Quantum Machine Learning for Conspicuity Detection in Production

Fatma Nur KILINÇ and Özgür Erdem ERTUĞRUL

(Dated: August 9, 2024)

This project, conducted as part of the Womanium Quantum&AI 2024 program which involves several steps aimed at familiarizing participants with the concept of quantum machine learning and its application in Conspicuity Detection in Production.

STEP 1: FAMILIARIZE YOURSELF WITH PENNYLANE

PennyLane, developed by Xanadu, is an open-source software framework designed for quantum machine learning, quantum chemistry, and quantum computing[1]. To familiarize ourselves with the framework, we will go thorough the Pennylane codebooks such as "Introduction to Quantum Computing", "Single-Qubit Gates" and "Circuits with Many Qubits".

Introduction to Quantum Computing

1. For a given quantum state $|\Psi\rangle$:

$$|\Psi\rangle = \alpha|0\rangle + \beta|1\rangle \tag{1}$$

to be a valid quantum state, it must satisfy the normalization condition:

$$\alpha^2 + \beta^2 = 1 \tag{2}$$

where α and β are complex coefficients (amplitudes). This condition ensures that the probabilities of all possible outcomes add up to 1, reflecting the certainty of finding the system in one of its possible states when measured.

2. Inner product of two orthonormal states $|\Psi\rangle$ and $|\Phi\rangle$ equals to 1.

$$\langle\Phi|\Psi\rangle = \delta_{\Phi,\Psi} \tag{3}$$

where $\delta_{\Phi,\Psi}$ is the Kronecker delta.

3. Probability of measurement of an outcome is mod square of the amplitude of the outcome. If the amplitude is α, then the probability of the outcome can be calculated in the following way,

$$|\alpha|^2 = \alpha * \alpha^* \tag{4}$$

where α^* is the conjugate of α.

4. Unitary matrices preserve the normalization of states. Therefore, any quantum state transformed by a unitary operation remains a valid quantum state.

$$|\Psi^{'}\rangle = U|\Psi\rangle \tag{5}$$

where U is a unitary operation. Their defining property is that $UU^{\dagger} = I$.

5. In quantum mechanics, the order in which operators are applied is crucial.

6. In PennyLane, to execute quantum circuits, you need two essential components:

   - **Device:** This represents the backend where your quantum circuit will run. It can be a simulator or a real quantum hardware device. You create a device using:

     ```
     dev = qml.device('device.name',
         wires=num_qubits)
     ```

     'default.qubit' is a standard quantum simulator used in PennyLane.

   - **QNode:** This encapsulates your quantum circuit and binds it to the specified device. There are two ways to create a QNode:
     - Using **'qml.QNode'** function:

       ```
       my_qnode =
           qml.QNode(quantum_function,
           dev)
       ```

     - Using the **'@qml.qnode(dev)'** decorator:

       ```
       @qml.qnode(dev)
       def quantum_function(params):
           #circuit definition
           return result
       ```

       Here, *quantum_function* automatically becomes a *QNode* associated with the device *dev*.

7. In quantum computing, circuit depth represents the number of sequential layers or time steps required to execute a quantum circuit (done as in-parallel as possible). Minimizing depth is crucial for efficient quantum algorithms.

8. In PennyLane, the *QubitUnitary* operation allows you to implement a unitary operation specified by a matrix U on a quantum circuit.

```
qml.QubitUnitary(U, wires=wire)
```

9. Unitary matrices can be parameterized. In PennyLane, the *Rot* gate allows you to implement a parameterized unitary operation using three real numbers corresponding to the angles in the unitary matrix representation.

```
qml.Rot(phi, theta, omega, wires=wire)
```

Single-Qubit Gates

1. **Pauli X Gate:** Represented by the unitary matrix X.

$$X|0\rangle = |1\rangle$$
$$X|1\rangle = |0\rangle$$

2. **Hadamard Gate:** Represented by the unitary matrix H. It can create a uniform superposition of $|0\rangle$ and $|1\rangle$.

$$H|0\rangle = \frac{1}{\sqrt{2}}(|0\rangle + |1\rangle) = |+\rangle$$
$$H|1\rangle = \frac{1}{\sqrt{2}}(|0\rangle - |1\rangle) = |-\rangle$$

3. **Pauli Z Gate:** Represented by the unitary matrix Z.

$$\mathbb{Z}|0\rangle = |0\rangle$$
$$\mathbb{Z}|1\rangle = -|1\rangle$$
$$Z|+\rangle = \frac{1}{\sqrt{2}}(|0\rangle - |1\rangle) = |-\rangle$$
$$Z|-\rangle = \frac{1}{\sqrt{2}}(|0\rangle + |1\rangle) = |+\rangle$$

It can be mathematically expressed as follows:

$$Z = HXH \tag{6}$$

4. **RZ Gate:** Parametric gate represented by the unitary matrix $RZ(\omega)$.

$$|\Psi\rangle = \alpha|0\rangle + \beta|1\rangle$$
$$RZ(\omega)|\Psi\rangle = \alpha|0\rangle + \beta e^{i\omega}|1\rangle$$

5. **S and T Gates:** The quarter turn $RZ(\frac{\pi}{2})$ and eighth turn $RZ(\frac{\pi}{4})$ gates also have their own names: the phase gate, S and the T gate, respectively.

