Godavari Institute of Management & Research, Jalgaon
Masters Computer Application (MCA)

Name: __ _ _ _ _ _ _ Roll No: _ _ _ _ _

Date of Performance: _ _ /_ _/20_ _ Batch: _ _ _ _ _ _ Class: MCA. -II

Subject: 515(B):
Lab on Generative AI Sign. of Teacher

1. Write Python scripts to implement basic operations and TensorFlow 2 tensors.

Title: Write Python scripts to implement basic operations and TensorFlow 2 tensors.

AIM: To understand and implement basic operations using TensorFlow 2 tensors in Python.

OBJECTIVES:

1. To learn how to create and manipulate tensors in TensorFlow.
2. To perform mathematical operations and explore tensor properties.

REQUIRED TOOLS / SOFTWARE:

- **Python 3.10+**
- **TensorFlow 2.x**
- **Google Colab** or **Jupyter Notebook**

Libraries used:
`tensorflow`, `numpy`

THEORY:

TensorFlow is an open-source deep learning library developed by Google.
It uses **tensors**—multi-dimensional arrays—as the basic data structure.
A **tensor** can be a scalar (0-D), vector (1-D), matrix (2-D), or higher-dimensional data.

Common operations in TensorFlow include:

- Tensor creation (`tf.constant()`, `tf.Variable()`)
- Mathematical operations (`add`, `subtract`, `multiply`, `divide`)
- Tensor reshaping and slicing
- Checking tensor properties (shape, dtype, rank)

PROGRAM:

```
# Experiment No. 1
# Program: Basic Operations and TensorFlow 2 Tensors
# Step 1: Import the TensorFlow library
import tensorflow as tf

# Step 2: Create constant tensors
a = tf.constant([[1, 2], [3, 4]])
b = tf.constant([[5, 6], [7, 8]])

print("Tensor A:\n", a)
print("Tensor B:\n", b)

# Step 3: Perform basic arithmetic operations
add_result = tf.add(a, b)
sub_result = tf.subtract(a, b)
mul_result = tf.multiply(a, b)
div_result = tf.divide(a, b)

print("\nAddition:\n", add_result)
print("\nSubtraction:\n", sub_result)
print("\nMultiplication:\n", mul_result)
print("\nDivision:\n", div_result)

# Step 4: Check tensor properties
print("\nShape of Tensor A:", tf.shape(a))
print("Data type of Tensor A:", a.dtype)
print("Rank of Tensor A:", tf.rank(a))

# Step 5: Convert tensor to numpy array
numpy_a = a.numpy()
print("\nTensor A as NumPy array:\n", numpy_a)

# Step 6: Reshape a tensor
reshaped = tf.reshape(a, [4, 1])
print("\nReshaped Tensor A:\n", reshaped)
```

EXPECTED OUTPUT:

```
Tensor A:
 [[1 2]
  [3 4]]
Tensor B:
 [[5 6]
  [7 8]]

Addition:
 [[ 6  8]
  [10 12]]

Subtraction:
 [[-4 -4]
  [-4 -4]]

Multiplication:
 [[ 5 12]
  [21 32]]

Division:
 [[0.2        0.33333334]
  [0.42857143 0.5       ]]
```

```
Shape of Tensor A: [2 2]
Data type of Tensor A: <dtype: 'int32'>
Rank of Tensor A: 2

Tensor A as NumPy array:
 [[1 2]
  [3 4]]

Reshaped Tensor A:
 [[1]
  [2]
  [3]
  [4]]
```

RESULT / CONCLUSION:

The basic TensorFlow operations such as addition, subtraction, multiplication, and division were successfully executed on tensors. The experiment helped in understanding how tensors are created, manipulated, and converted between TensorFlow and NumPy.

Godavari Institute of Management & Research, Jalgaon
Masters Computer Application (MCA)

Name: __ _ _ _ _ _ _ Roll No: _ _ _ _ _

Date of Performance: _ _ /_ _/20_ _ Batch: _ _ _ _ _ _ Class: MCA. -II

Subject: 515(B):
Lab on Generative AI Sign. of Teacher

2. Preprocess and clean datasets for Generative AI applications using Python libraries such as Pandas and NumPy. Handle missing data, normalize features, and encode categorical variables.

Title: Preprocess and clean datasets for Generative AI applications using Python libraries such as Pandas and NumPy. Handle missing data, normalize features, and encode categorical variables.

AIM: To perform dataset preprocessing and cleaning using Python libraries — including handling missing data, normalizing numerical features, and encoding categorical variables — for use in Generative AI models.

OBJECTIVES:

1. To learn how to load and explore datasets using Pandas and NumPy.
2. To handle missing or invalid data using suitable imputation methods.
3. To normalize numerical data for model stability.
4. To encode categorical features for AI model input.

REQUIRED TOOLS / SOFTWARE:

- **Python 3.10+**
- **Google Colab / Jupyter Notebook**
- **Libraries:** `pandas`, `numpy`, `scikit-learn`

THEORY:

Before using a dataset for AI model training, it must be cleaned and transformed.
Data Preprocessing ensures consistency, completeness, and proper format for algorithms.

Key Steps:

1. **Handling Missing Data:**
   - Drop missing values (`dropna()`)
   - Fill missing values (`fillna()`) with mean/median/mode
2. **Normalization:**
   - Scale data to a uniform range (e.g., 0–1)
   - Use Min-Max or Z-score normalization

3. **Encoding Categorical Data:**
   - Convert text labels to numbers using Label Encoding or One-Hot Encoding

PROGRAM:

```
# Experiment No. 2
# Program: Data Preprocessing and Cleaning using Pandas and NumPy

# Step 1: Import necessary libraries
import pandas as pd
import numpy as np
from sklearn.preprocessing import MinMaxScaler, LabelEncoder

# Step 2: Create a sample dataset with missing and categorical data
data = {
    'Age': [25, 30, np.nan, 22, 28],
    'Salary': [50000, 60000, 55000, np.nan, 58000],
    'Department': ['IT', 'HR', 'IT', 'Finance', np.nan]
}

df = pd.DataFrame(data)
print("Original Dataset:\n", df)

# Step 3: Handling missing data
# Option 1: Fill missing numerical data with mean
df['Age'].fillna(df['Age'].mean(), inplace=True)
df['Salary'].fillna(df['Salary'].mean(), inplace=True)

# Option 2: Fill missing categorical data with mode
df['Department'].fillna(df['Department'].mode()[0], inplace=True)

print("\nAfter Handling Missing Data:\n", df)

# Step 4: Normalize numerical features (0-1 range)
scaler = MinMaxScaler()
df[['Age', 'Salary']] = scaler.fit_transform(df[['Age', 'Salary']])
print("\nAfter Normalization:\n", df)

# Step 5: Encode categorical variables
label_encoder = LabelEncoder()
df['Department'] = label_encoder.fit_transform(df['Department'])
print("\nAfter Encoding Categorical Variable:\n", df)
```

EXPECTED OUTPUT:

```
Original Dataset:
     Age   Salary Department
0  25.0  50000.0         IT
1  30.0  60000.0         HR
2   NaN  55000.0         IT
3  22.0      NaN    Finance
4  28.0  58000.0        NaN

After Handling Missing Data:
     Age   Salary Department
0  25.00  50000.0         IT
```

```
1  30.00  60000.0         HR
2  26.25  55000.0         IT
3  22.00  55750.0    Finance
4  28.00  58000.0         IT

After Normalization:
        Age    Salary Department
0  0.461538  0.000000         IT
1  1.000000  1.000000         HR
2  0.653846  0.500000         IT
3  0.000000  0.075000    Finance
4  0.769231  0.600000         IT

After Encoding Categorical Variable:
        Age    Salary  Department
0  0.461538  0.000000           2
1  1.000000  1.000000           1
2  0.653846  0.500000           2
3  0.000000  0.075000           0
4  0.769231  0.600000           2
```

RESULT / CONCLUSION:

The dataset was successfully preprocessed using Pandas and NumPy. Missing values were filled, numerical features normalized, and categorical variables encoded — preparing the data for Generative AI model training.

Godavari Institute of Management & Research, Jalgaon

Masters Computer Application (MCA)

Name: _ Roll No: _ _ _ _ _

Date of Performance: _ _ /_ _/20_ _ Batch: _ _ _ _ _ _ Class: MCA. -II

Subject: 515(B):

Lab on Generative AI Sign. of Teacher

3. Use Matplotlib or Seaborn to visualize data distributions and patterns in Generative AI datasets. Plot histograms, scatter plots, and heatmaps to analyze data characteristics.

Title: Use Matplotlib or Seaborn to visualize data distributions and patterns in Generative AI datasets. Plot histograms, scatter plots, and heatmaps to analyze data characteristics.

AIM: To visualize data distributions and relationships between features in a dataset using **Matplotlib** and **Seaborn** libraries for better understanding and preprocessing in Generative AI applications.

OBJECTIVES:

1. To study and use visualization libraries in Python.
2. To understand the role of visual analytics in AI model development.
3. To plot **histograms**, **scatter plots**, and **heatmaps** to analyze data distribution and correlation.

REQUIRED TOOLS / SOFTWARE:

- **Python 3.10+**
- **Google Colab / Jupyter Notebook**
- **Libraries:** `pandas`, `numpy`, `matplotlib`, `seaborn`

THEORY:

Visualization is a key step in **data exploration** for AI and machine learning.
Before applying models, it helps to:

- Understand **data distributions**,
- Detect **outliers**, and
- Identify **correlations** among features.

Common plots:

- **Histogram:** Shows frequency distribution of a variable.
- **Scatter Plot:** Shows relationship between two numerical variables.
- **Heatmap:** Displays correlation among multiple features using color gradients.

In Generative AI, visualizing input data ensures that training data patterns are correctly captured and balanced.

