Hybrid quantum-classical Neural Networks with PyTorch and Qiskit

Machine learning (ML) has established itself as a successful interdisciplinary field which seeks to mathematically extract generalizable information from data. Throwing in quantum computing gives rise to interesting areas of research which seek to leverage the principles of quantum mechanics to augment machine learning or vice-versa. Whether you're aiming to enhance classical ML algorithms by outsourcing difficult calculations to a quantum computer or optimise quantum algorithms using classical ML architectures - both fall under the diverse umbrella of quantum machine learning (QML).

In this chapter, we explore how a classical neural network can be partially quantized to create a hybrid quantum-classical neural network. We will code up a simple example that integrates Qiskit with a state-of-the-art open-source software package - PyTorch. The purpose of this example is to demonstrate the ease of integrating Qiskit with existing ML tools and to encourage ML practitioners to explore what is possible with quantum computing.

Contents

1. How does it work?

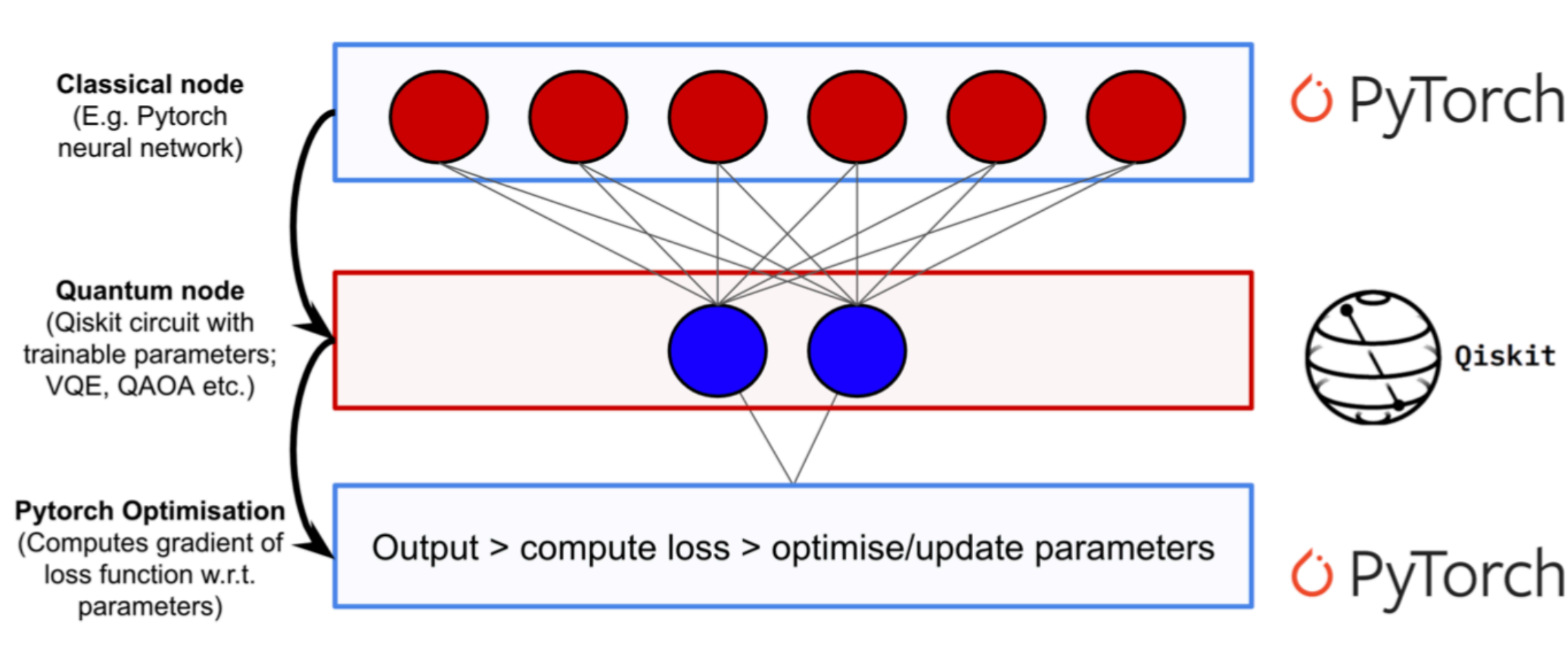


Fig.1 Illustrates the framework we will construct in this chapter. Ultimately, we will create a hybrid quantum-classical neural network that seeks to classify hand drawn digits. Note that the edges shown in this image are all directed downward; however, the directionality is not visually indicated.

1.1 Preliminaries

The background presented here on classical neural networks is included to establish relevant ideas and shared terminology; however, it is still extremely high-level. If you'd like to dive one step deeper into classical neural networks, see the well made video series by youtuber 3Blue1Brown. Alternatively, if you are already familiar with classical networks, you can skip to the next section.

Neurons and Weights

A neural network is ultimately just an elaborate function that is built by composing smaller building blocks called neurons. A neuron is typically a simple, easy-to-compute, and nonlinear function that maps one or more inputs to a single real number. The single output of a neuron is typically copied and fed as input into other neurons. Graphically, we represent neurons as nodes in a graph and we draw directed edges between nodes to indicate how the output of one neuron will be used as input to other neurons. It's also important to note that each edge in our graph is often associated with a scalar-value called a weight. The idea here is that each of the inputs to a neuron will be multiplied by a different scalar before being collected and processed into a single value. The objective when training a neural network consists primarily of choosing our weights such that the network behaves in a particular way.

