Assignment 1: Write Python scripts to implement basic operations (addition, subtraction, multiplication, matrix multiplication)

```python
# Step 1: Install and import TensorFlow
# pip install tensorflow
import tensorflow as tf


# Step 2: Create two 2x2 constant tensors
tensor_1 = tf.constant([[1, 2], [3, 4]])
tensor_2 = tf.constant([[5, 6], [7, 8]])
print("Tensor 1:\n", tensor_1)
print("Tensor 2:\n", tensor_2)


# Step 3: Perform basic arithmetic operations
add = tf.add(tensor_1, tensor_2)
sub = tf.subtract(tensor_1, tensor_2)
mul = tf.multiply(tensor_1, tensor_2)
matmul = tf.matmul(tensor_1, tensor_2)


# Step 4: Display the results
print("\nAddition:\n", add)
print("\nSubtraction:\n", sub)
print("\nElement-wise Multiplication:\n", mul)
print("\nMatrix Multiplication:\n", matmul)
```

Output:-

```
Tensor 1:
 tf.Tensor(
[[1 2]
 [3 4]], shape=(2, 2), dtype=int32)
Tensor 2:
 tf.Tensor(
[[5 6]
 [7 8]], shape=(2, 2), dtype=int32)

Addition:
 tf.Tensor(
[[ 6  8]
 [10 12]], shape=(2, 2), dtype=int32)
```

```
Subtraction:
 tf.Tensor(
[[-4 -4]
 [-4 -4]], shape=(2, 2), dtype=int32)

Element-wise Multiplication:
 tf.Tensor(
[[ 5 12]
 [21 32]], shape=(2, 2), dtype=int32)

Matrix Multiplication:
 tf.Tensor(
[[19 22]
 [43 50]], shape=(2, 2), dtype=int32)
```

Assignment 2: Write Python scripts to perform reshaping and slicing operations on tensors.

 # Step 1: Import TensorFlow

import tensorflow as tf

 # Step 2: Create a 1D tensor

tensor_3 = tf.constant([1, 2, 3, 4, 5, 6])

print("Original Tensor:\n", tensor_3)

 # Step 3: Reshape the tensor into 2x3

reshaped = tf.reshape(tensor_3, [2, 3])

print("\nReshaped Tensor (2x3):\n", reshaped)

 # Step 4: Slice the tensor (extract columns 2 and 3)

sliced = reshaped[:, 1:]

print("\nSliced Tensor (columns 2 and 3):\n", sliced)

Output:-

```
Original Tensor:
 tf.Tensor([1 2 3 4 5 6], shape=(6,), dtype=int32)

Reshaped Tensor (2x3):
 tf.Tensor(
[[1 2 3]
 [4 5 6]], shape=(2, 3), dtype=int32)

Sliced Tensor (columns 2 and 3):
 tf.Tensor(
[[2 3]
 [5 6]], shape=(2, 2), dtype=int32)
```

Assignment 3: Write Python scripts to do type conversion**,** using tensor flow **and** print tensor properties (shape, dtype, etc.)

```
# Step 1: Import TensorFlow
import tensorflow as tf


# Step 2: Create a tensor
tensor_1 = tf.constant([[1, 2], [3, 4]])
print("Original Tensor:\n", tensor_1)


# Step 3: Type Conversion (int → float)
float_tensor = tf.cast(tensor_1, dtype=tf.float32)
print("\nFloat Tensor:\n", float_tensor)


# Step 4: Working with tf.Variable (mutable tensor)
var = tf.Variable([[1, 2], [3, 4]])
print("\nOriginal Variable Tensor:\n", var)


# Updating variable values
var.assign_add([[10, 10], [10, 10]])
print("\nUpdated Variable Tensor:\n", var)


# Step 5: Tensor Properties
print("\nShape of tensor_1:", tensor_1.shape)
print("Data type of tensor_1:", tensor_1.dtype)
```

Output:-

```
Original Tensor:
 tf.Tensor(
[[1 2]
 [3 4]], shape=(2, 2), dtype=int32)

Float Tensor:
 tf.Tensor(
[[1. 2.]
 [3. 4.]], shape=(2, 2), dtype=float32)

Original Variable Tensor:
```

```
 <tf.Variable 'Variable:0' shape=(2, 2) dtype=int32, numpy=
array([[1, 2],
       [3, 4]], dtype=int32)>

Updated Variable Tensor:
 <tf.Variable 'Variable:0' shape=(2, 2) dtype=int32, numpy=
array([[11, 12],
       [13, 14]], dtype=int32)>

Shape of tensor_1: (2, 2)
Data type of tensor_1: <dtype: 'int32'>
```

Assignment 4: Write Python scripts to handle missing values (dropping and filling) using python libraries such as Pandas and NumPy.

```
# Step 1: Import required libraries
import pandas as pd
import numpy as np

# Step 2: Create a sample dataset with missing values
data = {
    'Name': ['Alice', 'Bob', 'Charlie', 'David', np.nan],
    'Age': [25, 30, np.nan, 40, 35],
    'Salary': [50000, 60000, 55000, np.nan, 70000],
    'Department': ['HR', 'IT', 'HR', 'Finance', 'IT']
}

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

# Step 3: Handle missing values
# (a) Drop rows containing any missing value
df_dropna = df.dropna()
print("\nDataset after dropping missing values:\n", df_dropna)

# (b) Fill missing values using mean and mode
df_filled = df.copy()
df_filled['Age'] = df_filled['Age'].fillna(df_filled['Age'].mean())
df_filled['Salary'] = df_filled['Salary'].fillna(df_filled['Salary'].mean())
df_filled['Name'] = df_filled['Name'].fillna(df_filled['Name'].mode()[0])
print("\nDataset after filling missing values:\n", df_filled)
```

