dl-assignment1-part1-1

February 11, 2025

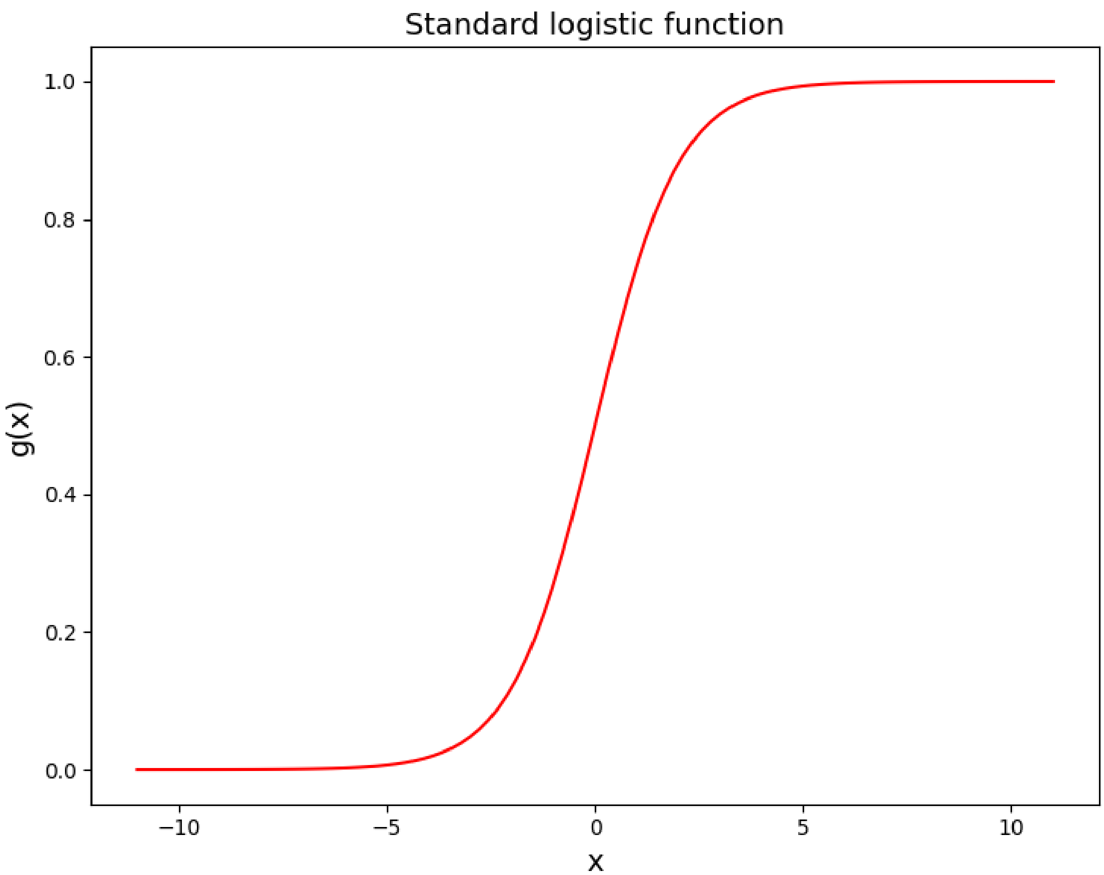
##Name : Ayush Fating ## ##PRN : 202201070127 ## ##Batch : T1 ##

#Logistic Regression from Scratch

[ ]:
```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from sklearn.datasets import load_iris

# Load Iris dataset
def load_iris_data():
    iris = load_iris()
    X = iris.data[:, :2]  # Using only two features for visualization
    y = (iris.target != 0).astype(int)  # Binary classification: Setosa vs␣
↪Non-Setosa
    return X, y

def logistic(x):
    y = 1 / (1 + np.exp(-x))
    return y

# Plotting the logistic function
plt.figure(figsize = (7.5, 6))
x = np.linspace(-11, 11, 100)
print(x)
plt.plot(x, logistic(x), color = 'red')
plt.xlabel("x", fontsize = 14)
plt.ylabel("g(x)", fontsize = 14)
plt.title("Standard logistic function", fontsize = 14)
plt.tight_layout()
plt.show()
```

```
[-11.         -10.77777778 -10.55555556 -10.33333333 -10.11111111
  -9.88888889  -9.66666667  -9.44444444  -9.22222222  -9.
  -8.77777778  -8.55555556  -8.33333333  -8.11111111  -7.88888889
  -7.66666667  -7.44444444  -7.22222222  -7.          -6.77777778
  -6.55555556  -6.33333333  -6.11111111  -5.88888889  -5.66666667
```

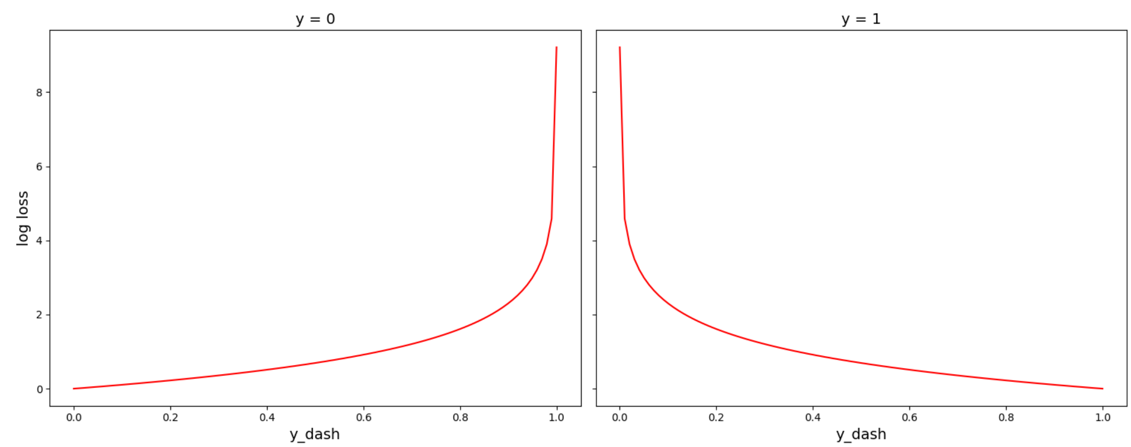
```
-5.44444444  -5.22222222  -5.          -4.77777778  -4.55555556
-4.33333333  -4.11111111  -3.88888889  -3.66666667  -3.44444444
-3.22222222  -3.          -2.77777778  -2.55555556  -2.33333333
-2.11111111  -1.88888889  -1.66666667  -1.44444444  -1.22222222
-1.          -0.77777778  -0.55555556  -0.33333333  -0.11111111
 0.11111111   0.33333333   0.55555556   0.77777778   1.
 1.22222222   1.44444444   1.66666667   1.88888889   2.11111111
 2.33333333   2.55555556   2.77777778   3.           3.22222222
 3.44444444   3.66666667   3.88888889   4.11111111   4.33333333
 4.55555556   4.77777778   5.           5.22222222   5.44444444
 5.66666667   5.88888889   6.11111111   6.33333333   6.55555556
 6.77777778   7.           7.22222222   7.44444444   7.66666667
 7.88888889   8.11111111   8.33333333   8.55555556   8.77777778
 9.           9.22222222   9.44444444   9.66666667   9.88888889
10.11111111  10.33333333  10.55555556  10.77777778  11.        ]
```