Feed Forward Neural Networks

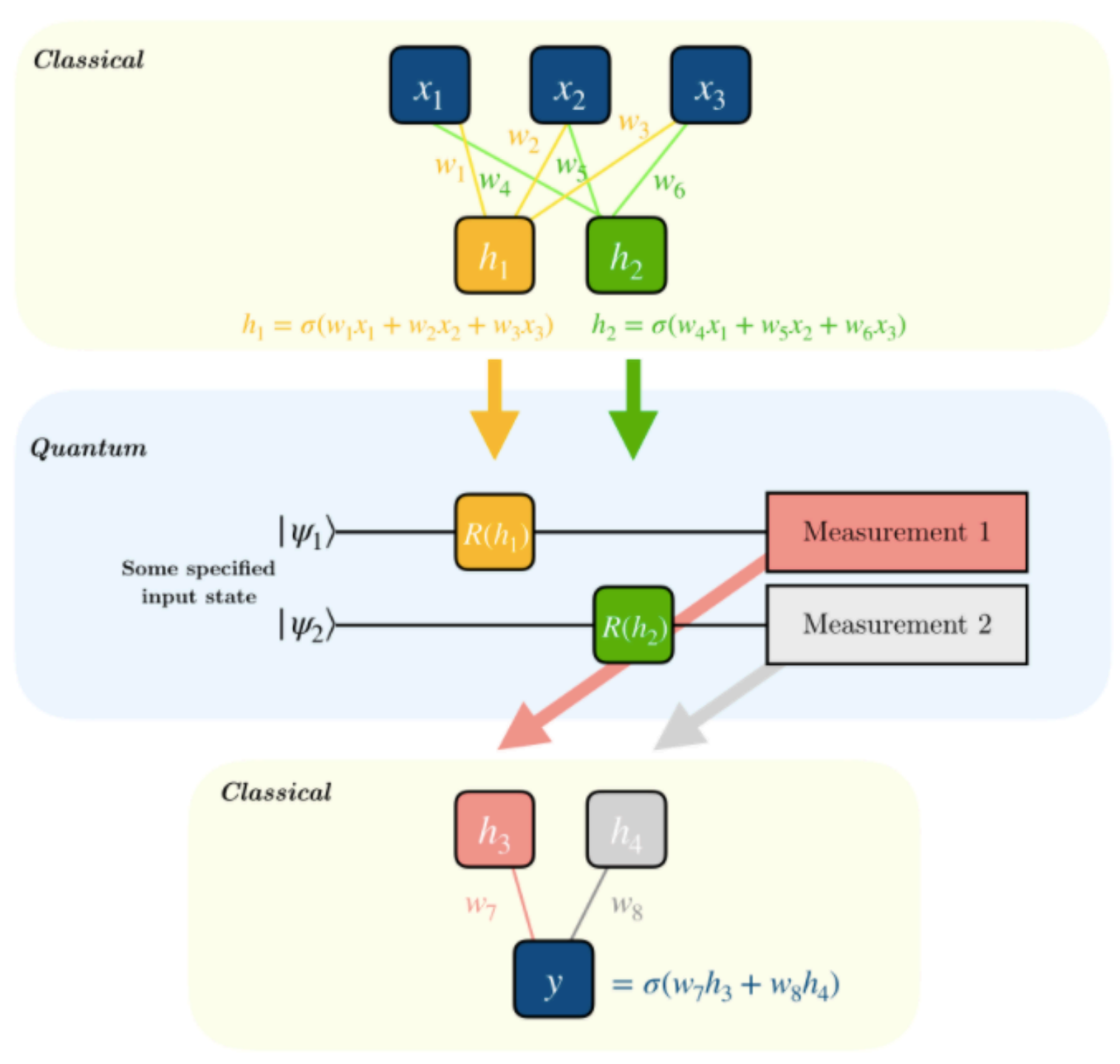
It is also worth noting that the particular type of neural network we will concern ourselves with is called a feed-forward neural network (FFNN). This means that as data flows through our neural network, it will never return to a neuron it has already visited. Equivalently, you could say that the graph which describes our neural network is a directed acyclic graph (DAG). Furthermore, we will stipulate that neurons within the same layer of our neural network will not have edges between them.

IO Structure of Layers

The input to a neural network is a classical (real-valued) vector. Each component of the input vector is multiplied by a different weight and fed into a layer of neurons according to the graph structure of the network. After each neuron in the layer has been evaluated, the results are collected into a new vector where the i'th component records the output of the i'th neuron. This new vector can then be treated as an input for a new layer, and so on. We will use the standard term hidden layer to describe all but the first and last layers of our network.

2. So How Does Quantum Enter the Picture?

To create a quantum-classical neural network, one can implement a hidden layer for our neural network using a parameterized quantum circuit. By "parameterized quantum circuit", we mean a quantum circuit where the rotation angles for each gate are specified by the components of a classical input vector. The outputs from our neural network's previous layer will be collected and used as the inputs for our parameterized circuit. The measurement statistics of our quantum circuit can then be collected and used as inputs for the following layer. A simple example is depicted below:



Here, σ is a nonlinear function and h_i is the value of neuron i at each hidden layer. $R(h_i)$ represents any rotation gate about an angle equal to h_i and y is the final prediction value generated from the hybrid network.

What about backpropagation?

