GENERATIVE AI LAB MANUAL

SUBJECT CODE : AD2602

CHENNAI INSTITUTE OF TECHNOLOGY
(Autonomous)

SL.No	Program	Date
1	Image Generation with Generative Adversarial Networks (GANs)	
2	Text Generation with Recurrent Neural Networks (RNNs)	
3	Music Generation with Variational Autoencoders (VAEs)	
4	Style Transfer with Neural Style Transfer Algorithms	
5	Data Augmentation with Generative Models	
6	Video Generation with Generative Adversarial Networks (GANs)	
7	Anomaly Detection with Generative Models	
8	Domain Adaptation with Generative Adversarial Networks (GANs)	

PROGRAM :O1

Image Generation with Generative Adversarial Networks (GANs)

AIM

To implement domain adaptation using Generative Adversarial Networks (GANs), enabling knowledge transfer from a labeled source domain to an unlabeled target domain and evaluating the performance on target domain tasks.

ALGORITHM

1. Load the dataset (e.g., CIFAR-10, CelebA).
2. Define two networks: a Generator and a Discriminator.
3. Takes random noise as input and generates a synthetic image.This network consists of several layers like dense, convolutional, and transposed convolutional layers to upsample the input noise into an image.
4. Takes an image (real or generated) as input and outputs a probability indicating whether the image is real or fake.The discriminator consists of convolutional layers to classify the input image.
5. The Generator is trained to generate images that can deceive the Discriminator into classifying them as real.The Discriminator is trained to distinguish between real and fake images.The objective is for the Generator to improve over time to generate more realistic images while the Discriminator gets better at distinguishing them.
6. During each iteration, feed real and fake images into the Discriminator.
7. Update the Generator and Discriminator based on their losses (using a loss function like Binary Cross-Entropy).
8. Use an optimizer (e.g., Adam optimizer) to minimize the loss.
9. Experiment with different architectures like DCGAN (Deep Convolutional GAN) or WGAN (Wasserstein GAN) and tune hyperparameters like learning rates, batch sizes, and network layers.
10. Generate images from the trained model and evaluate the image quality using metrics like Inception Score (IS) or Fréchet Inception Distance (FID).
11. Observe convergence speed by tracking the loss of both networks over epochs.

PROGRAM

```
import torch

import torch.nn as nn

import torch.optim as optim

import torchvision

import torchvision.transforms as transforms

from torch.utils.data import DataLoader
```

```
# Hyperparameters
batch_size = 128
epochs = 100
lr = 0.0002
beta1 = 0.5
latent_dim = 100

# Load dataset
transform = transforms.Compose([
    transforms.ToTensor(),
    transforms.Normalize(mean=[0.5, 0.5, 0.5], std=[0.5, 0.5, 0.5])
])

train_data = torchvision.datasets.CIFAR10(root='./data', train=True, download=True,
transform=transform)
train_loader = DataLoader(train_data, batch_size=batch_size, shuffle=True)

# Generator model
class Generator(nn.Module):
    def __init__(self):
        super(Generator, self).__init__()
        self.model = nn.Sequential(
            nn.Dense(256, input_dim=latent_dim),
            nn.ReLU(),
            nn.Dense(512),
            nn.ReLU(),
            nn.Dense(1024),
            nn.ReLU(),
            nn.Dense(3, activation='tanh')  # Output channels 3 for RGB image
        )
```

```
    def forward(self, z):
        return self.model(z)

# Discriminator model
class Discriminator(nn.Module):
    def __init__(self):
        super(Discriminator, self).__init__()
        self.model = nn.Sequential(
            nn.Conv2d(3, 64, kernel_size=4, stride=2, padding=1),
            nn.LeakyReLU(0.2),
            nn.Conv2d(64, 128, kernel_size=4, stride=2, padding=1),
            nn.LeakyReLU(0.2),
            nn.Conv2d(128, 256, kernel_size=4, stride=2, padding=1),
            nn.LeakyReLU(0.2),
            nn.Flatten(),
            nn.Linear(256*8*8, 1),
            nn.Sigmoid()
        )

    def forward(self, img):
        return self.model(img)

# Initialize models
generator = Generator()
discriminator = Discriminator()

# Loss function
adversarial_loss = nn.BCELoss()
```

```
# Optimizers
optimizer_g = optim.Adam(generator.parameters(), lr=lr, betas=(beta1, 0.999))
optimizer_d = optim.Adam(discriminator.parameters(), lr=lr, betas=(beta1, 0.999))

# Training loop
for epoch in range(epochs):
  for i, (imgs, _) in enumerate(train_loader):
    real_imgs = imgs
    batch_size = real_imgs.size(0)

    # Create labels
    valid = torch.ones(batch_size, 1)
    fake = torch.zeros(batch_size, 1)

    # Train Discriminator
    optimizer_d.zero_grad()

    # Real images
    real_loss = adversarial_loss(discriminator(real_imgs), valid)

    # Fake images
    z = torch.randn(batch_size, latent_dim)
    fake_imgs = generator(z)
    fake_loss = adversarial_loss(discriminator(fake_imgs.detach()), fake)

    # Total discriminator loss
    d_loss = (real_loss + fake_loss) / 2
    d_loss.backward()
    optimizer_d.step()
```

```
        # Train Generator
        optimizer_g.zero_grad()
        g_loss = adversarial_loss(discriminator(fake_imgs), valid)
        g_loss.backward()
        optimizer_g.step()

    print(f"Epoch {epoch+1}/{epochs}, D Loss: {d_loss.item()}, G Loss: {g_loss.item()}")
```

PROGRAM NO : 02

Text Generation with Recurrent Neural Networks (RNNs)

AIM

To implement a Recurrent Neural Network (RNN) for text generation using frameworks like TensorFlow or PyTorch. The model will be trained on a large corpus of text data, such as Shakespearean texts or Wikipedia articles, and will explore different RNN architectures (e.g., vanilla RNN, LSTM, GRU) and training techniques (e.g., teacher forcing, beam search) to improve the quality and coherence of the generated text.

