```
!pip install pennylane --upgrade
```

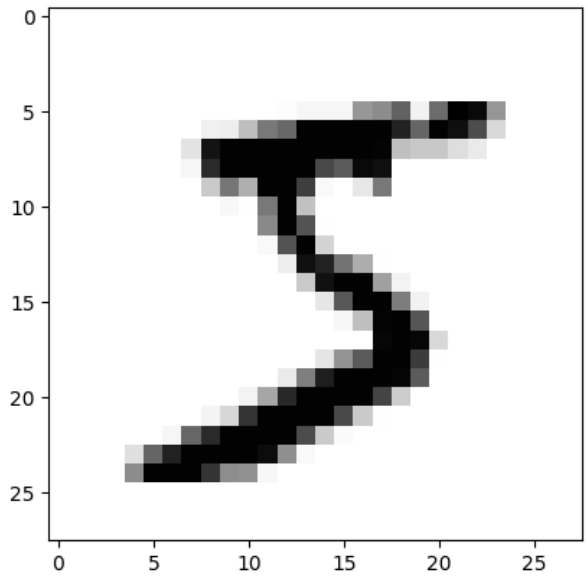
```
Collecting pennylane
  Downloading PennyLane-0.35.1-py3-none-any.whl (1.6 MB)
     ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 1.6/1.6 MB 6.1 MB/s eta 0:00:00
Requirement already satisfied: numpy in /usr/local/lib/python3.10/dist-packages (from pennylane) (1.25.2)
Requirement already satisfied: scipy in /usr/local/lib/python3.10/dist-packages (from pennylane) (1.11.4)
Requirement already satisfied: networkx in /usr/local/lib/python3.10/dist-packages (from pennylane) (3.2.1)
Collecting rustworkx (from pennylane)
  Downloading rustworkx-0.14.2-cp310-cp310-manylinux_2_17_x86_64.manylinux2014_x86_64.whl (2.1 MB)
     ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 2.1/2.1 MB 27.9 MB/s eta 0:00:00
Requirement already satisfied: autograd in /usr/local/lib/python3.10/dist-packages (from pennylane) (1.6.2)
Requirement already satisfied: toml in /usr/local/lib/python3.10/dist-packages (from pennylane) (0.10.2)
Requirement already satisfied: appdirs in /usr/local/lib/python3.10/dist-packages (from pennylane) (1.4.4)
Collecting semantic-version>=2.7 (from pennylane)
  Downloading semantic_version-2.10.0-py2.py3-none-any.whl (15 kB)
Collecting autoray>=0.6.1 (from pennylane)
  Downloading autoray-0.6.9-py3-none-any.whl (49 kB)
     ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 49.8/49.8 kB 3.0 MB/s eta 0:00:00
Requirement already satisfied: cachetools in /usr/local/lib/python3.10/dist-packages (from pennylane) (5.3.3)
Collecting pennylane-lightning>=0.35 (from pennylane)
  Downloading PennyLane_Lightning-0.35.1-cp310-cp310-manylinux_2_17_x86_64.manylinux2014_x86_64.whl (18.5 MB)
     ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 18.5/18.5 MB 33.8 MB/s eta 0:00:00
Requirement already satisfied: requests in /usr/local/lib/python3.10/dist-packages (from pennylane) (2.31.0)
Requirement already satisfied: typing-extensions in /usr/local/lib/python3.10/dist-packages (from pennylane) (4.10.0)
Requirement already satisfied: future>=0.15.2 in /usr/local/lib/python3.10/dist-packages (from autograd->pennylane) (0.18
Requirement already satisfied: charset-normalizer<4,>=2 in /usr/local/lib/python3.10/dist-packages (from requests->pennyl
Requirement already satisfied: idna<4,>=2.5 in /usr/local/lib/python3.10/dist-packages (from requests->pennylane) (3.6)
Requirement already satisfied: urllib3<3,>=1.21.1 in /usr/local/lib/python3.10/dist-packages (from requests->pennylane) (
Requirement already satisfied: certifi>=2017.4.17 in /usr/local/lib/python3.10/dist-packages (from requests->pennylane) (
Installing collected packages: semantic-version, rustworkx, autoray, pennylane-lightning, pennylane
Successfully installed autoray-0.6.9 pennylane-0.35.1 pennylane-lightning-0.35.1 rustworkx-0.14.2 semantic-version-2.10.0
```

