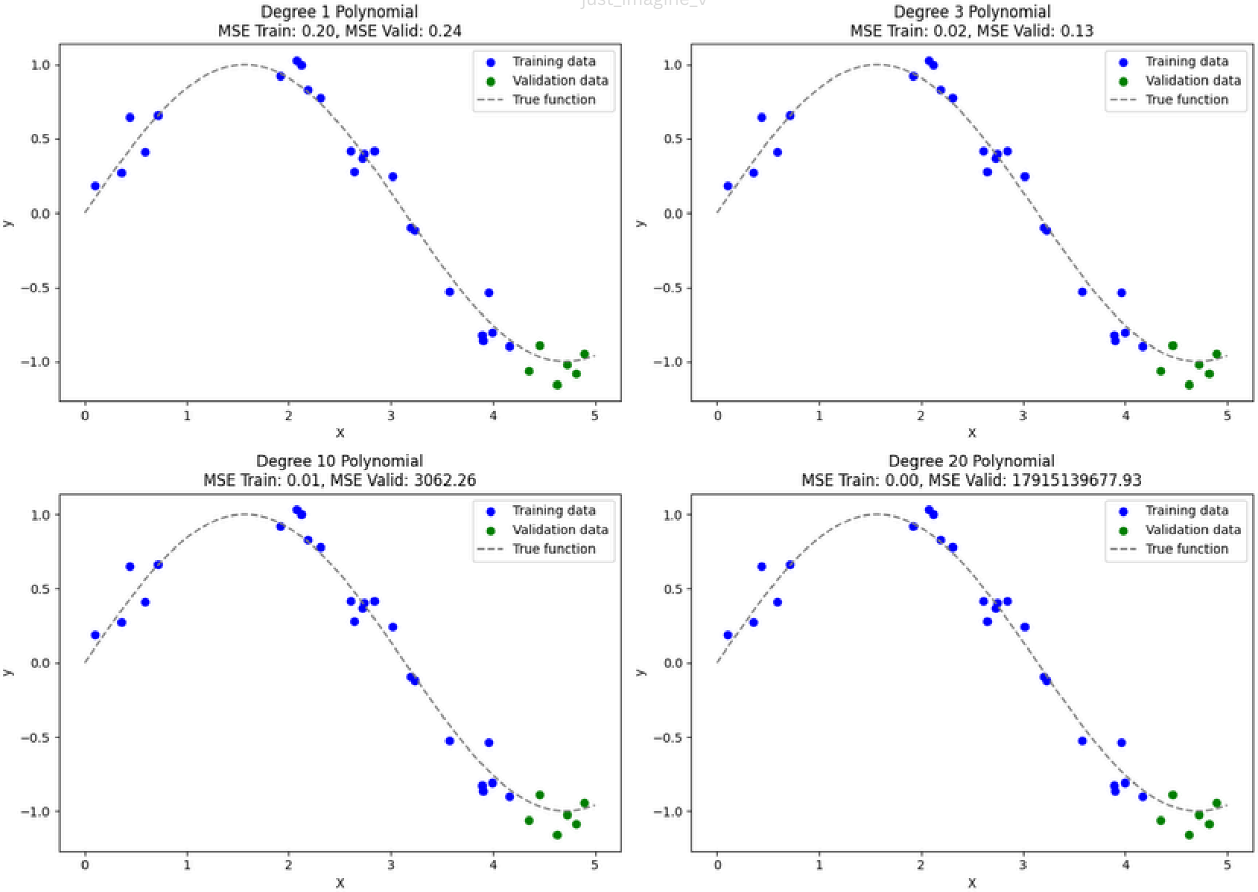
Practical 01

```python
import numpy as np
from sklearn.datasets import load_iris
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from sklearn.metrics import classification_report, confusion_matrix
import tensorflow as tf
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense

# Function to create and train the MLP model
def train_mlp(X_train, y_train):
    # Define the MLP model
    model = Sequential([
        Dense(10, activation='relu', input_shape=(4,)),
        Dense(10, activation='relu'),
        Dense(3, activation='softmax')
    ])

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

    # Train the model
    model.fit(X_train, y_train, epochs=50, batch_size=1, verbose=0)

    return model

# Function to predict using the trained model
def predict_species(model, input_data, scaler):
    # Standardize input data using the same scaler
    input_data = np.array(input_data).reshape(1, -1)  # Reshape input for single prediction
    input_data_scaled = scaler.transform(input_data)

    # Predict probabilities for each class
    probabilities = model.predict(input_data_scaled)

    # Determine the predicted class
    predicted_class = np.argmax(probabilities, axis=-1)[0]

    return predicted_class

# Load the Iris dataset
iris = load_iris()
X = iris.data
y = iris.target

# Splitting dataset 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)
# Standardize features by removing the mean and scaling to unit variance
scaler = StandardScaler()
X_train_scaled = scaler.fit_transform(X_train)
X_test_scaled = scaler.transform(X_test)
# Train the MLP model using all training data
model = train_mlp(X_train_scaled, y_train)
# Evaluate the model on test data
y_pred = np.argmax(model.predict(X_test_scaled), axis=-1)
print('Evaluation on Test Data:')
print('Confusion Matrix:')
print(confusion_matrix(y_test, y_pred))
print('\nClassification Report:')
print(classification_report(y_test, y_pred, target_names=iris.target_names))
```

```python
# Function to predict species based on user input
def predict_species_from_input(model, scaler):
    sepal_length = float(input("Enter sepal length in cm: "))
    sepal_width = float(input("Enter sepal width in cm: "))
    petal_length = float(input("Enter petal length in cm: "))
    petal_width = float(input("Enter petal width in cm: "))

    input_data = [sepal_length, sepal_width, petal_length, petal_width]
    predicted_class = predict_species(model, input_data, scaler)

    print(f"Predicted species: {iris.target_names[predicted_class]}")

# Predict species based on user input
predict_species_from_input(model, scaler)
```

```
Evaluation on Test Data:
Confusion Matrix:
[[10  0  0]
 [ 0  9  0]
 [ 0  0 11]]

Classification Report:
              precision    recall  f1-score   support

      setosa       1.00      1.00      1.00        10
  versicolor       1.00      1.00      1.00         9
   virginica       1.00      1.00      1.00        11

    accuracy                           1.00        30
   macro avg       1.00      1.00      1.00        30
weighted avg       1.00      1.00      1.00        30

Enter sepal length in cm: 1.2
Enter sepal width in cm: 2.5
Enter petal length in cm: 1.8
Enter petal width in cm: 3.9
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 55ms/step
Predicted species: virginica
```

