Reinforcement Learning for Quantum Control

Alberto Chimenti[†], Clara Eminente[†] and Matteo Guida[†]

[†] *Physics of Data M.D. - Dipartimento di Fisica e Astronomia G. Galilei, Università degli Studi di Padova, Via Marzolo 8, I-35131 Padova, Italy.*

November 15, 2020

Abstract

Controlling non-integrable many-body quantum systems of interacting qubits is crucial in many areas of physics and in particular in quantum information science. In the following work a Reinforcement Learning (RL) algorithm is implemented in order to find an optimal protocol that drives a quantum system from an initial to a target state in two study cases: a single isolated qubit and a closed chain of L coupled qubits. For both cases the obtained results are compared with the ones achieved through Stochastic Descent (SD). What has been found is that, for a single qubit, both methods find optimal protocols whenever the total protocol duration T allows it. When the number of qubits increases RL turns out to be more flexible and to require less tuning in order to find better solutions. We also find that both algorithms capture the role of T. The work is based on some of the results obtained in [1].

1 Theory

One of the possible ways to manipulate a system of spin-$\frac{1}{2}$ qubits in order to reach a desired target state $|\psi_{target}\rangle$ from an initial state $|\psi_{start}\rangle$ is to use a **static** magnetic field and a time dependent one labeled as **control field**.
In this scenario a protocol of duration T is defined as a sequence of values of the control field, labeled as $\{h(t)\}_T$, given the static field. We will restrict our choice to the so called "bang-bang" protocols, i.e. the control field will only take two discrete opposite values.
The quality of the protocol is measured by means of the fidelity between the final quantum state and the target one. This quantity, for pure states, is defined as follows:

$$F = |\langle\psi_{final}|\psi_{target}\rangle|^2 \tag{1}$$

See appendix for a more general definition [A]. The goal in quantum optimal control is to optimize a cost function aiming to efficiently find high quality solutions. The protocol is said to be optimal if the reached state coincides with the target state, hence it has fidelity equal to 1. If the final result is fairly adequate, the protocol is called sub-optimal. In the following natural units are used setting $\hbar = c = 1$.

1.1 Quantum States

The first treated quantum mechanical model is a two-state single qubit. In particular we considered a spin-$\frac{1}{2}$ particle in a **static** magnetic field $\mathbf{B}_1 = B^z$ oriented along the z-axis. The **control field** is represented by $\mathbf{B}_2(t) = B^x(t)$, oriented along the x-axis.
Hence, the Hamiltonian writes:

$$H(t) = -\gamma\mathbf{B}(t)\cdot\mathbf{S} = -\gamma B^z S^z - \gamma B^x(t) S^x \tag{2}$$

where γ is the nuclear gyromagnetic ratio, $S^a = \frac{1}{2}\sigma^a$ are the spin-1/2 operators with σ^a the Pauli matrices and $\omega_0 = \gamma B^z$ is the Larmor frequency. Those parameters are set as $\omega_0 = 1$ a.u. and $h^x(t) = \gamma B^x(t)$, in order to be consistent with the notation in [1]. The final Hamiltonian reads:

$$H(t) = -S^z - h^x(t) S^x \tag{3}$$

In the second part of the work a closed chain of L coupled qubits is considered with periodic boundary condition, i.e. $S_{L+1} = S_1$. Besides the previous contributions for each spin, we introduce the interactions among spins. These are assumed to occur between nearest-neighbors, only in the z-direction and all with the same strength J. In order to avoid phase transitions in the system the following values of the parameters are chosen: $J = \omega_0 = 1$, consistently with [1].
The Hamiltonian for N-coupled qubits reads:

$$H(t) = -\sum_{j=1}^{N}\left(S_j^z S_{j+1}^z + S_j^z + h^x(t) S_j^x\right) \tag{4}$$

where the following implicit notation is used:

$$S_j^z S_{j+1}^z = \underbrace{\mathbb{1}_2 \otimes \ldots \otimes \mathbb{1}_2}_{j-1} \otimes S_j^z \otimes S_{j+1}^z \otimes \underbrace{\mathbb{1}_2 \otimes \ldots \otimes \mathbb{1}_2}_{N-j-1}$$

in the spin-spin interaction terms and

$$S_j^a = \underbrace{\mathbb{1}_2 \otimes \ldots \otimes \mathbb{1}_2}_{j-1} \otimes S_j^a \otimes \underbrace{\mathbb{1}_2 \ldots \otimes \mathbb{1}_2}_{N-j}$$

in the spin-magnetic field interaction terms.

1.2 Time evolution function

Evolving a quantum state implies solving the time-dependent Schrödinger equation for the system. To maximize the reliability of the results obtained for the optimal protocols we decided to employ the **spectral method** for computing time evolution.
Evolving a state ψ from time t_0 to time t is done by exactly diagonalizing the Hamiltonian, getting its eigenstates $|E_i\rangle$ and eigenvalues E_i and computing:

$$|\psi(t)\rangle = \sum_{i=1}^{2^L} e^{-iE_i(t-t_0)} \langle E_i|\psi(t_0)\rangle |E_i\rangle \tag{5}$$

As the dimension of the Hilbert space describing a many-body quantum system scales exponentially with the system size L the associated Hamiltonian is represented by a matrix of size $2^L \times 2^L$. In light of this, the details of the choice of an exact method and its limits are explained and discussed in section 4.1.2.

2 Reinforcement Learning

Reiforcement Learning (RL) is a learning paradigm in ML in which an agent learns to maximize some reward function by "exploring" a given environment. An RL agent interacts with its environment in discrete time steps. At each time t, the agent receives the current state s_t and reward r_t. It then chooses an action a_t from the set of available actions, which is subsequently sent to the environment. The environment moves to a new state s_{t+1} and the reward r_{t+1} associated with the transition (s_t, a_t, s_{t+1}) is determined.
The computational model revolves around the definition of the "state-action" space of the problem over which a Markov Decision Process is defined. The task of the algorithm then is to learn the transition probabilities associated with the latter. So, starting from a given initialization of such probabilities, the agent gradually adjusts their value in order to maximize the obtained reward.