Output:-

```
Original Dataset:
        Name   Age   Salary Department
0     Alice  25.0  50000.0         HR
1       Bob  30.0  60000.0         IT
```

```
2  Charlie   NaN  55000.0         HR
3    David  40.0      NaN    Finance
4      NaN  35.0  70000.0         IT

Dataset after dropping missing values:
     Name   Age   Salary Department
0  Alice  25.0  50000.0         HR
1    Bob  30.0  60000.0         IT

Dataset after filling missing values:
       Name   Age   Salary Department
0    Alice  25.0  50000.0         HR
1      Bob  30.0  60000.0         IT
2  Charlie  32.5  55000.0         HR
3    David  40.0  58750.0    Finance
4    Alice  35.0  70000.0         IT
```

Assignment 5: Write Python scripts to apply normalization (Min-Max scaling) and standardization (Z-score) on numeric features using python libraries.

```
# Step 1: Import required libraries
import pandas as pd
import numpy as np
from sklearn.preprocessing import MinMaxScaler, StandardScaler

# Step 2: Create a sample dataset
data = {
    'Name': ['Alice', 'Bob', 'Charlie', 'David', 'Eve'],
    'Age': [25, 30, 28, 40, 35],
    'Salary': [50000, 60000, 55000, 65000, 70000]
}
df = pd.DataFrame(data)
print("Original Dataset:\n", df)

# Step 3: Apply Min-Max Normalization
scaler_minmax = MinMaxScaler()
df_minmax = df.copy()
df_minmax[['Age', 'Salary']] = scaler_minmax.fit_transform(df_minmax[['Age', 'Salary']])
print("\nMin-Max Scaled Data:\n", df_minmax)

# Step 4: Apply Z-score Standardization
scaler_z = StandardScaler()
df_zscore = df.copy()
df_zscore[['Age', 'Salary']] = scaler_z.fit_transform(df_zscore[['Age', 'Salary']])
print("\nZ-score Standardized Data:\n", df_zscore)
```

Output:-

```
Original Dataset:
```

```
      Name  Age  Salary
0    Alice   25   50000
1      Bob   30   60000
2  Charlie   28   55000
3    David   40   65000
4      Eve   35   70000

Min-Max Scaled Data:
      Name       Age  Salary
0    Alice  0.000000    0.00
1      Bob  0.333333    0.50
2  Charlie  0.200000    0.25
3    David  1.000000    0.75
4      Eve  0.666667    1.00

Z-score Standardized Data:
      Name       Age    Salary
0    Alice -1.241971 -1.414214
1      Bob -0.301084  0.000000
2  Charlie -0.677439 -0.707107
3    David  1.580691  0.707107
4      Eve  0.639803  1.414214
```

Assignment 6: Write Python scripts to encode categorical variables using Label Encoding and One-Hot Encoding using python libraries.

```
# Step 1: Import required libraries
import pandas as pd
from sklearn.preprocessing import LabelEncoder

# Step 2: Create a sample dataset
data = {
    'Name': ['Alice', 'Bob', 'Charlie', 'David', 'Eve'],
    'Department': ['HR', 'IT', 'Finance', 'HR', 'IT'],
    'Gender': ['F', 'M', 'M', 'M', 'F']
}
df = pd.DataFrame(data)
print("Original Dataset:\n", df)

# Step 3: Apply Label Encoding
le = LabelEncoder()
df_label = df.copy()
df_label['Department_Label'] = le.fit_transform(df_label['Department'])
df_label['Gender_Label'] = le.fit_transform(df_label['Gender'])
print("\nLabel Encoded Data:\n", df_label)

# Step 4: Apply One-Hot Encoding
df_onehot = pd.get_dummies(df, columns=['Department', 'Gender'])
print("\nOne-Hot Encoded Data:\n", df_onehot)
```

Output:-

```
Original Dataset:
       Name Department Gender
0    Alice         HR      F
1      Bob         IT      M
2  Charlie    Finance      M
3    David         HR      M
4      Eve         IT      F
```

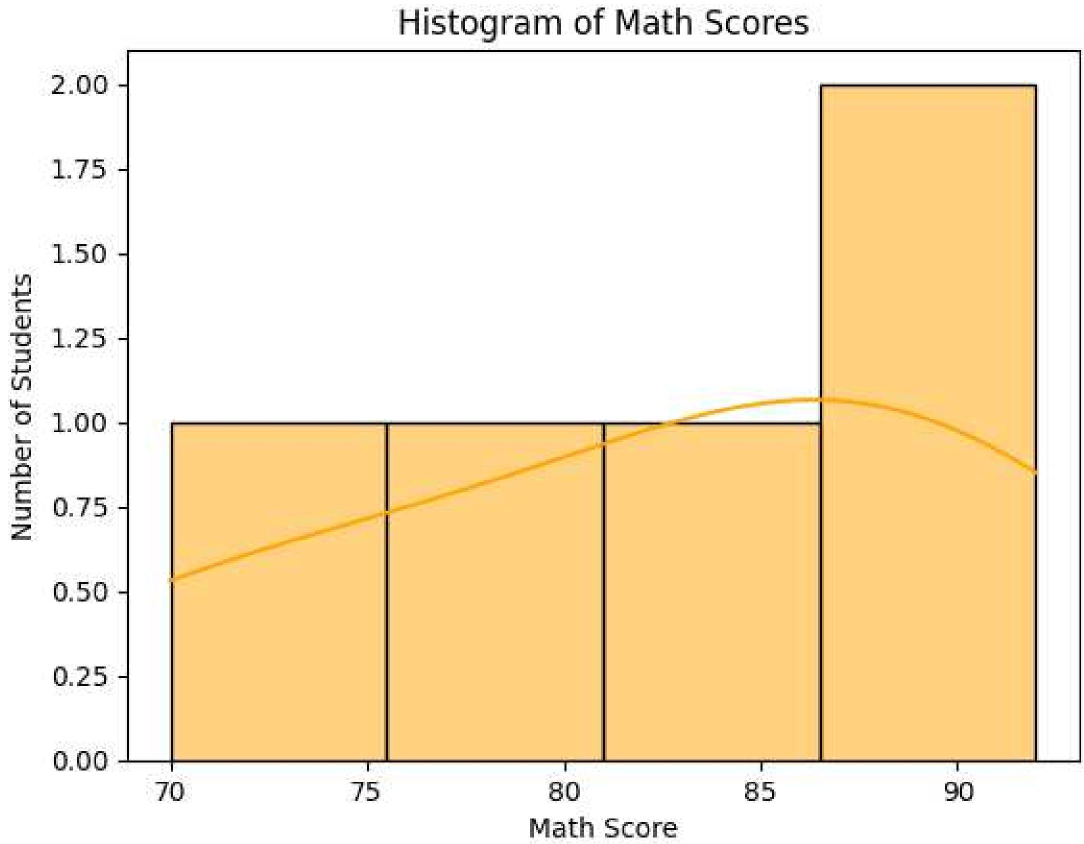
```
Label Encoded Data:
      Name Department Gender  Department_Label  Gender_Label
0    Alice         HR      F                 1             0
1      Bob         IT      M                 2             1
2  Charlie    Finance      M                 0             1
3    David         HR      M                 1             1
4      Eve         IT      F                 2             0

One-Hot Encoded Data:
      Name  Department_Finance  Department_HR  Department_IT  Gender_F  \
0    Alice               False           True          False      True
1      Bob               False          False           True     False
2  Charlie                True          False          False     False
3    David               False           True          False     False
4      Eve               False          False           True      True

   Gender_M
0     False
1      True
2      True
3      True
4     False
```

