Robot Learning - Laboratory 1: Comparison between classic control algorithms and reinforcement learning

Andrea Delli (S331998)
Robot Learning course (01HFNOV)
Master's Degree in Computer Engineering
Artificial Intelligence and Data Analytics
Politecnico di Torino

Abstract: The purpose of this laboratory is to explore and compare classical control and reinforcement learning (RL) approaches in the CartPole environment, a standard benchmark for control and decision-making tasks. Using the Linear Quadratic Regulator (LQR) method, we examine the effects of varying control parameters on stability and convergence. Then, we implement RL to train agents using default, random and custom reward functions to guide specific behaviors. This study investigates agent performance under different conditions, evaluates the impact of stochastic training processes, and highlights trade-offs between the predictability of classical control and the adaptability of RL.

1 Introduction

The Cart-Pole environment is a fundamental and widely-used benchmark in reinforcement learning (RL), designed to test control strategies and decision-making algorithms. The setup consists of a cart that can move along a one-dimensional track and a pole mounted on the cart, free to pivot around its attachment point. The primary goal of the task is to balance the pole upright by applying forces to the cart to move it left or right.

Key components of the environment include a state observation vector s containing four elements: the cart's position x and velocity $\dot{x}$, the pole's angle θ relative to the vertical axis, and the pole's angular velocity $\dot{\theta}$. The agent's objective is to maintain the pole in a vertical position while keeping the cart within the track's boundaries. Any significant deviation, such as the pole falling or the cart moving out of bounds, results in a failure.

$$s = \begin{bmatrix} x \\ \dot{x} \\ \theta \\ \dot{\theta} \end{bmatrix}$$

2 Linear Quadratic Regulator (LQR)

The LQR provides a foundational control approach for stabilizing the Cart-Pole system. By approximating the system dynamics linearly around the equilibrium point, the LQR computes an optimal control gain K that minimizes a quadratic cost function J. This cost function is parameterized by weight matrices Q and R. The Q matrix represents the gain of the cost function on the states of the system, while the R matrix represents the gain of the cost function on the input to the system. Therefore, Q and R define what we consider as optimal.

$$J = \int_0^{\infty} \left(\mathbf{x}^T Q \mathbf{x} + \mathbf{u}^T R \mathbf{u} \right) dt$$

In the initial experiments, we observe the evolution of system states over time. An analysis of the state trajectories (see Fig. 1) shows the convergence behavior, revealing the effectiveness of the LQR control strategy in stabilizing the pole.

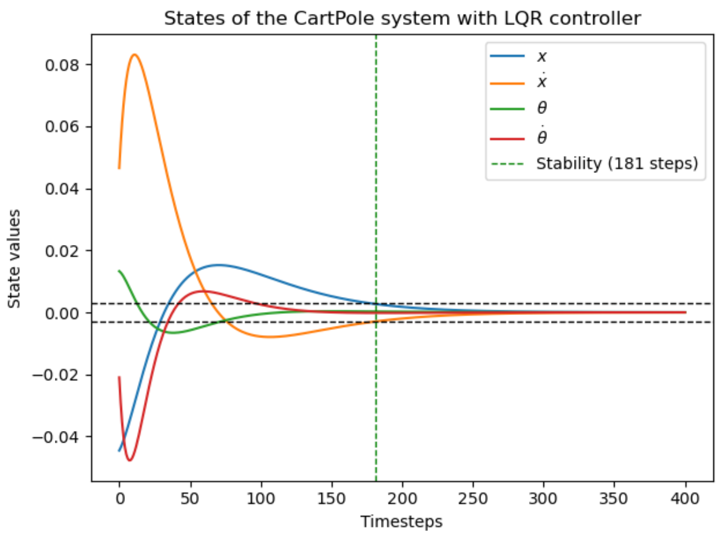


Figure 1: Task 1 - States of the system over the first 400 timesteps

By observing the plot in Fig. 2, all the state values converge in the range 0 ± 0.003 after the timestamp 181, confirming system's ability to reach steady-state behavior under LQR control.

