Using Reinforcement Learning to Optimize Quantum Circuits in the Presence of Noise

Khalil Guy[a], **Gabriel Perdue**[a]
[a]Fisk University, SIST Intern
[b]Fermilab, Fermi Quantum Institute

Abstract. Many quantum computing frameworks currently use noise aware algorithms for implementing quantum circuits which do not scale efficiently as the size of the hardware architecture increases. As we move towards devices which utilize more qubits, it becomes increasing more important to map quantum circuits in a way that uses resources efficiently as well as maximizes the reliability of the results of that circuit. However, as the hardware increases to the point where Quantum supremacy is attainable, it will infeasible for a brute-force algorithm to find the most optimal circuit layout for circuits of medium to large depth sizes. To this end, we will to rely on reinforcement learning (RL) as a method of building quantum circuits based on observations of the noise characteristics in its environment. In this work, we create a working reinforcement learning environment in which an agent is able to make action which will build the class of circuits which creates the GHZ state. In addition to this, we also get preliminary results of the performance of a Deep Q Neural Network, which initially does not perform as well as we believe it can. In the future, we want to improve the performance of the agent and potentially generalize this environment to more classes of circuits.

1 Introduction

Quantum computing is a type of computing which leverages several quantum mechanical properties such as superposition and entanglement to perform computations. This type of computing has been shown to offer a significant theoretical speed up for certain algorithms.[1] Currently, quantum computing takes place on Noisy Intermediate-Scale Quantum (NISQ) devices, a class of systems composed of devices with fewer than 1000 qubits.[2] With these systems, and at every stage of quantum devices, finding optimal ways of mapping quantum circuits onto some target hardware. However, as devices become larger and move into the regime of quantum supremacy, exact algorithms for optimization become intractable. Therefore, we propose training a reinforcement learning agent to build optimal circuits in noisy environments.

This manuscript has been authored by Fermi Research Alliance, LLC under Contract No. DE-AC02-07CH11359 with the U.S. Department of Energy, Office of Science, Office of High Energy Physics.

2 Methodology

2.1 Tools for Modeling the Quantum Hardware

Quantum circuits are currently the primary method of doing quantum computing. These circuits are composed of three main components: quantum bits (qubits), quantum gates, and measurement gates.A qubit is the elementary elementary unit in quantum computing. Unlike classical bits which can only hold values of 0 or 1, qubits are able to hold superpositions of states according to a probabilistic scaling of two basis states $|0\rangle$ and $|1\rangle$ in $\mathbb{C}^2$. The state of a qubit can be expressed as

$$|\psi\rangle = \alpha\,|0\rangle + \beta\,|1\rangle \tag{1}$$

where α and β are complex amplitudes such that $|\alpha|^2 + |\beta|^2 = 1$. Qubits are acted on by quantum operators or "gates" which changes their state by altering the probability amplitude. There are two kinds of gates: one-qubit gates, which acts on one qubits to change its state, and two qubit gates which uses a control-target pair of qubits to change the state of the target qubit. There are n-qubit gates, with n - 1 control qubits, but these are not considered in this study.

In order to model this behavior we use QuTip, an open-source framework for quantum computing based in Python. This package allows us to do low-level implementations of quantum circuits with fully connected qubits as well as implementation of custom gates for even more control. In order to model the environment seen in actual quantum hardware, we use the NetworkX graph-theory Python package to treat both the hardware and the circuit as a directed graph.

2.2 Creating Reinforcement Learning Environment with OpenAi Gym

Reinforcement learning is a type of machine learning which uses an agent to choose from a certain set of actions based on observations from an environment to complete a task or maximize some reward. Formally, this is know as a Markov Decision Process (MDP), where $\mathbf{S}$ is the finite set of states of the environment, $\mathbf{A}$ is the finite set of actions that can be taken, and $\mathbf{R}$ is the finite set of rewards that can be received. At each time step $t = 0, 1, 2, ...$ the agent observes a state $s_t \in \mathbf{S}$ chooses an action $a_t \in \mathbf{A}$ based on that state. At the next time-step $t + 1$, the environment transitions into a new state $s_{t+1} \in \mathbf{S}$ and the agent receives a reward $r_{t+1} \in \mathbf{R}$. The focus of this study is to create the environment in which an agent will be taking actions and making observations of states. Specifically, we will be defining the action space $\mathbf{A}$, state space $\mathbf{S}$, and criteria for receiving reward and the amount at each time step. This is implemented using OpenAI Gym, an open-source framework for creating reinforcement learning environments.

