Task11

March 25, 2025

1 Task 11

We will be implementing a hybrid neural network that combines MLP with quantum computing.

First import required libraries

[7]:
```
import torch
import torch.nn as nn
import torch.optim as optim
import pennylane as qml
```

We will be using 4 defualt qubits of qml

[8]:
```
torch.set_default_dtype(torch.float64) # to set datatype

n_qubits = 4
dev = qml.device("default.qubit", wires=n_qubits)
```

1.0.1 Circuit

First apply RY gate

Then apply Cnot gate to all adjacent

Then again apply RY gate to all

And finally use Pauliz to to get expectation value

[ ]:
```
@qml.qnode(dev, interface="torch")
def circuit(params):
    for i in range(n_qubits):
        qml.RY(params[0, i], wires=i)

    for i in range(n_qubits - 1):
        qml.CNOT(wires=[i, i+1])

    for i in range(n_qubits):
        qml.RY(params[1, i], wires=i)

    return [qml.expval(qml.PauliZ(i)) for i in range(n_qubits)]
```

```
trial_circuit = torch.randn((2, n_qubits), dtype=torch.float64)
# sample circuit for understanding
drawer = qml.draw(circuit)
print(drawer(trial_circuit))
```

```
0:  RY(0.30)   RY(-0.39)            <Z>
1:  RY(0.44)  X      RY(1.01)        <Z>
2:  RY(0.09)   X          RY(1.39)   <Z>
3:  RY(-1.36)        X     RY(-0.04)   <Z>
```

2 MLP

This Multi-Layer Perceptron (MLP) maps normally distributed data to PQC parameters.

`fc1`: Fully connected layer (`input_dim → hidden_dim`). `fc2`: Fully connected layer (`hidden_dim → hidden_dim`). `fc3`: Fully connected layer (`hidden_dim → output_dim`). Uses ReLU activations for non-linearity. Output (`output_dim = 2 * n_qubits`) provides the parameters for the PQC.

[10]:
```
class MLP(nn.Module):
    def __init__(self, input_dim, hidden_dim, output_dim):
        super(MLP, self).__init__()
        self.fc1 = nn.Linear(input_dim, hidden_dim)
        self.fc2 = nn.Linear(hidden_dim, hidden_dim)
        self.fc3 = nn.Linear(hidden_dim, output_dim)
        self.relu = nn.ReLU()

    def forward(self, x):
        x = self.relu(self.fc1(x))
        x = self.relu(self.fc2(x))
        x = self.fc3(x)
        return x
```

2.0.1 Hyperparameters

The tolerance defines how close quantum circuit outputs must be to targets to count as accurate.

[11]:
```
input_dim = 4                   # Dimensionality of input data
hidden_dim = 16                 # Hidden layer size for the MLP
output_dim = 2 * n_qubits       # Total parameters for the PQC (2 layers x 4␣
 ↪qubits)
learning_rate = 0.01
n_epochs = 200
batch_size = 16

# Instantiate the MLP and the optimizer
mlp = MLP(input_dim, hidden_dim, output_dim).double()
optimizer = optim.Adam(mlp.parameters(), lr=learning_rate)
mse_loss = nn.MSELoss()
```

```
# Tolerance
tol = 0.1
```

Data Generation

Generate normally distributed inputs (x).

Apply tanh to bound target values between -1 and 1.

Processing Each Sample

MLP predicts PQC parameters and reshapes them into [2, n_qubits].

The quantum circuit is executed with these parameters.

The outputs are collected into a batch tensor.

Loss Calculation

Mean Squared Error (MSE) loss is computed against the target.

Backpropagation

`loss.backward()` updates both the MLP and quantum parameters.

`optimizer.step()` updates the weights.

[ ]:
```
best_acc=0
for epoch in range(n_epochs):
    optimizer.zero_grad()

    # random data normally distributed
    x = torch.randn((batch_size, input_dim), dtype=torch.float64)

    # For a target, we use tanh as range is[-1,1]
    target = torch.tanh(x)

    outputs = []
    for sample in x:
        # Predict PQC parameters and reshape to [2, n_qubits]
        params = mlp(sample).reshape(2, n_qubits)
        # Run the quantum circuit and collect expectation values
        exp_vals = torch.stack(circuit(params))
        outputs.append(exp_vals)

    outputs = torch.stack(outputs)  # shape: [batch_size, n_qubits]

    # Compute MSE loss between circuit outputs and the target (first n_qubits␣
↪dimensions)
    loss = mse_loss(outputs, target[:, :n_qubits])

    # Backpropagation
```

```
    loss.backward()
    optimizer.step()

    # Compute accuracy: percentage of output elements within the tolerance of␣
↪the target.
    accuracy = (torch.abs(outputs - target[:, :n_qubits]) < tol).float().mean()␣
↪* 100.0
    if(accuracy>best_acc):
        best_acc=accuracy
    if(epoch%10==0):
        print(f"Epoch {epoch+1}/{n_epochs} - Loss: {loss.item():.6f} - Accuracy:
↪ {accuracy.item():.2f}%")

print(f"Best accuracy = {best_acc}")
```

```
Epoch 1/200 - Loss: 0.002825 - Accuracy: 92.19%
Epoch 11/200 - Loss: 0.001792 - Accuracy: 93.75%
Epoch 21/200 - Loss: 0.003642 - Accuracy: 95.31%
Epoch 31/200 - Loss: 0.000834 - Accuracy: 100.00%
Epoch 41/200 - Loss: 0.001483 - Accuracy: 100.00%
Epoch 51/200 - Loss: 0.001748 - Accuracy: 96.88%
Epoch 61/200 - Loss: 0.002243 - Accuracy: 96.88%
Epoch 71/200 - Loss: 0.001135 - Accuracy: 98.44%
Epoch 81/200 - Loss: 0.001370 - Accuracy: 98.44%
Epoch 91/200 - Loss: 0.004487 - Accuracy: 96.88%
Epoch 101/200 - Loss: 0.001008 - Accuracy: 100.00%
Epoch 111/200 - Loss: 0.001386 - Accuracy: 98.44%
Epoch 121/200 - Loss: 0.001377 - Accuracy: 98.44%
Epoch 131/200 - Loss: 0.001163 - Accuracy: 98.44%
Epoch 141/200 - Loss: 0.001521 - Accuracy: 96.88%
Epoch 151/200 - Loss: 0.001522 - Accuracy: 98.44%
Epoch 161/200 - Loss: 0.001091 - Accuracy: 100.00%
Epoch 171/200 - Loss: 0.000922 - Accuracy: 100.00%
Epoch 181/200 - Loss: 0.001470 - Accuracy: 98.44%
Epoch 191/200 - Loss: 0.001081 - Accuracy: 100.00%
Best accuracy = 100.0
```

We have achieved 100% accuracy beacuse the dataset is simple and randomly generated. This value will be changed if we will rerun the code.