If you're familiar with classical ML, you may immediately be wondering how do we calculate gradients when quantum circuits are involved? This would be necessary to enlist powerful optimisation techniques such as gradient descent. It gets a bit technical, but in short, we can view a quantum circuit as a black box and the gradient of this black box with respect to its parameters can be calculated as follows:

$$\nabla_\theta \boxed{\text{Quantum Circuit}(\theta)} = \boxed{\text{Quantum Circuit}(\theta + s)} - \boxed{\text{Quantum Circuit}(\theta - s)}$$

where θ represents the parameters of the quantum circuit and s is a macroscopic shift. The gradient is then simply the difference between our quantum circuit evaluated at $\theta + s$ and $\theta - s$. Thus, we can systematically differentiate our quantum circuit as part of a larger backpropagation routine. This closed form rule for calculating the gradient of quantum circuit parameters is known as the parameter shift rule.

3. Let's code!

3.1 Imports

First, we import some handy packages that we will need, including Qiskit and PyTorch.

In [ ]:

```python
import numpy as np
from IPython.display import Image
import matplotlib.pyplot as plt
```

In [ ]:

```
!pip uninstall imgaug && pip uninstall albumentations && pip install git+https://github.com/aleju/imgaug.git
```

```
Found existing installation: imgaug 0.2.9
Uninstalling imgaug-0.2.9:
  Would remove:
    /usr/local/lib/python3.7/dist-packages/imgaug-0.2.9.dist-info/*
    /usr/local/lib/python3.7/dist-packages/imgaug/*
Proceed (y/n)? y
  Successfully uninstalled imgaug-0.2.9
Found existing installation: albumentations 0.1.12
Uninstalling albumentations-0.1.12:
  Would remove:
    /usr/local/lib/python3.7/dist-packages/albumentations-0.1.12.dist-info/*
    /usr/local/lib/python3.7/dist-packages/albumentations/*
Proceed (y/n)? y
  Successfully uninstalled albumentations-0.1.12
Collecting git+https://github.com/aleju/imgaug.git
  Cloning https://github.com/aleju/imgaug.git to /tmp/pip-req-build-aatedtdv
  Running command git clone -q https://github.com/aleju/imgaug.git /tmp/pip-req-build-aatedtdv
Requirement already satisfied: six in /usr/local/lib/python3.7/dist-packages (from imgaug==0.4.0) (1.15.0)
Requirement already satisfied: numpy>=1.15 in /usr/local/lib/python3.7/dist-packages (from imgaug==0.4.0) (1.21.5)
Requirement already satisfied: scipy in /usr/local/lib/python3.7/dist-packages (from imgaug==0.4.0) (1.4.1)
Requirement already satisfied: Pillow in /usr/local/lib/python3.7/dist-packages (from imgaug==0.4.0) (7.1.2)
Requirement already satisfied: matplotlib in /usr/local/lib/python3.7/dist-packages (from imgaug==0.4.0) (3.2.2)
Requirement already satisfied: scikit-image>=0.14.2 in /usr/local/lib/python3.7/dist-packages (from imgaug==0.4.0) (0.18.3)
Requirement already satisfied: opencv-python in /usr/local/lib/python3.7/dist-packages (from imgaug==0.4.0) (4.1.2.30)
Requirement already satisfied: Shapely in /usr/local/lib/python3.7/dist-packages (from imgaug==0.4.0) (1.8.1.post1)
Requirement already satisfied: imageio in /usr/local/lib/python3.7/dist-packages (from imgaug==0.4.0) (2.4.1)
Requirement already satisfied: networkx>=2.0 in /usr/local/lib/python3.7/dist-packages (from scikit-image>=0.14.2->imgaug==0.4.0) (
2.6.3)
Requirement already satisfied: tifffile>=2019.7.26 in /usr/local/lib/python3.7/dist-packages (from scikit-image>=0.14.2->imgaug==0.
4.0) (2021.11.2)
Requirement already satisfied: PyWavelets>=1.1.1 in /usr/local/lib/python3.7/dist-packages (from scikit-image>=0.14.2->imgaug==0.4.
0) (1.2.0)
Requirement already satisfied: cycler>=0.10 in /usr/local/lib/python3.7/dist-packages (from matplotlib->imgaug==0.4.0) (0.11.0)
Requirement already satisfied: pyparsing!=2.0.4,!=2.1.2,!=2.1.6,>=2.0.1 in /usr/local/lib/python3.7/dist-packages (from matplotlib-
>imgaug==0.4.0) (3.0.7)
Requirement already satisfied: python-dateutil>=2.1 in /usr/local/lib/python3.7/dist-packages (from matplotlib->imgaug==0.4.0) (2.8
.2)
Requirement already satisfied: kiwisolver>=1.0.1 in /usr/local/lib/python3.7/dist-packages (from matplotlib->imgaug==0.4.0) (1.3.2)
Building wheels for collected packages: imgaug
  Building wheel for imgaug (setup.py) ... done
  Created wheel for imgaug: filename=imgaug-0.4.0-py3-none-any.whl size=971122 sha256=770b80e618c37ce5ae757e1e4a16a52afd260ca663140
e51fa64b7e2f4de73c6
  Stored in directory: /tmp/pip-ephem-wheel-cache-hxyv10kb/wheels/0c/78/b5/9303fae9d5e03df1f319adfe4e6534180b5c3232de11bc9a2f
Successfully built imgaug
Installing collected packages: imgaug
Successfully installed imgaug-0.4.0
```

