Experiment No. 8

Title: Dark Web Marketplace Image Classification using Quantum Convolutional Neural Network

Batch: B4 **Roll No. 16010420117** **Experiment No.:8**

Title: Dark Web Marketplace Image Classification using Quantum Convolutional Neural Network

---

Describe the following points with respect to the business under consideration,

---

1. Problem faced by the business

The system is designed such that it helps to estimate the price of a Car based upon the different features present, so it helps the business to give the correct amount of pricing to their Car Models, so that the customers are attracted and also are satisfied with price range assigned based upon it's features. So, assigning Price range accordingly is challenging without the use of ML algorithms.

2. Approach/ Methodology followed by the business

The methodology which we followed was that we took the dataset of Automobile Price Data. Then we split the dataset into train and test. Then we used Linear Regression and trained the machine learning model with that dataset and in the end we tested that model.

3. Skillsets , infrastructure and other impact on the business during

implementation Skillset: Cloud computing
Infrastructure: Microsoft Azure Cloud services
There were no as such impacts on the business, as any of the business service lines were not being used.

4. Similar approaches followed by other businesses

Step 1. Sign-in using Microsoft account on studio.azureml.net
Step 2. Creating workspace for our Machine Learning project.
Step 3. Select New option on bottom right:
Step 4. Click on Blank experiment and write name and summary of experiment
Step 5. Select From Saved Datasets-> Samples-> dataset of your choice
Step 6. Now, search 'Select columns in dataset' from items and drag it
Step 7. Now, click on launch column selector-> with rules->exclude column normalized-losses as that column contains many rows/records with empty values.
Step 8. Search and select 'Clean Missing Data' from items list
Step 9. Now, select cleaning mode -> Remove entire row as it will remove the entire row wherever missing value is found
Step 10. Again choose 'select columns in dataset'
Step 11. Now, launch column selector and include all the columns based on which prediction is to be done: make, body-style, wheel-base, engine-size, horsepower, peak-rpm, highway-mpg, price

Step 12. Now, select ‘split data’ from list and drag it
Step 13. For Split data, enter the fraction of data which is needed for training while rest will be used for testing
Step 14. Now, Select ‘Linear Regression’ as the algorithm to be used and ‘Train Model’ from list
Step 15. For training model, click on launch column selector, include price column as Price is what is to be predicted
Step 16. Add Score Model from list drag it and make connections
Step 17. Now, Add Evaluate Model from list and make connections
Step 18. Now, Click on Run
Step 19. To check prediction results, right click on Score Model, select visualize
Step 20. To check Evaluation results, right click on Evaluation Model, select visualize

1. **Problems faced:**

   Researchers have investigated the dark web for various purposes and with various approaches. Most of the dark web data investigation focused on analyzing text collected from HTML pages of websites hosted on the dark web. In addition, researchers have documented work on dark web image data analysis for a specific domain, such as identifying and analyzing Child Sexual Abusive Material (CSAM) on the dark web. However, image data from dark web marketplace postings and forums could also be helpful in forensic analysis of the dark web investigation. The presented work attempts to conduct image classification on classes other than CSAM. Nevertheless, manually scanning thousands of websites from the dark web for visual evidence of criminal activity is time and resource intensive. Therefore, the proposed work presented the use of quantum computing to classify the images using a Quantum Convolutional Neural Network (QCNN). Authors classified dark web images into four categories alcohol, drugs, devices, and cards. The provided dataset used for work discussed in the paper consists of around 1242 images. The image dataset combines an open source dataset and data collected by authors. The paper discussed the implementation of QCNN and offered related performance measures.

Approach/Methodology:

There has been a significant interest in quantum machine learning (QML) by researchers. This is because QML offers the ability to more effectively address the issues associated with massive data and the slow training process in existing conventional machine learning. Quantum Convolution Neural Networks (QCNNs) handle quantum data to recognize stages of quantum states and create a quantum error-correcting system. Quantum gates with controllable parameters approximate the pooling and convolutional layers. The authors present a quantum convolutional neural network based on parameterized quantum circuits in the proposed work. To be processed by quantum technology, image data must be encoded into quantum states. The authors use the QCNN model to handle grid-type input, such as photographs. A parametric rule is used in the proposed model to calculate the analytical gradients of loss functions on quantum circuits and to achieve faster yet more stable convergence.

