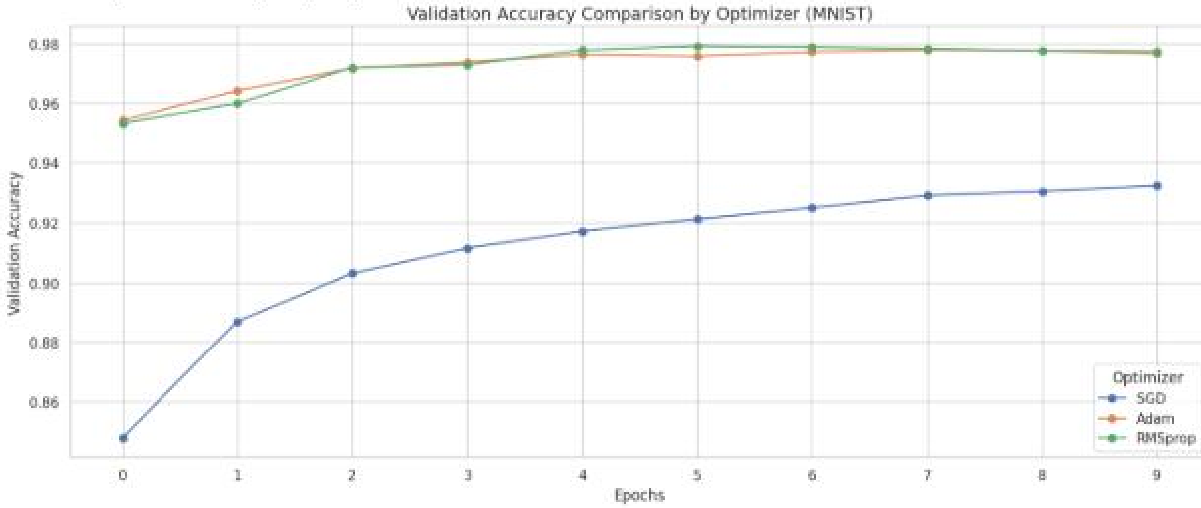
Practical 1

AIM- Implement Feed-forward Neural Network and train the network with different optimizers and compare the results.

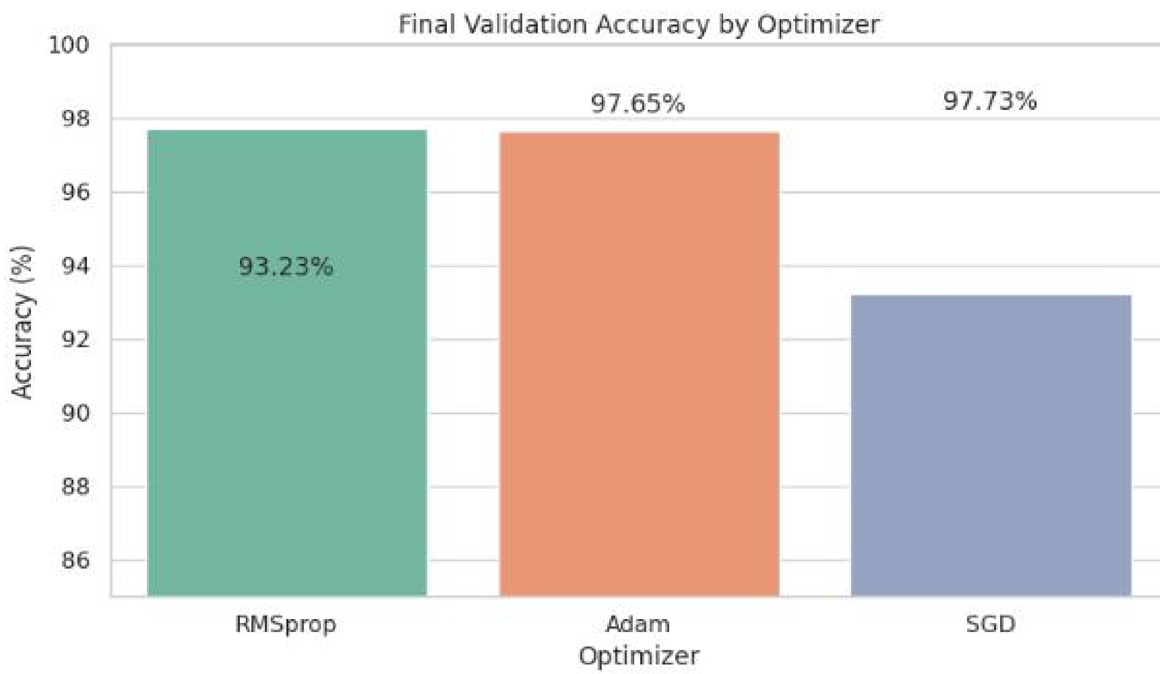
```
import matplotlib.pyplot as plt
import seaborn as sns
import pandas as pd
from keras.datasets import mnist
from keras.models import Sequential
from keras.layers import Dense, Flatten
from keras.utils import to_categorical
from keras.optimizers import SGD, Adam, RMSprop

# Load and preprocess data
(x_train, y_train), (x_test, y_test) = mnist.load_data()
x_train = x_train.astype('float32') / 255.0
x_test  = x_test.astype('float32') / 255.0
y_train = to_categorical(y_train, 10)
y_test  = to_categorical(y_test, 10)

# Build model function
def build_model(optimizer):
    model = Sequential([
        Flatten(input_shape=(28, 28)),
        Dense(128, activation='relu'),
        Dense(64, activation='relu'),
        Dense(10, activation='softmax')
    ])
    model.compile(optimizer=optimizer, loss='categorical_crossentropy', metrics=['accuracy'])
    return model

# Define optimizers
optimizers = {
    'SGD': SGD(),
    'Adam': Adam(),
    'RMSprop': RMSprop()
}

# Train models
histories = {}
final_results = []
for name, opt in optimizers.items():
    print(f"\nTraining with {name} optimizer...")
    model = build_model(opt)
    history = model.fit(x_train, y_train,
                        validation_data=(x_test, y_test),
```

```
                    epochs=10, batch_size=128, verbose=0)
    histories[name] = history
    final_acc = history.history['val_accuracy'][-1]
    final_results.append({'Optimizer': name, 'Final Val Accuracy': final_acc * 100})

# --- Plotting ---
sns.set(style='whitegrid')
plt.figure(figsize=(14, 6))

# 1. Validation accuracy over epochs
for name, history in histories.items():
    plt.plot(history.history['val_accuracy'], label=f'{name}', marker='o')

plt.title('Validation Accuracy Comparison by Optimizer (MNIST)', fontsize=14)
plt.xlabel('Epochs')
plt.ylabel('Validation Accuracy')
plt.xticks(range(10))
plt.legend(title='Optimizer')
plt.tight_layout()
plt.show()

# 2. Final accuracy table
results_df = pd.DataFrame(final_results).sort_values(by='Final Val Accuracy', ascending=False)
print("\nFinal Validation Accuracy (Sorted):")
print(results_df.to_string(index=False, float_format="%.2f"))

# Optional: Bar chart of final accuracy
plt.figure(figsize=(8, 5))
sns.barplot(x='Optimizer', y='Final Val Accuracy', data=results_df, palette='Set2')
plt.title('Final Validation Accuracy by Optimizer')
plt.ylabel('Accuracy (%)')
plt.ylim(85, 100)
for i, row in results_df.iterrows():
    plt.text(i, row['Final Val Accuracy'] + 0.5, f"{row['Final Val Accuracy']:.2f}%", ha='center')
plt.tight_layout()
plt.show()
```

```
Training with Adam optimizer...

Training with RMSprop optimizer...
```




```
Final Validation Accuracy (Sorted):
Optimizer  Final Val Accuracy
  RMSprop               97.73
     Adam               97.65
      SGD               93.23
```



