Reinforcement Learning Notes

Gianluca Peri

January 24, 2025

Contents

Introduction

Given a problem to solve we can usually frame its machine learning solution in 3 different ways:

- Supervised Learning: the model is given a dataset of input-output pairs and has to learn the function that maps the inputs to the outputs.
- Unsupervised Learning: the model is given a dataset of inputs and has to find some structure in it.
- Reinforcement Learning: the model (agent) is given a reward signal and has to learn the best actions to take in order to maximize the cumulative reward.

Of course the best frame depends on the problem at hand. Reinforcement Learning (RL) is particularly suited for problems with the following characteristics:

- The problem can be modeled with an agent interacting with an environment.
- The agent's actions lead to different future observed states of the environment (so the agent influences the learning material).
- Good actions do not always yeld an immediate reward.

In this notes we will focus on the basics of RL.

1 Basic Concepts of Reinforcement Learning

Reinforcement Learning deals with agents that can be in a set of states $s \in \mathcal{S}$ and can take actions $a \in \mathcal{A}$. The agent receives a reward $R \in \mathbb{R}$ after arriving in a new state (sometimes the reward can be zero, or negative); see Figure 1. In this context we usually like to work in discrete time.

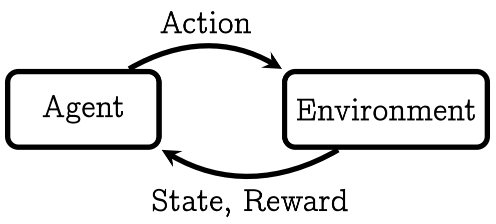

Figure 1: Simple depiction of the agent-environment interaction. We can see that the agent-environment interaction is (usually) modeled as a *Markov Decision Process* (MDP), where the agent can take actions that influence the environment and receive rewards.

The agent's goal is to maximize the expected cumulative reward, that is formally defined as:

$$G_t \doteq R_{t+1} + \gamma R_{t+2} + \ldots + \gamma^{T-t-1} R_T = \sum_{k=t+1}^{T} \gamma^{k-t-1} R_k \tag{1}$$

where $\gamma \in [0, 1]$ is the *discount rate*, this parameter exists to give more importance to immediate rewards. Note that γ can also be 0 or 1. Note also that the time T marks the end of context window for the agent: this window is usually called *episode*.
An agent is always equipped with a policy $\pi(a|s)$ that specifies the probability of taking action a in state s. Notice that if the state space and the action space are finite, the policy can always be represented tabularly as a tensor.

It is customary to define, for a policy, the value function $v_\pi(s)$ as the expected cumulative reward starting from state s and following policy π:

$$v_\pi(s) \doteq \mathbb{E}_\pi\left[G_t|S_t = s\right]. \tag{2}$$

We can also define another function that will become useful later on, the *action-value function* $q_\pi(s,a)$. This function represents the expected cumulative reward starting from state s *and* taking action a:

$$q_\pi(s,a) \doteq \mathbb{E}_\pi\left[G_t|S_t = s, A_t = a\right]. \tag{3}$$

It's important to notice the following relations between $v_\pi(s)$ and $q_\pi(s,a)$. We can derive the value function from the action-value function:

$$v_\pi(s) = \sum_a \pi(a|s) q_\pi(s,a). \tag{4}$$

and *vice versa*, we can derive the action-value function from the value function:

$$q_\pi(s,a) = \sum_{s',r} p(s',r|s,a)\Big[r + \gamma v_\pi(s')\Big]. \tag{5}$$

If equation (5) is not yet clear it will become evident once we have derived the Bellman equation; but just to give a hint: the main logic behind it is that the action-value function is simply the expected value of the next state's reward *plus* the discounted value of the other expected rewards (which are given by the value function $v_\pi(s')$).
Once again note that if the state space and the action space are finite, the value function and the action-value function can always be represented tabularly as tensors; in particular the tabular representation of the action-value function is usually called the *Q-table*.
By now it should be clear that one of the main conceptual points behind reinforcement learning, if not *the* main one, is recognizing the huge difference between the *reward* r associated to a state and the *value* v_π (or *quality* q_π) of that state.