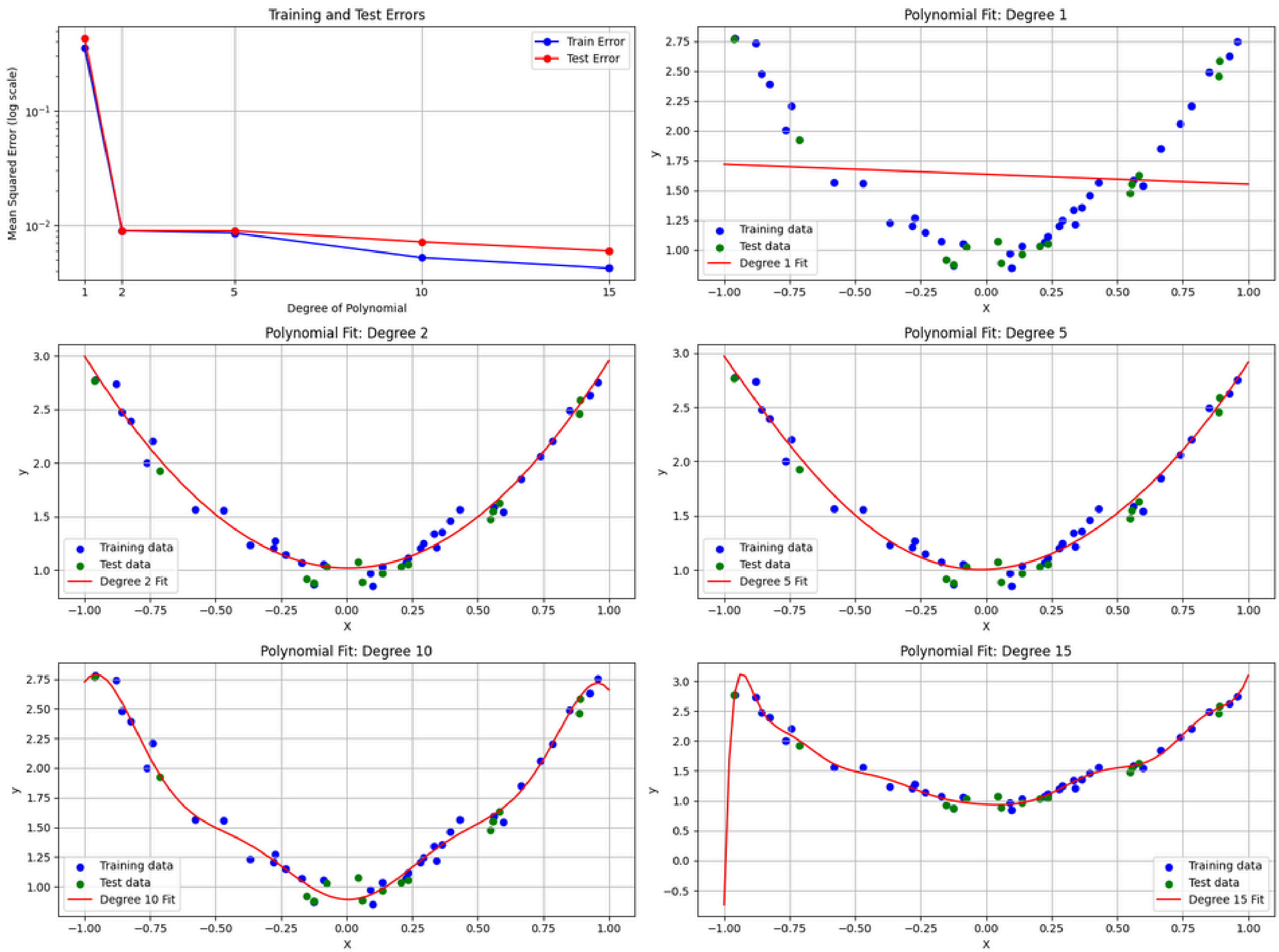



Practical 02

```python
import numpy as np
import matplotlib.pyplot as plt
from sklearn.preprocessing import PolynomialFeatures
from sklearn.linear_model import LinearRegression
from sklearn.metrics import mean_squared_error

# Function to generate synthetic data with a known relationship
def generate_data(n_samples):
    np.random.seed(0)
    X = np.random.rand(n_samples, 1) * 2 - 1  # Generate n_samples random numbers between -1 and 1
    y = 2 * X.flatten()**2 + 1 + np.random.randn(n_samples) * 0.1  # True relationship y = 2X^2 + 1 + noise
    return X, y

# Generate synthetic data
X, y = generate_data(n_samples=50)

# Split data into training and test sets
from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3, random_state=0)

# Function to train polynomial regression models of varying degrees
def polynomial_regression(X_train, y_train, X_test, y_test, degrees):
    train_errors = []
    test_errors = []
    models = []
    fitted_curves = []

    for d in degrees:
        # Transform input data to polynomial features of degree d
        poly_features = PolynomialFeatures(degree=d)
        X_train_poly = poly_features.fit_transform(X_train)
        X_test_poly = poly_features.transform(X_test)

        # Fit a linear regression model
        model = LinearRegression()
        model.fit(X_train_poly, y_train)

        # Predictions
        y_train_pred = model.predict(X_train_poly)
        y_test_pred = model.predict(X_test_poly)

        # Calculate MSE
        train_error = mean_squared_error(y_train, y_train_pred)
        test_error = mean_squared_error(y_test, y_test_pred)

        train_errors.append(train_error)
        test_errors.append(test_error)
        models.append(model)

        # Generate points for plotting the fitted curve
        X_range = np.linspace(-1, 1, 100).reshape(-1, 1)
        X_range_poly = poly_features.transform(X_range)
        y_range_pred = model.predict(X_range_poly)
        fitted_curves.append((X_range, y_range_pred))

    return train_errors, test_errors, models, fitted_curves
```

```python
# Degrees of polynomial features to test
degrees = [1, 2, 5, 10, 15]

# Train polynomial regression models
train_errors, test_errors, models, fitted_curves = polynomial_regression(X_train, y_train, X_test, y_test, degrees)

# Plotting the results
plt.figure(figsize=(16, 12))
```