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

# pennylane
import pennylane as qml
from functools import partial

# Tensorflow
import tensorflow as tf
from tensorflow.keras.preprocessing import image
from tensorflow.keras.models import Sequential

# MNIST Dataset
from tensorflow.keras.datasets import mnist
```

Dataset Preparation and other Info

```
# Load MNIST dataset using TensorFlow
(train_images, train_labels), (test_images, test_labels) = mnist.load_data()

print(train_images.shape, train_labels.shape, test_images.shape, test_labels.shape)

# extract one image data
plt.imshow(train_images[0, :, :], cmap='Greys')
```

```
(60000, 28, 28) (60000,) (10000, 28, 28) (10000,)
<matplotlib.image.AxesImage at 0x7e4a6414c580>
```



```
# get ranges for normalization
print('max:', np.max(train_images[0, :, :]), 'min:', np.min(train_images[0, :, :]))
```

```
max: 255 min: 0
```

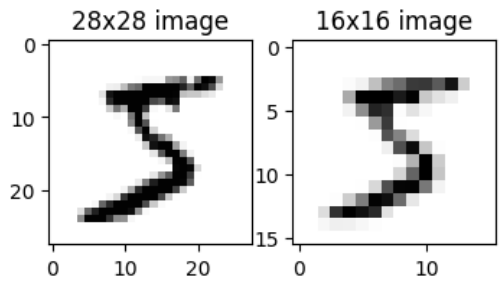
As we would see in the encoding section, the best we can do to store classical information in qubits limits to 2^n classical features into n qubits

To store 784 pixel data into amplitude, we need atleast a 10 qubit state to store the image data. This when put into a circuit to compare 2 images, the qubit requirements may go to 25-30 qubits. Simulating this circuit will go beyond the compute limits.

So we reduce the image size to 16x16 = 256 pixel values. This can be put into a 8 qubit state, reducing the qubit sizes as much as possible.

```
# visual comparison between 28x28 and 16x16 images

train_images = train_images.reshape(-1, 28, 28, 1).astype(np.float32) / 255.0  # Reshape into 4d array and normalize
print(train_images.shape)

# resize the images to 16x16
X_train = tf.image.resize(train_images, [16,16])
print(X_train.shape)

plt.figure(figsize=(4, 2))

plt.subplot(1, 2, 1)
plt.imshow(train_images[0, :, :], cmap='Greys')
plt.title('28x28 image')

plt.subplot(1, 2, 2)
plt.imshow(X_train[0, :, :], cmap='Greys')
plt.title('16x16 image')

plt.show()
```

```
(60000, 28, 28, 1)
(60000, 16, 16, 1)
```



We normalize the data arrays from ranges [0, 255] to [0, 1], And resize the images to 16x16, and also prepare a dataset to batch two random images and provide it every time we iterate through it. (custom dataset is made in tensorflow)

```
def prepare_dataset(batch_size):
    # Load MNIST dataset
    (x_train, y_train), _ = tf.keras.datasets.mnist.load_data()
    x_train = x_train.astype('float32') / 255.0  # Normalize pixel values
    print(x_train.shape)
    x_train = x_train.reshape(-1, 28, 28, 1)
    x_train = tf.image.resize(x_train, [16,16])
    x_train = tf.squeeze(x_train)
    x_train = tf.reshape(x_train, [60000, -1])
    print(x_train.shape)
    # Convert labels to one-hot encoding
    # y_train = tf.one_hot(y_train, depth=10)

