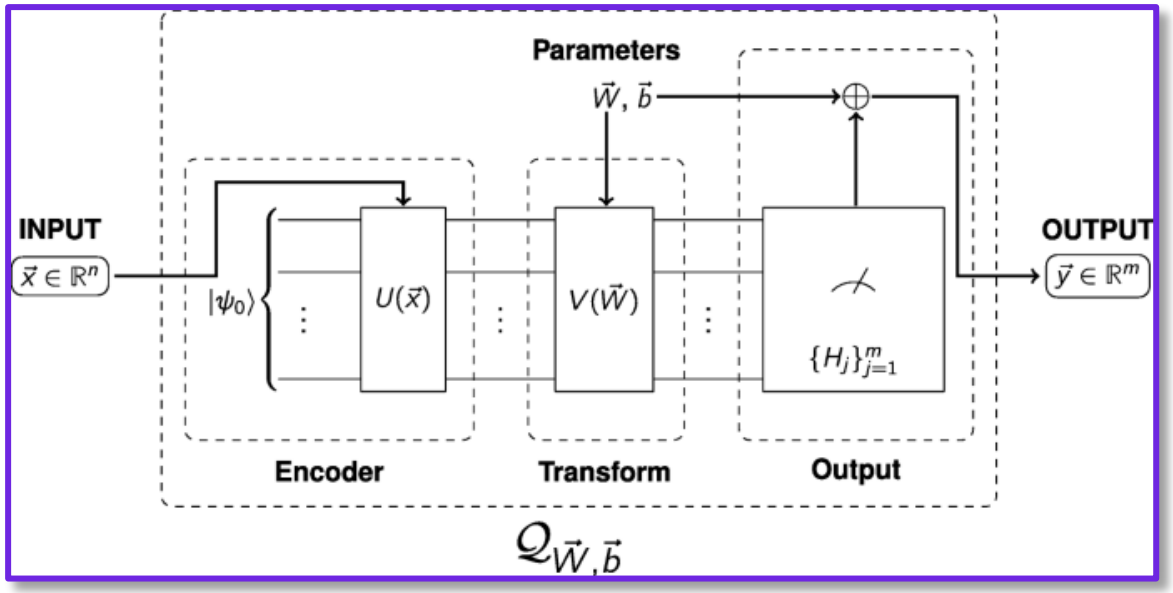
QNN Example

QNN Example

Recap

In previous presentations, we studied the fundamentals of quantum computing and quantum neural networks (QNNs). We learned what qubits are and how different gates manipulate these qubits. We also explored how to convert traditional data into qubits using angle encoding.
Additionally, we designed a quantum neural network with two main components: the encoding part and the variational part. We examined how to implement angle encoding in the encoding section of the QNN. Then, we discussed how to design the variational section using CNOT gates and rotation gates. Finally, we reviewed how to measure the output and update the parameters of the network based on the calculated loss.

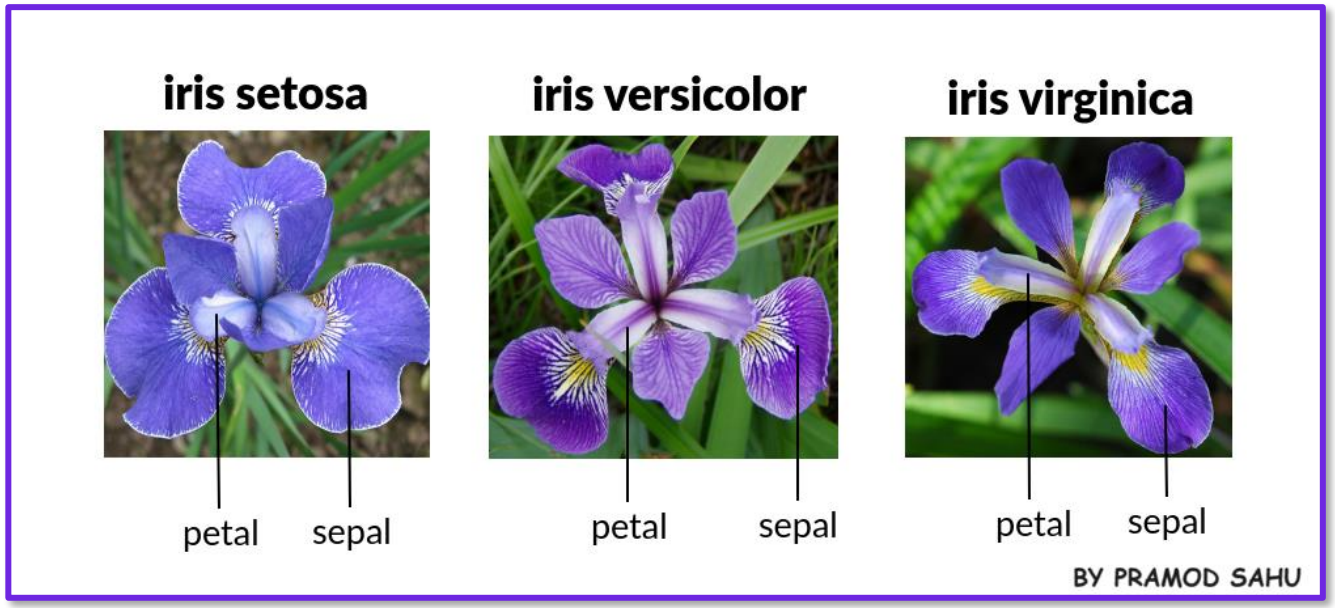


QNN Example

Problem Description

In this presentation, we will solve the Iris classification problem, which was introduced by the British biologist and statistician Ronald Fisher in 1936 as a benchmark for testing various statistical classification techniques. The Iris dataset consists of samples from three different species of the Iris flower, with four physical features measured for each sample.

