Branching Quantum Convolutional Neural Networks

Amir Barkam, Arnold Ying, Rain Zhang, Stella Wang

CPEN 400Q Final Project Report

Contents

1 Introduction

Neural networks are machine learning algorithms that allow us to teach computers to recognize complex patterns from large data sets. With recent developments in quantum computing, we are able to process and learn from quantum data sets such as Hamiltonians. Our project's paper focuses on the Branching Quantum Convolutional Neural Network, which is built on top of the Quantum Convolutional Neural Network, which is then built upon the classical Convolutional Neural Network. The below sections briefly explain the three aforementioned Neural Network architectures.

1.1 Convolutional Neural Network

A Convolutional Neural Network (CNN) is a specific type of neural network that is generally used for image recognition and feature detection. CNNs have the capability to extract spatial information by applying *filters* over a given input (typically an image). Typical internal structure for CNN's includes alternating convolution and pooling layers followed by a final fully connected layer.

- Convolution layers extract features from input data
- Pooling layers reduce dimensions to down sample data
- Fully connected layer are used to output a classification label for the given task

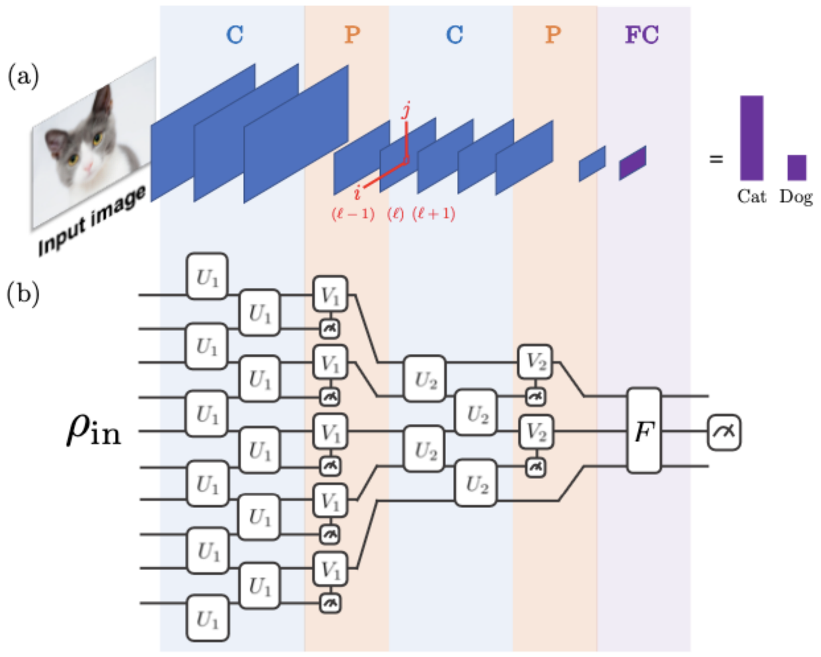

Figure 1: Comparison of (a) CNN and (b) QCNN architectures [4]

1.2 Quantum Convolutional Neural Network

Quantum Convolutional Neural Networks (QCNNs) are the quantum equivalent of traditional CNNs. They have a similar structure to traditional CNNs in that they have alternating convolution and pooling layers, followed by a final fully connected layer. QCNNs replace convolution layers in traditional CNNs with entangling gates that act on neighboring pairs of qubits. The pooling layers in QCNNs consist of single-qubit controlled rotations on a subset of the qubits in the layer. The remaining qubits act as the controls for the rotations on neighboring qubits. These qubits are then discarded to reduce dimensionality, similar to pooling layers in a traditional CNN. There is a final fully connected layer at the end of QCNNs that consists of a final measurement on the last remaining qubit to determine the label for the classification. Unlike classical CNNs that only have trainable parameters in the convolution layers, QCNNs have trainable parameters in both convolution and pooling layers. Each two-qubit entangling gate in the convolution layers has 15 trainable parameters, and each one-qubit controlled rotation in the pooling layers has 3 trainable parameters.

1.3 Branching Quantum Convolutional Neural Networks

Branching Quantum Convolutional Neural Networks (bQCNNs), the topic of this report, are very similar to QCNNs as they both have convolution layers with multi-qubit entangling gates, each followed by a pooling layer. The key difference between the two architectures is in their pooling layers. In QCNNs, some of the input qubits act as controls for rotations on their neighboring qubits, and are then discarded. In bQCNNs, mid-circuit measurements are performed on these qubits to determine which branch of the circuit should be taken, and they are then discarded. The qubits that are not measured in a bQCNN pooling layer have controlled rotations applied to them, similar to those in a QCNN. Each branch of the bQCNN circuit consists of a sequence of convolution and pooling layers with a final fully connected layer. At a bQCNN pooling layer, if n of the input qubits are measured, the number of branches that can be taken after that layer can be represented by 2^n. The layers in each branch contain their own set of trainable parameters, thus increasing the density of trainable parameters relative to circuit depth. For a QCNN with a 4-qubit input, where 2/3 of the qubits are discarded at each pooling layer, the total number of trainable parameters is 66. For a bQCNN with the same input size and circuit depth, there are 111 trainable parameters.

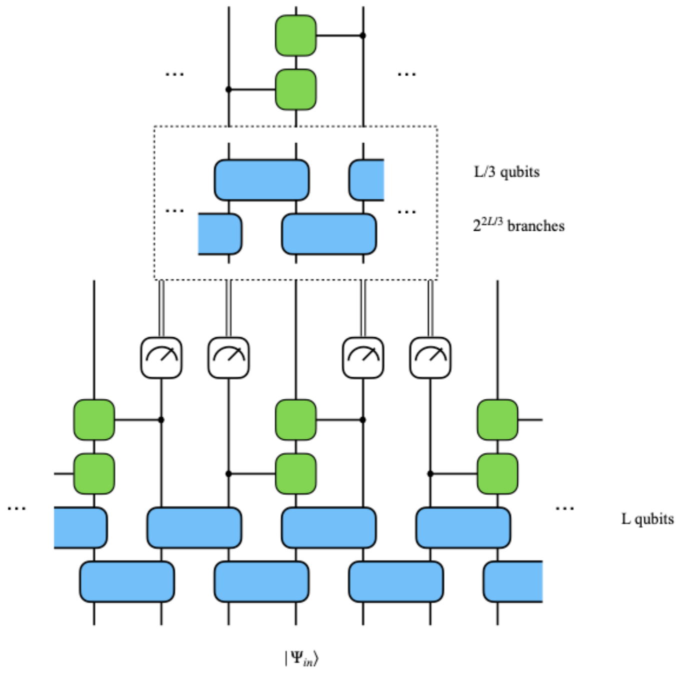

Figure 2: bQCNN architecture with convolution layers (blue) and pooling layers (green) [5]

2 Theory

The purpose of this report is to investigate whether bQCNNs can perform better than QCNNs on the same data set. The key difference between the two architectures is that due to the availability of branches after each pooling layer, bQCNNs have a greater number of trainable parameters than QCNNs with the same circuit depth and input size. With a greater number of trainable parameters, bQCNNs have greater expressibility. Expressibility is a measure of the extent to which a neural network can reach large volumes of the Hilbert space. Greater expressibility allows for the modeling of more complex functions. bQCNNs are able to achieve this increase in trainable parameters without increasing circuit depth, which is very important since increased circuit depth results in greater gate noise. In any bQCNN, the branch that is taken after each pooling layer can be the same as the circuit path taken in a regular QCNN. Therefore, the bQCNN should always be able to perform at least as well as an equivalent QCNN. With the addition of branches, the bQCNN also has the potential to improve upon a QCNN's performance by selecting a different branch with parameters that are better suited to model the function in the problem.

Another important distinction between the two architectures is that in the pooling layers, bQCNNs make use of a global view of the data to determine which branch of the circuit to take. QCNNs, on the other hand, only perform controlled rotations based on adjacent qubits, thus the rotation for each qubit only depends on its immediate neighbor. With the ability to make use of global information at each pooling layer, bQCNNs have greater potential to perform accurate predictions on data sets where correlations between data are more spaced out.

3 Results

3.1 Objective

Show that bQCNN has higher expressibility and correctness than QCNN using two different tasks.

3.2 Tasks

3.2.1 Random excitation to cluster state

One of the two training tasks that we have decided to conduct to test the effectiveness of bQCNN over QCNN is random excitation to cluster state used by the TensorFlow Quantum QCNN tutorial [3]. The input data to this training task is random parameterized PauliX rotations applied to all 4 input Qubits. And the label for a particular data point is 1 if the magnitude of the rotations applied to the wire is $>= \frac{\pi}{2}$ and 0 otherwise (i.e. $< \frac{\pi}{2}$). The input qubits are then passed through a cluster state circuit, as defined in the TensorFlow Quantum QCNN tutorial [3]. Then finally the Qubits are applied either the bQCNN or QCNN circuit for training. The results we obtained for this task is shown in Figure 3 and 4

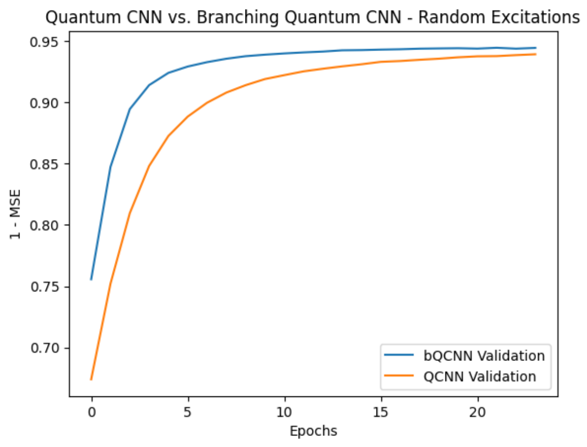

Figure 3: bQCNN vs QCNN loss in Random Excitation Task with MSE loss

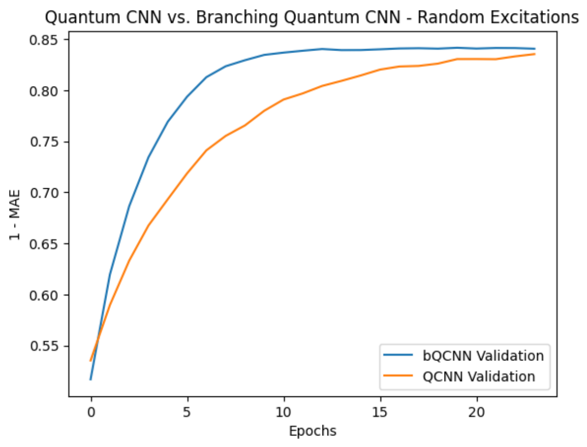

Figure 4: bQCNN vs QCNN loss in Random Excitation Task with MAE loss

For MSE loss function (Figure 3), both bQCNN and QCNN appears to reach an overall maximum correctness (1-MSE) of 0.94. However the bQCNN loss function curve appears to have a noticeably faster and steeper learning curve than the QCNN.

For MAE, loss function (Figure 4), both bQCNN and QCNN appears to reach an overall maximum correctness (1-MAE) of 0.835. Though the loss function curves appears to be even more steep for the bQCNN compared to QCNN

It is noted that most likely due to the simplicity of the training task, we are not able to fully capture the effects of the increase in expressibility of the bQCNN. An increase in expressibility of the bQCNN would mean an overall higher maximum correctness of bQCNN over QCNN however we are seeing the same results here.

3.2.2 Image classification

A common use of CNNs, in general, is image classification. The CNN extract particular features and patterns of the pixels in an image using kernels in the convolution layer. To compare the performance between QCNNs and bQCNNs in image classification, we used a script that generates a 2x2 pixel image from a Qiskit QCNN tutorial [2]. The patterns we will train the QCNN and bQCNN to distinguish is a horizontal or vertical line, which is randomly placed anywhere in the image, alongside a noisy background. Figure 7 shows some sample training data for this task.

The results we obtained for this task is shown in Figure 5 and 6

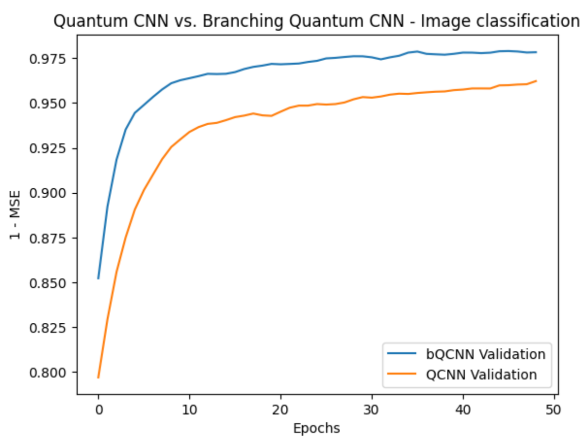

Figure 5: bQCNN vs QCNN loss in Image Classification with MSE loss

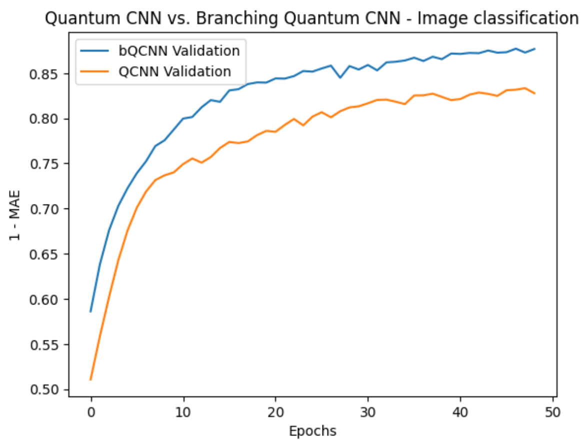

Figure 6: bQCNN vs QCNN loss in Image Classification with MAE loss

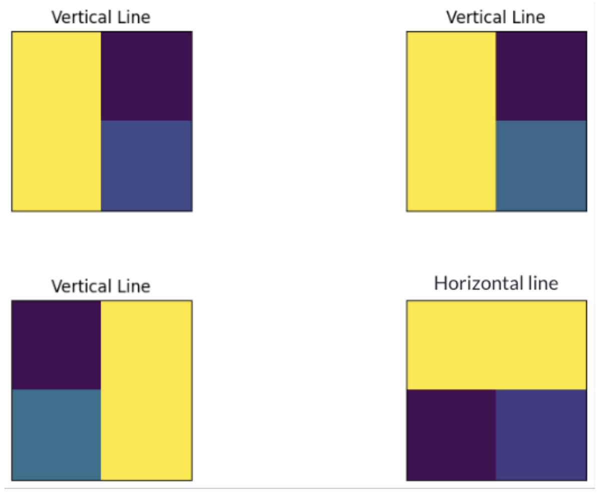

Figure 7: 2x2 pixels generated to have "horizontal" or "vertical" lines

For MSE loss function (Figure 5), bQCNN appears to reach a better overall maximum correctness (1-MSE) of 0.975 whereas QCNN appears to reach an overall maximum correctness of 0.950. Furthermore the bQCNN loss function curve seems to have a slightly steeper learning curve than the QCNNs.

For MAE, loss function (Figure 6), bQCNN continues to see a better performance than QCNN reaching a better overall maximum correctness (1-MAE) of 0.87 and QCNN reaching an overall maximum correctness of 0.82.

4 Design

4.1 Framework

Three quantum machine learning frameworks were considered to implement a bQCNN circuit: Tensorflow Quantum, Qiskit, and Pennylane. Table 1 compares the requirements and usability of each framework.

	TensorFlow Quantum	Qiskit	PennyLane
Branching support	✓	✗	✓
Relevant Docs	✓	✓	✗
Ease of Use	4	3	2
Visualization	3	5	3
Familiarity	4	2	5

Table 1: Comparison between different Quantum ML Frameworks (ratings are between 1-5 with 5 being the best)

4.1.1 TensorFlow Quantum

TensorFlow Quantum has well-detailed documentations and tutorials for QCNN's which served as a good reference for setting up the structure for our bQCNN circuit. [3]. It also has well-defined functionalities and straightforward model declarations, providing ease of use.

4.1.2 Qiskit

Qiskit, a quantum computing framework built by IBM, is capable of performing mid-circuit measurements which is vital in the branching layer of the bQCNN [1]. However, the article only demonstrates the capability of Qiskit to perform mid-circuit measurements on *IBM Quantum backends*, not in simulation.

Following Qiskit's documentation for QCNNs, we were able to reproduce a QCNN circuit and construct a bQCNN circuit but were met with errors during training for the bQCNN circuit [1]. Qiskit's 'circuit.decompose.draw()' is also helpful for visualizing and understanding the underlying QCNN and bQCNN circuits.

4.1.3 Pennylane

PennyLane was considered due to familiarity, however, it was not selected due to lack of relevant documentation and lack of support for Machine Learning functionalities. For example, Pennylane does not loss functions or training functions readily available.

4.1.4 Conclusion

Tensorflow Quantum was selected to implement our bQCNN circuit. All results in Section 3 are based on bQCNN and QCNN circuit implementation in Tensowflow Quantum.

Additionally, we experimented with Qiskit with varying successes. We were able to successfully train and validate a QCNN but were unable to train a bQCNN in Qiskit due to an error with classical bits in the quantum circuit.