    # Create TensorFlow dataset from the MNIST data
    dataset = tf.data.Dataset.from_tensor_slices((x_train, y_train))

    # Shuffle and batch the dataset
    dataset = dataset.shuffle(buffer_size=10000).batch(batch_size)

    return dataset


# obtain a train dataset for later use
batch_size = 2 # atleast 2 has to be chosen to get 2 images to be compared and contrasted
train_dataset = prepare_dataset(batch_size)
```

```
(60000, 28, 28)
(60000, 256)
```

Function Definitions

1. Image to State-Vector Function

We see that images are of size 16x16 ie., 256 pixels with each pixel having a float value. So to store their info, we require a state where we can encode atleast 256 values as amplitudes/phases.

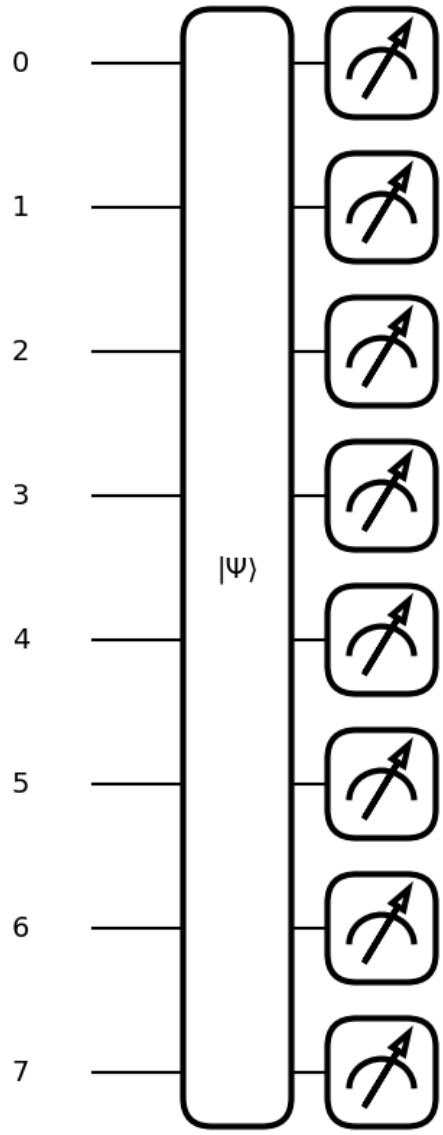
Available encoding methods are:

1. Amplitude encoding - 2^n features into n qubits
2. Phase angle encoding - Encodes Max of n features into n qubits
3. Basis Encoding - Encodes n binary features into a basis state of n qubits.

There are other encodings which are more complicated, but dont provide the high num_features to qubit ratio that amplitude encoding provides, So we go forward with amplitude encoding.

```
# image to state - Quantum Circuit / Pennylane function

num_qubits = 8

@qml.qnode(qml.device('default.qubit'), wires=8)
def imageToState(image_array, weights):
    # image_array is (16, 16) shaped 2d numpy array
    # weights are (256,) shaped numpy 1d array
    flattened = tf.reshape(image_array, [-1])
    features = flattened * weights
    qml.AmplitudeEmbedding(features=features, wires=range(num_qubits), normalize=True)

    return qml.state()


