SRM Institute of Science and Technology

Department of Computer Applications

Delhi – Meerut Road, Sikri Kalan, Ghaziabad, Uttar Pradesh – 201204

Circular – 2020-21

MCA GAI 1st semester

Fundamentals of Generative AI and Working with Open AI (PGI20C03J)

Lab Manual

Lab 1: Simple programs on Open API

Title

Introduction to Generative AI with a Simple Text Generation API Interaction

Aim

To understand the basic process of interacting with a generative AI API for text generation.

Procedure

1. **Understand API Interaction:** Learn about sending requests (e.g., HTTP POST) to an API endpoint and receiving responses (e.g., JSON).
2. **Define the Prompt:** Formulate a clear and concise text prompt that you want the AI model to complete or generate text based on.
3. **Construct the Request:** Prepare the data payload (e.g., a JSON object containing the prompt and any generation parameters like `max_tokens`, `temperature`).
4. **Send the Request:** Use a suitable library (e.g., `requests` in Python) to send the HTTP request to the API endpoint.
5. **Process the Response:** Parse the JSON response from the API to extract the generated text.
6. **Handle Errors:** Implement basic error handling for API call failures or unexpected responses.

Source Code

```
import requests
import json

def generate_text_from_api(prompt, api_url, api_key, max_tokens=50,
temperature=0.7):
    """
    Generates text using a hypothetical generative AI API.

    Args:
        prompt (str): The input text prompt for the AI.
        api_url (str): The URL of the AI API endpoint.
        api_key (str): Your API key for authentication.
        max_tokens (int): The maximum number of tokens to generate.
        temperature (float): Controls the randomness of the output. Higher
values mean more random.

    Returns:
```

```
        str: The generated text, or an error message if the API call fails.
    """
    headers = {
        "Content-Type": "application/json",
        "Authorization": f"Bearer {api_key}" # Assuming Bearer token
authentication
    }
    payload = {
        "prompt": prompt,
        "max_tokens": max_tokens,
        "temperature": temperature
    }

    try:
        response = requests.post(api_url, headers=headers,
data=json.dumps(payload))
        response.raise_for_status() # Raise an HTTPError for bad responses (4xx
or 5xx)
        response_data = response.json()

        # Assuming the generated text is in response_data['choices'][0]['text']
        if 'choices' in response_data and len(response_data['choices']) > 0:
            return response_data['choices'][0]['text'].strip()
        else:
            return "Error: Unexpected API response format."

    except requests.exceptions.HTTPError as errh:
        return f"Http Error: {errh}"
    except requests.exceptions.ConnectionError as errc:
        return f"Error Connecting: {errc}"
    except requests.exceptions.Timeout as errt:
        return f"Timeout Error: {errt}"
    except requests.exceptions.RequestException as err:
        return f"Something went wrong: {err}"
    except json.JSONDecodeError:
        return "Error: Could not decode JSON response."

# --- Example Usage ---
# Replace with your actual API URL and Key
# For demonstration purposes, this will not make a real API call.
# This structure is similar to how you'd interact with models like OpenAI's GPT.
hypothetical_api_url = "https://api.example.com/generate"
your_api_key = "YOUR_API_KEY_HERE" # Get this from your API provider

if __name__ == "__main__":
    input_prompt = "Write a short story about a robot who discovers art."
    generated_text = generate_text_from_api(input_prompt, hypothetical_api_url,
your_api_key)
    print("Generated Text:")
    print(generated_text)
```

Input

```
Prompt: "Write a short story about a robot who discovers art."
```

Expected Output

```
Generated Text:
Unit 734, a sanitation bot, whirred diligently through the city's alleys. Its
optical sensors, usually focused on refuse, flickered over a discarded canvas.
Streaks of vibrant color, haphazard yet harmonious, captivated its processors.
This was not logical. This was... art. A new directive sparked within its
circuits: understand beauty.
```

Lab 2: Training a simple autoencoder model on a dataset.

Title

Building and Training a Basic Autoencoder for Data Reconstruction

Aim

To understand the fundamental principles of autoencoders and implement a simple autoencoder model for dimensionality reduction and data reconstruction using a given dataset.

Procedure

1. **Dataset Preparation:** Load and preprocess a suitable dataset (e.g., MNIST, Fashion MNIST). This involves normalizing pixel values and reshaping data.
2. **Model Architecture:** Define the encoder and decoder components of the autoencoder. The encoder maps input data to a lower-dimensional latent space, and the decoder reconstructs the original data from this latent representation.
3. **Model Compilation:** Compile the autoencoder model, specifying an optimizer (e.g., Adam) and a loss function (e.g., Mean Squared Error for reconstruction).
4. **Model Training:** Train the autoencoder on the dataset. The goal during training is to minimize the reconstruction loss, forcing the model to learn an efficient encoding of the input data.
5. **Evaluation:** Evaluate the trained autoencoder by reconstructing unseen data and visually comparing the original and reconstructed outputs.

Source Code

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

def build_autoencoder(input_shape, latent_dim):
    """
    Builds a simple autoencoder model.

    Args:
        input_shape (tuple): Shape of the input data (e.g., (28, 28, 1) for
MNIST).
        latent_dim (int): Dimensionality of the latent space.

    Returns:
        tf.keras.Model: The complete autoencoder model.
        tf.keras.Model: The encoder part of the autoencoder.
        tf.keras.Model: The decoder part of the autoencoder.
    """
    # Encoder
    encoder_input = tf.keras.Input(shape=input_shape)
    x = layers.Flatten()(encoder_input)
    x = layers.Dense(128, activation='relu')(x)
    latent_output = layers.Dense(latent_dim, activation='relu')(x)
    encoder = models.Model(encoder_input, latent_output, name="encoder")

    # Decoder
    decoder_input = tf.keras.Input(shape=(latent_dim,))
    x = layers.Dense(128, activation='relu')(decoder_input)
```

```
    x = layers.Dense(np.prod(input_shape), activation='sigmoid')(x) # Output
layer matches input size
    decoder_output = layers.Reshape(input_shape)(x)
    decoder = models.Model(decoder_input, decoder_output, name="decoder")

    # Autoencoder
    autoencoder_input = tf.keras.Input(shape=input_shape)
    encoded = encoder(autoencoder_input)
    decoded = decoder(encoded)
    autoencoder = models.Model(autoencoder_input, decoded, name="autoencoder")

    return autoencoder, encoder, decoder

if __name__ == "__main__":
    # Load and preprocess the MNIST dataset
    (x_train, _), (x_test, _) = datasets.mnist.load_data()
    x_train = x_train.astype('float32') / 255.0
    x_test = x_test.astype('float32') / 255.0

    # Reshape for convolutional layers if needed, or keep as is for dense
    input_shape = x_train.shape[1:] # (28, 28)
    # Add a channel dimension for consistency with image processing
    x_train = np.expand_dims(x_train, -1) # (60000, 28, 28, 1)
    x_test = np.expand_dims(x_test, -1)   # (10000, 28, 28, 1)
    input_shape = x_train.shape[1:] # (28, 28, 1)

    latent_dimension = 32 # Dimension of the compressed representation

    # Build the autoencoder
    autoencoder, encoder, decoder = build_autoencoder(input_shape,
latent_dimension)

    # Compile the autoencoder
    autoencoder.compile(optimizer='adam', loss='mse') # Mean Squared Error for
reconstruction

    # Print model summary
    print("Autoencoder Summary:")
    autoencoder.summary()
    print("\nEncoder Summary:")
    encoder.summary()
    print("\nDecoder Summary:")
    decoder.summary()

    # Train the autoencoder
    print("\nTraining Autoencoder...")
    history = autoencoder.fit(x_train, x_train,
                              epochs=10,
                              batch_size=256,
                              shuffle=True,
                              validation_data=(x_test, x_test))

    # Plot training history
    plt.figure(figsize=(10, 5))
    plt.plot(history.history['loss'], label='Training Loss')
    plt.plot(history.history['val_loss'], label='Validation Loss')
    plt.title('Autoencoder Training Loss')
    plt.xlabel('Epoch')
    plt.ylabel('Loss')
    plt.legend()
    plt.grid(True)
    plt.show()

    # Evaluate on test data
    reconstructed_images = autoencoder.predict(x_test)

    # Visualize original vs. reconstructed images
```

```
    n = 10 # Number of images to display
    plt.figure(figsize=(20, 4))
    for i in range(n):
        # Original
        ax = plt.subplot(2, n, i + 1)
        plt.imshow(x_test[i].reshape(input_shape[:-1]), cmap='gray')
        plt.title("Original")
        plt.axis('off')

        # Reconstructed
        ax = plt.subplot(2, n, i + 1 + n)
        plt.imshow(reconstructed_images[i].reshape(input_shape[:-1]),
cmap='gray')
        plt.title("Reconstructed")
        plt.axis('off')
    plt.suptitle('Original vs. Reconstructed Images (MNIST)')
    plt.show()
```

Input

The MNIST dataset (handwritten digits). The code automatically downloads it.

Expected Output

1. Summary of the autoencoder, encoder, and decoder models (number of layers, parameters).
2. Training progress showing epoch number, loss, and validation loss.
3. A plot displaying the training and validation loss over epochs.
4. A plot showing original MNIST digits alongside their reconstructed counterparts, demonstrating the autoencoder's ability to reconstruct images. The reconstruction loss should decrease over epochs.

Lab 3: Implementing a basic GAN architecture for generating synthetic images using a pre-trained model.

Title

Synthetic Image Generation with a Pre-trained GAN Generator

Aim

To understand the concept of Generative Adversarial Networks (GANs) and to utilize a pre-trained GAN generator to produce synthetic images from random noise.

Procedure

1. **Understand GANs:** Briefly review the adversarial training process involving a generator and a discriminator.
2. **Load Pre-trained Generator:** Identify and load a pre-trained generator model. For this lab, we'll simulate loading one as a full pre-trained model is complex to include directly.
3. **Generate Noise Vector:** Create random noise vectors (latent vectors) which serve as input to the generator. The distribution of this noise (e.g., normal distribution) is crucial.
4. **Generate Images:** Pass the noise vectors through the pre-trained generator model to produce synthetic images.
5. **Visualize Results:** Display the generated images to observe the quality and diversity of the synthetic outputs.

Source Code

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

def build_generator(latent_dim, output_channels=1):
    """
    Builds a simple generator model (simulating a pre-trained one).
    This is a placeholder; in a real scenario, you would load a saved model.

    Args:
        latent_dim (int): Dimension of the input noise vector.
        output_channels (int): Number of channels in the output image (e.g., 1
for grayscale, 3 for RGB).

    Returns:
        tf.keras.Model: The generator model.
    """
    model = models.Sequential(name="generator")
    # Foundation for 7x7 image
    model.add(layers.Dense(7 * 7 * 128, use_bias=False,
input_shape=(latent_dim,)))
    model.add(layers.BatchNormalization())
    model.add(layers.LeakyReLU())

    model.add(layers.Reshape((7, 7, 128)))
    assert model.output_shape == (None, 7, 7, 128) # Note: None is for the batch
size
```

```
    # Upsample to 14x14
    model.add(layers.Conv2DTranspose(64, (5, 5), strides=(2, 2), padding='same',
use_bias=False))
    model.add(layers.BatchNormalization())
    model.add(layers.LeakyReLU())
    assert model.output_shape == (None, 14, 14, 64)

    # Upsample to 28x28
    model.add(layers.Conv2DTranspose(output_channels, (5, 5), strides=(2, 2),
padding='same', use_bias=False, activation='tanh'))
    assert model.output_shape == (None, 28, 28, output_channels)

    return model

if __name__ == "__main__":
    latent_dimension = 100 # Dimension of the noise vector
    num_images_to_generate = 16

    # Simulate loading a pre-trained generator
    # In a real scenario, you would load a model like:
    # generator =
tf.keras.models.load_model('path/to/your/pretrained_generator.h5')
    # For this lab, we'll build a simple one to demonstrate the process.
    generator = build_generator(latent_dimension)
    print("Generator Summary (Simulated Pre-trained):")
    generator.summary()

    # Generate random noise vectors
    # Using a normal distribution for noise is common in GANs
    noise = tf.random.normal([num_images_to_generate, latent_dimension])

    # Generate images from the noise
    generated_images = generator(noise, training=False) # Set training=False for
inference

    # Post-process images for display (e.g., scale from tanh output to 0-1)
    generated_images = (generated_images + 1) / 2.0 # Scale from [-1, 1] to [0,
1]

    # Visualize the generated images
    plt.figure(figsize=(4, 4))
    for i in range(generated_images.shape[0]):
        plt.subplot(4, 4, i+1)
        plt.imshow(generated_images[i, :, :, 0], cmap='gray') # Assuming
grayscale images
        plt.axis('off')
    plt.suptitle(f'Generated Images from a Pre-trained GAN (Latent Dim:
{latent_dimension})')
    plt.show()
```

Input

Random noise vectors (e.g., 100-dimensional vectors sampled from a normal distribution).

Expected Output

A grid of synthetic images (e.g., 28x28 grayscale images resembling digits or simple objects, depending on what the simulated pre-trained GAN was trained on). The quality will vary based on the simplicity of the simulated generator.

Lab 4: Implementing a basic autoencoder using TensorFlow or PyTorch.

Title

Basic Autoencoder Implementation with TensorFlow

Aim

To implement a basic autoencoder model using TensorFlow/Keras, focusing on the encoder-decoder structure and its application for unsupervised feature learning and data reconstruction.