2 Bellman Equation

Bellman equation is *the* foundamental equation in reinforcement learning. It is a recursive equation that relates the value function of a state to the value function of its successor state*s*. To derive it let's start from the definition of $v_\pi(s)$:

$$v_\pi(s) = \mathbb{E}_\pi[G_t|S_t = s]$$

from here we can observe that the argument G_t follows the following recursive relation:

$$v_\pi(s) = \mathbb{E}_\pi\Big[G_t|S_t = s\Big] = \mathbb{E}_\pi\Big[R_{t+1} + \gamma G_{t+1}|S_t = s\Big] \tag{6}$$

from here we can leverage the linearity of the expectation value operator:

$$v_\pi(s) = \mathbb{E}_\pi[R_{t+1}|S_t = s] + \gamma\mathbb{E}_\pi[G_{t+1}|S_t = s] \tag{7}$$

We can now make progress by expanding the two addends: calling the states at time $t+1$ s':[1]

$$\mathbb{E}_\pi[R_{t+1}|S_t = s] = \sum_{s',a,r} p(s',r|s,a)r \tag{8}$$

where $p(s',r|s,a)$ is the probability of transitioning to state s' (and receiving reward r), given to be in state s and take action a. Notice that if the reward is *deterministic* (one state, one reward), then the sum is over s' and a alone.
For the second addend we can write:

$$\gamma\mathbb{E}_\pi[G_{t+1}|S_t = s] = \gamma\sum_{s',a}\pi(a|s)p(s'|s,a)\mathbb{E}_\pi[G_{t+1}|S_{t+1} = s'] \tag{9}$$

but now we can insert a resolution of the identity (since $\sum_r p(r) = 1$):

$$\begin{aligned}\gamma\mathbb{E}_\pi[G_{t+1}|S_t = s] &= \gamma\sum_{s',a}\pi(a|s)p(s'|s,a)\mathbb{E}_\pi[G_{t+1}|S_{t+1} = s'] = \\ &= \gamma\sum_{s',a,r}\pi(a|s)p(s',r|s,a)\mathbb{E}_\pi[G_{t+1}|S_{t+1} = s'].\end{aligned} \tag{10}$$

In light of this we can rewrite (7) as:

$$v_\pi(s) = \sum_a \pi(a|s)\sum_{s',r}p(s',r|s,a)\Big[r + \gamma\mathbb{E}_\pi[G_{t+1}|S_{t+1} = s']\Big]. \tag{11}$$

And finally, from (11), keeping in mind the definition of the value function (2), we can write the canonical *Bellman Equation:*

$$v_\pi(s) = \sum_a \pi(a|s)\sum_{s',r}p(s',r|s,a)\Big[r + \gamma v_\pi(s')\Big]. \tag{12}$$

[1]If we assume that each action a be possible in each state s, our derivation obviously holds. But even if this wasn't the case, we could still derive the Bellman equation: it would be sufficient to pay attention to which actions are possible in which states, by defining the spaces of actions $\mathcal{A}(s)$), but given this the Bellman equation would still hold in the same form.

This equation is crucial, since it allows us to compute the value function of a state by knowing the value function of its successor states. It will be heavily used from here on out.

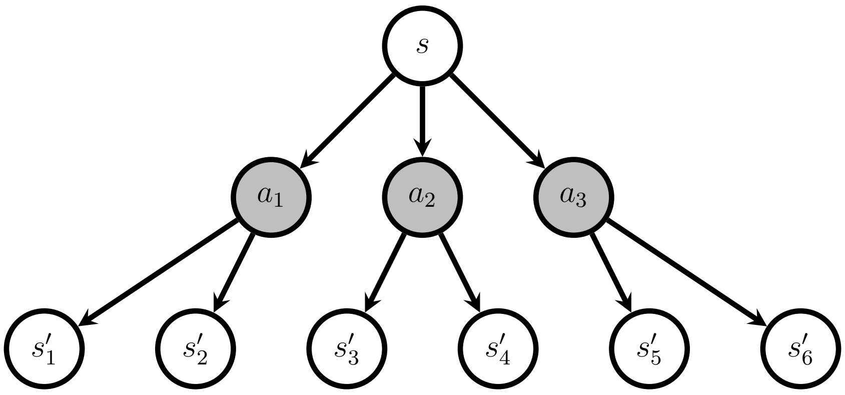

Figure 2: Example of a simple tree representation of the Bellman Equation. The value function $v_\pi(s)$ of a state s can be computed by knowing the value functions of its successor states s'. This kind of diagrams are usually called *Backup Diagrams*.

