# Designing Quantum Circuits using Reinforcement Learning

A Quantum Information Project

Owen Huisman (5653525) & Renze Suters (5616964) & Charles Renshaw-Whitman (5513812)

July 30, 2023

**Abstract**

Quantum computers run algorithms using circuits made up of unitary operations on qubits. Successfully designing quantum circuits is not always an easy task, however. When algorithms become more complex, with a large number of qubits, or when noise is involved, designing a circuit that can implement a particular algorithm is a highly non-trivial task. In this project, we approached the problem of designing quantum circuits under different conditions using Reinforcement Learning (RL). We developed an RL scheme that allows the so-called (reinforcement) learner to place gates in the circuit based on comparison with a desired target output state. We used Qiskit to classically simulate quantum circuits. The learner is able to successfully implement all 24 gates in the Clifford set using just $H$ and $T$ gates as well as avoiding miscalibrated gates when given a larger gate set to work with. Finally, the agent was trained to approximate arbitrary gates using either $H$ and $T$ gates alone or a wider set of gates, $H$, $T$, $X$, $Y$, $Z$, etc.

# 1 Introduction

Since the inception of the field of quantum information, much effort has been devoted to designing quantum algorithms based on unitary operations on qubits. In the perhaps not-so-distant future, many of these algorithms might actually be implemented when quantum computers with large numbers of qubits are available. Real-life qubits, however, are subject to all sorts of noise and decoherence processes, which is a major problem in creating working quantum computers. It is therefore interesting to investigate the possibility of designing quantum circuits that can deal with different types of noise and that can produce any required output state. In this project, we approach this problem using reinforcement learning (RL) and implement the first few steps required to create software that can, in principle, 'adapt' the quantum circuit to different types of input states subject to different noise for any required output state.

Reinforcement learning is part of the rapidly expanding field of machine learning. RL deals with the design and optimisation of fully autonomous machines. [1] A recent exciting development DeepMind's use of RL to train a single agent to superhuman performance on a number of Atari games. [2] RL is unique in that it lies at the interface between AI and control theory, and is closely related to the 'optimal control' problem. [3] In contrast to other types of AI, the RL agent not only does data analyses, but will actively interact with its environment in a feedback-like manner. Besides designing quantum computing circuits, exploring the possibility of using RL for a real physics experiment has been the main motivation behind this project.

In this work we present an RL implementation for designing quantum circuits. The several design considerations that went into this are discussed in the following sections. The resulting AI was able to decompose the full set of single qubit Clifford gates into gates of a discrete universal set, and could perform a reasonable approximation of a gate not exactly implementable using this discrete set. We wrote all code in the Python programming language, using the open source libraries Qiskit [4] from IBM, and Tensorforce [5] (itself based on Tensorflow [6] from Google).

First, we give a basic introduction to reinforcement learning in section 2. This is followed by a more precise description of the problems that we are trying to solve to get the reinforcement learner to work properly and qualitatively and quantitatively test its behaviour in section 3. Section 4 describes the way we implemented the reinforcement learner in practice and section 5 shows the results obtained with our implementation. Finally, the results are discussed and the report is concluded in section 6.

## 2 Reinforcement Learning Basics

At its conception, the goal of this project was to explore the use of reinforcement learning in quantum circuit design. We offer here a minimalist explanation of reinforcement learning in the service of interpreting the work discussed below. Figure 1 shows an example of how reinforcement learning terminology is used in practice.

Reductively, a reinforcement learning algorithm is a type of optimization algorithm. A reinforcement learning setup has the following key components:

- The *environment* encompasses everything interacting with the process of interest. It may be fully or only partially observable, and may be deterministic or stochastic. For our purposes, the state of the environment is given by the sequence of gates implemented, i.e. the whole quantum circuit. The *state* may refer either to the environment or to the most recent observation thereof.
- The *agent* or *learner* is an algorithm which observes the environment (i.e. the state), and outputs a desired *action* in an attempt to maximise reward. As the agent interacts with the environment, it (in theory) learns to make decisions which lead to greater rewards. What this means in practice is made clear below.
- The *reward* or *reward signal* is a scalar value which the agent observes at each time step. States with higher reward are 'better' than those with lower reward. All reinforcement-learning algorithms are designed to maximise the (possibly time-discounted) reward, and all aspects which determine a state's desirability must be reflected in the reward.

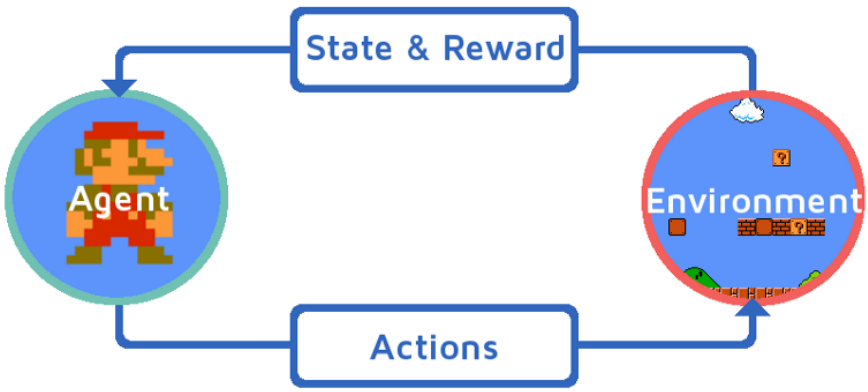



Figure 1: Illustration of the most important Reinforcement Learning terminology. In this example, the RL machine or agent is playing Mario and receives information about the environment around Mario's current position in the form of a state observation and reward. In turn, the agent playing Mario can decide to perform certain actions such as walking or jumping, thereby changing its surrounding environment, upon which the agent receives information regarding its new environment and this scheme is repeated over and over until the end of the episode. Image credit: https://www.educba.com/what-is-reinforcement-learning/.

Some more technical terminology will be used in discussing the difficulties of the reinforcement learning approach:

- A timestep is the smallest unit of 'time' considered and refers to the period in which an agent makes one observation, takes a single action, and receives the associated reward. Timesteps are discrete, and at each timestep, one gate is placed.
- An *episode* is a collection of timesteps. At the beginning of an episode, the environment is in the default state. Timesteps are taken in sequence until the episode terminates, either because the agent requests termination, or because some pre-set maximum episode length is surpassed.
- *Training* an agent consists of allowing it to 'play' multiple episodes. After each episode, the agent alters its behaviour in proportion to the reward received for an episode: high rewards incentivise the agent to repeat the behaviours taken, while low rewards cause it to disfavour these behaviours. What is meant concretely with 'behaviour' is explained below. The optimisation method that was used for our agents was 'Proximal Policy Optimisation'

(PPO). The details of this algorithm will not be treated in this work, but we mention here that it is state-of-the-art and that it has been used successfully in many machine learning implementations.[7]

- A *hyperparameter* is any parameter which affects the behaviour of the learning algorithm; such parameters do not effect the environment. These parameters may be scalars, such as the *learning rate* or more complex, such as the agent's internal representation of its observations.
- The *Policy Network* or *Neural Network* (NN) is the mechanism by which an agent converts observation of a state into an action. The neural network may be thought of as parameters $\theta$ for a function $p_\theta(\text{action} \mid \text{state})$, where $p$ outputs the action most favoured by the agent. Explicitly it is a set of *layers* $\mathbf{o}^{(k)}$, which contain *nodes* $o_i^{(k)}$. The nodes of layers $k$ and $k-1$ are connected by *weights* $w_{ij}^{(k)}$. The first 'input' layer of a policy network is obtained from the state of the system. The value of the nodes of the next layers are given by a cascaded calculation of the form:

$$o_i^{(k)} = g\left(\sum_j w_{ij}^{(k)} o_j^{(k-1)}\right)$$

  Here $g$ is a non-linear *activation* function, (e.g. $\tanh x$), which allows the NN to model systems that are non-linear. The final calculated 'output' layer is used to to determine new actions for the agent to take.

  Above the 'training' of an agent was mentioned. This comes down to optimizing the weights of the NN. The way that weights are optimised is part of the PPO algorithm.

# 3 Problem Statement

Our work is heavily inspired by that of Florian Marquardt [8], who has used reinforcement learning to, *inter alia*, develop noise-robust quantum circuits and to compile complex, many-qubit circuits (see references). Our goal with the project was to get a feel for the ways in which reinforcement learning techniques could act in service of quantum computation.

For the sake of originality, we developed several tasks which we wanted to use to test the reinforcement learner. As the project progressed, it became apparent that some of these objectives were either infeasible or would require substantial modifications to the existing framework, while other opportunities for interesting experiments presented themselves. We outline here only those experiments that were performed.

The main issues we are interested in exploring are: (i) the design of gates robust to gate-noise and (ii) implementation efficiency, especially in trade-off with accuracy.

## 3.1 Problem 1: Unitary Inference from Observations

All of our work has focused on the implementation of particular unitary operators. While a useful abstraction, this neglects the fact that in quantum computation, one generally measures the states of qubits and not the operators which act on them. Thus, in order to tighten the (admittedly loose) connection between simulation and reality, we seek to assess the learner's ability to infer a unitary operator on the basis of its effect on different states. More specifically, we seek to determine if the agent is capable of acting intelligently when its observations are not of the implemented unitary itself, but of the results of the unitary's actions on different basis states.

$$\text{Observation} = \begin{bmatrix} U_{11} & U_{12} \\ U_{21} & U_{22} \end{bmatrix} \longrightarrow \text{Observation} = \begin{bmatrix} U\,|{+x}\rangle \\ U\,|{-x}\rangle \\ U\,|{+y}\rangle \\ U\,|{-y}\rangle \\ U\,|{+z}\rangle \\ U\,|{-z}\rangle \end{bmatrix}$$

## 3.2 Problems 2: Calibration Error and 3: Noise Circumvention

In testing 'robustness', our goal was to use the reinforcement learner to design circuits in which the gates were not consistently implemented. This inconsistency we refer to as *manufacturing noise*, on account of the fact that this is not noise in the traditional sense of causing entanglement with the environment; instead, all gates are implemented with some small error. Note that the general unitary is specified using the three Euler angles:

$$U(\theta, \phi, \lambda) = \begin{pmatrix} \cos\frac{\theta}{2} & -e^{-i\lambda}\sin\frac{\theta}{2} \\ e^{i\phi}\sin\frac{\theta}{2} & e^{i(\phi+\lambda)}\cos\frac{\theta}{2} \end{pmatrix}$$

As an example, upon receipt of an order to 'add an $H$ gate', the program adds the parameterised U gate, which for $H$ is $U(\frac{\pi}{2}, 0, \pi)$. To each of these parameters is added some small quantity, drawn from a normal distribution:

$$\begin{gathered} H \longrightarrow U(\frac{\pi}{2} + \epsilon_1, 0 + \epsilon_2, \pi + \epsilon_3) \\ \epsilon_1 \sim \mathcal{N}\left(\mu_{\mathrm{H},1}, \sigma^2_{\mathrm{H},1}\right) \\ \epsilon_2 \sim \mathcal{N}(\mu_{\mathrm{H},2}, \sigma^2_{\mathrm{H},2}) \\ \epsilon_3 \sim \mathcal{N}(\mu_{\mathrm{H},3}, \sigma^2_{\mathrm{H},3}) \end{gathered}$$

The $\epsilon$ parameters represent the 'manufacturing noise' of the gate, so-called as its behaviour resembles that of an analog circuit manufactured to operate within tolerance of some nominal value - once the component is implemented this does not change, however. Note that all single-qubit transformations may be accomplished by an appropriate $U$ gate, so this formalism is valid for all single-qubit simulations. The $\mu$ corresponds to a fixed error which all gates of the type considered will have, while the $\sigma^2$ represents the individual variation between gates. There are a number of design choices made as to how this noise should be handled; more detail is given in Section 4.