4.2 Software Implementation

4.2.1 Number of Qubits

The paper worked with bQCNN circuits with either 4 input qubits or 8 input qubits. A 4-qubits bQCNN circuit has 111 trainable parameters while an 8-qubit bQCNN circuit scales to 914 trainable parameters. We chose to implement a 4 input qubit system due to limitations in time and resources. A smaller network means a faster training and testing time which would result in faster iterations.

[1]Refer to bQCNN-Quiskit.ipynb jupyter notebook for the detailed error message on training a bQCNN model

4.2.2 Loss Function

To measure correctness, we compared two potential loss functions:

1. Mean Squared Error (MSE)
2. Mean Absolute Error (MAE)

MSE is a popularly used loss function for machine learning. We were able to achieve a good loss of 0.945 (1 - MSE) for our random excitation task. However, we were only able to achieve a loss of 0.83 (1 - MAE) for the same task using MAE. The reason that MAE was included was because the bQCNN paper used MAE for its loss function so we wanted to compare our results with theirs to further prove their claim.

Ultimately, which is better depends on your project goal. If you want to train a model which focuses on reducing large outlier errors then MSE is the better choice, whereas if this isn't important and you would prefer greater interpretability then MAE would be better.

MAE is the average absolute distance between the real data and the predicted data, but fails to punish large errors in prediction. MSE measures the average squared difference between the estimated values and the actual value

4.2.3 Optimization Technique

In the paper, a genetic algorithm was used to train and update the bQCNN parameters, but we implemented gradient descent to update parameters due to limitations in scope and the great support for gradient descent optimizer in TensorFlow Quantum.

An attempt was made in trying to implement genetic algorithm using the PyGAD framework [2]. However adoption of it was unsuccessful and was not included in our final software implementation.

[2]PyGAD is a Python Genetic Algorithm library that supports Keras (TensorFlow) and PyTorch, the two most popular python machine learning libraries

5 Reproducibility

5.1 Software implementation

There were no issues reproducing the software implementation of a bQCNN circuit. The paper explained clearly the number of input qubits, trainable parameters, the decomposition of rotation gates in the circuit, and the branching setup.

5.2 Training tasks

5.2.1 Randomly generated data from inverse bQCNN circuit

The first task the authors used to demonstrate the performance of bQCNN is using training data generated by randomizing parameters and running a bQCNN circuit in reverse. However, the paper does not specifically outline how "running a bQCNN in reverse" can be implemented. We were not able to reproduce generating this training data, possibly due to varying software choices (ie, framework, circuit setup, etc) from the authors' implementation, as they did not specify any details regarding their software implementation or choices. Not feasible in Tensorflow Quantum and not working in Quiskit

5.2.2 Backwards QCNN

The authors also compared the results of generating random data from an inverse QCNN circuit. For the same reasons as above, this was not feasible with our choice of software library.

5.2.3 Detecting an SPT phase transition

The training task that the authors used to compare the expressibility of bQCNN vs QCNN was to recognize phase transitions. The authors left out a few crucial details necessary for recreating training data for this task. They explained that the ground states of the cluster Hamiltonian are used as training data and how to take the expectation value to label SPT phases. For this task, we were able to reproduce the Hamiltonian training data and the string order parameter for which the expectation value is the output label. However, potentially due to differences in design and framework choices again, we were not able to integrate this into our bQCNN implementation to train and measure performance.

5.3 Model Optimization Technique

They used a genetic algorithm to train 500 generations of a bQCNN, whereas we only used gradient descent with Adam optimizer provided by TensorFlow. This could be a contributing factor to the differences in our results and the inability to reproduce good results to mimic theirs.

5.4 Hyperparameter Tuning

Some key hyper-parameters such as: learning rate, number of epochs and batch size were not specified so we manually selected these hyper-parameters to achieve the best results.

References

[1] Mid-circuit measurement. https://quantum-computing.ibm.com/lab/docs/iql/manage/systems/midcircuit-measurement/, 2021.

[2] The quantum convolution neural network. https://qiskit.org/ecosystem/machine-learning/tutorials/11_quantum_convolutional_neural_networks.html, 2023.

[3] Quantum convolutional neural network. https://www.tensorflow.org/quantum/tutorials/qcnn, 03 2023.

[4] Iris Cong, Soonwon Choi, and Mikhail D. Lukin. Quantum convolutional neural networks. *Nature Physics*, 15(12):1273–1278, aug 2019.

[5] Ian MacCormack, Conor Delaney, Alexey Galda, Nidhi Aggarwal, and Prineha Narang. Branching quantum convolutional neural networks, 2020.