3 Optimality

We wish to define an ordering in the space of all the possible policies: we say that a policy π is better or equal (greater or equal) than a policy π' if it's expected return is greater or equal than the expected return of π' for all states $s \in S$. So:

$$\pi \geq \pi' \iff v_\pi(s) \geq v_{\pi'}(s) \; \forall s \in S.$$

This ordering allow us to define the *optimal policy* π_* as simply the policy that is greater or equal to all other policies.
All optimal policies in a given context share the same value function, called the *optimal value function* $v_*(s)$:

$$v_*(s) \doteq \max_\pi v_\pi(s) \; \forall s \in S. \tag{13}$$

Similarly, the optimal policies also share the optimal action-value function $q_*(s, a)$:

$$q_*(s, a) \doteq \max_\pi q_\pi(s, a) \; \forall s \in S, a \in A. \tag{14}$$

Finally note:

$$\begin{gathered} q_*(s,a) = \mathbb{E}_{\pi_*}\Big[G_t | S_t = s, A_t = a\Big] \;\Rightarrow \\ \Rightarrow q_*(s,a) = \mathbb{E}_{\pi_*}\Big[R_{t+1} + \gamma G_{t+1} | S_t = s, A_t = a\Big] \end{gathered} \tag{15}$$

but since π_* is optimal, and that it is given that we are in state s, we can essentially substitute G_{t+1} with $v_*(S_{t+1})$ in the last equation:

$$q_*(s,a) = \mathbb{E}\Big[R_{t+1} + \gamma v_*(S_{t+1}) | S_t = s, A_t = a\Big]. \tag{16}$$

3.1 Bellman Optimality Equations

There are two Bellman equations that are crucial in the context of optimality: one for the value function and one for the action-value function. Let's start with the former. For the optimal value function it obviously holds that:

$$v_*(s) = \max_a q_*(s,a) \tag{17}$$

but in light of (16) we can write:

$$v_*(s) = \max_a \mathbb{E}\Big[R_{t+1} + \gamma v_*(S_{t+1}) | S_t = s, A_t = a\Big] \tag{18}$$

and now, by definition of expectation value (with a conditional probability):

$$v_*(s) = \max_a \sum_{s',r} p(s',r|s,a)\big[r + \gamma v_*(s')\big] \tag{19}$$

which is the *Bellman optimality equation for the value function*.
This equation is crucial, since it allows us to compute the *optimal*s value function of a state by knowing the optimal value function of its successor states.
At last, to obtain the other Bellman optimality equation, we can start from (16):

$$q_*(s,a) = \mathbb{E}\Big[R_{t+1} + \gamma v_*(S_{t+1}) | S_t = s, A_t = a\Big]$$

but we can remember (17), and write:

$$q_*(s,a) = \mathbb{E}\Big[R_{t+1} + \gamma \max_{a'} q_*(S_{t+1}, a') | S_t = s, A_t = a\Big] \tag{20}$$

and so, by definition of expectation value:

$$q_*(s,a) = \sum_{s',r} p(s',r|s,a)\big[r + \gamma \max_{a'} q_*(s',a')\big] \tag{21}$$

and this is indeed the *Bellman optimality eq. for the action-value function*.
This last one in particular will be crucial for what follows.

4 Finding the optimal policy

4.1 Dynamic Programming

In a sense the problem of finding optimal policies for a given Markov Decision Process is completely solved. A collection of algorithms, called *Dynamic Programming* (DP), has been developed to solve exactly this problem. This can be rephrased as saying that if we have a perfect model of the environment, then tecnically we can surely find the optimal policy by using DP.
The cornerstone of DP is the *policy evaluation algorithm*, that allows us to compute the value function $v_\pi(s)$ for a given policy π. This algorithm is usually presented in its iterative form, called *iterative policy evaluation*: we start from an initial guess of the value function (with the obvious constraint that the terminal states must have $v_\pi = 0)^2$, and we iteratively update it until convergence to the true value function, using the Bellman equation. Specifically given the value function at iteration k, we can update it to the value function at iteration $k+1$ by:

