Report Laboratory 03 - Robot Learning

Federico Pulinas
s319881
Computer Engineering in Automation and Intelligent Cyber-Physical Systems
Polytechnic of Turin, Italy
`federicopulinas.pvt@gmail.com`

Abstract: The purpose of this exercise is to implement a simple reinforcement learning pipeline to solve the cartpole environment. In particular, it considers a tabular version of the off-policy, temporal-difference methods, the Q-learning, to learn how to control the Cart-Pole system.

Keywords: Cart-Pole, Reinforcement Learning, Q-Learning

Introduction

Task 3.1 - Behavious Policy: ϵ-greedy policy vs fixed epsilon policy

introduction

This task investigates the implementation of the Q-learning algorithm for the CartPole environment, focusing on addressing the exploration-exploitation trade-off during training. To ensure effective learning, an epsilon-greedy strategy is deployed to balance exploration (gathering new information about the environment) and exploitation (utilizing the current knowledge to maximize rewards). This strategy is examined in two configurations:

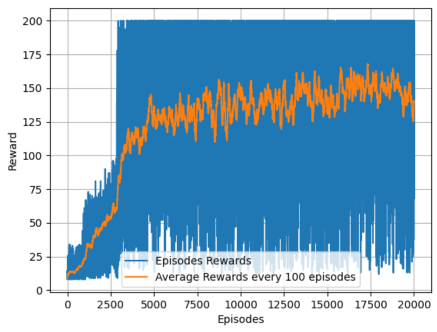
- using a constant epsilon value, $\epsilon = 0.2$, where the trade-off is fixed
- using a decaying epsilon value based on the GLIE (*greedy in limit with infinite exploration*), where exploration decreases over time, and the policy converges to a greedy strategy in the limit.
  The implementation of the GLIE schedule has been based on the following formula
  $$\epsilon_k = \frac{b}{b+k} \tag{1}$$
  tuning b such that ϵ reaches 0.1 after $k = 20000$ episodes.

To tune b to the nearest integer means to choose $b = 2223$ based on (1).

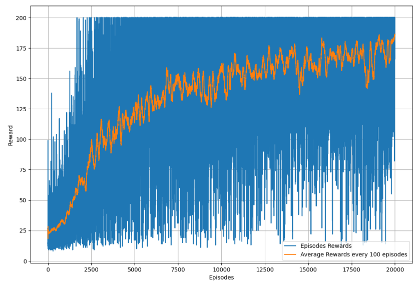
$$\epsilon_k = \frac{b}{b+k} \Rightarrow \ b = \left.\frac{k}{\frac{1}{\epsilon_k} - 1}\right|_{k=20000} \approxeq 2223 \tag{2}$$

Results Evaluation

To evaluate the performance difference between the fixed epsilon and greedy epsilon strategies, we analyzed the return achieved during each training episode. To emphasize the overall progress of the algorithm rather than focusing on the variability of individual episodes, we incorporated a smoothed trend line by averaging the rewards over every 100 episodes. Additionally, to gain deeper insights into the exploration-exploitation trade-off, we created a complementary plot that tracks the actions taken throughout the training. In this plot, each point represents a counter of random or greedy (depending on the curve) actions performed during a single episode.



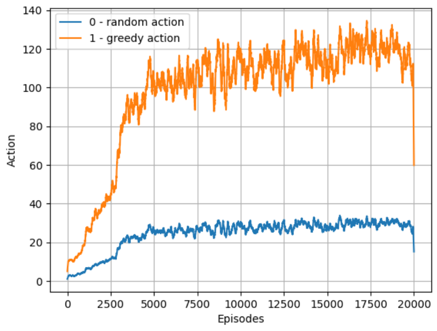
(a) Fixed ϵ - return obtained at each training episode