[ ]:
```
# Log loss
def log_loss(y, y_dash):
    loss = - (y * np.log(y_dash)) - ((1 - y) * np.log(1 - y_dash))
```

```python
    return loss

y, y_dash = 0, 0.6
print(f"log_loss({y}, {y_dash}) = {log_loss(y, y_dash)}")
y, y_dash = 1, 0.4
print(f"log_loss({y}, {y_dash}) = {log_loss(y, y_dash)}")
y, y_dash = 1, 0.8
print(f"log_loss({y}, {y_dash}) = {log_loss(y, y_dash)}")
y, y_dash = 0, 0.2
print(f"log_loss({y}, {y_dash}) = {log_loss(y, y_dash)}")

# Log loss for y = 0 and y = 1
fig, ax = plt.subplots(1, 2, figsize = (15, 6), sharex = True, sharey = True)
y_dash = np.linspace(0.0001, 0.9999, 100)
ax[0].plot(y_dash, log_loss(0, y_dash), color = 'red')
ax[0].set_title("y = 0", fontsize = 14)
ax[0].set_xlabel("y_dash", fontsize = 14)
ax[0].set_ylabel("log loss", fontsize = 14)
ax[1].plot(y_dash, log_loss(1, y_dash), color = 'red')
ax[1].set_title("y = 1", fontsize = 14)
ax[1].set_xlabel("y_dash", fontsize = 14)
plt.tight_layout()
plt.show()
```

```
log_loss(0, 0.6) = 0.916290731874155
log_loss(1, 0.4) = 0.916290731874155
log_loss(1, 0.8) = 0.2231435513142097
log_loss(0, 0.2) = 0.2231435513142097
```

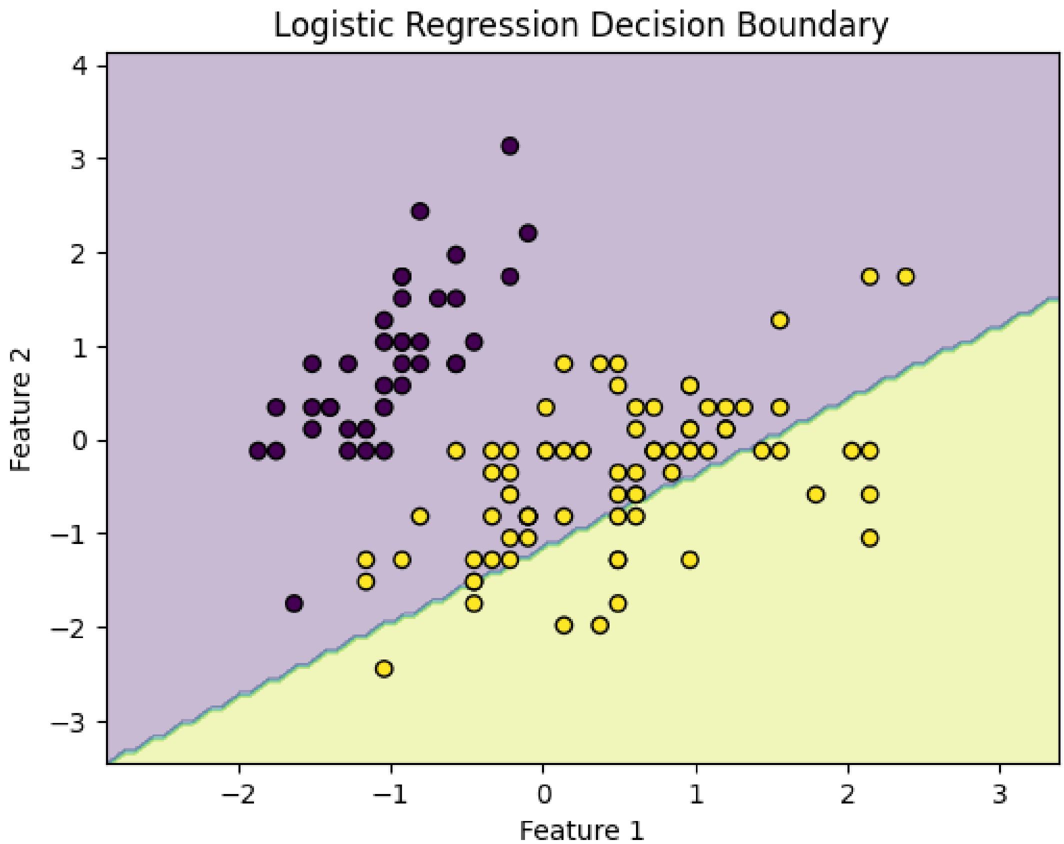


#Logistic Regression from Library

[ ]:

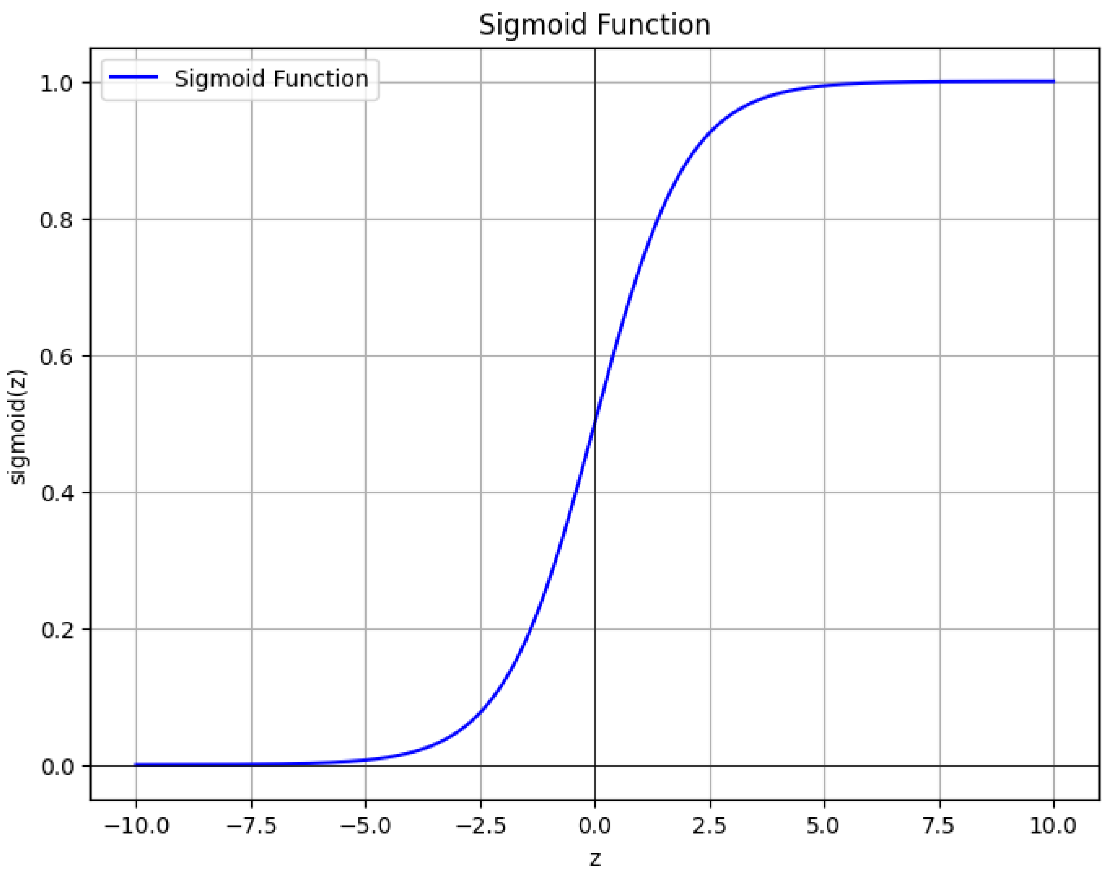
```
# Load Iris dataset
def load_iris_data():
    iris = load_iris()
    X = iris.data[:, :2]  # Using only two features for visualization
    y = (iris.target != 0).astype(int)  # Binary classification: Setosa vs␣
↪Non-Setosa
    return X, y

# Load and preprocess data
X, y = load_iris_data()
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2,␣
↪random_state=0)
scaler = StandardScaler()
X_train = scaler.fit_transform(X_train)
X_test = scaler.transform(X_test)

# Train logistic regression model
model = LogisticRegression()
model.fit(X_train, y_train)

# Predictions
y_pred = model.predict(X_test)
accuracy = accuracy_score(y_test, y_pred)
print(f'Accuracy: {accuracy:.2f}%')

# Plot decision boundary
def plot_decision_boundary(X, y, model):
    x_min, x_max = X[:, 0].min() - 1, X[:, 0].max() + 1
    y_min, y_max = X[:, 1].min() - 1, X[:, 1].max() + 1
    xx, yy = np.meshgrid(np.linspace(x_min, x_max, 100), np.linspace(y_min,␣
↪y_max, 100))
    grid = np.c_[xx.ravel(), yy.ravel()]
    grid = scaler.transform(grid)
    probs = model.predict(grid).reshape(xx.shape)
    plt.contourf(xx, yy, probs, alpha=0.3)
    plt.scatter(X[:, 0], X[:, 1], c=y, edgecolors='k')
    plt.xlabel('Feature 1')
    plt.ylabel('Feature 2')
    plt.title('Logistic Regression Decision Boundary')
    plt.show()

plot_decision_boundary(X_train, y_train, model)
```

```
Accuracy: 1.00%
```