$$v_{k+1}(s) = \sum_a \pi(a|s) \sum_{s',r} p(s',r|s,a) \Big[r + \gamma v_k(s') \Big] \tag{22}$$

in fact it can be proven that this algorithm converges to the true value function $v_\pi(s)$ as $k \to \infty$ (we are not interested in this proof).
Once in possession of the value function of a policy, we can use it to improve the policy itself.
To understand how to do this we have to remember that the value function is always calculated with respect to a policy, so we shall start with an arbitrary policy π, and calculate its value function $v_\pi(s)$ by iterating (22) until convergence. Having $v_\pi(s)$ we can now compute $q_\pi(s,a)$:

$$q_\pi(s,a) = \sum_{s',r} p(s',r|s,a) \Big[r + \gamma v_\pi(s') \Big], \tag{23}$$

and now improving the policy is as simple as choosing the action that maximizes $q_\pi(s,a)$ for each state s with probability 1. In other, more general,[3] words: we improve the policy by assigning probability 0 to all the suboptimal actions.
The fact that the resulting policy π' is better or equal to π can be rigorously proven (*policy improvement theorem* [1]), but it is intuitive: we are simply choosing the

[2]If this appears strange to you remember that the final states' rewards have nothing to do with the value of v_π for the final states, since G_t is influenced by R_{t+1}, and not R_t.

[3]This last way of phrasing the *policy improvement algorithm* is more general because it also takes into account the situations in which there are more than one optimal actions.

action that maximizes the expected cumulative reward, so it is natural that the new policy is better or equal to the old one.
The crucial point is that this procedure can be iterated! Moreover it can be proven than each iteration strictly improves the policy, until the optimal policy is reached. This means that this algorithm is guaranteed to converge to the optimal policy as the number of iterations goes to infinity (again, we are not interested in formally proving this facts ourselves). This way of finding the optimal policy is called *policy iteration*.
Of course this algorithm has huge drawbacks: the main one being its gargantuan computational cost. This problem emerges mainly from the need of sweeping the entire state space of the MDP at each iteration. You can train an agent to play tic-tac-toe with this, but surely not chess. One way to mitigate this problem is to use *Asynchronous Dynamic Programming*, where the values of the states are updated, indeed, in a non-synchronous way.

- This allows us to make progress in the optimization of the policy without having to wait for an entire sweep of the state space.
- Also this paradigm allows to intermix optimization and real-time interaction with the environment (exploration of the MDP).

We can create policy iteration algorithms that conform to the Asynchronous Dynamic Programming paradigm, and these are usually called *Generalized Policy Iteration* (GPI) algorithms. Almost all practical reinforcement learning algorithms are based on GPI. The main idea is simply to carry out policy evaluation and policy improvement at the same time: in practice this means alternating between policy evaluation on a subset of the state space and policy improvement (still on a subset). In a sense these two processes are adversarial: the policy evaluation tries to make the value function as accurate as possible, while the policy improvement tries to make the policy as greedy as possible, thus making the current value function inaccurate. This *'competition'* between the two processes allows convergence to the optimal policy.
Again we are not interested in the formal proof of convergence of GPI to the optimal policy, but it is important to note that conceptually the convergence is ensured by the fact that sooner or later the GPI will lead to a policy that *is greedy with respect to its own value function*, and thus satisfy the Bellman optimality equation!
The other huge problem with DP is that it requires a model of the environment,[4] and this is often not available. In the next section we will explore one of the main ways to overcome this problem.

[4]Concretely this means that we need to know the transition probabilities $p(s', r|s, a)$ and all the rewards r.

4.2 Monte Carlo Methods

The *Monte Carlo* (MC) class of methods holds one main advantage over DP: they *do not require a model of the environment*, they only need to interact with it. Mathematically this means that they do not need to know the transition probabilities $p(s', r|s, a)$ and all the rewards r; they only need to know the rewards of the states they visit.