This dataset is considered straightforward because it is structured in a way that allows for clear separation of the species based on the measured features. This makes it an ideal choice for our focus on building a quantum neural network (QNN) without the complications that may arise from more complex datasets.



Import the Modules

First, we need to import the modules that we will use throughout this example. We will utilize the Sklearn library to import our dataset. NumPy will be used to reshape our inputs and outputs to the desired format. The copy module will help us create deep copies of some of our lists, while Matplotlib will be used to plot the loss curve.
Additionally, we will introduce a new module called Qiskit. This module provides tools for creating and manipulating quantum programs, allowing us to run them on prototype quantum devices within the IBM Quantum Platform or on local simulators. We will use Qiskit to build, visualize, and test our quantum neural network (QNN).

```
from sklearn import model_selection, datasets
import numpy as np
import copy
import matplotlib.pyplot as plt
from qiskit import *

from qiskit.visualization import circuit_drawer
from qiskit_aer import Aer
```

Prepare the Dataset

Before designing and defining the model, we need to import our dataset. The Iris dataset is quite simple and consists of only four features, making it relatively small.
First, we load the dataset. Then, we determine which samples we will use throughout our program. We will use the first 100 samples, which means we will only include two classes of the Iris dataset, as there are 50 samples per class and they are arranged in order. Finally, we split our dataset into training and testing sets. We will use 20 percent of the dataset for testing and the remaining 80 percent for training.

```
iris=datasets.load_iris()
X=iris.data[0:100]
Y=iris.target[0:100]
X_train, X_test, Y_train, Y_test=
    model_selection.train_test_split(X,Y,test_size=0.20, random_state=42)
print(Y_train)
```

QNN Example

Encoding function

The first step in building the quantum neural network is to create the encoding section. We will define a function that takes one sample and returns the corresponding qubits and classical bits for that sample.
In our example, we have four features and two classes, so we define four quantum bits and one classical bit to store our output. Next, we create a quantum circuit that includes the qubits and the classical bit we defined. We then apply a rotation gate around the x-axis on each qubit. The magnitude of this rotation is determined by the value of the corresponding feature in the sample. Finally, we return the rotated qubits and the classical bit.

```
def feature_map(X):
 q=QuantumRegister(N)
 c=ClassicalRegister(1)
 qc=QuantumCircuit(q,c)
 for i, x in enumerate(X):
  qc.rx(x,i)
 return qc, c
```

QNN Example

Build the Variational Part

The next step in building the model is to define the variational part. We will create a function to add this section to our circuit. This function will accept the rotated qubits from the previous section and a list that specifies the values of the parameters.
The first set of gates we will add consists of the CNOT gates. We will place a CNOT gate between each pair of consecutive qubits, which will introduce non-linearity to our network. Next, we will add our rotation gates, this time rotating the qubits around the y-axis. The magnitude of the rotation is determined by the parameters passed to the function. Finally, we return the qubits after all the transformations.

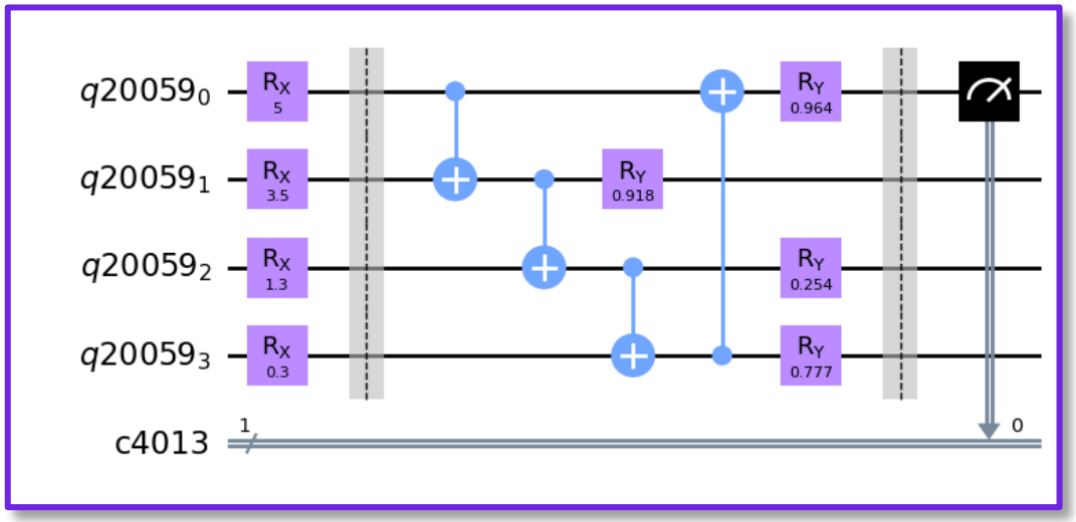
```
def variational_circuit (qc,theta):
  for i in range(N-1):
    qc.cnot(i,i+1)

  qc.cnot(N-1,0)
  for i in range(N):
      qc.ry(theta[i],i)
  return qc
```

