POLITECNICO
MILANO 1863

DATA ANALYTICS FOR MECHANICAL SYSTEMS PROJECT

GROUP MEMBERS:

Andrea Paludo

Jacopo Porzio

PROFESSORS:

Emanuele Riva

Beatrice Luciani

Academic year 2022/2023

1 CONTENTS

2 INTRODUCTION

The aim of this project is to solve a reinforcement learning (RL) problem by means of an existing algorithm.

We wanted to set problem in a way that, knowing which problem must be solved, the research is conducted with the aim of finding methods and an algorithm able to deal with the problem.
However, we wanted to find something which exploits artificial intelligence (AI), but still related to something which deals with mechanical systems. This led us to the choice of driving a car (mechanical system) by means of an AI algorithm.

As a first step we had to choose an environment: the choice fell on Gymnasium (Open-AI Gym), a very popular RL framework, specifically the environment *'Car – Racing'*.

Then, a deep deterministic policy gradient (DDPG) algorithm has been developed to train the model. DDPG has several advantages with respect to other methods because it is an off-policy actor-critic algorithm.
An off-policy algorithm can learn from experience generated either from other agents or from a different policy: in this case, from experience sampled from a replay buffer.
The actor-critic structure, then, allows the actor to take an action and to have a feedback from the critic, which evaluates the goodness of the undertaken action.
Finally, the name 'deep' in the algorithm suggests that both actor and critic are neural networks, i.e. complex functions approximators.

In the following we will introduce the basics of reinforcement learning (section 3), briefly discuss the environment (section 4), then we present the documentation on which the algorithm is developed, along with the papers which led us to choose this kind of architecture (section 5).

Finally, in section 6 we report the outcomes of our research along with some experimental results and the methods used to reach them.

3 BASICS OF REINFORCEMENT LEARNING

Reinforcement learning is a machine learning technique that involves training an artificial intelligence agent through the repetition of actions and associated rewards, in a specific environment. The architecture of the reinforcement learning can be summarized as follows:

- The RL agent receives a state S_0 from the environment.
- Based on that state S_0, the RL agent takes an action A_0.
- Now the environment is in a new state S_1.
- Environment gives a reward R_1 to the RL agent.
- Once a certain amount of experience is gained, we update the RL agent according to a metrics called loss function.

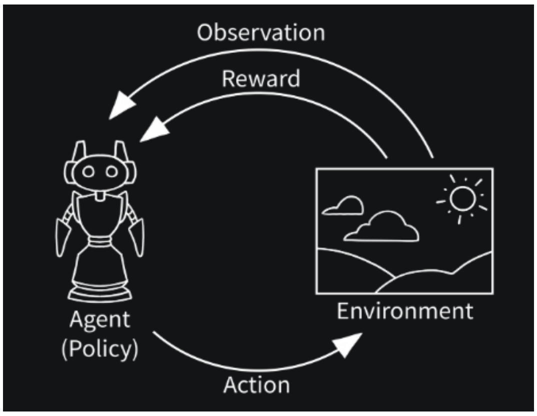


Figure 1: simplified scheme of the RL process.

To have a world-agent interaction which can grant sufficient exploration of the world itself and avoid poor local minima, the learning process takes advantage of the concept of episodes: the interaction is limited by certain flags that can trigger its termination, leading to the reset of the environment. This loop continues till the case specific objective is reached.

The update of the agent consists of a gradient step: its parameters are updated by means of the gradient of the loss function w.r.t. the parameters themselves. The step size is usually called *learning rate*.
Recently, model-based RL algorithms appeared in the literature, but we will focus only on model-free ones: the former is either based on a model of the agent and/or environment or tries to learn it, the latter doesn’t.

4 PRACTICAL GOAL

Open-AI Gym is a development and train environment for machine learning. It offers different environments which can be exploited by users to test and train some AI-agents through algorithms.