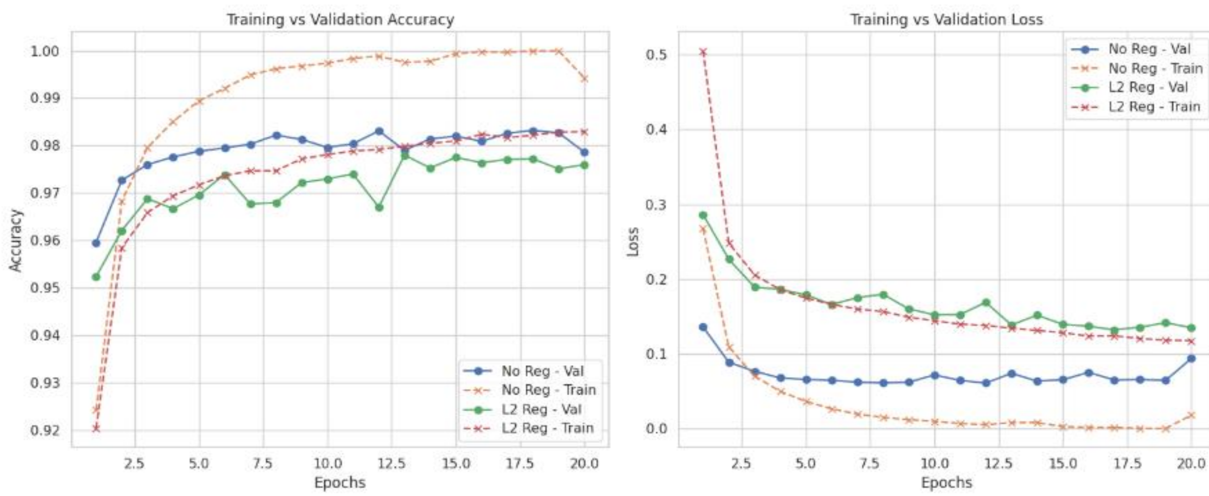
Practical 2

AIM - Write a Program to implement regularization to prevent the model from overfitting.

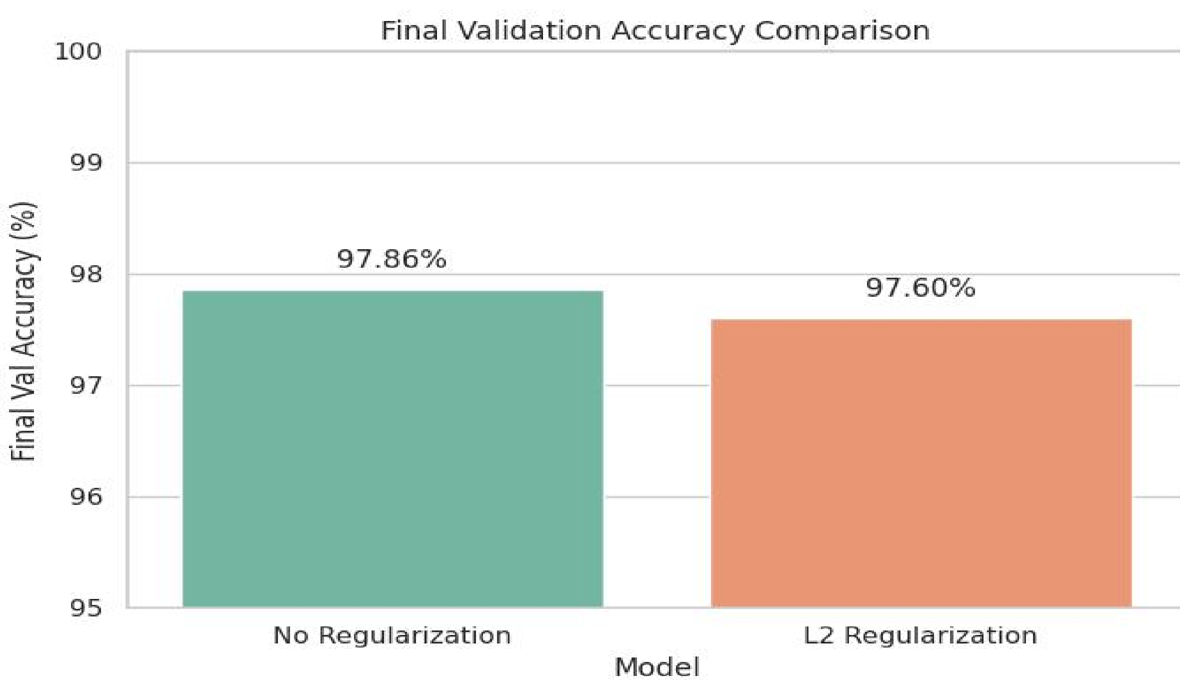
```
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
import pandas as pd
from tensorflow.keras.datasets import mnist
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, Flatten
from tensorflow.keras.regularizers import l2
from tensorflow.keras.utils import to_categorical

# Load and normalize MNIST data
(x_train, y_train), (x_test, y_test) = mnist.load_data()
x_train, x_test = x_train / 255.0, x_test / 255.0
y_train, y_test = to_categorical(y_train, 10), to_categorical(y_test, 10)

# ---- 1. Model without regularization ----
model_plain = Sequential([
    Flatten(input_shape=(28, 28)),
    Dense(512, activation='relu'),
    Dense(10, activation='softmax')
])
model_plain.compile(optimizer='adam', loss='categorical_crossentropy', metrics=['accuracy'])

# ---- 2. Model with L2 regularization ----
model_l2 = Sequential([
    Flatten(input_shape=(28, 28)),
    Dense(512, activation='relu', kernel_regularizer=l2(0.001)),
    Dense(10, activation='softmax')
])
model_l2.compile(optimizer='adam', loss='categorical_crossentropy', metrics=['accuracy'])

# ---- Train both models ----
history_plain = model_plain.fit(x_train, y_train,
                                validation_data=(x_test, y_test),
                                epochs=20, batch_size=128, verbose=0)

history_l2 = model_l2.fit(x_train, y_train,
                          validation_data=(x_test, y_test),
                          epochs=20, batch_size=128, verbose=0)

# ---- Create summary ----
results = pd.DataFrame({
    'Model': ['No Regularization', 'L2 Regularization'],
```

```
    'Final Val Accuracy (%)': [
        history_plain.history['val_accuracy'][-1] * 100,
        history_l2.history['val_accuracy'][-1] * 100
    ],
    'Final Val Loss': [
        history_plain.history['val_loss'][-1],
        history_l2.history['val_loss'][-1]
    ]
}).sort_values(by='Final Val Accuracy (%)', ascending=False)

# ---- Visualization ----
sns.set(style="whitegrid")
epochs = range(1, 21)

plt.figure(figsize=(14, 6))

# --- Accuracy Plot ---
plt.subplot(1, 2, 1)
plt.plot(epochs, history_plain.history['val_accuracy'], label='No Reg - Val', marker='o')
plt.plot(epochs, history_plain.history['accuracy'], label='No Reg - Train', linestyle='--', marker='x')
plt.plot(epochs, history_l2.history['val_accuracy'], label='L2 Reg - Val', marker='o')
plt.plot(epochs, history_l2.history['accuracy'], label='L2 Reg - Train', linestyle='--', marker='x')
plt.title('Training vs Validation Accuracy')
plt.xlabel('Epochs')
plt.ylabel('Accuracy')
plt.legend()

# --- Loss Plot ---
plt.subplot(1, 2, 2)
plt.plot(epochs, history_plain.history['val_loss'], label='No Reg - Val', marker='o')
plt.plot(epochs, history_plain.history['loss'], label='No Reg - Train', linestyle='--', marker='x')
plt.plot(epochs, history_l2.history['val_loss'], label='L2 Reg - Val', marker='o')
plt.plot(epochs, history_l2.history['loss'], label='L2 Reg - Train', linestyle='--', marker='x')
plt.title('Training vs Validation Loss')
plt.xlabel('Epochs')
plt.ylabel('Loss')
plt.legend()

plt.tight_layout()
plt.show()

# --- Summary Table ---
print("\nFinal Metrics Summary:")
print(results.to_string(index=False, float_format="%.2f"))

# Optional: Bar plot of final accuracy
plt.figure(figsize=(7, 5))
sns.barplot(x='Model', y='Final Val Accuracy (%)', data=results, palette='Set2')
```

```
plt.title('Final Validation Accuracy Comparison')
plt.ylim(95, 100)
for i, row in results.iterrows():
    plt.text(i, row['Final Val Accuracy (%)'] + 0.2,
             f"{row['Final Val Accuracy (%)']:.2f}%", ha='center')
plt.tight_layout()
plt.show()
```




```
Final Metrics Summary:
                Model  Final Val Accuracy (%)  Final Val Loss
No Regularization                       97.86            0.09
L2 Regularization                       97.60            0.14
```