QNN Example

Define the Network

Now that we have both the encoding and variational parts ready, we can define a function to build and apply the quantum neural network (QNN). First, we encode the sample using the previously defined function. Then, we pass the rotated qubits to the function responsible for adding the variational circuit in order to obtain the final states of the qubits.

Next, we need to produce the output. First, we will measure the first qubit to obtain the output. Then, we will specify the number of times we want to repeat the experiment. After that, we will define the simulator we want to use. Finally, we will generate a histogram of the results and return the probability that the measured qubit is in the state $|1\rangle$ as the output.




```
def quantum_nn(X, theta, simulator =True):
 qc,c=feature_map(X)
 qc=variational_circuit(qc,theta)
 qc.measure(0,c)

 shots=1E5
 backend=Aer.get_backend('qasm_simulator')
 job=execute(qc,backend, shots=shots)
 result=job.result()
 counts=result.get_counts(qc)
 return counts['1']/shots
```

QNN Example

Calculate the Loss and Gradient

For this example, we will use a simple loss function. We will calculate the loss by subtracting the actual label from the predicted label and then squaring the result. This calculated loss will be printed to closely monitor the training process.
We will compute the gradient using a method that differs slightly from that used in classical neural networks. For each parameter, we will add a small value to that parameter. Next, we will calculate the network's output and the corresponding loss. We will then subtract the current loss from the previous loss and divide this difference by the small value to determine the derivative of the loss function with respect to that parameter. We will repeat this process for all parameters to form the final gradient.

```
def loss(prediction, target):
    return(prediction-target)**2
```

```
def gradient(X,Y, theta):
  delta=0.01
  grad=[]
  for i in range(len(theta)):
    dtheta=copy.copy(theta)
    dtheta[i] +=delta
    pred1=quantum_nn(X, dtheta)
    pred2=quantum_nn(X, theta)
    grad.append((loss(pred1, Y)-loss(pred2, Y))/delta)
    return np.array(grad)
```

Calculate Accuracy

The metric we will monitor during training and evaluation is accuracy. We have to define a function that calculates the accuracy of our model on a given dataset and parameters.
This function will calculate the output of the model for each input. Then, it will interpret the output as 1 if the probability is greater than 0.5 and 0 otherwise. Next, it will compare the predicted label to the actual label and keep track of the number of correct classifications. In the end, it will return the accuracy of the model.

```
def accuracy (X,Y, theta):

    counter=0
    for X_i, Y_i in zip(X, Y):
        prediction=quantum_nn(X_i, theta)

        if prediction<0.5 and Y_i==0:
            counter+=1
        elif prediction>=0.5 and Y_i==1:
            counter+=1
    return counter/len(Y)
```

QNN Example

Training Process

First, we define the learning rate and initialize the parameters to one. We also create a list to keep track of the loss during training.
Next, we iterate through the dataset for 18 epochs. In each epoch, we define a temporary list to calculate the loss of the model for each sample. We then iterate through the samples to compute the output and the loss for each sample. After that, we calculate the gradient and update the parameters. At the end of each epoch, we append the mean of the temporary loss list to the loss list defined at the beginning, and we calculate the accuracy of the model, printing this information. This process will help us monitor the training progress more effectively.

```
eta=0.03
loss_list=[]
theta= np.ones(N)

print('Epoch\t Loss\t Training Accuracy')

for i in range(18):
  loss_tmp=[]
  for X_i, Y_i in zip( X_train, Y_train):
    prediction=quantum_nn(X_i, theta)
    loss_tmp.append(loss(prediction,Y_i))
    theta=theta-eta*gradient(X_i, Y_i, theta)

  loss_list.append(np.mean(loss_tmp))
  acc=accuracy(X_train, Y_train, theta)
  print(f'{i} \t {loss_list[-1]:.3f} \t {acc:.3f}')
```

QNN Example

Evaluation

If we visualize the loss list created during the training process, we can observe that the loss decreased over time, indicating that the model is learning effectively from the dataset. However, the model has not yet converged after 18 epochs, suggesting that further training could lead to improved results.

Once the training is complete, we can evaluate our model using the previously defined function to calculate its accuracy. We will compute and print the accuracy of our model on the testing dataset. The model achieves a high accuracy of 80 percent, which is impressive considering that we only trained it for 18 epochs.

```
accuracy(X_test,Y_test,theta)
```

```
<ipython-input-59-78d5960a0ade>:3:
  qc.cnot(i,i+1)
<ipython-input-59-78d5960a0ade>:5:
  qc.cnot(N-1,0)
0.8
```