# QNN with PennyLane and PyTorch

Stefano Markidis
KTH Royal Institute of Technology

# Binary Classification - Breast Cancer Wisconsin Dataset

- We will train a QNN model to implement a **binary classifier**.
  - Dataset provided by the `scikit-learn` package: the ”Breast cancer Wisconsin dataset”.
- This dataset contains 569 samples, each with 30 numerical variables.
  - These variables describe features that can be used to characterize whether a breast mass is benign or malignant.
    - Each sample's label is either 0 (malignant) or 1 (benign).
- Dataset online at https://scikit-learn.org/stable/datasets/toy_dataset.html#

```
from sklearn.datasets import load_breast_cancer

x,y = load_breast_cancer(return_X_y = True)
```

Ref: Dua, D. & Graff, C. UCI Machine Learning Repository. (University of California, Irvine, School of Information,2017), http://archive.ics.uci.edu/ml

# Training, Validation, and Test Dataset

- We can split our dataset into a training, validation, and test dataset as follows:

```
from sklearn.model_selection import train_test_split

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)
```

# Dataset Normalization

- All the variables in the dataset are non-zero, but they are not normalized. To use them with any of our feature maps, we **normalize the training data between 0 and 1** using `MaxAbsScaler`.

```
from sklearn.preprocessing import MaxAbsScaler

scaler = MaxAbsScaler()
x_tr = scaler.fit_transform(x_tr)
```

- We normalize the test and validation datasets in the same proportions as the training dataset:

```
x_test = scaler.transform(x_test)
x_val = scaler.transform(x_val)
# Restrict all the values to be between 0 and 1.
x_test = np.clip(x_test, 0, 1)
x_val = np.clip(x_val, 0, 1)
```

# Encoding Data into Quantum States

- Our dataset has 30 variables.
- Encoding options:
  - Use the amplitude encoding feature map on five qubits, which can accommodate up to $2^5 = 32$ variables
    - It is straightforward if you use the `qml.AmplitudeEmbedding` template that we studied for QSVM.
  - Use any of the other feature maps that we have used,
    - 30 Qubits > possible to simulate, but too long
    - In conjunction with a dimensionality reduction technique.
  - We will go for the latter choice.

# Data Reduction Dimensionality

- We use principal component analysis to reduce the number of variables in our dataset to 4:

```
from sklearn.decomposition import PCA
pca = PCA(n_components = 4)

xs_tr = pca.fit_transform(x_tr)

xs_test = pca.transform(x_test)
xs_val = pca.transform(x_val)
```

# QNN Encoding & Variational Form

- The ZZ feature map and the two-local variational form are not built into PennyLane
  - We provide the implementation of these variational circuits.

Encoding

```python
from itertools import combinations
def ZZFeatureMap(nqubits, data):
  # Number of variables that we will load:
  # could be smaller than the number of qubits.
  nload = min(len(data), nqubits)
  for i in range(nload):
    qml.Hadamard(i)
    qml.RZ(2.0 * data[i], wires = i)
  for pair in list(combinations(range(nload), 2)):
    q0 = pair[0]
    q1 = pair[1]
    qml.CZ(wires = [q0, q1])
    qml.RZ(2.0 * (np.pi - data[q0]) * (np.pi-data[q1]),
           wires = q1)
    qml.CZ(wires = [q0, q1])

def TwoLocal(nqubits, theta, reps = 1):
  for r in range(reps):
    for i in range(nqubits):
       qml.RY(theta[r * nqubits + i], wires = i)
    for i in range(nqubits - 1):
       qml.CNOT(wires = [i, i + 1])
  for i in range(nqubits):
    qml.RY(theta[reps * nqubits + i], wires = i)
```

Variational Circuit

# Observables and Measurements

- Observables are represented by Hermitian operators in quantum mechanics.
  - PennyLane allows us to work directly with these Hermitian representations.
- For instance, you may use return `qml.probs(wires = [0])` at the end of the definition of a circuit to get the probabilities of every possible measurement outcome on a computational basis.
- Additional possibilities:
  - Given any Hermitian matrix A (encoded as a numpy array A), we may retrieve **the expectation value of A on an array of wires w at the end of a circuit** with return `qml.expval(A, wires = w).`
  - The dimensions of A must be compatible with the length of w. This is useful in our case,
    - To get the expectation value on the first qubit, we will have to compute the expectation value of the Hermitian

