A Proposal for Examining Speedup of Quantum Machine Learning

Eliska Greplova

July 5, 2017

In this note we discuss the possibilities of gaining quantum advantage with quantum machine learning algorithm. We build up on the approach of arXiv:1706.01561 for QML based binary classification. We generalize the algorithm to multiple qubits and provide a code compilable for IBM quantum experience machine. We argue that the classification problems with minimal number of measured qubits provide the good testing ground in search for end-to-end speedup of machine learning algorithms.

1 Introduction

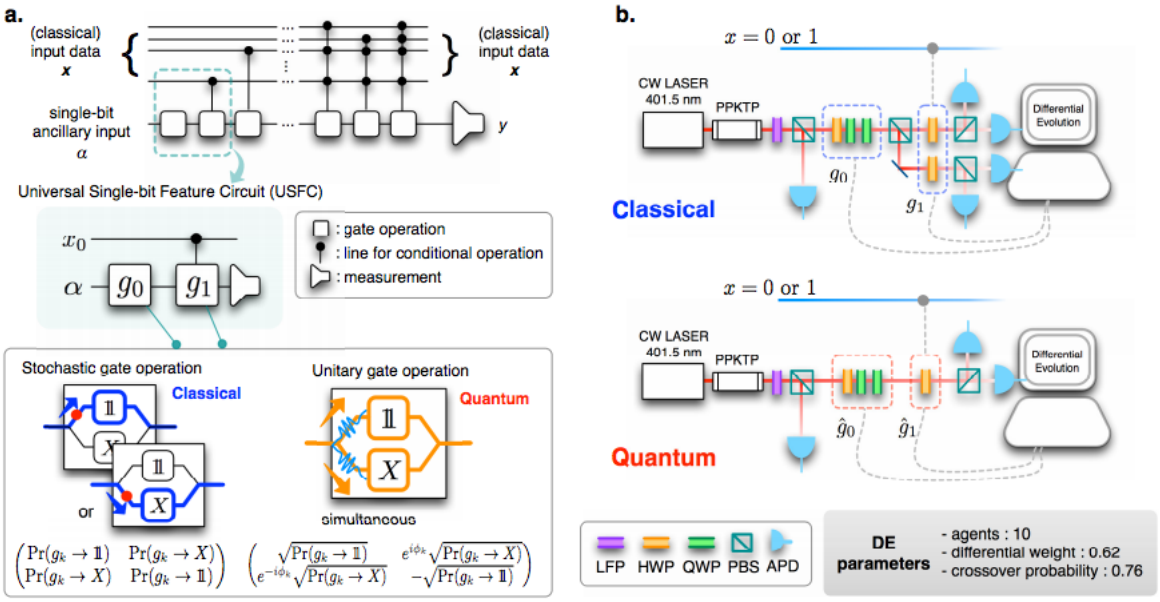

Figure 1: From arXiv:1706.01561. Binary classification protocol with the quantum optical implementation proposed there.

In arXiv:1706.01561 the authors propose binary classification quantum machine learning protocol and experimentally test it for one bit of input data, see Fig. 1. This is a very sound idea that can be readily generalized for bigger inputs. The reason why this problem is particularly attractive in the context of near-term quantum processors is that it minimizes the readout time. In particular measurement readout of superconducting qubits is orders of magnitude longer than performing the quantum operations on the chip. The potential integration of superconducting chip into the quantum/classical interface will then suffer from huge overhead of transforming the quantum readout into the classical data. When trying to find a good candidate algorithm that will show end-to-end quantum speed-up, binary classification (which can be readily generalized into the multiclass classification) seems to be natural candidate.

This note is organized as follows. In Section 2, we explain in the detail our core idea of the extension to the 5 qubit code and provide the ProjectQ code for simulating the chip that would perform the required operations. In Section 3, we give bigger-picture recipe on embedding this into the bigger classical data processing algorithm. In Section 4, we offer list of concrete steps that are needed to verify, quantify and extend the procedure proposed here.

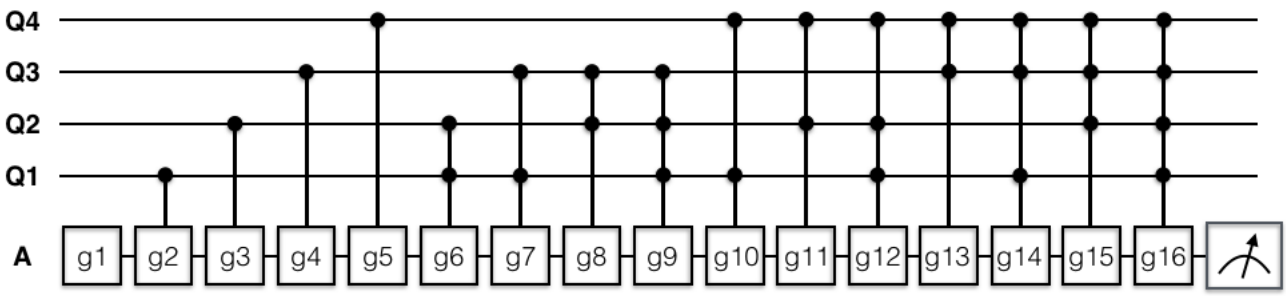

Figure 2: Circuit of Fig. 1a adapted for 5 qubits.

2 Generalization

We propose the extension of the protocol above for bigger inputs, that can be run on super-conducting quantum processor/IBM quantum experience machine. Let us adapt the circuit from Fig. 1a for 5 qubits (4 input, 1 ancillary) that are available on IBM machine. The resulting circuit we would like to implement is shown in Fig. 2. The learning part of the algorithm relies on optimization of conditional phase shift gates $G = \{g_1 \dots g_{16}\}$ such that the desired function (classification) input $\mapsto$ output is realized.

We would like to prepare a code simulating this circuit in the ProjectQ programming language introduced in arXiv:1612.08091 and output the expectation values of the measurements on ancilla A into the differential evolution optimization algorithm to find optimal conditional operations G for any input. In practice, we have two options how to use this code: we can either just run both together on the classical computer using ProjectQ where results of the 'quantum' measurements will be determined at random, or, we can compile the quantum circuit part for IBM Quantum Experience and output the measurement results into the python environment and run the optimization algorithm. Below we present in some more detail parts of code that realizes these steps.

First step clearly is the implementation of the gates G with the flexible rotation angle in ProjectQ. How to do this is shown in Fig. 3. Then we can implement the circuit from Fig. 2 as shown in Fig. 4.

```
# The gates are defined as a class
# Here we define the new gate based on the class: BasicRotationGate
class NewGate(BasicRotationGate):
      # The first function of the class is initialisation
      # Which will take two arguments: phi and pgk
      # See details here: https://github.com/ProjectQ-Framework/ProjectQ/blob/develop/projectq/ops/_basics.py
      def __init__(self, phi, pgk):
            BasicGate.__init__(self)
            self._angle = float(angle)
            self._pgk = float(pgk)

      # The gate will be defined by the matrix as for the Ph gate
      # https://github.com/ProjectQ-Framework/ProjectQ/blob/develop/projectq/ops/_gates.py
      @property
      def matrix(self):
            pgkX = self._pgk
            pgkI = 1 - self._pgk
            return np.matrix([[np.sqrt(pgkI), np.sqrt(pgkX)*cmath.exp(1j * self._angle)],
[np.sqrt(pgkX)*cmath.exp(-1j * self._angle), -np.sqrt(pgkI)]])
```

Figure 3: ProjectQ code for implementation of gates $G = \{g_1 \dots g_{16}\}$ depicted in Fig. 2