ALGORITHM

Data Preprocessing:

1. Load a large corpus of text data (e.g., Shakespearean texts, Wikipedia).
2. Tokenize the text into characters or words.
3. Convert the text into numerical representations (e.g., one-hot encoding or integer encoding).
4. Create input sequences and target sequences for training. For character-level generation, the input sequence could be a sliding window of characters, and the target sequence would be the following character.

RNN Architecture:

5. Define the RNN model. This can be a simple vanilla RNN, or more advanced models like LSTM (Long Short-Term Memory) or GRU (Gated Recurrent Unit).
6. The model should consist of an embedding layer (if working with words), one or more recurrent layers, and a dense output layer that predicts the next character or word.

Loss Function and Optimization:

7. Use a suitable loss function, such as categorical cross-entropy, to measure the difference between the predicted output and the actual target sequence.
8. Use an optimizer like Adam or SGD to update the weights of the network.

Training the Model:

9. Train the model using the training dataset (character or word sequences) and backpropagate the errors to update the network weights.
10. Implement techniques like teacher forcing, where the true output at each timestep is fed as the next input during training, and beam search for generating more coherent text during inference.

Text Generation:

11. After training the model, generate text by providing a seed input (e.g., an initial character or word).

12. Use the trained RNN to predict the next character/word, and feed the prediction back as the input for the next step.
13. Implement temperature sampling to control the randomness of the generated text (higher temperature for more randomness, lower temperature for more predictable text).

Evaluation:

14. Evaluate the generated text qualitatively for coherence and fluency.Optionally, compute metrics like perplexity to assess the performance of the model.

PROGRAM

```
import torch

import torch.nn as nn

import torch.optim as optim

import numpy as np

import random

import string


# Hyperparameters

batch_size = 128

epochs = 50

seq_length = 100

hidden_size = 256

lr = 0.002

temperature = 0.8


# Load dataset

with open('shakespeare.txt', 'r') as f:

    text = f.read().lower()


# Create mappings for characters to indices and vice versa

chars = sorted(list(set(text)))

char_to_idx = {char: idx for idx, char in enumerate(chars)}

idx_to_char = {idx: char for idx, char in enumerate(chars)}
```

```
# Prepare the data (sequence of characters)
def prepare_data(text, seq_length):
    inputs = []
    targets = []
    for i in range(0, len(text) - seq_length, seq_length):
        seq_in = text[i:i + seq_length]
        seq_out = text[i + 1:i + seq_length + 1]
        inputs.append([char_to_idx[char] for char in seq_in])
        targets.append([char_to_idx[char] for char in seq_out])
    return np.array(inputs), np.array(targets)

X, y = prepare_data(text, seq_length)

# Define the RNN model (Vanilla RNN)
class CharRNN(nn.Module):
    def __init__(self, input_size, hidden_size, output_size):
        super(CharRNN, self).__init__()
        self.hidden_size = hidden_size
        self.rnn = nn.RNN(input_size, hidden_size, batch_first=True)
        self.fc = nn.Linear(hidden_size, output_size)

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

    def init_hidden(self, batch_size):
        return torch.zeros(batch_size, self.hidden_size)
```

```
# Initialize model, loss function, and optimizer
model = CharRNN(input_size=len(chars), hidden_size=hidden_size, output_size=len(chars))
loss_fn = nn.CrossEntropyLoss()
optimizer = optim.Adam(model.parameters(), lr=lr)

# One-hot encode the inputs for training
def one_hot_encode(data, vocab_size):
    encoded = np.zeros((data.shape[0], data.shape[1], vocab_size))
    for i, seq in enumerate(data):
        for j, idx in enumerate(seq):
            encoded[i, j, idx] = 1
    return torch.tensor(encoded, dtype=torch.float32)

X_train = one_hot_encode(X, len(chars))
y_train = torch.tensor(y, dtype=torch.long)

# Training loop
for epoch in range(epochs):
    hidden = model.init_hidden(batch_size)
    model.train()
    optimizer.zero_grad()

    # Forward pass
    output, hidden = model(X_train, hidden)

    # Compute loss
    loss = loss_fn(output, y_train.view(-1))
    loss.backward()

    # Update weights
```

```
    optimizer.step()

    if (epoch + 1) % 10 == 0:
        print(f"Epoch [{epoch+1}/{epochs}], Loss: {loss.item():.4f}")

# Text Generation
def generate_text(model, seed, num_chars, temperature=1.0):
    model.eval()
    input_seq = [char_to_idx[char] for char in seed]
    input_tensor = torch.tensor(input_seq).unsqueeze(0)
    hidden = model.init_hidden(1)

    generated_text = seed

    for _ in range(num_chars):
        input_tensor_one_hot = one_hot_encode(input_tensor, len(chars))
        output, hidden = model(input_tensor_one_hot, hidden)

        # Apply temperature to output probabilities
        output = output.squeeze().detach().numpy()
        output = output / temperature
        exp_output = np.exp(output - np.max(output))  # Normalize to avoid overflow
        probs = exp_output / np.sum(exp_output)  # Softmax

        # Sample from the output distribution
        idx = np.random.choice(len(chars), p=probs)
        generated_text += idx_to_char[idx]

        # Prepare the next input
        input_tensor = torch.tensor([idx]).unsqueeze(0)
```

```
    return generated_text

# Generate text
seed = "shall i compare thee to a summer's day"
generated_text = generate_text(model, seed, 500, temperature)
print(generated_text)
```