In this case the optimization is performed in order to maximize the *fidelity* (1) of the final quantum state, which represents the chosen *reward function* of the environment.

The degrees of freedom of the learner define the action space and in this case are associated with the possible values assumed by $h^x(t)$. Each state is then represented by the pair $(t, h^x(t))$, respectively the time index of the protocol and the field value.

The objective, then, is for the learner to be able to output a protocol, i.e. a succession of magnetic control field values, which is supposed to lead the starting quantum state as close as possible to the target one in a given simulation time duration.

2.1 Q-learning Algorithm

The learning paradigm of the algorithm is built over the the definition of *Q-value* introduced in Appendix (B). The simplest implementation of it is on-policy and uses a *greedy* (i.e. chooses the one with maximum probability) behavioural policy $\pi(a|s)$.

1. The agent starts in a given state with a given starting action.
2. Chooses the next action by means of the chosen $\pi(a|s)$.
3. Moves to the next state through the chosen action and computes the reward associated with it.
4. The Q-value associated with the visited state-action pair is updated using bootstrap method with the update rule (11).
5. Repeat from point 2 for the desired number of steps or until the final goal is reached.

Each iteration of the above mentioned procedure represents an *episode*. After a certain amount of episodes, the Q-table is expected to converge and learn the state-action space.

It is worth to mention the fact that in order to "learn the space", a completely deterministic approach to the policy such as the greedy one is counterproductive since it is going to trap our agent in the first encountered local minimum. In these kind of optimization problems it is crucial to find a good trade-off between explorability and exploitability, i.e. being able to explore improbable states and promote the most probable pathways to reach convergence of the Q-table. Commonly used probability distributions over the Q-values ($\pi(a|s)$) are *softmax* and ϵ-*greedy*.

2.2 Eligibility Traces

The algorithm implementation slightly differs from the usual structure that an RL agent is supposed to have. The aim is to build an agent that can learn the optimal protocol with no actual information about the quantum system. Therefore the tricky concept is that in our description of the problem, the actual states of the system to be explored and learnt are represented by entire protocols instead of single time-action pairs. This will probably become more clear in section (4). Moreover, since on the physical point of view any transition path is equivalent as long as it leads to the same final fidelity, the rewards are computed only at the end of each episode.

This leads to a challenging learning process which was decided to be tackled with the use of *eligibility*

traces. This is an improvement of the temporal difference algorithm using Q-learning approach. The main concept behind it is to find a way to propagate the delayed reward to the previously visited states in the update procedure.

The *eligibility trace* is defined by the update rule and boundary condition:

$$E_0(s) = 0 \tag{6}$$

$$E_t(s) \leftarrow \gamma\lambda E_{t-1}(s) + \mathbb{1}(S_t = s) \tag{7}$$

where γ is still the *discount factor*, λ is a parameter which controls the importance of future states over older ones and $\mathbb{1}$ is the indicator function. Therefore one rewrites the Q-table update rule (11) as:

$$Q_t(s) \leftarrow Q_t(s) + \alpha\delta_t E_t(s) \tag{8}$$

Since our goal is to converge to an optimal protocol instead of a mix of states belonging to pseudo-optimal ones, in a practical sensed implementation, the eligibility trace should be unique to the episode. This will be clearly explained in Code Development.

3 Stochastic Descent

To benchmark the results obtained using RL and tackle the problem from a different perspective we employed **stochastic descent** (SD) as an alternative method. The idea is to map each protocol $\{h(t)\}_T$ into a chain of classical ± 1 spins and to search for the configuration that minimizes a cost function, which in our case can be easily identified with the **infidelity**, i.e. $I(\{h(t)\}_T) = 1 - F(\{h(t)\}_T)$.
This different point of view allows us to both compare the results obtained via RL with those obtained with a more basic approach and to highlight a different side of this problem: the role of the total protocol duration, T.
We limit the field values to only two values, $h^x(t) \in \{-4, 4\}$, and starting from a random configuration of spins (i.e. a random protocol) perform a *greedy* $1-flip$ *stochastic descent* (as in [1]). Note that this is basically equivalent to finding the ground state of a classical Ising model where the energy landscape is represented by the infidelity function.
We run SD multiple times at fixed protocol duration starting from random protocols and storing the final best protocols $\{h_x^\alpha(t)\}_{\alpha=1}^{N_{trials}}$ and their fidelity. Details of the algorithm can be found in the following section.

3.1 Phases of quantum control

Mapping the problem in this way one can explore the landscape of minima of the infidelity, represented by the protocols output at the end of each SD run.
As mentioned before, in this framework our problem is analogous to that of finding an optimal spin configuration where interactions between spins in the chain (i.e. the fields at each step of the protocol) are very complicated: something similar happens in systems of spin glasses. In analogy, we thus define the correlator $q(T)$:

$$q(T) = \frac{1}{16N_T}\sum_{n=1}^{N_T}\overline{(h_x(n) - \overline{h_x(n)})^2} \tag{9}$$

with N_T the number of steps in the protocol of total duration T. $\overline{h_x(n)} = \frac{1}{N_{trials}}\sum_{\alpha=1}^{N_{trials}} h_x^\alpha(n)$ is the value of the field at the n-th step of the protocol averaged over all the best protocols found for a given protocol duration T. The correlator $q(T)$ spans from 0 (i.e. all protocols are the same) to 1 (i.e. all the protocols with best fidelity are different). If $q(T) = 0$ the problem is convex and only one optimal configuration exists; if $q(T) = 1$ many different equivalent minima of the infidelity exist.

4 Code Development

In this section we introduce an overview of the modules composing the algorithm and illustrate the implementation choices which have been made.

4.1 Quantum Model

RL and SD share the same "physical" computations. We now offer a brief overview on the implementation choices made to tackle the physical part of the problem, which mainly revolve around the choice of the time evolution function.