In implementing this version of noise, we examined two separate problems. First, we attempt to see if the learner is capable of mitigating the effects of a 'calibration error'. This corresponds to the case where all of the $\sigma^2$ are 0, while some gates have $\mu \neq 0$, which corresponds to a persistent error in the way a gate is implemented (as might be the case if one miscalibrated, say, a laser). It is hoped that the agent recognises that some gates are faulty and refrains from using them.

The second type of noise-related problem we wanted to address is the reverse of the above: All $\mu = 0$ and some gates have $\sigma^2 \neq 0$, i.e. the gates are not implemented consistently, but display no average error. As above, it is hoped that the agent learns to refrain from using noisy components in the circuit (see Section 4 for details on penalties for noise).

The archetypal example of both of these issues may be illustrated if we consider a Z gate subject to noise/error of either variety, while X and Y gates are noiseless:

$$G_1\ G_2\ ...\ G_{n-1}\ \tilde{Z}\ G_{n+1}\ ...\ \longrightarrow\ G_1\ G_2\ ...\ G_{n-1}\ (XY)\ G_{n+1}...$$

That is, the equivalent noiseless circuit should be preferred to the noisy gate (the phase $-i$ is global and thus of no physical consequence in a single-qubit circuit).

### 3.3 Problems 4: Efficient Clifford-Gate Implementation and 5: Accuracy-Efficiency Frontier Discovery

To test the ability of the agent we consider two tests. The first test has been designed to probe this tradeoff and test the learner's flexibility by letting it implement the 24 Clifford gates, using only the $H$ and $T$ gates. This problem is a good test as the Clifford gates are implementable exactly in relatively short sequences of H and T; for this task the agent must plan effectively.

The second test concerns the ability of the agent to implement a gate that is not exactly implementable using the available gate set, but can be sequentially approximated up to smaller and smaller errors. This is familiar in the context of the Solovay-Kitaev theorem, which states that, from a discrete universal gate-set, any single-qubit unitary may be approximated to within error $\epsilon$ by application of $O\left(\log^c\left(\epsilon^{-1}\right)\right)$ such gates.[9] If the learner is sufficiently intelligent, one may hope to realise the $O\left(\log^c\left(\epsilon^{-1}\right)\right)$ bound.

# 4 Methodology and Design Considerations

## 4.1 Observing the State and Accounting for Noise

Above, we stated some considerations that went into finding a set of quantitative observations (the state) that (i) could be used to fully determine the unitary operation of the circuit in question, but (ii) which uses only information of the quantum state of the qubits. We decided to use the density matrix of the qubits as the agent's state observation. The density matrix formalism allows for the most natural implementation of noise - in particular, it allows measurement of a state's mixedness (see below). As it is a bounded quantity, which explicitly takes into account possible mixture of states, we use the fidelity as the basis of our reward function. Figure 2 illustrates the principal feedback loop.

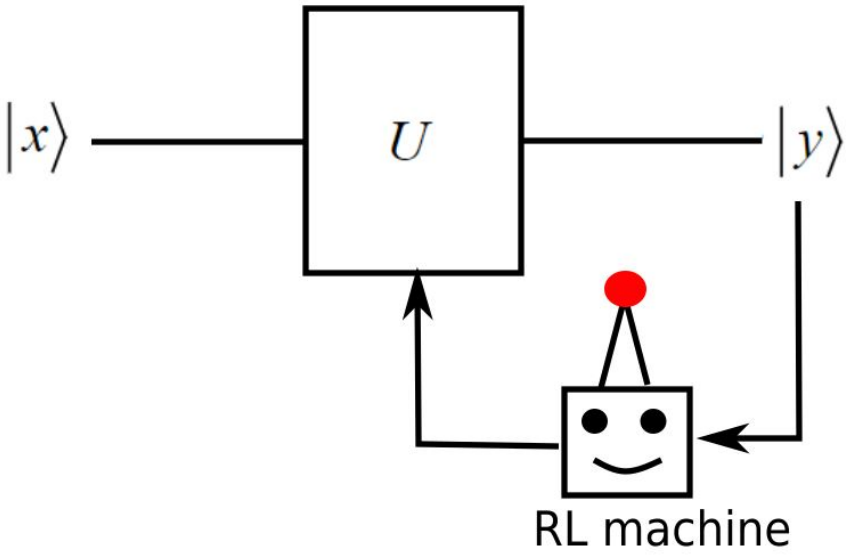



Figure 2: Illustration showing the main feedback loop the RL agent concerns itself with. The circuit applies a unitary to a state. The agent has access to the output, and makes changes to the circuit, to achieve a certain target state output.

A brief note is in order regarding how 'noise' is implemented, given that, as described above, we focus on 'manufacturing noise' as opposed to the traditional environmental decoherence noise. Using this admittedly odd definition of noise does provide the formalism to address the problems identified, however. In dealing with gates subject to 'manufacturing noise', the following procedure is used in order to estimate the mixed state of the noisy circuit: For each cardinal state, build $N$ copies of the planned circuit; for each gate that is applied, implement a universal U-gate drawn from the distribution corresponding to that gate. Then combine the density matrices:

$$\rho_{+x} = \frac{1}{N}\sum_{i}^{N} \rho_{+x}^{(i)} \quad \ldots \quad \rho_{-z} = \frac{1}{N}\sum_{i}^{N} \rho_{-z}^{(i)}$$

As mentioned above the state that the agent observes is the input layer to the neural network (NN), and consists of sequence of real numbers. We chose these to be the entries of the sample-averaged density matrices, omitting as many unnecessary entries as possible:

$$\text{State} = [\text{Re}((\rho_{+x})_{11}),\ \text{Re}((\rho_{+x})_{12}),\ \text{Im}((\rho_{+x})_{12})\ \ldots\ \text{Re}((\rho_{-z})_{22})]$$

Note that diagonal elements of $\rho$ are real, and because $\rho$ is Hermitian, only the lower triangular elements are included: $(\rho)_{ij}\ ,\ i > j$. See below for an explanation of why all six cardinal states are used.

## 4.2 The Need to use all Cardinal States

By design with this way of obtaining the state, the use of noisy gates will always reduce a reward based on the fidelity, but is it the case that the state now implicitly contains enough information to

infer the implemented unitary? We had considerable difficulty in diagnosing an issue related to this point; we began the project by reasoning that knowledge the kets $U|0\rangle$ and $U|1\rangle$ should suffice, as the action of $U$ on $|\alpha\rangle = \alpha_0|0\rangle + \alpha_1|1\rangle$ is just $U|\alpha\rangle = \alpha_0(U|0\rangle) + \alpha_1(U|1\rangle)$ (and analogously for any number of qubits). However, because we wanted to make our simulations more 'realistic', we chose to observe not $U|\psi\rangle$ but the (potentially mixed) density matrix, which is why it is not sufficient to observe only these binary states; information about the phase is lost in going from the state to density matrix representation; consider:

$$U|0\rangle = |\alpha\rangle\,, \quad U|1\rangle = |\beta\rangle$$

as compared to

$$U|0\rangle = |\alpha\rangle\,, \quad U|1\rangle = \mathrm{e}^{i\pi}|\beta\rangle$$

Both of these operators have the same density matrix (the phase cancels with its conjugate). Thus, if we wish to use density matrices (with the fidelity-based reward, that penalises noise), a more complicated tomography scheme must be used.

To address this difficulty, a number of workarounds were considered, such as explicitly calculating the accrued phase (this worked poorly). The method ultimately used was to apply the procedure outlined above, measuring $\rho$ for each of the six cardinal states, as one would do in experiments as well. Cardinal state observation was found to work quite well for the single-qubit case, but the generalisation to multiple qubits requires the use of all possible combinations of single-qubit cardinal states, which becomes computationally costly very quickly.

In sum, the choice of how to represent the state of the system has some important subtleties - the method chosen allows for full determination of the unitary, explicit penalisation/measurement of noisiness, and is reasonably physical in its underpinnings (if one revised the sampling procedure appropriately, the learner could just as well work with real lab measurements). The disadvantages of this approach are the relatively large number of distinct numbers provided, eighteen, in contrast to the ostensible four degrees of freedom that the unitary has (three if one parameterises using the appropriate Euler angles). Additionally, this representation is difficult to generalise to more than a single-qubit, computationally and practically speaking.

## 4.3 Actions

Depending on the experiment being performed, the set of possible actions was all permitted gates and a terminating NULL action. After the agent gave a new action output, the corresponding gate was appended to the circuit, up to the manufacturing noise drawn from the normal distribution. The use of the terminating-NULL immediately ends the episode, at which point the circuit is evaluated and the reward determined. Allowing termination of the episode is important, as in general, the agent's actions are chosen stochastically according to their favourability: if a non-terminating NULL were allowed, an agent may reach a perfect circuit implementation, it might 99% favour the choice of nothing and 1% favour the implementation of some new gate; if 1000 timesteps are taken, there is effectively no chance that this perfect implementation will remain undamaged. It is also standard practice to implement an 'exploration rate' (a hyperparameter) which governs the chance an agent will act randomly rather than according to the policy - this is done in order to decrease the attractiveness of local optima; a non-zero value for this exploration rate exacerbates the problem above. Figure 3 schematically shows the anatomy of the agent and how it ties into the environment.

## 4.4 Rewards

As mentioned above, we would like to base the reward on the fidelity between the output state of the circuit $U\rho' U^* = \rho$ and a specified target state $\sigma$. We can use the general formula for matrix fidelity:

$$F(\sigma, \rho) = \left(\mathrm{tr}(\sqrt{\sqrt{\sigma}\rho\sqrt{\sigma}})\right)^2 \tag{1}$$

Also here we note that there are $\rho$'s and $\sigma$'s for each of the cardinal states. The fidelity used for calculating rewards is the average of these six fidelities.