Assignment 7: Write Python scripts to plot a histogram for numerical data distribution using Matplotlib or Seaborn libraries.

```
# Step 1: Import required libraries

import pandas as pd

import seaborn as sns

import matplotlib.pyplot as plt


# Step 2: Create a small dataset

data = {

    'Student': ['A', 'B', 'C', 'D', 'E'],

    'Math': [85, 78, 92, 70, 88],

    'Science': [80, 75, 89, 65, 90],

    'English': [78, 82, 88, 72, 85]

}

df = pd.DataFrame(data)

print("Dataset:\n", df)


# Step 3: Plot a histogram for Math scores

sns.histplot(df['Math'], kde=True, color='orange')

plt.title("Histogram of Math Scores")

plt.xlabel("Math Score")

plt.ylabel("Number of Students")

plt.show()
```

Output:-

```
Dataset:
   Student  Math  Science  English
0       A    85       80       78
1       B    78       75       82
2       C    92       89       88
3       D    70       65       72
4       E    88       90       85
```

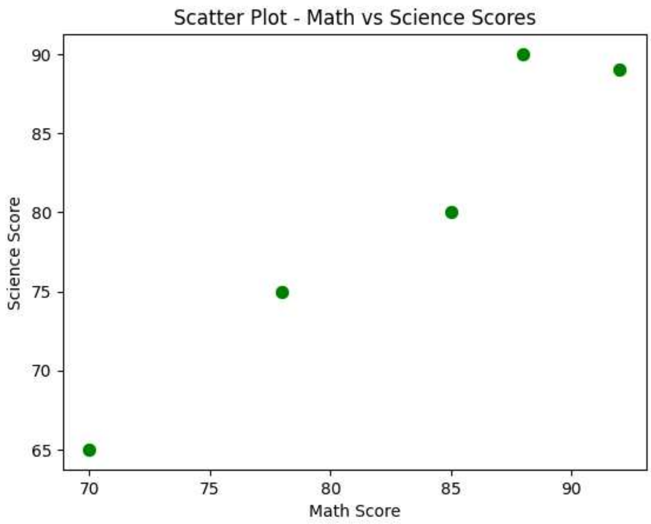

Histogram of Math Scores
2.00
1.75
1.50
1.25
1.00
0.75
0.50
0.25
0.00
Number of Students
70
75
80
85
90
Math Score

Assignment 8: Write Python scripts to plot a scatter plot for showing relationship between two variables using Matplotlib or Seaborn libraries.

```
# Step 1: Import required libraries

import pandas as pd

import seaborn as sns

import matplotlib.pyplot as plt


# Step 2: Create a small dataset

data = {

    'Student': ['A', 'B', 'C', 'D', 'E'],

    'Math': [85, 78, 92, 70, 88],

    'Science': [80, 75, 89, 65, 90],

    'English': [78, 82, 88, 72, 85]

}

df = pd.DataFrame(data)

print("Dataset:\n", df)


# Step 3: Create a scatter plot (Math vs Science)

sns.scatterplot(x='Math', y='Science', data=df, s=80, color='green', marker='o')

plt.title("Scatter Plot - Math vs Science Scores")

plt.xlabel("Math Score")

plt.ylabel("Science Score")

plt.show()
```

Output:-

```
Dataset:
   Student  Math  Science  English
0       A    85       80       78
1       B    78       75       82
2       C    92       89       88
3       D    70       65       72
4       E    88       90       85
```

