Multi-agent Reinforcement Learning

Miguel Suau de Castro

Kongens Lyngby 2018

Technical University of Denmark
Department of Applied Mathematics and Computer Science
Richard Petersens Plads, building 324,
2800 Kongens Lyngby, Denmark
Phone +45 4525 3031
compute@compute.dtu.dk
www.compute.dtu.dk

Summary

The goal of the thesis is to ...

Preface

This thesis was prepared at DTU Compute in fulfilment of the requirements for acquiring an M.Sc. in Engineering.

The thesis deals with ...

The thesis consists of ...

Lyngby, 22-July-2018

Miguel Suau de Castro

Acknowledgements

I would like to thank my....

Contents

Chapter 1

Reinforcement Learning

In [1] contrast to other Machine Learning areas such as supervised learning, where a static information buffer is used to build a function that can map input and output, or unsupervised learning, where the main goal is to find correlations and patterns in the data cloud itself, in reinforcement learning, no prior information is provided. The data consists of experiences that are continuously collected from the agent-environment interaction. The nature of this process produces a time series of events where samples are certainly not independent nor identically distributed. Although this sequential dataset can be used to find an optimal solution, the necessity of gathering more information that can better describe the environment makes it difficult to make the common split between training and testing and raises probably the greatest challenge in reinforcement learning, known as exploration-exploitation dilemma.

In this first chapter we will introduce the reinforcement learning framework, describe some of the most popular techniques used to tackle the control problem and show some examples that aim to illustrate the advantages but also the limitations of these methods.

[1] This initial chapter is based on [SB98]

1.1 Markov Decision Process

In order to address the reinforcement learning problem from a mathematical perspective the agent-environment interaction is represented as a Markov decision process MDP. In every time step $t = 0, 1, 2, ..., T$, the agent, using the information provided by the environment about the current state S_t, selects an action A_t that will lead it with certain probability to the next state S_{t+1}, and obtains a reward R_{t+1} as outcome of the chosen action. Eventually, if the process is finite, the agent will reach the terminal state and the sequence will conclude. In a finite MDP, the time series of events from the initial state S_0 to the terminal state S_T is known as episode and must always occur as follows:

$$S_0, A_0, R_1, S_1, A_1, R_2, S_2, A_2..., R_T, S_T \tag{1.1}$$

Because the problem is formulated as an MDP each state and reward in an episode is only dependent on the previous state and action pair. Therefore, in a non-deterministic environment the joint probability of a obtaining a specific reward and ending in a particular state can be written as.

$$p(s', r|s, a) = Pr\{S_{t+1} = s', R_{t+1} = r|S_t = s, A_t = a\} \tag{1.2}$$

Where s, s' and a are certain states and actions that lie within the complete set of possible states S and actions A respectively. Besides, since initial state and terminal state in the previous episode are completely uncorrelated, the first one follows its own independent distribution.

1.2 Partial rewards and expected return

A reward is a a feedback signal that the agent receives at every time step as an indicator of how beneficial is to take a particular action. Negative rewards are meant to penalize a decision, while positive ones encourage a particular behavior. Although, these rewards can be given by the environment, e.g. score in a video game, in certain applications they need to be carefully selected.

The overall objective in reinforcement learning is to maximize the expected return, that is, the sum of expected rewards in a episode. Therefore partial rewards must be perfectly aligned with the final goal of the agent. A more general

formulation which also considers the decaying importance of future rewards at the current time step is given by:

$$G_t = R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + ... = \sum_{k=t+1}^{T} \gamma^{k-t-1} R_k \tag{1.3}$$

The expression above is known as expected discounted return and the parameter $\gamma \in [0, 1]$ is the discount rate. Thus, for γ close to 1 more weight is given to future rewards, while when γ is small or 0 only proximal or immediate rewards are considered. Note that equation 1.3 is also suitable for continuing tasks, that is, when the process cannot be split into episodes, in that case $T = \infty$ and $\gamma < 1$ for the expected return to be bounded.

1.3 Value functions and Bellman equation

A common strategy to solve the reinforcement learning problem is to use the so-called value function, which maps every possible state in the environment with its expected return when following a certain behavior. This behavior, which is known in the literature as policy and is denoted by π, represents the agent's probability of selecting a particular action when in a certain state, $\pi(a|s)$. Using this concept we can write the state-value function under policy π as:

$$v_\pi(s) = \mathbb{E}_\pi \left[G_t \mid S_t = s \right] = \mathbb{E}_\pi \left[\sum_{k=t+1}^{T} \gamma^{k-t-1} R_k \;\middle|\; S_t = s \right] \tag{1.4}$$

The action-value function, on the other hand, not only evaluates the value of being in a state but also the effect of choosing a particular action and following a given policy π from that point onwards.

$$q_\pi(s, a) = \mathbb{E}_\pi \left[G_t \mid S_t = s, A_t = a \right] = \mathbb{E}_\pi \left[\sum_{k=t+1}^{T} \gamma^{k-t-1} R_k \;\middle|\; S_t = s, A_t = a \right] \tag{1.5}$$

Equation 1.4 can be expressed recursively in terms of subsequent states by

exploiting some of the properties of MDPs.

$$\begin{aligned} v_\pi(s) &= \mathbb{E}_\pi \left[G_t \mid S_t = s \right] \\ &= \mathbb{E}_\pi \left[R_{t+1} + \gamma G_{t+1} \mid S_t = s \right] \\ &= \sum_a \pi(a|s) \sum_{s'} \sum_r p(s', r|s, a) \left[r + \gamma \mathbb{E}_\pi \left[G_{t+1} | S_{t+1} = s' \right] \right] \\ &= \sum_a \pi(a|s) \sum_{s'} \sum_r p(s', r|s, a) \left[r + \gamma v_\pi(s') \right] \end{aligned} \tag{1.6}$$

As mentioned before, the goal of the agent is to maximize the expected return over a sequence of steps. Assuming that the true action-value function is known for every state an action, the problem could be easily solved by consistently selecting the action that maximizes the expected return in every state. In that case we say the agent is following the optimal policy. Consequently, the return of any other policy must be always smaller or equal to that of the optimal policy.

$$v_*(s) = \max_\pi v_\pi(s) \tag{1.7}$$

and the optimal action-value function

$$q_*(s, a) = \max_\pi q_\pi(s, a) \tag{1.8}$$

Therefore, considering that our policy is being selected so that the return is maximized the following must always hold

$$\begin{aligned} v_*(s) &= \max_a q_*(s, a) \\ &= \max_a \mathbb{E}_* \left[G_t | S_t = s, A_t = a \right] \\ &= \max_a \mathbb{E}_* \left[R_{t+1} + \gamma G_{t+1} | S_t = s, A_t = a \right] \\ &= \max_a \mathbb{E} \left[R_{t+1} + \gamma v_*(S_{t+1}) | S_t = s, A_t = a \right] \\ &= \max_a \sum_{s', r} p(s', r|a, s) \left[r + \gamma v_*(s') \right] \end{aligned} \tag{1.9}$$

which is known as Bellman optimality equation for the state-value function. Analogously we can write the optimality equation for the action-value function as.

$$\begin{aligned} q_*(s,a) &= \mathbb{E}\left[R_{t+1} + \gamma \max_{a'} q_*(S_{t+1}, a') \middle| S_t = s, A_t = a\right] \\ &= \sum_{s',r} p(s', r|a, s)\left[r + \gamma \max_{a'} q_*(s', a')\right] \end{aligned} \tag{1.10}$$

The exact solution of the above expressions can be found if the state space is finite and the dynamics of the environment known. Thereupon, the best policy can be found by acting greedily in every every state from the beginning until the end of the episode. Note that the greedy action in every state is not necessarily the one that maximizes the immediate reward but the final return, which is given at every state by the Bellman optimality equation.

