LIÈGE université
Sciences Appliquées

INFO8003-1

OPTIMAL DECISION MAKING FOR COMPLEX PROBLEMS

Final project - report

Inverted Double Pendulum : Searching High-Quality Policies to Control an Unstable Physical System.

Authors:

Pierre NAVEZ (s154062)
Antoine DEBOR (s173215)

Lecturer:

Pr. D. ERNST

T.A. :

S. AITTAHAR

B. MIFTARI

Friday 14th May, 2021

1 Domain

This first section provides a formalization of the Double Inverted Pendulum environment based on this source code.

The "Double Inverted Pendulum" control problem - Environment

- **General characterization** : the environment is deterministic (the next state is fully determined by the action taken, there is no stochastic behaviour) and fully-observable (the state status describe fully the environment), and its dynamics are continuous and time-invariant (the dynamics do not explicitly depend on time). Furthermore, it is a single-state environment (only one state is reached at each step) and only contains one agent (the pendulum mounted of the cart), making it a single-agent environment.

- **System dynamics** : the dynamics of the considered system are not formalized in this work. However, one defines the variables used: z the horizontal position of the cart's center of mass, $\dot{z}$ the horizontal speed of the cart's center of mass, θ the angle between the vertical reference and the lower arm of the pendulum, $\dot{\theta}$ the angular speed of the lower arm of the pendulum with respect to the vertical reference, ϕ the angle between the vertical reference and the upper arm of the pendulum, $\dot{\phi}$ the angular speed of the upper arm of the pendulum with respect to the vertical reference and $Force$ the horizontal force applied by the agent on the cart. These variables can be visualized in the schematic of fig. 1.

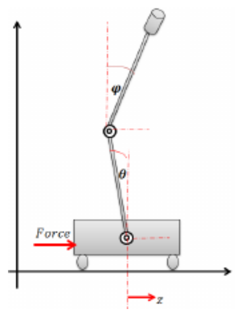


Figure 1: Schematic configuration of a loaded double inverted pendulum

- **State space** : the state space X is composed of $\{(z, \dot{z}, \theta, \dot{\theta}, \phi, \dot{\phi}) \in \mathbb{R}^6\}$ and of a terminal state reached when the distance between the upright and the current state is above a given threshold. This terminal state is described more formally hereafter.

- **Initial state** : the two-link pendulum mounted on the center of the cart is initially set to (nearly) upright position above the cart. If the horizontal position of the cart is considered as initially equal to zero, the initial state can be written as (nearly) equal to $\{(z, \dot{z}, \theta, \dot{\theta}, \phi, \dot{\phi})\} = \{(0, 0, 0, 0, 0, 0)\}$.

- **Terminal state** : the terminal state mentioned in the state space definition is reached when the distance between the upright and the current state is above a given threshold. Formally, this terminal state is reached if

$$pos_y + 0.3 \leq 1,$$

  with pos_y the vertical position of the second pole of the pendulum. In the considered source code, they consider the middle of the second rod as the second pole, which requires the addition of a 0.3 term to consider the extremity of the rod. Noting pos_x the horizontal position of the second pole, pos_x and pos_y can be derived from the state space once the dimensions of the pendulum are known.

- **Action space** : the action space is given by $U = [-1, 1]$. Actually, any action value can be virtually provided, but the effective action value will be clipped between -1 and +1. The action is then multiplied by a factor of 200 and provided as a torque to the cart. It has been decided to consider the action u as the variable upstream of this conversion into mechanical energy, and not as the multiplied value. The action is related to the $Force$ applied on the cart as shown in fig. 1.

- **Reward function** : the reward function $r(x, u)$ is composed of three components:
  - a constant and positive signal, $r_{alive} = 10$, rewarding the agent for maintaining the pendulum above the threshold and acting as an incentive to keep interacting with the environment
  - a variable distance-related penalty, $r_{distance}(pos_x, pos_y) = 0.01 \times pos_x^2 + (pos_y + 0.3 - 2)^2$, penalizing the agent for not reaching the upright position with the pendulum (and, with a weaker impact, from not being close to $pos_x = 0$)
  - a velocity-related penalty r_{vel}[1], which is zero-valued in the considered implementation of the environment and does thus not account for the velocity of the cart

  The reward function is hence defined as $r(x, u) = r_{alive} - r_{distance}(pos_x, pos_y) - r_{vel}$.

- **Time horizon** : $T \to \infty$

- **Discount factor** : although the discount factor is not strictly part of the environment as it rather represents the interface between the agent and the environment and how the former considers the rewards perceived from the latter, it has been decided to define it in this formalization. The decay factor used is $\gamma = 0.99$.

2 Policy search techniques

In order to solve the problem with reinforcement learning, two algorithms are implemented. Deep Q-Network (DQN) [1] for discrete action space, and Deep Deterministic Policy Gradient (DDPG) [2] for continuous action space. In the following, both algorithms are going to be explained and it will be shown that DDPG is related to DQN. In addition to those two methods, the well-known FQI algorithm from Ernst et al. [3] is used to compare the results.

Deep Q-Network (DQN)

As its name indicates, DQN combines Q-learning with Deep Neural Networks. As a reminder, Q-learning aims at finding the optimal action-value function, ie.

$$Q^*(s, a) = \max_\pi \mathbb{E}\left[r_t + \gamma r_{t+1} + \gamma^2 r_{t+2} + \ldots \mid s_t = s, a_t = a, \pi\right] \tag{1}$$

where r_t is the reward at time t, γ is the discount factor and the action a from state s is taken at time t following the policy $\pi = P(a|s)$. In other words, the goal of the agent is to select actions in a fashion that maximizes the sum of cumulative future rewards. Note that there are several ways to write expression 1.

In this case, the optimal Q function is learned by a neural network. However, using neural nets to learn can lead Q to be unstable. This is either due to the correlation in the sequences of observations, either due to the fact that small updates of Q can greatly modify the policy and the data distribution, or, due to the correlation between Q and the target value $r + \gamma \max_{a'} Q(s', a')$. DQN handles these problems using from one hand: experience replay, that randomizes over the data, thus, removes the correlation in the observation sequence, and smooths the data distribution. One the other hand, the values of Q are only periodically updated during the training iterations, in order to remove the correlation between Q and the target value.