By varying R we analyze how the control force evolves as this parameter changes. The plot in Fig. 2 illustrate how the applied force influences convergence, by weighting the control vector u differently.

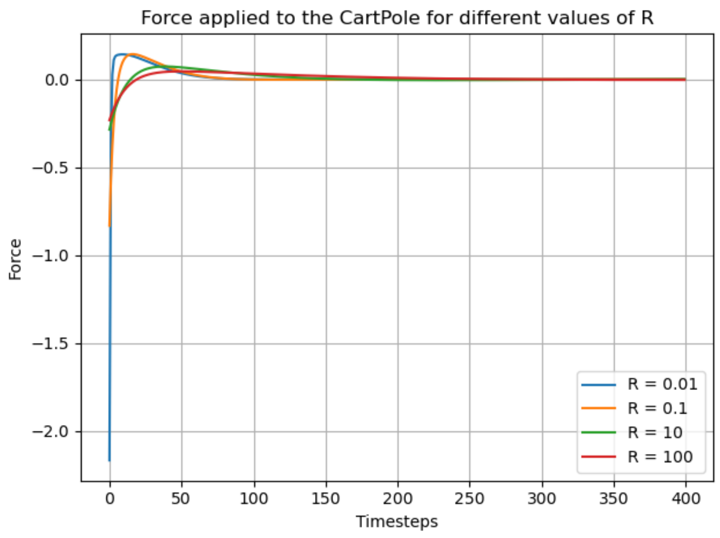


Figure 2: Task 2 - Forces applied to the system for different values of R

The plot in 2 shows the inverted proportionality between the control weight R and the initial force applied: for larger values of R, the applied force is lower (because J, which we want to minimize, increases), and vice-versa for smaller values. The higher force is obtained using the smallest R value (0.01) and has an absolute magnitude of $\simeq 0.23$.

Another important effect of changing R is the variation of the **overshoot**, which is defined as the maximum peak value of the response curve measured from the desired response of the system [1]. The overshoot is also inversely proportional to the value of R: higher values lead to lower overshoot, while lower values lead to higher overshoot.

3 Reinforcement Learning (RL)

Reinforcement learning offers a data-driven approach to solving the Cart-Pole balancing problem, relying on an agent that learns optimal actions through trial-and-error interactions with the environment. The training process employs a reward-based feedback mechanism, where the agent refines its policy to maximize cumulative rewards.

3.1 Default policy and Random policy

As reported in official documentation [2], the episode ends if any one of the following occurs:

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 200

The documentation also reports the default reward function: 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.

Training with the default reward function demonstrate the agent's ability to learn effective balancing strategies. Training and testing results reveal how the agent's policy evolves and stabilizes over time, as shown in Fig. 3.

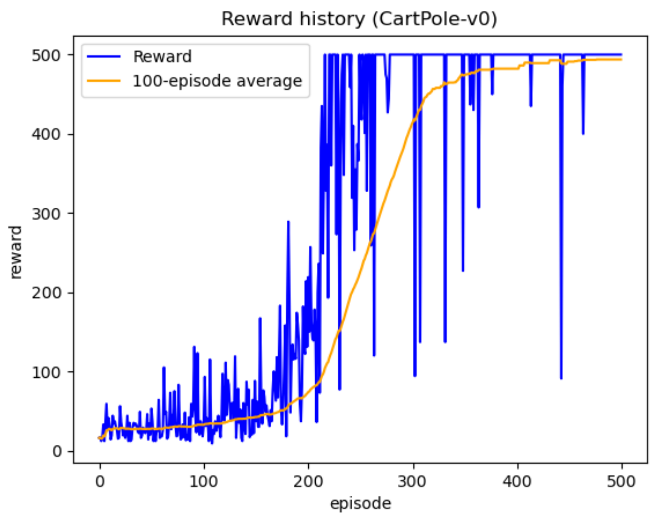


Figure 3: Reward history during training with default policy

If we consider a random policy (by sampling a random action from within the action space and performing it), the model doesn't learn anything and just keeps doing random actions, as shown in Fig. 4, and obviously has a worse performance than the default policy. The default policy has an average reward of 500, while the random policy has an average reward of 84.

