Deep Reinforcement Learning for Quantum Control

January 28, 2019

Contents

1 Introduction

There are three main subfields in modern machine learning: supervised learning, unsupervised learning, and reinforcement learning. Supervised learning can be summarized as training a neural network to learn a function that maps input data to some output. The function is inferred by comparing the current output of the network to the desired output and updating the network parameters. Unsupervised learning is used to find commonalities in the input data, and is used to group, classify or categorize unlabeled data.

Reinforcement learning differs from supervised and unsupervised learning and is based on letting an *actor* learn how to behave in a desired way by taking actions in an *environment* and observing the effect of the action on the environment. In order to define the "optimal" behavior of the agent, we give it feedback in the form of a *reward* based on the effect of its previous action. If the action change the environment into a more desirable state we give it a positive reward, while if it had negative consequences we give it a negative reward. A schematic of the basic reinforcement learning protocol is shown in Fig. 1. At time t the environment is in a given state S_t. The agent performs an action A_t which induces a state change of the environment from S_t to S_{t+1}. The agent then receives an observation, O_{t+1}, of the new state of the environment. This observation may be an observation of the full state, i.e. $O_{t+1} \equiv S_{t+1}$, or it can be a partial observation such that it is a subset of the full state, i.e. $O_{t+1} \subset S_{t+1}$.

As an example task thats suitable for reinforcement learning, consider the archetypal the cart-pole balancing problem. An inverted pendulum is attached to a cart as shown in Fig. 2. The task is to balance the inverted pendulum in its upright position by moving the cart right or left. In this task the state of the environment would be described by the position of the cart x, its velocity v_x, the angle of the pendulum with respect to the central axis θ, and the angular momentum of the pendulum ω. The state space would then be $S \in \{x, v_x, \theta, \omega\}$. Usually the more of a state the agent is allowed to observe, the easier it is to learn the desired behavior, so lets say the agent's observation is complete $O \equiv S$. The reward could be $r = +1$ for every timestep the pole has not fallen below a certain angle, and $r = -10$ if the pole falls down. The reward space is then $R \in \{+1 \vee -10\}$. In the simplest case the actions available to the agent could be to apply a certain amount of force, F_x, in the x-direction to either the left or right side of the cart. The action space is therefore $A \in \{-F_x \vee +F_x\}$. In order to understand how we could utilize this information to teach the agent to balance the inverted pendulum we first need to understand the mathematical foundation of modern reinforcement learning; Markov Decision Processes.

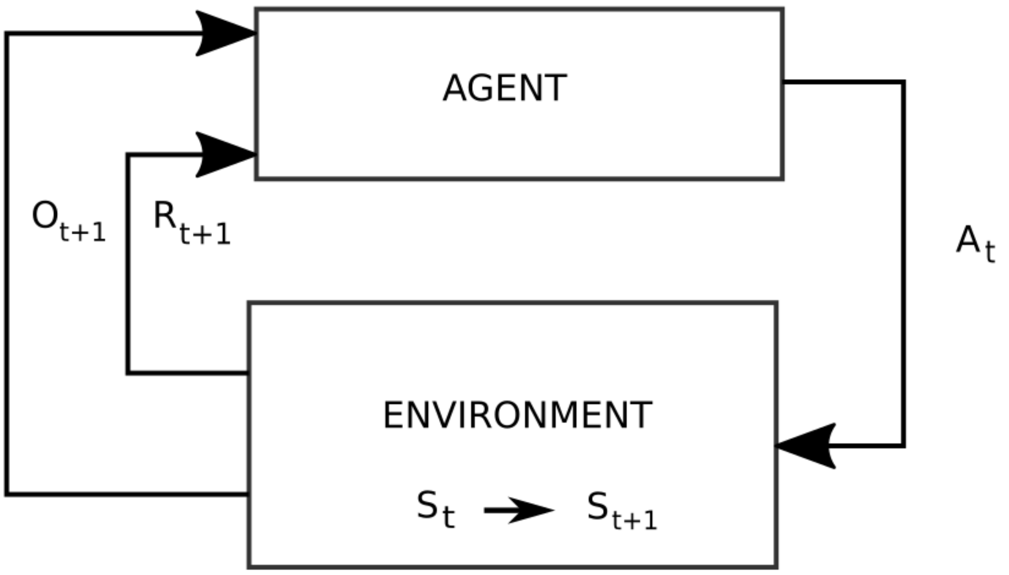

Figure 1: Schematic showing the basic formulation of the reinforcement learning. An agent performs an action A_t which induces a state change of the environment from S_t to S_{t+1}. The agent then receives an observation of the new state of the environment, O_{t+1}, and a reward, R_{t+1} that tells it how good the previous action was.

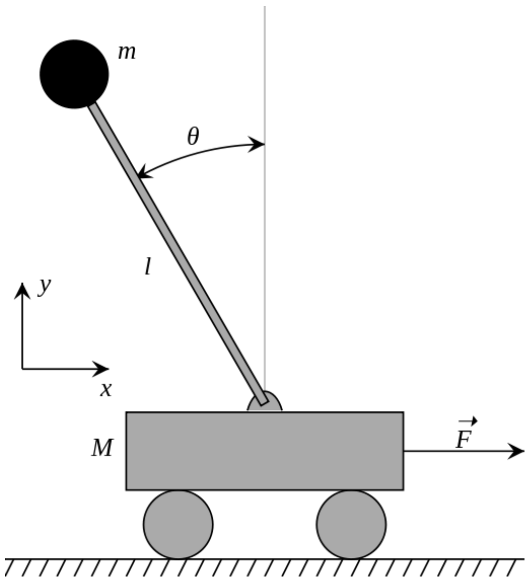

Figure 2: A schematic illustration of the inverted pendulum cartpole balancing problem.

