Project Documentation: Neural Network Models with NumPy and TensorFlow

Overview

This project comprises three Jupyter notebooks, each showcasing different neural network architectures for various tasks:

1. **Single Perceptron for Regression**
2. **Two-Layer Neural Network for Binary Classification**
3. **Multi-Layer Neural Network for Multi-Class Classification**

`linear_regression_single_perceptron.ipynb`

This notebook illustrates a neural network model utilizing a single perceptron for linear regression tasks. Two variants are presented: one with a single input feature and another with two input features.

Contents

1. **Single Input Perceptron**

   - **Data Generation**: Synthetic data is created using `make_regression` from sklearn.
   - **Model Implementation**:
     - `initialize_parameters`: Initializes weights and biases.
     - `forward_propagation`: Computes the predicted output.
     - `compute_cost`: Calculates the mean squared error cost.
     - `gradient_descent`: Updates parameters using gradient descent.
     - `nn_model`: Trains the model using the above functions.
   - **Visualization**: Plots the regression line and data points.

2. **Two Input Perceptron**

   - **Data Preparation**: Reads and preprocesses the house prices dataset.
   - **Model Implementation**: Reuses functions from the single input model.
   - **Visualization and Evaluation**: Plots the regression results and calculates RMSE and R^2 score.

`NeuralNet_with_Two_Layers.ipynb`

This notebook implements a neural network with one hidden layer for binary classification tasks. The hidden layer can have an arbitrary number of neurons.

Contents

1. **Data Generation**: Synthetic data is created using `make_blobs` from sklearn.
2. **Model Implementation**:
   - `initialize_parameters`: Initializes weights and biases for both layers.
   - `forward_propagation`: Computes the predicted output.
   - `compute_cost`: Calculates the binary cross-entropy loss.
   - `gradient_descent`: Updates parameters using gradient descent.
   - `nn_model`: Trains the model using the above functions.
   - `predict`: Makes predictions using the trained model.
   - `plot_decision_boundary`: Visualizes the decision boundary of the trained model.
3. **Visualization**: Plots decision boundaries for different datasets.
4. **Evaluation**: Calculates and displays the RMSE and R^2 score.

`multi_layer_nn.ipynb`

This notebook implements a multi-layer neural network for multi-class classification tasks using the MNIST digits dataset.

Contents

1. **Data Preparation**:
   - Loads and preprocesses the MNIST digits dataset.
   - Scales features using `MinMaxScaler`.
   - Splits the data into training and testing sets.
2. **Model Implementation**:
   - `initialize_parameters`: Initializes weights and biases for each layer.
   - `forward_propagation`: Computes the predicted output using softmax activation for the final layer.
   - `compute_cost`: Calculates the categorical cross-entropy loss.
   - `gradient_descent`: Updates parameters using gradient descent.
   - `learning_rate_decay`: Implements learning rate decay over epochs.

- `nn_model` : Trains the model using the above functions.
- `predict` : Makes predictions using the trained model.

3. **Evaluation**:
   - Evaluates the model using classification report and accuracy score.
   - Displays confusion matrix.
   - Visualizes misclassified examples.

Neural Network Model with a Single Perceptron and One Input Node

In [ ]:
```
# Import necessary libraries
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import tensorflow as tf
from sklearn.datasets import make_regression
```

```
2024-05-25 20:09:49.694609: I tensorflow/core/platform/cpu_feature_guard.cc:182] This TensorFlow binary is optimized to use available CPU instructions in performance-critical operations.
To enable the following instructions: SSE4.1 SSE4.2 AVX AVX2 FMA, in other operations, rebuild TensorFlow with the appropriate compiler flags.
```

In [ ]:
```
# Generate synthetic data for regression using sklearn's make_regression
# This creates a dataset with 30 samples and 1 feature with some noise added
X, y = make_regression(n_samples=30, n_features=1, random_state=1, noise=20)

# Transpose X to match the expected input shape for our model
X = X.T
# Reshape y to be a row vector
y = np.reshape(y, (1, -1))

# Print the shapes of X and y to verify their dimensions
print('The shape of X is: ' + str(X.shape))
print('The shape of y is: ' + str(y.shape))
```

```
The shape of X is: (1, 30)
The shape of y is: (1, 30)
```

In [ ]:
```
def initialize_parameters(n_x, n_y):
    """
    Initialize parameters for the neural network.

    Arguments:
    n_x -- size of the input layer
    n_y -- size of the output layer

    Returns:
    W -- initialized weight matrix of shape (n_y, n_x)
    b -- initialized bias vector of shape (n_y, 1)
    """
    W = tf.Variable(tf.random.normal((n_y, n_x)) * 0.1)  # Small random values for weights
    b = tf.Variable(tf.zeros((n_y, 1)))  # Biases initialized to zero
    return W, b
```

In [ ]:
```
def forward_propagation(X, W, b):
    """
    Perform forward propagation to predict the output.

    Arguments:
    X -- input data of shape (n_x, number of examples)
    W -- weight matrix of shape (n_y, n_x)
    b -- bias vector of shape (n_y, 1)

    Returns:
    y_hat -- predicted output
    """
    y_hat = W @ X + b  # Linear combination of inputs and weights plus bias
    return y_hat
```

In [ ]:
```
def compute_cost(y, y_hat):
    """
    Compute the cost using mean squared error.

    Arguments:
    y -- true "label" vector
    y_hat -- predicted output vector

    Returns:
    cost -- mean squared error cost
    """
    cost = tf.reduce_mean((y - y_hat) ** 2) / 2  # Mean squared error cost function
    return cost
```

In [ ]:
```
def gradient_descent(W, b, dj_dw, dj_db, learning_rate):
    """
```

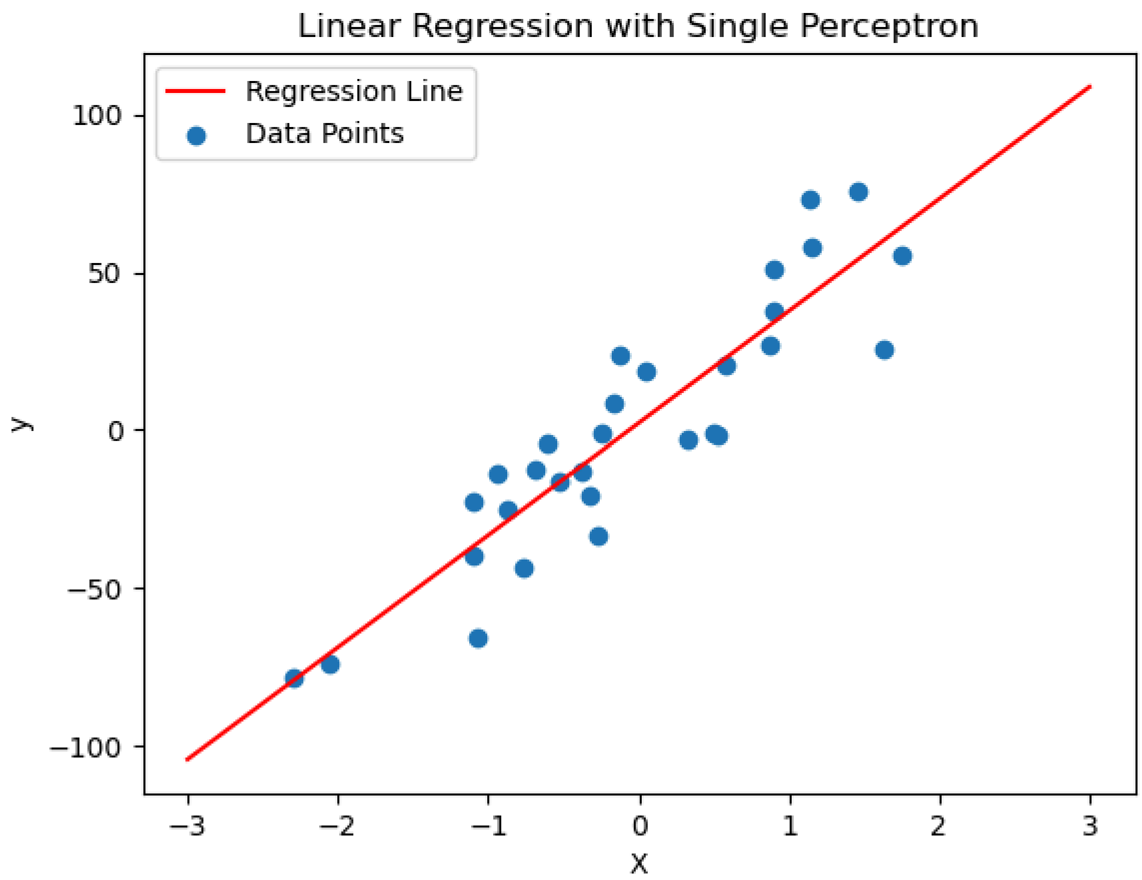
```
    Update parameters using gradient descent.

    Arguments:
    W -- weight matrix
    b -- bias vector
    dj_dw -- gradient of the cost with respect to W
    dj_db -- gradient of the cost with respect to b
    learning_rate -- learning rate for gradient descent

    Returns:
    W -- updated weight matrix
    b -- updated bias vector
    """
    W.assign_sub(learning_rate * dj_dw)  # Update weights
    b.assign_sub(learning_rate * dj_db)  # Update biases
    return W, b
```

