QFilter

CCH

Febrary 2021

0.1 Introduction

The emerging field of hybrid quantum-classical algorithms joins CPUs and QPUs to speed up specific calculations within a classical algorithm. This allows for shorter quantum executions that are less susceptible to the cumulative effects of noise and that run well on today's devices. This is why we intend to explore the performance of a hybrid convolutional neural network model that incorporates a trainable quantum layer, effectively replacing a convolutional filter, in both quantum simulators and QPU.

Our team proposes to design a trainable quantum convolutional filter in a quantum-classical hybrid neural network, appealing for the NISQ era, inspired by these papers: Hybrid quantum-classical Convolutional Neural Networks [1] and Quanvolutional Neural Networks [2], but generalizing these previous works to use cloud based QPU.

0.2 CNN

Convolutional neural networks (CNN) are used mainly to treat problems with a large number of images characteristic of Deep Learning. Regarding its operation, it could be divided into two stages. The first one will be in charge of passing the image through some filters to create new images that facilitate understanding the network. After this, we will give the previous phase's output through a full-connected neural network capable of learning thanks to a cost function. Finally, it is worth mentioning that both the layers of the neural network and the filters are parameterized, so it is expected that, with the training process, they will be updated towards the desired values.

0.2.1 Convolutional filters

Focusing on the first of the phases described, we will talk about convolutional filters. The process followed to transform the image with these follows a simple process. Initially, a filter is defined as an $n \times n$ matrix, then a window of the same dimension will go through the image performing the operation shown in figure (1).

As the filter is moved, the solution image's size will be reduced compared to the initial input. There are padding techniques with which it could be possible to make the final image retain its size, but we have chosen not to carry out this process to reduce the number of parameters. In this way, given an image of dimension $l \times w$ and a filter of $n \times n$, we will obtain an output of $[\frac{l}{n}] \times [\frac{w}{n}]$.

0.3 Our model

This project will create a convolutional network to classify the MNIST dataset (set of images with handwritten digits 0 to 9. We will work with a convolutional layer in which we will apply 4 filters, and later we will connect it to a neural layer whose output will have dimension 10, one for each digit. The outcome represents each class's probability, and we will say that an image belongs to the class whose probability is more outstanding. In these cases in which we want to approximate a probability, the crossed entropy is applied to calculate the defined error E as follows:

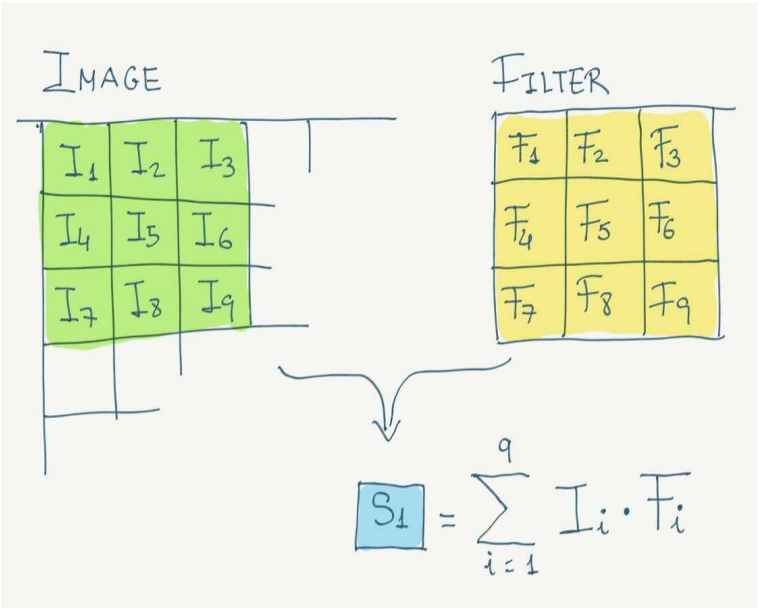


Figure 1: Suppose we take $n = 3$, that is, the filter will have a total of 9 elements. The result after performing the operation is to obtain $\sum_{i=0}^{9} I_i F_i$

$$E = \frac{-1}{M} \sum_{i=1}^{M} y_i log(\hat{y}_i) \tag{1}$$

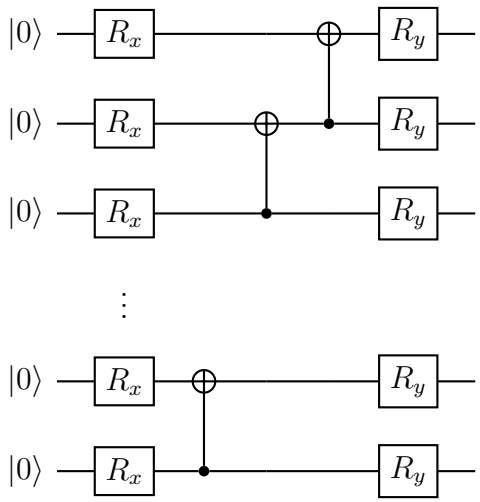
Where M is the number of classes, $\hat{y}_i$ is the probability obtained from the class i, and $y_i = 1$ if the label is i or 0 otherwise. Up to this point, the process followed could be interpreted as a classical development applying convolutional networks. However, we have decided to carry out a quantum approximation, replacing, in this case, the classical filters with quantum procedures. The birth of this idea arises when representing the elements of the image and the filter as vectors:

$$I := \begin{pmatrix} I_1 \\ I_2 \\ ... \\ I_n \end{pmatrix} \qquad F := \begin{pmatrix} F_1 \\ F_2 \\ ... \\ F_n \end{pmatrix} \tag{2}$$

In this way, it is easy to realize that the operation carried out is nothing more than the scalar product of said vectors. Therefore, we could denote it according to Dirac's notation as $\langle I|F\rangle$. Looking at it this way, you begin to get a sense of the idea behind filter quantization.

0.3.1 Quantum filters

Let's define a quantum filter as that filter that executes the dot product $\langle I|F\rangle$ in a quantum circuit. To transfer this process to the circuit, we must first encode input I. By definition, all its elements are numbers between 0 and 1 (grayscale), so the most logical embedding is to encode said values at angles with gate R_Y. In this way, the filter size will determine the number of qubits in the circuit, requiring n^2 qubits. We can then represent the input as $R_Y(I)\,|0\rangle^n$. Regarding the filter, we say to use an ansatz determined by n^2 parameters:



For simplicity, we will denote this ansatz as $F(\theta)\,|0\rangle^n$. Having all this notation, what we want is to obtain the dot product, that is $\langle 0^n | I^\dagger F(\theta) | 0^n \rangle$. To calculate this dot product, we have constructed the circuit $\left| I^\dagger F(\theta) | 0^n \right\rangle$ and obtained the probability of obtaining $|0\rangle^n$. One of our code's strengths is having managed to abstract a function that performs the entire training procedure's scalar product; thus, we can easily choose with which product we want to execute the whole process.

0.3.2 Implementation

To carry out the project, we have taken advantage of the fact that Pennylane provides an interface for Tensorflow and, in this way, use functions already created efficiently. Thanks to this, we have built a hybrid training flow in which we use two different optimizers: SGD to update the classical parameters and Adadelta for the quantum ones. We have made this distinction since, during the experimentation, we observed that the gradient of the quantum parameters was imposed on the classics without letting that part learn; however, we guarantee double training during the experimentation. Once the model is built, the first step is to see that our model is capable of learning and generalizing. To do this, we take 50 images from our dataset (MNIST) and train it for 30 epochs obtaining the following results:

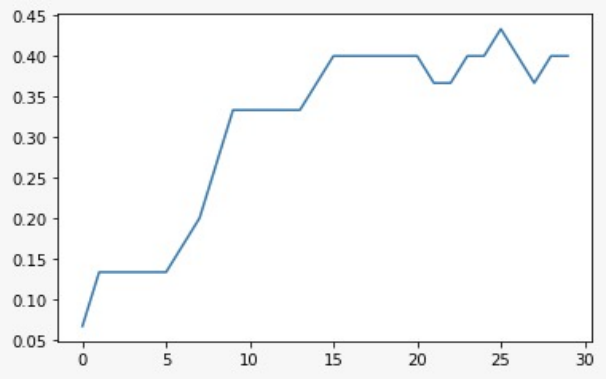


Figure 2: Here we can see the precision of the model depending on the epochs.

In this case, starting at the 15 epoch, the model begins to have a precision of 40%. It may seem like a bad result on the surface, but let's analyze it. As we said before, we have used 50 images, and we are trying to divide between a set of 10 different classes; that is; we are using no more

than five images for each class. Despite this, we have run this circuit with a traditional model reaching similar limits as a check.

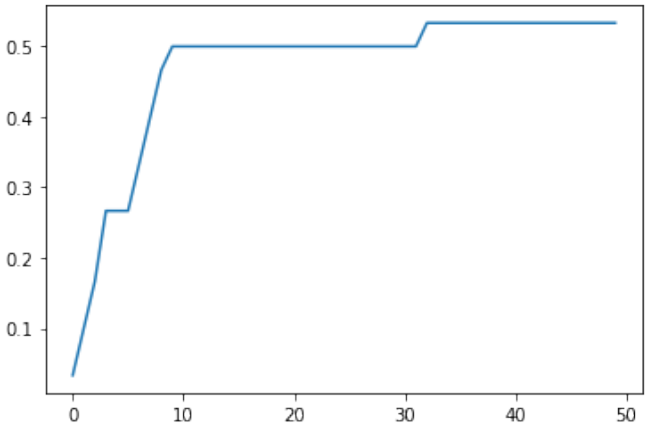


Figure 3: you can observe the filter learning phases.

On the other hand, when implementing the model for the first time, with the filters' size 2×2, we faced difficulty with the number of operations (dot products). To understand this, let's calculate said number of operations (S) in a generic way for p image of $l \times l$ and m filters of $n \times n$ during t epochs.

$$S = [\frac{l}{n}]^2 mpt \tag{3}$$

Therefore, in our initial case of 30 times 4 filters of size 2 and with 50 images of $28\times$ 28, we performed a total of $1,176,000$ operations. We decided to increase the filter size to 4 to solve this problem, reducing $294,000$. The reduction in the number of operations is significant, and really what we are doing is transferring the computational load of the classical part to the quantum scalar product because by increasing the size of the filter, we raise the size of the circuit from 4 qubits to 16. We were looking for this since we can easily enhance this gradient calculation with the parallelism offered by Amazon Braket with Pennylane API [3].

0.4 Results and Discussions

Based on the experiments we did, we found it essential to define the following guidelines to achieve the code's adequate scalability.

1. Spin up larger jupyter notebook instances, for example the type ml.c5.2xlarge (8 vCPU, 16 GiB Memory) to speed up CPU optimization and quantum simulation time.

2. Compare the computation time of remote/local simulator

3. Increase the number of qubits (filters of size 4x4 and 6x6)

4. Batch parallelization of the quantum circuits at the gradient and convolution translation operation levels.

Results on a local simulator (for example Amazon Braket Local simulator or Pennylane's default) are a bit discouraging in terms of running time. When increasing the filter window size, we are

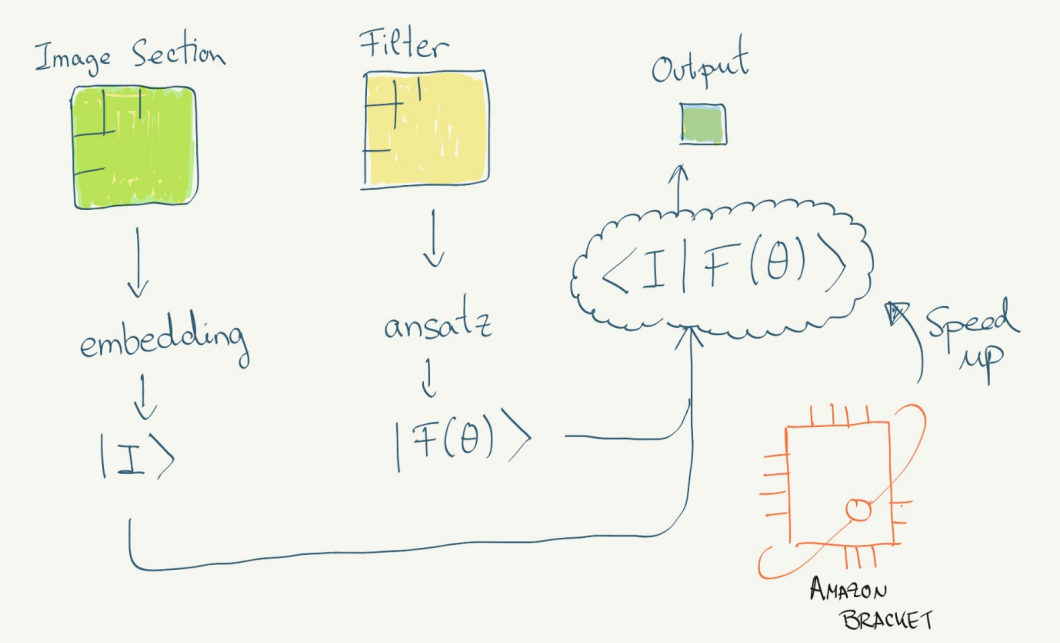


Figure 4: With Amazon Bracket we can enhance the quantum scalar product that we had defined as