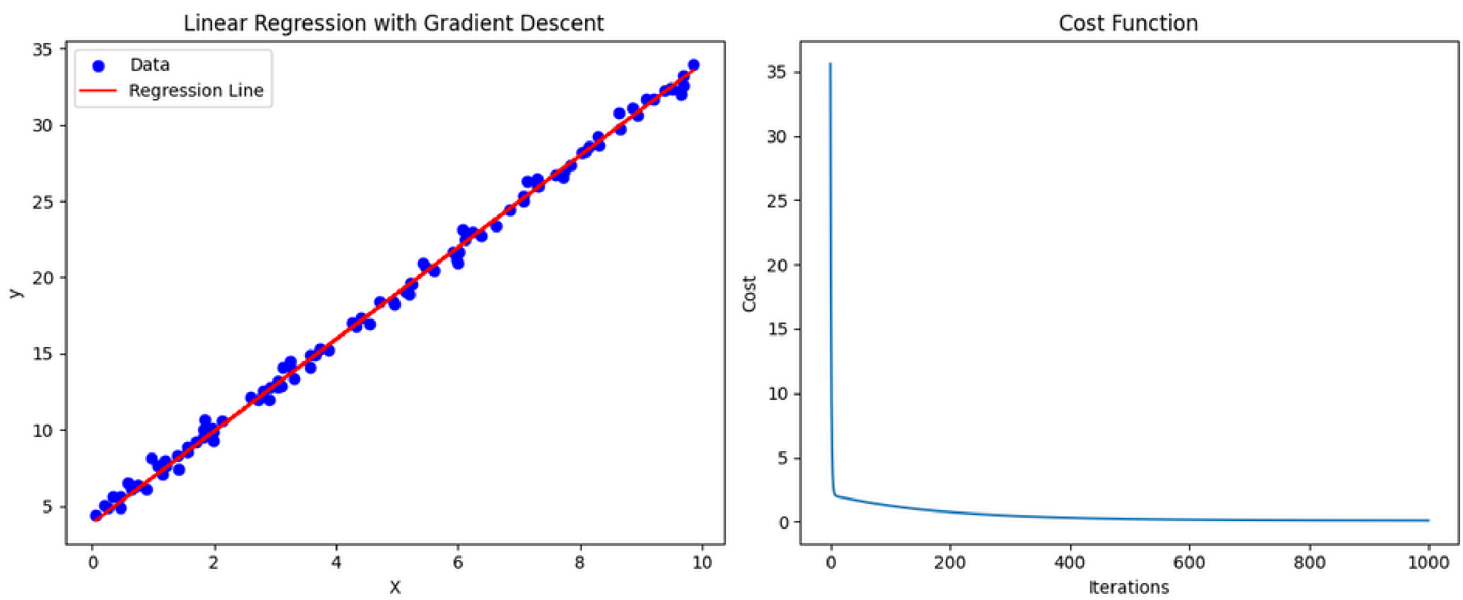
```python
# Plotting the results
plt.figure(figsize=(16, 12))

# Plot training and test errors
plt.subplot(3, 2, 1)
plt.plot(degrees, train_errors, marker='o', label='Train Error', color='blue')
plt.plot(degrees, test_errors, marker='o', label='Test Error', color='red')
plt.yscale('log')  # Log scale for better visualization of errors
plt.title('Training and Test Errors')
plt.xlabel('Degree of Polynomial')
plt.ylabel('Mean Squared Error (log scale)')
plt.xticks(degrees)
plt.legend()
plt.grid(True)

# Plotting the fitted curves
for i, degree in enumerate(degrees):
    plt.subplot(3, 2, i+2)
    plt.scatter(X_train, y_train, color='blue', label='Training data')
    plt.scatter(X_test, y_test, color='green', label='Test data')
    plt.plot(fitted_curves[i][0], fitted_curves[i][1], color='red', label=f'Degree {degree} Fit')
    plt.title(f'Polynomial Fit: Degree {degree}')
    plt.xlabel('X')
    plt.ylabel('y')
    plt.legend()
    plt.grid(True)

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

Practical 03

```
import numpy as np
import matplotlib.pyplot as plt

# Generate synthetic data
def generate_data(num_samples=100, noise_std=0.5):
    np.random.seed(42)
    X = np.random.rand(num_samples, 1) * 10
    y = 4 + 3 * X + np.random.randn(num_samples, 1) * noise_std
    return X, y

# Compute cost (Mean Squared Error)
def compute_cost(X, y, theta):
    m = len(y)
    predictions = X.dot(theta)
    cost = (1 / (2 * m)) * np.sum((predictions - y) ** 2)
    return cost

# Perform gradient descent
def gradient_descent(X, y, theta, alpha, num_iters):
    m = len(y)
    J_history = []

    for _ in range(num_iters):
        predictions = X.dot(theta)
        theta = theta - (alpha / m) * X.T.dot(predictions - y)
        J_history.append(compute_cost(X, y, theta))

    return theta, J_history

# Main function
def main():
    # Generate synthetic data
    X, y = generate_data()

    # Add intercept term to X
    X_b = np.c_[np.ones((len(X), 1)), X]

    # Initialize parameters
    theta = np.random.randn(2, 1)

    # Set hyperparameters
    alpha = 0.01
    num_iters = 1000

    # Run gradient descent
    theta_optimal, J_history = gradient_descent(X_b, y, theta, alpha, num_iters)

    # Print results
    print("Optimal theta:")
```

```python
    # Print results
    print("Optimal theta:")
    print(f"theta_0 (intercept): {theta_optimal[0][0]:.4f}")
    print(f"theta_1 (slope): {theta_optimal[1][0]:.4f}")