It is a known phenomena that the effectiveness of reinforcement learning depends on how rewards scale.[10] An effective scaling for the rewards was found to be:

$$R = e^{CF_{avg} - K}$$

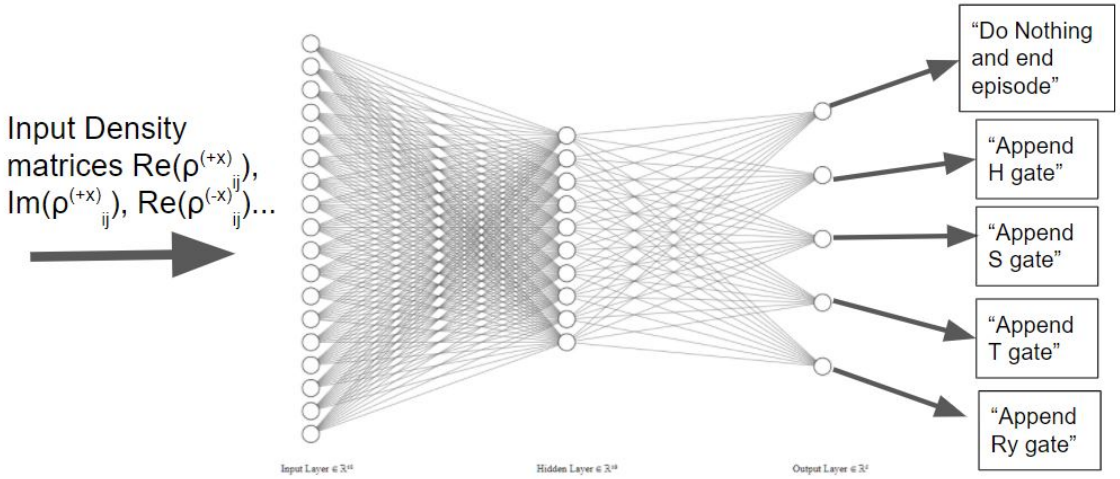



Figure 3: Illustration showing the anatomy of the agent used to obtain our results. This consists mainly of a neural network (NN), whose weights are optimised by training. The input layer contains information of the output density matrices for all 6 cardinal input states. The output layer corresponds to the different gates that the agent can add to the circuit. There is one hidden layer in between.

Where C and K, respectively, are the scale and offset parameters. The main reason for using this reward is that, since the fidelity is bounded between 0 and 1, a sequence of approximants to the desired unitary will receive a rapidly diminishing increase in reward; as the agent's behaviour changes in proportion to the reward received, this decreasing return-on-accuracy will make it much less likely that the agent will discover difficult or complex patterns of approximation.

In addition to scaling, another important choice was to only allocate the agent rewards at the end of the episode, instead of rewarding per timestep as is sometimes seen in literature. This ameliorated a number of subtle problems associated with the discount rate and episode termination; the tradeoff for this improvement is that the single reward value at episode-end is not as informative to the agent as a continuously updated stream of rewards.

### 4.5 General Rules of Thumb

In this paragraph we would like to mention some general considerations that came up during the progress of designing the agent and environment. These come down to some rules of thumb that were confirmed time and time again working with the reinforcement learner.

- All of the agent hyperparameters are important. It became clear very early that none of these were redundant, and that each of them seems to have an optimum. In the end we used a open-source set of optimised hyperparameters that the Tensorforce developers converged on when training their RL agents. [5]
- The size of the NN hidden layers should be between that of the input and the output layers. This leads us to the next point.
- It is best to reduce complexity as much as possible. For example, as mentioned above, elements of the density matrices that are redundant were not added to the state. In general a smaller agent is easier to train than a larger agent.
- As mentioned above, the state was based upon density matrices and the reward on the corresponding fidelities. This implies that both the observations and the rewards will lay in a finite interval. This boundedness is important to some implementations of RL algorithms like the one used: PPO. [10]

### 4.6 Miscellaneous Design Considerations

The problem setup arrived at and described above is the result of a large number of design decisions which we found to play an important role in ensuring that the agent was able to perform with even minimal intelligence. For illustrative purposes, we discuss several of these here.

#### 4.6.1 Reward Structure: Should an intermediate reward be included?

To begin with, quite a bit of time was spent in determining the optimal structure of the reward which would encourage the agent to behave as desired. One aspect of this design was the decision to offer rewards to the agent only at the end of an episode, rather than at intermediate timesteps (the latter of which is quite common in many applications of RL to control problems). On the one hand, these rewards provide a much richer feedback to the agent regarding performance and the favourability of any given state of the environment - this is valuable as it substantially improves the trainability of the agent. On the other hand, this continuous stream of rewards has the problem that it has a perverse incentive - consider the example of an agent which has implemented an $R_z(\pi/3)$ rotation, with the goal of implementing an $R_z(\pi/5)$ rotation: the implemented gate will provide a reward quite close to the maximum; if the intervening sequence of gates is quite long and 'detours' far from the goal, the reward lost along this 'detour' may outweigh the slight gain in reward from implementing the gate more accurately. In the end it was determined that the only way to ensure the agent behaved as desired was to use a reward function which only rewarded the 'final' circuit, i.e., one in which all intermediate steps receive no reward. Though this makes training the agent more difficult, this reward is analogous to the 'zero-one' reward used in two-player competitive games, for which good performance of RL algorithms can still be achieved, as exemplified by the landmark performance of AlphaGoZero [11].

#### 4.6.2 Reward Structure: How should noisiness of circuits be measured and penalised?

Another important consideration is associated with the way that noisiness of a circuit is considered, especially given the non-standard implementation of our 'gate-noise'. In this regard there are two related questions: how should the 'noisiness' of a circuit be measured, and how, on the basis of this measurement, should it be penalised? In measuring the noisiness, there are two main options: on the one hand, one may consider a single implementation of the circuit, with each of the gates drawn from the appropriate noisy distribution, and simply calculate the fidelity on the basis of this circuit. This has two main disadvantages: the reward is the principal way that the agent is informed about its environment, so any noisiness in this signal will decrease the ease with which it is trained. Further, this method is incapable of directly quantifying the amount of noise in the circuit. The alternative option, designed to ameliorate this issue, is to 'sample' different implementations of the circuit, in each case redrawing the gates from the appropriate distribution; the output density matrices are then averaged. This has two main advantages: for one, the averaging process will tend to reduce the purity of the output density matrix, encoding directly information about the circuit's noisiness; second, this allows for better characterisation of the noise between different elements of the circuit - the hope was that a sufficiently intelligent agent might use this information to perform as many intermediate steps as possible in a 'noiseless subspace' (whether such a subspace exists depends on the noise and on the problem to be solved, of course). These benefits of more directly characterising noise and allowing it to be directly penalised are the two main reasons we determined to use the density-matrix approach. This approach is, however, associated with increased computational cost (increasing the number of runs in proportion to the number of samples), and with the addition of this sample-size as an additional hyperparameter.

#### 4.6.3 Training and Hyperparameters: Exploration, Learning Rate, Discount, and Neural Network structure

The final design decision pertains less to the implementation of the quantum circuit environment than to problems arising from the inconsistent behaviour of the RL learner. The RL algorithm depends on a number of 'hyperparameters', which characterise the learning algorithm and policy network which constitute the agent. We will limit our discussion here to a brief mention of the most important of these: the exploration rate, the learning rate, the discount factor, and the structure of the policy network. Each of these hyperparameters has a substantial impact on the agent behaviour and exhibit no clear pattern that would indicate an optimal value. The arbitrariness in setting these parameters constituted one of the key difficulties encountered. The parameter values used to generate the final results are included from the appendix, and were taken from an open-source implementation of the PPO algorithm for an unrelated reinforcement learning problem [12].

The exploration rate is the probability that the agent will take an action at random (i.e., without regards to the policy-network). The purpose of this randomness is to force the agent to explore new parts of state space, preventing it from becoming 'stuck' in a local optimum [13]. Because rewards are offered only at the end of an episode, it was observed that for most cases, a non-zero exploration rate

simply resulted in the agent having no idea how to respond to the newly vandalised circuit, placing gates essentially at random and performing very poorly. Thus for many of our later simulations the exploration was set to zero or a very small value.

The learning rate governs the rate at which the policy network is altered in response to the received reward. It may be thought of as the 'step-size' of the gradient-descent problem [13]. A small learning rate will cause slower learning, but should correspond to steadier improvement; a large learning rate will cause the agent to learn rapidly, but may cause a high degree of instability [13]. The difficulty is not only in this computation-time vs performance tradeoff, but in that there is no readily available way to assess which regime one is operating in, i.e., it is not *a priori* clear whether a learning rate of, say, 0.01 is laughably small or ludicrously large.

The use of time-discounting of rewards is a standard technique in reinforcement-learning, designed to account for uncertainty about the environment [13]. It is expected that some amount of time-discounting would encourage the agent to end the episode with a circuit it thinks passable, rather than hit the assigned timestep limit, likely in the middle of some operation which will make the output useless. The difficulty with this is that, in conjunction with the 'reward only at end of episode' policy, a non-zero discount rate may encourage the agent to end an episode prematurely when it stands to make only slight accuracy gains. It is unclear what discount rate would be sufficient to ensure the agent ends an episode rather than start 'flailing', while simultaneously not being so punishing as to discourage incremental improvements to the circuit.

The final hyperparameter, the structure of the policy network, is not a scalar value like the previous ones, but instead refers to the number, type, and size of the layers which make up the policy network. In general a policy network may be made extremely complex, adding a large number of high-width layers, which will provide an enormous degree of representational flexibility; this improved representational capacity is generally counterbalanced by the fact that increasingly complex networks have more free parameters and become more difficult to train. Though the design of such neural networks is a field of study in its own right, we found that, for the amount of training performed in each case, the 'funnel-in' network structure with only one hidden layer, shown in Figure 3, provided acceptable results.

#### 4.6.4 Environment and Action Structure: Alternative Possibilities

The overall architecture of the environment and the manner in which the agent interacts with the environment was far from the only choice possible. We outline here a few possible alternatives that we considered before arriving at the present implementation.

At present, the agent is started 'from-scratch' each time we wish to have it implement a new unitary. One alternative to this approach would be to make the desired output an input to the agent's policy network; this would, in principle, allow the training of a single agent capable of implementing a wide variety of unitaries. In the end, given the large number of consistency problems encountered, this option was not chosen as it would require much more complex behaviour from the agent on the one hand, and on the other would require an increase in the size of the policy network, which we expect to correspond to increased training difficulty as noted above. Thus, though the training of such an agent would be an impressive feat, we deemed it unlikely that the substantial increase in training required would be within our capabilities.