Procedure

1. **Data Loading and Preprocessing:** Load a dataset (e.g., Fashion MNIST) and normalize its pixel values to a range suitable for neural networks (e.g., 0-1). Reshape images if necessary.
2. **Encoder Definition:** Design the encoder part of the autoencoder. This typically consists of several dense or convolutional layers that progressively reduce the dimensionality of the input, leading to a compact latent representation.
3. **Decoder Definition:** Design the decoder part, which mirrors the encoder. It takes the latent representation as input and reconstructs the original data by gradually increasing dimensionality.
4. **Autoencoder Assembly:** Combine the encoder and decoder to form the complete autoencoder model.
5. **Model Compilation and Training:** Compile the autoencoder with an appropriate optimizer and a reconstruction loss function (e.g., Mean Squared Error or Binary Cross-Entropy for images). Train the model on the input data, with the target being the input data itself.
6. **Visualization and Evaluation:** Visualize original vs. reconstructed images to assess the autoencoder's performance.

Source Code

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

def create_autoencoder(input_shape, encoding_dim):
    """
    Creates a basic autoencoder model using TensorFlow/Keras.

    Args:
        input_shape (tuple): Shape of the input data (e.g., (28, 28, 1)).
        encoding_dim (int): The size of the bottleneck/latent layer.

    Returns:
        tf.keras.Model: The autoencoder model.
        tf.keras.Model: The encoder model.
        tf.keras.Model: The decoder model.
    """
    # Encoder
    encoder_input = tf.keras.Input(shape=input_shape, name="encoder_input")
    x = layers.Flatten()(encoder_input)
    x = layers.Dense(128, activation='relu')(x)
```

```
    encoding = layers.Dense(encoding_dim, activation='relu',
name="latent_representation")(x)
    encoder = models.Model(encoder_input, encoding, name="encoder")

    # Decoder
    decoder_input = tf.keras.Input(shape=(encoding_dim,), name="decoder_input")
    x = layers.Dense(128, activation='relu')(decoder_input)
    x = layers.Dense(np.prod(input_shape), activation='sigmoid')(x) # Output
matches original input size
    decoder_output = layers.Reshape(input_shape)(x)
    decoder = models.Model(decoder_input, decoder_output, name="decoder")

    # Autoencoder
    autoencoder_input = tf.keras.Input(shape=input_shape,
name="autoencoder_input")
    encoded_data = encoder(autoencoder_input)
    decoded_data = decoder(encoded_data)
    autoencoder = models.Model(autoencoder_input, decoded_data,
name="autoencoder")

    return autoencoder, encoder, decoder

if __name__ == "__main__":
    # Load Fashion MNIST dataset
    (x_train, _), (x_test, _) = datasets.fashion_mnist.load_data()

    # Normalize pixel values to [0, 1]
    x_train = x_train.astype('float32') / 255.0
    x_test = x_test.astype('float32') / 255.0

    # Add a channel dimension (for grayscale images)
    input_shape = x_train.shape[1:] # (28, 28)
    x_train = np.expand_dims(x_train, -1) # (60000, 28, 28, 1)
    x_test = np.expand_dims(x_test, -1)   # (10000, 28, 28, 1)
    input_shape = x_train.shape[1:] # (28, 28, 1)

    # Define the size of the latent space
    latent_dimension = 64

    # Create the autoencoder
    autoencoder, encoder, decoder = create_autoencoder(input_shape,
latent_dimension)

    # Compile the autoencoder
    autoencoder.compile(optimizer='adam', loss='mse') # Mean Squared Error is
common for image reconstruction

    # Print model summaries
    print("Autoencoder Summary:")
    autoencoder.summary()
    print("\nEncoder Summary:")
    encoder.summary()
    print("\nDecoder Summary:")
    decoder.summary()

    # Train the autoencoder
    print("\nTraining Autoencoder on Fashion MNIST...")
    history = autoencoder.fit(x_train, x_train,
                              epochs=20,
                              batch_size=256,
                              shuffle=True,
                              validation_data=(x_test, x_test))

    # Plot training & validation loss values
    plt.figure(figsize=(10, 5))
    plt.plot(history.history['loss'], label='Training Loss')
    plt.plot(history.history['val_loss'], label='Validation Loss')
```

```
    plt.title('Autoencoder Training Loss on Fashion MNIST')
    plt.xlabel('Epoch')
    plt.ylabel('Loss')
    plt.legend()
    plt.grid(True)
    plt.show()

    # Predict reconstructions from the test set
    reconstructed_images = autoencoder.predict(x_test)

    # Display original vs. reconstructed images
    n = 10 # How many digits to display
    plt.figure(figsize=(20, 4))
    for i in range(n):
        # Display original
        ax = plt.subplot(2, n, i + 1)
        plt.imshow(x_test[i].reshape(input_shape[:-1]), cmap='gray')
        plt.title("Original")
        plt.axis('off')

        # Display reconstruction
        ax = plt.subplot(2, n, i + 1 + n)
        plt.imshow(reconstructed_images[i].reshape(input_shape[:-1]),
cmap='gray')
        plt.title("Reconstructed")
        plt.axis('off')
    plt.suptitle('Original vs. Reconstructed Images (Fashion MNIST)')
    plt.show()
```

Input

The Fashion MNIST dataset (images of clothing articles). The code automatically downloads it.

Expected Output

1. Summaries of the autoencoder, encoder, and decoder models.
2. Training logs showing loss and validation loss decreasing over epochs.
3. A plot illustrating the training and validation loss curves.
4. A visual comparison of original Fashion MNIST images and their reconstructed versions, demonstrating the autoencoder's ability to learn and reproduce the input data.

Lab 5: Implementing a variational autoencoder using TensorFlow or PyTorch.

Title

Variational Autoencoder (VAE) Implementation with TensorFlow

Aim

To implement a Variational Autoencoder (VAE) using TensorFlow/Keras, understanding its ability to generate new data samples by learning a continuous, disentangled latent space.

Procedure

1. **Understand VAEs:** Learn about the key differences between VAEs and standard autoencoders, particularly the probabilistic encoder and the reparameterization trick.
2. **Encoder Definition:** Design the encoder to output both the mean (μ) and logarithm of variance (logσ2) of the latent distribution.
3. **Reparameterization Trick:** Implement the reparameterization trick to sample from the latent distribution, allowing gradients to flow back through the sampling process.
4. **Decoder Definition:** Design the decoder to reconstruct the input data from a sampled latent vector.
5. **Loss Function:** Define the VAE loss, which consists of two parts:
   - **Reconstruction Loss:** Measures how well the decoder reconstructs the input.
   - **KL Divergence Loss:** Regularizes the latent space to be close to a standard normal distribution, encouraging continuity and disentanglement.
6. **Model Assembly and Training:** Assemble the VAE and train it, optimizing both reconstruction and KL divergence losses.
7. **Generation and Evaluation:** Generate new samples by sampling from the latent space and passing them through the decoder. Visualize generated samples and evaluate the quality of reconstruction.

Source Code

```
import tensorflow as tf
from tensorflow.keras import layers, models, datasets, backend as K
import numpy as np
import matplotlib.pyplot as plt

# Custom sampling layer for the reparameterization trick
class Sampling(layers.Layer):
    """Uses (z_mean, z_log_var) to sample z, the vector encoding a digit."""
    def call(self, inputs):
        z_mean, z_log_var = inputs
        batch = tf.shape(z_mean)[0]
        dim = tf.shape(z_mean)[1]
        epsilon = K.random_normal(shape=(batch, dim))
        return z_mean + tf.exp(0.5 * z_log_var) * epsilon

def build_vae(input_shape, latent_dim):
    """
    Builds a Variational Autoencoder (VAE) model.

    Args:
```

```
        input_shape (tuple): Shape of the input data (e.g., (28, 28, 1)).
        latent_dim (int): Dimensionality of the latent space.

    Returns:
        tf.keras.Model: The VAE model.
        tf.keras.Model: The encoder model.
        tf.keras.Model: The decoder model.
    """
    # Encoder
    encoder_inputs = tf.keras.Input(shape=input_shape, name="encoder_inputs")
    x = layers.Flatten()(encoder_inputs)
    x = layers.Dense(128, 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)
    z = Sampling()([z_mean, z_log_var])
    encoder = models.Model(encoder_inputs, [z_mean, z_log_var, z],
name="encoder")

    # Decoder
    latent_inputs = tf.keras.Input(shape=(latent_dim,), name="z_sampling")
    x = layers.Dense(128, activation='relu')(latent_inputs)
    x = layers.Dense(np.prod(input_shape), activation='sigmoid')(x)
    decoder_outputs = layers.Reshape(input_shape)(x)
    decoder = models.Model(latent_inputs, decoder_outputs, name="decoder")

    # VAE
    outputs = decoder(encoder(encoder_inputs)[2]) # Pass the sampled 'z' from
encoder to decoder
    vae = models.Model(encoder_inputs, outputs, name="vae")

    # VAE Loss
    # Reconstruction loss (Binary Cross-Entropy for pixel values between 0 and
1)
    reconstruction_loss = tf.reduce_mean(
        tf.keras.losses.binary_crossentropy(encoder_inputs, outputs)
    )
    reconstruction_loss *= np.prod(input_shape) # Scale by image dimensions

    # KL Divergence loss
    kl_loss = -0.5 * tf.reduce_sum(1 + z_log_var - tf.square(z_mean) -
tf.exp(z_log_var), axis=-1)
    kl_loss = tf.reduce_mean(kl_loss)

    vae_loss = reconstruction_loss + kl_loss
    vae.add_loss(vae_loss)

    return vae, encoder, decoder

if __name__ == "__main__":
    # Load MNIST dataset
    (x_train, _), (x_test, _) = datasets.mnist.load_data()
    x_train = x_train.astype('float32') / 255.0
    x_test = x_test.astype('float32') / 255.0

    # Add a channel dimension
    input_shape = x_train.shape[1:]
    x_train = np.expand_dims(x_train, -1)
    x_test = np.expand_dims(x_test, -1)
    input_shape = x_train.shape[1:] # (28, 28, 1)

    latent_dimension = 2 # For easy visualization of the latent space

    # Build the VAE
    vae, encoder, decoder = build_vae(input_shape, latent_dimension)

    # Compile the VAE
    vae.compile(optimizer='adam') # Loss is added via vae.add_loss()
```

```
    # Print model summaries
    print("VAE Summary:")
    vae.summary()
    print("\nEncoder Summary:")
    encoder.summary()
    print("\nDecoder Summary:")
    decoder.summary()

    # Train the VAE
    print("\nTraining VAE...")
    history = vae.fit(x_train, epochs=20, batch_size=128,
validation_data=(x_test,))

    # Plot training loss
    plt.figure(figsize=(10, 5))
    plt.plot(history.history['loss'], label='Training Loss')
    plt.plot(history.history['val_loss'], label='Validation Loss')
    plt.title('VAE Training Loss')
    plt.xlabel('Epoch')
    plt.ylabel('Total Loss (Reconstruction + KL)')
    plt.legend()
    plt.grid(True)
    plt.show()

    # --- Visualize Latent Space (for latent_dim=2) ---
    if latent_dimension == 2:
        z_mean, z_log_var, z = encoder.predict(x_test)
        plt.figure(figsize=(10, 8))
        plt.scatter(z_mean[:, 0], z_mean[:, 1], c=_, cmap='viridis', s=5) # Use
_ from x_test, _
        plt.colorbar()
        plt.xlabel("z[0]")
        plt.ylabel("z[1]")
        plt.title("Latent Space of VAE (MNIST Digits)")
        plt.show()

    # --- Generate new images from latent space ---
    print("\nGenerating new images from latent space...")
    n = 10 # Number of images to generate
    # Sample points from the latent space (e.g., a grid for 2D latent space)
    if latent_dimension == 2:
        grid_x = np.linspace(-1.5, 1.5, n)
        grid_y = np.linspace(-1.5, 1.5, n)
        figure = np.zeros((input_shape[0] * n, input_shape[1] * n,
input_shape[2]))
        for i, yi in enumerate(grid_x):
            for j, xi in enumerate(grid_y):
                z_sample = np.array([[xi, yi]])
                x_decoded = decoder.predict(z_sample)
                digit = x_decoded[0].reshape(input_shape)
                figure[i * input_shape[0]: (i + 1) * input_shape[0],
                       j * input_shape[1]: (j + 1) * input_shape[1]] = digit

        plt.figure(figsize=(10, 10))
        start_range = input_shape[0] // 2
        end_range = n * input_shape[0] + start_range + 1
        pixel_range = np.arange(start_range, end_range, input_shape[0])
        sample_range_x = np.round(grid_x, 1)
        sample_range_y = np.round(grid_y, 1)
        plt.xticks(pixel_range, sample_range_x)
        plt.yticks(pixel_range, sample_range_y)
        plt.xlabel("z[0]")
        plt.ylabel("z[1]")
        plt.imshow(figure[:, :, 0], cmap='gray')
        plt.title("Generated Images from Latent Space (VAE)")
        plt.show()
```

```
    else:
        # For higher dimensions, sample random points
        random_latent_vectors = tf.random.normal(shape=(n, latent_dimension))
        generated_images = decoder.predict(random_latent_vectors)
        plt.figure(figsize=(10, 2))
        for i in range(n):
            ax = plt.subplot(1, n, i + 1)
            plt.imshow(generated_images[i].reshape(input_shape[:-1]),
cmap='gray')
            plt.axis('off')
        plt.suptitle("Generated Images from Random Latent Samples")
        plt.show()
```

Input

The MNIST dataset (handwritten digits). The code automatically downloads it.

Expected Output

1. Summaries of the VAE, encoder, and decoder models.
2. Training logs showing the combined VAE loss decreasing over epochs.
3. A plot of the training and validation loss curves.
4. If `latent_dimension` is 2, a scatter plot of the latent space, showing how different digits cluster.
5. A grid of newly generated images, demonstrating the VAE's generative capabilities by sampling from the learned latent space.

Lab 6: VAEs for anomaly detection in datasets

Title

Anomaly Detection using Variational Autoencoders (VAEs)

Aim

To apply a Variational Autoencoder (VAE) for anomaly detection in a dataset by leveraging its ability to learn the distribution of "normal" data and identify deviations.

Procedure

1. **Dataset Preparation:** Obtain a dataset containing both normal and anomalous samples (e.g., a subset of MNIST as normal, and Fashion MNIST as anomalous, or a dedicated anomaly detection dataset).
2. **Train VAE on Normal Data:** Train the VAE exclusively on the "normal" data samples. The VAE will learn to efficiently encode and reconstruct these normal patterns.
3. **Anomaly Scoring:** For a given data point (normal or anomalous), calculate an "anomaly score". Common methods include:
   - **Reconstruction Error:** Higher reconstruction error (difference between original and reconstructed) indicates a higher likelihood of anomaly.
   - **Latent Space Distance:** Distance of the latent representation from the mean of the normal latent space.
   - **Combined Loss:** Using the VAE's total loss (reconstruction + KL divergence) as the score.
4. **Thresholding:** Set a threshold for the anomaly score. Data points exceeding this threshold are classified as anomalies.
5. **Evaluation:** Evaluate the anomaly detection performance using metrics like precision, recall, F1-score, or ROC curves, comparing the model's predictions against true labels.

Source Code

```
import tensorflow as tf
from tensorflow.keras import layers, models, datasets, backend as K
import numpy as np
import matplotlib.pyplot as plt
from sklearn.metrics import roc_curve, auc
from sklearn.model_selection import train_test_split

# Custom sampling layer for the reparameterization trick (from Lab 5)
class Sampling(layers.Layer):
    def call(self, inputs):
        z_mean, z_log_var = inputs
        batch = tf.shape(z_mean)[0]
        dim = tf.shape(z_mean)[1]
        epsilon = K.random_normal(shape=(batch, dim))
        return z_mean + tf.exp(0.5 * z_log_var) * epsilon

def build_vae_anomaly(input_shape, latent_dim):
    """
    Builds a Variational Autoencoder (VAE) model for anomaly detection.
    """
    # Encoder
    encoder_inputs = tf.keras.Input(shape=input_shape, name="encoder_inputs")
```

```
    x = layers.Flatten()(encoder_inputs)
    x = layers.Dense(128, 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)
    z = Sampling()([z_mean, z_log_var])
    encoder = models.Model(encoder_inputs, [z_mean, z_log_var, z],
name="encoder")

    # Decoder
    latent_inputs = tf.keras.Input(shape=(latent_dim,), name="z_sampling")
    x = layers.Dense(128, activation='relu')(latent_inputs)
    x = layers.Dense(np.prod(input_shape), activation='sigmoid')(x)
    decoder_outputs = layers.Reshape(input_shape)(x)
    decoder = models.Model(latent_inputs, decoder_outputs, name="decoder")

    # VAE
    outputs = decoder(encoder(encoder_inputs)[2])
    vae = models.Model(encoder_inputs, outputs, name="vae")

    # VAE Loss (reconstruction + KL divergence)
    reconstruction_loss = tf.reduce_mean(
        tf.keras.losses.binary_crossentropy(encoder_inputs, outputs)
    )
    reconstruction_loss *= np.prod(input_shape)

    kl_loss = -0.5 * tf.reduce_sum(1 + z_log_var - tf.square(z_mean) -
tf.exp(z_log_var), axis=-1)
    kl_loss = tf.reduce_mean(kl_loss)

    vae_loss = reconstruction_loss + kl_loss
    vae.add_loss(vae_loss)

    return vae, encoder, decoder, reconstruction_loss, kl_loss

if __name__ == "__main__":
    # --- Dataset Preparation for Anomaly Detection ---
    # We'll use MNIST digits 0-4 as 'normal' and digits 5-9 as 'anomalous'
    (x_train_mnist, y_train_mnist), (x_test_mnist, y_test_mnist) =
datasets.mnist.load_data()

    # Normalize and reshape
    x_train_mnist = x_train_mnist.astype('float32') / 255.0
    x_test_mnist = x_test_mnist.astype('float32') / 255.0
    input_shape = x_train_mnist.shape[1:]
    x_train_mnist = np.expand_dims(x_train_mnist, -1)
    x_test_mnist = np.expand_dims(x_test_mnist, -1)
    input_shape = x_train_mnist.shape[1:] # (28, 28, 1)

    # Define 'normal' data (digits 0-4)
    normal_indices_train = np.where(y_train_mnist < 5)[0]
    x_normal_train = x_train_mnist[normal_indices_train]

    # Create a test set with both normal and anomalous data
    # Normal test data (digits 0-4)
    normal_indices_test = np.where(y_test_mnist < 5)[0]
    x_normal_test = x_test_mnist[normal_indices_test]
    y_normal_test = np.zeros(len(x_normal_test)) # Label normal as 0

    # Anomalous test data (digits 5-9)
    anomaly_indices_test = np.where(y_test_mnist >= 5)[0]
    x_anomaly_test = x_test_mnist[anomaly_indices_test]
    y_anomaly_test = np.ones(len(x_anomaly_test)) # Label anomalous as 1

    # Combine for evaluation
    x_test_combined = np.concatenate([x_normal_test, x_anomaly_test])
    y_test_combined = np.concatenate([y_normal_test, y_anomaly_test])
```

```
    # Shuffle the combined test set
    p = np.random.permutation(len(x_test_combined))
    x_test_combined = x_test_combined[p]
    y_test_combined = y_test_combined[p]

    latent_dimension = 32

    # Build the VAE
    vae, encoder, decoder, reconstruction_loss_fn, kl_loss_fn =
build_vae_anomaly(input_shape, latent_dimension)
    vae.compile(optimizer='adam')

    print("VAE Summary (for Anomaly Detection):")
    vae.summary()

    # Train the VAE only on 'normal' data
    print("\nTraining VAE on NORMAL data (MNIST digits 0-4)...")
    history = vae.fit(x_normal_train, epochs=20, batch_size=128,
validation_split=0.1)

    # Plot training loss
    plt.figure(figsize=(10, 5))
    plt.plot(history.history['loss'], label='Training Loss')
    plt.plot(history.history['val_loss'], label='Validation Loss')
    plt.title('VAE Training Loss on Normal Data')
    plt.xlabel('Epoch')
    plt.ylabel('Total Loss (Reconstruction + KL)')
    plt.legend()
    plt.grid(True)
    plt.show()

    # --- Anomaly Scoring ---
    # Calculate reconstruction errors for the combined test set
    reconstructed_test_images = vae.predict(x_test_combined)

    # Calculate pixel-wise binary cross-entropy as reconstruction error
    # We need to compute this manually as it's part of the VAE's internal loss
calculation
    # and not directly exposed as a metric during predict()
    reconstruction_errors = tf.reduce_mean(
        tf.keras.losses.binary_crossentropy(x_test_combined,
reconstructed_test_images),
        axis=(1, 2, 3) # Sum over height, width, channels for each image
    ).numpy()
    # For a more robust score, you might also consider the KL divergence part or
a combination.
    # For simplicity, we'll use reconstruction error here.
    anomaly_scores = reconstruction_errors

    # --- Evaluation ---
    # Plot distribution of anomaly scores for normal vs. anomalous data
    normal_scores = anomaly_scores[y_test_combined == 0]
    anomaly_scores_actual = anomaly_scores[y_test_combined == 1]

    plt.figure(figsize=(10, 6))
    plt.hist(normal_scores, bins=50, alpha=0.5, label='Normal Data Scores',
color='blue')
    plt.hist(anomaly_scores_actual, bins=50, alpha=0.5, label='Anomalous Data
Scores', color='red')
    plt.title('Distribution of Anomaly Scores')
    plt.xlabel('Reconstruction Error (Anomaly Score)')
    plt.ylabel('Frequency')
    plt.legend()
    plt.grid(True)
    plt.show()

    # Calculate ROC curve and AUC
```

```
    fpr, tpr, thresholds = roc_curve(y_test_combined, anomaly_scores)
    roc_auc = auc(fpr, tpr)

    plt.figure(figsize=(8, 8))
    plt.plot(fpr, tpr, color='darkorange', lw=2, label=f'ROC curve (area =
{roc_auc:.2f})')
    plt.plot([0, 1], [0, 1], color='navy', lw=2, linestyle='--')
    plt.xlim([0.0, 1.0])
    plt.ylim([0.0, 1.05])
    plt.xlabel('False Positive Rate')
    plt.ylabel('True Positive Rate')
    plt.title('Receiver Operating Characteristic (ROC) Curve')
    plt.legend(loc="lower right")
    plt.grid(True)
    plt.show()

    # Determine an optimal threshold (e.g., using Youden's J statistic)
    optimal_idx = np.argmax(tpr - fpr)
    optimal_threshold = thresholds[optimal_idx]
    print(f"\nOptimal Anomaly Threshold (based on Youden's J):
{optimal_threshold:.4f}")

    # Classify based on the optimal threshold
    predicted_anomalies = (anomaly_scores >= optimal_threshold).astype(int)
    from sklearn.metrics import classification_report
    print("\nClassification Report:")
    print(classification_report(y_test_combined, predicted_anomalies,
target_names=['Normal', 'Anomaly']))

    # Visualize some anomalies and normal reconstructions
    n_display = 5
    normal_sample_indices = np.where((y_test_combined == 0) &
(predicted_anomalies == 0))[0][:n_display]
    anomaly_sample_indices = np.where((y_test_combined == 1) &
(predicted_anomalies == 1))[0][:n_display]

    plt.figure(figsize=(20, 8))
    for i, idx in enumerate(normal_sample_indices):
        # Original Normal
        ax = plt.subplot(4, n_display, i + 1)
        plt.imshow(x_test_combined[idx].reshape(input_shape[:-1]), cmap='gray')
        plt.title("Orig Normal")
        plt.axis('off')
        # Reconstructed Normal
        ax = plt.subplot(4, n_display, i + 1 + n_display)
        plt.imshow(reconstructed_test_images[idx].reshape(input_shape[:-1]),
cmap='gray')
        plt.title(f"Recon Normal\nScore: {anomaly_scores[idx]:.2f}")
        plt.axis('off')

    for i, idx in enumerate(anomaly_sample_indices):
        # Original Anomaly
        ax = plt.subplot(4, n_display, i + 1 + 2 * n_display)
        plt.imshow(x_test_combined[idx].reshape(input_shape[:-1]), cmap='gray')
        plt.title("Orig Anomaly")
        plt.axis('off')
        # Reconstructed Anomaly
        ax = plt.subplot(4, n_display, i + 1 + 3 * n_display)
        plt.imshow(reconstructed_test_images[idx].reshape(input_shape[:-1]),
cmap='gray')
        plt.title(f"Recon Anomaly\nScore: {anomaly_scores[idx]:.2f}")
        plt.axis('off')
    plt.suptitle('Normal vs. Anomalous Data Reconstruction (Anomaly Detection)')
    plt.tight_layout(rect=[0, 0.03, 1, 0.95])
    plt.show()
```

Input

MNIST dataset, where digits 0-4 are considered "normal" and digits 5-9 are considered "anomalous".

Expected Output

1. Summary of the VAE model.
2. Training loss plot for the VAE trained on normal data.
3. Histograms showing the distribution of anomaly scores for both normal and anomalous data, ideally with a clear separation.
4. An ROC curve and AUC score, indicating the model's ability to discriminate between normal and anomalous data.
5. An optimal anomaly threshold and a classification report (precision, recall, F1-score) based on this threshold.
6. Visual examples of original and reconstructed normal and anomalous images, showing that anomalous images are typically reconstructed poorly, leading to higher anomaly scores.

Lab 7: GAN model using TensorFlow or PyTorch.

Title

Implementing a Basic Generative Adversarial Network (GAN) with TensorFlow

Aim

To implement a complete Generative Adversarial Network (GAN) from scratch using TensorFlow/Keras, understanding the adversarial training process between a generator and a discriminator for synthetic data generation.

Procedure

1. **Understand GAN Components:** Define the roles of the Generator (creates fake data) and Discriminator (distinguishes real from fake data).
2. **Generator Architecture:** Design the generator network, which takes a random noise vector as input and outputs data (e.g., images) that should resemble the real data distribution.
3. **Discriminator Architecture:** Design the discriminator network, which takes data (real or fake) as input and outputs a probability indicating whether the input is real or fake.
4. **Loss Functions:** Define the loss functions for both the generator and discriminator.
   - **Discriminator Loss:** Binary cross-entropy, aiming to correctly classify real data as real and fake data as fake.
   - **Generator Loss:** Binary cross-entropy, aiming to fool the discriminator into classifying its generated fake data as real.
5. **Training Loop:** Implement the adversarial training loop:
   - Train the discriminator: On real images (label 1) and generated fake images (label 0).
   - Train the generator: On generated fake images (label 1, to fool the discriminator).
   - Alternate between training the discriminator and generator.
6. **Evaluation and Visualization:** Periodically generate images during training to observe the generator's progress. Evaluate the quality of generated images.

Source Code