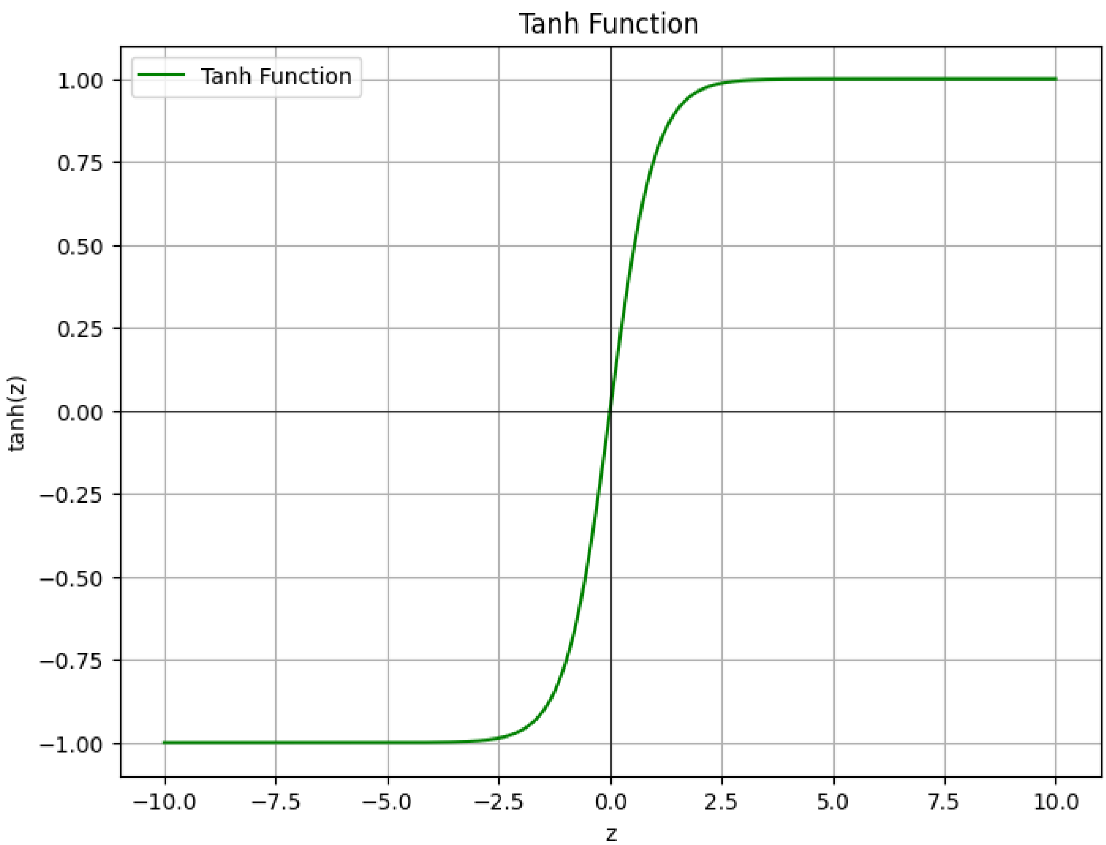


#Sigmoid, Tanh, Relu Function

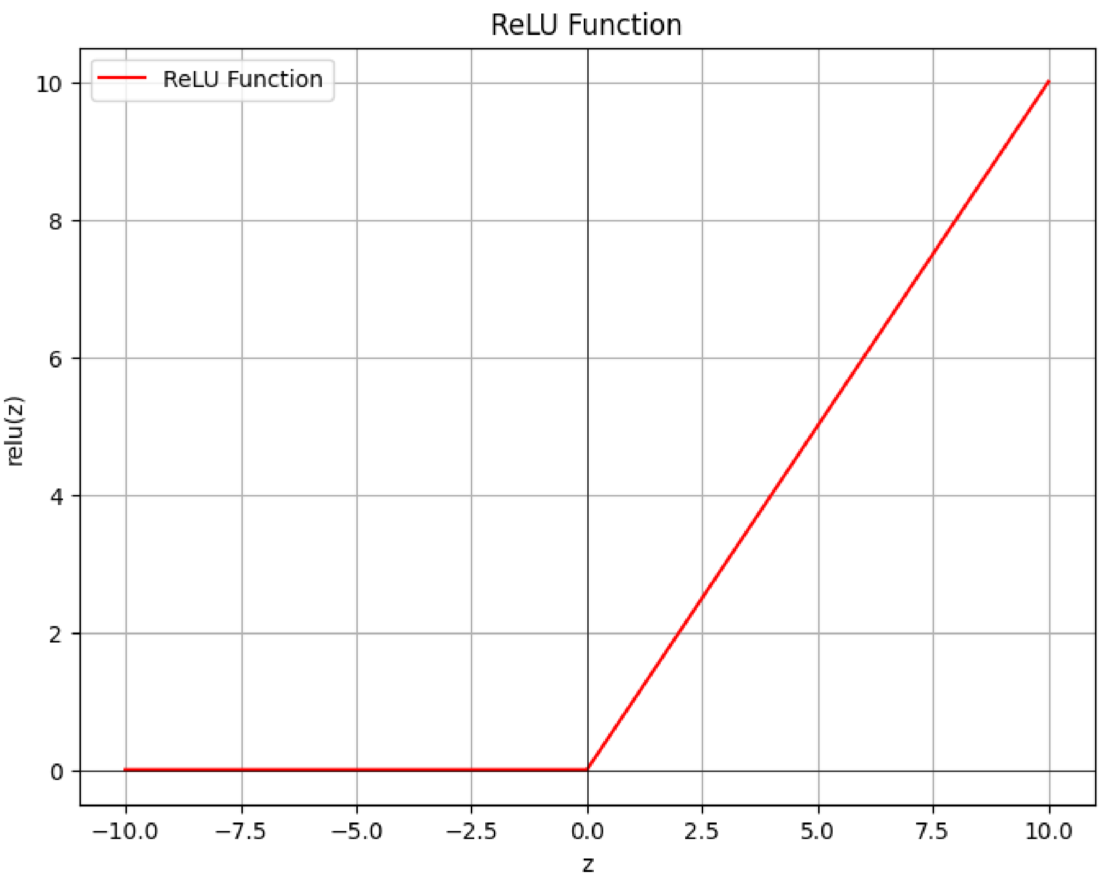
[ ]:
```
# Define the sigmoid activation function
def sigmoid(z):
    return 1 / (1 + np.exp(-z))

# Plotting the Sigmoid function
z_values = np.linspace(-10, 10, 1000)
sigmoid_values = sigmoid(z_values)

plt.figure(figsize=(8, 6))
plt.plot(z_values, sigmoid_values, label='Sigmoid Function', color='b')
plt.title('Sigmoid Function')
plt.xlabel('z')
plt.ylabel('sigmoid(z)')
plt.axhline(0, color='black', linewidth=0.5)
plt.axvline(0, color='black', linewidth=0.5)
plt.grid(True)
plt.legend(loc='best')
plt.show()
```



[ ]:
```
# Define the tanh activation function
def tanh(z):
    return np.tanh(z)

# Plotting the Tanh function
tanh_values = tanh(z_values)

plt.figure(figsize=(8, 6))
plt.plot(z_values, tanh_values, label='Tanh Function', color='g')
plt.title('Tanh Function')
plt.xlabel('z')
plt.ylabel('tanh(z)')
plt.axhline(0, color='black', linewidth=0.5)
plt.axvline(0, color='black', linewidth=0.5)
plt.grid(True)
plt.legend(loc='best')
plt.show()
```



[ ]:
```python
# Define the ReLU activation function
def relu(z):
    return np.maximum(0, z)

# Plotting the ReLU function
relu_values = relu(z_values)

plt.figure(figsize=(8, 6))
plt.plot(z_values, relu_values, label='ReLU Function', color='r')
plt.title('ReLU Function')
plt.xlabel('z')
plt.ylabel('relu(z)')
plt.axhline(0, color='black', linewidth=0.5)
plt.axvline(0, color='black', linewidth=0.5)
plt.grid(True)
plt.legend(loc='best')
plt.show()
```



#Log Loss for vector code

[ ]:
```
import numpy as np
import pandas as pd
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler, OneHotEncoder
from sklearn.datasets import load_iris

# Softmax Activation Function
def softmax(z):
    exp_z = np.exp(z - np.max(z, axis=1, keepdims=True))  # Stability trick
    return exp_z / np.sum(exp_z, axis=1, keepdims=True)

# Cross-Entropy Loss for Multiclass Classification
def compute_cross_entropy_loss(y_true, y_pred):
    epsilon = 1e-15  # To avoid log(0)
    y_pred = np.clip(y_pred, epsilon, 1 - epsilon)
    return -np.mean(np.sum(y_true * np.log(y_pred), axis=1))
```

```python
# Gradient Descent for Softmax Regression
def gradient_descent(X, y, weights, learning_rate, epochs):
    m = X.shape[0]  # Number of training examples
    for epoch in range(epochs):
        # Compute predictions
        z = np.dot(X, weights)
        predictions = softmax(z)

        # Compute gradients