    # Plot results
    plt.figure(figsize=(12, 5))

    # Plot data and regression line
    plt.subplot(121)
    plt.scatter(X, y, color='b', label='Data')
    plt.plot(X, X_b.dot(theta_optimal), color='r', label='Regression Line')
    plt.xlabel('X')
    plt.ylabel('y')
    plt.legend()
    plt.title('Linear Regression with Gradient Descent')

    # Plot cost function
    plt.subplot(122)
    plt.plot(range(num_iters), J_history)
    plt.xlabel('Iterations')
    plt.ylabel('Cost')
    plt.title('Cost Function')

    plt.tight_layout()
    plt.show()

if __name__ == "__main__":
    main()
```

Practical 04

```python
# Import necessary libraries
import tensorflow as tf
from tensorflow.keras.datasets import mnist
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, Flatten
from tensorflow.keras.utils import to_categorical
from tensorflow.keras.callbacks import EarlyStopping

# Load and preprocess the MNIST dataset
(x_train, y_train), (x_test, y_test) = mnist.load_data()

# Normalize the images to a range of 0 to 1
x_train = x_train.astype('float32') / 255.0
x_test = x_test.astype('float32') / 255.0

# Convert class vectors to binary class matrices
y_train = to_categorical(y_train, 10)
y_test = to_categorical(y_test, 10)

# Define the model
model = Sequential([
    Flatten(input_shape=(28, 28)),      # Flatten the 28x28 images into a 1D vector
    Dense(128, activation='relu'),      # First hidden layer with 128 neurons and ReLU activation
    Dense(64, activation='relu'),       # Second hidden layer with 64 neurons and ReLU activation
    Dense(10, activation='softmax')     # Output layer with 10 neurons (one for each digit) and softmax activation
])

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

# Define early stopping to prevent overfitting
early_stopping = EarlyStopping(monitor='val_loss', patience=3, restore_best_weights=True)

# Train the model
history = model.fit(x_train, y_train,
                    epochs=20,
                    batch_size=32,
                    validation_split=0.2,
                    callbacks=[early_stopping],
                    verbose=2)

# Evaluate the model
test_loss, test_acc = model.evaluate(x_test, y_test, verbose=2)
print(f'\nTest accuracy: {test_acc:.4f}')

# Optionally, plot training history
import matplotlib.pyplot as plt
```

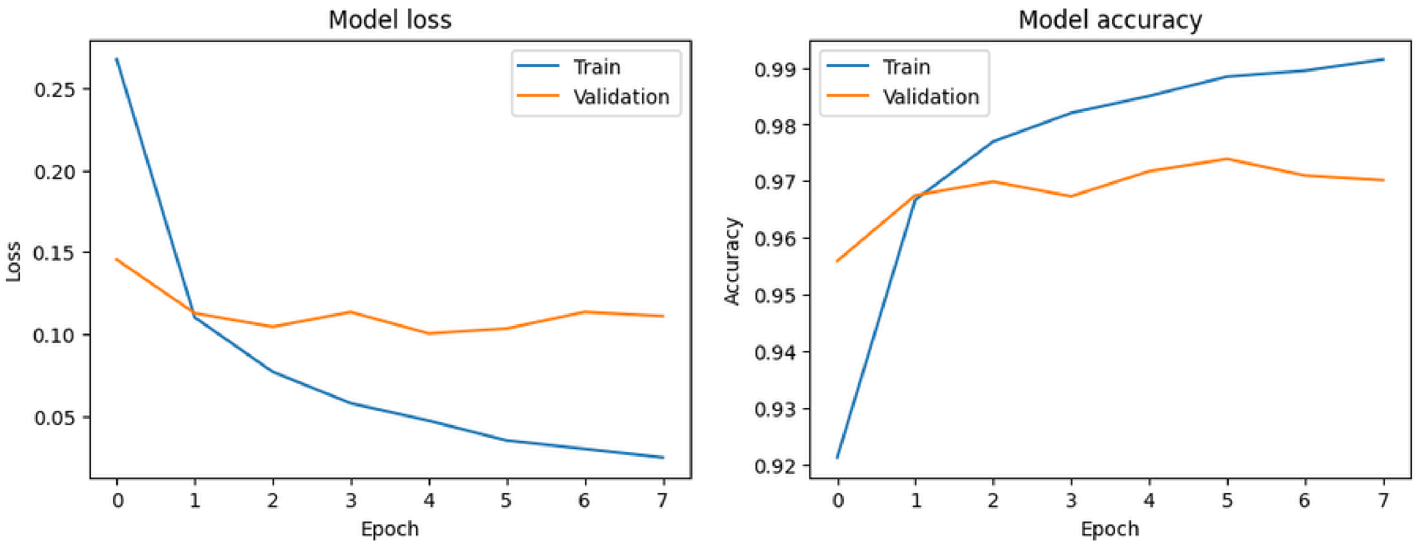


```python
plt.figure(figsize=(12, 4))

# Plot training & validation loss values
plt.subplot(1, 2, 1)
plt.plot(history.history['loss'])
plt.plot(history.history['val_loss'])
plt.title('Model loss')
plt.xlabel('Epoch')
plt.ylabel('Loss')
plt.legend(['Train', 'Validation'])
```

```python
# Plot training & validation accuracy values
plt.subplot(1, 2, 2)
plt.plot(history.history['accuracy'])
plt.plot(history.history['val_accuracy'])
plt.title('Model accuracy')
plt.xlabel('Epoch')
plt.ylabel('Accuracy')
plt.legend(['Train', 'Validation'])

plt.show()
```



```
1500/1500 - 5s - 3ms/step - accuracy: 0.9960 - loss: 0.0123 - val_accuracy: 0.9730 - val_loss: 0.1444
Epoch 19/20
1500/1500 - 5s - 3ms/step - accuracy: 0.9981 - loss: 0.0062 - val_accuracy: 0.9743 - val_loss: 0.1532
Epoch 20/20
1500/1500 - 6s - 4ms/step - accuracy: 0.9967 - loss: 0.0095 - val_accuracy: 0.9741 - val_loss: 0.1510
313/313 - 1s - 2ms/step - accuracy: 0.9757 - loss: 0.1399

