Assignment 3: QML Classification

Sunayana Gupta
1222389090
sgupt279@asu.edu

In this assignment, we had to work on the Pennylane library to perform Quantum Machine Learning Classification using a QML simulator provided by Pennylane.

Data:
For this I downloaded the original Iris dataset from the sklearn library using below code snippets:

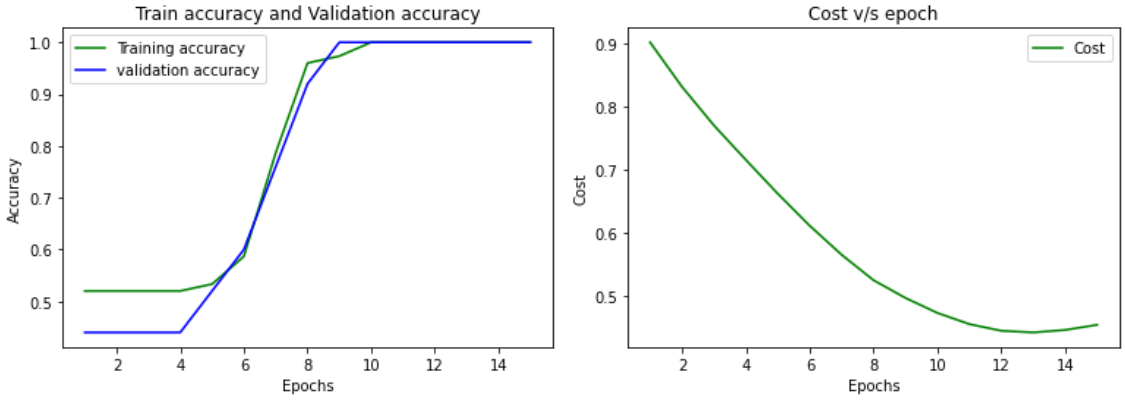
```
from sklearn import datasets

# load iris dataset
iris = datasets.load_iris()
# Since this is a bunch, create a dataframe
iris_df=pd.DataFrame(iris.data)
iris_df['class']=iris.target

iris_df.columns=['sepal_len', 'sepal_wid', 'petal_len', 'petal_wid', 'class']
iris_df = iris_df[iris_df['class'] != 2]
iris_df.loc[iris_df["class"] == 0, "class"] = 2
iris_df.loc[iris_df["class"] == 1, "class"] = -1
iris_df.loc[iris_df["class"] == 2, "class"] = 1

Y = iris_df['class']
X= iris_df.drop(columns=['class'])
X= X.to_numpy()
Y = Y.to_numpy()
```

Here I have removed the data samples from the last class and tried to work only on two of the classes.

Variation of embeddings:
Once this data was ready, I performed 4 types of embedding i.e., amplitude embedding, RX, RY, and RZ type of rotation embedding. This was done using the library functions from Pennylane.
We will first discuss amplitude embedding. Below is the snippet that I used to do amplitude embedding.

```
def statepreparation(a):
  qml.templates.embeddings.AmplitudeEmbedding(a, wires=list(range(2)), normalize=True)
```

I have used 2 qubits to do amplitude embedding. In this technique, the data is encoded into the amplitudes of a quantum state. For this experiment, I haven't changed the ansatz and used the default one. Ran the code for 6 layers with Adam Optimizer. And with these changes, I got a validation accuracy of 100% on the 9th epoch in comparison to the one given in the tutorial where 100% accuracy was achieved at the 47th iteration for the first time. This proves that using all 4 attributes without the padding of the dataset along with Adam Optimizer heavily improves the convergence. Below are the graphs showing the quick convergence.