        gradient = np.dot(X.T, (predictions - y)) / m

        # Update weights
        weights -= learning_rate * gradient

        # Compute and print loss every 100 epochs
        if epoch % 100 == 0:
            loss = compute_cross_entropy_loss(y, predictions)
            print(f"Epoch {epoch}, Cross-Entropy Loss: {loss:.4f}")

    return weights

# Load predefined Iris dataset
def load_iris_data():
    iris = load_iris()
    X = iris.data  # Using all features
    y = iris.target  # Target labels (Iris species)
    return X, y

# Load the data
X, y = load_iris_data()

# One-hot encode the target labels for multiclass classification
encoder = OneHotEncoder(sparse_output=False)  # Updated from sparse=False
y = encoder.fit_transform(y.reshape(-1, 1))

# Split the dataset into training (80%) and testing (20%) sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2,
 ↪random_state=42, stratify=y)

# Standardize the features
scaler = StandardScaler()
X_train = scaler.fit_transform(X_train)
X_test = scaler.transform(X_test)

# Add an intercept term (bias) to the features
X_train = np.hstack((np.ones((X_train.shape[0], 1)), X_train))
X_test = np.hstack((np.ones((X_test.shape[0], 1)), X_test))
```

```python
# Initialize weights (for 3 output classes)
num_classes = y.shape[1]
weights = np.zeros((X_train.shape[1], num_classes))

# Train the model using Gradient Descent
learning_rate = 0.1
epochs = 1000
weights = gradient_descent(X_train, y_train, weights, learning_rate, epochs)

# Make predictions on the test set
z_test = np.dot(X_test, weights)
y_test_pred_prob = softmax(z_test)  # Get class probabilities
y_test_pred = np.argmax(y_test_pred_prob, axis=1)  # Convert to class labels
y_test_true = np.argmax(y_test, axis=1)  # True class labels

# Compute test loss and accuracy
test_loss = compute_cross_entropy_loss(y_test, y_test_pred_prob)
accuracy = np.mean(y_test_pred == y_test_true)

print(f"\nTest Cross-Entropy Loss: {test_loss:.4f}")
print(f"Test Accuracy: {accuracy:.4f}")
```

```
Epoch 0, Cross-Entropy Loss: 1.0986
Epoch 100, Cross-Entropy Loss: 0.3196
Epoch 200, Cross-Entropy Loss: 0.2554
Epoch 300, Cross-Entropy Loss: 0.2174
Epoch 400, Cross-Entropy Loss: 0.1912
Epoch 500, Cross-Entropy Loss: 0.1721
Epoch 600, Cross-Entropy Loss: 0.1575
Epoch 700, Cross-Entropy Loss: 0.1461
Epoch 800, Cross-Entropy Loss: 0.1368
Epoch 900, Cross-Entropy Loss: 0.1292

Test Cross-Entropy Loss: 0.1433
Test Accuracy: 0.9667
```