Skillsets , infrastructure and other impact on the business

Implementation Skillset: Quantum Machine Learning
Infrastructure: Google Colab

Related Work Done by Researchers:

Researchers attempted image analysis with Compass Radius Estimation for Image Classification (CREIC) on the dark web data. The work is one of the few attempts to categorize dark web images into five categories. In other work approaches of perceptual hashing are discussed for dark web image classification. However, the proposed work aims to employ the advantage of quantum computing to classify dark web image data into four categories.

Numerous computer science research has been affected by the idea of quantum computation, particularly those in the fields of computational modeling, cryptography theory, and information theory. Information security may benefit from quantum computers, or it may suffer a detrimental effect. Many researchers have thoroughly examined the advantages of quantum computing in cybersecurity. The use of a quantum computer can potentially be advantageous in several domains.

Research on the application of many quantum concepts was given, such as Intruder detection systems for healthcare systems may be trained using mechanics and neural networks. The suggested method is tested on the KDD99 dataset.

Researchers investigated how quantum computing may reduce the need for domain-specific security. In it provided that quantum computing-based representations of standard AES and modified AES algorithms to underline that quantum computing will be a viable solution to improve cybersecurity. Quantum cryptography to encrypt communication between sensors and computers to protect cyber-physical systems.A look at the viability of fusing quantum and conventional computers.

A unique hybrid quantum-classical deep learning model for botnet detection using domain generation algorithms (DGA).

Researchers are also examining if applying quantum mechanical concepts to machine learning issues might enhance the outcome. The most recent research results in quantum machine learning were compiled under. A classification with Quantum Convolutional Neural Network strategy was suggested by the authors.

About Quantum Convolutional Neural Network

1. *Neurons and Weights*

A neural network is a complex function constructed from smaller building components known as neurons. A neuron is often a nonlinear function that translates one or more inputs to a single real number. It is also typically simple, straightforward to compute, and nonlinear. Usually, neurons copy their single output and provide it to other neurons as input. In order to visually depict how the output of one neuron will be utilized as the input to other neurons, we represent neurons as nodes in a graph and draw directed edges between nodes. Also noteworthy is that each edge in our graph frequently has a scalar number called a weight attached to it. According to this theory, each input to a neuron will be multiplied by a separate scalar before being gathered and processed into a single result. In order to train a neural network, the primary goal of the proposed work is to select weights that will cause the network to act in a specific manner.

2. Input Output Structure of neural network

A traditional (real-valued) vector serves as the input to a neural network. According to the network's graph topology, a layer of neurons receives each input vector component multiplied by a distinct weight. Then, the findings are compiled into a new vector, where the i'th component stores the output of the i'th neuron after each neuron in the layer has been assessed. After that, a new layer can use this new vector as an input, and so on. Except for the proposed network's initial and last levels, all other layers are hidden.

3. Feed Forward Neural Network

A feed-forward neural network is a name given to the type of neural network we will be working with (FFNN). This means that information will never hit a cell again as it passes through our brain network [18]. We may call the graph representing our neural network a directed acyclic graph (DAG). Furthermore, no edges will be allowed between neurons in the same neural network layer.

4. Backend

The backend acts as either a simulator or an actual quantum computer, operating quantum circuits and/or pulse schedules and providing results.

5. Shots

1. A single trip through each step of an entire quantum circuit on an IonQ(Trapped ion Quantum Computing), Rigetti, or OQC (Outgoing Quality Control) gate-based QPU(Quantum Processor) is called a "shot."