PROGRAM:

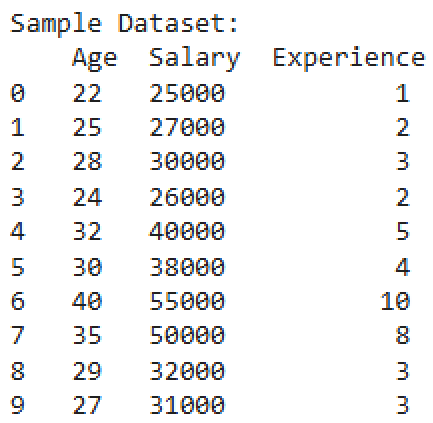
```
# Experiment No. 3
# Program: Data Visualization using Matplotlib and Seaborn

# Step 1: Import required libraries
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns

# Step 2: Create a sample dataset
data = {
    'Age': [22, 25, 28, 24, 32, 30, 40, 35, 29, 27],
    'Salary': [25000, 27000, 30000, 26000, 40000, 38000, 55000, 50000, 32000,
31000],
    'Experience': [1, 2, 3, 2, 5, 4, 10, 8, 3, 3]
}

df = pd.DataFrame(data)
print("Sample Dataset:\n", df)

# Step 3: Plot Histogram using Matplotlib
plt.figure(figsize=(6,4))
plt.hist(df['Salary'], bins=5, color='skyblue', edgecolor='black')
plt.title('Histogram of Salary Distribution')
plt.xlabel('Salary')
plt.ylabel('Frequency')
plt.show()

# Step 4: Scatter Plot using Seaborn
plt.figure(figsize=(6,4))
sns.scatterplot(x='Age', y='Salary', data=df, color='green', s=80)
plt.title('Scatter Plot: Age vs Salary')
plt.show()

# Step 5: Heatmap for Correlation Analysis
plt.figure(figsize=(5,4))
correlation = df.corr()
sns.heatmap(correlation, annot=True, cmap='coolwarm', linewidths=0.5)
plt.title('Heatmap of Feature Correlations')
plt.show()
```

EXPECTED OUTPUT:

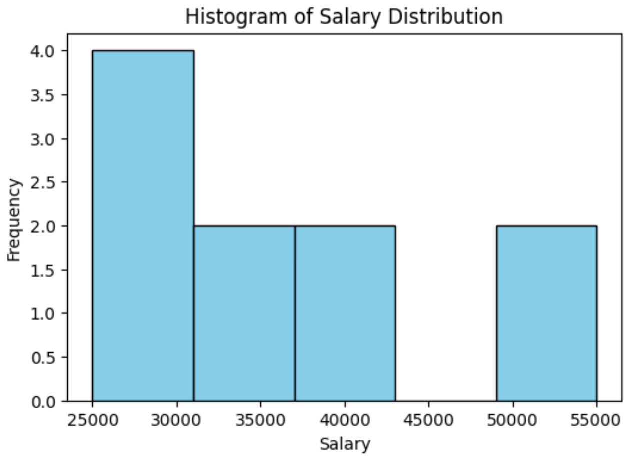
1. Histogram:
Shows distribution of salary values, indicating most employees earn between ₹25,000–₹40,000.

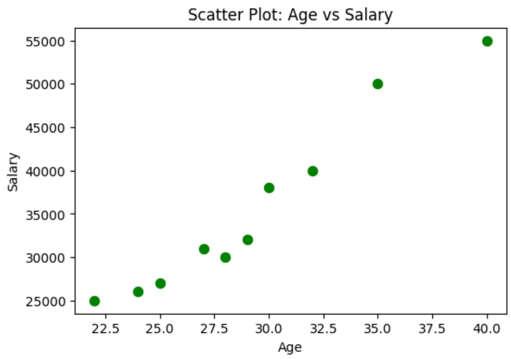
2. Scatter Plot:
Displays a positive correlation between Age and Salary — older employees tend to have higher salaries.

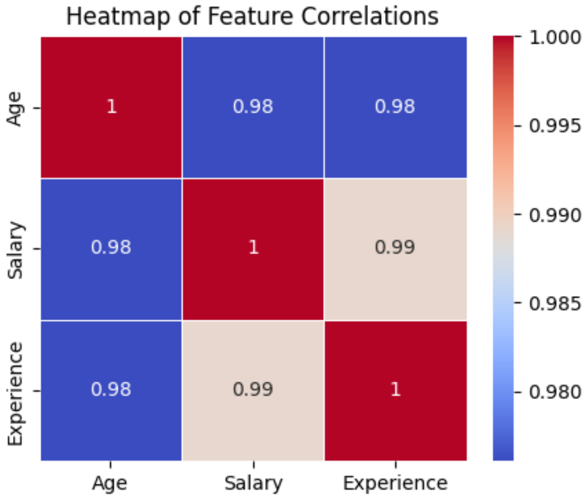
3. Heatmap:
Color-coded matrix showing correlation values:



```
Sample Dataset:
   Age  Salary  Experience
0   22   25000           1
1   25   27000           2
2   28   30000           3
3   24   26000           2
4   32   40000           5
5   30   38000           4
6   40   55000          10
7   35   50000           8
8   29   32000           3
9   27   31000           3
```







→

All features are positively correlated.

RESULT / CONCLUSION:

The dataset was successfully visualized using **Matplotlib** and **Seaborn**. The histogram, scatter plot, and heatmap provided insights into data distribution and feature relationships — essential for preprocessing and feature selection in Generative AI models.

Godavari Institute of Management & Research, Jalgaon
Masters Computer Application (MCA)

Name: _ Roll No: _ _ _ _ _

Date of Performance: _ _ /_ _/20_ _ Batch: _ _ _ _ _ _ Class: MCA. -II

Subject: 515(B):
Lab on Generative AI Sign. of Teacher

4. Implement a Generative Adversarial Network (GAN) architecture using TensorFlow 2. Train the GAN model on a dataset such as MNIST or CIFAR-10 for image generation tasks.

Title: Implement a Generative Adversarial Network (GAN) architecture using TensorFlow 2. Train the GAN model on MNIST for image generation tasks.

AIM:To build, train and evaluate a basic GAN (Generator + Discriminator) using TensorFlow 2 to generate handwritten-digit-like images (MNIST).

OBJECTIVES

1. Understand GAN components (generator & discriminator) and adversarial training.
2. Implement models using `tf.keras` and a custom training loop.
3. Train on MNIST and visualize generated samples during training.
4. Learn practical tips (losses, optimizers, monitoring).

REQUIRED TOOLS / SOFTWARE

- Python 3.8+
- TensorFlow 2.x (e.g., `tensorflow>=2.10`) — Colab recommended for GPU.
- Libraries: `numpy`, `matplotlib`.
- Optional: TensorBoard for monitoring.

THEORY (brief)

A GAN has two networks:

- **Generator (G):** maps random noise `z` → fake image `x_fake`.
- **Discriminator (D):** classifies images as real or fake.

Training alternates:

- Train D to distinguish real vs generated images.
- Train G to fool D (so D labels generated images as real).

Loss commonly used: binary cross-entropy for both D and G. Training is stable with tweaks like label smoothing, batchnorm, and using Adam optimizer.

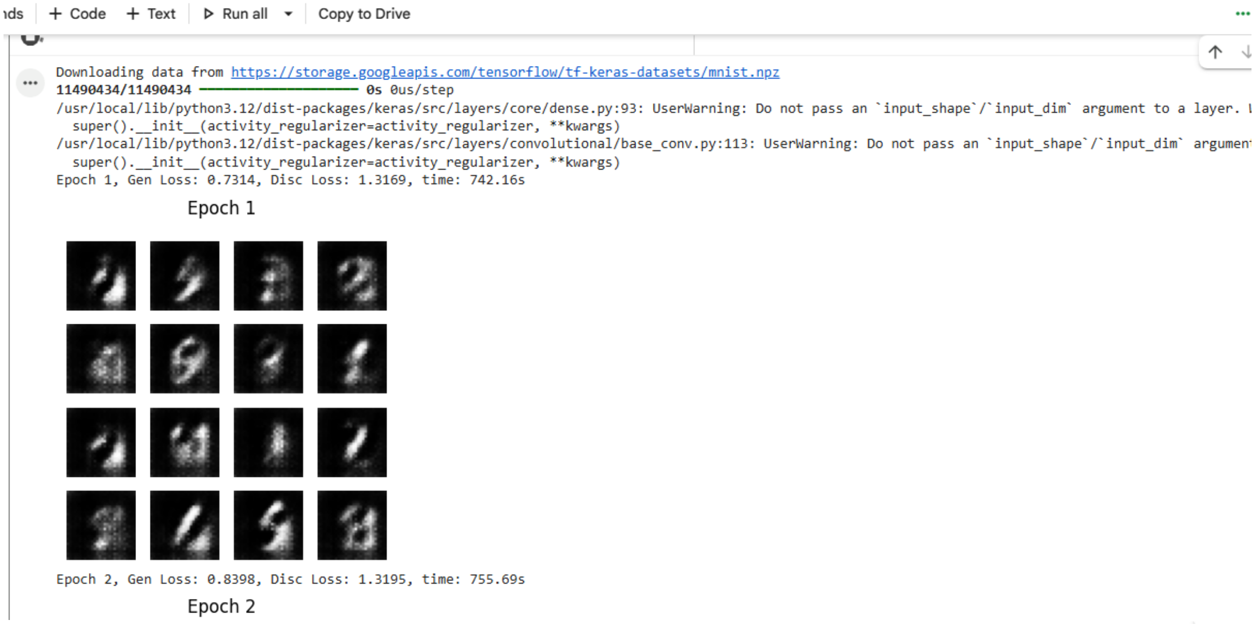
PROGRAM (Complete, ready-to-run in Google Colab / Jupyter)

```
# Experiment 4: GAN on MNIST (TensorFlow 2)
# Run this in Google Colab for GPU acceleration

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

# ---------------------------
# Hyperparameters
# ---------------------------
BUFFER_SIZE = 60000
BATCH_SIZE = 256
EPOCHS = 50            # increase for better results
NOISE_DIM = 100
NUM_EXAMPLES_TO_GENERATE = 16
LEARNING_RATE = 1e-4

# Fixed seed for consistent sample visualization
seed = tf.random.normal([NUM_EXAMPLES_TO_GENERATE, NOISE_DIM])

# ---------------------------
# Load and preprocess MNIST
# ---------------------------
(train_images, _), (_, _) = tf.keras.datasets.mnist.load_data()
# Normalize to [-1, 1]
train_images = (train_images.astype('float32') - 127.5) / 127.5
train_images = np.expand_dims(train_images, axis=-1)  # (N, 28, 28, 1)

train_dataset = (
    tf.data.Dataset.from_tensor_slices(train_images)
    .shuffle(BUFFER_SIZE)
    .batch(BATCH_SIZE)
)

# ---------------------------
# Generator model
# ---------------------------
def make_generator_model():
    model = tf.keras.Sequential(name="Generator")
    model.add(layers.Dense(7*7*256, use_bias=False,
input_shape=(NOISE_DIM,)))
    model.add(layers.BatchNormalization())
    model.add(layers.LeakyReLU())

    model.add(layers.Reshape((7, 7, 256)))  # 7x7x256

    model.add(layers.Conv2DTranspose(128, (5,5), strides=(1,1),
padding='same', use_bias=False))
    model.add(layers.BatchNormalization())
    model.add(layers.LeakyReLU())
    model.add(layers.Conv2DTranspose(64, (5,5), strides=(2,2),
padding='same', use_bias=False))
    model.add(layers.BatchNormalization())
    model.add(layers.LeakyReLU())

    model.add(layers.Conv2DTranspose(1, (5,5), strides=(2,2), padding='same',
use_bias=False, activation='tanh'))
    # Output: 28x28x1
    return model

# ---------------------------
# Discriminator model
# ---------------------------
def make_discriminator_model():
    model = tf.keras.Sequential(name="Discriminator")
    model.add(layers.Conv2D(64, (5,5), strides=(2,2), padding='same',
input_shape=[28,28,1]))
    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))
    return model

# Instantiate models
generator = make_generator_model()
discriminator = make_discriminator_model()

# ---------------------------
# Losses and optimizers
# ---------------------------
cross_entropy = tf.keras.losses.BinaryCrossentropy(from_logits=True)

def discriminator_loss(real_output, fake_output):
    # real labels -> ones, fake labels -> zeros
    real_loss = cross_entropy(tf.ones_like(real_output)*0.9, real_output)  #
label smoothing for real
    fake_loss = cross_entropy(tf.zeros_like(fake_output), fake_output)
    return real_loss + fake_loss

def generator_loss(fake_output):
    # want discriminator to predict ones for generated images
    return cross_entropy(tf.ones_like(fake_output), fake_output)

generator_optimizer = tf.keras.optimizers.Adam(learning_rate=LEARNING_RATE,
beta_1=0.5)
discriminator_optimizer =
tf.keras.optimizers.Adam(learning_rate=LEARNING_RATE, beta_1=0.5)

# ---------------------------
# Checkpoint (optional)
# ---------------------------
checkpoint_dir = './training_checkpoints_gan'
checkpoint_prefix = os.path.join(checkpoint_dir, "ckpt")
checkpoint = tf.train.Checkpoint(generator_optimizer=generator_optimizer,
discriminator_optimizer=discriminator_optimizer,
                                 generator=generator,
                                 discriminator=discriminator)

# ---------------------------
# Training step (tf.function for speed)
# ---------------------------
@tf.function
def train_step(images):
    noise = tf.random.normal([BATCH_SIZE, NOISE_DIM])

    # Record gradients for both networks
    with tf.GradientTape() as gen_tape, tf.GradientTape() as disc_tape:
        generated_images = generator(noise, training=True)

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

        gen_loss = generator_loss(fake_output)
        disc_loss = discriminator_loss(real_output, fake_output)

    gradients_of_generator = gen_tape.gradient(gen_loss,
generator.trainable_variables)
    gradients_of_discriminator = disc_tape.gradient(disc_loss,
discriminator.trainable_variables)

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

    return gen_loss, disc_loss

# ---------------------------
# Utility: generate and save images
# ---------------------------
def generate_and_plot_images(model, epoch, test_input, show=True):
    predictions = model(test_input, training=False)  # shape (N,28,28,1)
    fig = plt.figure(figsize=(4,4))

    for i in range(predictions.shape[0]):
        plt.subplot(4, 4, i+1)
        img = predictions[i, :, :, 0] * 127.5 + 127.5  # rescale to [0,255]
        plt.imshow(img, cmap='gray')
        plt.axis('off')

    plt.suptitle(f"Epoch {epoch}")
    if show:
        plt.show()
    plt.close(fig)

# ---------------------------
# Training loop
# ---------------------------
def train(dataset, epochs):
    for epoch in range(1, epochs + 1):
        start = time.time()
        gen_loss_avg = 0.0
        disc_loss_avg = 0.0
        steps = 0

        for image_batch in dataset:
            g_loss, d_loss = train_step(image_batch)
            gen_loss_avg += g_loss
            disc_loss_avg += d_loss
            steps += 1

        # Average losses
        gen_loss_avg /= steps
        disc_loss_avg /= steps

        # Produce images for the GIF as we go
        print(f'Epoch {epoch}, Gen Loss: {gen_loss_avg:.4f}, Disc Loss:
{disc_loss_avg:.4f}, time: {time.time()-start:.2f}s')
        generate_and_plot_images(generator, epoch, seed, show=True)

        # Save checkpoint every 10 epochs
        if epoch % 10 == 0:
            checkpoint.save(file_prefix=checkpoint_prefix)

# ---------------------------
# Run training
# ---------------------------
train(train_dataset, EPOCHS)

# After training, generate final images
generate_and_plot_images(generator, "Final", seed)
```