Also worth noting is the 'sequential' nature of the agent; at every timestep, the agent 'sees' only the output density matrices, but not the gates which make up the implemented unitary. Further, it is only able to append gates, without being able to 'edit' the circuit already constructed. Though this flexibility would likely be highly valuable, this implementation was not chosen on account of the fact that this representation would require variation of both the input and output of the policy network; this would have entailed an enormous increase in the program complexity, and would likely have posed equally large difficulties in training (given that the structure of the neural network must change).

#### 4.6.5 Summary and Design Takeaways

In sum, a number of design decisions were made early in the project which entailed exploring only one possible implementation structure to solve the problem given. From the environment architecture to the learner hyperparameters, many of these decisions must be made somewhat arbitrarily. The alternative architectures noted above may serve as a starting point for new and interesting projects,

but cannot really be said to correspond to 'tradeoffs' so much as they are simply different realisations of the same idea.

One possibility for definite improvement is offered, however, by the possibility of hyperparameter tuning algorithms. This was not implemented for two reasons: first, they are computationally very expensive, and should, strictly speaking, be redone at every redefinition of the problem (i.e. for each new unitary); second, the complexity of choosing hyperparameters was realised only late in the project, and the implementation of one of these algorithms would have required a complete rewriting of the code for compatibility. Using such an algorithm would significantly increase the complexity of the code by adding another 'black box'; it was also not clear that hyperparameters alone were responsible for the inconsistent performance observed, though this is possible, as the switch from hand-tuning (guessing) these values to using the open-source ones made available saw a marked improvement. Thus, tuning would be a good possibility to explore, but would likely not singlehandedly resolve the reliability issues we faced, which are known to be endemic to reinforcement learning[13].

# 5 Results

In this section we treat the results that have been achieved by the agent after sufficient training, and using suitable hyperparameters. We state again that using the set of PPO hyperparameters provided in the appendix and the finalised environment class, one should be able to reproduce each of these results.

## 5.1 Problem 1: Unitary Inference from Observations

We started off by training the agent to solve trivial problems, in order to confirm that it could correctly infer the unitary it has implemented on the basis of the density matrix entries available to the learner as state observations. The very first test consisted of training the agent to do nothing; the learner's goal was to implement the identity, which was only one possible action among the, sometimes many, other gates it was allowed to place. The agent is able to do this extremely trivial task with great consistency; it successfully converged on placing the identity and terminating the episode in each training session.

The learner also successfully passed the next few slightly harder, but still trivial, tests:

- Implementing all gates that are already in its allowed gate set. For instance, when giving the learner $H$ and $T$ gates to work with, its goal was to implement $H$ or $T$.
- Implement a particular rotation gate a few times in a row, without placing any other gates in between. For instance, when giving the learner a $\pi/8$ rotation about the $y$-axis and an $S$ gate to work with, its goal was to implement the rotation about $y$-gate 7 times in a row.

These basic training sessions show that the agent can infer the unitary it has implemented on the basis of the state observations we supply it with in our design and perform different actions based on these observations. It also shows that the learner can consistently converge on the state with the maximum reward.

## 5.2 Problems 2 and 3: Avoiding Miscalibration and Gate Noise

When allowing the agent to place gates from a larger set, namely $\mathfrak{G} = \{X, Y, Z, H, S, T, R_y(\pi/4)\}$, the algorithm showed its potential to avoid noise. For example, in a test run where the agent's target was to implement the $Z$ gate, the subset $\{X, Y, Z\}$ of implementable gates were purposefully miscalibrated, i.e. a fixed error $\mu \neq 0$ with $\sigma^2 = 0$. This resulted in the agent (consistently) converging on the equivalent, but more accurate circuit $HR_y^2(\pi/4)$. The agent also successfully avoids gates with higher manufacturing noise, when $\mu = 0$ but $\sigma^2 \neq 0$. It is worth noting that the agent's success in these cases is reliant on being able to make short sequences of actions on the basis of its observations, but does not necessitate very long or complex strings of gate placements planned out in advance. That is, the agent is quite capable of determining between immediately good and bad decisions with a horizon of a few timesteps, but this ability diminishes rapidly as more complex decisions are required, as is exhibited in the case of problem 5 below.

## 5.3 Problem 4: Clifford gates

With an agent able to place only $H$ and $T$ gates on a single qubit circuit, the learner is able to fully reproduce the set of 24 single qubit Clifford gates up to a global phase factor (which we deliberately made immaterial to both observations and the reward through using the density matrices as observations). This required training of 500 episodes. In the appendix the 24 Clifford gates are given, together with (some of the) equivalent circuits the agent converged upon. This is one of the more complex tasks the agent proved able to perform, so discussion of the results in greater detail may provide insight.

Figure 4 shows a (smoothed, 40-episode rolling average) plot of the rewards obtained by the agent in 10 different training sessions to implement the Clifford gate built out of $HT^2HT^2$ as a function of the episode number. All agents started with a 'clean sheet'. Of the 10 agents, only 7 succeeded in learning to implement the desired Clifford gate. The other 3 never explore the high-reward state space.

This figure offers the chance to provide a few remarks on the training process which apply to most of the agents trained for this project. First, the rewards shown are smoothed as the un-smoothed plot

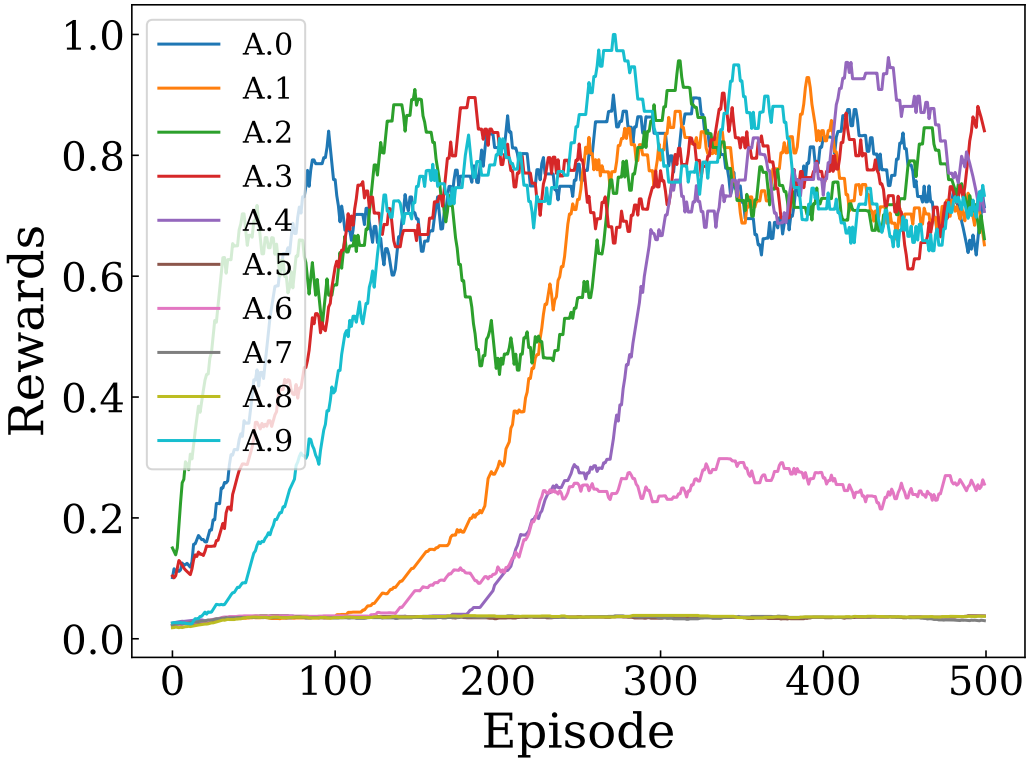


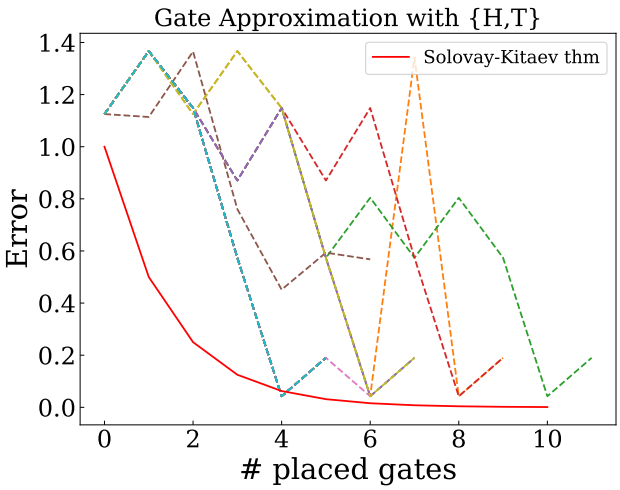

Figure 4: Figure showing the normalised smoothed reward obtained by 10 different agents during training over 500 episodes for the Clifford gate that is decomposed as $HT^2HT^2$. The normalised rewards shown here are smoothed using a moving average over 40 episodes. The plots shows that agents 7 and 8, as well as agent 6 to a lesser extent, never explore the state space associated with higher rewards (higher fidelity). The other learners start exploring the higher-reward state space at different episodes, but all of them eventually succeeded in implementing the Clifford gate using $H$ and $T$ gates. Not all agents reach the maximum normalised reward 1 in this plot because of the way the rewards are smoothed using the moving average.

is incomprehensible, all agents oscillating regularly between states of maximum and zero reward: excepting those agents which fail to converge and never achieve a high reward, regardless of how well agents perform on average, these agents are highly inconsistent (this is especially important as it relates to the exploration rate, as discussed previously). Second, the improvement is far from monotonic, and is extremely variable, with agents fluctuating greatly in their performance; convergence may take 50 episodes just as it may take 500, or may not occur at all. Lastly, it is worth noting that an agent is usually quite capable of eventually converging to whatever the maximum reward it has ever received is - thus if an agent only ever explores low-reward configurations, it has now way of 'knowing' that improvement is possible. We see agents with both extremely high variability and a tendency to become stuck in mediocre local optima, thus losing at both ends of the famous 'explore-exploit' tradeoff.