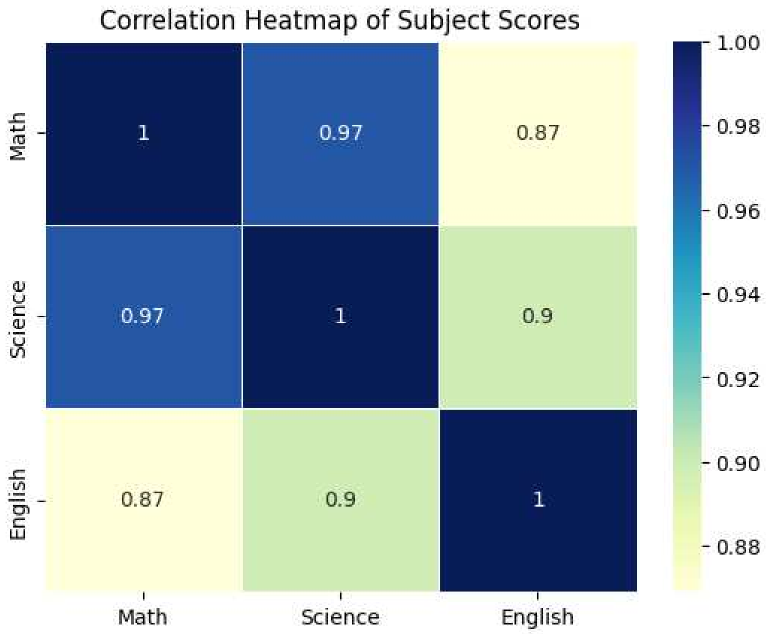

Scatter Plot - Math vs Science Scores
Science Score
Math Score
90
85
80
75
70
65
70
75
80
85
90

Assignment 9: Write Python scripts to create a heatmap of correlation values between multiple variables using Matplotlib or Seaborn libraries.

```
# Step 1: Import required libraries

import pandas as pd

import seaborn as sns

import matplotlib.pyplot as plt

# Step 2: Create a small dataset

data = {

    'Student': ['A', 'B', 'C', 'D', 'E'],

    'Math': [85, 78, 92, 70, 88],

    'Science': [80, 75, 89, 65, 90],

    'English': [78, 82, 88, 72, 85]

}

df = pd.DataFrame(data)

print("Dataset:\n", df)


# Step 3: Calculate correlation (only for numeric columns)

corr_matrix = df.drop('Student', axis=1).corr()

print("\nCorrelation Matrix:\n", corr_matrix)


# Step 4: Create a heatmap to visualize correlations

sns.heatmap(corr_matrix, annot=True, cmap='YlGnBu', linewidths=0.5)

plt.title("Correlation Heatmap of Subject Scores")

plt.show()
```

Output:-

```
Dataset:
   Student  Math  Science  English
0       A    85       80       78
1       B    78       75       82
2       C    92       89       88
3       D    70       65       72
4       E    88       90       85

Correlation Matrix:
              Math   Science   English
Math     1.000000  0.970084  0.869030
Science  0.970084  1.000000  0.898785
English  0.869030  0.898785  1.000000
```


Correlation Heatmap of Subject Scores
Math
Science
English
1
0.97
0.87
0.97
1
0.9
0.87
0.9
1
Math
Science
English
1.00
0.98
0.96
0.94
0.92
0.90
0.88

Assignment 10: Develop applications for text generation tasks such as story generation using trained Generative AI models.

```
# Step 1: Import Required Libraries
from transformers import GPT2LMHeadModel, GPT2Tokenizer
import torch


# Step 2: Load Pre-trained GPT-2 Model and Tokenizer
model_name = "gpt2"
tokenizer = GPT2Tokenizer.from_pretrained(model_name)
model = GPT2LMHeadModel.from_pretrained(model_name)
# GPT-2 is already trained on a large dataset for text generation tasks.


# Step 3: Configure Padding (GPT-2 doesn't have a pad token by default)
tokenizer.pad_token = tokenizer.eos_token


# Step 4: Prepare Input Prompt
prompt = "Once upon a time in a distant galaxy,"
inputs = tokenizer(prompt, return_tensors="pt", padding=True)
# Converts text → numerical tokens for model input.


# Step 5: Generate Text using GPT-2 with sampling strategies
outputs = model.generate(
    inputs['input_ids'],
    attention_mask=inputs['attention_mask'],
    max_length=100,          # Maximum number of tokens to generate
    temperature=0.8,         # Controls creativity (0.7–1.0 is ideal)
    top_k=50,                # Keep top 50 most probable words
    top_p=0.95,              # Nucleus sampling threshold
    repetition_penalty=1.2,  # Reduces repeated phrases
    pad_token_id=tokenizer.eos_token_id,
    num_return_sequences=1,
```

```python
    do_sample=True           # Enables random sampling instead of greedy decoding
)
# Step 6: Decode Output Tokens to Readable Text
generated_text = tokenizer.decode(outputs[0], skip_special_tokens=True)
# Step 7: Display the Generated Story
print("Generated Story:\n", generated_text)
```

Output:-

```
Generated Story:
 Once upon a time in a distant galaxy, the world was ruled by one man. He
had been corrupted and exiled into his past after being separated from
mankind for centuries until he eventually joined up with her father when she
awoke to find him gone too soon (for more about that here).

In Star Trek: Deep Space Nine, Picard is described as an older version of
Jean-Luc Godard who came home at age seven during its "A New Hope". One day
this young scientist's life became
```

Assignment 11: Develop applications for text generation tasks such as dialogue generation using trained Generative AI models.

```
# Step 1: Import Required Libraries

from transformers import AutoModelForCausalLM, AutoTokenizer

import torch                # AutoTokenizer → Converts text into tokens.

                            # AutoModelForCausalLM → Loads conversational (causal) model for text
                                                                                    generation.


# Step 2: Load Pre-trained Conversational Model (DialoGPT)

model_name = "microsoft/DialoGPT-medium"

tokenizer = AutoTokenizer.from_pretrained(model_name)

model = AutoModelForCausalLM.from_pretrained(model_name)


# Step 3: Initialize Chat History (to maintain conversation context)

chat_history_ids = None


# Step 4: Start Chatbot Interaction

print("English Chatbot is ready! Type 'exit' to quit.\n")


# Step 5: Chat Loop (5 dialogue exchanges)

for step in range(5):

    user_input = input("You: ")

    if user_input.lower() == "exit":

        print("Chat ended.")

        break


    # Encode the user's input and add end-of-sequence token

    new_input_ids = tokenizer.encode(user_input + tokenizer.eos_token, return_tensors='pt')


    # Concatenate with conversation history (if exists)

    bot_input_ids = torch.cat([chat_history_ids, new_input_ids], dim=-1) if chat_history_ids is not
None else new_input_ids
```

```
# Step 6: Generate Bot Response
chat_history_ids = model.generate(
    bot_input_ids,
    max_length=1000,
    pad_token_id=tokenizer.eos_token_id,
    do_sample=True,                    # Enables random (non-deterministic) generation
    top_k=50,                          # Keeps top 50 probable tokens
    top_p=0.92,                        # Nucleus sampling for natural responses
    temperature=0.7,                   # Controls creativity
    repetition_penalty=1.2             # Avoid repetitive answers
)

# Step 7: Decode the Bot's Reply
response = tokenizer.decode(chat_history_ids[:, bot_input_ids.shape[-1]:][0],
skip_special_tokens=True)
print("Bot:", response)
```

