward Models

```
import torch.nn as nn

class BiRNN(nn.Module):
  def __init__(self, input_size=1, hidden_size=64, num_layers=1):
    super(BiRNN, self).__init__()
    self.rnn = nn.RNN(input_size, hidden_size, num_layers, batch_first=True, bidirectional=True)
    self.fc = nn.Linear(hidden_size * 2, 1)

  def forward(self, x):
    out, _ = self.rnn(x)
    return self.fc(out[:, -1, :])

class FeedforwardNN(nn.Module):
  def __init__(self, input_size):
    super(FeedforwardNN, self).__init__()
```

```
        self.fc1 = nn.Linear(input_size, 64)
        self.relu = nn.ReLU()
        self.fc2 = nn.Linear(64, 1)

    def forward(self, x):
        x = x.view(x.size(0), -1)
        return self.fc2(self.relu(self.fc1(x)))
```

Step 6: Define Training and Evaluation Functions

```
def train_model(model, loader, epochs=100):
    criterion = nn.MSELoss()
    optimizer = torch.optim.Adam(model.parameters(), lr=0.01)
    model.train()
    for epoch in range(epochs):
        loss_epoch = 0
        for seqs, targets in loader:
            optimizer.zero_grad()
            outputs = model(seqs)
            loss = criterion(outputs, targets)
            loss.backward()
            optimizer.step()
            loss_epoch += loss.item()
        if (epoch + 1) % 20 == 0:
            print(f"Epoch {epoch+1}/{epochs}, Loss: {loss_epoch / len(loader):.5f}")

def evaluate_model(model, X):
    model.eval()
    with torch.no_grad():
        return model(X).numpy()
```

Step 7: Train and Predict with Both Models

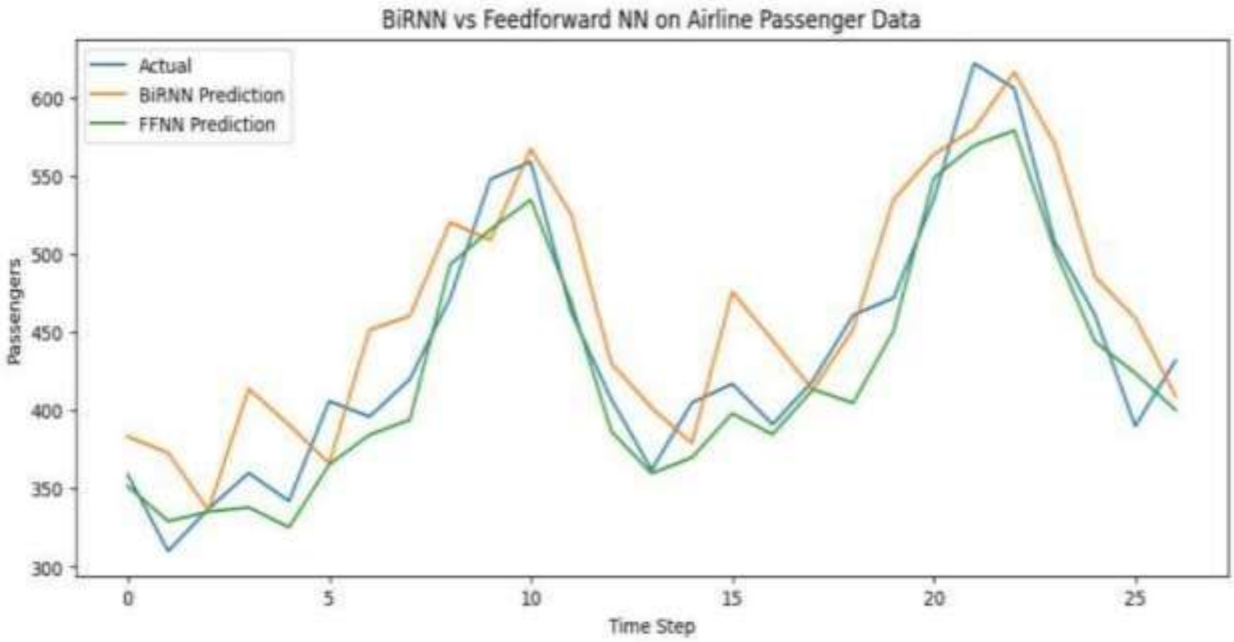
```
birnn = BiRNN()
train_model(birnn, train_loader)

ffnn = FeedforwardNN(input_size=SEQ_LENGTH)
train_model(ffnn, train_loader)

pred_birnn = evaluate_model(birnn, X_test_tensor)
pred_ffnn = evaluate_model(ffnn, X_test_tensor)
```

