FC2: Reinforcement Learning

Heriberto Espino Montelongo

Universidad de las Américas Puebla

P24-LIS3082-2: Inteligencia Artificial

Dra. Ingrid Kirsching

March 20, 2024

Table of Contents

Introduction

Reinforcement Learning is a branch of Machine Learning that deals with how software agents ought to take actions in an environment in order to maximize some notion of cumulative reward. It does so by exploration and exploitation of knowledge it learns by repeated trials of maximizing the reward. (DataCamp, 2018).

This document provides an implementation and analysis of different Reinforcement Learning algorithms applied to the *CartPole* problem from *Gymnasium*. The CartPole problem involves balancing a pole by applying forces to a cart. The goal is to keep the pole upright for as long as possible by moving the cart left or right.

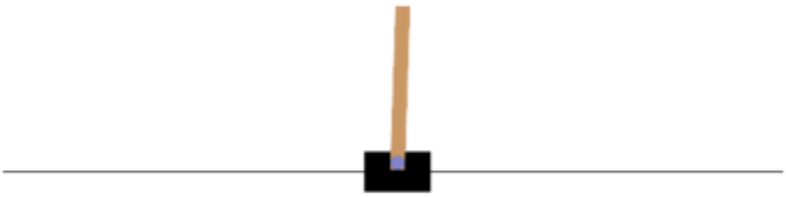
We begin by exploring a basic hard-coded policy that moves the cart in the direction opposite to the pole's lean. While this policy provides some intuition, it is far from optimal and fails to solve the problem. Next, we introduce a neural network-based policy that learns to balance the pole through the Policy Gradients algorithm. This approach demonstrates the power of learning from experience and adapting to the environment's dynamics. At the end, we use Q-Learning by discretizing the state space and iteratively updating the Q-values, we show how an agent can learn an effective policy for balancing the pole.

Environment

The goal is to balance a pole on a cart by applying forces to the cart. The environment we use, *CartPole-V1*, is part of Gymnasium toolkit, which provides a collection of environments for testing and benchmarking reinforcement learning algorithms.

Figure 1

Environment visualization



CartPole-V1, an environment of gymnasium.

The environment consists of a cart that can move horizontally on a surface, and a pole attached to the cart by an unactuated joint. The cart can be pushed left or right by applying a force, which in turn affects the angle and angular velocity of the pole. The objective is to keep the pole upright (within a specified angle range) by applying appropriate forces to the cart.

Observation Space:

The environment provides four continuous state variables as the observation at each time step:

1. Cart Position: The horizontal position of the cart on the surface.

2. Cart Velocity: The velocity of the cart.

3. Pole Angle: The angle of the pole from the vertical position.

4. Pole Angular Velocity: The angular velocity of the pole.

Action Space:

The action space is discrete, with two possible actions:

0. Push the cart to the left with a fixed force.

1. Push the cart to the right with a fixed force.

Rewards:

At each time step, the agent receives a reward of +1 if the pole remains upright. The episode terminates if the pole falls beyond the specified angle range or if the cart moves too far from the center.

Termination Conditions:

The episode terminates when the pole falls beyond a specified angle range.

Solving the Environment:

The CartPole environment is considered solved if the agent can keep the pole upright for at least 200 consecutive time steps.