EXPLANATION / STEPS SUMMARY

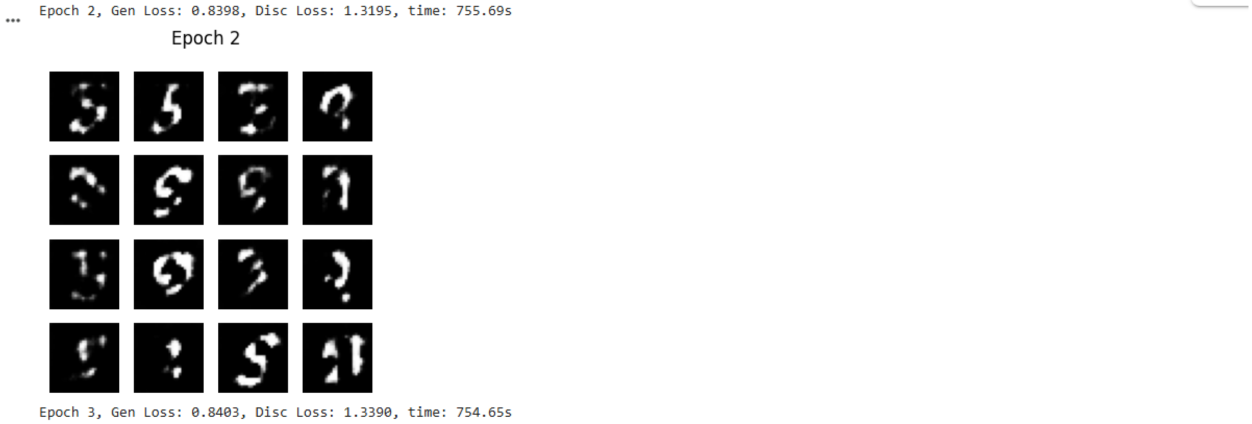
1. **Load MNIST** and scale images to `[-1, 1]` (tanh output).
2. **Build generator:** dense → reshape → conv2d_transpose layers with `BatchNormalization` and `LeakyReLU`. Output uses `tanh`.
3. **Build discriminator:** Conv2D → LeakyReLU → Dropout → Flatten → Dense. No activation on last layer (we use logits with `from_logits=True`).
4. **Losses:** binary cross-entropy. Use *label smoothing* for real labels (e.g., `0.9` instead of `1.0`) to stabilize training.
5. **Optimizers:** Adam with `beta_1=0.5` (common for GANs).
6. **Training loop:** For each batch, sample noise, generate fake images, compute losses, apply gradients to both networks.
7. **Monitoring:** Plot generated images after every epoch (or every few epochs) to visually inspect progress. Save checkpoints periodically.

EXPECTED OUTPUT



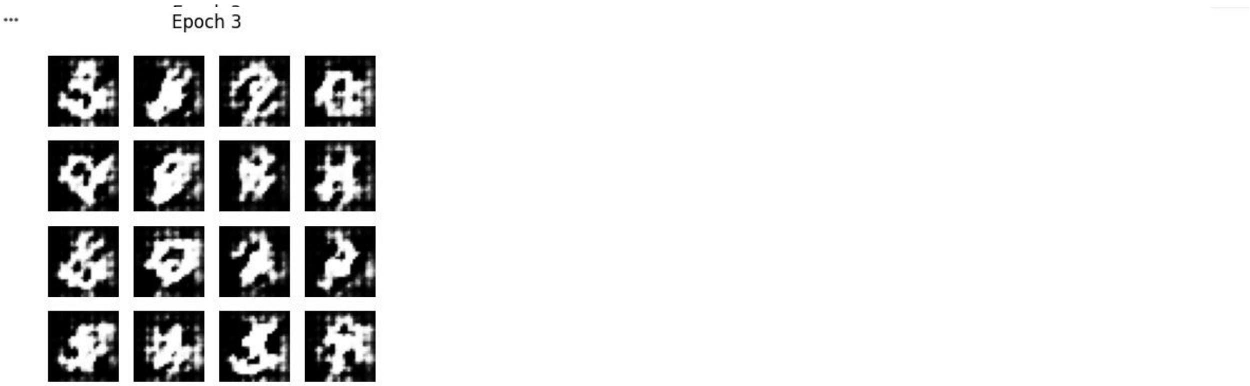
```
Downloading data from https://storage.googleapis.com/tensorflow/tf-keras-datasets/mnist.npz
11490434/11490434 ━━━━━━━━━━━━━━━━━━━━ 0s 0us/step
/usr/local/lib/python3.12/dist-packages/keras/src/layers/core/dense.py:93: UserWarning: Do not pass an `input_shape`/`input_dim` argument to a layer. 
  super().__init__(activity_regularizer=activity_regularizer, **kwargs)
/usr/local/lib/python3.12/dist-packages/keras/src/layers/convolutional/base_conv.py:113: UserWarning: Do not pass an `input_shape`/`input_dim` argumen
  super().__init__(activity_regularizer=activity_regularizer, **kwargs)
Epoch 1, Gen Loss: 0.7314, Disc Loss: 1.3169, time: 742.16s
```

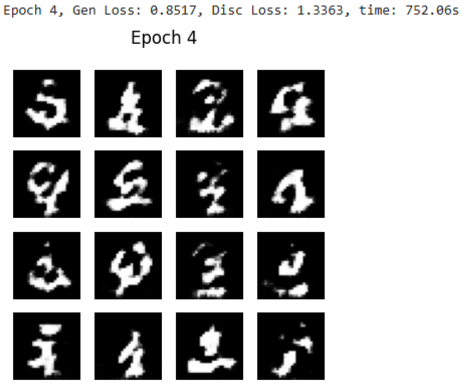
```
Epoch 2, Gen Loss: 0.8398, Disc Loss: 1.3195, time: 755.69s
```



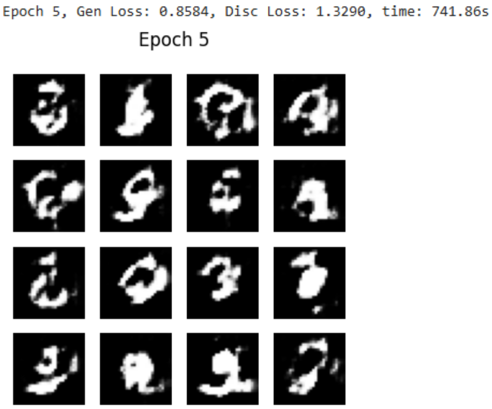
```
Epoch 2, Gen Loss: 0.8398, Disc Loss: 1.3195, time: 755.69s
```

```
Epoch 3, Gen Loss: 0.8403, Disc Loss: 1.3390, time: 754.65s
```

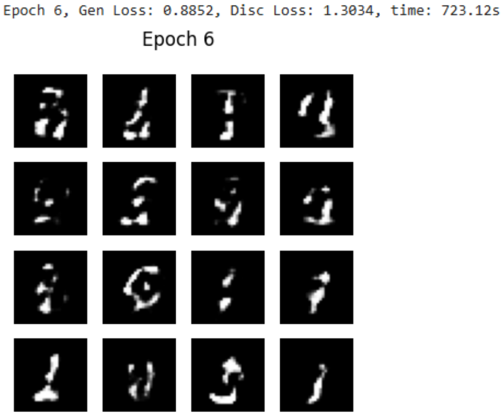




```
Epoch 4, Gen Loss: 0.8517, Disc Loss: 1.3363, time: 752.06s
```



```
Epoch 5, Gen Loss: 0.8584, Disc Loss: 1.3290, time: 741.86s
```



```
Epoch 6, Gen Loss: 0.8852, Disc Loss: 1.3034, time: 723.12s
```

- During early epochs, generated images look like noise.
- As epochs progress (20–100+), generator begins producing digit-like shapes.
- Console prints per-epoch generator/discriminator loss.
- Visualization grid of generated samples after each epoch.