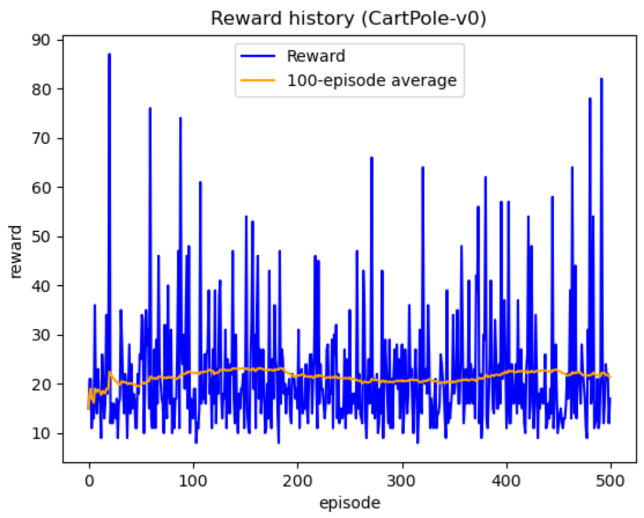


Figure 4: Task 1.1 - Reward history with a random policy

3.2 Generalization and Repeatability

Another test we performed is to train the same model with default policy with episodes of 200 timesteps, and test it to balance the pole for 500 timesteps. The resulting training reward history is shown in Fig. 5, and shows that for an episode length of 200 the model learns to balance the pole correctly. By testing it on episodes with length of 500 timesteps, it's visibly clear that the model has learned a valid policy to balance the pole, both from the test rendering and from the average reward, which is 500.

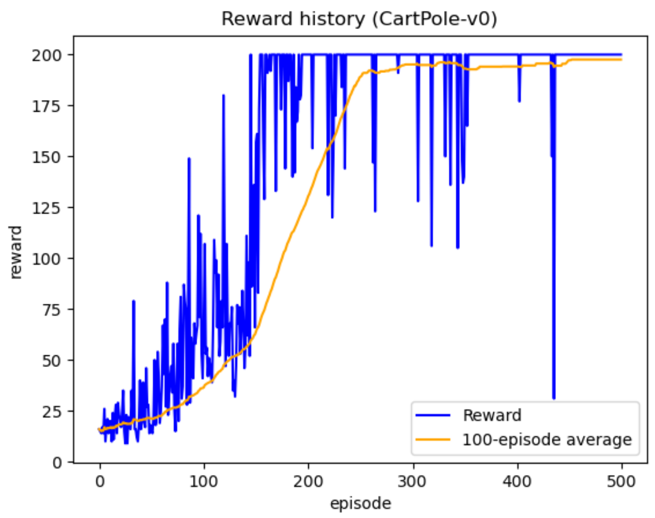


Figure 5: Task 1.2 - Reward history with default policy and 200 timesteps per episode

By training the same model multiple times and then testing it, it's clear that the performance isn't always the same. Some trained models achieve a good generalization and are able to make the pole stand, while in other cases the average reward is lower and the agent is capable of keeping the pole straight only for a shorter time. This is due to the stochasticity of the training process, which involves choosing an action to perform favoring exploration.

This occurs because training the model is a stochastic process and can lead to have different performance, even if the training is the same. In particular, in this case the randomness in training can be found in the action selection: `action, action_probabilities = agent.get_action(observation)`, which selects an action using a categorical distribution.

This leads to a variation of model performance, and implies a challenge in comparing different RL models, which performance might be "good by chance" and not because the model is actually good, or vice-versa (badly performing model, because the training process was unlucky).

The result of multiple training is shown in Fig. 6, where the variance of the reward between multiple runs is clearly visible.

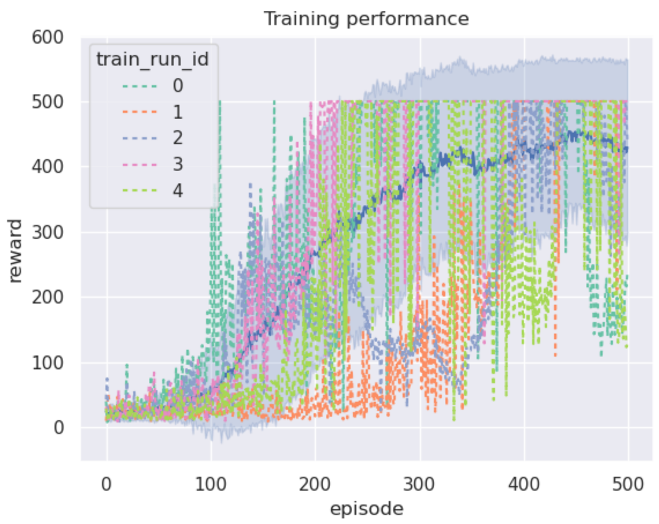


Figure 6: Variance in performance between multiple trainings of the default model

3.3 Reward functions

This section involves testing alternative reward functions to guide the agent toward specific behaviors, such as balancing at the center of the track or achieving high-speed movement while maintaining stability.

All models in this section have been trained for 1000 episodes, each with a duration of 500 timesteps.

Implemented reward functions goal:

1. Balance the pole close to the center of the screen (close to $x_0 = 0$)
2. Balance the pole in an arbitrary point of the screen ($x = x_0$)
3. Keep the cart moving from the leftmost to rightmost side of the track as fast as possible, while still balancing the pole. The minimum speed of the agent should be high enough that the agent is visibly moving from one side of the other (not just jittering in the center) and changes direction at least once and not just drifting in a random direction.

For the **first two reward functions** (stability in $x = 0$ and $x = x_0$) the same implementation has been used, because the default value of the parameter is $x_0 = 0$. The Algorithm 1 has the following structure:

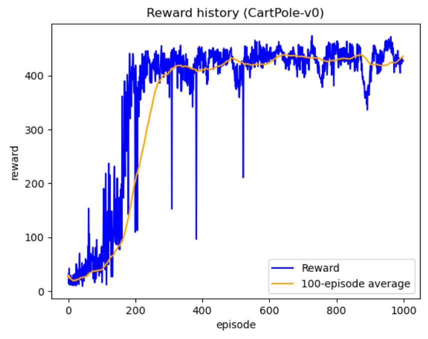
- `+1` for keeping the pole alive
- `-0.5 * abs(pos-x0)` as penalty to encourage the x_0 position

- `-abs(angle) - 0.1*abs(pos_velocity)` to penalize pole angles and movement, keeping the pole straight

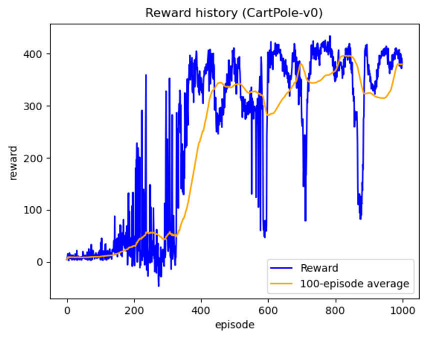
Algorithm 1 Reward function to keep the cart in a given position $x = x_0$

```
def reward_stable_in_position(state, x0):
    # Balance the pole close to the given point x0
    # The closer the pole is to x0, the higher the reward
    pos, pos_velocity, angle, angle_velocity = state
    return 1 - 0.5*abs(pos-x0) -abs(angle) - 0.1*abs(pos_velocity)
```

The reward history of the training using this reward function is presented in the plots in Fig. 7, with $x = 0$ on the left (Fig. 7a) and $x = -1$ on the right (Fig. 7b).



(a) Reward history with goal to center in $x_0 = 0$



(b) Reward history with goal to center in $x_0 = -1$

Figure 7: Reward history with policy to center the cart in $x = x_0$, given as argument, using reward function 1

The training has been done with 1000 episodes of 500 timesteps. In both cases the reward plot converges near an optimum value, as represented in the plateau in the plots. The learning in the second case converges more slowly, finding a good policy only after episode 800, while for $x = 0$ a good policy is found even after 400 episodes, due to the fact that the initial condition of the CartPole is always $x_{start} = 0$.

The final models however show a similar performance in terms of average reward ($\simeq$ 333, with 500 timesteps for each episode test, with the average episode length of 500, that is the maximum obtainable, meaning that the cart is stable).

For the **third reward function** (CartPole oscillation) two different implementations are reported (see Algorithm 2 and Algorithm 3). The first one is more complex, while the second one is pretty straightforward but still achieves a noticeable behavior.

In particular, the reward function 2 receives the state of the previous timestep (`previous_state`) and the current state (`state`), and is structured as follows:

- `-abs(angle)` to penalize large angles, doubled if the angle is above 0.1
- `+1` to encourage direction change, calculated as a change in the velocity sign between the previous and the current state. If there's no direction change, $+0.1$ is given
- `+abs(position_velocity)` to encourage high speed
- `-3*(abs(position) - 2.0)` to penalize being close to the environment limits (± 2.4), given only if $|x| > 2.0$

The weights of each of the terms have been chosen by running the training multiple times and visualizing the result to fine-tune their value.

Algorithm 2 Reward function to keep the cart oscillating

```
def reward_oscillation(previous_state, state):
    """
    Reward function to encourage oscillation while keeping the pole balanced.

    Args:
    previous_state: tuple with previous (position, velocity, angle, angle_velocity)
    state: tuple with current (position, velocity, angle, angle_velocity)

    Returns: reward: A scalar reward.
    """

    _, prev_position_velocity, _, _ = previous_state
    position, position_velocity, angle, angle_velocity = state

    angle_penalty = abs(angle) # Penalize large pole angles
    if abs(angle) > 0.1:
        angle_penalty *= 2  # Apply a stronger penalty for large deviations from balance

    # Reward for oscillation: Encourage movement from left to right with enough speed
    if ((prev_position_velocity > 0 and position_velocity < 0) or
        (prev_position_velocity < 0 and position_velocity > 0)):
        direction_change_reward = 1.0  # Reward for changing direction
    else:
        direction_change_reward = 0.1

    speed_reward = abs(position_velocity) # Speed reward: The faster the better

    edge_penalty = 0 # Penalize being too close to the edge (-2.4, 2.4).
    if abs(position) > 2.0:  # Penalize if position is within 0.4 units of the edge
        edge_penalty = 3.0 * (abs(position) - 2.0) # Penalize being closer to the edge

    return -angle_penalty + direction_change_reward + speed_reward - edge_penalty
```

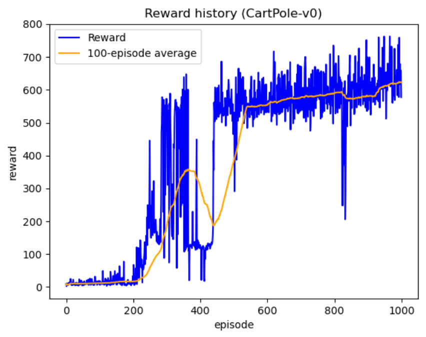
While the reward function 3 is simpler, and gives a default value of +1 for keeping the Cart alive, and +`abs(velocity)` to encourage higher speed.

Algorithm 3 Simplified reward function to keep the cart oscillate (also called "v2")

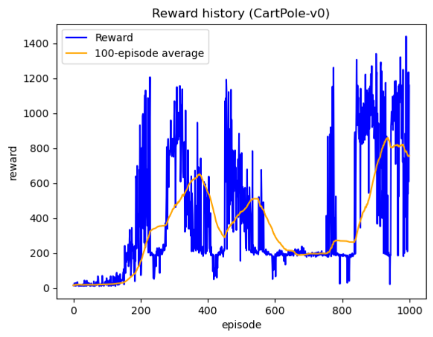
```
def reward_oscillation_v2(state):
    """
    reward function to encourage oscillation while keeping the pole balanced.
    args: state: tuple with current (position, velocity, angle, angle_velocity)
    returns: reward: a scalar reward.
    """
    return 1 + abs(state[1])    # reward for high speed
```

The plots in Fig. 8 show the reward history during the 1000 training episodes. On the left (Fig. 8a) we see the reward history of the more complex reward function, which is more clean and shows a convergence towards an optimal policy, while in the right plot (Fig. 8b) we can notice multiple spikes, representing an higher difficulty for the model to understand a good policy.

The two plots needs to be compared only in their shape, because the magnitude of the reward function result is intrinsically different. This is due to the fact that the functions try to incentivize or penalize different behaviors of the agent, using the weights accordingly.



(a) Reward history with goal to oscillate, using Algorithm 2



(b) Reward history with goal to oscillate, using Algorithm 3

Figure 8: Reward history with goal to make the CartPole oscillate along the environment

Even though the second reward implementation is very simple, the CartPole learns to have the proper behavior because it learns to stay alive for the entire duration of the episode (because of the `+1`), and to keep moving with sufficient speed (because of `+abs(speed)`).

The more complex reward function however has a better performance and makes the CartPole reach a higher velocity when compared with the simpler one, as shown in Table 1. The downside of that model is that after a few oscillations it exits the environment, making the episodes last less timesteps on average.

Model	**Max speed reached**	**Average episode length (max: 1000)**
Reward oscillation 2	3.2691169	431.75
Reward oscillation 3	2.0707452	680.73

Table 1: Performance of different oscillation models trained, in terms of maximum speed reached and average episode length (both evaluated at test time)

A visualization of the third reward function training performance can be visualized from the plot in Fig. 9, where the oscillation model with reward function 2 has been trained 10 times. The plot clearly shows the variance in the reward function during the training process, and it's possible to notice that in some cases the reward never increases to a decent value, showing that the training process could completely fail.

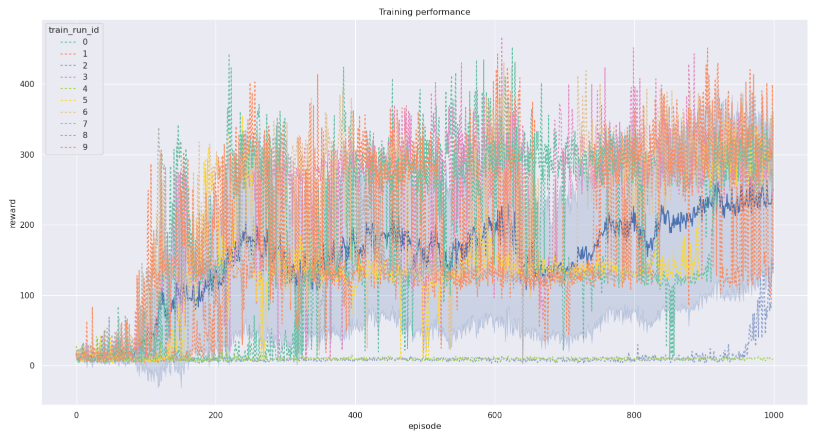


Figure 9: Reward variance between 10 trainings of the oscillation model with reward function 2

4 Conclusion

The experiments on the CartPole environment demonstrated the effectiveness of both classical control methods, such as LQR, and data-driven approaches like reinforcement learning (RL). The LQR provided a stable solution for balancing the pole, with predictable performance influenced by the weighting matrices Q and R. Variations in these weights showed the trade-off between control effort and system stability, highlighting its utility in deterministic control settings.

In contrast, reinforcement learning introduced flexibility by allowing agents to learn directly from interaction with the environment. The default reward function enabled the agent to achieve proficient balancing, but the training process exhibited stochastic behavior, leading to variability in outcomes. Modifying reward functions allowed tailoring of agent behavior, such as centering the cart or promoting oscillatory motion. However, achieving reliable performance required careful reward design and often multiple training iterations due to the inherent randomness in RL training.

Ultimately, while LQR offers a mathematically rigorous and predictable solution, RL's adaptability, flexibility and generalization makes it a powerful tool for more complex tasks. These results highlight the importance of choosing the appropriate approach based on the task requirements and constraints.

References

[1] Wikipedia. Overshoot. URL `https://en.wikipedia.org/wiki/Overshoot_(signal)`.

[2] Gym. Cartpole documentation. URL `https://www.gymlibrary.dev/environments/classic_control/cart_pole/`.