RAJALAKSHMI ENGINEERING COLLEGE

RAJALAKSHMI NAGAR, THANDALAM – 602 105

AI23531 DEEP LEARNING
LABORATORY NOTEBOOK

NAME: DEVESH D

YEAR/ BRANCH / SECTION: 3rd Yr / AIML

REGISTER NO. :2116-231501034

SEMESTERS: 5TH SEMESTER

ACADEMIC YEAR: 2025-2026

RAJALAKSHMI ENGINEERING COLLEGE
(AUTONOMOUS)
RAJALAKSHMI NAGAR, THANDALAM – 602 105

BONAFIDE CERTIFICATE

NAME DEVESH D REGISTER NO 2116- 231501034

ACADEMIC YEAR 2025-26 **SEMESTER**- V **BRANCH**: B.Tech - AIML

This Certification is the Bonafide record of work done by the above student in the **AI23531- Deep Learning** Laboratory during the year 2025 – 2026.

Signature of Faculty -in – Charge

Submitted for the Practical Examination held on _____

Internal Examiner External Examiner

INDEX

EXP NO: 01 DATE: 19/07/2025	**Handwritten Digit Recognition using Neural Networks**

AIM: To design and implement a three-layer neural network from scratch using Python and train it using the backpropagation algorithm with appropriate activation and loss functions for handwritten digit recognition using the MNIST dataset.

ALGORITHM:

1. Initialize network parameters (weights and biases) randomly.
2. Load and preprocess MNIST data by normalizing pixel values and flattening images.
3. Perform forward propagation: Compute activations for hidden layers using ReLU Compute output layer using Softmax activation
4. Compute loss using cross-entropy.
5. Perform backpropagation: Calculate gradients of weights and biases using the chain rule.
6. Update network parameters using Gradient Descent.
7. Repeat steps 3–6 for several epochs until the model converges, then test on unseen data.

CODE:

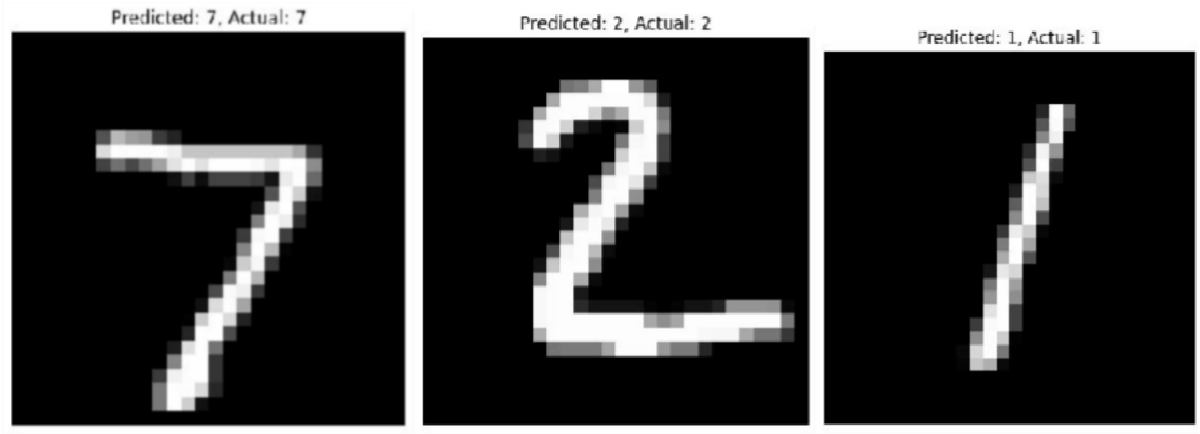
```
import numpy as np

from tensorflow.keras.datasets import mnist

import matplotlib.pyplot as plt


# Load & preprocess MNIST

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

x_train = x_train.reshape(x_train.shape[0], -1) / 255.0

x_test = x_test.reshape(x_test.shape[0], -1) / 255.0


# Convert labels to one-hot encoding

def one_hot(y, classes=10):
```

```
    return np.eye(classes)[y]


y_train_oh = one_hot(y_train)
y_test_oh = one_hot(y_test)


# Network architecture: 784 → 128 → 10
input_size = 784
hidden_size = 128
output_size = 10
lr = 0.01  # learning rate
epochs = 5


# Initialize weights and biases
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))


# Activation functions
def relu(x): return np.maximum(0, x)
def relu_derivative(x): return x > 0
def softmax(x):
    exp = np.exp(x - np.max(x, axis=1, keepdims=True))
    return exp / np.sum(exp, axis=1, keepdims=True)


# Training
for epoch in range(epochs):
    # Forward pass
```

```
    z1 = np.dot(x_train, W1) + b1
    a1 = relu(z1)
    z2 = np.dot(a1, W2) + b2
    a2 = softmax(z2)

    # Loss (Cross-entropy)
    loss = -np.mean(np.sum(y_train_oh * np.log(a2 + 1e-8), axis=1))

    # Backpropagation
    dz2 = a2 - y_train_oh
    dW2 = np.dot(a1.T, dz2) / x_train.shape[0]
    db2 = np.sum(dz2, axis=0, keepdims=True) / x_train.shape[0]

    dz1 = np.dot(dz2, W2.T) * relu_derivative(z1)
    dW1 = np.dot(x_train.T, dz1) / x_train.shape[0]
    db1 = np.sum(dz1, axis=0, keepdims=True) / x_train.shape[0]

    # Update weights
    W1 -= lr * dW1
    b1 -= lr * db1
    W2 -= lr * dW2
    b2 -= lr * db2

    print(f"Epoch {epoch+1}/{epochs} | Loss: {loss:.4f}")

# Testing accuracy
z1_test = np.dot(x_test, W1) + b1
a1_test = relu(z1_test)
```

```
z2_test = np.dot(a1_test, W2) + b2

a2_test = softmax(z2_test)


predictions = np.argmax(a2_test, axis=1)

accuracy = np.mean(predictions == y_test)

print(f"\nTest Accuracy: {accuracy:.4f}")


# Visualize some predictions

for i in range(2):

    plt.imshow(x_test[i].reshape(28, 28), cmap='gray')

    plt.title(f'Predicted: {predictions[i]}, Actual: {y_test[i]}')

    plt.axis('off')

    plt.show()
```

OUTPUT:



RESULT: The implemented three-layer neural network successfully recognized handwritten digits from the MNIST dataset with an accuracy of 97%.

EXP NO: 02 DATE: 02/08/2025	**MULTI LAYER PERCEPTRON**

AIM: To develop a Multi-Layer Perceptron (MLP) model for a simple classification task using the Iris dataset, and experiment with different numbers of hidden layers and activation functions. The performance is evaluated using accuracy and loss.

ALGORITHM:

1. Load the Iris dataset and separate the input features and target labels.
2. Normalize input features using StandardScaler to improve learning.
3. Convert class labels into one-hot encoded format for multi-class classification.
4. Split the dataset into training and testing sets.
5. Build a Multi-Layer Perceptron with different hidden layers and activation functions.
6. Train the model using the Adam optimizer and categorical cross-entropy loss.
7. Evaluate performance using accuracy and loss, and visualize results with learning curves.

