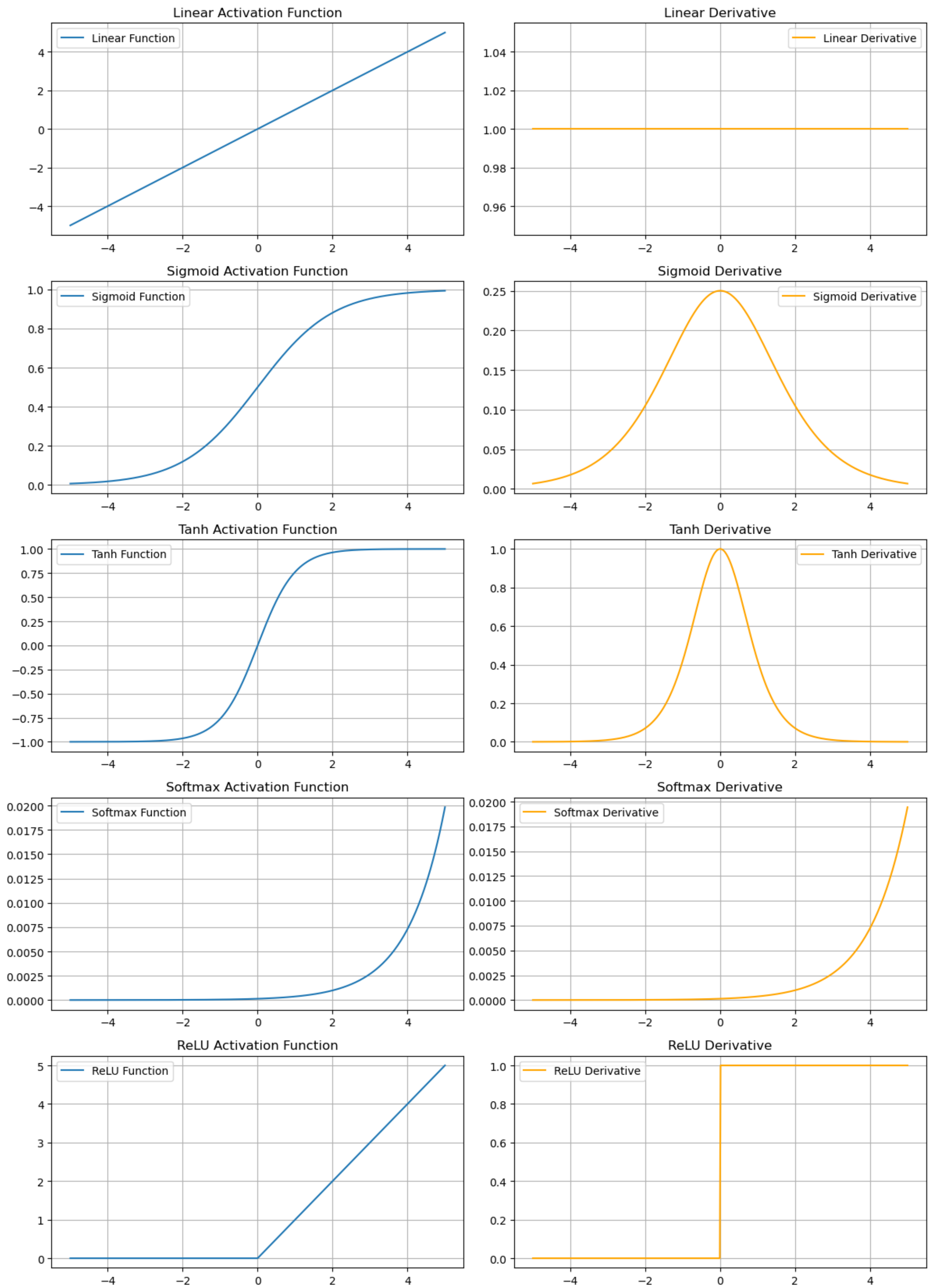
Assignment 1

All the activation functions and their derivatives

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

# Define activation functions and their derivatives
def linear(x):
    return x

def linear_derivative(x):
    return np.ones_like(x)