A Mathematical Approach to Quantum Convolutional Layer

Let X^l be the input and K^l be the Kernel for the layer l of a convolutional neural network,

And f: R $\rightarrow$[0, C] with C > 0 be a non-linear function so that $f(X^{l+1}) := f(X^l * K^l)$ is the output for layer l. The given X^l and K^l are stored in Quantum Random Access Memory (QRAM) ; there is a quantum algorithm that, for precision parameters

$\varepsilon > 0$ and $\eta > 0$, creates a quantum state $|f(\bar{X}^{l+1})\rangle$ such that $||f(\bar{X}^{l+1}) - f(X^{l+1})||_\infty \leq 2\varepsilon$ and retrieves classical tensor χ^{l+1} such that for each pixel j. [11, 17]

$$\begin{cases} \left|\chi_j^{l+1} - f\left(\chi_j^{l+1}\right)\right| \leq 2\varepsilon & if\ f\left(\chi_j^{l+1}\right) \geq \eta \\ \chi_j^{l+1} = 0 & if\ f\left(\chi_j^{l+1}\right) < \eta \end{cases}$$

The algorithm has time complexity as

$$O\left(\frac{1}{\varepsilon\eta^2} \cdot \frac{M\sqrt{C}}{\sqrt{E\left(f(\bar{X}^{l+1})\right)}}\right)$$

O hides the poly-logarithmic in the size of X^l and K^l.

Algorithms Used

1. Forward Pass for QCNN

The quantum analog of a single quantum convolutional layer is implemented in the QCNN forward pass method. To prepare the input for the following layer, it first applies a convolutional function to an input and a kernel, then applies a nonlinear function and performs pooling operations. [11]

2. Quantum Backpropagation Algorithm (Backward Pass)

A widely used algorithm to train feed-forward neural networks is backpropagation. The algorithm required for a quantum convolutional neural network is a quantum backpropagation algorithm. In classical feed-forward neural networks, the classical backpropagation algorithm updates all kernel weights according to the derivative of a given loss function *L*.

The algorithm calculates each element of the gradient tensor $\frac{\partial L}{\partial F^l}$ within additive error $\delta \| \frac{\partial L}{\partial F^l} \|$, which updates F^l as per the gradient descent update rule.

The time complexity of a single layer *l* for quantum backpropagation is:

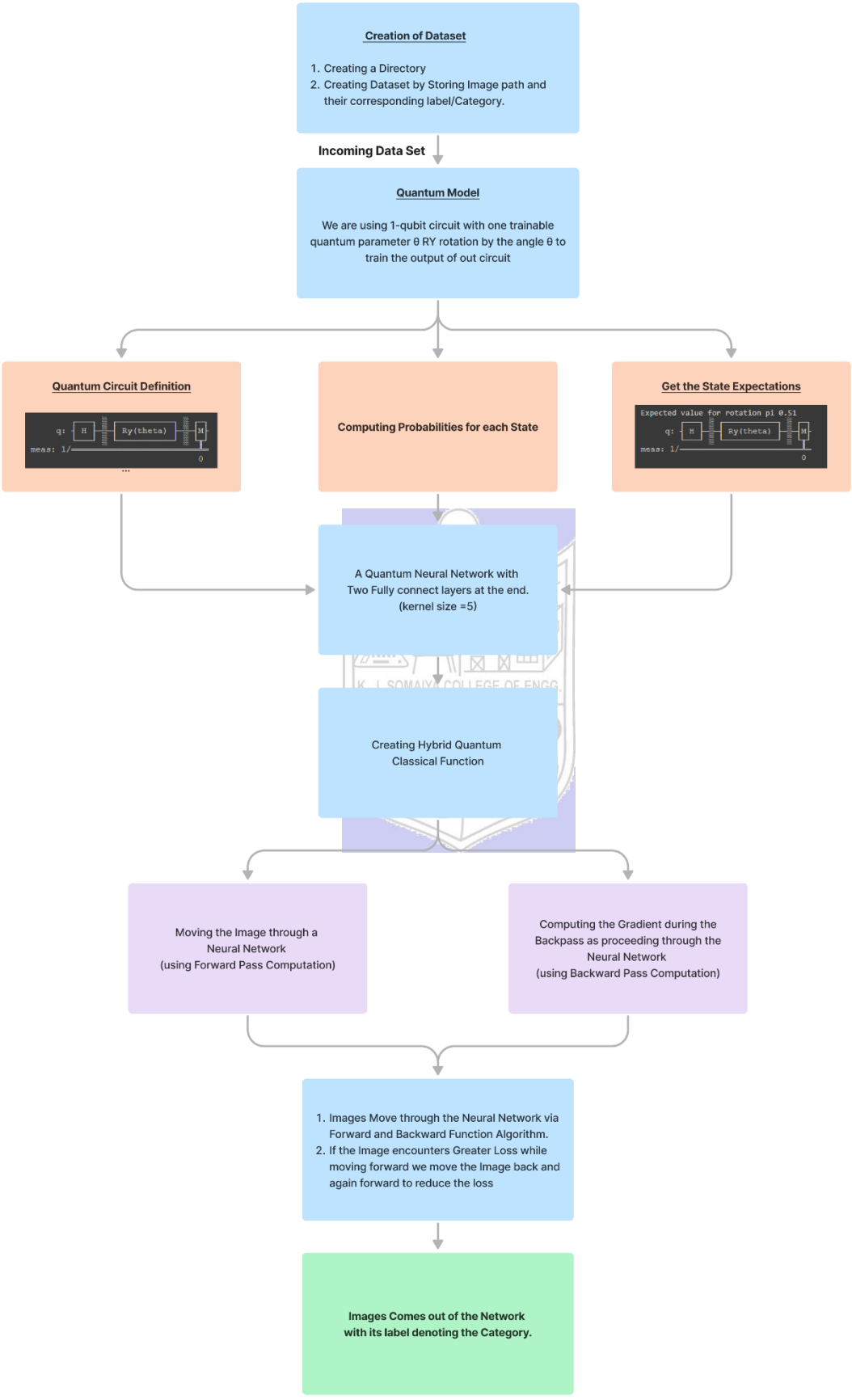
$$O\left(\left((\mu(A^l) + \mu(\frac{\partial L}{\partial Y^{l+1}})\right)\kappa\left(\frac{\partial L}{\partial F^l}\right) + (\mu\left(\frac{\partial L}{\partial Y^{l+1}}\right) + \mu(F^l))\kappa(\frac{\partial L}{\partial Y^l})\right)\frac{\log 1/\delta}{\delta^2}$$