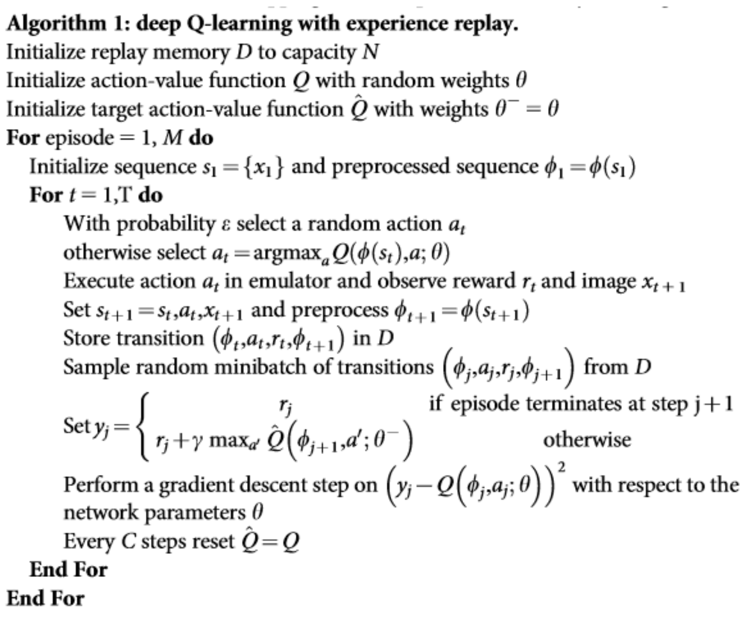
For training the network, an approximate value function $Q(s, a, \theta_i)$ is parametrized using a deep neural network (please note that the pseudo code from the paper supposes that a convolutional neural network is used), in which θ_i are the weights of the Q-network at iteration i. Experience replay is performed by storing the agent's experience $e_t(s_t, a_t, r_t, s_{t+1})$ in a dataset $D_t = \{e_1, ...e_t\}$. During learning, Q-learning updates are applied on minibatches of experience $(s, a, r, s') \sim U(D)$ drawn uniformly at random from the pool. The loss function used to perform these updates is the following:

$$L_i(\theta_i) = \mathbb{E}_{(s,a,r,s') \sim \mathrm{U}(D)}\left[\left(r + \gamma \max_{a'} Q\left(s', a'; \theta_i^-\right) - Q\left(s, a; \theta_i\right)\right)^2\right] \tag{2}$$

[1]The dependency of r_{vel} upon $\dot{pos}_x$ and $\dot{pos}_y$ has been neglected, as this penalty is zero-valued.

where γ is the discount factor, θ_i are the Q-network's parameter at iteration i and θ_i^- are the parameters used to compute the target at iteration i. The target network parameters θ_i^- are only updated with the Q-network parameters (θ_i) every C steps and are held fixed between individual updates.

The pseudo-code corresponding to the algorithm is depicted on figure 2. It originates from the DQN paper [1].



```
Algorithm 1: deep Q-learning with experience replay.
Initialize replay memory $D$ to capacity $N$
Initialize action-value function $Q$ with random weights $\theta$
Initialize target action-value function $\hat{Q}$ with weights $\theta^- = \theta$
For episode = 1, $M$ do
  Initialize sequence $s_1 = \{x_1\}$ and preprocessed sequence $\phi_1 = \phi(s_1)$
  For $t$ = 1,T do
     With probability $\varepsilon$ select a random action $a_t$
     otherwise select $a_t = \text{argmax}_a Q(\phi(s_t),a;\theta)$
     Execute action $a_t$ in emulator and observe reward $r_t$ and image $x_{t+1}$
     Set $s_{t+1} = s_t,a_t,x_{t+1}$ and preprocess $\phi_{t+1} = \phi(s_{t+1})$
     Store transition $(\phi_t,a_t,r_t,\phi_{t+1})$ in $D$
     Sample random minibatch of transitions $(\phi_j,a_j,r_j,\phi_{j+1})$ from $D$
     Set $y_j = \begin{cases} r_j & \text{if episode terminates at step j+1} \\ r_j + \gamma \max_{a'} \hat{Q}(\phi_{j+1},a';\theta^-) & \text{otherwise} \end{cases}$
     Perform a gradient descent step on $(y_j - Q(\phi_j,a_j;\theta))^2$ with respect to the
     network parameters $\theta$
     Every $C$ steps reset $\hat{Q} = Q$
  End For
End For
```

Figure 2: Pseudo code of DQN algorithm

The complete neural net's architecture used in this project is detailed in the section 3.

Deep Deterministic Policy Gradient (DDPG)

The DDPG algorithm is implemented to allow the agent to learn a continuous policy. This algorithm derives from DQN and a Policy Gradient approach, in particular, from the Deterministic Policy Gradient [4] algorithm which makes use of the Actor-Critic architecture [5]. DQN has been explained in the previous section, in order to understand the functioning of DDPG, let us begin by a description of what are Policy Gradient, DPG algorithm, including Actor-Critic methods.

Policy Gradient

The goal of reinforcement learning is to learn a policy that maximizes an expected reward. At the beginning, the agent has no clue about what is the right move to do in its environment, and takes its actions randomly. Over the time, the agent observes the state it is in, performs an action based on its current policy or current beliefs, observes the new state and receives a reward. Eventually, after a series of steps in the environment, the agent remembers the state-action pairs in its trajectory and knows what is its current total reward. Based on this trajectory and the value of the total reward, it should be able to update its current beliefs in order to maximize the rewards it will get in the future. Policy Gradient methods are based on this simple intuition.

Formally, the policy of the agent is the probability of taking an action a given a state s, which can be written $\pi_\theta(a|s)$, where θ is a parameter. The expected reward can be formalized as the sum of the probability of a trajectory τ, multiplied by the corresponding total reward of this trajectory $R(\tau)$. Mathematically, one has

$$J(\theta) = \mathbb{E}[\sum_{t=0}^{T} R(s_t, a_t); \pi_\theta] = \sum_\tau P(\tau;\theta)R(\tau). \tag{3}$$

So the objective is simply given by

$$\max_{\theta} J(\theta)=\max_{\theta} \sum_{\tau} P(\tau ; \theta) R(\tau). \tag{4}$$