def sigmoid(x):
    return 1 / (1 + np.exp(-x))

def sigmoid_derivative(x):
    s = sigmoid(x)
    return s * (1 - s)

def tanh(x):
    return np.tanh(x)

def tanh_derivative(x):
    return 1 - np.tanh(x)**2

def softmax(x):
    e_x = np.exp(x - np.max(x))  # For numerical stability
    return e_x / e_x.sum(axis=0)

def softmax_derivative(x):
    s = softmax(x)
    return s * (1 - s)  # Simplified for 1D case, not a full Jacobian

def relu(x):
    return np.maximum(0, x)

def relu_derivative(x):
    return np.where(x > 0, 1, 0)

def leaky_relu(x, alpha=0.01):
    return np.where(x > 0, x, alpha * x)

def leaky_relu_derivative(x, alpha=0.01):
    return np.where(x > 0, 1, alpha)

# Generate input range
x = np.linspace(-5, 5, 500)
```

```python
# Prepare for plotting
activations = {
    "Linear": (linear, linear_derivative),
    "Sigmoid": (sigmoid, sigmoid_derivative),
    "Tanh": (tanh, tanh_derivative),
    "Softmax":(softmax, softmax_derivative),
    "ReLU": (relu, relu_derivative),
    "Leaky ReLU": (leaky_relu, leaky_relu_derivative)
}

# Plot activation functions and their derivatives
fig, axes = plt.subplots(len(activations), 2, figsize=(12, 20))
for i, (name, (func, deriv)) in enumerate(activations.items()):
    # Compute function and derivative values
    y = func(x)
    dydx = deriv(x)

    # Plot activation function
    axes[i, 0].plot(x, y, label=f"{name} Function")
    axes[i, 0].set_title(f"{name} Activation Function")
    axes[i, 0].legend()
    axes[i, 0].grid(True)

    # Plot derivative
    axes[i, 1].plot(x, dydx, label=f"{name} Derivative",
color='orange')
    axes[i, 1].set_title(f"{name} Derivative")
    axes[i, 1].legend()
    axes[i, 1].grid(True)



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


Linear Activation Function
Linear Function
Linear Derivative
Linear Derivative
Sigmoid Activation Function
Sigmoid Function
Sigmoid Derivative
Sigmoid Derivative
Tanh Activation Function
Tanh Function
Tanh Derivative
Tanh Derivative
Softmax Activation Function
Softmax Function
Softmax Derivative
Softmax Derivative
ReLU Activation Function
ReLU Function
ReLU Derivative
ReLU Derivative

```python
import pandas as pd
import numpy as np

# Set random seed for reproducibility
np.random.seed(42)

# Number of samples
num_samples = 1000

soil_texture = np.random.uniform(0, 1, num_samples)  # Ratio of sand, silt, and clay
pH = np.random.uniform(4.5, 9.0, num_samples)  # Soil pH levels
moisture_content = np.random.uniform(5, 50, num_samples)  # Percentage moisture
organic_matter = np.random.uniform(1, 10, num_samples)  # Organic matter in %
bulk_density = np.random.uniform(1.1, 1.6, num_samples)  # Soil bulk density in g/cm^3


soil_types = []
for i in range(num_samples):
    if soil_texture[i] > 0.7 and pH[i] > 6.5 and moisture_content[i] < 15:
        soil_types.append("Sandy")
    elif organic_matter[i] > 5 and moisture_content[i] > 30:
        soil_types.append("Loamy")
    else:
        soil_types.append("Clayey")

# Create a DataFrame
data = pd.DataFrame({
    "Soil_Texture": soil_texture,
    "pH": pH,
    "Moisture_Content": moisture_content,
    "Organic_Matter": organic_matter,
    "Bulk_Density": bulk_density,
    "Soil_Type": soil_types
})

# Save the dataset to a CSV file
data.to_csv("synthetic_soil_dataset.csv", index=False)

print("Synthetic dataset created and saved as 'synthetic_soil_dataset.csv'.")
```

```
Synthetic dataset created and saved as 'synthetic_soil_dataset.csv'.
```

