Self Balancing AI

Siciliano Niccolò(1958541), Paossi Davide(1950062)

The Cartpole problem is a classic control problem in the field of reinforcement learning, aiming to maintain balance of a pole attached to a cart by applying appropriate actions. This paper presents an application of Q-learning, a popular model-free reinforcement learning algorithm, to solve a modified version of the Cartpole problem. We discuss the implementation of a self balancing inverted pendulum in a realistic simulated environment that takes in account gravity and air friction.

Artificial Intelligence | Reinforcement Learning | Inverted Pendulum | Q Learning

Introduction

The Cartpole problem, also known as the Inverted Pendulum problem, is a classic control problem widely used to evaluate RL algorithms. It involves balancing a pole attached to a cart by applying appropriate actions. In this paper we discuss an expanded version of the problem. The weight of the pendulum, in our implementation, can go also in the bottom half, requiring that the AI not only is able to keep the weight in an upright position, but that is also able to swing the weight back in the upper half. Moreover we added a random variable in the environment, the wind, to demonstrate the strength of the algorithm. In this paper we utilize the RL Q Learning algorithm to solve the problem.

Problem Formulation

Our modified Inverted Pendulum problem can be represented as a Markov decision process (MDP). The state space consists of three variables representing the pole's angle,pole's direction and pole's velocity. The actions space is discretized in 20 actions, each of them represents a different speed that can be given to the cart in order to balance the pole. The first half of the actions space correspond to a positive speed which means that the cart will move to the right, and the actions on the second half correspond to a movement to the left. The objective is to find an optimal policy that maximizes the cumulative reward while maintaining the pole's balance with the lowest velocity possible.

Q Learning Overview

Q-learning is a model-free reinforcement learning algorithm to learn the value of an action in a particular state. It does not require a model of the environment (hence "model-free"), and it can handle problems with stochastic transitions and rewards without requiring adaptations.

For any finite Markov decision process (FMDP), Q-learning finds an optimal policy in the sense of maximizing the expected value of the total reward over any and all successive steps, starting from the current state. Q-learning can identify an optimal action-selection policy for any given FMDP, given infinite exploration time and a partly-random policy.

"Q" refers to the function that the algorithm computes the expected rewards for an action taken in a given state. More information on the algorithm in the next section.

Q Learning Deepening

To deeply understand the Q Learning let's start from his formula [1]. The general meaning of the formula is: the new weight (or reward) of an action a in the state s is equal to a part of the actual weight plus a part of the reward of the action choosed combined with the maximum reward that we'll be able to reach after the actual action. In the specific, we have:

$1-\alpha$: How much importance we give to the actual weight.

$Q(s_t, a_t)$: The actual weight of the action a in the state s.

α : The importance of the new weight. This is in fact how much previous knowledge he "overwrite".

r_t : The reward for the action a in the state s.

γ : The discount rate. This is how much we weight the possible future reward reachable in the future over the actual reward.

$maxQ(s_{t+1}, a)$: The max weight reachable in the next state taking any action a

The weight of a combination (s,a) is the sum of all the reward taken, so higher the weight, more convenient is the combination. The key of this algorithm is the reward function, or $r_t(s_t, a_t)$, that dictate the behaviour of the AI.

It's important to keep in mind two important factors:

1. A value of learning rate of 0 means that the agent will not learn anything, and a value of 1 means that he will keep in mind only the current reward, completely forgetting the previous knowledge.
2. The discount rate, for mathematical reasons, must be lesser than 1.

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

[1]

Implementation details

We implemented Q-learning to solve the Inverted Pendulum problem using Python, PyGame(for the graphics visualization) and PyMunk(for the physics simulation). We discretized the continuous state space into a finite set of bins to facilitate Q-value table representation: the angle has been discretized in 40 different bins with the same size, while the velocity space has been discretized in 20 bins with different size, this is because we noticed that for high speeds the differences in the cart behavior were less relevant so the bins size increase as the velocity gets higher.The learning process involved initializing all the Q-Values to 0, executing several episodes with a random starting angle and updating Q-values using the Q-learning update rule[1]. Each training episode is divided into two phases: exploration phases which allow the algorithm to explore new possibilities choosing a random action and exploitation phase in which the agent choose the action with the highest reward.

