KARADENİZ TEKNİK ÜNİVERSİTESİ
FEN BİLİMLERİ ENSTİTÜSÜ
BİLGİSAYAR MÜHENDİSLİĞİ ANABİLİM DALI

(BIL5050 - Artificial Neural Systems)

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

Prepared by: 442599 MOHAMMAD HAROON RASIKH

Advisor: (Prof.Dr. MURAT EKİNCİ)

Single and Multi – Class Single Layer Neural Network projects

Objective

This Two project presents an interactive graphical tool for exploring and understanding Single Layer and Multi - Class Single Layer perceptron learning and visualization of the decision boundary. Users can intuitively interact with the application by adding data points and training a perceptron model to classify points into multiple classes using the sigmoid activation function.

Single Neuron: Two Classes classification:

This Python script will realize a GUI application using PyQt5 to visualize and interact with a Single-Layer Perceptron for binary classification. The key components of the code and the idea behind them are explained below:

Core Functionalities

1. **Sigmoid Activation Function**

```
# Sigmoid activation function
def sigmoid(x):
    # Implementing the sigmoid function
    return 1 / (1 + (2.71828 ** -x))  # Approximate value of e
```

This function gives outputs between 0 and 1, hence useful for probabilistic binary classification.

2. **Predict Function**

```
# Predict function for binary classification
def predict(X, weights, bias):
    # Compute the dot product manually
    total_activation = 0
    for i in range(len(X)):
        total_activation += X[i] * weights[i]
    total_activation += bias
    return sigmoid(total_activation)
```

The predict(X, weights, bias) function:

- Computes the weighted sum of inputs X and the weights and bias of the model.
- It passes this sum through the sigmoid activation to produce a prediction value.

3. **Training the Perceptron**

The train_perceptron(X, y, learning_rate, epochs) function:

```python
# Training function for single-layer perceptron (binary classification)
def train_perceptron(X, y, learning_rate=0.1, epochs=1000):
    weights = np.random.uniform(-1, 1, X.shape[1])  # Random weights for each feature
    bias = random.uniform(-1, 1)  # Random bias

    for epoch in range(epochs):
        for i in range(len(X)):
            prediction = predict(X[i], weights, bias)
            error = y[i] - prediction

            # Update weights and bias using gradient descent
            weights += learning_rate * error * prediction * (1 - prediction) * X[i]
            bias += learning_rate * error * prediction * (1 - prediction)

    return weights, bias
```

- Initializes random weights and bias.
- Iteratively adjusts the weights and bias using gradient descent based on the error between predicted and actual labels (y).

4. **Plotting the Decision Boundary**

The plot_decision_boundary(ax, X, y, weights, bias) function:

```python
# Plot decision boundary for binary classification
def plot_decision_boundary(ax, X, y, weights, bias):
    ax.clear()  # Clear the previous plot
    ax.set_xlim(-10, 10)
    ax.set_ylim(-10, 10)
    ax.axhline(0, color='black', linewidth=1)  # Draw horizontal axis line
    ax.axvline(0, color='black', linewidth=1)  # Draw vertical axis line
    ax.grid(True)

    # Plot data points
    for i, label in enumerate(y):
        color = 'red' if label == 0 else 'blue'  # Class 0 is red, class 1 is blue
        ax.scatter(X[i][0], X[i][1], color=color)

    # Calculate and plot decision boundary
    x_vals = np.linspace(-10, 10, 100)
    y_vals = -(weights[0] * x_vals + bias) / weights[1]  # Equation of decision boundary: w0*x + w1*y + b = 0
    ax.plot(x_vals, y_vals, color='green', label="Decision Boundary")

    ax.set_xlabel("Feature 1")
    ax.set_ylabel("Feature 2")
    ax.set_title("Single-Layer Perceptron - Decision Boundary")
    ax.legend()
```

- Clears the current plot.
- Plots the data points with color coding based on class labels.
- Calculation Equation of decision boundary: w0*x + w1*y + b = 0

Key Libraries

- PyQt5: For GUI components and event handling.
- Matplotlib: For plotting the data points and decision boundary.
- NumPy: For mathematical operations on arrays.

