**Practical 1:**

**Aim: Intro to TensorFlow**

import tensorflow as tf

# Creating tensors with different shapes and data types

t1 = tf.constant([1, 2, 3], dtype=tf.int32) # 1D tensor (vector)

t2 = tf.constant([[1.5, 2.5], [3.5, 4.5]], dtype=tf.float32) # 2D tensor (matrix)

# Performing basic tensor operations

add\_result = tf.add(t1, 2) # Adding a scalar

sub\_result = tf.subtract(t1, 1) # Subtracting a scalar

mul\_result = tf.multiply(t1, 2) # Element-wise multiplication

div\_result = tf.divide(t1, 2) # Element-wise division

# Reshaping, slicing, and indexing tensors

t3 = tf.reshape(t2, [4, 1]) # Reshape to a column vector

slice\_result = t2[:, 1] # Extracting second column

index\_result = t1[0] # Extracting first element

# Performing matrix multiplication

mat1 = tf.constant([[1, 2], [3, 4]], dtype=tf.float32)

mat2 = tf.constant([[5, 6], [7, 8]], dtype=tf.float32)

mat\_mul\_result = tf.matmul(mat1, mat2) # Matrix multiplication

# Finding eigenvalues and eigenvectors

eigenvalues, eigenvectors = tf.linalg.eig(mat1)

# Printing results

print("Addition:", add\_result.numpy())

print("Subtraction:", sub\_result.numpy())

print("Multiplication:", mul\_result.numpy())

print("Division:", div\_result.numpy())

print("Reshaped tensor:", t3.numpy())

print("Sliced tensor:", slice\_result.numpy())

print("Indexed element:", index\_result.numpy())

print("Matrix multiplication result:\n", mat\_mul\_result.numpy())

print("Eigenvalues:\n", eigenvalues.numpy())

print("Eigenvectors:\n", eigenvectors.numpy())

**Practical 1b:**

**Aim: Program to solve the XOR problem.**

import numpy as np

from tensorflow.keras.models import Sequential

from tensorflow.keras.layers import Input, Dense

from tensorflow.keras.optimizers import Adam

# XOR dataset

X = np.array([[0, 0], [0, 1], [1, 0], [1, 1]], dtype=np.float32)

y = np.array([[0], [1], [1], [0]], dtype=np.float32)

# Build the model

model = Sequential()

model.add(Input(shape=(2,))) # New Input layer

model.add(Dense(4, activation='relu'))

model.add(Dense(1, activation='sigmoid'))

# Compile and train

model.compile(optimizer=Adam(), loss='binary\_crossentropy', metrics=['accuracy'])

model.fit(X, y, epochs=1000, verbose=0)

# Predict

predictions = model.predict(X)

# Print results

print("XOR Predictions:")

for input\_val, pred\_val, actual in zip(X, predictions, y):

print(f"Input: {input\_val} => Prediction: {int(np.round(pred\_val[0]))}, Actual: {int(actual[0])}")

**Practical 2**

**Aim: Implement a simple linear regression model using TensorFlow's low level API (or tf. keras).**

import numpy as np

import tensorflow as tf

import matplotlib.pyplot as plt

# 1. Dataset: Square footage vs Housing prices

np.random.seed(42)

X = np.array([500, 1000, 1500, 2000, 2500, 3000], dtype=np.float32).reshape(-1, 1)

y = np.array([150000, 200000, 250000, 300000, 350000, 400000], dtype=np.float32).reshape(-1, 1)

# 2. Define a simple linear regression model

model = tf.keras.Sequential([

tf.keras.Input(shape=(1,)), # Proper input layer

tf.keras.layers.Dense(1) # Output layer

])

# 3. Compile the model

model.compile(optimizer='adam', loss='mse')

# 4. Train the model

history = model.fit(X, y, epochs=1000, verbose=0)

# 5. Plot training loss

plt.plot(history.history['loss'])

plt.title('Training Loss (MSE)')

plt.xlabel('Epochs')

plt.ylabel('Loss')

plt.grid(True)

plt.show()

# 6. Plot predictions vs actual data

plt.scatter(X, y, color='blue', label='Actual Data')

predicted\_y = model.predict(X)

plt.plot(X, predicted\_y, color='red', label='Model Prediction')

plt.title('Housing Prices vs Square Footage')

plt.xlabel('Square Footage')

plt.ylabel('Price')

plt.legend()

plt.grid(True)

plt.show()

# 7. Predict new values

new\_data = np.array([1200, 1800, 2200], dtype=np.float32).reshape(-1, 1)

predictions = model.predict(new\_data)

print("Predictions for new square footage:")

for sqft, price in zip(new\_data.flatten(), predictions.flatten()):

print(f"{sqft:.0f} sqft => ${price:,.2f}")