There exist a bunch of efficient dynamic programming algorithms that can be used to find the optimal policy of a finite state-space environment where the dynamics are known. Although all these methods ensure convergence they are computationally very demanding and only work well in practice when the total number of possible states is small. A more detailed description of these methods can be found in (reference to Sutton's book)

1.4 Unity Machine Learning Agents

1.4.1 Deterministic grid world

In this section we are going to introduce a sample environment that will be used throughout the entire thesis to explain the theory and evaluate the performance of some reinforcement learning algorithms. The most simple setting consist of the 4×4 grid world shown in figure 1.1.

The blue agent located at the center of the image is an UFO that is traveling around the space collecting gold and avoiding a collision with any asteroids. The UFO must decide in every step between four different actions: moving up, down, left or right. In our first example, the location of the objects, asteroid and gold, is fixed, and thus, the entire set of possible states can be defined only using the position of the UFO. The agent receives a reward of $+1$ if the gold is

Figure 1.1: 4x4 Grid world

collected, a reward of -1 when colliding with the Asteroid and a reward of -0.01 for every step that it takes until the episode is over. An episode finishes when either the agent crashes with the asteroid or the gold is picked up.

We solved this problem using a technique called Q-learning that we describe in section **. However, since the dynamics of the environment are known and the state space is finite, it can be solved using Bellman's equation and a dynamic programming method.

The color map in the left part of figure 1.2 represents action-value function at every state when following the optimal policy shown in the right graph.

1.5 Monte Carlo Methods

As mentioned before, dynamic programming algorithms require complete knowledge of the environment and the probability distributions of its transitions. The methods described in this section try to approximate the exact solution using simulations. These follow an empiric strategy based on experiences gathered by the agent when interacting with the environment.

The most simple Monte Carlo algorithm (MC) for evaluating a given policy consists of calculating the sample mean of the return at every state. In this case the value function is only updated at the end of the entire sequence for all the states visited in the episode. Depending on whether only the first visit or all

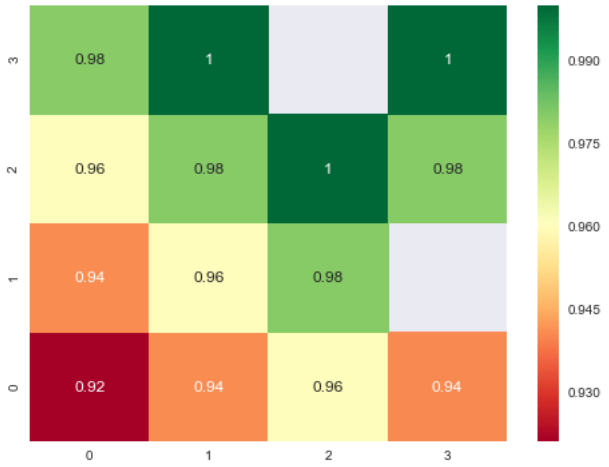


Figure 1.2: State-value function of deterministic grid world when following an optimal policy

the visits to the same state during an episode are used to calculate the average return, two algorithms with different theoretical properties are obtained. It can be proved that both methods converge to the optimal solution as the number of visits to all the states goes to infinity. The pseudo-code below corresponds to first-visit MC.

```
Define: π policy to evaluate
Initialize: V, s
Allocate memory: states, rewards, returns
repeat
    while not done do
        action ← choose an action following π
        state, reward, done ← take step
        states, rewards ← append state reward
    end
    for each different state in states do
        G ← sum rewards from first instance of state to end of episode
        returns(s) ← append G
        V(s) ← average returns(s)
    end
end
```

Algorithm 1: First visit MC method for policy evaluation

Algorithm 1 can be easily modified to calculate q_π instead of v_π. However, these

methods can only be used to evaluate a certain policy but not to find the optimal one. One way to do so, would be to update the policy as new information about the value of every state and action comes in. This method is known in dynamic programing as generalized policy iteration (GPI).

The idea is to start with an arbitrary policy, find the value function, select the greedy policy that maximizes the current value function and repeat the process again as shown in the diagram below ?. A policy can be updated after a few iterations of policy evaluation even if the exact value function for the previous policy has not been found yet. This strategy is meant to avoid the unnecessary computations of evaluating a sub-optimal policy. In practice a policy can be updated right after every episode, in this case it is common to use experiences coming from different policies to calculate the average return of every state. The main drawback of this method is that even if the process went on forever we could not assure that all states and actions would be explored infinitely many times during the process to be able to guarantee convergence to the optimal policy.

1.5.1 On-policy MC methods

On policy methods attempt to guarantee that all regions in the state-action space are sufficiently visited by initially deploying a more relaxed policy where the probability of selecting any of the actions in a given state is greater than 0. One such policy is called ϵ-greedy and the parameter ϵ represents the chance of selecting a random action instead of the greedy one. Hence, combining this strategy with policy evaluation and policy improvement we obtain an algorithm

that can be shown to converge to the optimal solution.

Randomly Initialize: π
Initialize: Q, s
Allocate memory: states, rewards, returns
repeat
  while *not done* **do**
    $a_{max} \leftarrow \max_a Q(s, a)$
    $a \leftarrow$ choose a_{max} or random a with probability ϵ and $1 - \epsilon$
    ns, r, done $\leftarrow$ take step
    states, actions, rewards $\leftarrow$ append s, a, r
    $s \leftarrow ns$
  end
  for *each different* $s - a$ *pair in states-actions* **do**
    G $\leftarrow$ sum rewards from first instance of state-action to end of episode
    returns$(s, a) \leftarrow$ append G
    $Q(s, a) \leftarrow$ average returns(s, a)
  end
end

Algorithm 2: On-policy ϵ-greedy MC method for control

1.5.2 Off-policy MC methods

Although in the previous section we introduced soft-policies to allow some environment exploration instead of acting greedily in every iteration, this had the consequence of making the control not entirely optimal. We will now present a different approach where two policies are used instead. The first one, known as behavior policy, is meant to explore and add new experiences, while the other, the target policy, uses the information provided by the first one to make improvements.

A key concept in off-policy algorithms is importance sampling. This is used to correct for the fact that the data generated to update our target policy comes from a different distribution. The probability of a sequence of events occurring when following a specific policy π can be calculated as:

$$P(A_t, S_{t+1}, A_{t+1}, ..., S_T | S_t, A_{t:T-1}\ \pi) = \prod_{k=t}^{T-1} \pi(A_k|S_k) p(S_{k+1}|S_k, A_k) \quad (1.11)$$

and the importance sampling ratio is then given by:

$$\rho_{t:T-1} = \frac{\prod_{k=t}^{T-1} \pi(A_k|S_k)p(S_{k+1}|S_k, A_k)}{\prod_{k=t}^{T-1} b(A_k|S_k)p(S_{k+1}|S_k, A_k)} = \prod_{k=t}^{T-1} \frac{\pi(A_k|S_k)}{b(A_k|S_k)} \tag{1.12}$$

Algorithm 1 can be then adapted to evaluate a target policy using a behavioral policy to create new experiences.

Define: π target policy
Define: b behavioral policy
Initialize: Q, s, $G = 0$, $W = 1$
Allocate memory: states, rewards, returns
repeat
  while *not done* **do**
    action $\leftarrow$ choose an action following b
    ns, r, done $\leftarrow$ take step
    states, actions, rewards $\leftarrow$ append s, a, r
    $s \leftarrow ns$
  end
  for *each state-action pair, from terminal to initial state* **do**
    $G \leftarrow \gamma G + R_{t+1}$
    $C(S_t, A_t) \leftarrow C(S_t, A_t) + W$
    $Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \frac{W}{C(S_t, A_t)}[G - Q(S_t, A_t)]$
    $W \leftarrow W \frac{\pi(A_t|S_t)}{b(A_t|S_t)}$
    if $W = 0$ **then**
      break
    end
  end
end

Algorithm 3: Off-policy MC method for policy evaluation

Where W CONTINUE ...

1.6 Temporal difference learning

Even though, as discussed in the previous section, Monte Carlo methods are proved to converge to the optimal solution they make an inadequate use of the information and can perform poorly in practice. The fact that we need to wait until the end of an episode to make a change in our value function can have a

strong negative impact specially in situations where episodes are long or even infinite.

While MC methods use the true sample return obtained at the end of an episode to update the value estimate of all the states and actions visited in that episode, temporal difference learning methods, TD, base the updates on previous estimations of the value function at subsequent states.

$$V(S_t) \leftarrow V(S_t) + \alpha \left[R_{t+1} + \gamma V(S_{t+1}) - V(S_t)\right] \tag{1.13}$$

where $R_{t+1} + \gamma V(S_{t+1})$, instead of G_t, is used as the approximation of the expected return at S_t. Hence, the TD learning alternative to algorithm 1 used for policy evaluation

Define: π policy to evaluate
Initialize: V
repeat
  Initialize: s
  while *not done* **do**
    action $\leftarrow$ choose an action following π
    ns, r, done $\leftarrow$ take step
    $V(s) \leftarrow V(s) + \alpha \left[r + \gamma V(ns) - V(s)\right]$
    $s \leftarrow ns$
  end
end

Algorithm 4: TD learning for policy evaluation

The technique applied in algorithm 4 is known as bootstrapping and it allows to perform updates more frequently than MC methods while still converging to the optimal solution. In practice, the efficient use of data and memory in TD learning methods, normally leads to faster learning. However, there is no mathematical prove supporting this empirical fact. TALK ABOUT ELIGIBILITY TRACES

1.6.1 On-policy TD methods: Sarsa

The on-policy version of TD control algorithms is known as Sarsa. As in MC we also apply Generalized Policy Iteration to evaluate and improve the action choices but in this case the value function update is done online and not at the end of every episode.

Again, the goal is to estimate the action value function $Q(s, a)$ at every state-action pair so as to be able to make decisions on what actions to take, by comparing the expected returns in a particular state. Because this is an on-policy method we need to deploy soft policies in order to allow some exploration around the state-action space. Algorithm 5 presents the Sarsa TD method for control with ϵ-greedy policy

Randomly Initialize: π Initialize: Q
repeat
  Initialize: s
  $a_{max} \leftarrow \max_a Q(s, a)$
  $a \leftarrow$ choose a_{max} or random a with probability ϵ and $1 - \epsilon$
  while *not done* **do**
    s', r, done $\leftarrow$ take step
    $a'_{max} \leftarrow \max_{a'} Q(s', a')$
    $a' \leftarrow$ choose a'_{max} or random a' with probability ϵ and $1 - \epsilon$
    $Q(s, a) \leftarrow Q(s, a) + \alpha \left[r + \gamma Q(s', a') - Q(s, a)\right]$
    $s \leftarrow s', \quad a \leftarrow a'$
  end
end

Algorithm 5: On-policy TD method for control: Sarsa

Algorithm 5 is guaranteed to converge to the optimal policy if all the state-action pairs are visited infinite times and the soft policy gradually becomes the greedy policy.

1.6.2 Off-policy TD methods: Q-Learning

Q-learning appears as a simple but interesting off-policy variant of Sarsa. In this method the value function is updated after a single step using the immediate reward and the current estimate of the greedy action at the next state.

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha \left[R_{t+1} + \gamma \max_a Q(s_{t+1}, a) - Q(S_t, A_t)\right] \qquad (1.14)$$

Randomly Initialize: π Initialize: Q
repeat
  Initialize: s
  while *not done* **do**
    $a_{max} \leftarrow \max_a Q(s, a)$
    $a \leftarrow$ choose a_{max} or random a with probability ϵ and $1 - \epsilon$
    s', r, done $\leftarrow$ take step
    $Q(s, a) \leftarrow Q(s, a) + \alpha \left[r + \gamma \max_{a'} Q(s', a') - Q(s, a) \right]$
    $s \leftarrow s', \quad a \leftarrow a'$
  end
end

Algorithm 6: Off-policy TD method for control: Qlearning

Q-learning is considered an off-policy method because the policy being followed is different from the greedy policy used to update the action value function. Yet, algorithm 6 varies from the off-policy methods described in SECTION MC in the sense that the target and behavioral policies are the same and the update is performed using the next state's greedy action. Therefore, the policy must be soft to ensure some exploration.

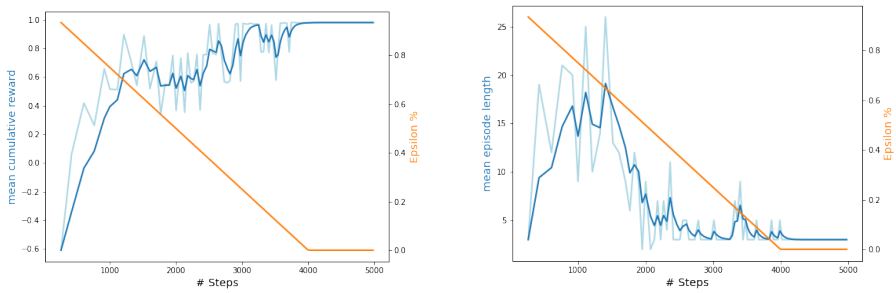
The method works very well in practice and has been proved to converge if all state-action pairs are visited enough times. Another variant, called Expected Sarsa, uses the expectation of the action value function at the next state to make the update.

1.6.3 Static Grid World with tabular methods

Algorithm 6 is implemented in Python and tested on our grid world example. The action-value function $Q(s, a)$ consists of a lookup table where every row represents a state and every column an action. Since the objects always remain in the same position, states are completely determined by the position of the UFO. Consequently, the size of the lookup table is 16×4 because there are 16 squares on the grid and 4 possible actions.

Because we assume no prior knowledge about the position of the objects all state-action pairs are initialized to 0. To encourage the UFO to explore the environment we plan a decay for ϵ from 1, entirely random policy, to 0, greedy policy. The results are shown in figure 1.3

Figure (a) shows that, as the agent learns more about the value function and the chance of random action decays, the UFO begins to consistently collect the gold and avoids colliding with the asteroid. We can see that after 4000 steps the



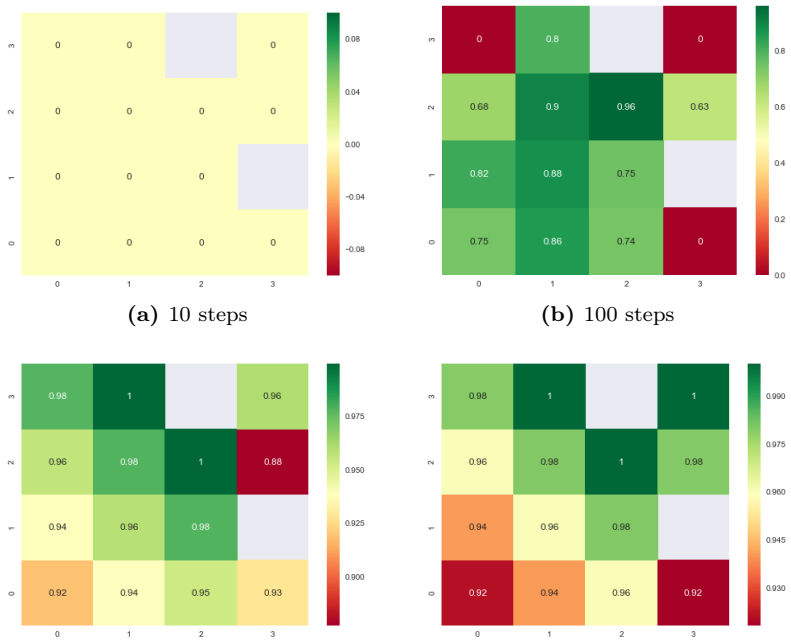
(a) **(b)**

Figure 1.3: Q learning with lookup table. Figure a: The left vertical axis shows the sum of rewards per episode averaged over 10 episodes. Figure b: The left vertical axis shows the total number of steps per episode averaged over 10 episodes. The right vertical axis in both plots corresponds to the value of ϵ at every step. We have applied exponential smoothing to the two blue curves in order to attenuate the peaks. The light blue curves are the true values.

agent starts to act always greedily. This has an obvious effect on the returns which from that point onwards lie always close to 1. The mean episode length depicted in figure 1.3 plot (b) shows that the episodes are short at first. The UFO is moving randomly which means that it can easily either collide with an asteroid or unintentionally collect the gold nugget. Around 1000 steps the agent has learned to avoid the asteroids and thus, the episodes are longer. From step 1500 onwards the agent tries to find the optimal path so that it can solve the task in the fewest possible number of moves.