Test accuracy: 0.9757
```

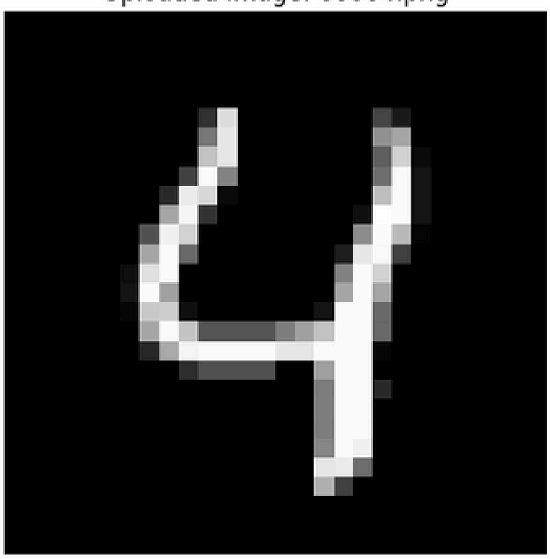


Choose Files 60004.png

- **60004.png**(image/png) - 275 bytes, last modified: 27/8/2024 - 100% done

```
Saving 60004.png to 60004.png
```

Uploaded Image: 60004.png



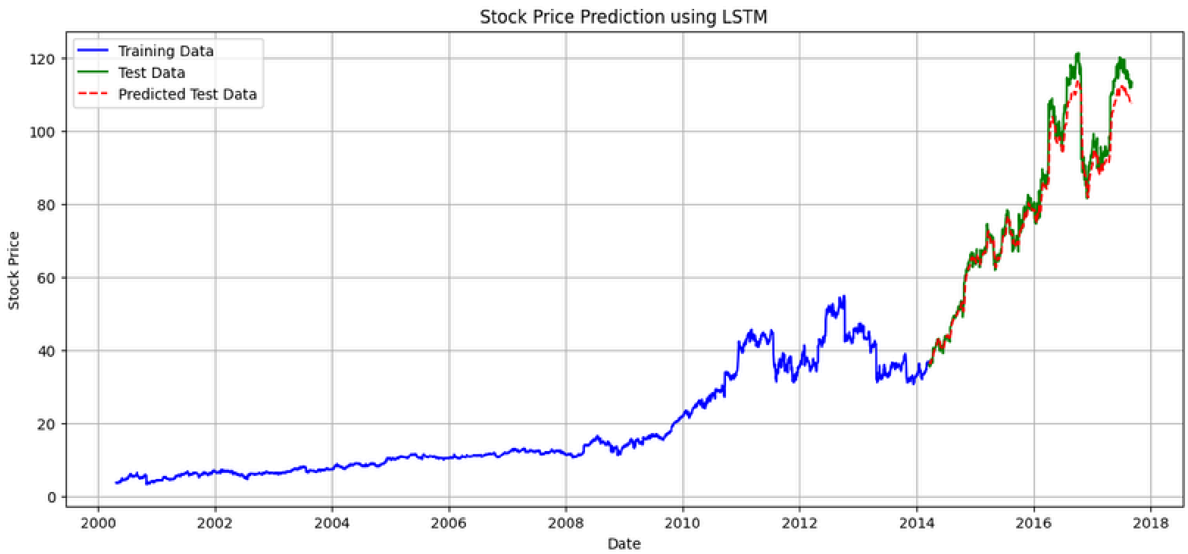
```
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 23ms/step
Predicted digit: 4
```

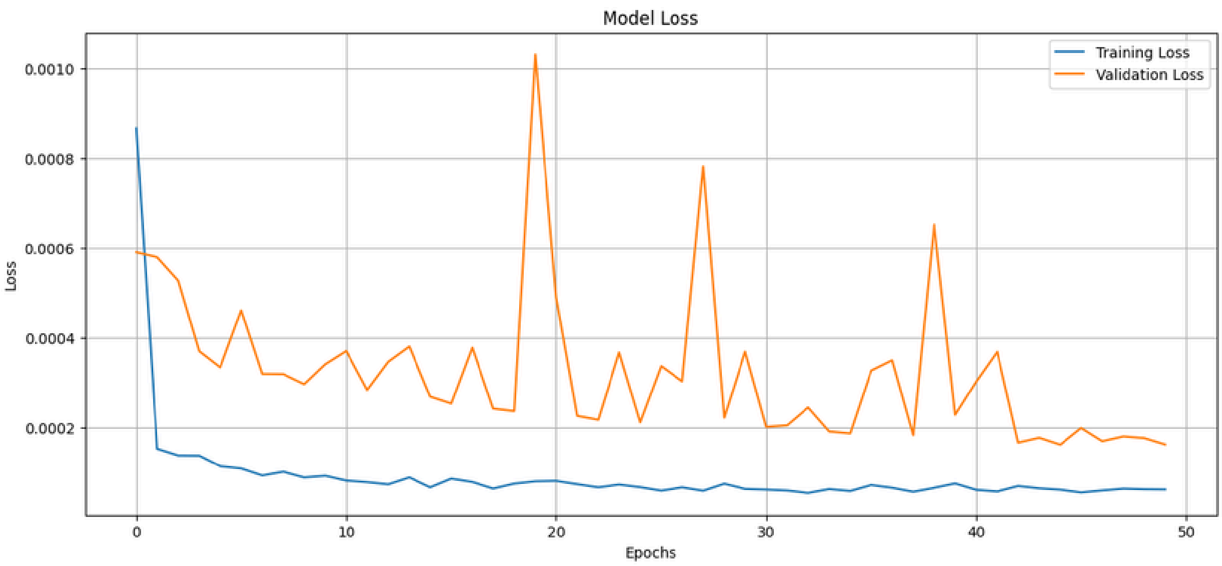
Practical 05