dl-assignment1-part2-1

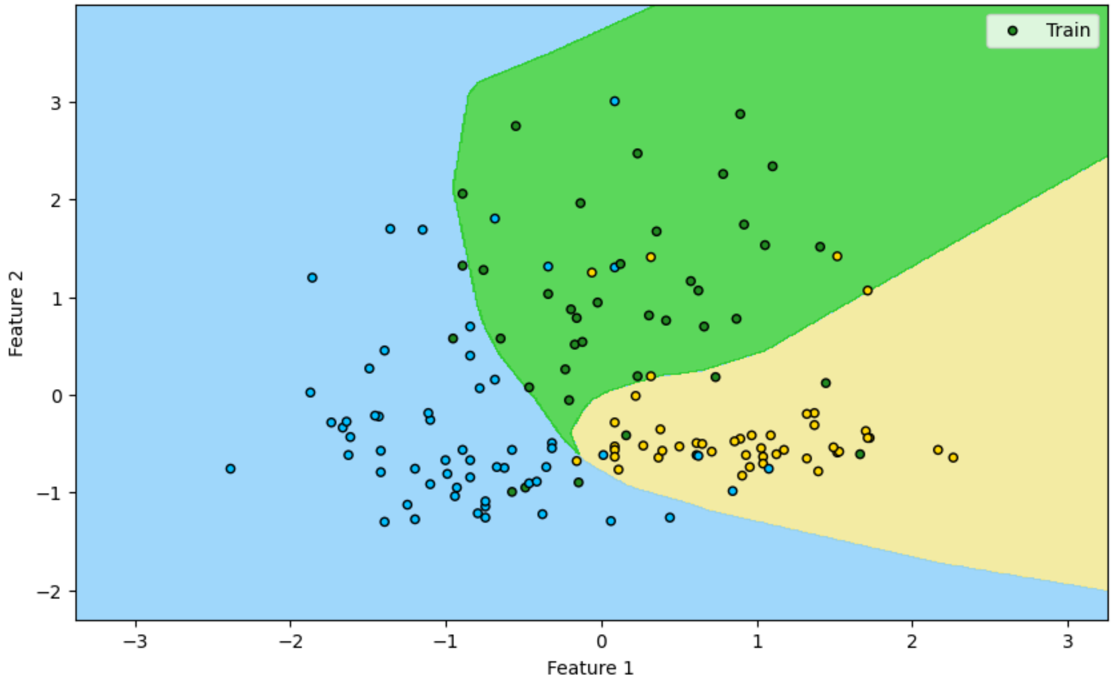
February 11, 2025

#Sklearn Implementation (Wine Dataset)

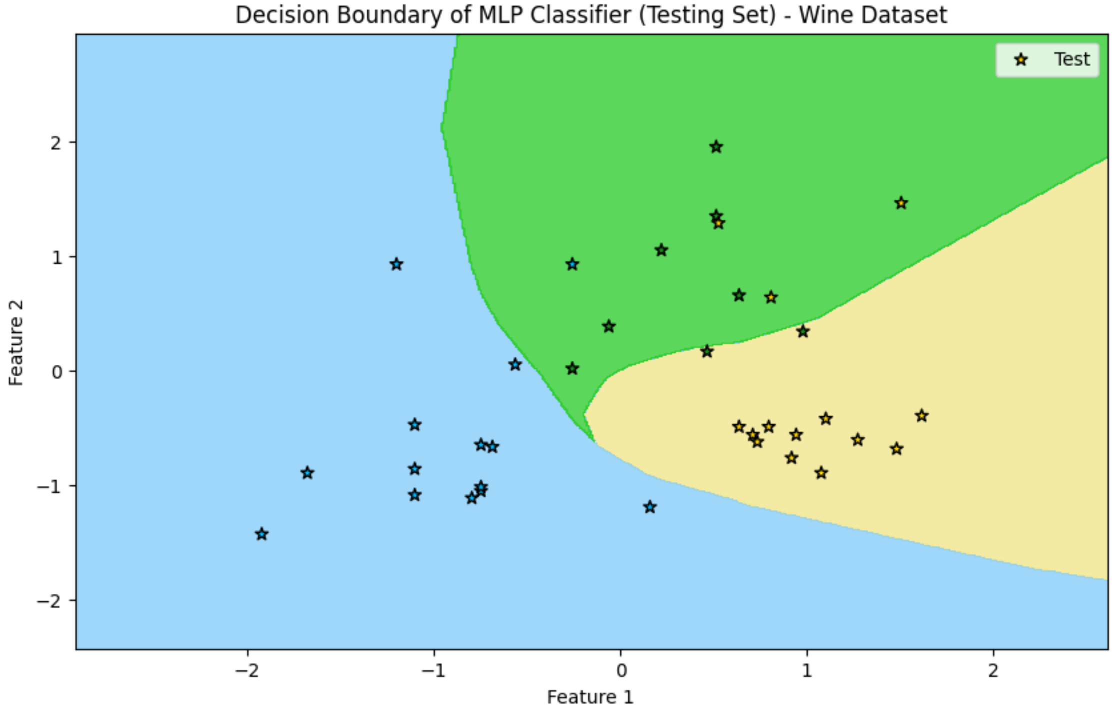
[ ]:
```python
from sklearn.datasets import load_wine
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from sklearn.neural_network import MLPClassifier
from sklearn.metrics import accuracy_score, classification_report
import numpy as np
import matplotlib.pyplot as plt
from matplotlib.colors import ListedColormap

# Load the Wine dataset
data = load_wine()
X = data.data[:, :2]  # Use the first two features for 2D visualization
y = data.target  # Target labels

# Split the 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 the features
scaler = StandardScaler()
X_train = scaler.fit_transform(X_train)
X_test = scaler.transform(X_test)

# Define the MLP classifier
mlp = MLPClassifier(hidden_layer_sizes=(10, 10), max_iter=1000, random_state=42)

# Train the MLP classifier
mlp.fit(X_train, y_train)

# Make predictions
y_pred = mlp.predict(X_test)

# Plot decision boundaries for training set
x_min, x_max = X_train[:, 0].min() - 1, X_train[:, 0].max() + 1
y_min, y_max = X_train[:, 1].min() - 1, X_train[:, 1].max() + 1
xx, yy = np.meshgrid(np.arange(x_min, x_max, 0.01),
```

```
                     np.arange(y_min, y_max, 0.01))

Z_train = mlp.predict(np.c_[xx.ravel(), yy.ravel()])
Z_train = Z_train.reshape(xx.shape)

# Define new color maps
cmap_light = ListedColormap(['#F0E68C', '#87CEFA', '#32CD32'])
cmap_bold = ListedColormap(['#FFD700', '#00BFFF', '#228B22'])

plt.figure(figsize=(10, 6))
plt.contourf(xx, yy, Z_train, alpha=0.8, cmap=cmap_light)
plt.scatter(X_train[:, 0], X_train[:, 1], c=y_train, cmap=cmap_bold,␣
 ↪edgecolor='k', s=20, label='Train')
plt.title("Decision Boundary of MLP Classifier (Training Set) - Wine Dataset")
plt.xlabel("Feature 1")
plt.ylabel("Feature 2")
plt.legend()
plt.show()

# Plot decision boundaries for testing set
x_min, x_max = X_test[:, 0].min() - 1, X_test[:, 0].max() + 1
y_min, y_max = X_test[:, 1].min() - 1, X_test[:, 1].max() + 1
xx, yy = np.meshgrid(np.arange(x_min, x_max, 0.01),
                     np.arange(y_min, y_max, 0.01))

Z_test = mlp.predict(np.c_[xx.ravel(), yy.ravel()])
Z_test = Z_test.reshape(xx.shape)

plt.figure(figsize=(10, 6))
plt.contourf(xx, yy, Z_test, alpha=0.8, cmap=cmap_light)
plt.scatter(X_test[:, 0], X_test[:, 1], c=y_test, cmap=cmap_bold,␣
 ↪edgecolor='k', s=50, label='Test', marker='*')
plt.title("Decision Boundary of MLP Classifier (Testing Set) - Wine Dataset")
plt.xlabel("Feature 1")
plt.ylabel("Feature 2")
plt.legend()
plt.show()
```