In [ ]:

```
!pip install qiskit
```

```
Collecting qiskit
  Downloading qiskit-0.34.2.tar.gz (13 kB)
Collecting qiskit-terra==0.19.2
  Downloading qiskit_terra-0.19.2-cp37-cp37m-manylinux_2_12_x86_64.manylinux2010_x86_64.whl (6.5 MB)
     |████████████████████████████████| 6.5 MB 4.2 MB/s
Collecting qiskit-aer==0.10.3
  Downloading qiskit_aer-0.10.3-cp37-cp37m-manylinux_2_12_x86_64.manylinux2010_x86_64.whl (18.0 MB)
     |████████████████████████████████| 18.0 MB 104 kB/s
Collecting qiskit-ibmq-provider==0.18.3
  Downloading qiskit_ibmq_provider-0.18.3-py3-none-any.whl (238 kB)
     |████████████████████████████████| 238 kB 79.1 MB/s
Collecting qiskit-ignis==0.7.0
  Downloading qiskit_ignis-0.7.0-py3-none-any.whl (200 kB)
     |████████████████████████████████| 200 kB 78.5 MB/s
Requirement already satisfied: scipy>=1.0 in /usr/local/lib/python3.7/dist-packages (from qiskit-aer==0.10.3->qiskit) (1.4.1)
Requirement already satisfied: numpy>=1.16.3 in /usr/local/lib/python3.7/dist-packages (from qiskit-aer==0.10.3->qiskit) (1.21.5)
Collecting requests-ntlm>=1.1.0
  Downloading requests_ntlm-1.1.0-py2.py3-none-any.whl (5.7 kB)
Collecting websocket-client>=1.0.1
  Downloading websocket_client-1.3.1-py3-none-any.whl (54 kB)
     |████████████████████████████████| 54 kB 3.2 MB/s
Requirement already satisfied: python-dateutil>=2.8.0 in /usr/local/lib/python3.7/dist-packages (from qiskit-ibmq-provider==0.18.3-
>qiskit) (2.8.2)
Requirement already satisfied: requests>=2.19 in /usr/local/lib/python3.7/dist-packages (from qiskit-ibmq-provider==0.18.3->qiskit)
(2.23.0)
Requirement already satisfied: urllib3>=1.21.1 in /usr/local/lib/python3.7/dist-packages (from qiskit-ibmq-provider==0.18.3->qiskit
) (1.24.3)
Collecting retworkx>=0.8.0
  Downloading retworkx-0.11.0-cp37-cp37m-manylinux_2_5_x86_64.manylinux1_x86_64.manylinux_2_12_x86_64.manylinux2010_x86_64.whl (1.6
MB)
     |████████████████████████████████| 1.6 MB 56.5 MB/s
Requirement already satisfied: setuptools>=40.1.0 in /usr/local/lib/python3.7/dist-packages (from qiskit-ignis==0.7.0->qiskit) (57.
4.0)
Collecting stevedore>=3.0.0
  Downloading stevedore-3.5.0-py3-none-any.whl (49 kB)
     |████████████████████████████████| 49 kB 6.4 MB/s
Requirement already satisfied: dill>=0.3 in /usr/local/lib/python3.7/dist-packages (from qiskit-terra==0.19.2->qiskit) (0.3.4)
Collecting symengine>=0.8
  Downloading symengine-0.9.0-cp37-cp37m-manylinux2010_x86_64.whl (37.5 MB)
     |████████████████████████████████| 37.5 MB 415 kB/s
Collecting scipy>=1.0
  Downloading scipy-1.7.3-cp37-cp37m-manylinux_2_12_x86_64.manylinux2010_x86_64.whl (38.1 MB)
     |████████████████████████████████| 38.1 MB 1.2 MB/s
Collecting tweedledum<2.0,>=1.1
  Downloading tweedledum-1.1.1-cp37-cp37m-manylinux_2_12_x86_64.manylinux2010_x86_64.whl (943 kB)
     |████████████████████████████████| 943 kB 67.7 MB/s
Requirement already satisfied: sympy>=1.3 in /usr/local/lib/python3.7/dist-packages (from qiskit-terra==0.19.2->qiskit) (1.7.1)
Requirement already satisfied: psutil>=5 in /usr/local/lib/python3.7/dist-packages (from qiskit-terra==0.19.2->qiskit) (5.4.8)
Collecting python-constraint>=1.4
  Downloading python-constraint-1.4.0.tar.bz2 (18 kB)
Collecting ply>=3.10
  Downloading ply-3.11-py2.py3-none-any.whl (49 kB)
     |████████████████████████████████| 49 kB 7.3 MB/s
Requirement already satisfied: six>=1.5 in /usr/local/lib/python3.7/dist-packages (from python-dateutil>=2.8.0->qiskit-ibmq-provide
r==0.18.3->qiskit) (1.15.0)
Requirement already satisfied: certifi>=2017.4.17 in /usr/local/lib/python3.7/dist-packages (from requests>=2.19->qiskit-ibmq-provi
der==0.18.3->qiskit) (2021.10.8)
Requirement already satisfied: idna<3,>=2.5 in /usr/local/lib/python3.7/dist-packages (from requests>=2.19->qiskit-ibmq-provider==0
.18.3->qiskit) (2.10)
Requirement already satisfied: chardet<4,>=3.0.2 in /usr/local/lib/python3.7/dist-packages (from requests>=2.19->qiskit-ibmq-provid
er==0.18.3->qiskit) (3.0.4)
Collecting ntlm-auth>=1.0.2
  Downloading ntlm_auth-1.5.0-py2.py3-none-any.whl (29 kB)
Collecting cryptography>=1.3
  Downloading cryptography-36.0.1-cp36-abi3-manylinux_2_24_x86_64.whl (3.6 MB)
     |████████████████████████████████| 3.6 MB 51.7 MB/s
Requirement already satisfied: cffi>=1.12 in /usr/local/lib/python3.7/dist-packages (from cryptography>=1.3->requests-ntlm>=1.1.0->
qiskit-ibmq-provider==0.18.3->qiskit) (1.15.0)
Requirement already satisfied: pycparser in /usr/local/lib/python3.7/dist-packages (from cffi>=1.12->cryptography>=1.3->requests-nt
lm>=1.1.0->qiskit-ibmq-provider==0.18.3->qiskit) (2.21)
Requirement already satisfied: importlib-metadata>=1.7.0 in /usr/local/lib/python3.7/dist-packages (from stevedore>=3.0.0->qiskit-t
erra==0.19.2->qiskit) (4.11.1)
Collecting pbr!=2.1.0,>=2.0.0
  Downloading pbr-5.8.1-py2.py3-none-any.whl (113 kB)
     |████████████████████████████████| 113 kB 70.5 MB/s
Requirement already satisfied: zipp>=0.5 in /usr/local/lib/python3.7/dist-packages (from importlib-metadata>=1.7.0->stevedore>=3.0.
0->qiskit-terra==0.19.2->qiskit) (3.7.0)
Requirement already satisfied: typing-extensions>=3.6.4 in /usr/local/lib/python3.7/dist-packages (from importlib-metadata>=1.7.0->
stevedore>=3.0.0->qiskit-terra==0.19.2->qiskit) (3.10.0.2)
Requirement already satisfied: mpmath>=0.19 in /usr/local/lib/python3.7/dist-packages (from sympy>=1.3->qiskit-terra==0.19.2->qiski
t) (1.2.1)
Building wheels for collected packages: qiskit, python-constraint
  Building wheel for qiskit (setup.py) ... done
  Created wheel for qiskit: filename=qiskit-0.34.2-py3-none-any.whl size=11805 sha256=c4366adef9933e1a6870486763a3f6da21fe40f987633
20f4d7fb4747b96e92a
  Stored in directory: /root/.cache/pip/wheels/62/77/65/cda6eedfdd2a525bd3f479a4386930ae3088a1eb01f8c944ed
  Building wheel for python-constraint (setup.py) ... done
  Created wheel for python-constraint: filename=python_constraint-1.4.0-py2.py3-none-any.whl size=24081 sha256=4dd8a531157e2d82f3a2
9b3f1fb08dccee17b08c0e71d6b2b0abe22064acf10b
  Stored in directory: /root/.cache/pip/wheels/07/27/db/1222c80eb1e431f3d2199c12569cb1cac60f562a451fe30479
Successfully built qiskit python-constraint
Installing collected packages: pbr, tweedledum, symengine, stevedore, scipy, retworkx, python-constraint, ply, ntlm-auth, cryptogra
phy, websocket-client, requests-ntlm, qiskit-terra, qiskit-ignis, qiskit-ibmq-provider, qiskit-aer, qiskit
  Attempting uninstall: scipy
    Found existing installation: scipy 1.4.1
    Uninstalling scipy-1.4.1:
      Successfully uninstalled scipy-1.4.1
Successfully installed cryptography-36.0.1 ntlm-auth-1.5.0 pbr-5.8.1 ply-3.11 python-constraint-1.4.0 qiskit-0.34.2 qiskit-aer-0.10
.3 qiskit-ibmq-provider-0.18.3 qiskit-ignis-0.7.0 qiskit-terra-0.19.2 requests-ntlm-1.1.0 retworkx-0.11.0 scipy-1.7.3 stevedore-3.5
.0 symengine-0.9.0 tweedledum-1.1.1 websocket-client-1.3.1
```