Gymnasium is a project that provides Python APIs (Application programming interface) for single agent RL environments: they come with a lot of different features and a large space for customization. The Gymnasium provides a standardized approach to RL problems consistent with its definition given in section 3: a simulated world and functions which allow to interact with it. Interacting means receiving an observation, apply an action given the observation, obtain the new observation, the reward and knowing whether the episode is terminated.

Concerning our work, we referred to *‘Box2D’* environments. They all involve toy-games based around physics control, using box2d-based physics and PyGame-based rendering.

4.1 ‘Car – Racing’ environment

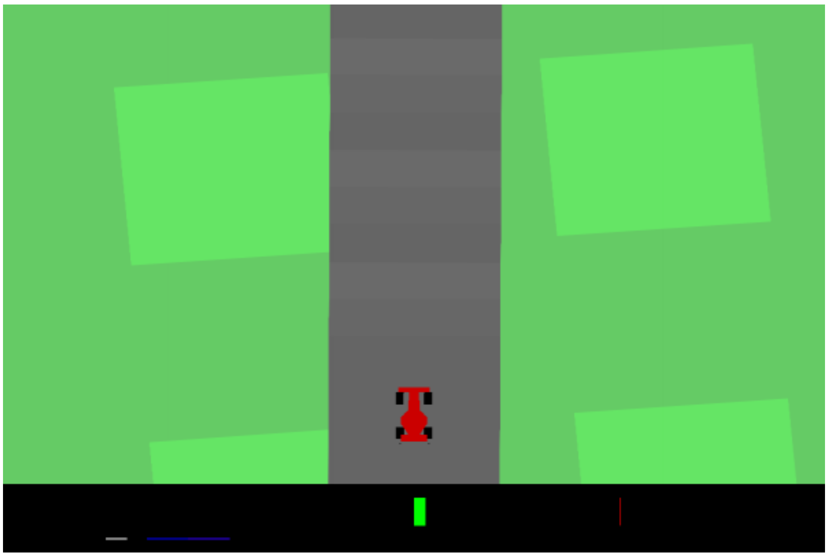


Figure 2: screenshot from *Car-Racing* environment.

- Description:
  - It is a top-down racing environment, where the control task is to be learned from pixels. The generated track is random every episode.
    Furthermore, the involved physics is a good representation of the real physics behind a racing vehicle, including different levels of friction for the track and the grass.
- Action space:
  - Continuous. There are three actions: steering, braking, accelerating.
- Observation space:
  - A top - down 96 x 96 RGB image of the car and racetrack.
- Reward:
  - The reward is -0.1 every frame and +1000/N for every track tile visited, where N is the total number of tiles visited in the track.
- Starting state:
  - The car starts at rest in the center of the road.
- Episode termination:
  - The episode finishes when all the tiles are visited. The car can also go outside the playfield, i.e., far off the track. In this case it will receive -100 reward and die.

As already mentioned, we want to train the car to make it able to drive autonomously while staying inside the track.
For this reason some modifications on the code have been made, to improve the learning performance. In section 6 we will discuss which are these modifications and how they work.

5 Documentation

First, it is worth to explain what is intended for ‘Off/On-policy Actor – critic’ structure and a brief definition of ‘Action – Value’ methods.

5.1 OFF – POLICY ALGORITHMS

The actor exploits the past experiences from other agents or from policies different from the one being currently used.
Thus, the actor can either learn from old experience or from other sources without the need of fully exploring the current environment. Following this approach, the learning process results to be more stable, due to the decorrelation of the exploration policy from the policies that generated the experience.
This allows to make use of already available training data and collected experience: therefore, the model gains in sampling efficiency. The sampling from experience happens with what are usually called batches.

Off-policy methods are widely deployed for tackling complex system control, robotics applications and games, since the agent learns from past experiences and the performance becomes better over time.

5.2 ON – POLICY ALGORITHMS

The actor can only learn from experience based on the current agent-environment interaction. It cannot exploit data from other policies or from other agents already gathered from past simulations.

The exploration policy coincides with the learned policy.
This kind of approach, which is simpler than an off-policy one, can be exploited in cases where we must deal with dynamic environment and later experience weight more than past one.

5.3 ACTION – VALUE METHOD

Action value method, Q-learning, is less advanced than the actor-critic structure. The aim is to estimate the value function 'Q' for each state-action couple. This method holds for discrete action spaces, since the optimal action is found through maximization of the value function over all the actions: in fact, the loss function for Q-learning is based on the Bellman equation and dynamic programming. It holds that $\max_{a}\big(Q(s,a)\big) = V(s)$, where Q is the Q-value function and V is the V-value function from Bellman's theory.
Even if this method is less performant than actor-critic method, its simpler architecture makes it suitable for many simple reinforcement learning tasks.

5.4 ACTOR – CRITIC METHOD

Actor-critic (AC) architecture is powerful when dealing with continuous action space environments. It is an extension of Q-learning to continuous action spaces, where the advantage is that it allows to avoid solving $\max_{a}\big(Q(s,a)\big)$.

AC algorithm is a RL technique employed to train models, in order to take optimal decisions in the environment. There are two main components:

- actor model: the actor is responsible for the selection of the actions to perform, given a state;
- critic model: the critic evaluates the performance of the actor, assessing the goodness of the chosen actions performed in each state. It serves as an approximator of the Q-value function.

The actor learns a policy which maps the state of the environment to action to perform. Its objective, which can be a defined performance measure (e.g., the cumulate reward over time) is to maximize the value function.

The critic, based on the states of the environment, tries to learn a Q-value function to estimate if the action taken by the actor is good or bad.

While training, the actor exploits the feedback from the critic to update and improve the policy; the critic uses the actors' actions to update the value function. The two updates happen independently, each one according to its own loss function. This leads to an iterative process of interaction between actor and critic, which has the aim of converging to an optimal policy that maximizes the value function.

To conclude, actor-critic method can be considered a more advanced method than action-value, since it combines the estimation of the action-value with the learning of the optimal policy. This allows more flexibility to adapt the agent's policy looking at the information given by the environment.

Let's now go more in depth about the references used for the experiment.

5.5 OFF-POLICY ACTOR-CRITIC, THOMAS DEGRIS ET AL., 2013, QUOTED BY 542

We now present the first actor-critic algorithm for off-policy reinforcement learning. The idea under the off-policy method is that the agent can learn from data gained from another policy, hence this can be a powerful way to improve the learning performance.
The authors wanted to combine the generality and learning potential of off-policy learning with the flexibility in action selection given by the actor-critic methods. Finally, they want to highlight how actor-critic structure is a good starting point for improving the learning performance.
The actor and critic models were implemented by mean of linear function approximators. *Off-PAC* is a stochastic method, as the actor model learns a distribution over the actions.

They develop the *Off-PAC* algorithm and compare its performance versus other three already existing off-policy action-value algorithms, which are: Q-Learning, Greedy-GQ, SoftMax-GQ.
Then, three different benchmark environments have been considered (Mountain car, Pendulum, Continuous grid world) with the common feature of having a discrete state space and a continuous action space. Deploying these algorithms to solve the problems, in all the tasks Off-PAC showed up to be the best performing.

The results of the article are that *Off-PAC* not only ended up being the best performing algorithm, but, in the last problem - the most challenging one -, it was the only one to learn and reach the goal reliably.
It can be said that, *Off-PAC* is a significant step toward robust off-policy control, with its biggest limit being the use of linear function approximators as actor and critic models.

5.6 Deterministic policy gradient algorithms. Silver et. al., 2014, quoted by 4097

Policy gradient algorithms are RL algorithms whose loss functions' gradients depend, indeed, on the gradient of the policy, and they are widely employed in RL problems with continuous action spaces.

A stochastic policy is a parametric probability distribution that stochastically selects an action in a state according to a parameter vector. Typically, a policy gradient algorithm proceeds by sampling this stochastic policy and adjusting the policy parameters in the direction of greater cumulative reward.

In this paper it is highlighted how deterministic policy gradient (DPG) has a convenient formulation. The main difference with the stochastic case is that DPG can be estimated much more efficiently than the usual stochastic policy gradient.
In addition, it is demonstrated that DPG outperforms their stochastic counterparts in high dimensional action spaces.