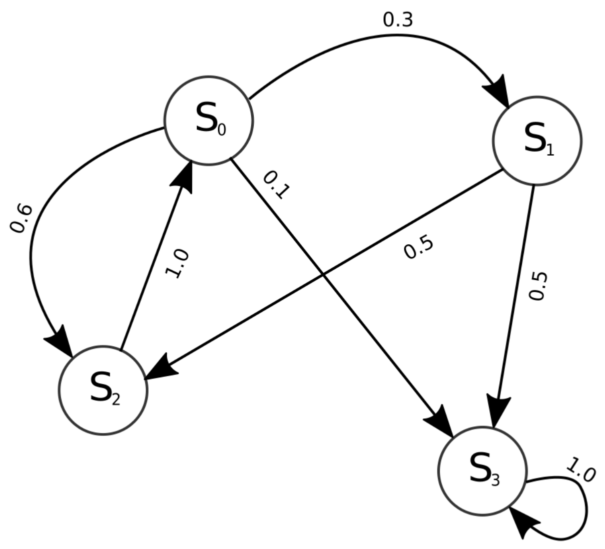

Figure 3: An example of a simple Markov chain model. This one have 4 different states, and the transition probability between each state is indicated by the numbers next to the arrows.

2 Markov Decision Process

A Markov Chain is a stochastic model for the transitions between different states, where the transition probability only depends on the current state. The independence of the future state on the full history of the states visited is called the Markov property. Mathematically it can be defined as

$$P(s_{t+1}|s_t) \equiv P(s_{t+1}|s_t, s_{t-1}, \ldots, s_0), \tag{1}$$

i.e., the conditional probability for transitioning to the state s_{t+1} from the state s_t does not depend on the preceding states $s_{t-1}, s_{t-2}, \ldots, s_0$. In Fig. 3 a simple Markov chain model is shown. It consist of four different states, S_0, S_1, S_2, and S_3. The numbers next to the arrows indicate the transition probability from each state, and the sum of the transition probabilities from each state is equal to 1. In this example the state S_3 is a terminal state, since the only transition available is to itself. Markov decision processes was introduced by Richard Bellman in 1957 [1]. A MDP is similar to a Markov chain, but now transitions between states are mediated by the choice of a set of actions available at each state. In addition some state transitions result in a reward, which can be either negative or positive. An example of a Markov decision process is shown in Fig. 4. Here the actions are chosen by the agent,

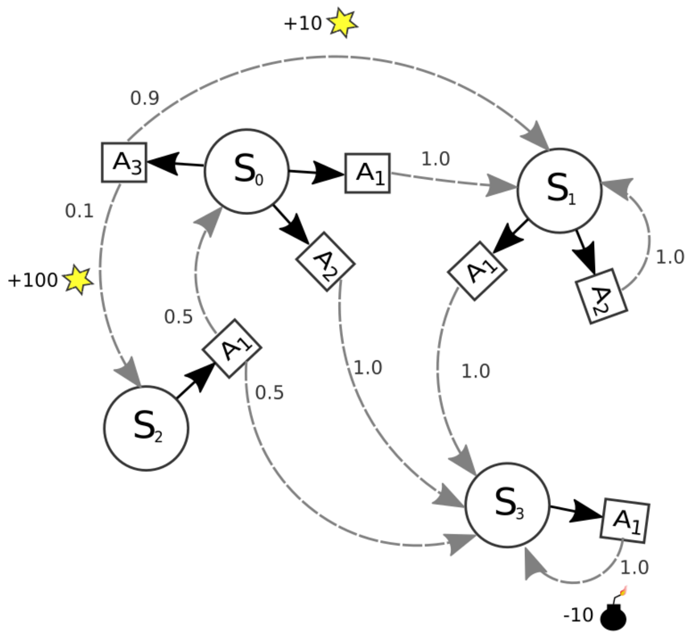

Figure 4: An example of a Markov decision process. As in the Markov chain model there are four possible states, but now each state has between one and three possible actions available. Some of the actions also result in negative or positive rewards.

resulting in a stochastic transition to other states. In the state S_0 there are three available actions to choose from, A_1, A_2 and A_3. After choosing the action A_3 there is a 10% chance to move to state S_2, receiving a reward of $R = +100$, and a 90% chance to move to state S_3, receiving $R = +10$. Assuming we ended up in state S_2 there is only one available action, A_1, which have 50%/50% probability to take us either back to S_0 or to the terminal state S_3. Once we reach state S_3 there is again only one available action, which takes us back to S_3 with a reward (punishment) of $R = -10$. In this example the transitions after choosing an action are stochastic, but a MDP can also be deterministic, i.e. a given action in a given state always result in the same transition.

The goal of the agent is to find a policy (what actions to take at each state), which maximizes the total reward received. In the example given one might be able to find the optimal policy by inspection, but for larger more complicated MDPs this approach quickly becomes impossible. Richard Bellman found a way to estimate the optimal policy of a MPD, but to understand it we first need to define the value and quality functions.

3 Value function and Quality function

Consider the 2D gridworld example shown in Fig. 5. Here the state of the system is given by the coordinates of the grid, e.g. $s = (1, 1)$ is the state in the upper left corner. There are two special states in this example; if $s = (1, 4)$ we get a reward of $R = +10$, while for $s = (3, 4)$ we get a reward of $R = -10$. Both of these states are also terminal states, so if we reach that state the game is over. The state $s = (2, 2)$ is also in a sense special, because it is unattainable. For each state there are four possible actions available to the actor; it can move in either of the cardinal directions. If the agent move into any wall, it ends up in the same state it started in. A *policy* (denoted $\pi(a|s)$) in RL is an instruction to how the agent should act for any given state. On the left side of Fig. 5 the possible actions available for each state is indicated by the arrows. This is a random policy, and if the goal is to get to the state that gives you a reward of $R = +10$, it is not a very good policy. In this example the optimal policy is easy to find by inspection, and it is shown on the left side of Fig. 5.

The value function $V_\pi(s)$ for a given policy π is just the expected cumulative reward gained by following the policy from the state s onwards. The value function for the optimal policy of the gridworld example is shown on the left side of Fig. 6. Since the maximum reward we can get is $R = +10$, and the optimal policy we found always takes us to this state, the value function is $V_\pi(s) = 10$ for all states except the state $s = (3, 4)$ which have the value $V_\pi(3, 4) = -10$. An important concept we have to introduce now is the discount factor γ. Because of inflation and the possibility to gain interest on bank deposits, receiving 100 \$ now is better than receiving 100 \$ later, and the discount factor is meant to account for situations where this concept is applicable. For the cart-pole example from the previous section, the actions the agent have recently performed are more important to stabilize the inverted pendulum than actions it performed 100 timesteps earlier. Applying a discount factor of $\gamma = 0.9$ to the value function of the gridworld example gives us the value function as shown on the right side of Fig. 6.

