# Exploring Quantum Convolutional Neural Networks with TensorFlow and TensorFlowQuantum

Nasir Ali
CDAC, Noida

## Contents

# 1 Introduction

This project explores the integration of quantum computing principles with convolutional neural networks, utilizing TensorFlow Quantum. The aim is to leverage the quantum mechanical properties to enhance the efficiency and performance of traditional neural network models, particularly in image processing tasks. Quantum Convolutional Neural Networks (QCNN) represent a frontier in combining quantum physics with deep learning, potentially offering significant computational advantages.

# 2 Background

Quantum computing harnesses quantum mechanics to process information, offering parallelism that classical computing cannot achieve. TensorFlow Quantum is an extension of TensorFlow that allows for the construction and training of quantum machine learning models. Convolutional Neural Networks (CNNs) are widely used in image recognition and processing tasks. This project is at the intersection of these technologies, aiming to explore how quantum computing can augment the capabilities of CNNs.

# 3 Methodology

The methodology involves preprocessing the MNIST dataset, a standard benchmark in machine learning for handwritten digit recognition. The images are resized and normalized for the quantum model. The architecture includes a fully connected model, a traditional CNN, and a QCNN, each trained and evaluated on the dataset. TensorFlow Quantum is used to implement the quantum layers in the QCNN.

## 3.1 Quantum Convolutional Neural Network Code

```python
# Importing libraries
import tensorflow as tf
import tensorflow_quantum as tfq
import cirq
import sympy
import numpy as np
from tensorflow.keras import datasets, layers, models
from cirq.contrib.svg import SVGCircuit

# Loading and preprocessing the MNIST dataset
(x_train, y_train), (x_test, y_test) = tf.keras.datasets.mnist.load_data()
x_train, x_test = x_train[..., np.newaxis]/255.0, x_test[...,
np.newaxis]/255.0
x_train = tf.image.resize(x_train[:], (10,10)).numpy()
x_test = tf.image.resize(x_test[:], (10,10)).numpy()

```

```
# Defining the fully connected model
width = np.shape(x_train)[1]
height = np.shape(x_train)[2]

fc_model = models.Sequential()

fc_model.add(layers.Flatten(input_shape=(width,height,1)))
fc_model.add(layers.Dense(32, activation='relu'))
fc_model.add(layers.Dense(10, activation='softmax'))

fc_model.compile(optimizer='adam',
loss='sparse_categorical_crossentropy',
metrics=['accuracy'])

fc_history = fc_model.fit(x_train, y_train, steps_per_epoch=500,
validation_data=(x_test, y_test),
epochs=50, batch_size=5)

# Defining the CNN model
width = np.shape(x_train)[1]
height = np.shape(x_train)[2]

cnn_model = models.Sequential()

cnn_model.add(layers.Conv2D(8, (2, 2), activation='relu', input_shape=(width, height, 1)))
cnn_model.add(layers.Flatten())
cnn_model.add(layers.Dense(32, activation='relu'))
cnn_model.add(layers.Dense(10, activation='softmax'))



cnn_model.compile(optimizer=tf.optimizers.Adam(),
loss='sparse_categorical_crossentropy',
metrics=['accuracy'])

cnn_history = cnn_model.fit(x_train, y_train, steps_per_epoch=500,
validation_data=(x_test, y_test),
epochs=10, batch_size=5)



# Defining the QCNN model
class QConv(tf.keras.layers.Layer):
def __init__(self, filter_size, depth, activation=None, name=None, kernel_regularizer=None, **kwangs):
super(QConv, self).__init__(name=name, **kwangs)
self.filter_size = filter_size
self.depth = depth
self.learning_params = []
self.QCNN_layer_gen()
# self.circuit_tensor = tfq.convert_to_tensor([self.circuit])
self.activation = tf.keras.layers.Activation(activation)
self.kernel_regularizer = kernel_regularizer