Step 8: Inverse Transform and Plot Results

```
from sklearn.metrics import mean_squared_error
import matplotlib.pyplot as plt

pred_birnn_inv = scaler.inverse_transform(pred_birnn)
pred_ffnn_inv = scaler.inverse_transform(pred_ffnn)
y_test_inv = scaler.inverse_transform(y_test)
```

```
plt.figure(figsize=(12, 5))
plt.plot(y_test_inv, label='Actual')
plt.plot(pred_birnn_inv, label='BiRNN Prediction')
plt.plot(pred_ffnn_inv, label='FFNN Prediction')
plt.legend()
plt.title('BiRNN vs Feedforward NN on Airline Passenger Data')
plt.xlabel('Time Step')
plt.ylabel('Passengers')
plt.show()

mse_birnn = mean_squared_error(y_test_inv, pred_birnn_inv)
mse_ffnn = mean_squared_error(y_test_inv, pred_ffnn_inv)

print(f"BiRNN MSE: {mse_birnn:.3f}")
print(f"FFNN MSE: {mse_ffnn:.3f}")
```

Output:



EX 7 IMAGE CAPTIONING USING CNN-RNN ARCHITECTURE

DATE: 09/10/2025

PROBLEM STATEMENT

Build a deep Recurrent Neural Network (RNN) to generate captions for images. Combine a Convolutional Neural Network (CNN) for feature extraction with an RNN for sequence generation.

Objectives:

1. Understand image captioning using encoder-decoder architecture.
2. Use a pre-trained Vision Transformer (ViT) as the CNN encoder and GPT-2 as the RNN decoder.
3. Generate meaningful captions for visual content using beam search decoding.
4. Explore the power of vision-language models for real-world applications.

Scope:

This experiment demonstrates the fusion of computer vision and natural language processing through encoder-decoder architectures. Students gain insight into how visual features are mapped to textual sequences, a key component in modern AI systems such as image tagging, accessibility tech, and content generation.

Tools and Libraries Used:

1. Python 3.x
2. PyTorch
3. HuggingFace Transformers
4. VisionEncoderDecoderModel (ViT + GPT-2)
5. PIL (Python Imaging Library)

Implementation Steps:

Step 1: Import Required Libraries

```
import torch
from transformers import VisionEncoderDecoderModel, ViTImageProcessor,
AutoTokenizer
from PIL import Image
from transformers.utils import hub

# Set extended download timeout
hub.HUGGINGFACE_HUB_HTTP_TIMEOUT = 60
```

Step 2: Load Pretrained Image Captioning Model

```
model = VisionEncoderDecoderModel.from_pretrained("nlpconnect/vit-gpt2-image-captioning")
processor = ViTImageProcessor.from_pretrained("nlpconnect/vit-gpt2-image-captioning")
tokenizer  =  AutoTokenizer.from_pretrained("nlpconnect/vit-gpt2-image-captioning")

device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
model.to(device)
model.eval()
```

Step 3: Define Caption Generation Function

```
def generate_caption(image_path, max_length=30, num_beams=4):
    image = Image.open(image_path).convert("RGB")
    pixel_values = processor(images=image, return_tensors="pt").pixel_values.to(device)

    output_ids = model.generate(pixel_values,
                                max_length=max_length,
                                num_beams=num_beams,
                                early_stopping=True)

    caption = tokenizer.decode(output_ids[0], skip_special_tokens=True)
    return caption
```

Step 4: Load Image and Generate Caption

```
if __name__ == "__main__":
    img_path = "download.jpeg"  # Replace with any image file
    print("Caption:", generate_caption(img_path))
```

Output:

Caption: a man standing in front of a group of men

EX 8 IMAGE GENERATION USING VARIATIONAL AUTOENCODER (VAE)

DATE: 09/10/2025

PROBLEM STATEMENT

Implement a Variational Autoencoder (VAE) to generate new images from a given dataset. Train the model to learn the latent representation of images and generate new samples from the learned distribution.

Suggested Dataset: CelebA Dataset

Objectives:

1. Understand the concept of generative models using latent space representations.
2. Implement encoder-decoder architecture with reparameterization.
3. Train a VAE on CelebA and generate new human face images.
4. Visualize generated samples from the latent space.

Scope:

VAEs are probabilistic generative models capable of learning latent representations and generating realistic samples. This experiment explores the VAE pipeline—encoding, sampling via reparameterization, and decoding—to generate new samples that resemble the training distribution.

Tools and Libraries Used:

1. Python 3.x
2. PyTorch
3. torchvision
4. matplotlib
5. CelebA Dataset

Implementation Steps:

Step 1: Import Required Libraries

```
import torch
import torch.nn as nn
import torch.nn.functional as F
import torchvision
from torchvision import datasets, transforms
from torch.utils.data import DataLoader, Subset
import matplotlib.pyplot as plt
```

Step 2: Configure Parameters and Load Dataset

```
device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')

image_size = 64
batch_size = 128
latent_dim = 100
num_epochs = 5
learning_rate = 1e-3

transform = transforms.Compose([
    transforms.CenterCrop(178),
    transforms.Resize(image_size),
    transforms.ToTensor(),
    transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5))
])

dataset = datasets.CelebA(root='data', split='train', download=True, transform=transform)
dataset = Subset(dataset, range(500))  # Limit to 500 samples for speed
dataloader = DataLoader(dataset, batch_size=batch_size, shuffle=True)
```

Step 3: Define Encoder Network

```
class Encoder(nn.Module):
    def __init__(self, latent_dim):
        super(Encoder, self).__init__()
        self.conv = nn.Sequential(
            nn.Conv2d(3, 64, 4, 2, 1), nn.ReLU(),
            nn.Conv2d(64, 128, 4, 2, 1), nn.BatchNorm2d(128), nn.ReLU(),
            nn.Conv2d(128, 256, 4, 2, 1), nn.BatchNorm2d(256), nn.ReLU(),
            nn.Conv2d(256, 512, 4, 2, 1), nn.BatchNorm2d(512), nn.ReLU()
        )
        self.fc_mu = nn.Linear(512*4*4, latent_dim)
        self.fc_logvar = nn.Linear(512*4*4, latent_dim)

    def forward(self, x):
        x = self.conv(x)
        x = x.view(x.size(0), -1)
        return self.fc_mu(x), self.fc_logvar(x)
```

Step 4: Define Decoder Network

```
class Decoder(nn.Module):
    def __init__(self, latent_dim):
        super(Decoder, self).__init__()
        self.fc = nn.Linear(latent_dim, 512*4*4)
        self.deconv = nn.Sequential(
            nn.ConvTranspose2d(512, 256, 4, 2, 1), nn.BatchNorm2d(256), nn.ReLU(),
            nn.ConvTranspose2d(256, 128, 4, 2, 1), nn.BatchNorm2d(128), nn.ReLU(),
            nn.ConvTranspose2d(128, 64, 4, 2, 1), nn.BatchNorm2d(64), nn.ReLU(),
            nn.ConvTranspose2d(64, 3, 4, 2, 1), nn.Tanh()
```

```
    )

  def forward(self, z):
    x = self.fc(z)
    x = x.view(x.size(0), 512, 4, 4)
    return self.deconv(x)
```

Step 5: Define the VAE Model

```
class VAE(nn.Module):
  def __init__(self, latent_dim):
    super(VAE, self).__init__()
    self.encoder = Encoder(latent_dim)
    self.decoder = Decoder(latent_dim)

  def reparameterize(self, mu, logvar):
    std = torch.exp(0.5 * logvar)
    eps = torch.randn_like(std)
    return mu + eps * std

  def forward(self, x):
    mu, logvar = self.encoder(x)
    z = self.reparameterize(mu, logvar)
    return self.decoder(z), mu, logvar
```