The Q-function (quality function) is closely related to the value function, the only difference is that it gives the expected cumulative reward given a state-action pair. $Q_\pi(s, a)$ gives you the expected cumulative reward given that you are in state s, take the action a and follow the policy thereafter. The Q-function for the gridworld example with a discount factor of $\gamma = 0.9$ is shown in Fig. 7. The grids are now divided in four, one triangle for each possible action, and the brightness of the color illustrates the Q-function

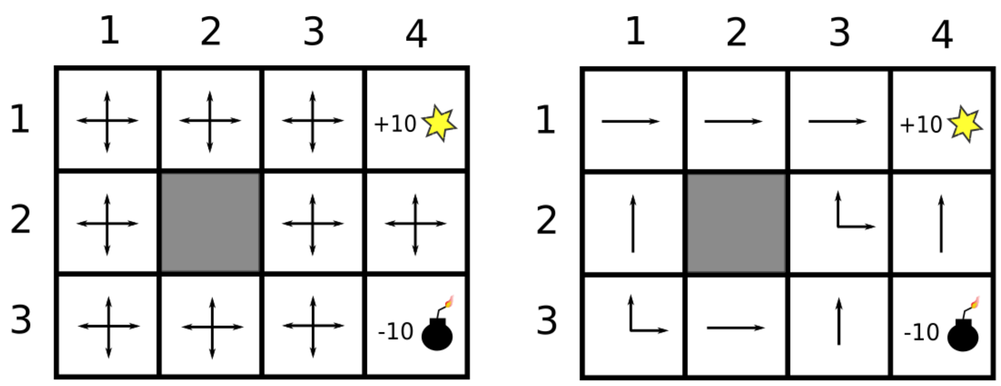

Figure 5: A 2D gridworld with $4 \times 3 - 1$ possible states and two terminal states, $(3, 4)$ and $(1, 4)$ which give rewards $+10$ and -10 respectively. The arrows indicate the actions available at each state. On the left side we have a random policy, and on the right side we show the optimal policy.

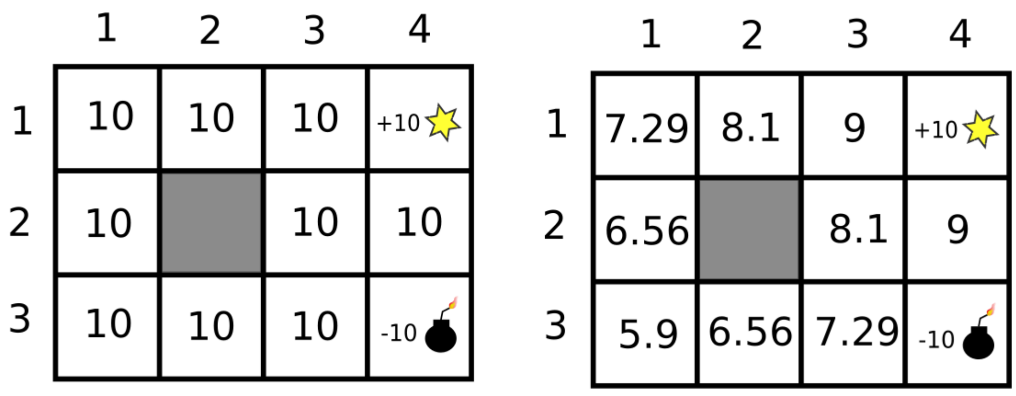

Figure 6: In this figure we show the same gridworld as above, only now we have filled in the value of each state, given that we follow the optimal policy. The right one is shown without a discount factor, and on the left we have applied a discount factor of $\gamma = 0.9$.

value for each state-action pair. For a given state, actions that takes us away from the goal is worse than actions that takes us towards it, and the total quality of a state increases progressively as we get closer to the goal. The actions that take us to the negative terminal state are obviously the worst possible, so they are indicated by a red color. The advantage of describing the system by the Q-function instead of the value function is that the former encodes both the value of being in a certain state, and the policy to follow. So by finding the optimal Q-value of all the state-action pairs,denoted $Q^*_\pi(s,a)$, we also find the optimal policy $\pi^*(a|s)$. If we always choose the brightest shade of green in each state of Fig. 7 we see that they correspond to the arrows of the optimal policy shown in Fig. 5.

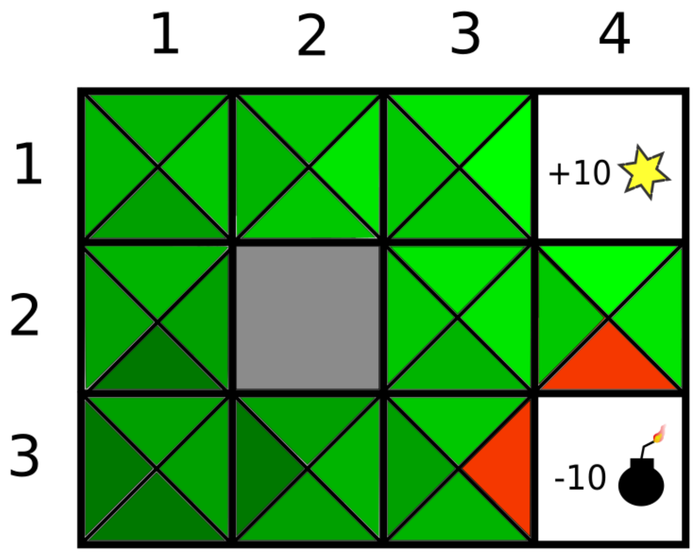

Figure 7: Here we illustrate the quality function of the gridworld example with a discount factor of $\gamma = 0.9$. The brightness of the green color indicates how good the state-action pair is (brighter means higher expected reward).