Data Preprocessing:

```python
import pandas as pd
import numpy as np
from sklearn.impute import SimpleImputer
from sklearn.preprocessing import StandardScaler, LabelEncoder

# Load the synthetic soil dataset
data = pd.read_csv("synthetic_soil_dataset.csv")

# Introduce some missing values for demonstration purposes
data.iloc[10:15, 2] = np.nan  # Simulate missing values in
'Moisture_Content'

# 1. Handle Missing Data
imputer = SimpleImputer(strategy="mean")  # Replace missing values
with the column mean
data.iloc[:, :-1] = imputer.fit_transform(data.iloc[:, :-1])  # Handle
numerical columns

# 2. Separate Features and Labels
X = data.iloc[:, :-1].values  # Features (inputs)
y = data.iloc[:, -1].values   # Labels (outputs)

# 3. Encode Categorical Labels (Simple Label Encoding)
label_encoder = LabelEncoder()
y_encoded = label_encoder.fit_transform(y)

# 4. Scale the Features (Standardization)
scaler = StandardScaler()
X_scaled = scaler.fit_transform(X)

# Output Preprocessed Data
print("Features after scaling:\n", X_scaled[:5])
print("Encoded Labels:\n", y_encoded[:5])

# Optional: Save the preprocessed data
preprocessed_data = pd.DataFrame(X_scaled, columns=["Soil_Texture",
"pH", "Moisture_Content", "Organic_Matter", "Bulk_Density"])
preprocessed_data["Soil_Type"] = y_encoded
preprocessed_data.to_csv("preprocessed_soil_dataset.csv", index=False)

print("\nPreprocessed data saved as 'preprocessed_soil_dataset.csv'.")
```

```
Features after scaling:
 [[-0.39630103 -1.10217857 -0.83490537  0.63673984  0.2717114 ]
 [ 1.57695733  0.11944663 -0.8858098   1.06970686  1.08602506]
 [ 0.82789256  1.25299211  1.39301993 -0.83782241  0.92810149]
 [ 0.37125061  0.77114327 -0.87693542  0.46970805 -1.18676377]
 [-1.14468466  1.02568136 -0.79949617  0.28416991 -1.20298622]]
Encoded Labels:
 [0 0 0 0 0]
```

```
Preprocessed data saved as 'preprocessed_soil_dataset.csv'.
```

```
datset = pd.read_csv('preprocessed_soil_dataset.csv')
```