4.1.1 Computing Eigenvalues and Eigenvectors

For relatively large system sizes L diagonalizing and computing spectral components of an Hamiltonian can be computationally expensive. In principle we would need to compute eigenvalues and eigenvectors every time the quantum state is evolved in order to apply the spectral method. To avoid heavy computational bottlenecks in this sense we wanted to exploit the fact that we deal with a very limited discrete number of previously known field values. We thus precompute eigenvalues and eigenvectors for each field value at the beginning of the learning and store them, so that they can be accessed when needed.

4.1.2 Time Evolution

As mentioned in section (1.2) time evolution is implemented using the spectral method. The function evolves the current quantum state for a single time step $dt = T/N_T$ in the protocol by accessing the precomputed eigenvalues and eigenvectors of the Hamiltonian corresponding to the istantaneous field value. It then computes all the quantities in eq. (5) using built-in numpy functions for fast and efficient computations. The function also checks if the norm of the state is conserved during the evolution.
We care to specify that in our case we knew we would not have the need to deal with large system sizes, for

which we know approximated methods are necessary; the small value of L played an important role in our choice of the time evolution function and its implementation.

4.2 Reinforcement Learning

We summarize in the following, the most important aspects of the implementation of a Watkins's Q-learning algorithm with eligibility traces and softmax behavioural policy.

4.2.1 Environment

As explained before, an important step is to efficiently map the state space of our quantum model into the state-action space seen by the agent. The *environment* module is responsible of carrying the information about the state of the agent and translate it from one space to the other. Each state in the agent space corresponds to one step in the protocol and it is idexed as:

```
state_idx = tstep_idx*len(all_actions) +
    action_idx
```

Where `all_actions` is simply a tuple containing all the possible magnetic field values corresponding to all the actions that the agent can cast. Through such mapping, given an action, the environment moves to the next state by means of the method `move`.

4.2.2 Action Selection and Replay

The agent implements two different behavioural policies for action selection.

- **softmax**:

$$\pi(a|s) = \frac{e^{Q_a(s)/\epsilon}}{\sum_a e^{Q_a(s)/\epsilon}}$$

- **ϵ-greedy**:

$$\pi(a|s) = \frac{\epsilon}{N_{actions} - 1}$$
$$\pi(a_{greedy}|s) = 1 - \epsilon$$

where

$$a_{greedy} = argmax_a \left(Q_a(s)\right);$$

The ϵ parameter serves as a temperature parameter and to tune greediness. Such parameter is crucial to tune the learning and make the Q-table gradually converge to the optimal protocol. Its tuning is better explained in Section (4.2.4).

Here we contextualize the role of the eligibility trace and its use. If the action selection leads to an action choice different to the completely greedy one, we break the learned path by resetting the trace. Therefore, the trace only tracks down the deterministic parts of the protocol without biasing past optimal paths with purely explorative ones.

Replay Episodes

As we will clearly show in the Results, the agent reaches very high rewards during the very first episodes. In order to consolidate the experience of these protocols, we decided to make the algorithm able to replay them for a given amount of iterations. This biases explorability in the successive episodes and leads the agent to choose less "randomized" protocols in the future. Such procedure is important when dealing with an environment like ours. Recall that what we want our agent to learn is a full protocol, corresponding to one final quantum state, not just assigning higher probabilities to part of its steps.

4.2.3 Q-table Update

The `update` method implements the update rules (6) and (8). It checks at every call whether the last time step is reached and, if so, computes the reward associated with the final state, which otherwise is always set to zero. We decided to use a completely greedy policy to compute the "estimate of optimal future value" in (11). Since we need quite a deterministic learning, this seemed the most suitable choice. The behavioural policy which performed best was the softmax.

4.2.4 Convergence: Reward-Based-ϵ-Decay

Tuning of the greediness of the algorithm depending on the learning stage was decisive for convergence. Since our plan was to make the agent flexible to different quantum model initialization (e.g. variable number of qubits), we decided to implement a reward based method to update the greedy parameter. Given an initial and a final value for ϵ, what it does is:

- check whether the moving average reward is above a certain threshold, if so:
- update the threshold by adding 10% of the difference between the old value and the new average reward
- update the ϵ value as:

$$\epsilon = \epsilon_f + (\epsilon_i - \epsilon_f)\, e^{-T\frac{episode\_idx}{N_{episodes}}}$$

This check is performed every 20 episodes. To avoid getting stuck with an homogeneous table after a given number of check failures, the ϵ value is forced to update.

4.3 Stochastic Descent

We now offer an overview of the implementation choices of the SD algorithm.
At each iteration one of the spins is flipped, the initial quantum state is evolved according to the new protocol and if the final infidelity is lower that the one obtained without the trial flip the new configuration is kept. It's important to notice that this kind of implementation does not guarantee to find the optimal

protocol, as the exploration capability of the algorithm is very low: flipping one spin at a time and accepting only actions which instantly increase the fidelity may result in the algorithm getting stuck in local minima. Moreover, we are aware that this kind of optimization problems are usually tackled using Metropolis and Wolff algorithms, mainly to accelerate convergence, especially for larger spin systems (which in our case would correspond to longer protocols). We decided to follow [1] to be able to reproduce the same results and because, as we will show, this choice allows us to implement a completely parameter free algorithm with surprisingly good time performances. However, as the complexity of the problem increases (i.e. for L>1) we will show the limits of this choice.
The stochastic descent algorithm starts by preparing a random protocol in the form of an array of size N_T whose entries take values ± 4 with uniform probability. A list of possible flips is generated: if just 1 flip at a time is allowed this will be merely represented by a list of indexes from 0 to $N_T - 1$. If n flips are allowed the list contains all possible combinations of at most n indexes from 0 to $N_T - 1$ (without repetitions). The list of possible flips is shuffled and the algorithm starts by iteratively checking over the possible flips: once the flip is applied the new configuration is kept if, after evolving the starting state according to the new protocol, the infidelity is lowered. Otherwise, we move to the next combination in our flip list. If the flip is accepted the list of possible flips is reshuffled and the iteration restarts. If the whole list is scrolled through and no flip is accepted it means that no matter what we try to change, the maximum *possible* fidelity was reached and the algorithm stops.
Running the code we noticed that sometimes the starting protocol leads the quantum state so far from the target one that, given its very poor exploration capability, the algorithm cannot improve it, even though the total protocol duration would in principle allow us to find a nearly optimal protocol. In this cases the algorithm would return the initial fidelity between the starting state and the target. To avoid this problem, when the whole flip list has been scrolled through and there has been no improvement in fidelity, the algorithm restarts from a new random protocol.