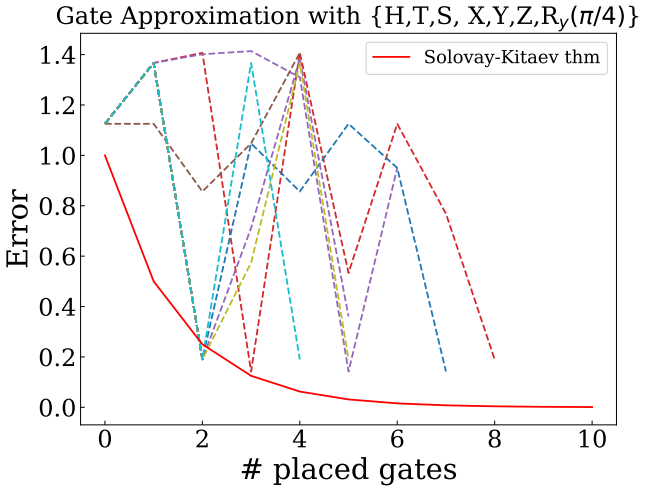
### 5.4 Problem 5: Gate approximation

As was discussed in section 3 we desired to compare the efficiency of the agent in gate approximation to the asymptotic limit as stated by the Solovay-Kitaev theorem. We repeat here that the theorem states that there is an asymptotic bounds on the amount of gates needed in the approximation, given a upper bounds on the error $\epsilon$: a single qubit gate can be approximated up to accuracy $\epsilon$ with $O(\log^2(1/\epsilon))$ gates from the discrete universal set $\mathfrak{U} = \{H, T\}$. The error $\epsilon$ is given by a distance measure for unitary operators. Namely, for a unitary $U$, with approximation $\tilde{U}$, we have:

$$\epsilon = \max_{|\psi\rangle \in \mathcal{H}} |\, \langle\psi|(U - \tilde{U})|\psi\rangle \,|$$




(a) Gate approximation using the gate set $\mathfrak{U} = \{H, T\}$ for several trained agents.




(b) Gate approximation using the gate set $\mathfrak{G} = \{X, Y, Z, H, S, T, R_y(\pi/4)\}$ for several trained agents.

Figure 5: Plots comparing the gate approximating ability of the agent w.r.t. the bounds of the Solovay-Kitaev theorem. The x-axis shows the number of placed gates, the y-axis the error $\epsilon$. The red solid curve is the theoretical asymptotic bounds. The dashed lines show the performance of the agent after training using the gate set $\mathfrak{U}$, resp. $\mathfrak{G}$.

For certain $U, \tilde{U}$, let the state that gives the maximum expectation value for $(U - \tilde{U})$ be denoted by $|\phi\rangle$, which for our purposes is assumed to be pure. We relate $\epsilon$ to the fidelity $F(U|\phi\rangle, \tilde{U}|\phi\rangle) = \langle\phi|U^*\tilde{U}|\phi\rangle$ (since $U|\phi\rangle$, $\tilde{U}|\phi\rangle$ both are pure normalised states).

$$\begin{aligned}\epsilon = \max_{|\psi\rangle\in\mathcal{H}} |\langle\psi|(U-\tilde{U})|\psi\rangle| &= |\langle\phi|(U-\tilde{U})|\phi\rangle| \\ &= \sqrt{|\langle\phi|(U^*-\tilde{U}^*)|\phi\rangle\langle\phi|(U-\tilde{U})|\phi\rangle|} \\ = \sqrt{|\langle U^*U\rangle| + \langle\tilde{U}^*\tilde{U}\rangle - \langle\tilde{U}^*U\rangle - \langle U^*\tilde{U}\rangle|} &= \sqrt{2|1-F|}\end{aligned}$$

Expressing $\epsilon$ in $F$ gives us a nice way to calculate the error in a phase invariant way. For this, $|\phi\rangle$ can be calculated numerically by finding the eigenstate that corresponds the the max. eigenvalue of $(U - \tilde{U})$, for which Python libraries exist.

The agent was trained twice for 1000 episodes of 100 timesteps. The target was implementing $R_y(\sqrt{2}\pi)$. On first instance with the universal gate set $\mathfrak{U}$, and a second time with the extended gate set $\mathfrak{G}$. For the episodes near the end of training the development of $\epsilon$ after each timestep was calculated.

An example of this is given in figure 5, showing the result of a few trained agents approximating the gate. This figure is very representative for all of the episodes near end of training. Using the gates of $\mathfrak{U}$ or $\mathfrak{G}$ the learner was not capable of approximating the target effectively; the Solovay-Kitaev 'bound' is drawn for illustrative purposes, but is in reality, of course, asymptotic. The agent could perform a reasonable approximation w.r.t. this limit, but would however stop after a handful of gates. This means that the current agent hyperparameters, and corresponding reward structure do not lead the learner to explore approximations consisting of larger gates sequences. In consequence, it is not possible to infer the asymptotic limit of the RL approximation efficiency.

# 6 Discussion & Conclusion

To conclude, we offer a recapitulation of the problem setup and the results achieved, followed by a discussion of possible further directions of research.

A discussion of noteworthy design decisions made throughout the project is offered in Section 4, with special attention given to difficult points and alternative methodologies in Section 4.6. The main points may be summarised briefly: the architecture chosen is not unique, and different environment, action, and observation architectures may allow an agent to interact much more adroitly with the quantum circuit environments. It is not clear which, if any, of these different architectures would enhance performance. Further, the tuning of hyperparameters is thought to be a key difficulty hindering the agent's effective learning for tasks which require more than a few timesteps of planned action. Tuning these hyperparameters would be a substantial, but possibly amply rewarded task.

Considering the design and its results presented above, the following improvements and directions of future research can be considered. We implemented the first steps toward using Reinforcement Learning as a tool to design quantum circuits. To this end, we wrote an RL enviroment in Python, using the Tensorforce library. This environment allows us to train a PPO agent to find single-qubit circuits that are equivalent to target unitaries up to a global phase. We used the agent hyperparameters published by OpenAI. The agent was able to find equivalent circuits consisting of $H$ and $T$ gates for the set of 24 Clifford gates. The agent could also be trained to avoid errors by miscalibration, and to approximate an irrational angle rotation with reasonable error rate with respect to the asymptotic bounds predicted by the Solovay-Kitaev theorem for a small number of allowed gates. In sum, the agent is capable of performing interesting tasks, but is limited by a lack of consistency on the one hand, and an inability to 'plan' long gate sequences on the other.

Finally, we may offer a few directions for further investigation, aside from the architectural changes mentioned above:

- The scheme where the observation takes into account the effect of the circuit on all of the six cardinal states has not yet been successfully been implemented for general $n$ qubits. Adding more qubits, implies the consideration of entanglement and mixing of qubits within each of the three Bloch sphere bases. Already for $n = 2$ this would greatly increase the complexity in terms of code architecture, but also in terms of required agent ability. Extending the RL environment to $n = 2$ or $n = 3$ would be an interesting future project on its own. Beyond this, architectural changes would be needed, though such methodologies are available, such as the mentor-pupil strategy employed by Marquardt et al. [8].
- The agents trained for implementing single qubit circuits could still be subject to improvement. Specifically in the choice of hyperparameters progress may be gained. It is known in literature that RL hyperparamers are a delicate matter, and specific tuning algorithms have been designed for finding harmonious combinations. As of writing, the trained agent does not always converge to implementing its target correctly, which might be overcome by using such a hyperparameter 'tuner'.
- Naturally it would be interesting to explore different noise models. For example decohering noise, as was done by Marquardt. [8]
- The agent was not able to find long strings of gates, approximating a certain unitary better and better. This prevented us to make a prediction on the asymptotic behaviour of the approximation error. Tests have shown that this is due to a part of the parameter space was not being explored. This is a problem that could be solved by changing the scheme by which rewards are allocated, or by improvement of the aforementioned hyperparamters.

We are of the opinion that Reinforcement Learning holds the potential to become very useful for research in quantum computing. It might not only be used to perform quantum circuit simulations and design circuits, but RL might also be applied in a more direct way as part of a real-time feedback loop in an experimental set-up.

# References


[1] Arulkumaran, K. et al. (2017) 'A Brief Survey of Deep Reinforcement Learning'. IEEE Signal Processing Magazine, Special Issue on Deep Learning for Image Understanding

[2] Mnih, V. et al. (2013) 'Playing Atari with Deep Reinforcement Learning'. DeepMind.

[3] Brunton, S.L. Kutz, J.N. (2019) 'Data-Driven Science and Engineering'. Cambridge University Press.

[4] https://qiskit.org/

[5] https://tensorforce.readthedocs.io/en/latest/

[6] https://www.tensorflow.org/

[7] Schulman, J. et al. (2017) 'Proximal Policy Optimization Algorithms'. arXiv:1707.06347v2

[8] Fösel, T. et al. (2021) 'Quantum circuit optimization with deep reinforcement learning'. arXiv:2103.07585v1

[9] Nielsen, M.A. Chuang, I.L. (2010) 'Quantum Computation and Quantum Information'. 10th Anniversary Edition. Cambridge University Press.

[10] Engstrom, L. et al. (2020) 'Implementation matters in deep policy gradients: a case study on PPO and TRPO'. arXiv:2005.12729v1

[11] Silver, D. et al. (2017) 'Mastering the game of Go without human knowledge'. doi:10.1038/nature24270

[12] https://github.com/tensorforce/tensorforce/blob/master/benchmarks/gym-cartpole/ppo.json

[13] Sutton, R. S. and Barto, A. G. (2018) 'Reinforcement Learning: An Introduction'. 2nd Edition. The MIT Press

## Appendix: Single qubit Clifford gates

$$\begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix} = I$$

$$\begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix} = HT^4H$$

$$\begin{pmatrix} 0 & -i \\ i & 0 \end{pmatrix} = HT^4HT^4$$

$$\frac{1}{2}\sqrt{2}\begin{pmatrix} 1 & -i \\ -i & 1 \end{pmatrix} = HT^2H$$

$$\frac{1}{2}\sqrt{2}\begin{pmatrix} 1 & i \\ i & 1 \end{pmatrix} = T^2HT^2$$

$$\frac{1}{2}\sqrt{2}\begin{pmatrix} 1 & -1 \\ 1 & 1 \end{pmatrix} = HT^4$$

$$\frac{1}{2}\sqrt{2}\begin{pmatrix} 1 & 1 \\ -1 & 1 \end{pmatrix} = T^4H$$

$$\frac{1}{2}\sqrt{2}\begin{pmatrix} 1-i & 0 \\ 0 & 1+i \end{pmatrix} = T^2$$

$$\frac{1}{2}\sqrt{2}\begin{pmatrix} 1+i & 0 \\ 0 & 1-i \end{pmatrix} = T^6$$

$$\frac{1}{2}\sqrt{2}\begin{pmatrix} 0 & -1-i \\ 1-i & 0 \end{pmatrix} = T^2HT^4H$$

$$\frac{1}{2}\sqrt{2}\begin{pmatrix} 1 & 1 \\ 1 & -1 \end{pmatrix} = H$$

$$\frac{1}{2}\sqrt{2}\begin{pmatrix} -i & -1 \\ 1 & i \end{pmatrix} = T^2HT^6$$

$$\frac{1}{2}\sqrt{2}\begin{pmatrix} 0 & -1+i \\ 1+i & 0 \end{pmatrix} = HT^4HT^2$$

