```
!pip install qiskit
!pip install qiskit_aer
!pip install qiskit_machine_learning
```

```
import torch
import torch.nn as nn
import numpy as np
from qiskit import QuantumCircuit
from qiskit.primitives import StatevectorSampler
from qiskit.primitives import StatevectorEstimator
from qiskit_machine_learning.neural_networks import EstimatorQNN
from qiskit_machine_learning.connectors import TorchConnector
from qiskit.circuit.library import RealAmplitudes
from qiskit.quantum_info import SparsePauliOp
from qiskit.circuit import ParameterVector

# Number of qubits defines quantum feature/input dimension
num_qubits = 2

# Create a parameterized quantum circuit (ansatz)
qc = RealAmplitudes( num_qubits, reps = 1 )

# Define input parameters for features
feature_params = ParameterVector( "x", num_qubits )

# Create a new circuit that includes the feature parameters
feature_map = QuantumCircuit( num_qubits )
for i in range(num_qubits):
    feature_map.rx( feature_params[i], i ) # Example feature encoding

# Combine feature map and ansatz
full_circuit = feature_map.compose( qc )


# Define an observable for the EstimatorQNN
observable = SparsePauliOp.from_list( [ ( "Z" * num_qubits, 1 ) ] )


# Qiskit estimator backend simulating statevector
estimator = StatevectorEstimator() # Use V2 Estimator


# Create the quantum neural network wrapping the circuit and estimator
qnn = EstimatorQNN( circuit = full_circuit,
                    estimator = estimator,
                    observables = [observable],
                    input_params = feature_params,
                    weight_params = qc.parameters )

# Connect quantum neural network as a PyTorch layer
quantum_layer = TorchConnector( qnn )

# Define a hybrid PyTorch neural network
class HybridQuantumClassifier(nn.Module):
    def __init__(self):
        super( HybridQuantumClassifier, self ).__init__()
        # Classical layer: input 4 features, output matches quantum input params (num_qubits)
        self.fc1 = nn.Linear(4, num_qubits) # Changed output dimension to match num_qubits
        self.quantum = quantum_layer          # Quantum neural network layer
        self.fc2 = nn.Linear(qnn.output_shape[0], 2)           # Classical output layer (2 classes)

    def forward( self, x ):
        x = torch.relu( self.fc1(x) )      # Classical NN activation, output is num_qubits dimensions
        x = self.quantum(x)              # Quantum layer forward pass with num_qubits input
        x = self.fc2(x)                  # Classical output layer
        return torch.log_softmax( x, dim = 1 )  # Log probabilities for classification

# Instantiate model, loss function and optimizer
model = HybridQuantumClassifier()
criterion = nn.NLLLoss()              # Negative log likelihood loss for classification
optimizer = torch.optim.Adam( model.parameters(), lr = 0.01 )

# Dummy training data: 8 samples, 4 features each
X_train = torch.tensor( np.random.rand(8, 4),
                        dtype = torch.float32 )
```

```
y_train = torch.tensor( [0, 1, 0, 1, 0, 1, 0, 1],
                        dtype = torch.long )

# Training loop example
for epoch in range(10):
    optimizer.zero_grad()
    output = model( X_train )
    loss = criterion( output, y_train )
    loss.backward()
    optimizer.step()
    print( f"Epoch {epoch+1}, Loss: {loss.item():.4f}" )

# Evaluate model predictions on training data
with torch.no_grad():
    preds = model( X_train ).argmax( dim = 1 )
    print( "Predictions:", preds )
```

```
WARNING:qiskit_machine_learning.neural_networks.estimator_qnn:No gradient function provided, creating a gradient function. If your
Epoch 1, Loss: 0.6986
Epoch 2, Loss: 0.6967
Epoch 3, Loss: 0.6954
Epoch 4, Loss: 0.6943
Epoch 5, Loss: 0.6927
Epoch 6, Loss: 0.6881
Epoch 7, Loss: 0.6891
Epoch 8, Loss: 0.6887
Epoch 9, Loss: 0.6871
Epoch 10, Loss: 0.6877
Predictions: tensor([0, 0, 0, 0, 0, 0, 0, 0])
```