In [ ]:
```
def nn_model(X, y, n_x, n_y, epochs, learning_rate, print_cost=True):
    """
    Train the neural network model.

    Arguments:
    X -- input data
    y -- true "label" vector
    n_x -- size of the input layer
    n_y -- size of the output layer
    epochs -- number of epochs to train the model
    learning_rate -- learning rate for gradient descent
    print_cost -- if True, print the cost every 10 epochs

    Returns:
    W -- trained weight matrix
    b -- trained bias vector
    """
    W, b = initialize_parameters(n_x, n_y)  # Initialize parameters

    for epoch in range(epochs):
        with tf.GradientTape() as tape:
            y_hat = forward_propagation(X, W, b)  # Forward propagation
            cost = compute_cost(y, y_hat)  # Compute cost

            if epoch % 10 == 0 and print_cost:
                print(f'Epoch:{epoch}, Cost: {cost}')

        dj_dw, dj_db = tape.gradient(cost, [W, b])  # Compute gradients
        W, b = gradient_descent(W, b, dj_dw, dj_db, learning_rate)  # Update parameters

    W = W.numpy()  # Convert TensorFlow variables to NumPy arrays
    b = b.numpy()
    return W, b
```

In [ ]:
```
# Set hyperparameters
LEARNING_RATE = 0.05
EPOCHS = 100
n_x = X.shape[0]  # Number of input features
n_y = 1  # Number of output features (single output)

# Train the model and get the final parameters
W, b = nn_model(X, y, n_x, n_y, EPOCHS, LEARNING_RATE, print_cost=True)
```

```
Epoch:0, Cost: 792.9966430664062
2024-05-25 20:09:52.945854: I tensorflow/core/common_runtime/process_util.cc:146] Creating new thread pool with defa
ult inter op setting: 2. Tune using inter_op_parallelism_threads for best performance.
Epoch:10, Cost: 370.9805603027344
Epoch:20, Cost: 222.5522918701172
Epoch:30, Cost: 170.1344451904297
Epoch:40, Cost: 151.54661560058594
Epoch:50, Cost: 144.92808532714844
Epoch:60, Cost: 142.5618438720703
Epoch:70, Cost: 141.7124481201172
Epoch:80, Cost: 141.40638732910156
Epoch:90, Cost: 141.29568481445312
```

In [ ]:
```
# Generate a range of x values for plotting the regression line
x = np.linspace(-3, 3, 50)
# Compute the predicted y values using the trained parameters
y_pred = W @ x.reshape(1, -1) + b

# Plot the regression line and the data points
plt.plot(x, y_pred[0], c='r', label='Regression Line')
plt.scatter(X, y, label='Data Points')
plt.xlabel('X')
plt.ylabel('y')
plt.title('Linear Regression with Single Perceptron')
plt.legend()
plt.show()
```



Neural Network Model with a Single Perceptron and Two Input Nodes

In [ ]:
```
import seaborn as sns
from sklearn.model_selection import train_test_split
```

In [ ]:
```
# Load house prices dataset
df = pd.read_csv('house_prices_train.csv', index_col='Id')
```

In [ ]:
```
# Check and print the percentage of missing values in each column
for column in df.columns:
    if np.sum(df[column].isna()) / 1460 > 0:
        print(f'{np.sum(df[column].isna()) / 1460 * 100 :0.2f}% of "{column}" is null.')
```

```
17.74% of "LotFrontage" is null.
93.77% of "Alley" is null.
59.73% of "MasVnrType" is null.
0.55% of "MasVnrArea" is null.
2.53% of "BsmtQual" is null.
2.53% of "BsmtCond" is null.
2.60% of "BsmtExposure" is null.
2.53% of "BsmtFinType1" is null.
2.60% of "BsmtFinType2" is null.
0.07% of "Electrical" is null.
47.26% of "FireplaceQu" is null.
5.55% of "GarageType" is null.
5.55% of "GarageYrBlt" is null.
5.55% of "GarageFinish" is null.
5.55% of "GarageQual" is null.
5.55% of "GarageCond" is null.
99.52% of "PoolQC" is null.
80.75% of "Fence" is null.
96.30% of "MiscFeature" is null.
```

In [ ]:
```
# List to store columns with more than 6% missing values
cols = []
for column in df.columns:
    if np.sum(df[column].isna()) / 1460 * 100 > 6:
        cols.append(column)

# Drop columns with more than 6% missing values and rows with any missing values
df_new = df.drop(columns=cols)
df_new = df_new.dropna()

# Print the shapes of the original and cleaned datasets
print(f'df.shape: {df.shape}, df_new.shape: {df_new.shape}')
```

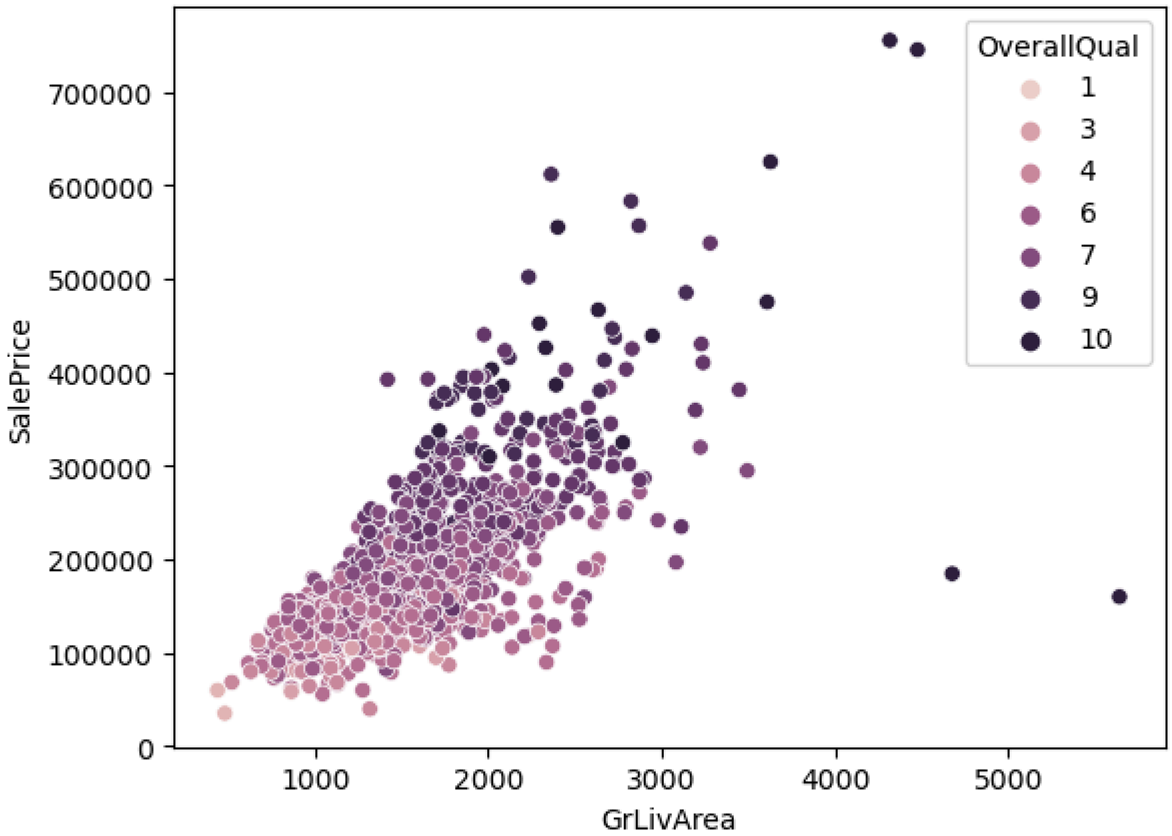
```
df.shape: (1460, 80), df_new.shape: (1338, 73)
```