4.4 Implementation Remarks

The choice of the time discretization may affect the precision of the algorithm when approximated methods are used e.g. in time evolution. As we decided to employ spectral method for time evolution this was not our case. We then checked fidelity performances using SD both fixing dt and the protocol length, obtaining absolutely comparable results. We eventually chose to use fixed protocol lengths for two main reasons:

- *Fine tuning RL hyperparameters*:
  as the explorable state-action space of the RL algorithm increases as:
  $$\frac{N_T!}{N_{actions}!(N_T - N_{actions})!}$$
  fixing dt (i.e. making protocols longer when T increases) requires a lot of fine tuning at different protocol lengths T in order to get the Q-table to converge and make the learning successful. As we wanted to compare performances for different T, fine tuning for every T was infeasible. Moreover, note that in case RL would be applied to real life experiments T will probably be known and set in advance, making it possible to tune the learning accordingly.
- *Comparison using $q(T)$*:
  to tackle the investigation of the role of protocol duration we also computed $q(T)$ (see paragraph 3.1). Given its definition, it looked reasonable, in order to compare $q(T)$ values at different times, for protocols to have the same length.

5 Results

Figure 1 and 2 show the reward obtained by the RL agent at the end of each episode as the learning moves on. It is possible to note that the trade off between exploration and exploitation: the learner explores protocols with very high final fidelity very early in the training but also a lot of lower fidelity paths. As the training moves on the choices become more greedy (as ϵ decreases), exploration is less probable and convergence is reached at a point where the learning becomes almost deterministic (i.e. a single final path is found). The method employed to update the greediness of the algorithm is the one that achieved best results in both reaching the desired final state and generalizing over different simulation parameters for the quantum model initialization (e.g. T or L). Yet the temperature in the exponential decay in (4.2.4) was arbitrarily chosen and set to 8 in the plots showed.
Observing the two plots one can note there is a difference in the span of the obtained rewards: for $T = 1$ not only they never reach 0 but they also appear to never be worse than the initial one (which is approximately 0.2 for our choice of $|\psi_{start}\rangle$ and $|\psi_{target}\rangle$). This is purely due to the choice of the initial and target states. In our case the static field alone moves the initial state in a favorable direction. In order to show intuitively what is happening we take into consideration the one qubit case along with its representation in the Bloch sphere. For lower values of T the control field has very little influence on the evolution which is mainly "dominated" by the static one. Hence, the state is not able to move significantly (only by means of the contribution of the static field) and the worst reachable quantum state in terms of final fidelity seems to be the starting state itself. On the other hand, for higher values of T after a certain protocol duration (which we will analyze later) the role of the control field becomes crucial in adjusting

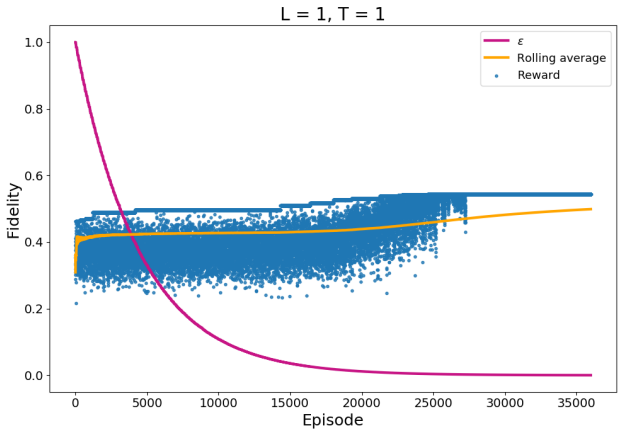

Figure 1: Episode fidelity of the RL agent during the training procedure. ($\alpha = 0.9$, $\lambda = 0.8$, $\gamma = 1$, action_list $= \{-4, 4\}$)

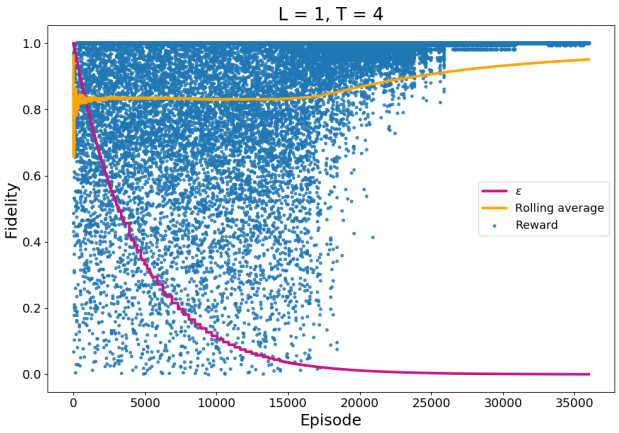

Figure 2: Episode fidelity of the RL agent during the training procedure. ($\alpha = 0.9$, $\lambda = 0.8$, $\gamma = 1$, action_list $= \{-4, 4\}$)

the final quantum state. The static field tends to bring the final state past the target and in the first phases of the training, when the agent is still learning, this can lead to very "bad" final states. What the agent eventually learns is how to adjust the protocol in order to compensate the static field.

Figure (3) and (4) show the final fidelity obtained for $L = 1$ and $L = 3$ at different values of the total protocol duration T.