PROGRAM NO : 03

Music Generation with Variational Autoencoders (VAEs)

AIM

Design and implement a Variational Autoencoder (VAE) using frameworks like TensorFlow or PyTorch for generating novel music sequences. The VAE will be trained on a dataset of MIDI files or audio samples to learn a meaningful latent representation of the music data. Techniques for sampling from the latent space will be investigated to generate creative and novel music sequences.

ALGORITHM

1. Initialize an empty dataset.
2. For each MIDI file in the input paths:
3. Load the MIDI file.
4. Convert the MIDI file to a pianoroll representation.
5. Slice the pianoroll into fixed-length sequences and normalize values.
6. Return the processed dataset.
7. Create a neural network with:
8. Input layer matching sequence dimensions.
9. Hidden dense layers with activation functions.
10. Two output layers for latent mean (z_mean) and log variance (z_log_var).
11. Create a neural network with:
12. Input layer matching the latent dimensions.
13. Hidden dense layers with activation functions.
14. Output layer matching the original sequence dimensions, activated with sigmoid.
15. Combine the encoder and decoder.
16. Sample from the latent space using reparameterization trick:
17. z = z_mean + exp(0.5 * z_log_var) * epsilon, where epsilon is Gaussian noise.
18. Compute reconstruction loss and KL divergence loss.
19. Add losses for optimization.
20. Compile the VAE using an optimizer (e.g., Adam) and a loss function (e.g., MSE).
21. Train the model on the preprocessed data for multiple epochs with a defined batch size.
22. Sample random points in the latent space.
23. Pass the sampled points through the decoder.
24. Reshape the generated output into MIDI-compatible sequences.
25. Convert generated sequences to MIDI format for playback.

PROGRAM

```
import tensorflow as tf

from tensorflow.keras import layers

import numpy as np
```

```
import pretty_midi

# Define constants
LATENT_DIM = 128
INPUT_DIM = 128  # Example for MIDI pianoroll representation
SEQ_LENGTH = 100  # Length of sequence
BATCH_SIZE = 64
EPOCHS = 50

# Data preprocessing: Load MIDI files and convert them to pianoroll representation
def preprocess_midi(file_paths, seq_length):
    data = []
    for file_path in file_paths:
        try:
            midi_data = pretty_midi.PrettyMIDI(file_path)
            piano_roll = midi_data.get_piano_roll(fs=10)  # Example resolution: 10 Hz
            for i in range(0, piano_roll.shape[1] - seq_length, seq_length):
                data.append(piano_roll[:, i:i + seq_length].T)
        except Exception as e:
            print(f"Error processing {file_path}: {e}")
    return np.array(data) / 127.0  # Normalize MIDI velocities

# Define the encoder
class Encoder(layers.Layer):
    def __init__(self, latent_dim):
        super(Encoder, self).__init__()
        self.dense1 = layers.Dense(256, activation='relu')
        self.dense2 = layers.Dense(128, activation='relu')
        self.z_mean = layers.Dense(latent_dim)
        self.z_log_var = layers.Dense(latent_dim)
```

```
    def call(self, inputs):
        x = self.dense1(inputs)
        x = self.dense2(x)
        z_mean = self.z_mean(x)
        z_log_var = self.z_log_var(x)
        return z_mean, z_log_var

# Define the decoder
class Decoder(layers.Layer):
    def __init__(self, output_dim):
        super(Decoder, self).__init__()
        self.dense1 = layers.Dense(128, activation='relu')
        self.dense2 = layers.Dense(256, activation='relu')
        self.dense3 = layers.Dense(output_dim, activation='sigmoid')

    def call(self, inputs):
        x = self.dense1(inputs)
        x = self.dense2(x)
        return self.dense3(x)

# Define the VAE
class VAE(tf.keras.Model):
    def __init__(self, encoder, decoder):
        super(VAE, self).__init__()
        self.encoder = encoder
        self.decoder = decoder

    def call(self, inputs):
        z_mean, z_log_var = self.encoder(inputs)
```

```
        epsilon = tf.random.normal(shape=z_mean.shape)
        z = z_mean + tf.exp(0.5 * z_log_var) * epsilon
        reconstructed = self.decoder(z)
        kl_loss = -0.5 * tf.reduce_sum(1 + z_log_var - tf.square(z_mean) - tf.exp(z_log_var),
axis=-1)
        self.add_loss(kl_loss)
        return reconstructed

# Compile the model
def build_and_train_vae(data, latent_dim, input_dim, batch_size, epochs):
    encoder = Encoder(latent_dim)
    decoder = Decoder(input_dim)
    vae = VAE(encoder, decoder)

    vae.compile(optimizer=tf.keras.optimizers.Adam(),
loss=tf.keras.losses.MeanSquaredError())
    vae.fit(data, data, batch_size=batch_size, epochs=epochs)

    return vae

# Example usage:
file_paths = ["example1.mid", "example2.mid"]  # Replace with actual MIDI file paths
data = preprocess_midi(file_paths, SEQ_LENGTH)
vae = build_and_train_vae(data, LATENT_DIM, INPUT_DIM * SEQ_LENGTH,
BATCH_SIZE, EPOCHS)

# Sampling from the latent space
def generate_music(vae, num_samples, seq_length):
    latent_samples = tf.random.normal(shape=(num_samples, LATENT_DIM))
    generated_sequences = vae.decoder(latent_samples)
    return generated_sequences.numpy().reshape(num_samples, seq_length, INPUT_DIM)
```

```
# Generate new music samples
generated_music = generate_music(vae, 10, SEQ_LENGTH)
```