Figure 1.4 shows the estimated action-value of every position after 10, 100, 500, 1000 steps. The values correspond to the action that maximizes the return at each location.

The color map demonstrates how after very few iterations the agent is able to find an accurate estimate of the action-value function and use it to improve its policy. However, one can argue that the method works very well due to its over-simplistic complexity and that using a table to represent the action-state space is not always possible. In the next section we shall study methods with better scaling properties that use functions instead of tables that allow to map more complex environments.



(a) 10 steps **(b)** 100 steps

(c) 500 steps **(d)** 1000 steps

Figure 1.4: Action-value function estimation using Q-learning with tabular methods. The values shown in the color map correspond to the greedy action at every state.

1.7 Value function approximation

As shown in the deterministic grid world example, tabular methods are very useful for solving simple tasks where the number of states and actions is relatively small. Recall that in the example, there were only 16 different states and 4 actions. Nonetheless, if we think of any real control environment the number of possible states and actions can be tremendously large or even infinite. That is why, using a table to represent states and actions is not always possible, because it does not scale when moving to more complex settings. In this section we will present a solution to this problem that implies a dramatic change in the way we apply and optimize our value function. Yet, the reinforcement learning framework remains unchanged and all the algorithms presented can still be applied.

As opposed to tabular methods, the value function is approximated using a parametric function instead of a matrix where the number of parameters is often smaller than the number of state-action pairs. The function used to represent the environment depends very much on its complexity. The range of possible options go from a simple weighted linear combination of state-action space features, to a nonlinear function obtained by training a neural network.

The idea sounds very much like a supervised learning problem. We can think of past observations as samples in our training set that we use to find a function that can map state observations to action values. However, in this case the training is done online, while the agent is exploring the environment. This raises the problem of samples being strongly correlated. Besides, while updating a particular element in the state-action table had no effect on the rest, changing a single parameter in our function can affect multiple states and actions.

1.7.1 Loss function

Finding a function that can effectively map state-action pairs to action values reduces to obtaining the set parameters that best approximate the action values in all regions. However, since the end goal is to find the optimal policy we might want to be able to approximate some states and actions better than others. Thus we need to decide on a suitable loss function than can help us asses performance. Like in most regression problems the obvious choice is the Mean Squared Error MSE, which can be weighted according to how much importance is given to each

particular state-action pair

$$L(\mathbf{w}) = \sum_{s \in S, a \in A} \mu(s,a)[q_\pi(s,a) - \hat{q}(s,a,\mathbf{w})]^2, \tag{1.15}$$

with $\mu(s,a)$ proportional to the expected number of visits to a particular state and action compared to the rest. We can estimate the time spent at each pair as

$$\eta(s,a) = h(s) + \sum_{s'} \eta(s',a)\pi(a|s')p(s|s',a) \tag{1.16}$$

where h(s) is the probability that s is the initial state, and thus

$$\mu(s) = \frac{\eta(s)}{\sum_s \eta(s)}. \tag{1.17}$$

Hence if we knew the true action values for the optimal policy, the optimal set of parameters could be easily obtained by solving an optimization problem using equation 1.15 as objective function. However, in a real scenario, since there is no prior knowledge of q_π, an approximation must be used instead.

(Add that is not clear that this is the optimal loss function because the end goal is to optimize the policy?)

1.7.2 Stochastic gradient and semi-gradient descent

We have already discussed various ways of approximating the value function. We could, for instance, substitute q_π by the MC estimate G_t in equation 1.15, and then, using stochastic gradient descent (SGD), update the parameters of our value function. Here we assume that $\mu(s,a)$ is the same for all state-action pairs.

$$\begin{aligned} \mathbf{w_{t+1}} &= \mathbf{w_t} - \frac{1}{2}\alpha \nabla \left[G_t - \hat{q}(S_t, A_t, \mathbf{w_t})\right]^2 \\ &= \mathbf{w_t} + \alpha \left[G_t - \hat{q}(S_t, A_t, \mathbf{w_t})\right] \nabla \hat{q}(S_t, A_t, \mathbf{w_t}) \end{aligned} \tag{1.18}$$

where α is the step size and $\nabla \hat{q}(S_t, A_t, \mathbf{w_t})$ represents the gradient of $\hat{q}$ with respect to $\mathbf{w_t}$

Conversely, if we wanted to apply any of the TD methods described in section ** we would have to use what is known in the literature as semi-gradient descent. The reason for this is that now, although our target also depends on the value function, it is not considered when calculating the gradient.

$$U_t = R_{t+1} + \hat{q}(S_{t+1}, A_{t+1}, \mathbf{w_t}) \tag{1.19}$$

$$\mathbf{w_{t+1}} = \mathbf{w_t} + \alpha \left[U_t - \hat{q}(S_t, A_t, \mathbf{w_t})\right] \nabla \hat{q}(S_t, A_t, \mathbf{w_t}) \tag{1.20}$$

Although possibly compromising convergence, the use of equation 1.20 is justified in practice because it can notably speed up the learning process.

1.8 Deep Q-learning

The algorithm presented in this section is in essence the same Q-learning method that helped us solve the deterministic grid world example except that now we use a neural network instead of a table to approximate the value function.

Randomly Initialize: π Initialize: Q
repeat
  Initialize: s
  while *not done* **do**
    $a_{max} \leftarrow \max_a \hat{q}(s, a, \mathbf{w})$
    $a \leftarrow$ choose a_{max} or random a with probability ϵ and $1 - \epsilon$
    $s', r, done \leftarrow$ take step
    $\mathbf{w} \leftarrow \mathbf{w} + \alpha \left[r + \gamma \max_{a'} \hat{q}(s', a', \mathbf{w}) - \hat{q}(s, a, \mathbf{w})\right] \nabla \hat{q}(s, a, \mathbf{w})$
    $s \leftarrow s'$
  end
end

Algorithm 7: Off-policy TD method for control: Qlearning

The pseudo-code shown in algorithm 7 corresponds to the standard Q-learning algorithm with function approximation. A more complex version that incorporates a few additional elements, described below, was implemented in Python using Tensorflow. A link to the GitHub repository that contains the code is provided in the Methods chapter.

1.8.1 Experience Replay

As we mentioned in section **, one important difference between reinforcement learning and supervised learning is that the samples we collect are not independent. Updating the network parameters in every iteration as we obtain new experiences can very much affect the stability of our optimization method.

Experience replay [Lin92] intends to decouple information and break up correlations between successive samples by first storing the information in a big replay memory buffer and then sampling at random mini-batches of experiences in every iteration to effectively update the model. This has the effect of not only reducing variance and preventing instability issues, but also accelerating the learning process because experiences can be used more than once.

1.8.2 Independent target network

We saw that when using the TD estimate as target to update our parameters a full gradient step could not be taken because our target was also dependent on the same parametric function. Although we saved this by using the so called semi gradient (equation 1.20), the fact that the function being updated is also present in our target means that they are both strongly correlated, which can provoke dangerous oscillations and make the algorithm diverge. The solution proposed in [MKS+15] consists of separating the two functions, the main network and the target network, and clipping the parameters of the target network. While the main network is updated in every iteration the weights of the target network are only periodically moved toward the main network after a certain number of iterations.