The gradient is calculated as follows:

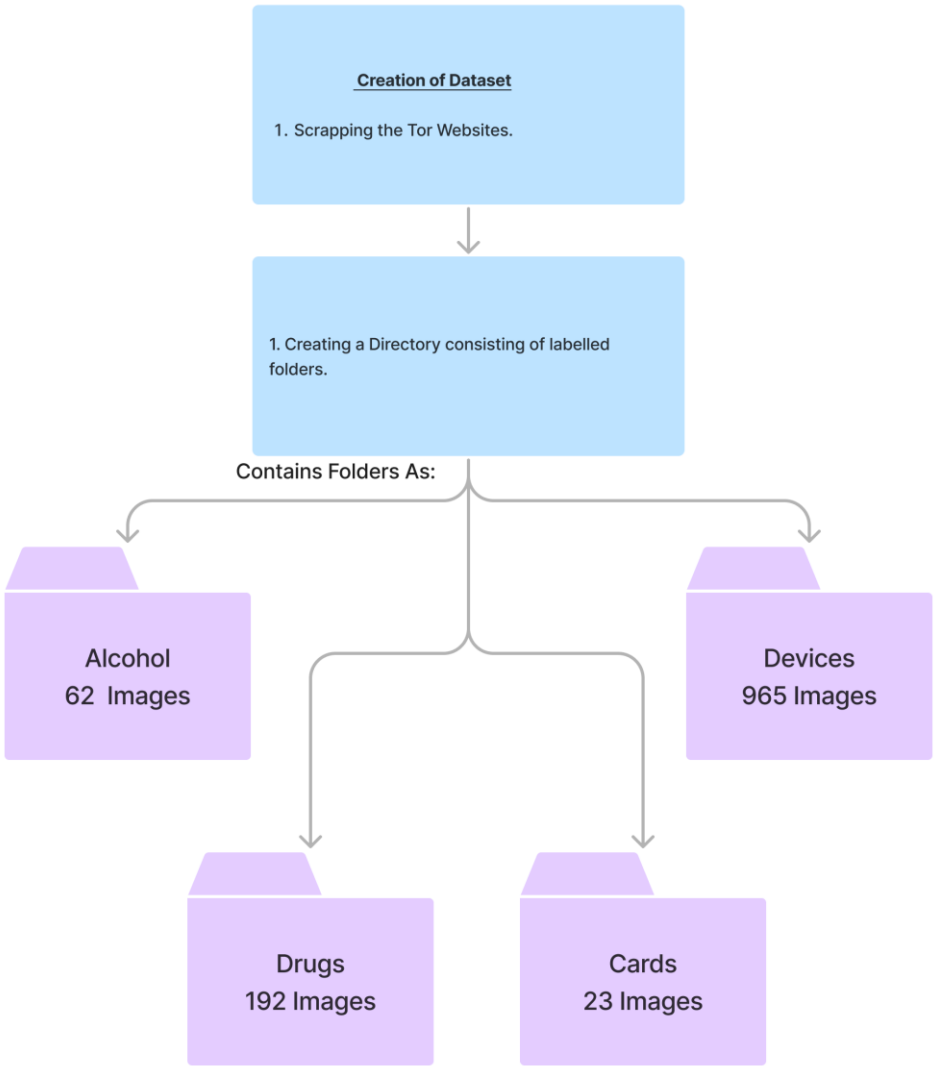
$$\Delta_{x_0} QuantumCircuit(x_0) = QuantumCircuit(\alpha + \beta) - QuantumCircuit(\alpha - \beta)$$