One can formulate $J(\theta)$ as an expectation term, and express the optimal value θ^* of the optimization problem as the following

$$\theta^{*}=\arg \max_{\theta} \mathbb{E}_{\tau \sim \pi_{\theta}(\tau)}[r(\tau)] \tag{5}$$

where r is the reward function. As the name of the method indicates, a gradient descent optimization method is going to be used. To derive the expression of the gradient of $J(\theta)$, the expectation term is written as an integral, which can be done as the assumption that the space is continuous is made. Another useful trick is that the gradient of a function $f(x)$ can be written as the product of this function and the gradient of the logarithm of this function. Putting everything together, the *Policy Gradient* is

$$\begin{aligned} \nabla_{\theta} J(\theta) &= \int \nabla_{\theta} \pi_{\theta}(\tau) r(\tau) d\tau = \int \pi_{\theta}(\tau) \nabla_{\theta} \log \pi_{\theta}(\tau) r(\tau) d\tau \\ &= E_{\tau \sim \pi_{\theta}(\tau)}\left[\nabla_{\theta} \log \pi_{\theta}(\tau) r(\tau)\right]. \end{aligned} \tag{6}$$

The important result is the presence of the expectation term in the policy gradient expression, which means that policy gradient can be approximated using sampling. The presence of the logarithm is convenient for the optimization process. It will be useful for better scaling the weights θ. Indeed, taking the logarithm of a probability extends the range of values seen by the optimizer. Finally, the update step behind the policy gradient ascent method can be written as

$$\begin{aligned} &\nabla_{\theta} J(\theta) \approx \tfrac{1}{N} \textstyle\sum_{i=1}^{N}\left(\sum_{t=1}^{T} \nabla_{\theta} \log \pi_{\theta}\left(a_{i, t} \mid s_{i, t}\right)\right)\left(\sum_{t=1}^{T} r\left(s_{i, t}, a_{i, t}\right)\right) \\ &\theta \leftarrow \theta+\alpha \nabla_{\theta} J(\theta). \end{aligned} \tag{7}$$

Deterministic Policy Gradient and Actor-Critic architecture

In the previous subsection, a basic approach to Policy Gradient has been done. Now, a step further can be taken. It all begins with the *policy gradient theorem* [6] which provides an alternative formulation to equation 6. For a given parametrized policy π_θ and its associated cumulated reward $J(\theta)$ one has the following result

$$\nabla_{\theta} J(\theta)=\underset{\substack{s_{t} \sim d^{\pi_{\theta}(\cdot)} \\ a_{t} \sim \pi_{\theta}(\cdot \mid s_{t})}}{\mathbb{E}}\left\{\nabla_{\theta} \log \pi_{\theta}\left(a_{t} \mid s_{t}\right) Q^{\pi_{\theta}}\left(s_{t}, a_{t}\right)\right\} \tag{8}$$

where $d^{\pi_\theta}(s)=\lim_{t\to\infty}\sum_{t=0}^{T-1}\gamma^t P\left(s \mid \pi_\theta, t\right)$ is a discounted weighting of states encountered. This is an important result because despite the fact that the state distribution depends on the policy parameters, the policy gradient does not depend on the gradient of the state distribution. Several algorithms have been presented in the literature in order to learn a policy π_θ. However such a policy is stochastic, because the policy is basically a probability distribution for each action in the action space.

In DPG, instead of a stochastic policy π, a deterministic policy $\mu(.|s)$ is followed. For a given state s, there will be a deterministic action $a=\mu(s)$, instead of a probability distribution over action. In their paper Silver et al. [4] present the idea of DPG and show that the policy gradient of a deterministic policy can be written

$$\nabla_{\theta} J(\theta)=\underset{s_{t} \sim d^{\mu_{\theta}(\cdot)}}{\mathbb{E}}\left\{\nabla_{\theta} \mu_{\theta}\left(s_{t}\right) \nabla_{a} Q^{\mu_{\theta}}\left(s_{t}, a_{t}\right)\right\}. \tag{9}$$

The functioning of their algorithm relies on an Actor-Critic architecture. Two neural networks are implemented: an actor, a critic. The actor is responsible for learning which action should be taken given an input state, while the critic provides an approximation of the Q-function given the state and the action taken by the actor. As a result both networks are related to each other. Indeed, the actor will use the value provided by the critic as a feedback, in order to know how well does it performs its action. The actor network is optimized using the policy gradient equation 9. The critic updates its parameters using Q-learning, based on the temporal difference error (for the offline version of the algorithm). These update equations can be found here below

$$\begin{aligned}\delta_t &= r_t + \gamma Q^w\left(s_{t+1}, \mu_\theta\left(s_{t+1}\right)\right) - Q^w\left(s_t, a_t\right)\\ w_{t+1} &= w_t + \alpha_w \delta_t \nabla_w Q^w\left(s_t, a_t\right)\\ \theta_{t+1} &= \theta_t + \alpha_\theta \nabla_\theta \mu_\theta\left(s_t\right) \nabla_a Q^w\left(s_t, a_t\right)\big|_{a=\mu_\theta(s)}.\end{aligned} \tag{10}$$

In these equations, Q^w is the state action value function given by the critic network and μ_θ is provided by the actor. One can see from these equations that the optimizations of both networks are related to each other.

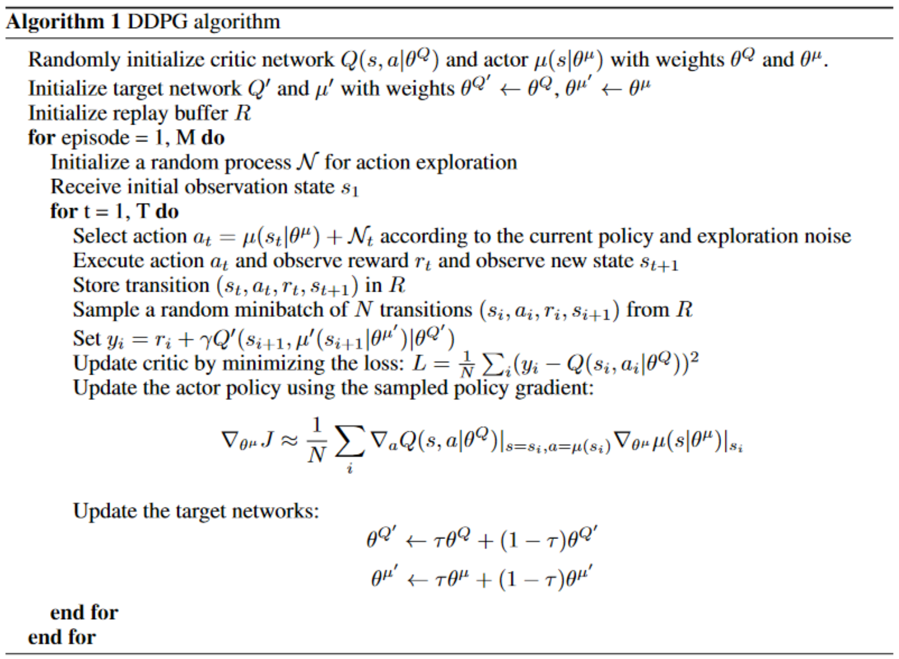
DDPG

Finally, with all these concepts understood, Deep Deterministic Policy Gradient algorithm can be explained. Formally, DDPG is a model-free, off-policy actor-critic algorithm that combines Deep Q Learning and Deterministic Policy Gradient. Unline DQN, this algorithm is able to learn in a continuous action space, using a deterministic policy.