Decision Boundary of MLP Classifier (Training Set) - Wine Dataset



Decision Boundary of MLP Classifier (Testing Set) - Wine Dataset



[ ]:
```
accuracy = accuracy_score(y_test, y_pred)
print("Accuracy:", accuracy)
print("\nClassification Report:\n", classification_report(y_test, y_pred))
```

```
Accuracy: 0.8333333333333334

Classification Report:
               precision    recall  f1-score   support

           0       0.85      0.79      0.81        14
           1       1.00      0.93      0.96        14
           2       0.60      0.75      0.67         8

    accuracy                           0.83        36
   macro avg       0.82      0.82      0.81        36
weighted avg       0.85      0.83      0.84        36
```

#Keras Implementation (Wine Dataset)

[ ]:
```
# Import necessary libraries
import numpy as np
import matplotlib.pyplot as plt
from keras.models import Sequential
from keras.layers import Dense
from sklearn.datasets import load_wine
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from sklearn.metrics import classification_report, ConfusionMatrixDisplay
from tensorflow.keras.utils import to_categorical

# Load the Wine dataset
wine = load_wine()
X = wine.data
y = wine.target

# Convert labels to categorical (One-Hot Encoding)
y = to_categorical(y, num_classes=3)  # 3 classes in the Wine dataset

# Split the dataset into training (80%) and testing (20%) sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2,␣
 ↪random_state=42)

# Standardize the data (normalize features)
scaler = StandardScaler()
X_train = scaler.fit_transform(X_train)
X_test = scaler.transform(X_test)
```

```
# Define the neural network model
model = Sequential([
    Dense(32, activation='relu', input_dim=X_train.shape[1]),  # Input layer
    Dense(16, activation='relu'),  # Hidden layer
    Dense(3, activation='softmax')  # Output layer with 3 neurons (multi-class␣
↪classification)
])

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

# Train the model
history = model.fit(X_train, y_train, epochs=50, batch_size=8,␣
↪validation_data=(X_test, y_test), verbose=1)
```

```
Epoch 1/50
```

```
/usr/local/lib/python3.11/dist-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)
```

```
18/18             1s 21ms/step -
accuracy: 0.3066 - loss: 1.3579 - val_accuracy: 0.4167 - val_loss: 1.0558
Epoch 2/50
18/18             0s 9ms/step -
accuracy: 0.5203 - loss: 0.9760 - val_accuracy: 0.5833 - val_loss: 0.8548
Epoch 3/50
18/18             0s 9ms/step -
accuracy: 0.6097 - loss: 0.8018 - val_accuracy: 0.7222 - val_loss: 0.6927
Epoch 4/50
18/18             0s 5ms/step -
accuracy: 0.7235 - loss: 0.6658 - val_accuracy: 0.8611 - val_loss: 0.5715
Epoch 5/50
18/18             0s 6ms/step -
accuracy: 0.8218 - loss: 0.5142 - val_accuracy: 0.9444 - val_loss: 0.4763
Epoch 6/50
18/18             0s 5ms/step -
accuracy: 0.9348 - loss: 0.4493 - val_accuracy: 0.9444 - val_loss: 0.3894
Epoch 7/50
18/18             0s 6ms/step -
accuracy: 0.9730 - loss: 0.3336 - val_accuracy: 0.9722 - val_loss: 0.3136
Epoch 8/50
18/18             0s 6ms/step -
accuracy: 0.9878 - loss: 0.2583 - val_accuracy: 1.0000 - val_loss: 0.2432
Epoch 9/50
```

```
18/18            0s 8ms/step -
accuracy: 0.9649 - loss: 0.2458 - val_accuracy: 1.0000 - val_loss: 0.1819
Epoch 10/50
18/18            0s 8ms/step -
accuracy: 0.9645 - loss: 0.1711 - val_accuracy: 1.0000 - val_loss: 0.1361
Epoch 11/50
18/18            0s 6ms/step -
accuracy: 0.9697 - loss: 0.1681 - val_accuracy: 1.0000 - val_loss: 0.0997
Epoch 12/50
18/18            0s 6ms/step -
accuracy: 0.9959 - loss: 0.1109 - val_accuracy: 1.0000 - val_loss: 0.0736
Epoch 13/50
18/18            0s 6ms/step -
accuracy: 0.9865 - loss: 0.1216 - val_accuracy: 1.0000 - val_loss: 0.0582
Epoch 14/50
18/18            0s 5ms/step -
accuracy: 0.9887 - loss: 0.0777 - val_accuracy: 1.0000 - val_loss: 0.0470
Epoch 15/50
18/18            0s 5ms/step -
accuracy: 0.9952 - loss: 0.0830 - val_accuracy: 1.0000 - val_loss: 0.0402
Epoch 16/50
18/18            0s 5ms/step -
accuracy: 0.9980 - loss: 0.0619 - val_accuracy: 1.0000 - val_loss: 0.0335
Epoch 17/50
18/18            0s 7ms/step -
accuracy: 0.9832 - loss: 0.0571 - val_accuracy: 1.0000 - val_loss: 0.0290
Epoch 18/50
18/18            0s 5ms/step -
accuracy: 0.9952 - loss: 0.0652 - val_accuracy: 1.0000 - val_loss: 0.0245
Epoch 19/50
18/18            0s 5ms/step -
accuracy: 0.9952 - loss: 0.0445 - val_accuracy: 1.0000 - val_loss: 0.0219
Epoch 20/50
18/18            0s 5ms/step -
accuracy: 1.0000 - loss: 0.0357 - val_accuracy: 1.0000 - val_loss: 0.0195
Epoch 21/50
18/18            0s 5ms/step -
accuracy: 1.0000 - loss: 0.0417 - val_accuracy: 1.0000 - val_loss: 0.0168
Epoch 22/50
18/18            0s 5ms/step -
accuracy: 1.0000 - loss: 0.0337 - val_accuracy: 1.0000 - val_loss: 0.0154
Epoch 23/50
18/18            0s 5ms/step -
accuracy: 1.0000 - loss: 0.0317 - val_accuracy: 1.0000 - val_loss: 0.0135
Epoch 24/50
18/18            0s 5ms/step -
accuracy: 1.0000 - loss: 0.0169 - val_accuracy: 1.0000 - val_loss: 0.0124
Epoch 25/50
```

```
18/18             0s 5ms/step -
accuracy: 1.0000 - loss: 0.0331 - val_accuracy: 1.0000 - val_loss: 0.0111
Epoch 26/50
18/18             0s 5ms/step -
accuracy: 1.0000 - loss: 0.0221 - val_accuracy: 1.0000 - val_loss: 0.0102
Epoch 27/50
18/18             0s 5ms/step -
accuracy: 1.0000 - loss: 0.0210 - val_accuracy: 1.0000 - val_loss: 0.0095
Epoch 28/50
18/18             0s 5ms/step -
accuracy: 1.0000 - loss: 0.0226 - val_accuracy: 1.0000 - val_loss: 0.0085
Epoch 29/50
18/18             0s 5ms/step -
accuracy: 1.0000 - loss: 0.0141 - val_accuracy: 1.0000 - val_loss: 0.0079
Epoch 30/50
18/18             0s 5ms/step -
accuracy: 1.0000 - loss: 0.0170 - val_accuracy: 1.0000 - val_loss: 0.0075
Epoch 31/50
18/18             0s 5ms/step -
accuracy: 1.0000 - loss: 0.0179 - val_accuracy: 1.0000 - val_loss: 0.0067
Epoch 32/50
18/18             0s 5ms/step -
accuracy: 1.0000 - loss: 0.0142 - val_accuracy: 1.0000 - val_loss: 0.0063
Epoch 33/50
18/18             0s 5ms/step -
accuracy: 1.0000 - loss: 0.0166 - val_accuracy: 1.0000 - val_loss: 0.0057
Epoch 34/50
18/18             0s 6ms/step -
accuracy: 1.0000 - loss: 0.0102 - val_accuracy: 1.0000 - val_loss: 0.0055
Epoch 35/50
18/18             0s 5ms/step -
accuracy: 1.0000 - loss: 0.0110 - val_accuracy: 1.0000 - val_loss: 0.0052
Epoch 36/50
18/18             0s 5ms/step -
accuracy: 1.0000 - loss: 0.0114 - val_accuracy: 1.0000 - val_loss: 0.0047
Epoch 37/50
18/18             0s 5ms/step -
accuracy: 1.0000 - loss: 0.0087 - val_accuracy: 1.0000 - val_loss: 0.0045
Epoch 38/50
18/18             0s 6ms/step -
accuracy: 1.0000 - loss: 0.0086 - val_accuracy: 1.0000 - val_loss: 0.0044
Epoch 39/50
18/18             0s 5ms/step -
accuracy: 1.0000 - loss: 0.0096 - val_accuracy: 1.0000 - val_loss: 0.0039
Epoch 40/50
18/18             0s 5ms/step -
accuracy: 1.0000 - loss: 0.0115 - val_accuracy: 1.0000 - val_loss: 0.0037
Epoch 41/50
```

```
18/18              0s 6ms/step -
accuracy: 1.0000 - loss: 0.0083 - val_accuracy: 1.0000 - val_loss: 0.0036
Epoch 42/50
18/18              0s 7ms/step -
accuracy: 1.0000 - loss: 0.0078 - val_accuracy: 1.0000 - val_loss: 0.0034
Epoch 43/50
18/18              0s 5ms/step -
accuracy: 1.0000 - loss: 0.0088 - val_accuracy: 1.0000 - val_loss: 0.0032
Epoch 44/50
18/18              0s 5ms/step -
accuracy: 1.0000 - loss: 0.0063 - val_accuracy: 1.0000 - val_loss: 0.0030
Epoch 45/50
18/18              0s 5ms/step -
accuracy: 1.0000 - loss: 0.0069 - val_accuracy: 1.0000 - val_loss: 0.0029
Epoch 46/50
18/18              0s 5ms/step -
accuracy: 1.0000 - loss: 0.0054 - val_accuracy: 1.0000 - val_loss: 0.0028
Epoch 47/50
18/18              0s 5ms/step -
accuracy: 1.0000 - loss: 0.0063 - val_accuracy: 1.0000 - val_loss: 0.0026
Epoch 48/50
18/18              0s 6ms/step -
accuracy: 1.0000 - loss: 0.0060 - val_accuracy: 1.0000 - val_loss: 0.0025
Epoch 49/50
18/18              0s 6ms/step -
accuracy: 1.0000 - loss: 0.0051 - val_accuracy: 1.0000 - val_loss: 0.0024
Epoch 50/50
18/18              0s 8ms/step -
accuracy: 1.0000 - loss: 0.0046 - val_accuracy: 1.0000 - val_loss: 0.0024
```