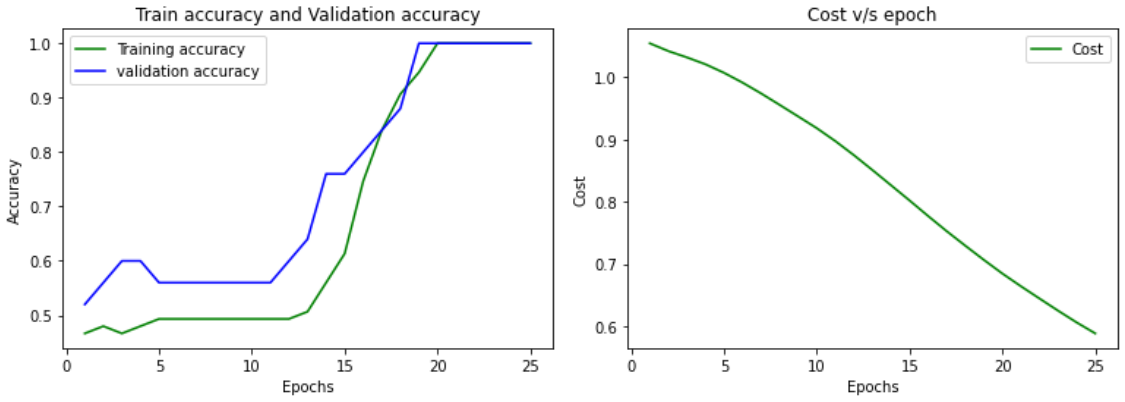


As we can see, the model converged at the 13th epoch in this experiment as the cost is the minimum in that iteration.

Next, I worked on the RX rotation embedding.

```
def statepreparation(a):
  qml.AngleEmbedding(features=a, wires=list(range(4)), rotation='X')
```

For rotation embedding, if n features are there then n qubits are required so for our Iris data, I took 4 qubits. Also, for rotation embedding the normalization is done between 0 and pi. Here again, I have taken the default layer structure from the tutorial for 4 qubits for the comparison between all 4 kinds of embedding. After running the model, it converged on the 19th iteration, and below are the result graphs:

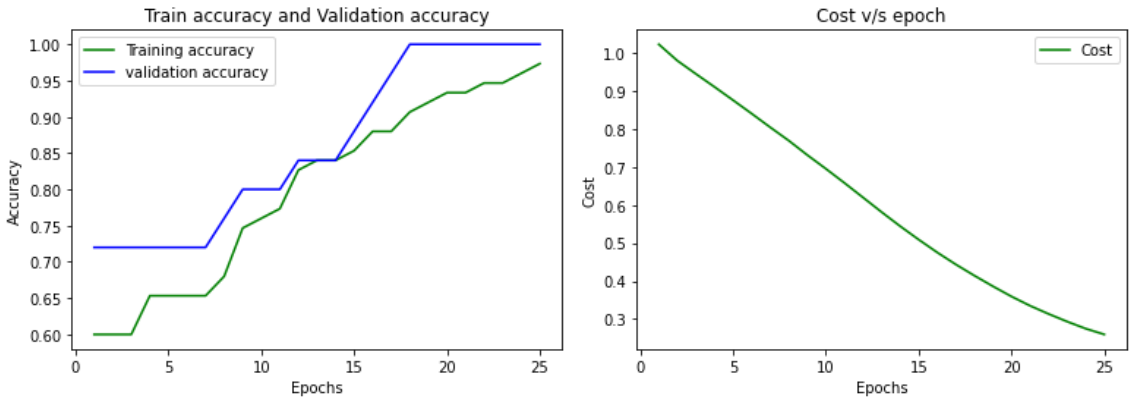


From the graphs, we can see that the RX embedding model converged comparatively slowly as compared to the amplitude embedding. This is because amplitude embedding took only log(n) qubits whereas rotation embedding took n qubits so to converge all the n qubits, 4 in our case it takes more resources and more computation and hence it converges slowly as compared to the amplitude embedding model. But angle embedding might be more useful where we want to look at more granularity of the data while it trains.