This two ideas, experience replay and the use of an independent target network, helped DeepMind build a model that achieved, and in some cases excelled, human level performance in a series of Atari video games.

1.8.3 Double Q-learning

Double Q-learning [VHGS16] aims to reduce the positive bias caused by the maximum operator used in the Q-learning update formula (equation 1.14) for both action selection and action evaluation. This effect, also known as overoptimism, has been shown to affect convergence, specially if suboptimal policies are repeatedly being favored.

The original paper [Has10], which is focused on tabular Q-learning, proposes storing two different tables, A and B, that learn the value function independently. The update is then performed using one of the functions to select an action and the other to estimate the value

$$Q^A(S_t, A_t) = Q^A(S_t, A_t) + \alpha \left[R_{t+1} + \gamma Q^B \left(S_{t+1}, \arg\max_a Q^A(S_{t+1}, a) \right) \right] \tag{1.21}$$

and analogously for table B.

In every iteration, by random selection, only one of the two tables is updated so that they learn from different sets of experiences. The underlaying idea is to decouple the estimate from the action selection to remove the positive bias.

The authors successfully applied this same method to DQN but used the target network instead of a secondary function. This resulted in the following equation for calculating the target

$$U_t = R_{t+1} + \gamma \hat{q} \left(S_{t+1}, \arg\max_a \hat{q}(S_{t+1}, a, \mathbf{w}_t), \mathbf{w}'_t \right). \tag{1.22}$$

Although the two functions are now dependent, the use of the target network is justified because it only implies a minimal change in the algorithm. Besides, the fact that target network is only updated every once in a while has been proven enough to reduce bias and increase performance.

1.8.4 Static Grid World with function approximation

We showed that the tabular Q learning method could solve the static grid world example in very few iterations because the number of states-action pairs is small and the use of a parametric function is not necessary. Yet, we would like to test how well our DQN does in the same task. Even though in [MKS+15] they use images as input and a convolutional neural network to encode observations, we will keep it simple for now and use a one-hot encoded vector representing the UFO's position at every time step.

The function approximator consists of a single fully connected layer with four nodes, each one corresponding to one of the four actions. (INCLUDE THIS:

we can do this because the action space is discrete and there are not many actions otherwise we would have to include the actions as input and maximize the function to find the greedy action) The algorithm stores experiences in a replay buffer and samples from it to update the function parameters. To make the optimization process more robust we added a secondary network to calculate the targets. Its weights are updated by a small amount toward the main network in every iteration. This has the same effect as a full update every certain number of steps but it prevents potential instability issues provoked by abrupt changes in the targets. Finally, we also applied double Q-learning, as described in the previous section, to reduce overoptimism.

The results of running DQN on the grid world example are shown in figure 1.5

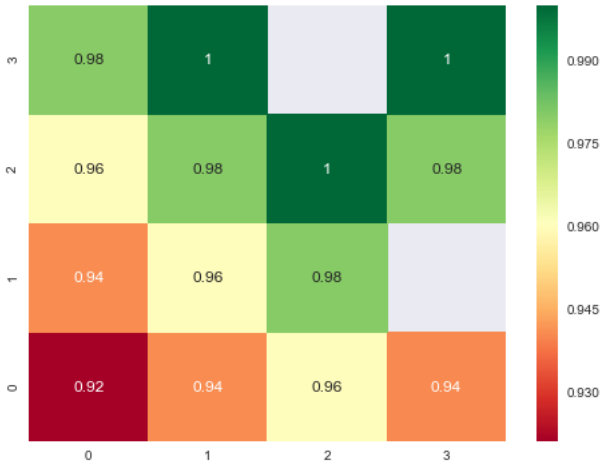


Figure 1.5: Value function of the static grid world example obtained using DQN. The values are equal to those obtained with tabular Q-learning

As we can see, using this binary state space representation, the method is able to match the values to the ones obtained with the tabular version. In fact, as long as the number of parameters is larger than the number of possible states-action pairs the neural network should be able to act like a regular table. In our case, because we use exactly 64 parameters and no bias terms, the weight values are equal to the 64 elements in the lookup table.

The above mentioned should be no surprise considering that our neural network is nothing but a linear regression between the binary vector as input, and the state-action values as output. Figure 1.6 shows the mean squared difference between the weights at the end of every episode and the exact true values.

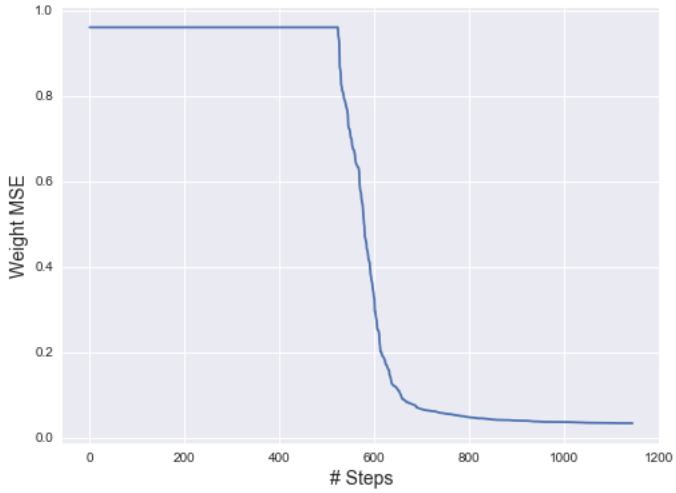


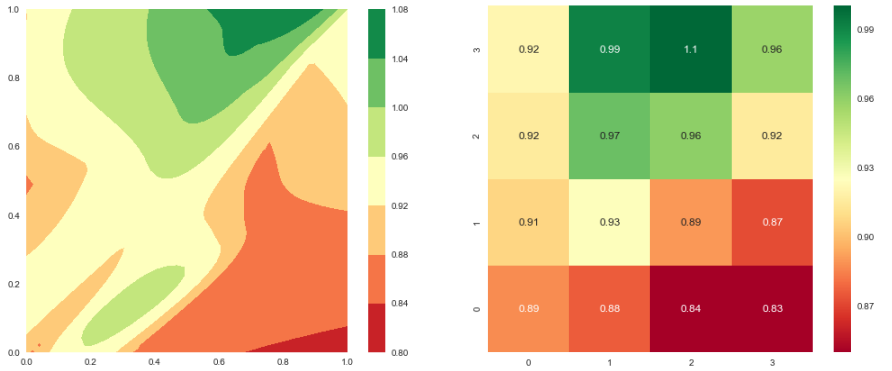
Figure 1.6: Mean Squared Error between weights at the end of every episode and true values. The first 10000 steps correspond to a period of random exploration where we only populate the replay memory but do not update the model.

We could also feed our neural network with a more compact state representation. For this example we used a state space vector of two dimensions that correspond to the coordinates of the agent at each position. Figure 1.7 shows a contour plot of the value function in a continuous state space. Even though our agent is only allowed to occupy certain locations on the grid, the model can calculate the action value at any point.

Now the function is not flexible enough to reproduce the optimal value function and needs to find a compromise solution. Yet, one can see that, although the values are not the true ones, the policy obtained when acting greedily is indeed the optimal policy. This example supports the use of a function approximator in complex reinforcement learning settings. As long as the function is able to describe the important elements in the environment it can very well be used to obtain the optimal policy.