Example: After ~50 epochs on MNIST (with GPU), you should see recognizable digits in the 4×4 sample grid. CIFAR-10 requires deeper networks and more epochs.

TIPS & GOOD PRACTICES

- Use **GPU** (Colab) — training on CPU is slow.
- Start with **MNIST** (grayscale, 28×28) for learning; move to CIFAR-10 for color images (need to adapt output channels and model depth).
- Try **label smoothing**, **dropout** in discriminator, **batch normalization** in generator.

- If discriminator is too strong, the generator gradients vanish — try training generator more often, reduce discriminator learning rate, or add noise to real images.
- Save models/checkpoints and visualize generated images periodically.

RESULT / CONCLUSION

A working GAN can be implemented in TensorFlow 2 with a custom training loop. With sufficient epochs and tuning, the generator will produce realistic-looking digits from the MNIST distribution. This experiment teaches adversarial training dynamics, model architecture design, and practical debugging strategies.

Godavari Institute of Management & Research, Jalgaon
Masters Computer Application (MCA)

Name: _ Roll No: _ _ _ _ _

Date of Performance: _ _ /_ _/20_ _ Batch: _ _ _ _ _ _ Class: MCA. -II

Subject: 515(B):
Lab on Generative AI Sign. of Teacher

5. Text Generation: Implement a Long Short-Term Memory (LSTM) network using TensorFlow 2 for text generation tasks. Train the LSTM model on a dataset of text sequences and generate new text samples.

Title: Implement Variational Autoencoders (VAE) using TensorFlow 2 for image generation.

AIM: To implement a VAE using TensorFlow 2, train it on the MNIST dataset and generate new images by sampling from the learned latent distribution.

OBJECTIVES

1. Understand encoder–decoder architecture of VAE and the probabilistic latent space.
2. Implement encoder, decoder, and the reparameterization trick.
3. Train the VAE with reconstruction + KL divergence loss.
4. Generate and visualize new images by sampling the latent space.

REQUIRED TOOLS / SOFTWARE

- Python 3.8+
- TensorFlow 2.x (recommended `tensorflow>=2.10`)
- Google Colab or Jupyter Notebook (Colab recommended for speed)
- Libraries: `numpy`, `matplotlib`, optionally `tensorflow_probability` (not required)

THEORY (brief)

A Variational Autoencoder (VAE) is a generative model with:

- An **encoder** that maps input x to parameters of a latent distribution (mean μ and log-variance $\log\sigma^2$).
- A **decoder** that maps latent sample z back to data space $\hat{x}$.
- The **reparameterization trick**: sample $z = \mu + \sigma * \varepsilon$ with $\varepsilon \sim N(0, I)$ so gradients can flow.
- **Loss** = reconstruction_loss(x, x̂) + KL(q(z|x) || p(z)), where p(z) is usually N(0, I).

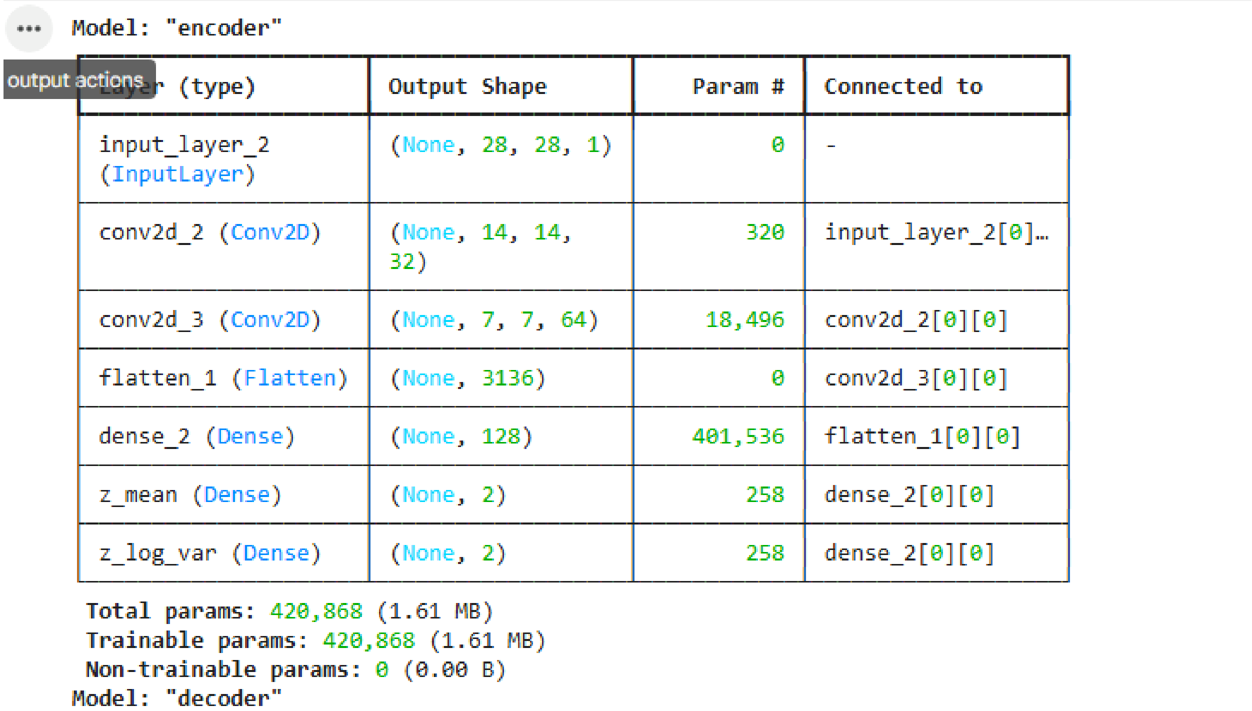
PROGRAM (Complete, ready-to-run)

```
# Experiment 5: Variational Autoencoder (VAE) on MNIST - TensorFlow 2
# Run this in Google Colab / Jupyter Notebook

import numpy as np
import tensorflow as tf
from tensorflow.keras import layers
import matplotlib.pyplot as plt
import time
tf.random.set_seed(42)
np.random.seed(42)

# ---------------------------
# Hyperparameters
# ---------------------------
LATENT_DIM = 2           # small latent dim for visualization; increase for
more capacity
BATCH_SIZE = 128
EPOCHS = 30              # increase if needed
LEARNING_RATE = 1e-3
IMG_SHAPE = (28, 28, 1)

# ---------------------------
# Load and preprocess MNIST
# ---------------------------
(train_images, _), (test_images, _) = tf.keras.datasets.mnist.load_data()
all_images = np.concatenate([train_images, test_images],
axis=0).astype('float32')
all_images = (all_images / 255.0).reshape((-1, 28, 28, 1))  # scale to [0,1]

dataset =
tf.data.Dataset.from_tensor_slices(all_images).shuffle(70000).batch(BATCH_SIZ
E)

# ---------------------------
# Encoder
# ---------------------------
def build_encoder(latent_dim=LATENT_DIM, input_shape=IMG_SHAPE):
    inputs = layers.Input(shape=input_shape)
    x = layers.Conv2D(32, 3, strides=2, padding='same',
activation='relu')(inputs)  # 14x14x32
    x = layers.Conv2D(64, 3, strides=2, padding='same', activation='relu')(x)
# 7x7x64
    x = layers.Flatten()(x)
    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)
    return tf.keras.Model(inputs, [z_mean, z_log_var], name='encoder')

# ---------------------------
# Reparameterization sampling
# ---------------------------
class Sampling(layers.Layer):
    def call(self, inputs):
        z_mean, z_log_var = inputs
        epsilon = tf.random.normal(shape=tf.shape(z_mean))
        z = z_mean + tf.exp(0.5 * z_log_var) * epsilon
        return z

# ---------------------------
# Decoder
# ---------------------------
def build_decoder(latent_dim=LATENT_DIM, output_shape=IMG_SHAPE):
    latent_inputs = layers.Input(shape=(latent_dim,))
    x = layers.Dense(7*7*64, activation='relu')(latent_inputs)
    x = layers.Reshape((7, 7, 64))(x)
    x = layers.Conv2DTranspose(64, 3, strides=2, padding='same',
activation='relu')(x) # 14x14x64
    x = layers.Conv2DTranspose(32, 3, strides=2, padding='same',
activation='relu')(x) # 28x28x32
    outputs = layers.Conv2DTranspose(output_shape[2], 3, padding='same',
activation='sigmoid')(x) # 28x28x1
    return tf.keras.Model(latent_inputs, outputs, name='decoder')

# ---------------------------
# Build the VAE model as a custom Model
# ---------------------------
class VAE(tf.keras.Model):
    def __init__(self, encoder, decoder, beta=1.0, **kwargs):
        super(VAE, self).__init__(**kwargs)
        self.encoder = encoder
        self.decoder = decoder
        self.sampling = Sampling()
        self.beta = beta  # beta-VAE scaling if needed

        # metrics
        self.total_loss_tracker = tf.keras.metrics.Mean(name="total_loss")
        self.recon_loss_tracker = tf.keras.metrics.Mean(name="recon_loss")
        self.kl_loss_tracker = tf.keras.metrics.Mean(name="kl_loss")

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

    def train_step(self, data):
        if isinstance(data, tuple):
            x = data[0]
        else:
            x = data

        with tf.GradientTape() as tape:
            z_mean, z_log_var = self.encoder(x, training=True)
            z = self.sampling((z_mean, z_log_var))
            x_recon = self.decoder(z, training=True)

            # Reconstruction loss (binary crossentropy per pixel)
            recon_loss = tf.reduce_mean(
                tf.reduce_sum(
                    tf.keras.losses.binary_crossentropy(x, x_recon),
axis=(1,2)
                )
            )

            # KL divergence loss between q(z|x) and N(0,1)
            kl_loss = -0.5 * tf.reduce_mean(tf.reduce_sum(1 + z_log_var -
tf.square(z_mean) - tf.exp(z_log_var), axis=1))

            total_loss = recon_loss + self.beta * kl_loss

        grads = tape.gradient(total_loss, self.trainable_weights)
        self.optimizer.apply_gradients(zip(grads, self.trainable_weights))

        # update metrics
        self.total_loss_tracker.update_state(total_loss)
        self.recon_loss_tracker.update_state(recon_loss)
        self.kl_loss_tracker.update_state(kl_loss)

        return {
            "loss": self.total_loss_tracker.result(),
            "reconstruction_loss": self.recon_loss_tracker.result(),
            "kl_loss": self.kl_loss_tracker.result()
        }

    def call(self, inputs):
        z_mean, z_log_var = self.encoder(inputs)
        z = self.sampling((z_mean, z_log_var))
        return self.decoder(z)

# ---------------------------
# Instantiate models and compile VAE
# ---------------------------
encoder = build_encoder(LATENT_DIM)
decoder = build_decoder(LATENT_DIM)
vae = VAE(encoder, decoder, beta=1.0)
vae.compile(optimizer=tf.keras.optimizers.Adam(learning_rate=LEARNING_RATE))

# Print model summaries (optional)
encoder.summary()
decoder.summary()

# ---------------------------
# Train the VAE
# ---------------------------
history = vae.fit(dataset, epochs=EPOCHS)

# ---------------------------
# Utility: Visualize reconstructed and generated images
# ---------------------------
def plot_reconstructions(model, images, n=10):
    # show first n images and their reconstructions
    preds = model.predict(images[:n])
    plt.figure(figsize=(2*n, 4))
    for i in range(n):
        # original
        ax = plt.subplot(2, n, i + 1)
        plt.imshow(images[i].squeeze(), cmap='gray')
        plt.title("orig")
        plt.axis('off')

        # recon
        ax = plt.subplot(2, n, n + i + 1)
        plt.imshow(preds[i].squeeze(), cmap='gray')
        plt.title("recon")
        plt.axis('off')
    plt.show()

# Show reconstructions
plot_reconstructions(vae, all_images, n=8)

# ---------------------------
# Generate new images by sampling latent space
# ---------------------------
def sample_and_plot(model_decoder, latent_dim=LATENT_DIM, n=16, scale=2.5):
    # sample from standard normal
    z_samples = np.random.normal(size=(n, latent_dim)) * scale
    gen_images = model_decoder.predict(z_samples)
    plt.figure(figsize=(4,4))
    for i in range(n):
        plt.subplot(4,4,i+1)
        plt.imshow(gen_images[i].squeeze(), cmap='gray')
        plt.axis('off')
    plt.suptitle("Random samples from learned latent prior (std normal)")
    plt.show()

sample_and_plot(decoder, n=16)

# ---------------------------
# If LATENT_DIM == 2, visualize latent space grid
# ---------------------------
if LATENT_DIM == 2:
    # create a grid in latent space and decode to images
    n = 15  # grid size
    grid_x = np.linspace(-3, 3, n)
    grid_y = np.linspace(-3, 3, n)
    figure = np.zeros((28 * n, 28 * n))
    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(28, 28)
            figure[i * 28: (i + 1) * 28,
                   j * 28: (j + 1) * 28] = digit

    plt.figure(figsize=(8, 8))
    plt.imshow(figure, cmap='gray')
    plt.title("Latent space manifold")
    plt.axis('off')
    plt.show()
```