Thereafter I worked on RY embedding.

```
def statepreparation(a):
  qml.AngleEmbedding(features=a, wires=list(range(4)), rotation='Y')
```

I kept every configuration the same as RZ embedding except the embedding axis from X to Y. Below are the results I got after performing this experiment:

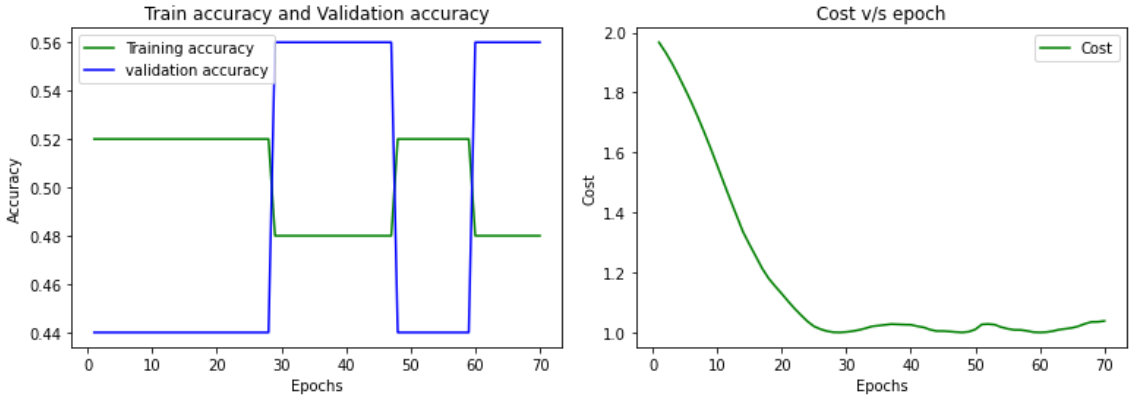


Here also we got 100% validation accuracy in the 18th epoch and therefore the RY embedding model performed in a similar way to the RX embedding model.

Next, I tried the RZ embedding again with the same configurations as RY and RX embedding models but just changed the embedding axis from 'Y' to 'Z'.

```
def statepreparation(a):
  qml.AngleEmbedding(features=a, wires=list(range(4)), rotation='Z')
```

Below are the results when I ran the model with RZ embedding.



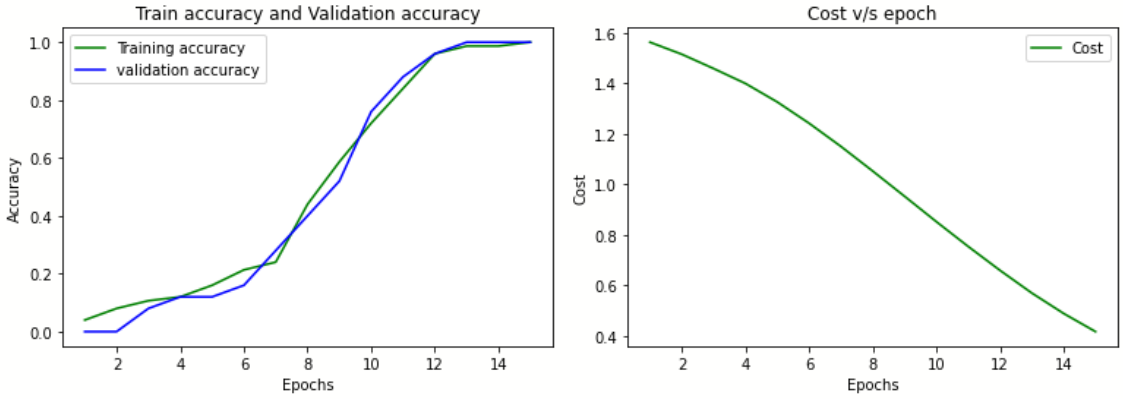
As we can see the model didn't converge at all with RZ embedding. I got similar results when I changed the exp value from PauliZ() to PauliX() and hence concluded that RZ embedding models are not converging. This is because in the case of RZ rotation every data sample will be mapped to $|0\rangle$ state and the encoded value will be lost. Hence we got a model with RZ that didn't converge at all.