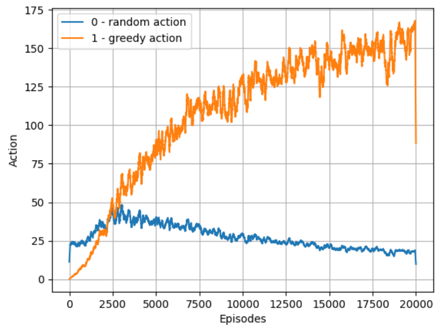
(b) Greedy ϵ - return obtained at each training episode

Figure 1

Both the approaches show steady improvement in average rewards, but the GLIE schedule achieves slightly higher rewards in the long run, due to its decreasing exploration rate. While the fixed epsilon case maintains some degree of suboptimality because it continues to explore with a constant probability, even after identifying near-optimal policies. From the exploration-exploitation balance point of view we can see that the fixed epsilon maintains it constant throughout the training, which is beneficial for preventing premature convergence but less efficient for fine-tuning optimal behavior. On the opposite side, the GLIE dynamically adjusts the balance, allowing for robust exploration early on and focuses exploitation later, resulting in better long-term performance stability.



(a) Fixed ϵ - counter of actions performed during training averaged every 100 episodes



(b) Greedy ϵ - counter of actions performed during training averaged every 100 episodes

Figure 2

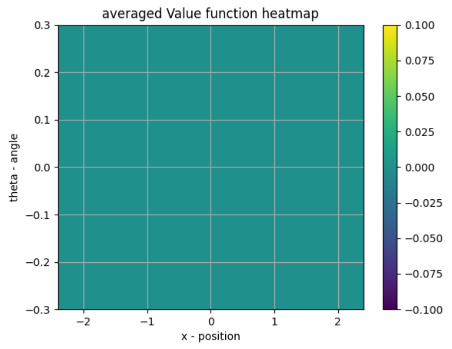
If we take a look at the plots containing the counter of the actions, in the fixed epsilon case the exploration-exploitation proportions are visible, i.e. 20% exploration (random actions) and 80% exploitation (greedy actions). Even for the GLIE schedule we can have a clear display of its behaviour. From an efficiency point of view we can state that the GLIE schedule aligns better with the task, ensuring better convergence to optimal rewards over time. This is given by the growth of greedy actions, indicating a clearer exploitation phase after sufficient exploration.

Task 3.2

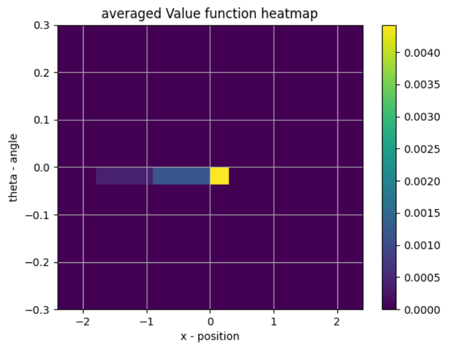
Having proved that with a variable ϵ the learned Q function should converge to an optimal action-value function q_*, we can exploit to recover the optimal state-value function ν_* through the following expression:

$$\nu_*(s) = \max_{a \in A(s)} q_*(s,a) \tag{3}$$

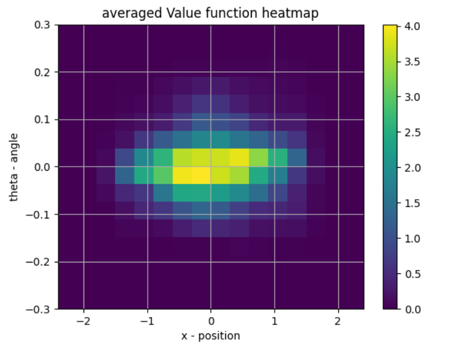
It is then possible to plot the heatmap of the computed state-value function in terms of x and θ, averaged over $\dot{x}$ and $\dot{\theta}$.



