KARADENİZ TEKNİK ÜNİVERSİTESİ
FEN BİLİMLERİ ENSTİTÜSÜ
BİLGİSAYAR MÜHENDİSLİĞİ ANABİLİM DALI

(BIL5050 - Artificial Neural Systems)

(**Report:** Multi-Layer Multi - Class Connected Neural Network)

Prepared by: 442599 MOHAMMAD HAROON RASIKH

Advisor: (Prof.Dr. MURAT EKİNCİ)

Multi - Layer Neural Network Project

It's a simple GUI application based on PyQt5 to interactively train and visualize an MLP. It mainly comprises three components: **inputting the data**, **training of the model**, and **visualization**.

Activation function

Implementing the activation functions (Sigmoid and ReLU) and their derivatives to use in the Multi-Layer Perception (MLP)

- **sigmoid**: Squashes input to a range between 0 and 1.
- **relu**: Keeps positive values and zeroes out negatives.

Derivatives:

- Used for backpropagation to calculate gradients.

```
# Sigmoid and ReLU activation functions
def sigmoid(x):
    return 1 / (1 + (2.718281828459045) ** -x)


def relu(x):
    return [i if i > 0 else 0 for i in x]


# Derivatives for backpropagation
def sigmoid_derivative(x):
    return sigmoid(x) * (1 - sigmoid(x))

def relu_derivative(x):
    return [1 if i > 0 else 0 for i in x]
```

Predict Function

```python
# Predict function for multi-layer model
def predict_multilayer(X, weights, biases, hidden_layer_size):
    # Forward pass through hidden layer
    hidden_layer_input = np.dot(X, weights["hidden"]) + biases["hidden"]
    hidden_layer_output = relu(hidden_layer_input)

    # Forward pass through output layer
    output_layer_input = np.dot(hidden_layer_output, weights["output"]) + biases["output"]
    output = sigmoid(output_layer_input)

    return output
```

Purpose: Implements forward pass of the Multi – Layer Perception

- Hidden layer input = X * W_{hidden} + b_{hidden}
- Apply **ReLU** to hidden layer outputs
- Output layer input = HiddenOutput * W_{output} + b_{output}
- Apply **Sigmoid** for classification probabilities

Training Function

```python
# Training function with backpropagation
def train_multilayer_perceptron(X, y, num_classes, hidden_layer_size=5, learning_rate=0.1, epochs=100, hidden_input_layer=None):
    total_errors = []  # Collect total error for each epoch
    input_size = len(X[0])

    # Initialize weights and biases randomly
    def random_matrix(rows, cols):
        return [[random.uniform(-1, 1) for _ in range(cols)] for _ in range(rows)]

    def random_vector(size):
        return [random.uniform(-1, 1) for _ in range(size)]

    weights = {
        "hidden": random_matrix(input_size, hidden_layer_size),
        "output": random_matrix(hidden_layer_size, num_classes)
    }
    biases = {
        "hidden": random_vector(hidden_layer_size),
        "output": random_vector(num_classes)
    }
```

```python
    # Use hidden_input_layer if provided
    if hidden_input_layer is not None:
        hidden_input = hidden_input_layer
    else:
        hidden_input = [0] * hidden_layer_size

    for epoch in range(epochs):
        epoch_error = 0
        for i in range(len(X)):
            X_i = X[i]
            y_i = y[i]

            # Compute hidden layer inputs and outputs
            hidden_input = []
            for k in range(hidden_layer_size):
                dot_product = 0
                for j in range(input_size):
                    dot_product += X_i[j] * weights["hidden"][j][k]
                hidden_input.append(dot_product + biases["hidden"][k])

            hidden_output = []
```

```python
for val in hidden_input:
    hidden_output.append(max(0, val))  # ReLU activation

# Compute output layer inputs and outputs
output_input = []
for k in range(num_classes):
    dot_product = 0
    for j in range(hidden_layer_size):
        dot_product += hidden_output[j] * weights["output"][j][k]
    output_input.append(dot_product + biases["output"][k])

output = []
for val in output_input:
    output.append(sigmoid(val))  # Sigmoid activation

# Calculate error
output_error = []
for k in range(num_classes):
    output_error.append(y_i[k] - output[k])
```

```python
# Calculate deltas
output_delta = []
for k in range(num_classes):
    output_delta.append(output_error[k] * output[k] * (1 - output[k]))

hidden_error = []
for k in range(hidden_layer_size):
    backprop_error = 0
    for m in range(num_classes):
        backprop_error += output_delta[m] * weights["output"][k][m]
    hidden_error.append(backprop_error)

hidden_delta = []
for k in range(hidden_layer_size):
    hidden_delta.append(hidden_error[k] * (1 if hidden_input[k] > 0 else 0))
```

```python
        # Update weights and biases for output layer
        for j in range(hidden_layer_size):
            for k in range(num_classes):
                weights["output"][j][k] += learning_rate * hidden_output[j] * output_delta[k]

        for k in range(num_classes):
            biases["output"][k] += learning_rate * output_delta[k]

        # Update weights and biases for hidden layer
        for j in range(input_size):
            for k in range(hidden_layer_size):
                weights["hidden"][j][k] += learning_rate * X_i[j] * hidden_delta[k]

        for k in range(hidden_layer_size):
            biases["hidden"][k] += learning_rate * hidden_delta[k]

    total_errors.append(epoch_error)  # Store error for the epoch

return weights, biases, total_errors
```