4 Deep Q-Learning

So how do we find the optimal Q-value? The value function can be defined as

$$V_\pi(s_t) = E\left[R(s_t, a_t, s_{t+1}) + \gamma R(s_{t+1}, a_{t+1}, s_{t+2}) + \cdots + \gamma^n R(s_{\tau-1}, a_{\tau-1}, s_\tau)\right], \tag{2}$$

where $R(s_t, a_t, s_{t+1})$ is the reward received by going from state s_t to s_{t+1} via the action a_t, and s_τ is a terminal state. The discount factor is applied for all the future rewards and $E[\ ]$ indicate the expectation value, in case there is stochasticity in the transitions. This equation can be rewritten as

$$V_\pi(s_t) = E\left[R(s_t, a_t, s_{t+1}) + \gamma V_\pi(s_{t+1})\right], \tag{3}$$

since

$$V_\pi(s_{t+1}) = E\left[R(s_{t+1}, a_{t+1}, s_{t+2}) + \cdots + \gamma^{n-1} R(s_{\tau-1}, a_{\tau-1}, s_\tau)\right]. \tag{4}$$

We define the transition probability $T(s_t, a_t, s_{t+1})$ as the probability to go from state s_t to s_{t+1} given that we choose the action a_t, and write the expectation explicitly as

$$V_\pi(s_t) = \sum_{s_{t+1}} T(s_t, a_t, s_{t+1}) \left[R(s_t, a_t, s_{t+1}) + \gamma V_\pi(s_{t+1})\right]. \tag{5}$$

If we have found the optimal Value function $\pi^*(a_t|s_t)\ \forall\ t$ the following relation is obviously true

$$V_\pi^*(s_t) = \max_{a_t} \sum_{s_{t+1}} T(s_t, a_t, s_{t+1}) \left[R(s_t, a_t, s_{t+1}) + \gamma V_\pi^*(s_{t+1})\right]. \tag{6}$$

That is, the optimal Value function (maximum expected future reward) is obtained by taking the action a_t that maximizes the expected immediate reward obtained plus the expected reward from all possible future states that this action leads to. This equation is known as the Bellman Optimality Equation [3]. It has a very similar form for the Q-value function;

$$Q_\pi^*(s_t, a_t) = \max_{a_t} \sum_{s_{t+1}} T(s_t, a_t, s_{t+1}) \left[R(s_t, a_t, s_{t+1}) + \gamma \max_{a_{t+1}} Q_\pi^*(s_{t+1}, a_{t+1})\right]. \tag{7}$$

This equation can be used to iteratively update the estimated value for the optimal Q-value for every possible state-action pair:

$$Q_\pi^{k+1}(s_t, a_t) \leftarrow \sum_{s_{t+1}} T(s_t, a_t, s_{t+1}) \left[R(s_t, a_t, s_{t+1}) + \gamma \max_{a_{t+1}} Q_\pi^k(s_{t+1}, a_{t+1})\right]. \tag{8}$$

Here $Q_\pi^k(s_t, a_t)$ is the estimate of the Q-value for the kth iteration of the algorithm. These estimates are guaranteed to converge to the optimal Q-value, given enough iterations [3]. Once the optimal Q-value is found, the optimal policy is given by

$$\pi^*(a_t|s_t) = \underset{a_t}{\text{argmax}}\ Q^*(s_t, a_t) \tag{9}$$

This is an example of dynamic programming; we break down the complex problem of directly finding the optimal policy to the subproblems of finding the optimal Q-value for each state-action pair, which we then use to extract the optimal policy.

Finding the optimal policy using Eq. 8 can certainly be effective for small systems. However, the algorithm scales very poorly for larger MPDs with many states and actions. Using it we have to calculate values for all the state-action pairs. This is certainly possible for the gridworld example introduced earlier, where we have approximately $\sim (3 \times 4) \times 4 = 48$ state-action pairs, but for the inverted pendulum the state-space is continuous so we have in principle an infinite number of state-action pairs. The solution was introduced by DeepMind in 2015 [4], and involves approximating the Q-function using a neural network.

$$Q(s, a) \simeq Q(s, a, \theta). \tag{10}$$

Here θ are the parameters of the neural network (its weights and biases). Neural networks are excellent function approximators, and with this innovation DeepMind was able to greatly outperform humans in several Atari games [4]. Training a neural network to approximate the Q-values is called *Deep Q-Learning* (DQL).

A neural network trains by minimizing a loss function (also called a cost function). Lets say you want to predict housing prices in Oslo by looking at features such as the latitude and longitude, number of bedrooms, age of house, and so on. You have a obtained a certain amount of data where you have the prices of the houses as well as those features. A common loss function is the mean squared error (MSE)

$$\text{MSE} = \frac{1}{n} \sum_{i=1}^{n} \left(y_i - \hat{y}_i \right)^2 , \tag{11}$$

where n is the number of samples, y_i is the true housing price (the target value) and $\hat{y}_i$ is the price predicted by the neural network. The MSE of the output from a neural network can be minimized by calculating its gradient and adjusting the weights and biases of the network accordingly. When the gradient of the MSE is zero we have reached a minimum (hopefully a global minimum) of the loss function. The details on the special back-propagation method used for training neural networks can be found in several articles, blog posts, and books, including [5].

When training the network in DQL, we do not actually know the real optimal Q-value. However we know that according to the Bellman equation the optimal Q-value satisfies

$$Q^*(s_t, a_t, \theta) = E\left[r_t + \gamma \max_{a_{t+1}} Q^*_\pi(s_{t+1}, a_{t+1}, \theta)\right], \tag{12}$$

where we have introduced the short-hand notation $r_t = R(a_t, s_t, s_{t+1})$. We can use this estimate as our target when training the network. The MSE loss function for DQL then becomes

$$L(\theta) = E\left[\left(r_t + \gamma \max_{a_{t+1}} Q^*_\pi(s_{t+1}, a_{t+1}, \theta) - Q(s_t, a_t, \theta)\right)^2\right] \tag{13}$$

For DQL to work properly and converge to a good estimate of the optimal Q-values, there are several improvements we can make. Some of these were introduced in [4], and will be covered in the following.

4.1 Exploration vs Exploitation

When the network is initialized its predictions for the Q-values are of course totally wrong. So if we always chose the actions that maximizes the current predicted Q-values, $\underset{a_t}{\text{argmax}}\ Q^*(s_t, a_t)$, the agent would not learn anything. We need to let the agent explore the state-action space by randomly performing actions. A typical exploration policy is the ϵ-greedy policy. In the beginning of the training the agent chooses random actions with probability ϵ, or the ones with the highest Q-value (greedily) with probability $1 - \epsilon$. As time goes and the agent explores more and more of the environment, ϵ is decreased so that it focuses more on the areas of the state-action space with higher Q-values. Typically we start by taking completely random actions, $\epsilon = 1$, and let ϵ converge to some finite number $\epsilon \sim 0.05$, so that there is always some exploration going on.