Here x_0 and α represented as a parameter for Quantum Circuit and β a macroscopic shift. Thus the gradient is simply the difference the Quantum Circuit evaluated at $\alpha + \beta$ And $\alpha - \beta$.

Flow of Quantum Classification Model



Flow of Dataset



Code for the above approach/ methodology

```
from pydrive.auth import GoogleAuth
from pydrive.drive import GoogleDrive
from google.colab import auth
from oauth2client.client import GoogleCredentials
auth.authenticate_user()
gauth = GoogleAuth()
gauth.credentials = GoogleCredentials.get_application_default()
drive = GoogleDrive(gauth)
```

```
! ls
! mkdir dataset
! unzip train_set.zip -d dataset
```

```
sample_data  train_set.zip
Archive:  train_set.zip
warning [train_set.zip]:  1048576 extra bytes at beginning or within zipfile
  (attempting to process anyway)
file #1:  bad zipfile offset (local header sig):  1048576
  (attempting to re-compensate)
  inflating: dataset/Alcohol/0005D4E7C1UJ0F8EmEDD78F3F2A.jpg
```

```
[ ] import seaborn as sns
    import cv2
    import os
    import numpy as np
    import matplotlib.pyplot as plt
    from sklearn.metrics import classification_report,confusion_matrix
```

```
#creating dataset by storing image path and their corresponding label
categories = os.listdir('/content/dataset/')

img_size= 224

def data_gen(data_dir):
    data=[]
    for category in categories:
        path= os.path.join(data_dir,category)
        class_num= categories.index(category)
        print(path)
        for img in os.listdir(path):
            try:
                img_arr= cv2.imread(os.path.join(path,img))
                resized_arr= cv2.resize(img_arr,(img_size,img_size))
                data.append([os.path.join(path,img), class_num])
            except Exception as e:
                print(e)
    return np.array(data)
```

```
[ ] temp = data_gen('/content/dataset/')#training dataset
    val_data= data_gen('/content/dataset/')
```


```
[ ] temp = data_gen('/content/dataset/')#training dataset
    val_data= data_gen('/content/dataset/')
```


```
/content/dataset/Device
/content/dataset/Alcohol
/content/dataset/Card
/content/dataset/Drugs
/content/dataset/Device
/content/dataset/Alcohol
/content/dataset/Card
/content/dataset/Drugs
```

```
%prun
X=[]
y=[]
lenofimage = len(temp)
for categories, label in temp:
    X.append(categories)
    y.append(label)
# X= np.array(X).reshape(lenofimage,-1)
for i in range(len(y)):
  if(y[i]!= '1'):
    y[i] = 0
  else:
    y[i]=1
temp1=list(zip(X,y))
#converting numpy array into dictionary
data_dict = {}
for elem in temp1:
    try:
        data_dict[elem[0]].append(elem[1])
    except KeyError:
        data_dict[elem[0]] = [elem[1]]
print(data_dict)
```

```
{'/content/dataset/Device/8C5C23E4WFB95DD2CF9945CE59ED494.jpg': [0], '/content/dataset/Device/qp8323CC6F7BDB89ZJqD1sep5D.png': [0],
```

```
#shuffling data
import random
l = list(data_dict.items())
random.shuffle(l)
data_dict = dict(l)
```

```
import numpy as np
import matplotlib.pyplot as plt

import torch
from torch.autograd import Function
from torchvision import datasets, transforms
import torch.optim as optim
import torch.nn as nn
import torch.nn.functional as F
```

```
! pip install qiskit
import qiskit
from qiskit import transpile, assemble
from qiskit.visualization import *
```

```
!pip install memory_profiler
```