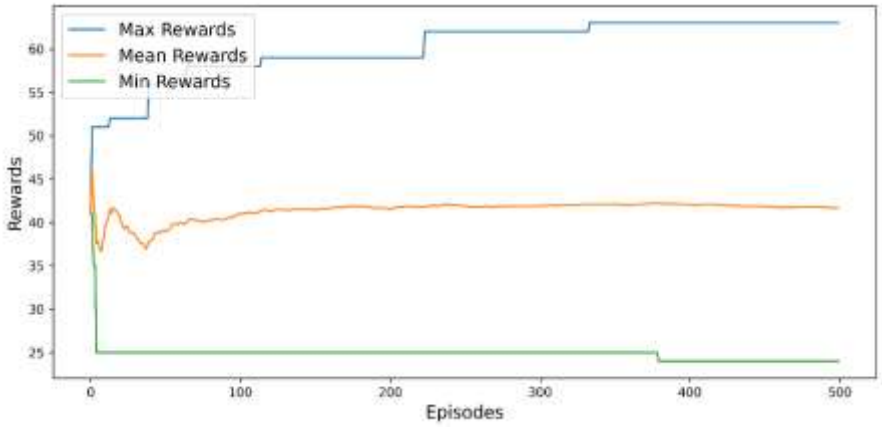
Simple Hard-Coded Policy

This basic strategy operates on a simple principle: if the pole leans left, the cart moves left, and if it leans right, the cart moves right. It is a reactive approach, without any learning involved. The decision-making process is binary: if the pole's angle is less than 0, the cart moves left; otherwise, it moves right.

To see if this basic plan is any good, we tried it out 500 times. Each time, we kept track of how well it did by looking at the total points it got. The total points add up to all the little rewards we got at each step during the episode.

Figure 2

Simple Hard-Coded Policy rewards



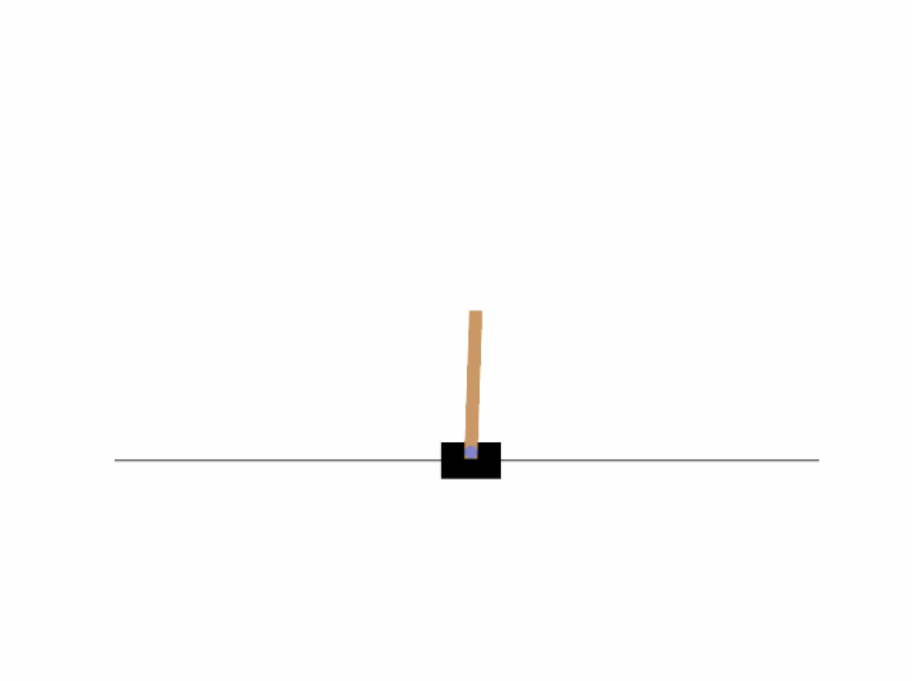
Simple Hard-Coded Policy with 500 iterations.

After all the episodes, it turns out this simple plan did not do so great. The best it did was keep the pole balanced for only 63 steps, having an average of 41. That is way below the goal of 200 steps for considering the task solved. The average points it got each time were also low, showing it is not doing a great job overall.

This basic strategy of just reacting to how the pole leans without any learning mechanism is not adequate for balancing the pole effectively. It is too simple and can´t learn from its mistakes. While simplicity is nice, in this case, it is not helping us get the job done well. We need to try out more advanced strategies that can learn and adapt to do better at this task.

Figure 3

Simple Hard-Coded Policy visualization



Visualization of one episode of Simple Hard-Coded Policy.

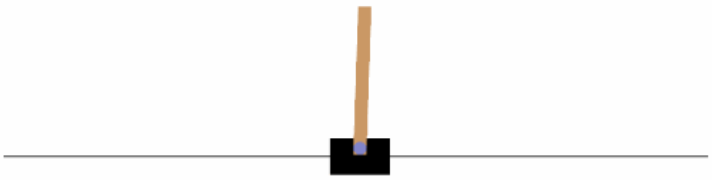
Neural Network Policies

We design a model that takes observations as inputs and outputs probabilities of actions to take. For the CartPole environment, which has two possible actions, left or right, we only need one output neuron. The output neuron provides the probability *p* of taking action 0 (left), with the probability of action 1 (right) being *1−p.*

When it is time to decide what move to make, we do not just pick the action with the highest probability. Instead, we use a mix of randomness and the probabilities with exploration-exploitation calculated by the neural network. This helps us strike a balance between trying out new actions and sticking with actions that have worked well in the past.