Practical 3

AIM - Implement deep learning for recognizing classes for datasets like CIFAR10 images for previously unseen images and assign them to one of the 10 classes

```
import tensorflow as tf
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
from tensorflow.keras.layers import Input, Conv2D, Dense, Flatten, Dropout, BatchNormalization, MaxPooling2D
from tensorflow.keras.models import Model
from tensorflow.keras.preprocessing.image import ImageDataGenerator
from tensorflow.keras.callbacks import EarlyStopping, ReduceLROnPlateau
from tensorflow.keras.mixed_precision import set_global_policy

# Enable mixed precision for better performance
set_global_policy('mixed_float16')

# Load CIFAR-10 dataset
(x_train, y_train), (x_test, y_test) = tf.keras.datasets.cifar10.load_data()

# Normalize pixel values
x_train, x_test = x_train / 255.0, x_test / 255.0
y_train, y_test = y_train.flatten(), y_test.flatten()

# Number of classes
K = len(set(y_train))

# Build the deeper CNN model
i = Input(shape=x_train[0].shape)

# Block 1
x = Conv2D(32, (3, 3), activation='relu', padding='same')(i)
x = BatchNormalization()(x)
x = Conv2D(32, (3, 3), activation='relu', padding='same')(x)
x = BatchNormalization()(x)
x = MaxPooling2D((2, 2))(x)

# Block 2
x = Conv2D(64, (3, 3), activation='relu', padding='same')(x)
x = BatchNormalization()(x)
x = Conv2D(64, (3, 3), activation='relu', padding='same')(x)
x = BatchNormalization()(x)
x = MaxPooling2D((2, 2))(x)

# Block 3
x = Conv2D(128, (3, 3), activation='relu', padding='same')(x)
```

```
x = BatchNormalization()(x)
x = Conv2D(128, (3, 3), activation='relu', padding='same')(x)
x = BatchNormalization()(x)
x = MaxPooling2D((2, 2))(x)

# Fully connected
x = Flatten()(x)
x = Dropout(0.3)(x)
x = Dense(1024, activation='relu')(x)
x = Dropout(0.3)(x)
x = Dense(K, activation='softmax', dtype='float32')(x)  # Ensure output is float32 for mixed precision

model = Model(i, x)

# Compile the model
model.compile(optimizer='adam',
              loss='sparse_categorical_crossentropy',
              metrics=['accuracy'])

# Model summary
model.summary()

# Data augmentation
batch_size = 32
datagen = ImageDataGenerator(
    width_shift_range=0.1,
    height_shift_range=0.1,
    horizontal_flip=True,
    rotation_range=10,
    zoom_range=0.1
)
train_gen = datagen.flow(x_train, y_train, batch_size=batch_size)

# Callbacks
early_stopping = EarlyStopping(monitor='val_accuracy', patience=5, restore_best_weights=True)
lr_scheduler = ReduceLROnPlateau(monitor='val_loss', factor=0.5, patience=3, min_lr=1e-5)

# Fit model
history = model.fit(train_gen,
                    validation_data=(x_test, y_test),
                    steps_per_epoch=len(x_train) // batch_size,
                    epochs=50,
                    callbacks=[early_stopping, lr_scheduler])

# Plot accuracy using Seaborn
sns.set(style="whitegrid")
plt.figure(figsize=(10, 5))
plt.plot(history.history['accuracy'], label='Train Accuracy', color='darkorange')
```

```
plt.plot(history.history['val_accuracy'], label='Validation Accuracy', color='teal')
plt.title("Training vs Validation Accuracy", fontsize=14)
plt.xlabel("Epoch")
plt.ylabel("Accuracy")
plt.legend()
plt.tight_layout()
plt.savefig("accuracy_plot.png", dpi=150)  # higher quality
plt.show()

# Class labels
labels = ['airplane', 'automobile', 'bird', 'cat', 'deer',
          'dog', 'frog', 'horse', 'ship', 'truck']

# Predict a test image
image_number = 0
test_img = x_test[image_number]
true_label = labels[y_test[image_number]]
predicted_label = labels[model.predict(test_img.reshape(1, 32, 32, 3)).argmax()]

# Show image + result with smooth rendering
plt.figure(figsize=(4, 4))
plt.imshow(test_img, interpolation='bilinear')  # better visual quality
plt.title(f"True: {true_label} | Pred: {predicted_label}", fontsize=12, color='navy')
plt.axis('off')
plt.tight_layout()
plt.savefig("test_prediction.png", dpi=150)  # save with good resolution
plt.show()

# Save the model
model.save('cifar10_deep_model.h5')
```