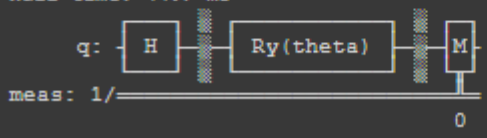
```
Looking in indexes: https://pypi.org/simple, https://us-python.pkg.dev/colab-wheels/public/simple/
Collecting memory_profiler
  Downloading memory_profiler-0.60.0.tar.gz (38 kB)
Requirement already satisfied: psutil in /usr/local/lib/python3.7/dist-packages (from memory_profiler) (5.4.8)
Building wheels for collected packages: memory-profiler
  Building wheel for memory-profiler (setup.py) ... done
  Created wheel for memory-profiler: filename=memory_profiler-0.60.0-py3-none-any.whl size=31284 sha256=0803e657bed3a3b9cb4aa16f4a465299e51f5f8ecaeb08cf70c5c9d1dbad19b3
  Stored in directory: /root/.cache/pip/wheels/67/2b/fb/326e30d638c538e69a5eb0aa47f4223d979f502bbdb403950f
Successfully built memory-profiler
Installing collected packages: memory-profiler
Successfully installed memory-profiler-0.60.0
```

```
#we're using 1-
qubit circuit with one trainable quantum parameter  θ RY rotation by the angle θ to train the output of our circuit
# %lprun
%prun
%load_ext memory_profiler

class QuantumCircuit:

  def __init__(self, n_qubits, backend, shots):
    # --- Circuit definition ---
    self._circuit = qiskit.QuantumCircuit(n_qubits)

    all_qubits = [i for i in range(n_qubits)]
    self.theta = qiskit.circuit.Parameter('theta')

    self._circuit.h(all_qubits)
    self._circuit.barrier()
    self._circuit.ry(self.theta, all_qubits)

    self._circuit.measure_all()
    # ---------------------------

    self.backend = backend
    self.shots = shots

  def run(self, thetas):
    t_qc = transpile(self._circuit,
                     self.backend)
    qobj = assemble(t_qc,
                    shots=self.shots,
                    parameter_binds = [{self.theta: theta} for theta in thetas])
    job = self.backend.run(qobj)
    result = job.result().get_counts()

    counts = np.array(list(result.values()))
    states = np.array(list(result.keys())).astype(float)
```

```
    # Compute probabilities for each state
    probabilities = counts / self.shots
    # Get state expectation
    expectation = np.sum(states * probabilities)

    return np.array([expectation])
```

