Reinforcement Learning: From Principles to Variational Quantum Circuits Implementation

Andrea Baldanza, Supervisor: Duarte Magano and Ariel Guerreiro

2024

Abstract

Starting from the principles of Reinforcement Learning, I propose an analysis of some classical algorithms (SARSA, Q-Learning, and Deep-Q-Learning) in a stochastic Frozen Lake environment. A variational quantum circuit implementation is discussed. The report can be a valuable resource for newcomers interested in exploring reinforcement machine learning and its modern quantum developments.

Contents

1 Introduction

Reinforcement Learning (RL) involves a set of techniques that enables Artificial Intelligence to improve its performance through experience, also without prior knowledge about the environment. RL is a continuously evolving field, that has gained significant importance and attention in recent years from both the scientific community and technology companies. Google's research team is testing reinforcement learning techniques to achieve superhuman performance in various games. A significant breakthrough occurred in 2016 when AlphaGo, combining supervised and reinforcement learning techniques, defeated the world champion in the board game Go[2]. The following year, a more advanced version of AlphaGo, based solely on reinforcement learning and without any prior human knowledge, defeated its predecessor 100-0 [3]. Despite the excellent results, training and improving neural networks for reinforcement learning is expensive. Recent studies proved that hybrid classical-quantum algorithms may give speedups in training, and may help to reduce the consumption of resources [9, 10, 1, 6].

In this work, I first introduce *Markov decision processes*, where an agent interacts with a stochastic environment. I introduce the Bellman equations in a stochastic frozen lake environment(the environment is shown in Figure 1). The Bellman equations form the foundation of RL techniques. When the environment is not fully understood, it's impossible to solve the Bellman equations directly. This challenge led to the development of Reinforcement Learning (RL) techniques. In Chapter 3 I propose an analysis of some RL classical algorithms (SARSA, Q-Learning, and Deep-Q-Learning). In Chapter 4, I discuss the implementation and possible advantages of Variational Quantum Circuits as substitutes for deep neural networks in Deep-Q-Learning.

2 Markov decision processes

2.1 Theory: definitions and structure

A Markov Decision Process (MDP) involves an agent in a stochastic environment. At each step, the agent can act, but the stochasticity of the environment causes the action to not be reliable.

More formally, a MDP[18, 20] is a 4-Tuple $(\mathcal{S}, \mathcal{A}, P, r)$, where $\mathcal{S}$ is called **state space**, $\mathcal{A}$ is the **action space**, P is the

transition probability tensor and r represents the **reward function**. Our agent can be in different specific states $s \in \mathcal{S}$, and can move between states by performing actions $a \in \mathcal{A}$. The stochasticity of the environment is encapsulated in the transition probability tensor $P(s'|s,a)$, which represents the probability of going from the state s to the state s' in one step by performing the action a:

$$P(s'|s,a) = \mathbb{P}[s_{t+1} = s'|s_t = s, a_t = a] \tag{1}$$

Where t is the time index. The Markovianity of the process is given by the fact that the probability of arriving in a state depends only on the previous state and action. At each state, the agent receives a reward r: if the state is a "good" state, the reward will be positive, if it is a "bad" state, the reward will be negative. Starting from a state s and performing an action a, the *expected reward* is:

$$\begin{aligned} R(s,a) &= \mathbb{E}_\pi[r_{t+1}|s_t = s, a_t = a] \\ &= \sum_{s'} r(s')P(s'|s,a) \end{aligned} \tag{2}$$

Sometimes it is possible to use also a transition tensor that records the probability of transitioning from s to s' after taking action and while obtaining a specific reward $P(s', r|s, a) = \mathbb{P}[s_{t+1} = s', r_{t+1} = r|s_t = s, a_t = a]$. This represents a more general transition tensor, and it is linked with the previous through a marginal sum:

$$P(s'|s,a) = \sum_r P(s', r|s, a) \tag{3}$$

If the rewards are deterministic depending only on the states, obviously it is not necessary to explicit the dependence in r, and $P(s'|s,a)$ gives all the necessary information to face the problem.
In general, the goal of the agent is to accumulate the highest rewards in the shortest possible time.
To reach this goal, it is useful to define 2 important quantities:

1. **Policy**. The policy represents what action the agent should take for any state s. A deterministic policy takes as input a state and gives as output an action:

   $$\pi : \mathcal{S} \to \mathcal{A}$$

   $$\pi(s) \equiv a^\pi(s)$$

   A stochastic policy is the probability of taking an action given a state:

   $$\pi : \mathcal{S} \otimes \mathcal{A} \to [0,1]$$

   $$\pi(a|s) \equiv \mathbb{P}[a = a_t|s = s_t] \tag{4}$$

   A deterministic policy can be written as a stochastic policy with $\pi(a^\pi(s)|s) = 1$.

2. **Return function G**
   The return function considers the rewards accumulated over an entire path taken by the agent. Let be $\{s\} = [s_t, s_{t+1}, ..., s_{t+T}]$ a specific possible path of our agent. The return function can be defined as

   $$G_t([s_t, ..., s_{t+T}]) = \sum_{t'=1}^{T} \gamma^{t'-1} r(s_{t+t'}),\ \gamma \in (0,1] \tag{5}$$

   This definition with the parameter γ reflects the concept of "discounted rewards": rewards received at later times have less importance.

Given the return function, we define two important goodness metrics:

1. **Expected utility**. The *expected utility*, also called *state-value function* is the expected value of the Return function starting, from a state s and following a policy π :

   $$V_\pi(s) = \mathbb{E}_\pi[G_t|s_t = s] \tag{6}$$

   This represents how good is the current policy π.