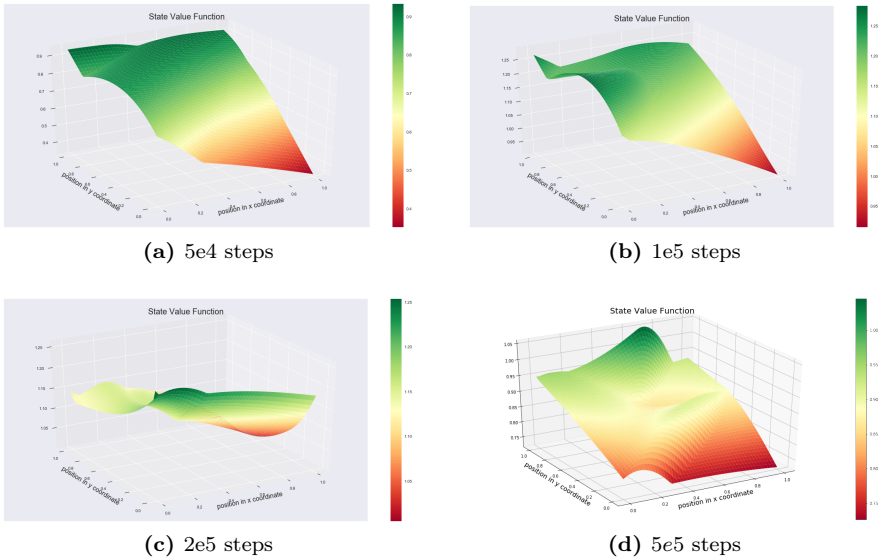
1.9 Policy Gradient Methods

All methods described so far were focused on learning a value function that was later used for selecting actions at every state. Considering that we are essentially



(a) **(b)**

Figure 1.7: State-value function estimation using DQN. The values shown in the contour plot (figure a) correspond to the maximum estimated return of all actions at every possible state in the continuous state space. The values estimated by the neural network when feeding the allowed locations only, are depicted in figure (b).



(a) 5e4 steps **(b)** 1e5 steps

(c) 2e5 steps **(d)** 5e5 steps

Figure 1.8: State-value function estimation using DQN. The values shown in the 3D graph correspond to the maximum estimated return of all the actions at every possible state in the continuous state space.

interested in finding the best strategy that will allow the agent to act optimally, this was an oblique way of solving the control problem.

We therefore present in this section a completely different approach in which we define a parametric function that can translate state observations into action probabilities.

$$\pi(a|s,\theta) = Pr\{A_t = a|S_t = s, \theta_\mathbf{t} = \theta\} \tag{1.23}$$

We saw in the static grid world example that policies tend to be simpler than value functions and thus, easier to learn in practice. While before the value of an action had to increase over the rest in a given state in order to become selected, now any slight change to our parameters modifies the distribution over the actions, therefore affecting directly to our policy. Besides, policies are now non-deterministic, in the sense that unless the probabilities become exactly 0 actions have always some chance of being selected, this can help solve certain tasks where the optimal policy is a stochastic policy (e.g. the game of poker).

1.9.1 Policy objective function and gradient

As we did when we presented the function approximation methods, we need to define a loss function that will help us evaluate the performance of a particular policy.

$$\begin{aligned} J(\theta) &= E_\pi \left[\sum_{t=0}^{T} R_t \right] \\ &= \sum_a \pi_\theta(S_0, a) q_{\pi_\theta}(S_0, a). \end{aligned} \tag{1.24}$$

This is nothing else than the expected return from the initial state S_0 when following policy π_θ from that point onwards. As mentioned earlier, because the policies are stochastic, updating our parametric function will slightly modify the chance of selecting certain actions but it can never cause dramatic changes to our policy as it occurs with value based methods. This means that the objective function in 1.24 is smooth and thus, the method has better convergence properties.

The idea to find the optimal policy is to repeatedly adjust the parameters moving them in the direction that maximizes equation 1.24. A way to do so is by following the gradient, which according to the policy gradient theorem, whose prove is given in [SB98], is calculated as follows

$$\begin{aligned}\nabla_\theta J(\theta) &\propto \sum_s \mu(s) \sum_a \nabla_\theta \pi_\theta(s,a) q_{\pi_\theta}(s,a) \\ &= E_{\pi_\theta}\left[\sum_a \pi_\theta(S_t,a) \frac{\nabla_\theta \pi_\theta(S_t,a)}{\pi_\theta(S_t,a)} q_{\pi_\theta}(S_t,a)\right] \\ &= E_{\pi_\theta}\left[\nabla_\theta \log \pi_\theta(S_t,A_t) q_{\pi_\theta}(S_t,A_t)\right]\end{aligned} \tag{1.25}$$

where $\mu(s)$ is the on-policy distribution over the state space under policy π as defined in equation 1.17.

The term $\nabla_\theta \log \pi_\theta(s,a)$ is often called eligibility vector and it indicates the direction along which a change in our parameters will most increase the chance of selecting a particular action. On the other hand, $q_{\pi_\theta}(s,a)$, gives the expected return of a particular action given a state. Therefore, taking a step in the gradient direction will increase the probability of selecting actions that yield a high positive return while decreasing the probability of the ones whose return is negative.

1.9.2 MC Policy Gradient: REINFORCE

The most simple version of policy gradient methods is called REINFORCE and it uses an MC estimate of the action value to adjust the policy weights by applying stochastic gradient ascent.

$$\theta_{t+1} = \theta_t + \alpha \nabla_\theta \log \pi_\theta(S_t,A_t) G_t \tag{1.26}$$

where $q_{\pi_\theta}(s,a)$ in equation 1.25 has been substituted by the sample return G_t. We first collect the experiences of an entire episode, and then update the parameters at every state using the sum of rewards from the current state until the end of the episode as an unbiased sample of the expected return. Algorithm

8 contains the pseudo-code describing the method.

Initialize: θ
Allocate memory: states, actions, rewards
repeat
  Initialize: s, G = 0
  while *not done* **do**
    a $\leftarrow$ choose an action at random from $\pi_\theta(s, a)$
    s', r, done $\leftarrow$ take step
    states, actions, rewards $\leftarrow$ append s, a, r
    $s \leftarrow s'$
  end
  for *each state-action pair, from terminal to initial state* **do**
    $G \leftarrow \gamma G + r$
    $\theta \leftarrow \theta + \alpha \nabla_\theta \log \pi_\theta(a, s) G$
  end
end

Algorithm 8: MC policy gradient: REINFORCE

1.9.3 TD Policy Gradient: Actor-Critic

As in any MC method, REINFORCE only updates the parameters of the policy once the episode is over which can slow down a lot the learning process. Besides, replacing the expected return with a sample estimate introduces high variance. One-step actor-critic methods use a value function approximator as an estimate of the return after a single step. The approach to obtain the action values is essentially the same as in value based methods with the only difference that now we only need the function to evaluate the actions and not to make decisions. Hence, we can distinguish between the actor π_θ, which defines our policy and therefore is the one influencing the agent's behavior, and the critic q_w, whose only task is to approximate the expected return.

The pseudo-code in algorithm 9 describes step by step how the one-step actor-critic method works. More sophisticated algorithms combine the properties of TD learning and MC methods using eligibility traces to trade off bias and

variance.

Randomly Initialize: π Initialize: Q
repeat
  Initialize: s
  $a \leftarrow$ sample an action at random from $\pi_\theta(s, a)$
  while *not done* **do**
    $s', r, done \leftarrow$ take step using a
    $a' \leftarrow$ sample an action at random from $\pi_\theta(s', a')$
    $\mathbf{w} \leftarrow \mathbf{w} + \alpha \left[r + \gamma \hat{q}_{\mathbf{w}}(s', a') - \hat{q}_{\mathbf{w}}(s, a) \right] \nabla \hat{q}_{\mathbf{w}}(s, a)$
    $\theta \leftarrow \theta + \alpha \nabla_\theta \log \pi_\theta(a, s) \hat{q}_{\mathbf{w}}(s, a)$
    $s \leftarrow s'$, $a \leftarrow a'$
  end
end

Algorithm 9: TD policy gradient: one-step actor-critic

Although using a function approximator introduces some bias, it can also make the algorithm converge faster. Moreover, since the updates are done online the method is suitable for continuing control tasks.