The environment plays an important role. Through the air resistance we fight the inertia of the weight, so that the AI need to constantly work to give enough speed at the weight to overcome the gravity and reach the upper half. Through the wind we force the AI to constantly adapt to maintain the weight in the right position, this way he can't just sit and do nothing once reached the goal the first time.

The reward formula used in our implementation is [2], with alpha = beta = 0.5 and 20 discretized speed buckets. As shown in the graph [1] the highest reward is obtained at 90 degrees and velocity 0. More on reward function experiments in the following chapter.

$$\alpha * sin(angle) + \beta * (20 - speed)/20$$

[2]

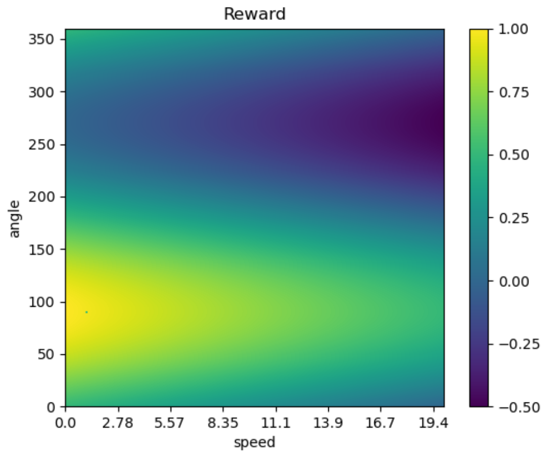

Fig. 1. Reward Function

Experimental Results and discussion

We conducted experiments to evaluate the performance of Q-learning in solving the Inverted Pendulum problem. We measured the average reward obtained per training session. The experiments were conducted over multiple iterations with varying hyperparameters such as learning rate, discount factor and epsilon decay. In details, for each test we repeated the training and the simulation six times and get the average of the reward to be sure to avoid unlucky(or lucky) cases. The results demonstrated a steady improvement in the agent's performance, achieving stable weight balancing for extended periods but our implementation is vulnerable to the catastrophic forgetting. The hyperparameters selected by us help to reduce drastically, but not eliminate, this phenomenon. How we can see in figure [2], we have a steady increase in performance until about 10k episodes (with an average reward of 0.66), than the performance starts to drop. Nonetheless, the agent will continue to always be able to balance the weight going forward with the episodes, keeping an average reward around 0.5. The results indicate the effectiveness of Q-learning in solving the given problem.

Another series of experiments concerning the learning rate show us that the best performance were achieved around a learning rate of 0.1 as shown in the graph [3]. At the end we tried to gain some improvement by adopting some decay strategies for both learning rate and the epsilon parameter. In the first place we tried an exponential epsilon decay and then a linear decay. The experiments show better performances with the linear solution. About learning rate, we compare three solutions, exponential decay, linear decay and fixed learning rate resulting in a winning for the fixed solution.

We also tried different training methods to find the best effective. Initially all the episodes started with the weight in upright position, but this was making train the "swing" very hard and not much consistent. Hence we made the starting position totally random, this gave us a better table exploration and therefore better performance. Another good modification is what we call "grace period". At the start of each episode, the agent has a limited number of actions before we start checking the failing of the task. In this way, mostly in the beginning of the training, it can train much faster and learn techniques otherwise impossible, like swing back and fourth the weight to gain speed. After this period, in the instant the weight enters the failing condition (the weight touches the farthest angle from the target), the episode ends.