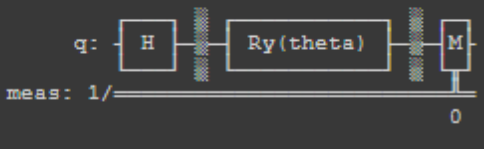
```
%%time
simulator = qiskit.Aer.get_backend('aer_simulator')

circuit = QuantumCircuit(1, simulator, 100)
circuit._circuit.draw()
```

```
CPU times: user 37.2 ms, sys: 5.72 ms, total: 42.9 ms
Wall time: 44.7 ms
```




```
%%time
%prun
%load_ext memory_profiler
simulator = qiskit.Aer.get_backend('aer_simulator')

circuit = QuantumCircuit(1, simulator, 100)
print('Expected value for rotation pi {}'.format(circuit.run([np.pi])[0]))
circuit._circuit.draw()
```

```
 The memory_profiler extension is already loaded. To reload it, use:
  %reload_ext memory_profiler
Expected value for rotation pi 0.38
CPU times: user 14.8 ms, sys: 0 ns, total: 14.8 ms
Wall time: 14.1 ms
```



```
%prun
%load_ext memory_profiler
class HybridFunction(Function):
    """ Hybrid quantum - classical function definition """

    @staticmethod
    def forward(ctx, input, quantum_circuit, shift):
        """ Forward pass computation """
        ctx.shift = shift
        ctx.quantum_circuit = quantum_circuit

        expectation_z = ctx.quantum_circuit.run(input[0].tolist())
        result = torch.tensor([expectation_z])
        ctx.save_for_backward(input, result)

        return result

    @staticmethod
    def backward(ctx, grad_output):#compute gradient directly during backpass
        """ Backward pass computation """
        input, expectation_z = ctx.saved_tensors
        input_list = np.array(input.tolist())

        shift_right = input_list + np.ones(input_list.shape) * ctx.shift
        shift_left = input_list - np.ones(input_list.shape) * ctx.shift

        gradients = []
        for i in range(len(input_list)):
            expectation_right = ctx.quantum_circuit.run(shift_right[i])
            expectation_left  = ctx.quantum_circuit.run(shift_left[i])

            gradient = torch.tensor([expectation_right]) - torch.tensor([expectation_left])
            gradients.append(gradient)
        gradients = np.array([gradients]).T
        return torch.tensor([gradients]).float() * grad_output.float(), None, None

class Hybrid(nn.Module):
    """ Hybrid quantum - classical layer definition """

    def __init__(self, backend, shots, shift):
        super(Hybrid, self).__init__()
        self.quantum_circuit = QuantumCircuit(1, backend, shots)
        self.shift = shift
```

```
  def forward(self, input):
    return HybridFunction.apply(input, self.quantum_circuit, self.shift)
```