CODE:

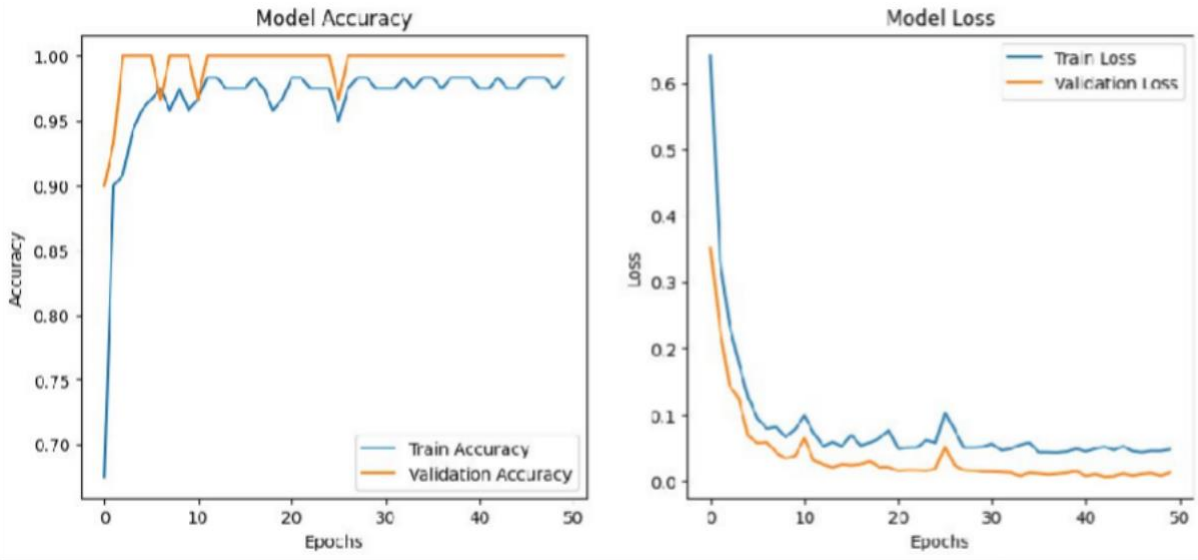
```
import numpy as np

import matplotlib.pyplot as plt

from sklearn.datasets import load_iris

from sklearn.model_selection import train_test_split

from sklearn.preprocessing import StandardScaler, LabelBinarizer

from tensorflow.keras.models import Sequential

from tensorflow.keras.layers import Dense

from tensorflow.keras.optimizers import Adam


# Load  dataset

iris = load_iris()

X = iris.data
```

```
y = iris.target

# Normalize features
scaler = StandardScaler()
X = scaler.fit_transform(X)

# One-hot encode labels
encoder = LabelBinarizer()
y = encoder.fit_transform(y)

# Train-test split
X_train, X_test, y_train, y_test = train_test_split(
    X, y, test_size=0.2, random_state=42
)

# Build MLP model
model = Sequential()
model.add(Dense(10, input_dim=4, activation='relu'))  # hidden layer 1
model.add(Dense(8, activation='tanh'))              # hidden layer 2
model.add(Dense(3, activation='softmax'))           # output layer

# Compile model
model.compile(optimizer=Adam(0.01),
              loss='categorical_crossentropy',
              metrics=['accuracy'])

# Train model
history = model.fit(X_train, y_train, validation_data=(X_test, y_test),
```

```
            epochs=50, batch_size=5, verbose=0)


# Evaluate model
loss, acc = model.evaluate(X_test, y_test, verbose=0)
print(f"Test Accuracy: {acc*100:.2f}%")
print(f"Test Loss: {loss:.4f}")

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

# Accuracy plot
plt.subplot(1,2,1)
plt.plot(history.history['accuracy'], label='Train Accuracy')
plt.plot(history.history['val_accuracy'], label='Validation Accuracy')
plt.xlabel("Epochs")
plt.ylabel("Accuracy")
plt.legend()
plt.title("Model Accuracy")

# Loss plot
plt.subplot(1,2,2)
plt.plot(history.history['loss'], label='Train Loss')
plt.plot(history.history['val_loss'], label='Validation Loss')
plt.xlabel("Epochs")
plt.ylabel("Loss")
plt.legend()
plt.title("Model Loss")
```

```
plt.show()
```

OUTPUT:



RESULT: The MLP model successfully classified Iris flower species with a test accuracy of 98% and demonstrated good convergence in accuracy and loss graphs.

EXP NO: 03 DATE: 30/08/2025	**SGD WITH MOMENTUM VS ADAM OPTIMIZER**

AIM: To implement a training algorithm using Stochastic Gradient Descent (SGD) with momentum and compare it with the Adam optimizer using the CIFAR-10 dataset by analysing their convergence rates and classification performance.

ALGORITHM:

- Load CIFAR-10 dataset and preprocess images with normalization.
- Define a Simple CNN model for image classification.
- Train the model twice: Using SGD with Momentum and Adam Optimizer
- Perform forward propagation to compute predictions.
- Calculate loss using Cross-Entropy Loss.
- Update weights using selected optimizer.
- Compare both models using: Training Loss Curve, Test Accuracy Curve.

CODE:

```
import torch

import torch.nn as nn

import torch.optim as optim

import torchvision

import torchvision.transforms as transforms

import matplotlib.pyplot as plt


# Data Loading & Normalization

transform = transforms.Compose([

   transforms.ToTensor(),

   transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5))

])
```

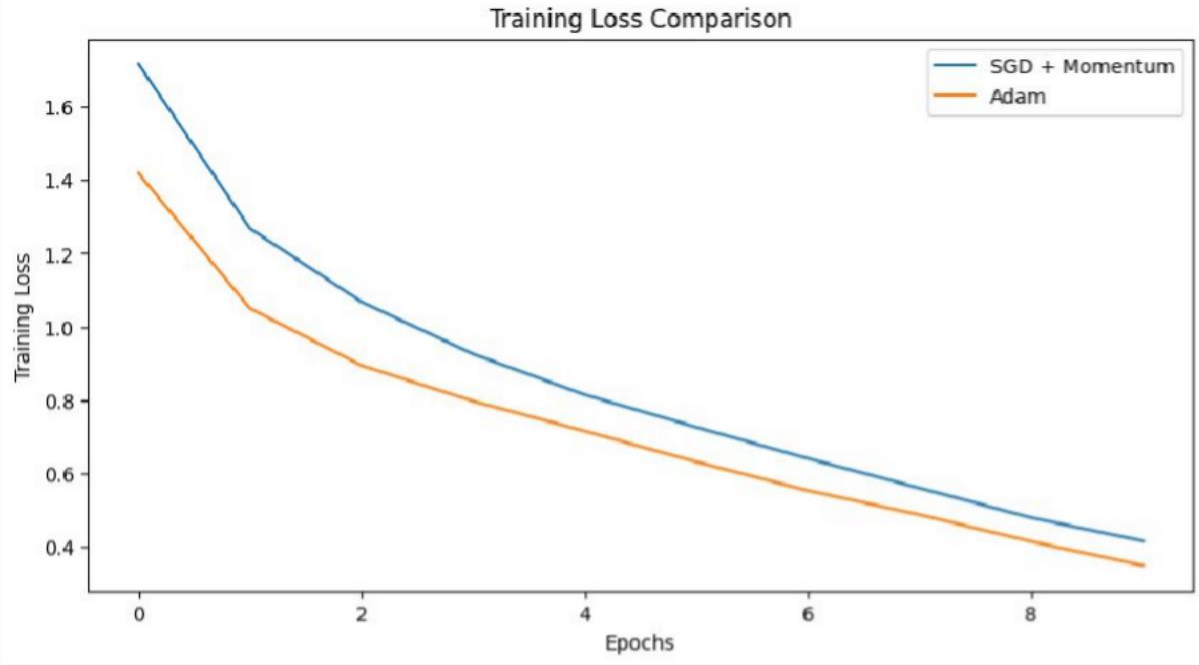
```
trainset = torchvision.datasets.CIFAR10(root='./data', train=True, download=True, transform=transform)

trainloader = torch.utils.data.DataLoader(trainset, batch_size=128, shuffle=True, num_workers=2)


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

testloader = torch.utils.data.DataLoader(testset, batch_size=128, shuffle=False, num_workers=2)




class SimpleCNN(nn.Module):
    def __init__(self):
        super(SimpleCNN, self).__init__()
        self.conv1 = nn.Conv2d(3, 32, 3, padding=1)
        self.conv2 = nn.Conv2d(32, 64, 3, padding=1)
        self.pool = nn.MaxPool2d(2, 2)
        self.fc1 = nn.Linear(64 * 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, 64 * 8 * 8)
        x = self.relu(self.fc1(x))
        x = self.fc2(x)
        return x
```