- Purpose: Train the multilayer perceptron with backpropagation:
- Initialize Weights & Biases: Randomly initialize matrices for hidden and output layers.

Forward Pass:

- Calculate inputs/outputs for hidden and output layers.
- Use ReLU for hidden and Sigmoid for output activations.

Error Calculation:

- Compute errors using squared differences.

Backpropagation:

- Compute gradients and adjust weights/biases.

Plot Decision Boundary

```python
# Plot decision boundary for multiclass classification
def plot_decision_boundary_multilayer(ax, X, y, weights, biases, num_classes):
    ax.clear()
    ax.set_xlim(-20, 20)
    ax.set_ylim(-20, 20)
    ax.axhline(0, color='black', linewidth=1)
    ax.axvline(0, color='black', linewidth=1)
    ax.grid(True)

     # Find x_min, x_max, y_min, y_max without using min and max
    x_min, x_max = float('inf'), float('-inf')
    y_min, y_max = float('inf'), float('-inf')
    for x in X:
        if x[0] < x_min:
            x_min = x[0]
        if x[0] > x_max:
            x_max = x[0]
        if x[1] < y_min:
            y_min = x[1]
        if x[1] > y_max:
            y_max = x[1]
   # Adjust the boundaries
    x_min, x_max = x_min - 1, x_max + 1
    y_min, y_max = y_min - 1, y_max + 1
# Generate grid points for decision boundary
    xx, yy = np.meshgrid(np.arange(x_min, x_max, 0.1),
                         np.arange(y_min, y_max, 0.1))

    Z = np.array([
        predict_multilayer([xx[i, j], yy[i, j]], weights, biases, hidden_layer_size=len(weights["hidden"][0])).argmax()
        for i in range(xx.shape[0]) for j in range(xx.shape[1])
    ])
    Z = Z.reshape(xx.shape)
```

```python
Z = Z.reshape(xx.shape)

ax.contourf(xx, yy, Z, cmap=plt.cm.rainbow, alpha=0.3)

colors = ['red', 'blue', 'green', 'purple']
for i, label in enumerate(y):
    class_idx = label.index(1)
    ax.scatter(X[i][0], X[i][1], color=colors[class_idx], label=f'Class {class_idx}' if i == 0 or label not in y[:i] else "")

ax.set_xlabel("Feature 1")
ax.set_ylabel("Feature 2")
ax.set_title(f"Multiclass Classification with Decision Boundaries ({num_classes} classes)")
ax.legend()
```

Purpose: Visualize decision boundaries for the model:

- Generate grid points in the feature space.
- Use **predict_multilayer** to classify each point.
- Plot results using **matplotlib**.