[ ]:
```
test_loss, test_acc = model.evaluate(X_test, y_test)
print(f"Test Loss: {test_loss:.4f}")
print(f"Test Accuracy: {test_acc:.4f}")
```

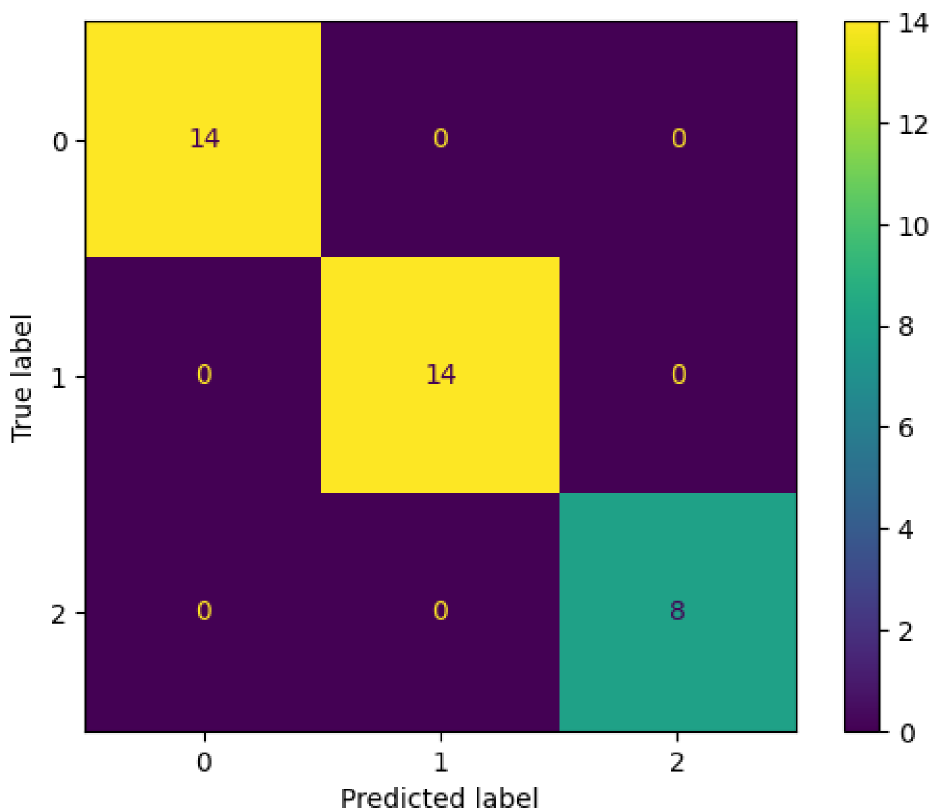
```
2/2              0s 33ms/step -
accuracy: 1.0000 - loss: 0.0024
Test Loss: 0.0024
Test Accuracy: 1.0000
```

[ ]:
```
# Predict on test data
y_pred = model.predict(X_test)
y_pred_classes = np.argmax(y_pred, axis=1)
y_test_classes = np.argmax(y_test, axis=1)
```

```
2/2              0s 32ms/step
```