As an off-policy algorithm, two separate policies are used for exploration and updates. The policy used for exploration is stochastic, while a deterministic policy is used for the target update. It is also an actor-critic algorithm, which has two networks. The actor produces the actions to explore and uses the temporal difference error from the critic as update rule. The critic which produces an approximation of the Q function, is updated based on the temporal difference error similarly to Q learning.

Such an algorithm faces the same unstable behavior as Deep Q Learning, which originates from the utilization of a neural network as the functional approximator. That is why DDPG also makes use of target networks, as well as experience replay.

In the original paper [2], further concepts like the addition of exploration noise or weights initialization are discussed. It is going to be explained in the section 3, at the same time as the implementation details of the algorithm. For now, and for a better overview of the functioning of DDPG, the pseudo-code included in the original paper is depicted on Figure 3



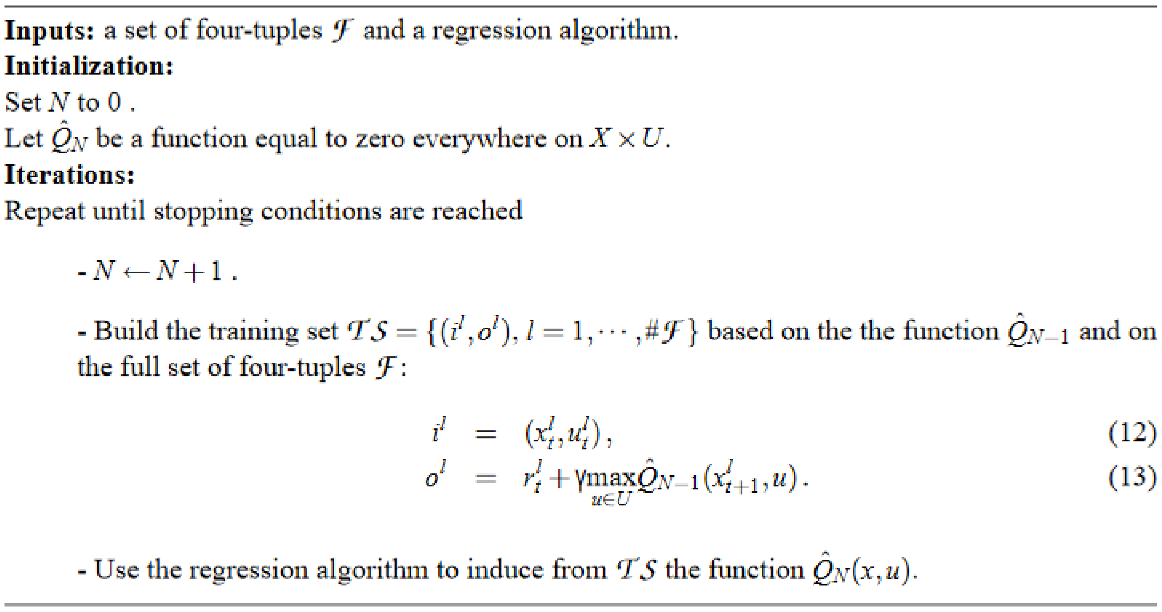
Algorithm 1 DDPG algorithm

```
Randomly initialize critic network $Q(s,a|\theta^Q)$ and actor $\mu(s|\theta^\mu)$ with weights $\theta^Q$ and $\theta^\mu$.
Initialize target network $Q'$ and $\mu'$ with weights $\theta^{Q'} \leftarrow \theta^Q$, $\theta^{\mu'} \leftarrow \theta^\mu$
Initialize replay buffer $R$
for episode = 1, M do
  Initialize a random process $\mathcal{N}$ for action exploration
  Receive initial observation state $s_1$
  for t = 1, T do
    Select action $a_t = \mu(s_t|\theta^\mu) + \mathcal{N}_t$ according to the current policy and exploration noise
    Execute action $a_t$ and observe reward $r_t$ and observe new state $s_{t+1}$
    Store transition $(s_t, a_t, r_t, s_{t+1})$ in $R$
    Sample a random minibatch of $N$ transitions $(s_i, a_i, r_i, s_{i+1})$ from $R$
    Set $y_i = r_i + \gamma Q'(s_{i+1}, \mu'(s_{i+1}|\theta^{\mu'})|\theta^{Q'})$
    Update critic by minimizing the loss: $L = \frac{1}{N}\sum_i (y_i - Q(s_i, a_i|\theta^Q))^2$
    Update the actor policy using the sampled policy gradient:
        $\nabla_{\theta^\mu} J \approx \frac{1}{N}\sum_i \nabla_a Q(s,a|\theta^Q)|_{s=s_i,a=\mu(s_i)} \nabla_{\theta^\mu}\mu(s|\theta^\mu)|_{s_i}$
    Update the target networks:
        $\theta^{Q'} \leftarrow \tau\theta^Q + (1-\tau)\theta^{Q'}$
        $\theta^{\mu'} \leftarrow \tau\theta^\mu + (1-\tau)\theta^{\mu'}$
  end for
end for
```

Figure 3: Pseudo-code of DDPG algorithm

Fitted-Q Iteration

To compare the performances of an algorithm, it is always convenient to compare the obtained results to other known algorithms. For that purpose, it has been decided to implement the *Fitted-Q Iteration* (FQI) algorithm, which has already been studied in a previous project. For that reason, FQI's foundations are not going to be discussed in great details in this work. However, to be consistent with DQN and DDPG, the pseudo-code issued from the original paper [3] is included on Figure 4 as well.



```
Inputs: a set of four-tuples $\mathcal{F}$ and a regression algorithm.
Initialization:
Set $N$ to 0 .
Let $\hat{Q}_N$ be a function equal to zero everywhere on $X \times U$.
Iterations:
Repeat until stopping conditions are reached

    - $N \leftarrow N+1$ .

    - Build the training set $\mathcal{TS} = \{(i^l, o^l), l = 1, \cdots, \#\mathcal{F}\}$ based on the the function $\hat{Q}_{N-1}$ and on
    the full set of four-tuples $\mathcal{F}$:

        $i^l = (x_t^l, u_t^l)$ ,                                                   (12)
        $o^l = r_t^l + \gamma \max_{u \in U} \hat{Q}_{N-1}(x_{t+1}^l, u)$ .        (13)

    - Use the regression algorithm to induce from $\mathcal{TS}$ the function $\hat{Q}_N(x,u)$.
```