```python
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from sklearn.preprocessing import MinMaxScaler
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import LSTM, Dense, Dropout

# Step 1: Load the data
data = pd.read_csv('/content/EW-MAX.csv', parse_dates=['Date'])
data = data[['Date', 'Close']]  # Selecting Date and Close price columns

# Step 2: Preprocess the data
# Scale the data
scaler = MinMaxScaler(feature_range=(0, 1))
scaled_data = scaler.fit_transform(data['Close'].values.reshape(-1, 1))

# Step 3: Prepare the data for LSTM
def create_dataset(data, time_step=20):
    X, y = [], []
    for i in range(len(data) - time_step):
        X.append(data[i:(i + time_step), 0])
        y.append(data[i + time_step, 0])
    return np.array(X), np.array(y)

# Using 20 time steps as per requirement
time_step = 20
X, y = create_dataset(scaled_data, time_step)

# Reshape data to be [samples, time steps, features]
X = X.reshape((X.shape[0], X.shape[1], 1))

# Split data into training and test sets
train_size = int(len(X) * 0.8)
X_train, X_test = X[:train_size], X[train_size:]
y_train, y_test = y[:train_size], y[train_size:]

# Step 4: Build the LSTM model
model = Sequential()
model.add(LSTM(units=50, return_sequences=True, input_shape=(X_train.shape[1], 1)))
model.add(Dropout(0.2))
model.add(LSTM(units=50, return_sequences=False))
model.add(Dropout(0.2))
model.add(Dense(units=1))
```

```
Epoch 49/50
88/88 ━━━━━━━━━━━━━━━━━━━━ 3s 28ms/step - loss: 6.3925e-05 - val_loss: 1.7549e-04
Epoch 50/50
88/88 ━━━━━━━━━━━━━━━━━━━━ 2s 19ms/step - loss: 5.7022e-05 - val_loss: 1.6098e-04
110/110 ━━━━━━━━━━━━━━━━━━━━ 1s 8ms/step
28/28 ━━━━━━━━━━━━━━━━━━━━ 0s 6ms/step
```

```python
# Compile the model
model.compile(optimizer='adam', loss='mean_squared_error')

# Step 5: Train the model
history = model.fit(X_train, y_train, epochs=50, batch_size=32, validation_split=0.2)

# Step 6: Make predictions
train_predictions = model.predict(X_train)
test_predictions = model.predict(X_test)

# Inverse transform the predictions and actual values
train_predictions = scaler.inverse_transform(train_predictions)
test_predictions = scaler.inverse_transform(test_predictions)
y_train_actual = scaler.inverse_transform(y_train.reshape(-1, 1))
y_test_actual = scaler.inverse_transform(y_test.reshape(-1, 1))

# Plotting the results
plt.figure(figsize=(14, 6))

# Plotting the training data predictions
plt.plot(data['Date'][time_step:train_size + time_step], y_train_actual, color='blue', label='Training Data')

# Plotting the test data and predictions
plt.plot(data['Date'][train_size + time_step:], y_test_actual, color='green', label='Test Data')
plt.plot(data['Date'][train_size + time_step:], test_predictions, color='red', linestyle='--', label='Predicted Test Data')

plt.xlabel('Date')
plt.ylabel('Stock Price')
plt.title('Stock Price Prediction using LSTM')
plt.legend()
plt.grid()
plt.show()
```

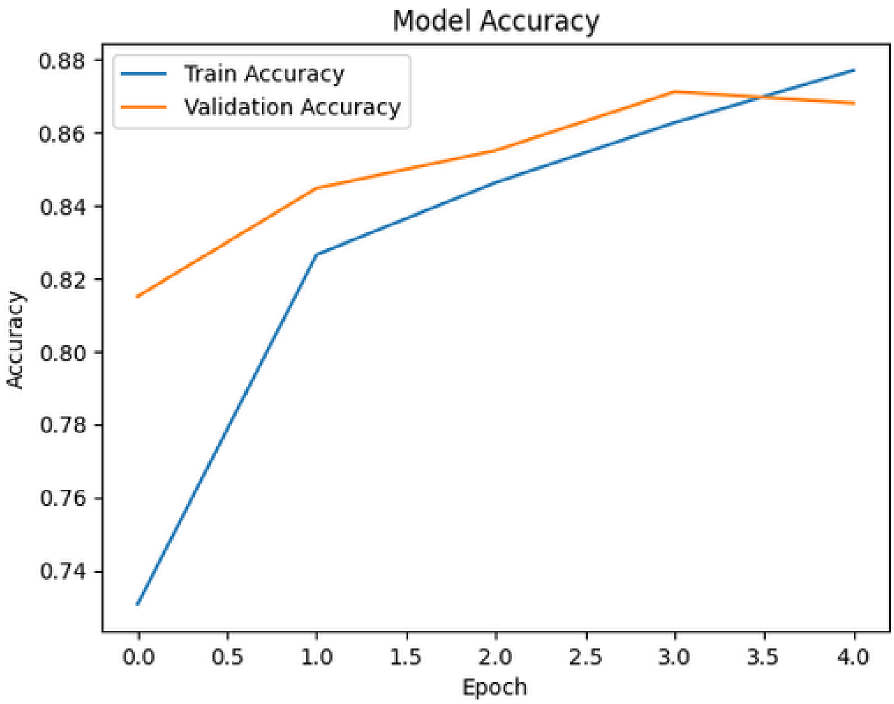


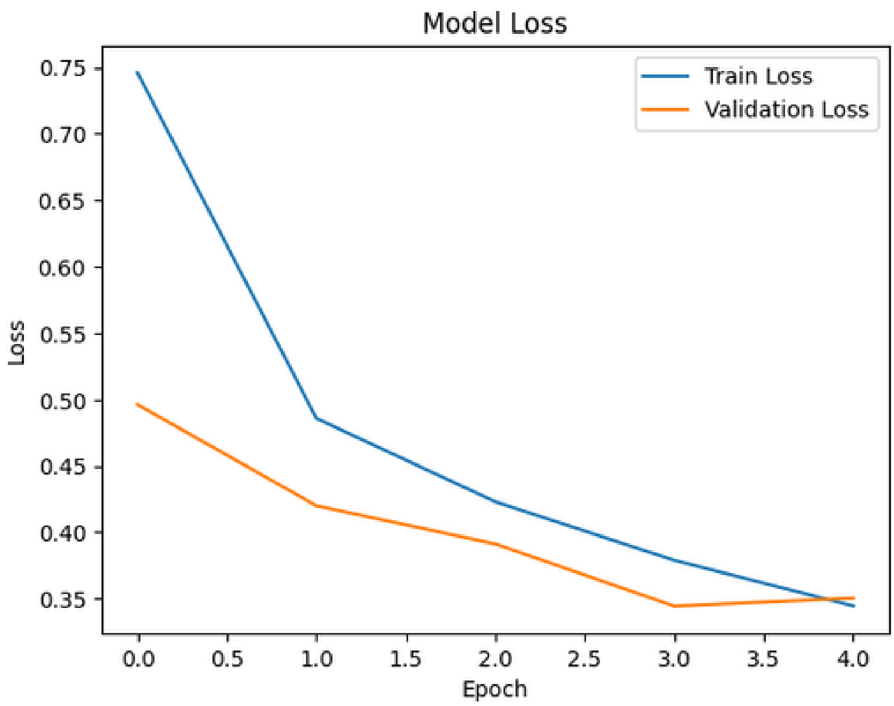


Practical 06