Starting from stochastic policy gradient, the authors prove the existence of deterministic policy gradient methods and they demonstrate that they are the limiting case of the stochastic policy gradient when the variance tends to zero.
The main difference between stochastic and deterministic policy gradient are:

- stochastic case: the policy gradient integrates over both state and action spaces;
- deterministic case: the policy gradient integrates over the state space only.

Therefore, the stochastic case requires more samples, especially in case of big size of the action spaces.
Hence, it is reasonable to tackle complex RL tasks with a deterministic approach.

Also, the authors wanted to highlight how an off-policy learning algorithm is useful to ensure proper exploration; hence, the proposed DPG chooses the action according to a stochastic behavioral policy, but it learns a deterministic policy, exploiting the sampling efficiency of DPG. The behavioral policy is said to be stochastic, because it's the composition of the learned deterministic policy and a stochastic superimposed noise, which grants for good exploration. Since the behavioral policy and the learned policy are different, DPG is said to be off-policy.

Summarizing, DPG is an off-policy actor-critic algorithm that estimates the action-value function through a linear function approximator, and then updates the policy parameters – linear approximations as well - in the direction of the approximate action-value gradient.
The algorithm has been tested versus its stochastic counterpart on different benchmarks and the deterministic approach outperformed by several orders of magnitude.

Usefulness of this deterministic approach can be highlighted also by the fact that it can deal with problems where there's no possibility to introduce noise, hence the stochastic policy gradient cannot be applied.

5.7 BATCH NORMALIZATION: ACCELERATING DEEP NETWORK TRAINING BY REDUCING INTERNAL COVARIATE SHIFT. SERGEY IOFFEE ET. AL., 2015, QUOTED BY 48507

This paper describes the *Batch Normalization* (BN) method which aims to improve the training of neural networks reducing the *internal covariate shift* effect.
First, the authors describe the phenomenon of internal covariate shift: the distribution of data changes while training, making it difficult for the layers to continuously adapt to the new input distributions. Hence the convergence becomes difficult to achieve.

To overcome this problem, the proposed solution is to normalize the activation of each input batch by means of statistical standardization, thus reducing the covariate shift. This method takes the name of batch normalization.
It is explained how the BN is applied to the activation of a layer, normalizing with zero mean and unitary variance, followed by a linear transformation and a shift.

This technique makes it possible to increase the learning rates and to be less careful about parameters initialization. In addition, in some cases it allows to avoid using dropout and increase the accuracy of the trained model.
Let's consider training and focus on the *i-th* node of a layer *J*, its activation x_i and a mini batch ***B*** of size *m.* The values can be normalized as follow:

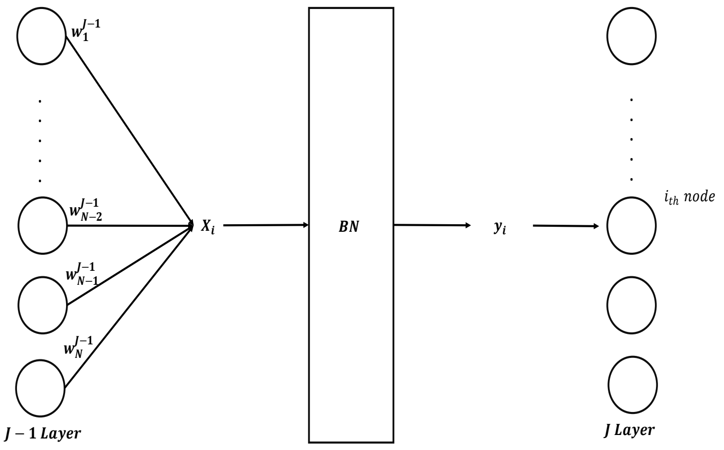


Figure 3: batch normalization scheme.

$$\mu_B = \frac{1}{m}\sum_{i=1}^{m} x_i \qquad (1)$$

$$\sigma_B^2 = \frac{1}{m}\sum_{i=1}^{m}(x_i - \mu_B)^2 \qquad (2)$$

$$\hat{x_i} = \frac{x_i - \mu_B}{\sqrt{\sigma_B^2 + \varepsilon}} \qquad (3)$$

$$y_i = \gamma\hat{x}_i + \beta \;\equiv BN_{\gamma,\beta}(x_i) \qquad (4)$$

γ and β are additional parameters to be learned while training, in order to obtain the linear output y_i.
This procedure ensures that the new input of the layer exhibit less internal covariate shift, thus accelerating the training process.