**Practical 3A:**

**Aim:Implementing deep neural network for performing binary classification task**

import numpy as np

import tensorflow as tf

from tensorflow.keras.models import Sequential

from tensorflow.keras.layers import Dense

from tensorflow.keras.optimizers import Adam

from sklearn.model\_selection import train\_test\_split

from sklearn.preprocessing import StandardScaler

from sklearn.datasets import make\_classification

# 1. Prepare the Data (You can replace this with your dataset)

# Let's generate a synthetic dataset for binary classification

X, y = make\_classification(n\_samples=1000, n\_features=20, n\_classes=2,

random\_state=42)

# Split the data into training and testing sets

X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, test\_size=0.2,

random\_state=42)

# Feature scaling (important for neural networks)

scaler = StandardScaler()

X\_train = scaler.fit\_transform(X\_train)

X\_test = scaler.transform(X\_test)

# 2. Build the Model

model = Sequential()

# Input layer (input shape matches the number of features in the data)

model.add(Dense(64, input\_dim=X\_train.shape[1], activation='relu'))

# Hidden layer

model.add(Dense(32, activation='relu'))

# Output layer (for binary classification, use sigmoid activation)

model.add(Dense(1, activation='sigmoid'))

# 3. Compile the Model

model.compile(optimizer=Adam(learning\_rate=0.001),

loss='binary\_crossentropy', # Binary classification loss function

metrics=['accuracy'])

# 4. Train the Model

history = model.fit(X\_train, y\_train, epochs=20, batch\_size=32,

validation\_split=0.2)

# 5. Evaluate the Model

loss, accuracy = model.evaluate(X\_test, y\_test)

print(f"Test Loss: {loss:.4f}")

print(f"Test Accuracy: {accuracy \* 100:.2f}%")

**Practical 6b**

**Aim : Applying the Autoencoder algorithms for encoding real-world data**

# Full Autoencoder

autoencoder = Model(input\_layer, decoded)

# Encoder model

encoder = Model(input\_layer, encoded)

# Step 5: Compile the Model

autoencoder.compile(optimizer=Adam(learning\_rate=0.001), loss='mse')

# Step 6: Train the Model

history = autoencoder.fit(

X\_train, X\_train,

epochs=100,

batch\_size=16,

shuffle=True,

validation\_data=(X\_test, X\_test)

)

# Step 7: Encode Real-World Data

X\_encoded = encoder.predict(X\_test)

print(f"Encoded data shape: {X\_encoded.shape}")

print("\nSample Encoded Data:\n", X\_encoded[:5])

# Step 8: Plot Loss Curves

plt.plot(history.history['loss'], label='Training Loss')

plt.plot(history.history['val\_loss'], label='Validation Loss')

plt.title('Model Loss During Training')

plt.xlabel('Epochs')

plt.ylabel('Loss')

plt.legend()

plt.show()

# Step 9: Save Encoded Data (Optional)

encoded\_df = pd.DataFrame(X\_encoded, columns=[f'encoded\_{i+1}' for i in

range(encoding\_dim)])

encoded\_df.to\_csv('encoded\_wine\_data.csv', index=False)

print("\nEncoded data saved to 'encoded\_wine\_data.csv'")

**Practical 7: Aim :Write a program for character recognition using RNN and compare it with CNN**

import tensorflow as tf

from tensorflow.keras import layers, models

from tensorflow.keras.datasets import mnist

# Load and preprocess MNIST dataset

(x\_train, y\_train), (x\_test, y\_test) = mnist.load\_data()

x\_train, x\_test = x\_train / 255.0, x\_test / 255.0

# Reshape for RNN (samples, timesteps, features)

x\_train\_rnn, x\_test\_rnn = x\_train.reshape(-1, 28, 28), x\_test.reshape(-1, 28,

28)

# Reshape for CNN (samples, height, width, channels)

x\_train\_cnn, x\_test\_cnn = x\_train.reshape(-1, 28, 28, 1), x\_test.reshape(-1,

28, 28, 1)

# RNN Model

def create\_rnn():

    model = models.Sequential([

        layers.SimpleRNN(128, activation='relu', input\_shape=(28, 28)),

        layers.Dense(10, activation='softmax')

    ])

    model.compile(optimizer='adam', loss='sparse\_categorical\_crossentropy',

metrics=['accuracy'])

    return model

# CNN Model

def create\_cnn():

    model = models.Sequential([

        layers.Conv2D(32, (3, 3), activation='relu', input\_shape=(28, 28, 1)),

        layers.MaxPooling2D((2, 2)),

        layers.Flatten(),

        layers.Dense(128, activation='relu'),

        layers.Dense(10, activation='softmax')

    ])

    model.compile(optimizer='adam', loss='sparse\_categorical\_crossentropy',

metrics=['accuracy'])

    return model

# Train and evaluate RNN

rnn\_model = create\_rnn()

rnn\_model.fit(x\_train\_rnn, y\_train, epochs=3, batch\_size=64,

validation\_data=(x\_test\_rnn, y\_test))

rnn\_loss, rnn\_acc = rnn\_model.evaluate(x\_test\_rnn, y\_test)