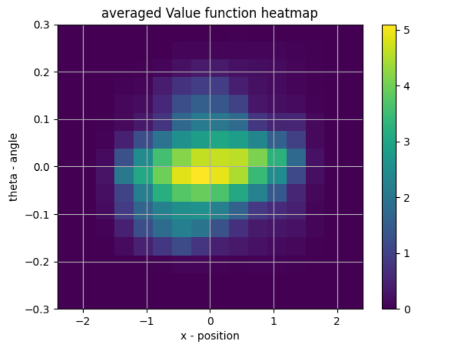
(a) before the training



(b) after 1 episode



(c) after 10000 episodes



(d) after 20000 episodes

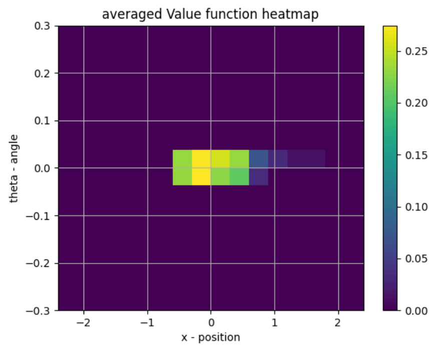
Figure 3: Heatmap during training

In Figure 3a, we observe the heatmap of $Q(S, A)$ values before training. At this stage, all values remain at their initialized levels, results monochromatic. After the first iteration, depicted in Figure 3b, the values start to update. Given that only limited data has been sampled, we notice just one point on the map turning yellow. This point represents the combination of cart position and pole angle that yields the highest return.

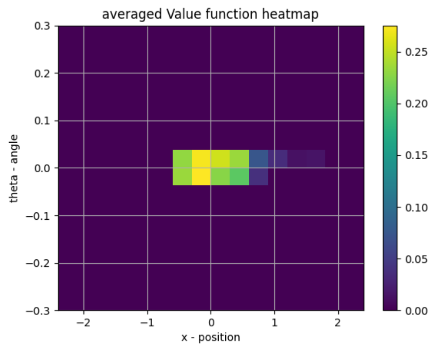
As training progresses, two additional plots emerge: Figure 3c, showing the heatmap halfway through the training episodes, and Figure 3d, illustrating the values at the end of the training. Comparing Figure 3b and Figure 3c, we observe that the area with high-value regions broadens significantly, indicating that the system has explored a wider range of actions. However, the comparison between Figure 3c and Figure 3d reveals a shift in focus. Instead of further broadening the high-value region, the system concentrates on exploiting known information to maximize the value function around previously identified optimal points.

Task 3.3

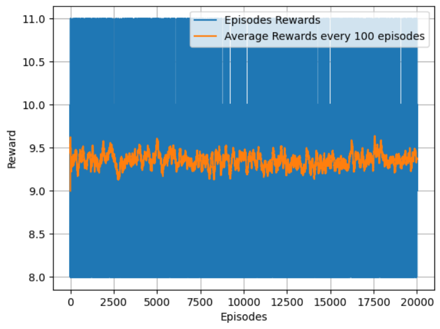
The previous tasks were performed considering the initial Q-function initialized at zero. By changing the initial values of the Q-function we can obtain new results. Having set ϵ to 0 and considering a greedy policy w.r.t. the current Q-function estimate, we are going to study two different cases, initialize Q-function values at zero and at 50 for all states and actions.



(a) Heatmap after half of the training episodes



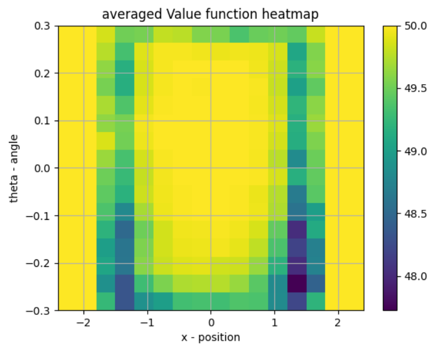
(b) Heatmap after half of the training episodes



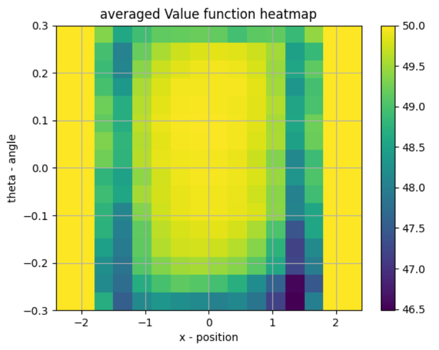
(c) Greedy ϵ - return obtained at each training episode

Figure 4: Initial Q set to zero

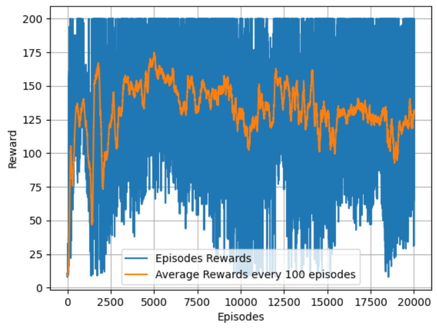
In the case where the Q-values are initialized to 0, the agent starts with no preference for any specific action, meaning all actions are considered equally valuable. This leads to the agent relying entirely on exploration to discover new possible actions. Despite following a greedy policy, the agent cannot explore the environment due to $\epsilon = 0$. As a result, it fails to discover better actions, leading to poor performance and low returns, which is the opposite of what we would expect from a greedy policy. Without exploration, the system is stuck and cannot improve its behavior.



(a) Heatmap after half of the training episodes



(b) Heatmap after half of the training episodes



(c) Greedy ϵ - return obtained at each training episode

Figure 5: Initial Q set to 50

In contrast, when the Q-values are initialized to 50, we introduce a bias that encourages the system to explore, even though $\epsilon = 0$. This bias leads to some initial exploration of the environment. As a result, the system performs better compared to the case where Q-values are initialized to 0, but its performance still falls short relative to the results from the first task. The noticeable difference in the heatmap is due to the initial conditions, which unintentionally promote exploration of the environment. This exploration arises as a natural consequence of the biased initial Q-values, while the greedy policy, has no freedom to explore as it attempts to optimize its actions.

Can Q-learning be used in environments with continuous state space?

Q-Learning can be used in environments with continuous state space with the use of function approximation, since the state space is too large to store Q-values in memory for every state-action pair explicitly.
Instead of storing Q-values for each individual state-action pair, Q-values are approximated using a parameterized function, typically a neural network or linear function. This function learns a mapping from state-action pairs to Q-values, allowing generalization over continuous states and actions. The function approximator is trained to estimate Q-values for new state-action pairs that were not explicitly visited during training.
This adaptation allows Q-Learning to handle continuous state spaces effectively, but it also introduces challenges. The most notable difficulty is maintaining stability during training. Since Q-Learning is off-policy, the target values used to update the function approximator depend on the same function being trained, which can lead to instability or even divergence. Moreover, the function approximator introduces approximation errors that can accumulate over time, potentially degrading the quality of the learned policy.

Can Q-learning be used in environments with continuous action spaces? If any, which step of the algorithm would be problematic to compute in the continuous action space case?

Q-Learning can be used in environments with continuous action spaces, but it becomes more challenging because the algorithm was originally designed for discrete action spaces. The key issue lies in the computation of the maximization step, i.e. the selection of the optimal action, in the update rule.

$$Q(s,a) \leftarrow Q(S,A) + \alpha[r + \gamma \max_{a'} Q(s',a') - Q(s,a)] \tag{4}$$

In a continuous action space, the set of possible actions is infinite, making this maximization step computationally infeasible. the term $\max_{a'} Q(s',a')$ requires identifying the action a' that maximizes the Q-value for the next state s'. In a discrete action space, this is straightforward since we can iterate over all possible actions. However, in a continuous action space, finding this maximum requires solving a complex optimization problem, which is computationally expensive and may not be solvable analytically.