Ansatz Variation:

After finishing all the embedding experiments, I started varying the ansatz of the model, specifically the layers of the model. I have kept the RZ angle embedding constant for all the models with different ansatz.
Firstly I created below layer structure:

```
def layer(W):

        qml.RX(W[0,1], wires=0)
        qml.RX(W[1,1], wires=1)
        qml.RX(W[2,1], wires=2)
        qml.RX(W[3,1], wires=3)

        qml.broadcast(qml.CNOT, wires=[0, 1, 2, 3], pattern="ring")
        qml.RY(W[0,2], wires=0)
        qml.RY(W[1,2], wires=1)
        qml.RY(W[2,2], wires=2)
        qml.RY(W[3,2], wires=3)
        qml.broadcast(qml.CNOT, wires=[0, 1, 2, 3], pattern="ring")
        qml.RZ(W[0,0], wires=0)
        qml.RZ(W[1,0], wires=1)
        qml.RZ(W[2,0], wires=2)
        qml.RZ(W[3,0], wires=3)
        qml.Hadamard(wires=1)

        qml.broadcast(qml.CNOT, wires=[0, 1, 2, 3], pattern="ring")
```

This structure is inspired from the basic structure that was provided in the tutorial. Since our weight matrix had a shape of 4,3 I had decided to use different rotation axis for all different columns of the weight matrix and hence kept RX, RY and RZ as the single qubit rotation instead of the Rot. And I have kept a broadcast of CNOT gates after each kind of rotation that works on the gates ina ring fashion where firstly it works on 0,q wire then 1,2 then 2,3 and lastly 3,1. This way the CNOT gate entangles all the values. The basic function of CNOT is to flip the second qubit if first is $|1\rangle$. I designed this structure so that it will give much more variation to the plain structure given in the tutorial. Below were the results that I obtained working on this ansatz.

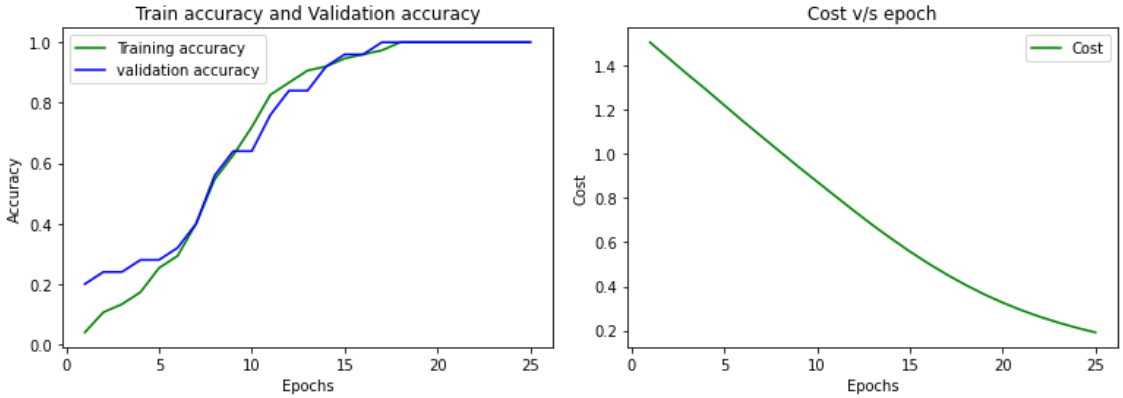


As expected this structure converged way before the plain RY model which converged on 18th epoch but with this structure it converged on 13th epoch only. This experiment was done using the same number of layers that was 6 across all the models made here. This was because of more variation in the model which in turn makes the model reach the local minima of cost quicker.

Then I decided to experiment more on the ansatz and work on the quantum chemistry gates for constructing the ansatz layers. Below is the layer structure that I created for the classification:

```
def layer(W):

        qml.SingleExcitation(W[0,0],wires=[0, 1])
        qml.SingleExcitation(W[0,1],wires=[1, 2])
        qml.SingleExcitation(W[0,2],wires=[2, 3])


        qml.broadcast(qml.CNOT, wires=[0, 1, 2, 3], pattern="ring")
        qml.SingleExcitation(W[1,0],wires=[0, 1])
        qml.SingleExcitation(W[1,1],wires=[1, 2])
        qml.SingleExcitation(W[1,2],wires=[2, 3])


        qml.broadcast(qml.CNOT, wires=[0, 1, 2, 3], pattern="ring")
        qml.SingleExcitation(W[2,0],wires=[0, 1])
        qml.SingleExcitation(W[2,1],wires=[1, 2])
        qml.SingleExcitation(W[2,2],wires=[2, 3])

        qml.Hadamard(wires=1)

        qml.broadcast(qml.CNOT, wires=[0, 1, 2, 3], pattern="ring")
```