4.2 Experience Replay

As seen in Eq. 13 a single update of the network weights requires the following input: the current state s_t, the action chosen a_t, the immediate reward r_t, and the next state s_{t+1}. We call this tuple, $e_t = (s_t, a_t, r_t, s_{t+1})$, that the network trains on an experience. Instead of training on consecutive experiences we store them all in a memory $M_t = \{e_0, e_1, \ldots, e_t\}$, and then train on randomly drawn batches of samples from the memory. The memory have a finite capacity, and new experiences replace older ones when the memory is full. There are three main advantages of training on the replay memory:

It is data efficient since a single experience can be drawn many times. Only training on consecutive experiences is inefficient, since the network tends to forget previous experiences by overwriting them with new experiences. The time-correlation of consecutive experiences means that the network update due to the current experience determines what the next experience will be, so training can be dominated by experiences from a certain area in the state-action space.

4.3 Target Network

Finally we see that in Eq. 13 the current weights of the network determines both the target Q-value and the predicted Q-value. Thus every network update changes the target Q-value that we are trying to reach. This is like a dog chasing its own tail, and makes it hard for the network weights to converge. A simple way to circumvent this problem is to use two neural networks, one for the target Q-value (θ^-), and one for the current Q-value (θ).

$$L(\theta) = E\left[\left(r_t + \gamma \max_{a_{t+1}} Q^*_\pi(s_{t+1}, a_{t+1}, \theta^-) - Q(s_t, a_t, \theta)\right)^2\right]. \tag{14}$$

The target network weights θ^- are updated to the current network weights, $\theta^- \rightarrow \theta$, every N iteration of the algorithm. A pseudocode of the DQL algorithm presented in [4] is shown below

```
Initialize memory M
Initialize Q-value network with weights $\theta$
Initialize target Q-value network with weights $\theta^- = \theta$
for episode = 1 to $N_e$ do
    Reset environment to initial state $s_0$
    for $t$ = 1 to $T$ do
        Select random action $a_t$ with probability $\epsilon$
        Else select $a_t = \underset{a_t}{\mathrm{argmax}}\, Q^*(s_t, a_t)$
        Execute action in environment and observe $r_t$ and $s_{t+1}$
        Store experience $(s_t, a_t, r_t, s_{t+1})$ in memory M
        Sample random experiences $(s_i, a_i, r_i, s_{i+1})$ from M
        If $s_{i+1}$ is a terminal state set $y_i = r_i$
        Else set $y_i = r_i + \gamma \max_{a_{i+1}} Q(s_{i+1}, a_{i+1}, \theta^-)$
        Perform a gradient decent step on $(y_i - Q(s_i, a_i, \theta))^2$
        Every C steps set $\theta^- = \theta$
        Set $s_{t+1} = s_t$ and decrease $\epsilon$
    end
end
```

5 Qubit spin control

5.1 State-to-state transfer

Consider a two level system described by the time-dependent Hamiltonian

$$H[h_x(t)] = -\sigma_z - h_x(t)\sigma_x, \tag{15}$$

where σ_z and σ_x are Pauli spin matrices. The goal of our reinforcement learning agent is to drive the system from some initial state $|\psi_i\rangle$ to a final state $|\psi_f\rangle$, within a given time τ. Its available actions are to choose the value of the applied magnetic field in the x-direction, $h_x(t)$, for each time interval dt. A suitable reward function for this task is then to give a positive reward based on the overlap between the target state $|\psi_f\rangle$ and the state at the end of an episode $|\psi(\tau)\rangle$, $F[\psi(t)] = |\langle\psi_f|\psi(t)\rangle|^2$. The specific reward shaping used for the qubit spin control is

$$r(t) = \begin{cases} -1, & \text{if} \quad F[\psi(t)] < 1 - \delta \\ 100 \cdot F[\psi(t)] & \text{if} \quad F[\psi(t)] > 1 - \delta. \\ 100 \cdot F[\psi(t)] & \text{if} \quad t = \tau. \end{cases} \tag{16}$$

Here the reward is -1 for each time-step passed without the state being within the desired distance (defined by the tolerance δ). The episode terminates if we reach the maximum alloted time τ, or if the overlap becomes unity, within the desired tolerance. The reward shaping determines what the agent learns, and one of the main challenges for RL is how to construct the feedback to make the agent achieve the task we want it to, and minimize the time it takes to learn it. There are many loopholes the agent can find that does not fit the task we actually want it to perform. Here we have multiplied the reward gained by reaching the target state by 100. If instead multiplied it by 1000, the agent could for example take a 10 steps extra (being rewarded -1 each time) just so it could go from 0.98 overlap to 0.99. If we multiplied it by a lower number the opposite would happen; the agent could be content with 0.95 overlap instead of using one extra time-step to get 0.99 overlap. Finally, to help the agent learn faster we give it a reward based on the overlap at $t = \tau$, even if its not within the desired distance. Without this reward it could take a long time (depending on δ) before the agent making random actions gets any positive reward to guide its behavior. The necessity of this additional reward depends on what value we choose for δ. If for example $\delta = 0.05$ we can be confident that the agent will sometimes randomly reach the desired state by random actions, and learn from those experiences. On the other hand if $\delta \simeq 0$, that would never happen.

The state of the system at time t in the σ_z basis is given by

$$|\psi(t)\rangle = c_0(t)\,|0\rangle + c_1(t)\,|1\rangle\,, \tag{17}$$

where $|0\rangle$ is the spin-down eigenstate and $|1\rangle$ is the spin up eigenstate. A full description of the environment is then given by $S = (H[h_x(t)], c_0(t), c_1(t), t)$. As previously stated, the observations of the agent does not need to be complete. To begin with we choose the observation of the agent to be incomplete, in the sense that the input to the neural network is

$$O_t = (\text{Re } c_0, \text{Re } c_1, \text{Im } c_0, \text{Im } c_1, t)\,. \tag{18}$$

The coefficients c_0 and c_1 are complex numbers, which is difficult to deal with for neural networks, so we have to split them up into their real and imaginary parts. For the simplest case we limit the control field $h_x(t)$ to a bang-bang protocol where the agent chooses between two values, $h_x(t) = -4 \vee +4$, for each time interval $t \to t + dt$. We set the initial state $|\psi_i\rangle$ to be the ground-state of $H = -\sigma_z - 2\ \sigma_x$, and the target state the ground-state of $H = -\sigma_z + 2\ \sigma_x$. Therefore the goal is to drive the system from the orange arrow to the green arrow shown in Fig. 8. We set the maximum alloted time to $\tau = 3.75$ s, the size of the time-step to dt $= 0.05$, and $\delta = 0.01$. This gives us a maximum of 75 actions to perform before the episode terminates.