# Train and evaluate CNN

cnn\_model = create\_cnn()

cnn\_model.fit(x\_train\_cnn, y\_train, epochs=3, batch\_size=64,

validation\_data=(x\_test\_cnn, y\_test))

cnn\_loss, cnn\_acc = cnn\_model.evaluate(x\_test\_cnn, y\_test)

# Compare results

print(f'RNN Accuracy: {rnn\_acc:.4f}, CNN Accuracy: {cnn\_acc:.4f}')

**Practical 08**

**Aim : Write a program to develop Autoencoders using MNIST Handwritten Digits**

# Step 1: Import Libraries

import numpy as np

import matplotlib.pyplot as plt

from tensorflow.keras.datasets import mnist

from tensorflow.keras.models import Model

from tensorflow.keras.layers import Input, Dense, Flatten, Reshape

from tensorflow.keras.optimizers import Adam

# Step 2: Load and Preprocess Data

(x\_train, \_), (x\_test, \_) = mnist.load\_data()

# Normalize pixel values between 0 and 1

x\_train = x\_train.astype('float32') / 255.0

x\_test = x\_test.astype('float32') / 255.0

# Flatten the images (28x28 -> 784)

x\_train = x\_train.reshape((len(x\_train), np.prod(x\_train.shape[1:])))

x\_test = x\_test.reshape((len(x\_test), np.prod(x\_test.shape[1:])))

print(f"x\_train shape: {x\_train.shape}")

print(f"x\_test shape: {x\_test.shape}")

# Step 3: Build Autoencoder

input\_dim = x\_train.shape[1]  # 784

encoding\_dim = 32  # Size of the bottleneck

# Input layer

input\_img = Input(shape=(input\_dim,))

# Encoder

encoded = Dense(encoding\_dim, activation='relu')(input\_img)

# Decoder

decoded = Dense(input\_dim, activation='sigmoid')(encoded)

# Autoencoder Model

autoencoder = Model(input\_img, decoded)

# Encoder Model (for getting encoded data separately)

encoder = Model(input\_img, encoded)

# Step 4: Compile the Autoencoder

autoencoder.compile(optimizer=Adam(learning\_rate=0.001),

loss='binary\_crossentropy')

# Step 5: Train the Autoencoder

history = autoencoder.fit(

    x\_train, x\_train,

    epochs=50,

    batch\_size=256,

    shuffle=True,

    validation\_data=(x\_test, x\_test)

)

# Step 6: Encode and Decode Some Digits

encoded\_imgs = encoder.predict(x\_test)

decoded\_imgs = autoencoder.predict(x\_test)

print(f"Encoded images shape: {encoded\_imgs.shape}")

# Step 7: Visualize the Original and Reconstructed Images

n = 10  # Number of digits to display

plt.figure(figsize=(20, 4))

for i in range(n):

    # Display original digits

    ax = plt.subplot(2, n, i + 1)

    plt.imshow(x\_test[i].reshape(28, 28))

    plt.gray()

    ax.axis('off')

    # Display reconstructed digits

    ax = plt.subplot(2, n, i + 1 + n)

    plt.imshow(decoded\_imgs[i].reshape(28, 28))

    plt.gray()

    ax.axis('off')

plt.show()

# Step 8: Plot Training History (Loss curve)

plt.figure(figsize=(8, 4))

plt.plot(history.history['loss'], label='Training Loss')

plt.plot(history.history['val\_loss'], label='Validation Loss')

plt.title('Autoencoder Training vs Validation Loss')

plt.xlabel('Epochs')

plt.ylabel('Loss')

plt.legend()

plt.show()

**Practical 9 :Aim : Demonstrate recurrent neural network that learns to perform sequence analysis for stock price.(google stock price)**

# Predict future stock prices

future\_steps = 30

future\_input = test\_data[-time\_steps:].reshape(1, time\_steps, 1)

future\_predictions = []

for \_ in range(future\_steps):

    pred = model.predict(future\_input, verbose=0)

    future\_predictions.append(pred[0, 0])

    future\_input = np.append(future\_input[:,1:, :], pred.reshape(1, 1, 1), axis=1)

# Inverse transform all predictions once

future\_predictions = scaler.inverse\_transform(np.array(future\_predictions).reshape(-1, 1))

# Visualization

plt.figure(figsize=(14, 5))

future\_dates = pd.date\_range(df.index[-1], periods=future\_steps + 1, freq='B')[1:]

plt.plot(future\_dates, future\_predictions, label='Future Predictions', color='green')

plt.title("Future Stock Price Predictions")

plt.xlabel("Date")

plt.ylabel("Price")

plt.legend()

plt.grid(True)

plt.show()