Then, there's the inference phase, where the trained model is used to make predictions on new samples. BN is deployed differently from the training phase.
Once the model has been trained for K training steps, each node will have seen K different means μ_B and variances σ_B^2. In order to recycle this information coming from the training, the procedure to be followed is the following one:

$$\bar{\mu} = \frac{1}{K}\sum_{i=1}^{K} \mu_{B,i} \qquad (5)$$

$$\bar{\sigma^2} = \frac{m}{m-1}\frac{1}{K}\sum_{i=1}^{K} \sigma_{B,i}^2 \qquad (6)$$

$$y = \frac{\gamma}{\sqrt{\overline{\sigma^2} + \varepsilon}} x + \left(\beta - \frac{\gamma\,\bar{\mu}}{\sqrt{\overline{\sigma^2} + \varepsilon}}\right) \qquad (7)$$

In equation (5) and (6) we compute the new values of mean and variance (unbiased variance) considering all the means and variances found while training. Basically, we compute the average over all the batches appeared in the training phase.
Since from now on, mean and variance are fixed during inference, the batch normalization is completed scaling by γ and shifting by β (equation 7), to yield a single linear transform.

5.8 CONTINUOUS CONTROL WITH DEEP REINFORCEMENT LEARNING. THIMOTHY P. LILLICRAP ET. AL., 2016, QUOTED BY 18831

This paper is the core of this work, since it explains the Deep deterministic policy gradient (DDPG) algorithm.
The paper presents an actor-critic, model-free algorithm based on deterministic policy gradient (DPG) that can operate over continuous action spaces.

Mainly two works are of utter importance for DDPG:

- DPG (Silver et al., 2014): model free, off-policy actor-critic algorithm using deep function approximators that learn policies in high dimensional, continuous action spaces. Unfortunately, this ends up being unstable when using neural function approximators in challenging problems.
- Deep Q-Network (DQN) (Mnih et al., 2015): it learns a Q-value function using large, non-linear, neural function approximators in a stable and robust manner for discrete action spaces, thanks to:
  - the network is trained off-policy with samples from a replay buffer to minimize correlations between samples;
  - the loss function is calculated with both the Q-network and a target one, i.e. a delayed copy of the network;
  - batch normalization (Sergey Ioffee et. al., 2015) was used.

The resulting algorithm takes the name of Deep deterministic policy gradient (DDPG); it can learn competitive policies using low dimensional observations, such as cartesian coordinates or joint angles, as well as using more structured inputs such as images.

Key feature of this approach is its simplicity. It requires an actor-critic architecture and it results being a learning algorithm with few hyperparameters, making it simple to implement and tune depending on the complexity of the problem.

Main contribution of the paper is to provide modifications to DPG, inspired by the success of DQN, which allows to use neural network function approximators to learn in large state and action spaces.

The off-policy actor-critic structure of DPG, ideal for continuous action spaces, is preserved, but its effectiveness is amplified thanks to the intuitions behind DQN: specifically, the use of target neural networks, both for actor and critic, to improve the stability of the algorithm, and a use of a replay buffer. The replay buffer is basically a finite size cache where transitions sampled from the environment according to the exploration policy are stored.