PROGRAM :O4

Style Transfer with Neural Style Transfer Algorithms

AIM

To implement a Neural Style Transfer algorithm to blend the artistic style of one image into the content of another, using TensorFlow.

ALGORITHM

1. Load content and style images.
2. Resize and normalize them to fit the VGG19 input requirements.
3. Define the Neural Network:
4. Use a pre-trained VGG19 model to extract style and content features.
5. Select specific layers for style (shallow) and content (deeper) representation.
6. Content Loss: Measure similarity between the content image and generated image at the content layer.
7. Style Loss: Compare Gram matrices of style layers between the style image and the generated image.
8. Total Loss: Combine content and style loss with respective weights.
9. Start with the content image as the initial generated image.
10. Use an optimizer (e.g., Adam) to iteratively update the generated image to minimize the total loss.
11. Convert the output image from model format back to displayable RGB format.
12. Save or show the resulting stylized image.

PROGRAM

```
# Neural Style Transfer using TensorFlow

import tensorflow as tf

from tensorflow.keras.applications import VGG19

from tensorflow.keras.models import Model

import numpy as np

import matplotlib.pyplot as plt

from PIL import Image


# Aim:

# To implement a Neural Style Transfer algorithm to blend the artistic style of one image into the content of another.


# Load and preprocess images
```

```
def load_and_preprocess_image(image_path, target_dim):
    img = Image.open(image_path)
    img = img.resize((target_dim, target_dim))
    img = tf.keras.applications.vgg19.preprocess_input(np.array(img, dtype=np.float32))
    return tf.convert_to_tensor(img[np.newaxis, ...])

# Postprocess the image to display
def postprocess_image(img):
    img = img[0].numpy()
    img = np.clip(img + [103.939, 116.779, 123.68], 0, 255).astype(np.uint8)
    return Image.fromarray(img[..., ::-1])  # Convert BGR to RGB

# Load the VGG19 model for feature extraction
def get_vgg_model():
    vgg = VGG19(include_top=False, weights="imagenet")
    content_layers = ["block5_conv2"]  # Content layer
    style_layers = ["block1_conv1", "block2_conv1", "block3_conv1", "block4_conv1", "block5_conv1"]

    outputs = [vgg.get_layer(name).output for name in (style_layers + content_layers)]
    return Model(inputs=vgg.input, outputs=outputs), style_layers, content_layers

# Compute content loss
def content_loss(base_content, target):
    return tf.reduce_mean(tf.square(base_content - target))

# Compute style loss
def gram_matrix(input_tensor):
    channels = int(input_tensor.shape[-1])
    a = tf.reshape(input_tensor, [-1, channels])
    n = tf.shape(a)[0]
```

```
    gram = tf.matmul(a, a, transpose_a=True)
    return gram / tf.cast(n, tf.float32)

def style_loss(base_style, gram_target):
    gram_style = gram_matrix(base_style)
    return tf.reduce_mean(tf.square(gram_style - gram_target))

# Define the total loss
def compute_loss(model, loss_weights, init_image, gram_style_features, content_features):
    style_weight, content_weight = loss_weights

    model_outputs = model(init_image)
    style_output_features = model_outputs[:len(gram_style_features)]
    content_output_features = model_outputs[len(gram_style_features):]

    style_loss_value = tf.add_n([style_loss(style_output, target) for style_output, target in zip(style_output_features, gram_style_features)])
    style_loss_value *= style_weight / len(gram_style_features)

    content_loss_value = tf.add_n([content_loss(content_output, target) for content_output, target in zip(content_output_features, content_features)])
    content_loss_value *= content_weight / len(content_features)

    total_loss = style_loss_value + content_loss_value
    return total_loss

# Training step
@tf.function
def train_step(image, model, optimizer, loss_weights, gram_style_features, content_features):
    with tf.GradientTape() as tape:
        loss = compute_loss(model, loss_weights, image, gram_style_features, content_features)
```

```
    grad = tape.gradient(loss, image)
    optimizer.apply_gradients([(grad, image)])
    image.assign(tf.clip_by_value(image, -103.939, 255 - 103.939))  # Keep image values within range


# Main function for style transfer
def style_transfer(content_path, style_path, iterations=1000, style_weight=1e-2, content_weight=1e4, target_dim=512):
    content_image = load_and_preprocess_image(content_path, target_dim)
    style_image = load_and_preprocess_image(style_path, target_dim)

    model, style_layers, content_layers = get_vgg_model()
    model.trainable = False

    # Extract features
    style_features = model(style_image)[:len(style_layers)]
    gram_style_features = [gram_matrix(feature) for feature in style_features]
    content_features = model(content_image)[len(style_layers):]

    # Initialize the generated image
    init_image = tf.Variable(content_image, dtype=tf.float32)

    # Optimizer
    optimizer = tf.keras.optimizers.Adam(learning_rate=0.02)

    # Train the model
    for i in range(iterations):
        train_step(init_image, model, optimizer, (style_weight, content_weight), gram_style_features, content_features)
        if i % 100 == 0:
            print(f"Iteration {i}/{iterations} completed.")
```

```
    return postprocess_image(init_image)

# Example Usage:
content_path = "path_to_content_image.jpg"
style_path = "path_to_style_image.jpg"
output_image = style_transfer(content_path, style_path, iterations=500)
output_image.show()
```