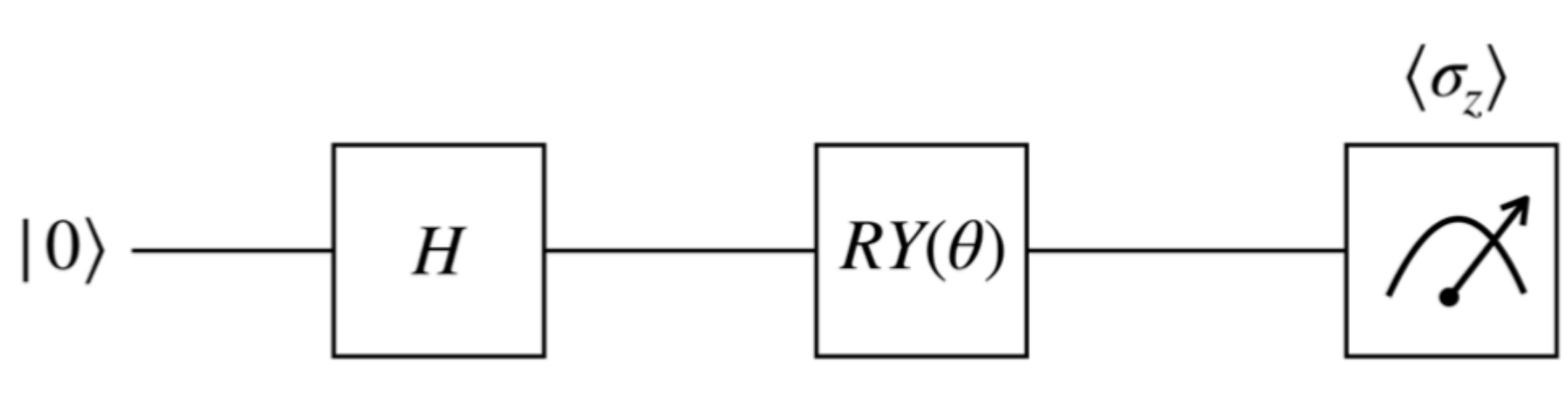
In [ ]:

```python
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

import qiskit
from qiskit import transpile, assemble
from qiskit.visualization import *
```