2.Train-Test Split:

```
from sklearn.model_selection import train_test_split

# Perform Train-Test Split
X_train, X_test, y_train, y_test = train_test_split(
    X_scaled, y_encoded, test_size=0.2, random_state=42,
stratify=y_encoded
)

# Output the shapes of the splits
print(f"Training Features Shape: {X_train.shape}")
print(f"Test Features Shape: {X_test.shape}")
print(f"Training Labels Shape: {y_train.shape}")
print(f"Test Labels Shape: {y_test.shape}")
```

```
Training Features Shape: (800, 5)
Test Features Shape: (200, 5)
Training Labels Shape: (800,)
Test Labels Shape: (200,)
```

```
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, Dropout

# Define the Sequential model
model = Sequential([
    # First hidden layer with 64 neurons and ReLU activation
    Dense(64, activation='relu', input_shape=(X_train.shape[1],)),
    Dropout(0.3),  # Dropout with a rate of 30%

    # Second hidden layer with 32 neurons and ReLU activation
    Dense(32, activation='relu'),
    Dropout(0.3),  # Dropout with a rate of 30%

    # Output layer for classification
    Dense(len(set(y_encoded)), activation='softmax')  # Adjust output
neurons based on the number of classes
])

# Print model summary
model.summary()
```

```
C:\Users\HP\AppData\Roaming\Python\Python312\site-packages\keras\src\
layers\core\dense.py:87: UserWarning: Do not pass an
`input_shape`/`input_dim` argument to a layer. When using Sequential
```

```
models, prefer using an `Input(shape)` object as the first layer in
the model instead.
  super().__init__(activity_regularizer=activity_regularizer,
**kwargs)

Model: "sequential"

┏━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━┳━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━┳
━━━━━━━━━━━━━━━━━┓
┃ Layer (type)                          ┃ Output Shape                 ┃
Param # ┃
┡━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━╇━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━╇
━━━━━━━━━━━━━━━━━┩
│ dense (Dense)                         │ (None, 64)                   │
384 │
├───────────────────────────────────────┼──────────────────────────────┼
─────────────────┤
│ dropout (Dropout)                     │ (None, 64)                   │
0 │
├───────────────────────────────────────┼──────────────────────────────┼
─────────────────┤
│ dense_1 (Dense)                       │ (None, 32)                   │
2,080 │
├───────────────────────────────────────┼──────────────────────────────┼
─────────────────┤
│ dropout_1 (Dropout)                   │ (None, 32)                   │
0 │
├───────────────────────────────────────┼──────────────────────────────┼
─────────────────┤
│ dense_2 (Dense)                       │ (None, 3)                    │
99 │
└───────────────────────────────────────┴──────────────────────────────┴
─────────────────┘

 Total params: 2,563 (10.01 KB)

 Trainable params: 2,563 (10.01 KB)

 Non-trainable params: 0 (0.00 B)

from tensorflow.keras.optimizers import Adam

# Compile the model
model.compile(
    optimizer=Adam(learning_rate=0.001),  # Adam optimizer with a
default learning rate of 0.001
    loss='sparse_categorical_crossentropy',  # Loss function for
integer-labeled multi-class classification
    metrics=['accuracy']  # Track accuracy during training
)
```

```python
print("Model compiled successfully!")
```

```
Model compiled successfully!
```

```python
history = model.fit(X_train, y_train, validation_data=(X_test,
y_test), epochs=50, batch_size=32)
```