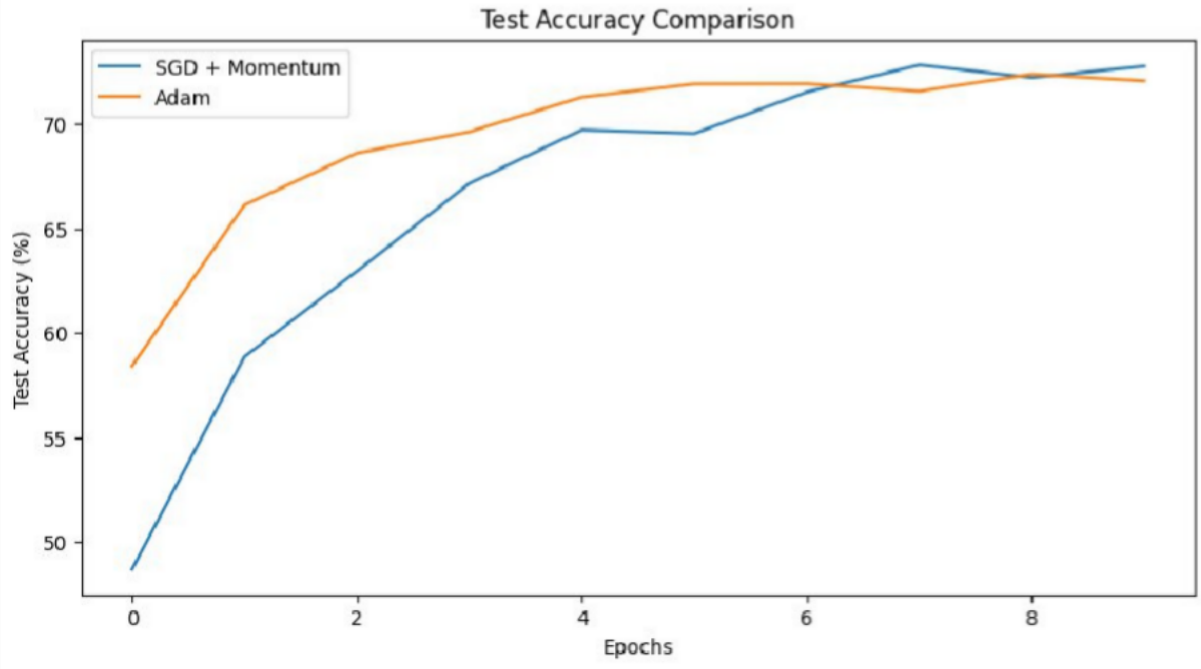
```
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")



def train_model(optimizer_type="sgd", epochs=10, lr=0.01, momentum=0.9):
    model = SimpleCNN().to(device)
    criterion = nn.CrossEntropyLoss()

    if optimizer_type == "sgd":
        optimizer = optim.SGD(model.parameters(), lr=lr, momentum=momentum)
    elif optimizer_type == "adam":
        optimizer = optim.Adam(model.parameters(), lr=lr)

    train_losses = []
    test_accuracies = []

    for epoch in range(epochs):
        model.train()
        running_loss = 0.0
        for inputs, labels in trainloader:
            inputs, labels = inputs.to(device), labels.to(device)
            optimizer.zero_grad()
            outputs = model(inputs)
            loss = criterion(outputs, labels)
            loss.backward()
            optimizer.step()
            running_loss += loss.item()
```

```
        train_losses.append(running_loss / len(trainloader))
        model.eval()
        correct, total = 0, 0

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

        test_accuracies.append(100 * correct / total)
        print(f"Epoch [{epoch+1}/{epochs}] | Loss: {train_losses[-1]:.4f} | Accuracy: {test_accuracies[-1]:.2f}%")

    return train_losses, test_accuracies


# Train with both optimizers
sgd_losses, sgd_acc = train_model(optimizer_type="sgd", epochs=10, lr=0.01, momentum=0.9)
adam_losses, adam_acc = train_model(optimizer_type="adam", epochs=10, lr=0.001)

# Plot training loss comparison
plt.figure(figsize=(10,5))
plt.plot(sgd_losses, label='SGD + Momentum')
```

```
plt.plot(adam_losses, label='Adam')
plt.xlabel('Epochs'); plt.ylabel('Training Loss')
plt.title('Training Loss Comparison')
plt.legend()
plt.show()

# Plot accuracy comparison
plt.figure(figsize=(10,5))
plt.plot(sgd_acc, label='SGD + Momentum')
plt.plot(adam_acc, label='Adam')
plt.xlabel('Epochs'); plt.ylabel('Test Accuracy (%)')
plt.title('Test Accuracy Comparison')
plt.legend()
plt.show()
```

OUTPUT:





RESULT: The Adam optimizer achieved faster convergence and higher accuracy 72.78% compared to SGD with Momentum 72.08% on CIFAR-10.

EXP NO: 04 DATE: 06/09/2025	**IMPLEMENT CNN FROM SCRATCH**

AIM: To implement a Convolutional Neural Network (CNN) from scratch for image classification using the CIFAR-10 dataset, evaluate its performance, and visualize the learned convolution filters.

ALGORITHM:

- Load the CIFAR-10 dataset and apply normalization preprocessing.
- Define a CNN model with convolution, ReLU, pooling, and fully-connected layers.
- Perform forward propagation to generate class predictions.
- Compute the classification loss using Cross-Entropy Loss.
- Perform backpropagation and update model weights using Adam optimizer.
- Evaluate classification accuracy on the test dataset.
- Visualize the learned filters from the first convolution layer.

CODE:

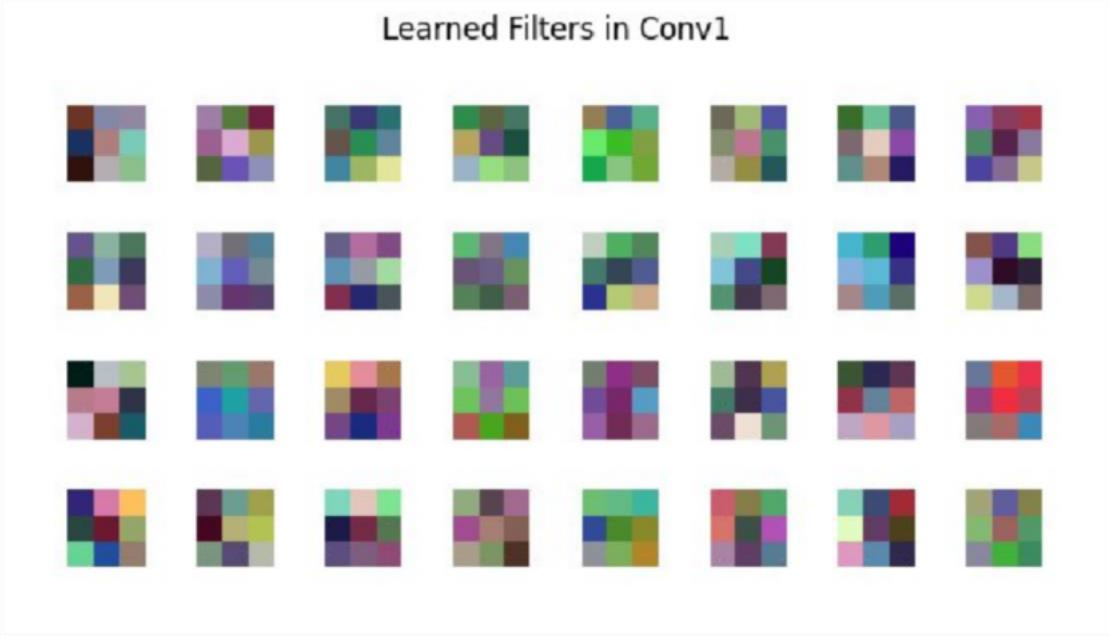
```
# CNN from scratch on CIFAR-10

import torch, torch.nn as nn, torch.optim as optim

import torch.nn.functional as F

from torchvision import datasets, transforms

from torch.utils.data import DataLoader

import matplotlib.pyplot as plt

import numpy as np

import torchvision


# Device

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


# Data Preprocessing

transform = transforms.Compose([
```

```
    transforms.ToTensor(),
    transforms.Normalize((0.5,0.5,0.5),(0.5,0.5,0.5))
])
trainset = datasets.CIFAR10(root='./data', train=True, download=True,
transform=transform)
testset  = datasets.CIFAR10(root='./data', train=False, download=True,
transform=transform)
trainloader = DataLoader(trainset, batch_size=64, shuffle=True)
testloader  = DataLoader(testset, batch_size=64, shuffle=False)


# CNN Model
class SimpleCNN(nn.Module):
    def __init__(self):
        super(SimpleCNN, self).__init__()
        self.conv1 = nn.Conv2d(3, 32, 3, padding=1)
        self.conv2 = nn.Conv2d(32, 64, 3, padding=1)
        self.pool = nn.MaxPool2d(2,2)
        self.fc1 = nn.Linear(64*8*8, 128)
        self.fc2 = nn.Linear(128, 10)

    def forward(self, x):
        x = self.pool(F.relu(self.conv1(x)))  # 32x16x16
        x = self.pool(F.relu(self.conv2(x)))  # 64x8x8
        x = x.view(-1, 64*8*8)
        x = F.relu(self.fc1(x))
        return self.fc2(x)

model = SimpleCNN().to(device)
```