3.2 Create a "Quantum Class" with Qiskit

We can conveniently put our Qiskit quantum functions into a class. First, we specify how many trainable quantum parameters and how many shots we wish to use in our quantum circuit. In this example, we will keep it simple and use a 1-qubit circuit with one trainable quantum parameter θ. We hard code the circuit for simplicity and use a $RY-$ rotation by the angle θ to train the output of our circuit. The circuit looks like this:



In order to measure the output in the $z-$ basis, we calculate the σ_z expectation.

$$\sigma_z = . \sum_i z_i p(z_i)$$

We will see later how this all ties into the hybrid neural network.

In [ ]:

```python
class QuantumCircuit:
    """ 
    This class provides a simple interface for interaction 
    with the quantum circuit 
    """
    
    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])
```

Let's test the implementation

In [ ]:

```python
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()
```

```
Expected value for rotation pi 0.48
```

Out[ ]:

```
        ┌───┐ ░ ┌───────────┐ ░ ┌─┐
     q: ┤ H ├─░─┤ Ry(theta) ├─░─┤M├
        └───┘ ░ └───────────┘ ░ └╥┘
meas: 1/═════════════════════════╩═
                                 0 
```

3.3 Create a "Quantum-Classical Class" with PyTorch

Now that our quantum circuit is defined, we can create the functions needed for backpropagation using PyTorch. The forward and backward passes contain elements from our Qiskit class. The backward pass directly computes the analytical gradients using the finite difference formula we introduced above.

In [ ]:

```python
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):
        """ 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)
```

3.4 Data Loading and Preprocessing

Putting this all together:

We will create a simple hybrid neural network to classify images of two types of digits (0 or 1) from the MNIST dataset. We first load MNIST and filter for pictures containing 0's and 1's. These will serve as inputs for our neural network to classify.

Training data

In [ ]:

```python
# Concentrating on the first 100 samples
n_samples = 100

X_train = datasets.MNIST(root='./data', train=True, download=True,
                         transform=transforms.Compose([transforms.ToTensor()]))

# Leaving only labels 0 and 1 
idx = np.append(np.where(X_train.targets == 0)[0][:n_samples], 
                np.where(X_train.targets == 1)[0][:n_samples])

X_train.data = X_train.data[idx]
X_train.targets = X_train.targets[idx]

train_loader = torch.utils.data.DataLoader(X_train, batch_size=1, shuffle=True)
```

```
Downloading http://yann.lecun.com/exdb/mnist/train-images-idx3-ubyte.gz
Downloading http://yann.lecun.com/exdb/mnist/train-images-idx3-ubyte.gz to ./data/MNIST/raw/train-images-idx3-ubyte.gz

Extracting ./data/MNIST/raw/train-images-idx3-ubyte.gz to ./data/MNIST/raw

Downloading http://yann.lecun.com/exdb/mnist/train-labels-idx1-ubyte.gz
Downloading http://yann.lecun.com/exdb/mnist/train-labels-idx1-ubyte.gz to ./data/MNIST/raw/train-labels-idx1-ubyte.gz

Extracting ./data/MNIST/raw/train-labels-idx1-ubyte.gz to ./data/MNIST/raw

Downloading http://yann.lecun.com/exdb/mnist/t10k-images-idx3-ubyte.gz
Downloading http://yann.lecun.com/exdb/mnist/t10k-images-idx3-ubyte.gz to ./data/MNIST/raw/t10k-images-idx3-ubyte.gz

Extracting ./data/MNIST/raw/t10k-images-idx3-ubyte.gz to ./data/MNIST/raw

Downloading http://yann.lecun.com/exdb/mnist/t10k-labels-idx1-ubyte.gz
Downloading http://yann.lecun.com/exdb/mnist/t10k-labels-idx1-ubyte.gz to ./data/MNIST/raw/t10k-labels-idx1-ubyte.gz

Extracting ./data/MNIST/raw/t10k-labels-idx1-ubyte.gz to ./data/MNIST/raw
```