The value of the starting fidelity at T=0 is determined by the choice of the initial $|\psi_{start}\rangle$ and targhet $|\psi_{target}\rangle$ states which are the ground states at $h_x = -2$ and $h_x = 2$ of (4). This decision was taken in order to better compare the obtained results with [1].

The first behaviour that can be observed in both plots is the increase of the obtained fidelity with the time duration of the protocol. As one can imagine, this parameter must play an important role in the control procedure: intuitively if T is too small and the control field is finite the quantum state does not have enough time to reach the target. We underline that the dependency on T must be a physical characteristic of the system, as the same behaviour is found using different approaches as RL and SD.

Focusing on Fig. 3 it is possible to note that the fidelity eventually reaches a stationary stage where it is equal to 1 and the corresponding protocols are said to be optimal.

In this case optimal fidelity is reached after a point in time which may be identified with the *quantum speed limit* (T_{QSL}). An exhaustive analysis of this phenomenon and its meaning is beyond the scope of this work and we limit ourselves to underline how it is not possible to reach optimal fidelity for shorter protocol times.

From an intuitive point of view, what happens after T_{QSL} is that the protocol has always enough time to adjust itself, to a point where an infinite number of optimal protocols exists. This intuition is reflected and supported by the value of $q(T)$.

At the beginning $q(T)$ takes very small values (almost zero for the first values of T): as we previously mentioned this means that there exist a single protocol that leads to the maximum achievable fidelity. The growth of $q(T)$ suggests that the number of infidelity minima, corresponding to the maximum achievable fidelity, increase accordingly. This behaviour reaches a saturation value which basically corresponds to finding different protocols each time. As we increase the simulation time, the number of optimal solutions potentially tends to infinity. Given the trend found in $q(T)$ and how it was introduced, it can be identified as an order parameter of a spin-glass-like system.

The value of $q(T)$ was computed running SD multiple times for each protocol length T. In principle, given how the algorithm was implemented, one could argue that the value of $q(T)$ could be affected by the fact that SD gets stuck in local minima. Since, however, we find almost the exact same fidelity values using RL (which has way better exploration capabilities) this scenario seems unlikely.

Moving to Fig. 4 it is possible to notice how the two different methods lead to perfectly comparable results. However it is worth noting that in this case we had to allow 2 flips at a time in SD to achieve these results. By allowing just 1 flip the algorithm has not enough explorability in the protocol space to move adequately in the infidelity landscape and to find the infidelity minima corresponding to the maximum achievable fidelity for the considered protocol time T.

This behaviour is due to the increasing complexity of the problem as the dimension of the Hilbert space grows and can be observed in Fig. 5. It is possible to note that, especially for higher T, the variance of the found fidelity is much higher, which is due to the algorithm getting trapped in local minima in some iterations. In general we noted that this behaviour worsens for high values of T as L grows and it is actually also present in Fig.4 since, as we previously mentioned, allowing 2 flips is neither an optimal nor ultimate solution to the problem.

We finish by underlining how, graphically, it is no more

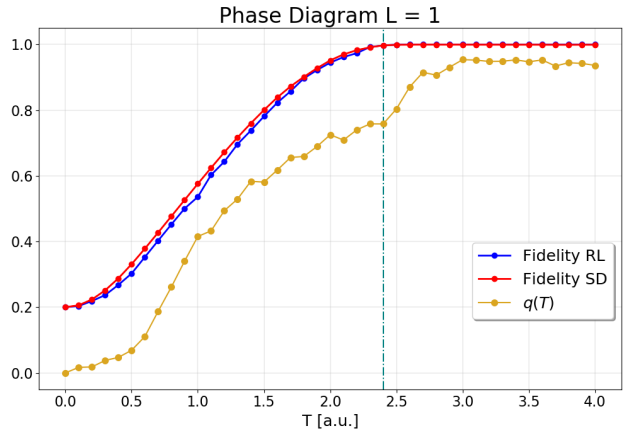

Figure 3: Obtained fidelity for different protocol duration T with fixed $N_T = 100$. SD was run with 1-flip and the results are averaged over $N_{trials} = 20$. The hyperparameters for RL are the same as in Figure (1-2)

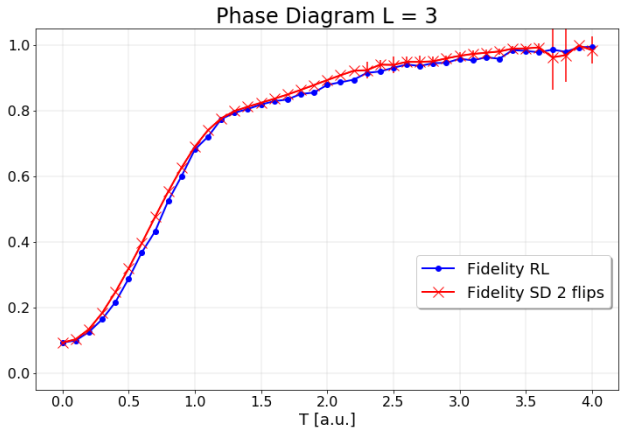

Figure 4: Obtained fidelity for different protocol duration T with fixed $N_T = 100$. Here, we confront the results for SD with 2-flips and RL. The parameters are the same as in Figure (3). The SD results are averaged over $N_{trials} = 20$.

possible to identify T_{QSL}, i.e. a point beyond which the fidelity is permanently 1.

6 Self-evaluation

- **Correctness**: both RL and SD are able to find optimal protocols (when the value of T allows it) for $L = 1$. As we showed, the chosen implementation of SD does not guarantee for the best protocol to be found for $L > 1$, whereas RL manages to give adequate results without compromising the computational cost. However, both methods manage to capture the physics behind the problem, as the dependency on T emerges using both methods.
  Besides the correctness of the final results, as previously mentioned, the conservation of the norm is checked along with the correct convergence to the final learned path in the Q-learning algorithm, in order to highlight possible inaccuracies.