simulating more qubits, therefore simulation time also increases exponentially with the number of qubits, at least in full wave-function or state vectors simulators. The gradient computation involves many quantum circuits executions,it scales approximately as $2 \times$ num of trainable parameters, due to the parameter shift rule used to calculate the trainable quantum filter gradients that propagate through the network, and that really blows up the running time, although we expect high performing remote simulators that are able to batch/parallelize quantum circuits to help. Here is a benchmarking for a fixed set of hyper-parameters of the 16-qubit quantum filter, to get a sense of the running time.

Another aspect to consider is network latency between the CPU and remote QPU/ high performance simulators, this can quickly be the major overhead as the feedback loop between classical and quantum computation is iterated many , circuit device executions that are sent to cloud based instances of QPU and simulators must be minimized. For that, the geographical colocalcalization of both processors is a must.

0.4.1 Benchmark

To test our system, we decided to compare with a similar case already studied [2]. In this case, the MNIST is also used, and a fixed quantum filter is applied; that is, it does not train any parameter. The reason for taking random parameters is defined in [2]. It is detailed that this type of filters is suitable for detecting vertices, particularly in this dataset; this becomes a relevant skill.

As we can see, both models end up converging around a 40% precision, which, as we have said before, given the small number of images with which we are training, is a successful result.

2X2 Filter		
Local Mac Pro (image processing time per second)	Amazon Braket (image processing time per second)	
	ml.t3.medium	ml.c5.2xlarge
3.52	19.30	4.52

4X4 Filter		
Local Mac Pro (image processing time per second)	Amazon Braket (image processing time per second)	
	ml.t3.medium	ml.c5.2xlarge
140.05	229.30	170.52

Figure 5: Table that highlights the importance of latency, the type of instance on which we run our project. This table summarizes in some way the simulation and comparison strategy to take into account for future benchmarks. In addition to all this, the device to be used must be taken into account.

0.4.2 Conclusion and future directions

Throughout the project, we have been able to test different strategies to tackle somewhat larger than usual hybrid programming problems. This is thanks to being working with a filter that acts locally on the image. However, the number of operations increases considerably, so parallelism will be included in the classic part (and not only in the circuits) to work with batches of more images in future improvements of the project.

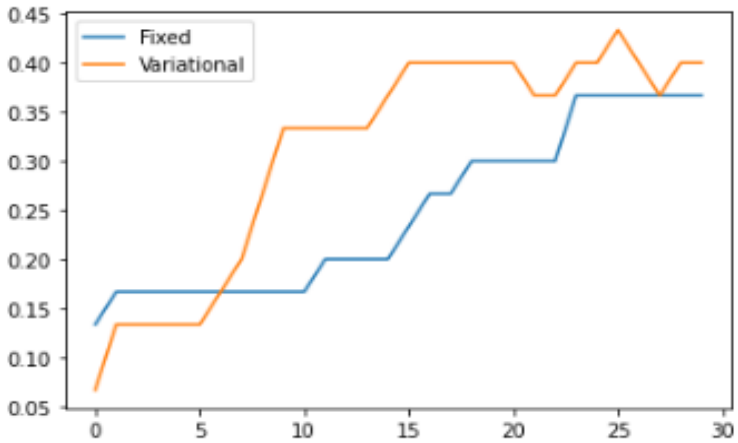


Figure 6: the blue line marks the precision of our model throughout the training process, while the yellow line refers to the learning of the fixed model

0.4.3 acknowledgements

We also want to thank the Xanadu/Pennylane team for all the effort put into this super event, and all the sponsors involved. Specially, Amazon Web Services for providing access to its resources (S3, Amazon Braket) to carry out part of this project. We plan to continue learning on this platform.

Bibliography

[1] Junhua Liu, Kwan Hui Lim, Kristin L Wood, Wei Huang, Chu Guo, and He-Liang Huang. Hybrid quantum-classical convolutional neural networks. *arXiv preprint arXiv:1911.02998*, 2019.

[2] Maxwell Henderson, Samriddhi Shakya, Shashindra Pradhan, and Tristan Cook. Quanvolutional neural networks: Powering image recognition with quantum circuits, 2019.

[3] PennyLane Amazon Braket. Pennylane-braket plugin. `https://amazon-braket-pennylane-plugin-python.readthedocs.io/en/latest/`, 2021 (accessed February 26, 2021).