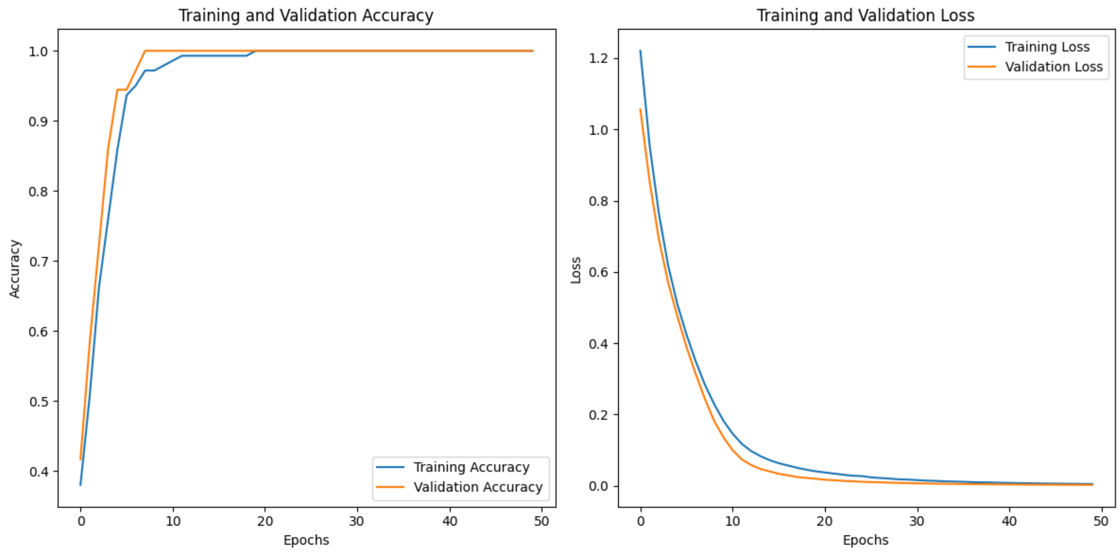
[ ]:
```
# Display classification report
print(classification_report(y_test_classes, y_pred_classes))

# Show confusion matrix
ConfusionMatrixDisplay.from_predictions(y_test_classes, y_pred_classes)
plt.show()
```

```
              precision    recall  f1-score   support

           0       1.00      1.00      1.00        14
           1       1.00      1.00      1.00        14
           2       1.00      1.00      1.00         8

    accuracy                           1.00        36
   macro avg       1.00      1.00      1.00        36
weighted avg       1.00      1.00      1.00        36
```



[ ]:
```
# Step 9: Plot Training and Validation Loss/Accuracy
# Extract metrics from the history object (collected during training)
acc = history.history['accuracy']         # Training accuracy
```

```python
val_acc = history.history['val_accuracy']  # Validation accuracy
loss = history.history['loss']            # Training loss
val_loss = history.history['val_loss']    # Validation loss

# Plot Accuracy
plt.figure(figsize=(12, 6))
plt.subplot(1, 2, 1)  # Create the first subplot for accuracy
plt.plot(acc, label='Training Accuracy')
plt.plot(val_acc, label='Validation Accuracy')
plt.title('Training and Validation Accuracy')  # Add a title
plt.xlabel('Epochs')                           # Label the x-axis
plt.ylabel('Accuracy')                         # Label the y-axis
plt.legend()                                   # Add a legend

# Plot Loss
plt.subplot(1, 2, 2)  # Create the second subplot for loss
plt.plot(loss, label='Training Loss')
plt.plot(val_loss, label='Validation Loss')
plt.title('Training and Validation Loss')      # Add a title
plt.xlabel('Epochs')                           # Label the x-axis
plt.ylabel('Loss')                             # Label the y-axis
plt.legend()                                   # Add a legend

# Adjust layout to prevent overlap and display the plots
plt.tight_layout()
plt.show()
```



#Back Propogation (Wine dataset)

[2]:

```
import numpy as np
import matplotlib.pyplot as plt
from sklearn.datasets import load_wine
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler

# Load the Wine dataset
wine = load_wine()
X = wine.data  # Features
y = wine.target  # Labels

# Convert labels to one-hot encoding
y_one_hot = np.zeros((y.size, y.max() + 1))
y_one_hot[np.arange(y.size), y] = 1

# Split the dataset
X_train, X_test, y_train, y_test = train_test_split(X, y_one_hot, test_size=0.
↪2, random_state=42)

# Normalize the dataset
scaler = StandardScaler()
X_train = scaler.fit_transform(X_train)
X_test = scaler.transform(X_test)

# Initialize network parameters
input_neurons = X_train.shape[1]
hidden_neurons = 8
output_neurons = 3
np.random.seed(42)
weights_input_hidden = np.random.randn(input_neurons, hidden_neurons)
weights_hidden_output = np.random.randn(hidden_neurons, output_neurons)
bias_hidden = np.zeros((1, hidden_neurons))
bias_output = np.zeros((1, output_neurons))
learning_rate = 0.01
epochs = 500
losses = []

# Activation function and derivative
def sigmoid(x):
    return 1 / (1 + np.exp(-x))

def sigmoid_derivative(x):
    return x * (1 - x)

# Training loop
for epoch in range(epochs):
    # Forward pass
```

```python
    hidden_layer_input = np.dot(X_train, weights_input_hidden) + bias_hidden
    hidden_layer_output = sigmoid(hidden_layer_input)
    final_layer_input = np.dot(hidden_layer_output, weights_hidden_output) +␣
 ↪bias_output
    final_output = sigmoid(final_layer_input)

    # Compute error
    error = y_train - final_output

    # Backpropagation
    d_output = error * sigmoid_derivative(final_output)
    error_hidden = d_output.dot(weights_hidden_output.T)
    d_hidden = error_hidden * sigmoid_derivative(hidden_layer_output)

    # Update weights and biases
    weights_hidden_output += hidden_layer_output.T.dot(d_output) * learning_rate
    weights_input_hidden += X_train.T.dot(d_hidden) * learning_rate
    bias_output += np.sum(d_output, axis=0, keepdims=True) * learning_rate
    bias_hidden += np.sum(d_hidden, axis=0, keepdims=True) * learning_rate

    # Store loss every 50 epochs
    if epoch % 50 == 0:
        loss = np.mean(np.abs(error))
        losses.append(loss)
        print(f"Epoch {epoch}, Loss: {loss:.4f}")

# Testing the network
hidden_layer_input = np.dot(X_test, weights_input_hidden) + bias_hidden
hidden_layer_output = sigmoid(hidden_layer_input)
final_layer_input = np.dot(hidden_layer_output, weights_hidden_output) +␣
 ↪bias_output
final_output = sigmoid(final_layer_input)

predictions = np.argmax(final_output, axis=1)
y_true = np.argmax(y_test, axis=1)

# Plot training loss curve
plt.plot(range(0, epochs, 50), losses)
plt.xlabel('Epochs')
plt.ylabel('Loss')
plt.title('Training Loss Curve')
plt.show()
```

```
Epoch 0, Loss: 0.5701
Epoch 50, Loss: 0.1756
Epoch 100, Loss: 0.1136
Epoch 150, Loss: 0.0860
```

```
Epoch 200, Loss: 0.0707
Epoch 250, Loss: 0.0607
Epoch 300, Loss: 0.0537
Epoch 350, Loss: 0.0482
Epoch 400, Loss: 0.0440
Epoch 450, Loss: 0.0406
```