Model: "functional_2"

Layer (type)	Output Shape	Param #
input_layer_2 (InputLayer)	(None, 32, 32, 3)	0
cast_4 (Cast)	(None, 32, 32, 3)	0
conv2d_11 (Conv2D)	(None, 32, 32, 32)	896
batch_normalization_10 (BatchNormalization)	(None, 32, 32, 32)	128
conv2d_12 (Conv2D)	(None, 32, 32, 32)	9,248
batch_normalization_11 (BatchNormalization)	(None, 32, 32, 32)	128
max_pooling2d_5 (MaxPooling2D)	(None, 16, 16, 32)	0
conv2d_13 (Conv2D)	(None, 16, 16, 64)	18,496
batch_normalization_12 (BatchNormalization)	(None, 16, 16, 64)	256
conv2d_14 (Conv2D)	(None, 16, 16, 64)	36,928
batch_normalization_13 (BatchNormalization)	(None, 16, 16, 64)	256

```
 Total params: 2,397,226 (9.14 MB)
 Trainable params: 2,396,330 (9.14 MB)
 Non-trainable params: 896 (3.50 KB)
Epoch 1/50
1562/1562 ━━━━━━━━━━━━━━━━━━━━ 51s 25ms/step - accuracy: 0.3830 - loss: 1.9720 - val_accuracy: 0.5679 - val_loss: 1
Epoch 2/50
1562/1562 ━━━━━━━━━━━━━━━━━━━━ 1s 538us/step - accuracy: 0.5938 - loss: 1.2919 - val_accuracy: 0.5705 - val_loss: 1
Epoch 3/50
1562/1562 ━━━━━━━━━━━━━━━━━━━━ 35s 22ms/step - accuracy: 0.5917 - loss: 1.1624 - val_accuracy: 0.6596 - val_loss: 1
Epoch 4/50
1562/1562 ━━━━━━━━━━━━━━━━━━━━ 1s 838us/step - accuracy: 0.6562 - loss: 1.0369 - val_accuracy: 0.6677 - val_loss: 0
Epoch 5/50
1562/1562 ━━━━━━━━━━━━━━━━━━━━ 39s 22ms/step - accuracy: 0.6669 - loss: 0.9614 - val_accuracy: 0.6718 - val_loss: 0
Epoch 6/50
1562/1562 ━━━━━━━━━━━━━━━━━━━━ 1s 540us/step - accuracy: 0.8125 - loss: 0.4995 - val_accuracy: 0.6735 - val_loss: 0
Epoch 7/50
1562/1562 ━━━━━━━━━━━━━━━━━━━━ 35s 22ms/step - accuracy: 0.7048 - loss: 0.8661 - val_accuracy: 0.7335 - val_loss: 0
Epoch 8/50
1562/1562 ━━━━━━━━━━━━━━━━━━━━ 1s 838us/step - accuracy: 0.7500 - loss: 0.6291 - val_accuracy: 0.7356 - val_loss: 0
Epoch 9/50
1562/1562 ━━━━━━━━━━━━━━━━━━━━ 35s 22ms/step - accuracy: 0.7287 - loss: 0.7904 - val_accuracy: 0.7399 - val_loss: 0
Epoch 10/50
```


Training vs Validation Accuracy
Train Accuracy
Validation Accuracy
Accuracy
Epoch
1.0
0.9
0.8
0.7
0.6
0.5
0.4
0
10
20
30
40
50


True: cat | Pred: cat

Practical 4

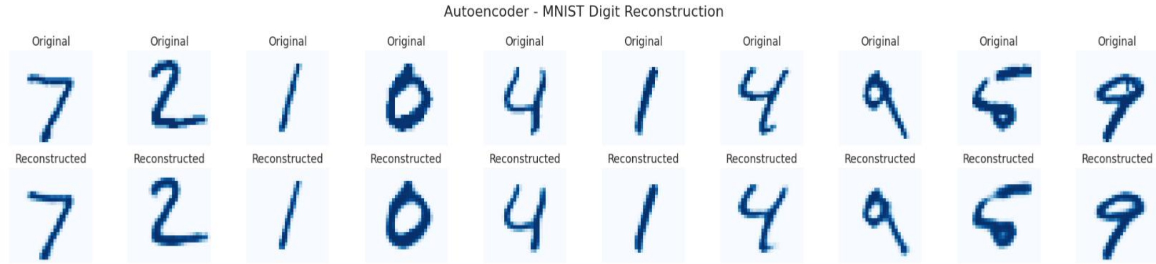
AIM - Implement deep learning for the Prediction of the autoencoder from the test data (e.g. MNIST data set).

```
import numpy as np
import tensorflow as tf
from tensorflow.keras import layers
import matplotlib.pyplot as plt
import seaborn as sns

sns.set(style="whitegrid")

# Load and Preprocess the MNIST Dataset
(x_train, _), (x_test, _) = tf.keras.datasets.mnist.load_data()
x_train = x_train.astype('float32') / 255.
x_test = x_test.astype('float32') / 255.
x_train = np.reshape(x_train, (len(x_train), 28, 28, 1))
x_test = np.reshape(x_test, (len(x_test), 28, 28, 1))

latent_dim = 64

# Encoder
encoder_input = layers.Input(shape=(28, 28, 1))
x = layers.Conv2D(32, (3, 3), activation='relu', padding='same')(encoder_input)
x = layers.MaxPooling2D((2, 2), padding='same')(x)
x = layers.Conv2D(64, (3, 3), activation='relu', padding='same')(x)
x = layers.MaxPooling2D((2, 2), padding='same')(x)
x = layers.Flatten()(x)
latent_output = layers.Dense(latent_dim, activation='relu')(x)
encoder = tf.keras.Model(encoder_input, latent_output, name="encoder")

# Decoder
decoder_input = layers.Input(shape=(latent_dim,))
x = layers.Dense(7 * 7 * 64, activation='relu')(decoder_input)
x = layers.Reshape((7, 7, 64))(x)
x = layers.Conv2DTranspose(64, (3, 3), strides=2, activation='relu', padding='same')(x)
x = layers.Conv2DTranspose(32, (3, 3), strides=2, activation='relu', padding='same')(x)
decoder_output = layers.Conv2DTranspose(1, (3, 3), activation='sigmoid', padding='same')(x)
decoder = tf.keras.Model(decoder_input, decoder_output, name="decoder")

# Connect encoder to decoder
autoencoder_input = encoder_input
encoded = encoder(autoencoder_input)
decoded = decoder(encoded)
autoencoder = tf.keras.Model(autoencoder_input, decoded, name="autoencoder")
```

```
# Compile and Train
autoencoder.compile(optimizer='adam', loss='binary_crossentropy')
autoencoder.fit(x_train, x_train,
                epochs=10,
                batch_size=128,
                shuffle=True,
                validation_data=(x_test, x_test))

# Predict
decoded_imgs = autoencoder.predict(x_test)

# Visualize with Seaborn styling
n = 10
plt.figure(figsize=(20, 4))
for i in range(n):
    ax = plt.subplot(2, n, i + 1)
    plt.imshow(x_test[i].reshape(28, 28), cmap='Blues')
    plt.title("Original")
    plt.axis('off')

    ax = plt.subplot(2, n, i + 1 + n)
    plt.imshow(decoded_imgs[i].reshape(28, 28), cmap='Blues')
    plt.title("Reconstructed")
    plt.axis('off')

plt.suptitle("Autoencoder - MNIST Digit Reconstruction", fontsize=16)
plt.tight_layout()
plt.show()
```

```
Epoch 1/10
469/469 ━━━━━━━━━━━━━━━━━━━━ 72s 149ms/step - loss: 0.2571 - val_loss: 0.0940
Epoch 2/10
469/469 ━━━━━━━━━━━━━━━━━━━━ 67s 144ms/step - loss: 0.0913 - val_loss: 0.0823
Epoch 3/10
469/469 ━━━━━━━━━━━━━━━━━━━━ 67s 143ms/step - loss: 0.0815 - val_loss: 0.0780
Epoch 4/10
469/469 ━━━━━━━━━━━━━━━━━━━━ 67s 143ms/step - loss: 0.0776 - val_loss: 0.0751
Epoch 5/10
469/469 ━━━━━━━━━━━━━━━━━━━━ 67s 143ms/step - loss: 0.0754 - val_loss: 0.0734
Epoch 6/10
469/469 ━━━━━━━━━━━━━━━━━━━━ 67s 143ms/step - loss: 0.0739 - val_loss: 0.0724
Epoch 7/10
469/469 ━━━━━━━━━━━━━━━━━━━━ 67s 143ms/step - loss: 0.0725 - val_loss: 0.0715
Epoch 8/10
469/469 ━━━━━━━━━━━━━━━━━━━━ 67s 143ms/step - loss: 0.0718 - val_loss: 0.0710
Epoch 9/10
469/469 ━━━━━━━━━━━━━━━━━━━━ 67s 143ms/step - loss: 0.0710 - val_loss: 0.0709
Epoch 10/10
469/469 ━━━━━━━━━━━━━━━━━━━━ 67s 143ms/step - loss: 0.0706 - val_loss: 0.0697
313/313 ━━━━━━━━━━━━━━━━━━━━ 5s 14ms/step
```

```
313/313 ━━━━━━━━━━━━━━━━━━━━ 5s 14ms/step
```