In [ ]:
```
# Number of features to select for the model
n_features = 2

# Convert categorical variables to dummy variables
df_new = pd.get_dummies(df_new, drop_first=True)

# Select top correlated features with the target variable 'SalePrice'
columns = df_new.corrwith(df_new['SalePrice']).abs().nlargest(n_features + 1).keys()[1:]
columns
```

Out[ ]: Index(['OverallQual', 'GrLivArea'], dtype='object')

In [ ]:
```
# Assign the selected features to X and target variable to y
X = df_new[columns]
y = df_new['SalePrice']

# Visualize the relationship between selected features and target variable
sns.scatterplot(data=df, x='GrLivArea', y=y, hue='OverallQual')
```

Out[ ]: <Axes: xlabel='GrLivArea', ylabel='SalePrice'>



In [ ]:
```
# Split the data into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.33, random_state=101)
```

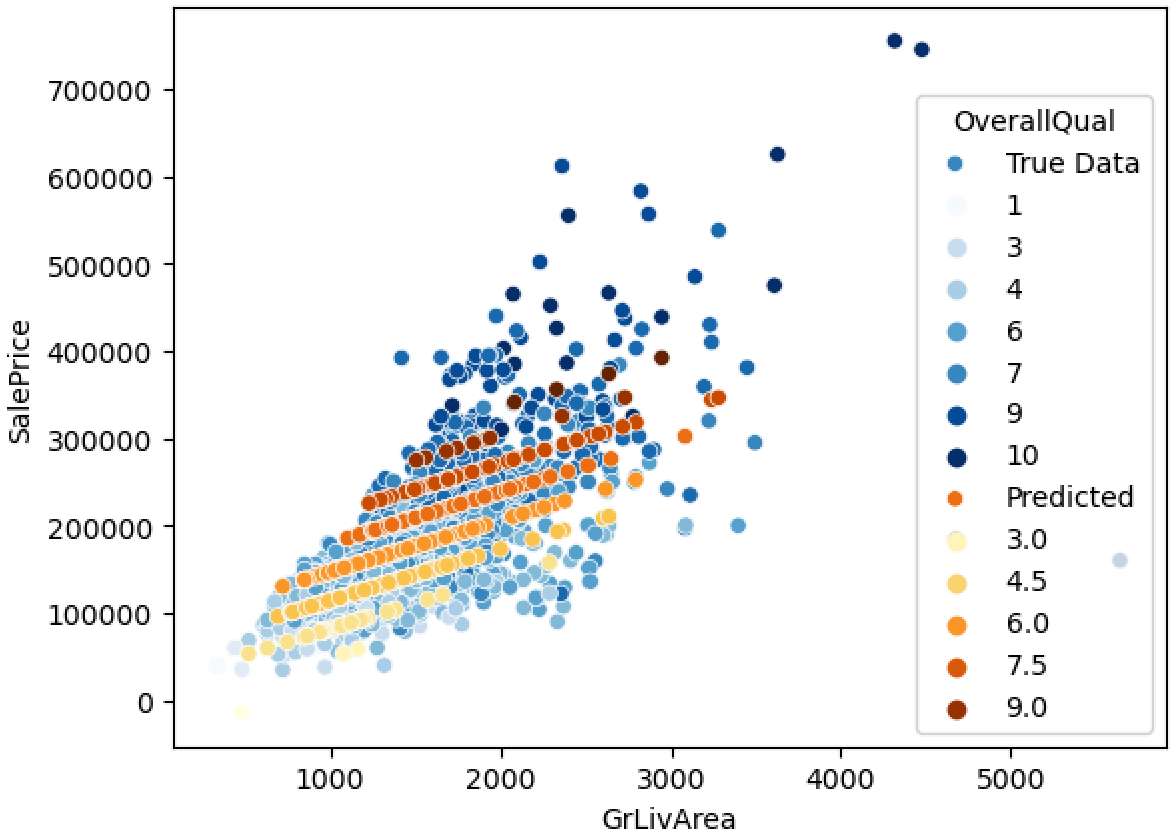
In [ ]:
```
# Normalize the features for training and testing sets
X_train_norm = ((X_train - X.mean()) / X.std()).T.to_numpy()
X_test_norm = ((X_test - X.mean()) / X.std()).T.to_numpy()

# Normalize the target variable for training set
y_train_norm = (y_train - y.mean()) / y.std()
y_train_norm = np.reshape(y_train_norm, (1, -1))
y_test = np.reshape(y_test, (1, -1))
```

In [ ]:
```
# Set hyperparameters
LEARNING_RATE = 0.05
EPOCHS = 100
n_x = X_train_norm.shape[0]  # Number of input features
n_y = 1  # Number of output features (single output)

# Train the model and get the final parameters
W, b = nn_model(X_train_norm, y_train_norm, n_x, n_y, EPOCHS, LEARNING_RATE, print_cost=True)
```

```
Epoch:0, Cost: 0.4375437796115875
Epoch:10, Cost: 0.20974504947662354
Epoch:20, Cost: 0.16759221255779266
Epoch:30, Cost: 0.1587148904800415
Epoch:40, Cost: 0.15617524087429047
Epoch:50, Cost: 0.15507946908473969
Epoch:60, Cost: 0.15445576608181
Epoch:70, Cost: 0.154059499502182
Epoch:80, Cost: 0.15379902720451355
Epoch:90, Cost: 0.15362612903118134
```

In [ ]:
```
# Visualize the true data points
sns.scatterplot(data=df, x='GrLivArea', y='SalePrice', hue='OverallQual', palette='Blues', label='True Data')

# Predict the normalized target values using the trained model
y_values_norm = W @ X_test_norm + b
# Convert the normalized predicted values back to original scale
y_values = y_values_norm * y.std() + y.mean()

# Create a dataframe with the test features and predicted target values
df2 = pd.DataFrame(np.hstack([X_test, y_values.T]), columns=[*columns, 'SalePrice'])

# Visualize the predicted data points
sns.scatterplot(data=df2, x='GrLivArea', y='SalePrice', hue='OverallQual', palette='YlOrBr', label='Predicted')
```

Out[ ]: <Axes: xlabel='GrLivArea', ylabel='SalePrice'>



In [ ]:
```python
# Evaluate the model using RMSE and R² score
from sklearn.metrics import mean_squared_error, r2_score

RMSE = np.sqrt(mean_squared_error(y_test.T, y_values.T))
print(f'RMSE : {RMSE}')
print(f'R2 Score: {r2_score(y_test.T, y_values.T)}')
```