# Expectation Value on the First Qubit

- To get the expectation value on the first qubit, we will just have to compute the expectation value of the Hermitian.

$$M = \begin{pmatrix} 1 & 0 \\ 0 & 0 \end{pmatrix}$$

- The matrix M can be constructed as follows:

```
state_0 = [[1], [0]]
M = state_0 * np.conj(state_0).T
```

- **This will give us, as output, a value between 0 and 1**, which is perfect for constructing a classifier:
  - We will assign class 1 to every data instance with a value of 0.5 or higher and class 0 to all the rest.

# QNN Combining Layers and Measurement

- A PennyLane quantum node has two arguments: inputs and theta.
  - Its first argument must accept an **array with all the inputs** to the network, and the **name of this argument must be input**.
  - After this argument, we provide the optimizable parameters in the variational form.

```
n_qubits = 4
reps = 2 # depth for TwoLocal
dev = qml.device("default.qubit", wires=n_qubits)

@qml.qnode(dev, interface="torch", diff_method="best")
def qnode(x, theta):

    ZZFeatureMap(n_qubits, x)

    TwoLocal(n_qubits, theta, reps=reps)

    return qml.expval(qml.PauliZ(0))
```

- **Note:** we have added the argument interface = "torch" to the quantum node initializer.
- Had we used the `@qml.qnode` decorator, we would've had to include the argument in its call.

# Batched QNode Layer

- A `nn.Module` wrapper around a PennyLane `QNode`.
- Trainable quantum parameters:
  - Uses a `nn.ParameterDict` driven by `weight_shapes` to register learnable angles (e.g., theta).
- Forward pass (batched):
  - Casts inputs to `float64`, ensures a batch dimension, then calls the QNode per sample.
  - Each QNode returns $\langle Z \rangle \in [-1, 1]$, which is mapped to a probability via (ev + 1) / 2

```
class BatchedQNodeLayer(nn.Module):
    def __init__(self, qnode, weight_shapes):
        super().__init__()
        …

    def forward(self, x):
       x = x.to(dtype=torch.float64)
       for i in range(x.shape[0]):
…
```

# Integrating PyTorch with PennyLane

- Thanks to PennyLane's interoperability, **we train our QNN with PyTorch**

```
import torch.nn as nn
weight_shapes = {"theta": ((reps + 1) * n_qubits,)} # flat vector for TwoLocal

model = nn.Sequential(BatchedQNodeLayer(qnode, weight_shapes)).double()
print(model)
```

- Tell the quantum layer how many trainable params it needs:
  - `weight_shapes = {"theta": ((reps + 1) * n_qubits,)}` defines a flat vector of parameters for the TwoLocal ansatz: reps RY layers + one final RY layer, each with n_qubits angles → total (reps + 1) * n_qubits learnable angles stored in theta.
- Build the model as a <u>single quantum layer</u>:
  - `model = nn.Sequential(BatchedQNodeLayer(qnode, weight_shapes)).double()` creates a model consisting of one custom layer that runs the QNode per sample and outputs a probability;

# Training

- Run for 15 epochs with Adam (`lr=5e-3`) and binary cross-entropy
- Training step (per epoch):
  1. `model.train()`
  2. For each batch:
     1. `opt.zero_grad()`
     2. forward (`preds = model(xb)`)
     3. loss (`criterion(preds, yb)`)
     4. backprop (`loss.backward()`)
     5. update (`opt.step()`)

```
epochs = 15
opt = torch.optim.Adam(model.parameters(), lr=5e-3)
criterion = nn.BCELoss()

history = {"loss": [], "val_loss": []}

for epoch in range(1, epochs+1):
    model.train()
    train_loss = 0.0
    for xb, yb in train_loader:
        opt.zero_grad()
        preds = model(xb)
        loss = criterion(preds, yb)
        loss.backward()
        opt.step()
        train_loss += loss.item() * xb.size(0)

    train_loss /= len(train_loader.dataset)
    model.eval()
    val_loss = 0.0
```

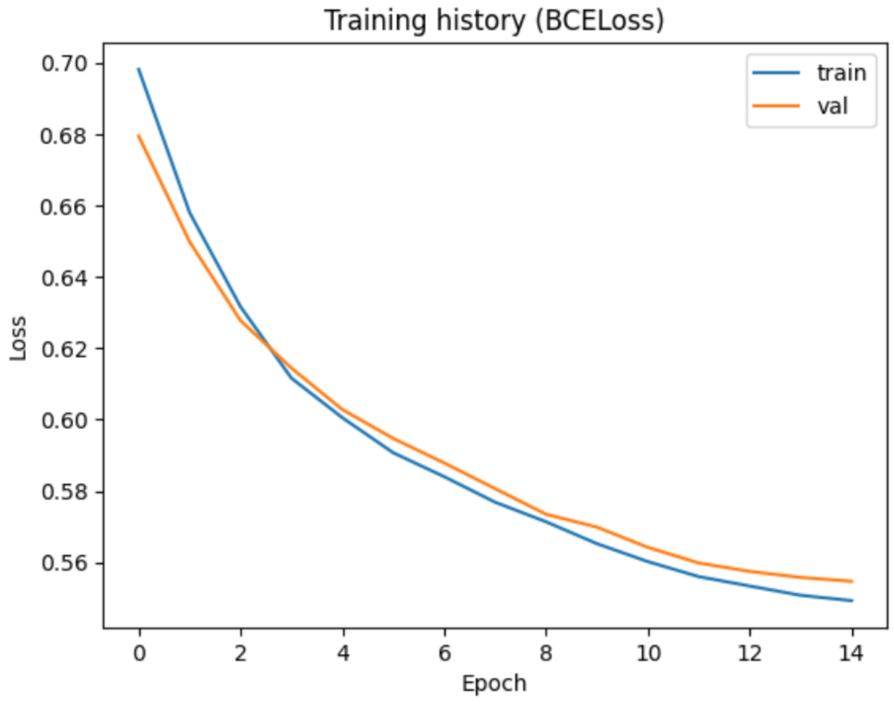
# Training Execution

```
Epoch 01/15 - loss: 0.6981 - val_loss: 0.6794
Epoch 02/15 - loss: 0.6580 - val_loss: 0.6498
Epoch 03/15 - loss: 0.6316 - val_loss: 0.6277
Epoch 04/15 - loss: 0.6116 - val_loss: 0.6144
Epoch 05/15 - loss: 0.6005 - val_loss: 0.6028
Epoch 06/15 - loss: 0.5907 - val_loss: 0.5947
Epoch 07/15 - loss: 0.5840 - val_loss: 0.5878
Epoch 08/15 - loss: 0.5768 - val_loss: 0.5806
Epoch 09/15 - loss: 0.5713 - val_loss: 0.5735
Epoch 10/15 - loss: 0.5652 - val_loss: 0.5698
Epoch 11/15 - loss: 0.5602 - val_loss: 0.5642
Epoch 12/15 - loss: 0.5560 - val_loss: 0.5598
Epoch 13/15 - loss: 0.5533 - val_loss: 0.5574
Epoch 14/15 - loss: 0.5507 - val_loss: 0.5558
Epoch 15/15 - loss: 0.5492 - val_loss: 0.5547
```

**The model is learning** because the training and validation losses are dropping.

# Plotting the QNN Losses

```python
import matplotlib.pyplot as plt
plt.figure()
plt.plot(history['loss'], label='train')
plt.plot(history['val_loss'], label='val')
plt.xlabel('Epoch')
plt.ylabel('Loss')
plt.title('Training history (BCELoss)')
plt.legend()
plt.show()
```



# Evaluating the Performance

```
Test Accuracy: 0.7193
Confusion Matrix [ [TN FP], [FN TP] ]:
[[ 6 15]
 [ 1 35]]
```