Figure 4

Neural Network Policy visualization



Visualization of one episode of Neural Network Policy.

The neural network learns to improve its performance through training iterations. Initially, in this case, is bad.

Policy Gradients

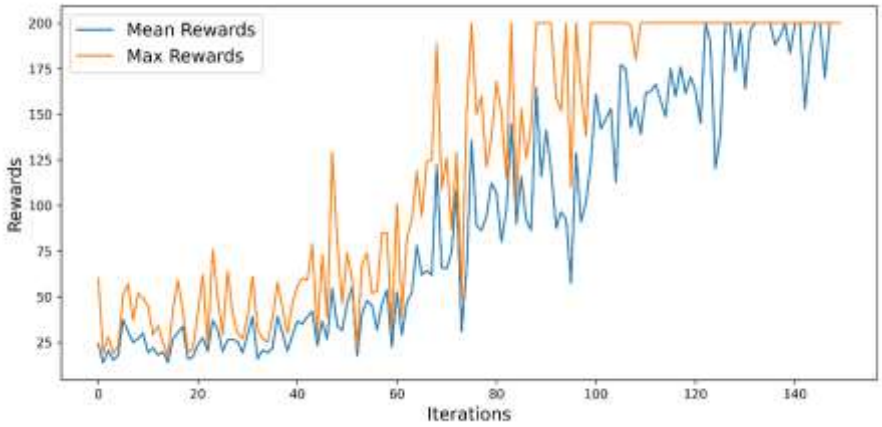
The Policy Gradients algorithm addresses the problem by playing multiple episodes in the environment and adjusting probabilities based on the observed rewards. Actions associated with positive rewards are made slightly more likely, while those associated with negative rewards are made slightly less likely. This approach allows the agent to learn from experience and improve its decision making over time.

We start by creating a function to play a single step using the model. The action is selected based on the model's predicted probabilities, and the loss and gradients are computed based on the selected action. Next, we create a function to play multiple episodes in the environment. For each episode, we play multiple steps, accumulating y and gradients along the way. We compute discounted rewards to account for the delayed effects of actions on rewards. This involves discounting future rewards based on a discount factor and summing them up. Finally, we normalize the discounted rewards across all episodes to ensure consistency and stability in training.

To train the neural network policy, we use the Nadam optimizer and the binary cross-entropy loss function. We play a total of 150 and through training iterations, the model learns to improve its performance.

Figure 5

Policy Gradients rewards



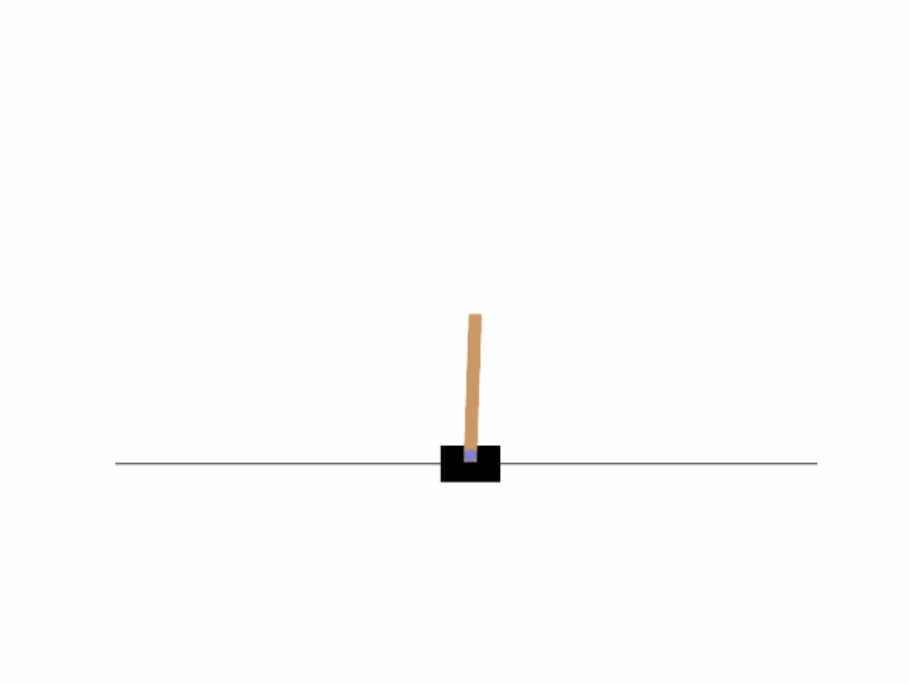
Policy Gradient with 150 iterations.