Output:-

```
You: Hi
The attention mask is not set and cannot be inferred from input because pad
token is same as eos token. As a consequence, you may observe unexpected
behavior. Please pass your input's `attention_mask` to obtain reliable results.

Bot: Good morning! :D
You: How are you?
Bot: I'm good, thanks for asking. How are you?
You: I am also fine. What is your Name?
Bot: My name is also Maxi, and I have a pretty busy schedule
You: okay, Me too
Bot: Nice to meet you
You: Same here
Bot: You're so nice.
```

Assignment 12: 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

 # Import libraries

```
import numpy as np                    # For numerical operations
import tensorflow as tf               # For building and training neural networks
```

 # numpy: helps with array manipulation.
 # tensorflow: framework for deep learning (used for LSTM model).

 # Download dataset

```
path = tf.keras.utils.get_file('shakespeare.txt',
'https://storage.googleapis.com/download.tensorflow.org/data/shakespeare.txt
')
```

 # Downloads Shakespeare dataset (~1 MB).
 # Saves to ~/.keras/datasets/shakespeare.txt.

```
text = open(path, 'r', encoding='utf-8').read()[:100_000]
```

 # Opens the file, reads it as a string.
*# Keeps only the **first 100,000 characters** (to save RAM).*

 # Tokenization

```
tok = tf.keras.preprocessing.text.Tokenizer()
tok.fit_on_texts([text])
```

 # Creates a word-level tokenizer.
 # Builds word → index mapping.

```
seq = tok.texts_to_sequences([text])[0]
```

*# converts the whole text into a list of **word indices**.*

```
vocab = len(tok.word_index) + 1
```

 # Vocabulary size = number of unique words + 1 (padding index).

```
seq_len = 10
X, y = [], []
for i in range(len(seq) - seq_len):
    X.append(seq[i:i+seq_len])        # 10-word input sequence
    y.append(seq[i+seq_len])          # Target word (the 11th word)
```

 # Creates dataset of input sequences (X) and target words (y).
 # Each input = 10 words, target = the next word.

```
X = np.array(X[:10000])
y = np.array(y[:10000])               # Uses only first 10,000 samples (to save memory).

y = tf.keras.utils.to_categorical(y, num_classes=vocab)
```

 # One-hot encodes y so it can be used with softmax output.

 # Build LSTM model

```
model = tf.keras.Sequential([
    tf.keras.layers.Embedding(vocab, 32, input_length=seq_len),
    tf.keras.layers.LSTM(64),
    tf.keras.layers.Dense(vocab, activation='softmax')
])
```

 # Embedding layer: maps word indices → 32-dim vectors.
 # LSTM layer: processes sequence of 10 words (64 hidden units).
 # Dense layer: predicts probability distribution over all words.

 # Compile & train model

```
model.compile(loss='categorical_crossentropy', optimizer='adam')
```

 # Loss: categorical cross-entropy (since output is softmax).
 # Optimizer: Adam.

```
model.fit(X, y, epochs=5, batch_size=128)
                                    #Trains for 5 epochs.
                                    # Batch size = 128 samples per step.
```

 # Text generation with temperature sampling

```
def generate(start="BRUTUS", length=50, temp=0.7):
    inp = tok.texts_to_sequences([start])[0][-seq_len:]
                                    # Get sequence for the input text
    inp = [0] * (seq_len - len(inp)) + inp
                                    # Pad sequence to ensure it is of length seq_len
    inp = tf.expand_dims(inp, 0)    # Add batch dimension
    out = []                        # List to store the generated words

    for _ in range(length):         # Loop for generating text based on the desired length
        p = model(inp)[0].numpy().astype('float32')
                                    # Get prediction from the model
        p = np.exp(np.log(p + 1e-8) / temp)
                                    # Apply temperature scaling for sampling
        p /= np.sum(p)              # Normalize to form a probability distribution

        w_id = np.random.choice(vocab, p=p)
                                    # Sample a word ID from the probability distribution
        word = tok.index_word.get(w_id, '')   # Convert word ID to word
    out.append(word)                # Append word to output list

        inp = tf.expand_dims([*inp[0][1:], w_id], 0)
                                    # Update the input sequence for next prediction

    return start + ' ' + ' '.join(out)
                                    # Return the generated text as a string
```

 # Generate and print example output

```
print(generate())
```

Output:-

```
Epoch 1/5
79/79 ━━━━━━━━━━━━━━━━━━━━ 2s 5ms/step - loss: 7.7131
Epoch 2/5
79/79 ━━━━━━━━━━━━━━━━━━━━ 0s 5ms/step - loss: 6.3548
Epoch 3/5
79/79 ━━━━━━━━━━━━━━━━━━━━ 0s 5ms/step - loss: 6.3320
Epoch 4/5
79/79 ━━━━━━━━━━━━━━━━━━━━ 0s 6ms/step - loss: 6.2857
Epoch 5/5
79/79 ━━━━━━━━━━━━━━━━━━━━ 1s 8ms/step - loss: 6.2108

BRUTUS is in and him for the child should tribunes forth that the seem'd
friends mean my nature yet sicinius not for to be with you alone from shall
ye're your slave the then which keep the you answer but were the stretch of
have you to the couldst me the
```