qml.draw_mpl(imageToState)(np.random.rand(16, 16), np.random.rand(256))
```

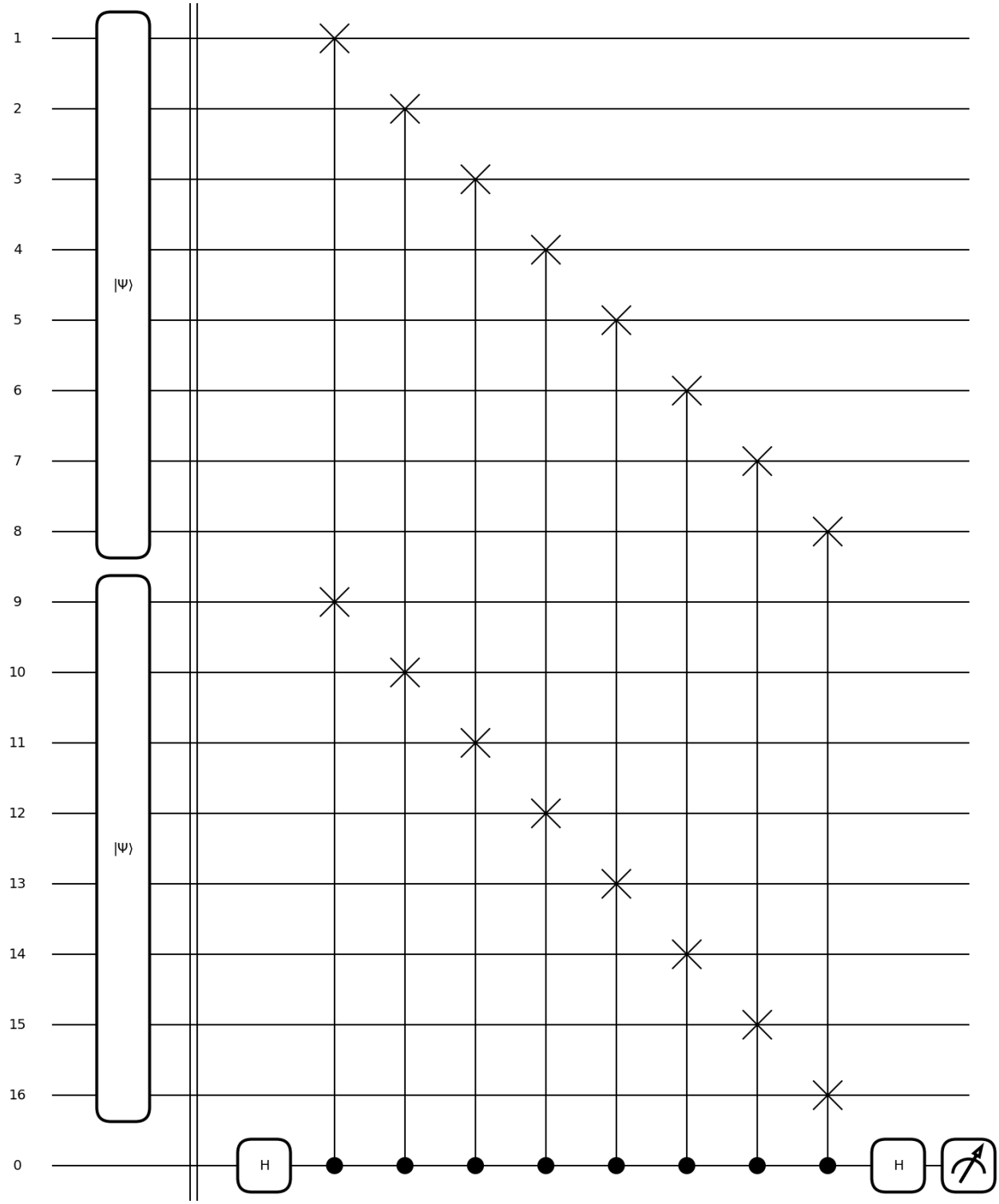
```
/usr/local/lib/python3.10/dist-packages/pennylane/workflow/qnode.py:475: UserWarning: Received gradient_kwarg wires, w
  warnings.warn(
(<Figure size 400x900 with 1 Axes>, <Axes: >)
```



2. Circuit function for Swap Test and Fidelity

```
@qml.qnode(qml.device('default.qubit'))
def swapTestCircuit(state1, state2):
    # state1, state2 are 8 qubit states initialized from image arrays
    # function returns the measurement from the topmost qubit (wire0) where the swap test results are stored

    # initialize states
    qml.QubitStateVector(state1, wires=range(1, 9))
    qml.QubitStateVector(state2, wires=range(9, 17))

    qml.Barrier(wires=range(16))

    # swap test
    qml.Hadamard(0)
    for i in range(1, 9):
        qml.CSWAP(wires = [0, i, 8+i])
    qml.Hadamard(0)

    return qml.probs(wires=0)

def getFidelity(state1, state2):
    probs = swapTestCircuit(state1, state2)
    return probs[0]


sv = imageToState(X_train[0, :, :, 0], np.random.rand(256))
qml.draw_mpl(swapTestCircuit)(sv, sv)
```

```
(<Figure size 1500x1800 with 1 Axes>, <Axes: >)
```



3. Combining them into single circuit

This makes the circuit to be easily embedded into the neural network

```
dev = qml.device("default.qubit")

# batch_params function is used to enable batched training of the model on mnist data
@partial(qml.batch_params, all_operations=True)
@qml.qnode(dev)
def swapTest(inputs, weights):
    image1 = inputs[:, 0:256]
    image2 = inputs[:, 256:512]

    # image to state and initialize
    features1 = image1 * weights
    qml.AmplitudeEmbedding(features=features1, wires=range(1, 9), normalize=True)
    features2 = image2 * weights
    qml.AmplitudeEmbedding(features=features2, wires=range(9, 17), normalize=True)

    # state1, state2 are 8 qubit states initialized from image arrays
    # function returns the measurement from the topmost qubit (wire0) where the swap test results are stored

    qml.Barrier(wires=range(17))

    # swap test
    qml.Hadamard(0)
    for i in range(1, 9):
        qml.CSWAP(wires = [0, i, 8+i])
    qml.Hadamard(0)

    return qml.probs(wires=0)

# uncomment and run to see the circuit
# qml.draw_mpl(swapTest)(np.random.rand(256*2), np.random.rand(256))
```

4. Hybrid Quantum-Classical NN model definition

using Keras Layer functionality of pennylane. Same can be done from pytorch as well. This task I will be doing this with tensorflow as I already explored dataloading and training Graph neural networks with pytorch (task-2)

```
# define model
```

This model consumes a lot of ram, as it has to simulate and train 17 qubit quantum network. Running on colab will not go past this stage, so try running this cell to verify if the model is working for a forward pass

```
qlayer = qml.qnn.KerasLayer(swapTest, weight_shapes, output_dim=output_dimen)
# run this to verify, batched outputs out of quantum circuits

model(np.random.rand(32, 512))
```

```
<tf.Tensor: shape=(32,), dtype=float32, numpy=
array([0.7471916 , 0.77812773, 0.797869  , 0.7657964 , 0.81011903,
       0.7801069 , 0.7720183 , 0.7733801 , 0.74419475, 0.7894096 ,
       0.76696897, 0.76363134, 0.7868311 , 0.75211626, 0.7458878 ,
       0.7674922 , 0.79894775, 0.76400054, 0.8009435 , 0.7927398 ,
       0.79217494, 0.79748845, 0.76757455, 0.8003708 , 0.75577444,
       0.78264344, 0.77935374, 0.81031466, 0.8117033 , 0.77034324,
       0.75336236, 0.7830683 ], dtype=float32)>
```

5. Training Phase

```
# other hyper-parameters
```

You can either use the customized training loop, or `model.fit()` to train the model. I was able to run it for 2 epochs max using the custom training loop, but the training process crashed before that using the fit method due to excessive ram usage (in both laptop and colab).

```
# customized training loop

epochs = 2
for epoch in range(epochs):
    print("\nStart of epoch %d" % (epoch,))
    for step, (images, labels) in enumerate(train_dataset.take(1)):

        # start merging 2 images
        # Randomly select two indices
        indices = tf.random.shuffle(tf.range(batch_size))
        first_indices = indices[:batch_size // 2]
        second_indices = indices[batch_size // 2:]

        # Randomly shuffle the images and labels within the batch
        first_images = tf.gather(images, first_indices)
        first_labels = tf.gather(labels, first_indices)
```