The algorithm was successful in training the neural network policy. The mean rewards increase over iterations. The agent was able to achieve optimal performance in individual episodes, successfully keeping the pole upright for the entire duration. the end of the iterations, both mean and maximum rewards consistently reach the maximum value.

While the algorithm successfully solved the problem, the training took a lot, a lot, a lot of time. While Policy Gradients can effectively train neural network policies, they may require considerable computational resources and time for solving a problem.

Figure 6

Policy Gradients visualization



Visualization of one episode of Policy Gradient.

Q-Learning

Q-learning is a model-free, value-based, off-policy algorithm that will find the best series of actions based on the agent`s current state. The “Q” stands for quality. Quality represents how valuable the action is in maximizing future rewards. The model-based algorithms use transition and reward functions to estimate the optimal policy and create the model. In contrast, model-free algorithms learn the consequences of their actions through the experience without transition and reward function. The value-based method trains the value function to learn which state is more valuable and take action. On the other hand, policy-based methods train the policy directly to learn which action to take in a given state. In the off-policy, the algorithm evaluates and updates a policy that differs from the policy used to take an action. Conversely, the on-policy algorithm evaluates and improves the same policy used to take an action. (DataCamp, 2018)

In Q-Learning it is necessary to discretize the continuous state space into manageable bins. Discretization transforms a continuous state space into discrete bins, making it easier for the learning agent to understand and learn from its environment. Parameters such as maximum training episodes, reward update intervals, and coefficients for learning and exploration rates are defined to fine-tune the learning process.

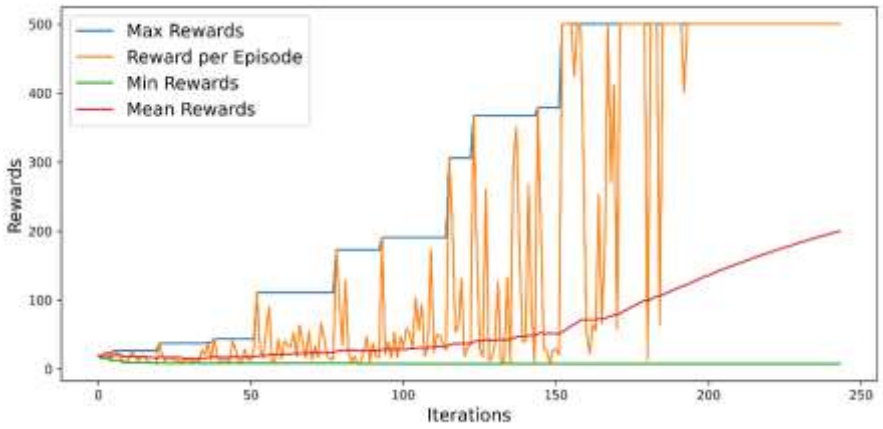
The Q-table, initialized with zeros, stores Q-values representing the expected rewards for different state-action pairs. These values are updated iteratively as the agent interacts with the environment. To discretize state values, a function called discretizer is used, which maps continuous state variables to discrete bins.

Exploration and learning parameters, such as exploration and learning rates, determine how the agent explores the environment and updates its Q-values. These rates decrease over time to balance between exploring new actions and exploiting known ones. The discount factor influences the importance of future rewards in the learning process.

The qlearning function executes episodes of the environment, updating the Q-table based on observed rewards and states. It chooses actions based on the current state, updates Q-values accordingly, and tracks rewards over iterations. Once the mean reward reaches a certain threshold, the learning process may halt.