Now we need to calculate and evaluate fidelity

$$F = \left(\prod_x \sum_y \sqrt{Pr(y|x) Pr_\tau(y|x)} \right)^{\frac{1}{2^n}}, \tag{1}$$

where $Pr(y|x)$ is the probability obtaining output y given the input x, or, in quantum terms, the mean value of the resulting measured ancillary state. $Pr_\tau(y|x)$ is the target probability, i.e. the probability distribution we want to obtain based on the function, f, we want the circuit to perform. For example, if we want the function $f = 0$ that maps any input string on the state $|0\rangle$, we obtain

$$F = \left(\sqrt{Pr(0|0000) Pr(0|1000) \dots Pr(0|1111)} \right)^{\frac{1}{2^4}}, \tag{2}$$

```
def New_Circuit(phis,pgks,xinput):
        qubit1 = eng.allocate_qubit()
        qubit2 = eng.allocate_qubit()
        qubit3 = eng.allocate_qubit()
        qubit4 = eng.allocate_qubit()
        ancilla = eng.allocate_qubit()

        if numpy.mod(xinput,2) == 1:
                X | qubit1
        if numpy.mod(numpy.floor(xinput/2),2) == 1:
                X | qubit2
        if numpy.mod(numpy.floor(xinput/4),2) == 1:
                X | qubit3
        if numpy.mod(numpy.floor(xinput/8),2) == 1:
                X | qubit4

        NewGate(phis[0], pgks[0]) | ancilla

        CNewGate(1,phis[1],pgks[1]) | (qubit1, ancilla)
        CNewGate(1,phis[2],pgks[2]) | (qubit2, ancilla)
        CNewGate(1,phis[3],pgks[3]) | (qubit3, ancilla)
        CNewGate(1,phis[4],pgks[4]) | (qubit4, ancilla)
        eng.flush()

        CNewGate(2,phis[5],pgks[5]) | (qubit2, qubit1, ancilla)
        eng.flush()
        CNewGate(2,phis[6],pgks[6]) | (qubit3, qubit1, ancilla)
        eng.flush()
        CNewGate(2,phis[7],pgks[7]) | (qubit3, qubit2, ancilla)
        eng.flush()

        CNewGate(3,phis[8],pgks[8]) | (qubit3, qubit2, qubit1, ancilla)
        eng.flush()
        CNewGate(2,phis[9],pgks[9]) | (qubit4, qubit1, ancilla)
        eng.flush()
        CNewGate(2,phis[10],pgks[10]) | (qubit4, qubit2, ancilla)
        eng.flush()

        CNewGate(3,phis[11],pgks[11]) | (qubit4, qubit2, qubit1, ancilla)
        eng.flush()
        CNewGate(2,phis[12],pgks[12]) | (qubit4, qubit3, ancilla)
        eng.flush()
        CNewGate(3,phis[13],pgks[13]) | (qubit4, qubit3, qubit1, ancilla)
        eng.flush()
        CNewGate(3,phis[14],pgks[14]) | (qubit4, qubit3, qubit2, ancilla)
        eng.flush()

        CNewGate(4,phis[15],pgks[15]) | (qubit4, qubit3, qubit2, qubit1, ancilla)
        eng.flush()

        prob0 = eng.backend.get_probability([0],ancilla)

        Measure | qubit1
        Measure | qubit2
        Measure | qubit3
        Measure | qubit4
        Measure | ancilla
        return prob0
```

Figure 4: ProjectQ code for implementation of quantum circuit depicted in Fig. 2. This can be either simulated classically, or compiled and simulated on IBM chip

```
def get_F(x,args=[]):
        phis = [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
        pgks = x

        prob0_array = []
        for n in range(16):
                xinput = n
                prob0n = New_Circuit(phis,pgks,xinput)
                prob0_array.append(prob0n)
        prob0_array = numpy.asarray(prob0_array)
        F = np.prod(prob0_array)**(1.0/32)
        return 1-F
```

Figure 5: Function that call quantum circuit function and calculates the fidelity (2)

```
boundarray=[(0,1), (0, 1), (0, 1), (0, 1), (0, 1),(0,1), (0, 1), (0, 1), (0, 1), (0, 1), (0,1), (0, 1), (0, 1), (0, 1), (0, 1), (0,1)]
A=scipyopt.differential_evolution(get_F, boundarray,args=[],mutation=(0,1.8))
```