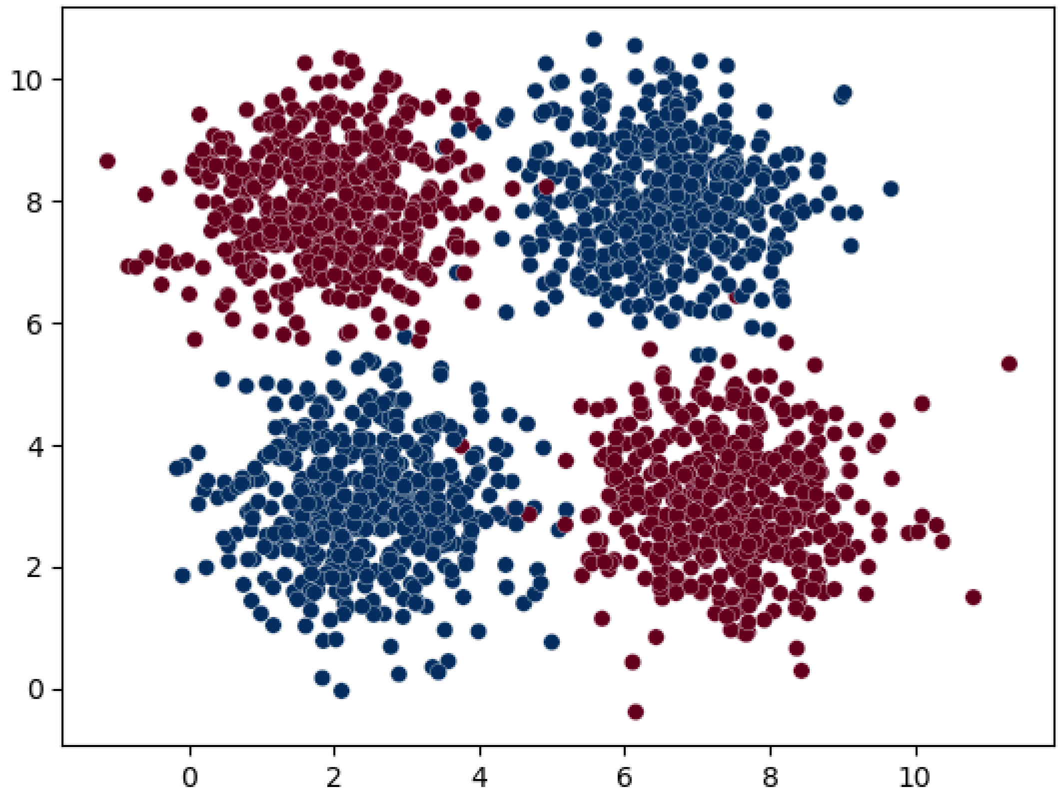
```
RMSE : 40488.32085402417
R2 Score: 0.7128511001638731
```

Neural Network Model with One Hidden Layer

In [ ]:
```python
# Import necessary libraries
import numpy as np
import matplotlib.pyplot as plt
import tensorflow as tf
import pandas as pd
from sklearn.datasets import make_blobs
```

In [ ]:
```python
# Generate synthetic data for classification using sklearn's make_blobs
m = 2000
X, y = make_blobs(m, centers=([2.5, 3], [6.5, 8], [2, 8], [7.5, 3]), random_state=0)
y[(y == 0) | (y == 1)] = 1
y[(y == 2) | (y == 3)] = 0

# Transpose X to match the expected input shape for our model
X = X.T
# Reshape y to be a row vector
y = np.reshape(y, (1, -1))

# Plot the data points
plt.scatter(X[0, :], X[1, :], c=y[0, :], cmap='RdBu', edgecolors='white', linewidths=0.2);

# Print the shapes of X and y to verify their dimensions
print('The shape of X is: ' + str(X.shape))
print('The shape of y is: ' + str(y.shape))
```

```
The shape of X is: (2, 2000)
The shape of y is: (1, 2000)
```



In [ ]:
```
def initialize_parameters(n_x, n_h, n_y):
    """
    Initialize parameters for the neural network with two layers.

    Arguments:
    n_x -- size of the input layer
    n_h -- size of the hidden layer
    n_y -- size of the output layer

    Returns:
    params -- dictionary containing initialized parameters
    """
    W1 = tf.Variable(tf.random.normal(shape=(n_h, n_x)) * tf.sqrt(2/n_x))  # He initialization for weights
    b1 = tf.Variable(tf.zeros(shape=(n_h, 1)))  # Biases initialized to zero
    W2 = tf.Variable(tf.random.normal(shape=(n_y, n_h)) * tf.sqrt(2/n_y))  # He initialization for weights
    b2 = tf.Variable(tf.zeros(shape=(n_y, 1)))  # Biases initialized to zero

    params = {
        'W1': W1,
        'b1': b1,
        'W2': W2,
        'b2': b2
    }

    return params
```

In [ ]:
```
def sigmoid(z):
    """
    Compute the sigmoid activation function.

    Arguments:
    z -- input to the sigmoid function

    Returns:
    sigmoid(z) -- output of the sigmoid function
    """
    return 1 / (1 + tf.exp(-z))
```

In [ ]:
```
def forward_propagation(X, params):
    """
    Perform forward propagation to predict the output.

    Arguments:
    X -- input data of shape (n_x, number of examples)
    params -- dictionary containing initialized parameters

    Returns:
    y_hat -- predicted output
    """
    W1 = params['W1']
    b1 = params['b1']
    W2 = params['W2']
    b2 = params['b2']

    Z1 = W1 @ X + b1  # Linear transformation
    A1 = tf.nn.relu(Z1)  # ReLU activation function
    Z2 = W2 @ A1 + b2  # Linear transformation
    y_hat = sigmoid(Z2)  # Sigmoid activation function

    return y_hat
```

In [ ]:
```python
def compute_cost(y, y_hat):
    """
    Compute the cost using binary cross-entropy.

    Arguments:
    y -- true "label" vector
    y_hat -- predicted output vector

    Returns:
    cost -- binary cross-entropy cost
    """
    logloss = tf.keras.losses.binary_crossentropy(y, y_hat)  # Binary cross-entropy loss function
    return tf.reduce_mean(logloss)
```

In [ ]:
```python
def gradient_descent(params, grads, learning_rate):
    """
    Update parameters using gradient descent.

    Arguments:
    params -- dictionary containing parameters
    grads -- dictionary containing gradients of the cost with respect to parameters
    learning_rate -- learning rate for gradient descent

    Returns:
    params -- updated parameters
    """
    for i in params.keys():
        params[i].assign_sub(learning_rate * grads[i])  # Update parameters using gradients

    return params
```

In [ ]:
```python
def nn_model(X, y, n_x, n_h, n_y, epochs, learning_rate):
    """
    Train the neural network model.

    Arguments:
    X -- input data
    y -- true "label" vector
    n_x -- size of the input layer
    n_h -- size of the hidden layer
    n_y -- size of the output layer
    epochs -- number of epochs to train the model
    learning_rate -- learning rate for gradient descent

    Returns:
    params -- trained parameters
    """
    params = initialize_parameters(n_x, n_h, n_y)  # Initialize parameters
    for epoch in range(epochs):
        with tf.GradientTape() as tape:
            y_hat = forward_propagation(X, params)  # Forward propagation
            cost = compute_cost(y, y_hat)  # Compute cost

            if epoch % 100 == 0:
                print(f'Epoch:{epoch}, Cost: {cost}')

        grads = tape.gradient(cost, params)  # Compute gradients
        params = gradient_descent(params, grads, learning_rate)  # Update parameters

    return params
```