Here I have used the single excitation gate. These excitation are related to the excitation of electron in the orbitals of an atom Hartree-Fock method is used to minimize the total energy function of the electron in these models which help it to converge. Pennylane provides all the functionality to compute the gradient of such function inherently. This is called single excitation as both the used states $|10\rangle$,$|01\rangle$ differ by the excitation of a single particle. Each of these electronic excitations is represented by a gate that excites electrons from the occupied orbitals of a reference state to the unoccupied ones. Here instead of the Rotation gates I have used the chemistry excitation gates and used CNOT gates in between. This gave below performance:

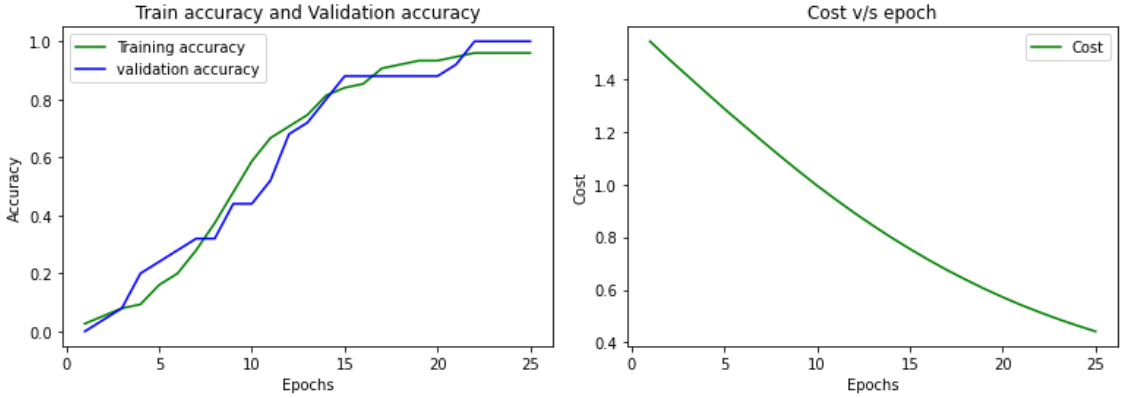


This gave 100% validation accuracy on the 17th iteration. So the performance of this layer structure is the same as the one we got with the basic structure given in the Pennylane tutorial.

Next I performed the same experiment with double excitation thinking it would improve the performance of the model.

```
def layer(W):

        qml.DoubleExcitation(W[0,0],wires=[0,1,2,3])
        qml.DoubleExcitation(W[0,1],wires=[0,1,2,3])
        qml.DoubleExcitation(W[0,2],wires=[0,1,2,3])

        qml.broadcast(qml.CNOT, wires=[0, 1, 2, 3], pattern="ring")
        qml.DoubleExcitation(W[1,0],wires=[0,1,2,3])
        qml.DoubleExcitation(W[1,1],wires=[0,1,2,3])
        qml.DoubleExcitation(W[1,2],wires=[0,1,2,3])


        qml.broadcast(qml.CNOT, wires=[0, 1, 2, 3], pattern="ring")
        qml.DoubleExcitation(W[2,0],wires=[0,1,2,3])
        qml.DoubleExcitation(W[2,1],wires=[0,1,2,3])
        qml.DoubleExcitation(W[2,2],wires=[0,1,2,3])

        qml.Hadamard(wires=1)

        qml.broadcast(qml.CNOT, wires=[0, 1, 2, 3], pattern="ring")
```

Below are the results that I got using double excitation gates.