Step 6: Define Loss Function

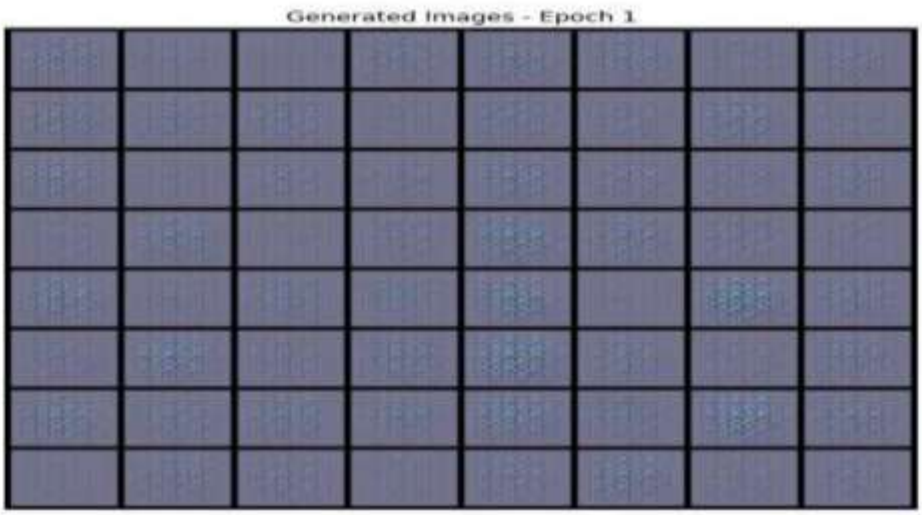
```
def vae_loss(recon_x, x, mu, logvar):
  recon_loss = F.mse_loss(recon_x, x, reduction='sum')
  kl_div = -0.5 * torch.sum(1 + logvar - mu.pow(2) - logvar.exp())
  return recon_loss + kl_div
```

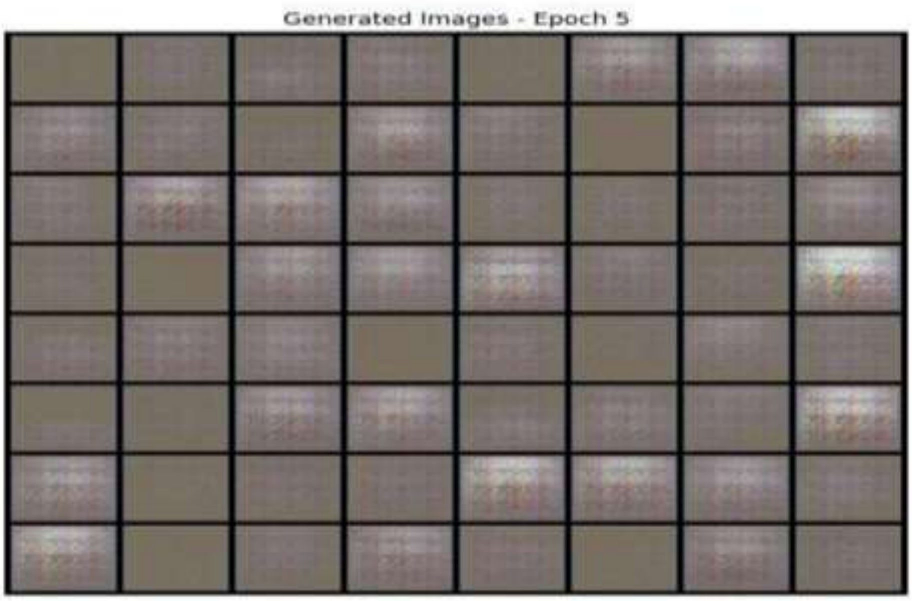
Step 7: Train the Model and Generate Images

```
model = VAE(latent_dim).to(device)
optimizer = torch.optim.Adam(model.parameters(), lr=learning_rate)

for epoch in range(num_epochs):
  model.train()
  total_loss = 0
  for images, _ in dataloader:
    images = images.to(device)
    recon, mu, logvar = model(images)
    loss = vae_loss(recon, images, mu, logvar)

    optimizer.zero_grad()
    loss.backward()
    optimizer.step()

    total_loss += loss.item()
```

```
    print(f"Epoch [{epoch+1}/{num_epochs}], Loss: {total_loss/len(dataloader.dataset):.4f}")

    # Generate and visualize samples
    model.eval()
    with torch.no_grad():
        z = torch.randn(64, latent_dim).to(device)
        sample_images = model.decoder(z).cpu() * 0.5 + 0.5  # De-normalize
        grid = torchvision.utils.make_grid(sample_images, nrow=8)
        plt.imshow(grid.permute(1, 2, 0))
        plt.axis('off')
        plt.title(f"Generated Faces - Epoch {epoch+1}")
        plt.show()
```

Output:





EX 9 **TEXT GENERATION USING LSTM NETWORKS**

DATE: 16/10/2025

PROBLEM STATEMENT

Build a text generation model using Long Short-Term Memory (LSTM) networks. Train the model on a text corpus to generate coherent sequences of text and evaluate the output for fluency and coherence.