1.9.4 Policy Gradient with Baseline

In order to reduce even more the variance of our estimates we define a baseline function, $B(S)$, and we subtract its value from the policy gradient. This operation must have no effect on the direction of the gradient and should only change its scale.

$$\nabla J(\theta) \propto \sum_s \mu(s) \sum_a \nabla_\theta \pi_\theta(s, a)(q_{\pi_\theta}(s, a) - b(s)) \tag{1.27}$$

and as long as the function does not vary with a

$$\sum_a b(s) \nabla_\theta \pi_\theta(s, a) = b(s) \nabla_\theta \sum_a \pi_\theta(s, a) = b(s) \nabla_\theta 1 = 0 \tag{1.28}$$

Most methods use the state value function $V_\pi(s)$ for the baseline so that the term $(q_{\pi_\theta}(s, a) - b(s))$ in equation 1.27 becomes

$$D_\pi(S_t, A_t) = Q_\pi(S_t, A_t) - V_\pi(S_t). \tag{1.29}$$

$D_\pi(S_t, A_t)$ is known as advantage function and it measures the deviation of an action value from the current state estimate. Although it is common in the reinforcement learning literature to use A_π to denote the advantage function, note that this terminology would cause a name collision between our action variable A_t and the new function.

Whereas before we were updating actions based on their value estimate and, unless the estimated return was negative, we would adjust the parameters more or less strongly along the positive direction, now we penalize actions that are worse than the state value estimate no matter whether the outcome is positive or negative. In other words, the idea of using a baseline is that, in states where all actions have associated a positive return, we would like to discriminate between good and bad actions and increase or decrease their probabilities of being selected accordingly. This pushes our all gradient values to be in a neighborhood of 0 which is preferable to having gradients of very different magnitude depending on which state we are in.

One could then use this approach to reduce the variance of an actor-critic method and consequently, accelerate its learning rate. We could, of course, introduce a new function to approximate V_π but a more reasonable choice would be to use the critic. For the latter, we would modify the value function approximator so that it computes the state-value instead of the action value and then estimate the advantage at each time step using an approximation of the TD error.

$$\delta_{\mathbf{w}} = R_{t+1} + \gamma v_{\mathbf{w}}(S_{t+1}) - v_{\mathbf{w}}(S_t). \tag{1.30}$$

We can show that this is acceptable by substituting v_w with the the true value function V_π and then taking the expected value.

$$\begin{aligned} E_\pi[\delta_\pi | S_t, A_t] &= E[R_{t+1} + \gamma V_\pi(S_{t+1}) - V_\pi(S_t) | S_t, A_t] \\ &= Q_\pi(S_t, A_t) - V_\pi(S_t) \\ &= D_\pi(S_t, A_t). \end{aligned} \tag{1.31}$$

The objective function that we are trying to optimize is then

$$J(\theta) = E[\log \pi_\theta(S_t, A_t) D_\pi(S_t, A_t)] \tag{1.32}$$

Algorithm 10 shows a variant of the Actor-Critic method that introduces a baseline function.

Randomly Initialize: π Initialize: Q
repeat
  Initialize: s
  $a \leftarrow$ sample an action at random from $\pi_\theta(s, a)$
  while *not done* **do**
    $s', r, done \leftarrow$ take step using a
    $a' \leftarrow$ sample an action at random from $\pi_\theta(s', a')$
    $\delta \leftarrow r + \gamma \hat{v}_{\mathbf{w}}(s') - \hat{v}_{\mathbf{w}}(s)$
    $\mathbf{w} \leftarrow \mathbf{w} + \alpha\delta\nabla\hat{q}_{\mathbf{w}}(s, a)$
    $\theta \leftarrow \theta + \alpha\nabla_\theta \log \pi_\theta(a, s)\delta$
    $s \leftarrow s'$, $a \leftarrow a'$
  end
end

Algorithm 10: TD policy gradient: one-step actor-critic with baseline

1.9.5 Proximal Policy Optimization

Adding a critic in place of the MC sample return and a baseline function to scale the size of the objective gradient, certainly helped reducing the variance of the policy gradient methods. But considering that one major complication in reinforcement learning is the lack of a training set, performing a single parameter update in every iteration after collecting one sample does not seem to be the best way to use the limited amount of data that is available.

Because Q learning is an off-policy method when we implemented our Deep Q-learning algorithm, we used a big replay buffer that we would sample from to adjust the weights of our value function. This is not the case of policy gradient methods where the policy being optimized is the same we use to make decisions. Nonetheless, we can overcome this problem with importance sampling. This technique, which we already presented in section **, allows us to optimize our parameters employing experiences obtained under a different policy. Following this idea Proximal Policy Gradient (PPO) [SWD+17] modifies the loss function in equation 1.32 as follows

$$J(\theta) = E\left[\frac{\pi_\theta(S_t, A_t)}{\pi_{\theta_{\text{old}}}(S_t, A_t)} D_\pi(S_t, A_t)\right], \tag{1.33}$$

where $\pi_{\theta_{\text{old}}}(S_t, A_t)$ are the weights of our policy before the update. Substituting

the objective function in algorithm 10 by equation 1.33, and repeatedly update our parameters using mini-batches of our experience buffer will, in most cases, make our policy diverge. As long as the old and new policies are similar there should no problem in using this new loss function, but as they become more and more different the magnitude of the gradient increases and so does the step size of our updates. Taking excessively large steps could modify the policy dramatically leading the agent to regions of the space from where it could be very difficult to recover.

To solve this issue PPO applies a variant of a numerical optimization technique called trust region. This idea was first introduced in [SLA+15] where the authors added an inequality constraint to the optimization problem that prevents old and new policies from diverging, by limiting their relative entropy (KL divergence).(SHOULD I INCLUDE A LINK TO THE KL WIKIPEDIA SITE?) It turns out that this arrangement complicated the algorithm substantially and made it incompatible with certain deep learning architectures. Equation 1.34 is the objective function akin to trust region proposed in [SWD+17].

$$J^{\text{CLIP}}(\theta) = E\left[\min(r_t(\theta)D_\pi(S_t, A_t), \text{clip}(r_t(\theta), 1-\epsilon, 1+\epsilon)D_\pi(S_t, A_t))\right], \tag{1.34}$$

with

$$r_t(\theta) = \frac{\pi_\theta(S_t, A_t)}{\pi_{\theta_{\text{old}}}(S_t, A_t)} \tag{1.35}$$

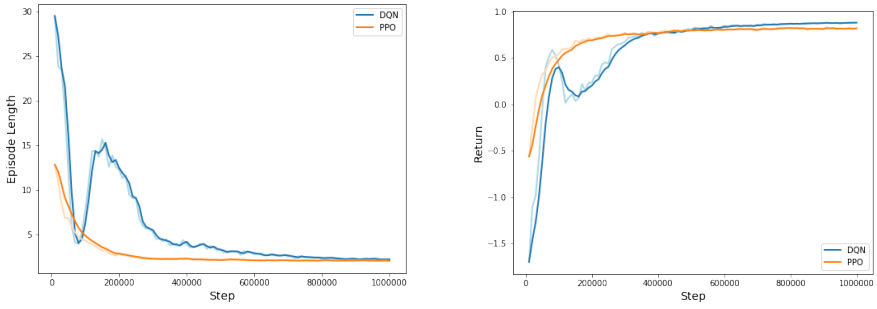
The clip operator acts like a boundary to the policy ratio. When the policies are too far from each other, the formula replaces the value of r_t by either $1-\epsilon$ or $1+\epsilon$, depending on whether D_t is positive or negative. We can then compute the gradient of equation 1.34 and use it to update our parameters multiple times sampling from a big pool of experiences.

PPO is considered the state of the art method in RL. It has been successfully applied to a wide range of different problems. Due to its versatility PPO is the method chosen by Unity ML-Agents. Since it is an open-source project the algorithm implementation and all the necessary functions that allow to train agents in the Unity engine are publicly available in GitHub. The link to the repository can be found in the Methods chapter.

1.9.6 Dynamic grid world DQ-learning and PPO

In this section we will test and compare the performance of Unity's PPO implementation with our own version of DQN on the grid world example. In order to make it more challenging, we modify the environment so that the position of the objects is randomly selected in every episode.

