AASMA Project - CartPole

Group 07

David Cruz
89377

Matheus Franco
92523

Pedro Almeida
92538

1 - ABSTRACT

CartPole is a game in the Open-AI Gym [1] reinforced learning environment. This game has always been widely used in scientific exploration articles and books to demonstrate the power of machine learning. The game consists of a single agent, the cartpole, and the objective is to keep it balanced while applying forces to a pivot point.

The agent will learn how to act through the reinforcement learning (RL) algorithms: Q-learning, DQN and PPO. In each of them, we reached a solution with a different number of training episodes. PPO was the fastest to reach the model, then it was DQN and, at last, Q-learning.

2 - INTRODUCTION

2.1 Motivation

Taking into account that physics has always been a very fascinating area and part of young people's education and culture, it seemed interesting to bring this subject together with the area of autonomous agents and multi-agent systems. Being our main motivation to deepen our knowledge in agents and machine learning algorithms, we can even better understand the distance needed to correct the angle to its actual amplitude and understand if perfect balance is ever possible to achieve.

2.2 Related Work

There has been a lot of attention directed at solving the problem presented by Cartpole. Some solutions focus on using physics and mathematical knowledge, taking into account the angle of the inverted pendulum, to find a policy to keep it always balanced [9]. Other solutions use machine learning algorithms such as reinforcement learning to solve the problem [10].

2.3 Problem Definition

CartPole, known also as an Inverted Pendulum, is a pendulum with a center of gravity above its pivot point. It's unstable, but can be controlled by moving the pivot point under the center of mass. The CartPole consists of a pole attached by an un-actuated joint to a cart. The pole starts in an upright position with a small perturbation. The goal is to move the cart left and right to keep the pole from falling.

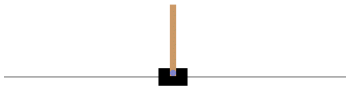

Fig.1: Image of Cartpole [2]

2.4 Objectives

Our main objective is to deepen our knowledge in the field of Autonomous Agents and Multi Agent Systems, while we implement algorithms by which an agent tries to find the best strategy to achieve its goal. To do that, we will evaluate different policies in the choice of actions and analyze the results obtained to understand which can best achieve what is required.

3 - APPROACH

3.1 Environment

The environment consists of a cart which has a pole connected by an un-actuated joint and a track on which the cart will move where there is no friction effect. The pole is placed upright on the cart and the goal is to balance it by applying horizontal forces to the right and left on the cart.

The environment is:

- **Accessible**: the agent has full access to the environment state.
- **Deterministic**: an action has a single guaranteed effect.
- **Static**: the state doesn't change while the agent is deliberating.
- **Continuous**: there is an infinite number of states, but a finite number of actions.

The environment also offers the following methods to enable interaction:

- *reset*: resets the environment to its initial state.
- *step*: receive as input an action and apply it to the environment, then, returns the reward, the next observation, an end flag and extra info.
- *close*: end the game.

3.2 A Single-Agent System

The system has only one agent, CartPole, and its purpose is only to maintain and balance the pole. For it to be done, the agent must try to counteract the fall of the pole, so it is necessary to move the cart to the left when the pole's center of mass starts to fall to the left and move the cart to the right when the pole's center of mass starts to fall to the right.
Therefore, the agent has only two options of actions he can perform, move left or move right.
As the agent complies with its objective, it is rewarded with an increment in reward so that it understands what behavior it should follow.
The agent is reactive, for example the agent's action is influenced by the angle of the inverted pendulum, and is proactive, since he has a goal directed behavior (to maximize its points). The agent also has mobility, autonomy, adaptivity and rationality.

3.3 Architecture

The agent and environment interaction is the traditional one.

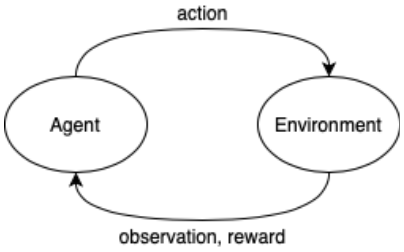

Fig.2: Agent and Environment Interaction

The agent stores an internal state $s_t = x_0, a_1, x_1, \ldots, a_t$ that remembers the previous observations and actions.

The action consists of an array of size 1 that can take on values {0, 1} indicating the direction of the force with which the car is pushed:

- **0**: Push cart to the **left**
- **1**: Push cart to the **right**

The increase and decrease of the cart's speed taking into account the applied force is not fixed, it also depends on the angle of the pole. The center of gravity of the pole varies the amount of energy required to move the cart.

The observation consists of an array of size 4 where each entry has different meanings and can reach the following values:

- 0: **Cart Position** (-4.8, 4.8)
- 1: **Cart Velocity** (-∞, ∞)
- 2: **Pole Angle** (-0.418 rad, 0.418 rad)
- 3: **Pole Angular Velocity** (-∞, ∞)

Although the cart position can take on values between (-4.8, 4.8) the episode ends if the cart leaves the (-2.4, 2.4) range. Although the pole angle can be between (-0.418 rad, 0.418 rad), around ± 24º, the episode ends if the pole angle is not in the range (-0.2095 rad, 0.2095 rad) , around ± 12º.

Since the goal is to keep the pole upright as long as possible while moving the cart in a given space of limited size, the agent receives a reward increment of 1 unit for every timestep where the pole and cart are between the defined angle and position limits, respectively.

In the first state all observations are assigned a uniformly random value in (-0.05, 0.05). The episode terminates if pole angle is greater than ±12° or cart position is greater than ±2.4 or episode length is greater than 500.

3.4 Markov Decision Process

The problem presented in this paper can be modeled as a Markov Decision Process (MDP), a tuple of size 4 (S, A, P_a, R_a). Where:

- **S** is a set of states and is equal to all possible observations.
- **A** is a set of actions and consists of only two actions {0, 1}.
- **P_a(s, s')** is the probability of transitioning to the state *s'* starting from the state *s* and performing the action *a*. In our case, as the environment is deterministic, whenever we execute an action from a given state we can only transit to a single possible state.
- **R_a(s, s')** is the immediate reward received after transitioning from state *s* to state *s'* due to action *a*, generally this reward consists of +1 if the new state complies with the defined angle and position limits, otherwise it is 0.

3.5 Random Agent

The simplest agent possible is the random agent, i.e. it selects a random action at each step. We ran 500 episodes to test this agent. The scores are shown below (fig.3), the highest one was approximately 70 and the mean score was 21,5.

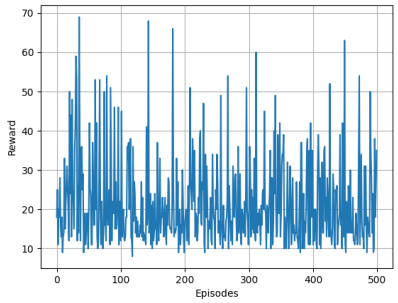

Fig.3: Scores of the Random Agent by episode

3.6 Q-learning Agent

A more complex agent that we implemented uses an off-policy, model free reinforcement learning algorithm called Q-learning. The function $Q: S \times A \rightarrow R$ [5] associates to a pair of state and action an estimated reward for the whole episode, considering an underlying policy π. In other words,

$$Q^{\pi}(s_t, a_t) = E[r_{t+1} + \gamma r_{t+2} + \gamma^2 r_{t+3} + \ldots | s_t, a_t]$$

where γ is the discount and r_t is the reward at time t. The action to be chosen is, therefore, the one which maximizes Q in the given state. The Q matrix is initialized as zeros and then is updated by the Bellman recursive equation [6]

$$Q^{new}(s_t, a_t) = Q(s_t, a_t) + \alpha\,(r_t + \gamma\, max_a Q(s_{t+1}, a) - Q(s_t, a_t))$$

where α is the learning rate. In this algorithm, we made use of the Explore-Exploit dilemma. That is, we choose the Q-best action with probability $1 - \epsilon$ and a random action with probability ϵ. We used an annealing policy for ϵ in which it started as 1.0 and ended as 0.1, decreasing by the equation

$$\epsilon = 1 - log_{10}(\tfrac{ep+1}{25})$$

where ep is the episode number. This allows the algorithm to explore random paths at the beginning and then focus on the Q-best actions. The same was done with the learning rate.

We ran 500 episodes to train the agent and the scores are shown below (fig.4).

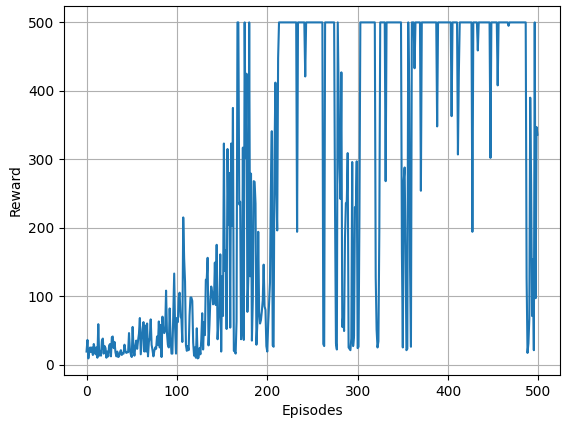

Fig.4: Scores of the Q-learning Agent by episode

Note that the Q-learning algorithm accepts only a discretized observation space and, therefore, we discretized the observation simply by dividing it in bins. We used 3,3,6 and 6 bins (respectively) for each observation feature. More bins were used for the angle's measures since we thought that these informations were more relevant.

3.7 DQN Agent

Now, we dive into a more complex agent that implements a variant of the Q-learning.

The Q-learning approach has some limitations. For example, it becomes very difficult to maintain a Q-function for a very large number of states [4]. Also, Q-learning does not accept continuous states [4]. Furthermore, the correlation between the sequence of observations can represent a bias in the learning process.

Solving these issues and arising from the challenge of playing atari games (observation of a grid of pixels), DQN (Deep Q Learning) was developed [3].

3.7.1 Background on DQN

The goal of the agent is to select actions such that the function $R_t = \sum_{t'=t}^{T} \gamma^{t'-t} r_t$,where T is the time-step in which the game terminates and γ is a discount factor, is maximized.

Instead of maintaining a large table, a neural network can be constructed to compute $Q(s,a)$ given s and a, as the scheme below (fig.5).

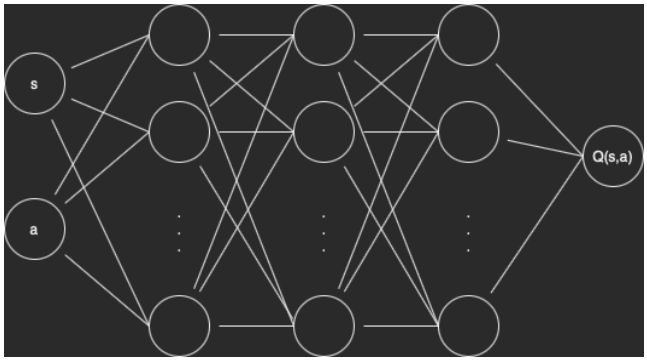

Fig.5: Neural network with inputs s, a and output $Q(s,a)$

But, in such a way, we would have as many forward passes (which takes much computational time) as the number of actions. Another, better, option is to input the state and train it to output $Q(s,a)$ for each action (Deep Q Network), as in the scheme below (fig.6).

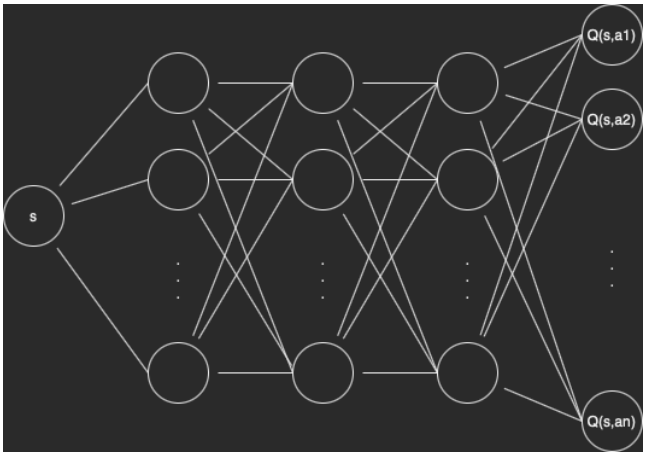

Fig.6: Neural network with inputs s and outputs $Q(s, a_i)$ for every action a_i

To train it, online reinforcement learning can be applied as in Q-learning, i.e. an episode is played and stored in a dataset with the entry $(s_t, a_t, s_{t+1}, r_t, d_t)$ which is later used to improve the neural network by the Bellman recurrence property (as in Q-learning)

$$Q^*(s,a) = E_{s' \sim \varepsilon}[r + \gamma \, max_{a'} Q^*(s', a') | s, a]$$

where Q^* is the optimal Q function and ε is the environment. The neural network loss function [3] is

$$L_i(\theta_i) = E_{s,a \sim p(\cdot)}[(y_i - Q(s,a;\theta_i))^2]$$

where $p(s,a)$ is a probability distribution for state s and action a (behavior distribution), $Q(s,a;\theta_i) \approx Q^*(s,a)$ (i-th iteration approximation), $y = E_{s' \sim \varepsilon}[r + \gamma \, max_{a'} Q(s', a'; \theta_{i-1}) | s, a]$ (the computed reward by the current network). The behavior distribution is often replaced by the ε-greedy strategy.

3.7.2 DQN Algorithm and Preprocessings

If we apply standard online Q-learning, consecutive samples would be used and the result would be inefficient due to the strong correlation between samples. Also, learning on-policy would bias the training distribution and, consequently, the result. The solution is to use Experience Replay [3] which consists in storing the agent's experience, $e_t = (s_t, a_t, r_t, s_{t+1}, d_t)$, at each time-step in a data-set D and picking up random samples for the learning process.
The algorithm of DQN [3] is as follows

Algorithm 1 Deep Q-learning with Experience Replay

Initialize replay memory $\mathcal{D}$ to capacity N
Initialize action-value function Q with random weights
for episode $= 1, M$ **do**
  Initialise sequence $s_1 = \{x_1\}$ and preprocessed sequenced $\phi_1 = \phi(s_1)$
  for $t = 1, T$ **do**
    With probability ϵ select a random action a_t
    otherwise select $a_t = \max_a Q^*(\phi(s_t), a; \theta)$
    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 $\mathcal{D}$
    Sample random minibatch of transitions $(\phi_j, a_j, r_j, \phi_{j+1})$ from $\mathcal{D}$
    Set $y_j = \begin{cases} r_j & \text{for terminal } \phi_{j+1} \\ r_j + \gamma \max_{a'} Q(\phi_{j+1}, a'; \theta) & \text{for non-terminal } \phi_{j+1} \end{cases}$
    Perform a gradient descent step on $(y_j - Q(\phi_j, a_j; \theta))^2$
  end for
end for

Also, a preprocessing of the input is recommended for training efficiency. Note that the current observation doesn't fully translate the actual state. For example, suppose an environment with a ball and the current observation shows that ball in the air. It doesn't give you enough information to know the ball's velocity or in which direction it's going. To solve this, the actual state will store the 4 last observations. Also, when possible, the input pixel image should be rescaled and putted in grayscale for reducing input shape.
Also, frame-skipping technique can be used. I.e. agents sees and selects actions on every k-th frame and its last action is repeated on the skipped frames.

3.7.3 DQN Evaluation

We ran a DQN agent with a different network from the original paper. The network had 2 hidden dense layers with 24 tensors. No convolution layer was used since we are not processing any image. Also, our replay buffer had a capacity of 100000 (10% of the size in the original paper) due to our computer limitations. The results of the experiment are shown below (fig.7).

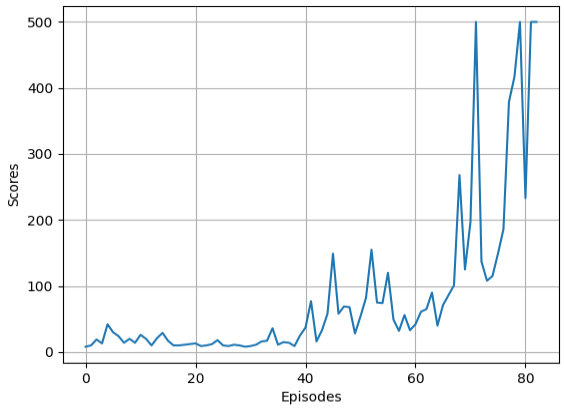

Fig.7: Scores of the DQN Agent by episode

3.8 PPO Agent

The final reinforcement learning agent that we evaluated used the PPO (Proximal Policy Evaluation) algorithm [7]. The main idea of PPO algorithm is to estimate a "safe region" nearby the current policy and depending on that

evaluation we can make fast or small improvements. A key for reaching this goal is using the of the concept of advantage of an action in a state, as defined by

$$A(s,a) = Q(s,a) - V(s)$$

where $V(s)$ is the average of $Q(s,a')$ for all actions $a' \in A$.

Also, to check that "safe region", PPO algorithm maintains two separate policies networks: $\pi_\theta(a|s)$ and an older one $\pi_{\theta_k}(a|s)$ established performing some experience samples. To verify how much the new policy deviates from the older one, the ratio $r(s) = \pi_\theta(a|s)/\pi_{\theta_k}(a|s)$ between policies is estimated and used to calculate the policy update

$$\theta_{k+1} = arg\,max_\theta L_{\theta_k}^{CLIP}(\theta)$$

where [8]

$$L_{\theta_k}^{CLIP}(\theta) = E[\sum_{t=0}^{T}[min(r_t(\theta)\hat{A}_t, clip(r_t(\theta), 1-\epsilon, 1+\epsilon)\hat{A}_t)]]$$

where $\hat{A}_t$ is the estimator of the advantage function and $\epsilon = 0.2$ (as used in the original paper) is a hyperparameter. The algorithm is as shown below [8]

Algorithm 5 PPO with Clipped Objective

Input: initial policy parameters θ_0, clipping threshold ϵ
for $k = 0, 1, 2, \ldots$ **do**
  Collect set of partial trajectories $\mathcal{D}_k$ on policy $\pi_k = \pi(\theta_k)$
  Estimate advantages $\hat{A}_t^{\pi_k}$ using any advantage estimation algorithm
  Compute policy update

$$\theta_{k+1} = \arg\max_\theta \mathcal{L}_{\theta_k}^{CLIP}(\theta)$$

  by taking K steps of minibatch SGD (via Adam), where

$$\mathcal{L}_{\theta_k}^{CLIP}(\theta) = \mathop{\mathrm{E}}_{\tau\sim\pi_k}\left[\sum_{t=0}^{T}\left[\min(r_t(\theta)\hat{A}_t^{\pi_k}, \text{clip}\,(r_t(\theta), 1-\epsilon, 1+\epsilon)\,\hat{A}_t^{\pi_k})\right]\right]$$

end for

We used PPO with clipped objective since it showed better results in the original paper. The scores results through training is shown below (fig.8).

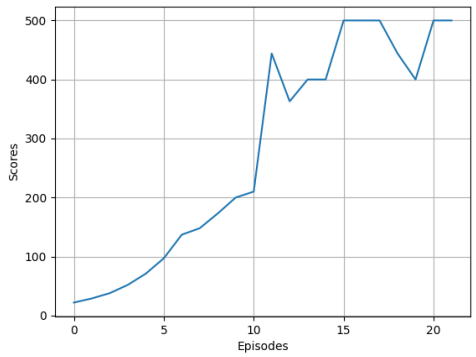

Fig.8: Scores of the PPO Agent by episode

4 - EMPIRICAL EVALUATION

Different agents with different reinforcement learning algorithms were applied to the CartPole problem. We concluded that the random agent isn't a good solution to the problem but both agents with Q-learning, DQN and PPO solved the problem (reached the 500 maximum points). The performances were different though and this can be seen by the number of episodes each of them took to learn a good model. Q-learning stabilized between 400 episodes, DQN around 80 episodes and PPO by 20 episodes. The result found does match what was theoretically expected, i.e. PPO to outperform DQN and DQN to outperform Q-learning.

5 - REFERENCES

[1] https://www.gymlibrary.ml/environments/classic_control/cart_pole/

[2] https://www.gymlibrary.ml/_images/cart_pole.gif

[3] Mnih, V., Kavukcuoglu, K. Silver, D., Graves, A., Antonoglou, I., Wierstra, D., & RIedmiller, M. (2013). Playing Atari with Deep Reinforcement Learning. Retrieved from https://doi.org/10.48550/arXiv.1312.5602.

[4] Mocanu, E., Nguyen, P. & Gibescu, M. (2018). Deep Learning for Power System Data Analysis. Elsevier. Big Data Application in Power Systems. https://doi.org/10.1016/B978-0-12-811968-6.00007-3.

[5] Russel, Stuart J. (Stuart Jonathan). (2010). Artificial Intelligence: A Modern Approach. Upper Saddle River, N.J. :Prentice Hall

[6] Baeldung. (2021) Epsilon-Greedy Q-learning. https://www.baeldung.com/cs/epsilon-greedy-q-learning

[7] Schulman, J. Wolski, F., Dhariwal, P., Radford, A. & Klimov, O. (2017). Proximal Policy Optimization Algorithms. https://doi.org/10.48550/arXiv.1707.06347

[8] Nema, V. (2020). Proximal Policy Optimization. https://aarl-ieee-nitk.github.io/reinforcement-learning./policy-gradient-methods./sampled-learning./optimization/theory/2020/03/25/Proximal-Policy-Optimization.html

[9] Xu, J. (2021, February 21). *How to beat the Cartpole game in 5 lines*. Medium. Retrieved from https://towardsdatascience.com/how-to-beat-the-cartpole-game-in-5-lines-5ab4e738c93f

[10] Surma, G. (2019, November 10). *Cartpole - introduction to reinforcement learning (DQN - deep Q-learning)*. Medium. Retrieved from https://gsurma.medium.com/cartpole-introduction-to-reinforcement-learning-ed0eb5b58288