- **Numerical stability and accurate discretization**: the main issue that might emerge is when some form of discretization is performed when using approximated methods for time evolution. Since we employed an exact method we do not expect to incur in such problems.

- **Flexibility**: the most interesting part of using RL for tackling quantum control problems is that the algorithm is completely model free. This fact has two main advantages.
  Firstly, the algorithm has no knowledge of the quantum system and its complications: this became clear when we successfully trained the algorithm for $L > 1$ without any further effort compared to the $L = 1$ case. As a matter of fact, the computational complexity of the training procedure (excluding the time evolution of the quantum state) does not depend on the Hilbert space dimension, but only on the chosen time discretization. Therefore, finding the optimal protocol for a N-qubit system is potentially equivalent to solving just a simple one qubit case. Secondly, this means that in principle the same procedure we used to learn an optimal protocol can be used to tackle very different scenarios: one would just need to substitute the fidelity in the reward function with a quantity adequate for the new problem.
  In this sense, SD is a less flexible method but as long as the problem can be mapped into finding the ground state of a chain of spins, the implementation is the same. However, as we underlined multiple times, when the space dimension becomes larger the current implementation lacks the exploration capability to optimally perform. We proposed some adjustments but these come either at very high efficiency costs (allowing more flips) or needing some hyper-parameters tuning (Metropolis, Wolff).

- **Efficiency**: we noticed that RL and SD present different computational bottlenecks.
  As we showed, in case of SD, when the system size increases a single flip is not enough to obtain good results but if multiple flips are allowed the number of accessible trial protocols becomes very large and runtimes become inefficiently long (i.e. the increment in precision, doesn't balance the computational tradeoff). Remarkably, the just explained inefficiency impacts the computational runtime much more that the quantum state time evolution. In these cases it would be probably better to prefer algorithms such as Metropolis and Wolff which require some fine tuning of the parameters but will probably be more efficient in leading

to optimal results.
As for RL, the main bottleneck turned out to be time evolution of the quantum state. Even though diagonalization of the Hamiltonian is not performed each time the state is evolved, the size of the matrices to deal with (e.g. the number of eigenvectors) grows as 2^L and this impacts runtime when the quantities in (5) are computed. Exploiting the python built-in methods for matrix-vector multiplication turned out to increase the efficiency of the process but we reckon that for larger systems approximated methods would be preferable.

As previusly mentioned in the *Flexibility* comments, it seems that the possibility of solving optimization problems over quantum system is actually limited by the need of simulating the system itself.

We forward the attention to the appendix section (D) for further analysis about the scalability with system size and profiling of the training function for RL.

References

[1] M. Bukov, A. G. R. Day, D. Sels, P. Weinberg, A. Polkovnikov, and P. Mehta, Reinforcement learning in different phases of quantum control, *Phys. Rev. X* **8**, 031086 (2018).

Appendices

A Fidelity of Quantum States

Given the density matrices describing two quantum states ρ and σ the *fidelity* describes the closeness of such states and it is defined as:

$$F(\rho,\sigma) = \left[Tr\sqrt{\sqrt{\rho}\sigma\sqrt{\rho}}\right]^2$$

It is symmetric and in case of pure states, it is equivalent to the expression (1).

Proof:
Namely, $\rho = |\psi_\rho\rangle\langle\psi_\rho|$ and $\sigma = |\psi_\sigma\rangle\langle\psi_\sigma|$

$$\begin{aligned} F(\rho,\sigma) &= \left[Tr\sqrt{|\psi_\rho\rangle\langle\psi_\rho|\sigma|\psi_\rho\rangle\langle\psi_\rho|}\right]^2 \\ &= \langle\psi_\rho|\sigma|\psi_\rho\rangle \left[Tr\sqrt{|\psi_\rho\rangle\langle\psi_\rho|}\right]^2 \\ &= \langle\psi_\rho|\sigma|\psi_\rho\rangle \\ &= |\langle\psi_\rho|\psi_\sigma\rangle|^2 \end{aligned}$$

□

B Introduction to Q-learning

We denote A_t as the set of actions that the agent can take at time t; $\mathcal{S}$ is the set of states that the agent can occupy and R_t is the reward at time t.

The agent tries to maximize the *Long-Term Expected Reward*:

$$G_t = \sum_{k=0}^{\infty} \gamma^k R_{t+1+k} \qquad \gamma \in [0,1)$$

One can define the *value function* of the state as the expected reward in the state:

$$\begin{aligned} v_t(s) &= \mathbb{E}\Big[G_t | S_t = s\Big] && \text{(Bellman Equation)} \\ &= \mathbb{E}\big[R_{t+1} + \gamma v(S_{t+1}) | S_t = s\big] && (10) \end{aligned}$$

γ is called *discount factor* and it represents the foresightedness of the agent. Solving such equation needs the total knowledge of the agent's policy:

$$\pi(a|s) = \mathbb{P}\big[A_t = a | S_t = s\big]$$

this is known as agent *behavioural policy* and it selects the next action given the state. Unfortunately 10 is exponentially complex to solve for S. For this reason we approximate the value function with an estimate called *Q-value* which has the following update rule:

$$Q(s_t,a_t) \leftarrow Q(s_t,a_t) + \alpha \underbrace{\Big(\overbrace{r_t}^{\text{reward}} + \underbrace{\gamma}_{\text{discount factor}} \cdot \overbrace{max_a Q(s_{t+1},a)}^{\text{estimate of optimal future value}} - \underbrace{Q(s_t,a_t)}_{\text{old value}} \Big)}_{\delta_t} \tag{11}$$

Therefore, the *Q-learning algorithm* is model-free and uses *bootstrapping* technique to estimate the value function starting from a default initial estimate. The agent's optimal future value estimator of this algorithm is called *greedy update policy*. It must not be confused with the behavioural policy; if the two policies coincide, the learning algorithm is called *on-policy*. There is various behavioural policy options, among which, as previously mentioned, the most common are ϵ-*greedy* and *softmax*. As we can tell, the use of one or the other yields some differences on the explorability of the state space.

C Further Comments on SD

In Fig. 5 it is possible to note that for lower values of the total protocol duration T we obtain the same values of fidelity using 1 flip and 2 flips as the control field has little impact on the evolution of the state, as explained in detail in section (5). For longer protocol durations the role of the control field is no more marginal and the explorability limits related to flipping one spin at a time emerge. If we compare the results for 1 and 2 flips, allowing just 1 flip results in lower fidelity and larger standard deviation. This is due to the algorithm frequently falling into local minima.

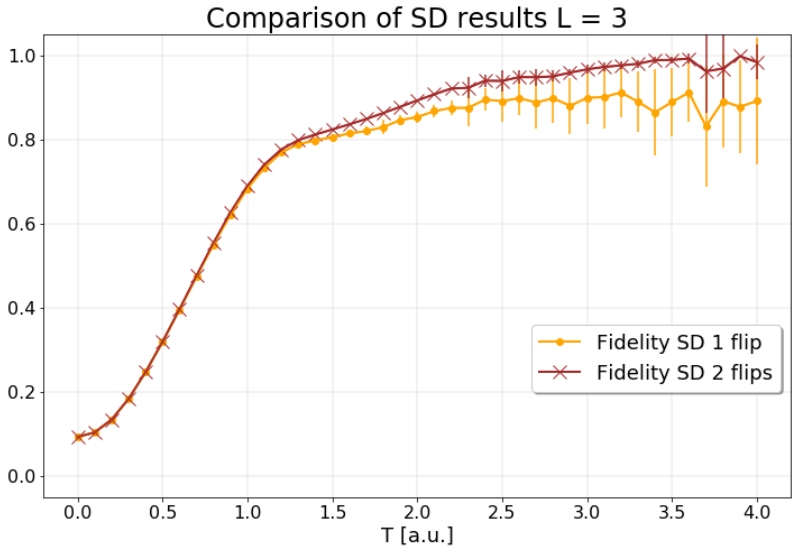

Figure 5: Comparison of rewards obtained using SD for L=3 when allowing 1 or 2 flips. The data refers to 20 repetitions for each protocol duration T.

D Training Profiling and Scalability with L for RL

The following plot shows the scaling of the training procedure computation time as a function of the system size L:

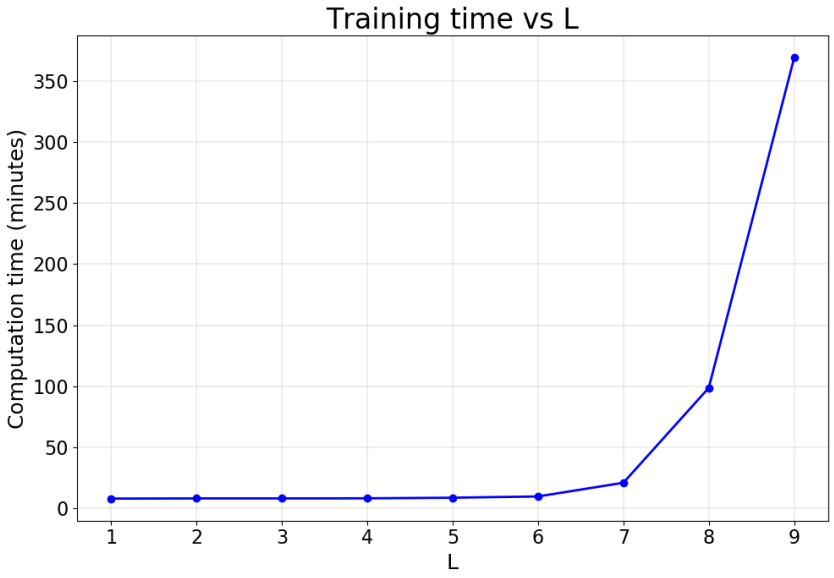

Figure 6: Total RL agent training runtime as a function of the system size L at fixed protocol lenght

Finally, we show a representative example of the computational runtime profiling of the agent class method `train_episode` obtained using a customized profiling python decorator (*profiler_decorator.py*). We report the

stdout of the code obtained by running the algorithm for $L = 9$ and $N_T = 100$. It is important to notice that among all of the results listed, the most important contribution comes from `evolve` (highlighted in light blue). Such class method takes 0.016 s per call which have to be multiplied by the number of iterations contained in each episode (N_T). Therefore even considering the limited precision of these results by comparing the total runtime per call of `train_episode` one can conclude that a major contribution is due to the quantum time evolution.

```
   Ordered by: function name

   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
     4767    0.003    0.000    0.003    0.000 {built-in method builtins.getattr}
     2397    0.002    0.000    0.002    0.000 {built-in method builtins.isinstance}
     8753    0.008    0.000    0.008    0.000 {built-in method builtins.issubclass}
      806    0.001    0.000    0.001    0.000 {built-in method builtins.len}
     2395    0.024    0.000    0.024    0.000 {built-in method numpy.array}
11152/6385    5.478    0.000    5.701    0.001 {built-in method numpy.core._multiarray_umath.
    implement_array_function}
     3196    0.002    0.000    0.002    0.000 {built-in method numpy.geterrobj}
     1598    0.005    0.000    0.005    0.000 {built-in method numpy.seterrobj}
      544    0.006    0.000    0.006    0.000 {built-in method numpy.zeros}
     3187    0.007    0.000    0.007    0.000 fromnumeric.py:74(<dictcomp>)
     2409    0.002    0.000    0.002    0.000 {method 'append' of 'list' objects}
     2388    0.214    0.000    0.289    0.000 {method 'choice' of 'numpy.random.mtrand.
    RandomState' objects}
        9    0.000    0.000    0.000    0.000 {method 'disable' of '_lsprof.Profiler' objects}
     1589    0.009    0.000    0.009    0.000 {method 'flatten' of 'numpy.ndarray' objects}
     1589    0.006    0.000    0.006    0.000 {method 'format' of 'str' objects}
     1598    0.001    0.000    0.001    0.000 {method 'get' of 'dict' objects}
     3187    0.002    0.000    0.002    0.000 {method 'items' of 'dict' objects}
     1589    0.005    0.000    0.005    0.000 {method 'nonzero' of 'numpy.ndarray' objects}
      799    0.001    0.000    0.001    0.000 {method 'pop' of 'dict' objects}
     3984    0.848    0.000    0.848    0.000 {method 'reduce' of 'numpy.ufunc' objects}
      797    0.013    0.000    0.799    0.001 {method 'sum' of 'numpy.ndarray' objects}
     2388    0.005    0.000    0.005    0.000 {method 'transpose' of 'numpy.ndarray' objects}
      799    0.003    0.000    0.025    0.000 _ufunc_config.py:441(__enter__)
      799    0.003    0.000    0.011    0.000 _ufunc_config.py:446(__exit__)
      799    0.004    0.000    0.004    0.000 _ufunc_config.py:437(__init__)
        9    0.000    0.000    0.000    0.000 environment.py:14(__init__)
     1598    0.007    0.000    0.008    0.000 getlimits.py:365(__new__)
     2388    0.001    0.000    0.001    0.000 fromnumeric.py:2546(_amax_dispatcher)
     1589    0.001    0.000    0.001    0.000 numeric.py:533(_argwhere_dispatcher)
      799    0.017    0.000    0.024    0.000 _util.py:200(_asarray_validated)
        7    0.000    0.000    0.000    0.000 function_base.py:726(_copy_dispatcher)
      544    0.004    0.000    0.010    0.000 QctRL.py:53(_init_trace)
     1589    0.004    0.000    0.004    0.000 _dtype.py:36(_kind_name)
     1589    0.020    0.000    0.059    0.000 _dtype.py:333(_name_get)
     1589    0.010    0.000    0.028    0.000 _dtype.py:319(_name_includes_bit_suffix)
     1589    0.001    0.000    0.001    0.000 fromnumeric.py:3033(_ndim_dispatcher)
     1589    0.001    0.000    0.001    0.000 fromnumeric.py:1800(_nonzero_dispatcher)
      797    0.003    0.000    0.786    0.001 _methods.py:36(_sum)
      799    0.000    0.000    0.000    0.000 fromnumeric.py:2087(_sum_dispatcher)
     1589    0.001    0.000    0.001    0.000 fromnumeric.py:600(_transpose_dispatcher)
     3178    0.011    0.000    0.065    0.000 fromnumeric.py:55(_wrapfunc)
     1589    0.017    0.000    0.046    0.000 fromnumeric.py:42(_wrapit)
     3187    0.028    0.000    0.102    0.000 fromnumeric.py:73(_wrapreduction)
      806    0.008    0.000    0.009    0.000 environment.py:54(action_state_map)
     2388    0.015    0.000    0.103    0.000 fromnumeric.py:2551(amax)
     2388    0.009    0.000    0.120    0.000 <__array_function__ internals>:2(amax)
     1589    0.014    0.000    0.107    0.000 numeric.py:537(argwhere)
     1589    0.005    0.000    0.116    0.000 <__array_function__ internals>:2(argwhere)
     2388    0.003    0.000    0.026    0.000 _asarray.py:16(asarray)
        7    0.000    0.000    0.002    0.000 Qmodel.py:102(compute_fidelity)
      797    0.031    0.000    0.061    0.000 Qmodel.py:120(compute_fidelity_ext)
        7    0.000    0.000    0.000    0.000 function_base.py:730(copy)
        7    0.000    0.000    0.000    0.000 <__array_function__ internals>:2(copy)
      798    0.001    0.000    0.001    0.000 multiarray.py:707(dot)
      798    0.009    0.000    5.444    0.007 <__array_function__ internals>:2(dot)

      799    6.604    0.008   12.910    0.016 Qmodel.py:87(evolve)

     1598    0.011    0.000    0.011    0.000 _ufunc_config.py:139(geterr)
      799    0.001    0.000    0.002    0.000 core.py:6293(isMaskedArray)
      799    0.002    0.000    0.003    0.000 base.py:1192(isspmatrix)
     4776    0.009    0.000    0.016    0.000 numerictypes.py:293(issubclass_)
     2388    0.010    0.000    0.026    0.000 numerictypes.py:365(issubdtype)
```

```
     799    0.056    0.000    0.169    0.000 _logsumexp.py:9(logsumexp)
     797    0.017    0.000    0.028    0.000 environment.py:68(move)
    1589    0.002    0.000    0.002    0.000 fromnumeric.py:3037(ndim)
    1589    0.003    0.000    0.009    0.000 <__array_function__ internals>:2(ndim)
    1589    0.003    0.000    0.014    0.000 fromnumeric.py:1804(nonzero)
    1589    0.003    0.000    0.020    0.000 <__array_function__ internals>:2(nonzero)
       9    0.000    0.000    0.000    0.000 Qmodel.py:65(reset)
       9    0.000    0.000    0.000    0.000 environment.py:36(reset)
    1589    0.085    0.000    0.783    0.000 QctRL.py:72(select_action)
    1598    0.012    0.000    0.030    0.000 _ufunc_config.py:39(seterr)
     799    0.011    0.000    0.180    0.000 _logsumexp.py:132(softmax)
     799    0.005    0.000    0.019    0.000 fromnumeric.py:2092(sum)
     799    0.002    0.000    0.024    0.000 <__array_function__ internals>:2(sum)

       9    0.711    0.079   14.511    1.612 QctRL.py:151(train_episode)

    1589    0.003    0.000    0.057    0.000 fromnumeric.py:604(transpose)
    1589    0.004    0.000    0.064    0.000 <__array_function__ internals>:2(transpose)
     797    0.077    0.000    0.349    0.000 QctRL.py:111(update)
     804    0.001    0.000    0.001    0.000 multiarray.py:795(vdot)
     804    0.004    0.000    0.031    0.000 <__array_function__ internals>:2(vdot)
```