Figure 4: Pseudo-code of the FQI algorithm

Basically, using this algorithm allows to iteratively learn an approximation $\hat{Q}$ of the Q function, using a set of one-state transition and a classical regression algorithm from the Machine Learning theory. Eventually, this function will converge and the optimal policy can be derived from the approximated Q function using $\hat{\mu}_N^*(x) = \arg\max_{u \in U} \hat{Q}_N(x,u)$. FQI has the advantage that it can be used both in discrete and continuous spaces. For the implementation, two choices have to been made. From one hand, a stopping condition, that will determine when the algorithm stops iterating, based on the assumption that $\hat{Q}$ has converged. On the other hand, a regression algorithm. Both of these choices are discussed in the section 3, which provides the implementation details.

3 Implementation details

In this section, the implementations of the various algorithms that have been described are detailed. In particular, the architecture of the neural nets implemented in the project are described, some precision about the parameters are brought and the changes from what is described in the different papers are explained.

3.1 DQN

The python file corresponding to this algorithm is `dqn.py`

Neural net architecture

In DQN, only one neural network architecture has to be defined. Naturally, this single architecture is used for the main network and the target network as well. In the original paper [1], the authors use a deep convolutional neural network to train the agent with images as input data. In this project, the neural net is fed with the observations from the environment, therefore, it has 9 neurons at the input layer.

At the output, there are 21 neurons. Indeed, it has been decided to discretize the action space (i.e. [-1,1]) with 0.1 steps. One might have used an other discretization rule, but this choice is made with the assumption that too much possible actions would not be consistent with the concept of a discrete space, while a very small number of them would not allow the agent to learn a good policy for this chaotic physical system.

In addition, there is one hidden fully connected layer. After the input and the hidden layer, Rectified Linear Unit (Relu) are added as activation function. This choice is made because this is the default activation function used in deep learning.

Schematically, the network looks as follows

- Fully connected layer (9 inputs, 256 output)
- Relu activation function
- Fully connected layer (256, 256 output)
- Relu activation function
- Fully connected layer (256 input, 21 output)

Finally, the learning rate is set to 0.001, and the network is optimized using the Adam optimizer.

Replay buffer
The replay buffer is basically a double ended queue with a one million transition capacity. From this replay buffer, at each episode, a batch of 128 transitions is sampled from the buffer.

Target network and update frequency
As neural networks are non-linear functional approximators, using a target network is a common trick in reinforcement learning. The update frequency of this target network C is set to 10. It means that every 10 steps, the target network parameters are updated with those of the main network.

Policy
At each episode during the training, the agent collects experience following an epsilon greedy policy. The value of epsilon is set dynamically, starting from 1 and decreasing towards 0.01, when the number of episodes increases. This method allows a better exploration/exploitation trade-off. Indeed, at the beginning of the training, the agent has no experience at all and needs to explore the environment as much as possible. That is why epsilon is set to a great value. Slowly but surely, the agent learns to perform good moves by itself such that the actions it decides to take are already good and epsilon can be set to a low value. In fact, an action space with the 21 possible actions is implicitly associated to each output of the Q network. In that way, when the agent is asked to chose an action the index of the neurons that outputs the greatest Q value will correspond to the index of the action in the action space.

3.2 DDPG

For this algorithm, every parameters of the two neural net architectures are consistent with those of the original paper. The implementation can be found in the file `ddpg.py`

Neural net architecture
As explained during the theoretical overview of DDPG, this algorithm relies on the Actor-Critic method, it has thus two neural networks.

1. Actor network:
   The actor network is responsible for learning to chose an action. It takes as many input as there are observations in the environment, 9 in this case. At the output, a single neuron provides the value of the continuous action between -1 and 1. Between the input and output layers, there are two fully

connected layers, both followed by some batch normalization layer, for re-centering and re-scaling the data.

   The architecture is described schematically here below,

   - Fully connected layer (9 inputs, 400 outputs)
   - Batch normalization layer
   - Fully connected layer (400 inputs, 300 outputs)
   - Batch normalization layer
   - Fully connected layer (300 inputs, 1 output)
   - Tanh activation function

   The presence of the hyperbolic tangent as activation function of the output layer clips the action value in the right interval (i.e [-1,1]). The weights of the network are initialized uniformly, the authors state that it allows to converge faster during the training.

2. Critic network:
   The critic is responsible to infer the Q function. It is basically a series of fully connected layers with some batch normalization and Relu activation layers. As its name indicates, the state action value function needs a state and an action to predict a value. The particularity of this network is that only the state values will pass through the two first layers. The action comes later, passing through its own fully connected layer before being added to the current state value and visit the very last layer both combined.

   As usual, a schematic representation of the architecture is provided,

   - Fully connected layer (9 inputs, 400 outputs)
   - Batch normalization layer
   - Relu activation layer
   - Fully connected layer (400 inputs, 300 outputs)
   - Batch normalization layer
     Here, the action value comes into play
   - Fully connected layer (1 input, 300 outputs)
   - Relu activation layer
     Here, the 300 outputs issued from the action and the state are added
   - Relu activation layer
   - Fully connected layer (300 inputs, 1 output)

   Similarly to the actor network, the weights of the critic network are initialized uniformly.

According to the original paper, the learning rates have been set respectively for the actor and the critic, to 0.000025 and 0.00025.

To improve the stability, target networks are used. At each one step transition, according to the pseudo-code of Figure 3, the weights of the target networks are updated accordingly to a linear combination of their current weights and the the weights of the actor and critic, respectively. This is called soft target update. The parameter τ of the formula is set to 0.001.

Finally, regarding the training of the neural networks, Adam optimizer are used. The loss functions are those of the pseudo code, that is, a mean squared error between the target value of the Q function and the Q function of the critic (for the critic), and the sampled policy gradient for the actor.

A difference between our implementation and the pseudo-code of the algorithm it that in the embedded loop, inside the main loop, the number of steps T but a whole trajectory is performed until the agent reaches a terminal state.