6. **RX and RY Gates:**

$$RX(\theta) = \begin{pmatrix} cos(\frac{\theta}{2}) & -isin(\frac{\theta}{2}) \\ -isin(\frac{\theta}{2}) & cos(\frac{\theta}{2}) \end{pmatrix}$$

$$RY(\theta) = \begin{pmatrix} cos(\frac{\theta}{2}) & -sin(\frac{\theta}{2}) \\ sin(\frac{\theta}{2}) & cos(\frac{\theta}{2}) \end{pmatrix}$$

7. The most general expression of a single-qubit unitary matrix looks like this:

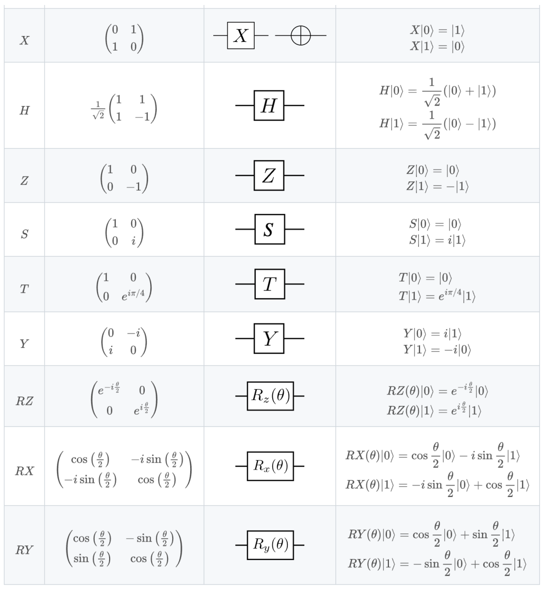
$$U(\phi,\theta,\omega) = \begin{pmatrix} e^{-i(\phi+\omega)/2}cos(\frac{\theta}{2}) & -e^{i(\phi-\omega)/2}sin(\frac{\theta}{2}) \\ e^{-i(\phi-\omega)/2}sin(\frac{\theta}{2}) & e^{i(\phi+\omega)/2}cos(\frac{\theta}{2}) \end{pmatrix} \tag{7}$$

8. **A universal gate set** in quantum computing refers to a minimal set of quantum gates that can be used to approximate any arbitrary unitary operation up to arbitrary precision. Quantum circuit synthesis uses universal gate sets to efficiently achieve diverse quantum computations, minimizing gate types while maintaining versatility in hardware design.

9. Any single-qubit unitary operation can be implemented using any two rotation of RX, RY, RZ.

$$U(\phi,\theta,\omega) = RZ(\omega + \frac{\pi}{2})RX(-\theta)RZ(\phi - \frac{\pi}{2}) \tag{8}$$
$$U(\phi,\theta,\omega) = RZ(\omega)RY(\theta)RZ(\phi) \tag{9}$$

10. Reference table for the single-qubit gates learned so far is given below.



X	$\begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}$	$\boxed{X}$ $\oplus$	$X\|0\rangle = \|1\rangle$ $X\|1\rangle = \|0\rangle$
H	$\frac{1}{\sqrt{2}}\begin{pmatrix} 1 & 1 \\ 1 & -1 \end{pmatrix}$	$\boxed{H}$	$H\|0\rangle = \frac{1}{\sqrt{2}}(\|0\rangle + \|1\rangle)$ $H\|1\rangle = \frac{1}{\sqrt{2}}(\|0\rangle - \|1\rangle)$
Z	$\begin{pmatrix} 1 & 0 \\ 0 & -1 \end{pmatrix}$	$\boxed{Z}$	$Z\|0\rangle = \|0\rangle$ $Z\|1\rangle = -\|1\rangle$
S	$\begin{pmatrix} 1 & 0 \\ 0 & i \end{pmatrix}$	$\boxed{S}$	$S\|0\rangle = \|0\rangle$ $S\|1\rangle = i\|1\rangle$
T	$\begin{pmatrix} 1 & 0 \\ 0 & e^{i\pi/4} \end{pmatrix}$	$\boxed{T}$	$T\|0\rangle = \|0\rangle$ $T\|1\rangle = e^{i\pi/4}\|1\rangle$
Y	$\begin{pmatrix} 0 & -i \\ i & 0 \end{pmatrix}$	$\boxed{Y}$	$Y\|0\rangle = i\|1\rangle$ $Y\|1\rangle = -i\|0\rangle$
RZ	$\begin{pmatrix} e^{-i\frac{\theta}{2}} & 0 \\ 0 & e^{i\frac{\theta}{2}} \end{pmatrix}$	$\boxed{R_z(\theta)}$	$RZ(\theta)\|0\rangle = e^{-i\frac{\theta}{2}}\|0\rangle$ $RZ(\theta)\|1\rangle = e^{i\frac{\theta}{2}}\|1\rangle$
RX	$\begin{pmatrix} \cos\left(\frac{\theta}{2}\right) & -i\sin\left(\frac{\theta}{2}\right) \\ -i\sin\left(\frac{\theta}{2}\right) & \cos\left(\frac{\theta}{2}\right) \end{pmatrix}$	$\boxed{R_x(\theta)}$	$RX(\theta)\|0\rangle = \cos\frac{\theta}{2}\|0\rangle - i\sin\frac{\theta}{2}\|1\rangle$ $RX(\theta)\|1\rangle = -i\sin\frac{\theta}{2}\|0\rangle + \cos\frac{\theta}{2}\|1\rangle$
RY	$\begin{pmatrix} \cos\left(\frac{\theta}{2}\right) & -\sin\left(\frac{\theta}{2}\right) \\ \sin\left(\frac{\theta}{2}\right) & \cos\left(\frac{\theta}{2}\right) \end{pmatrix}$	$\boxed{R_y(\theta)}$	$RY(\theta)\|0\rangle = \cos\frac{\theta}{2}\|0\rangle + \sin\frac{\theta}{2}\|1\rangle$ $RY(\theta)\|1\rangle = -\sin\frac{\theta}{2}\|0\rangle + \cos\frac{\theta}{2}\|1\rangle$

FIG. 1

Single-Qubit Unitaries Table[2].