Figure 7

Q-Learning rewards



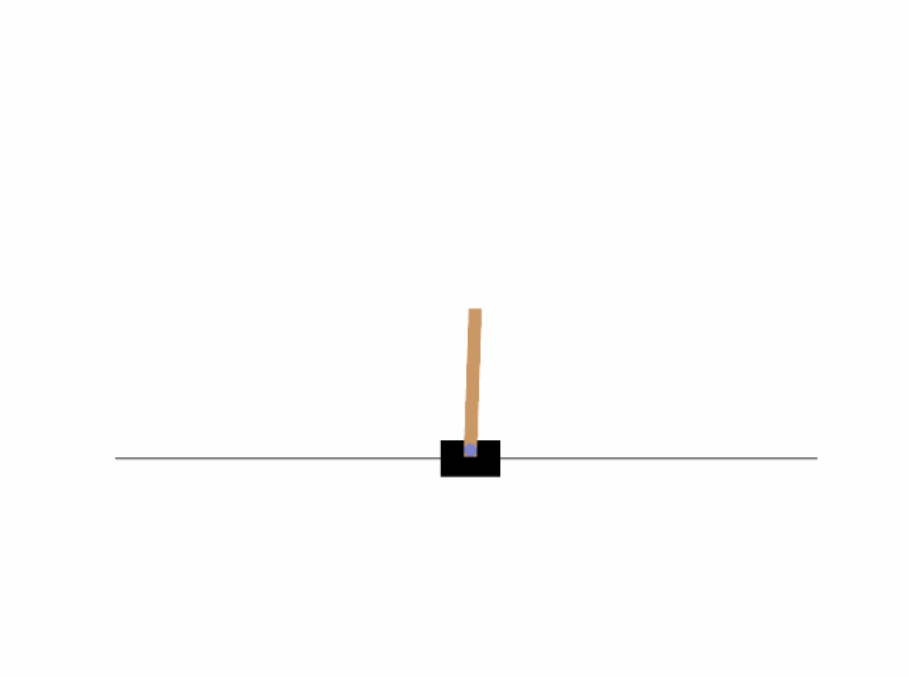
Q-Learning with 243 iterations.

In the early iterations the agent is exploring the environment and trying different actions. The maximum reward achieved in these iterations gradually increases, indicating that the agent is learning to balance the pole for longer periods. As the iterations progress, the maximum reward achieved by the agent continues to increase. The agent is learning to perform better in the environment over time. The mean reward also shows an increasing trend, indicating overall improvement in performance across episodes.

It is worth noting that the first time the episode reward exceeds 200 occurs in Iteration 115. A significant improvement is observed around Iteration 160, where the maximum reward jumps from to 500, meaning that the agent has discovered a successful strategy or policy that allows it to achieve much higher rewards. Lastly, by Iteration 243, the mean reward exceeds 200, indicating that the agent has successfully mastered the task of balancing the pole for extended periods.

Figure 8

Q-Learning visualization



Visualization of one episode of Q-Learning.

Conclusions

Each approach has its advantages and limitations when are applied. Simple hard-coded policies are easy to implement but lack adaptability, it did not even succeed. Neural Network policies and Policy Gradient offer flexibility but require extensive computational resources. Q-Learning is simple and effective but requires careful parameter tuning and discretization of the state space. The choice of method depends on the specific requirements of the problem and the available computational resources.

References

ageron. (2024, January 19). *Chapter 18 – Reinforcement Learning*. handson-ml3/18_reinforcement_learning.ipynb at main · ageron/handson-ml3 (github.com)

DataCamp. (2018, November). *Introduction to Reinforcement Learning.* Introduction to Reinforcement Learning | DataCamp

DTANGERS. (2022, October 4). *Q-Learning with Cartpole*. PASTEBIN. https://pastebin.com/wqKECmck#google_vignette

OpenAI. (2024). ChatGPT [Large language model]. https://chat.openai.com/chat

GitHub Repository

Personal repository: heritaco/Prolog (github.com)

Code: IA/3 Intelligent Agents/FC2_Reinforcement_Learning.ipynb at main · heritaco/IA (github.com)