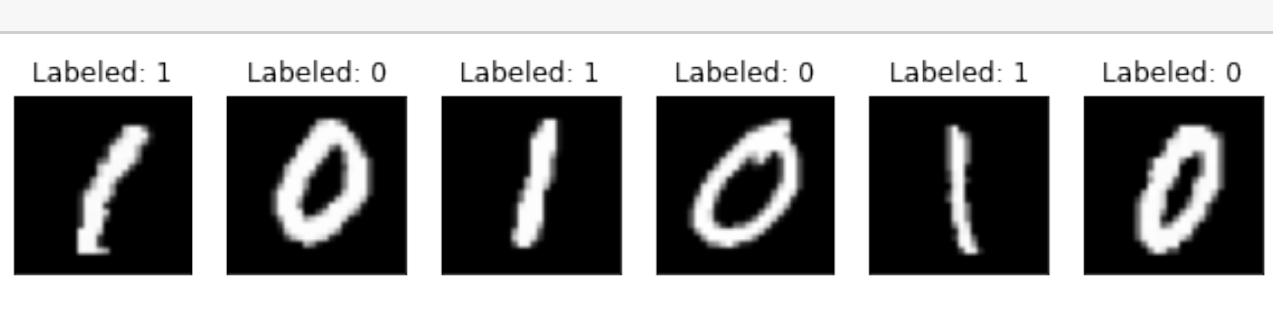
In [ ]:

```python
n_samples_show = 6

data_iter = iter(train_loader)
fig, axes = plt.subplots(nrows=1, ncols=n_samples_show, figsize=(10, 3))

while n_samples_show > 0:
    images, targets = data_iter.__next__()

    axes[n_samples_show - 1].imshow(images[0].numpy().squeeze(), cmap='gray')
    axes[n_samples_show - 1].set_xticks([])
    axes[n_samples_show - 1].set_yticks([])
    axes[n_samples_show - 1].set_title("Labeled: {}".format(targets.item()))
    
    n_samples_show -= 1
```



Testing data

In [ ]:

```python
n_samples = 50

X_test = datasets.MNIST(root='./data', train=False, download=True,
                        transform=transforms.Compose([transforms.ToTensor()]))

idx = np.append(np.where(X_test.targets == 0)[0][:n_samples], 
                np.where(X_test.targets == 1)[0][:n_samples])

X_test.data = X_test.data[idx]
X_test.targets = X_test.targets[idx]

test_loader = torch.utils.data.DataLoader(X_test, batch_size=1, shuffle=True)
```

So far, we have loaded the data and coded a class that creates our quantum circuit which contains 1 trainable parameter. This quantum parameter will be inserted into a classical neural network along with the other classical parameters to form the hybrid neural network. We also created backward and forward pass functions that allow us to do backpropagation and optimise our neural network. Lastly, we need to specify our neural network architecture such that we can begin to train our parameters using optimisation techniques provided by PyTorch.

3.5 Creating the Hybrid Neural Network

We can use a neat PyTorch pipeline to create a neural network architecture. The network will need to be compatible in terms of its dimensionality when we insert the quantum layer (i.e. our quantum circuit). Since our quantum in this example contains 1 parameter, we must ensure the network condenses neurons down to size 1. We create a typical Convolutional Neural Network with two fully-connected layers at the end. The value of the last neuron of the fully-connected layer is fed as the parameter θ into our quantum circuit. The circuit measurement then serves as the final prediction for 0 or 1 as provided by a σ_z measurement.

In [ ]:

```python
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(64, 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)
```

3.6 Training the Network

We now have all the ingredients to train our hybrid network! We can specify any PyTorch optimiser, learning rate and cost/loss function in order to train over multiple epochs. In this instance, we use the Adam optimiser, a learning rate of 0.001 and the negative log-likelihood loss function.

In [ ]:

```python
model = Net()
optimizer = optim.Adam(model.parameters(), lr=0.001)
loss_func = nn.NLLLoss()

epochs = 20
loss_list = []

model.train()
for epoch in range(epochs):
    total_loss = []
    for batch_idx, (data, target) in enumerate(train_loader):
        optimizer.zero_grad()
        # Forward pass
        output = model(data)
        # Calculating loss
        loss = loss_func(output, target)
        # Backward pass
        loss.backward()
        # 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]))
```

```
/usr/local/lib/python3.7/dist-packages/ipykernel_launcher.py:11: UserWarning: Creating a tensor from a list of numpy.ndarrays is ex
tremely slow. Please consider converting the list to a single numpy.ndarray with numpy.array() before converting to a tensor. (Trig
gered internally at  ../torch/csrc/utils/tensor_new.cpp:201.)
  # This is added back by InteractiveShellApp.init_path()
/usr/local/lib/python3.7/dist-packages/ipykernel_launcher.py:32: FutureWarning: The input object of type 'Tensor' is an array-like
implementing one of the corresponding protocols (`__array__`, `__array_interface__` or `__array_struct__`); but not a sequence (or
0-D). In the future, this object will be coerced as if it was first converted using `np.array(obj)`. To retain the old behaviour, y
ou have to either modify the type 'Tensor', or assign to an empty array created with `np.empty(correct_shape, dtype=object)`.
Training [5%]	Loss: -0.7978
Training [10%]	Loss: -0.9329
Training [15%]	Loss: -0.9456
Training [20%]	Loss: -0.9521
Training [25%]	Loss: -0.9550
Training [30%]	Loss: -0.9606
Training [35%]	Loss: -0.9623
Training [40%]	Loss: -0.9705
Training [45%]	Loss: -0.9727
Training [50%]	Loss: -0.9706
Training [55%]	Loss: -0.9819
Training [60%]	Loss: -0.9809
Training [65%]	Loss: -0.9822
Training [70%]	Loss: -0.9843
Training [75%]	Loss: -0.9849
Training [80%]	Loss: -0.9869
Training [85%]	Loss: -0.9908
Training [90%]	Loss: -0.9893
Training [95%]	Loss: -0.9923
Training [100%]	Loss: -0.9920
```