```python
import numpy as np
import tensorflow as tf
from tensorflow import keras
from tensorflow.keras import layers
from tensorflow.keras.datasets import fashion_mnist
import matplotlib.pyplot as plt
# Load and preprocess the Fashion MNIST dataset
(x_train, y_train), (x_test, y_test) = fashion_mnist.load_data()
# Normalize the images to [0, 1]
x_train = x_train.astype('float32') / 255.0
x_test = x_test.astype('float32') / 255.0
# Reshape the data to add a channel dimension
x_train = np.expand_dims(x_train, axis=-1)
x_test = np.expand_dims(x_test, axis=-1)
# One-hot encode the labels
y_train = keras.utils.to_categorical(y_train, num_classes=10)
y_test = keras.utils.to_categorical(y_test, num_classes=10)
# Define the CNN model
model = keras.Sequential([
    layers.Conv2D(32, (3, 3), activation='relu', input_shape=(28, 28, 1)),
    layers.MaxPooling2D((2, 2)),
    layers.Conv2D(64, (3, 3), activation='relu'),
    layers.MaxPooling2D((2, 2)),
    layers.Conv2D(128, (3, 3), activation='relu'),
    layers.MaxPooling2D((2, 2)),
    layers.Flatten(),
    layers.Dense(128, activation='relu'),
    layers.Dropout(0.5),  # Dropout for regularization
    layers.Dense(10, activation='softmax')  # 10 classes for Fashion MNIST
])
# Compile the model
model.compile(optimizer='adam',
              loss='categorical_crossentropy',
              metrics=['accuracy'])
# Train the model
history = model.fit(x_train, y_train, epochs=5, batch_size=64, validation_split=0.2)
# Evaluate the model
test_loss, test_accuracy = model.evaluate(x_test, y_test)
print(f'Test accuracy: {test_accuracy:.4f}')
# Plot training & validation accuracy values
plt.plot(history.history['accuracy'], label='Train Accuracy')
plt.plot(history.history['val_accuracy'], label='Validation Accuracy')
plt.title('Model Accuracy')
plt.ylabel('Accuracy')
plt.xlabel('Epoch')
plt.legend()
plt.show()
# Plot training & validation loss values
plt.plot(history.history['loss'], label='Train Loss')
plt.plot(history.history['val_loss'], label='Validation Loss')
plt.title('Model Loss')
plt.ylabel('Loss')
plt.xlabel('Epoch')
plt.legend()
plt.show()
```

```
Epoch 1/5
750/750 ━━━━━━━━━━━━━━━━━━━━ 61s 71ms/step - accuracy: 0.6220 - loss: 1.0426 - val_accuracy: 0.8151 - val_loss: 0.4962
Epoch 2/5
750/750 ━━━━━━━━━━━━━━━━━━━━ 80s 69ms/step - accuracy: 0.8216 - loss: 0.5045 - val_accuracy: 0.8447 - val_loss: 0.4201
Epoch 3/5
750/750 ━━━━━━━━━━━━━━━━━━━━ 80s 67ms/step - accuracy: 0.8457 - loss: 0.4233 - val_accuracy: 0.8551 - val_loss: 0.3913
Epoch 4/5
534/750 ━━━━━━━━━━━━━━━━━━━━ 13s 63ms/step - accuracy: 0.8602 - loss: 0.3870
```



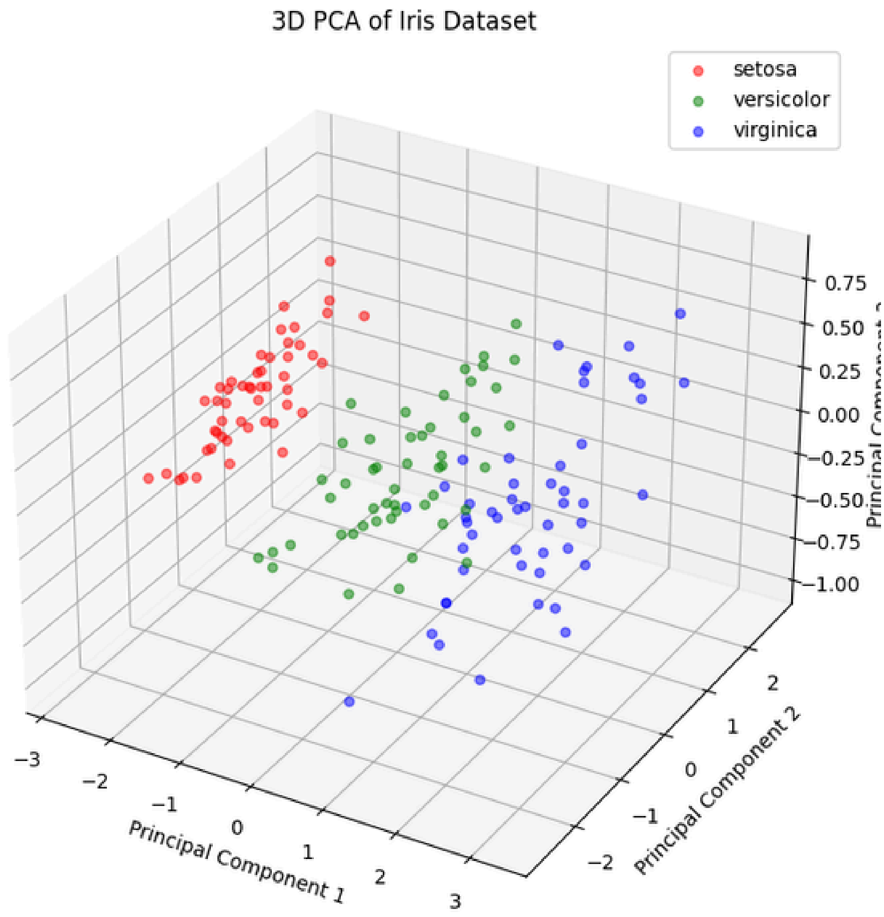


Practical 07