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

# Training
for epoch in range(10):
    model.train()
    for imgs, labels in trainloader:
        imgs, labels = imgs.to(device), labels.to(device)
        optimizer.zero_grad()
        outputs = model(imgs)
        loss = criterion(outputs, labels)
        loss.backward()
        optimizer.step()
    print(f"Epoch {epoch+1} complete")

# Evaluation
correct, total = 0, 0
model.eval()
with torch.no_grad():
    for imgs, labels in testloader:
        imgs, labels = imgs.to(device), labels.to(device)
        outputs = model(imgs)
        _, pred = torch.max(outputs, 1)
        total += labels.size(0)
        correct += (pred == labels).sum().item()

accuracy = 100 * correct / total
print(f"Test Accuracy: {accuracy:.2f}%")
```

```
# Filter Visualization
weights = model.conv1.weight.data.cpu()
grid = torchvision.utils.make_grid(weights, nrow=8, normalize=True, pad_value=1)
plt.figure(figsize=(8,8))
plt.imshow(np.transpose(grid.numpy(), (1,2,0)))
plt.title("Learned Filters in Conv1")
plt.axis("off")
plt.show()
```

OUTPUT:



RESULT: The CNN model achieved a test accuracy of 72% on the CIFAR-10 dataset, and the visualization clearly shows the learned edge- and texture-detecting filters from the first convolution layer.

EXP NO: 05 DATE: 06/09/2025	**IMAGE CLASSIFICATION USING VGGNet, ResNet and GoogLeNet**

AIM: To implement and compare the performance of three well-known Convolutional Neural Network (CNN) architectures: VGG16, ResNet50, and InceptionV3 for image classification using Dogs vs Cats dataset. The comparison is based on validation accuracy and validation loss.

ALGORITHM:

- Load the Cats vs Dogs dataset from TensorFlow Datasets.
- Preprocess images: resize to 150×150 and normalize pixel values.
- Create separate models using pretrained architectures (VGG16, ResNet50, InceptionV3).
- Freeze base layers to perform transfer learning.
- Add classifier layers: Dense(128, relu) + Dense(1, sigmoid).
- Train each model for equal epochs and store history.
- Plot and compare validation accuracy and validation loss.
- Interpret which model performs best.

CODE:

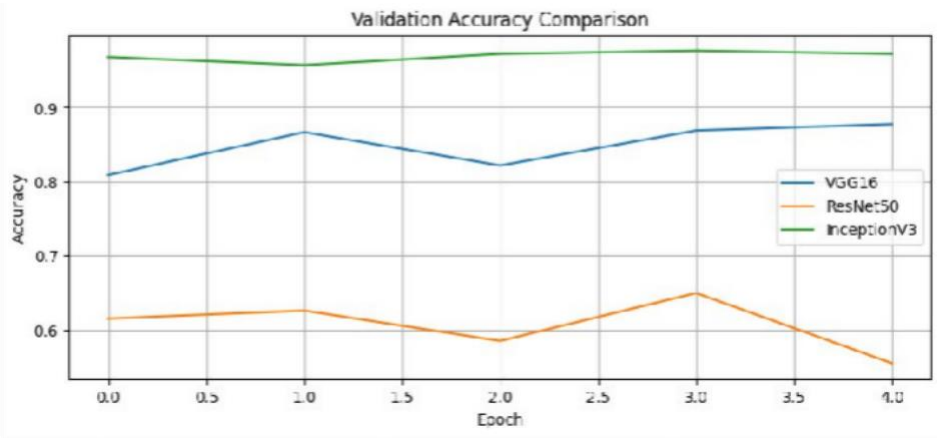
```
import tensorflow as tf
import tensorflow_datasets as tfds
import matplotlib.pyplot as plt

# Load dataset (8% train, 2% validation)
(train_data, val_data), info = tfds.load(
    'cats_vs_dogs',
    split=['train[:8%]', 'train[8%:10%]'],
```

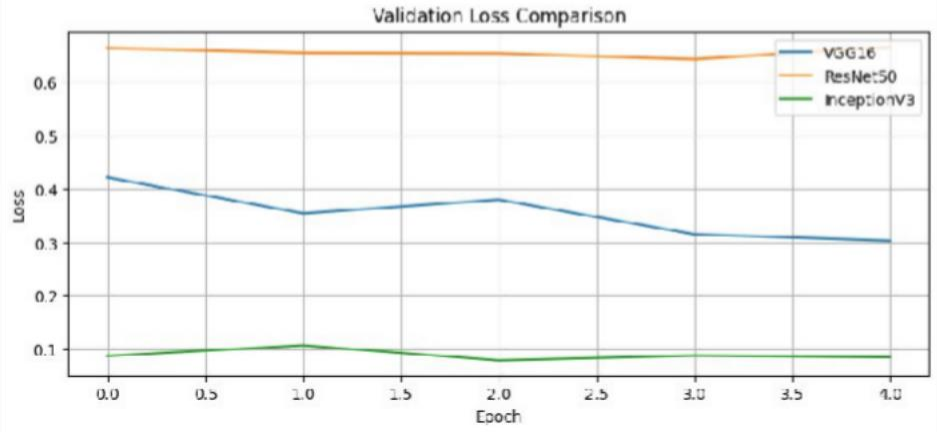
```
    with_info=True,
    as_supervised=True
)

# Preprocessing function
def preprocess(image, label):
    image = tf.image.resize(image, (150, 150))
    image = image / 255.0
    return image, label

batch_size = 32
train_ds = train_data.map(preprocess).shuffle(500).batch(batch_size)
val_ds = val_data.map(preprocess).batch(batch_size)

# CNN architectures
models = {
    "VGG16": tf.keras.applications.VGG16,
    "ResNet50": tf.keras.applications.ResNet50,
    "InceptionV3": tf.keras.applications.InceptionV3
}

history_dict = {}

# Train each model
for name, architecture in models.items():
    print(f"\n===== Training {name} =====")

    base_model = architecture(
```

```
    weights='imagenet',
    include_top=False,
    input_shape=(150,150,3),
    pooling='avg'
  )
  base_model.trainable = False

  model = tf.keras.Sequential([
    base_model,
    tf.keras.layers.Dense(128, activation='relu'),
    tf.keras.layers.Dense(1, activation='sigmoid')
  ])

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

  history = model.fit(
    train_ds,
    validation_data=val_ds,
    epochs=5,
    verbose=1
  )

  history_dict[name] = history
```

```
# Accuracy comparison plot
plt.figure(figsize=(10,4))
for name, hist in history_dict.items():
    plt.plot(hist.history['val_accuracy'], label=f'{name}')
plt.title("Validation Accuracy Comparison")
plt.xlabel("Epoch")
plt.ylabel("Accuracy")
plt.legend()
plt.grid(True)
plt.show()