Just knowing its own location the agent would be incapable of solving the problem. That's why, we also include among its state observations the position of the asteroid and the gold nugget at every time step. Therefore, the state representation consists of a 6 element vector containing the coordinates x and y for each of the objects. It might seem that from the agent's perspective the information fully describes the environment and should be enough to quickly obtain the optimal policy. However, since the vector of observations is continuous, the function approximator not only needs to estimate action values or actions distributions for the few allowed coordinate locations in the grid, but also for the infinite number of positions in between. In other words, the algorithm has to come up with a smooth and flexible NN that can sufficiently describe a continuous version of the environment.



(a) Mean episode length per time step

(b) Mean cumulative reward per time step

Figure 1.9: Comparison between DQN and PPO algorithms on the dynamic grid world environment. Although for the same number of iterations DQN seems to reach a higher mean cumulative reward, PPO's implementation uses very little memory compared to DQN and thus, it runs much faster. Note that the blue curve also contains more oscillations which can be attributed to abrupt changes in the policies

The performance comparison between DQN and PPO is depicted in figure 1.9. The plot on the right shows how the two methods are able to converge to a

near optimal solution in 1×10^6 steps. Although in the static version of the environment we only penalized the agent with a negative reward of -0.01 per time step. With this choice the agent quickly learned to reach the goal but not always following the shortest path. It was only after more than 3×10^6 that both algorithms found an optimal solution. In figure 1.1 we used a penalty of -0.05 instead. This makes the agent worry more about reaching the gold nugget in fewer movements. Note that this change only decreases the value of the returns after every episode but it does not affect the optimal policy. The left plot in figure 1.9 shows how as the number of steps increases the episodes get shorter and shorter.

On the other hand, since the position of the objects is not fixed and changes from one episode to another, the agent is implicitly forced to explore all the states and thus, there is no need for applying a soft policy. The agent can solve the problem by always acting greedily. This is proved by the blue curve in figure 1.9.

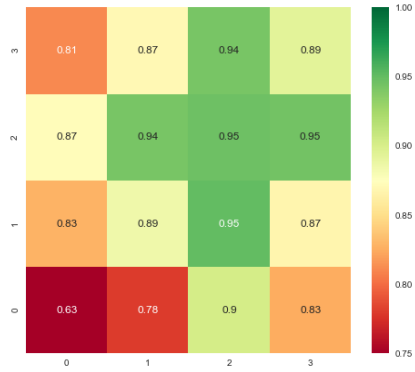
It is also worth mentioning that, although both algorithms take a similar number of steps to find the optimum, PPO does so much faster. This is because in order for DQN to work well we need to keep a considerably large buffer of experiences which makes the code more demanding in terms of memory, and consequently slower. One solution that we adopted to speed up the process, was to maintain a fixed size for our array of samples, throwing away the oldest experiences as new ones came in. This worked well as long as the array was kept large enough, but the method failed to converge when making the buffer smaller.

It is not clear, that the oldest experiences are less important than the others. In fact, they are more susceptible to contain examples of bad action choices, which can actually be useful to approximate our value function. With this in mind we modified our code so that experiences are removed at random. This allowed us to reduce even more the size of the buffer without compromising convergence. A more sophisticated approach would be to smartly select which samples we want to remove. This is somewhat similar to the method called Prioritized Experience Replay proposed in [SQAS15]. The authors of this paper designed a framework where relevant experiences more likely to be sampled from the buffer when the model is updated.

As mentioned before, because in DQN we select the actions that maximize the estimated return at every step, our policies can change very abruptly provoking dramatic oscillations in both episode length and cumulative reward (figure 1.9). In contrast, adjusting the parameters in policy gradient methods only causes slight changes in the distribution over actions. That is why the orange curves consistently move towards the optimal values.

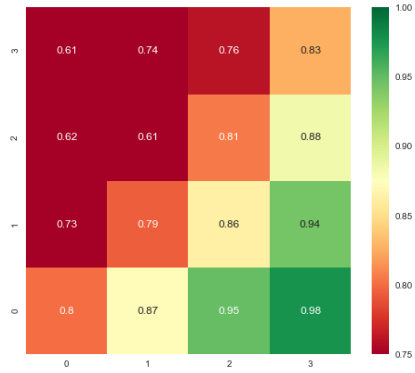
Finally, in figure 1.10 we plotted the state-value function for different locations of the agent while maintaining the asteroid and the gold nugget fixed.

(a) 1e4 steps



(b) 5e5 steps

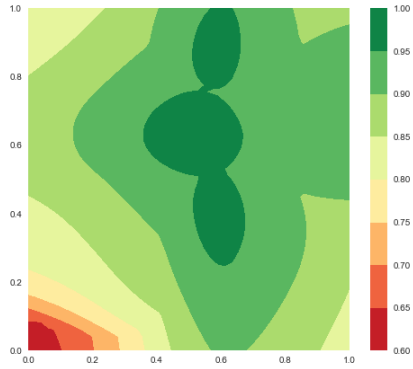
(c) 1e5 steps



(d) 2e5 steps

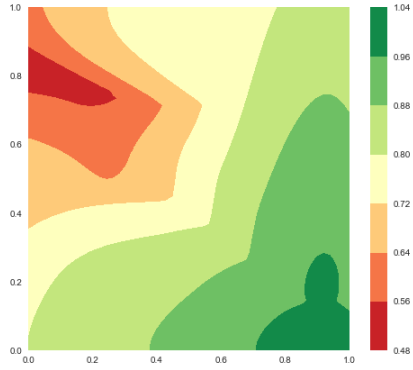
Figure 1.10: State-value function estimation using DQN. The values shown in the heat map on the right correspond to the state-values at each agent location when fixing the two objects position as shown on the screen shots. The values might differ from the true ones but they still lead to an optimal policy when acting greedily. Note that, in order to improve performance, the step penalty has been increased to -0.05, therefore altering the expected return at every state.

(a) 1e4 steps



(b) 5e4 steps

(c) 1e5 steps



(d) 2e5 steps

Figure 1.11: State-value function estimation using DQN. The values shown in the contours on the right correspond to the state-values at each agent location when fixing the two objects position as shown on the screen shots. The values might differ from the true ones but they still lead to an optimal policy when acting greedily.

Chapter 2

Multi-Agent Reinforcement Learning

2.1 Stochastic Markov Games

Methods

Unity Engine

Unity ML-agents

Python and Tensorflow

Stuff

This appendix is full of stuff ...

Bibliography

[Has10] Hado V Hasselt. Double q-learning. In *Advances in Neural Information Processing Systems*, pages 2613–2621, 2010.

[Lin92] Long-Ji Lin. Self-improving reactive agents based on reinforcement learning, planning and teaching. *Machine learning*, 8(3-4):293–321, 1992.

[MKS+15] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A Rusu, Joel Veness, Marc G Bellemare, Alex Graves, Martin Riedmiller, Andreas K Fidjeland, Georg Ostrovski, et al. Human-level control through deep reinforcement learning. *Nature*, 518(7540):529, 2015.

[SB98] Richard S Sutton and Andrew G Barto. *Reinforcement learning: An introduction*, volume 1. MIT press Cambridge, 1998.

[SLA+15] John Schulman, Sergey Levine, Pieter Abbeel, Michael Jordan, and Philipp Moritz. Trust region policy optimization. In *International Conference on Machine Learning*, pages 1889–1897, 2015.

[SQAS15] Tom Schaul, John Quan, Ioannis Antonoglou, and David Silver. Prioritized experience replay. *arXiv preprint arXiv:1511.05952*, 2015.

[SWD+17] John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal policy optimization algorithms. *arXiv preprint arXiv:1707.06347*, 2017.

[VHGS16] Hado Van Hasselt, Arthur Guez, and David Silver. Deep reinforcement learning with double q-learning. In *AAAI*, volume 16, pages 2094–2100, 2016.