11. PennyLane offers automated tools for state preparation through its library of templates. One such template is *MottonenStatePreparation*, which can automatically prepare any normalized qubit state vector, up to a global phase.

12. **Projective Measurement:** Given a quantum state $|\Psi\rangle$, the probability of observing the state in an eigenstate $|\phi\rangle$ when measuring with respect to a basis that includes $|\phi\rangle$ is given by:

$$Pr(\phi) = |\langle\phi|\Psi\rangle|^2 \tag{10}$$

13. To measure in a different basis using only computational basis measurements, perform a basis rotation before the measurement. This "tricks" the quantum computer by rotating the states so that the desired basis states map back to the computational basis. Specifically, apply the adjoint (inverse) of the operation that defines the desired basis.

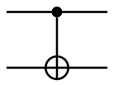
```
dev = qml.device("default.qubit", wires=1)
@qml.qnode(dev)
def measure_in_y_basis():
    # PREPARE THE STATE
    prepare_psi()
    y_basis_rotation()
    # PERFORM THE ROTATION BACK TO
        COMPUTATIONAL BASIS
    # (ADJOINT OF BASIS ROTATION)
    qml.adjoint(qml.Hadamard(wires=0))
    qml.adjoint(qml.S(wires=0))
    # RETURN THE MEASUREMENT OUTCOME
        PROBABILITIES
    return qml.probs(wires=0)
```

14. In quantum computing, physical quantities such as energy are associated with observables, which are Hermitian matrices with real eigenvalues representing possible measurement results. **Expectation value** of an observable is the weighted average of outcomes over many experiments. Mathematically, the expectation value of some observable B for a state $|\Psi\rangle$ is given by,

$$\langle B\rangle = \langle\Psi|B|\Psi\rangle \tag{11}$$

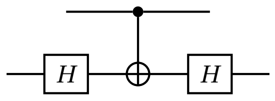
Circuits with Many Qubits

1. In PennyLane, qubits are indexed numerically from left to right. Some other libraries use the opposite numbering systems. While circuits are read from left to right, the matrices are applied to the states in reverse order, from right to left.
2. When combining multiple qubits, we use the tensor product to combine their Hilbert spaces.
3. A state is entangled if it cannot be described as a tensor product of individual qubit states; if it can, it is separable. Quantum states with more than two qubits can exhibit different levels of entanglement.
4. **C-NOT Gate:** One of the entangling gate is the controlled-NOT (CNOT) gate, a two-qubit gate that applies a Pauli-X (NOT) operation on one qubit depending on the state of another.

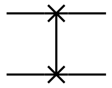


The qubit denoted with the solid dot is the control qubit.

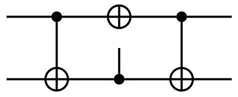
5. For multi-qubit operations, we only need the CNOT gate in addition to single-qubit gates to achieve universal computation. Any multi-qubit operation can be implemented as a controlled operation using some or all of the other qubits as control.
6. Controlled-Z can be implemented using controlled-X and H gates.



7. **SWAP Gate:** It swaps the states of two qubits.



It can be implemented using CNOT gates.



8. **Toffoli Gate:** Controlled-controlled-NOT operation, flipping the third qubit only if the controlled qubits are both $|1\rangle$. (Reversible AND operation)

STEP 2: FAMILIARIZE YOURSELF WITH QML

In order to familiarize ourselves with QML, we will go through Variational Classifier tutorial.

Variational Classifier[3]

Variational quantum classifiers, quantum circuits, are trained on labeled data to classify new samples. Examples used in this tutorial are based on two pioneering papers: Farhi and Neven[4] and Schuld et al.[5].

1. *Fitting the parity function*

Will show that a variational circuit can be trained to emulate the parity function.

$$f : x \in \{0,1\}^{\otimes n} \to y = \begin{cases} 1 & \text{if there is an odd number of 1's in } x \\ 0 & \text{else} \end{cases}$$

The elementary structure of variational classifiers is called *layers*, or *blocks*. Layers are repeated to form the complete variational circuit.

In this problem, four qubits, or wires are used. The layer is consists of two parts:

- **Arbitrary Rotations** are applied on every wires. Rotation requires three angles for each qubit. These layer parameters are called *weights*.
- **CNOT Gates:** are applied to entangle each qubit with its neighbor and looping back to entangle the last qubit with the first.

```
def layer(layer_weights):
    for wire in range(4):
        qml.Rot(*layer_weights[wire],
            wires=wire)

    for wires in ([0, 1], [1, 2], [2, 3], [3,
        0]):
        qml.CNOT(wires)
```

In this example, we require a method to encode data inputs x into the quantum circuit. The *BasisState* function in PennyLane is designed to perform this kind of encoding. It takes x as input, represented as a list of 0s and 1s, and sets the qubits accordingly.

```
def state_preparation(x):
    qml.BasisState(x, wires=[0, 1, 2, 3])
```

The variational quantum circuit can be defined by combining state preparation and repeated layer structure.

```
@qml.qnode(dev)
def circuit(weights, x):
    state_preparation(x)

    for layer_weights in weights:
        layer(layer_weights)

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

We can also add trainable bias to the circuit to improve the performance of the classifier.