4.2.1 Classic First Visit Monte Carlo

The most popular Monte Carlo RL algorithm is the *First Visit MC*: it simply estimates the state-action value function $q_\pi(s, a)$ by averaging the returns that follow the first visit to the state-action pair (s, a). The main problem with this strategy is that *a priori* only a small fraction of all the possible state-action pairs will be visited, and so the estimates of the value function won't be accurate. This problem can be solved by using *exploring starts methed*, in witch the agent starts each episode from a random state-action pair. We can then use GPI to improve the policy.
At an abstract level we can conceptualize the First Visit MC algorithm as follows:

1. Initialize the policy π and the action-value function q randomly (in practice is sufficient to initialize only the action-value function randomly, since the policy can simply always be the greedy policy with respect to q).

2. Generate an episode using the current policy π.

3. Compute the discounted cumulative reward $G(s, a)$ for each state-action pair (s, a) in the episode (but only for the first visit).

4. Update the value $q_\pi(s, a)$ of the visited state-action pairs to a new value $q'_\pi(s, a)$, given by:

$$q'_\pi(s, a) = \frac{1}{N}\sum_{e=1}^{N} G_i(s, a) \tag{24}$$

   where the index e runs over all the episodes done so far (we are performing a mean over the episodes).

   It is customary in the RL literature to find equations like (24) written as:

$$q_\pi(s, a) \;\leftarrow\; \frac{1}{N}\sum_{e=1}^{N} G_i(s, a) \tag{25}$$

where the symbol $\leftarrow$ is used to indicate that the value of $q_\pi(s, a)$ is updated with the mean of the returns.

5. Update the policy π to be greedy with respect to the new action-value function (if the policy is already greedy with respect to the q tensor given then in practice there will be nothing to do).

Concretely here is a somewhat decent implementation of the First Visit MC algorithm in Python pseudo-code.

```python
from collections import defaultdict
policy.random_initialization()
q.random_initialization()
# This dictionary of list will allow us to perform the means needed
# for the MC algorithm
# Specifically we have one list for each state-action pair
# Notice that this mean will be performed on different episodes
# and so on different policies, because we also update the policy
# at each episode
returns = defaultdict(list)
for episode in range(n_episodes):
    # This is the exploring starts method
    s_0, t_0 = chose_random_state_action_pair()
    # Generate the episode with the current policy
    chain_of_the_episode = generate_episode(policy, s_0, t_0)
    lenght_of_the_episode = len(chain_of_the_episode)
    # Compute the returns for each state-action pair in the episode
    # We start from the end of the episode since this way we can
    # compute G in one pass
    G = 0
    for t in reversed(range(lenght_of_the_episode)):
        s, a, r = chain_of_the_episode[t]
        G = gamma * G + r
        # We only consider the first visit to the state-action pair
        if not (s, a) in chain_of_the_episode[:t]:
            returns[(s,a)].append(G)
            # Update the value function with the mean
            q[(s,a)] = mean(returns[(s,a)])
            # Update the policy
            policy[s] = argmax_on_actions(q, s)
```

This MC algorithm is an example of an *on-policy* method, since it improves the policy used to generate the episodes. One possible drawback is the fact that sometimes is not possible to initialize the episode in each possible state-action pair, and so the exploring starts method is not always feasible. This problem can be solved, for example, by using *off-policy* methods, where the policy used to generate the episodes is different from the policy that is being improved. A simple class of off-policy methods are the *ε-greedy* algorithms, where the agent chooses the action that maximizes the value function with probability $1-\varepsilon$, and a random action with probability ε. This ensures exploration during the learning process. Once deployment is needed, the agent can simply use the greedy policy, putting in light the off-policy nature of the paradigm.
Another possible drawback is that MC methods need to wait until the end of the episode to update the value function: if episodes are long, or *in game* learning is needed, this can be problematic.