$$\frac{1}{2}\sqrt{2}\begin{pmatrix} 1 & -1 \\ -1 & -1 \end{pmatrix} = T^4HT^4$$

$$\frac{1}{2}\sqrt{2}\begin{pmatrix} -i & 1 \\ -1 & i \end{pmatrix} = T^4HT^2H$$

$$\frac{1}{2}\begin{pmatrix} 1-i & -1-i \\ 1+i & 1+i \end{pmatrix} = T^2HT^2H$$

$$\frac{1}{2}\begin{pmatrix} 1+i & 1+i \\ -1+i & 1-i \end{pmatrix} = T^2H$$

$$\frac{1}{2}\begin{pmatrix} 1+i & 1-i \\ -1-i & 1-i \end{pmatrix} = HT^6HT^2$$

$$\frac{1}{2}\begin{pmatrix} 1+i & 1-i \\ -1-i & 1-i \end{pmatrix} = T^6HT^2H$$

$$\frac{1}{2}\begin{pmatrix} 1-i & 1-i \\ -1-i & 1+i \end{pmatrix} = HT^2HT^2$$

$$\frac{1}{2}\begin{pmatrix} 1+i & -1+i \\ 1+i & 1-i \end{pmatrix} = HT^2$$

$$\frac{1}{2}\begin{pmatrix} 1+i & -1-i \\ 1-i & 1-i \end{pmatrix} = HT^4HT^2H$$

$$\frac{1}{2}\begin{pmatrix} 1-i & 1+i \\ -1+i & 1+i \end{pmatrix} = T^2HT^6H$$

# Appendix: code

## Cwrapper.py

```
'''
Wrapper to train agent to implement the Clifford gates using H and T
'''

import numpy as np
from itertools import product
from qiskit import QuantumCircuit
import qiskit.quantum_info as qi
from qiskit.quantum_info.operators import Operator
from QuantumRunner import QuantumRunner
import matplotlib.pyplot as plt

def generate_clifford_gate(theta,nx,ny,nz):
    global Id, PauliX, PauliY, PauliZ
    return Id*np.cos(theta/2)-1j*np.sin(theta/2)*(nx*PauliX+ny*PauliY+nz*PauliZ)

clifford_gates = np.zeros((24,2,2),dtype='complex')

Id = np.array([[1,0],[0,1]],dtype="complex")
PauliX = np.array([[0,1],[1,0]],dtype="complex")
PauliY = np.array([[0,-1j],[1j,0]],dtype="complex")
PauliZ = np.array([[1,0],[0,-1]],dtype="complex")

clifford_coords = np.array([
[0,0,0,0],
[np.pi,1,0,0],
[np.pi,0,1,0],
[np.pi,0,0,1],
[np.pi/2,1,0,0],
[-1*np.pi/2,1,0,0],
[np.pi/2,0,1,0],
[-1*np.pi/2,0,1,0],
[np.pi/2,0,0,1],
[-1*np.pi/2,0,0,1],
[np.pi,1/np.sqrt(2),1/np.sqrt(2),0],
[np.pi,1/np.sqrt(2),0,1/np.sqrt(2)],
[np.pi,0,1/np.sqrt(2),1/np.sqrt(2)],
[np.pi,-1*1/np.sqrt(2),1/np.sqrt(2),0],
[np.pi,1/np.sqrt(2),0,-1*1/np.sqrt(2)]
,[np.pi,0,-1*1/np.sqrt(2),1/np.sqrt(2)],
[2*np.pi/3,1/np.sqrt(3),1/np.sqrt(3),1/np.sqrt(3)],
[-1*2*np.pi/3,1/np.sqrt(3),1/np.sqrt(3),1/np.sqrt(3)],
[2*np.pi/3,-1*1/np.sqrt(3),1/np.sqrt(3),1/np.sqrt(3)],
[-1*2*np.pi/3,-1*1/np.sqrt(3),1/np.sqrt(3),1/np.sqrt(3)],
[2*np.pi/3,1/np.sqrt(3),-1*1/np.sqrt(3),1/np.sqrt(3)],
[-1*2*np.pi/3,1/np.sqrt(3),-1*1/np.sqrt(3),1/np.sqrt(3)],
[2*np.pi/3,1/np.sqrt(3),1/np.sqrt(3),-1*1/np.sqrt(3)],
[-1*2*np.pi/3,1/np.sqrt(3),1/np.sqrt(3),-1*1/np.sqrt(3)]
])



for idx,coords in enumerate(clifford_coords):
    tmp0 = coords[0]
    tmp1 = coords[1]
    tmp2 = coords[2]
```

```
    tmp3 = coords[3]
    clifford_gates[idx] = generate_clifford_gate(tmp0,tmp1,tmp2,tmp3)

epsilon = np.ones((len(clifford_gates)))

for lv,clifford_gate in enumerate(clifford_gates[20:21]):

    #For each clifford gate, start by creating target:
    Clifford = Operator(clifford_gate)
    num_qubits = 1
    targetBigRho = np.zeros((6**(num_qubits),2**(num_qubits), 2**(num_qubits)), dtype=complex)
    cart_prod_states = list(product(['0', '1', '+', '-', 'r', 'l',], repeat=num_qubits))
    initstring = ''
    for basis_id in range(6**(num_qubits)):
        initstring = ''
        for bit in range(num_qubits):
            initstring = initstring + cart_prod_states[basis_id][bit]
        circ = QuantumCircuit(num_qubits)
        circ.initialize(initstring,circ.qubits)
        circ.append(Clifford, [0])
        targetBigRho[basis_id] = qi.DensityMatrix.from_instruction(circ)
    for sim in range(0,10):
        qr_output = QuantumRunner(targetBigRho,lv).get_stored_data()
        action_list = qr_output[1][-1]
        w = 20 #half-window-size for avg
        smoothed_rewards = [ np.average( qr_output[0][ max(0,lv-w) : \
        min(lv+w,len(qr_output[0])-1)] ) for lv,r in enumerate(qr_output[0])]
        #np.savetxt('rewards'+str(lv)+'_'+str(sim)+'.csv', smoothed_rewards, delimiter=',')
```

### Skwrapper.py

```
'''
Wrapper for gate approximation and comparison to Solovay-Kitaev theorem.
To be run with extended gateset.
'''
import numpy as np
from itertools import product
from qiskit import QuantumCircuit
import qiskit.quantum_info as qi
from qiskit.quantum_info.operators import Operator
from QuantumRunner import QuantumRunner
import matplotlib.pyplot as plt

def Solovay_Kitaev_limit(timesteps):
    return [2**(-1*t) for t in range(timesteps)]

def reconstruct_unitary(actions):
    '''Function to rebuild a unitary given some episode actions.
    Check that unused gates are commented out!! '''
    Ry = np.array([[np.cos(np.pi/8),-1*np.sin(np.pi/8)],\
    [np.sin(np.pi/8),np.cos(np.pi/8)]],dtype='complex')
    S = np.array([[1,0],[0,1j]], dtype='complex')
    X = np.array([[0,1],[1,0]],dtype="complex")
    Y = np.array([[0,-1j],[1j,0]],dtype="complex")
    Z = np.array([[1,0],[0,-1]],dtype="complex")
    U = np.array([[1,0],[0,1]],dtype='complex')
    H = np.array([[1,1],[1,-1]],dtype='complex')/np.sqrt(2)
    T = np.array([[1,0],[0,np.exp(np.pi*1j/4)]], dtype='complex')
    if len(actions) != 0:
        for action in actions:
```

```
            if action == 0:
                pass
            elif action == 1:
                U = np.matmul(Ry,U)
            elif action == 2:
                U = np.matmul(S,U)
            elif action == 3:
                U = np.matmul(X,U)
            elif action == 4:
                U = np.matmul(Y,U)
            elif action == 5:
                U = np.matmul(Z,U)
            elif action == 6:
                U = np.matmul(H,U)
            elif action == 7:
                U = np.matmul(T,U)
    return U

def reconstruct_circuit(actions):
    '''Function to rebuild a circuit given some episode actions.
    Check that unused gates are commented out!! '''
    circ = QuantumCircuit(n)
    qbit_idx = 0
    if actions.ndim != 0:
        for action in actions:
            if action == 0:
                pass

            #commented below to test with JUST H and T
            elif action == 1:
                circ.u(np.pi/4,0,0, qbit_idx)
            elif action == 2:
                circ.s(qbit_idx)
            elif action == 3:
                circ.x(qbit_idx)
            elif action == 4:
                circ.y(qbit_idx)
            elif action == 5:
                circ.z(qbit_idx)
            elif action == 6: #change 1*-->6* if you add gates back
                circ.h(qbit_idx)
            elif action == 7: #change 2*-->7* if you add gates back
                circ.t(qbit_idx)
    return Operator(circ)

def calc_errors(action_list, target_unitary):
    '''Function to calculate approximation error for target unitary,
    given episode actions '''
    errors = []
    for idx in range(len(action_list)):
        approx_unitary = reconstruct_unitary(action_list[:idx])
        eigs, vects = np.linalg.eigh(target_unitary-approx_unitary)
        eigvector = qi.Statevector(vects[:,np.argmax(np.abs(eigs))])
        U, V = qi.Operator(target_unitary), qi.Operator(approx_unitary)
        psi, phi = eigvector.evolve(U), eigvector.evolve(V)
        errors.append(np.sqrt(2-2*qi.state_fidelity(psi,phi)))
    return errors


num_qubits = 1
```

```
targetBigRho = np.zeros((6**(num_qubits),2**(num_qubits), 2**(num_qubits)), dtype=complex)
cart_prod_states = list(product(['0', '1', '+', '-', 'r', 'l',], repeat=num_qubits))
initstring = ''
for basis_id in range(6**(num_qubits)):
    initstring = ''
    for bit in range(num_qubits):
        initstring = initstring + cart_prod_states[basis_id][bit]
    circ = QuantumCircuit(num_qubits)
    circ.initialize(initstring,circ.qubits)
    circ.ry(np.sqrt(2)*np.pi,0)
    targetBigRho[basis_id] = qi.DensityMatrix.from_instruction(circ)

#Now the target is created, run the learner:
qr_output = QuantumRunner(targetBigRho,'SK_thm_exp').get_stored_data()
epsilon = []

target_unitary = np.array([[np.cos(np.sqrt(2)/2*np.pi),-1*np.sin(np.sqrt(2)/2*np.pi)],\
[np.sin(np.sqrt(2)/2*np.pi),np.cos(np.sqrt(2)/2*np.pi)]],dtype='complex')

plt.figure()
for episode,action_list in enumerate(qr_output[1]):
    epsilon.append(calc_errors(action_list, target_unitary))
    plt.plot(range(len(epsilon[episode])), epsilon[episode], linestyle='--')
plt.plot(range(11), Solovay_Kitaev_limit(11), label = 'Solovay-Kitaev thm')
plt.legend()
plt.xlabel('# placed gates')
plt.ylabel('Error')
plt.show()
```