PROGRAM :O5

Data Augmentation with Generative Models

AIM

To utilize generative models like GANs for augmenting training data in classification tasks and compare the performance of classifiers trained with and without augmented data.

ALGORITHM

1. Load the dataset (e.g., MNIST) and preprocess the images (normalize, reshape).
2. Build a simple GAN with a generator and discriminator.
3. Compile the discriminator and GAN model.
4. Train the discriminator using real and generated images.
5. Train the generator to fool the discriminator.
6. Use the trained generator to create synthetic images.
7. Assign labels to the generated images for the classification task.
8. Build and compile a CNN for classification tasks.
9. Train one classifier using the original dataset.
10. Train another classifier using the augmented dataset (original + generated).
11. Compare the test set accuracy of both classifiers to measure the impact of augmentation.

PROGRAM

```
import tensorflow as tf

from tensorflow.keras import layers, models

import numpy as np

from sklearn.model_selection import train_test_split

from sklearn.metrics import accuracy_score


# Aim:

# To utilize generative models like GANs or VAEs for data augmentation and compare the
performance

# of a classifier trained with and without the augmented data.


# Load a sample dataset (e.g., MNIST for demonstration)

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

x_train = x_train.astype('float32') / 255.0

x_test = x_test.astype('float32') / 255.0
```

```
x_train = np.expand_dims(x_train, axis=-1)
x_test = np.expand_dims(x_test, axis=-1)
y_train = tf.keras.utils.to_categorical(y_train, 10)
y_test = tf.keras.utils.to_categorical(y_test, 10)

# Split into training and validation
x_train, x_val, y_train, y_val = train_test_split(x_train, y_train, test_size=0.2,
random_state=42)

# Define a simple GAN for data generation
class GAN:
    def __init__(self, input_dim):
        self.input_dim = input_dim
        self.generator = self.build_generator()
        self.discriminator = self.build_discriminator()
        self.gan = self.build_gan()

    def build_generator(self):
        model = models.Sequential([
            layers.Dense(256, activation='relu', input_dim=self.input_dim),
            layers.BatchNormalization(),
            layers.Dense(512, activation='relu'),
            layers.BatchNormalization(),
            layers.Dense(28 * 28, activation='sigmoid'),
            layers.Reshape((28, 28, 1))
        ])
        return model

    def build_discriminator(self):
        model = models.Sequential([
            layers.Flatten(input_shape=(28, 28, 1)),
```

```
            layers.Dense(512, activation='relu'),
            layers.Dense(256, activation='relu'),
            layers.Dense(1, activation='sigmoid')
        ])
        model.compile(optimizer=tf.keras.optimizers.Adam(0.0002),
loss='binary_crossentropy', metrics=['accuracy'])
        return model

    def build_gan(self):
        self.discriminator.trainable = False
        model = models.Sequential([
            self.generator,
            self.discriminator
        ])
        model.compile(optimizer=tf.keras.optimizers.Adam(0.0002),
loss='binary_crossentropy')
        return model

    def train(self, x_train, epochs, batch_size):
        real_labels = np.ones((batch_size, 1))
        fake_labels = np.zeros((batch_size, 1))

        for epoch in range(epochs):
            # Train discriminator
            idx = np.random.randint(0, x_train.shape[0], batch_size)
            real_images = x_train[idx]

            noise = np.random.normal(0, 1, (batch_size, self.input_dim))
            generated_images = self.generator.predict(noise)

            d_loss_real = self.discriminator.train_on_batch(real_images, real_labels)
```

```
            d_loss_fake = self.discriminator.train_on_batch(generated_images, fake_labels)

            # Train generator
            noise = np.random.normal(0, 1, (batch_size, self.input_dim))
            g_loss = self.gan.train_on_batch(noise, real_labels)

            if epoch % 100 == 0:
                print(f"Epoch {epoch}, D Loss: {0.5 * np.add(d_loss_real, d_loss_fake)}, G Loss:
{g_loss}")

# Train GAN
gan = GAN(input_dim=100)
gan.train(x_train, epochs=5000, batch_size=64)

# Generate augmented data
num_generated = 10000
noise = np.random.normal(0, 1, (num_generated, 100))
generated_images = gan.generator.predict(noise)
generated_labels = tf.keras.utils.to_categorical(np.random.randint(0, 10, num_generated), 10)

x_augmented = np.concatenate((x_train, generated_images), axis=0)
y_augmented = np.concatenate((y_train, generated_labels), axis=0)

# Define a CNN classifier
def build_classifier():
    model = models.Sequential([
        layers.Conv2D(32, kernel_size=(3, 3), activation='relu', input_shape=(28, 28, 1)),
        layers.MaxPooling2D(pool_size=(2, 2)),
        layers.Conv2D(64, (3, 3), activation='relu'),
        layers.MaxPooling2D(pool_size=(2, 2)),
        layers.Flatten(),
```

```
        layers.Dense(128, activation='relu'),
        layers.Dense(10, activation='softmax')
    ])
    model.compile(optimizer='adam', loss='categorical_crossentropy', metrics=['accuracy'])
    return model

# Train classifiers
classifier_original = build_classifier()
classifier_augmented = build_classifier()

print("Training on original data...")
classifier_original.fit(x_train, y_train, validation_data=(x_val, y_val), epochs=10,
batch_size=64)

print("Training on augmented data...")
classifier_augmented.fit(x_augmented, y_augmented, validation_data=(x_val, y_val),
epochs=10, batch_size=64)

# Evaluate classifiers
original_acc = classifier_original.evaluate(x_test, y_test, verbose=0)[1]
augmented_acc = classifier_augmented.evaluate(x_test, y_test, verbose=0)[1]

print(f"Accuracy on original data: {original_acc:.2f}")
print(f"Accuracy with augmented data: {augmented_acc:.2f}")
```