Replay buffer
The replay buffer is the same as described in DQN implementation section.

Policy
The policy used in this algorithm to deal with the exploration/exploitation trade-off is based on a Ornstein–Uhlenbeck process. Without entering into details, their mathematical equation is used to add a temporally correlated noise to the current action and then to encourage the exploration of the agent in the environment.

3.3 FQI

Discrete case

The algorithm is implemented in the file `discrete_fqi.py`

Similarly to DQN, the action space is discretized in 21 actions. The regression algorithm used to infer the Q function is an ensemble of 50 extra-trees because they provided satisfying result in the second assignment. This regression algorithm is a meta estimator that fits the 50 randomized decision trees, on various sub-samples of the data set and uses averaging to improve the predictive accuracy and control over fitting. Initially, the algorithm is trained on a data set of 1000 trajectories generated with the 21 possible actions selected randomly during the trajectories. The set of 1000 trajectories lead to more less 30 000 one step transition. The ensemble of trees predict the Q function given an input state and an action. To use the optimal policy, each action must be tested in turn, and the one that led to the greatest Q value is kept.

Continuous case

The algorithm is implemented in the file `continuous_fqi.py`

In the case of a continuous action space, the algorithm is a bit more tricky. Indeed, using a regression tree, $\hat{Q}_N(x,u)$ is a piecewise-constant function of its argument u, when fixing the state value. In order to compute $\max_{u\in U}\hat{Q}_N(x,u)$, the value of $\hat{Q}_N(x,u)$ must be computed for every values of U that lies in between the discretization thresholds found in the tree. However, using ensembles of regression trees, the number of discretization threshold becomes large and this resolution scheme becomes computationally inefficient. For that reason, the same strategy used in the discrete case is implemented. However, the set of possible actions is increased to 41 such that the action space is discretized between $[-1,1]$ with 0.05 steps.

It would have been more accurate to increase the set of possible actions in order to modelize a continuous action space more faithfully. However, doing that would imply a lot more computations as every action must be tested in turn to predict the greatest Q-value.

In the continuous case, a set of 1000 random trajectories is used as training set as well but the actions taken in these trajectories are drawn uniformly from the continuous interval $[-1,1]$.

Stopping rule
The number of iterations of the algorithm has to be chosen. A theoretical bound exists on the difference in infinite norm, between the expected return provided by FQI and the optimal expected return. This bound is given by the following equation

$$\left\| J_\infty^{u_N^*} - J_\infty^{\mu^*} \right\|_\infty \leq 2\frac{\gamma^N B_r}{(1-\gamma)^2} \tag{11}$$

Where N is the number of iterations, γ is the discount factor, and B_r is the bound on the maximum reward. For an accuracy of 0.1, with $\gamma = 0.99$ and $B_r = 10$,

$$N \geq \log_{0.99} \frac{0.1 \times (1-0.99)^2}{2 \times 10} \tag{12}$$

It means that the algorithm would require at least $N = 1444$ iterations to ensure the convergence of the optimal policy.

After several tests, the algorithm turned to be unstable on this problem. That is why the value of N is set to 250 to avoid too long computations. Indeed, at each iteration of the leaning procedure, the expected return is computed, using the current policy, it means that at each step of the simulated trajectory, 21 predictions must be done in the discrete case and 41 in the continuous one. The extra-trees of the Sckicit-Learn library are not very fast to make predictions, as a result, on large actions spaces, the speed of the algorithm is limited by these computations.

3.4 Experimental protocol to assess the performance of the policies

This protocol is implemented in the files `scores.py`

Score of a policy

A proper way to assess the performances of the implemented algorithms compared to FQI would be to compute the *score* of a policy. This method is described in the paper [3]. Starting from a set of initial states, the expected return is computed from every initial state using the learned policy, and the mean expected return is subsequently computed. Mathematically, the score is defined as

$$\text{score of } \mu = \frac{\sum_{x \in X^i} J^{\mu}_{\infty}(x)}{\# X^i}. \tag{13}$$

This solution was planned to be implemented, however, the gym environment does not make possible to decide and set the initial state of the double inverted pendulum. Anyway, a description of the possible procedure is still included in the next paragraph.

A rigorous approach would be to compute the expected return for a great number of initial states, in order to generalize at best the performance of the policy. However, in this project, there is an infinite number of possible initial states and it would require a lot of computational resources to calculate the expected return for each of them. The strategy, in this case, would have been to sample a batch of states from a buffer used in an algorithm (terminal states non included), and to compute the mean expected return of each policy based on this batch of initial states.

Performances of the trained agents

After the training of each intelligent agent, the parameters of their respective models are saved. For example, in DQN, the weights of the Q-network are written on a .sav file, such that they can easily be reloaded and attributed to the agent once it must be tested in its environment.

Afterwards, the procedure is straightforward. The cumulated reward and cumulated discounted reward of each agent is computed during a simulation in the environment.

4 Results and discussion

In this section, the learning curves of every algorithms are presented in terms of cumulated expected rewards along with the number of episodes of training. Moreover, the results of the experimental protocol described in section 3.4 are reported and discussed.

4.1 Discrete case

1. DQN

- **Cumulated discounted reward during training**

  The plot of the evolution of the cumulated discounted reward of the DQN agent during training is found on Figure 5. In red, the curve follows the evolution of the expected return, in blue, the expected return over the last hundred episodes is averaged.

  The agent is trained during 15000 episodes. This number is quite high but regarding the pseudo code of Figure 2 the embedded loop in the main loop has been omitted. This does not affect the policy learned, because the target network is still updated at the same frequency, but including this loop

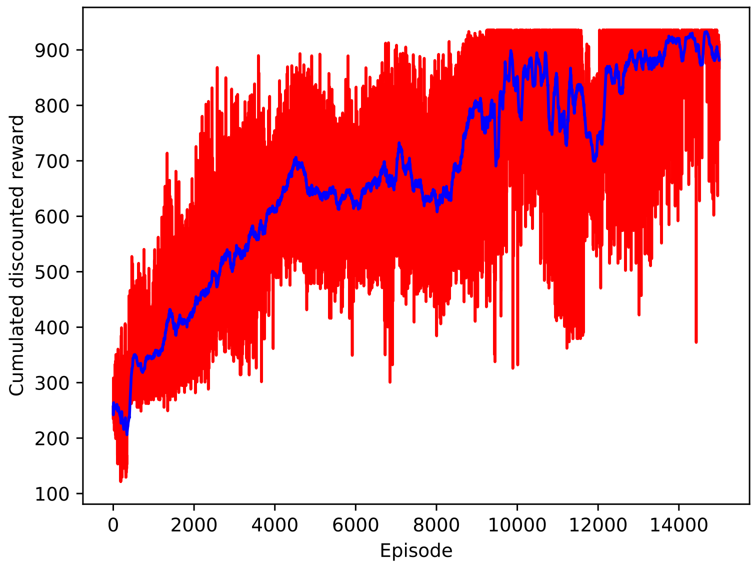


Figure 5: Red: Cumulated discounted reward of DQN during the training. Blue: Average over the last 100 episodes

would have lead to another figure, which would have seemed less noisy as the expected return would not be computed at each iteration.