2. **Q-value**. The *Q-value*, also called *Action-value function*, is the expected value of the Return function, starting from a state s, making an immediate action a, and following the policy π for the rest of the time:

   $$Q_\pi(s,a) = \mathbb{E}_\pi[G_t|s_t = s, a_t = a] \tag{7}$$

   This Q-value is important because permits to compare different policies. This can be achieved by defining an *Advantage function* as $A_\pi(s,a) = Q_\pi(s,a) - V_\pi(s)$: if it is positive the action a is better with respect to the action previously predicted by the current policy π.
   The state values V and the Q values are linked by the following relation:

   $$V_\pi(s) = \sum_{a \in \mathcal{A}} Q_\pi(s,a)\pi(a|s) \tag{8}$$

   For deterministic policies, an action $a_\pi(s)$ is chosen with probability 1, so:

   $$V_\pi(s) = Q_\pi(s, a_\pi(s)) \tag{9}$$

The final goal of the agent is to choose the policy π^* that maximizes the expected utility. I call $V(s)$ the maximum utility:

$$V(s) \equiv V_{\pi^*}(s) = \max_\pi V_\pi(s) \tag{10}$$

$\pi^* = \text{argmax} V_\pi(s)(s)$ is called **optimal policy**.
It is possible to explicitly express the expectation values in function of the policy and transition probabilities. Let $\mathcal{O} = \mathcal{O}(\{s\})$ a quantity that depends on the path of the agent $\{s\} = [s_0, s_1, ..., s_T]$. So the expected value of $\mathcal{O}$ is given by:

$$\mathbb{E}_\pi[\mathcal{O}] = \sum_{\{s\}} \mathcal{O}(\{s\})P_\pi(\{s\}) \tag{11}$$

With $P_\pi(\{s\})$ is the probability of taking a path $\{s\}$ while following the policy π. To find $P_\pi(\{s\})$, notice that the probability to move in a step from a state s_{t-1} to a state s_t following a policy π is given by:

$$P(s_t|s_{t-1}) = \sum_{a_{t-1}} P(s_t|s_{t-1}, a_{t-1})\pi(a_{t-1}, s_{t-1}) \tag{12}$$

So the probability of a specific path $s = [s_1, s_2, s_3, ..., s_T]$, following a policy π, is given by the probability to start from a state s_0, times the probability to move from s_0 to s_1 in one step, times the probability to move from s_1 to s_2, until s_T. Explicitly:

$$\begin{aligned} P_\pi(\{s\}) &\equiv P_\pi([s_0, s_1, s_2, ..., s_T]) = \\ &= P(s_0) \prod_{t'=1}^{T} \sum_{a_{t'-1} \in \mathcal{A}} \{P(s_{t'}|s_{t'-1}, a_{t'-1})\pi(a_{t'-1}, s_{t'-1})\} \end{aligned} \tag{13}$$

If the policy is deterministic, the agent chooses an action that depends only on the current state $a^\pi_{t'} = a^\pi(s_{t'})$ with probability 1:

$$P_\pi(\{s\}) = P(s_0) \prod_{t'=1}^{T} P(s_{t'}|s_{t'-1}, a^\pi(s_{t'-1})) \tag{14}$$

So, explicitly, the expected value of $\mathcal{O}$ is:

$$\mathbb{E}_\pi[\mathcal{O}] = \sum_{\{s\}} \mathcal{O}(\{s\}) P_\pi(\{s\}) = \sum_{\{s\}} \mathcal{O}([s_0, s_1, ..., s_T]) \left\{ P(s_0) \cdot \prod_{t'=1}^{T} \left[\sum_{a_{t'-1}} P(s_t|s_{t'-1}, a_{t'-1})\pi(a_{t'-1}, s_{t'-1}) \right] \right\} \tag{15}$$

And for deterministic policies:

$$\mathbb{E}_\pi[\mathcal{O}] = \sum_{\{s\}} \mathcal{O}([s_0, s_1, ..., s_T]) \left\{ P(s_0) \cdot \prod_{t'=1}^{T} [P(s_t|s_{t'-1}, a^\pi(s_{t'-1}))] \right\} \tag{16}$$

2.1.1 Characterization of the optimal policy

To determine the optimal policy, it is useful to explicit the equation $\pi^*(s) = \text{argmax}_\pi V_\pi(s)$ in function of the transition probability. It is possible to use the equation (16) for expectation values for deterministic policy. Setting $\mathcal{O} = G$, $s = s_0$, $a = a_0 = a(s_0)$, and $s' = s_1$, the optimal policy takes the following form:

Optimal policy

$$\pi^*(s) = \text{argmax}_{a \in \mathcal{A}} \left(V(s')P(s'|s, a) \right) \tag{17}$$

Where $V(s')$ are the maximum utilities. So, knowing the maximum utilities and the transition probabilities, it is possible to reconstruct the optimal policy with this equation.

2.2 The Bellman equations

A good way to evaluate the optimal V and the Q values is to exploit the Markovian nature of our processes, i.e. rewriting the equations exploiting the dependence between the time t and the time $t+1$. This can be done for example for the Value function, taking the equation (6), exploiting G using eq. (5), and using (15) to exploit all terms. Doing the same with Q we end up in the Bellman equations:

$$\begin{cases} V_\pi(s_t) = \mathbb{E}_\pi[r_{t+1} + \gamma V_\pi(s_{t+1})|s_t = s] \\ Q_\pi(s_t, a_t) = \mathbb{E}_\pi[r_{t+1} + \gamma Q_\pi(s_{t+1}, a_{t+1})|s_t = s, a_t = a] \end{cases} \tag{18}$$

Exploiting the dependencies to the transition probability we have the Bellman expectation equations:

$$\begin{cases} V_\pi(s) = \sum_{a \in \mathcal{A}} \pi(a|s) \left[R(s, a) + \gamma \sum_{s' \in \mathcal{S}} P(s'|s, a) V_\pi(s') \right] \\ Q_\pi(s, a) = R(s, a) + \gamma \sum_{s', a'} P(s'|s, a)\pi(a'|s') Q_\pi(s', a') \end{cases} \tag{19}$$

If we maximize respect to a, we end up in the *Bellman optimality equations*:

Bellman optimality equations

$$\begin{cases} V(s) = \max_{a \in \mathcal{A}} \sum_{a \in \mathcal{A}} \pi(a|s) \left[R(s, a) + \gamma \sum_{s' \in \mathcal{S}} P(s'|s, a) V(s') \right] \\ Q_(s, a) = R(s, a) + \gamma \sum_{s' \in \mathcal{S}} P(s'|s, a) \max_{a'} Q(s', a') \end{cases} \tag{20}$$

If we know about the environment and so if we know the transition probability $P(s'|s, a)$, it is possible to solve the Bellman Optimality equation for V, and finally reconstruct the optimal policy thanks to the equation (17). In the next section, I discuss a method to solve the Bellman optimality equations.

2.3 The stochastic frozen lake environment and Bellman solutions

We take into consideration the simple example in Russell's book [18] chapter 17(figure 3). This is equivalent to a stochastic frozen lake environment, a simple and well-known example in the reinforcement learning scientific community [10]. Note that the agent starts also in a random initial state.