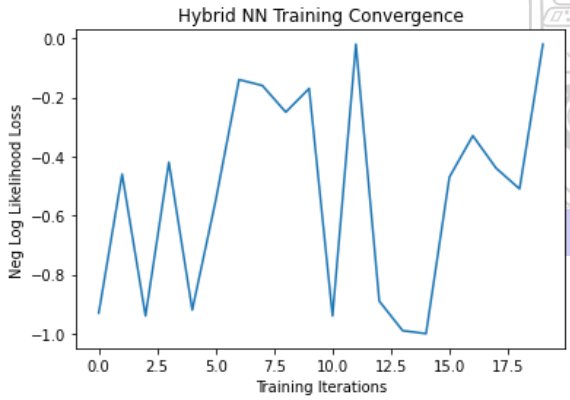
```
#normal CNN with two fully connected layers at the end
%prun
class Net(nn.Module):
    def __init__(self):
        super(Net, self).__init__()
        self.conv1 = nn.Conv2d(1, 6, kernel_size=5)
        self.conv2 = nn.Conv2d(6, 16, kernel_size=5)
        self.dropout = nn.Dropout2d()
        self.fc1 = nn.Linear(256, 64)
        self.fc2 = nn.Linear(44944, 1)
        self.hybrid = Hybrid(qiskit.Aer.get_backend('aer_simulator'), 100, np.pi / 2)

    def forward(self, x):
        x = F.relu(self.conv1(x))
        x = F.max_pool2d(x, 2)
        x = F.relu(self.conv2(x))
        x = F.max_pool2d(x, 2)
        x = self.dropout(x)
        x = x.view(1, -1)
        # x = F.relu(self.fc1(x))
        x = self.fc2(x)
        x = self.hybrid(x)
        return torch.cat((x, 1 - x), -1)
```

```
#adijusting hyperparameters and training data

%prun
%load_ext memory_profiler

model = Net()
optimizer = optim.Adam(model.parameters(), lr=0.00001)
loss_func = nn.NLLLoss()

epochs = 20
loss_list = []

model.train()
for epoch in range(epochs):
  total_loss = []
  optimizer.zero_grad()
  for img_path,target_x in data_dict.items():
    # Forward pass
    img_arr= cv2.imread(img_path)
    resized_arr= cv2.resize(img_arr,(img_size,img_size))
    resized_arr= cv2.cvtColor(resized_arr, cv2.COLOR_BGR2GRAY)
    data=torch.from_numpy(resized_arr)
    data=torch.unsqueeze(data,0)
```

```
    data=torch.unsqueeze(data,0)
    data=data.type(torch.float32)
    output = model(data)
    target=torch.ones(1,dtype=torch.long)
    # Calculating loss
    loss = loss_func(output, target)
    # Backward pass
    loss.backward(retain_graph=True)
# Optimize the weights
optimizer.step()
total_loss.append(loss.item())
loss_list.append(sum(total_loss)/len(total_loss))
print('Training [{:.0f}%]\tLoss: {:.4f}'.format(
    100. * (epoch + 1) / epochs, loss_list[-1]))
```

```
import matplotlib.pyplot as plt

plt.plot(loss_list)
plt.show()
#the result of loss function seems to be fluctuating, however at the end it decreases than initial loss.
```

```
plt.plot(loss_list)
plt.title('Hybrid NN Training Convergence')
plt.xlabel('Training Iterations')
plt.ylabel('Neg Log Likelihood Loss')
```



```
model.eval()
with torch.no_grad():

    correct = 0
    for img_path,target_x in data_dict.items():
        output = model(data)

        pred = output.argmax(dim=1, keepdim=True)
        target = torch.tensor(target_x)
        # target =
        correct += pred.eq(target.view_as(pred)).sum().item()

        loss = loss_func(output, target)
        total_loss.append(loss.item())

    print('Performance on test data:\n\tLoss: {:.4f}\n\tAccuracy: {:.1f}%'.format(
        sum(total_loss) / len(total_loss),
        correct / len(data_dict) * 100)
        )
```

Questions:

- Differentiate between linear and nonlinear regression and write a note on converting non-linear model into linear model

Ans: Nonlinear regression modeling is similar to linear regression modeling in that both seek to track a particular response from a set of variables graphically. Nonlinear models are more complicated than linear models to develop because the function is created through a series of approximations (iterations) that may stem from trial-and-error. Mathematicians use several established methods, such as the Gauss-Newton method and the Levenberg-Marquardt method.

Often, regression models that appear nonlinear upon first glance are actually linear. The curve estimation procedure can be used to identify the nature of the functional relationships at play in your data, so you can choose the correct regression model, whether linear or nonlinear. Linear regression models, while they typically form a straight line, can also form curves, depending on the form of the linear regression equation. Likewise, it's possible to use algebra to transform a nonlinear equation so that it mimics a linear equation—such a nonlinear equation is referred to as "intrinsically linear."

Outcomes: CO Apply concepts of learning and neural network

Explain how to convert nonlinear regression to linear regression.

Ans) Linear regression always uses a linear equation, Y = a +bx, where x is the explanatory variable and Y is the dependent variable. In multiple linear regression, multiple equations are added together but the parameters are still linear.
If the model equation does not follow the Y = a +bx form then the relationship between the dependent and independent variables will not be linear. There are many different forms of non-linear models. A random forest regression is considered a non-linear model. Random forest models are ensemble learning methods for regression which grow a forest of regression trees and then average the outcomes. This cannot be expressed as an equation

Conclusion: (Conclusion to be based on the objectives and outcomes achieved)

I was able to perform an experiment on Azure ML Studio using Machine Learning algorithm(linear regression).

Google Colab Notebook Link:
https://colab.research.google.com/drive/1rhXk7pKg6qHDWBGuTSDzItSb5TtHN9E4?usp=sharing

Grade: AA / AB / BB / BC / CC / CD /DD

Signature of faculty in-charge with date

References:

Books/ Journals/ Websites: