# Cart Pole

**Francesco Paolo Carmone**
s308126
Computer Engineering in Automation and Cyber-Physical Systems
Politecnico di Torino
`francescopaolo.carmone@studenti.polito.it`

## 1 Linear Quadratic Regulator

### 1.1 Task 1

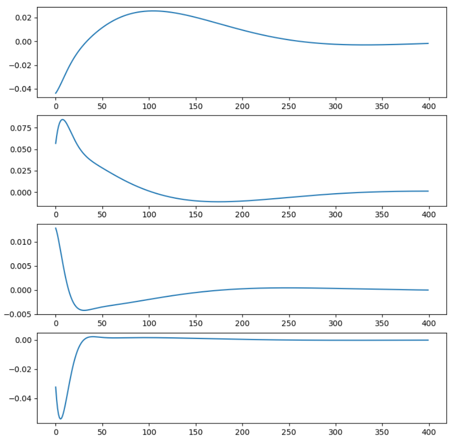


Figure 1: Plots of the state vector $s$ with respect to time. In order from top to bottom: Kart position $x$, Kart velocity $\dot{x}$, Pole angle $\theta$, Pole Angular velocity $\dot{\theta}$

#### 1.1.1 Question 1

*Looking at the plot, after which timestep do all states converge to within* $0 \pm 0.05$*? Explain what it means for all states to converge to zero from a control standpoint.*

By looking at the plot and reading its saved values, I was able to conclude that:

1. The kart position $x$ is always within range
2. The kart velocit $\dot{x}$ converges after $34$ iterations
3. The pole angle $\theta$ is always within range
4. The pole angular velocity $\dot{\theta}$ converges after 7 iterations

Considering the most restrictive of them all, I can conclude that the system converges after the 34-th iteration

## 1.2 Task 2

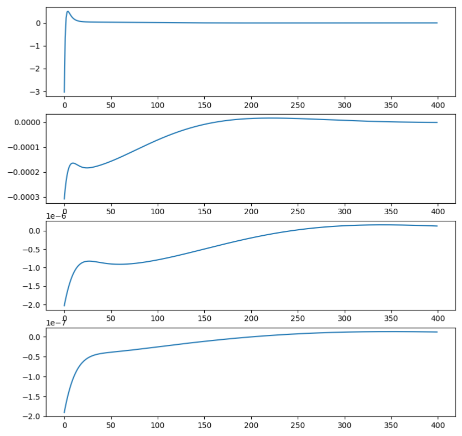


Figure 2: Plots of the kart forces varying R, from top to bottom: $R = 0.01, R = 0.1, R = 10, R = 100$

### 1.2.1 Question 1

*Analyzing the plot, how does the choice of R affect the force? Are they proportional or inversely proportional?*

Lower values of $R$ result in forces with higher magnitudes, while higher values of $R$ result in forces with lower magnitudes. We can conclude that $R$ is inversely proportional to the forces.

We could have reached the same conclusion without running the simulation and just considering the cost function associated to the LQR:

$$\min_u J = \int_0^\infty x^T Q x + u^T R u \quad dx \tag{1}$$

Infact $R$ is the covariance matrix associated to the command effort, and $Q$ is the covariance matrix associated to the state, that governs how fastly the system converges. By increasing $R$, we can notice that $Q$ loses importance, which results in a slower convergence. Such a behaviour is intuitively characterized by lower forces.

## 2 Reinforcement Learning

## 2.1 Task 1

### 2.1.1 Question 1.0.1

*What are the conditions under which the episode ends?*

In the gym version we're currently using, the episode ends when the boolean variable `done` is set to `True` inside the function `env.step()`. That happens when if any of the following conditions are met:

1. Termination: Pole Angle is greater than ±12°
2. Termination: Cart Position is greater than ±2.4 (center of the cart reaches the edge of the display)

3. Truncation: Episode length is greater than 500 (200 for v0)

#### 2.1.2 Question 1.0.2

*How is the reward internally computed?*

The documentation mentions here that *since the goal is to keep the pole upright for as long as possible, a reward of $+1$ for every step taken, including the termination step, is allotted*

The threshold for rewards is $195.0$ for v0. It's possible to access that variable, stored in `env > spec > reward_threshold`, through a debugger.

### 2.2 Task 1.1

#### 2.2.1 Question 1.1

*Evaluate the learned random policy. What is the performance in terms of the average test reward compared to training with the normal policy?*

When the normal policy is on, the model learns from its past iterations, and this is shown by significantly improving its rewards. When the random policy is on, this doesn't happen, leading to the same model being run several times and obtaining comparable results.

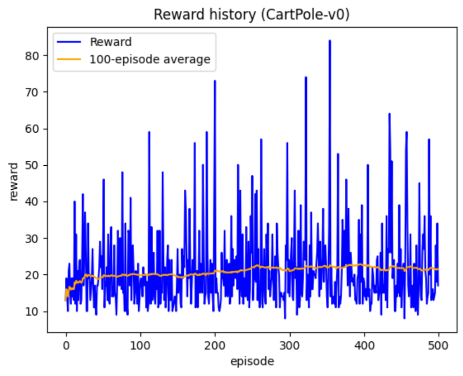


Figure 3: Random policy, the average test is always around 20, meaning the never learns how to fully balance the pole

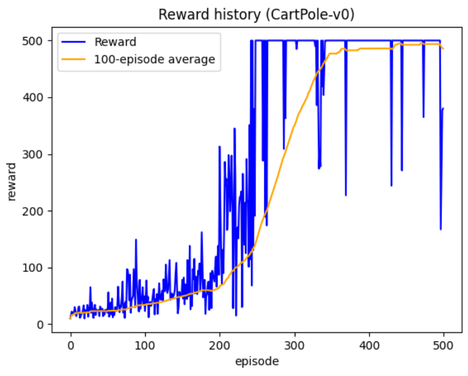


Figure 4: Normal policy, the average test result increases over time until it is maximized (500) in the last few cases, meaning the kart learned how to balance the pole

### 2.3 Task 1.2

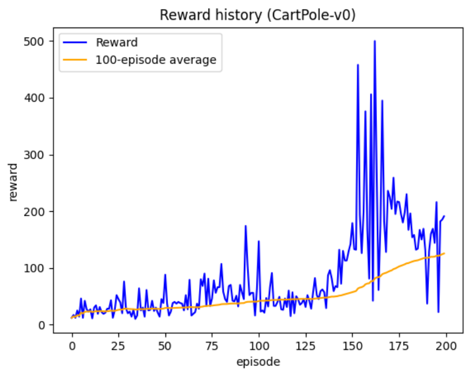


Figure 5: Reward history with 200 timesteps per episode. Notice how the mean is at the end: the kart has still some trouble balancing the pole!

#### 2.3.1 Question 1.2

*Can the same model, trained to balance for 200 timesteps, also balance the pole for 500 timesteps? Briefly justify your answer*

Theoretically yes. Should the algorithm learn the optimal policy within 200 steps, it would definitely be able to balance the pole for 500 steps. However, as shown in Figure **??**, in that particular training experiment the reward at episode 200 was still low and far from 500, which indicates it wouldn't have been able to balance the pole.

### 2.4 Task 1.3

#### 2.4.1 Question 1.3

*Are the behavior and performance of the trained model the same every time? Analyze the causes briefly*

The behaviour and performance of the trained model are very similar from one run to another, but never exactly identical. One must consider the intrinsic stochasticity within algorithm.

## 3 Reinforcement Learning

### 3.1 Repeatability

The graph illustrates training across a maximum of 500 episodes per training iteration. With this restricted episode count, the algorithm faces a limited opportunity to thoroughly explore the solution space and must make the most of the acquired data. The variability in rewards for similar actions arises from the stochastic nature of the environment and the learning process, as the agent lacks perfect knowledge of the true reward function.

Given these considerations, I anticipate that increasing the number of episodes will likely lead to reduced variance in results. However, it's important to note that tuning this parameter will be an iterative process, requiring experimentation and refinement to find the optimal balance

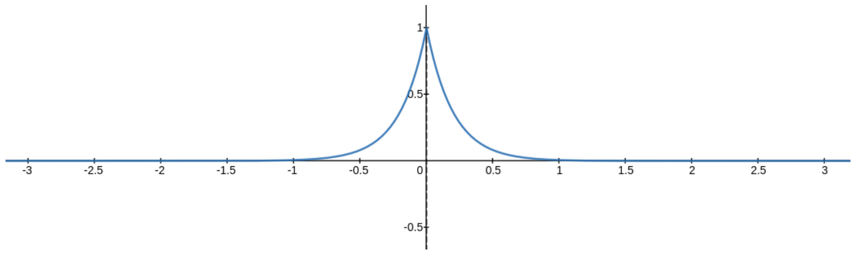
### 3.2 Reward Functions

An interactive version of the first two reward functions can be found in the associated repository's `README.md` file, that can be found by clicking here.

### 3.2.1 Task 3.1

#### First Reward Function

In order to keep the pole in place, the $\theta$ angle must be null, moreover to force it to stay in place, I'm also rewarding positions close to zero.



Where $a$ is a parameter I set to $+5$. Check out the interactive graph here.

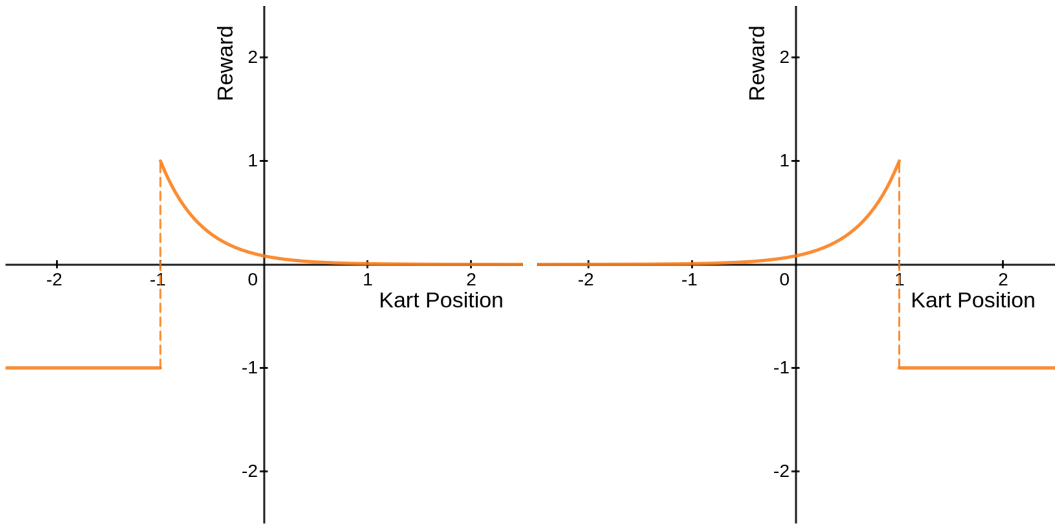
$$r(s) = \exp(-a(|x| + |\theta|)) \tag{2}$$

#### Second Reward Function

Starting from the previous reward function, I added penalties for overreaching the target position. This is, however, practically identical to the previous case.

$$r(s) = \begin{cases} \exp(-a(|x - x_0| + |\theta|) & \text{if } x > x_0 \\ -1 & \text{elsewhere} \end{cases} \tag{3}$$

Where $a$ is a parameter I set to $+5$. Check out the interactive graph here.



(a) $x_0 < 0$ (b) $x_0 > 0$

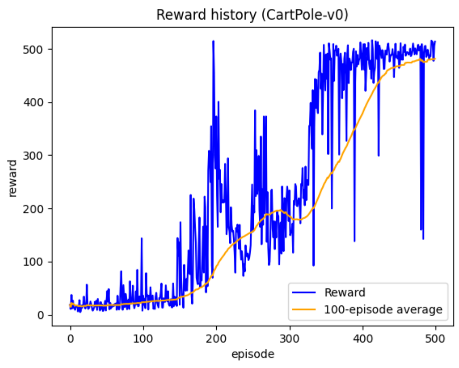
While testing, I noticed two issues:

1. The reward calculation does not consider the relative kart position, i.e. if it was on the left or on the right side of the cartesian plane. I have to give positive rewards only if the state is in the correct half plane.
2. Sometimes not moving gives a better reward than doing anything else

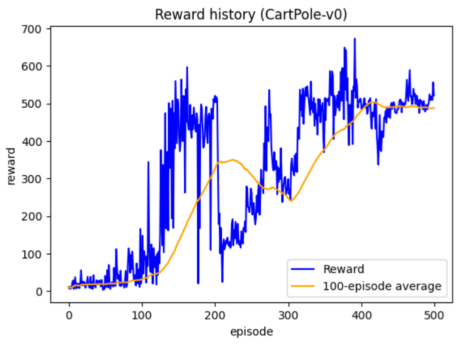
I realized the issue could be the on over-engineering the reward function. Finally, after lots of experiments and tuning weights, this **reward function** gave satisfactory results:

$$1.5\exp(-10|x - x_0|) + 0.75\exp(-|\theta|) \tag{4}$$

I also added a penalties and bonuses respectively for being in the wrong or in the correct side of the screen. The models I trained for this function can be found in the GitHub repository.



(a) $x_0 < 0$



(b) $x_0 > 0$

#### 3.2.2 Reward Function 3

My starting idea was to reward the agents for keeping close to $x_0 = 0$, having an high speed in module, and keeping balance ($\theta = 0$). After countless failed attempts, all of which revolved around increasing the episode length and changing the weights of a reward function like this

$$a\exp(-\alpha|x|) + b\exp(-\beta/|\dot{x}|) + c\exp(-\gamma|\theta|) \tag{5}$$

I ended up with a satisfactory result by choosing the following weights

$$a = 1, b = 0.25, c = 2.25, \alpha = -8, \beta = -1, \gamma = -1$$

Unfortunately, I didn't save the model nor the reward history, and have been unable to replicate the model despite my best effort. An animation can be found here.

My following attempts revolved around giving negative rewards for actions I was trying to avoid, mainly going off screen and getting stuck in the middle.

To solve that issue, I added a negative reward for being slow and close to zero. That fix led to an awkward result, in which the kart started giggling sighlty further away from that, as it's possible to see here.

Another solution that crossed my mind was to give a reward proportional to the walk lenght, but I'm sure it would have ended up.

My last reward function has been

$$1.5\exp(-|1/\dot{x}|) + \exp(-|x|) + 0.01 \tag{6}$$

Where I didn't give any reward for stabilizing the pole, and gave negative one for leaving the screen. The animation can be found here.

## 4 Extras

### 4.1 Question 1

*Discuss briefly the advantages of using an LQR controller or an RL agent to solve a general task of controlling a robotic system. Analyze the differences and similarities between the two methods*

LQR controllers follows a deterministic, model-based optimization approach, and provide an analytical solution for known linear systems that can be modeled with quadratic cost functions, it leds to controllers that compute an optimal input signal $u(t)$ quickly making them suitable for real-time applications

It has a few disadvantages: the system has to be linear, and its dynamics known. This is a very restrictive assumption, as almost all real case scenarios in robot applications are nonlinear. Obtaining an analytical solution to the Ricatti equation is hard besides obvious cases, and thesolution to a particular LQR problem is obtained under the implicit assumption that the desired final state is reachable from the given initial state.

RL Agents embrace a trial-and-error, model-free learning approach, since they learn from experience, and explore the environment to discover optimal policies. This allows the agents to deal with any kind of system, including nonlinear ones, at the expense of computational power and lots of time. Moreover, as I've found out from this simple scenario, finding a working reward function is really challenging.

The LQR and RL are different tools used in different contexts. An RL agent can learn to do what an LQR can do, but not viceversa, but not necessairly better. On the example studied on this paper, and similar simple simple systems, using a controller over an AI algorithm is definitely a better choice.