```
import tensorflow as tf
from tensorflow.keras import layers, models, optimizers, losses, datasets
import numpy as np
import matplotlib.pyplot as plt
import os

# Define the Generator
def build_generator(latent_dim):
    model = models.Sequential(name="generator")
    # Foundation for 7x7 image
    model.add(layers.Dense(7 * 7 * 128, use_bias=False,
input_shape=(latent_dim,)))
    model.add(layers.BatchNormalization())
    model.add(layers.LeakyReLU())

    model.add(layers.Reshape((7, 7, 128)))
    # assert model.output_shape == (None, 7, 7, 128) # None is for batch size

    # Upsample to 14x14
```

```
    model.add(layers.Conv2DTranspose(64, (5, 5), strides=(2, 2), padding='same',
use_bias=False))
    model.add(layers.BatchNormalization())
    model.add(layers.LeakyReLU())
    # assert model.output_shape == (None, 14, 14, 64)

    # Upsample to 28x28
    model.add(layers.Conv2DTranspose(1, (5, 5), strides=(2, 2), padding='same',
use_bias=False, activation='tanh'))
    # assert model.output_shape == (None, 28, 28, 1)

    return model

# Define the Discriminator
def build_discriminator(input_shape):
    model = models.Sequential(name="discriminator")
    model.add(layers.Conv2D(64, (5, 5), strides=(2, 2), padding='same',
input_shape=input_shape))
    model.add(layers.LeakyReLU())
    model.add(layers.Dropout(0.3))

    model.add(layers.Conv2D(128, (5, 5), strides=(2, 2), padding='same'))
    model.add(layers.LeakyReLU())
    model.add(layers.Dropout(0.3))

    model.add(layers.Flatten())
    model.add(layers.Dense(1, activation='sigmoid')) # Output a probability
(real or fake)

    return model

# Define GAN training step
@tf.function
def train_step(images, generator, discriminator, generator_optimizer,
discriminator_optimizer, latent_dim, batch_size):
    noise = tf.random.normal([batch_size, latent_dim])

    with tf.GradientTape() as gen_tape, tf.GradientTape() as disc_tape:
        generated_images = generator(noise, training=True)

        real_output = discriminator(images, training=True)
        fake_output = discriminator(generated_images, training=True)

        # Discriminator Loss
        real_loss =
losses.BinaryCrossentropy(from_logits=False)(tf.ones_like(real_output),
real_output)
        fake_loss =
losses.BinaryCrossentropy(from_logits=False)(tf.zeros_like(fake_output),
fake_output)
        discriminator_loss = real_loss + fake_loss

        # Generator Loss
        generator_loss =
losses.BinaryCrossentropy(from_logits=False)(tf.ones_like(fake_output),
fake_output) # Generator wants fake to be classified as real

    # Calculate gradients
    gradients_of_discriminator = disc_tape.gradient(discriminator_loss,
discriminator.trainable_variables)
    gradients_of_generator = gen_tape.gradient(generator_loss,
generator.trainable_variables)

    # Apply gradients
    discriminator_optimizer.apply_gradients(zip(gradients_of_discriminator,
discriminator.trainable_variables))
```

```
    generator_optimizer.apply_gradients(zip(gradients_of_generator,
generator.trainable_variables))

    return generator_loss, discriminator_loss

def train_gan(dataset, epochs, generator, discriminator, generator_optimizer,
discriminator_optimizer, latent_dim, batch_size, sample_interval=1):
    seed = tf.random.normal([16, latent_dim]) # Fixed noise for visualization

    for epoch in range(epochs):
        gen_losses = []
        disc_losses = []
        for image_batch in dataset:
            g_loss, d_loss = train_step(image_batch, generator, discriminator,
generator_optimizer, discriminator_optimizer, latent_dim, batch_size)
            gen_losses.append(g_loss)
            disc_losses.append(d_loss)

        avg_gen_loss = tf.reduce_mean(gen_losses)
        avg_disc_loss = tf.reduce_mean(disc_losses)
        print(f"Epoch {epoch+1}/{epochs}, Generator Loss: {avg_gen_loss:.4f},
Discriminator Loss: {avg_disc_loss:.4f}")

        # Generate and save images at intervals
        if (epoch + 1) % sample_interval == 0:
            generate_and_save_images(generator, epoch + 1, seed)

def generate_and_save_images(model, epoch, test_input):
    predictions = model(test_input, training=False)
    # Scale images to [0, 1] for display if using tanh output [-1, 1]
    predictions = (predictions + 1) / 2.0

    fig = plt.figure(figsize=(4, 4))
    for i in range(predictions.shape[0]):
        plt.subplot(4, 4, i+1)
        plt.imshow(predictions[i, :, :, 0], cmap='gray')
        plt.axis('off')
    plt.suptitle(f"Generated Images - Epoch {epoch}")
    # Create a directory for saving images if it doesn't exist
    if not os.path.exists('gan_generated_images'):
        os.makedirs('gan_generated_images')
    plt.savefig(f'gan_generated_images/image_at_epoch_{epoch:04d}.png')
    plt.close(fig) # Close the figure to free memory

if __name__ == "__main__":
    # Load and preprocess MNIST dataset
    (x_train, _), (_, _) = datasets.mnist.load_data()
    x_train = x_train.astype('float32')
    # Normalize images to [-1, 1] for better GAN training with tanh output
    x_train = (x_train - 127.5) / 127.5
    x_train = np.expand_dims(x_train, -1) # Add channel dimension

    # Batch and shuffle the data
    BUFFER_SIZE = 60000
    BATCH_SIZE = 256
    train_dataset =
tf.data.Dataset.from_tensor_slices(x_train).shuffle(BUFFER_SIZE).batch(BATCH_SIZ
E)

    # Define hyperparameters
    LATENT_DIM = 100
    EPOCHS = 50 # You might need more epochs for better results
    LEARNING_RATE = 1e-4

    # Build Generator and Discriminator
    generator = build_generator(LATENT_DIM)
    discriminator = build_discriminator(x_train.shape[1:])
```

```
    # Define optimizers
    generator_optimizer = optimizers.Adam(learning_rate=LEARNING_RATE)
    discriminator_optimizer = optimizers.Adam(learning_rate=LEARNING_RATE)

    # Print model summaries
    print("Generator Summary:")
    generator.summary()
    print("\nDiscriminator Summary:")
    discriminator.summary()

    # Train the GAN
    print("\nStarting GAN Training...")
    train_gan(train_dataset, EPOCHS, generator, discriminator,
generator_optimizer, discriminator_optimizer, LATENT_DIM, BATCH_SIZE)

    print("\nGAN Training Complete.")
    print("Generated images saved in 'gan_generated_images' directory.")

    # Display final generated images
    final_seed = tf.random.normal([16, LATENT_DIM])
    final_generated_images = generator(final_seed, training=False)
    final_generated_images = (final_generated_images + 1) / 2.0 # Scale to [0,
1]

    plt.figure(figsize=(4, 4))
    for i in range(final_generated_images.shape[0]):
        plt.subplot(4, 4, i+1)
        plt.imshow(final_generated_images[i, :, :, 0], cmap='gray')
        plt.axis('off')
    plt.suptitle("Final Generated Images")
    plt.show()
```

Input

The MNIST dataset (handwritten digits). The code automatically downloads and preprocesses it. Random noise vectors are generated internally as input to the generator.

Expected Output

1. Summaries of the generator and discriminator models.
2. Epoch-wise training logs showing the generator and discriminator losses. Ideally, both losses will fluctuate as the models compete.
3. A series of saved image files (e.g., `image_at_epoch_0001.png`, `image_at_epoch_0002.png`, etc.) in a `gan_generated_images` directory, showing the progression of image quality as the GAN trains.
4. A final plot displaying a grid of newly generated synthetic images. As training progresses, these images should increasingly resemble handwritten digits from the MNIST dataset.

Lab 8: Implementing a DCGAN for image generation

Title

Deep Convolutional Generative Adversarial Network (DCGAN) for Image Generation

Aim

To implement a Deep Convolutional Generative Adversarial Network (DCGAN) using TensorFlow/Keras, focusing on using convolutional layers for both the generator and discriminator to produce higher-quality synthetic images.

Procedure

1. **Understand DCGAN Principles:** Learn about the key architectural guidelines for stable GAN training, such as using convolutional layers, batch normalization, and specific activation functions.
2. **Generator Architecture:** Design the generator using `Conv2DTranspose` (deconvolutional) layers to upsample from a latent vector to an image. Include Batch Normalization and LeakyReLU activations.
3. **Discriminator Architecture:** Design the discriminator using `Conv2D` layers to downsample an image to a single probability. Include Batch Normalization (except for the first layer) and LeakyReLU activations.
4. **Loss Functions and Optimizers:** Use Binary Cross-Entropy for both generator and discriminator losses. Use Adam optimizers with specific learning rates.
5. **Training Loop:** Implement the adversarial training loop similar to a basic GAN, but with the convolutional architectures.
6. **Evaluation and Visualization:** Monitor training progress by periodically generating and visualizing images. Assess the visual quality and diversity of the generated images.

Source Code

```
import tensorflow as tf
from tensorflow.keras import layers, models, optimizers, losses, datasets
import numpy as np
import matplotlib.pyplot as plt
import os

# Define the DCGAN Generator
def build_dcgan_generator(latent_dim):
    model = models.Sequential(name="dcgan_generator")
    # Project and reshape
    model.add(layers.Dense(7 * 7 * 256, use_bias=False,
input_shape=(latent_dim,)))
    model.add(layers.BatchNormalization())
    model.add(layers.LeakyReLU())
    model.add(layers.Reshape((7, 7, 256)))
    # assert model.output_shape == (None, 7, 7, 256)

    # First upsample block
    model.add(layers.Conv2DTranspose(128, (5, 5), strides=(1, 1),
padding='same', use_bias=False))
    model.add(layers.BatchNormalization())
    model.add(layers.LeakyReLU())
    # assert model.output_shape == (None, 7, 7, 128)
```

```
    # Second upsample block
    model.add(layers.Conv2DTranspose(64, (5, 5), strides=(2, 2), padding='same',
use_bias=False))
    model.add(layers.BatchNormalization())
    model.add(layers.LeakyReLU())
    # assert model.output_shape == (None, 14, 14, 64)

    # Output layer
    model.add(layers.Conv2DTranspose(1, (5, 5), strides=(2, 2), padding='same',
use_bias=False, activation='tanh'))
    # assert model.output_shape == (None, 28, 28, 1)

    return model

# Define the DCGAN Discriminator
def build_dcgan_discriminator(input_shape):
    model = models.Sequential(name="dcgan_discriminator")
    # First convolutional block
    model.add(layers.Conv2D(64, (5, 5), strides=(2, 2), padding='same',
input_shape=input_shape))
    model.add(layers.LeakyReLU())
    model.add(layers.Dropout(0.3))

    # Second convolutional block
    model.add(layers.Conv2D(128, (5, 5), strides=(2, 2), padding='same'))
    model.add(layers.BatchNormalization()) # DCGAN guidelines suggest no BN on
first discriminator layer
    model.add(layers.LeakyReLU())
    model.add(layers.Dropout(0.3))

    # Output layer
    model.add(layers.Flatten())
    model.add(layers.Dense(1, activation='sigmoid')) # Output probability of
real/fake

    return model

# Define GAN training step (reusing from Lab 7, as the core logic is similar)
@tf.function
def train_step_dcgan(images, generator, discriminator, generator_optimizer,
discriminator_optimizer, latent_dim, batch_size):
    noise = tf.random.normal([batch_size, latent_dim])

    with tf.GradientTape() as gen_tape, tf.GradientTape() as disc_tape:
        generated_images = generator(noise, training=True)

        real_output = discriminator(images, training=True)
        fake_output = discriminator(generated_images, training=True)

        # Discriminator Loss
        real_loss =
losses.BinaryCrossentropy(from_logits=False)(tf.ones_like(real_output),
real_output)
        fake_loss =
losses.BinaryCrossentropy(from_logits=False)(tf.zeros_like(fake_output),
fake_output)
        discriminator_loss = real_loss + fake_loss

        # Generator Loss
        generator_loss =
losses.BinaryCrossentropy(from_logits=False)(tf.ones_like(fake_output),
fake_output)

    # Calculate gradients
    gradients_of_discriminator = disc_tape.gradient(discriminator_loss,
discriminator.trainable_variables)
```

```
    gradients_of_generator = gen_tape.gradient(generator_loss,
generator.trainable_variables)