Figure 6: Differential evolution optimization function

where under the square root there is a product of conditional probabilities for 16 different possibilities on input string. Eq. (2) is indeed the quantity we want to optimize as a function of gates $g_1, \ldots, g_{16}$. This function is implemented in Python as shown in Fig. 5. Finally we just need to optimize with differential evolution optimization function that is a part of the SciPy library. That's the machine learning part of the algorithm and is implemented as shown in Fig. 6. This code reaches the fidelities around 90% in very short training times. Further study could be done for different set of relative phases and for a variety of different classification problems.

3 Bigger-Picture

The small algorithm described above is a core idea for implementing architectures like the one schematically depicted in Fig. 7. First we need to implement some compression or encoding algorithm to decrease the volume of the input data, then we initialize the quantum circuit in the 'classical' states 0, 1 and run a quantum circuit as the one proposed in the previous section, applying a sequence of conditional operations on the ancillary state, which we finally measure. Note, that this proposal is very classical in a sense, there is no entanglement generated, only quantum super position. As discussed above, this is a binary classification circuit, as we are measuring one qubit only, so we can decide whether an input is in a given class or not (like deciding whether email is a spam etc.). This can be potentially extended to more general classification schemes like One-vs-All. In other words, this covers quite big class of problems without extending measurement time. From our point of view that might be main bottleneck of end-to-end speedup. While the gates applied on super conducting qubits are quite fast, lengthy readout can push back against quantum advantage. This can make a huge difference especially when considering near-term $<$50 qubits chips.

4 Next Steps and Outlook

In this section, we would like introduce a set of concrete steps that can be taken to build up on the approach suggested above.

1. Get better (analytical) idea about quantum advantage of this approach for multiple qubits. The calculation

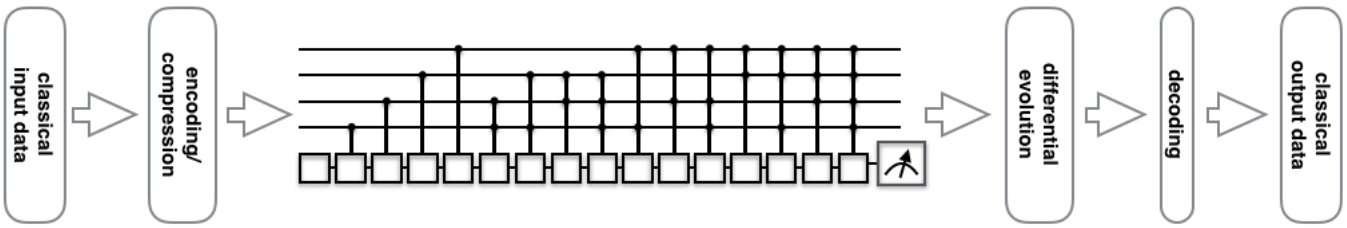

Figure 7: End-to-end visualization of the final algorithm

performed in Supplemental Material of arXiv:1706.01561, while complicated from the combinatorial point of view, would give us more concrete idea about how quantum advantage is scaling.

2. Perform the classical simulation of the procedure above and compare with the speed of ProjectQ simulation.

3. Compile the code of Fig. 4 and run on IBM quantum experience. Compare with the length of the classical simulation. With limited number of measured qubits, hopefully we will be able to see that quantum advantage over weights the disadvantage caused by readout delays.

4. The presently discussed protocol has no entangling gates. Looking into entanglement advantage even for the price of additional measurements might lead to speedup increase.

5. Study of classical and quantum auto-encoding schemes that will perform the outer parts of the circuit in Fig. 7. Currently there is a huge mismatch between the volumes of classical data and number of qubits available, it is therefore imperative to embed the encoding schemes into any practical QML algorithm.