CS269Q: Quantum Programming **Spring 2019**

Project #3

Due: 11:59pm on Wed., May 8 2019, by Gradescope

This project contains both coded solutions (labelled practical) and written problems (labelled theory). These are to be submitted separately to gradescope. For the coding problems there are two template files provided. The file `compilation_template.py` includes further instructions for coding up Problems 1 and 2. The file `grad_template.py` includes further instructions for coding up Problem 3. You will need to submit two files to gradescope `compilation.py` and `grad.py` with your coded solutions.

Problem 1. Gate Compilation
(Practical - 20pts) Write a pyQuil function that compiles all CNOT gates in a program into CZ gates.

Problem 2. GHZ implementations
A GHZ state on n qubits is defined by $|0...0\rangle + |1...1\rangle/\sqrt{2}$. This state can be made in quantum memory with

```
H 0
CNOT 0 1
CNOT 0 2
...
CNOT 0 (n-1)
```

Similarly, one can do:

```
H 0
CNOT 0 1
CNOT 1 2
...
CNOT (n-2) (n-1)
```

However, as we saw in lecture, some QPUs don't allow us to perform CNOT operations between arbitrary qubit pairs directly.

a. (Practical - 10pts) Program each of these and verify that they produce the same operator for $n = 5$.

b. (Theory - 5pts) Depending on the architecture, the first implementation may be more inefficient than the second implementation. Why?

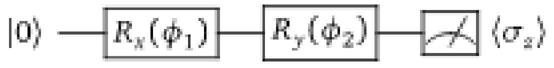
c. (Practical - 30pts) Implement a function called `ghz_compile` which takes as an argument a graph of qubit indices (defined by a list of edges) which defines where two qubit gates are allowed to be applied. Your function should then return a pyQuil program that produces a GHZ state over all those qubits. You can assume that the graph you are provided is connected. *HINT* The `networkx`library is a very helpful library for working with graphs in Python.

Problem 3. Quantum Program Gradients

In this problem we will reproduce what Pennylane does under the hood to calculate gradients of quantum circuits. You will find [1] to be a useful reference.

A key strategy for machine learning on quantum computers is to use the standard *gradient descent* algorithm. For this, it is necessary to compute the gradient of some cost function C with respect to all free parameters α of a circuit, i.e., $\nabla_\alpha C$. This gradient information can be used to determine update rules for the free parameters, namely in the direction of steepest descent: $\alpha \to \alpha + \nabla_\alpha C$

For example, let us consider the following quantum circuit:



In this example, a qubit starts in the $|0\rangle$ state, rotations are done about the x and y axes, and the expectation value of the Pauli-Z observable is obtained. The free parameters are given by $\alpha = (\theta_1, \theta_2)$.

One of the key features of PennyLane is that it performs *automatic differentiation* of quantum circuits, i.e., it automatically computes $\nabla_\alpha C$ for you.

This makes computing gradients a simple one-liner. Note that since there are two parameters, the gradient is a two-dimensional vector:

```
grad_value = qml.grad(circuit, argnum=[0,1])(theta1, theta2)
# Note: we could also do qml.grad(circuit, argnum=[0])([theta1, theta2])
print("Gradient of C:", np.stack(grad_value))
```

a. (Practical - 10pts) Implement this circuit as a qnode in Pennylane. See the template code for details.

b. (Theory - 7pts) Pennylane's gradient computations are based on analytic formulas (i.e., it is not a numerical finite-difference method) and leverage the quantum hardware itself to evaluate them. In this notebook, we will walk through an example of how this is done, focusing on computing the gradient for the specific circuit above.

Our first step is to derive the analytic formula for the gradient of our objective function. The objective function is C, which has the form

$$C = \langle 0|\hat{R}_X^\dagger(\theta)\hat{R}_Y^\dagger(\phi)\hat{P}_Z\hat{R}_Y(\phi)\hat{R}_X(\theta)|0\rangle,$$

where, for later convenience, we use the notation $\hat{P}_j$ to denote the Pauli operator for $j = X, Y, Z$.

Use the product rule and formula $\nabla_\theta \hat{R}_X(\theta) = -\frac{i}{2}\hat{R}_X(\theta)\hat{P}_X$ to show:

$$\nabla_\theta C = -\frac{i}{2}\mathrm{Tr}\Big(\hat{P}_Z\hat{R}_Y(\phi)[\hat{P}_X, \hat{\rho}]\hat{R}_Y^\dagger(\phi)\Big)$$

where $[\cdot,\cdot]$ denotes the commutator and we have combined some terms together to form the new operator $\hat{\rho} = \hat{R}_X(\theta)|0\rangle\langle 0|\hat{R}_X^\dagger(\theta)$.

c. (Theory - 8pts) Simplify this expression (see Eqn (2-3) in [1]) to show:

$$\begin{aligned}\nabla_\theta C = &\tfrac{1}{2}\langle 0|\hat{R}_X^\dagger(\theta+\tfrac{\pi}{2})\hat{R}_Y^\dagger(\phi)\hat{P}_Z\hat{R}_Y(\phi)\hat{R}_X(\theta+\tfrac{\pi}{2})|0\rangle \\ &- \tfrac{1}{2}\langle 0|\hat{R}_X^\dagger(\theta-\tfrac{\pi}{2})\hat{R}_Y^\dagger(\phi)\hat{P}_Z\hat{R}_Y(\phi)\hat{R}_X(\theta-\tfrac{\pi}{2})|0\rangle\end{aligned}$$

d. (Practical - 10) Here we can make a key observation about this final gradient formula: *each term above can be evaluated using the same circuit as the original function C, but with shifted parameters.*

This is the key insight behind the "parameter-shift" gradient trick. Gradients of quantum circuits can often be evaluated using the same circuit, but with shifted arguments. Note that the shift $\frac{1}{2}$ is not infinitesimal, as might be expected if this were a finite-difference numerical method.

Hand-code this gradient as a difference of two circuits and compare to the gradient result from Pennylane in a function called `circuit_grad`. See the template for details.

Acknowledgements

Special thanks for Robert Smith and Nathan Killoran for help with suggesting questions for this project.

References

[1] Kosuke Mitarai, Makoto Negoro, Masahiro Kitagawa, and Keisuke Fujii. Quantum Circuit Learning. 3 2018.