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

```
Requirement already satisfied: pennylane in /usr/local/lib/python3.10/dist-packages (0.35.1)
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)
Requirement already satisfied: rustworkx in /usr/local/lib/python3.10/dist-packages (from pennylane) (0.14.2)
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)
Requirement already satisfied: semantic-version>=2.7 in /usr/local/lib/python3.10/dist-packages (from pennylane) (2.10.0
Requirement already satisfied: autoray>=0.6.1 in /usr/local/lib/python3.10/dist-packages (from pennylane) (0.6.9)
Requirement already satisfied: cachetools in /usr/local/lib/python3.10/dist-packages (from pennylane) (5.3.3)
Requirement already satisfied: pennylane-lightning>=0.35 in /usr/local/lib/python3.10/dist-packages (from pennylane) (0.
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) (0.1
Requirement already satisfied: charset-normalizer<4,>=2 in /usr/local/lib/python3.10/dist-packages (from requests->penny
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)
```

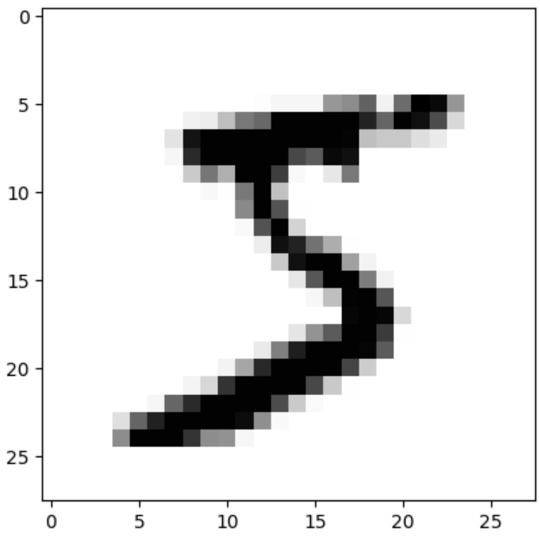
```
# QML imports
import pennylane as qml

# MNIST Dataset
from tensorflow.keras.datasets import mnist

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

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

## Dataset preparation and 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 0x79d34d392800>
```




```
# flat_img = train_images[0, :, :].flatten()
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

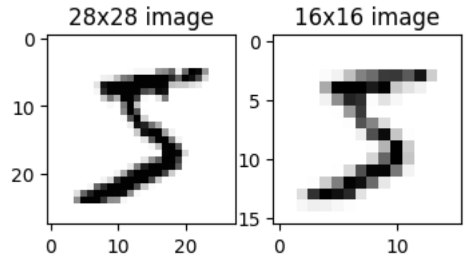
To store 784 pixel data into amplitude, we need atleast a 10 qubit state to store the image data. This when put into a circuit to compare 2 images, the qubit requirements may go to 25-30 qubits. Simulating this circuit will go beyond the compute limits.

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.

We normalize the data arrays from ranges [0, 255] to [0, 1], And resize the images to 16x16