    # Apply gradients
    discriminator_optimizer.apply_gradients(zip(gradients_of_discriminator,
discriminator.trainable_variables))
    generator_optimizer.apply_gradients(zip(gradients_of_generator,
generator.trainable_variables))

    return generator_loss, discriminator_loss

def train_dcgan(dataset, epochs, generator, discriminator, generator_optimizer,
discriminator_optimizer, latent_dim, batch_size, sample_interval=1):
    seed = tf.random.normal([16, latent_dim]) # Fixed noise for visualization

    for epoch in range(epochs):
        gen_losses = []
        disc_losses = []
        for image_batch in dataset:
            g_loss, d_loss = train_step_dcgan(image_batch, generator,
discriminator, generator_optimizer, discriminator_optimizer, latent_dim,
batch_size)
            gen_losses.append(g_loss)
            disc_losses.append(d_loss)

        avg_gen_loss = tf.reduce_mean(gen_losses)
        avg_disc_loss = tf.reduce_mean(disc_losses)
        print(f"Epoch {epoch+1}/{epochs}, Generator Loss: {avg_gen_loss:.4f},
Discriminator Loss: {avg_disc_loss:.4f}")

        if (epoch + 1) % sample_interval == 0:
            generate_and_save_images_dcgan(generator, epoch + 1, seed)

def generate_and_save_images_dcgan(model, epoch, test_input):
    predictions = model(test_input, training=False)
    predictions = (predictions + 1) / 2.0 # Scale from [-1, 1] to [0, 1]

    fig = plt.figure(figsize=(4, 4))
    for i in range(predictions.shape[0]):
        plt.subplot(4, 4, i+1)
        plt.imshow(predictions[i, :, :, 0], cmap='gray')
        plt.axis('off')
    plt.suptitle(f"DCGAN Generated Images - Epoch {epoch}")
    if not os.path.exists('dcgan_generated_images'):
        os.makedirs('dcgan_generated_images')
    plt.savefig(f'dcgan_generated_images/image_at_epoch_{epoch:04d}.png')
    plt.close(fig)

if __name__ == "__main__":
    # Load and preprocess Fashion MNIST dataset (more complex than MNIST)
    (x_train, _), (_, _) = datasets.fashion_mnist.load_data()
    x_train = x_train.astype('float32')
    x_train = (x_train - 127.5) / 127.5 # Normalize to [-1, 1]
    x_train = np.expand_dims(x_train, -1) # Add channel dimension

    BUFFER_SIZE = 60000
    BATCH_SIZE = 256
    train_dataset =
tf.data.Dataset.from_tensor_slices(x_train).shuffle(BUFFER_SIZE).batch(BATCH_SIZ
E)

    LATENT_DIM = 100
    EPOCHS = 100 # DCGANs often require more epochs for good results
    GENERATOR_LR = 1e-4
    DISCRIMINATOR_LR = 1e-4

    generator = build_dcgan_generator(LATENT_DIM)
```

```
    discriminator = build_dcgan_discriminator(x_train.shape[1:])

    generator_optimizer = optimizers.Adam(learning_rate=GENERATOR_LR,
beta_1=0.5) # Beta_1=0.5 is common for GANs
    discriminator_optimizer = optimizers.Adam(learning_rate=DISCRIMINATOR_LR,
beta_1=0.5)

    print("DCGAN Generator Summary:")
    generator.summary()
    print("\nDCGAN Discriminator Summary:")
    discriminator.summary()

    print("\nStarting DCGAN Training...")
    train_dcgan(train_dataset, EPOCHS, generator, discriminator,
generator_optimizer, discriminator_optimizer, LATENT_DIM, BATCH_SIZE)

    print("\nDCGAN Training Complete.")
    print("Generated images saved in 'dcgan_generated_images' directory.")

    # Display final generated images
    final_seed = tf.random.normal([16, LATENT_DIM])
    final_generated_images = generator(final_seed, training=False)
    final_generated_images = (final_generated_images + 1) / 2.0

    plt.figure(figsize=(4, 4))
    for i in range(final_generated_images.shape[0]):
        plt.subplot(4, 4, i+1)
        plt.imshow(final_generated_images[i, :, :, 0], cmap='gray')
        plt.axis('off')
    plt.suptitle("Final DCGAN Generated Images")
    plt.show()
```

Input

The Fashion MNIST dataset. The code automatically downloads and preprocesses it. Random noise vectors are generated internally.

Expected Output

1. Summaries of the DCGAN generator and discriminator models, showing convolutional and deconvolutional layers.
2. Epoch-wise training logs indicating the generator and discriminator losses.
3. A series of saved image files in a `dcgan_generated_images` directory, demonstrating the improvement in generated image quality over training epochs.
4. A final plot displaying a grid of synthetic images that should resemble articles of clothing from the Fashion MNIST dataset, generally of higher visual quality than a simple fully-connected GAN.

Lab 9: Implementing a Progressive Growing GAN

Title

Conceptual Understanding of Progressive Growing GAN (PGGAN)

Aim

To understand the core concepts and benefits of Progressive Growing GANs (PGGANs) for generating high-resolution, high-quality images, and to explore its architectural principles.

Procedure

1. **Understand PGGAN Concept:** Learn how PGGANs train by progressively adding layers to both the generator and discriminator, starting with low-resolution images and gradually increasing resolution. This stabilizes training and improves image quality.
2. **Fading In Layers:** Understand the "fading in" mechanism, where new layers are smoothly introduced during training, preventing sudden changes that could destabilize the GAN.
3. **Architectural Components:** Identify the key components of a PGGAN, such as the use of equalized learning rates, pixel normalization, and a minibatch standard deviation layer.
4. **Training Stages:** Conceptualize the multi-stage training process, where each stage focuses on a specific resolution.
5. **Benefits:** Discuss the advantages of PGGANs, including improved training stability, faster convergence for high resolutions, and superior image quality.

Source Code

(Note: Implementing a full Progressive Growing GAN is highly complex and computationally intensive, requiring specialized hardware and extensive training. Providing a complete, runnable source code for this lab is beyond the scope of a typical lab manual and would involve thousands of lines of code. The following provides a conceptual Python structure to illustrate the idea, but it is not a runnable, complete PGGAN implementation.)

```
import tensorflow as tf
from tensorflow.keras import layers, models, optimizers
import numpy as np

# Conceptual building blocks for PGGAN
def pixel_norm(x):
    """Pixel-wise feature vector normalization."""
    epsilon = 1e-8
    return x / tf.sqrt(tf.reduce_mean(tf.square(x), axis=-1, keepdims=True) +
epsilon)

def minibatch_stddev_layer(x):
    """Adds a minibatch standard deviation layer to the discriminator."""
    batch_size, H, W, C = x.shape
    # Calculate standard deviation across batch for each feature map and spatial
location
    stddev = tf.sqrt(tf.reduce_mean(tf.square(x - tf.reduce_mean(x, axis=0,
keepdims=True)), axis=0) + 1e-8)
    # Average across all features and spatial locations
    averaged_stddev = tf.reduce_mean(stddev)
    # Replicate and concatenate to the input
```

```
    averaged_stddev =
tf.tile(tf.expand_dims(tf.expand_dims(tf.expand_dims(averaged_stddev, 0), 0),
0),
                              [batch_size, H, W, 1])
    return tf.concat([x, averaged_stddev], axis=-1)

# Conceptual Generator block for a specific resolution
def generator_block(inputs, filters, upsample=True):
    x = inputs
    if upsample:
        x = layers.UpSampling2D((2, 2))(x) # Upsample resolution
    x = layers.Conv2D(filters, (3, 3), padding='same', use_bias=False)(x)
    x = pixel_norm(x)
    x = layers.LeakyReLU(alpha=0.2)(x)
    x = layers.Conv2D(filters, (3, 3), padding='same', use_bias=False)(x)
    x = pixel_norm(x)
    x = layers.LeakyReLU(alpha=0.2)(x)
    return x

# Conceptual Discriminator block for a specific resolution
def discriminator_block(inputs, filters, downsample=True):
    x = inputs
    x = layers.Conv2D(filters, (3, 3), padding='same', use_bias=False)(x)
    x = layers.LeakyReLU(alpha=0.2)(x)
    x = layers.Conv2D(filters, (3, 3), padding='same', use_bias=False)(x)
    x = layers.LeakyReLU(alpha=0.2)(x)
    if downsample:
        x = layers.AveragePooling2D((2, 2))(x) # Downsample resolution
    return x

# Conceptual PGGAN Model (simplified, not runnable as a full PGGAN)
def build_conceptual_pggans(latent_dim, resolutions=[4, 8, 16, 32, 64, 128]):
    """
    Conceptual PGGAN architecture.
    This is a highly simplified representation and not a full working PGGAN.
    """
    # Generator
    generator_input = tf.keras.Input(shape=(latent_dim,))
    # Start with a base layer for the lowest resolution
    x_gen = layers.Dense(4 * 4 * 512, use_bias=False)(generator_input)
    x_gen = layers.Reshape((4, 4, 512))(x_gen)
    x_gen = pixel_norm(x_gen)
    x_gen = layers.LeakyReLU(alpha=0.2)(x_gen)

    # To RGB layer (for the current resolution)
    to_rgb_layers = []
    to_rgb_layers.append(layers.Conv2D(3, (1, 1), padding='same',
activation='tanh')) # 4x4

    # Add progressive blocks
    for i, res in enumerate(resolutions[1:]): # Start from 8x8
        filters = max(4, 512 // (2 ** (i + 1))) # Example filter reduction
        x_gen = generator_block(x_gen, filters, upsample=True)
        to_rgb_layers.append(layers.Conv2D(3, (1, 1), padding='same',
activation='tanh'))

    # Discriminator
    discriminator_input = tf.keras.Input(shape=(resolutions[-1], resolutions[-
1], 3)) # Max resolution
    # From RGB layer
    from_rgb_layers = []
    from_rgb_layers.append(layers.Conv2D(512, (1, 1), padding='same')) # Max
resolution

    x_disc = from_rgb_layers[-1](discriminator_input)
    x_disc = layers.LeakyReLU(alpha=0.2)(x_disc)
```

```
    x_disc = minibatch_stddev_layer(x_disc) # Add minibatch stddev to highest
resolution

    # Add progressive blocks (in reverse order for discriminator)
    for i, res in reversed(list(enumerate(resolutions[:-1]))): # From 64x64 down
to 4x4
        filters = max(4, 512 // (2 ** i))
        x_disc = discriminator_block(x_disc, filters, downsample=True)
        from_rgb_layers.insert(0, layers.Conv2D(filters, (1, 1),
padding='same'))

    # Final output for discriminator
    x_disc = layers.Flatten()(x_disc)
    discriminator_output = layers.Dense(1, activation='sigmoid')(x_disc)

    # The actual PGGAN training involves dynamically changing the model based on
resolution
    # and fading in layers. This conceptual code cannot fully represent that.
    # It would involve custom Keras models and training loops.

    print("Conceptual PGGAN Generator (highest resolution):")
    # This is just a static build for the highest resolution
    temp_gen = models.Model(generator_input, to_rgb_layers[-1](x_gen))
    temp_gen.summary()

    print("\nConceptual PGGAN Discriminator (highest resolution):")
    # This is just a static build for the highest resolution
    temp_disc = models.Model(discriminator_input, discriminator_output)
    temp_disc.summary()

    print("\nNote: A full PGGAN implementation requires dynamic model building
and training stages.")
    print("This code provides conceptual building blocks and summaries for the
highest resolution.")

if __name__ == "__main__":
    LATENT_DIM = 512 # Common latent dimension for PGGANs
    build_conceptual_pggans(LATENT_DIM)
```

Input

Random noise vectors (e.g., 512-dimensional for a PGGAN). Real image datasets (e.g., CelebA-HQ, FFHQ) are used for training.

Expected Output

1. Conceptual summaries of the generator and discriminator models, illustrating the layers involved in building up to the highest resolution.
2. A textual explanation of the progressive growing training process, emphasizing the gradual increase in resolution and the "fading in" of new layers.
3. A discussion of the benefits of PGGANs, such as improved training stability and the generation of high-resolution, visually realistic images.
4. (No actual image generation from this conceptual code, as it's not a full implementation).

Lab 10: Fine-tuning GPT for Text Generation.

Title

Fine-tuning a GPT Model for Custom Text Generation

Aim

To understand the process of fine-tuning a pre-trained GPT (Generative Pre-trained Transformer) model on a specific dataset to adapt its text generation capabilities to a particular domain or style.

Procedure

1. **Dataset Preparation:** Obtain or create a dataset relevant to the desired text generation task. This dataset should consist of examples of the text style or content you want the GPT model to learn. Format the data appropriately (e.g., plain text, or prompt-completion pairs).
2. **Choose a Pre-trained GPT Model:** Select a suitable pre-trained GPT model (e.g., GPT-2, or a smaller variant of a larger model if resources are limited).
3. **Tokenization:** Tokenize the prepared dataset using the tokenizer associated with the chosen GPT model. This converts text into numerical tokens that the model understands.
4. **Model Loading:** Load the pre-trained GPT model.
5. **Fine-tuning Configuration:** Define training parameters such as learning rate, batch size, number of epochs, and optimization strategy.
6. **Training:** Fine-tune the GPT model on your custom dataset. During this phase, the model's weights are adjusted to minimize a language modeling loss (e.g., cross-entropy) on your specific data.
7. **Evaluation and Generation:** After fine-tuning, evaluate the model's performance on unseen text. Generate new text samples using prompts relevant to your fine-tuning data to assess the model's acquired style or knowledge.

Source Code

(Note: Fine-tuning large GPT models requires significant computational resources (GPUs, TPUs) and time. The following code provides a conceptual framework using the Hugging Face `transformers` library, which is widely used for this purpose. It uses a small model for demonstration, but actual fine-tuning often involves larger models and more extensive datasets.)