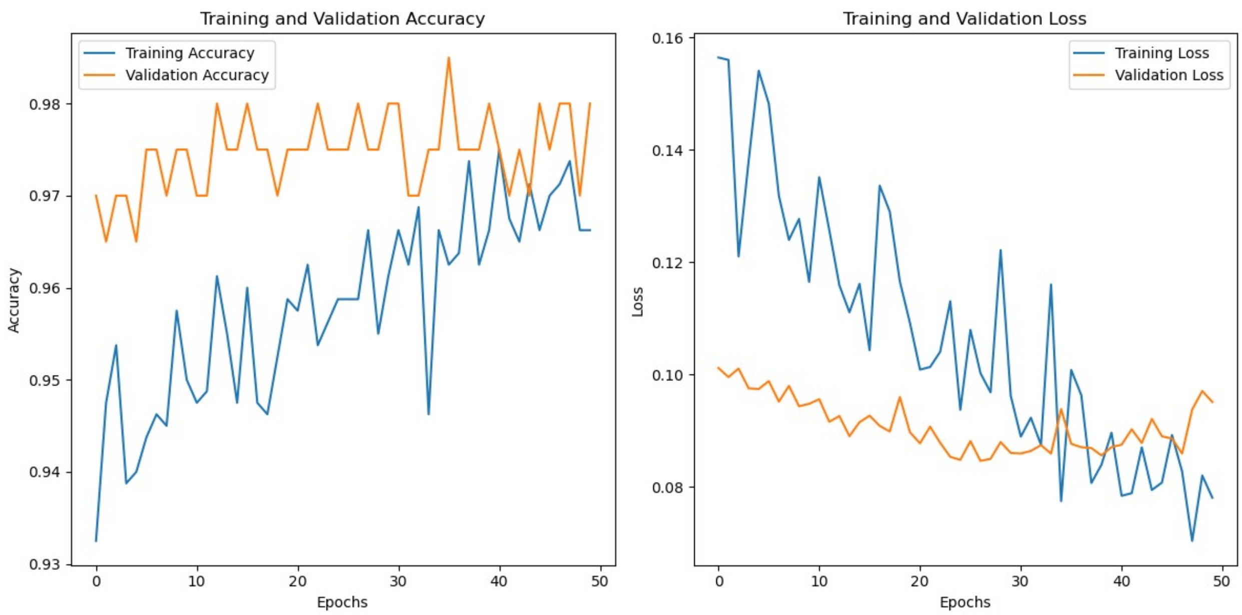
```
Epoch 1/50
25/25 ━━━━━━━━━━━━━━━━━━━━ 1s 18ms/step - accuracy: 0.9283 - loss:
0.1559 - val_accuracy: 0.9700 - val_loss: 0.1012
Epoch 2/50
25/25 ━━━━━━━━━━━━━━━━━━━━ 0s 14ms/step - accuracy: 0.9596 - loss:
0.1417 - val_accuracy: 0.9650 - val_loss: 0.0995
Epoch 3/50
25/25 ━━━━━━━━━━━━━━━━━━━━ 1s 13ms/step - accuracy: 0.9601 - loss:
0.1201 - val_accuracy: 0.9700 - val_loss: 0.1011
Epoch 4/50
25/25 ━━━━━━━━━━━━━━━━━━━━ 0s 13ms/step - accuracy: 0.9377 - loss:
0.1352 - val_accuracy: 0.9700 - val_loss: 0.0975
Epoch 5/50
25/25 ━━━━━━━━━━━━━━━━━━━━ 1s 19ms/step - accuracy: 0.9455 - loss:
0.1385 - val_accuracy: 0.9650 - val_loss: 0.0974
Epoch 6/50
25/25 ━━━━━━━━━━━━━━━━━━━━ 0s 12ms/step - accuracy: 0.9457 - loss:
0.1391 - val_accuracy: 0.9750 - val_loss: 0.0988
Epoch 7/50
25/25 ━━━━━━━━━━━━━━━━━━━━ 0s 14ms/step - accuracy: 0.9503 - loss:
0.1243 - val_accuracy: 0.9750 - val_loss: 0.0952
Epoch 8/50
25/25 ━━━━━━━━━━━━━━━━━━━━ 0s 14ms/step - accuracy: 0.9496 - loss:
0.1230 - val_accuracy: 0.9700 - val_loss: 0.0980
Epoch 9/50
25/25 ━━━━━━━━━━━━━━━━━━━━ 0s 14ms/step - accuracy: 0.9513 - loss:
0.1445 - val_accuracy: 0.9750 - val_loss: 0.0944
Epoch 10/50
25/25 ━━━━━━━━━━━━━━━━━━━━ 0s 13ms/step - accuracy: 0.9529 - loss:
0.1177 - val_accuracy: 0.9750 - val_loss: 0.0948
Epoch 11/50
25/25 ━━━━━━━━━━━━━━━━━━━━ 0s 13ms/step - accuracy: 0.9460 - loss:
0.1385 - val_accuracy: 0.9700 - val_loss: 0.0956
Epoch 12/50
25/25 ━━━━━━━━━━━━━━━━━━━━ 0s 13ms/step - accuracy: 0.9375 - loss:
0.1427 - val_accuracy: 0.9700 - val_loss: 0.0916
Epoch 13/50
25/25 ━━━━━━━━━━━━━━━━━━━━ 0s 14ms/step - accuracy: 0.9740 - loss:
0.0996 - val_accuracy: 0.9800 - val_loss: 0.0926
Epoch 14/50
25/25 ━━━━━━━━━━━━━━━━━━━━ 0s 13ms/step - accuracy: 0.9627 - loss:
0.1017 - val_accuracy: 0.9750 - val_loss: 0.0890
```

```
Epoch 15/50
25/25 ━━━━━━━━━━━━━━━━━━━━ 0s 15ms/step - accuracy: 0.9500 - loss:
0.1165 - val_accuracy: 0.9750 - val_loss: 0.0915
Epoch 16/50
25/25 ━━━━━━━━━━━━━━━━━━━━ 0s 14ms/step - accuracy: 0.9543 - loss:
0.1082 - val_accuracy: 0.9800 - val_loss: 0.0927
Epoch 17/50
25/25 ━━━━━━━━━━━━━━━━━━━━ 0s 15ms/step - accuracy: 0.9609 - loss:
0.1172 - val_accuracy: 0.9750 - val_loss: 0.0909
Epoch 18/50
25/25 ━━━━━━━━━━━━━━━━━━━━ 0s 14ms/step - accuracy: 0.9580 - loss:
0.1167 - val_accuracy: 0.9750 - val_loss: 0.0899
Epoch 19/50
25/25 ━━━━━━━━━━━━━━━━━━━━ 0s 14ms/step - accuracy: 0.9501 - loss:
0.1127 - val_accuracy: 0.9700 - val_loss: 0.0960
Epoch 20/50