```
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)
```




## Quantum Circuits and Encoding

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.

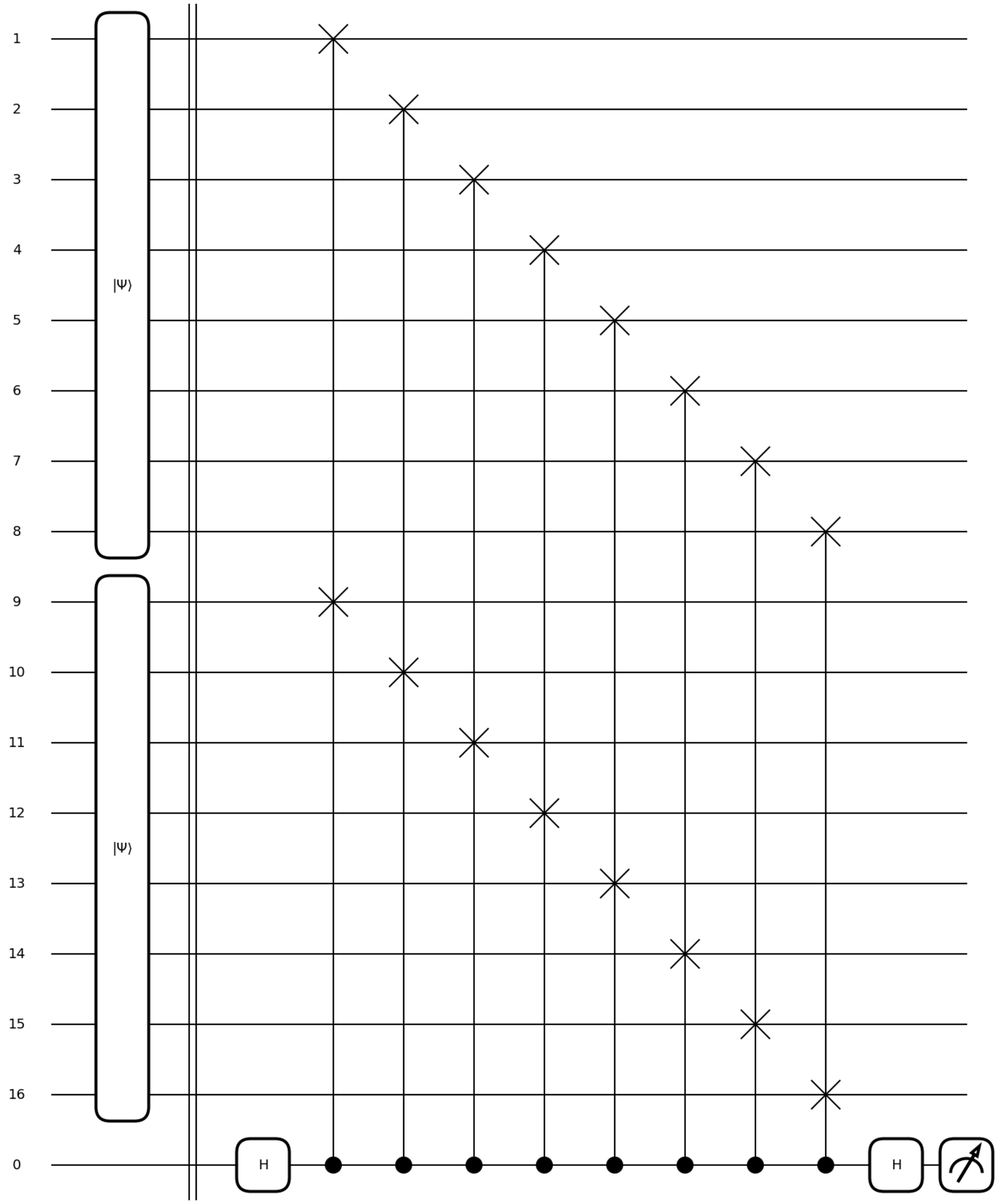
There are other encodings which are more complicated, but dont provide the high num_features to qubit ratio that amplitude encoding provides, So we go forward with amplitude encoding.

```
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()
```

```
/usr/local/lib/python3.10/dist-packages/pennylane/workflow/qnode.py:475: UserWarning: Received gradient_kwarg wires, whi
  warnings.warn(
```

```
num_qubits = 8

@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: >)
```



```
# define the final model and loss function
# both individual functions are combined into a single circuit, to speed up execution

@qml.qnode(qml.device('default.qubit'))
def quantumRepModel(inputs, weights0):
    image1, image2 = inputs
    # Embed images into quantum states
    im_arr1 = image1[:, :, 0]
    im_arr2 = image2[:, :, 0]
    #state1 = imageToState(im_arr1, weights)

    flattened1 = tf.reshape(im_arr1, [-1])
    features1 = flattened1 * weights0
    qml.AmplitudeEmbedding(features=features1, wires=range(1, 9), normalize=True)

    #state2 = imageToState(im_arr2, weights)
    flattened2 = np.array(im_arr2).flatten()
    features2 = flattened2 * weights0
    qml.AmplitudeEmbedding(features=features2, wires=range(9, 17), normalize=True)

    # Calculate fidelity between quantum states
    # fidelity = swapTestCircuit(state1, state2)[0]
    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)

    # # Define contrastive loss based on image labels
    # loss = tf.cond(tf.equal(label1, label2),
    #                lambda: 1 - fidelity,
    #                lambda: tf.maximum(0., fidelity))

    # return loss


k = quantumRepModel((X_train[0], X_train[0]), np.random.rand(256))


np.shape(k)
```

```
(2,)
```

```
# Define the quantum circuit model
class QuantumModel(tf.keras.Model):
    def __init__(self):
        super(QuantumModel, self).__init__()
        self.num_parameters = 256
        self.params = tf.Variable(tf.random.normal(shape=(self.num_parameters,)), trainable=True)

    def train_step(self, data):
        # Unpack the data. Its structure depends on your model and
        # on what you pass to `fit()`.
        img1, y1, img2, y2 = data

        with tf.GradientTape() as tape:
            fidelity = self(x, training=True)  # Forward pass
            # Compute the loss value
            # (the loss function is configured in `compile()`)
            loss = self.compute_loss(y=fidelity, y_pred=(y1, y2))

        # Compute gradients
        trainable_vars = self.trainable_variables
        gradients = tape.gradient(loss, trainable_vars)
# Initialize the quantum model
quantum_model = QuantumModel()

# Define optimizer
optimizer = tf.keras.optimizers.Adam(learning_rate=0.001)

# Define training step
batch_size = 32

@tf.function
def train_step(image1, image2, label1, label2):

    with tf.GradientTape() as tape:
        # Forward pass: compute contrastive loss
        loss = quantum_model.compute_loss(quantum_model((image1, image2, label1, label2)), label1, label2)

    # Compute gradients
    gradients = tape.gradient(loss, quantum_model.trainable_variables)

    # Update weights
    optimizer.apply_gradients(zip(gradients, quantum_model.trainable_variables))

    return loss

# Training loop
num_epochs = 10
for epoch in range(num_epochs):

        image_batch, label_batch = next(image.ImageDataGenerator().flow(X_train, train_labels, batch_size=batch_size))
        # Sample another batch for contrastive loss (ideally with different labels)
        image_batch2, label_batch2 = next(image.ImageDataGenerator().flow(X_train, train_labels, batch_size=batch_size, shuf

        # Perform training step
        loss = train_step(image_batch[0], image_batch2[0], label_batch[0], label_batch2[0])

        print(f"Epoch {epoch + 1}, Step {i + 1}, Loss: {loss.numpy()}")
```

```
WARNING:tensorflow:AutoGraph could not transform <function _c3_mro at 0x79d3eea2c4c0> and will run it as-is.
Cause: for/else statement not yet supported
To silence this warning, decorate the function with @tf.autograph.experimental.do_not_convert
WARNING: AutoGraph could not transform <function _c3_mro at 0x79d3eea2c4c0> and will run it as-is.
Cause: for/else statement not yet supported
To silence this warning, decorate the function with @tf.autograph.experimental.do_not_convert
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-39-a06f547d5c60> in <cell line: 21>()
     26
     27         # Perform training step
---> 28         loss = train_step(image_batch[0], image_batch2[0], label_batch[0], label_batch2[0])
     29
     30         print(f"Epoch {epoch + 1}, Step {i + 1}, Loss: {loss.numpy()}")

                                    4 frames
/tmp/__autograph_generated_fileuaa25116.py in <lambda>()
     15                 ag__.converted_call(ag__.ld(qml).Barrier, (), dict(wires=ag__.converted_call(ag__.ld(range),
(16,), None, fscope)), fscope)
     16                 fidelity = ag__.converted_call(ag__.ld(getFidelity), (ag__.ld(state1), ag__.ld(state2)),
None, fscope)
---> 17                 loss = ag__.converted_call(ag__.ld(tf).cond, (ag__.converted_call(ag__.ld(tf).equal,
(ag__.ld(label1), ag__.ld(label2)), None, fscope), ag__.autograph_artifact(lambda : 1 - ag__.ld(fidelity)),
```