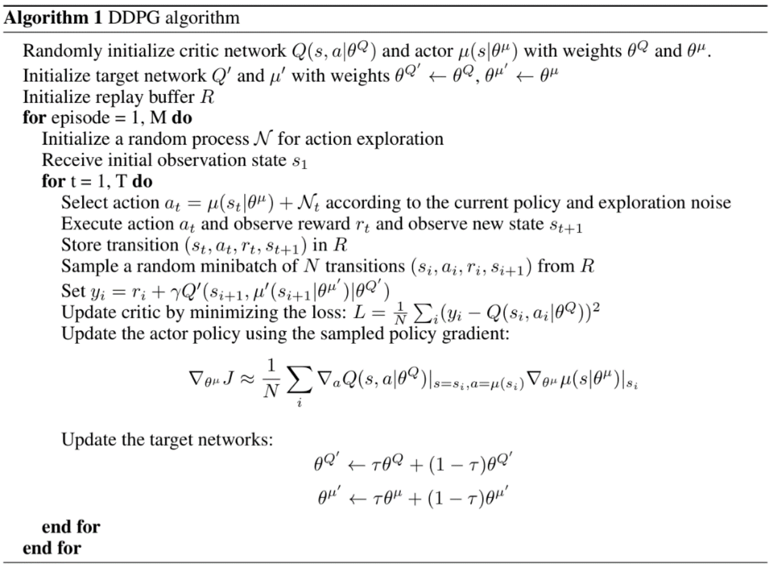
DDPG is an off-policy algorithm because the loss function for the critic is calculated both from the actions sampled from buffer and from actions calculated from the target actor network, and because the behavioral policy consists of a noisy version of the deterministic learned policy.
Due to its off-policy nature, the buffer should be large, allowing the algorithm to benefit from learning across a set of uncorrelated transitions.

It is worth noting that, when learning from low dimensional feature vector observation, the different components of the observation may have different physical units (e.g., position and velocities) and the ranges may vary across the environment. The same holds for images as well, where pixels intensity values can change a lot within the image itself and within images of a batch.
This can reduce the efficiency of the learning, hence, the authors address this issue by adapting batch normalization technique (Ioffe & Szegedy, 2015), vd. Sec. 5.7.

The final issue to be addressed is the problem of ensuring good exploration in continuous action spaces. Being DDPG off-policy, the problem of exploration is treated independently from the learning algorithm. Constructing a new exploration policy by adding noise sampled from a noise process (Ornstein-Uhlenbeck noise) to the learned policy helps increasing the exploration efficiency.

To conclude, this work combines insights from the then recent advances in deep learning and reinforcement learning, resulting in an algorithm that robustly solves challenging problems across a variety of domains with continuous action spaces, even when using raw pixels for observation.

Few limitations to the approach remain. Most notably, as with most model-free reinforcement approaches, DDPG requires many training episodes to find solutions, since the sampling efficiency is still a challenge; furthermore, a fine and very precise tuning of the hyperparameters is required, since the algorithm is very sensitive to them.



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 4: DDPG algorithm.

6 OBTAINED RESULTS

As already explained, the OpenAI Gymnasium environment *Car-Racing* was used for the experiment.
We modified the Python code, to penalize two conditions:

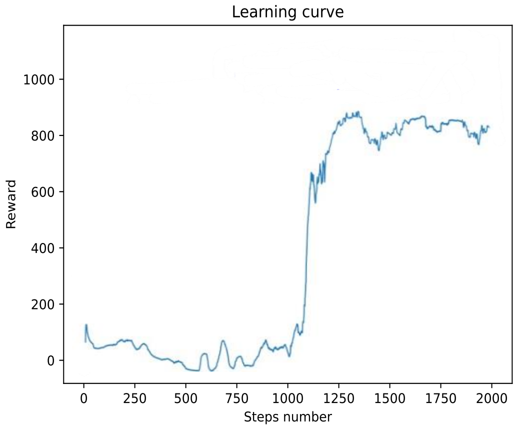
1) car staying still for too much time: a penalty is assigned and the episode is terminated.
2) car driving on grass: a penalty is assigned.