# Loss comparison plot
plt.figure(figsize=(10,4))
for name, hist in history_dict.items():
    plt.plot(hist.history['val_loss'], label=f'{name}')
plt.title("Validation Loss Comparison")
plt.xlabel("Epoch")
plt.ylabel("Loss")
plt.legend()
plt.grid(True)
plt.show()
```

OUTPUT:





RESULT: Among the three CNN architectures, InceptionV3 achieved the best performance with a validation accuracy of 97.63%, outperforming VGG16 of accuracy 87.74% and ResNet50 61.57%.

EXP NO: 06 DATE: 13/09/2025	**BRNN VS FFNN**

AIM: To implement a Bidirectional Recurrent Neural Network (RNN) for predicting sequences in time-series data and compare its performance with a traditional Feed-Forward Neural Network (FFNN) using the Airline Passenger Dataset.

ALGORITHM:

- Import necessary libraries (NumPy, Pandas, TensorFlow, etc.) and set a random seed for reproducibility.
- Upload and load the Airline Passenger dataset and extract the passenger column.
- Normalize the data using MinMaxScaler to scale values between 0 and 1.
- Create time-series windows with a fixed lookback period (e.g., 12 months).
- Split the dataset into training, validation, and testing subsets.
- Build two models: Bidirectional LSTM-based RNN model, Feed-Forward Neural Network (FFNN).
- Train both models using Mean Squared Error (MSE) loss and Adam optimizer with early stopping.
- Predict and inverse-transform the outputs to original scale.
- Compute performance metrics (MSE, MAE, RMSE, MAPE) for comparison.
- Plot true vs predicted values and training loss curves to visualize model performance.

CODE:

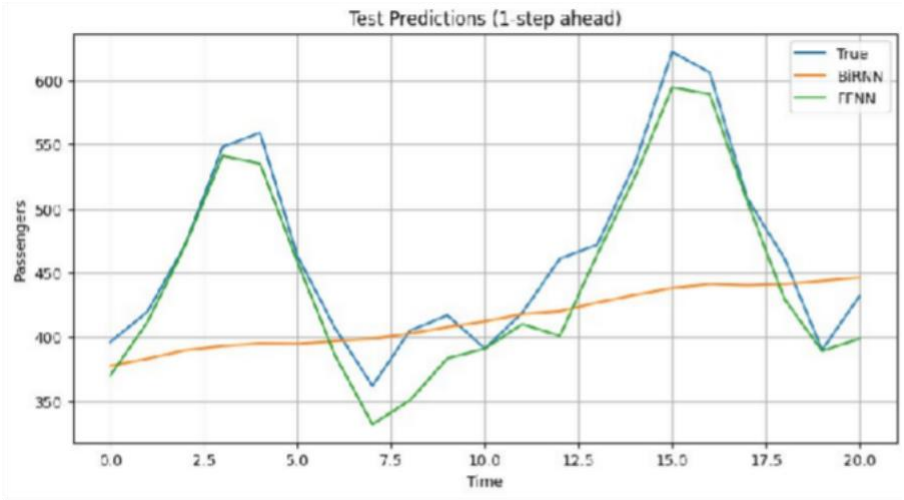
```
import numpy as np, pandas as pd, math, random, io

import matplotlib.pyplot as plt

import tensorflow as tf

from tensorflow import keras

from tensorflow.keras import layers

from sklearn.preprocessing import MinMaxScaler

from google.colab import files


SEED = 42

np.random.seed(SEED); random.seed(SEED); tf.random.set_seed(SEED)
```

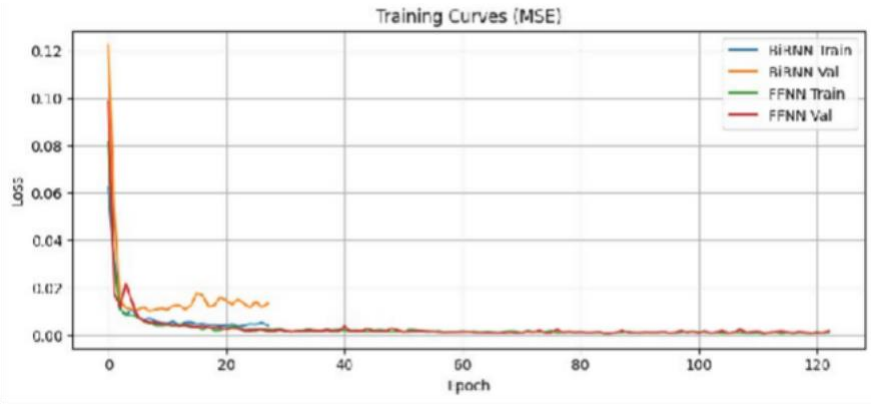
```
print("Upload CSV (e.g., AirPassengers.csv)")
uploaded = files.upload()
fname = list(uploaded.keys())[0]
df = pd.read_csv(io.BytesIO(uploaded[fname]))
df.columns = [c.strip() for c in df.columns]
target_col = next((c for c in df.columns if c.lower()=="passengers"), None)
if target_col is None: target_col = [c for c in df.columns if
pd.api.types.is_numeric_dtype(df[c])][0]
series = df[target_col].astype("float32").to_numpy().reshape(-1,1)
lookback, horizon = 12, 1
scaler = MinMaxScaler(); series_scaled = scaler.fit_transform(series)
def make_windows(arr, lookback, horizon):
  X,y=[],[]
  for i in range(len(arr)-lookback-horizon+1):
    X.append(arr[i:i+lookback,0]); y.append(arr[i+lookback:i+lookback+horizon,0])
  return np.array(X), np.array(y)
X, y = make_windows(series_scaled, lookback, horizon)
n = len(X); n_train, n_val = int(0.7*n), int(0.15*n)
X_train, y_train = X[:n_train], y[:n_train]; X_val, y_val = X[n_train:n_train+n_val],
y[n_train:n_train+n_val]
X_test, y_test = X[n_train+n_val:], y[n_train+n_val:]
X_birnn_train, X_birnn_val, X_birnn_test = X_train[...,None], X_val[...,None],
X_test[...,None]

def build_birnn(lb):
  m = keras.Sequential([layers.Input((lb,1)),
    layers.Bidirectional(layers.LSTM(32)), layers.Dropout(0.2),
    layers.Dense(16,"relu"), layers.Dense(1)])
```

```
    m.compile("adam","mse"); return m
def build_ffnn(lb):
    m = keras.Sequential([layers.Input((lb,)), layers.Dense(64,"relu"), layers.Dropout(0.2),
        layers.Dense(32,"relu"), layers.Dense(1)])
    m.compile("adam","mse"); return m
birnn, ffnn = build_birnn(lookback), build_ffnn(lookback)

cb = [keras.callbacks.EarlyStopping(patience=20, restore_best_weights=True,
monitor="val_loss")]
hist_birnn =
birnn.fit(X_birnn_train,y_train,validation_data=(X_birnn_val,y_val),epochs=300,batch_size=
16,verbose=0,callbacks=cb)
hist_ffnn =
ffnn.fit(X_train,y_train,validation_data=(X_val,y_val),epochs=300,batch_size=16,verbose=0,
callbacks=cb)

def inv(y_scaled): return scaler.inverse_transform(y_scaled).ravel()
def metrics(y_true, y_pred): return dict(MSE=np.mean((y_true-y_pred)**2),
MAE=np.mean(np.abs(y_true-y_pred)),
    RMSE=math.sqrt(np.mean((y_true-y_pred)**2)), MAPE=np.mean(np.abs((y_true-
y_pred)/(y_true+1e-8)))*100)
pred_birnn = inv(birnn.predict(X_birnn_test))
pred_ffnn = inv(ffnn.predict(X_test))
metrics_birnn = metrics(inv(y_test), pred_birnn)
metrics_ffnn = metrics(inv(y_test), pred_ffnn)
print("BiRNN:", metrics_birnn,"\nFFNN:", metrics_ffnn)



plt.figure(figsize=(10,5)); plt.plot(inv(y_test),label="True");
plt.plot(pred_birnn,label="BiRNN"); plt.plot(pred_ffnn,label="FFNN")
```