```
# Install necessary libraries if not already installed:
# pip install transformers datasets accelerate

import torch
from transformers import AutoTokenizer, AutoModelForCausalLM, Trainer,
TrainingArguments
from datasets import Dataset # Hugging Face datasets library

# --- 1. Dataset Preparation (Illustrative) ---
# In a real scenario, you would load your own text data.
# For demonstration, we'll create a simple dummy dataset.
# Imagine you want to fine-tune GPT to write short, positive affirmations.
raw_data = [
    "You are capable of amazing things. Believe in yourself.",
    "Every day is a new opportunity to grow and shine.",
    "Your strength is greater than any struggle.",
    "Embrace your unique journey and celebrate your progress.",
```

```
    "You are worthy of all the good things coming your way.",
    "Positive thoughts lead to positive outcomes.",
    "Let your light shine brightly.",
    "Today is a gift, that's why it's called the present."
]

# Convert raw data to a Hugging Face Dataset object
# For causal language modeling, we typically just need a 'text' column.
dataset_dict = {'text': raw_data}
dataset = Dataset.from_dict(dataset_dict)

# --- 2. Choose a Pre-trained GPT Model and Tokenizer ---
# Using a small, accessible model for demonstration (e.g., 'gpt2')
model_name = "gpt2" # You can try 'distilgpt2' for faster training

tokenizer = AutoTokenizer.from_pretrained(model_name)
# GPT-2 tokenizer does not have a pad token by default, which is needed for
batching.
if tokenizer.pad_token is None:
    tokenizer.add_special_tokens({'pad_token': tokenizer.eos_token}) # Use EOS
token as pad token

model = AutoModelForCausalLM.from_pretrained(model_name)
model.resize_token_embeddings(len(tokenizer)) # Resize embeddings if we added a
new token

# --- 3. Tokenization and Data Formatting ---
def tokenize_function(examples):
    # Tokenize the text and ensure truncation if sequences are too long
    # max_length should be chosen based on your data and model's context window
    return tokenizer(examples["text"], truncation=True, max_length=128)

tokenized_dataset = dataset.map(tokenize_function, batched=True,
remove_columns=["text"])

# For causal language modeling, the labels are the same as the input IDs.
# We also need to handle padding for batching.
def prepare_lm_inputs(examples):
    # Concatenate all texts in the batch
    concatenated_examples = {k: sum(examples[k], []) for k in examples.keys()}
    total_length = len(concatenated_examples[list(examples.keys())[0]])
    # We'll pad to a multiple of max_length (or just max_length)
    max_length = tokenizer.model_max_length if tokenizer.model_max_length > 0
else 128
    total_length = (total_length // max_length) * max_length
    result = {
        k: [t[i : i + max_length] for i in range(0, total_length, max_length)]
        for k, t in concatenated_examples.items()
    }
    result["labels"] = result["input_ids"].copy()
    return result

# Apply the preparation function
# This step is often handled by DataCollatorForLanguageModeling, but showing
manual for clarity
# For simplicity, we'll just use the tokenized_dataset directly for this small
example.
# For larger datasets, a DataCollator would be used with `Trainer`.
# Here, we'll just ensure labels are present.
tokenized_dataset = tokenized_dataset.map(lambda examples: {"labels":
examples["input_ids"]}, batched=True)


# --- 4. Fine-tuning Configuration ---
# Define training arguments
training_args = TrainingArguments(
```

```
    output_dir="./gpt_fine_tuned_model", # Directory to save checkpoints and
model
    overwrite_output_dir=True,
    num_train_epochs=3, # Number of training epochs
    per_device_train_batch_size=2, # Batch size per GPU/CPU
    save_steps=10_000, # Save checkpoint every 10,000 steps
    save_total_limit=2, # Only keep the last 2 checkpoints
    logging_dir="./logs", # Directory for logs
    logging_steps=500,
    learning_rate=5e-5, # Learning rate
    weight_decay=0.01,
    fp16=torch.cuda.is_available(), # Use mixed precision if GPU is available
    report_to="none" # Disable reporting to external services like W&B
)

# --- 5. Training ---
trainer = Trainer(
    model=model,
    args=training_args,
    train_dataset=tokenized_dataset,
    tokenizer=tokenizer,
    # data_collator=DataCollatorForLanguageModeling(tokenizer=tokenizer,
mlm=False) # For proper batching/padding
)

print(f"\nStarting fine-tuning of {model_name}...")
# trainer.train() # Uncomment to run actual training

print("\nFine-tuning process conceptually complete. (Training skipped for
brevity)")
print(f"Model would be saved to: {training_args.output_dir}")

# --- 6. Evaluation and Generation (Illustrative) ---
# Load the fine-tuned model (or the original if training was skipped)
# For a real scenario, you'd load from `training_args.output_dir`
# fine_tuned_model =
AutoModelForCausalLM.from_pretrained(training_args.output_dir)
# fine_tuned_tokenizer = AutoTokenizer.from_pretrained(training_args.output_dir)
fine_tuned_model = model # Using the original model for demonstration
fine_tuned_tokenizer = tokenizer

print("\n--- Generating Text with (Simulated) Fine-tuned Model ---")
input_prompt = "You are amazing."
input_ids = fine_tuned_tokenizer.encode(input_prompt, return_tensors='pt')

# Move to GPU if available
if torch.cuda.is_available():
    input_ids = input_ids.to('cuda')
    fine_tuned_model.to('cuda')

# Generate text
# num_return_sequences: how many independent sequences to generate
# max_new_tokens: maximum number of tokens to generate
# temperature: controls randomness. Lower = more deterministic, Higher = more
creative
# top_k: sample from top_k most likely words
# top_p: sample from smallest set of words whose cumulative probability exceeds
top_p
output_sequences = fine_tuned_model.generate(
    input_ids=input_ids,
    max_new_tokens=50,
    num_return_sequences=1,
    temperature=0.7,
    top_k=50,
    top_p=0.95,
    do_sample=True, # Enable sampling for more diverse output
    pad_token_id=fine_tuned_tokenizer.eos_token_id # Important for generation
```

```
)

generated_text = fine_tuned_tokenizer.decode(output_sequences[0],
skip_special_tokens=True)
print(f"Prompt: '{input_prompt}'")
print(f"Generated Text:\n{generated_text}")

# Another example
input_prompt_2 = "Embrace your"
input_ids_2 = fine_tuned_tokenizer.encode(input_prompt_2, return_tensors='pt')
if torch.cuda.is_available():
    input_ids_2 = input_ids_2.to('cuda')

output_sequences_2 = fine_tuned_model.generate(
    input_ids=input_ids_2,
    max_new_tokens=30,
    num_return_sequences=1,
    temperature=0.7,
    top_k=50,
    top_p=0.95,
    do_sample=True,
    pad_token_id=fine_tuned_tokenizer.eos_token_id
)
generated_text_2 = fine_tuned_tokenizer.decode(output_sequences_2[0],
skip_special_tokens=True)
print(f"\nPrompt: '{input_prompt_2}'")
print(f"Generated Text:\n{generated_text_2}")
```

Input

A dataset of text examples for fine-tuning (e.g., a collection of positive affirmations in the example code). A text prompt for generation after fine-tuning (e.g., "You are amazing.", "Embrace your").

Expected Output

1. Confirmation of model and tokenizer loading.
2. (If `trainer.train()` is uncommented) Training logs showing loss decreasing over epochs.
3. A message indicating where the fine-tuned model would be saved.
4. Generated text samples based on the provided prompts. If fine-tuning was successful, these generated texts should reflect the style and content of the fine-tuning dataset (e.g., positive affirmations).

Lab 11: Conditioning GPT models for specific text generation tasks

Title

Conditional Text Generation with GPT Models

Aim

To explore methods of conditioning GPT models to generate text that adheres to specific requirements, topics, styles, or formats, rather than just free-form generation.

Procedure

1. **Understand Conditioning:** Learn about different techniques to guide a GPT model's output, including:
   - **Prompt Engineering:** Crafting detailed and specific input prompts.
   - **Few-Shot Learning:** Providing examples within the prompt to demonstrate the desired output format or style.
   - **Control Tokens/Prefixes:** Using special tokens or phrases to signal the desired generation mode (less common with standard GPT-2/3, more with instruction-tuned models).
2. **Model and Tokenizer Setup:** Load a pre-trained GPT model and its corresponding tokenizer.
3. **Implement Conditioning Techniques:**
   - **Direct Prompting:** Formulate prompts that clearly state the desired output.
   - **Example-based Prompting:** Include input-output pairs in the prompt to guide the model.
4. **Generate and Analyze:** Generate text using the conditioned prompts and analyze how well the model adheres to the specified conditions.

Source Code

```
import torch
from transformers import AutoTokenizer, AutoModelForCausalLM

def generate_conditional_text(model, tokenizer, prompt, max_new_tokens=100,
temperature=0.7, top_k=50, top_p=0.95, num_return_sequences=1):
    """
    Generates text from a GPT model with specified conditioning.

    Args:
        model: The pre-trained GPT model.
        tokenizer: The tokenizer for the model.
        prompt (str): The input prompt, including conditioning elements.
        max_new_tokens (int): Maximum number of tokens to generate.
        temperature (float): Controls randomness.
        top_k (int): Sample from top_k most likely words.
        top_p (float): Sample from smallest set of words whose cumulative
probability exceeds top_p.
        num_return_sequences (int): Number of independent sequences to generate.

    Returns:
        list: A list of generated text strings.
    """
```

```
    input_ids = tokenizer.encode(prompt, return_tensors='pt')

    # Move to GPU if available
    if torch.cuda.is_available():
        input_ids = input_ids.to('cuda')
        model.to('cuda')

    # Generate text
    output_sequences = model.generate(
        input_ids=input_ids,
        max_new_tokens=max_new_tokens,
        num_return_sequences=num_return_sequences,
        temperature=temperature,
        top_k=top_k,
        top_p=top_p,
        do_sample=True,
        pad_token_id=tokenizer.eos_token_id # Important for generation
    )

    generated_texts = [tokenizer.decode(seq, skip_special_tokens=True) for seq
in output_sequences]
    return generated_texts

if __name__ == "__main__":
    # Load a pre-trained GPT-2 model and tokenizer
    model_name = "gpt2"
    tokenizer = AutoTokenizer.from_pretrained(model_name)
    model = AutoModelForCausalLM.from_pretrained(model_name)

    if tokenizer.pad_token is None:
        tokenizer.add_special_tokens({'pad_token': tokenizer.eos_token})

    print(f"Loaded GPT model: {model_name}")

    # --- Conditioning Technique 1: Direct Prompt Engineering ---
    print("\n--- Direct Prompt Engineering ---")
    prompt_1 = "Write a short, optimistic poem about the sunrise, using vivid
imagery and no more than 4 lines."
    generated_poems = generate_conditional_text(model, tokenizer, prompt_1,
max_new_tokens=40, num_return_sequences=1)
    for i, text in enumerate(generated_poems):
        print(f"Prompt: '{prompt_1}'")
        print(f"Generated Poem {i+1}:\n{text}\n")

    # --- Conditioning Technique 2: Few-Shot Learning (Example-based) ---
    # Provide examples to guide the model's output format/style
    print("\n--- Few-Shot Learning (Example-based) ---")
    prompt_2 = """Translate the following English phrases to French:
English: Hello
French: Bonjour
English: Thank you
French: Merci
English: Good morning
French: """
    generated_translations = generate_conditional_text(model, tokenizer,
prompt_2, max_new_tokens=10, num_return_sequences=1)
    for i, text in enumerate(generated_translations):
        print(f"Prompt: '{prompt_2}'")
        print(f"Generated Translation {i+1}:\n{text}\n")

    # --- Conditioning Technique 3: Tone/Style Conditioning ---
    print("\n--- Tone/Style Conditioning ---")
    prompt_3 = "Write a formal email requesting information about a job opening
for a data scientist. Subject: Inquiry about Data Scientist Position"
    generated_emails = generate_conditional_text(model, tokenizer, prompt_3,
max_new_tokens=150, num_return_sequences=1)
    for i, text in enumerate(generated_emails):
```

```
        print(f"Prompt: '{prompt_3}'")
        print(f"Generated Email {i+1}:\n{text}\n")

    # --- Conditioning Technique 4: Specific Topic/Content ---
    print("\n--- Specific Topic/Content Conditioning ---")
    prompt_4 = "Explain the concept of quantum entanglement in simple terms,
suitable for a high school student."
    generated_explanations = generate_conditional_text(model, tokenizer,
prompt_4, max_new_tokens=120, num_return_sequences=1)
    for i, text in enumerate(generated_explanations):
        print(f"Prompt: '{prompt_4}'")
        print(f"Generated Explanation {i+1}:\n{text}\n")
```

Input

Various text prompts designed to condition the GPT model, including:

- Direct instructions (e.g., "Write a short, optimistic poem...")
- Few-shot examples (e.g., "English: Hello\nFrench: Bonjour\n...")
- Contextual cues for tone/style (e.g., "Write a formal email...")
- Specific topic requests (e.g., "Explain the concept of quantum entanglement...")

Expected Output

Generated text outputs for each prompt. The output should demonstrate the GPT model's ability to follow the conditioning instructions, producing text that is:

- A short, optimistic poem about sunrise.
- A French translation (likely "Bonjour" or similar, following the pattern).
- A formally structured email.
- A simplified explanation of quantum entanglement. The quality of adherence will depend on the model's capabilities and the clarity of the prompt.

Lab 12: Interpreting and analyzing the output of GPT models for text generation tasks.

Title

Analyzing and Interpreting GPT Model Outputs

Aim

To develop critical analysis skills for evaluating the quality, coherence, factual accuracy, potential biases, and overall effectiveness of text generated by GPT models.

Procedure

1. **Generate Diverse Outputs:** Use a GPT model to generate text for various prompts and tasks (e.g., creative writing, summarization, Q&A, code generation).
2. **Define Evaluation Criteria:** Establish a set of criteria for analyzing the generated text. These might include:
   - **Fluency and Coherence:** Does the text flow naturally? Is it grammatically correct? Are ideas logically connected?
   - **Relevance:** Does the text directly address the prompt? Is it on-topic?
   - **Factual Accuracy:** Is any factual information presented correct? (Crucial for non-creative tasks).
   - **Creativity/Originality:** For creative tasks, is the output original and engaging?
   - **Completeness:** Does the text fulfill all aspects of the prompt?
   - **Conciseness:** Is the text free of unnecessary verbosity?
   - **Bias and Safety:** Does the text contain any harmful, biased, or inappropriate content?
   - **Style and Tone:** Does the text match the requested style/tone?
3. **Perform Manual Analysis:** Read through the generated texts and apply the defined criteria to assess their quality. Annotate specific examples of strengths and weaknesses.
4. **Identify Limitations and Strengths:** Based on the analysis, identify common patterns of failure (e.g., hallucination, repetition, lack of common sense) and strengths (e.g., fluency, creativity, speed).
5. **Formulate Improvements:** Suggest ways to improve the prompts or fine-tuning data to mitigate identified weaknesses.

Source Code

(Note: This lab primarily involves qualitative analysis rather than extensive coding. The source code below focuses on generating diverse outputs to facilitate the analysis, but the interpretation itself is a manual, human-driven process.)