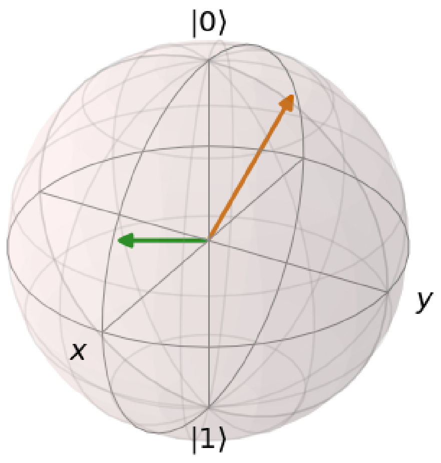

Figure 8: Bloch state representation of the initial state (orange) and the target state (green).

In Fig. 9 we plot the total reward received per episode as a function of the episode number. The blue dots are the reward received per episode while the red line is a running average of the 250 neighboring episodes. In the first 4000 episodes there is a lot of exploration, giving us randomly distributed rewards. There is a clear distinction above and below $R \simeq 20$ between those episodes that randomly reached the target state before $t = \tau$ and those that did not. As we tune the exploration parameter down per episode according to the $\epsilon -$ greedy strategy the reward received increases until it converges to values around $R \simeq 70$. The stochasticity observed even in the final episodes is due to the fact that we never turn the exploration completely off, the probability of taking a random action is bounded below at 5%.

In Fig. 10 we show the number of time-steps taken before the episode terminates. In the early phase, with high exploration probability, the state transfer is rarely achieved within our desired time. The agent almost always performs all its maximum allowed 75 steps. As exploration is decreased and the actions that the agent considers best are more deterministically chosen, the number of steps per episode decreases. It converges to about 25 steps. The very minimum amount of steps is 23 steps, which corresponds to $t = 1.15$ s. The quantum speed limit, which is the minimum amount of time it takes to perform a state transfer for a given Hamiltonian, is about $\tau_{QSL} \simeq 1.2$ for this system [2]. This is already quite close (when considering the time-step is $dt = 0.05$), but we can get an even better approximation of τ_{QSL} by reducing

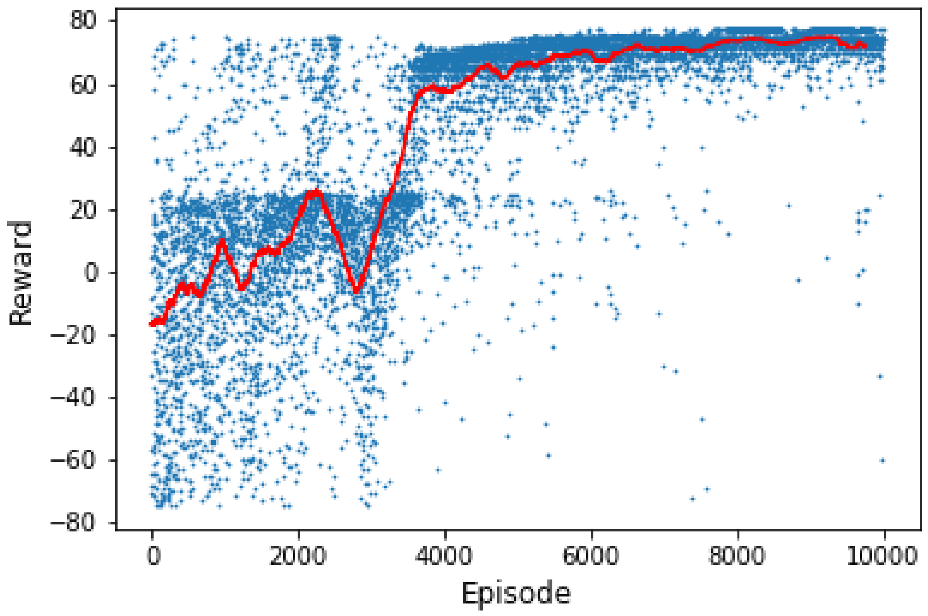

Figure 9: Plot of the cumulative reward received for each episode. The blue dots are the rewards, and the red line is a running average (250 neighbors).

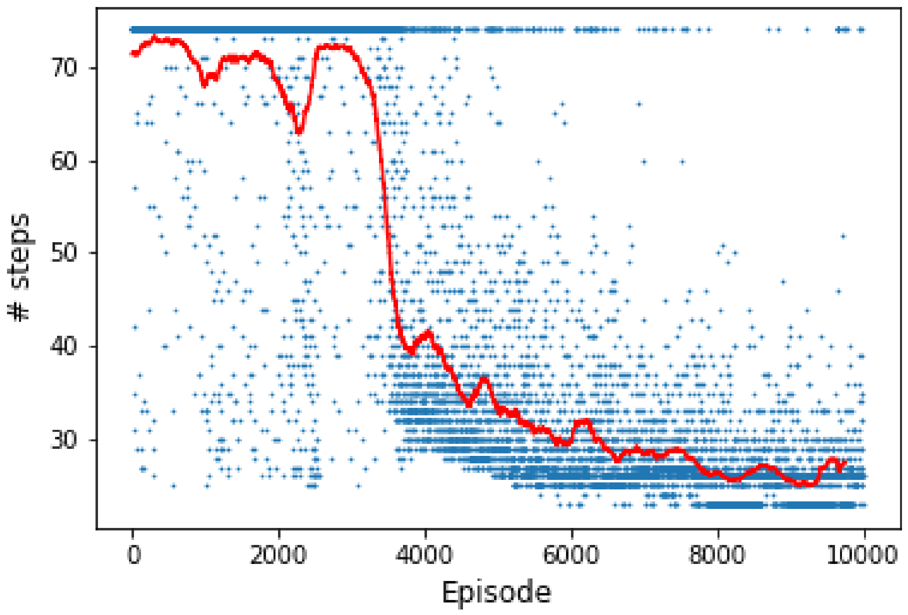

Figure 10: Plot of the number of time-steps taken before termination. The blue dots is the number of steps per episode, and the red line is a running average (250 neighbors).

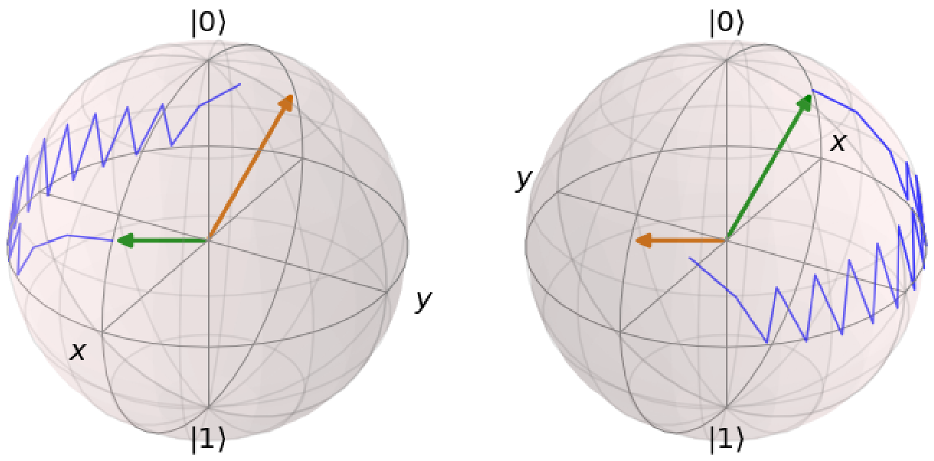

Figure 11: Two views (180 degrees apart) of the fastest state-to-state transfer found. The green arrow points to the initial state, while the orange arrow points to the target state.

δ.

The fastest and best solution found by the agent is shown in Fig. 11. The green arrow points to the initial state, while the orange arrow points to the target state. The two Bloch-spheres show the same solution, they have a relative rotation of 180 degrees for better viewing. The blue lines is a connected line between the states $|\psi(t_n)\rangle$ for $t_n = 0, dt, 2dt, \dots,$ and does not show the exact time evolution of the state. A striking feature is that even though this is a model free algorithm, meaning that the agent does not know anything about the system it is influencing, it nevertheless learns that the fastest way is to travel along the equator.

5.2 Starting from random initial state

We can easily extend the previous analysis, using the exact same agent. Instead of finding the fastest way from one particular initial state, we can find the fastest to a target state from any random initial state. The only goal of the agent is to maximize the *expected* cumulative future reward for any given state-action pair, so this type of stochasticity is no hindrance for it. Of course it takes much longer to learn the optimal sequence of actions for *all* states rather than just one. For this task we allowed the agent to take a total of 100 steps, to further increase the probability of gaining positive rewards in the high exploration phase. Except from that we use the exact same algorithm and parameters as in the previous section, the only difference being that every time an episode terminate, the next initial state is a random point on the Bloch-sphere. Fig. 12 shows the total reward received per episode, as before. Due to the added randomness of the initial condition we no longer see a clear divide between the episodes where a state transfer was achieved and those where it was not. However as before we see clear tendencies of learning as the exploration is tuned down with the number of episodes passed. The reward converges to a band around $R \simeq 80 \pm 20$. In Fig. 13 we show the number of steps taken per episode. In the beginning there are a lot of episodes where the agent performs all of the 100 allowed steps. Nevertheless, the agent eventually learns to minimize the amount of steps for all states on the Bloch sphere. It reaches an average of ~ 20 steps per episode. This is of course averaged over all initial conditions also, so some of the times the initial condition is already very close to the target state, so that it only performs one or two actions. After about 3000 episodes, the agent always reaches the target state before taking all of its allowed steps. Even better, for the later episodes the maximum amount of steps it needs is about ~ 30 steps, which translates to about $t = 1.5$ s. And this is for *any* initial state on the Bloch-sphere.

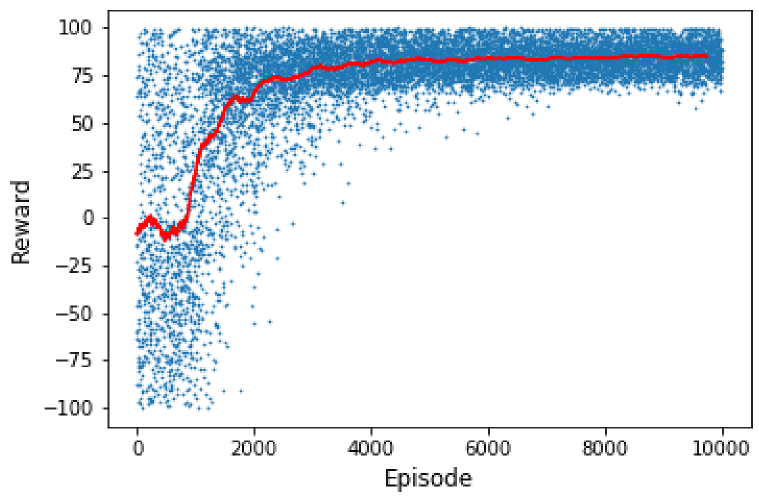

Figure 12: The total reward received per episode. The initial state is a random uniformly sampled point on the Bloch sphere.

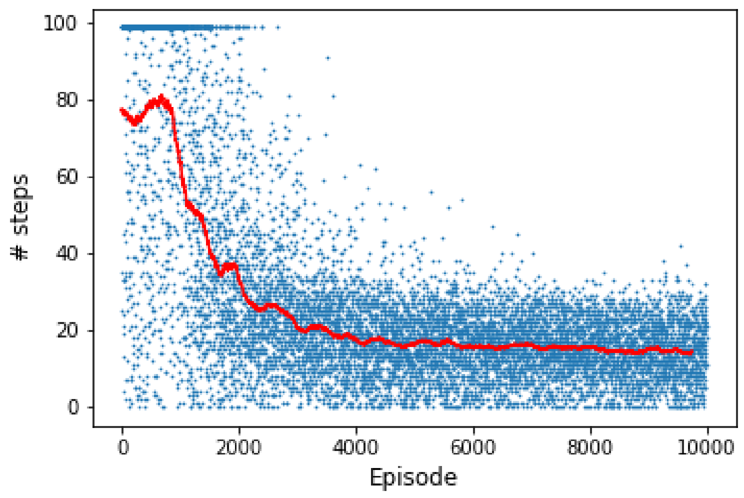

Figure 13: The number of steps the agent took before the episode terminated. The initial state is a random uniformly sampled point on the Bloch sphere.

6 Source Code

In this project we used QuTiP 4.3.1 to perform the time evolution of the quantum states, and Keras 2.2.4 for the neural network.

6.1 Environment Class

The following is the code for making a Class environment that the agent can act in.

```
class bloch_env:

H1 = -sigmaz() - 2*sigmax()
H2 = -sigmaz() + 2*sigmax()

def __init__(self):


self.initial_state = self.H1.groundstate()[1]
self.target_state = self.H2.groundstate()[1]

self.dt = 5e-2
self.max_steps = 100
self.t_max = self.dt*self.max_steps
self.H = [- sigmaz() - 4*sigmax() , - sigmaz() + 4*sigmax()]
self.actions = [4,-4]



self.action_interval = np.array([self.actions[0],self.actions[-1]])
self.reset()
self.action_space = len(self.actions)
self.observation_space = len(self.state)



def reset(self):

self.cum_reward = 0
self.time = 0
self.alfa = 0

# self.q_state = rand_ket(2) # uncomment for random initial state
self.q_state = self.initial_state
self.state_real = np.real(np.reshape(self.q_state[:], [1,2])[0])
self.state_imag = np.imag(np.reshape(self.q_state[:], [1,2])[0])

self.state = np.append(self.state_real, self.state_imag)

self.steps = 0


return self.state

def time_evolution(self,action):

self.H = - sigmaz() - action*sigmax()
self.U = (-1j*self.H*self.dt).expm()

return self.U

def step(self,action):

self.steps += 1
self.time += self.dt
self.alfa = self.actions[action]
self.q_state = self.time_evolution(self.actions[action]) *
    self.q_state

self.next_state_real = np.real(np.reshape(self.q_state[:],
    [1,2])[0])
self.next_state_imag = np.imag(np.reshape(self.q_state[:],
    [1,2])[0])

next_state = np.append(self.next_state_real, self.next_state_imag)

reward = -1

self.overlap = abs(self.q_state.overlap(self.target_state))**2

if self.overlap >= 0.99:
reward = 100*(self.overlap)
done = True

elif self.steps == self.max_steps:

done = True

reward = reward + 100*(self.overlap)

else:
done = False

self.cum_reward += reward

return next_state, reward, done
```

6.2 Agent Class

The following is the code to create a class in Python that acts as the agent. Modified from github.com/keon/deep-q-learning.

```
class DQNAgent:

def __init__(self, state_size, action_size):

self.state_size = state_size
self.action_size = action_size

self.memory = deque(maxlen = 2000)

self.gamma = 1

self.epsilon = 1.0
self.epsilon_decay = 0.99995
self.epsilon_min = 0.01

self.learning_rate = 0.001

self.model = self._build_model()
self.target_model = self._build_model()

def _build_model(self):

model = Sequential()
model.add(Dense(24, input_dim = self.state_size, activation =
    'relu'))
model.add(Dense(48, activation = 'relu'))
model.add(Dense(24, activation = 'relu'))
model.add(Dense(self.action_size, activation = 'linear'))

model.compile(loss='mse', optimizer = Adam(lr =
    self.learning_rate))

return model


def remember(self, state, action, reward, next_state, done):

self.memory.append((state, action, reward, next_state, done))

def act(self,state):

if np.random.rand() <= self.epsilon:
return random.randrange(self.action_size)
act_values = self.model.predict(state)
return np.argmax(act_values[0])

def replay(self, batch_size):
```

```
minibatch = random.sample(self.memory, batch_size)


for state, action, reward, next_state, done in minibatch:

target = self.target_model.predict(state)

if done:
target[0][action] = reward

else:
Q_future = max(self.target_model.predict(next_state)[0])
target[0][action] = reward + Q_future*self.gamma

self.model.fit(state, target, epochs = 1, verbose = 0)

def load(self, name):
self.model.load_weights(name)

def save(self,name):
self.model.save_weights(name)
```

6.3 Main File

The following is the main file where we initialize the agent and the environment, and make them interact.

```
env = bloch_env()
state_size = env.observation_space
action_size = env.action_space

batch_size = 32
n_episodes = 10000

done = False
agent = DQNAgent(state_size, action_size)

agent.epsilon_decay = (0.05)**(1/n_episodes)

score = [None]*n_episodes
for e in range(n_episodes):
```

```
state = env.reset()
state = np.reshape(state, [1,state_size])

for t in range(env.max_steps):

action = agent.act(state)

next_state, reward, done = env.step(action)

next_state = np.reshape(next_state, [1,state_size])

agent.remember(state, action, reward, next_state, done)

state = next_state

if done:
print('episode: {}/{}, score: {:.3}, epsilon:
    {:.2}'.format(e,n_episodes, env.cum_reward, agent.epsilon))
score[e] = [env.cum_reward,t]
break

if len(agent.memory) > batch_size:
agent.replay(batch_size)

if e % 10 == 0:
agent.target_model.set_weights(agent.model.get_weights())

if agent.epsilon > agent.epsilon_min:
agent.epsilon *= agent.epsilon_decay
```

References

[1] RICHARD BELLMAN. A markovian decision process. *Journal of Mathematics and Mechanics*, 6(5):679–684, 1957.

[2] Marin Bukov, Alexandre G. R. Day, Dries Sels, Phillip Weinberg, Anatoli Polkovnikov, and Pankaj Mehta. Reinforcement learning in different phases of quantum control. *Phys. Rev. X*, 8:031086, Sep 2018.

[3] Aurelién Géron. *Hands-on Machine Learning with Scikit-Learn and Tensorflow*. O'Reilly Media, Inc.

[4] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A. Rusu, Joel Veness, Marc G. Bellemare, Alex Graves, Martin Riedmiller, Andreas K. Fidjeland, Georg Ostrovski, Stig Petersen, Charles Beattie, Amir Sadik, Ioannis Antonoglou, Helen King, Dharshan Kumaran, Daan Wierstra, Shane Legg, and Demis Hassabis. Human-level control through deep reinforcement learning. *Nature*, 518(7540):529–533, February 2015.

[5] Jerome Friedman Trevor Hastie, Robert Tibshirani. *The Elements of Statistical Learning: Data Mining, Inferene, and Prediction.* Springer Science + Business Media.