REWARD FUNCTION. At first we trained the algorithm with the reward function[3], we simply used the UPorDOWN function, which returns 0 when the cube is moving downwards and 1 if the cube is moving upwards. Despite the simplicity of the function, it worked just fine for the first seconds of the simulation than the cube started to gain speed and eventually the base was not able to balance the cube. So we introduced a penalty on the speed of the cube and trained the algorithm with reward function:

$$\alpha * UPorDOWN() + \beta * (20 - speed)/20 \quad [3]$$

(where alpha and beta are the weight of the two rewards and 20 is the highest speed) so higher was the speed and lower was the reward. This was enough to maintain the cube in the correct position for as long as we liked. Then we introduced the swing up problem, so the cube was initialized at 270 degrees and the algorithm had to take it up. At the end of the training the algorithm was not able to balance the cube, in fact the cart attached to the pendulum remained still and didn't move at all. Probably this was caused by the fact that from 180 to 359 degrees there's no way to gain a constant increasing reward and the highest reward possible at this position is at 270 degrees with speed equals to 0.

We tried another approach, changing totally the formula. We came up with the idea to slow down the weight when close to the target angle, and speeds it up when far from the target. The formula[4] does that, but we noticed that with it the agent ended up with just speeding up the cube as fast as possible when in the bottom half, making impossible to slow and stop the weight later on.

$$\alpha * sin(angle) + \beta * sin(angle) * (20 - speed)/20 \quad [4]$$

So we worked on a more precise reward function and came up with the following formula [2]. In this case the model was able to take the cube at 90 degrees not just starting from 270 but from every angle we initialize the cube.

The bucket dilemma. At the start we had a tight speed bucket distribution, with 1 bucket for each speed value between 0 and 10. This seems a good idea, that could give a better control over slow speeds, but that's not totally true. The difference between those speeds is so small that it became difficult to visit them all with all the different conditions. So they tend to need an exaggerated amount on training and also in this case, they hardly converge. This makes the agent bulky and can make impossible to balance the weight.

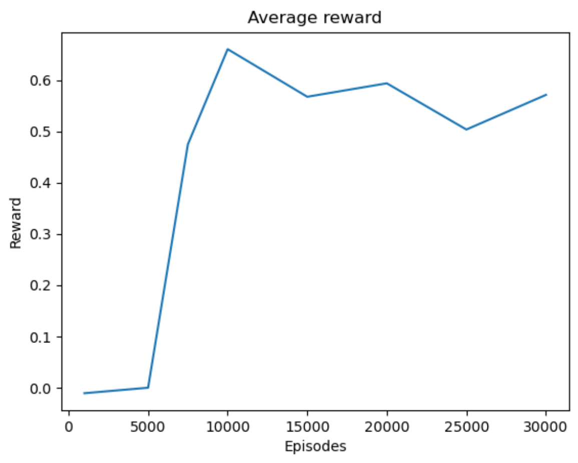

Fig. 2. Average reward per episodes

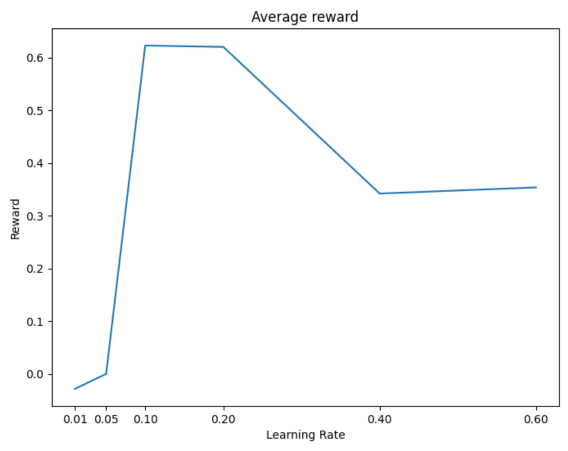

Fig. 3. Average reward per Learning Rate

Known Problems

The code has been implemented by hand, without the use of an external library such sklearn or pytorch. This gave us total control on the code and it's behavior, but it came with his drawbacks. First of all, our code is cpu-based. We have not implemented the gpu support and so training the agent can be very slow and tedious. The second problem is a consequence of the first one. Do 1 million episodes on a cpu is not feasible, and also 100.000 episodes can take hours. So our implementation has a relatively simple table (about 32.000 states) because we tried exploring a larger table(about a quarter million of states) and we noticed that also with 100/200 thousand episodes about a quarter of the table was not visited even once. A bigger state space would certainly help to make the agent less jerky and more fluid, but the performance should remain similar.