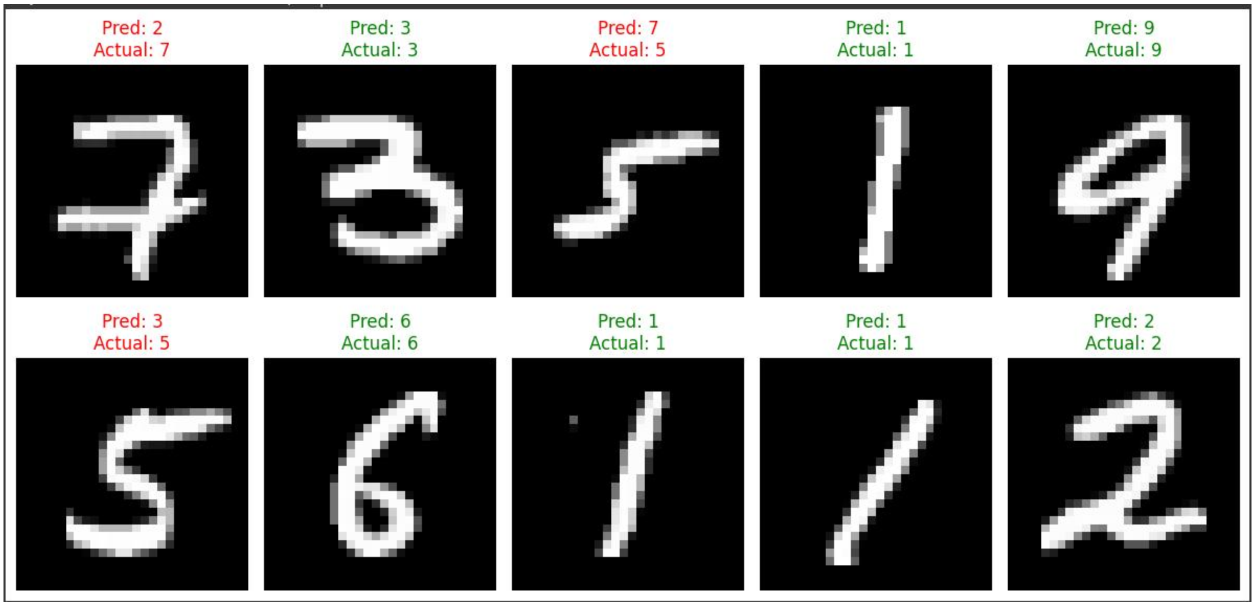
Practical 5

AIM - Implement Convolutional Neural Network for Digit Recognition on the MNIST Dataset.

```
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Conv2D, MaxPooling2D, Flatten, Dense, Dropout
from tensorflow.keras import regularizers
from tensorflow.keras.preprocessing.image import ImageDataGenerator
from tensorflow.keras.callbacks import EarlyStopping
import numpy as np
from tensorflow.keras.datasets import mnist
from tensorflow.keras.utils import to_categorical
import matplotlib.pyplot as plt
import random

# Load data
(x_train, y_train), (x_test, y_test) = mnist.load_data()

# Preprocess data
x_train = x_train.reshape(-1, 28, 28, 1).astype('float32') / 255
x_test = x_test.reshape(-1, 28, 28, 1).astype('float32') / 255
y_train = to_categorical(y_train, 10)
y_test = to_categorical(y_test, 10)

# Create data augmentation object
datagen = ImageDataGenerator(
   rotation_range=15,
   width_shift_range=0.1,
   height_shift_range=0.1,
   shear_range=0.2,
   zoom_range=0.1,
   horizontal_flip=True,
   fill_mode='nearest'
)
datagen.fit(x_train)

# Define the model with regularization and dropout
model = Sequential([
   Conv2D(8, kernel_size=(3, 3), activation='relu', input_shape=(28, 28, 1),
kernel_regularizer=regularizers.l2(0.001)),
   MaxPooling2D(pool_size=(2, 2)),
   Flatten(),
   Dense(16, activation='relu', kernel_regularizer=regularizers.l2(0.001)),
   Dropout(0.5),
   Dense(10, activation='softmax')
])
```

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

# Set up early stopping
early_stop = EarlyStopping(monitor='val_loss', patience=3, restore_best_weights=True)

# Train the model with augmented data and early stopping
model.fit(datagen.flow(x_train, y_train, batch_size=128), epochs=10, validation_data=(x_test, y_test),
callbacks=[early_stop], verbose=1)

# Evaluate the model
loss, accuracy = model.evaluate(x_test, y_test, verbose=0)
print(f"Test Loss: {loss:.4f}")
print(f"Test Accuracy: {accuracy:.4f}")

# Predict on the test set
predictions = model.predict(x_test)
predicted_classes = np.argmax(predictions, axis=1)
true_classes = np.argmax(y_test, axis=1)

# Visualize 10 random images along with their predictions
plt.figure(figsize=(12, 6))
for i in range(10):
    index = random.randint(0, len(x_test) - 1)
    image = x_test[index].reshape(28, 28)
    plt.subplot(2, 5, i + 1)
    plt.imshow(image, cmap='gray')
    pred = predicted_classes[index]
    true = true_classes[index]
    color = 'green' if pred == true else 'red'  # Green if correctly classified, red if misclassified
    plt.title(f"Pred: {pred}\nActual: {true}", color=color)
    plt.axis('off')

plt.tight_layout()
plt.show()
```

```
Epoch 1/10
469/469 ━━━━━━━━━━━━━━━━━━━━ 22s 40ms/step - accuracy: 0.2838 - loss: 1.9887 - val_accuracy: 0.7803 - val_loss: 0.9173
Epoch 2/10
469/469 ━━━━━━━━━━━━━━━━━━━━ 18s 39ms/step - accuracy: 0.4638 - loss: 1.4745 - val_accuracy: 0.7814 - val_loss: 0.7695
Epoch 3/10
469/469 ━━━━━━━━━━━━━━━━━━━━ 17s 36ms/step - accuracy: 0.4977 - loss: 1.3843 - val_accuracy: 0.7971 - val_loss: 0.7417
Epoch 4/10
469/469 ━━━━━━━━━━━━━━━━━━━━ 18s 38ms/step - accuracy: 0.5172 - loss: 1.3256 - val_accuracy: 0.8241 - val_loss: 0.6572
Epoch 5/10
469/469 ━━━━━━━━━━━━━━━━━━━━ 18s 38ms/step - accuracy: 0.5326 - loss: 1.2987 - val_accuracy: 0.8296 - val_loss: 0.6401
Epoch 6/10
469/469 ━━━━━━━━━━━━━━━━━━━━ 18s 39ms/step - accuracy: 0.5443 - loss: 1.2759 - val_accuracy: 0.8350 - val_loss: 0.5884
Epoch 7/10
469/469 ━━━━━━━━━━━━━━━━━━━━ 18s 38ms/step - accuracy: 0.5482 - loss: 1.2502 - val_accuracy: 0.8352 - val_loss: 0.5748
Epoch 8/10
469/469 ━━━━━━━━━━━━━━━━━━━━ 18s 38ms/step - accuracy: 0.5515 - loss: 1.2432 - val_accuracy: 0.8310 - val_loss: 0.5965
Epoch 9/10
469/469 ━━━━━━━━━━━━━━━━━━━━ 17s 36ms/step - accuracy: 0.5516 - loss: 1.2336 - val_accuracy: 0.8448 - val_loss: 0.5582
Epoch 10/10
469/469 ━━━━━━━━━━━━━━━━━━━━ 22s 39ms/step - accuracy: 0.5614 - loss: 1.2144 - val_accuracy: 0.8467 - val_loss: 0.5251
Test Loss: 0.5251
Test Accuracy: 0.8467
313/313 ━━━━━━━━━━━━━━━━━━━━ 1s 2ms/step
```



Practical 6

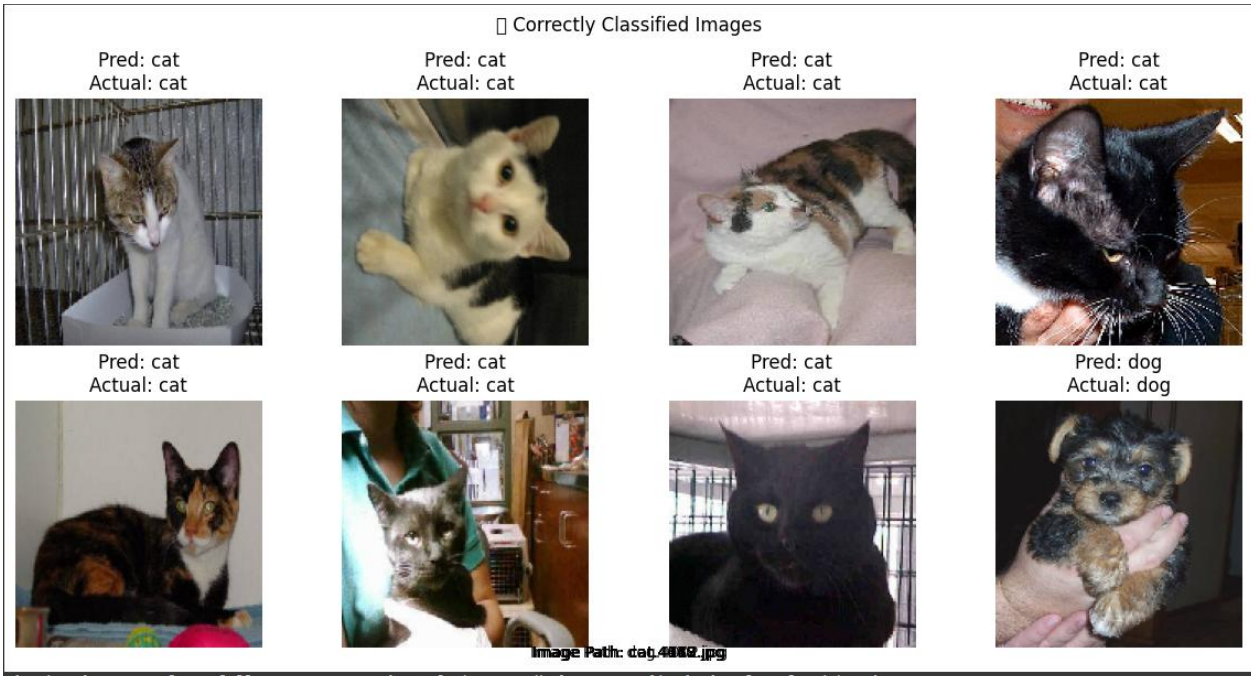
AIM - Write a program to implement Transfer Learning on the suitable dataset (e.g. classify the cats versus dogs dataset from Kaggle).

```
import os
import zipfile
import requests
import shutil
import random
import numpy as np
import matplotlib.pyplot as plt
from tensorflow.keras.preprocessing.image import ImageDataGenerator, load_img, img_to_array
from keras.models import Sequential
from keras.layers import Conv2D, MaxPooling2D, Flatten, Dense

# Step 1: Download and unzip the dataset
url = "https://github.com/Junaid2003/Cat_vs_dog_dataset/raw/main/cats_vs_dogs_dataset.zip"
zip_path = "/content/cats_vs_dogs_dataset.zip"

response = requests.get(url)
with open(zip_path, "wb") as f:
    f.write(response.content)

with zipfile.ZipFile(zip_path, "r") as zip_ref:
    zip_ref.extractall("/content/")

# Step 2: Organize paths
dataset_path = "/content"
train_path = os.path.join(dataset_path, "train")
test_path = os.path.join(dataset_path, "test")
cats_set_path = os.path.join(dataset_path, "cats_set")
dogs_set_path = os.path.join(dataset_path, "dogs_set")

os.makedirs(os.path.join(train_path, 'cats'), exist_ok=True)
os.makedirs(os.path.join(train_path, 'dogs'), exist_ok=True)
os.makedirs(os.path.join(test_path, 'cats'), exist_ok=True)
os.makedirs(os.path.join(test_path, 'dogs'), exist_ok=True)

# Step 3: Move files into train/test folders
def move_files(src_dir, dest_train_dir, dest_test_dir, train_ratio=0.8):
    files = os.listdir(src_dir)
    random.shuffle(files)
    split_index = int(len(files) * train_ratio)
    for f in files[:split_index]:
        shutil.move(os.path.join(src_dir, f), os.path.join(dest_train_dir, f))
    for f in files[split_index:]:
```

```
        shutil.move(os.path.join(src_dir, f), os.path.join(dest_test_dir, f))

move_files(cats_set_path, os.path.join(train_path, 'cats'), os.path.join(test_path, 'cats'))
move_files(dogs_set_path, os.path.join(train_path, 'dogs'), os.path.join(test_path, 'dogs'))

# Step 4: Create ImageDataGenerators
train_gen = ImageDataGenerator(rescale=1./255)
test_gen = ImageDataGenerator(rescale=1./255)

train_generator = train_gen.flow_from_directory(train_path,
                                                target_size=(150, 150),
                                                batch_size=32,
                                                class_mode='binary')

validation_generator = test_gen.flow_from_directory(test_path,
                                                    target_size=(150, 150),
                                                    batch_size=32,
                                                    class_mode='binary')

# Step 5: Build CNN model
model = Sequential([
    Conv2D(32, (3, 3), activation='relu', input_shape=(150, 150, 3)),
    MaxPooling2D(2, 2),
    Conv2D(64, (3, 3), activation='relu'),
    MaxPooling2D(2, 2),
    Flatten(),
    Dense(128, activation='relu'),
    Dense(1, activation='sigmoid')
])

model.compile(optimizer='adam',
              loss='binary_crossentropy',
              metrics=['accuracy'])

# Step 6: Train model
history = model.fit(train_generator,
                    epochs=5,
                    validation_data=validation_generator,
                    verbose=1)

print("✅ Model training completed.")

# Step 7: Evaluate model
loss, accuracy = model.evaluate(validation_generator, verbose=0)
print(f"Test Loss: {loss:.4f}")
print(f"Test Accuracy: {accuracy:.4f}")

# Step 8: Predict & show classified/misclassified images
```

```
classified = []
misclassified = []

# Sample 20 random images for evaluation
for category in ['cats', 'dogs']:
    test_dir = os.path.join(test_path, category)
    files = random.sample(os.listdir(test_dir), 10)

    for fname in files:
        path = os.path.join(test_dir, fname)
        img = load_img(path, target_size=(150, 150))
        arr = img_to_array(img) / 255.0
        arr = np.expand_dims(arr, axis=0)

        pred = model.predict(arr)[0][0]
        predicted_label = 'dog' if pred > 0.5 else 'cat'
        actual_label = 'dog' if 'dog' in category else 'cat'

        if predicted_label == actual_label:
            classified.append((img, predicted_label, actual_label, path))
        else:
            misclassified.append((img, predicted_label, actual_label, path))

# Step 9: Display sample predictions with pet images
def show_images(images, title):
    plt.figure(figsize=(12, 6))
    for i, (img, pred, actual, path) in enumerate(images[:8]):
        plt.subplot(2, 4, i + 1)
        plt.imshow(img)
        plt.title(f"Pred: {pred}\nActual: {actual}")
        plt.axis('off')

        # Adding path of the image (e.g., displaying a preview of the filename as an extra detail)
        plt.figtext(0.5, 0.01, f"Image Path: {os.path.basename(path)}", wrap=True, horizontalalignment='center',
fontsize=10)

    plt.suptitle(title)
    plt.tight_layout()
    plt.show()

show_images(classified, "✅ Correctly Classified Images")
show_images(misclassified, "❌ Misclassified Images")
```

```
Found 999 images belonging to 2 classes.
Found 741 images belonging to 2 classes.
Epoch 1/5
32/32 ━━━━━━━━━━━━━━━━━━━━ 7s 154ms/step - accuracy: 0.5048 - loss: 1.6286 - val_accuracy: 0.6383 - val_loss: 0.6890
Epoch 2/5
32/32 ━━━━━━━━━━━━━━━━━━━━ 2s 78ms/step - accuracy: 0.6239 - loss: 0.6833 - val_accuracy: 0.6478 - val_loss: 0.6361
Epoch 3/5
32/32 ━━━━━━━━━━━━━━━━━━━━ 3s 79ms/step - accuracy: 0.6561 - loss: 0.6320 - val_accuracy: 0.7503 - val_loss: 0.5457
Epoch 4/5
32/32 ━━━━━━━━━━━━━━━━━━━━ 3s 80ms/step - accuracy: 0.7259 - loss: 0.5481 - val_accuracy: 0.8367 - val_loss: 0.4056
Epoch 5/5
32/32 ━━━━━━━━━━━━━━━━━━━━ 4s 122ms/step - accuracy: 0.8036 - loss: 0.4107 - val_accuracy: 0.8475 - val_loss: 0.3268
✅ Model training completed.
Test Loss: 0.3268
Test Accuracy: 0.8475
```



Practical 7

AIM - Write a program for the Implementation of a Generative Adversarial Network for generating synthetic shapes (like digits)

```
import tensorflow as tf
tf.__version__

import glob
import imageio
import matplotlib.pyplot as plt
import numpy as np
import os
import PIL
from tensorflow.keras import layers
import time

from IPython import display
(train_images, train_labels), (_, _) = tf.keras.datasets.mnist.load_data()
train_images = train_images.reshape(train_images.shape[0], 28, 28, 1).astype('float32')
train_images = (train_images - 127.5) / 127.5  # Normalize the images to [-1, 1]
BUFFER_SIZE = 60000
BATCH_SIZE = 256
# Batch and shuffle the data
train_dataset = tf.data.Dataset.from_tensor_slices(train_images).shuffle(BUFFER_SIZE).batch(BATCH_SIZE)

def make_generator_model():
    model = tf.keras.Sequential()
    model.add(layers.Dense(7*7*256, use_bias=False, input_shape=(100,)))
    model.add(layers.BatchNormalization())
    model.add(layers.LeakyReLU())

    model.add(layers.Reshape((7, 7, 256)))
    assert model.output_shape == (None, 7, 7, 256)  # Note: None is the batch size

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

    model.add(layers.Conv2DTranspose(64, (5, 5), strides=(2, 2), padding='same', use_bias=False))
    assert model.output_shape == (None, 14, 14, 64)
    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'))
    assert model.output_shape == (None, 28, 28, 1)
```