4.2.2 Constant-α Monte Carlo

In the following it also will be useful knowing about an alternative version of the (first visit) Monte Carlo method: the *constant-α Monte Carlo*. This method is essentially the same as the first visit MC, but we substitute the mean of (25) with an *exponential moving average*. Another way of saying this is that we ditch the mean calculation in favour of a new idea: giving the state-action evaluation a sort of *inertia*. In any case this is done by updating the value function with the following formula:

$$q_\pi(s,a) \;\leftarrow\; (1-\alpha)q_\pi(s,a) + \alpha G_i(s,a) \tag{26}$$

We see that the new estimate of the state-action value function (tensor) is a mixture between the old estimate and the new candidate one. The parameter $\alpha \in [0,1]$ is called the *learning rate*, and is crucial not to mistake it with the learning rate found in gradient descent algorithms. This method is particularly useful when the value function is noisy, maybe due to an intrinsic stochasticity in the environment, and the learning process needs to smooth out the noise.
Notice that with this new method we still need to wait until the end of the episode to update the action-value function.

4.3 Temporal-Difference Learning

The last class of methods we will explore is also the more popular: *Temporal-Difference* (TD) methods. These methods do not require a model of the environment, but also don't need to wait until the end of the episode to update the value function. The trick to achieving this is essentially to cut the Monte Carlo

evaluation short, and use the current estimate of the value function to guess the rewards far into the future. This practice, estimating future rewards with v_π to update v_π itself, is usually called *bootstrapping*.
We shall focus on the simplest and most popular TD method: the TD(0) method. Here the *0* indicates that we are only truly looking one step into the future, and guessing everything else with v_π.
In the following subsections we will explore two of the most popular TD(0) algorithms: *Sarsa* and *Q-learning*. But before diving deep into the inner workings of these algorithms a little spoiler is in order: both of them will employ the constant-α paradigm, and the main difference between the two will be that Sarsa is an *on-policy* method, while Q-learning is an *off-policy* method.

4.3.1 Sarsa

The Sarsa algorithm is an *on-policy* TD(0) method. It is the most intuitive way to implement TD(0) as described in the previous paragraphs. Concretely the update rule for the action-value function is the following:

$$q_\pi(s,a) \;\leftarrow\; (1-\alpha)q_\pi(s,a) + \alpha\Big[r + \gamma q_\pi(s',a')\Big] \tag{27}$$

notice, crucially, that r is the reward recived *after* taking action a in state s; in other words it's the reward associated with the state s' reached after taking the action a. Also (as always) $q_\pi(s,a)$ estimates the discounted cumulative reward starting from the reward *after* taking action a in state s.
It's also important to note why Sarsa is an *on-policy* method: the action a' in (27) is chosen with the start policy π, the same that is being evaluated and improved. In other words, considering a fixed greedy policy with respect to the given tensor q, the action a' is the argmax of $q_\pi(s',a)$, before the improvement.

4.3.2 Q-learning

Q-learning is by far the <u>most popular</u> TD(0) method. It is an *off-policy* method, and it is based on the following update rule for the action-value function:

$$q_\pi(s,a) \;\leftarrow\; (1-\alpha)q_\pi(s,a) + \alpha\Big[r + \gamma \max_{a'} q_\pi(s',a')\Big] \tag{28}$$

The difference between this method and Sarsa is subtle: in sarsa we select a' on the basis of the current policy, that may not be greedy, while in Q-learning we select a' by taking the argmax of the action-value function; in a sense Q-learning simply *forces* all policies to be greedy during the evaluation of future rewards (for this reason it is *off-policy*).

Note that in practice this method is often also coupled with the ε-greedy policy paradigm, to ensure exploration during the learning process.
At last we cannot avoid mentioning another lens with which to look at Q-learning: it can be seen as a method that tries to learn the optimal action-value function q_*, by iterating an update rule based on the Bellman optimality equation for the action-value function. This interpretation is particularly useful when we want to understand the convergence properties of the algorithm, however we will not dive into this topic any further.

5 What about boundless state spaces?

All the methods explored so far seem to break down when the state space is infinite or non discretizable. However it turns out we can extend our methods to the continuum case with just one main idea: *function approximation*. In particular we will be interested in extending the Q-learning paradigm to the continuum case, using *Artificial Neural Networks* (ANN) as function approximators. This paradigm is often called *Deep Q-learning*.
The continuum case and the discrete case are not so different after all: the former is just the limit case of the latter (see Figure 3).