PROGRAM :O6

Video Generation with Generative Adversarial Networks (GANs)

AIM

To extend GAN architectures to generate video sequences, train the GAN on a dataset of video clips, and evaluate the generated sequences based on realism and diversity.

ALGORITHM

1. Load video clips and preprocess them into a suitable format (e.g., normalized 5D tensors: [samples, time, height, width, channels]).
2. Generator: Use a 3D Convolutional Neural Network (Conv3DTranspose) to generate video sequences.
3. Discriminator: Use a 3D Convolutional Neural Network (Conv3D) to classify sequences as real or fake.
4. Train the discriminator with real video clips labeled as 1 and generated video clips labeled as 0.
5. Train the generator by fooling the discriminator into classifying fake videos as real.
6. Generate video sequences using the trained generator.
7. Evaluate realism and diversity using qualitative (visual inspection) and quantitative metrics (e.g., FID for videos).
8. Save generated video clips for further analysis or visualization.

PROGRAM

```
import tensorflow as tf

from tensorflow.keras import layers, models

import numpy as np

import os

# Define 3D GAN for Video Generation

class VideoGAN:

    def __init__(self, noise_dim, video_shape):

        self.noise_dim = noise_dim

        self.video_shape = video_shape

        self.generator = self.build_generator()

        self.discriminator = self.build_discriminator()

        self.gan = self.build_gan()
```

```
    def build_generator(self):
        model = models.Sequential([
            layers.Dense(8 * 8 * 256, activation="relu", input_dim=self.noise_dim),
            layers.Reshape((8, 8, 1, 256)),
            layers.Conv3DTranspose(128, kernel_size=(4, 4, 4), strides=(2, 2, 2),
padding="same", activation="relu"),
            layers.BatchNormalization(),
            layers.Conv3DTranspose(64, kernel_size=(4, 4, 4), strides=(2, 2, 2), padding="same",
activation="relu"),
            layers.BatchNormalization(),
            layers.Conv3DTranspose(3, kernel_size=(4, 4, 4), strides=(2, 2, 2), padding="same",
activation="tanh")
        ])
        return model

    def build_discriminator(self):
        model = models.Sequential([
            layers.Conv3D(64, kernel_size=(4, 4, 4), strides=(2, 2, 2), padding="same",
input_shape=self.video_shape),
            layers.LeakyReLU(alpha=0.2),
            layers.Conv3D(128, kernel_size=(4, 4, 4), strides=(2, 2, 2), padding="same"),
            layers.LeakyReLU(alpha=0.2),
            layers.Flatten(),
            layers.Dense(1, activation="sigmoid")
        ])
        model.compile(optimizer=tf.keras.optimizers.Adam(0.0002, 0.5),
loss="binary_crossentropy", metrics=["accuracy"])
        return model

    def build_gan(self):
        self.discriminator.trainable = False
        model = models.Sequential([
```

```
            self.generator,
            self.discriminator
        ])
        model.compile(optimizer=tf.keras.optimizers.Adam(0.0002, 0.5),
loss="binary_crossentropy")
        return model

    def train(self, video_data, epochs, batch_size):
        half_batch = batch_size // 2

        for epoch in range(epochs):
            # Train Discriminator
            idx = np.random.randint(0, video_data.shape[0], half_batch)
            real_videos = video_data[idx]
            noise = np.random.normal(0, 1, (half_batch, self.noise_dim))
            generated_videos = self.generator.predict(noise)

            real_labels = np.ones((half_batch, 1))
            fake_labels = np.zeros((half_batch, 1))

            d_loss_real = self.discriminator.train_on_batch(real_videos, real_labels)
            d_loss_fake = self.discriminator.train_on_batch(generated_videos, fake_labels)

            # Train Generator
            noise = np.random.normal(0, 1, (batch_size, self.noise_dim))
            valid_labels = np.ones((batch_size, 1))
            g_loss = self.gan.train_on_batch(noise, valid_labels)

            # Print losses every epoch
            if epoch % 10 == 0:
```

```
            print(f"Epoch {epoch}/{epochs}, D Loss: {0.5 * np.add(d_loss_real,
d_loss_fake)}, G Loss: {g_loss}")

# Preprocess Video Data (Example assumes dataset of shape [num_samples, time_steps,
height, width, channels])
def preprocess_videos(video_paths, video_shape):
    videos = []
    for path in video_paths:
        video = np.random.random(video_shape)  # Replace with video loading logic
        videos.append(video)
    return np.array(videos, dtype="float32") / 127.5 - 1.0

# Example Usage
video_shape = (32, 64, 64, 3)  # (time_steps, height, width, channels)
noise_dim = 100
video_paths = ["path_to_video1", "path_to_video2"]  # Replace with actual video file paths
video_data = preprocess_videos(video_paths, video_shape)

video_gan = VideoGAN(noise_dim=noise_dim, video_shape=video_shape)
video_gan.train(video_data, epochs=1000, batch_size=16)

# Generate and save video sequences
num_videos = 5
noise = np.random.normal(0, 1, (num_videos, noise_dim))
generated_videos = video_gan.generator.predict(noise)

# Save generated videos (Replace with actual saving logic)
for i, video in enumerate(generated_videos):
    save_path = f"generated_video_{i}.mp4"
    print(f"Generated video saved to {save_path}")
```