25/25 ━━━━━━━━━━━━━━━━━━━━ 0s 12ms/step - accuracy: 0.9531 - loss:
0.1028 - val_accuracy: 0.9750 - val_loss: 0.0897
Epoch 21/50
25/25 ━━━━━━━━━━━━━━━━━━━━ 0s 13ms/step - accuracy: 0.9471 - loss:
0.1192 - val_accuracy: 0.9750 - val_loss: 0.0878
Epoch 22/50
25/25 ━━━━━━━━━━━━━━━━━━━━ 0s 14ms/step - accuracy: 0.9601 - loss:
0.1095 - val_accuracy: 0.9750 - val_loss: 0.0907
Epoch 23/50
25/25 ━━━━━━━━━━━━━━━━━━━━ 0s 12ms/step - accuracy: 0.9457 - loss:
0.1163 - val_accuracy: 0.9800 - val_loss: 0.0878
Epoch 24/50
25/25 ━━━━━━━━━━━━━━━━━━━━ 0s 13ms/step - accuracy: 0.9518 - loss:
0.1308 - val_accuracy: 0.9750 - val_loss: 0.0853
Epoch 25/50
25/25 ━━━━━━━━━━━━━━━━━━━━ 0s 13ms/step - accuracy: 0.9637 - loss:
0.1003 - val_accuracy: 0.9750 - val_loss: 0.0848
Epoch 26/50
25/25 ━━━━━━━━━━━━━━━━━━━━ 0s 15ms/step - accuracy: 0.9483 - loss:
0.1262 - val_accuracy: 0.9750 - val_loss: 0.0882
Epoch 27/50
25/25 ━━━━━━━━━━━━━━━━━━━━ 0s 13ms/step - accuracy: 0.9652 - loss:
0.0844 - val_accuracy: 0.9800 - val_loss: 0.0846
Epoch 28/50
25/25 ━━━━━━━━━━━━━━━━━━━━ 0s 15ms/step - accuracy: 0.9600 - loss:
0.1093 - val_accuracy: 0.9750 - val_loss: 0.0850
Epoch 29/50
25/25 ━━━━━━━━━━━━━━━━━━━━ 1s 18ms/step - accuracy: 0.9614 - loss:
0.1086 - val_accuracy: 0.9750 - val_loss: 0.0880
Epoch 30/50
25/25 ━━━━━━━━━━━━━━━━━━━━ 0s 14ms/step - accuracy: 0.9630 - loss:
0.0922 - val_accuracy: 0.9800 - val_loss: 0.0860
Epoch 31/50
```

```
25/25 ━━━━━━━━━━━━━━━━━━━━ 0s 14ms/step - accuracy: 0.9762 - loss:
0.0741 - val_accuracy: 0.9800 - val_loss: 0.0859
Epoch 32/50
25/25 ━━━━━━━━━━━━━━━━━━━━ 0s 14ms/step - accuracy: 0.9690 - loss:
0.0829 - val_accuracy: 0.9700 - val_loss: 0.0864
Epoch 33/50
25/25 ━━━━━━━━━━━━━━━━━━━━ 0s 14ms/step - accuracy: 0.9704 - loss:
0.0837 - val_accuracy: 0.9700 - val_loss: 0.0874
Epoch 34/50
25/25 ━━━━━━━━━━━━━━━━━━━━ 0s 14ms/step - accuracy: 0.9519 - loss:
0.1070 - val_accuracy: 0.9750 - val_loss: 0.0859
Epoch 35/50
25/25 ━━━━━━━━━━━━━━━━━━━━ 0s 14ms/step - accuracy: 0.9699 - loss:
0.0739 - val_accuracy: 0.9750 - val_loss: 0.0938
Epoch 36/50
25/25 ━━━━━━━━━━━━━━━━━━━━ 0s 14ms/step - accuracy: 0.9638 - loss:
0.0909 - val_accuracy: 0.9850 - val_loss: 0.0877
Epoch 37/50
25/25 ━━━━━━━━━━━━━━━━━━━━ 0s 13ms/step - accuracy: 0.9705 - loss:
0.0832 - val_accuracy: 0.9750 - val_loss: 0.0870
Epoch 38/50
25/25 ━━━━━━━━━━━━━━━━━━━━ 0s 13ms/step - accuracy: 0.9747 - loss:
0.0769 - val_accuracy: 0.9750 - val_loss: 0.0869
Epoch 39/50
25/25 ━━━━━━━━━━━━━━━━━━━━ 0s 13ms/step - accuracy: 0.9722 - loss:
0.0748 - val_accuracy: 0.9750 - val_loss: 0.0856