```

```
def _next_qubit_set(self, original_size, next_size, qubits):
step = original_size // next_size
qubit_list = []
for i in range(0, original_size, step):
for j in range(0, original_size, step):
qubit_list.append(qubits[original_size*i + j])
return qubit_list

def _get_new_param(self):
"""
return new learnable parameter
all returned parameter saved in self.learning_params
"""
new_param = sympy.symbols("p"+str(len(self.learning_params)))
self.learning_params.append(new_param)
return new_param

def _QConv(self, step, target, qubits):
"""
apply learnable gates each quantum convolutional layer level
"""
yield cirq.CZPowGate(exponent=self._get_new_param())(qubits[target], qubits[target+step])
yield cirq.CXPowGate(exponent=self._get_new_param())(qubits[target], qubits[target+step])

def QCNN_layer_gen(self):
"""
make quantum convolutional layer in QConv layer
"""
pixels = self.filter_size**2
# filter size: 2^n only for this version!
if np.log2(pixels) % 1 != 0:
raise NotImplementedError("filter size: 2^n only available")
cirq_qubits = cirq.GridQubit.rect(self.filter_size, self.filter_size)
# mapping input data to circuit
input_circuit = cirq.Circuit()
input_params = [sympy.symbols('a%d' %i) for i in range(pixels)]
for i, qubit in enumerate(cirq_qubits):
input_circuit.append(cirq.rx(np.pi*input_params[i])(qubit))
# apply learnable gate set to QCNN circuit
QCNN_circuit = cirq.Circuit()
step_size = [2**i for i in range(np.log2(pixels).astype(np.int32))]
for step in step_size:
for target in range(0, pixels, 2*step):
QCNN_circuit.append(self._QConv(step, target, cirq_qubits))
# merge the circuits
full_circuit = cirq.Circuit()
full_circuit.append(input_circuit)
full_circuit.append(QCNN_circuit)
self.circuit = full_circuit # save circuit to the QCNN layer obj.
self.params = input_params + self.learning_params
self.op = cirq.Z(cirq_qubits[0])
```

```
def build(self, input_shape):
self.width = input_shape[1]
self.height = input_shape[2]
self.channel = input_shape[3]
self.num_x = self.width - self.filter_size + 1
self.num_y = self.height - self.filter_size + 1

self.kernel = self.add_weight(name="kenel",
shape=[self.depth,
self.channel,
len(self.learning_params)],
initializer=tf.keras.initializers.glorot_normal(),
regularizer=self.kernel_regularizer)
self.circuit_tensor = tfq.convert_to_tensor([self.circuit] * self.num_x * self.num_y * self.channel)

def call(self, inputs):
# input shape: [N, width, height, channel]
# slide and collect data
stack_set = None
for i in range(self.num_x):