2.2.1 Workflow of the RL Agent

The environment the RL agent will be working in is noisy quantum hardware with nearest neighbor connectivity. Therefore, the environment is a directed square lattice graph, where each node q_n represents a qubit and each edge v_{ij} represents an available connection between a qubit q_n, where n, i, j are elements in the node space $Q = [0, 1, 2, ..., m]$ where m is the number of nodes in the graph.

In this environment, the agent will start at some random node on the grid. This location is described by a pointer which is limited to moving towards other nodes which directly adjacent to it. In this beginning, the pointer will move without placing gates in order find the best location to start the circuit. It evaluates the goodness of a particular location by evaluating average fidelity

of random GHZ circuits from its current location. After the agent has made some determination of where to start the circuit, it will begin the circuit by placing a Hadamard gate at the pointer's current position. After this step, the agent proceeds by choosing a gate-direction pair and placing placing these gates with respect to the direction, where the control qubit is the pointer's current position and the target qubit is the node the pointer moves to the choosen direction. It will do this, placing either a CNOT gate or a SWAP gate,until the desired circuit width is reached.

2.2.2 State Representation and Action Space

The only information that is given to the agent in terms of what it observes is the state of the circuit at each time step. The circuit is represented by an initially edgeless, directed graph and the agent observes the the adjacency matrix of the graph. The adjacency matrix of a graph with m nodes is an mxm matrix. As the agent adds gates to the to the circuit, edges are added to the graph and elements are added to matrix according to the nodes that they connect.

Noise Model For our noise model, we want to use the concept of a "Noisy Unitary" (NU), N to simulate coherent noise on a quantum device. This NU is represented by as superposition of two gates. Initially, we are able to construct a gate $\hat{N}$ such that

$$\hat{N} = aU + bT \tag{2}$$

where U and T are unitary kxk two-qubit gate operations and a and b are constants. We then need to transform $\hat{N}$ into a unitary matrix N, which can be done through the Grahm-Schmit process. This process will transform the columns of $\hat{N}$, n_l into an orthonormal basis set of $\mathbb{C}^k$ such

that

$$n_l = \frac{\hat{n}_l - \sum_{j=1}^{l-1} \text{proj}_{n_j}(\hat{n}_l)}{\|\hat{n}_l - \sum_{j=1}^{l-1} \text{proj}_{n_j}(\hat{n}_l)\|} \tag{3}$$

where

$$\text{proj}_{n_j}(\hat{n}_l) = \frac{\hat{n}_j \cdot n_l}{n_l \cdot n_l} n_l$$

. For our environment, we model the cohenrent noise for a CNOT gate as superposition of itself and the CZ gate where

$$CNOT = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{pmatrix} \quad CZ = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & -1 \end{pmatrix}$$

The matrix for this noisy unitary according to the above formula is

$$NU(a) = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & \frac{b}{\sqrt{a^2+b^2}} & \frac{a}{\sqrt{a^2+b^2}} \\ 0 & 0 & \frac{a}{\sqrt{a^2+b^2}} & -\frac{b}{\sqrt{a^2+b^2}} \end{pmatrix} \tag{4}$$

Notice that while NU is unitary for any $a, b \in \mathbb{R}$ or any $a = b \in \mathbb{C}$, we want NU to represent some percentage of its constituent gates. Therefore, we restrict $a = c \in [0, 1]$ and $b = 1 - a$. We can verify this matrix by substituting $a = 1$ and $a = 0$ to get the CNOT and CZ gate, respectively.

From this gate, we create the SWAP gate, which is composed of three CNOT gates. To generate random noise in our device, each of the edges in the graph are assigned a normally distributed random element $c \in [0, 1]$ and a $NU(c)$ is applied along that edge.

2.2.3 Calculating Intermediate Reward Using Random Tree Search

At each time step, we need to have some measure of goodness of the potential completed circuits that can be created from the agent's current position in the hardware at it current stage in completing the desired circuit. Therefore, we use a random tree search in order to create random circuits, calculate their fidelity, and use the average as an intermediate reward.

