The below code implements a Quantum Machine Learning (QML) model with the goal of reproducing the values of the sine function. Let's break down how the code achieves this:

Overview

The goal is to create a quantum model that approximates the sine function using a parameterized quantum circuit. The model is trained using a quantum optimization technique to minimize the difference between the model's output and the true sine function values.

Detailed Explanation

1. Quantum Model Definition

```
@qml.qnode(dev)def qml_model(phi, weights):
    qml.RX(phi, wires=0)
    qml.RY(weights[0], wires=0)
    qml.CNOT(wires=[0, 1])
    qml.RY(weights[1], wires=1)
    qml.CNOT(wires=[1, 2])
    qml.RY(weights[2], wires=2)
    return qml.expval(qml.PauliZ(0))
```

Quantum Circuit: The quantum model is defined using a quantum circuit with three qubits.

RX and RY Gates: Parameterized rotation gates are applied to the qubits. The RX gate applies a rotation around the x-axis, and the RY gate applies a rotation around the y-axis. The angles for these gates are specified by phi and weights.
CNOT Gates: These entangling gates create correlations between the qubits. Entanglement helps in capturing more complex relationships between input and output.

Measurement: The output of the quantum circuit is the expectation value of the Pauli-Z operator on the first qubit (qml.expval(qml.PauliZ(0))). This output is used to approximate the sine function.

2. **Training Data**

```
x_train = np.linspace(0, 2 * np.pi, 200)
y_train = np.sin(x_train)
```

Training Data: The training data consists of input values (x_train) evenly spaced between 0 and 2π, and corresponding sine function values (y_train).

2. Loss Function

```
def loss(weights):
    predictions = np.array([qml_model(x, weights) for x in x_train])
    return np.mean((predictions - y_train) ** 2)
```

Loss Function: The loss function measures the difference between the quantum model's predictions and the actual sine values.

Prediction: For each training data point x, the quantum model generates a prediction using the current parameters (weights).

Mean Squared Error (MSE): The loss is computed as the mean squared error between the predicted values and the true sine values.

4. **Optimization**

```
opt = qml.AdamOptimizer(0.01)
weights = np.random.normal(0, 0.1, (3,), requires_grad=True)
for epoch in range(200):
    weights, _ = opt.step_and_cost(lambda w: loss(w), weights)
```

Optimizer: The Adam optimizer is used to minimize the loss function.
Weights Initialization: The parameters (weights) are initialized randomly.
Training Loop: The optimizer adjusts the parameters to minimize the loss function over multiple epochs.

5. **Evaluation**

```
x_eval = np.linspace(0, 2 * np.pi, 200)
y_eval = np.array([qml_model(x, weights) for x in x_eval])
```

Evaluation: After training, the model is evaluated on a new set of input values (x_eval) to generate predictions (y_eval).

6. **Plotting Results**

```
plt.figure(figsize=(10, 6))
plt.plot(x_train, y_train, label="Sine function")
plt.plot(x_eval, y_eval, label="QML model", linestyle='--')
plt.xlabel("x")
plt.ylabel("y")
plt.legend()
plt.title("Sine Function Approximation using Quantum Machine Learning")
plt.grid(True)
plt.show()
```

Visualization: The true sine function and the model's predictions are plotted to visualize how well the quantum model approximates the sine function.
How It Works

Parameterization: The quantum circuit is parameterized with angles that control the gates. These angles are adjusted during training to minimize the loss.

Training: The optimizer adjusts the quantum circuit parameters to reduce the error between the model's predictions and the true sine values.

Approximation: By tuning the parameters of the quantum circuit, the model learns to approximate the sine function.

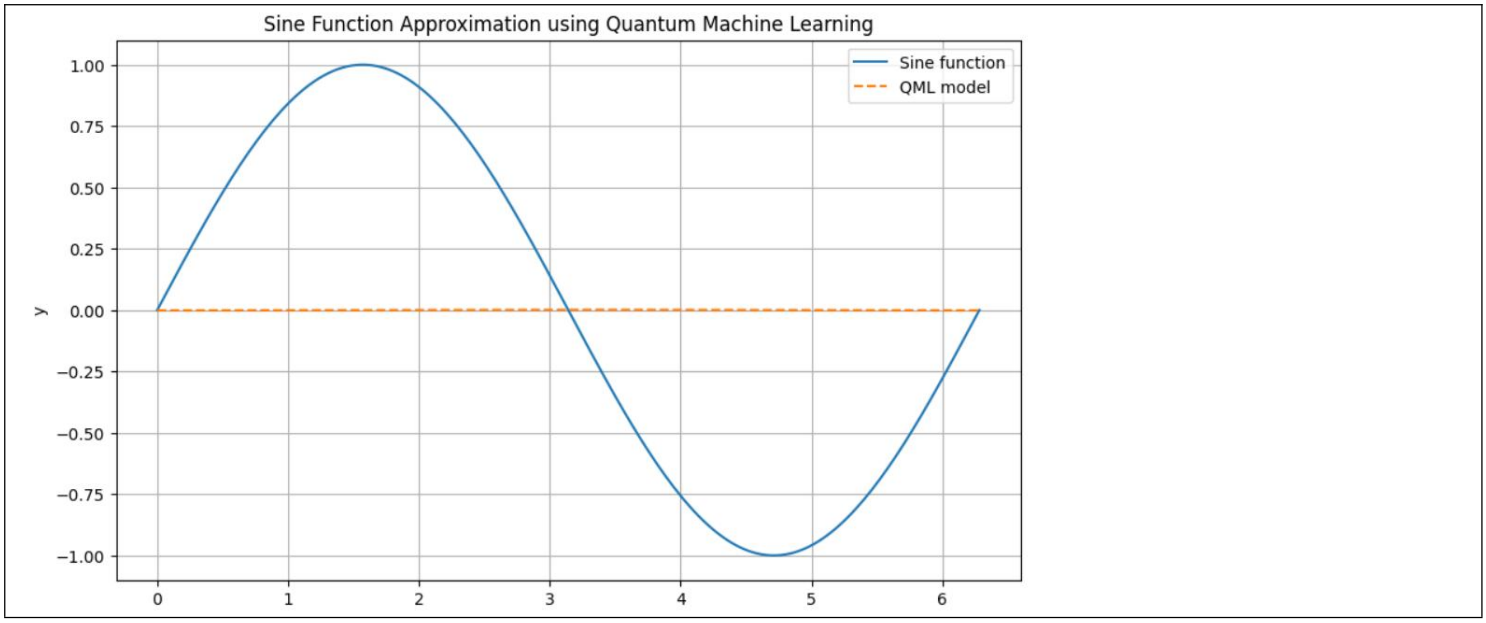
Summary

The code implements a quantum model using a quantum circuit with parameterized gates and optimizes it to fit the sine function. The model is

trained to minimize the difference between its predictions and the true sine values using the Adam optimizer. After training, the model is evaluated and plotted to show how well it reproduces the sine function.

This is a sample example, and the model's performance can be improved by experimenting with different circuit architectures, training data, and hyperparameters.

```
import pennylane as qml
from pennylane import numpy as np
import matplotlib.pyplot as plt

# Define the QML model with increased complexity
dev = qml.device("default.qubit", wires=3)

@qml.qnode(dev)
def qml_model(phi, weights):
    qml.RX(phi, wires=0)
    qml.RY(weights[0], wires=0)
    qml.CNOT(wires=[0, 1])
    qml.RY(weights[1], wires=1)
    qml.CNOT(wires=[1, 2])
    qml.RY(weights[2], wires=2)
    return qml.expval(qml.PauliZ(0))

# Define the sine function
def sine(x):
    return np.sin(x)

# Define the training data with more points
x_train = np.linspace(0, 2 * np.pi, 200)
y_train = np.sin(x_train)

# Define the loss function
def loss(weights):
    predictions = np.array([qml_model(x, weights) for x in x_train])
    return np.mean((predictions - y_train) ** 2)

# Train the QML model with adjusted hyperparameters
opt = qml.AdamOptimizer(0.01)
weights = np.random.normal(0, 0.1, (3,), requires_grad=True)

for epoch in range(200):  # Increased number of epochs
    weights, _ = opt.step_and_cost(lambda w: loss(w), weights)

# Evaluate the trained QML model
x_eval = np.linspace(0, 2 * np.pi, 200)
y_eval = np.array([qml_model(x, weights) for x in x_eval])

# Plot the results
plt.figure(figsize=(10, 6))
plt.plot(x_train, y_train, label="Sine function")
plt.plot(x_eval, y_eval, label="QML model", linestyle='--')
plt.xlabel("x")
plt.ylabel("y")
plt.legend()
plt.title("Sine Function Approximation using Quantum Machine Learning")
plt.grid(True)
plt.show()
```


Sine Function Approximation using Quantum Machine Learning
Sine function
QML model
y
1.00
0.75
0.50
0.25
0.00
−0.25
−0.50
−0.75
−1.00
0
1
2
3
4
5
6