Epoch 40/50
25/25 ━━━━━━━━━━━━━━━━━━━━ 1s 20ms/step - accuracy: 0.9642 - loss:
0.0912 - val_accuracy: 0.9800 - val_loss: 0.0871
Epoch 41/50
25/25 ━━━━━━━━━━━━━━━━━━━━ 0s 13ms/step - accuracy: 0.9680 - loss:
0.0753 - val_accuracy: 0.9750 - val_loss: 0.0875
Epoch 42/50
25/25 ━━━━━━━━━━━━━━━━━━━━ 0s 14ms/step - accuracy: 0.9625 - loss:
0.0880 - val_accuracy: 0.9700 - val_loss: 0.0902
Epoch 43/50
25/25 ━━━━━━━━━━━━━━━━━━━━ 0s 13ms/step - accuracy: 0.9685 - loss:
0.0803 - val_accuracy: 0.9750 - val_loss: 0.0878
Epoch 44/50
25/25 ━━━━━━━━━━━━━━━━━━━━ 0s 15ms/step - accuracy: 0.9798 - loss:
0.0676 - val_accuracy: 0.9700 - val_loss: 0.0921
Epoch 45/50
25/25 ━━━━━━━━━━━━━━━━━━━━ 0s 13ms/step - accuracy: 0.9643 - loss:
0.0773 - val_accuracy: 0.9800 - val_loss: 0.0890
Epoch 46/50
25/25 ━━━━━━━━━━━━━━━━━━━━ 0s 14ms/step - accuracy: 0.9701 - loss:
0.0820 - val_accuracy: 0.9750 - val_loss: 0.0886
Epoch 47/50
25/25 ━━━━━━━━━━━━━━━━━━━━ 1s 15ms/step - accuracy: 0.9750 - loss:
```

```
0.0884 - val_accuracy: 0.9800 - val_loss: 0.0859
Epoch 48/50
25/25 ━━━━━━━━━━━━━━━━━━━━ 0s 12ms/step - accuracy: 0.9723 - loss:
0.0828 - val_accuracy: 0.9800 - val_loss: 0.0938
Epoch 49/50
25/25 ━━━━━━━━━━━━━━━━━━━━ 0s 15ms/step - accuracy: 0.9581 - loss:
0.0994 - val_accuracy: 0.9700 - val_loss: 0.0971
Epoch 50/50
25/25 ━━━━━━━━━━━━━━━━━━━━ 0s 14ms/step - accuracy: 0.9596 - loss:
0.0830 - val_accuracy: 0.9800 - val_loss: 0.0951

# Evaluate the model on the test dataset
test_loss, test_accuracy = model.evaluate(X_test, y_test, verbose=1)

# Print the results
print(f"Test Loss: {test_loss:.4f}")
print(f"Test Accuracy: {test_accuracy:.4f}")

7/7 ━━━━━━━━━━━━━━━━━━━━ 0s 15ms/step - accuracy: 0.9756 - loss:
0.1593
Test Loss: 0.0951
Test Accuracy: 0.9800

# Extract training and validation metrics
training_accuracy = history.history['accuracy'][-1]
validation_accuracy = history.history['val_accuracy'][-1]
training_loss = history.history['loss'][-1]
validation_loss = history.history['val_loss'][-1]

print("Training Accuracy:", training_accuracy)
print("Validation Accuracy:", validation_accuracy)
print("Training Loss:", training_loss)
print("Validation Loss:", validation_loss)

Training Accuracy: 0.9662500023841858
Validation Accuracy: 0.9800000190734863
Training Loss: 0.07806619256734848
Validation Loss: 0.09511351585388184

import matplotlib.pyplot as plt

# Plot training and validation accuracy
plt.figure(figsize=(12, 6))
plt.subplot(1, 2, 1)
plt.plot(history.history['accuracy'], label='Training Accuracy')