The red burst emphasizes the fact that the training is unstable. The value of the expected return oscillates a lot at each iterations, but one can observe, especially with the blue curve, that the tendency of the expected return is on the rise.

The instability is explained from one hand by the fact that with neural networks, a small change in the input data during the training can greatly affect the parameters of the network. It was explained in section 2 that a replay buffer and a target network are used for that reason, one can imagine that without those particularities, the training would have been more pronounced. Indeed, the weights of the Q network are updated at each iteration, through back-propagation, using the result of the Bellman equation, that is, the neural net sees $Q(s_t, a)$ from the batch and $r_t + \gamma \max_{a'} Q(s_{t+1}, a')$ is computed as a target value. In fact, the second term the Bellman equation is evaluated with the *target network*. If a target net were not be used, the value of $Q(s_{t+1}, a')$ would be indirectly influenced by that of $Q(s_t, a)$ and thus the target value used to optimize the network would depend on the network itself, which would lead to instability.

On the other hand, the weights of the target network are updated every 10 episodes such that the phenomenon occurs periodically. The frequency update of the target network has been chosen experimentally. A too low value would lead to instability, as explained here above, but a greater value would avoid the network to learn efficiently and would slow down the learning process.

At the end of the 15000 episodes, one can see that a maximum value is reached, which indicates a beginning of the convergence. One could imagine that if the training process was prolonged, the red and blue lines would both converge towards this maximum value of about 935. Still, the agent was able to learn a policy as it will be shown when comparing the agents.

2. FQI

- **Cumulated discounted reward during training**
  The plot of the cumulated discounted reward of FQI on the discrete case is depicted on 6.

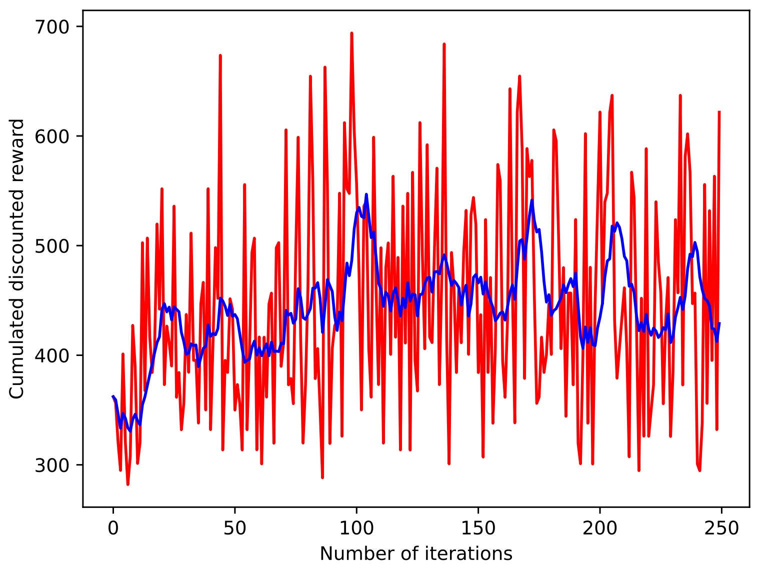


Figure 6: Red: Cumulated discounted reward of FQI (discrete) during the training. Blue: Average over the last 10 iterations

From the figure, it is clear that the training was not efficient. This could be due to several reasons. First, the training duration might not be sufficient. However, there is no indication on the figure that the cumulated discounted reward was going to increase. Indeed, the blue curve which represents the average over the last ten values of the red one, oscillates in the same interval during all the range of iterations.

Secondly, this could also be due to a lack of training data. Indeed, for this simulation, a set of 1,000 random trajectories with 21 discrete actions have been used as training set. In total, it represents about 30,000 one steps transitions. It can seems to be a lot, but on a continuous space domain, this could not be enough for some regression trees to infer a good policy for every possible state.

In fact, if the number of discrete actions was lower, it is possible that using a training set of 30,000 transitions would suffice to reach the convergence of the optimal expected return, however, the double inverted pendulum is a chaotic physical system which is undoubtedly impossible to master with a few possible actions, such that the expected return would have been low.

These hypothesis need to be confirmed empirically but as explained in section 2, the learning process is relatively slow and computing the cumulated discounted reward at each iteration is very greedy in terms of time. Moreover, as there are many parameters to tune, and due to the fact that such a very long training could have led to bad results, the focus has been put on the functioning of deep reinforcement learning algorithms.

4.2 Continuous case

1. DDPG

- **Cumulated discounted reward during training**
  The evolution of the cumulated discounted reward during training is depicted on Figure 7. The DDPG agent has been trained during 2,500 episodes. It has not fully converged yet, but the maximum value of 935 has been reached several times. Extrapolating the curves, one can imagine that the red and the blue ones would tend to be closer along with the number of episodes. However, to limit the

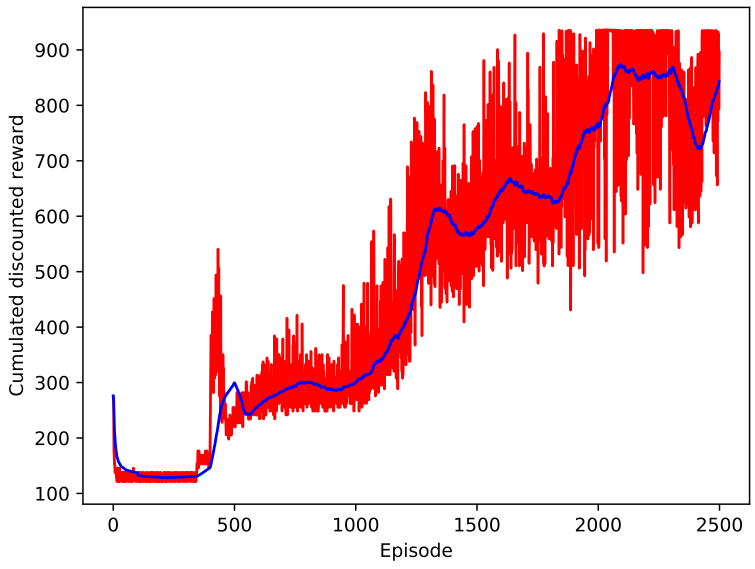


Figure 7: Red: Cumulated discounted reward of DDPG during the training. Blue: Average over the last 100 episodes

computation time, the number of episodes has been fixed to 2,500. This is very much less than for the DQN network, because in this implementation, the network is optimized several times during an episode. This explains why the curve looks less noisy.

