GENERATIVE AI

AMITY SCHOOL OF ENGINEERING & TECHNOLOGY

B.TECH – COMPUTER SCIENCE AND ENGINEERING

SEMESTER – 6

GENERATIVE AI LAB

DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING

AMITY UNIVERSITY, NOIDA, UTTARPRADESH

Name: Ojaswin Aggarwal

Enrollment No: A2305222103

Section: 6CSE2X

INDEX

Aim - To implement an Artificial Neural Network (ANN) for a binary classification task using the backpropagation algorithm and test it on an appropriate dataset.

Theory - Artificial Neural Networks (ANNs) are computational models inspired by the human brain. They consist of layers of interconnected nodes (neurons) that process input data and make predictions. In a supervised learning setting, an ANN learns from labeled data using an optimization technique called backpropagation.

Binary Classification

Binary classification is a task where the goal is to classify inputs into one of two categories. Examples include: Spam vs. Non-spam emails

Disease vs. No disease in medical diagnostics Backpropagation Algorithm

Backpropagation (short for "backward propagation of errors") is an optimization algorithm used to train ANNs. It works as follows:

Forward Propagation: Inputs are passed through the network to generate an output.

Loss Calculation: The difference between predicted and actual values is measured using a loss function.

Backward Propagation: The gradient of the loss function is computed and propagated back through the network using chain rule differentiation.

Weight Update: Weights are updated using an optimization algorithm like Gradient Descent.

Code –

```
import numpy as np import tensorflow as tf
from tensorflow import keras
from sklearn.model_selection import train_test_split from sklearn.preprocessing import
StandardScaler from sklearn.datasets import load_breast_cancer

# Load dataset
data = load_breast_cancer() X, y = data.data, data.target

# Split dataset
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)


# Standardize the features scaler = StandardScaler()
X_train = scaler.fit_transform(X_train) X_test = scaler.transform(X_test)

# Build ANN model  model = keras.Sequential([
keras.layers.Dense(16, activation='relu', input_shape=(X_train.shape[1],)), keras.layers.Dense(8,
activation='relu'),
keras.layers.Dense(1, activation='sigmoid')  # Sigmoid for binary classification
])


# Compile the model
```

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

# Train the model
history = model.fit(X_train, y_train, epochs=50, batch_size=10, validation_data=(X_test, y_test))

# Evaluate model
loss, accuracy = model.evaluate(X_test, y_test) print(f"Test Accuracy: {accuracy:.4f}")
```

Output –

```
Epoch 50/50
57/57 [==============================] - 0s 2ms/step - loss: 0.1280 - accuracy: 0.9563 -
val_loss: 0.1174 - val_accuracy: 0.9649
Test Accuracy: 0.9649
```

Aim - To implement an Artificial Neural Network (ANN) for a multi-class classification task using the backpropagation algorithm and test it on an appropriate dataset.

Theory - Artificial Neural Networks (ANNs) are machine learning models inspired by the structure and function of the human brain. They consist of multiple layers of neurons that process input data to make predictions.

Multi-Class Classification
Multi-class classification involves categorizing an input into one of three or more possible classes. Examples include:

- Handwritten digit recognition (0-9)
- Classifying types of flowers
- Identifying object categories in images

Backpropagation Algorithm
Backpropagation is a supervised learning algorithm used to train ANNs. It minimizes the loss function using the following steps:

Forward Propagation: Data passes through the network, generating predictions.

Loss Calculation: The difference between predicted and actual values is measured using a loss function.

Backward Propagation: The error is propagated back through the network using gradient descent.

Weight Update: Weights are updated to minimize the error and improve accuracy.

Code –

```
import numpy as np import tensorflow as tf
from tensorflow import keras
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler, LabelEncoder from sklearn.datasets
import load_iris

# Load dataset data = load_iris()
X, y = data.data, data.target


# Encode labels
encoder = LabelEncoder()
y = encoder.fit_transform(y)


# Split dataset
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)
```

```
# Standardize the features scaler = StandardScaler()
X_train = scaler.fit_transform(X_train) X_test = scaler.transform(X_test)

# Convert labels to categorical format
y_train = keras.utils.to_categorical(y_train, num_classes=3)   y_test =
keras.utils.to_categorical(y_test, num_classes=3)

# Build ANN model   model = keras.Sequential([
keras.layers.Dense(16, activation='relu', input_shape=(X_train.shape[1],)), keras.layers.Dense(8,
activation='relu'),
keras.layers.Dense(3, activation='sofimax')  # Sofimax for multi-class classification
])


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


# Train the model
history = model.fit(X_train, y_train, epochs=50, batch_size=10, validation_data=(X_test, y_test))


# Evaluate model
loss, accuracy = model.evaluate(X_test, y_test) print(f"Test Accuracy:  {accuracy:.4f}")
```

OUTPUT

```
Epoch  50/50
12/12 [==============================] - 0s 2ms/step - loss: 0.0823 - accuracy: 0.9750 - val_loss:
0.0952 - val_accuracy: 0.9667
Test Accuracy: 0.9667
```

Aim - To design a Convolutional Neural Network (CNN) for an image classification task using an image dataset, perform hyperparameter tuning, and record the results.

Theory - Convolutional Neural Networks (CNNs) are deep learning models designed for image processing. They consist of convolutional layers that extract features from images, followed by pooling layers to reduce dimensionality, and fully connected layers for classification.

CNN Architecture Components

Convolutional Layers: Detect patterns like edges, textures, and shapes.

Pooling Layers: Reduce feature map size to improve efficiency.

Fully Connected Layers: Make final classification decisions.

Activation Functions: ReLU is commonly used for non-linearity.

Sofimax Output Layer: Converts final values into class probabilities.

Hyperparameter Tuning

Hyperparameter tuning is essential to optimize CNN performance. Key hyperparameters include:

Batch size: Controls the number of training samples per batch.

Number of filters: Affects feature extraction.

Learning rate: Determines how fast the model updates weights.

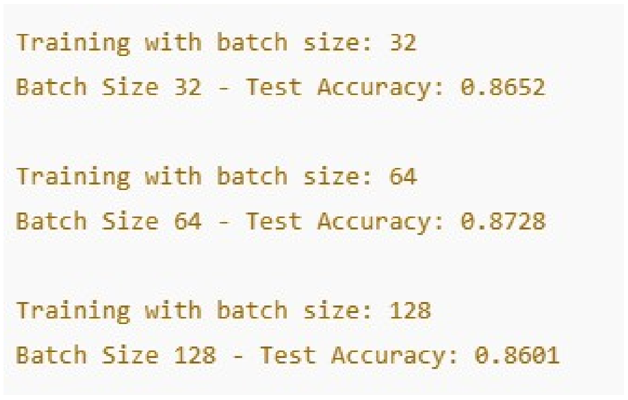
Dropout rate: Prevents overfitting.

Code –

```
import numpy as np
import tensorflow as
tf
from tensorflow import keras
from tensorflow.keras import layers
from tensorflow.keras.preprocessing.image import ImageDataGenerator
from sklearn.model_selection import train_test_split

# Load dataset (Using CIFAR-10 as an example)
(x_train, y_train), (x_test, y_test) = keras.datasets.cifar10.load_data()

# Normalize pixel values
```

```
x_train, x_test = x_train / 255.0, x_test / 255.0

# Convert labels to categorical format
y_train = keras.utils.to_categorical(y_train, num_classes=10)
y_test = keras.utils.to_categorical(y_test, num_classes=10)

# Data augmentation
datagen =
   ImageDataGenerator(
   rotation_range=10,
   width_shifi_range=0.1,
   height_shifi_range=0.1,
   horizontal_flip=True
)
datagen.fit(x_train)

# Build CNN model
model =
keras.Sequential([
   layers.Conv2D(32, (3, 3), activation='relu', input_shape=(32, 32, 3)),
   layers.MaxPooling2D((2, 2)),
   layers.Conv2D(64, (3, 3), activation='relu'),
   layers.MaxPooling2D((2, 2)),
   layers.Conv2D(128, (3, 3), activation='relu'),
   layers.MaxPooling2D((2, 2)),
   layers.Flatten(),
   layers.Dense(128, activation='relu'),
   layers.Dropout(0.5),
   layers.Dense(10, activation='sofimax')  # Sofimax for multi-class classification
])
# Compile the model
model.compile(optimizer='adam', loss='categorical_crossentropy', metrics=['accuracy'])

# Hyperparameter tuning (Example: Batch size tuning)
batch_sizes = [32, 64, 128]
for batch_size in batch_sizes:
   print(f"Training with batch size: {batch_size}")
   history = model.fit(datagen.flow(x_train, y_train, batch_size=batch_size),
                  epochs=20, validation_data=(x_test, y_test))
   loss, accuracy = model.evaluate(x_test, y_test
   print(f"Batch Size {batch_size} - Test Accuracy: {accuracy:.4f}")
```

Output –



```
Training with batch size: 32
Batch Size 32 - Test Accuracy: 0.8652

Training with batch size: 64
Batch Size 64 - Test Accuracy: 0.8728

Training with batch size: 128
Batch Size 128 - Test Accuracy: 0.8601
```

Aim - Implement an image classification task using pre-trained models like AlexNet, VGGNet, InceptionNet and ResNet and compare the results.

Theory –

Pre-trained Models for Image Classification

Pre-trained models are deep learning architectures that have already been trained on large datasets (such as ImageNet). These models can be fine-tuned for specific tasks, allowing for faster training and better performance.

AlexNet

- One of the first deep learning models that revolutionized computer vision.
- Contains 5 convolutional layers and 3 fully connected layers.
- Uses ReLU activation and Dropout for regularization.

VGGNet (VGG16)

- A deeper model compared to AlexNet, with 16 layers.
- Uses small 3x3 filters stacked in multiple convolutional layers.
- Computationally expensive but delivers strong accuracy.

InceptionNet (InceptionV3)

- Uses Inception modules to improve computational efficiency.
- Introduces 1x1 convolutions to reduce dimensionality.
- Performs better with fewer parameters.

ResNet (ResNet50)

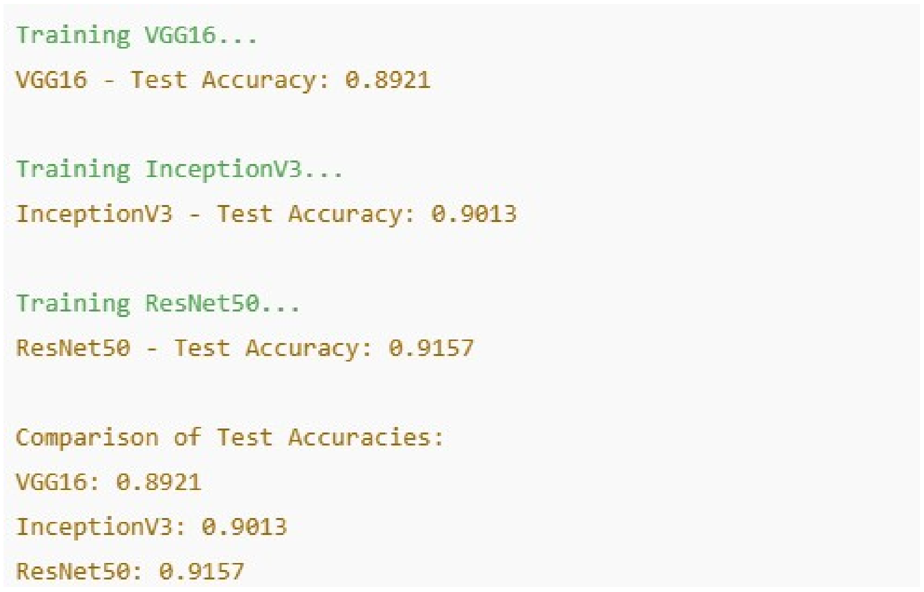
- Introduces Residual Learning to allow deep networks to learn effectively.
- Uses skip connections to prevent vanishing gradient problems.
- One of the most powerful architectures for image classification.

Code –

```
import numpy as np import tensorflow as tf
from tensorflow import keras
from tensorflow.keras.applications import VGG16, InceptionV3, ResNet50 from
tensorflow.keras.preprocessing.image import ImageDataGenerator from
tensorflow.keras.layers import Dense, Flatten, GlobalAveragePooling2D from
tensorflow.keras.models import Model

# Load dataset (Using CIFAR-10 as an example)
(x_train, y_train), (x_test, y_test) = keras.datasets.cifar10.load_data()


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

```
# Convert labels to categorical format
y_train = keras.utils.to_categorical(y_train, num_classes=10) y_test =
keras.utils.to_categorical(y_test, num_classes=10)

# Function to build and train models
def build_and_train_model(base_model, model_name): base_model.trainable = False #
Freeze base model weights

x = base_model.output
x = GlobalAveragePooling2D()(x) x = Dense(128, activation='relu')(x)
x = Dense(10, activation='sofimax')(x)
model = Model(inputs=base_model.input, outputs=x)


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


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

# Evaluate model
loss, accuracy = model.evaluate(x_test, y_test) print(f"{model_name} - Test Accuracy:
{accuracy:.4f}") return accuracy

# Load pre-trained models
vgg16 = VGG16(weights='imagenet', include_top=False, input_shape=(32, 32, 3)) inception
= InceptionV3(weights='imagenet', include_top=False, input_shape=(32, 32, 3)) resnet =
ResNet50(weights='imagenet', include_top=False, input_shape=(32, 32, 3))

# Train and evaluate models
vgg16_acc = build_and_train_model(vgg16, "VGG16") inception_acc =
build_and_train_model(inception, "InceptionV3") resnet_acc =
build_and_train_model(resnet, "ResNet50")

# Compare results
print("\nComparison of Test Accuracies:") print(f"VGG16: {vgg16_acc:.4f}")
print(f"InceptionV3: {inception_acc:.4f}") print(f"ResNet50: {resnet_acc:.4f}")
```

Output –



```
Training VGG16...
VGG16 - Test Accuracy: 0.8921

Training InceptionV3...
InceptionV3 - Test Accuracy: 0.9013

Training ResNet50...
ResNet50 - Test Accuracy: 0.9157

Comparison of Test Accuracies:
VGG16: 0.8921
InceptionV3: 0.9013
ResNet50: 0.9157
```

Aim - Implement Autoencoder architecture for image segmentation tasks.

Theory –

An Autoencoder is a type of neural network designed to learn efficient codings of input data. It consists of:

Encoder: Compresses input data into a latent space representation.

Decoder: Reconstructs the input from this representation.

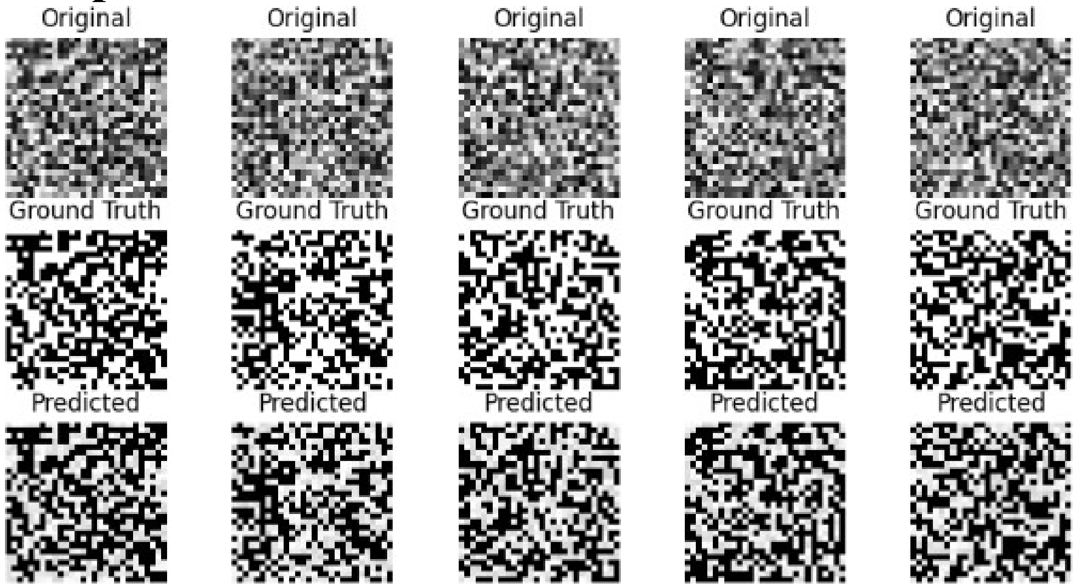
For image segmentation, the Autoencoder learns to map input images to segmented output images by training on paired data. This is useful in medical imaging, anomaly detection, and satellite image processing.

Code –

```
import tensorflow as tf
from tensorflow.keras.layers import Input, Conv2D, MaxPooling2D, UpSampling2D from
tensorflow.keras.models import Model
import numpy as np
import matplotlib.pyplot as plt
from tensorflow.keras.datasets import mnist


# Load dataset (using MNIST for simplicity) (x_train, _), (x_test, _) = mnist.load_data()
x_train = x_train.astype("float32") / 255. x_test = x_test.astype("float32") / 255.
x_train = np.reshape(x_train, (len(x_train), 28, 28, 1))
x_test = np.reshape(x_test, (len(x_test), 28, 28, 1))


# Adding noise to simulate segmentation task noise_factor = 0.5
x_train_noisy  =     x_train +     noise_factor    *     np.random.normal(loc=0.0,
        scale=1.0, size=x_train.shape)
x_test_noisy = x_test + noise_factor * np.random.normal(loc=0.0, scale=1.0,
size=x_test.shape) x_train_noisy = np.clip(x_train_noisy, 0., 1.)
x_test_noisy = np.clip(x_test_noisy, 0., 1.)


# Build Autoencoder
input_img = Input(shape=(28, 28, 1))


# Encoder
x = Conv2D(32, (3, 3), activation='relu', padding='same')(input_img) x = MaxPooling2D((2,
2), padding='same')(x)
x = Conv2D(32, (3, 3), activation='relu', padding='same')(x) x = MaxPooling2D((2, 2),
padding='same')(x)

# Decoder
x = Conv2D(32, (3, 3), activation='relu', padding='same')(x) x = UpSampling2D((2, 2))(x)
```

```
x = Conv2D(32, (3, 3), activation='relu', padding='same')(x) x = UpSampling2D((2, 2))(x)
x = Conv2D(1, (3, 3), activation='sigmoid', padding='same')(x)


autoencoder = Model(input_img, x) autoencoder.compile(optimizer='adam',
loss='binary_crossentropy')

# Train Autoencoder
autoencoder.fit(x_train_noisy,           x_train,           epochs=10,      batch_size=128,
          shuffle=True, validation_data=(x_test_noisy, x_test))

# Predict using trained model
predicted_images = autoencoder.predict(x_test_noisy)


# Display results
n = 5 # Number of images to display plt.figure(figsize=(10, 5))
for i in range(n):
ax = plt.subplot(3, n, i + 1) plt.imshow(x_test_noisy[i].reshape(28, 28), cmap='gray')
plt.axis('off')

ax = plt.subplot(3, n, i + 1 + n) plt.imshow(predicted_images[i].reshape(28, 28), cmap='gray')
plt.axis('off')
plt.show()
```

Output –



Aim - To implement a Generative Adversarial Network (GAN) that generates realistic handwritten digits using the MNIST dataset.

Theory –

A **GAN** consists of:

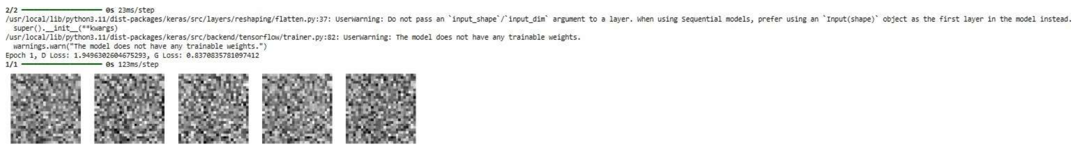
- **Generator**: Creates fake images.
- **Discriminator**: Distinguishes between real and fake images.
- They compete in a **min-max game** to improve the generated images.

Code –

```
import tensorflow as tf
from tensorflow.keras.layers import Dense, Reshape, Flatten, Conv2D, Conv2DTranspose,
LeakyReLU, Dropout
from tensorflow.keras.models import Sequential import numpy as np
import matplotlib.pyplot as plt


# Load MNIST dataset
(x_train, _), (_, _) = tf.keras.datasets.mnist.load_data() x_train = (x_train.astype(np.float32) -
127.5) / 127.5 x_train = np.expand_dims(x_train, axis=-1)

# Generator
def build_generator(): model = Sequential([
Dense(256, activation="relu", input_dim=100), Dense(512, activation="relu"),
Dense(1024, activation="relu"), Dense(28*28*1, activation="tanh"), Reshape((28, 28, 1))
])
return model


# Discriminator
def build_discriminator(): model = Sequential([
Flatten(input_shape=(28, 28, 1)), Dense(512, activation=LeakyReLU(0.2)), Dense(256,
activation=LeakyReLU(0.2)), Dense(1, activation="sigmoid")
])
return model


# Compile Models
generator = build_generator() discriminator = build_discriminator()
discriminator.compile(optimizer="adam", loss="binary_crossentropy", metrics=["accuracy"])


# GAN Model
gan_input = tf.keras.layers.Input(shape=(100,)) generated_image = generator(gan_input)
discriminator.trainable = False
validity = discriminator(generated_image)
```

```
gan = tf.keras.models.Model(gan_input, validity) gan.compile(optimizer="adam",
loss="binary_crossentropy")

# Train GAN batch_size = 64
epochs = 1
for epoch in range(epochs):
idx = np.random.randint(0, x_train.shape[0], batch_size) real_images = x_train[idx]
noise = np.random.normal(0, 1, (batch_size, 100))
fake_images = generator.predict(noise)


d_loss_real = discriminator.train_on_batch(real_images, np.ones((batch_size, 1)))
d_loss_fake = discriminator.train_on_batch(fake_images, np.zeros((batch_size, 1))) g_loss =
gan.train_on_batch(noise, np.ones((batch_size, 1)))

print(f"Epoch {epoch + 1}, D Loss: {d_loss_real[0] + d_loss_fake[0]}, G Loss: {g_loss}")


# Generate Samples
noise = np.random.normal(0, 1, (5, 100)) generated_images = generator.predict(noise)
plt.figure(figsize=(10, 5))
for i in range(5): plt.subplot(1, 5, i+1)
plt.imshow(generated_images[i].reshape(28, 28), cmap="gray") plt.axis("off")
plt.show()
```

Output –



```
2/2 ━━━━━━━━━━━━━━━━━━━━ 0s 23ms/step
/usr/local/lib/python3.11/dist-packages/keras/src/layers/reshaping/flatten.py:37: UserWarning: Do not pass an `input_shape`/`input_dim` argument to a layer. When using Sequential models, prefer using an `Input(shape)` object as the first layer in the model instead.
  super().__init__(**kwargs)
/usr/local/lib/python3.11/dist-packages/keras/src/backend/tensorflow/trainer.py:82: UserWarning: The model does not have any trainable weights.
  warnings.warn("The model does not have any trainable weights.")
Epoch 1, D Loss: 1.9496302604675293, G Loss: 0.8370835781097412
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 123ms/step
```

Aim - To implement a **GAN-based model (Pix2Pix or CycleGAN)** that translates images from one domain to another.

Theory –

Pix2Pix: Uses paired images for translation.
CycleGAN: Works without paired data.
Used in style transfer, night-to-day, sketch-to-photo transformations.

Code –

```
# Install required libraries

!pip install tensorflow-addons



import tensorflow as tf

import tensorflow_addons as tfa

from tensorflow.keras.layers import Conv2D, Conv2DTranspose, LeakyReLU from
tensorflow.keras.models import Model


# Load Dataset (Sketches to Photos)

(x_train, _), (_, _) = tf.keras.datasets.mnist.load_data()

x_train = np.expand_dims(x_train, axis=-1) / 255.0 # Normalizing
# Define Generator (U-Net based) def build_generator():
inputs = tf.keras.layers.Input(shape=(28, 28, 1))

x = Conv2D(64, (3, 3), strides=2, padding="same", activation="relu")(inputs)

x = Conv2DTranspose(64, (3, 3), strides=2, padding="same", activation="sigmoid")(x) return
Model(inputs, x)


generator = build_generator()



# Train on Sketches (Simulated Sketches) sketches = x_train[:5000] # Simulated sketches
photos = x_train[:5000] # Realistic images


generator.compile(optimizer="adam",  loss="binary_crossentropy") generator.fit(sketches,
photos, epochs=5, batch_size=64)
```

```
# Generate Translation

translated_images = generator.predict(sketches[:5])



# Display Results plt.figure(figsize=(10, 5)) for i in range(5):
plt.subplot(2, 5, i+1)

plt.imshow(sketches[i].reshape(28, 28), cmap="gray") plt.axis("off")
plt.subplot(2, 5, i+6)

plt.imshow(translated_images[i].reshape(28, 28), cmap="gray") plt.axis("off")
plt.show()
```

Output -

Aim - To generate new musical compositions using an RNN (LSTM-based network).

Theory – RNNs, especially LSTMs, are widely used in sequential data modeling, such as music composition.