### runner.py

```
from QuantumEnvironment import QuantumCircuitEnvironment

class QuantumRunner():

    def __init__(self,targetBigRho,clifford_gate_label):
        num_qubits = 1
        num_gates = 7 # not including NUL #Gates are (Ry,S,X,Y,Z) H,T
        num_samples = 1
        sig = 0.0
        sigs = [sig,sig,sig]
        gateNoiseParams = ([sigs]*num_gates)#.append(sig)



        reward_offset=1 - 1e-0
        reward_scale = 1e1
        is_reward_exp = True
        rwd_discount = 0.99

        #print(targetBigstate)
        myQuantumCircuitEnvironment = \
            QuantumCircuitEnvironment(num_qubits, num_gates,
                                      num_samples, gateNoiseParams, targetBigRho,
                                      reward_offset, reward_scale, is_reward_exp, rwd_discount)

        from tensorforce.environments import Environment
        environment = Environment.create(
            environment=myQuantumCircuitEnvironment, max_episode_timesteps=100
        )
```

```
    import json

    # Opening JSON file
    f = open('C:\\Users\\owent\\Documents\\Master Jaar 1\\Quantum Project\\JSON')

    # returns JSON object as
    # a dictionary
    hyperparameters = json.load(f)


    agent = Agent.create(agent=hyperparameters ,environment=environment
        #agent='ppo', environment=environment,
        #network=[dict(type='dense', size=5, activation='tanh')],
        #batch_size=1, learning_rate=1e-2, tracking = 'all',#memory = 200,
        #exploration=dict(type='exponential', unit='timesteps', \
        num_steps=1000, initial_value=1.0, decay_rate=0.01)
        )

    # Train for num_episodes
    num_episodes = 1000
    episode_rewards = []
    episode_actions = []
    #plt.figure()
    for _ in range(num_episodes):
        store_actions =[]
        # Initialize episode
        states = environment.reset()
        terminal = False

        while not terminal:
            # Episode timestep
            actions = agent.act(states=states)
            states, terminal, reward = environment.execute(actions=actions)
            agent.observe(terminal=terminal, reward=reward)
            #store the action taken at each step:
            store_actions.append(actions)

        #at the end of an episode:
        episode_actions.append(store_actions)
        episode_rewards.append(reward)
        #tensors = agent.tracked_tensors()

    #save agent
    agent_save_dir ='Agent_CliffordGate'+str(clifford_gate_label)
    agent.save(directory='data',filename=agent_save_dir, format='numpy', append='episodes')

    agent.close()
    environment.close()

    #all the data to return
    self.stored_data = episode_rewards, episode_actions, agent_save_dir

def get_stored_data(self):
    return self.stored_data
```