for j in range(self.num_y):
slice_part = tf.slice(inputs, [0, i, j, 0], [-1, self.filter_size, self.filter_size, -1])
slice_part = tf.reshape(slice_part, shape=[-1, 1, self.filter_size, self.filter_size, self.channel])
if stack_set == None:
stack_set = slice_part
else:
stack_set = tf.concat([stack_set, slice_part], 1)
# -> shape: [N, num_x*num_y, filter_size, filter_size, channel]
stack_set = tf.transpose(stack_set, perm=[0, 1, 4, 2, 3])
# -> shape: [N, num_x*num_y, channel, filter_size, fiter_size]
stack_set = tf.reshape(stack_set, shape=[-1, self.filter_size**2])
# -> shape: [N*num_x*num_y*channel, filter_size^2]

# total input citcuits: N * num_x * num_y * channel
circuit_inputs = tf.tile([self.circuit_tensor], [tf.shape(inputs)[0], 1])
circuit_inputs = tf.reshape(circuit_inputs, shape=[-1])
tf.fill([tf.shape(inputs)[0]*self.num_x*self.num_y, 1], 1)
outputs = []
for i in range(self.depth):
controller = tf.tile(self.kernel[i], [tf.shape(inputs)[0]*self.num_x*self.num_y, 1])
outputs.append(self.single_depth_QCNN(stack_set, controller, circuit_inputs))
# shape: [N, num_x, num_y]

output_tensor = tf.stack(outputs, axis=3)
output_tensor = tf.math.acos(tf.clip_by_value(output_tensor, -1+1e-5, 1-1e-5)) / np.pi
# output_tensor = tf.clip_by_value(tf.math.acos(output_tensor)/np.pi, -1, 1)
return self.activation(output_tensor)
```

```
    def single_depth_QCNN(self, input_data, controller, circuit_inputs):
    """
    make QCNN for 1 channel only
    """
    # input shape: [N*num_x*num_y*channel, filter_size^2]
    # controller shape: [N*num_x*num_y*channel, len(learning_params)]
    input_data = tf.concat([input_data, controller], 1)
    # input_data shape: [N*num_x*num_y*channel, len(learning_params)]
    QCNN_output = tfq.layers.Expectation()(circuit_inputs,
    symbol_names=self.params,
    symbol_values=input_data,
    operators=self.op)
    # QCNN_output shape: [N*num_x*num_y*channel]
    QCNN_output = tf.reshape(QCNN_output, shape=[-1, self.num_x, self.num_y, self.channel])
    return tf.math.reduce_sum(QCNN_output, 3)



    width = np.shape(x_train)[1]
    height = np.shape(x_train)[2]

    qcnn_model = models.Sequential()


    qcnn_model.add(QConv(filter_size=2, depth=8, activation='relu',
    name='qconv1', input_shape=(width, height, 1)))
    #model.add(layers.Conv2D(16, (2, 2), activation='relu'))
    qcnn_model.add(layers.Flatten())
    qcnn_model.add(layers.Dense(32, activation='relu'))
    qcnn_model.add(layers.Dense(10, activation='softmax'))
```

## 3.2 Analysis of Code

The implementation begins with importing necessary libraries, including TensorFlow for machine learning models, TensorFlow Quantum for integrating quantum computing with TensorFlow, and Cirq for quantum circuit simulation. The MNIST dataset, a standard dataset for handwritten digit recognition, is loaded and preprocessed by resizing and normalizing the images to fit the quantum model's requirements.

The code defines three types of models. The first is a fully connected (dense) neural network, which serves as a baseline for comparison. The second is a conventional Convolutional Neural Network (CNN), well-known for its effectiveness in image processing tasks.

The most notable part of the implementation is the Quantum Convolutional Neural Network (QCNN). It involves creating a custom quantum convolutional layer ('QConv') using TensorFlow Quantum and Cirq. This layer is designed to apply quantum gates and measurements to process the image data. The quantum convolutional layer leverages quantum mechanical properties to enhance the model's capability to capture complex patterns in the data.

The QCNN model's architecture includes this ‘QConv‘ layer followed by flattening and dense layers, resembling the structure of traditional CNNs but with a quantum computing twist. This unique integration showcases the potential of quantum computing in enhancing classical deep learning models, particularly in processing and analyzing visual data.

# 4 Results

## 4.1 Training and Validation Curves

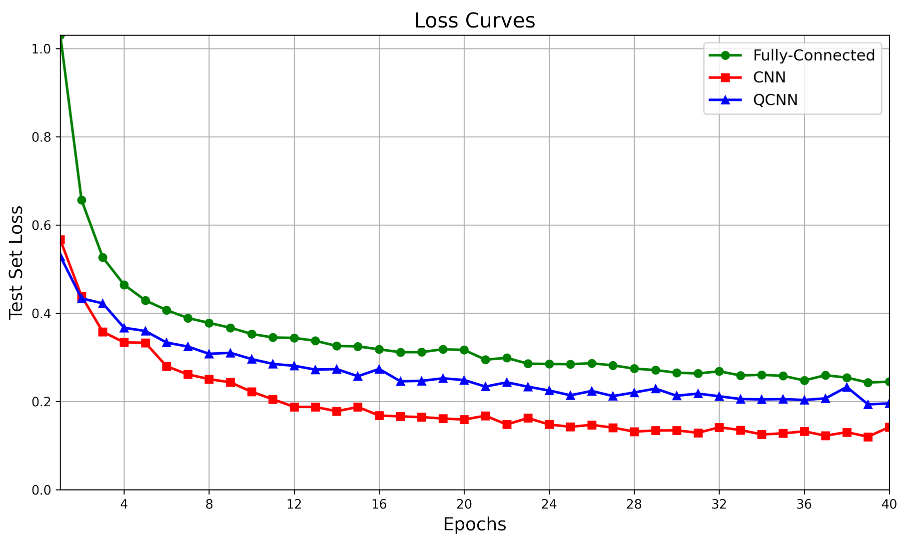


Figure 1: Training and Validation Curves

### 4.2 Performance Comparison

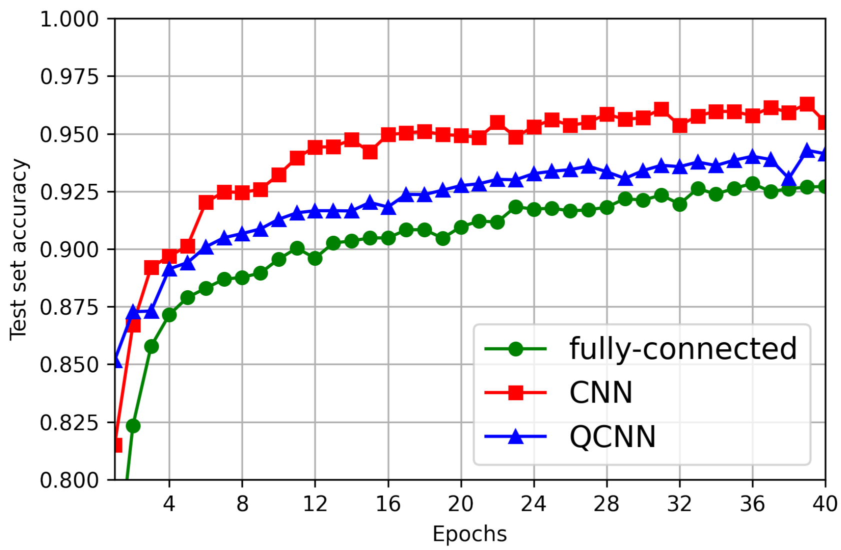


Figure 2: Performance Comparison

The results show how the QCNN model compares with traditional models in terms of accuracy and loss over epochs. The training and validation curves indicate the learning efficiency of each model. The performance comparison highlights the advantages or limitations of using quantum layers in neural networks.

# 5 Discussion

The QCNN model exhibits unique characteristics due to its quantum nature, which are reflected in the training and performance metrics. These results are indicative of how quantum computing principles can impact machine learning, specifically in areas requiring complex pattern recognition like image processing.

# 6 Conclusion

This project demonstrates a foundational exploration of Quantum Convolutional Neural Networks using TensorFlow Quantum. The results provide insights into the potential synergies between quantum computing and deep learning, paving the way for further research in this promising field.

# 7 References

## References


[1] Simple QCNN MNIST Implementation. *GitHub*. Available at: https://github.com/Menborong/Simple-QCNN/blob/master/Simple_QCNN_MNIST.ipynb.

[2] Cong, I., Choi, S. & Lukin, M.D. (2019). *Quantum convolutional neural networks*. Nature Physics, 15, 1273–1278. Available at: https://doi.org/10.1038/s41567-019-0648-8.

[3] Convolutional Neural Networks. *IBM*. Available at: https://www.ibm.com/cloud/learn/convolutional-neural-networks.