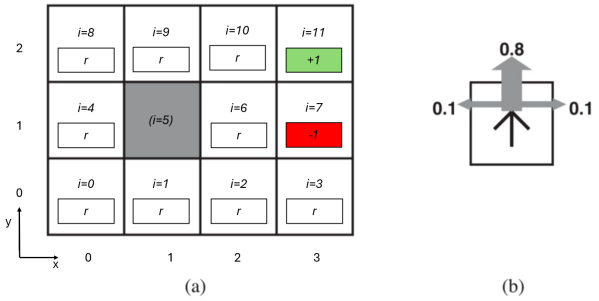


Figure 1: (a) A simple 4x3 environment with a sequential decision problem. Inside the state, I show the indices i associated with the states and the relative reward in rectangular boxes. (b) Illustration of the transition model of the environment: the indented outcome occurs with probability 0.8, but with probability 0.2 the agent moves at right angles to the intended direction. A collision with a wall results in no movement. The two terminal states have reward +1 and -1, respectively. All other states have a fixed reward r. The optimal policy is not always trivial: transition phases occurs depending on the fixed reward of the states r[18].

The way to solve the Bellman optimality equation (20), knowing the full transition probability function, is to initialize $V_0(s) = 0$ and iterate the so-called **Bellman update** for the utility function:

$$\begin{cases} V_{\tau+1}(s) \leftarrow \max_{a \in \mathcal{A}} \left(R(s,a) + \gamma \sum_{s'} P(s'|s,a) V_\tau(s') \right) \\ R(s,a) = \sum_{r \in \mathcal{R}} r \sum_{s' \in \mathcal{S}} P(s', r|s, a) \end{cases} \tag{21}$$

Where $R(s,a)$ is the expected reward. Remember that if the reward values r are deterministic and depend only on the states, we can write

$$R(s,a) = \sum_{s'} r(s') P(s'|s,a).$$

So, explicitly, the Bellman update:

Bellman Update

$$V_{\tau+1}(s) \leftarrow \max_{a \in \mathcal{A}} \left\{ \sum_{s'} P(s'|s,a) \left(r(s') + \gamma V_\tau(s') \right) \right\} \tag{22}$$

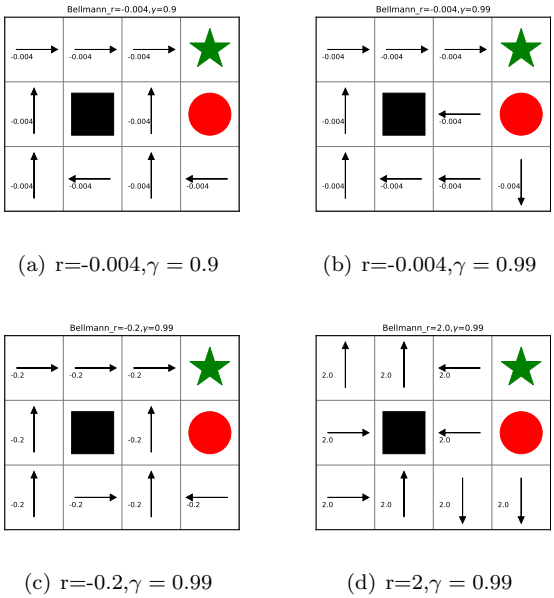
To find the best policy update now (not before), the utility of terminal states as $V(s_{terminal}) = r(s_{terminal})$ and use the equation of the optimal policy (17).

WARNING

Note that this update is slightly different from the Bellman update present in the book of Russel and Norvig [18]. They proposed the following update:

$$V_{\tau+1}(s) \leftarrow r(s) + \gamma \max_a \sum_{s'} P(s'|s,a) V_i(s') \tag{23}$$

Russel and Norvig also consider the starting state's reward when evaluating the return function G, starting the sum from $t' = 0$. In my work instead, I want not to consider that: for a path, the first reward to take into account is *after* our agent took the first action (in fact in the return equation (5) the sum starts from $t' = 1$). The update used in this work (equation (22)) is coherent with the previous theoretical treatment and with the work of Lilian Weng[20]. This difference affects only the reward values, which are changed by a simple factor γ.
My choice of not considering the starting state is justified by the fact that the partially observable algorithms (SARSA and Q-Learning, presented in the next chapter) cannot converge to utility values proposed by Russel, because a reward is always taken after an action.



(a) r=-0.004,$\gamma = 0.9$ (b) r=-0.004,$\gamma = 0.99$

(c) r=-0.2,$\gamma = 0.99$ (d) r=2,$\gamma = 0.99$

Figure 2: Some Bellman solutions varying the state-reward r and the discount factor γ. The optimal policy depends strongly on r. Also γ plays an important role (see the difference between (a) and (b))

3 Reinforcement Learning

In real problems the transition probability $P(s'|s,a)$ is not known. In this case, we have to reconstruct the optimal policy in different ways, without solving directly the Bellman optimality equations. A way to treat this problem is using Reinforcement Machine Learning: the agent needs to explore the environment, see how it responds to his actions, and acquire experience, and only after a good exploration can have the ability to recognize good policies. To do that we can simulate a different number of *episodes*, for a total of $N_{episodes}$, each one starting from different positions allowing our agent to discover the environment.
With SARSA and Q-Learning algorithms, it is possible to infer the value functions from experiences. The implementation of SARSA and Q-learning becomes computationally expensive when the states space becomes large and dense: in this case, it is possible to approximate the solution through a deep neural network (deep Q-learning).