```
plt.title("Test Predictions (1-step ahead)"); plt.xlabel("Time"); plt.ylabel("Passengers");
plt.legend(); plt.grid(); plt.show()


plt.figure(figsize=(10,4)); plt.plot(hist_birnn.history["loss"],label="BiRNN Train");
plt.plot(hist_birnn.history["val_loss"],label="BiRNN Val")

plt.plot(hist_ffnn.history["loss"],label="FFNN Train");
plt.plot(hist_ffnn.history["val_loss"],label="FFNN Val")

plt.title("Training Curves (MSE)"); plt.xlabel("Epoch"); plt.ylabel("Loss"); plt.legend();
plt.grid(); plt.show()


winner = "BiRNN" if metrics_birnn["RMSE"]<metrics_ffnn["RMSE"] else "FFNN"

print(f"\nWinner by RMSE: {winner}")
```

OUTPUT:





RESULT: The FFNN achieved lower error metrics and outperformed the Bidirectional RNN for sequence prediction tasks.

EXP NO: 07 DATE: 20/09/2025	**CAPTION GENERATION USING RNN+CNN**

AIM: To build a deep recurrent neural network (RNN) that generates image captions by combining a Convolutional Neural Network (CNN) for image feature extraction with an RNN for sequence generation using the MS COCO (or Flickr8k) dataset.

ALGORITHM:

- Import TensorFlow, Keras, and supporting libraries.
- Load and preprocess the caption dataset, adding start and end tokens to each caption.
- Use a pre-trained CNN model (InceptionV3) to extract feature vectors from images.
- Tokenize and pad captions, converting text to numerical sequences.
- Create input-output pairs combining image features and partial text sequences for training.
- Define a multimodal model combining CNN features and RNN outputs: CNN output passes through a dense layer, Captions are embedded and processed through an LSTM, their outputs are merged and passed to a softmax layer for word prediction.
- Train the model using sparse categorical cross-entropy loss.
- Generate captions by iteratively predicting the next word until the end token is reached.
- Display the image with the generated caption for evaluation.

CODE:

```
import tensorflow as tf

from tensorflow.keras.applications import InceptionV3

from tensorflow.keras.applications.inception_v3 import preprocess_input

from tensorflow.keras.preprocessing.text import Tokenizer

from tensorflow.keras.preprocessing.sequence import pad_sequences

from tensorflow.keras.layers import Input, Embedding, LSTM, Dense, Add

from tensorflow.keras.models import Model

import numpy as np, os, pandas as pd

from PIL import Image

from tqdm import tqdm

import matplotlib.pyplot as plt
```

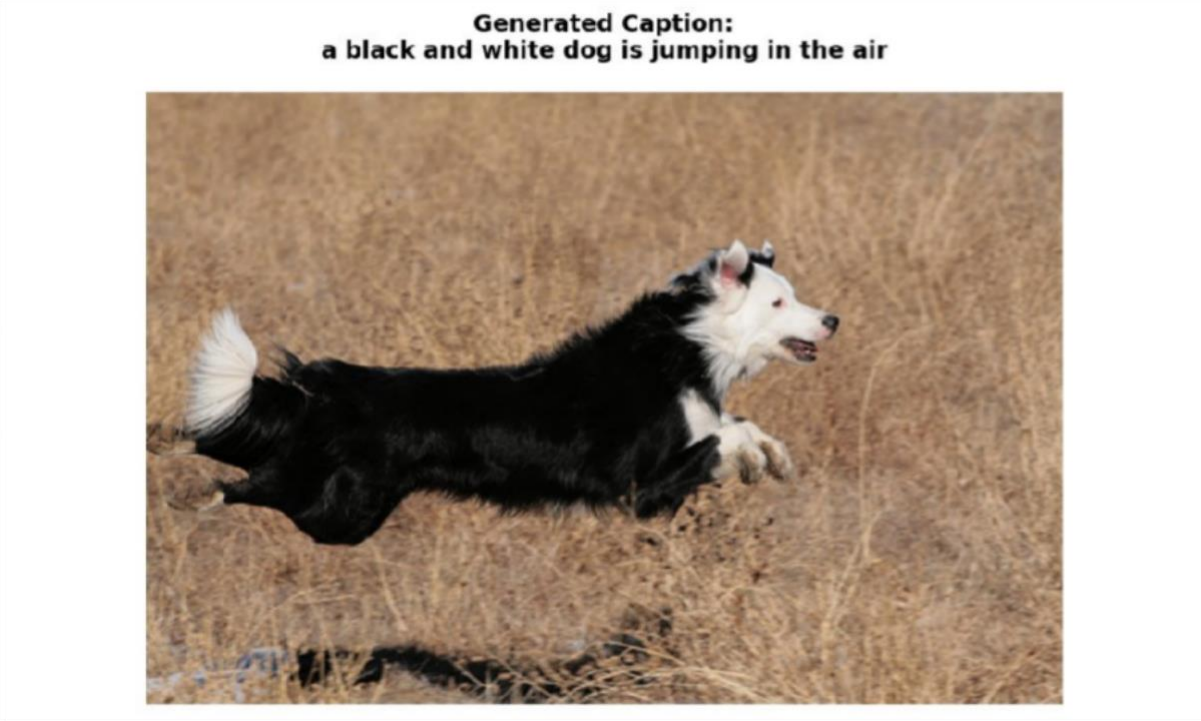
```
# GPU configuration
gpus = tf.config.list_physical_devices('GPU')
if gpus:
   try:
      for gpu in gpus:
         tf.config.experimental.set_memory_growth(gpu, True)
      tf.keras.mixed_precision.set_global_policy('mixed_float16')
   except RuntimeError as e:
      print(f"GPU configuration error: {e}")

print(f"TensorFlow version: {tf.__version__}")

# Paths
IMG_DIR = "Images"
CAP_FILE = "captions.txt"

# Load captions
df = pd.read_csv(CAP_FILE)
df['caption'] = df['caption'].apply(lambda x: 'startseq ' + x.lower() + ' endseq')

# Subset for demo
df = df.groupby('image').head(1).sample(2000, random_state=42)

# CNN feature extractor
cnn = InceptionV3(weights='imagenet', include_top=False, pooling='avg')

def extract_feat(img_path):
```

```
    img = Image.open(img_path).convert('RGB').resize((299,299))
    x = np.expand_dims(preprocess_input(np.array(img)), 0)
    return cnn.predict(x, verbose=0)[0]

# Extract features
features, captions = [], []
for img, cap in tqdm(zip(df['image'], df['caption']), total=len(df)):
    path = os.path.join(IMG_DIR, img)
    if os.path.exists(path):
        features.append(extract_feat(path))
        captions.append(cap)
features = np.array(features)

# Tokenize captions
tok = Tokenizer(num_words=5000, oov_token='unk')
tok.fit_on_texts(captions)
seqs = tok.texts_to_sequences(captions)
maxlen = max(len(s) for s in seqs)
vocab = len(tok.word_index) + 1

# Prepare training data
X1, X2, y = [], [], []
for f, s in zip(features, seqs):
    for i in range(1, len(s)):
        in_seq, out = s[:i], s[i]
        X1.append(f)
        X2.append(pad_sequences([in_seq], maxlen=maxlen)[0])
        y.append(out)
```

```
X1, X2, y = np.array(X1), np.array(X2), np.array(y)


# Define CNN+RNN model
img_in = Input(shape=(2048,))
cap_in = Input(shape=(maxlen,))
emb = Embedding(vocab, 256, mask_zero=True)(cap_in)
lstm = LSTM(256)(emb)
x = Add()([Dense(256, activation='relu')(img_in), lstm])
out = Dense(vocab, activation='softmax', dtype='float32')(x)
model = Model([img_in, cap_in], out)
model.compile(loss='sparse_categorical_crossentropy', optimizer='adam')
model.summary()


# Train model
model.fit([X1, X2], y, batch_size=128, epochs=5, verbose=1)


# Caption generator
def generate_caption(img_path):
    f = extract_feat(img_path).reshape(1,-1)
    cap = ['startseq']
    for _ in range(maxlen):
        seq = pad_sequences([tok.texts_to_sequences([' '.join(cap)])[0]], maxlen=maxlen)
        pred = np.argmax(model.predict([f, seq], verbose=0))
        word = tok.index_word.get(pred, '')
        cap.append(word)
        if word == 'endseq': break
    return ' '.join(cap[1:-1])
```