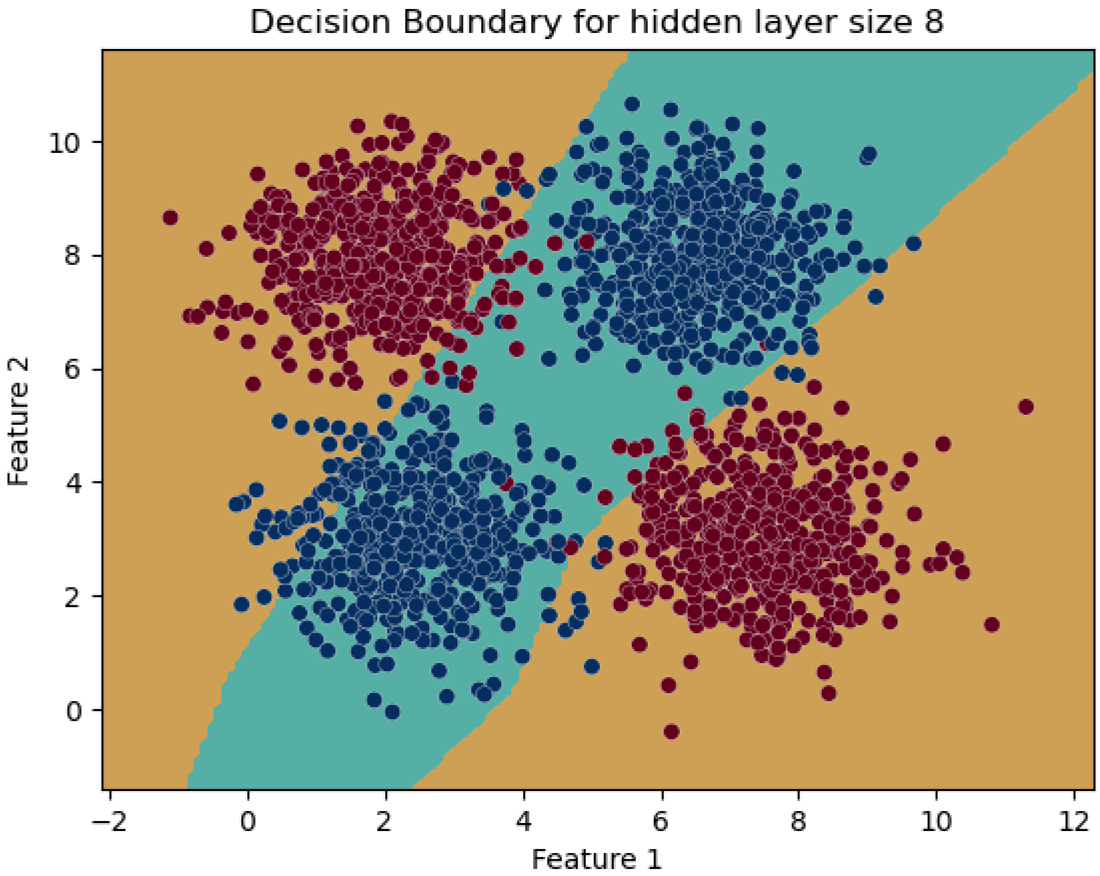
In [ ]:
```python
# Set hyperparameters
LEARNING_RATE = 0.08
EPOCHS = 1000
n_x = X.shape[0]  # Number of input features
n_h = 8  # Number of units in hidden layer
n_y = y.shape[0]  # Number of output units

# Train the model and get the final parameters
params = nn_model(X, y, n_x, n_h, n_y, EPOCHS, LEARNING_RATE)
```

```
Epoch:0, Cost: 4.5431084632873535
Epoch:100, Cost: 0.34587621688842773
Epoch:200, Cost: 0.291795551776886
Epoch:300, Cost: 0.25851231813430786
Epoch:400, Cost: 0.2341194450855255
Epoch:500, Cost: 0.21590639650821686
Epoch:600, Cost: 0.20175601541996002
Epoch:700, Cost: 0.1903807371854782
Epoch:800, Cost: 0.18064361810684204
Epoch:900, Cost: 0.17172929644584656
```

In [ ]:
```python
def predict(X, params):
    """
    Make predictions using the trained model.
```

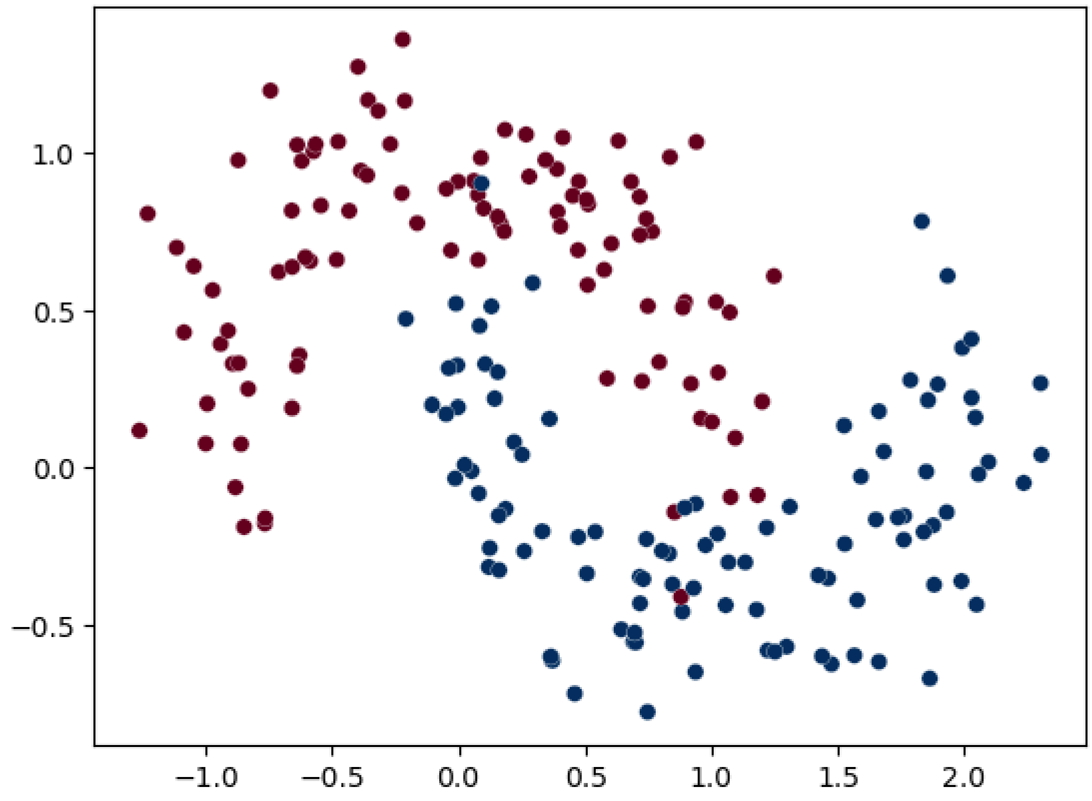
```
    Arguments:
    X -- input data
    params -- trained parameters

    Returns:
    predictions -- array of predictions
    """
    A2 = forward_propagation(X, params)  # Forward propagation
    predictions = A2 > 0.5  # Convert probabilities to binary predictions

    return predictions.numpy()
```

In [ ]:
```
def plot_decision_boundary(X, y, params):
    """
    Plot the decision boundary of the trained model.

    Arguments:
    X -- input data
    y -- true "label" vector
    params -- trained parameters
    """
    min1, max1 = X[0, :].min() - 1, X[0, :].max() + 1
    min2, max2 = X[1, :].min() - 1, X[1, :].max() + 1

    # Generate a grid of points within the feature space
    x1grid = np.arange(min1, max1, 0.1)
    x2grid = np.arange(min2, max2, 0.1)
    xx, yy = np.meshgrid(x1grid, x2grid)
    r1, r2 = xx.flatten(), yy.flatten()
    r1, r2 = r1.reshape((1, len(r1))), r2.reshape((1, len(r2)))
    grid = np.vstack((r1,r2))

    # Make predictions on the grid points
    predictions = predict(grid, params)
    zz = predictions.reshape(xx.shape)

    # Plot decision boundary and data points
    plt.contourf(xx, yy, zz, cmap='BrBG')
    plt.scatter(X[0, :], X[1, :], c=y[0, :], cmap='RdBu', edgecolors='white', linewidths=0.2)
    plt.title("Decision Boundary for hidden layer size " + str(n_h));
    plt.xlabel('Feature 1')
    plt.ylabel('Feature 2')
    plt.show()

# Plot decision boundary for the synthetic data
plot_decision_boundary(X, y, params)
```