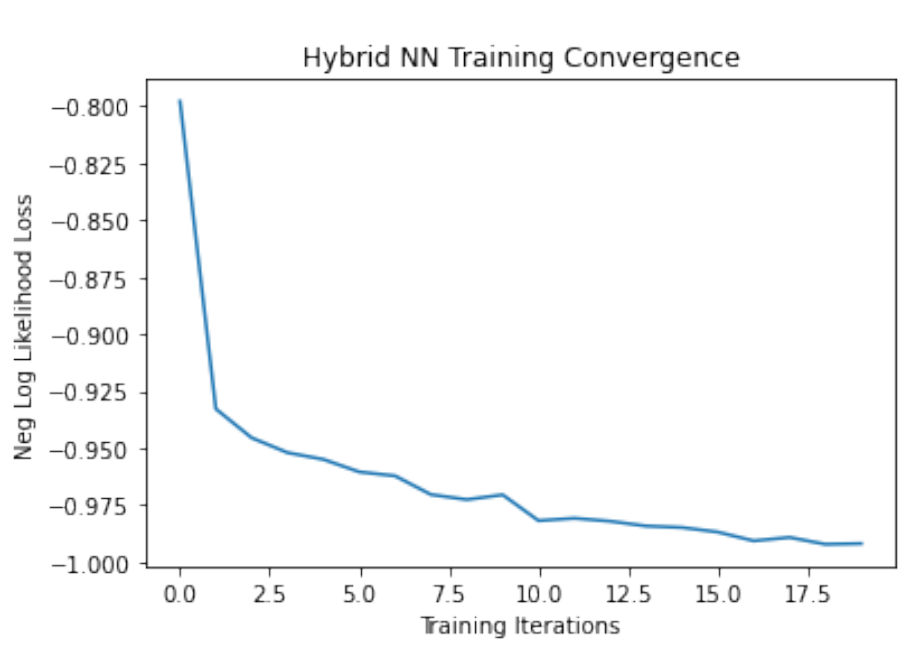
Plot the training graph

In [ ]:

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

Out[ ]:

```
Text(0, 0.5, 'Neg Log Likelihood Loss')
```



3.7 Testing the Network

In [ ]:

```python
model.eval()
with torch.no_grad():
    
    correct = 0
    for batch_idx, (data, target) in enumerate(test_loader):
        output = model(data)
        
        pred = output.argmax(dim=1, keepdim=True) 
        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(test_loader) * 100)
        )
```

```
Performance on test data:
	Loss: -0.9851
	Accuracy: 100.0%
```

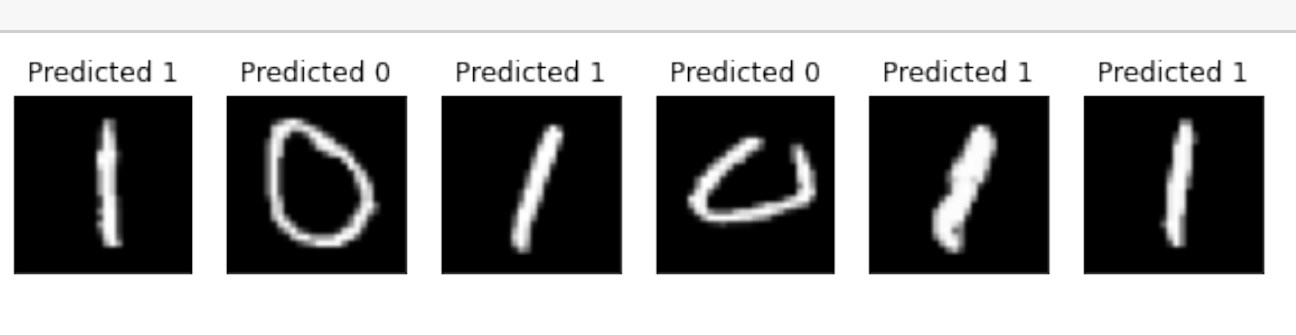
In [ ]:

```python
n_samples_show = 6
count = 0
fig, axes = plt.subplots(nrows=1, ncols=n_samples_show, figsize=(10, 3))

model.eval()
with torch.no_grad():
    for batch_idx, (data, target) in enumerate(test_loader):
        if count == n_samples_show:
            break
        output = model(data)
        
        pred = output.argmax(dim=1, keepdim=True) 

        axes[count].imshow(data[0].numpy().squeeze(), cmap='gray')

        axes[count].set_xticks([])
        axes[count].set_yticks([])
        axes[count].set_title('Predicted {}'.format(pred.item()))
        
        count += 1
```



4. What Now?

While it is totally possible to create hybrid neural networks, does this actually have any benefit?

In fact, the classical layers of this network train perfectly fine (in fact, better) without the quantum layer. Furthermore, you may have noticed that the quantum layer we trained here generates no entanglement, and will, therefore, continue to be classically simulatable as we scale up this particular architecture. This means that if you hope to achieve a quantum advantage using hybrid neural networks, you'll need to start by extending this code to include a more sophisticated quantum layer.

The point of this exercise was to get you thinking about integrating techniques from ML and quantum computing in order to investigate if there is indeed some element of interest - and thanks to PyTorch and Qiskit, this becomes a little bit easier.

In [ ]:

```python
import qiskit.tools.jupyter
%qiskit_version_table
```

Version Information

Qiskit Software	Version
`qiskit-terra`	0.19.2
`qiskit-aer`	0.10.3
`qiskit-ignis`	0.7.0
`qiskit-ibmq-provider`	0.18.3
System information	
Python version	3.7.12
Python compiler	GCC 7.5.0
Python build	default, Jan 15 2022 18:48:18
OS	Linux
CPUs	1
Memory (Gb)	12.686653137207031
	Mon Feb 28 21:02:36 2022 UTC