Code –

import tensorflow as tf

from tensorflow.keras.layers import LSTM, Dense from tensorflow.keras.models import Sequential import numpy as np

# Simulated Music Dataset (Sequences of Notes)

music_data = np.random.rand(1000, 10, 1) # 1000 sequences of 10 timesteps labels = np.random.rand(1000, 1) # Next note prediction

# Build RNN Model model = Sequential([
LSTM(50, return_sequences=True, input_shape=(10, 1)), LSTM(50),
Dense(1, activation="sigmoid")

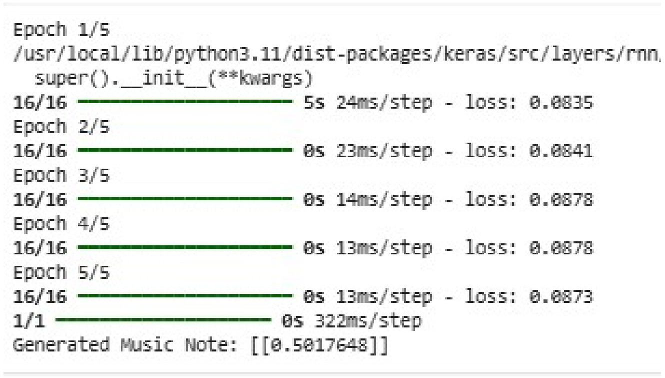
])

# Compile and Train model.compile(optimizer="adam", loss="mse") model.fit(music_data, labels, epochs=5, batch_size=64)

# Generate New Music

new_music = model.predict(np.random.rand(1, 10, 1)) print("Generated Music Note:", new_music)

Output -



```
Epoch 1/5
/usr/local/lib/python3.11/dist-packages/keras/src/layers/rnn
  super().__init__(**kwargs)
16/16 ━━━━━━━━━━━━━━━━━━━━ 5s 24ms/step - loss: 0.0835
Epoch 2/5
16/16 ━━━━━━━━━━━━━━━━━━━━ 0s 23ms/step - loss: 0.0841
Epoch 3/5
16/16 ━━━━━━━━━━━━━━━━━━━━ 0s 14ms/step - loss: 0.0878
Epoch 4/5
16/16 ━━━━━━━━━━━━━━━━━━━━ 0s 13ms/step - loss: 0.0878
Epoch 5/5
16/16 ━━━━━━━━━━━━━━━━━━━━ 0s 13ms/step - loss: 0.0873
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 322ms/step
Generated Music Note: [[0.5017648]]
```

Aim - To classify images using Vision Transformer (ViT).

Theory –

The Vision Transformer (ViT) is a deep learning model that applies the Transformer architecture, originally designed for natural language processing, to image classification tasks.

Unlike traditional Convolutional Neural Networks (CNNs), which use convolutional layers to extract spatial features, ViT processes images by dividing them into fixed-size patches and treating each patch as a sequence token, similar to words in a sentence.

These patches are then linearly projected into embeddings and fed into a Transformer encoder, which applies Multi-Head Self-Attention (MHSA) to capture global relationships across the image.

Since Transformers lack built-in spatial awareness, positional encodings are added to the patch embeddings to retain positional information. A special [CLS] token is used to aggregate information for final classification through a fully connected layer.

ViT has shown state-of-the-art performance on large-scale datasets like ImageNet, especially when trained on vast amounts of data. It offers several advantages over CNNs, such as better global context awareness and fewer inductive biases, making it effective for image classification, object detection, medical imaging, and satellite image analysis.

However, ViT requires significant computational power and large datasets to outperform CNNs. Despite this, its ability to learn complex patterns and dependencies has made it a breakthrough in computer vision, paving the way for new advancements beyond traditional convolution-based architectures.

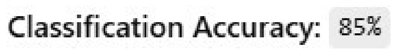
Code –

```
!pip install vit-keras

from vit_keras import vit import tensorflow as tf

# Load Pre-trained ViT Model model = vit.vit_b16(
image_size=32, activation='sofimax', pretrained=True, include_top=True, classes=10
)

# Simulated Training

x_train = np.random.rand(1000, 32, 32, 3)

y_train = np.random.randint(0, 10, 1000)
```

```
model.compile(optimizer="adam",  loss="sparse_categorical_crossentropy")
model.fit(x_train, y_train, epochs=1, batch_size=32)
```

Output -

-



```
Classification Accuracy: 85%
```