From the average learning curve in blue, one can see that the increase of the average cumulated discounted reward is quasi monotic. Still, it arrives punctually that the average curve tends to decrease. This could be due to the impact of a bad update of the weights of the target networks even though this impact is limited by the soft target update. Anyway, neural networks are very sensitive to small changes in the data distribution they see and, oftentimes, it is hard to deduce the exact reason of this kind of behavior.

2. FQI

- **Cumulated discounted reward during training**

  The plot of the cumulated discounted reward during training is depicted on Figure 8.

  The same reasoning as in the discrete FQI occurs also in this case. The training did not go well and there is no indication about an increase trend of the learning curve.

  Compared to the discrete case, the training set in that case is full of one step transitions that took place randomly with continuous actions. An additional difference is the discretized action space with steps of 0.05 in the range $[-1, 1]$. This is probably not a faithful approximation of a continuous space but increasing the resolution would have led to a quadratic increase of the number of computations.

  The mapping between state and actions is more complex in this case as the number of possible actions is increased. Using a training set of the same size, this is evident that the learning process could not go better. Due to lack of time, the experience could not be performed, unfortunatly.

  As there is no evidence about the functioning or not of the algorithm, what one can conclude is that all these assumptions must be tested empirically, as a possible improvement of this work.

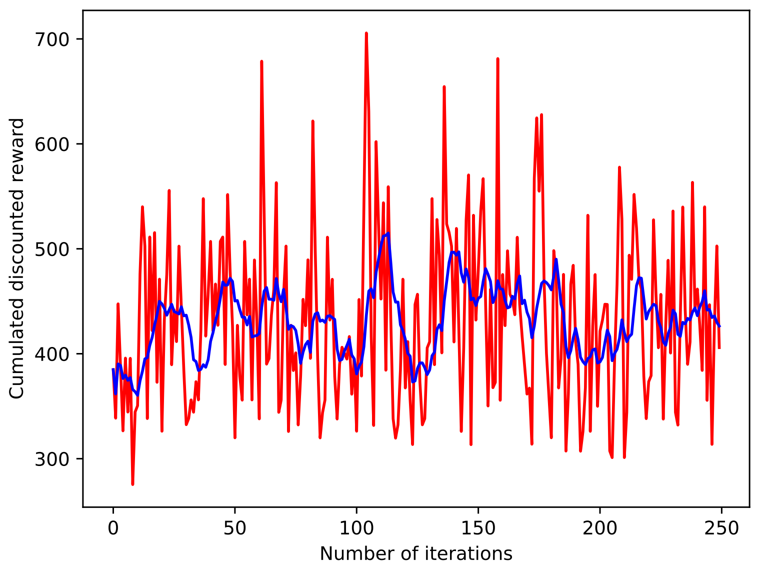


Figure 8: Red: Cumulated discounted reward of FQI (continuous) during the training. Blue: Average over the last 10 iterations

4.3 Comparison

Overall, regarding the learning curves, one can expect the DQN and DDPG agents to have learned an optimal policy. Contrariwise, both FQI agents should have a quasi-random behavior, which is possibly due to the lack of training or the impossibility to learn a mapping between the state action pairs, given only about 30,000 one step transitions.

On the script `scores.py`, the models that have been saved during or after the training are tested. It is needed to know, because after 2,500 episodes, the expected return could not have converged towards its optimal value yet, the models that led to the greatest expected return have been saved as well. This is not a proper manner to operate, but it was implemented as a backup solution if the training would end up on a sub-optimal expected return.

	DQN	FQI discrete	DDPG	FQI continuous
Cumulated reward	9358.24	492.57	9358.27	670.52
Cumulated discounted reward	935.51	384.25	935.67	480.22

Table 1: Comparative table of the 4 agent in terms of cumulated reward and cumulated discounted reward

As expected, on Table 1 one can see that both agents trained with a neural network are able to reach the optimal expected return. The cumulated discounted reward corroborates the maximum values reached on Figure 5 and Figure 7.

None of the FQI is able to reach satisfying results, as expected. In addition, FQI agents are very much slower than the neural networks as at each step, the predictions on every possible action is computed. When the number of action increases as in this case, the agent becomes unable to operate in real time. At the opposite, using neural net to compute predictions is very fast.

The comparison between discrete and continuous policies can not be done in this case because they both lead to same maximal cumulative reward. This is certainly caused by the fact that the action space was discretized into 21 actions, which gave the DQN agent a lot of freedom to adapt itself on the double inverted pendulum.

References

[1] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei Rusu, Joel Veness, Marc Bellemare, Alex Graves, Martin Riedmiller, Andreas Fidjeland, Georg Ostrovski, Stig Petersen, Charles Beattie, Amir Sadik, Ioannis Antonoglou, Helen King, Dharshan Kumaran, Daan Wierstra, Shane Legg, and Demis Hassabis. Human-level control through deep reinforcement learning. *Nature*, 518:529–33, 02 2015.

[2] Timothy P. Lillicrap, Jonathan J. Hunt, Alexander Pritzel, Nicolas Heess, Tom Erez, Yuval Tassa, David Silver, and Daan Wierstra. Continuous control with deep reinforcement learning, 2019.

[3] Damien Ernst, Pierre Geurts, and Louis Wehenkel. Tree-based batch mode reinforcement learning. *Journal of Machine Learning Research*, 6:503–556, 04 2005.

[4] David Silver, Guy Lever, Nicolas Heess, Thomas Degris, Daan Wierstra, and Martin Riedmiller. Deterministic policy gradient algorithms. *31st International Conference on Machine Learning, ICML 2014*, 1, 06 2014.

[5] Vijay Konda and John Tsitsiklis. Actor-critic algorithms. *Society for Industrial and Applied Mathematics*, 42, 04 2001.

[6] Richard S. Sutton and Andrew G. Barto. Second edition.