PROGRAM :O7

Anomaly Detection with Generative Models

AIM

To train a Variational Autoencoder (VAE) on a dataset containing only normal instances and detect anomalies using reconstruction errors or latent space distances.

ALGORITHM

1. Load and preprocess a dataset containing normal instances (e.g., MNIST for simplicity).
2. Create an encoder to map input data to a latent space representation.
3. Use a decoder to reconstruct the input data from the latent representation.
4. Implement reparameterization for generating latent variables.
5. Train the VAE on normal data using a combination of reconstruction loss and KL divergence.
6. Pass both normal and anomalous data through the trained VAE.
7. Compute reconstruction errors for each sample.
8. Determine a threshold for reconstruction errors using a statistical method (e.g., 95th percentile of errors).
9. Flag instances with reconstruction errors above the threshold as anomalies.

PROGRAM

```
import tensorflow as tf

from tensorflow.keras import layers, models

import numpy as np

# Define the VAE Architecture

class VAE(tf.keras.Model):

    def __init__(self, latent_dim):

        super(VAE, self).__init__()

        self.latent_dim = latent_dim


        # Encoder
```

```
        self.encoder = models.Sequential([
            layers.InputLayer(input_shape=(28, 28, 1)),
            layers.Conv2D(32, (3, 3), activation="relu", strides=2, padding="same"),
            layers.Conv2D(64, (3, 3), activation="relu", strides=2, padding="same"),
            layers.Flatten(),
            layers.Dense(latent_dim + latent_dim),  # Mean and log-variance
        ])

        # Decoder
        self.decoder = models.Sequential([
            layers.InputLayer(input_shape=(latent_dim,)),
            layers.Dense(7 * 7 * 64, activation="relu"),
            layers.Reshape((7, 7, 64)),
            layers.Conv2DTranspose(64, (3, 3), activation="relu", strides=2, padding="same"),
            layers.Conv2DTranspose(32, (3, 3), activation="relu", strides=2, padding="same"),
            layers.Conv2DTranspose(1, (3, 3), activation="sigmoid", padding="same"),
        ])

    def encode(self, x):
        mean, logvar = tf.split(self.encoder(x), num_or_size_splits=2, axis=1)
        return mean, logvar

    def reparameterize(self, mean, logvar):
        eps = tf.random.normal(shape=mean.shape)
        return eps * tf.exp(logvar * 0.5) + mean

    def decode(self, z, apply_sigmoid=False):
        logits = self.decoder(z)
        return logits
```

```
    def call(self, x):
        mean, logvar = self.encode(x)
        z = self.reparameterize(mean, logvar)
        return self.decode(z)

# Loss function
@tf.function
def compute_loss(model, x):
    mean, logvar = model.encode(x)
    z = model.reparameterize(mean, logvar)
    x_logit = model.decode(z)
    reconstruction_loss = tf.reduce_mean(
        tf.reduce_sum(tf.keras.losses.binary_crossentropy(x, x_logit), axis=(1, 2))
    )
    kl_divergence = -0.5 * tf.reduce_sum(1 + logvar - tf.square(mean) - tf.exp(logvar))
    return reconstruction_loss + kl_divergence

# Training step
@tf.function
def train_step(model, x, optimizer):
    with tf.GradientTape() as tape:
        loss = compute_loss(model, x)
    gradients = tape.gradient(loss, model.trainable_variables)
    optimizer.apply_gradients(zip(gradients, model.trainable_variables))

# Load and preprocess dataset
(x_train, _), (x_test, _) = tf.keras.datasets.mnist.load_data()
x_train = x_train.astype("float32") / 255.0
x_test = x_test.astype("float32") / 255.0
x_train = np.expand_dims(x_train, axis=-1)
```

```
x_test = np.expand_dims(x_test, axis=-1)

# Train VAE
latent_dim = 2
vae = VAE(latent_dim=latent_dim)
optimizer = tf.keras.optimizers.Adam(learning_rate=1e-3)
epochs = 20
batch_size = 64

dataset = tf.data.Dataset.from_tensor_slices(x_train).shuffle(60000).batch(batch_size)
for epoch in range(epochs):
    for batch in dataset:
        train_step(vae, batch, optimizer)
    print(f"Epoch {epoch + 1}/{epochs} completed.")

# Anomaly Detection
reconstruction_errors = []
for x in x_test:
    x = tf.expand_dims(x, axis=0)
    reconstruction = vae(x)
    error = tf.reduce_mean(tf.abs(x - reconstruction))
    reconstruction_errors.append(error.numpy())

# Define threshold
threshold = np.percentile(reconstruction_errors, 95)

# Evaluate anomalies
def detect_anomalies(data, model, threshold):
    anomalies = []
    for x in data:
```

```
        x = tf.expand_dims(x, axis=0)
        reconstruction = model(x)
        error = tf.reduce_mean(tf.abs(x - reconstruction))
        if error > threshold:
            anomalies.append(True)
        else:
            anomalies.append(False)
    return anomalies

# Example usage
anomalies = detect_anomalies(x_test, vae, threshold)
print(f"Detected anomalies: {np.sum(anomalies)} out of {len(x_test)} samples.")
```