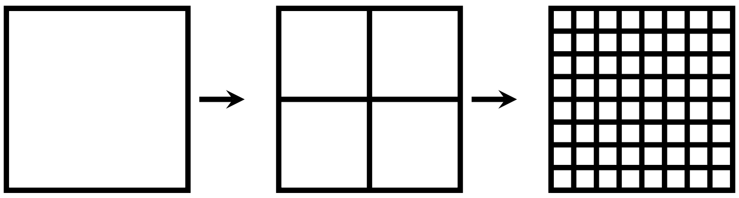

Figure 3: The continuum case is just the limit case of the discrete case. We can just cut the space finer and finer to approximate the continuum more and more. *On the other hand* we can render a function more and more constant in definite regions to approximate the discrete case better and better.

The idea then is to approximate what the tensor $q(s,a)$ was doing in the discrete case with a function $q(s,a;\theta)$, where θ are the parameters of the function. In particular is common to use ANN in place of $q(s,a;\theta)$, since they are universal function approximators. It is customary to employ a neural network with a number of output neurons equal to the number of possible actions [3]; the network is effectively trying to estimate all the available actions' qualities at once (we are still bound by the assumption of finite action space).
The glaring problem now is how to adapt the update rule of Q-learning to this

continuous case. To solve this last hurdle multiple tricks are employed, the main one being the *experience replay* technique. Assuming that the reader is familiar with ANN's training procedures the best course of action to explain this bag of tricks is simply to show the pseudo-code of a Deep Q-learning algorithm [2]:

```
# We don't need to initialize the policy (is always greedy)
# Initialize the ANN with random weights
neural_network.random_initialization()
# Initialize the memory of the agent
memory = []
for episode in range(n_episodes):
    s = initial_state()
    for t in range(max_episode_length):
        q_values = neural_network(s)
        # Choose the action with the epsilon-greedy policy
        if random() < epsilon:
            a = random_action()
        else:
            a = argmax_on_actions(q_values)
        # Perform the action and observe the reward and the new state
        s_prime, r = take_action(s, a)
        # Store the experience in the memory
        memory.append((s, a, r, s_prime))
        # Advance the state
        s = s_prime
        # NOW COMES THE LEARNING PART
        # Sample batch of memory (if still too small sample what available)
        batch = sample(memory, batch_size)
        # Initialize the supervised learning batch
        supervised_learning_batch = []
        for old_s, old_a, old_r, old_s_prime in batch:
            # Compute the single target modification to the q_vector
            # suggested by the memory
            if old_s_prime is terminal:
                single_target = old_r
            else:
                single_target = old_r + gamma *
                ↪  max(neural_network(old_s_prime))
            # Compute the prediction
            y = neural_network(old_s)
            # Generate the target q vector by swapping only one element
```

```
                y[old_a] = (1-alpha) * y[old_a] + alpha * single_target
                # Add one element to the supervised learning batch
                supervised_learning_batch.append((old_s, y))
            # Train on batch
            neural_network.train(supervised_learning_batch)
```

By now it should be clear what the name of the game is: we are essentially operating an almost real-time conversion between a reinforcement learning paradigm and a supervised learning one. Essentially we are leveraging the memories of the agent to build a regression minibatch on the fly, and then train the ANN with it. The target values for the regression are calculated using the Q-learning update rule.
The main difference between this and the discrete paradigm is that we are not substituting directly the quality values with the new ones, but instead we are penalizing the network for not conforming to the new values, incetivizing it to learn them.
At last we go down into the details, to ensure that the previous pseudocode is well understood. We output a vector of Q-values, so the regression problem is from $\mathbb{R}^n$ to $\mathbb{R}^m$, where n is the number of dimensions of the state space and m is the number of possible actions. Since a single memory gives us only a single update to the action-value function, and not an entirely new vector of Q-values, we have to be careful to update only the action that was taken in the memory. This is done by swapping only one element of the old Q-vector, and then training the network with a loss that is a measure of distance between the old Q-vector and the new one (for example the mean squared error loss).

References

[1] Richard S. Sutton and Andrew G. Barto, *Reinforcement Learning: An Introduction*, 2nd Edition, The MIT Press, 2018.

[2] Volodymyr Mnih et al., *Playing Atari with Deep Reinforcement Learning*, 2013.

[3] Adam Paszke & Mark Towers, *Reinforcement Learning (DQN) Tutorial*, https://pytorch.org/tutorials/intermediate/reinforcement_q_learning.html