```
def variational_classifier(weights, bias, x):
    return circuit(weights, x) + bias
```

The cost function evaluates how well the model's predictions match the target labels. It usually combines a loss function and a regularizer. A regularizer is an additional term in the cost function to prevent overfitting and improve model generalization. It penalizes certain model parameters to keep them small.[6]

```
def square_loss(labels, predictions):
    # The square loss function calculates the
        mean squared error between the target
        labels and the model's predictions
    # We use a call to qml.math.stack to
        allow subtracting the arrays directly
    return np.mean((labels -
        qml.math.stack(predictions)) ** 2)

def accuracy(labels, predictions):
    # The accuracy function measures the
        proportion of correct predictions
    acc = sum(abs(l - p) < 1e-5 for l, p in
        zip(labels, predictions))
    acc = acc / len(labels)
    return acc

def cost(weights, bias, X, Y):
    predictions =
        [variational_classifier(weights,
        bias, x) for x in X]
    return square_loss(Y, predictions)
```

Loading the parity train data (download from):

```
data =
np.loadtxt("variational_classifier/data/parity_train.txt",
    dtype=int)
X = np.array(data[:, :-1])
Y = np.array(data[:, -1])
Y = Y * 2 - 1 # shift label from {0, 1} to
    {-1, 1}
```

Initialize the variables:

```
np.random.seed(0) # for reproducibility
num_qubits = 4
num_layers = 2
weights_init = 0.01 *
    np.random.randn(num_layers, num_qubits,
    3, requires_grad=True)
bias_init = np.array(0.0, requires_grad=True)
```

An optimizer in machine learning is an algorithm used to adjust the parameters of a model to minimize the cost function during training[7]. In the context of variational quantum classifiers, the optimizer iteratively updates the weights and biases of the quantum circuit. In this example, the Nesterov Momentum Optimizer is used, which is a kind of gradient descent optimizer[8].

```
opt = NesterovMomentumOptimizer(0.5)
batch_size = 5

weights = weights_init
bias = bias_init
for it in range(100):
        # Update the weights by one optimizer
            step, using only a limited batch of
            data
        batch_index = np.random.randint(0,
            len(X), (batch_size,))
        X_batch = X[batch_index]
        Y_batch = Y[batch_index]
        weights, bias = opt.step(cost, weights,
            bias, X=X_batch, Y=Y_batch)

        # Compute accuracy
        predictions =
            [np.sign(variational_classifier(weights,
            bias, x)) for x in X]

        current_cost = cost(weights, bias, X, Y)
        acc = accuracy(Y, predictions)

        print(f"Iter: {it+1:4d} | Cost:
            {current_cost:0.7f} | Accuracy:
            {acc:0.7f}")
```

The variational classifier correctly classified the training data but must generalize to unseen data. The challenge in (quantum) machine learning is to design models that generalize well. We need to check if it can generalize well using the training data (download from):

```
    data =
    np.loadtxt("variational_classifier/data/parity_test.t
        dtype=int)
    X_test = np.array(data[:, :-1])
    Y_test = np.array(data[:, -1])
    Y_test = Y_test * 2 - 1 # shift label from
        {0, 1} to {-1, 1}

    predictions_test =
        [np.sign(variational_classifier(weights,
        bias, x)) for x in X_test]

    for x,y,p in zip(X_test, Y_test,
        predictions_test):
        print(f"x = {x}, y = {y}, pred={p}")

    acc_test = accuracy(Y_test, predictions_test)
    print("Accuracy on unseen data:", acc_test)
```

If you run the code, you will see that the quantum circuit has learned to predict unseen data perfectly, which is remarkable. Despite many possible relations, the classifier correctly labels bit strings according to the parity function.

2. *Iris Classification*

Iris dataset consists of 2-D vectors. We will use *"latent dimensions"*, which helps to embed the higher dimensional data to lower dimensional space, and encode the input data to 2-qubits[9].

State preparation for iris data is more complex than bistring input. Each input vector x is translated into angles for state preparation. For simplicity, only positive subspace will be used.

The circuit is based on the schemes from Möttönen et al.[10] and Schuld & Petruccione[11] for positive vectors, with controlled Y-axis rotations decomposed into basic gates as per Nielsen & Chuang[12].

```
def get_angles(x):
    beta0 = 2 * np.arcsin(np.sqrt(x[1] ** 2)
        / np.sqrt(x[0] ** 2 + x[1] ** 2 +
        1e-12))
    beta1 = 2 * np.arcsin(np.sqrt(x[3] ** 2)
        / np.sqrt(x[2] ** 2 + x[3] ** 2 +
        1e-12))
    beta2 = 2 *
        np.arcsin(np.linalg.norm(x[2:]) /
        np.linalg.norm(x))

    return np.array([beta2, -beta1 / 2, beta1
        / 2, -beta0 / 2, beta0 / 2])
```

```
def state_preparation(a):
    qml.RY(a[0], wires=0)

    qml.CNOT(wires=[0, 1])
    qml.RY(a[1], wires=1)
    qml.CNOT(wires=[0, 1])
    qml.RY(a[2], wires=1)

    qml.PauliX(wires=0)
    qml.CNOT(wires=[0, 1])
    qml.RY(a[3], wires=1)
    qml.CNOT(wires=[0, 1])
    qml.RY(a[4], wires=1)
    qml.PauliX(wires=0)
```

Test if this actually works:

```
x = np.array([0.53896774, 0.79503606,
    0.27826503, 0.0], requires_grad=False)
ang = get_angles(x)

@qml.qnode(dev)
def test(angles):
    state_preparation(angles)

    return qml.state()

state = test(ang)

print("x : ", np.round(x, 6))
print("angles : ", np.round(ang, 6))
print("amplitude vector: ",
    np.round(np.real(state), 6))
```

The output:

```
x : [0.538968 0.795036 0.278265 0. ]
angles : [ 0.563975 -0. 0. -0.975046 0.975046]
amplitude vector: [ 0.538968 0.795036
    0.278265 -0. ]
```

The method successfully calculated the appropriate angles to achieve the target state preparation.

The *default.qubit* simulator can prepare states using *qml.StatePrep(x, wires=[0, 1])*. On state simulators, this replaces the state with our input. On hardware, it uses more advanced methods than implemented above.

```
def layer(layer_weights):
    for wire in range(2):
        qml.Rot(*layer_weights[wire],
            wires=wire)
    qml.CNOT(wires=[0, 1])


def cost(weights, bias, X, Y):
    # Transpose the batch of input data in
        order to make the indexing
    # in state_preparation work
    predictions =
        variational_classifier(weights, bias,
        X.T)
    return square_loss(Y, predictions)
```

Now, load the data set from here. We need tp preprocess the data to encode inputs into quantum state amplitudes.

- Extract the first two columns of the data. Only these two features will be used.
- Add padding to the data. Padding expands each 2-dimensional data point to 4 dimensions to match the quantum state vector size for a 2-qubit system. Padding keeps important information from the original data that might be lost during normalization if the vectors weren't expanded.
- Normalize the data points. The sum of the squares of state amplitudes must be equal to 1.
- Calculate rotation angles from the normalized data points to prepare them for quantum state preparation.

```
data =
    np.loadtxt("variational_classifier/data/
    iris_classes1and2_scaled.txt")
X = data[:, 0:2]
print(f"First X sample (original) : {X[0]}")

# pad the vectors to size 2^2=4 with constant
    values
padding = np.ones((len(X), 2)) * 0.1
X_pad = np.c_[X, padding]
print(f"First X sample (padded) : {X_pad[0]}")

# normalize each input
normalization = np.sqrt(np.sum(X_pad**2, -1))
X_norm = (X_pad.T / normalization).T
print(f"First X sample (normalized):
    {X_norm[0]}")

# the angles for state preparation are the
    features
features = np.array([get_angles(x) for x in
    X_norm], requires_grad=False)
print(f"First features sample :
    {features[0]}")

#Extracting labels
Y = data[:, -1]
```

The output:

```
First X sample (original) : [0.4 0.75]
First X sample (padded) : [0.4 0.75 0.1 0.1 ]
First X sample (normalized): [0.46420708
    0.87038828 0.11605177 0.11605177]
First features sample : [ 0.32973573
    -0.78539816 0.78539816 -1.080839 1.080839
    ]
```

These angles are our new features, so we renamed X to "*features*".

Data will be divided into training and testing sets.

```
np.random.seed(0)
num_data = len(Y)
num_train = int(0.75 * num_data)
index = np.random.permutation(range(num_data))
feats_train = features[index[:num_train]]
Y_train = Y[index[:num_train]]
feats_val = features[index[num_train:]]
Y_val = Y[index[num_train:]]

# We need these later for plotting
X_train = X[index[:num_train]]
X_val = X[index[num_train:]]
```

Initializing variables for training:

```
num_qubits = 2
num_layers = 6

weights_init = 0.01 *
    np.random.randn(num_layers, num_qubits,
    3, requires_grad=True)
bias_init = np.array(0.0, requires_grad=True)
```

Training using *NesterovMomentumOptimizer*:

```
opt = NesterovMomentumOptimizer(0.01)
batch_size = 5

# train the variational classifier
weights = weights_init
bias = bias_init
for it in range(60):
    # Update the weights by one optimizer step
    batch_index = np.random.randint(0,
        num_train, (batch_size,))
    feats_train_batch =
        feats_train[batch_index]
    Y_train_batch = Y_train[batch_index]
    weights, bias, _, _ = opt.step(cost,
        weights, bias, feats_train_batch,
        Y_train_batch)

    # Compute predictions on train and
        validation set
    predictions_train =
        np.sign(variational_classifier(weights,
        bias, feats_train.T))
    predictions_val =
        np.sign(variational_classifier(weights,
        bias, feats_val.T))

    # Compute accuracy on train and
        validation set
    acc_train = accuracy(Y_train,
        predictions_train)
    acc_val = accuracy(Y_val, predictions_val)
```

The plot the classifier's output for the first two dimensions of the Iris dataset is given below. We see that the variational classifier learned to separate the two classes with a line, allowing it to classify even the unseen data perfectly.

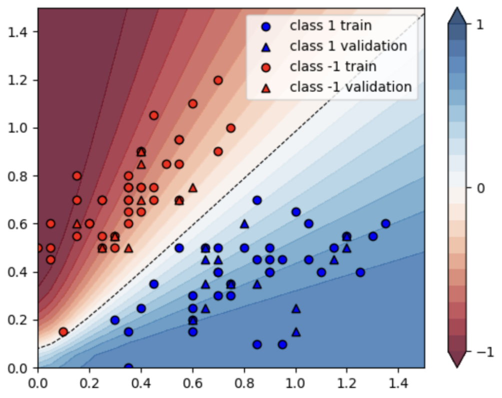


FIG. 2
The classifier's output for the first two dimensions of the Iris dataset[3].

STEP 3: QUANVOLUTIONAL NEURAL NETWORKS[13]

In this demo, we create a Quanvolutional Neural Network, a type of quantum machine learning model first introduced by Henderson et al. in 2019[14].

Convolutional Neural Networks (CNNs) are commonly used in classical machine learning for image processing by applying local convolutions to small regions of an image with the same kernel. The results are combined into output pixels, forming a new image-like object for further processing.

Quantum variational circuits can extend convolution to quantum computing. In this approach, small regions of an image, like 2×2 squares, are embedded into quantum circuits using parameterized rotations. A unitary operation is performed, and the system is measured to obtain classical expectation values, which are mapped to channels of an output pixel. Repeating this across different regions processes the entire image into a multi-channel output. Quantum circuits can create complex kernels that are difficult to compute classically. The tutorial uses a fixed, non-trainable "quanvolution" kernel, but these kernels can also be trained using PennyLane's gradient evaluation capabilities.

Set the hyperparameters:

```
n_epochs = 30 # Number of optimization epochs
n_layers = 1 # Number of random layers
n_train = 50 # Size of the train dataset
n_test = 30 # Size of the test dataset

SAVE_PATH =
    "../_static/demonstration_assets/quanvolution/"
    # Data saving folder
PREPROCESS = True # If False, skip quantum
    processing and load data from SAVE_PATH
np.random.seed(0) # Seed for NumPy random number
    generator
tf.random.set_seed(0) # Seed for TensorFlow
    random number generator
```

Import the MNIST dataset from Keras.

```
mnist_dataset = keras.datasets.mnist
(train_images, train_labels), (test_images,
    test_labels) = mnist_dataset.load_data()

# Reduce dataset size
train_images = train_images[:n_train]
train_labels = train_labels[:n_train]
test_images = test_images[:n_test]
test_labels = test_labels[:n_test]

# Normalize pixel values within 0 and 1
train_images = train_images / 255
test_images = test_images / 255

# Add extra dimension for convolution channels
train_images = np.array(train_images[...,
    tf.newaxis], requires_grad=False)
```

```
test_images = np.array(test_images[...,
    tf.newaxis], requires_grad=False)
```

We set up a PennyLane *default.qubit* device to simulate a system with 4 qubits. QNode includes the following components:

- An embedding layer with local (R_y) rotations, where angles are scaled by a factor of (π).
- A random circuit with a specified number of layers (*n_layers*).
- A final measurement in the computational basis to estimate 4 expectation values.

```
dev = qml.device("default.qubit", wires=4)
# Random circuit parameters
rand_params = np.random.uniform(high=2 * np.pi,
    size=(n_layers, 4))

@qml.qnode(dev)
def circuit(phi):
    # Encoding of 4 classical input values
    for j in range(4):
        qml.RY(np.pi * phi[j], wires=j)

    # Random quantum circuit
    RandomLayers(rand_params,
        wires=list(range(4)))

    # Measurement producing 4 classical output
        values
    return [qml.expval(qml.PauliZ(j)) for j in
        range(4)]
```

The convolution procedure:

- The image is divided into 2×2 pixel squares.
- Each square is processed by the quantum circuit.
- The 4 expectation values obtained are mapped to 4 different channels of a single output pixel.

```
def quanv(image):
    """Convolves the input image with many
        applications of the same quantum
        circuit."""
    out = np.zeros((14, 14, 4))

    # Loop over the coordinates of the top-left
        pixel of 2X2 squares
    for j in range(0, 28, 2):
        for k in range(0, 28, 2):
            # Process a squared 2x2 region of the
                image with a quantum circuit
            q_results = circuit(
                [
                    image[j, k, 0],
                    image[j, k + 1, 0],
                    image[j + 1, k, 0],
                    image[j + 1, k + 1, 0]
                ]
            )
            # Assign expectation values to
                different channels of the output
                pixel (j/2, k/2)
            for c in range(4):
                out[j // 2, k // 2, c] =
                    q_results[c]
    return out
```

Since the quantum convolution layer won't be trained, it's more efficient to use it as a pre-processing step for the entire dataset. Pre-processed images are saved to SAVE_PATH and can be loaded by setting PREPROCESS = False. If PREPROCESS is True, the quantum convolution is applied each time the code runs.

```
if PREPROCESS == True:
    q_train_images = []
    print("Quantum pre-processing of train
        images:")
    for idx, img in enumerate(train_images):
        print("{}/{} ".format(idx + 1, n_train),
            end="\r")
        q_train_images.append(quanv(img))
    q_train_images = np.asarray(q_train_images)

    q_test_images = []
    print("\nQuantum pre-processing of test
        images:")
    for idx, img in enumerate(test_images):
        print("{}/{} ".format(idx + 1, n_test),
            end="\r")
        q_test_images.append(quanv(img))
    q_test_images = np.asarray(q_test_images)

    # Save pre-processed images
    np.save(SAVE_PATH + "q_train_images.npy",
        q_train_images)
    np.save(SAVE_PATH + "q_test_images.npy",
        q_test_images)


# Load pre-processed images
q_train_images = np.load(SAVE_PATH +
    "q_train_images.npy")
q_test_images = np.load(SAVE_PATH +
    "q_test_images.npy")
```

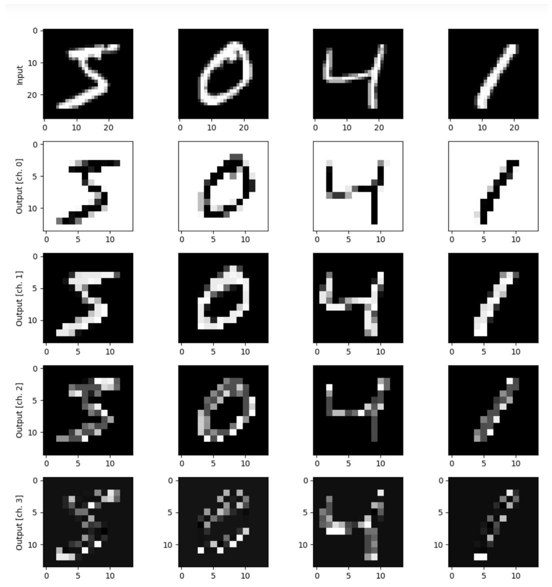


FIG. 3
Below each input image, the 4 output channels from the quantum convolution are displayed in grayscale. The 4 grayscale output channels from the quantum convolution show reduced resolution and some local distortion, but the overall image shape remains preserved.

After applying the quantum convolution layer, the features are input into a simple classical neural network for classifying the 10 MNIST digits. The model consists of a fully connected layer with 10 output nodes and a softmax activation function, and is compiled with stochastic gradient descent and a cross-entropy loss function.

```
def MyModel():
    """Initializes and returns a custom Keras
        model
    which is ready to be trained."""
    model = keras.models.Sequential([
        keras.layers.Flatten(),
        keras.layers.Dense(10,
            activation="softmax")
    ])

    model.compile(
        optimizer='adam',
        loss="sparse_categorical_crossentropy",
        metrics=["accuracy"],
    )
    return model
```

We first initialize the model, then train and validate it using the pre-processed dataset from the quantum convolution.

```
q_model = MyModel()

q_history = q_model.fit(
    q_train_images,
    train_labels,
    validation_data=(q_test_images, test_labels),
    batch_size=4,
    epochs=n_epochs,
    verbose=2,
)
```

To compare results, we also initialize a "classical" model that is trained and validated directly with the raw MNIST images, without quantum pre-processing.

```
c_model = MyModel()

c_history = c_model.fit(
    train_images,
    train_labels,
    validation_data=(test_images, test_labels),
    batch_size=4,
    epochs=n_epochs,
    verbose=2,
)
```

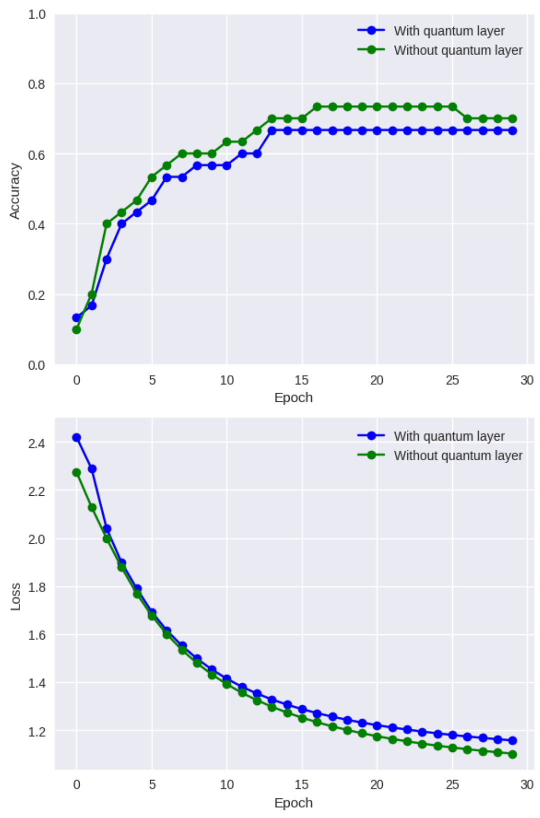


FIG. 4
Plot of the test accuracy and the test loss of classical and quantum-classical models with respect to the number of training epochs.

STEP 4: LEARNING SINE FUNCTION WITH QNN

In this step, we developed Quantum Neurel Network model to learn sine function on the interval $[0, 2\pi]$.

First, we need to install pennylane library. You may need to restart the kernel after the installation.

```
pip install pennylane
```

Import the necessary libraries.

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

We need to create train and test data for the model.

```
# Generate angles between 0 and 2pi
angles = np.linspace(0, 2 * np.pi, 1000)
# Suffle the angles to get randomized data
    points for train and test
angles_shuffled =
    np.random.permutation(angles)

# Compute sine values for these angles
sine_values = np.sin(angles_shuffled)

# Generate training data
x_train = angles_shuffled[:80]
y_train = sine_values[:80]

# Generate testing data
x_test = angles_shuffled[80:]
y_test = sine_values[80:]
```

Initialize the variables. We will use one qubit which is enough for encoding an angle variable. Set up the optimizer.

```
# Define the number of qubits and layers
n_qubits = 1
n_layers = np.arange(5)
n_epochs = 100
#np.random.seed(0)

# Set up the optimizer
opt =
    qml.GradientDescentOptimizer(stepsize=0.1)
```

Define QNN circuit. We use angle embedding to embed clasical data to the quantum circuit.

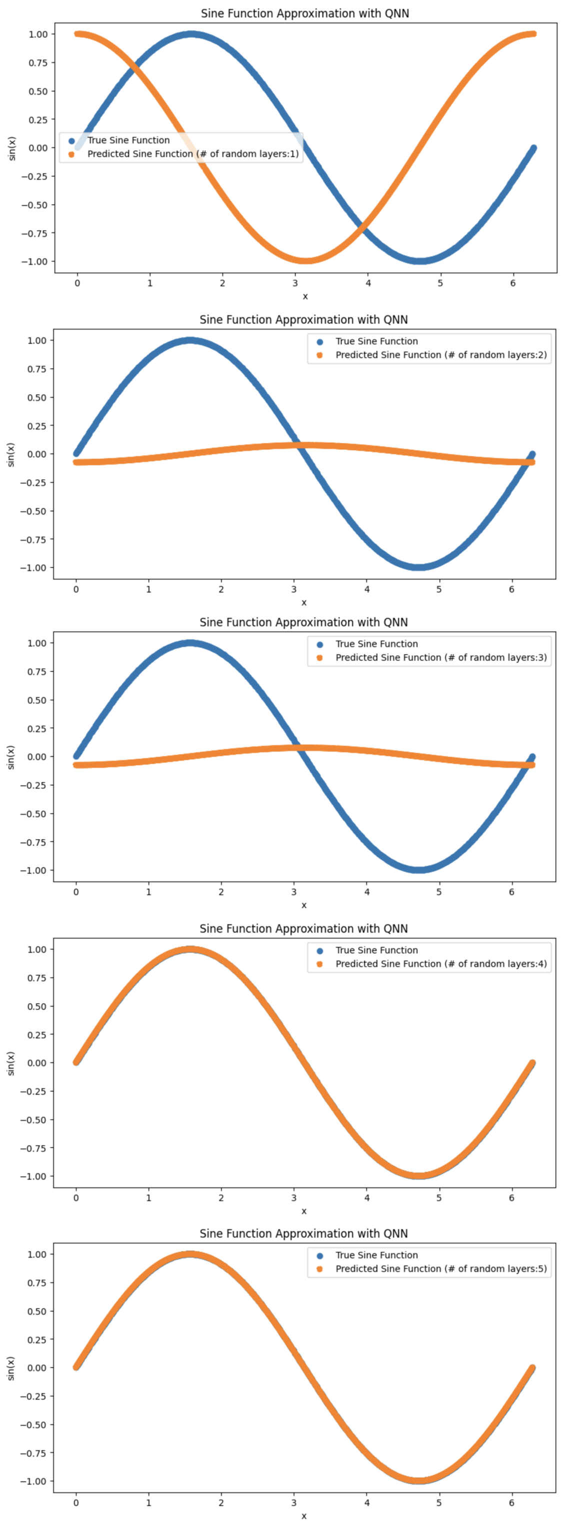
```
# Initialize a device
dev = qml.device("default.qubit",
    wires=n_qubits)

# Define the QNN circuit
@qml.qnode(dev)
def qnn_circuit(params, x):
    qml.AngleEmbedding([x],
        wires=range(n_qubits))
    qml.RandomLayers(params,
        wires=range(n_qubits))
    return
        qml.expval(qml.PauliZ(wires=range(n_qubits)))
```

Define the cost function.

```
# Define the cost function
def cost(params, x, y):
    predictions = [qnn_circuit(params, x_i)
        for x_i in x]
    return np.mean((np.array(predictions) -
        y) ** 2)
```

Train and test the model for different number of layers.

```
for l in n_layers:

    print(f"Number of layers: {l+1}")

    params = 0.01 * np.random.randn(l,
        n_qubits)

    # Train the model
    for epoch in range(n_epochs):
        params, cost_val =
            opt.step_and_cost(lambda v:
            cost(v, x_train, y_train), params)

        if (epoch + 1) % 20 == 0:
            print(f"Epoch {epoch + 1}: Cost =
                {cost_val:.4f}")

    # Testing
    # Predict using the trained QNN
    y_pred = [qnn_circuit(params, x_i) for
        x_i in x_test]

    # Plot the results
    plt.figure(figsize=(10, 5))
    plt.scatter(x_test, y_test, label='True
        Sine Function')
    plt.scatter(x_test, y_pred, label=
        f'Predicted Sine Function (# of
        random layers:{l+1})',
        linestyle='dashed')
    plt.title('Sine Function Approximation
        with QNN')
    plt.xlabel('x')
    plt.ylabel('sin(x)')
    plt.legend()
    plt.show()
```



.3

FIG. 5: QNN model trained with different number of random layers.

It is observed from above that 4 or 5 random layers are enough for model to learn sine function.

[1] Xanadu, “PennyLane: An open-source software framework for quantum machine learning, quantum chemistry, and quantum computing,” https://pennylane.ai/ (Accessed: 2024-07-20).

[2] PennyLane, "Prepare yourself," (2024), accessed: 2024-07-25.

[3] P. AI, "Tutorial: Variational quantum classifiers," (2024), accessed: 2024-07-27.

[4] E. Farhi and H. Neven, "Classification with quantum neural networks on near term processors," (2018), arXiv:1802.06002 [quant-ph].

[5] M. Schuld, A. Bocharov, K. M. Svore, and N. Wiebe, Physical Review A **101** (2020), 10.1103/physreva.101.032308.

[6] Simplilearn, "Regularization in machine learning," (2023), accessed: 2024-07-27.

[7] A. Vidhya, (2021).

[8] PennyLane, "Nesterovmomentumoptimizer," (2024).

[9] T. S. Francisco, "What is latent space in ai?" (2024), accessed: 2024-07-29.

[10] M. Mottonen, J. J. Vartiainen, V. Bergholm, and M. M. Salomaa, "Transformation of quantum states using uniformly controlled rotations," (2004), arXiv:quant-ph/0407010 [quant-ph].

[11] M. Schuld and F. Petruccione, *Supervised Learning with Quantum Computers*, 1st ed., Quantum Science and Technology (Springer Cham, 2018) pp. XIII, 287, published: 22 September 2018, Softcover ISBN: 978-3-030-07188-2, Published: 03 January 2019.

[12] M. A. Nielsen and I. L. Chuang, *Quantum Computation and Quantum Information* (Cambridge University Press, Cambridge, 2000).

[13] P. Team, "Quanvolutional neural networks with pennylane," `https://pennylane.ai/qml/demos/tutorial_quanvolution/` (2024), accessed: 2024-08-09.

[14] M. Henderson, S. Shakya, S. Pradhan, and T. Cook, arXiv preprint arXiv:1904.04767 (2019).