Additional dataset

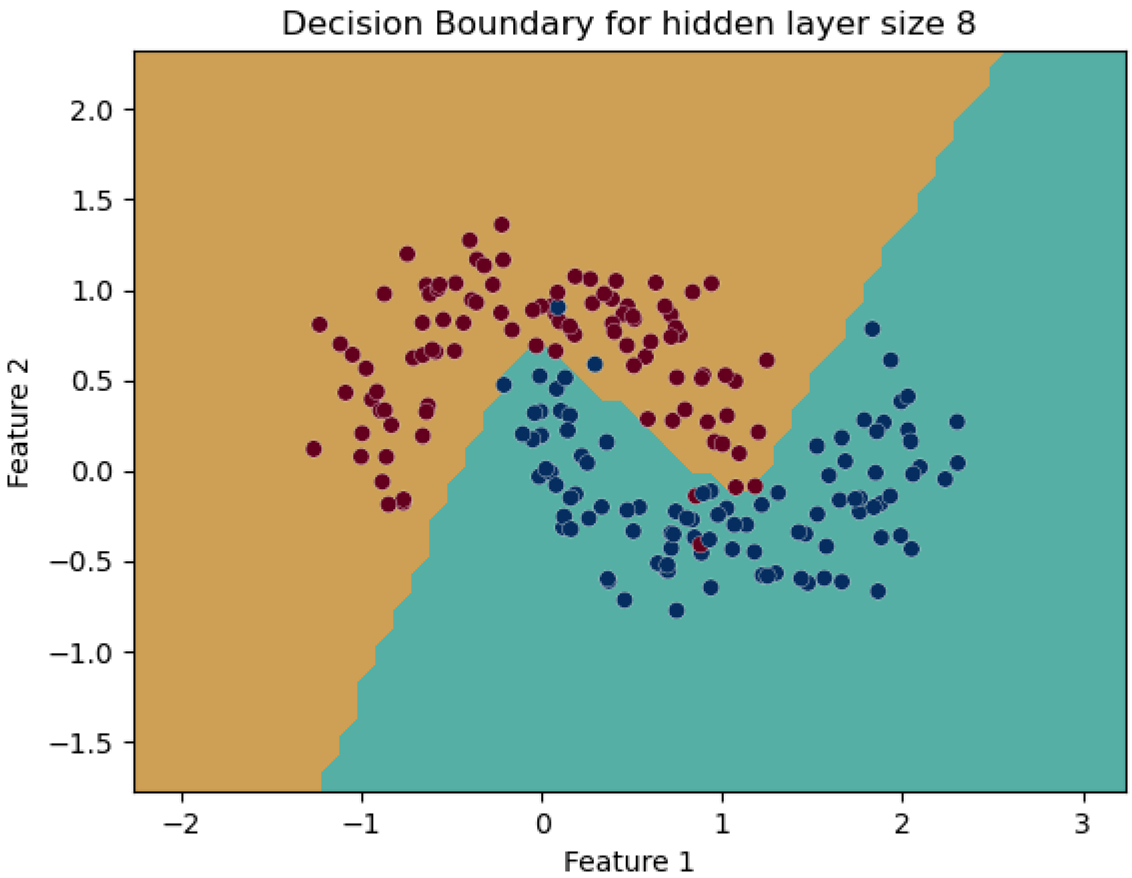
In [ ]:
```
data = pd.read_csv('Arcs.csv')
X = data.iloc[:, :-1].T.to_numpy()
y = np.reshape(data.iloc[:, -1], (1, -1))
plt.scatter(X[0, :], X[1, :], c=y[0, :], cmap='RdBu', edgecolors='white', linewidths=0.2);
```



```
In [ ]: LEARNING_RATE = 0.5
        EPOCHS = 1000

        n_x = X.shape[0]  # Number of input features
        n_h = 8  # Number of units in hidden layer
        n_y = y.shape[0]  # Number of output units

        params = nn_model(X, y, n_x, n_h, n_y, EPOCHS, LEARNING_RATE)
```

```
Epoch:0, Cost: 2.0575194358825684
Epoch:100, Cost: 0.2534295320510864
Epoch:200, Cost: 0.20632652938365936
Epoch:300, Cost: 0.1616419553756714
Epoch:400, Cost: 0.13303911685943604
Epoch:500, Cost: 0.11789369583129883
Epoch:600, Cost: 0.10948474705219269
Epoch:700, Cost: 0.10379713773727417
Epoch:800, Cost: 0.09918693453073502
Epoch:900, Cost: 0.09636596590280533
```

```
In [ ]: plot_decision_boundary(X, y, params);
```



Neural Network Model with Multiple Layers

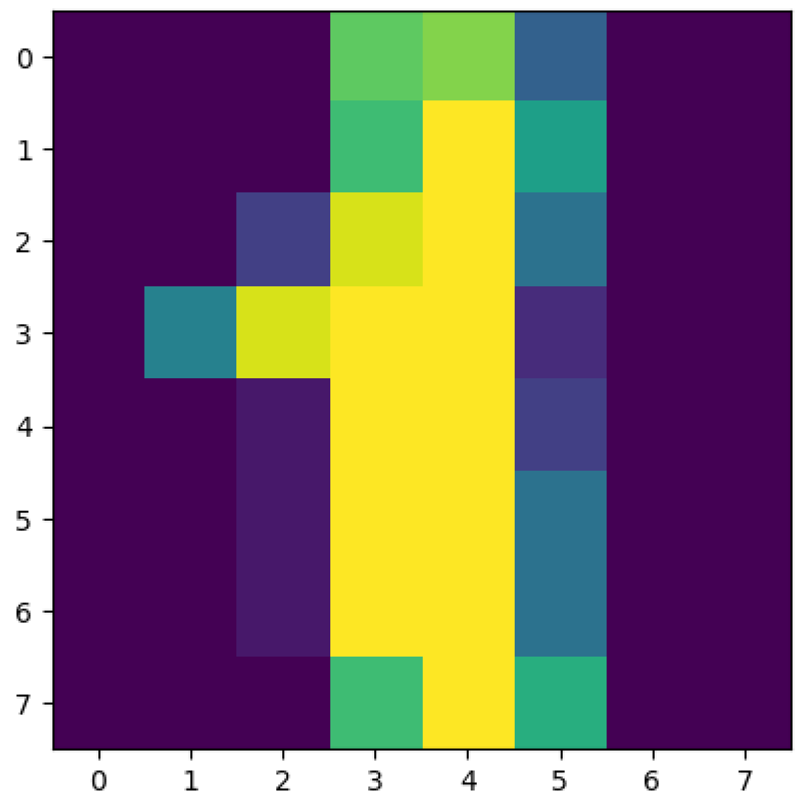
```
In [ ]: # Import necessary libraries
        import numpy as np
        import pandas as pd
        import tensorflow as tf
        import matplotlib.pyplot as plt
        from sklearn.datasets import load_digits
        from sklearn.preprocessing import MinMaxScaler
        from sklearn.model_selection import train_test_split
```

In [ ]:
```python
# Load the digits dataset
digits = load_digits()

# Extract features (X) and labels (y) from the dataset
X = digits['data']
y = digits['target']

# Print the shape of X and y to verify their dimensions
print(f'X.shape: {X.shape}')
print(f'y.shape: {y.shape}')
```

```
X.shape: (1797, 64)
y.shape: (1797,)
```