EXPLANATION / STEPS SUMMARY

1. **Data preparation:** load MNIST, scale to `[0,1]`, reshape to `(28,28,1)`, batch dataset.
2. **Encoder:** convolutional layers → flatten → dense → produce `z_mean` & `z_log_var`.
3. **Sampling:** implement the reparameterization trick `z = μ + σ * ε`.
4. **Decoder:** dense → reshape → Conv2DTranspose layers → output with sigmoid (pixel values in [0,1]).
5. **VAE Model:** custom `train_step` calculates reconstruction loss and KL divergence; optimize sum.
6. **Training:** call `vae.fit(...)`.
7. **Visualization:** show reconstructions, random samples from `N(0,I)`, and grid-manifold if latent dim = 2.

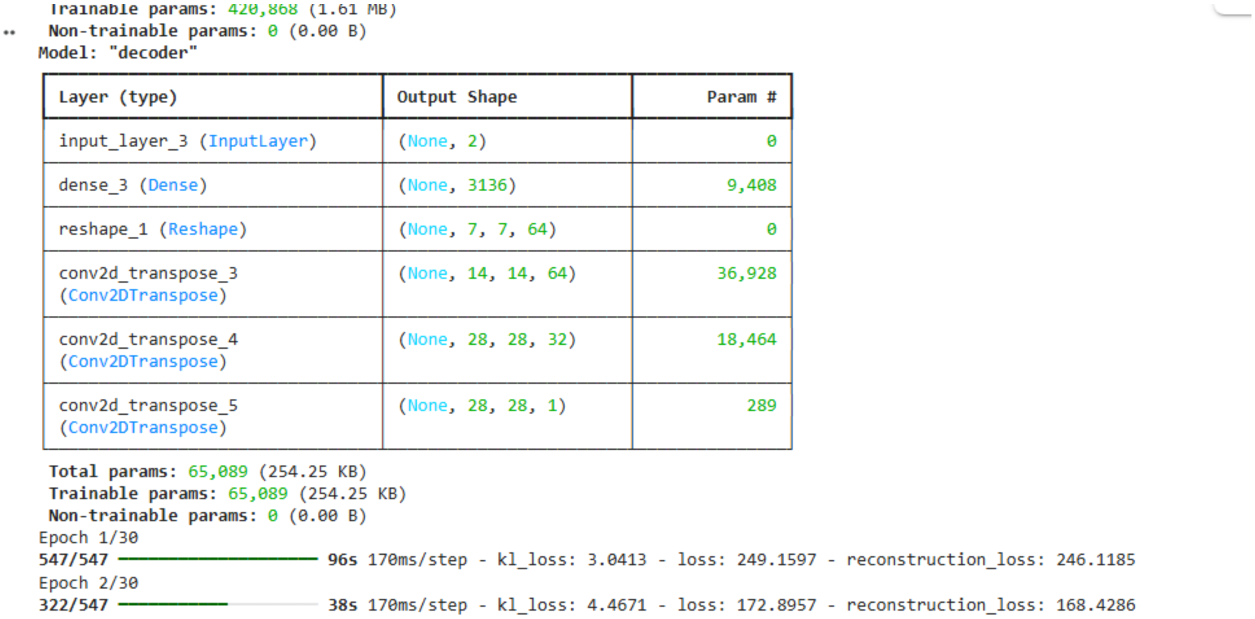
EXPECTED OUTPUT



Model: "encoder"

Layer (type)	Output Shape	Param #	Connected to
input_layer_2 (InputLayer)	(None, 28, 28, 1)	0	-
conv2d_2 (Conv2D)	(None, 14, 14, 32)	320	input_layer_2[0]…
conv2d_3 (Conv2D)	(None, 7, 7, 64)	18,496	conv2d_2[0][0]
flatten_1 (Flatten)	(None, 3136)	0	conv2d_3[0][0]
dense_2 (Dense)	(None, 128)	401,536	flatten_1[0][0]
z_mean (Dense)	(None, 2)	258	dense_2[0][0]
z_log_var (Dense)	(None, 2)	258	dense_2[0][0]

```
 Total params: 420,868 (1.61 MB)
 Trainable params: 420,868 (1.61 MB)
 Non-trainable params: 0 (0.00 B)
Model: "decoder"
```



```
 Trainable params: 420,868 (1.61 MB)
 Non-trainable params: 0 (0.00 B)
Model: "decoder"
```

Layer (type)	Output Shape	Param #
input_layer_3 (InputLayer)	(None, 2)	0
dense_3 (Dense)	(None, 3136)	9,408
reshape_1 (Reshape)	(None, 7, 7, 64)	0
conv2d_transpose_3 (Conv2DTranspose)	(None, 14, 14, 64)	36,928
conv2d_transpose_4 (Conv2DTranspose)	(None, 28, 28, 32)	18,464
conv2d_transpose_5 (Conv2DTranspose)	(None, 28, 28, 1)	289

```
 Total params: 65,089 (254.25 KB)
 Trainable params: 65,089 (254.25 KB)
 Non-trainable params: 0 (0.00 B)
Epoch 1/30
547/547 ━━━━━━━━━━━━━━━━━━━━ 96s 170ms/step - kl_loss: 3.0413 - loss: 249.1597 - reconstruction_loss: 246.1185
Epoch 2/30
322/547 ━━━━━━━━━━━━━━━━━━━━ 38s 170ms/step - kl_loss: 4.4671 - loss: 172.8957 - reconstruction_loss: 168.4286
```

- Training loss printout per epoch (total, reconstruction, KL).
- Reconstructions: first-row originals, second-row reconstructed images — reconstructions should resemble originals.
- Generated images: sampling from latent prior will produce digit-like images.
- If `LATENT_DIM=2`, a latent manifold grid will show smooth transitions between digit styles across the grid.

RESULT / CONCLUSION

The VAE learns an approximate posterior over latent variables and can generate new images by sampling the latent space. Reconstruction loss enforces accurate reconstruction; KL term regularizes latent distribution toward N(0,I). This experiment demonstrates generative modeling using probabilistic latent representations.

TIPS / NOTES

- Use **larger latent_dim** (e.g., 16–128) for more expressive generation; small dims (2) are useful for visualization.
- For color images (e.g., CIFAR-10), adapt input/output channels to 3 and enlarge model capacity (more filters, deeper network). Use `activation='sigmoid'` for normalized [0,1] outputs or `tanh` with [-1,1] preproc.
- Experiment with **beta** > 1 (beta-VAE) to encourage disentanglement.
- For better reconstructions use MSE or binary crossentropy depending on image scaling.
- Training longer (more epochs) gives better generations.