For what it concerns the DDPG algorithm, we exploit the code from a Keras tutorial.
Since the observation from the environment consists of an image, we decided to add some convolutional layers in our code to increase the performance: the attempt of using raw pixels as input revealed to be unsatisfying, when compared to the presence of convolutional layers.
Convolutional neural networks (CNN) are very effective when dealing with bidimensional data, such as images. The scope of CNN is to make the model learn the most important features of the image.

Two main concepts are exploited in CNNs:

1) Convolution: it involves the product between a reduced size window, called kernel/filter and the input image. This process extracts local information such as lines, patterns and textures. The outcome of convolution highlights the relevant regions of the image.
2) Max-pooling: it reduces the size of the image coming from convolution maintaining itmost important features.

CNNs are composed by both convolutional layers and pooling layers until the outcome is a unidimensional vector. Then, other fully connected (or dense) layers may be present to increase the capability of the model.
As we can see from the table summarizing the structures of the models, we didn't use max-pooling layers. Thanks to the presence of batch normalization layers, we didn't have the need for a dropout strategy.
Finally, all the hyperparameters such as learning rate, buffer size, episode duration have been fine-tuned accordingly to our needs. This algorithm proved to be very sensitive to the choice of the hyperparameters. We obtained the following learning curve:



Critic learning rate	0.00095
Actor learning rate	0.00085
γ	0.99
τ	0.005
Cyclic buffer size	50000
Batch size	64

Table 1: model hyperparameters.

Figure 5: plot of the trend of the reward while training.

ACTOR NETWORK (TRAINABLE PARAMETERS: 283026**):**

Layer (type)	Output shape
InputLayer	(96, 96, 1)
Conv2D	(24, 24, 4)
Conv2D	(11, 11, 8)
Conv2D	(4, 4, 12)
Flatten	(192)
BatchNormalization	(192)
Dense	(400)
BatchNormalization	(400)
Activation **(ReLU)**	(400)
Dense	(500)
BatchNormalization	(500)
Activation **(ReLU)**	(500)
Dense	(2)
BatchNormalization	(2)
Activation **(tanh)**	(2)

Table 2: actor network structure.

CRITIC NETWORK (TRAINABLE PARAMETERS: 493033**):**

State		Action	
Layer (type)	**Output Shape**	**Layer (type)**	**Output Shape**
InputLayer	(96, 96, 1)	InputLayer	(2)
Conv2D	(24, 24, 4)		
Conv2D	(11, 11, 8)		
Conv2D	(4, 4, 12)	BatchNormalization	(2)
Flatten	(192)		
BatchNormalization	(192)	Dense	(64)
Dense	(400)		
BatchNormalization	(400)	BatchNormalization	(64)
Activation **(ReLU)**	(400)		
Dense	(500)	Activation **(ReLU)**	(64)
BatchNormalization	(500)		
Activation **(ReLU)**	(500)		
Layer (type)		**Output Shape**	
Concatenate		(564)	
Dense		(256)	
BatchNormalization		(256)	
Dense		(256)	
BatchNormalization		(256)	
Dense		(1)	

Table 3: critic network structure.

With the weights that the model learned in 2000 episodes of training, corresponding to 15 hours, we obtain a good result.

A much longer training can lead to a better behavior in some cases, but training was stopped when the reward was consistently larger than a threshold considered satisfying enough, for the sake of brevity of the experiment. For a demonstration of the behavior of the car after this training check the following site: https://sites.google.com/view/dataanalytics22-23

7 References

[1] Thomas Degris, Marta White, Richard S. Sutton. *Off-Policy Actor-Critic*. 2013.

[2] David Silver, Guy Lever, Nicolas Hees, Thomas Degris, Daan Wierstra, Martin Riedmiller. *Deterministic Policy Gradient Algorithms*. 2014.

[3] Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift. Sergey Ioffe, Christian Azegedy. *Deterministic Policy Gradient Algorithms*. 2015.

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

[5] Mnih V., Kavukcuoglu K., Silver D., Rusu A., Veness J., Bellemare M., Graves A., Riedmiller M., Fidjeland A., Ostrovski G., et al. *Human-level control through deep reinforcement learning.* 2015.

[6] Useful links: Data Analytics website.