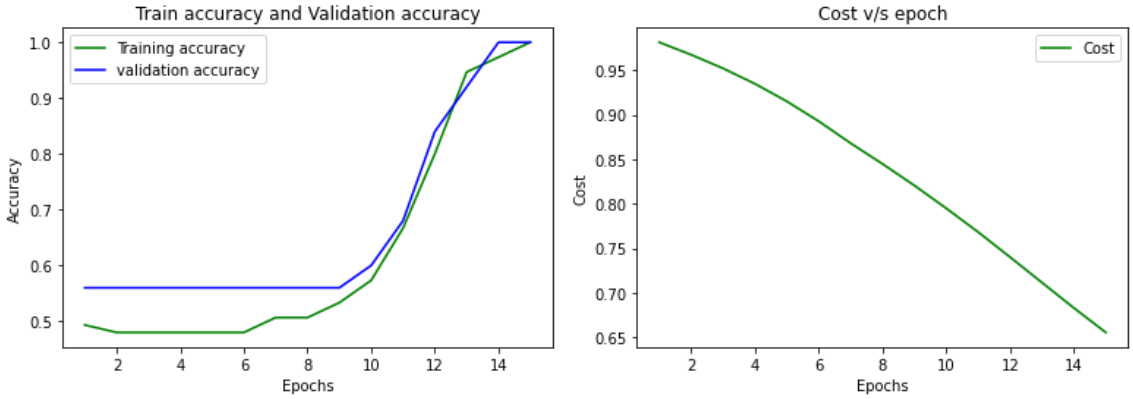


To my surprise, this gave a worse performance to what we got with the basic tutorial structure. This Double excitation layer model gave 100% validation accuracy in the 22nd iteration whereas the basic one achieved 100% accuracy in the 18th iteration. Therefore quantum chemistry gates proved to be not that useful for our use case.

Finally, I tried to work on other MultiRZ (arbitrary multi z rotation) and Pauli Rot gates for making the layer structure and got rid of all the CNOT gates and kept the Hadamard gate on each subsequent wire. Below is the structure that I created:

```
def layer(W):

        qml.MultiRZ(W[0,0],wires=[0,1,2,3])
        qml.MultiRZ(W[0,1],wires=[0,1,2,3])
        qml.MultiRZ(W[0,2],wires=[0,1,2,3])
        qml.Hadamard(wires=0)

        qml.PauliRot(W[1,0],'X',  wires=0)
        qml.PauliRot(W[1,1],'X',  wires=1)
        qml.PauliRot(W[1,2],'X',  wires=2)
        qml.Hadamard(wires=2)

        qml.PauliRot(W[1,0],'Y',  wires=1)
        qml.PauliRot(W[1,1],'Y',  wires=2)
        qml.PauliRot(W[1,2],'Y',  wires=3)

        qml.Hadamard(wires=1)

        qml.broadcast(qml.CY, wires=[0, 1, 2, 3], pattern="ring")
```

Below are the results I got after using the above layer:



I also added a CY gate (controlled Y rotation) at the end with a broadcast in a ring pattern. The addition of Hadamard gates instead of CNOT gave better results and the model gave a validation accuracy of 100% within 14 epochs only. This is because MultiRZ and PauliRot all give different kinds of rotations to the qubits. Here I have made sure to have rotations w.r.t to all the axes in different parts of the layer. So here the variation is not just on the axis but also on the kind of rotation hence this makes the gradient reach a local minima even faster as could be clearly seen in the graph.

Below are the URLs of the public repository where I have uploaded the and code

Code:
I have uploaded the code to the below public repository. Ran this in google Colab.

Code for amplitude encoding:
https://github.com/sunayana17/CSE598IDL/blob/master/Assignment4/AmplitudeEmbedding.ipynb

Code for RX embedding:
https://github.com/sunayana17/CSE598IDL/blob/master/Assignment4/RXEmbedding.ipynb

Code for RY embedding:
https://github.com/sunayana17/CSE598IDL/blob/master/Assignment4/RYEmbedding.ipynb

Code for RZ embedding:
https://github.com/sunayana17/CSE598IDL/blob/master/Assignment4/RZEmbedding.ipynb

Code for various ansatz variations:
https://github.com/sunayana17/CSE598IDL/blob/master/Assignment4/AnzatzVariation.ipynb

Below is the URL from where I got the Iris dataset:
https://scikit-learn.org/stable/auto_examples/datasets/plot_iris_dataset.html