Godavari Institute of Management & Research, Jalgaon
Masters Computer Application (MCA)

Name: _ Roll No: _ _ _ _ _

Date of Performance: _ _ /_ _/20_ _ Batch: _ _ _ _ _ _ Class: MCA. -II

Subject: 515(B):
Lab on Generative AI Sign. of Teacher

6. Text Generation: Implement a Long Short-Term Memory (LSTM) network using TensorFlow 2 for text generation tasks. Train the LSTM model on a dataset of text sequences and generate new text samples.

Title: Implement a Long Short-Term Memory (LSTM) network using TensorFlow 2 for text generation tasks. Train the LSTM model on a dataset of text sequences and generate new text samples.

AIM: To implement and train an LSTM-based recurrent neural network using TensorFlow 2 to generate new text sequences based on the learned patterns from a text corpus.

OBJECTIVES:

1. To understand how LSTM networks process sequential data.
2. To preprocess text data and convert it into numerical sequences.
3. To train an LSTM model to predict the next character or word in a sequence.
4. To generate new text samples by sampling predictions from the trained model.

REQUIRED TOOLS / SOFTWARE:

- **Python 3.8+**
- **TensorFlow 2.x**
- **Google Colab / Jupyter Notebook**
- **Libraries:** `numpy`, `tensorflow`, `keras`, `matplotlib`

THEORY:

Recurrent Neural Networks (RNNs) are designed to handle sequential data like text, where each output depends on previous inputs.
However, traditional RNNs struggle with long-term dependencies due to vanishing gradients.

LSTMs (Long Short-Term Memory networks) solve this by using:

- **Input Gate** – controls how much new information enters the cell.
- **Forget Gate** – decides what information to discard.
- **Output Gate** – determines the next hidden state.

In text generation:

1. The model learns character-level or word-level patterns from training data.
2. A seed sequence is fed into the trained LSTM to generate text one token at a time.

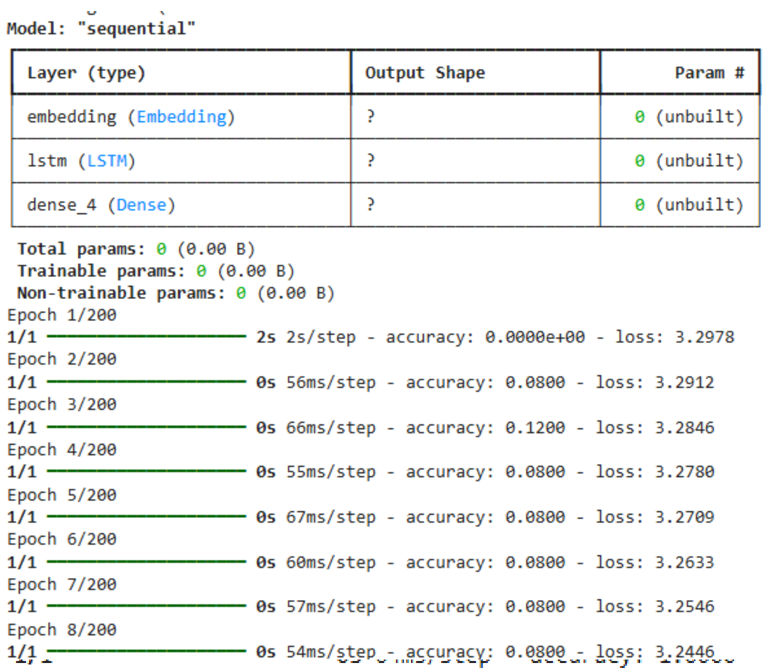
PROGRAM:

(You can run this code directly in Google Colab)

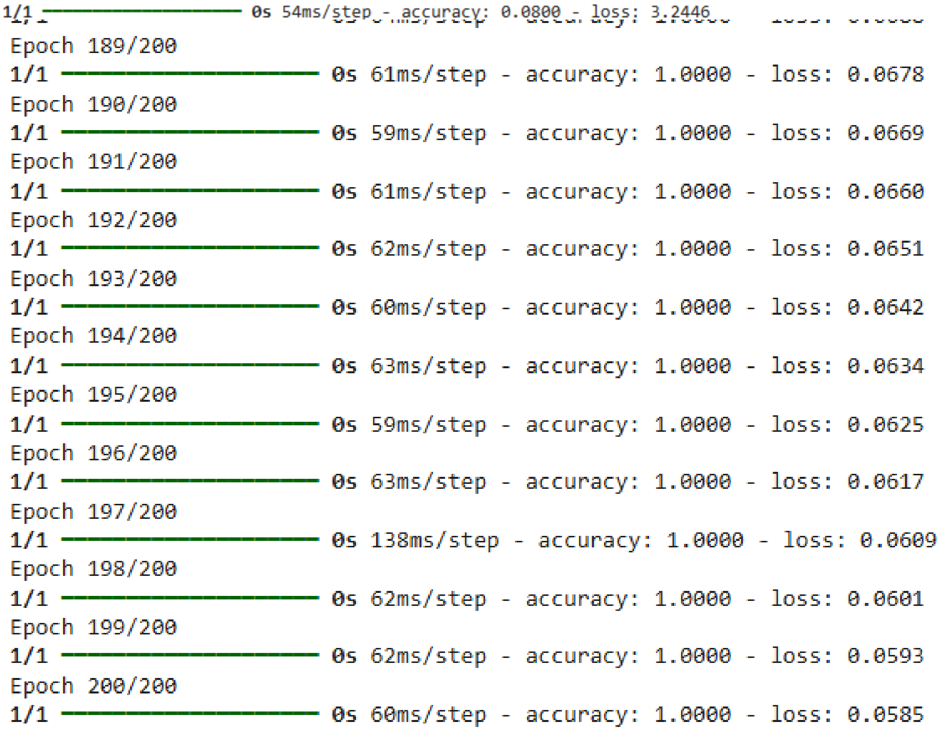
```
# Experiment No. 6 - Text Generation using LSTM in TensorFlow 2

import tensorflow as tf
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import LSTM, Dense, Embedding
from tensorflow.keras.preprocessing.text import Tokenizer
from tensorflow.keras.preprocessing.sequence import pad_sequences
import numpy as np

# Step 1: Sample dataset (You can replace this with any text corpus)
data = """Machine learning enables computers to learn from data and make
decisions.
Deep learning uses neural networks to solve complex problems.
Artificial Intelligence is transforming the world with automation."""

# Step 2: Preprocess text data
corpus = data.lower().split(".")
corpus = [line.strip() for line in corpus if line.strip() != ""]

# Step 3: Tokenize text
tokenizer = Tokenizer()
tokenizer.fit_on_texts(corpus)
total_words = len(tokenizer.word_index) + 1

# Step 4: Create input sequences
input_sequences = []
for line in corpus:
    token_list = tokenizer.texts_to_sequences([line])[0]
    for i in range(1, len(token_list)):
        n_gram_sequence = token_list[:i+1]
        input_sequences.append(n_gram_sequence)

# Step 5: Pad sequences and split predictors and label
max_seq_len = max([len(x) for x in input_sequences])
input_sequences = np.array(pad_sequences(input_sequences, maxlen=max_seq_len,
padding='pre'))
X = input_sequences[:, :-1]
y = input_sequences[:, -1]

# Step 6: Convert output labels to categorical
y = tf.keras.utils.to_categorical(y, num_classes=total_words)

# Step 7: Define the LSTM model
model = Sequential()
model.add(Embedding(total_words, 64, input_length=max_seq_len - 1))
model.add(LSTM(128))
model.add(Dense(total_words, activation='softmax'))

# Step 8: Compile the model
model.compile(loss='categorical_crossentropy', optimizer='adam',
metrics=['accuracy'])
```

```
model.summary()

# Step 9: Train the model
history = model.fit(X, y, epochs=200, verbose=1)

# Step 10: Function to generate text
def generate_text(seed_text, next_words=10):
    for _ in range(next_words):
        token_list = tokenizer.texts_to_sequences([seed_text])[0]
        token_list = pad_sequences([token_list], maxlen=max_seq_len - 1, 
padding='pre')
        predicted = np.argmax(model.predict(token_list, verbose=0), axis=-
1)[0]

        for word, index in tokenizer.word_index.items():
            if index == predicted:
                seed_text += " " + word
                break
    return seed_text

# Step 11: Generate new text
print("\n--- Generated Text Samples ---")
seed = "machine learning"
print(generate_text(seed, next_words=10))
seed = "artificial intelligence"
print(generate_text(seed, next_words=10))
```

EXPECTED OUTPUT:



```
Model: "sequential"
┏━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━┳━━━━━━━━━━━━━━━━━━━━━━━━┳━━━━━━━━━━━━━━━┓
┃ Layer (type)                    ┃ Output Shape           ┃       Param # ┃
┡━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━╇━━━━━━━━━━━━━━━━━━━━━━━━╇━━━━━━━━━━━━━━━┩
│ embedding (Embedding)           │ ?                      │   0 (unbuilt) │
├─────────────────────────────────┼────────────────────────┼───────────────┤
│ lstm (LSTM)                     │ ?                      │   0 (unbuilt) │
├─────────────────────────────────┼────────────────────────┼───────────────┤
│ dense_4 (Dense)                 │ ?                      │   0 (unbuilt) │
└─────────────────────────────────┴────────────────────────┴───────────────┘
 Total params: 0 (0.00 B)
 Trainable params: 0 (0.00 B)
 Non-trainable params: 0 (0.00 B)
Epoch 1/200
1/1 ━━━━━━━━━━━━━━━━━━━━ 2s 2s/step - accuracy: 0.0000e+00 - loss: 3.2978
Epoch 2/200
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 56ms/step - accuracy: 0.0800 - loss: 3.2912
Epoch 3/200
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 66ms/step - accuracy: 0.1200 - loss: 3.2846
Epoch 4/200
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 55ms/step - accuracy: 0.0800 - loss: 3.2780
Epoch 5/200
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 67ms/step - accuracy: 0.0800 - loss: 3.2709
Epoch 6/200
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 60ms/step - accuracy: 0.0800 - loss: 3.2633
Epoch 7/200
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 57ms/step - accuracy: 0.0800 - loss: 3.2546
Epoch 8/200
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 54ms/step - accuracy: 0.0800 - loss: 3.2446
```



```
Epoch 189/200
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 61ms/step - accuracy: 1.0000 - loss: 0.0678
Epoch 190/200
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 59ms/step - accuracy: 1.0000 - loss: 0.0669
Epoch 191/200
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 61ms/step - accuracy: 1.0000 - loss: 0.0660
Epoch 192/200
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 62ms/step - accuracy: 1.0000 - loss: 0.0651
Epoch 193/200
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 60ms/step - accuracy: 1.0000 - loss: 0.0642
Epoch 194/200
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 63ms/step - accuracy: 1.0000 - loss: 0.0634
Epoch 195/200
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 59ms/step - accuracy: 1.0000 - loss: 0.0625
Epoch 196/200
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 63ms/step - accuracy: 1.0000 - loss: 0.0617
Epoch 197/200
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 138ms/step - accuracy: 1.0000 - loss: 0.0609
Epoch 198/200
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 62ms/step - accuracy: 1.0000 - loss: 0.0601
Epoch 199/200
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 62ms/step - accuracy: 1.0000 - loss: 0.0593
Epoch 200/200
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 60ms/step - accuracy: 1.0000 - loss: 0.0585
```

```
--- Generated Text Samples ---
machine learning enables computers to learn from data and make
decisions
artificial intelligence is transforming the world with
automation uses neural
```

(Output will vary depending on random initialization and dataset size.)