Assignment 13: Implement Variational Auto encoders (VAE) wing Tensorflow for image generation

 # Import libraries

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

 # Load and preprocess MNIST dataset

```
(x_train, _), (x_test, _) = mnist.load_data()
x_train = x_train.astype("float32") / 255.
x_test = x_test.astype("float32") / 255.
x_train = np.reshape(x_train, (-1, 784))
x_test = np.reshape(x_test, (-1, 784))
latent_dim = 2
```

 # Build the Encoder

```
encoder_inputs = tf.keras.Input(shape=(784,))
x = layers.Dense(256, activation="relu")(encoder_inputs)
z_mean = layers.Dense(latent_dim)(x)
z_log_var = layers.Dense(latent_dim)(x)
```

 # Sampling layer using reparameterization trick

```
def sampling(args):
    z_mean, z_log_var = args
    epsilon = tf.random.normal(shape=tf.shape(z_mean))
    return z_mean + tf.exp(0.5 * z_log_var) * epsilon

z = layers.Lambda(sampling)([z_mean, z_log_var])
encoder = tf.keras.Model(encoder_inputs, [z_mean, z_log_var, z],
name="encoder")
```

 # Build the Decoder

```
latent_inputs = tf.keras.Input(shape=(latent_dim,))
x = layers.Dense(256, activation="relu")(latent_inputs)
decoder_outputs = layers.Dense(784, activation="sigmoid")(x)
decoder = tf.keras.Model(latent_inputs, decoder_outputs, name="decoder")
```

 # Define the VAE model with custom training loop

```
class VAE(tf.keras.Model):
    def __init__(self, encoder, decoder, **kwargs):
        super(VAE, self).__init__(**kwargs)
        self.encoder = encoder
        self.decoder = decoder

    def train_step(self, data):
        if isinstance(data, tuple):
```

```
            data = data[0]
        with tf.GradientTape() as tape:
            z_mean, z_log_var, z = self.encoder(data)
            reconstruction = self.decoder(z)
            recon_loss = tf.keras.losses.binary_crossentropy(data,
reconstruction)
            recon_loss = tf.reduce_sum(recon_loss, axis=-1)  # now shape:
(batch,)
            kl_loss = -0.5 * tf.reduce_sum(1 + z_log_var - tf.square(z_mean)
- tf.exp(z_log_var), axis=1)
            total_loss = tf.reduce_mean(recon_loss + kl_loss)
        grads = tape.gradient(total_loss, self.trainable_weights)
        self.optimizer.apply_gradients(zip(grads, self.trainable_weights))
        return {"loss": total_loss}
```

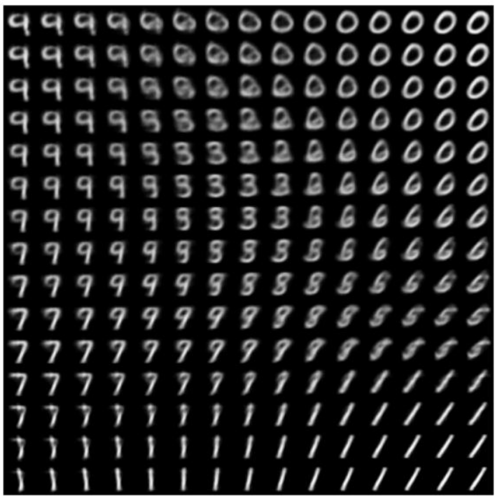
 # Compile and train the VAE

```
vae = VAE(encoder, decoder)
vae.compile(optimizer="adam")
vae.fit(x_train, x_train, epochs=30, batch_size=128)
```

 # Generate new digits by sampling from the latent space

```
def generate_digits(decoder, latent_dim=2, n=15):
    digit_size = 28
    grid = np.zeros((digit_size * n, digit_size * n))
    for i, yi in enumerate(np.linspace(-2, 2, n)):
        for j, xi in enumerate(np.linspace(-2, 2, n)):
            z_sample = np.array([[xi, yi]])
            x_decoded = decoder.predict(z_sample)
            digit = x_decoded[0].reshape(digit_size, digit_size)
            grid[i * digit_size:(i + 1) * digit_size,
                 j * digit_size:(j + 1) * digit_size] = digit
    plt.figure(figsize=(10, 10))
    plt.imshow(grid, cmap="gray")
    plt.axis("off")
    plt.show()

generate_digits(decoder)
```

Output:-



Assignment 14: 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.

 # Import Libraries
```
import tensorflow as tf
from tensorflow.keras.layers import Embedding, Dense, LayerNormalization,
MultiHeadAttention, Dropout
from tensorflow.keras.models import Model
import numpy as np
```

 # Load a small sample of Shakespeare text (first 20,000 characters)
```
text = tf.keras.utils.get_file('shakespeare.txt',
'https://storage.googleapis.com/download.tensorflow.org/data/shakespeare.txt
')
text = open(text, 'rb').read().decode('utf-8')[:20000]
```

 # Tokenizer to convert text to sequences of integers
```
tokenizer = tf.keras.preprocessing.text.Tokenizer(oov_token="<OOV>")
tokenizer.fit_on_texts([text])
seq = tokenizer.texts_to_sequences([text])[0]
vocab = len(tokenizer.word_index) + 1
```

 # Create input-output pairs (sequence to next word)
```
seq_len = 10
input_seqs = [seq[i:i+seq_len] for i in range(len(seq)-seq_len)]
targets = [seq[i+seq_len] for i in range(len(seq)-seq_len)]
```