plt.plot(history.history['val_accuracy'], label='Validation Accuracy')
plt.title('Training and Validation Accuracy')
plt.xlabel('Epochs')
plt.ylabel('Accuracy')
plt.legend()
```

```python
# Plot training and validation loss
plt.subplot(1, 2, 2)
plt.plot(history.history['loss'], label='Training Loss')
plt.plot(history.history['val_loss'], label='Validation Loss')
plt.title('Training and Validation Loss')
plt.xlabel('Epochs')
plt.ylabel('Loss')
plt.legend()

# Display the plots
plt.tight_layout()
plt.show()
```




```python
# Save the entire model to a file
model.save("soil_classification_model.h5")
print("Model saved as 'soil_classification_model.h5'")
```

```
WARNING:absl:You are saving your model as an HDF5 file via
`model.save()` or `keras.saving.save_model(model)`. This file format
is considered legacy. We recommend using instead the native Keras
format, e.g. `model.save('my_model.keras')` or
`keras.saving.save_model(model, 'my_model.keras')`.

Model saved as 'soil_classification_model.h5'
```

```python
from tensorflow.keras.models import load_model

# Load the model from the file
loaded_model = load_model("soil_classification_model.h5")
print("Model loaded successfully!")
```

```
WARNING:absl:Compiled the loaded model, but the compiled metrics have
yet to be built. `model.compile_metrics` will be empty until you train
or evaluate the model.

Model loaded successfully!
```

```python
# Evaluate the loaded model on the test set
loss, accuracy = loaded_model.evaluate(X_test, y_test)
print(f"Loaded Model Accuracy: {accuracy:.2f}")

# Make predictions with the loaded model
new_soil_data = np.array([[2, 6.5, 15.0, 3.5, 1.2]])  # Example new
data
new_soil_data_scaled = scaler.transform(new_soil_data)
predictions = loaded_model.predict(new_soil_data_scaled)
predicted_class = np.argmax(predictions, axis=1)
print("Predicted Soil Type:", predicted_class)
```

```
7/7 ━━━━━━━━━━━━━━━━━━━━ 1s 13ms/step - accuracy: 0.9262 - loss:
0.2066
Loaded Model Accuracy: 0.93
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 259ms/step
Predicted Soil Type: [0]
```

```python
import os

# Get the file size of the saved model
model_file = "soil_classification_model.h5"

# Check if the file exists
if os.path.exists(model_file):
    model_size = os.path.getsize(model_file) / (1024 * 1024)  #
Convert bytes to MB
    print(f"Model size: {model_size:.2f} MB")
else:
    print(f"Model file '{model_file}' not found. Please save the model
first.")
```

```
Model size: 0.06 MB
```