```python
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from sklearn.datasets import load_iris
from sklearn.decomposition import PCA
from sklearn.preprocessing import StandardScaler

# Load the Iris dataset
iris = load_iris()
X = iris.data
y = iris.target
feature_names = iris.feature_names
target_names = iris.target_names

# Standardize the data
scaler = StandardScaler()
X_scaled = scaler.fit_transform(X)

# Apply PCA
pca = PCA()  # Fit PCA without specifying components to analyze all
X_pca = pca.fit_transform(X_scaled)

# Explained variance
explained_variance = pca.explained_variance_ratio_
cumulative_variance = np.cumsum(explained_variance)

# Create a DataFrame for easier plotting
df_pca = pd.DataFrame(data=X_pca, columns=[f'Principal Component {i+1}' for i in range(X.shape[1])])
df_pca['Target'] = y

# Plotting the results for the first two principal components
plt.figure(figsize=(8, 6))
colors = ['r', 'g', 'b']
for target, color in zip(range(len(target_names)), colors):
    plt.scatter(df_pca[df_pca['Target'] == target]['Principal Component 1'],
                df_pca[df_pca['Target'] == target]['Principal Component 2'],
                color=color, alpha=0.5, label=target_names[target])

plt.title('PCA of Iris Dataset')
plt.xlabel('Principal Component 1')
plt.ylabel('Principal Component 2')
plt.legend()
plt.grid()
plt.show()
```

```python
# Scree plot to show explained variance by each principal component
plt.figure(figsize=(8, 6))
plt.plot(range(1, len(explained_variance) + 1), explained_variance, marker='o', linestyle='--')
plt.title('Scree Plot')
plt.xlabel('Principal Component')
plt.ylabel('Variance Explained')
plt.xticks(range(1, len(explained_variance) + 1))
plt.grid()
plt.show()
```

PCA of Iris Dataset
Principal Component 1
Principal Component 2
setosa
versicolor
virginica



3D PCA of Iris Dataset
setosa
versicolor
virginica
Principal Component 1
Principal Component 2

Practical 08

```python
import numpy as np
import matplotlib.pyplot as plt
from tensorflow.keras.layers import Input, Dense
from tensorflow.keras.models import Model
from tensorflow.keras.datasets import mnist

# Step 1: Load and preprocess the dataset
# Example with MNIST dataset
(X_train, _), (X_test, _) = mnist.load_data()

# Normalize pixel values to [0, 1]
X_train = X_train.astype('float32') / 255.
X_test = X_test.astype('float32') / 255.

# Flatten images into 784-dimensional vectors (28x28)
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:])))

# Step 2: Build the autoencoder model
input_dim = X_train.shape[1]
encoding_dim = 32  # Dimension of the encoded representation

# Encoder
input_img = Input(shape=(input_dim,))
encoded = Dense(encoding_dim, activation='relu')(input_img)

# Decoder
decoded = Dense(input_dim, activation='sigmoid')(encoded)

# Autoencoder model
autoencoder = Model(input_img, decoded)

# Compile the autoencoder model
autoencoder.compile(optimizer='adam', loss='mse')

# Step 3: Train the autoencoder
epochs = 20
batch_size = 128
history = autoencoder.fit(X_train, X_train,
                          epochs=epochs,
                          batch_size=batch_size,
                          shuffle=True,
                          validation_data=(X_test, X_test))
```

```
# Step 4: Evaluate the autoencoder
# Evaluate on test data
mse = autoencoder.evaluate(X_test, X_test, verbose=0)
print(f'Reconstruction error (MSE) on test set: {mse}')

# Step 5: Visualize original vs. reconstructed images
decoded_imgs = autoencoder.predict(X_test)

n = 10  # Number of digits to display
plt.figure(figsize=(20, 4))
for i in range(n):
    # Original images
    ax = plt.subplot(2, n, i + 1)
    plt.imshow(X_test[i].reshape(28, 28), cmap='gray')
    plt.title('Original')
    plt.axis('off')

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

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

```
Downloading data from https://storage.googleapis.com/tensorflow/tf-keras-datasets/mnist.npz
11490434/11490434 ━━━━━━━━━━━━━━━━━━━━ 0s 0us/step
Epoch 1/20
469/469 ━━━━━━━━━━━━━━━━━━━━ 4s 6ms/step - loss: 0.0854 - val_loss: 0.0305
Epoch 2/20
469/469 ━━━━━━━━━━━━━━━━━━━━ 2s 5ms/step - loss: 0.0277 - val_loss: 0.0206
Epoch 3/20
```