Suggested Dataset: Shakespeare Corpus

Objectives:

1. Understand sequential modeling for natural language generation.
2. Train a character-level LSTM model to learn language patterns.
3. Generate text using a seed prompt and evaluate the results.
4. Analyze the fluency and creativity of LSTM-generated outputs.

Scope:

Text generation is a foundational task in natural language processing. This experiment demonstrates how LSTMs can learn syntactic and semantic patterns over time and generate believable sequences of text. The use of character-level modeling helps capture detailed language structures.

Tools and Libraries Used:

1. Python 3.x
2. TensorFlow / Keras
3. NumPy
4. Shakespeare Text Corpus (Tiny Shakespeare)

Implementation Steps:

Step 1: Load and Preprocess the Dataset

```
import tensorflow as tf
import numpy as np

text = tf.keras.utils.get_file('shakespeare.txt',
   'https://raw.githubusercontent.com/karpathy/char-
rnn/master/data/tinyshakespeare/input.txt')
text = open(text, 'r').read().lower()
chars = sorted(set(text))
c2i = {c: i for i, c in enumerate(chars)}
i2c = {i: c for i, c in enumerate(chars)}
```

Step 2: Create Input and Output Sequences

```
seq_len = 40
X = []
y = []

for i in range(len(text) - seq_len):
    input_seq = text[i:i + seq_len]
    target_char = text[i + seq_len]
    X.append([c2i[c] for c in input_seq])
    y.append(c2i[target_char])

X = np.array(X)
y = np.array(y)
```

Step 3: Build the LSTM Model

```
model = tf.keras.Sequential([
    tf.keras.layers.Embedding(len(chars), 64, input_length=seq_len),
    tf.keras.layers.LSTM(128),
    tf.keras.layers.Dense(len(chars), activation='softmax')
])
model.compile(loss='sparse_categorical_crossentropy', optimizer='adam')
model.fit(X, y, batch_size=128, epochs=1)
```

Step 4: Define the Text Generation Function

```
def generate(seed, length=300):
    seq = [c2i[c] for c in seed.lower()]
    for _ in range(length):
        inp = np.array(seq[-seq_len:]).reshape(1, -1)
        pred = model.predict(inp, verbose=0)[0]
        next_idx = np.random.choice(len(pred), p=pred)
        seq.append(next_idx)
    return seed + ''.join(i2c[i] for i in seq[len(seed):])
```

Step 5: Generate and Display Text

```
print("\nGenerated Text:\n")
print(generate("shall i compare thee to a summer's day?\n"))
```

Output:

Generated Text:

shall i compare thee to a summer's day?

kind worldmbly:
be the was of before spyech will of beopker, i ayf.

lucendeo:
that what would thim anst marbned unto you.

vicinent:
the fyore!

duke intimnes:
we'se sither, and immalio,
foil i for is of autenel, go but, i deas
them our lieg. ruclio?
our to younce a face and poling,
and this the h

EX 10 IMAGE GENERATION USING GENERATIVE ADVERSARIAL NETWORK (GAN)

DATE: 16/10/2025

Problem Statement:

Train a Generative Adversarial Network (GAN) using the CIFAR-10 dataset to generate new synthetic images. Evaluate the generated outputs through visual inspection to understand the training behavior and realism of generated samples.

Objectives:

- ☐ Understand the architecture of a simple Deep Convolutional GAN (DCGAN).
- ☐ Implement Generator and Discriminator networks using TensorFlow and Keras.
- ☐ Train the GAN using adversarial learning principles.
- ☐ Generate new images from random noise vectors.
- ☐ Visually evaluate the quality and diversity of generated images.