PROGRAM :O8

Domain Adaptation with Generative Adversarial Networks (GANs)

AIM

To train a Variational Autoencoder (VAE) on a dataset containing only normal instances and detect anomalies using reconstruction errors or latent space distances.

ALGORITHM

1. Load and preprocess a dataset containing normal instances (e.g., MNIST for simplicity).
2. Create an encoder to map input data to a latent space representation.
3. Use a decoder to reconstruct the input data from the latent representation.
4. Implement reparameterization for generating latent variables.
5. Train the VAE on normal data using a combination of reconstruction loss and KL divergence
6. Pass both normal and anomalous data through the trained VAE.
7. Compute reconstruction errors for each sample.
8. Determine a threshold for reconstruction errors using a statistical method (e.g., 95th percentile of errors).
9. Flag instances with reconstruction errors above the threshold as anomalies.

PROGRAM

```
import tensorflow as tf

from tensorflow.keras import layers, models

import numpy as np


# Aim:

# To implement domain adaptation using Generative Adversarial Networks (GANs), transferring knowledge

# from a labeled source domain to an unlabeled target domain and evaluating the adapted model on target domain tasks.


# Define the Domain Adaptation GAN

class DomainAdaptationGAN:

    def __init__(self, input_shape):

        self.input_shape = input_shape

        self.generator = self.build_generator()
```

```
        self.discriminator = self.build_discriminator()
        self.adversarial_model = self.build_adversarial_model()

    def build_generator(self):
        model = models.Sequential([
            layers.InputLayer(input_shape=self.input_shape),
            layers.Conv2D(64, (3, 3), strides=1, padding='same', activation='relu'),
            layers.BatchNormalization(),
            layers.Conv2D(128, (3, 3), strides=2, padding='same', activation='relu'),
            layers.BatchNormalization(),
            layers.Conv2DTranspose(128, (3, 3), strides=2, padding='same', activation='relu'),
            layers.BatchNormalization(),
            layers.Conv2D(3, (3, 3), strides=1, padding='same', activation='tanh'),
        ])
        return model

    def build_discriminator(self):
        model = models.Sequential([
            layers.InputLayer(input_shape=self.input_shape),
            layers.Conv2D(64, (3, 3), strides=2, padding='same', activation='leaky_relu'),
            layers.BatchNormalization(),
            layers.Conv2D(128, (3, 3), strides=2, padding='same', activation='leaky_relu'),
            layers.BatchNormalization(),
            layers.Flatten(),
            layers.Dense(1, activation='sigmoid'),
        ])
        model.compile(optimizer=tf.keras.optimizers.Adam(0.0002,                    beta_1=0.5),
loss='binary_crossentropy', metrics=['accuracy'])
        return model

    def build_adversarial_model(self):
```

```
        self.discriminator.trainable = False
        model = models.Sequential([
            self.generator,
            self.discriminator
        ])
        model.compile(optimizer=tf.keras.optimizers.Adam(0.0002, beta_1=0.5), loss='binary_crossentropy')
        return model

    def train(self, source_data, target_data, epochs, batch_size):
        for epoch in range(epochs):
            # Train discriminator with source and target samples
            idx_s = np.random.randint(0, source_data.shape[0], batch_size // 2)
            idx_t = np.random.randint(0, target_data.shape[0], batch_size // 2)

            real_source = source_data[idx_s]
            real_target = target_data[idx_t]

            fake_target = self.generator.predict(real_source)

            real_labels = np.ones((batch_size // 2, 1))
            fake_labels = np.zeros((batch_size // 2, 1))

            d_loss_real = self.discriminator.train_on_batch(real_target, real_labels)
            d_loss_fake = self.discriminator.train_on_batch(fake_target, fake_labels)

            # Train generator to fool discriminator
            g_loss = self.adversarial_model.train_on_batch(real_source, real_labels)

            if epoch % 10 == 0:
```

```
        print(f"Epoch {epoch}, D Loss: {0.5 * np.add(d_loss_real, d_loss_fake)}, G Loss: {g_loss}")

# Prepare Source and Target Data (Example data)
def preprocess_data(data, input_shape):
    data = tf.image.resize(data, (input_shape[0], input_shape[1]))
    return data / 127.5 - 1.0

# Example usage
input_shape = (64, 64, 3)
source_data = np.random.random((1000, 64, 64, 3))  # Replace with actual source data
target_data = np.random.random((1000, 64, 64, 3))  # Replace with actual target data

source_data = preprocess_data(source_data, input_shape)
target_data = preprocess_data(target_data, input_shape)

# Initialize and train the GAN
adaptation_gan = DomainAdaptationGAN(input_shape)
adaptation_gan.train(source_data, target_data, epochs=100, batch_size=32)

# Evaluate on target domain tasks (Custom evaluation logic here)
print("Domain adaptation completed.")
```