3.1 SARSA and Q-Learning

SARSA and Q-Learning are two algorithms of *Temporal Difference learning*[20], i.e. our agent can learn from the experience, also with incomplete episodes. We want to find value functions, i.e., the expected value of G of our states, from experiences. It is possible to use the self-consistency of the Bellman equation to update the value functions at each step of the simulation. For example for V values, to approximate solutions of the Bellman equation (19), it is possible at each experience collected by the agent to reduce the distance between $V(s_t)$ and $r_{t+1}+\gamma V(s_{t+1})$. The update has to take into account also old computed values. This can be done defining the hyperparameter α as a *learning rate*:

$$V(s_t) \leftarrow V(s_t) + \alpha(r_{t+1} + \gamma V(s_{t+1}) - V(s_t)) \tag{24}$$

The same for Q-value:

$$Q(s_t,a_t) \leftarrow Q(s_t,a_t) + \alpha(r_{t+1} + \gamma Q(s_{t+1},a_{t+1}) - Q(s_t,a_t)) \tag{25}$$

This is the idea of SARSA, and it is called *on-policy*-algorithm because it updates the value function depending on the action a_{t+1} chosen by the agent (i.e. his policy). So, to reach a good convergence, it is necessary that the agent chooses optimal actions.
The Q-learning algorithm instead is called *off-policy* because the values are updated independently of the policy, and it can reach convergence also without choosing optimal actions:

$$Q(s_t,a_t) \leftarrow Q(s_t,a_t) + \alpha(r_{t+1} + \gamma \max_a Q(s_{t+1},a) - Q(s_t,a_t)) \tag{26}$$

Initially, the values of Q (and so also V) are completely unknown. So it is a good idea initially to start with random actions (*exploration*). After some exploration, the agent can choose optimal actions (*exploitation*). The fact that we do not know the transition probability tensor, i.e., we do not know the environment, introduces in SARSA a new self-consistent term: the action must follow the optimal policy, but the optimal policy depends on the Q values.

SARSA Algorithm

1. Starting in a generical t, you are in s_t, choose an action $a_t = \text{argmax}_{a\in\mathcal{A}} Q(s_t,a)$. If Applying an ϵ-greedy algorithm (where ϵ is the *exploration parameter*), you choose instead a random action with probability ϵ;
2. Take the reward r_{t+1};
3. Take another action
$$a_{t+1} = \text{argmax}_{a\in\mathcal{A}} Q(s_{t+1},a)$$
4. Now you can update Q:
$$\begin{aligned} Q(s_t,a_t) \leftarrow Q(s_t,a_t) + \\ + \alpha(r_{t+1} + \gamma Q(s_{t+1},a_{t+1}) - Q(s_t,a_t)) \end{aligned} \tag{27}$$
5. Set $t \leftarrow t+1$ and repeat.

Q-Learning Algorithm

1. Starting in a generical t, you are in s_t, choose $a_t = \text{argmax}_{a\in\mathcal{A}} Q(s_t,a)$. Applying an ϵ-greedy algorithm, you choose instead a random action with probability ϵ;
2. Take the reward r_{t+1};
3. Update Q:
$$\begin{aligned} Q(s_t,a_t) \leftarrow Q(s_t,a_t) + \\ + \alpha(r_{t+1} + \gamma \max_a Q(s_{t+1},a) - Q(s_t,a_t)) \end{aligned} \tag{28}$$
4. Set $t \leftarrow t+1$ and repeat.

The implementation of terminal states can be achieved automatically by initializing the Q values at zero.

3.2 Deep Q-learning

It is not always possible to use SARSA and Q-learning, especially for complex problems where the computational cost becomes easily big. For this reason, a powerful way to solve a Markov decision problem is to use deep neural networks to approximate the Q function (*Deep-Q learning*).

3.2.1 The Deep Q Network

A generic Deep Neural Network (DNN) consists of processing units called neurons. A neuron can store a real value x, and process it through a non-linear function $a(x)$, also called

activation function (a possible choice for example is ReLU $a(x) = max(0, x)$). The neurons of a DNN are organized in layers[4]:

1. Input layer: in the input neurons are stored the input values of the network;
2. Hidden layers: a set of neurons between input and output layers;
3. Output layer: the output neurons store the output values of the networks.

Each neuron of one layer l is linked with the neurons of the successive layer $l+1$, from the input layer ($l = 1$) to the output layer ($l = L$). Denoting with $x_i^{[l]}$ the value stored in the i-th neuron of the l-th layer, the general formula for the values stored in the $l+1$-th layer's neurons in a DNN is:

$$x_i^{[l+1]} = \sum_j \theta_{ij}^{[l]} a\left(x_j^{[l]}\right) + b_i^{[l+1]} \tag{29}$$

Where $\theta_{ij}^{[l]}$ is a weight matrix that *links* the neurons from the l-th activated layer to the successive, and $b_i^{[l+1]}$ are *biases* of the neuron. In a more compact way:

$$\vec{x}^{[l+1]} = \overleftrightarrow{\theta}^{[l]} \vec{a}(\vec{x}^{[l]}) + \vec{b}^{[l+1]} \tag{30}$$

The output $\vec{f}_{DNN}$ of a deep neural network is so a non-linear function of the input $\vec{x}^{[1]}$ and of all the weights θ. For example for a DNN consisting of 3 layers, with zero biases:

$$\vec{f}_{DNN}(\vec{x}^{[1]}, \theta) \equiv \vec{a}_{out}(\vec{x}^{[3]}) = \vec{a}_{out}\left(\overleftrightarrow{\theta}^{[2]} \vec{a}\left(\overleftrightarrow{\theta}^{[1]} \vec{a}(\vec{x}^{[1]})\right)\right) \tag{31}$$

Often the same activation function $\vec{a}$ is chosen along the network, except for the output layers. In fact $\vec{a}_{out}$ is chosen depending on the specific problem, and it may also be omitted. Deep neural networks are useful because they are *universal function approximators*, i.e. it is possible to approximate every function with arbitrary precision with a big enough DNN[17]. The approximation is reached by *training* the neural networks: it is possible to minimize the distance between the output and the target function using iterative techniques such as *backpropagation* [19]. This will be discussed further in the next section.

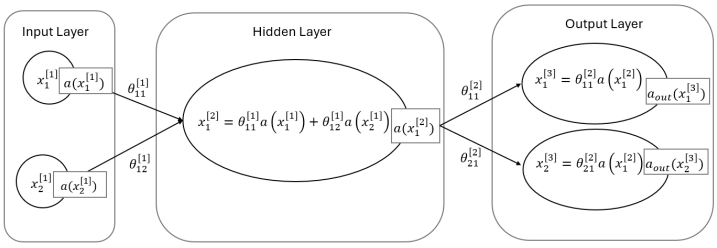


Figure 3: An explicit illustration of a simple neural network with 3 layers: 2 input neurons, 1 hidden neuron, and 2 output neurons.