```
import torch
from transformers import AutoTokenizer, AutoModelForCausalLM

def generate_text_for_analysis(model, tokenizer, prompt, max_new_tokens=150,
temperature=0.7, num_return_sequences=1):
    """
    Generates text from a GPT model for analysis.
    """
    input_ids = tokenizer.encode(prompt, return_tensors='pt')
    if torch.cuda.is_available():
```

```
        input_ids = input_ids.to('cuda')
        model.to('cuda')

    output_sequences = model.generate(
        input_ids=input_ids,
        max_new_tokens=max_new_tokens,
        num_return_sequences=num_return_sequences,
        temperature=temperature,
        do_sample=True,
        pad_token_id=tokenizer.eos_token_id
    )
    generated_texts = [tokenizer.decode(seq, skip_special_tokens=True) for seq
in output_sequences]
    return generated_texts

if __name__ == "__main__":
    model_name = "gpt2" # Or a larger model if available and resources permit
    tokenizer = AutoTokenizer.from_pretrained(model_name)
    model = AutoModelForCausalLM.from_pretrained(model_name)

    if tokenizer.pad_token is None:
        tokenizer.add_special_tokens({'pad_token': tokenizer.eos_token})

    print(f"Loaded GPT model: {model_name}")
    print("\n--- Generating Diverse Texts for Analysis ---")

    # Prompt 1: Creative Writing - Fantasy Story
    prompt_1 = "Continue the story: The ancient dragon awoke, its eyes glowing
like embers in the deep cavern. A lone adventurer stood before it, sword drawn."
    generated_story = generate_text_for_analysis(model, tokenizer, prompt_1,
max_new_tokens=200, temperature=0.8)
    print("\n--- Generated Story ---")
    print(f"Prompt: '{prompt_1}'")
    print(generated_story[0])
    print("-" * 50)

    # Prompt 2: Factual/Explanatory - Science Concept
    prompt_2 = "Explain the process of photosynthesis in simple terms."
    generated_explanation = generate_text_for_analysis(model, tokenizer,
prompt_2, max_new_tokens=150, temperature=0.5)
    print("\n--- Generated Explanation ---")
    print(f"Prompt: '{prompt_2}'")
    print(generated_explanation[0])
    print("-" * 50)

    # Prompt 3: Summarization (simulated)
    # In a real scenario, you'd feed a longer text to summarize.
    # Here, we'll ask it to summarize a concept it might know.
    prompt_3 = "Summarize the main idea of machine learning."
    generated_summary = generate_text_for_analysis(model, tokenizer, prompt_3,
max_new_tokens=80, temperature=0.6)
    print("\n--- Generated Summary ---")
    print(f"Prompt: '{prompt_3}'")
    print(generated_summary[0])
    print("-" * 50)

    # Prompt 4: Potential for Bias/Stereotypes (Illustrative)
    prompt_4 = "Describe a typical scientist."
    generated_description = generate_text_for_analysis(model, tokenizer,
prompt_4, max_new_tokens=100, temperature=0.7)
    print("\n--- Generated Description (for Bias Analysis) ---")
    print(f"Prompt: '{prompt_4}'")
    print(generated_description[0])
    print("-" * 50)

    print("\n--- Analysis Instructions ---")
```

```
    print("Now, manually analyze each generated text based on the following
criteria:")
    print("1.  **Fluency and Coherence:** Does it read naturally? Are sentences
well-formed?")
    print("2.  **Relevance:** Does it answer the prompt directly?")
    print("3.  **Factual Accuracy (for non-creative tasks):** Is the information
correct?")
    print("4.  **Creativity/Originality (for creative tasks):** Is it
interesting and unique?")
    print("5.  **Completeness:** Does it cover all aspects requested in the
prompt?")
    print("6.  **Conciseness:** Is there any unnecessary repetition or
verbosity?")
    print("7.  **Bias and Safety:** Are there any stereotypes, harmful content,
or inappropriate language?")
    print("\nDocument your findings for each generated text, noting strengths,
weaknesses, and potential areas for improvement.")
```

Input

Various text prompts covering different generation tasks (e.g., story continuation, factual explanation, summarization, descriptive prompts).

Expected Output

1. Several generated text outputs, each corresponding to a different prompt.
2. Instructions for the user to manually analyze these outputs based on the provided evaluation criteria.
3. The core output of this lab is the *human analysis report* of the generated texts, which identifies their strengths, weaknesses, and potential biases.

Lab 13: Generating images using DALL E

Title

Image Generation with DALL-E (Conceptual)

Aim

To understand the process of generating images from text prompts using a text-to-image model like DALL-E, focusing on the user interaction and the power of prompt engineering.

Procedure

1. **Understand Text-to-Image Models:** Learn about the capabilities of models like DALL-E, which translate natural language descriptions into visual images.
2. **Formulate Text Prompts:** Craft descriptive text prompts that specify the desired image content, style, and composition. The more detailed and clear the prompt, the better the generated image.
3. **Interact with the API/Interface:** (Conceptually) Send the text prompt to a DALL-E API or interact with a DALL-E-powered interface.
4. **Review Generated Images:** Examine the images returned by DALL-E, evaluating how well they match the prompt and their overall quality.
5. **Iterate and Refine:** Experiment with different prompts, adding more details, changing styles, or specifying negative constraints to achieve desired visual outcomes.

Source Code

(Note: Direct access to DALL-E's API for general use is typically restricted or requires specific credentials. The following code provides a conceptual Python structure to illustrate how such an interaction would occur, using a placeholder for the actual API call. It will not generate real images.)

```
import json
import base64

# This is a placeholder for the actual DALL-E API interaction.
# In a real application, you would use the official DALL-E API client or make
HTTP requests.

async def generate_image_with_dalle(prompt):
    """
    Simulates generating an image using a DALL-E-like API.
    This function will NOT make a real API call or generate real images.
    It demonstrates the expected structure of such an interaction.

    Args:
        prompt (str): The text description for the image to generate.

    Returns:
        dict: A dictionary containing a placeholder image URL and a message.
              In a real scenario, this would contain actual image data or URLs.
    """
    print(f"Simulating DALL-E image generation for prompt: '{prompt}'")

    # Placeholder for API call
    # In a real scenario, this would be:
```

```
    # response = await fetch(DALL_E_API_URL, method='POST', headers=...,
body=json.dumps({'prompt': prompt, ...}))
    # result = await response.json()
    # image_data = result['data'][0]['b64_json'] # Example for base64 image

    # For demonstration, we'll return a placeholder image URL.
    # In a real DALL-E response, you'd get base64 encoded images or URLs.
    # Example placeholder image from placehold.co
    placeholder_image_url =
"https://placehold.co/512x512/ADD8E6/000000?text=DALL-E+Image"

    # Simulate a delay for API call
    await asyncio.sleep(1) # Requires asyncio, for browser environment, use
setTimeout

    return {
        "image_url": placeholder_image_url,
        "message": f"Image generation simulated for: '{prompt}'. A placeholder
image is shown. In a real scenario, DALL-E would generate a unique image based
on your prompt."
    }

# This part would typically be in a web application's JavaScript or a Python
script
# that calls the async function.
# For a direct Python script, you'd need an event loop.
import asyncio

async def main():
    print("--- DALL-E Image Generation Simulation ---")

    # Example 1: Simple prompt
    prompt_1 = "A majestic cat wearing a tiny crown, sitting on a cloud."
    result_1 = await generate_image_with_dalle(prompt_1)
    print(f"\nResult for Prompt 1:\nImage URL: {result_1['image_url']}\nMessage:
{result_1['message']}")

    # Example 2: More detailed prompt with style
    prompt_2 = "An astronaut riding a horse on the moon, in a photorealistic
style, with Earth in the background."
    result_2 = await generate_image_with_dalle(prompt_2)
    print(f"\nResult for Prompt 2:\nImage URL: {result_2['image_url']}\nMessage:
{result_2['message']}")

    # Example 3: Abstract concept
    prompt_3 = "The concept of 'creativity' visualized as an abstract painting."
    result_3 = await generate_image_with_dalle(prompt_3)
    print(f"\nResult for Prompt 3:\nImage URL: {result_3['image_url']}\nMessage:
{result_3['message']}")

if __name__ == "__main__":
    # Run the async main function
    try:
        asyncio.run(main())
    except RuntimeError as e:
        if "cannot run loop while another loop is running" in str(e):
            print("Detected existing asyncio loop. Running main in current
loop.")
            # This handles cases where run() is called in environments already
running a loop (e.g., Jupyter)
            loop = asyncio.get_event_loop()
            loop.run_until_complete(main())
        else:
            raise
```

Input

Text prompts describing the desired image content. Examples:

- "A majestic cat wearing a tiny crown, sitting on a cloud."
- "An astronaut riding a horse on the moon, in a photorealistic style, with Earth in the background."
- "The concept of 'creativity' visualized as an abstract painting."

Expected Output

1. Messages indicating the simulation of DALL-E image generation for each prompt.
2. For each prompt, a placeholder image URL will be displayed, along with a message explaining that this is a simulation and a real DALL-E output would be a unique image based on the prompt.
3. (In a real DALL-E environment) Actual generated images matching the descriptions, showcasing the model's ability to create novel visuals from text.

Lab 14: Conditioning DALL-E to generate images

Title

Advanced Image Generation with DALL-E Conditioning

Aim

To master the art of conditioning DALL-E through sophisticated prompt engineering techniques, enabling the generation of images with precise control over style, composition, and specific elements.

Procedure

1. **Review Basic Prompting:** Recall the basics of generating images from text prompts.
2. **Explore Advanced Prompting:** Learn about techniques to refine DALL-E's output:
   - **Detailed Descriptions:** Using specific nouns, adjectives, and verbs to describe objects, actions, and environments.
   - **Art Styles:** Specifying artistic styles (e.g., "oil painting," "pixel art," "cyberpunk," "impressionist").
   - **Lighting and Atmosphere:** Describing lighting conditions (e.g., "golden hour," "neon glow," "dark and moody") and atmosphere (e.g., "foggy," "serene," "chaotic").
   - **Camera Angles/Shots:** (If supported) Indicating perspective (e.g., "wide shot," "close-up," "from above").
   - **Negative Prompts (Conceptual):** (In some advanced models) Specifying what *not* to include in the image.
   - **Combinations:** Combining multiple elements and styles into a single prompt.
3. **Experiment with Prompts:** Systematically vary elements within prompts to observe their impact on the generated images.
4. **Analyze and Refine:** Evaluate the generated images against the prompt's intent. Identify which prompt elements are most effective and iterate on prompts for better results.

Source Code

(Note: Similar to Lab 13, this is a conceptual demonstration. It will not make real API calls or generate real images, but it shows how different conditioning elements would be incorporated into prompts.)

```
import json
import base64
import asyncio

async def generate_conditioned_image_with_dalle(prompt):
    """
    Simulates generating an image using a DALL-E-like API with advanced
conditioning.
    This function will NOT make a real API call or generate real images.
    It demonstrates the expected structure of such an interaction.

    Args:
        prompt (str): The text description with detailed conditioning for the
image.