In [ ]:
```python
# Visualize a sample image from the dataset
plt.imshow(X[1].reshape(8, 8))
```

Out[ ]: <matplotlib.image.AxesImage at 0x7b4bface28d0>



In [ ]:
```python
# Scale the features to the range [0, 1]
scaler = MinMaxScaler()
X = scaler.fit_transform(X)
```

In [ ]:
```python
# Split the data into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.33, random_state=42)
```

In [ ]:
```python
# Transpose the data to match the expected input shape for our model
X_train = X_train.T
X_test = X_test.T
y_train = tf.keras.utils.to_categorical(y_train).T

# Print the shapes of training and testing data to verify their dimensions
print(f'X_train.shape: {X_train.shape}')
print(f'X_test.shape: {X_test.shape}')
print(f'y_train.shape: {y_train.shape}')
print(f'y_test.shape: {y_test.shape}')
```

```
X_train.shape: (64, 1203)
X_test.shape: (64, 594)
y_train.shape: (10, 1203)
y_test.shape: (594,)
```

In [ ]:
```python
def initialize_parameters(layer_dims):
    """
    Initialize parameters for the neural network with multiple layers.

    Arguments:
    layer_dims -- list containing the number of units in each layer

    Returns:
    params -- dictionary containing initialized parameters
    """
    params = {}
    for i in range(1, len(layer_dims)):
        params[f'W{i}'] = tf.Variable(tf.random.normal(shape=(layer_dims[i], layer_dims[i-1])) * tf.sqrt(2/layer_di
        params[f'b{i}'] = tf.Variable(tf.zeros(shape=(layer_dims[i], 1)))

    return params
```

In [ ]:
```python
def forward_propagation(X, params):
    """
    Perform forward propagation to predict the output.
```

```
    Arguments:
    X -- input data of shape (n_x, m)
    params -- dictionary containing initialized parameters

    Returns:
    y_hat -- predicted output
    """
    l = len(params) // 2
    A = X
    for i in range(1, l):
        Z = params[f'W{i}'] @ A + params[f'b{i}']
        A = tf.nn.relu(Z)

    Z = params[f'W{l}'] @ A + params[f'b{l}']
    y_hat = tf.nn.softmax(Z)

    return y_hat
```

In [ ]:
```
def compute_cost(y, y_hat):
    """
    Compute the cost using categorical cross-entropy.

    Arguments:
    y -- true labels
    y_hat -- predicted probabilities

    Returns:
    cost -- categorical cross-entropy cost
    """
    loss = tf.keras.losses.categorical_crossentropy(y, y_hat)
    return tf.reduce_mean(loss)
```

In [ ]:
```
def gradient_descent(params, grads, learning_rate):
    """
    Update parameters using gradient descent.

    Arguments:
    params -- dictionary containing parameters
    grads -- dictionary containing gradients of the cost with respect to parameters
    learning_rate -- learning rate for gradient descent

    Returns:
    params -- updated parameters
    """
    for i in params.keys():
        params[i].assign_sub(learning_rate * grads[i])

    return params
```

In [ ]:
```
def learning_rate_decay(learning_rate, epoch_num, decay_rate=1, time_interval=1000):
    """
    Decay the learning rate over time.

    Arguments:
    learning_rate -- initial learning rate
    epoch_num -- current epoch number
    decay_rate -- rate of decay
    time_interval -- time interval for decay

    Returns:
    updated_learning_rate -- decayed learning rate
    """
    updated_learning_rate = learning_rate / (1 + decay_rate * epoch_num / time_interval)
    return updated_learning_rate
```

In [ ]:
```
def nn_model(X, y, layer_dims, epochs, learning_rate, decay_rate=1, print_cost=False):
    """
    Train the neural network model.

    Arguments:
    X -- input data
    y -- true labels
    layer_dims -- list containing the number of units in each layer
    epochs -- number of epochs to train the model
    learning_rate -- initial learning rate
    decay_rate -- rate of decay for learning rate
    print_cost -- whether to print the cost during training

    Returns:
    params -- trained parameters
    """
    params = initialize_parameters(layer_dims)
    for epoch in range(epochs):
        with tf.GradientTape() as tape:
            y_hat = forward_propagation(X, params)
```

```python
            cost = compute_cost(y, y_hat)

            if print_cost and epoch % 100 == 0:
                print(f'Epoch:{epoch}, Cost: {cost:0.2f}')

        grads = tape.gradient(cost, params)
        decayed_learning_rate = learning_rate_decay(learning_rate, epoch, decay_rate)
        params = gradient_descent(params, grads, decayed_learning_rate)

    return params
```

In [ ]:

```python
# Set hyperparameters
LEARNING_RATE = 0.025
EPOCHS = 1000
LAYER_DIMS = [X_train.shape[0], 64, 32, y_train.shape[0]]  # Number of units in each layer

# Train the model and get the final parameters
params = nn_model(X_train, y_train, LAYER_DIMS, EPOCHS, LEARNING_RATE, decay_rate=1.5, print_cost=True)
```

```
Epoch:0, Cost: 865.67
Epoch:100, Cost: 597.35
Epoch:200, Cost: 581.50
Epoch:300, Cost: 579.44
Epoch:400, Cost: 578.02
Epoch:500, Cost: 577.70
Epoch:600, Cost: 577.15
Epoch:700, Cost: 576.96
Epoch:800, Cost: 576.86
Epoch:900, Cost: 576.76
```

In [ ]:

```python
def predict(X, params):
    """
    Make predictions using the trained model.

    Arguments:
    X -- input data
    params -- trained parameters

    Returns:
    predictions -- array of predictions
    """
    y_hat = forward_propagation(X, params)
    predictions = np.argmax(y_hat, axis=0)

    return predictions
```

In [ ]:

```python
# Make predictions using the trained model
y_pred = predict(X_test, params)
```

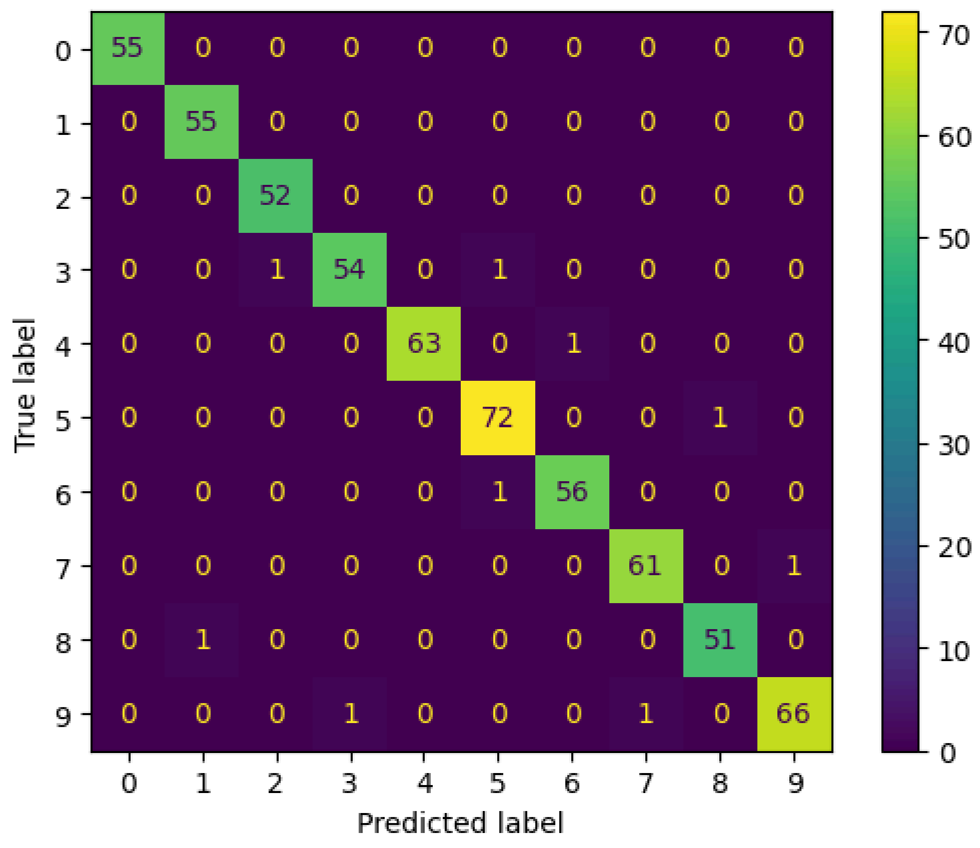
In [ ]:

```python
from sklearn.metrics import accuracy_score, classification_report, confusion_matrix, ConfusionMatrixDisplay

# Evaluate the model using classification report and confusion matrix
print(classification_report(y_test, y_pred))
ConfusionMatrixDisplay(confusion_matrix(y_test, y_pred)).plot()
```

```
              precision    recall  f1-score   support

           0       1.00      1.00      1.00        55
           1       0.98      1.00      0.99        55
           2       0.98      1.00      0.99        52
           3       0.98      0.96      0.97        56
           4       1.00      0.98      0.99        64
           5       0.97      0.99      0.98        73
           6       0.98      0.98      0.98        57
           7       0.98      0.98      0.98        62
           8       0.98      0.98      0.98        52
           9       0.99      0.97      0.98        68

    accuracy                           0.98       594
   macro avg       0.99      0.99      0.99       594
weighted avg       0.98      0.98      0.98       594
```

Out[ ]:

```
<sklearn.metrics._plot.confusion_matrix.ConfusionMatrixDisplay at 0x7b4c14375950>
```



In [ ]:
```
# Make predictions using the trained model
y_pred_train = predict(X_train, params)

# Calculate and print the accuracy score on the training set
accuracy_train = np.mean(y_pred_train == y_train.argmax(axis=0))
print(f'Training Accuracy: {accuracy_train}')
```

```
Training Accuracy: 1.0
```

In [ ]:
```
# Calculate and print the accuracy score on the test set
accuracy_test = np.mean(y_pred == y_test)
print(f'Test Accuracy: {accuracy_test}')
```

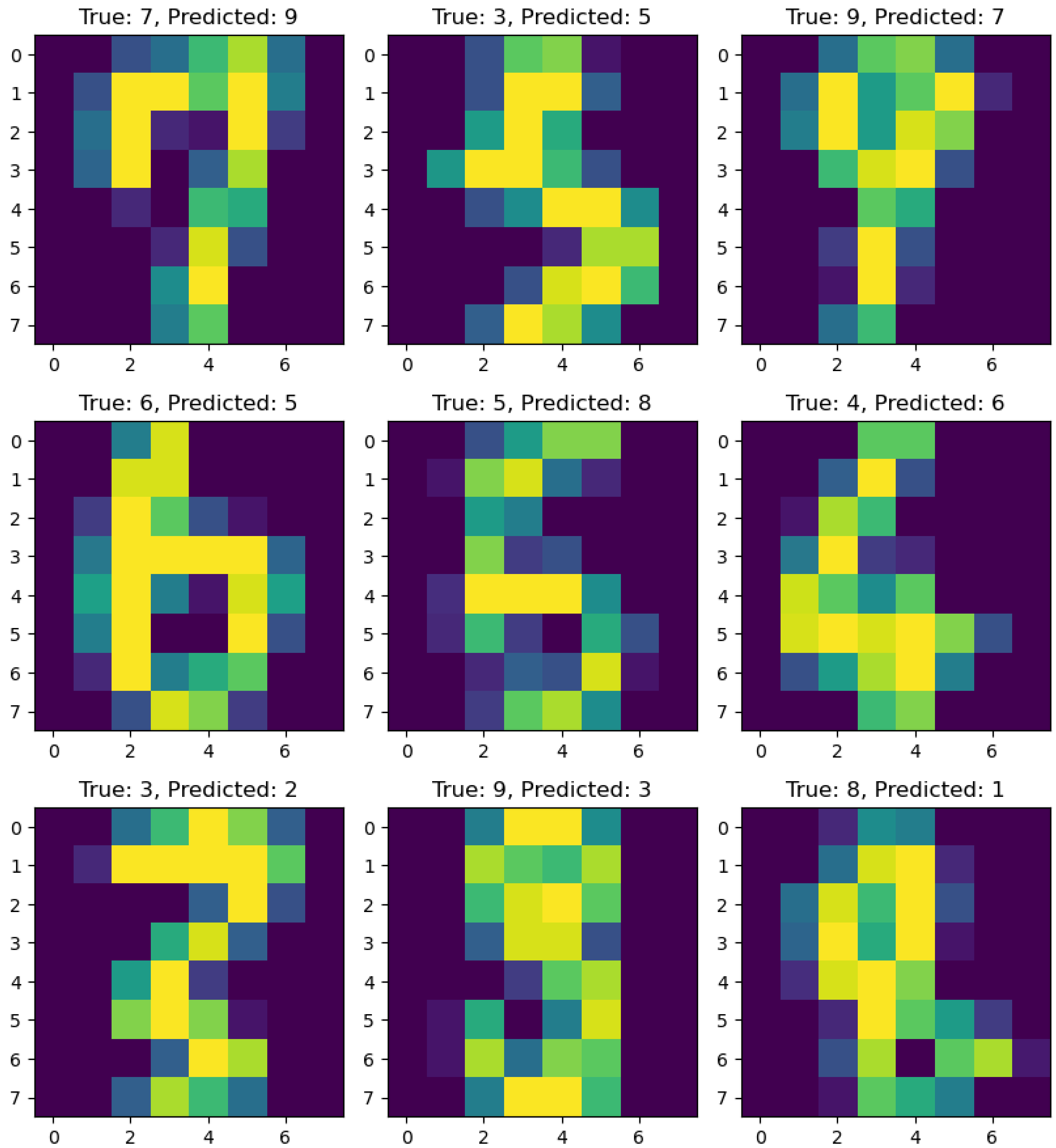
```
Test Accuracy: 0.9848484848484849
```

In [ ]:
```
def find_closest_factors(number):
    """
    Find the closest factors of a number.

    Arguments:
    number -- input number

    Returns:
    a -- one factor
    b -- another factor
    """
    a = int(np.sqrt(number))
    for i in range(a, 0, -1):
        if number % i == 0:
            return i, number // i
```

In [ ]:
```
# Calculate the number of misclassifications
errors = np.sum(y_test != y_pred)

# Find and print the closest factors of the number of misclassifications
a, b = find_closest_factors(errors)
print(f'Number of Misclassifications: {errors} = {a} * {b}')
```

```
Number of Misclassifications: 9 = 3 * 3
```

In [ ]:
```
# Visualize some of the misclassified digits
# Plot a grid of images of misclassified digits along with their true and predicted labels
misclassified_indices = np.where(y_test != y_pred)[0]
num_misclassified = len(misclassified_indices)

# Create subplots for each misclassified digit
fig, axs = plt.subplots(a, b, figsize=(b*2.75, a*3))
for i, ax in enumerate(axs.flat):
    ax.imshow(X_test[:, misclassified_indices[i]].reshape(8, 8))
    ax.set_title(f'True: {y_test[misclassified_indices[i]]}, Predicted: {y_pred[misclassified_indices[i]]}')

# Show the plot
plt.tight_layout()
plt.show()
```


True: 7, Predicted: 9
True: 3, Predicted: 5
True: 9, Predicted: 7
True: 6, Predicted: 5
True: 5, Predicted: 8
True: 4, Predicted: 6
True: 3, Predicted: 2
True: 9, Predicted: 3
True: 8, Predicted: 1
0
1
2
3
4
5
6
7
0
2
4
6