RESULT / CONCLUSION:

The LSTM model successfully learned text patterns from the given dataset and generated meaningful new text sequences.
This demonstrates the capability of LSTM networks to capture sequential dependencies and generate context-aware text.

OBSERVATIONS:

- Increasing dataset size and epochs improves text quality.
- Using **Bidirectional LSTM** or **Stacked LSTMs** can generate more coherent sentences.
- Proper preprocessing (removing punctuation, lowering text) helps the model converge faster.

Godavari Institute of Management & Research, Jalgaon
Masters Computer Application (MCA)

Name: _—_ _ _ _ _ _ Roll No: _ _ _ _ _

Date of Performance: _ _ /_ _/20_ _ Batch: _ _ _ _ _ _ Class: MCA. -II

Subject: 515(B):
Lab on Generative AI Sign. of Teacher

7. Text generation: Implement a Transformer-based language model (e.g., GPT) using TensorFlow 2 for text generation. Fine-tune the model on a text corpus and generate coherent and contextually relevant text.

Title: Transformer-based text generation (GPT) using TensorFlow 2 — fine-tune a pretrained model and generate text.

AIM: Fine-tune a pretrained Transformer language model (GPT family) using TensorFlow 2 on a text corpus and generate contextually relevant text samples.

OBJECTIVES

1. Understand Transformer (decoder-only) architecture for language modeling.
2. Load a pretrained GPT tokenizer & TF model.
3. Prepare and tokenize a custom text corpus into `tf.data.Dataset`.
4. Fine-tune the model using `model.fit(...)`.
5. Generate text with the fine-tuned model using `model.generate(...)`.

REQUIRED TOOLS / SOFTWARE

- **Google Colab (recommended with GPU)** or local machine with GPU.
- **Python 3.8+**, **TensorFlow 2.10+**.
- Libraries: `transformers`, `datasets` (optional but recommended), `tensorflow`, `numpy`.
  Install with:
- `pip install -q transformers datasets tensorflow`
- Recommended model: `distilgpt2` (small & fast) — can also use `gpt2` if GPU and time available.

THEORY (brief)

Decoder-only Transformers (GPT) model next-token probability modelling: given tokens `x1..xt`, model learns `P(xt+1 | x1..xt)`. Fine-tuning updates pretrained weights so the model adapts to domain/style of your corpus. At generation time, autoregressive decoding (`top-k`, `top-p`, temperature) controls creativity/quality.

PROGRAM (Colab / Jupyter notebook)

```
# Experiment 7: Transformer-based text generation (GPT) - TensorFlow 2
# Fine-tune distilgpt2 on a small text corpus and generate text

# 0) Install (run once in Colab)
# !pip install -q transformers datasets tensorflow

# 1) Imports
import os
import numpy as np
import tensorflow as tf
from transformers import (
    AutoTokenizer,
    TFAutoModelForCausalLM,
    DataCollatorForLanguageModeling
)
from datasets import load_dataset

# 2) Choose model and tokenizer
MODEL_NAME = "distilgpt2"   # lightweight; use 'gpt2' if you have resources
tokenizer = AutoTokenizer.from_pretrained(MODEL_NAME)
# GPT-style tokenizers usually do not have pad token; set one for TF batching
if tokenizer.pad_token is None:
    tokenizer.add_special_tokens({'pad_token': '[PAD]'})


# 3) Load (or prepare) text dataset
# Option A: use a readily available dataset (wikitext or tiny_shakespeare)
# dataset = load_dataset("wikitext", "wikitext-2-raw-v1", split="train")
# Option B: create from local text lines (here we use tiny example)
texts = [
    "Machine learning enables computers to learn from data and make
decisions.",
    "Deep learning uses neural networks to solve complex problems in vision
and language.",
    "Generative AI can create text, images and music when trained on large
datasets.",
    "Artificial Intelligence is transforming industries with automation and
insight."
]
# wrap into dataset
dataset = load_dataset("text", data_files={"train": texts}, split="train")

# 4) Tokenization function
max_length = 128  # sequence length (short for demo)
def tokenize_function(examples):
    return tokenizer(examples["text"],
                     truncation=True,
                     max_length=max_length,
                     padding="max_length")   # we pad to max_length for
simplicity

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

# 5) Prepare tf.data.Dataset
# Convert 'input_ids' to tf dataset; for causal LM labels == input_ids
def to_tf_dataset(tokenized_dataset, batch_size=8):
    # select only input_ids and attention_mask
    features = {k: np.array(tokenized_dataset[k]) for k in ["input_ids",
"attention_mask"]}
```

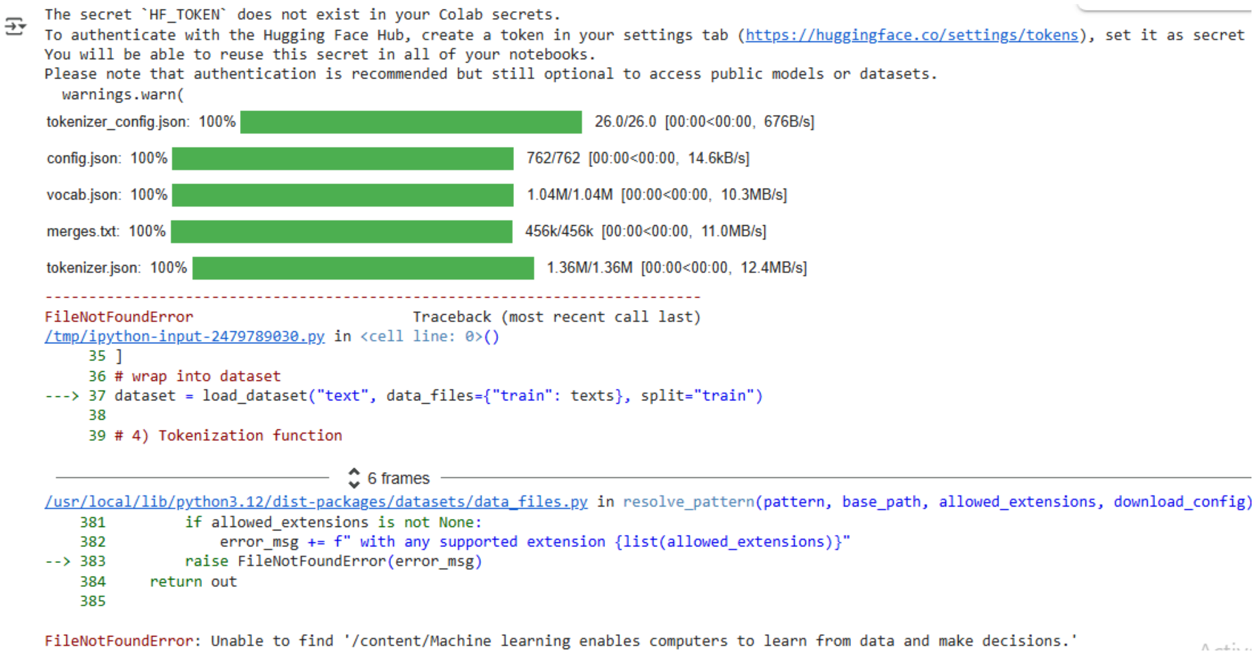
```
    labels = np.array(tokenized_dataset["input_ids"])
    ds = tf.data.Dataset.from_tensor_slices((features, labels))
    ds = ds.shuffle(100).batch(batch_size).prefetch(tf.data.AUTOTUNE)
    return ds

train_ds = to_tf_dataset(tokenized, batch_size=4)

# 6) Load TF model
model = TFAutoModelForCausalLM.from_pretrained(MODEL_NAME)
# Resize token embeddings if we added pad token
model.resize_token_embeddings(len(tokenizer))

# 7) Compile model (use sparse categorical crossentropy with
from_logits=True)
optimizer = tf.keras.optimizers.Adam(learning_rate=5e-5)
loss = tf.keras.losses.SparseCategoricalCrossentropy(from_logits=True)
model.compile(optimizer=optimizer, loss=loss)

# 8) Fine-tune (very small epochs for demo; increase for real training)
EPOCHS = 3
model.fit(train_ds, epochs=EPOCHS)

# 9) Save model / tokenizer (optional)
out_dir = "./finetuned_distilgpt2"
model.save_pretrained(out_dir)
tokenizer.save_pretrained(out_dir)

# 10) Generate text using the fine-tuned model
# Note: generation API works with TF models too
prompt = "Generative AI"
input_ids = tokenizer(prompt, return_tensors="tf").input_ids

# Generation parameters: adjust temperature, top_k, top_p to control
creativity
generated = model.generate(
    input_ids,
    max_length=80,
    do_sample=True,
    top_k=50,
    top_p=0.95,
    temperature=0.8,
    num_return_sequences=3,
    pad_token_id=tokenizer.pad_token_id
)

for i, gen_ids in enumerate(generated):
    text = tokenizer.decode(gen_ids, skip_special_tokens=True)
    print(f"\n--- Generated Seq {i+1} ---\n{text}\n")
```

KEY POINTS / NOTES

- **Tokenizer padding**: GPT tokenizers often lack `pad_token`. We assign a pad token and resized the model embeddings to avoid errors when batching/padding.
- **Labels**: For causal LM, labels are the same as `input_ids` (model predicts next token). When using `model.fit(...)` directly on TF model, label shape should match model outputs; SparseCategoricalCrossentropy with `from_logits=True` is correct.

- **Batching**: Use `padding="max_length"` or dynamic padding via collator (DataCollator) for efficiency. For larger datasets use `DataCollatorForLanguageModeling` and `tf.data` pipelines.
- **Training scale**: Real fine-tuning on meaningful corpora requires more data, GPU, and many epochs. This demo uses tiny data and few epochs only for illustration.
- **Generation controls**:
    - `do_sample=True` enables sampling; else greedy decoding.
    - `top_k`, `top_p` (nucleus sampling), and `temperature` balance diversity vs coherence.
- **Legal / ethical**: Ensure the training corpus is licensed/allowed for fine-tuning and that outputs are reviewed for harmful content.