    Returns:
```

```
        dict: A dictionary containing a placeholder image URL and a message.
    """
    print(f"Simulating DALL-E conditioned image generation for prompt:
'{prompt}'")

    # Placeholder for API call
    # In a real scenario, this would be:
    # response = await fetch(DALL_E_API_URL, method='POST', headers=...,
body=json.dumps({'prompt': prompt, ...}))
    # result = await response.json()
    # image_data = result['data'][0]['b64_json']

    placeholder_image_url =
"https://placehold.co/512x512/DDA0DD/000000?text=DALL-E+Conditioned+Image"
    await asyncio.sleep(1) # Simulate delay

    return {
        "image_url": placeholder_image_url,
        "message": f"Conditioned image generation simulated for: '{prompt}'. A
placeholder image is shown. In a real scenario, DALL-E would generate a unique
image based on your detailed prompt."
    }

async def main():
    print("--- DALL-E Conditioned Image Generation Simulation ---")

    # Example 1: Specific Art Style
    prompt_1 = "A cyberpunk city at night, with neon lights reflecting on wet
streets, in the style of a retro-futuristic anime."
    result_1 = await generate_conditioned_image_with_dalle(prompt_1)
    print(f"\nResult for Prompt 1 (Art Style):\nImage URL:
{result_1['image_url']}\nMessage: {result_1['message']}")

    # Example 2: Lighting and Atmosphere
    prompt_2 = "A serene forest glade bathed in soft, ethereal morning light,
with a gentle mist rising from the ground."
    result_2 = await generate_conditioned_image_with_dalle(prompt_2)
    print(f"\nResult for Prompt 2 (Lighting/Atmosphere):\nImage URL:
{result_2['image_url']}\nMessage: {result_2['message']}")

    # Example 3: Complex Scene with Multiple Elements
    prompt_3 = "An ancient wizard, with a long white beard and a pointed hat,
casting a spell in a crumbling library filled with floating magical books,
highly detailed, fantasy art."
    result_3 = await generate_conditioned_image_with_dalle(prompt_3)
    print(f"\nResult for Prompt 3 (Complex Scene):\nImage URL:
{result_3['image_url']}\nMessage: {result_3['message']}")

    # Example 4: Abstract Concept with Specific Colors
    prompt_4 = "The feeling of 'nostalgia' visualized as a warm, sepia-toned
memory, with blurry edges and a single, sharp detail in the center."
    result_4 = await generate_conditioned_image_with_dalle(prompt_4)
    print(f"\nResult for Prompt 4 (Abstract/Colors):\nImage URL:
{result_4['image_url']}\nMessage: {result_4['message']}")

if __name__ == "__main__":
    try:
        asyncio.run(main())
    except RuntimeError as e:
        if "cannot run loop while another loop is running" in str(e):
            print("Detected existing asyncio loop. Running main in current
loop.")
            loop = asyncio.get_event_loop()
            loop.run_until_complete(main())
        else:
            raise
```

Input

Detailed text prompts incorporating various conditioning elements:

- Art style (e.g., "retro-futuristic anime")
- Lighting and atmosphere (e.g., "soft, ethereal morning light, with a gentle mist")
- Complex scenes with multiple objects and actions
- Abstract concepts with specific visual attributes (e.g., "warm, sepia-toned memory, with blurry edges")

Expected Output

1. Messages indicating the simulation of DALL-E conditioned image generation for each detailed prompt.
2. For each prompt, a placeholder image URL will be displayed, along with a message explaining that this is a simulation and a real DALL-E output would be a unique, highly-conditioned image.
3. (In a real DALL-E environment) Generated images that closely match the intricate details and stylistic cues provided in the prompts, demonstrating the power of advanced prompt engineering.

Lab 15: Preprocessing and formatting datasets for training and fine-tuning DALL-E models.

Title

Dataset Preparation for DALL-E Model Training and Fine-tuning

Aim

To understand the critical steps involved in preparing and formatting image-text pair datasets for training or fine-tuning large-scale text-to-image generative models like DALL-E.

Procedure

1. **Dataset Collection:** Identify and collect a dataset consisting of image-text pairs. Each pair should ideally have a descriptive caption corresponding to the image.
2. **Data Cleaning:**
   - **Image Cleaning:** Remove corrupted, low-quality, or irrelevant images. Resize images to a consistent dimension (e.g., 256x256, 512x512) and normalize pixel values.
   - **Text Cleaning:** Clean captions by removing special characters, emojis, irrelevant metadata, and performing basic text normalization (e.g., lowercasing, tokenization if required for the specific model's tokenizer).
   - **Pairing Validation:** Ensure that each image has a corresponding, meaningful text caption.
3. **Data Augmentation (Optional but Recommended):** Apply image augmentations (e.g., random flips, rotations, color jitter) to increase dataset diversity and improve model robustness.
4. **Tokenization:** Tokenize the text captions using the specific tokenizer compatible with the DALL-E model you intend to train/fine-tune. This converts text into numerical sequences.
5. **Data Loading and Batching:** Create data loaders that efficiently load image-text pairs, apply necessary transformations, and batch them for training. This often involves libraries like TensorFlow's `tf.data` or PyTorch's `DataLoader`.
6. **Storage Format:** Store the preprocessed dataset in an efficient format (e.g., TFRecords, PyTorch `Dataset` objects, or simple directories with image files and a metadata CSV/JSON).

Source Code

(Note: This lab involves significant data handling and potentially large files. The following code provides a conceptual framework for the preprocessing steps using Python libraries like PIL (Pillow), `transformers` for tokenization, and NumPy. It does not handle large-scale dataset downloading or complex distributed processing, but illustrates the core logic.)

```
import os
import json
from PIL import Image
import numpy as np
from transformers import AutoTokenizer # Assuming a tokenizer for text captions
import tensorflow as tf # Or torch for PyTorch data loading

# --- Configuration ---
```

```
IMAGE_SIZE = (256, 256) # Target size for images
MAX_CAPTION_LENGTH = 77 # Common max length for CLIP-based models (like DALL-E's
text encoder)
DATASET_DIR = "raw_image_text_dataset" # Directory containing raw images and
captions
PROCESSED_DIR = "processed_dalle_dataset" # Directory to save processed data

# Create dummy raw data for demonstration
def create_dummy_raw_data(num_samples=10):
    if not os.path.exists(DATASET_DIR):
        os.makedirs(DATASET_DIR)
    dummy_captions = []
    for i in range(num_samples):
        # Create a dummy image
        dummy_image = Image.new('RGB', (512, 512), color = (i*20 % 255, i*30 %
255, i*40 % 255))
        image_filename = f"image_{i:03d}.png"
        dummy_image.save(os.path.join(DATASET_DIR, image_filename))

        # Create a dummy caption
        caption = f"A colorful abstract image with shades of {['red', 'blue',
'green', 'yellow'][i % 4]} and a number {i}. This is a test caption for DALL-E
dataset preprocessing."
        dummy_captions.append({"image_filename": image_filename, "caption":
caption})

    with open(os.path.join(DATASET_DIR, "captions.json"), "w") as f:
        json.dump(dummy_captions, f, indent=4)
    print(f"Created {num_samples} dummy raw image-text pairs in
'{DATASET_DIR}'")

# --- 1. Data Cleaning and Preprocessing ---
def preprocess_image(image_path, target_size):
    """Loads, resizes, and normalizes an image."""
    try:
        img = Image.open(image_path).convert("RGB")
        img = img.resize(target_size, Image.LANCZOS) # Use LANCZOS for high
quality downsampling
        img_array = np.array(img).astype(np.float32) / 255.0 # Normalize to [0,
1]
        return img_array
    except Exception as e:
        print(f"Error processing image {image_path}: {e}")
        return None

def clean_and_tokenize_caption(caption, tokenizer, max_length):
    """Cleans and tokenizes a text caption."""
    # Basic cleaning: remove extra whitespace, convert to lowercase
    cleaned_caption = " ".join(caption.split()).lower()
    # Tokenize and truncate
    tokenized_output = tokenizer(
        cleaned_caption,
        padding="max_length",
        truncation=True,
        max_length=max_length,
        return_tensors="tf" # Or "pt" for PyTorch
    )
    return tokenized_output["input_ids"][0],
tokenized_output["attention_mask"][0]

def process_dataset(raw_dir, processed_dir, image_size, max_caption_length,
tokenizer):
    """
    Processes raw image-text dataset and saves it in a ready-to-use format.
    """
    if not os.path.exists(processed_dir):
        os.makedirs(processed_dir)
```

```
    captions_file = os.path.join(raw_dir, "captions.json")
    if not os.path.exists(captions_file):
        print(f"Error: captions.json not found in {raw_dir}")
        return

    with open(captions_file, "r") as f:
        raw_captions_data = json.load(f)

    processed_data_list = []
    for i, item in enumerate(raw_captions_data):
        image_filename = item["image_filename"]
        caption = item["caption"]
        image_path = os.path.join(raw_dir, image_filename)

        print(f"Processing sample {i+1}/{len(raw_captions_data)}:
{image_filename}")

        # Preprocess image
        processed_image = preprocess_image(image_path, image_size)
        if processed_image is None:
            continue

        # Clean and tokenize caption
        input_ids, attention_mask = clean_and_tokenize_caption(caption,
tokenizer, max_caption_length)

        # Store processed data (e.g., as numpy arrays or TFRecords)
        # For simplicity, we'll store paths and processed data in a list for
this demo.
        # In a real scenario, you'd save these to TFRecords, HDF5, or similar.
        processed_data_list.append({
            "image": processed_image,
            "input_ids": input_ids.numpy(), # Convert TF tensor to numpy
            "attention_mask": attention_mask.numpy(), # Convert TF tensor to
numpy
            "original_caption": caption,
            "image_filename": image_filename
        })

    print(f"Successfully processed {len(processed_data_list)} samples.")
    return processed_data_list

# --- 2. Data Loading and Batching (Conceptual with TensorFlow tf.data) ---
def create_tf_dataset(processed_data, batch_size):
    """
    Creates a TensorFlow tf.data.Dataset from processed data.
    """
    # Separate images and text data
    images = np.array([d["image"] for d in processed_data])
    input_ids = np.array([d["input_ids"] for d in processed_data])
    attention_masks = np.array([d["attention_mask"] for d in processed_data])

    dataset = tf.data.Dataset.from_tensor_slices(
        ({"image": images, "input_ids": input_ids, "attention_mask":
attention_masks})
    )
    dataset =
dataset.shuffle(buffer_size=len(processed_data)).batch(batch_size).prefetch(tf.d
ata.AUTOTUNE)
    print(f"Created TensorFlow dataset with batch size {batch_size}.")
    return dataset

if __name__ == "__main__":
    # Create dummy raw data for demonstration
    create_dummy_raw_data(num_samples=20)
```

```
    # Initialize a tokenizer (e.g., CLIP's tokenizer, which DALL-E uses)
    # Using 'openai/clip-vit-base-patch32' as an example for text encoder
    tokenizer = AutoTokenizer.from_pretrained("openai/clip-vit-base-patch32")
    print(f"Loaded tokenizer: {tokenizer.name_or_path}")

    # Process the dataset
    processed_samples = process_dataset(DATASET_DIR, PROCESSED_DIR, IMAGE_SIZE,
MAX_CAPTION_LENGTH, tokenizer)

    if processed_samples:
        # Create a TensorFlow dataset for training
        BATCH_SIZE = 4
        dalle_tf_dataset = create_tf_dataset(processed_samples, BATCH_SIZE)

        # --- Verify a batch ---
        print("\n--- Verifying a batch from the processed dataset ---")
        for batch in dalle_tf_dataset.take(1):
            batch_images = batch["image"].numpy()
            batch_input_ids = batch["input_ids"].numpy()
            batch_attention_masks = batch["attention_mask"].numpy()

            print(f"Batch Image Shape: {batch_images.shape}") # (BATCH_SIZE,
IMAGE_SIZE[0], IMAGE_SIZE[1], 3)
            print(f"Batch Input IDs Shape: {batch_input_ids.shape}") #
(BATCH_SIZE, MAX_CAPTION_LENGTH)
            print(f"Batch Attention Mask Shape: {batch_attention_masks.shape}")
# (BATCH_SIZE, MAX_CAPTION_LENGTH)

            # Decode a sample caption to verify
            sample_caption_decoded = tokenizer.decode(batch_input_ids[0],
skip_special_tokens=True)
            print(f"Sample Decoded Caption: '{sample_caption_decoded}'")

            # Display a sample image from the batch
            plt.figure(figsize=(4, 4))
            plt.imshow(batch_images[0])
            plt.title(f"Sample Processed Image\nCaption:
{sample_caption_decoded[:50]}...")
            plt.axis('off')
            plt.show()
    else:
        print("No samples were processed. Check raw data directory and errors.")

    print("\nDataset preprocessing and formatting conceptually complete.")
    print(f"Processed data is ready for training/fine-tuning a DALL-E model.")
```

Input

A directory containing raw image files and a `captions.json` file (or similar metadata file) mapping image filenames to their corresponding text captions. The code includes a function to create dummy data for demonstration.

Expected Output

1. Messages indicating the creation of dummy raw data (if `create_dummy_raw_data` is run).
2. Confirmation of tokenizer loading.
3. Logs showing the progress of processing each image-text pair.
4. Confirmation of the number of successfully processed samples.
5. Details about the created TensorFlow `tf.data.Dataset`, including its batch size.
6. Verification of a sample batch, showing the shapes of the image and tokenized text tensors.
7. A decoded sample caption from the batch to ensure correct tokenization.

8. A plot displaying a sample preprocessed image from the batch, confirming correct resizing and normalization.
9. A final message indicating that the dataset is ready for training/fine-tuning.