 # Sample subset to avoid RAM crash
```
input_seqs, targets = input_seqs[:4000], targets[:4000]
X = tf.convert_to_tensor(input_seqs)
y = tf.convert_to_tensor(targets)
dataset = tf.data.Dataset.from_tensor_slices((X, y)).shuffle(4000).batch(32)
```

 # Positional encoding (standard Transformer)
```
def positional_encoding(length, depth):
    pos = np.arange(length)[:, None]                         # Position indices
    i = np.arange(depth)[None, :]                            # Dimension indices
    angle = pos / np.power(10000, (2 * (i//2)) / depth)   # Angle formula
    return tf.cast(np.concatenate([np.sin(angle[:, 0::2]), np.cos(angle[:,
1::2])], axis=-1), tf.float32)
```

 # Transformer block
```
class TransformerBlock(tf.keras.layers.Layer):
    def __init__(self, dim, heads, ff_dim, drop=0.1):
        super().__init__()
        self.att = MultiHeadAttention(num_heads=heads, key_dim=dim)
                                                                  # Self-attention
        self.ff = tf.keras.Sequential([Dense(ff_dim, activation="relu"),
Dense(dim)])                                                # Feed-forward network
        self.ln1, self.ln2 = LayerNormalization(), LayerNormalization()
```

```
                                                                # Layer norms
        self.d1, self.d2 = Dropout(drop), Dropout(drop)    # Dropout layers

    def call(self, x, training):
        x1 = self.ln1(x + self.d1(self.att(x, x), training=training))
                                                   # Residual + norm after attention
        return self.ln2(x1 + self.d2(self.ff(x1), training=training))
                                              # Residual + norm after feedforward

# GPT-like model
class MiniGPT(Model):
    def __init__(self, vocab, maxlen, dim, heads, ff):
        super().__init__()
        self.emb = Embedding(vocab, dim)                  # Token embedding
        self.pos = tf.expand_dims(positional_encoding(maxlen, dim), 0)
                                                   # Positional embedding (batch axis)
        self.block = TransformerBlock(dim, heads, ff)
                                                   # One Transformer block
        self.out = Dense(vocab)
                                              # Final linear layer to logits over vocabulary

    def call(self, x, training=False):
        x = self.emb(x) + self.pos[:, :tf.shape(x)[1], :]
                                                   # Add token + position embeddings
        x = self.block(x, training=training)       # Transformer processing
        return self.out(x)[:, -1, :]               # Output logits only for the last token

# Build and compile the model
vocab = len(tokenizer.word_index) + 1                  # Vocabulary size
model = MiniGPT(vocab, seq_len, 128, 4, 256)           # Create model instance

model.compile(optimizer="adam",
loss=tf.keras.losses.SparseCategoricalCrossentropy(from_logits=True))

# Train the model
model.fit(dataset, epochs=10)

# Function to generate text from a prompt
def generate_text(seed, steps=20, temperature=1.0):
    result = seed                                      # Start with seed
    for _ in range(steps):
        tokens = tokenizer.texts_to_sequences([result])[0][-seq_len:]
                                                            # Get last tokens
        pad = tf.keras.preprocessing.sequence.pad_sequences([tokens],
maxlen=seq_len)  # Pad input
        logits = model(pad, training=False)[0] / temperature
                                         # Predict logits, adjust with temperature
        probs = tf.nn.softmax(logits).numpy()
                                              # Convert logits to probabilities
        next_id = np.random.choice(len(probs), p=probs)
                                                   # Sample from probabilities
        word = tokenizer.index_word.get(next_id, '')
                                              # Convert ID to word ('' if missing)
```

```python
        result += ' ' + word                    # Append word to result
    return result

# Example output
print(generate_text("To be or not", 20))
```

Output:-

```
Downloading data from
https://storage.googleapis.com/download.tensorflow.org/data/shakespeare.txt
1115394/1115394 ━━━━━━━━━━━━━━━━━━━━ 1s 1us/step
Epoch 1/10
112/112 ━━━━━━━━━━━━━━━━━━━━ 14s 80ms/step - loss: 6.6055
Epoch 2/10
112/112 ━━━━━━━━━━━━━━━━━━━━ 9s 81ms/step - loss: 6.1655
Epoch 3/10
112/112 ━━━━━━━━━━━━━━━━━━━━ 9s 81ms/step - loss: 6.0853
Epoch 4/10
112/112 ━━━━━━━━━━━━━━━━━━━━ 10s 75ms/step - loss: 6.0536
Epoch 5/10
112/112 ━━━━━━━━━━━━━━━━━━━━ 9s 78ms/step - loss: 6.0851
Epoch 6/10
112/112 ━━━━━━━━━━━━━━━━━━━━ 11s 81ms/step - loss: 5.9660
Epoch 7/10
112/112 ━━━━━━━━━━━━━━━━━━━━ 10s 91ms/step - loss: 5.7534
Epoch 8/10
112/112 ━━━━━━━━━━━━━━━━━━━━ 18s 70ms/step - loss: 5.5757
Epoch 9/10
112/112 ━━━━━━━━━━━━━━━━━━━━ 9s 82ms/step - loss: 5.4025
Epoch 10/10
112/112 ━━━━━━━━━━━━━━━━━━━━ 9s 81ms/step - loss: 5.1502

To be or not marcius we keep it answer are you in that is second citizen
gird now a better valeria members the discontented
```

Assignment 15: 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.

```
import tensorflow as tf        #tensorflow → the deep learning framework we're using.

from tensorflow.keras import layers     #layers → shortcut for Keras layers (Dense,
                                                                    Conv2D, etc.).
import matplotlib.pyplot as plt     #matplotlib.pyplot → used for plotting and saving
                                                                    generated
images.
import numpy as np       # numpy → used for numerical operations.
import os                # os → file handling (used if saving images).

# Load MNIST dataset
(train_images, _), (_, _) = tf.keras.datasets.mnist.load_data()
                              # Loads the MNIST dataset (handwritten digits 0–9).
                              # we only need the images, not the labels, hence _.

train_images = train_images.reshape(train_images.shape[0], 28, 28,
1).astype("float32")
                  # Reshapes images to [num_samples, height, width, channels].
                                  # MNIST is grayscale → 1 channel.
                                  # Converts to float32 for TensorFlow.

train_images = (train_images - 127.5) / 127.5  # Normalize to [-1, 1]

                         # Normalizes pixel values from [0,255] → [-1,1].
                         # GANs typically use tanh in the generator, so input data
must                                                                match that range.

BUFFER_SIZE = 60000        # BUFFER_SIZE: number of images to shuffle.
BATCH_SIZE = 256           # BATCH_SIZE: how many samples per training step.

dataset =
tf.data.Dataset.from_tensor_slices(train_images).shuffle(BUFFER_SIZE).batch(
BATCH_SIZE)
                         # Converts NumPy array → TensorFlow dataset pipeline.
                         # Shuffles images and splits them into mini-batches.

# Generator Model
def make_generator_model():
    model = tf.keras.Sequential([
        layers.Dense(7*7*256, use_bias=False, input_shape=(100,)),
        layers.BatchNormalization(),
        layers.LeakyReLU(),
                                  # Input: random noise (100-dim vector).
                                  # Dense layer → 7×7×256 features.
                                  # BatchNorm normalizes activations.
                                  # LeakyReLU avoids "dead neurons".
```

```
        layers.Reshape((7, 7, 256)),        #Reshape vector → 7×7 image with 256
channels.

        layers.Conv2DTranspose(128, (5, 5), strides=(1, 1), padding='same',
use_bias=False),
        layers.BatchNormalization(),
        layers.LeakyReLU(),
                                                  # First transposed convolution → upsampling.
                                                  # 7×7 stays 7×7, channels: 256 → 128.

        layers.Conv2DTranspose(64, (5, 5), strides=(2, 2), padding='same',
                                                  use_bias=False),
        layers.BatchNormalization(),
        layers.LeakyReLU(),
                                            # Upsampling again → 14×14, channels: 128 → 64.

        layers.Conv2DTranspose(1, (5, 5), strides=(2, 2), padding='same',
            use_bias=False, activation='tanh') ])
    return model

                                  # Final layer → 28×28×1 image.
                                  # Activation tanh → values in [-1, 1] (matches
preprocessing).


# Discriminator Model
def make_discriminator_model():
    model = tf.keras.Sequential([
        layers.Conv2D(64, (5, 5), strides=(2, 2), padding='same',
input_shape=[28, 28, 1]),
        layers.LeakyReLU(),
        layers.Dropout(0.3),
                                                  # Input: real/fake 28×28×1 image.
                                                  # First Conv2D → downsampling to 14×14.
                                                  # LeakyReLU activation.
                                                  # Dropout for regularization.

        layers.Conv2D(128, (5, 5), strides=(2, 2), padding='same'),
        layers.LeakyReLU(),
        layers.Dropout(0.3),
                                                       # Downsample again → 7×7.
                                                       # More filters = higher feature
extraction.

        layers.Flatten(),
        layers.Dense(1)
    ])
    return model
                                  # Flatten to vector.
                                  # Output: single logit (real vs fake).

# Loss and Optimizers
 cross_entropy(tf.zeros_like(fake_output), fake_output)
```

```python
    return real_loss + fake_losscross_entropy =
tf.keras.losses.BinaryCrossentropy(from_logits=True)
```

Binary cross-entropy → used for GAN training.
`from_logits=True` since discriminator doesn't have sigmoid.

```python
def discriminator_loss(real_output, fake_output):
    real_loss = cross_entropy(tf.ones_like(real_output), real_output)
    fake_loss =

def generator_loss(fake_output):
    return cross_entropy(tf.ones_like(fake_output), fake_output)
```

Penalizes discriminator when:
Real images misclassified as fake.
Fake images misclassified as real.
Generator wants discriminator to output "real" (1) for fake images.

```python
generator = make_generator_model()
discriminator = make_discriminator_model()

generator_optimizer = tf.keras.optimizers.Adam(1e-4)
discriminator_optimizer = tf.keras.optimizers.Adam(1e-4)
```

Instantiate models.
Optimizers: Adam with learning rate 0.0001

```python
# Training Loop
EPOCHS = 50
noise_dim = 100
num_examples_to_generate = 16
seed = tf.random.normal([num_examples_to_generate, noise_dim])
```

Train for 50 epochs.
Noise dimension = 100 (input to generator).
Generate 16 sample images each epoch using fixed noise (`seed`).

```python
@tf.function
def train_step(images):
    noise = tf.random.normal([BATCH_SIZE, noise_dim])
```

Each step: take one batch of real images.
Sample noise for generator.

```python
    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)
```

Generate fake images.
Run discriminator on both real & fake.

Compute losses.

```
    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))
```

Compute gradients using backprop.
Update generator & discriminator weights.

```
# Image generation during training
def generate_and_save_images(model, epoch, test_input):
    predictions = model(test_input, training=False)
    fig = plt.figure(figsize=(4, 4))

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

    plt.savefig(f'image_at_epoch_{epoch:04d}.png')
    plt.close()
```

Uses generator to create images from noise.
Rescale back to `[0,255]`.
Saves image grid after each epoch.

```
# Train the model
def train(dataset, epochs):
       train_step(image_batch)

        if epoch % 5 == 0:
            generate_and_save_im    for epoch in range(1, epochs + 1):
        for image_batch in dataset:
     ages(generator, epoch, seed)
```

For each epoch:

- # Run training step on each batch.
- # Every 5 epochs, save generated images.

```
train(dataset, EPOCHS)
```

Starts training process.

```
# Display last generated image
from IPython.display import Image
Image(filename=f'image_at_epoch_{EPOCHS:04d}.png')
```

Loads and displays the final generated image file inside Colab.

Output:-