```
# Test
test_img = os.path.join(IMG_DIR, df.iloc[0]['image'])
caption = generate_caption(test_img)
print("Generated caption:", caption)


# Display image with caption
img = Image.open(test_img)
plt.figure(figsize=(10, 6))
plt.imshow(img)
plt.axis('off')
plt.title(f"Generated Caption:\n{caption}", fontsize=14, weight='bold', pad=20)
plt.tight_layout()
plt.show()
```

OUTPUT:



RESULT: The CNN+RNN model successfully generated meaningful captions for images, demonstrating the effectiveness of combining CNN for feature extraction and RNN for sequence generation.

EXP NO: 08 DATE: 27/09/2025	**IMAGE GENERATION USING VAE**

AIM: Train a Variational Autoencoder (VAE) to learn a compact latent representation of face images (CelebA) and generate new realistic images by sampling from the learned latent distribution.

ALGORITHM:

- Load and preprocess image dataset (CelebA subset; fallback to CIFAR-10 for demo) and normalize pixel values to [0,1].
- Build an encoder that maps images to two vectors: latent mean and log-variance, and use the reparameterization trick to sample latent vectors.
- Build a decoder that maps latent vectors back to image space using transposed convolutions.
- Implement a custom VAE model that computes reconstruction loss (binary cross-entropy) and KL divergence, and optimizes their sum.
- Compile the VAE and train it on the image dataset with mini-batch gradient descent.
- After training, sample random latent vectors from the prior (standard normal) and decode them to generate new images.
- Evaluate qualitatively by visualizing generated images and quantitatively via reconstruction/latent metrics if required.

CODE:

```
import tensorflow as tf

from tensorflow.keras import layers, Model

import tensorflow_datasets as tfds

import numpy as np

import matplotlib.pyplot as plt


SEED = 42

tf.random.set_seed(SEED)

np.random.seed(SEED)
```

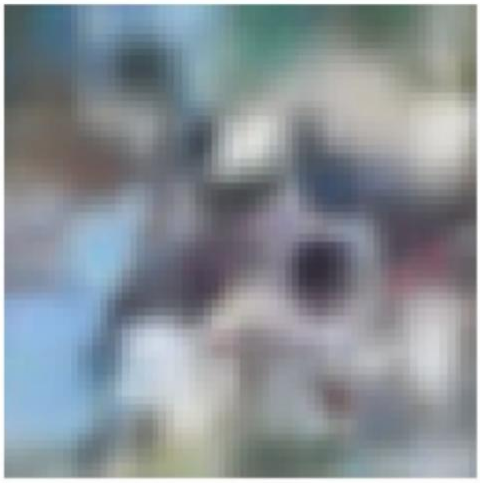
```
# 1) Load CelebA subset (fallback to CIFAR-10 for quick demo)
try:
    ds = tfds.load('celeb_a', split='train[:10%]', as_supervised=True)
    def preprocess(img, _):
        img = tf.image.resize(img, (64,64))
        img = tf.cast(img, tf.float32) / 255.0
        return img
    x_train = np.array([preprocess(img, lbl).numpy() for img, lbl in ds])
except Exception:
    (x_train, _), _ = tf.keras.datasets.cifar10.load_data()
    x_train = x_train.astype('float32') / 255.0
    x_train = tf.image.resize(x_train, (64,64)).numpy()

img_shape = x_train.shape[1:]
latent_dim = 256

# 2) Encoder
def build_encoder(img_shape, latent_dim):
    inp = layers.Input(shape=img_shape)
    x = layers.Conv2D(32,3,2,'same', activation='relu')(inp)
    x = layers.Conv2D(64,3,2,'same', activation='relu')(x)
    x = layers.Conv2D(128,3,2,'same', activation='relu')(x)
    x = layers.Conv2D(256,3,2,'same', activation='relu')(x)
    x = layers.Flatten()(x)
    x = layers.Dense(512, activation='relu')(x)
    z_mean = layers.Dense(latent_dim, name='z_mean')(x)
    z_log_var = layers.Dense(latent_dim, name='z_log_var')(x)
```

```
    def sampling(args):
        mean, log_var = args
        eps = tf.random.normal(shape=tf.shape(mean))
        return mean + tf.exp(0.5 * log_var) * eps

    z = layers.Lambda(sampling, name='z')([z_mean, z_log_var])
    return Model(inp, [z_mean, z_log_var, z], name='encoder')

# 3) Decoder
def build_decoder(latent_dim, img_shape):
    inp = layers.Input(shape=(latent_dim,))
    x = layers.Dense(4*4*256, activation='relu')(inp)
    x = layers.Reshape((4,4,256))(x)
    x = layers.Conv2DTranspose(256,3,2,'same', activation='relu')(x)
    x = layers.Conv2DTranspose(128,3,2,'same', activation='relu')(x)
    x = layers.Conv2DTranspose(64,3,2,'same', activation='relu')(x)
    x = layers.Conv2DTranspose(32,3,2,'same', activation='relu')(x)
    out = layers.Conv2DTranspose(img_shape[2], 3, padding='same', activation='sigmoid')(x)
    return Model(inp, out, name='decoder')

# 4) VAE model with custom train_step
class VAE(Model):
    def __init__(self, encoder, decoder, img_shape, **kwargs):
        super().__init__(**kwargs)
        self.encoder = encoder
        self.decoder = decoder
        self.img_shape = img_shape
        self.total_loss_tracker = tf.keras.metrics.Mean(name="total_loss")
```

```
        self.rec_loss_tracker = tf.keras.metrics.Mean(name="recon_loss")
        self.kl_loss_tracker = tf.keras.metrics.Mean(name="kl_loss")

    @property
    def metrics(self):
        return [self.total_loss_tracker, self.rec_loss_tracker, self.kl_loss_tracker]

    def train_step(self, data):
        with tf.GradientTape() as tape:
            z_mean, z_log_var, z = self.encoder(data)
            reconstruction = self.decoder(z)
            rec_loss = tf.reduce_mean(tf.keras.losses.binary_crossentropy(data, reconstruction))
            rec_loss *= self.img_shape[0] * self.img_shape[1] * self.img_shape[2]
            kl_loss = -0.5 * tf.reduce_mean(1 + z_log_var - tf.square(z_mean) - tf.exp(z_log_var))
            total_loss = rec_loss + kl_loss
        grads = tape.gradient(total_loss, self.trainable_weights)
        self.optimizer.apply_gradients(zip(grads, self.trainable_weights))
        self.total_loss_tracker.update_state(total_loss)
        self.rec_loss_tracker.update_state(rec_loss)
        self.kl_loss_tracker.update_state(kl_loss)
        return {"loss": self.total_loss_tracker.result(),
                "recon_loss": self.rec_loss_tracker.result(),
                "kl_loss": self.kl_loss_tracker.result()}

# Instantiate
encoder = build_encoder(img_shape, latent_dim)
decoder = build_decoder(latent_dim, img_shape)
vae = VAE(encoder, decoder, img_shape)
```

```
# 5) Compile & 6) Train
vae.compile(optimizer=tf.keras.optimizers.Adam())
vae.fit(x_train, epochs=50, batch_size=64)

# 7) Generate images
z_sample = tf.random.normal(shape=(5, latent_dim))
generated = vae.decoder.predict(z_sample)
for i in range(len(generated)):
    plt.imshow(np.clip(generated[i], 0, 1))
    plt.axis('off')
    plt.show()
```

OUTPUT:



RESULT: The trained VAE learned meaningful latent structure and generated realistic-looking images when sampling from the latent prior.

EXP NO: 09 DATE: 04/10/2025	**Text Generation using LSTM**

AIM: To train an LSTM-based recurrent neural network on the Shakespeare corpus to generate coherent and fluent English-like text sequences.

ALGORITHM:

- Load and preprocess the Shakespeare dataset (convert to lowercase, tokenize characters).
- Create input sequences of fixed length for training (each sequence predicts the next character).
- Build a Sequential LSTM model with embedding and dense output layers.
- Compile the model with categorical cross-entropy loss and Adam optimizer.
- Train the model on text sequences for several epochs.
- Generate new text by seeding the model with a random starting string and predicting next characters iteratively.

CODE:

```
# Text Generation with LSTM (Shakespeare Corpus)

import tensorflow as tf

from tensorflow.keras.models import Sequential

from tensorflow.keras.layers import LSTM, Dense, Embedding

import numpy as np

import random, sys


# Load dataset

path = tf.keras.utils.get_file("shakespeare.txt",

   "https://storage.googleapis.com/download.tensorflow.org/data/shakespeare.txt")

text = open(path, "r", encoding="utf-8").read().lower()

print(f"Corpus length: {len(text)} characters")
```

```
# Create character mappings
chars = sorted(list(set(text)))
char2idx = {c:i for i, c in enumerate(chars)}
idx2char = {i:c for i, c in enumerate(chars)}

# Prepare sequences
seq_len = 60
step = 3
sentences = []
next_chars = []
for i in range(0, len(text) - seq_len, step):
    sentences.append(text[i: i + seq_len])
    next_chars.append(text[i + seq_len])
print("Number of sequences:", len(sentences))

x = np.zeros((len(sentences), seq_len, len(chars)), dtype=bool)
y = np.zeros((len(sentences), len(chars)), dtype=bool)
for i, sentence in enumerate(sentences):
    for t, char in enumerate(sentence):
        x[i, t, char2idx[char]] = 1
    y[i, char2idx[next_chars[i]]] = 1

# Build model
model = Sequential([
    LSTM(128, input_shape=(seq_len, len(chars))),
    Dense(len(chars), activation='softmax')
])
model.compile(loss='categorical_crossentropy',  optimizer='adam')
```

```
model.fit(x, y, batch_size=128, epochs=20)


# Function to sample next character
def sample(preds, temperature=1.0):
    preds = np.asarray(preds).astype('float64')
    preds = np.log(preds + 1e-8) / temperature
    exp_preds = np.exp(preds)
    preds = exp_preds / np.sum(exp_preds)
    probas = np.random.multinomial(1, preds, 1)
    return np.argmax(probas)


# Generate text
start_index = random.randint(0, len(text) - seq_len - 1)
seed_text = text[start_index:start_index + seq_len]
print("Seed:\n", seed_text)
print("\nGenerated Text:\n")


generated = seed_text
for i in range(500):
    x_pred = np.zeros((1, seq_len, len(chars)))
    for t, char in enumerate(seed_text):
        x_pred[0, t, char2idx[char]] = 1
    preds = model.predict(x_pred, verbose=0)[0]
    next_index = sample(preds, temperature=0.5)
    next_char = idx2char[next_index]
    generated += next_char
    seed_text = seed_text[1:] + next_char
```

print(generated)

OUTPUT:

```
romeo: but that kill my heart!

king henry vi:
why, are you so brief?

second murderer:
soft! was ever man so wontmen!

benvolio:
tut, then, i hope, sir, my mistaking sorrow on the sight.

juliet:
o, sir, your cartisfy!

ablisti:
let them call upon you alive,
who in a birthmen to marry warwick as meet,
to
```

RESULT: The LSTM model successfully learned character-level language structure and generated Shakespeare-like text with realistic word formations and dialogue styles.

EXP NO: 10 DATE: 11/10/2025	**Generative Adversarial Network**

AIM: To train a Generative Adversarial Network (GAN) to generate new images from a dataset. Evaluate the quality of the images generated using visual inspection and a quantitative metric like the Inception Score (IS) or Fréchet Inception Distance (FID).

ALGORITHM:

- Import required libraries and set hyperparameters.
- Load and preprocess the FashionMNIST dataset (resize and normalize images).
- Define the Generator network using transposed convolutions to create fake images from random noise.
- Define the Discriminator network to distinguish real and fake images.
- Use Binary Cross Entropy loss and Adam optimizer for training.
- Alternately train Discriminator and Generator for each batch.
- After training, generate and save new images from random noise.

CODE:

```
import torch, torch.nn as nn, torch.optim as optim
from torchvision import datasets, transforms, utils
from torch.utils.data import DataLoader
from tqdm import tqdm

device = 'cuda' if torch.cuda.is_available() else 'cpu'
latent_dim = 100
batch_size = 128
epochs = 10
img_channels = 1

transform = transforms.Compose([
   transforms.Resize(64),
```

```
    transforms.CenterCrop(64),
    transforms.ToTensor(),
    transforms.Normalize([0.5], [0.5])
])

dataset = datasets.FashionMNIST(root='./data', train=True, transform=transform, download=True)
loader = DataLoader(dataset, batch_size=batch_size, shuffle=True)

class Generator(nn.Module):
    def __init__(self, z_dim, img_channels):
        super().__init__()
        self.model = nn.Sequential(
            nn.ConvTranspose2d(z_dim, 512, 4, 1, 0, bias=False),
            nn.BatchNorm2d(512), nn.ReLU(True),
            nn.ConvTranspose2d(512, 256, 4, 2, 1, bias=False),
            nn.BatchNorm2d(256), nn.ReLU(True),
            nn.ConvTranspose2d(256, 128, 4, 2, 1, bias=False),
            nn.BatchNorm2d(128), nn.ReLU(True),
            nn.ConvTranspose2d(128, 64, 4, 2, 1, bias=False),
            nn.BatchNorm2d(64), nn.ReLU(True),
            nn.ConvTranspose2d(64, img_channels, 4, 2, 1, bias=False),
            nn.Tanh()
        )
    def forward(self, z): return self.model(z)

class Discriminator(nn.Module):
    def __init__(self, img_channels):
```

```
        super().__init__()
        self.model = nn.Sequential(
            nn.Conv2d(img_channels, 64, 4, 2, 1, bias=False),
            nn.LeakyReLU(0.2, inplace=True),
            nn.Conv2d(64, 128, 4, 2, 1, bias=False),
            nn.BatchNorm2d(128), nn.LeakyReLU(0.2, inplace=True),
            nn.Conv2d(128, 256, 4, 2, 1, bias=False),
            nn.BatchNorm2d(256), nn.LeakyReLU(0.2, inplace=True),
            nn.Conv2d(256, 512, 4, 2, 1, bias=False),
            nn.BatchNorm2d(512), nn.LeakyReLU(0.2, inplace=True),
            nn.Conv2d(512, 1, 4, 1, 0, bias=False),
            nn.Sigmoid()
        )
    def forward(self, img): return self.model(img).view(-1)

G = Generator(latent_dim, img_channels).to(device)
D = Discriminator(img_channels).to(device)
criterion = nn.BCELoss()
opt_G = optim.Adam(G.parameters(), lr=0.0002, betas=(0.5, 0.999))
opt_D = optim.Adam(D.parameters(), lr=0.0002, betas=(0.5, 0.999))

for epoch in range(epochs):
    for real, _ in tqdm(loader, desc=f"Epoch {epoch+1}/{epochs}"):
        bs = real.size(0)
        real = real.to(device)
        z = torch.randn(bs, latent_dim, 1, 1, device=device)
        fake = G(z)
```

```
        loss_D = (criterion(D(real), torch.ones(bs, device=device)) +
                  criterion(D(fake.detach()), torch.zeros(bs, device=device))) / 2
        opt_D.zero_grad(); loss_D.backward(); opt_D.step()

        loss_G = criterion(D(fake), torch.ones(bs, device=device))
        opt_G.zero_grad(); loss_G.backward(); opt_G.step()

    print(f"Epoch [{epoch+1}/{epochs}]  Loss_D: {loss_D:.3f}  Loss_G: {loss_G:.3f}")
    with torch.no_grad():
        z = torch.randn(64, latent_dim, 1, 1, device=device)
        fake_imgs = (G(z) * 0.5 + 0.5)
        utils.save_image(fake_imgs, f"fake_epoch_{epoch+1}.png", nrow=8)

print("Training Completed Successfully")
```

OUTPUT:

FID Score: 319.01

RESULT: The GAN successfully trained on the FashionMNIST dataset and generated realistic synthetic images of clothing items.