It is possible to use a DNN to approximate the Q values: in this case, the network is called Deep-Q Network(DQN)[8]. It takes in input a preprocessed image of the state $\phi(s)$, and returns as output a vector with the dimensionality of the number of possible actions(see figure 4). The approximation of Q is stored in each entry of the network's output, which I denote as $Q(s, a, \theta)$.

3.2.2 Backpropagation and Loss Function

Training a DNN requires defining a loss function $\mathcal{L}$, minimizing it with respect to the neural network weights. The gradient descend rule for the weights is:

$$\theta \leftarrow \theta - \alpha \nabla_\theta \mathcal{L}(\theta) \tag{32}$$

Where α is the learning rate. The backpropagation[14, 19] is a method based on chain rules to evaluate analytically the gradients of the loss. In particular, the backpropagation uses the fact that the derivatives of each calculation done in each evaluation step, from the input of the neural network to the evaluation of the loss, have simple analytical forms. From the value of the loss function, it is possible to find iteratively the derivatives relative to the precedent evaluation step (i.e. the derivatives are computed *backward*). For example, the loss function can be always expressed as a simple function of the output of the neural network $\mathcal{L}(f_{DNN})$, and the analytical form of the derivative $\frac{\partial \mathcal{L}}{\partial f_{DNN}}$ is known: from the value of $\mathcal{L}$ and f_{DNN} it is easily possible to estimate the analytical value of these derivatives. Using the chain rule, the derivatives with respect to a weight θ_i can be expressed as:

$$\frac{\partial \mathcal{L}}{\partial \theta_i} = \frac{\partial \mathcal{L}}{\partial f_{DNN}} \frac{\partial f_{DNN}}{\partial \theta_i}$$

The first term was easily evaluated before, and the second can be evaluated by reapplying the chain rule inside the neural network (as seen in eq (29), all the steps involve compositions of linear and activation functions, with known derivatives). Now it should be clear that the evaluation of the derivatives proceeds backward because the chain rule proceeds backward. The backpropagation can be easily implemented using Pytorch library.

Using the Q-Learning approach (equation (26)), a good loss function to use in our problem is the mean square error (MSE) between $Q(\theta, s, a)$ and $y \equiv r + \gamma \max_a Q(\theta^-, s, a)$, where $Q(\theta^-, s, a)$ is evaluated by another Deep Q-Networks, that uses some old parameters θ^-.

Every C epochs we update the old Q:

$$Q(\theta^-, s, a) \leftarrow Q(\theta, s, a)$$

I use also the concept of replay memory[16]: the recent history of the agent is stored in a memory, and the training of the NN is done on a minibatch sampled from a uniform distribution of this memory. In this way the training steps are uncorrelated, giving more stability. Calling D the replay memory, so the

loss function is the expected value of the MSE with respect to a minibatch sampled uniformly from the replay memory [8]:

DQL loss function

$$\begin{cases} \mathcal{L}(\theta) = \underset{\mathcal{U}(D)}{\mathbb{E}}\left[(y - Q(s,a,\theta))^2\right] \\ y = r + \gamma \max\limits_{a'} Q(s',a',\theta^-) \end{cases} \tag{33}$$

3.2.3 Deep-Q-Learning algorithm with experience replay

The deep Q-learning algorithm is a Q-learning algorithm with replay memory. Calling and $\phi(s)$ the a preprocessed image of the state s, the full Deep-Q-Algorithm[16] (see figure 4 for an intuitive illustration):

Deep Q Learning algorithm

Initialize replay memory D to capacity N, initialize the neural networks Q and Q^- with same weights.
Simulate M episodes.
For each episode and for each time t of the episode:

1. With probability ϵ select a random action, otherwise select $a_t = \text{argmax}Q(\phi(s_t), a, \theta)$
2. Execute the action, and observing the the state s_{t+1} and the relative reward r_t, store the experience $(\phi(s_t), a_t, r_t, \phi(s_{t+1}))$ in D
3. Sample random minibatch of experiences $(\phi_j, a_j, r_j, \phi_{j+1})$ from D
4. Perform a gradient descend step on $(y_j - Q(\phi_j, a_j, \theta))^2$ with respect to the weights θ, with

$$\begin{cases} y_j = r_j, \text{ if the episode terminates at step } j+1 \\ y_j = r_j + \gamma \max\limits_{a} Q^-(\phi_{j+1}, a, \theta^-), \text{ otherwise} \end{cases} \tag{34}$$

5. Every C step reset $Q^- \leftarrow Q$

This method is really powerful because it can be used in different scenarios: it is possible to use as input a compressed and preprocessed image, that represents a state. Through the Deep Neural network, you can infer the best action to do given that input, without having in memory the whole Q tensor. This makes it possible to find good policies also for really complex problems[8, 11].
For our simple problem a state can be represented by a hot-encoded vector of 12 entries, each representing a position in our discrete space (the first entry represents the position $[x, y] = [0, 0]$, the second $[1, 0]$, the third $[2, 0]$, the fourth $[3, 0]$, ...). For example if our agent is in the position $[0, 0]$, it can be represented by a hot-encoded-vector:

$s = \text{agent in}[0,\ 0] \Rightarrow \phi(s) = [1,\ 0,\ 0,\ 0,\ 0,\ 0,\ 0,\ 0,\ 0,\ 0,\ 0,\ 0]$

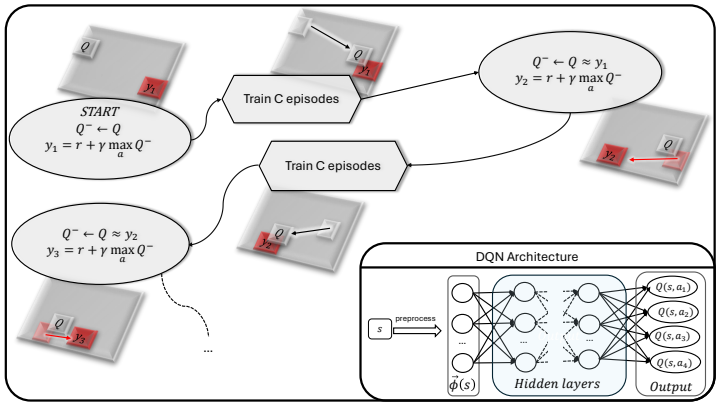


Figure 4: Intuitive illustration of the Deep Q-Learning algorithm and Deep Q-Network architecture. It is necessary to initialize 2 identical neural networks: Q^- has fixed weights and it is used to evaluate y, while Q is trained to try to approximate y. Q^- is updated every C episodes (or steps, you are free to choose), changing the target y.

3.3 Comparisions between classical algorithms

It is possible to compare the performances of SARSA, Q-learning, and Deep-Q-Learning, in the stochastic frozen lake environment. A parametrical analysis brings to the following conclusions (referring to figure 5):

1. The optimal values of the learning parameters can vary changing the environment statistic parameters. It is not always easy to reach the optimal policy convergence, particularly in environments where different policies result in only minor differences in path values (and so may be difficult for the algorithms to comprehend which one is better). This often happens near phase transitions. In these cases, it is better to use more episodes. Note that in reality, this is not a big problem, because often the goal is to reach a good policy, and may be not worth stressing too much for converging to a policy that gives an irrelevant improvement of the value function.

2. Optimizing the hyperparameters the algorithms converge to best policies. An initial exploration (i.e. start with ϵ high) is necessary to reach fast good convergence.

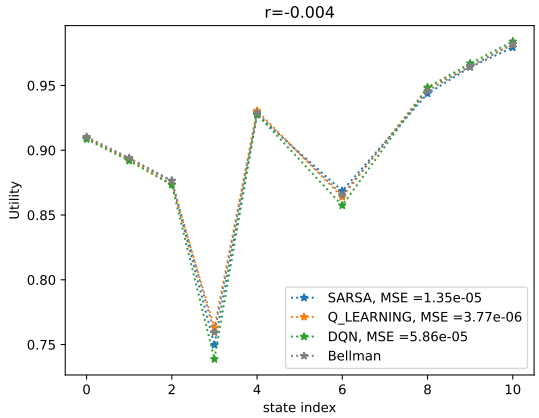
3. Q-learning and deep-Q-learning, being *off-policy* algorithms, converge to the correct policy also with a fixed ϵ, while for SARSA is necessary to start with a finite ϵ and make it decay to zero during the training (see figure 5 (a) and (b)). Decaying ϵ works well also in Q-learning and DQL.

4. Often for this problem, with good parameter setting, $N_{episodes} \approx \mathcal{O}(10^4 - 10^5)$ is enough to reach a good convergence. Generally SARSA requires more time because it has to stabilize after ϵ reaches zero (see figure 5 (c));

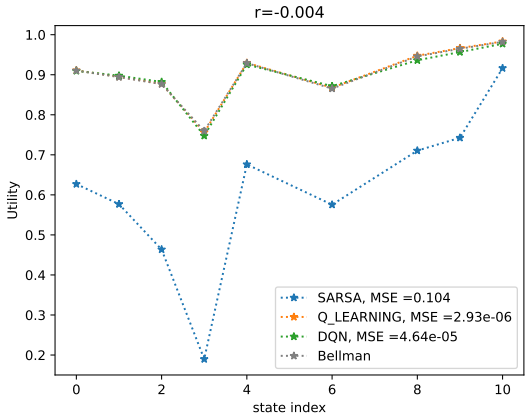
5. For SARSA and Q-learning, a good value of learning rate α is between $0,01 - 0,2$ (depending also on the number of episodes of the simulation). To reach better results it is possible to start with a high learning rate and reduce it during the training. For DQN it is good to start from a learning rate of $\approx 10^{-3}$ and make it decay until $\approx 10^{-5}$, but these values depend also on the architecture of the specific DQN. Xavier Uniform initialization[12] of weights works tendentially well.

6. Generally, Q-learning converges more easily and faster than SARSA. DQN converges faster when using batch sizes greater than 1, but excessively large batch sizes can reduce precision. The replay memory should not be too large, as initial experiences are random. As training progresses, the network accumulates newer and better experiences: after a certain amount of time, we want to throw away old memories.

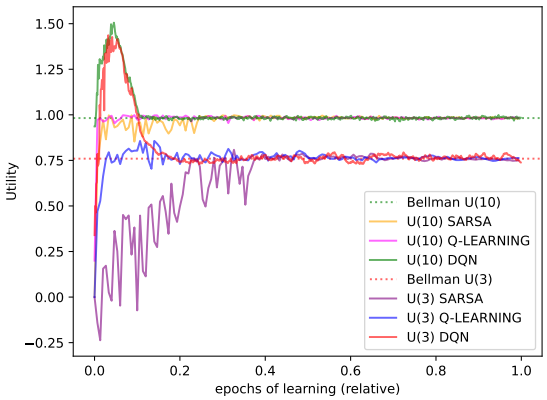
7. The quality of the convergence of DQN depends strongly on the number of neurons, the complexity of the network, the initialization of the weights, and all the parameters related to the training loop. This makes the optimization of DQN not easy: it is important to find the correct amount of neurons and layers that make the training process fast. Too complex networks in fact may be hard to train, and they may require a way larger amount of steps, while a small amount of neurons may be not enough to approximate correctly the Q-values. A good initialization is necessary to avoid divergences.



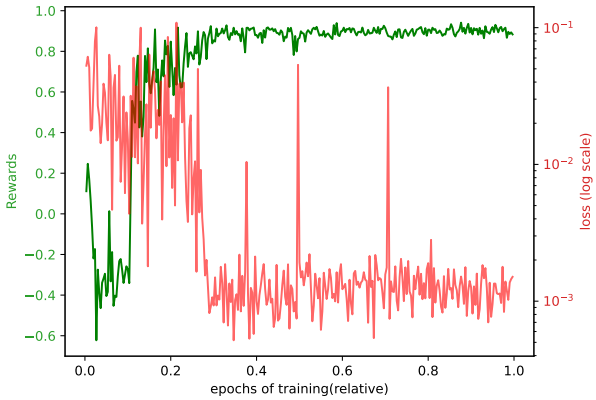
(a) Decaying ϵ



(b) Static $\epsilon = 0,6$



(c) Convergence of utilities of states 3 and 10 during the training (decaying ϵ).



(d) DQN training loss and reward (decaying ϵ)

Figure 5: Confront between SARSA, QL, and DQN convergences in the stochastic frozen lake environment with $r = -0,004$, $\gamma = 0,99$, $N_{episodes} = 30000$, $\epsilon_0 = 0,6$. For QL and SARSA, I set the learning rate $\alpha_0 = 0,2$, halving it every 5000 episodes with $\alpha_{min} = 0,01$.
For DQN I use 2 hidden layers of 32 neurons each, using uniform Xavier initialization and ReLU activation functions (and no activation function in output). The learning rate is $LR_0 = 5 \cdot 10^{-3}$, decaying during the training loop until $LR_{min} = 5 \cdot 10{-}5$. I use a replay memory size of 800 experiences and batch_size= 2. The state indices in (a), (b), and (c) plots are related to the 2D positions (x,y) through the following identity: state_index = x%L_x + y $\cdot$ L_x. Positions relative to ending states or obstacles are excluded in (a) and (b).
Figure (a): with decaying ϵ the algorithms converge well: it is implemented a linear decay for the first half of the total training epochs, with $\epsilon_{min}^{SARSA} = 0$; $\epsilon_{min}^{QL} = 0,01$; $\epsilon_{min}^{DQN} = 0,001$.
Figure (b): With a static $\epsilon = 0,6$ SARSA is not able to reach convergence to the optimal policy, with respect to QL and DQN.
Figure (c): it shows the convergence of the utility function of states 3 and 10 for the algorithms during the training (decaying ϵ). SARSA is slower with respect to the other algorithms due to the fact that it is an *on-policy* algorithm. Q-learning is the one that converges fastest. Note that reaching convergence for state 3 is harder because it is a state harder to reach respect to 10.
Figure (d): Rewards taken(green) and loss function (red) related to DQN during the training loop.

4 Variational quantum circuits for RL: a hybrid approach

Training a Deep-Q-Network is still computationally expensive, especially for large-scale problems. Hybrid quantum-classical

algorithms may help to reduce the workload and the number of parameters required[10]. In recent years lots of quantum architectures have been developed, with quite promising results. N. Meyer et al. published a complete survey recently[9]. A possible and simple approach is to substitute the Deep-Q-Network with a Variational Quantum Circuit to approximate the Q-values. This approach was proposed by Chen et al. in 2020 [10], and improved in 2022 by Skolik et al. [1]. Variational quantum circuits use parametrized unitary transformations $\hat{\mathcal{U}}_\theta$ to encode and process input data and solve problems using fewer parameters concerning classical neural networks. A variational quantum circuit consists of 3 mains blocks (see figure 6 b):

1. **Encoding input block**: some unitary operations are used to encode the input. The input in our Neural network is a state s, i.e. a position in the frozen lake environment. I transform the index related to the state ($i = \mathrm{x}\%Lx + y \cdot Lx$) in binary representation(12 states $\rightarrow$ 4 bits) and encode it in 4 qubits:

$$|\psi\rangle\ (i=0) = |0\rangle_0 |0\rangle_1 |0\rangle_2 |0\rangle_3$$

$$|\psi\rangle\ (i=1) = |0\rangle_0 |0\rangle_1 |0\rangle_2 |1\rangle_3$$

$$\dots$$

Since a Pauli rotation is $R_\alpha(\theta) = e^{-i\frac{\theta}{2}\sigma_\alpha}$, with σ_α Pauli matrices and $\alpha = x, y, z$, it is possible to transform a ket $|0\rangle$ to $|1\rangle$ with two rotations:

$$\hat{R}_z(\pi)\hat{R}_x(\pi)|0\rangle = |1\rangle$$

This means that, starting from $\prod_j |0\rangle_j$, the input state representation can be achieved by applying these two operators to the qubits of the circuit:

$$\prod_j |a_j\rangle_j = \prod_j \hat{R}_z^{[j]}(\pi\delta_{a_j,1})\hat{R}_x^{[j]}(\pi\delta_{a_j,1})|0\rangle_j$$

With $\delta_{a_j,1}$ is the Kronecker δ.

2. **Variational block**: entangling transformations and generic rotations are performed, with some parameters to optimize. The variational block is often composed by some entangling controlled gates (for example CNOT), and general parametrized rotations. To increase the expressivity the variational block can be repeated. To improve the performances, it is possible to implement also the so-called *data-reuploading*: in this case, also the encoding input block is repeated with the variational block[1].

3. **Output**: The output is given by measuring the expected value of some observable $\langle O\rangle$ on the output qubits. This can be rescaled and post-processed. It is also possible to build hybrid Networks: the output of the Quantum Deep Network can be post-processed by a classical Deep Neural Network.
In our case the agent has 4 possible actions(0 = up, 1 = right, 2 = down, 3 = left), and we want to approximate the Q values: $Q(s, a_j)$ can be represented by the expected value z-component of the jth output spin of the circuit(with values bound between -1 and 1). Depending on the initialization of the stochastic frozen lake environment (on the choice of the state rewards r in Figure 1), rescaling of the output may be necessary to achieve better convergence.

4.1 Evaluating gradients in quantum circuits

To solve reinforcement learning problems, the deep-Q-learning algorithm(figure 4) is preserved: we have just replaced the deep Q network with a Quantum Variational Circuit. To optimize the parameters of the variational block we still use the gradient descend rule:

$$\theta \leftarrow \theta - \alpha\nabla_\theta\mathcal{L}(\theta)$$

When simulating quantum circuits it is still possible to use chain rules from analytical results to do the backpropagation. In a physical quantum computer it is not possible to use the chain rule, but it is still possible to evaluate analytically the gradients using the *parameter-shift-rule*[13, 7, 5]:

Parameter shift rule

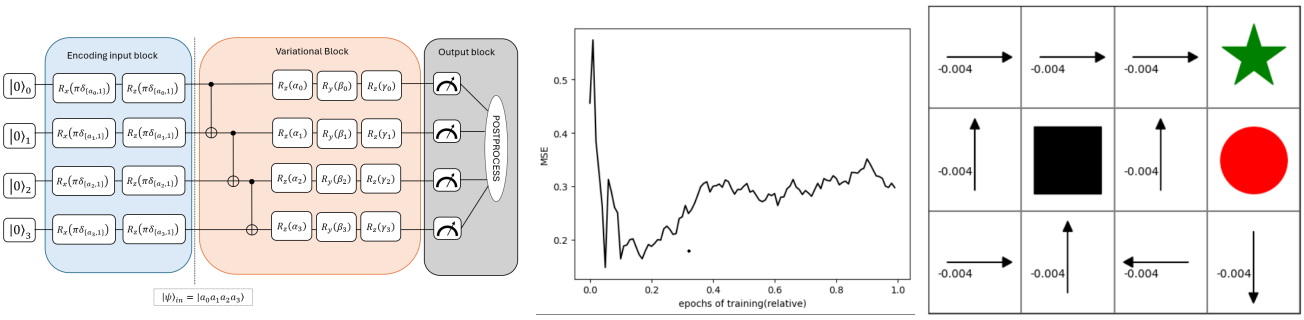
Let be O an observable on the output of the quantum circuit, and suppose that the quantum circuit is composed by some parametric unitary rotations $R_i(\theta_i) = e^{-i\frac{\theta_i}{2}\sigma_i}$. So it is possible to evaluate the analytical gradients of $\langle O\rangle$ respect to θ_i, by evaluating the expectation values of O shifting $\theta_i \pm \frac{\pi}{2}$:

$$\nabla_{\theta_i}\langle O(\theta)\rangle = \frac{1}{2}\left(\langle O(\theta_i + \pi/2)\rangle - \langle O(\theta_i - \pi/2)\rangle\right) \quad (35)$$

4.2 Possible advantages of quantum approaches

Quantum/hybrid reinforcement learning is still an open field of research, and in recent years, interest in it has significantly increased, due to the possibility that variational quantum circuits sometimes learn optimal policy faster[6, 15], and with less amount of parameters related to the architecture as well[10], with respect to classical Neural Networks. Implementation difficulties arise due to the limited availability of quantum hardware, and simulating quantum circuits is costly. Furthermore, not always it is easy to choose a good architecture and find good hyperparameters to make them converge: these Variational Quantum Deep Q Learning approaches are subject to instabilities that may cause the learned policy to diverge[6]. I made some attempts to implement in the stochastic Frozen Lake Environment(Figure 1) the simplest Deep Q-Learning variational circuit (the circuit in figure 6 b), the same used by Chen for the deterministic Frozen Lake Environment[10], but without success. Indeed, I have encountered instabilities, and

the network easily becomes trapped in local minima. Notice that the optimization of hyperparameters requires more time than the classical neural networks because the time of simulations is drastically increased. Probably more attempts have to be made, and stabilization techniques, proposed by Maya et al.[6], have to be implemented to reach good results.



(a) Illustration of the variational quantum circuit proposed by Chen[10]

(b) MSE of utilities(respect to Bellman solutions) in function of the training epochs

(c) Policy

Figure 6: (a)Variational Quantum Deep Q Learning algorithm and (b)an example of Variational Quantum Circuit, used by Chen et al. [10] for solving deterministic Frozen lake environment. in the original idea, only the Variational block has to be repeated. In (c) and (d) it is shown an attempt with 5 variational blocks,$N_{episodes} = 1000$, and torch RMSprop optimizer (Lr=0,01). Initially, the Mean Square Error (of Utilities respect to Bellman solutions) is going better, but after it restarts to diverge. The optimal policy is better than a random one, but still it needs further optimizations .

5 Conclusion

The conclusions are summed up in the following points:

- From the analysis of the classical algorithms SARSA and Q-Learning, Q-Learning is faster, being an *off-policy* algorithm, i.e. does not need to wait for the decaying of the exploration parameter to be 0.

- Deep Neural Networks can be used to approximate the Q-values (*Deep-Q-Learning*), allowing to solve complex problems without storing the full Q tensor. The optimization of training hyperparameters is generally more challenging than in supervised machine learning problems. However, with optimized parameters, rapid convergence can be achieved.

- I discussed the implementation of Variational Quantum Circuits as substitutes for Deep Neural Networks in Deep-Q-Learning. An attempt to simulate and solve the stochastic-frozen-lake environment has been made with the simplest Variational Quantum Circuit proposed by Chen[10], but the time required to train and simulate didn't permit me to reach really good results. More attempts and further optimizations, maybe based on stabilization techniques [1] could improve the performances.

Bibliografy

[1] Andrea Skolik et al. "Quantum agents in the Gym: a variational quantum algorithm for deep Q-learning". In: (2022).

[2] David Silver et al. "Mastering the game of Go with deep neural networks and tree search". In: (2016).

[3] David Silver et al. "Mastering the game of Go without human knowledge". In: (2017).

[4] Harsh Kukreja et al. "AN INTRODUCTION TO ARTIFICIAL NEURAL NETWORK". In: (2016).

[5] K. Mitarai et al. "Quantum Circuit Learning". In: (2019).

[6] Maja Franza et al. "Uncovering Instabilities in Variational-Quantum Deep Q-Networks". In: (2022).

[7] Maria Schuld et al. "Evaluating analytic gradients on quantum hardware". In: (2018).

[8] Mnih et al. "Human-level control through deep reinforcement learning". In: (2015).

[9] Nico Meyer et al. "A Survey on Quantum Reinforcement Learning". In: (2024).

[10] S.Y. Chen et al. “Variational Quantum Circuits for Deep Reinforcement Learning”. In: (2020).

[11] Volodymyr Mnih et al. “Playing Atari with Deep Reinforcement Learning”. In: (2013).

[12] Siddharth Krishna Kumar. “On weight initialization in deep neural networks”. In: (2017).

[13] Oleksandr Kyriienko, Annie E. Paine, and Vincent E. Elfving. “Solving nonlinear differential equations with differentiable quantum circuits”. In: (2021).

[14] Fei-Fei Li, Justin Johnson, and Serena Yeung. *Lecture 4: Backpropagation and Neural Networks*. 2017. URL: https://cs231n.stanford.edu/slides/2017/cs231n_2017_lecture4.pdf.

[15] Owen Lockwood and Mei Si. “REINFORCEMENT LEARNING WITH QUANTUM VARIATIONAL CIRCUITS”. In: (2020).

[16] S. Tellex M. Roderick J. MacGlashan. “Implementing the Deep Q-Network”. In: (2017).

[17] Vitaly Maiorov and Allan Pinkus. “Lower bounds for approximation by MLP neural networks”. In: (1999).

[18] P. Norvig S. Russell. *Artificial Intelligence, A Modern Approach*. Pearson, 2010.

[19] Golnaz Moharrer Saeed Damadi, Mostafa Cham, and Jinglai Shen. “The backpropagation algorithm for a math student”. In: (2023).

[20] Lilian Weng. “A (Long) Peek into Reinforcement Learning”. In: *lilianweng.github.io* (2018). URL: https://lilianweng.github.io/posts/2018-02-19-rl-overview/.