GUI Implementation with PyQt5

1. **Class:** PerceptronApp(Qwidget):

```
class PerceptronApp(QWidget):
    def __init__(self):
        super().__init__()
        self.points = []  # List to store points as (x, y, class)
        self.X = []  # Input features
        self.y = []  # Labels
        self.weights = None  # To store trained weights
        self.bias = None  # To store trained bias
        self.current_class = 0  # Default class is 0

        self.setWindowTitle("Single-Layer Perceptron")
        layout = QVBoxLayout()
        self.setLayout(layout)

        # Dropdown for selecting the current class
        layout.addWidget(QLabel("Select current class:"))
        self.class_selector = QComboBox()
        self.class_selector.addItem("Class 0")
        self.class_selector.addItem("Class 1")
        self.class_selector.setCurrentIndex(0)  # Default to class 0
        self.class_selector.currentIndexChanged.connect(self.select_class)
        layout.addWidget(self.class_selector)
```

```
    # Matplotlib plot
    self.figure, self.ax = plt.subplots()
    self.canvas = FigureCanvas(self.figure)
    layout.addWidget(self.canvas)

    # Button to train the perceptron
    self.train_button = QPushButton("Train Model")
    self.train_button.clicked.connect(self.train_model)
    layout.addWidget(self.train_button)

    # Clear button to reset points
    self.clear_button = QPushButton("Clear Points")
    self.clear_button.clicked.connect(self.clear_points)
    layout.addWidget(self.clear_button)

    # Connect click event on canvas
    self.canvas.mpl_connect("button_press_event", self.onclick)

    # Initialize the plot with proper axes
    self.initialize_plot()

def initialize_plot(self):
    # Set initial axes limits and grid
    self.ax.set_xlim(-10, 10)
    self.ax.set_ylim(-10, 10)
    self.ax.axhline(0, color='black', linewidth=1)
    self.ax.axvline(0, color='black', linewidth=1)
    self.ax.grid(True)
    self.canvas.draw()

def select_class(self, index):
    # Update the current class based on dropdown selection
    self.current_class = index
```

```python
def select_class(self, index):
    # Update the current class based on dropdown selection
    self.current_class = index

def onclick(self, event):
    # Only add points if a valid click (xdata and ydata exist)
    if event.xdata is not None and event.ydata is not None:
        self.points.append((event.xdata, event.ydata, self.current_class))
        self.X.append([event.xdata, event.ydata])
        self.y.append(self.current_class)

        # Plot the point in the selected class color
        color = 'red' if self.current_class == 0 else 'blue'
        self.ax.scatter(event.xdata, event.ydata, color=color)
        self.canvas.draw()

def train_model(self):
    # Convert points to numpy arrays for training
    X = np.array(self.X)
    y = np.array(self.y)

    # Train the perceptron
    self.weights, self.bias = train_perceptron(X, y)

    # Plot decision boundary
    plot_decision_boundary(self.ax, X, y, self.weights, self.bias)
    self.canvas.draw()
```

```python
def clear_points(self):
    # Clear the list of points, X, and y
    self.points = []
    self.X = []
    self.y = []

    # Re-initialize the plot with grid and axes intact
    self.ax.clear()  # Clear all existing data on the axes
    self.initialize_plot()  # Reinitialize the axes with grid and limits

    # Redraw the canvas
    self.canvas.draw()
```

```python
# Run the application
app = QApplication(sys.argv)
window = PerceptronApp()
window.show()
sys.exit(app.exec_())
```

This is the main GUI class that inherits from QWidget.

Key components:

1. **Attributes**:

   - points, X, y: To store clicked data points and their labels.
   - weights, bias: To hold the parameters of the trained perceptron.
   - current_class: The active class (0 or 1) for the clicked points.