```
    return model

generator = make_generator_model()

noise = tf.random.normal([1, 100])
generated_image = generator(noise, training=False)

plt.imshow(generated_image[0, :, :, 0], cmap='gray')

def make_discriminator_model():
    model = tf.keras.Sequential()
    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

discriminator = make_discriminator_model()
decision = discriminator(generated_image)
print (decision)


# This method returns a helper function to compute cross entropy loss
cross_entropy = tf.keras.losses.BinaryCrossentropy(from_logits=True)

def discriminator_loss(real_output, fake_output):
    real_loss = cross_entropy(tf.ones_like(real_output), real_output)
    fake_loss = cross_entropy(tf.zeros_like(fake_output), fake_output)
    total_loss = real_loss + fake_loss
    return total_loss

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

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

checkpoint_dir = './training_checkpoints'
checkpoint_prefix = os.path.join(checkpoint_dir, "ckpt")
```

```
checkpoint = tf.train.Checkpoint(generator_optimizer=generator_optimizer,
                                 discriminator_optimizer=discriminator_optimizer,
                                 generator=generator,
                                 discriminator=discriminator)

EPOCHS = 50
noise_dim = 100
num_examples_to_generate = 16

# You will reuse this seed overtime (so it's easier)
# to visualize progress in the animated GIF)
seed = tf.random.normal([num_examples_to_generate, noise_dim])


# Notice the use of `tf.function`
# This annotation causes the function to be "compiled".
@tf.function
def train_step(images):
    noise = tf.random.normal([BATCH_SIZE, noise_dim])

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

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

      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))


def train(dataset, epochs):
  for epoch in range(epochs):
    start = time.time()

    for image_batch in dataset:
      train_step(image_batch)

    # Produce images for the GIF as you go
    display.clear_output(wait=True)
    generate_and_save_images(generator,
                             epoch + 1,
                             seed)
```

```
    # Save the model every 15 epochs
    if (epoch + 1) % 15 == 0:
      checkpoint.save(file_prefix = checkpoint_prefix)

    print ('Time for epoch {} is {} sec'.format(epoch + 1, time.time()-start))

  # Generate after the final epoch
  display.clear_output(wait=True)
  generate_and_save_images(generator,
                           epochs,
                           seed)



def generate_and_save_images(model, epoch, test_input):
  # Notice `training` is set to False.
  # This is so all layers run in inference mode (batchnorm).
  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('image_at_epoch_{:04d}.png'.format(epoch))
  plt.show()

train(train_dataset, EPOCHS)

checkpoint.restore(tf.train.latest_checkpoint(checkpoint_dir))

# Display a single image using the epoch number
def display_image(epoch_no):
  return PIL.Image.open('image_at_epoch_{:04d}.png'.format(epoch_no))

display_image(EPOCHS)

anim_file = 'dcgan.gif'

with imageio.get_writer(anim_file, mode='I') as writer:
  filenames = glob.glob('image*.png')
  filenames = sorted(filenames)
  for filename in filenames:
    image = imageio.imread(filename)
    writer.append_data(image)
```

```
  image = imageio.imread(filename)
  writer.append_data(image)

import tensorflow_docs.vis.embed as embed
embed.embed_file(anim_file)
```

Input digit image

Output digit image

Practical 8

AIM - Write a program to implement a simple form of a recurrent neural network.
E.g. (4-to-1 RNN) to show that the quantity of rain on a certain day also depends on the values of the previous day

Steps: Rainfall data is collected from the opencity database. This dataset contains rainfall information from chennai Monsoon season in mm. Due to its large size, the dataset contains data from the year 1973 and contains 100 records for simplicity.

```
import pandas as pd
import numpy as np
import tensorflow as tf
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import SimpleRNN, Dense
from sklearn.preprocessing import MinMaxScaler
import seaborn as sns
import matplotlib.pyplot as plt
import plotly.graph_objects as go

# Load your Excel dataset
url = "https://raw.githubusercontent.com/Junaid2003/rainfall_dataset/main/rainfall_dataset_small.csv"
df = pd.read_csv(url)

# Display first few rows
print("First few rows of the dataset:")
print(df.head())

# --- Check if 'Rainfall' column exists ---
if 'Rainfall' not in df.columns:
    raise ValueError("The dataset must contain a 'Rainfall' column.")

# Plot distribution using Seaborn
sns.set(style="whitegrid")
plt.figure(figsize=(10, 4))
sns.histplot(df['Rainfall'], bins=30, kde=True)
plt.title("Rainfall Distribution")
plt.xlabel("Rainfall")
plt.ylabel("Frequency")
plt.show()

# --- Data Preprocessing ---
rainfall_data = df['Rainfall'].values.reshape(-1, 1)

# Normalize using MinMaxScaler
scaler = MinMaxScaler()
rainfall_scaled = scaler.fit_transform(rainfall_data)
```

```
# Sequence function: 4-day input → 1-day output
def create_sequences(data, window_size):
    X, y = [], []
    for i in range(len(data) - window_size):
        X.append(data[i:i + window_size])
        y.append(data[i + window_size])
    return np.array(X), np.array(y)

window_size = 4
X, y = create_sequences(rainfall_scaled, window_size)
X = X.reshape((X.shape[0], window_size, 1))

# --- RNN Model ---
model = Sequential([
    SimpleRNN(10, activation='tanh', input_shape=(window_size, 1)),
    Dense(1)
])
model.compile(optimizer='adam', loss='mse')
model.fit(X, y, epochs=100, batch_size=8, verbose=0)

# --- Make Predictions ---
predictions = model.predict(X)
predicted_rainfall = scaler.inverse_transform(predictions)
actual_rainfall = scaler.inverse_transform(y)

# --- Prepare Plot Data ---
results_df = pd.DataFrame({
    "Day": np.arange(len(actual_rainfall)),
    "Actual Rainfall": actual_rainfall.flatten(),
    "Predicted Rainfall": predicted_rainfall.flatten()
})

# --- Interactive Plotly Chart ---
fig = go.Figure()
fig.add_trace(go.Scatter(
    x=results_df["Day"], y=results_df["Actual Rainfall"],
    mode='lines+markers', name='Actual Rainfall',
    line=dict(color='blue')
))
fig.add_trace(go.Scatter(
    x=results_df["Day"], y=results_df["Predicted Rainfall"],
    mode='lines+markers', name='Predicted Rainfall',
    line=dict(color='orange')
))
fig.update_layout(
    title="Interactive Rainfall Forecast Using 4-to-1 RNN",
    xaxis_title="Day",
    yaxis_title="Rainfall (mm)",
```

```
    hovermode="x unified"
)
fig.show()
```

```
First few rows of the dataset:
   _id District              Station  Rainfall                 Date
0    1  Chennai    Chennai port trust      10.0  1993-03-03T00:00:00
1    2  Chennai    Chennai port trust      10.0  1993-03-03T00:00:00
2    3  Chennai        Sholinganallur       2.0  1993-03-03T00:00:00
3    4  Chennai  Chennai nungambakkam       2.0  1993-03-03T00:00:00
4    5  Chennai        Sholinganallur      46.1  1993-05-16T00:00:00
```