Interactive Plot GUI

```
class InteractivePlot(QWidget):
    def __init__(self):
        super().__init__()
        self.num_classes = 4
        self.points = []
        self.current_class = 0

        self.setWindowTitle("Multilayer Perceptron Input")
        layout = QVBoxLayout()
        self.setLayout(layout)

        layout.addWidget(QLabel("Select number of classes:"))
        self.class_count_selector = QComboBox()
        for i in range(2, 11):
            self.class_count_selector.addItem(str(i))
        self.class_count_selector.setCurrentText(str(self.num_classes))
        self.class_count_selector.currentIndexChanged.connect(self.update_class_count)
        layout.addWidget(self.class_count_selector)

        layout.addWidget(QLabel("Select current class:"))
        self.class_selector = QComboBox()
        self.update_class_selector()
        self.class_selector.currentIndexChanged.connect(self.select_class)
        layout.addWidget(self.class_selector)

        # Add input for hidden_input_layer
        layout.addWidget(QLabel("Enter Hidden Input Layer (comma separated):"))
        self.hidden_input_layer_input = QLineEdit()
        layout.addWidget(self.hidden_input_layer_input)
```

```
    self.figure, (self.ax, self.error_ax) = plt.subplots(1, 2, figsize=(10, 5))
    self.canvas = FigureCanvas(self.figure)
    self.ax.set_xlim(-20, 20)
    self.ax.set_ylim(-20, 20)
    self.ax.axhline(0, color='black', linewidth=1)
    self.ax.axvline(0, color='black', linewidth=1)
    self.ax.grid(True)
    layout.addWidget(self.canvas)

    self.train_button = QPushButton("Train Model")
    self.train_button.clicked.connect(self.train_model)
    layout.addWidget(self.train_button)

    self.clear_button = QPushButton("Clear Points")
    self.clear_button.clicked.connect(self.clear_points)
    layout.addWidget(self.clear_button)

    self.canvas.mpl_connect("button_press_event", self.onclick)
```

```python
    def update_class_count(self):
        self.num_classes = int(self.class_count_selector.currentText())
        self.update_class_selector()

    def update_class_selector(self):
        self.class_selector.clear()
        for i in range(self.num_classes):
            self.class_selector.addItem(f"Class {i}")

    def select_class(self, index):
        self.current_class = index

    def onclick(self, event):
        if event.xdata is not None and event.ydata is not None:
            self.points.append((event.xdata, event.ydata, self.current_class))
            self.ax.plot(event.xdata, event.ydata, 'o', label=f"Class {self.current_class}", color=f"C{self.current_class}")
            self.canvas.draw()

    def train_model(self):
        X = [[p[0], p[1]] for p in self.points]
        y = [[1 if p[2] == i else 0 for i in range(self.num_classes)] for p in self.points]

        # Get the hidden input layer from the QLineEdit input
        hidden_input_layer_text = self.hidden_input_layer_input.text()

        if hidden_input_layer_text:
            try:
```

Handle Mouse Clicks with onclick()

- Capture and store clicked points in the feature space.

Train Model

```python
if hidden_input_layer_text:
    try:
        hidden_input_layer = [float(x.strip()) for x in hidden_input_layer_text.split(',')]
    except ValueError:
        print("Invalid input for hidden input layer.")
        return
else:
    hidden_input_layer = None

weights, biases, total_errors = train_multilayer_perceptron(
    X, y, self.num_classes, hidden_layer_size=5, learning_rate=0.1, epochs=100, hidden_input_layer=hidden_input_layer
)

# Plot decision boundaries
plot_decision_boundary_multilayer(self.ax, X, y, weights, biases, self.num_classes)

# Plot total error
self.error_ax.clear()
self.error_ax.plot(total_errors, label="Total Error", color="red")
self.error_ax.set_title("Total Error Over Epochs")
self.error_ax.set_xlabel("Epoch")
self.error_ax.set_ylabel("Error")
self.error_ax.legend()

self.canvas.draw()
```

- Train the model using collected data points.
- Plot:
- Decision boundary.
- Total error over epochs

Clear Points

```
    def clear_points(self):
        self.points = []
        self.ax.clear()
        self.ax.set_xlim(-20, 20)
        self.ax.set_ylim(-20, 20)
        self.ax.axhline(0, color='black', linewidth=1)
        self.ax.axvline(0, color='black', linewidth=1)
        self.ax.grid(True)

        self.error_ax.clear()
        self.error_ax.set_title("Total Error Over Epochs")
        self.error_ax.set_xlabel("Epoch")
        self.error_ax.set_ylabel("Error")
        self.canvas.draw()
```

- Clear stored data points and reset plots.