2. **Widgets**:

   - A **dropdown menu (QComboBox)** to select the current class.
   - A **plot canvas (Matplotlib)** for data visualization and decision boundary display.
   - Buttons for training the model and clearing points.

3. **Methods**:

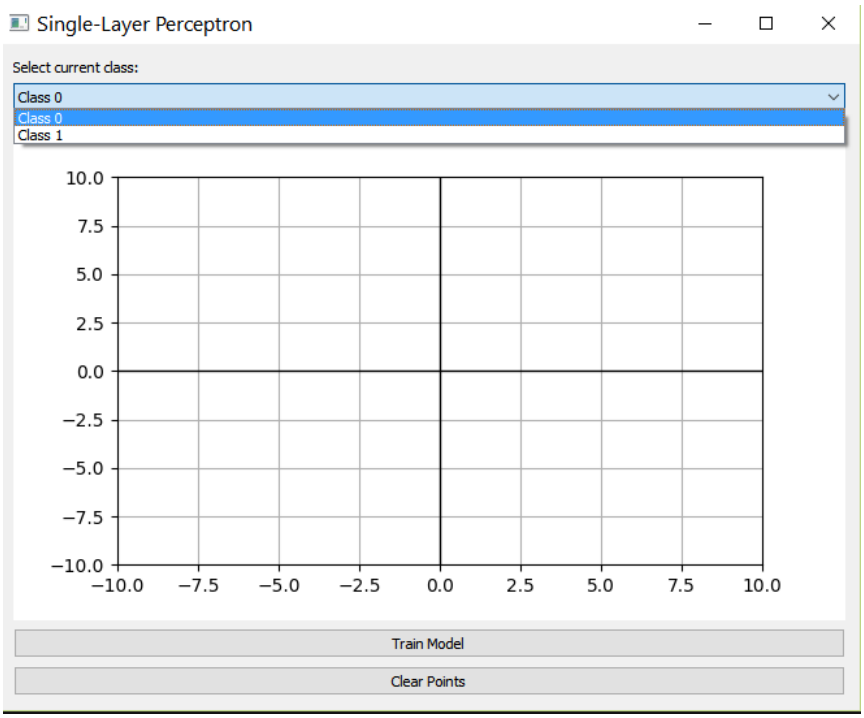
   - **initialize_plot**: Sets up an empty plot with grid lines and axes.
   - **select_class**: Updates the active class when the dropdown changes.
   - **onclick**: Adds a point to the plot when clicked.
   - **train_model**: Trains the perceptron and updates the plot with the decision boundary.
   - **clear_points**: Clears all points and resets the plot.

2. **Interactive Features**

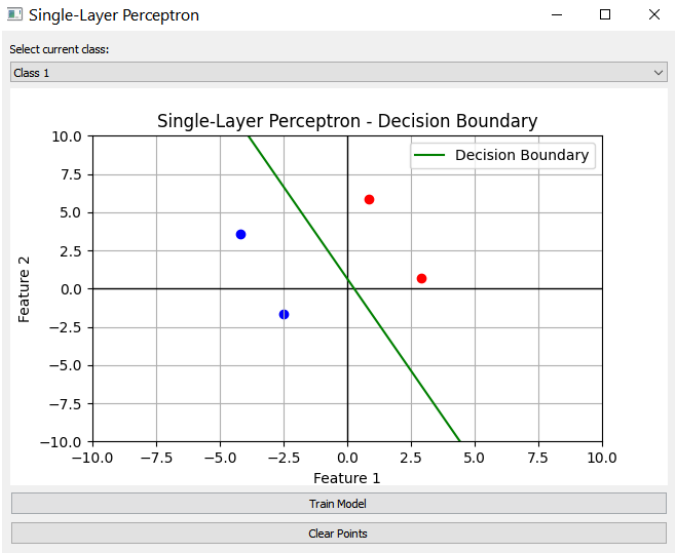
- Users click on the plot to add data points. These are saved as feature vectors (X) and labels (y).
- The dropdown toggles between class 0 (red points) and class 1 (blue points).
- When the Train Model button is clicked, the perceptron trains on the clicked points and shows the decision boundary in green.
- The button Clear Points resets the app.

Example

Input window before training single layer model



Output of Single Layer after training model



Multi Neuron : Multiple classes classification

This Python script implements a multiclass perceptron using a GUI designed in PyQt5 and visualized in Matplotlib. It trains a perceptron by gradient descent to perform a multiclass classification of points clicked by users on the plot. Here is the breakdown:

Sigmoid Activation Function

This function is applied to the output of each perceptron to transform it into a probability for classification.

```python
# Sigmoid activation function
def sigmoid(x):
    return 1 / (1 + (2.718281828459045) ** -x)
```

Predict Function for Multiclass Classification

This function calculates the output of the perceptron for each class:

- It computes the weighted sum of the inputs (X[j] * weights[i][j]) and adds the bias for each class.
- It then applies the sigmoid function to each class's total activation and stores the results in outputs.

```python
# Predict function for multiclass using the sigmoid activation function
def predict_multiclass(X, weights, biases):
    outputs = []
    for i in range(len(weights)):
        total_activation = 0
        for j in range(len(X)):
            total_activation += X[j] * weights[i][j]
        total_activation += biases[i]
        outputs.append(sigmoid(total_activation))  # Apply the sigmoid function

    return outputs
```

The train_multicategory_perceptron function does the following:

```python
# Training function with error propagation for multiclass
def train_multicategory_perceptron(X, y, num_classes, learning_rate=1, epochs=100):
    # Initialize weights and biases randomly
    weights = [[random.uniform(-1, 1) for _ in range(len(X[0]))] for _ in range(num_classes)]
    biases = [random.uniform(-1, 1) for _ in range(num_classes)]

    for epoch in range(epochs):
        for i in range(len(X)):
            # Forward pass: get outputs
            outputs = predict_multiclass(X[i], weights, biases)

            # Calculate error for each class (binary error)
            errors = [y[i][j] - outputs[j] for j in range(num_classes)]
            print('errors',errors)

            # Backward pass: update weights and biases based on error gradients
            for k in range(num_classes):
                # Calculate gradient for each weight and update weights
                for j in range(len(weights[k])):
                    weights[k][j] += learning_rate * errors[k] * outputs[k] * (1 - outputs[k]) * X[i][j]

                # Update bias with error propagation
                biases[k] += learning_rate * errors[k] * outputs[k] * (1 - outputs[k])

    return weights, biases
```

- It randomly initializes weights and biases for each class
- It trains the perceptron using gradient descent for a number of iterations specified by epochs.
- For every point:
    1. It makes a forward pass to calculate the output probabilities for each class.
    2. It calculates the error between the predicted output and actual labels using a binary error function.
    3. It updates the weights and biases using gradient descent to minimize the error.

Plotting Decision Boundary

This function visualizes the decision boundary for multi-class classification:

```python
# Plot decision boundary for multiclass classification
def plot_decision_boundary_multiclass(ax, X, y, weights, biases, num_classes):
    ax.clear()  # Clear the previous plot

    # Set axis limits and draw grid and axes lines
    ax.set_xlim(-20, 20)
    ax.set_ylim(-20, 20)
    ax.axhline(0, color='black', linewidth=1)  # Draw horizontal axis line
    ax.axvline(0, color='black', linewidth=1)  # Draw vertical axis line
    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))

    # Predict class for each grid point
    Z = np.array([
        predict_multiclass([xx[i, j], yy[i, j]], weights, biases).index(max(predict_multiclass([xx[i, j], yy[i, j]], weights, biases)))
```

```python
# Predict class for each grid point
Z = np.array([
    predict_multiclass([xx[i, j], yy[i, j]], weights, biases).index(max(predict_multiclass([xx[i, j], yy[i, j]], weights, biases)))
    for i in range(xx.shape[0]) for j in range(xx.shape[1])
])
Z = Z.reshape(xx.shape)

# Plot the decision boundary by color regions
ax.contourf(xx, yy, Z, cmap=plt.cm.rainbow, alpha=0.3)

# Plot data points
colors = ['red', 'blue', 'green', 'purple']  # Adjust colors if you have more classes
for i, label in enumerate(y):
    class_idx = label.index(1)  # Assuming one-hot encoding
    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()
```

It clears the plot and sets axis limits and grid.

It then computes the decision boundary by iterating through a grid of points, xx, yy.

It then uses the **predict_multiclass** function, obtaining an output for each class at each point on the grid, and assigns the point to the class with highest probability.

Decision boundaries are plotted by filling the regions of the plot by using contour filling (contourf).

It also plots the data points, color-coded for each class, (for example, red, blue, green, purple.)

GUI: Interactive Plot for Adding Data Points and Training

```
class InteractivePlot(QWidget):
    def __init__(self):
        super().__init__()
        self.num_classes = 4  # Default number of classes to 4
        self.points = []  # Store points as (x, y, class)
        self.current_class = 0  # Default class

        # PyQt Layout
        self.setWindowTitle("Multiclass Perceptron Input")
        layout = QVBoxLayout()
        self.setLayout(layout)

        # Dropdown for selecting the number of classes
        layout.addWidget(QLabel("Select number of classes:"))
        self.class_count_selector = QComboBox()
        for i in range(2, 11):  # Allow up to 10 classes for flexibility
            self.class_count_selector.addItem(str(i))
        self.class_count_selector.setCurrentText(str(self.num_classes))  # Default to 4 classes
        self.class_count_selector.currentIndexChanged.connect(self.update_class_count)
        layout.addWidget(self.class_count_selector)

        # Dropdown for selecting the current class
        layout.addWidget(QLabel("Select current class:"))
        self.class_selector = QComboBox()
        self.update_class_selector()  # Initialize with the default number of classes
        self.class_selector.currentIndexChanged.connect(self.select_class)
        layout.addWidget(self.class_selector)
```

```
# Matplotlib plot
self.figure, self.ax = plt.subplots()
self.canvas = FigureCanvas(self.figure)
self.ax.set_xlim(-20, 20)  # Set limits for Cartesian grid (4 quadrants)
self.ax.set_ylim(-20, 20)
self.ax.axhline(0, color='black', linewidth=1)  # Draw horizontal axis line
self.ax.axvline(0, color='black', linewidth=1)  # Draw vertical axis line
self.ax.grid(True)  # Add grid for clarity
layout.addWidget(self.canvas)

# Button to start training after adding points
self.train_button = QPushButton("Train Model")
self.train_button.clicked.connect(self.train_model)
layout.addWidget(self.train_button)

# Clear button to reset points
self.clear_button = QPushButton("Clear Points")
self.clear_button.clicked.connect(self.clear_points)
layout.addWidget(self.clear_button)

# Connect click event on canvas
self.canvas.mpl_connect("button_press_event", self.onclick)
```

```
def update_class_count(self):
    # Update the number of classes based on dropdown selection
    self.num_classes = int(self.class_count_selector.currentText())
    self.update_class_selector()  # Update class dropdown options

def update_class_selector(self):
    # Update the class selection dropdown based on the number of classes
    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
```

The **InteractivePlot** class creates the GUI to interact with the perceptron model as follows:

- **Attributes**: points - stores the coordinates and class labels of the clicked points; num_classes - determines how many classes will be classified by the model.
- **GUI Components:**
  1. **Class count drop-down** (QComboBox): User can choose the number of classes (2-10).
  2. **Class selector drop-down**: User can choose which class to assign to a new point.
  3. **Matplotlib canvas**: For plotting points and decision boundaries.
  4. **Buttons**: For starting training of the model and to clear all points.

Adding Points

```
def onclick(self, event):
    # Capture click on plot and add point with selected class
    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()
```

When the user clicks on the plot:

- The point is added to the points list with its class.
- The point is then plotted on the canvas using Matplotlib, color-coded based on the selected class.

```
def train_model(self):
    # Prepare X and y from clicked points
    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]

    # Call the training function
    weights, biases = train_multicategory_perceptron(X, y, self.num_classes, learning_rate=0.1, epochs=100)

    # Plot decision boundary on the same canvas
    plot_decision_boundary_multiclass(self.ax, X, y, weights, biases, self.num_classes)
    self.canvas.draw()
```

When the Train Model button is clicked,

- Data points will be first changed to feature and one-hot encoded label data (X and y).
- The perceptron is trained by the function of **train_multicategory_perceptron.**
- Updates the decision boundary and then draws on the canvas.

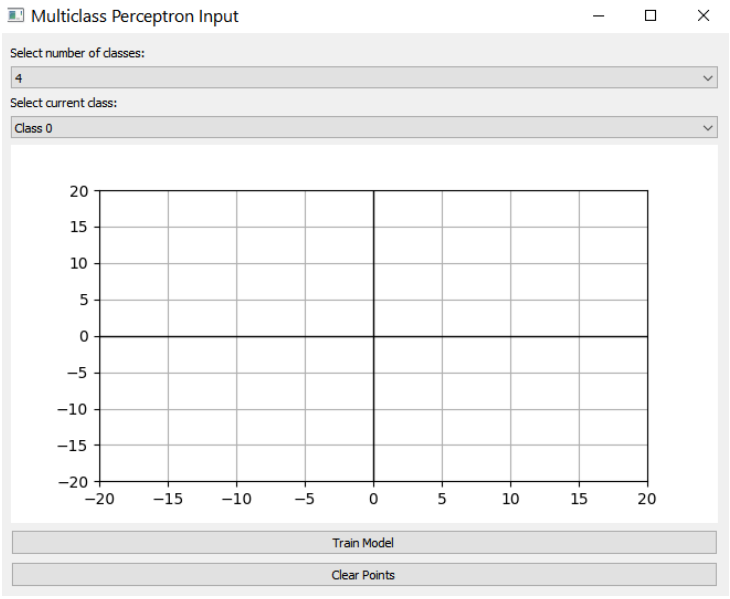
Clearing Points

```
def clear_points(self):
    # Clear points and reset the plot
    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.canvas.draw()
```

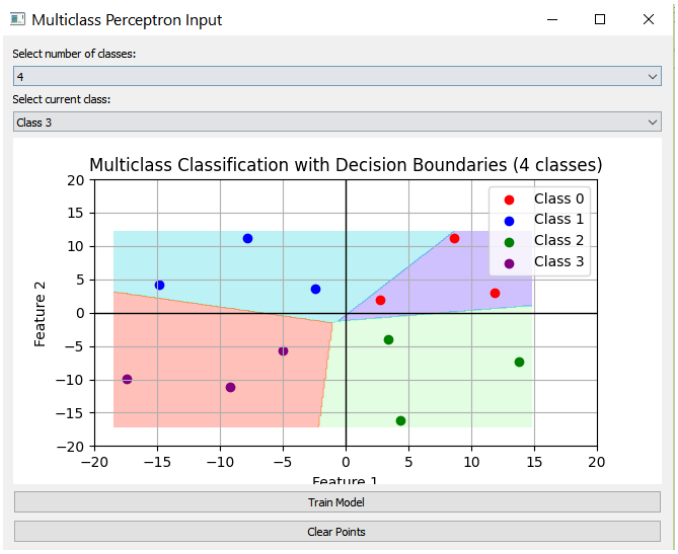
The **Clear Points** button resets the plot and removes all data points.

Example:

Input window before Training Model



Output after Training Model



Total error list

```
errors [-0.01321358756037814, -0.0742549385936915, -0.21550379040601556, 0.8433908753555807]
errors [-0.0012079240323808666, -0.024677308729590872, -0.3537381070687387, 0.35825028379441015]
errors [-0.0003958294473306788, -0.3749634356638766, -0.009136647035666832, 0.006464705344473742]
```