### environment.py

```
from tensorforce import Environment
import numpy as np
from scipy import linalg
from qiskit import QuantumCircuit
from qiskit.providers.aer import QasmSimulator
import qiskit.quantum_info as qi
from itertools import product



class QuantumCircuitEnvironment(Environment):
    def __init__(self, num_qubits, num_gates, num_samples, gateNoiseParams, targetBigRho, \
    reward_offset=0.5, reward_scale=1, is_reward_exp=False,reward_discount=0.99):
        '''num_qubits = int
        num_samples = int
        gateNoiseParams is a list of (mu,sigma) pairs for each gate
        For now we will assume the 1-qubit gates, when written as R_n (theta),
        have n and theta Gaussian noise
        For 2 single-qubit gates, would be ((mu_th,sig_th),(mu_n,sig_n))'''

        self.num_qubits = num_qubits # Number of qubits to simulate
        self.num_cardinal_states = 6**num_qubits
        self.num_basis_states = 2**num_qubits
        self.num_samples = num_samples # number of times to simulate a given QC to find rho
        self.num_gates = num_gates
        # this is NOT the number of distinct 1-qubit gates,
        # but is the total possible number of actions

        self.num_triangle_perstate = int(self.num_basis_states * (self.num_basis_states-1)/2)
        self.num_diag_perstate = self.num_basis_states-1
        self.num_obs_perstate = 2*self.num_triangle_perstate + self.num_diag_perstate
        self.num_obs_total_datapoints = self.num_obs_perstate * self.num_cardinal_states

        self.simulator = QasmSimulator()
        self.gateNoiseParams = gateNoiseParams
        self.action_list = []
        self.targetBigRho = targetBigRho
        self.reward_offset = reward_offset
        self.reward_scale = reward_scale
        self.is_reward_exp = is_reward_exp
        self.reward_discount = 0.9985351346308641 #copy-pasted from JSON - not best practice

        # create list of all possible basis states for tomography
        all_idx = range(self.num_qubits)
        self.cart_prod_idx = [(x,y) for x in all_idx for y in all_idx]
        for x in self.cart_prod_idx:
            if x[0]==x[1]:
                self.cart_prod_idx.remove(x)

        n_pairs = len(self.cart_prod_idx)
        assert n_pairs == self.num_qubits*(self.num_qubits-1)

        self.cart_prod_states = list(product(['0', '1', '+', '-', 'r', 'l',], \
        repeat=self.num_qubits))
        assert len(self.cart_prod_states) == self.num_cardinal_states


    def states(self):
```

```
        #return dict(type='float', shape=(2**(self.num_qubits*2+1) * self.num_cardinal_states,))
        return dict(type='float', shape=(self.num_obs_total_datapoints,))


    def actions(self):
        '''The +1 is to account for the NULL action'''
        return dict(type='int', num_values=self.num_gates+1)


    def max_episode_timesteps(self):
        '''Optional: should only be defined if environment has a natural fixed
        maximum episode length; otherwise specify maximum number of training
        timesteps via Environment.create(..., max_episode_timesteps=???)'''
        return super().max_episode_timesteps()


    def close(self):
        '''Optional additional steps to close environment'''
        super().close()


    def reset(self):
        '''state = np.zeros(self.targetBigRho.shape).flatten()
        state = np.zeros(2**(self.num_qubits*3+1))
        state[0] = 1'''

        idealCircuit = self.constructIdealCircuit()
        print(idealCircuit)

        self.action_list = []
        state, _rwd , _tmnl = self.getSampledBigRho()

        return state


    def execute(self, actions):
        '''update ideal circuit based on action
        assert len(actions) == 1'''

        self.action_list.append(actions)
        next_state, reward ,terminal = self.getSampledBigRho()

        terminal = False  # Always False if no "natural" terminal state
        if actions == 0:
            terminal = True

            #compensate for lost reward:
            remaining_steps = self.max_episode_timesteps() - len(self.action_list)
            #reward *= (1-self.reward_discount**(remaining_steps)) / (1-self.reward_discount)
            #should NOT have this if reward is given only for terminal state!

        if not terminal:
            reward = 0

        return next_state, terminal, reward


    def constructIdealCircuit(self):

        idealCircuit = QuantumCircuit(self.num_qubits)
```

```
        idealCircuit = self.constructCircuitRealisation(idealCircuit, noise=False)

        return idealCircuit


    def constructCircuitRealisation(self, noisyCircuit, noise=True):
        '''add gates from the given action list to the noisyCircuit
        action is an int between 0 and self.num_gates
        need to specify the gate to be applied and the qubit line on which to apply it
        Assume gate order is NUL, Rz1,Rz2... S1,S2,.... CNOT(1,1) CNOT(1,2)...'''

        n = self.num_qubits

        for action in self.action_list:
            if action == 0:
                self.applyNUL()
            elif action < n+1: #change 1*-->6* if you add gates back
                qbit_idx = action - 1*n -1
                self.applyH(noisyCircuit, qbit_idx, noise)
            elif action < 2*n+1: #change 2*-->7* if you add gates back
                qbit_idx = action - 2*n -1
                self.applyT(noisyCircuit, qbit_idx, noise)
            #commented below to test with JUST H and T
            '''elif action < n+1:
                qbit_idx = action - 1
                self.applyR(noisyCircuit, qbit_idx, noise)
            elif action < 2*n+1:
                qbit_idx = action - 2*n -1
                self.applyS(noisyCircuit, qbit_idx, noise)
            elif action < 3*n+1:
                qbit_idx = action - 3*n -1
                self.applyX(noisyCircuit, qbit_idx, noise)
            elif action < 4*n+1:
                qbit_idx = action - 4*n -1
                self.applyY(noisyCircuit, qbit_idx, noise)
            elif action < 5*n+1:
                qbit_idx = action - 5*n -1
                self.applyZ(noisyCircuit, qbit_idx, noise)
            '''



        return noisyCircuit


    def applyNUL(self):
        '''action corresponding to doing nothing, i.e. identity'''
        pass


    def applyR(self, circuit, qbit_idx, noise):
        '''add a noisy rotation over Z by pi/8 gate to the end of the circuit on qubit index
        qbit_idx. The noise is given by a normal distribution with standard deviation
        given by gateNoiseParams. Forms a discrete universal single qubit gate set
        with the S gate.
        U(theta,0,0) = R(theta)'''

        if noise:
            sig = self.gateNoiseParams[0]
        else:
```

```
            sig = [0,0,0]

        mu_theta = np.pi/4
        sig_theta = sig[0]
        mu_phi = 0
        sig_phi = sig[1]
        mu_lam = 0
        sig_lam = sig[2]

        theta = np.random.normal(loc=mu_theta, scale = sig_theta)
        phi = np.random.normal(loc=mu_phi, scale = sig_phi)
        lam = np.random.normal(loc=mu_lam, scale = sig_lam)

        circuit.u(theta,phi,lam, qbit_idx)


    def applyS(self, circuit, qbit_idx, noise):
        '''add a noisy S gate to the end of the circuit on qubit index qbit_idx
        the noise is given by a normal distribution with standard deviation
        given by gateNoiseParams.
        forms a discrete universal single qubit gate set with the R gate above
        U(0,pi/2,0) = S'''

        if noise:
            sig = self.gateNoiseParams[1]
        else:
            sig = [0,0,0]

        mu_theta = 0
        sig_theta = sig[0]
        mu_phi = np.pi/2
        sig_phi = sig[1]
        mu_lam = 0
        sig_lam = sig[2]

        theta = np.random.normal(loc=mu_theta, scale = sig_theta)
        phi = np.random.normal(loc=mu_phi, scale = sig_phi)
        lam = np.random.normal(loc=mu_lam, scale = sig_lam)

        circuit.u(theta,phi,lam, qbit_idx)


    def applyX(self, circuit, qbit_idx, noise):
        '''add a noisy X gate to the end of the circuit on qubit index qbit_idx
        the noise is given by a normal distribution with standard deviation
        given by gateNoiseParams.
        U(pi,0,0) = S'''

        if noise:
            sig = self.gateNoiseParams[2]
        else:
            sig = [0,0,0]

        mu_theta = np.pi
        sig_theta = sig[0]
        mu_phi = 0
        sig_phi = sig[1]
        mu_lam = 0
        sig_lam = sig[2]
```

```
        theta = np.random.normal(loc=mu_theta, scale = sig_theta)
        phi = np.random.normal(loc=mu_phi, scale = sig_phi)
        lam = np.random.normal(loc=mu_lam, scale = sig_lam)

        circuit.u(theta,phi,lam, qbit_idx)

    def applyY(self, circuit, qbit_idx, noise):
        '''add a noisy Y gate to the end of the circuit on qubit index qbit_idx
        the noise is given by a normal distribution with standard deviation
        given by gateNoiseParams.
        U(pi,pi/2,pi/2) = Y'''

        if noise:
            sig = self.gateNoiseParams[3]
        else:
            sig = [0,0,0]

        mu_theta = np.pi
        sig_theta = sig[0]
        mu_phi = np.pi/2
        sig_phi = sig[1]
        mu_lam = np.pi/2
        sig_lam = sig[2]

        theta = np.random.normal(loc=mu_theta, scale = sig_theta)
        phi = np.random.normal(loc=mu_phi, scale = sig_phi)
        lam = np.random.normal(loc=mu_lam, scale = sig_lam)

        circuit.u(theta,phi,lam, qbit_idx)

    def applyZ(self, circuit, qbit_idx, noise):
        '''add a noisy Z gate to the end of the circuit on qubit index qbit_idx
        the noise is given by a normal distribution with standard deviation
        given by gateNoiseParams.
        U(0,pi,0) = S'''

        if noise:
            sig = self.gateNoiseParams[4]
        else:
            sig = [0,0,0]

        mu_theta = 0
        sig_theta = sig[0]
        mu_phi = np.pi
        sig_phi = sig[1]
        mu_lam = 0
        sig_lam = sig[2]

        theta = np.random.normal(loc=mu_theta, scale = sig_theta)
        phi = np.random.normal(loc=mu_phi, scale = sig_phi)
        lam = np.random.normal(loc=mu_lam, scale = sig_lam)

        circuit.u(theta,phi,lam, qbit_idx)

    def applyH(self, circuit, qbit_idx, noise):
        '''add a noisy H gate to the end of the circuit on qubit index qbit_idx
        the noise is given by a normal distribution with standard deviation
        given by gateNoiseParams.
        U(pi/2,0,pi) = H'''
```

```
        if noise:
            sig = self.gateNoiseParams[0] #change 0-->5 when adding gates back
        else:
            sig = [0,0,0]

        mu_theta = np.pi/2
        sig_theta = sig[0]
        mu_phi = 0
        sig_phi = sig[1]
        mu_lam = np.pi
        sig_lam = sig[2]

        theta = np.random.normal(loc=mu_theta, scale = sig_theta)
        phi = np.random.normal(loc=mu_phi, scale = sig_phi)
        lam = np.random.normal(loc=mu_lam, scale = sig_lam)

        circuit.u(theta,phi,lam, qbit_idx)

    def applyT(self, circuit, qbit_idx, noise):
        '''add a noisy T gate to the end of the circuit on qubit index qbit_idx
        the noise is given by a normal distribution with standard deviation
        given by gateNoiseParams.
        U(0,pi/4,0) = T'''

        if noise:
            sig = self.gateNoiseParams[1] #change to 6 if add gates back
        else:
            sig = [0,0,0]

        mu_theta = 0
        sig_theta = sig[0]
        mu_phi = np.pi/4
        sig_phi = sig[1]
        mu_lam = 0
        sig_lam = sig[2]

        theta = np.random.normal(loc=mu_theta, scale = sig_theta)
        phi = np.random.normal(loc=mu_phi, scale = sig_phi)
        lam = np.random.normal(loc=mu_lam, scale = sig_lam)

        circuit.u(theta,phi,lam, qbit_idx)


    def applyCNOT(self, circuit, tmp_idx, noise):
        '''add a noisy CNOT gate to the end of the circuit on qubits tmp_idx
        noisy in the sense that we add it with probability 1-p_error
        and we don't add it with prob. p_error'''

        if noise:
            p_error = self.gateNoiseParams[-1]
        else:
            p_error = 0

        implement = np.random.choice(2, 1, p = [p_error, 1-p_error])[0]

        if implement:
            circuit.cnot(self.cart_prod_idx[tmp_idx])


    def getSampledBigRho(self):
```

```
        '''The main observation function: when run,
        gets the density matrices for each of several
        sample realisations.  It then compares these to
        the ideal, and determines the reward.

        Reward = Fidelity of sample-averaged DMs compared to ideal
        Observed = average of sampled DMs
        Terminal = false (no natural end-state, ceiling on timesteps is implemented elsewhere)
        TargetRho should be 2^n*n*n rho' = U rho U* for input rho'''

        # to store all sample DMs: is 2**n , s , n, n
        rho_samples = np.zeros((self.num_cardinal_states, self.num_samples, \
        self.num_basis_states, self.num_basis_states), dtype='complex')

        # sample-averaged DM
        rho_sample_averaged = np.zeros((self.num_cardinal_states, self.num_basis_states,\
        self.num_basis_states), dtype='complex')

        # store reward for each input cardinal state
        rewards = np.zeros((self.num_cardinal_states))

        # Loop over all basis states
        for input_state_id in range(self.num_cardinal_states):
            qbit_init_string = self.cart_prod_states[input_state_id]

            #sampling loop
            for sample_id in range(self.num_samples):
                #store each sample rho
                rho_samples[input_state_id, sample_id,:,:] = self.runCircuit(qbit_init_string)

            # average the results of all samples
            rho_sample_averaged[input_state_id] = np.average(rho_samples[input_state_id], axis=0)

            #store the reward for this basis state
            rewards[input_state_id] = self.calculateReward(rho_sample_averaged, input_state_id)

        terminal = False
        observation = self.observeRho(rho_sample_averaged)
        reward = np.average(rewards, axis=0)

        if self.is_reward_exp:
            reward = np.exp(reward)

        return observation, reward, terminal


    def observeRho(self, rho_sample_averaged):
        '''Takes as input the sample-averaged DMs, and reduces #datapoints.
        Reduction in the sense that it takes only upper right triangle of rho
        for each cardinal state,
        without last diagonal entry, plus diagonal entries only real part.
        use only upper right triangle, without diagonal, since rho is hermitian.
        Also, leave out last entry on diagonal of rho, since trace always 1
        diagonal entries only real part since rho hermitian.
        '''

        num_triangle = self.num_triangle_perstate
        num_diag = self.num_diag_perstate
        obs_perstate = self.num_obs_perstate
        obs_size = self.num_obs_total_datapoints #total number of CPLX datapts to store
```

```
        observation = np.zeros(obs_size)
        # sample-averaged DMs + number of entries reduced as far as possible
        # but flat + split into RE and IM parts
        assert num_diag+2*num_triangle==obs_perstate
        assert obs_perstate*self.num_cardinal_states == obs_size

        tridx = np.triu_indices(self.num_basis_states, k=1)

        for state in range(self.num_cardinal_states):
            diagonal = np.diagonal(rho_sample_averaged[state])
            reduced_diagonal = diagonal[:-1]
            # leave out final entry on diagonal, because trace of rho is always 1

            triangle_dm = rho_sample_averaged[state][tridx]
            # take only upper triangle of density matrix rho
            observation[obs_perstate*state: obs_perstate*state+num_diag]\
            = np.real(reduced_diagonal)
            observation[state*obs_perstate+num_diag: state*obs_perstate + num_diag + \
            num_triangle] = np.real(triangle_dm)
            observation[state*obs_perstate+ num_diag + num_triangle: \
            (state+1)*obs_perstate] = np.imag(triangle_dm)

        return observation


    def runCircuit(self, qbit_init_string):
        '''create one instance of a noisy ciruit and then calculate its density matrix
        First, initialize the circuit in one particular basis state given by binary.
        Then, call consTtructCircuitRealisation to add (noisy) gates to the circuit.
        Finally, get the density matrix of the (noisy) circuit
        (one particular instance, so always pure density matrix).'''

        init_caller = ''
        for qubit in range(self.num_qubits):
            init_caller = init_caller + qbit_init_string[qubit]

        noisyCircuit = QuantumCircuit(self.num_qubits)
        noisyCircuit.initialize(init_caller)
        noisyCircuit = self.constructCircuitRealisation(noisyCircuit)

        rho = qi.DensityMatrix.from_instruction(noisyCircuit)
        rho = np.array(rho)

        return rho


    def calculateReward(self, rhomixed, input_state):
        '''calculate the RL reward given a mixed density matrix rhomixed
        and the pure target density matrix targetrho'''

        fidelity = self.calculateFidelity(self.targetBigRho[input_state], rhomixed[input_state])
        # fidelity always between 0 and 1, is 1 if perfect match
        try:
            np.abs(np.imag(fidelity)) < 1e-10
        except:
            print(np.abs(np.imag(fidelity)))
            assert False
        fidelity = np.real(fidelity)
        reward = (fidelity - self.reward_offset) * self.reward_scale
```

```
        return reward


    def calculateFidelity(self,target_rho, realized_rho):
        '''calculate the fidelity of the realized_rho density matrix and the target_rho
        Fidelity is always between 0 and 1.
        The fidelity is 1 iff the density matrices are exactly the same.'''

        sqrt_target = linalg.sqrtm(target_rho)
        matrix1 = np.matmul(realized_rho, sqrt_target)
        matrix2 = np.matmul(sqrt_target, matrix1)
        sqrt_matrix = linalg.sqrtm(matrix2)
        fidelity = (np.trace(sqrt_matrix))**2

        return fidelity
```

### hyperparameters.json

```
    {
    "agent": "ppo",
    "network": {"type": "auto", "rnn": false},
    "use_beta_distribution": false,
    "memory": "minimum",
    "batch_size": 12,
    "update_frequency": 1,
    "learning_rate": 0.001813150053725916,
    "multi_step": 5,
    "subsampling_fraction": 0.9131375430837279,
    "likelihood_ratio_clipping": 0.09955676846552193,
    "discount": 0.9985351346308641,
    "return_processing": null,
    "advantage_processing": null,
    "predict_terminal_values": false,
    "reward_processing": null,
    "baseline": {"type": "auto", "rnn": false},
    "baseline_optimizer": {"optimizer": "adam", "learning_rate": 0.003670157218888348,
    "multi_step": 10},
    "l2_regularization": 0.0,
    "entropy_regularization": 0.0011393096635237982,
    "state_preprocessing": "linear_normalization",
    "exploration": 0.1,
    "variable_noise": 0.0,
    "max_episode_timesteps": 1000
}
```