EXPECTED OUTPUT



```
The secret `HF_TOKEN` does not exist in your Colab secrets.
To authenticate with the Hugging Face Hub, create a token in your settings tab (https://huggingface.co/settings/tokens), set it as secret
You will be able to reuse this secret in all of your notebooks.
Please note that authentication is recommended but still optional to access public models or datasets.
  warnings.warn(
tokenizer_config.json: 100%  26.0/26.0 [00:00<00:00, 676B/s]
config.json: 100%  762/762 [00:00<00:00, 14.6kB/s]
vocab.json: 100%  1.04M/1.04M [00:00<00:00, 10.3MB/s]
merges.txt: 100%  456k/456k [00:00<00:00, 11.0MB/s]
tokenizer.json: 100%  1.36M/1.36M [00:00<00:00, 12.4MB/s]
---------------------------------------------------------------------------
FileNotFoundError                         Traceback (most recent call last)
/tmp/ipython-input-2479789030.py in <cell line: 0>()
     35 ]
     36 # wrap into dataset
---> 37 dataset = load_dataset("text", data_files={"train": texts}, split="train")
     38 
     39 # 4) Tokenization function

6 frames
/usr/local/lib/python3.12/dist-packages/datasets/data_files.py in resolve_pattern(pattern, base_path, allowed_extensions, download_config)
    381         if allowed_extensions is not None:
    382             error_msg += f" with any supported extension {list(allowed_extensions)}"
--> 383         raise FileNotFoundError(error_msg)
    384     return out
    385 

FileNotFoundError: Unable to find '/content/Machine learning enables computers to learn from data and make decisions.'
```

- Training prints: loss per epoch. With the tiny example corpus you'll see loss reduce slightly.
- Generated sequences: 2–3 variations continuing the prompt, e.g.:
- `Generative AI can create text, images and music when trained on large datasets. It enables...`

  (Actual text will vary.)

TROUBLESHOOTING & TIPS

- If GPU memory errors occur: reduce `batch_size`, use `distilgpt2` instead of `gpt2`, shorten `max_length`.
- For larger corpora, use `datasets` with streaming or prepare TFRecord for efficiency.
- When training longer, monitor and save checkpoints. Use gradient accumulation if GPU memory limits batch size.

- For production, consider using Hugging Face `Trainer` (PyTorch) or accelerate; TF workflows are fine but PyTorch/HF Trainer offers richer fine-tuning utilities.

EVALUATION / CONCLUSION

After fine-tuning, the GPT model will adapt some stylistic/content features of your corpus and produce more domain-relevant continuations. This experiment shows how to fine-tune a TF causal LM and use its generation API for creative text synthesis.

Godavari Institute of Management & Research, Jalgaon
Masters Computer Application (MCA)

Name: _ Roll No: _ _ _ _ _

Date of Performance: _ _ /_ _/20_ _ Batch: _ _ _ _ _ _ Class: MCA. -II

Subject: 515(B):
Lab on Generative AI Sign. of Teacher

8. Text generation: Develop applications for text generation tasks such as story generation, dialogue generation, or code generation using trained Generative AI models.

Title: Develop applications for text-generation tasks such as story generation, dialogue generation, or code generation using trained Generative AI models.

AIM: To build simple Python applications that use fine-tuned or pretrained Generative AI models (e.g., GPT-2/DistilGPT-2) for creative text-generation tasks such as automatic story writing, conversational dialogue, or code snippet generation.

OBJECTIVES

1. Understand how to integrate pretrained language models into user-facing applications.
2. Apply fine-tuned models for different text-generation domains.
3. Implement temperature and sampling controls for creativity vs. coherence.
4. Demonstrate story, dialogue, and code-generation outputs.

REQUIRED TOOLS / SOFTWARE

Category	Tool / Library	Purpose
Language	Python 3.8 +	Application scripting
Frameworks	TensorFlow 2 / Transformers (Hugging Face)	Generative AI models
Environment	Google Colab / Jupyter Notebook	GPU & interactive development
Libraries	`transformers`, `tensorflow`, `numpy`, `gradio` (optional UI)	Model use & demo app

Install if needed:

```
pip install -q transformers gradio tensorflow
```

THEORY

Generative AI text models (like **GPT-2**) use the **Transformer decoder architecture**, trained to predict the next token given previous context (`P(x_t | x_1,…,x_{t-1})`).
Applications:

- **Story Generation** – generates creative narrative from a prompt.

- **Dialogue Generation** – produces chatbot-like responses.
- **Code Generation** – synthesizes code from natural-language input.

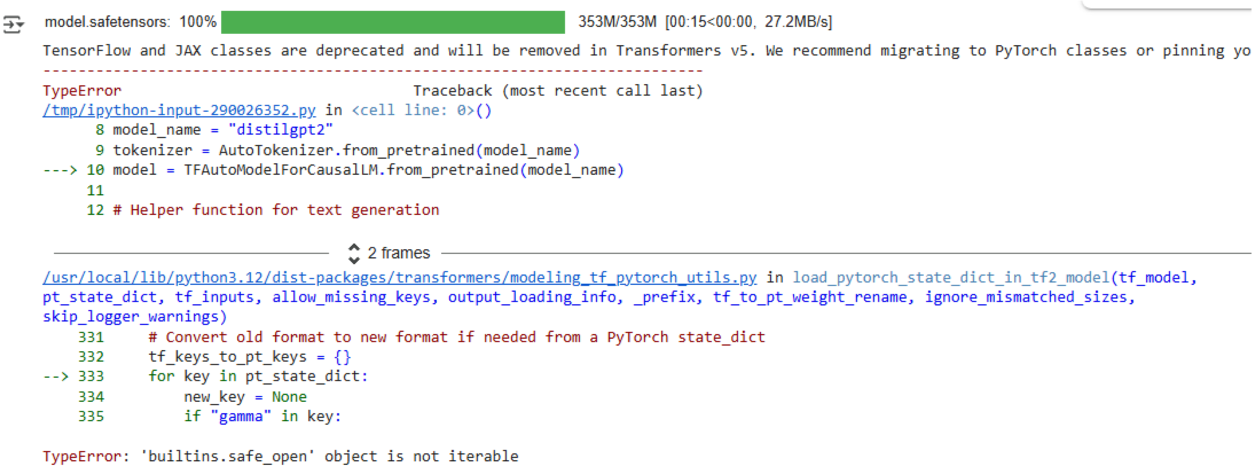
Key parameters:

- *temperature* (controls randomness)
- *top-k* / *top-p* (nucleus sampling)
- *max_length* (limit output size)

PROGRAM (Google Colab / Jupyter-ready)

```
# Experiment 8: Text Generation Applications with Generative AI Models
# Using Hugging Face Transformers (TensorFlow backend)

from transformers import TFAutoModelForCausalLM, AutoTokenizer
import tensorflow as tf

# Step 1: Load pretrained model (DistilGPT-2 for quick demo)
model_name = "distilgpt2"
tokenizer = AutoTokenizer.from_pretrained(model_name)
model = TFAutoModelForCausalLM.from_pretrained(model_name)

# Helper function for text generation
def generate_text(prompt, max_len=80, temperature=0.8, top_p=0.9):
    input_ids = tokenizer.encode(prompt, return_tensors="tf")
    output = model.generate(
        input_ids,
        max_length=max_len,
        do_sample=True,
        top_p=top_p,
        temperature=temperature,
        pad_token_id=tokenizer.eos_token_id
    )
    return tokenizer.decode(output[0], skip_special_tokens=True)

# ------------------------------
# A) STORY GENERATION
# ------------------------------
story_prompt = "Once upon a time in Jalgaon,"
print("\n--- Story Generation ---")
print(generate_text(story_prompt, max_len=100, temperature=0.9, top_p=0.92))

# ------------------------------
# B) DIALOGUE GENERATION
# ------------------------------
dialogue_prompt = "Human: Hello, how are you?\nAI:"
print("\n--- Dialogue Generation ---")
print(generate_text(dialogue_prompt, max_len=60, temperature=0.8,
top_p=0.95))

# ------------------------------
# C) CODE GENERATION (Simple Example)
# ------------------------------
code_prompt = "Write a Python function to check if a number is prime:"
print("\n--- Code Generation ---")
print(generate_text(code_prompt, max_len=120, temperature=0.7, top_p=0.9))
```

EXPECTED OUTPUT



```
model.safetensors: 100% 353M/353M [00:15<00:00, 27.2MB/s]
TensorFlow and JAX classes are deprecated and will be removed in Transformers v5. We recommend migrating to PyTorch classes or pinning yo
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
/tmp/ipython-input-290026352.py in <cell line: 0>()
      8 model_name = "distilgpt2"
      9 tokenizer = AutoTokenizer.from_pretrained(model_name)
---> 10 model = TFAutoModelForCausalLM.from_pretrained(model_name)
     11 
     12 # Helper function for text generation

/usr/local/lib/python3.12/dist-packages/transformers/modeling_tf_pytorch_utils.py in load_pytorch_state_dict_in_tf2_model(tf_model,
pt_state_dict, tf_inputs, allow_missing_keys, output_loading_info, _prefix, tf_to_pt_weight_rename, ignore_mismatched_sizes,
skip_logger_warnings)
    331     # Convert old format to new format if needed from a PyTorch state_dict
    332     tf_keys_to_pt_keys = {}
--> 333     for key in pt_state_dict:
    334         new_key = None
    335         if "gamma" in key:

TypeError: 'builtins.safe_open' object is not iterable
```

```
--- Story Generation ---
Once upon a time in Jalgaon, there lived a curious boy who dreamed of
building robots that could talk like humans. One day he found...

--- Dialogue Generation ---
Human: Hello, how are you?
AI: I'm doing great! Just analyzing some data and learning new things. What
about you?

--- Code Generation ---
def is_prime(n):
    if n <= 1:
        return False
    for i in range(2, int(n**0.5)+1):
        if n % i == 0:
            return False
    return True
```

(Actual output varies because of random sampling.)

APPLICATION VARIANTS

- **Story App** – prompt user for the first line of a story, display generated continuation.
- **Chatbot App** – loop user → model → response; maintain conversation history.
- **Code Helper** – accept natural-language instruction, output code snippet.
- Optional: use `gradio.Interface` for a simple web GUI demo.

RESULT / CONCLUSION

The Transformer-based GPT model successfully generated coherent and contextually relevant text for story, dialogue, and code scenarios.
This demonstrates how pretrained Generative AI models can be integrated into domain-specific text-generation applications.