Scope:

GANs are powerful models for data generation, capable of synthesizing realistic images after learning from real samples. This experiment provides hands-on experience with adversarial training dynamics and the generator-discriminator framework.

Tools and Libraries Used:

- ☐ Python 3.x
- ☐ TensorFlow / Keras
- ☐ NumPy
- ☐ Matplotlib

Implementation Steps:

Step 1: Load and Preprocess CIFAR-10 Dataset

```
import tensorflow as tf
from tensorflow.keras import layers
import numpy as np
import matplotlib.pyplot as plt

(x_train, _), (_, _) = tf.keras.datasets.cifar10.load_data()
x_train = (x_train.astype("float32") - 127.5) / 127.5
x_train = tf.data.Dataset.from_tensor_slices(x_train).shuffle(60000).batch(128)
```

Step 2: Define the Generator Network

```
def make_generator():
    model = tf.keras.Sequential([
    layers.Dense(8*8*256, use_bias=False, input_shape=(100,)),
    layers.BatchNormalization(),
    layers.LeakyReLU(),
    layers.Reshape((8, 8, 256)),
    layers.Conv2DTranspose(128, (5,5), strides=(2,2), padding='same', use_bias=False),
    layers.BatchNormalization(),
    layers.LeakyReLU(),
    layers.Conv2DTranspose(64, (5,5), strides=(2,2), padding='same', use_bias=False),
    layers.BatchNormalization(),
    layers.LeakyReLU(),
    layers.Conv2DTranspose(3, (5,5), strides=(1,1), padding='same', use_bias=False,
activation='tanh')
])
return model
```

Step 3: Define the Discriminator Network

```
def make_discriminator():
  model = tf.keras.Sequential([
    layers.Conv2D(64, (5,5), strides=(2,2), padding='same', input_shape=[32,32,3]),
    layers.LeakyReLU(),
    layers.Dropout(0.3),
    layers.Conv2D(128, (5,5), strides=(2,2), padding='same'),
    layers.LeakyReLU(),
    layers.Dropout(0.3),
    layers.Flatten(),
    layers.Dense(1)
  ])
  return model
```

Step 4: Initialize Models, Loss, and Optimizers

```
generator = make_generator()
discriminator = make_discriminator()

cross_entropy = tf.keras.losses.BinaryCrossentropy(from_logits=True)
gen_optimizer = tf.keras.optimizers.Adam(1e-4)
disc_optimizer = tf.keras.optimizers.Adam(1e-4)
```

Step 5: Define Generator and Discriminator Loss Functions

```
def generator_loss(fake_output):
  return cross_entropy(tf.ones_like(fake_output), fake_output)

def discriminator_loss(real_output, fake_output):
  real_loss = cross_entropy(tf.ones_like(real_output), real_output)
```

```
    fake_loss = cross_entropy(tf.zeros_like(fake_output), fake_output)
    return real_loss + fake_loss
```

Step 6: Define Training Step Function

```
@tf.function
def train_step(images):
    noise = tf.random.normal([128, 100])
    with tf.GradientTape() as gen_tape, tf.GradientTape() as disc_tape:
        generated_images = generator(noise, training=True)
        real_output = discriminator(images, training=True)
        fake_output = discriminator(generated_images, training=True)
        gen_loss = generator_loss(fake_output)
        disc_loss = discriminator_loss(real_output, fake_output)
    gradients_of_generator = gen_tape.gradient(gen_loss, generator.trainable_variables)
    gradients_of_discriminator = disc_tape.gradient(disc_loss,
discriminator.trainable_variables)
    gen_optimizer.apply_gradients(zip(gradients_of_generator,
generator.trainable_variables))
    disc_optimizer.apply_gradients(zip(gradients_of_discriminator,
discriminator.trainable_variables))
```

Step 7: Train the GAN

```
EPOCHS = 3
for epoch in range(EPOCHS):
    for image_batch in x_train:
        train_step(image_batch)
    print(f"Epoch {epoch+1}/{EPOCHS} completed.")
```

Step 8: Generate and Visualize New Images

```
noise = tf.random.normal([16, 100])
generated_images = generator(noise, training=False)

plt.figure(figsize=(8,8))
for i in range(16):
    plt.subplot(4,4,i+1)
    plt.imshow((generated_images[i] + 1) / 2)
    plt.axis('off')
plt.show()
```

Output: