Quantum Machine Learning for Conspicuity Detection in Production

Task1:

Pennylane Tutorial Codebook:

The Pennylane tutorial codebook for Learning Quantum Computing consist of the basic theory and theory formulations via Pennylane code for Understanding the basics of Quantum Computation which lays as the foundation for Quantum Machine learning.

A) The Introductory part gives us an understanding about the Single Qubit representations, Qubit operations and Normalization of Quantum states.
B) The single-qubit state part gives idea about state manipulation such as state preparation, qubit rotation, single qubit gate operations etc.
C) The third part gives us understanding about multi qubit quantum circuits, entanglement , circuit manipulation for desired quantum state and a multi qubit gate challenge.

Task2:

Quantum Machine Learning:

Classical Machine Learning Focuses on Statistical Analysis of Data to Predict patterns which can be used to classify new data samples. Quantum Machine Learning is a hybrid approach. The combination of Quantum Circuits with Classical Machine Learning models.

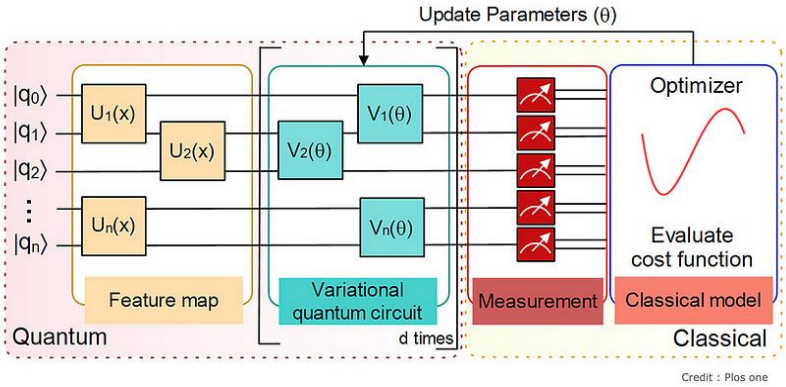
Concepts to Remember:

1) **Cost Function:** The cost function is generally an Optimization Parameter used in Classical as well as Quantum circuits. This cost function is Optimized according to an Optimization Strategy which will lead us towards the minimum value of the function. In a supervised learning model, the cost function is a combination of a loss function and a regularizer.
2) **Gradient:** Gradient points in the direction of the steepest Ascent. Basically, it acts as a useful constrain on the cost function. It gives us the idea to Move in a direction opposite to the Steep to get a lower value.
3) **Layer:** Layer forms the elementary building block for a VQ circuit which is replicated to form the entire circuit.
4) **QNode:** The parameterized quantum circuit in QML part is encoded as QNode in the QML model. We Optimize these parameters with the help of a classical ML model to minimize the cost function.
5) **Optimization:** Optimization is generally finding the best classical ML algorithm to optimize the parameters in VQC circuit. (Training using different models.)

Variational Quantum Classifiers:

VQC are quantum circuits that can be trained form labelled data to classify new data samples. The VQC consists of a sequence of parameter dependent unitary transformations which acts on an input quantum state.
The Input state is fed into the parameterized circuit and the resulting cost function is given to a classical optimizer which evaluates the cost function and give better parameter suggestions. We update the parameters according to the suggestions and repeat the process till maximum optimality is obtained.

Image from Medium Article



Our objective for the task is to represent Basis Encoding (Encoding the binary inputs into the basis states of Variational Circuits) and Amplitude Encoding (Real vectors as Amplitude vectors into quantum states).

Objective 1: Basis Encoding

Basis encoding shows us how to optimize Variational Quantum Circuit to emulate the Parity function. The parity function is a binary function defined form x to y where x is an n bit binary string such that

$$f : x \in \{0,1\}^{\otimes n} \rightarrow y = \begin{cases} 1 \text{ if uneven number of 1's in } x \\ 0 \text{ else.} \end{cases}$$

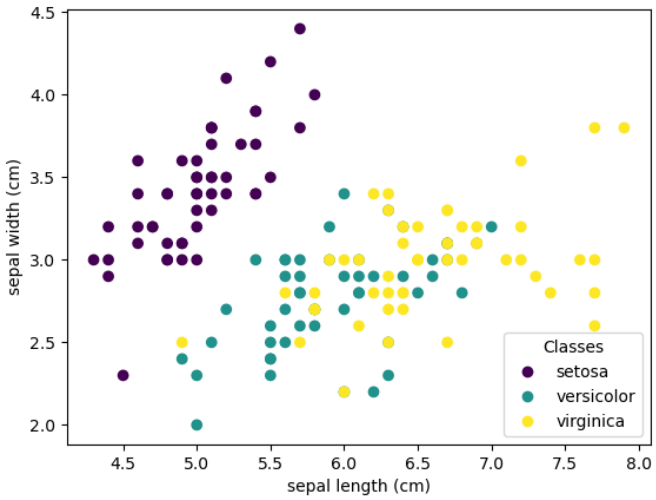
This is done via the standard steps followed in QML.

a) Importing the Necessary libraries and Optimizers. (Here the NesterovMomentumOptimizer is used which is a classical ML optimizer.)
b) Defining a Device – We need a device specified for running our quantum algorithm.
c) Defining Layer, Encoding the Classical bit string into quantum state and defining QNode.
d) Defining the cost function.
e) Optimization

Here the cost function is a standard square loss function which outputs the distance between the target labels and our model predictions.
We also define a function for getting the accuracy as a measure of the proportion of prediction that aggress with the actual label.

Objective 2: Amplitude Encoding (The IRIS Dataset Classification)

The Iris dataset consist of 3 different types of irises' (Sentosa, Versicolour, and Virginica) petal and sepal length, stored in a 150x4 NumPy array. The datapoints in the dataset are 2-dimensional vectors.



Here the state preparation is different from that of Basis Encoding. Each input data (2 dim vector) is converted to a set of angles which is fed into a routine for state preparation.
We need to update the layer function as we are working with two qubits and update the cost function as well.

Task 3:

Quanvolutional Neural Network

The objective of task 3 is to implement the Quantum version of Convolutional Neural Network model which is widely used for image recognition and related applications.

Convolutional Neural Network:

Convolutional Neural Network is a type of Deep Learning Neural Network Architecture which is used for visual recognition of Images and data.

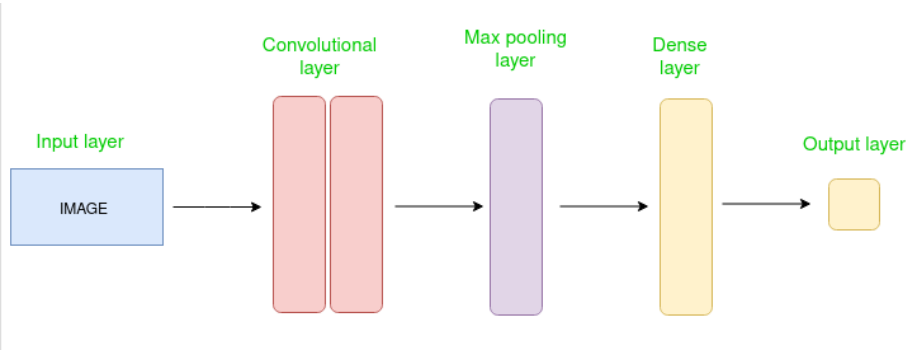


Image form Geeks for Geeks

Convolution: Convolution is a technique used in a variety of image processing algorithms. Convolution provides a way of multiplying two arrays of same dimension to produce another array of the same dimension.

The Convolution layer applies filters to the image to extract features. The convolutional layer consists of a set of learnable layers or filters which strides over the image to produce a new set of layers which gets stacked together as a new image.

This new image is called a feature map.

Quantum Convolution

QNN's are quantum networks similar to classical CNN's with an extra Quantum Convolution layer. Unlike the simple element wise matrix multiplication that Classical model applies Quantum convolution model transforms the input data using quantum network.

In task 3 we apply Quantum Convolution on the MINST dataset.

General Steps in Quantum Convolution of an Image Dataset:

a) We initially take a small part of the image. For example, if the image is of the order m*m pixel matrix we take the par n*n where n < m.
b) We then encode the n*n matrix into a quantum circuit by performing unitary transformations on the qubits in ground state.
c) We use a Quantum Circuit which is either structured or Random as a Quanvolutional filter on the Input quantum state and we measure the output to obtain the classical expectation value of the output data.
d) Analogous to the Future mapping step in classical convolution each expectation dataset value is mapped to a different channel of a single output pixel.
e) Similarly all the parts of the Input image will undergo the same process (stride) and a new output image is created (multi-channel-image).

Implementation:

a) In the code book initially, we import all the necessary functions required for Quantum as well as Classical Training models.

```
import pennylane as qml

from pennylane import numpy as np

from pennylane.templates import RandomLayers

import tensorflow as tf

from tensorflow import keras

import matplotlib.pyplot as plt
```

b) We then initialize the hyper parameters of the model which includes the number of layers, size of test and train data, path for saving the data and the value of preprocessing as "True".

```
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
```

c) We then load the MNIST data set using 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)
```

d) We then define a quantum device and Q node using penny lane as done in task 2 and would encode the n*n (Here 2*2) image data into the qubits defined by performing unitary transformations.

```
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)
```

Then we define a random quantum circuit.

```
    # 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)]
```

e) Next, we define a function that does the quanvoloution of the image. Each part (2*2 matrix array) of the image is processed by the quantum circuit and the 4 output expectation values 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
```

Quantum Pre-Processing:

The quantum convolution layer is applied as a pre-processing layer to all the images in the dataset. That is the expectation values plotted as different channel of the output pixel after quantum processing is used to train and classify an entire classical data set and in order to avoid repetition of training the Quantum model again, we use the PREPROCESS variable.

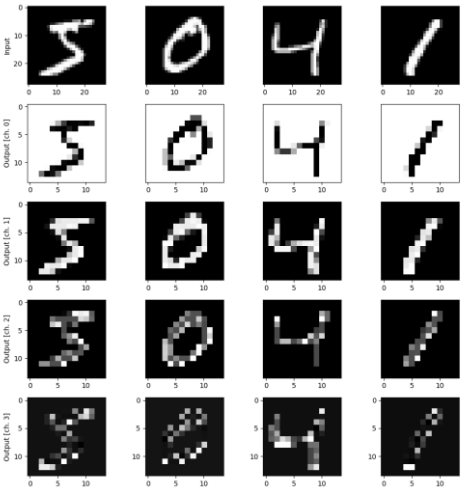
```
if PREPROCESS == True:
    q_train_images = []
    print("Quantum pre-processing of train images:")
```

```
    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")
```

Visualizing the effect of Quantum Convolution on the batch sample:

```
n_samples = 4
n_channels = 4
fig, axes = plt.subplots(1 + n_channels, n_samples, figsize=(10, 10))
for k in range(n_samples):
    axes[0, 0].set_ylabel("Input")
    if k != 0:
        axes[0, k].yaxis.set_visible(False)
    axes[0, k].imshow(train_images[k, :, :, 0], cmap="gray")
    # Plot all output channels
    for c in range(n_channels):
        axes[c + 1, 0].set_ylabel("Output [ch. {}]".format(c))
        if k != 0:
            axes[c, k].yaxis.set_visible(False)
        axes[c + 1, k].imshow(q_train_images[k, :, :, c], cmap="gray")
plt.tight_layout()
plt.show()
```



Four different output models of the input images are obtained after passing through the Quanvolutiona Network.

The Hybrid Approach:

This output features obtained via training a part of the dataset via the QNN is fed into a classical training model and is used to train and classify the MINST data set.

1) Defining the Model:

```
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
```

2) Training the pre-processed dataset

```
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,
)
```

3) Visualization: