

# QUANTUM MACHINE LEARNING

There are certain types of problems that would take thousands of the years to be solved by classical computers with existing algorithms. There comes the need of Quantum Computing, they have the potential to recast computations by making certain types of intractable problems solvable. It is more power efficient and could speed up the learning process of AI using Quantum Machine Learning, reducing thousands of years of learning to mere seconds. **Quantum Machine Learning** is an intersection of Quantum Physics and Machine Learning, this term refers to machine learning algorithms for the analysis of classical data executed on a quantum computer.

**Quantum computing** relies on the properties of quantum mechanics to compute problems that would be nearly impossible for classical computers. **Qubits** used in this type of computing. Qubits are like regular bits in a computer, but with the added ability to be put into a superposition and share entanglement. By controlling superposition and entanglement, it is easy for a quantum computer to perform quantum operations that are difficult to emulate with classical computers. It also includes quantum sampling, annealing, optimization, cryptography, neural network and machine learning.

## Two Main concepts of Quantum Machine Learning

- **Quantum Data**

It is the data which is generated by a quantum system. It exhibits superposition and entanglement, leading to joint probability distributions. Joint probability distribution is a probability distribution having two or more independent random variables. In a joint distribution, each random variable will still have its own probability distribution, expected value, variance, and standard deviation. An

easy example would be of throwing two dice together and getting the possibilities. Similarly, there would be extremely complex probability distributions on the generation of Quantum Data, which would be exponential. It requires an exponential amount of classical computational resources to store or represent.

For example, the data samples generated from the Sycamore processor for Google's demonstration of quantum supremacy. A processor with programmable superconducting qubits to create quantum states on 53 qubits, corresponding to a computational state-space of dimension  $2^{53}$  (about  $10^{16}$ ). A Sycamore processor takes about 200 seconds to sample one instance of a quantum circuit a million times but the equivalent task, classical supercomputer, would take approximately 10,000 years.

Heuristic Machine learning techniques are capable of maximizing the extraction of useful data from noisy entangled data. Also, TensorFlow Quantum library can be used to develop models that disentangle and generalize correlations in quantum data.

- **Hybrid Quantum-Classical Models**

As quantum processors are still small and noisy, so cannot generalize quantum data using quantum processors alone.

TensorFlow supports heterogeneous computing across CPUs, GPUs, and TPUs, it is used as the base platform to experiment with hybrid quantum-classical algorithms.

A Quantum Neural Network (QNN) is used to describe a parameterized quantum computational model that is best executed on a quantum computer. Also called, Parameterized Quantum Circuit (PQC).

# RUNNING QML CODES ON JUPYTER

## Building a quantum neural network (QNN) to classify a simplified version of MNIST

- Loading the raw data from Keras.
- Filters the dataset to only 3s and 6s.
- Downscales the images so they fit in a quantum computer.
- Removes any contradictory examples.
- Converts the binary images to Cirq circuits.
- Converts the Cirq circuits to TensorFlow Quantum circuits.

### Setup

```
$ !pip install tensorflow==<version>
```

```
$ !pip install tensorflow-quantum
```

### Importing TensorFlow and module dependencies

```
$ import tensorflow as tf
import tensorflow_quantum as tfq
import cirq
import sympy
import numpy as np
import seaborn as sns
import collections

# visualization tools
%matplotlib inline
import matplotlib.pyplot as plt
from cirq.contrib.svg import SVGCircuit
```

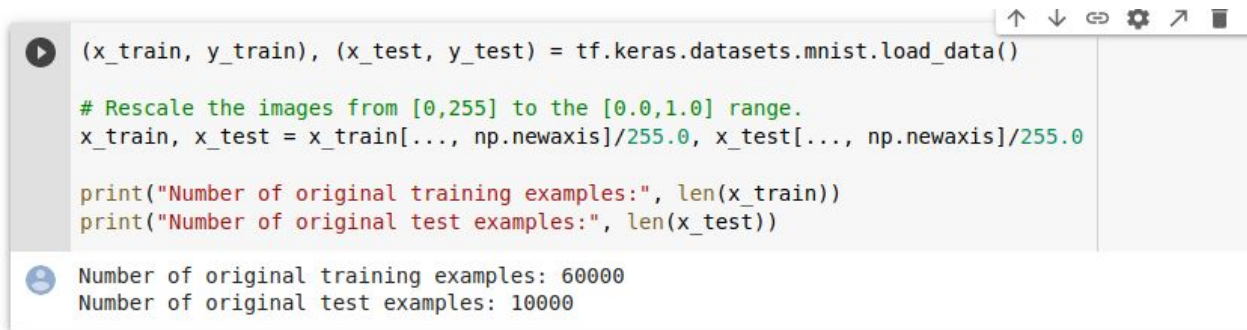
## Loading the data

### Loading the MNIST dataset distributed with Keras

```
(x_train, y_train), (x_test, y_test) = tf.keras.datasets.mnist.load_data()

# Re-scale the images from [0,255] to the [0.0,1.0] range.
x_train, x_test = x_train[...]/255.0, x_test[...]/255.0

print("Number of original training examples:", len(x_train))
print("Number of original test examples:", len(x_test))
```



```
(x_train, y_train), (x_test, y_test) = tf.keras.datasets.mnist.load_data()

# Rescale the images from [0,255] to the [0.0,1.0] range.
x_train, x_test = x_train[...]/255.0, x_test[...]/255.0

print("Number of original training examples:", len(x_train))
print("Number of original test examples:", len(x_test))
```

Number of original training examples: 60000  
Number of original test examples: 10000

### Filter the dataset to keep **3s** and **6s**. Convert Label **y** to boolean: True for 3 and False for 6

```
$ def filter_36(x, y):
    keep = (y == 3) | (y == 6)
    x, y = x[keep], y[keep]
    y = y == 3
    return x,y

x_train, y_train = filter_36(x_train, y_train)
x_test, y_test = filter_36(x_test, y_test)
print("Number of filtered training examples:", len(x_train))
print("Number of filtered test examples:", len(x_test))
```

```

x_train, y_train = filter_36(x_train, y_train)
x_test, y_test = filter_36(x_test, y_test)

print("Number of filtered training examples:", len(x_train))
print("Number of filtered test examples:", len(x_test))

```

```

Number of filtered training examples: 12049
Number of filtered test examples: 1968

```

```

print(y_train[0])
plt.imshow(x_train[0, :, :, 0])
plt.colorbar()

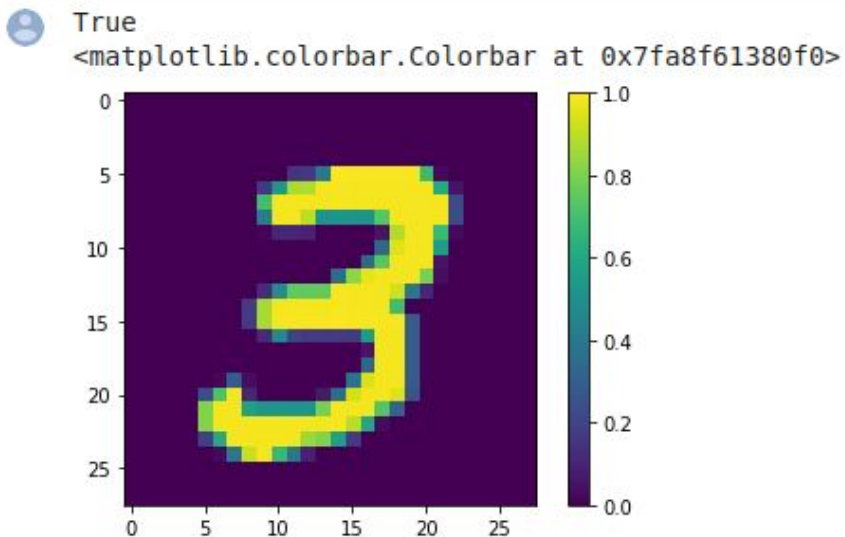
```

```

print(y_train[0])

plt.imshow(x_train[0, :, :, 0])
plt.colorbar()

```



Downscale the images (28x28 image size is too large for current quantum computers. Resize to 4x4)

```

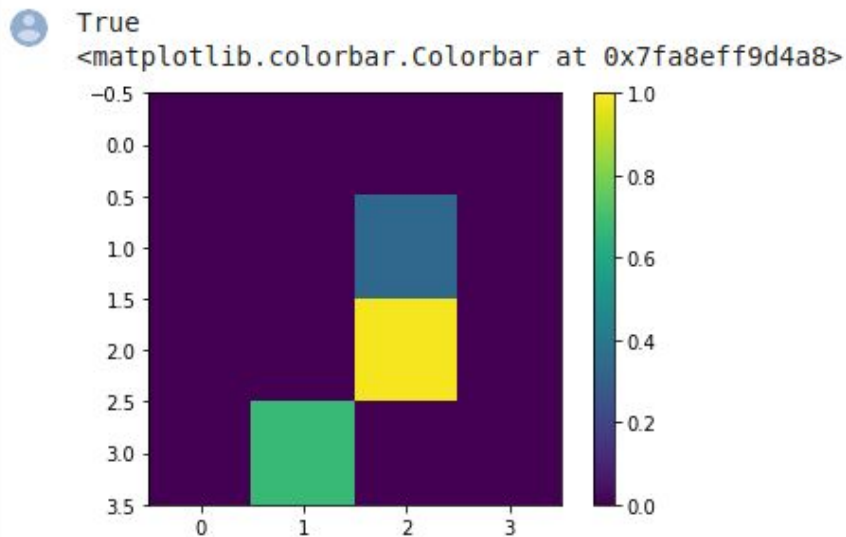
$ x_train_small = tf.image.resize(x_train, (4,4)).numpy()
  x_test_small = tf.image.resize(x_test, (4,4)).numpy()

```

```
print(y_train[0])
plt.imshow(x_train_small[0,:,:,:0], vmin=0, vmax=1)
plt.colorbar()
```

```
▶ x_train_small = tf.image.resize(x_train, (4,4)).numpy()
   x_test_small = tf.image.resize(x_test, (4,4)).numpy()

   print(y_train[0])
   plt.imshow(x_train_small[0,:,:,:0], vmin=0, vmax=1)
   plt.colorbar()
```



Filtering the dataset to remove images that are labeled as belonging to both classes

```
$ def remove_contradicting(xs, ys):
    mapping = collections.defaultdict(set)
    # Determine the set of labels for each unique image:
    for x,y in zip(xs,ys):
        mapping[tuple(x.flatten())].add(y)

    new_x = []
    new_y = []
```

```

for x,y in zip(xs, ys):
    labels = mapping[tuple(x.flatten())]
    if len(labels) == 1:
        new_x.append(x)
        new_y.append(list(labels)[0])
    else:
        # Throw out images that match more than one label.
        pass

num_3 = sum(1 for value in mapping.values() if True in value)
num_6 = sum(1 for value in mapping.values() if False in value)
num_both = sum(1 for value in mapping.values() if len(value) == 2)

print("Number of unique images:", len(mapping.values()))
print("Number of 3s: ", num_3)
print("Number of 6s: ", num_6)
print("Number of contradictory images: ", num_both)
print()
print("Initial number of examples: ", len(xs))
print("Remaining non-contradictory examples: ", len(new_x))

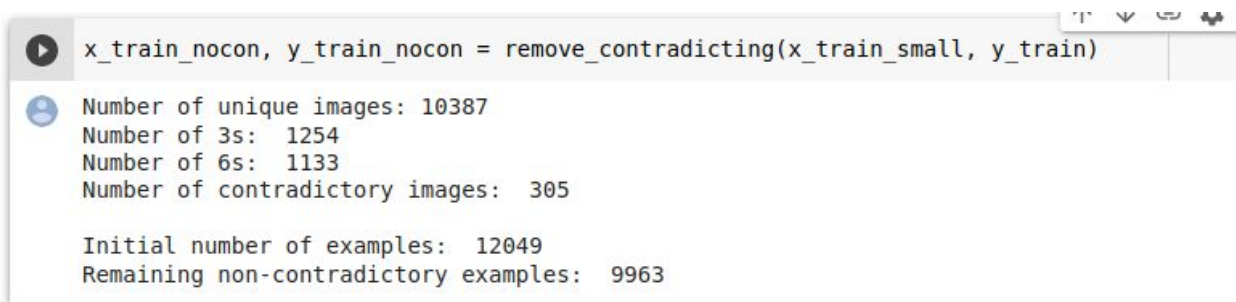
return np.array(new_x), np.array(new_y)

```

```

$ x_train_nocon, y_train_nocon = remove_contradicting(x_train_small,
y_train)

```



A screenshot of a Jupyter Notebook interface. The top part shows a code cell with the line: `x_train_nocon, y_train_nocon = remove_contradicting(x_train_small, y_train)`. Below the code cell is an output cell containing the following text:   
 Number of unique images: 10387  
 Number of 3s: 1254  
 Number of 6s: 1133  
 Number of contradictory images: 305  
  
 Initial number of examples: 12049  
 Remaining non-contradictory examples: 9963

### Encode the data as quantum circuits

- The process is representing each pixel with a qubit.
- First step is to convert to a binary encoding.

```
$ THRESHOLD = 0.5
   x_train_bin = np.array(x_train_nocon > THRESHOLD,
dtype=np.float32)
   x_test_bin = np.array(x_test_small > THRESHOLD,
dtype=np.float32)
```

[ Removing contradictory images ]

```
$ _ = remove_contradicting(x_train_bin, y_train_nocon)
```

```
▶ THRESHOLD = 0.5

x_train_bin = np.array(x_train_nocon > THRESHOLD, dtype=np.float32)
x_test_bin = np.array(x_test_small > THRESHOLD, dtype=np.float32)
_ = remove_contradicting(x_train_bin, y_train_nocon)

Number of unique images: 193
Number of 3s: 105
Number of 6s: 99
Number of contradictory images: 24

Initial number of examples: 9963
Remaining non-contradictory examples: 54
```

[ The qubits at pixel indices with values that exceed a threshold, are rotated through an X gate ]

```
$ def convert_to_circuit(image):
    """Encode truncated classical image into quantum datapoint."""
    values = np.ndarray.flatten(image)
    qubits = cirq.GridQubit.rect(4, 4)
    circuit = cirq.Circuit()
    for i, value in enumerate(values):
        if value:
            circuit.append(cirq.X(qubits[i]))
```



```

        return circuit
    x_train_circ = [convert_to_circuit(x) for x in x_train_bin]
    x_test_circ = [convert_to_circuit(x) for x in x_test_bin]

```

[ Circuit created ]

\$ SVGCircuit(x\_train\_circ[0])

```

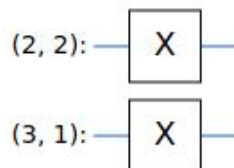
def convert_to_circuit(image):
    """Encode truncated classical image into quantum datapoint."""
    values = np.ndarray.flatten(image)
    qubits = cirq.GridQubit.rect(4, 4)
    circuit = cirq.Circuit()
    for i, value in enumerate(values):
        if value:
            circuit.append(cirq.X(qubits[i]))
    return circuit

x_train_circ = [convert_to_circuit(x) for x in x_train_bin]
x_test_circ = [convert_to_circuit(x) for x in x_test_bin]

SVGCircuit(x_train_circ[0])

```

findfont: Font family ['Arial'] not found. Falling back to DejaVu Sans.



[ Comparing this circuit to the indices where the image value exceeds the threshold ]

```

$ bin_img = x_train_bin[0,:,:,0]
  indices = np.array(np.where(bin_img)).T
  indices

```

```

bin_img = x_train_bin[0, :, :, 0]
indices = np.array(np.where(bin_img)).T
indices

```

```

array([[2, 2],
       [3, 1]])

```

[ Converting this Cirq circuits to tensors for tfq ]

```

$ x_train_tfcirc = tfq.convert_to_tensor(x_train_circ)
  x_test_tfcirc = tfq.convert_to_tensor(x_test_circ)

```

## Quantum Neural Network

### Building the model circuit

```

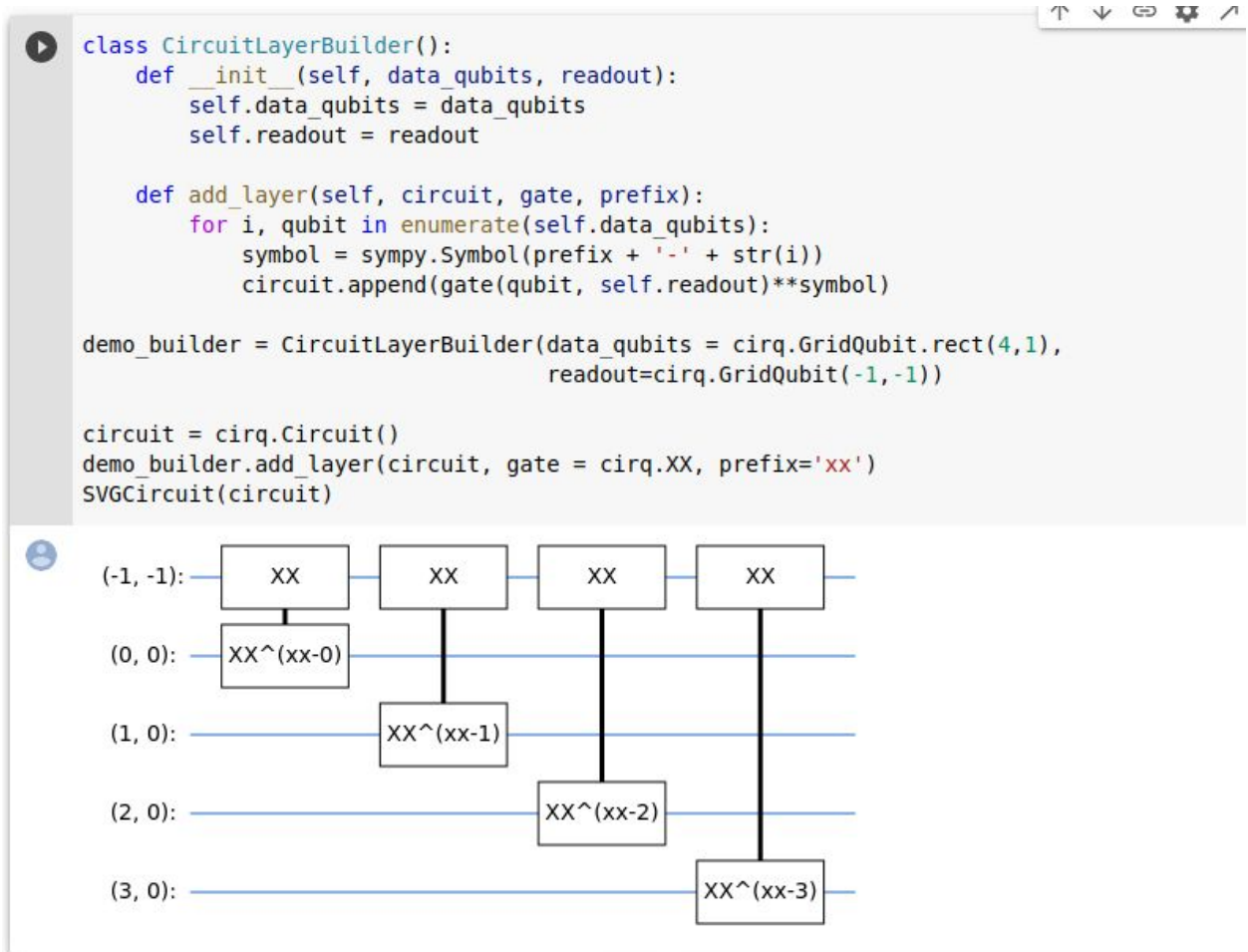
$ class CircuitLayerBuilder():
    def __init__(self, data_qubits, readout):
        self.data_qubits = data_qubits
        self.readout = readout

    def add_layer(self, circuit, gate, prefix):
        for i, qubit in enumerate(self.data_qubits):
            symbol = sympy.Symbol(prefix + '-' + str(i))
            circuit.append(gate(qubit, self.readout)**symbol)

$ demo_builder = CircuitLayerBuilder(data_qubits = cirq.GridQubit.rect(4,1),
                                     readout=cirq.GridQubit(-1,-1))

circuit = cirq.Circuit()
demo_builder.add_layer(circuit, gate = cirq.XX, prefix='xx')
SVGCircuit(circuit)

```



[ Now building a two-layered model ]

```

$ def create_quantum_model():
    """Create a QNN model circuit and readout operation to go along with it."""
    data_qubits = cirq.GridQubit.rect(4, 4) # a 4x4 grid.
    readout = cirq.GridQubit(-1, -1)      # a single qubit at [-1,-1]
    circuit = cirq.Circuit()

    # Prepare the readout qubit.
    circuit.append(cirq.X(readout))
    circuit.append(cirq.H(readout))

    builder = CircuitLayerBuilder(
        data_qubits = data_qubits,
        readout=readout)

```

```
# Then add layers (experiment by adding more).
builder.add_layer(circuit, cirq.XX, "xx1")
builder.add_layer(circuit, cirq.ZZ, "zz1")
```

```
# Finally, prepare the readout qubit.
circuit.append(cirq.H(readout))
return circuit, cirq.Z(readout)
```

```
$    model_circuit, model_readout = create_quantum_model()
```

### Wrapping the model-circuit in a tfq-keras model

```
$    # Build the Keras model.
    model = tf.keras.Sequential([
        # The input is the data-circuit, encoded as a tf.string
        tf.keras.layers.Input(shape=(), dtype=tf.string),
        # The PQC layer returns the expected value of the readout gate,
        range [-1,1].
        tfq.layers.PQC(model_circuit, model_readout),
    ])
```

[ Converting the labels, `y_train_nocon`, from boolean to [-1, 1], as expected by the hinge loss ]

```
$    y_train_hinge = 2.0*y_train_nocon-1.0
    y_test_hinge = 2.0*y_test-1.0
```

[ Use a custom hinge\_accuracy metric that handles [-1, 1] as the `y_true` labels argument ]

```
$    def hinge_accuracy(y_true, y_pred):
        y_true = tf.squeeze(y_true) > 0.0
        y_pred = tf.squeeze(y_pred) > 0.0
        result = tf.cast(y_true == y_pred, tf.float32)
        return tf.reduce_mean(result)
```

```
model.compile(
    loss=tf.keras.losses.Hinge(),
    optimizer=tf.keras.optimizers.Adam(),
    metrics=[hinge_accuracy])
print(model.summary())
```

```
▶ model.compile(
    loss=tf.keras.losses.Hinge(),
    optimizer=tf.keras.optimizers.Adam(),
    metrics=[hinge_accuracy])
print(model.summary())
```

Model: "sequential"

Layer (type)	Output Shape	Param #
pqc (PQC)	(None, 1)	32

Total params: 32  
Trainable params: 32  
Non-trainable params: 0

None

## Train the Quantum Model

Training this model to convergence should achieve >%85 accuracy on the test set. Training process would take about 45 min, but we can use small subset of the data by setting NUM\_EXAMPLE=300.

```
$ EPOCHS = 3
  BATCH_SIZE = 32
  #NUM_EXAMPLES = len(x_train_tfcirc)
  NUM_EXAMPLES = 300
```

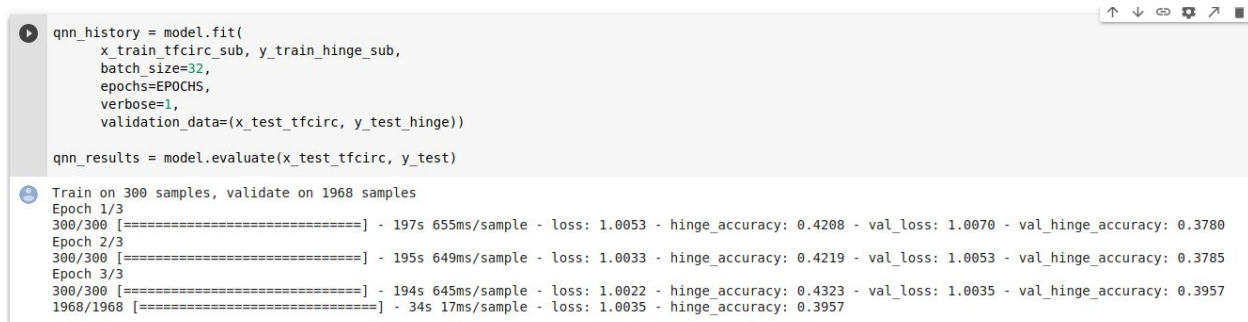
```
x_train_tfcirc_sub = x_train_tfcirc[:NUM_EXAMPLES]
y_train_hinge_sub = y_train_hinge[:NUM_EXAMPLES]
```

```

qnn_history = model.fit(
    x_train_tfcirc_sub, y_train_hinge_sub,
    batch_size=32,
    epochs=EPOCHS,
    verbose=1,
    validation_data=(x_test_tfcirc, y_test_hinge))

qnn_results = model.evaluate(x_test_tfcirc, y_test)

```



```

qnn_history = model.fit(
    x_train_tfcirc_sub, y_train_hinge_sub,
    batch_size=32,
    epochs=EPOCHS,
    verbose=1,
    validation_data=(x_test_tfcirc, y_test_hinge))

qnn_results = model.evaluate(x_test_tfcirc, y_test)

```

Train on 300 samples, validate on 1968 samples

Epoch	Loss	Hinge Accuracy	Val Loss	Val Hinge Accuracy
Epoch 1/3	1.0053	0.4208	1.0070	0.3780
Epoch 2/3	1.0033	0.4219	1.0053	0.3785
Epoch 3/3	1.0022	0.4323	1.0035	0.3957

## Classical Neural Network

**After a single epoch, a classical neural network can achieve >98% accuracy on the holdout set.**

```

$ def create_classical_model():
    # A simple model based off LeNet from https://keras.io/examples/mnist_cnn/
    model = tf.keras.Sequential()
    model.add(tf.keras.layers.Conv2D(32, [3, 3], activation='relu',
input_shape=(28,28,1)))
    model.add(tf.keras.layers.Conv2D(64, [3, 3], activation='relu'))
    model.add(tf.keras.layers.MaxPooling2D(pool_size=(2, 2)))
    model.add(tf.keras.layers.Dropout(0.25))
    model.add(tf.keras.layers.Flatten())
    model.add(tf.keras.layers.Dense(128, activation='relu'))
    model.add(tf.keras.layers.Dropout(0.5))
    model.add(tf.keras.layers.Dense(1))
    return model

model = create_classical_model()

```

```
model.compile(loss = tf.keras.losses.BinaryCrossentropy(from_logits=True),
              optimizer=tf.keras.optimizers.Adam(),
              metrics=['accuracy'])
```

```
model.summary()
```

```
def create_classical_model():
    # A simple model based off LeNet from https://keras.io/examples/mnist_cnn/
    model = tf.keras.Sequential()
    model.add(tf.keras.layers.Conv2D(32, [3, 3], activation='relu', input_shape=(28,28,1)))
    model.add(tf.keras.layers.Conv2D(64, [3, 3], activation='relu'))
    model.add(tf.keras.layers.MaxPooling2D(pool_size=(2, 2)))
    model.add(tf.keras.layers.Dropout(0.25))
    model.add(tf.keras.layers.Flatten())
    model.add(tf.keras.layers.Dense(128, activation='relu'))
    model.add(tf.keras.layers.Dropout(0.5))
    model.add(tf.keras.layers.Dense(1))
    return model
```

```
model = create_classical_model()
model.compile(loss=tf.keras.losses.BinaryCrossentropy(from_logits=True),
              optimizer=tf.keras.optimizers.Adam(),
              metrics=['accuracy'])
```

```
model.summary()
```

Model: "sequential\_1"

Layer (type)	Output Shape	Param #
conv2d (Conv2D)	(None, 26, 26, 32)	320
conv2d_1 (Conv2D)	(None, 24, 24, 64)	18496
max_pooling2d (MaxPooling2D)	(None, 12, 12, 64)	0
dropout (Dropout)	(None, 12, 12, 64)	0
flatten (Flatten)	(None, 9216)	0
dense (Dense)	(None, 128)	1179776
dropout_1 (Dropout)	(None, 128)	0
dense_1 (Dense)	(None, 1)	129
Total params: 1,198,721		
Trainable params: 1,198,721		
Non-trainable params: 0		

\$ model.fit(x\_train,  
y\_train,  
batch\_size=128,  
epochs=1,



```

verbose=1,
validation_data=(x_test, y_test))

cnn_results = model.evaluate(x_test, y_test)

```

```

model.fit(x_train,
          y_train,
          batch_size=128,
          epochs=1,
          verbose=1,
          validation_data=(x_test, y_test))

cnn_results = model.evaluate(x_test, y_test)

```

Train on 12049 samples, validate on 1968 samples  
 12049/12049 [=====] - 30s 2ms/sample - loss: 0.0407 - accuracy: 0.9826 - val\_loss: 0.0037 - val\_accuracy: 0.9985  
 1968/1968 [=====] - 1s 735us/sample - loss: 0.0037 - accuracy: 0.9985

## [Trying 37-parameter model]

```

$ def create_fair_classical_model():
    # A simple model based off LeNet from https://keras.io/examples/mnist_cnn/
    model = tf.keras.Sequential()
    model.add(tf.keras.layers.Flatten(input_shape=(4,4,1)))
    model.add(tf.keras.layers.Dense(2, activation='relu'))
    model.add(tf.keras.layers.Dense(1))
    return model

model = create_fair_classical_model()
model.compile(loss=tf.keras.losses.BinaryCrossentropy(from_logits=True),
              optimizer=tf.keras.optimizers.Adam(),
              metrics=['accuracy'])

model.summary()

```



```

def create_fair_classical_model():
    # A simple model based off LeNet from https://keras.io/examples/mnist\_cnn/
    model = tf.keras.Sequential()
    model.add(tf.keras.layers.Flatten(input_shape=(4,4,1)))
    model.add(tf.keras.layers.Dense(2, activation='relu'))
    model.add(tf.keras.layers.Dense(1))
    return model

model = create_fair_classical_model()
model.compile(loss=tf.keras.losses.BinaryCrossentropy(from_logits=True),
              optimizer=tf.keras.optimizers.Adam(),
              metrics=['accuracy'])

model.summary()

```

Model: "sequential\_2"

Layer (type)	Output Shape	Param #
flatten_1 (Flatten)	(None, 16)	0
dense_2 (Dense)	(None, 2)	34
dense_3 (Dense)	(None, 1)	3
Total params: 37		
Trainable params: 37		
Non-trainable params: 0		

```

$ model.fit(x_train_bin,
            y_train_nocon,
            batch_size=128,
            epochs=20,
            verbose=2,
            validation_data=(x_test_bin, y_test))

```

```

fair_nn_results = model.evaluate(x_test_bin, y_test)

```

```

Train on 11520 samples, validate on 1968 samples
Epoch 1/20
11520/11520 - 0s - loss: 0.6404 - accuracy: 0.5855 - val_loss: 0.6039 - val_accuracy: 0.6042
Epoch 2/20
11520/11520 - 0s - loss: 0.5732 - accuracy: 0.6791 - val_loss: 0.5293 - val_accuracy: 0.7154
Epoch 3/20
11520/11520 - 0s - loss: 0.4939 - accuracy: 0.7614 - val_loss: 0.4554 - val_accuracy: 0.7612
Epoch 4/20
11520/11520 - 0s - loss: 0.4237 - accuracy: 0.8225 - val_loss: 0.3932 - val_accuracy: 0.8435
Epoch 5/20
11520/11520 - 0s - loss: 0.3669 - accuracy: 0.8619 - val_loss: 0.3451 - val_accuracy: 0.8445
Epoch 6/20
11520/11520 - 0s - loss: 0.3238 - accuracy: 0.8639 - val_loss: 0.3094 - val_accuracy: 0.8481
Epoch 7/20
11520/11520 - 0s - loss: 0.2921 - accuracy: 0.8707 - val_loss: 0.2837 - val_accuracy: 0.8577
Epoch 8/20
11520/11520 - 0s - loss: 0.2692 - accuracy: 0.8732 - val_loss: 0.2651 - val_accuracy: 0.8577
Epoch 9/20
11520/11520 - 0s - loss: 0.2529 - accuracy: 0.8757 - val_loss: 0.2522 - val_accuracy: 0.9111
Epoch 10/20
11520/11520 - 0s - loss: 0.2410 - accuracy: 0.9056 - val_loss: 0.2428 - val_accuracy: 0.9126
Epoch 11/20
11520/11520 - 0s - loss: 0.2323 - accuracy: 0.9059 - val_loss: 0.2359 - val_accuracy: 0.9126
Epoch 12/20
11520/11520 - 0s - loss: 0.2258 - accuracy: 0.9059 - val_loss: 0.2308 - val_accuracy: 0.9126
Epoch 13/20
11520/11520 - 0s - loss: 0.2209 - accuracy: 0.9072 - val_loss: 0.2269 - val_accuracy: 0.9131
Epoch 14/20
11520/11520 - 0s - loss: 0.2171 - accuracy: 0.9077 - val_loss: 0.2239 - val_accuracy: 0.9131
Epoch 15/20
11520/11520 - 0s - loss: 0.2142 - accuracy: 0.9093 - val_loss: 0.2219 - val_accuracy: 0.9131
Epoch 16/20
11520/11520 - 0s - loss: 0.2119 - accuracy: 0.9113 - val_loss: 0.2205 - val_accuracy: 0.9141
Epoch 17/20
11520/11520 - 0s - loss: 0.2102 - accuracy: 0.9122 - val_loss: 0.2194 - val_accuracy: 0.9141
Epoch 18/20
11520/11520 - 0s - loss: 0.2089 - accuracy: 0.9122 - val_loss: 0.2183 - val_accuracy: 0.9141
Epoch 19/20
11520/11520 - 0s - loss: 0.2078 - accuracy: 0.9122 - val_loss: 0.2176 - val_accuracy: 0.9141
Epoch 20/20
11520/11520 - 0s - loss: 0.2069 - accuracy: 0.9122 - val_loss: 0.2175 - val_accuracy: 0.9141
1968/1968 [=====] - 0s 164us/sample - loss: 0.2175 - accuracy: 0.9141

```

## Comparison

Higher resolution input and a more powerful model make this problem easy for CNN. While a classical model of similar power (~32 parameters) trains to a similar accuracy in a fraction of the time. **Classical Neural Network easily beats the Quantum Neural Network.**

```

$   qnn_accuracy = qnn_results[1]
    cnn_accuracy = cnn_results[1]
    fair_nn_accuracy = fair_nn_results[1]

```

```
sns.barplot(["Quantum", "Classical, full", "Classical, fair"],  
            [qnn_accuracy, cnn_accuracy, fair_nn_accuracy])
```

```
▶ qnn_accuracy = qnn_results[1]  
   cnn_accuracy = cnn_results[1]  
   fair_nn_accuracy = fair_nn_results[1]  
  
   sns.barplot(["Quantum", "Classical, full", "Classical, fair"],  
               [qnn_accuracy, cnn_accuracy, fair_nn_accuracy])
```

⦿ <matplotlib.axes.\_subplots.AxesSubplot at 0x7f31e0a06320>

