



Hybrid Architectures with PennyLane and PyTorch

A small hybrid quantum-classical classifier

Stefano Markidis
KTH Royal Institute of Technology

Setting things up

- We use NumPy, PyTorch and scikit-learn to build a simple binary classification task.
- PennyLane provides the quantum circuits and differentiable QNodes.
- PyTorch handles classical layers, training loop, and optimizers.
- We set random seeds for reproducibility and use `float64` to match PennyLane's default.

```
import numpy as np
import torch
from sklearn.metrics import accuracy_score
from sklearn.model_selection import train_test_split
from sklearn.datasets import make_classification
seed = 1234
np.random.seed(seed)
torch.manual_seed(seed)
# match PennyLane default
torch.set_default_dtype(torch.float64)
```

Synthetic dataset and precision

- We generate a synthetic binary dataset with `make_classification` (e.g. 1,000 samples, 20 features).
- Data is split into train, validation, and test sets.
- Inputs remain classical features; the quantum layer will see a reduced representation.

```
x, y = make_classification(  
    n_samples=1000,  
    n_features=20,  
    n_informative=15,  
    n_redundant=5,  
    random_state=seed,  
)  
  
x_tr, x_test, y_tr, y_test = train_test_split(x, y, train_size=0.8)  
x_val, x_test, y_val, y_test = train_test_split(x_test, y_test, train_size=0.5)
```

PennyLane device and Measurement

- We use the 'lightning.qubit' simulator as a backend.
- The number of qubits sets the size of the quantum layer (e.g., four qubits).
- We measure the projector $|0\rangle\langle 0|$ on a single qubit, which yields values in $[0, 1]$.
- This expectation value can be interpreted as a class probability in a binary classifier.

```
import pennylane as qml
n_qubits = 4
dev = qml.device("lightning.qubit", wires=n_qubits)

state_0 = np.array([[1.0], [0.0]])
M = state_0 @ state_0.conj().T    #  $|0\rangle\langle 0|$ 
```

Variational Ansatz: TwoLocal

- We construct a parameterized circuit with layers of RY rotations and CNOT entanglers.
 - Pattern: RY on each qubit -> CNOT chain -> repeat for 'reps' layers.
 - A final RY layer adds extra expressivity.
 - Number of parameters scales as $(\text{reps}+1) \times n_{\text{qubits}}$.

```
def TwoLocal(n_qubits, theta, reps=1):  
    for r in range(reps):  
        # Single-qubit rotations  
        for i in range(n_qubits):  
            qml.RY(theta[r * n_qubits + i], wires=i)  
        # Entangling CNOT layer  
        for i in range(n_qubits - 1):  
            qml.CNOT(wires=[i, i + 1])  
        # Final rotation layer  
    for i in range(n_qubits):  
        qml.RY(theta[reps * n_qubits + i], wires=i)
```

Quantum node (QNode) for the QNN

- Inputs are embedded with AngleEmbedding on all qubits.
- The TwoLocal circuit is applied with trainable parameters θ .
- We return `qml.expval` of the Hermitian observable $|\theta\rangle\langle\theta|$ on one qubit.
- We use `interface='torch'` and `diff_method='adjoint'` for efficient gradients.

```
@qml.qnode(dev, interface="torch", diff_method="adjoint")
def qnn(inputs, theta):
    qml.AngleEmbedding(inputs, wires=range(n_qubits))
    TwoLocal(n_qubits, theta, reps=2)
    return qml.expval(qml.Hermitian(M, wires=[0]))

weight_shapes = {"theta": (12,)} # 4 qubits x (2+1) layers
```

Hybrid model: Classical + Quantum in PyTorch

- Classical front-end: Linear layer $20 \rightarrow 4$ with a nonlinear activation (e.g., sigmoid).
- Quantum layer: the QNode wrapped as `qml.qnn.TorchLayer`, acting on four features / four qubits.
- The `TorchLayer` behaves like a `torch.nn.Module` inside the network.
- The model outputs a single scalar per sample, interpreted as $P(\text{class}=1)$.

```
qlayer = qml.qnn.TorchLayer(qnn,
weight_shapes)
import torch.nn as nn
class HybridQNN(nn.Module):
    def __init__(self):
        super().__init__()
        self.fc = nn.Linear(20, 4)
        self.qlayer = qlayer

    def forward(self, x):
        x = torch.sigmoid(self.fc(x))
        x = self.qlayer(x)
        return x
```

Classical input layer

Quantum layer

Training loop in PyTorch

- Define a Binary Cross-Entropy loss (BCELoss or BCEWithLogitsLoss).
- Use an optimizer such as Adam with a small learning rate (e.g., 0.005).
- In each epoch:
 - forward pass → compute loss → backpropagate → optimizer step.

```
model = HybridQNN()
criterion = nn.BCELoss()
optimizer = torch.optim.Adam(model.parameters(),
                               lr=0.005)
x_tr_t = torch.tensor(x_tr)
y_tr_t = ...
x_val_t = torch.tensor(x_val)
y_val_t = ...

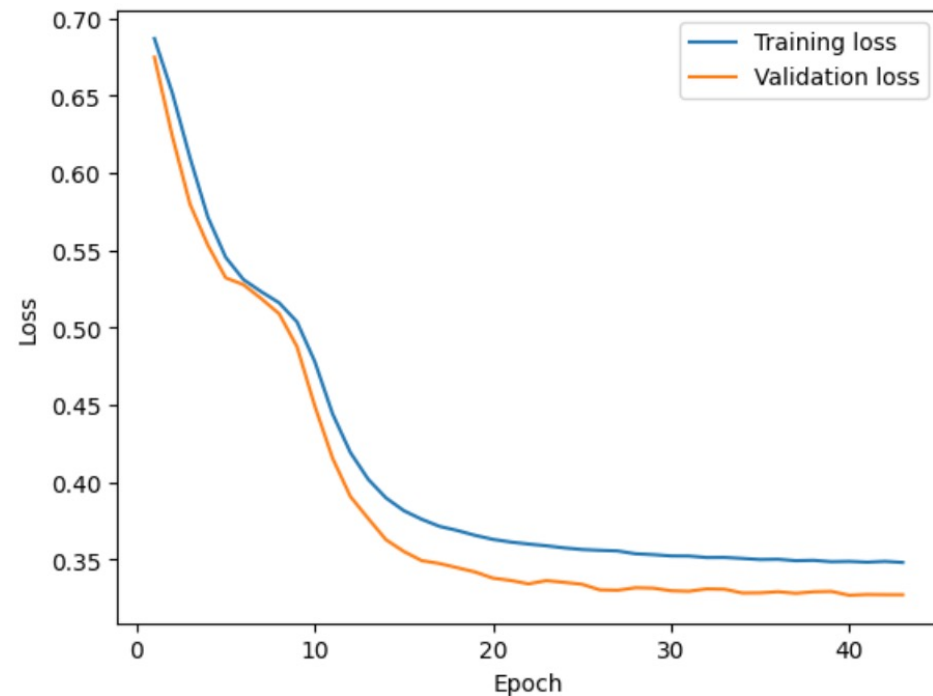
for epoch in range(50):
    model.train()
    optimizer.zero_grad()
    y_pred = model(x_tr_t)
    loss = criterion(y_pred, y_tr_t)
    loss.backward()
    optimizer.step()
    model.eval()
    with torch.no_grad():
        val_pred = model(x_val_t)
        val_loss = criterion(val_pred, y_val_t)
```

Evaluating Classification Accuracy

- After training, evaluate on the train, validation, and test sets.
- Apply a threshold of 0.5 to convert probabilities into class labels.
- Compute accuracy scores with scikit-learn (`accuracy_score`).
- Compare train/val/test accuracies to assess the hybrid model's generalization.

Binary classifier results

Train accuracy: 0.9475
Validation accuracy: 1.0
Test accuracy: 0.88



Summary

- Shows how to embed a quantum layer (QNode) inside a familiar PyTorch model.
- Uses a small, realistic binary classification task for experimentation.
- Pattern generalizes to other datasets and deeper classical front-ends.
- Provides a template for more advanced hybrid models built with PennyLane + PyTorch.