Application Runner

```
# Run the application
if __name__ == '__main__':
    app = QApplication(sys.argv)
    window = InteractivePlot()
    window.show()
    sys.exit(app.exec_())
```

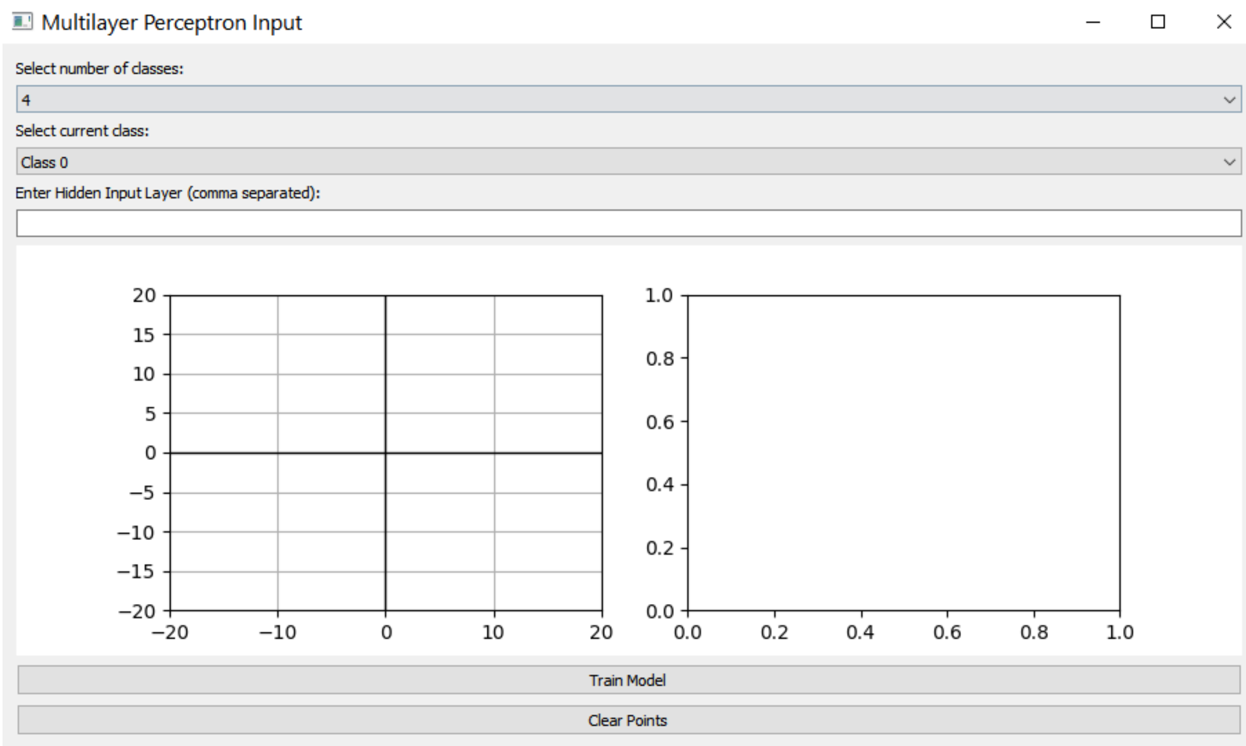
- **Purpose:** Run the PyQt5 application.

How It Works Together

1. Set Up GUI: Run the Application: create dropdowns, buttons, and plots.
2. Interactive Input: Add points by clicking and classify them using the dropdown.
3. Train Model Use the "Train Model" button to make training and plot decision boundaries.
4. Clear Points: Clear the plot with the "Clear Points" button.

Example:

Input window before training model



Output window after selecting four classes and their samples