2.2.4 RL Agent Goal and Restrictions

To start, the goal of the agent will be to create simple circuits and we hope to develop something which can be applied more generally in the future. For our study, the agent's goal is to create a circuit which creates the GHZ state for a specified number of working qubits. This circuit is very simple in that its implementation is straightforward. The GHZ state circuit adheres to a line topology on any graph which causes entanglement between qubits. Therefore, a minor, but important, detail that must be included in the environment is that the agent is not able to make moves which would form a loop or non-line topology in the graph.

3 Preliminary Agent Results

For the agent, we use a Deep-Q Network (DQN) as the agent for our environment. We use a standard implementation of the agent provided in the Tensorflow Agents Python package. As seen below, the agent does perform better after a certain number of time steps but it does not perform optimally, as when as noiseless path is introduced to the environment, the agent is not able to find

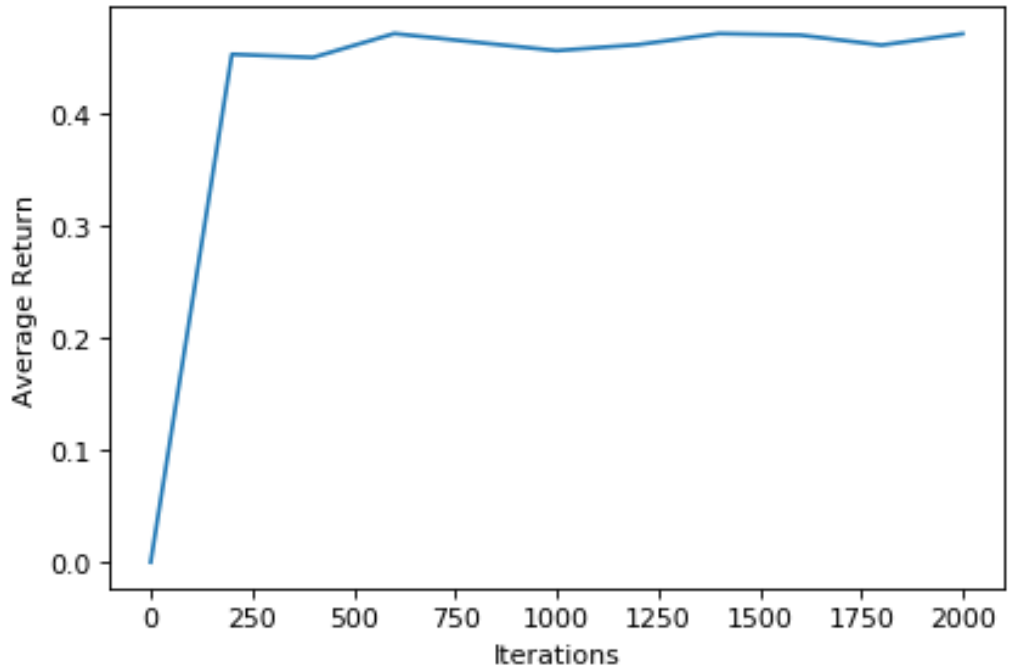

Fig 1 A plot of the agent's average return vs. the number of iterations.

it. At the moment, more testing and fine tuning are needed in order to improve the performance of the agent.

4 Future Works

From this study, we build a functional reinforcement learning environment in which a reinforcement learning agent can perform actions and receive rewards. Additionally, we also obtain preliminary results of a DQN agent's performance in this environment, which is not optimal when a noiseless, optimal path is introduced and performance remains relatively low. Thus, going foward, we want to improve the performance of our RL Agent with the following questions in mind: What is the best state representation of the environment for the RL agent? Which RL model will have the best performance in the proposed environment? How do we generalize the environment to more classes of circuits?

References

1 F. Arute, K. Arya, R. Babbush, *et al.*, "Quantum supremacy using a programmable superconducting processor," 505 – 510 (2019). https://doi.org/10.1038/s41586-019-1666-5.

2 P. Murali, J. Baker, A. Javadi-Abhari, *et al.*, "Noise-adaptive compiler mappings for noisy intermediate-scale quantum computers," 1015–1029 (2019).