Universidad de las Américas Puebla

Artificial Intelligence

FC2: Reinforcement Learning

Cart-Pole Experiment

Sofía Graham Coello

174291

March 19°, 2024.

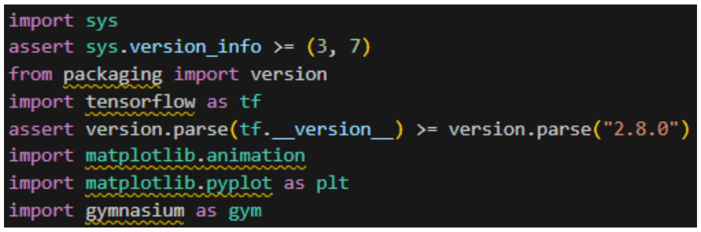
Introduction

Reinforcement Learning constitutes a paradigm where an agent learns to make sequential decisions by maximizing cumulative rewards obtained from interactions with an environment. (Sutton, 2018). On the other hand, Gymnasium, developed by OpenAI, is a toolkit which provides a diverse collection of environments to experiment and evaluate RL algorithms in various domains. Gymnasium environments are designed to be easy to use, with standardized interfaces for interacting with agents, enabling seamless integration with a multitude of RL frameworks and algorithms. (Sonawane, 2023).

The Cart Pole experiment, also known as the Inverted Pendulum problem, is a classic example used in the field of reinforcement learning (RL) to demonstrate the concept of balancing control. The experiment involves a simple setup consisting of a cart, a pole attached to the cart via a joint, and an environment where the cart can move along a one-dimensional track. The objective of the Cart Pole experiment is to keep the pole balanced upright by controlling the movement of the cart. The cart can move left or right along the track, and the goal is to apply appropriate forces to the cart to prevent the pole from falling over. (Sahoo, 2021).

Running the code

To start running the simulations, first we need to set the environment needed for the code to work, the project requires Python 3.7 or above and uses the package TensorFlow version 2.8 or higher. Additionally, to illustrate the experiments better we use the animations provided by Mathplot and finally we import the gymnasium, which includes our Cart Pole experiment.



```python
import sys
assert sys.version_info >= (3, 7)
from packaging import version
import tensorflow as tf
assert version.parse(tf.__version__) >= version.parse("2.8.0")
import matplotlib.animation
import matplotlib.pyplot as plt
import gymnasium as gym
```

Code 1 *Packages load.*

Cart Pole (version 1)

This is a very simple environment composed of a cart that can move left or right, and pole placed vertically on top of it. The objective of the agent is to move the cart left or right to keep the pole upright. In the case of the Cart Pole, each observation is a 1D NumPy array composed of 4 floats: they represent the cart's horizontal position, its velocity, the angle of the pole and the angular velocity.

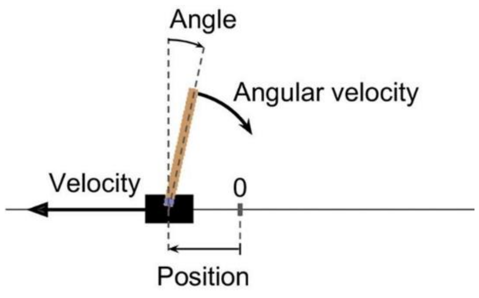
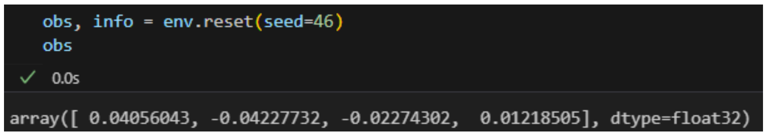


Figure 1 *Simulation elements.*

We initialize the environment by calling it's 'reset ()' method. This returns an observation, as well as a dictionary that may contain extra information. Both are environment specific. We set a seed value to make sure that the initial state of the environment after calling reset () will be the same every time. Additionally, we can visualize this environment and verify how it is interacting with us.



```
obs, info = env.reset(seed=46)
obs
✓ 0.0s
array([ 0.04056043, -0.04227732, -0.02274302,  0.01218505], dtype=float32)
```

Code 2 *Output of obs function.*

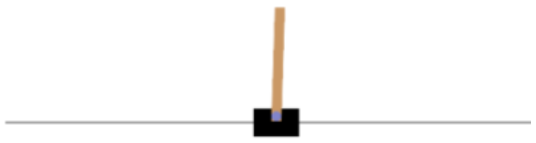
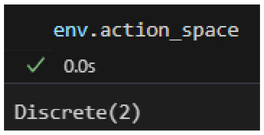


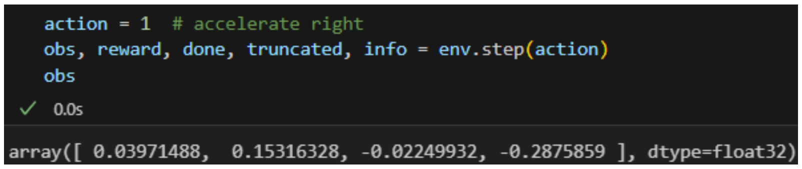
Figure 2 *Pole illustration with seed = 46.*

With the function included in the gymnasium we can check how the agent will interact, in this case there are two possible actions, accelerate towards the left (0) or towards the right (1). Since the pole is leaning toward the right ('obs [2] > 0'), the cart must move toward the right.



```
env.action_space
✓ 0.0s
Discrete(2)
```

Code 3 *Environment function to show possible actions.*



```
action = 1  # accelerate right
obs, reward, done, truncated, info = env.step(action)
obs
✓ 0.0s

array([ 0.03971488,  0.15316328, -0.02249932, -0.2875859 ], dtype=float32)
```

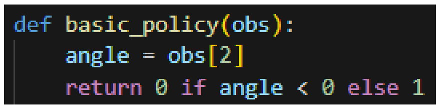
Code 4 *Result in elements of the Cart Pole after acting.*

Comparting these values with the first simulation, there are changes in all the values, which confirms the agent is moving as intended to keep the pole upright. The current position has the cart moving toward the right (`obs [1] > 0`). The pole is still tilted toward the right (`obs [2] > 0`), but its angular velocity is now negative (`obs [3] < 0`), so it will likely be tilted toward the left after the next step.

The other remaining elements in these simulations are the *reward*, which the gymnasium keeps track of, when the simulation is *done* and if the simulations are *truncated* in any way (time limit, out of bounds, etc.). As we know the basics elements to work with the simulations, we need to define the policy which the agent will follow to keep the pole upright indefinitely.

Simple hard-coded policy

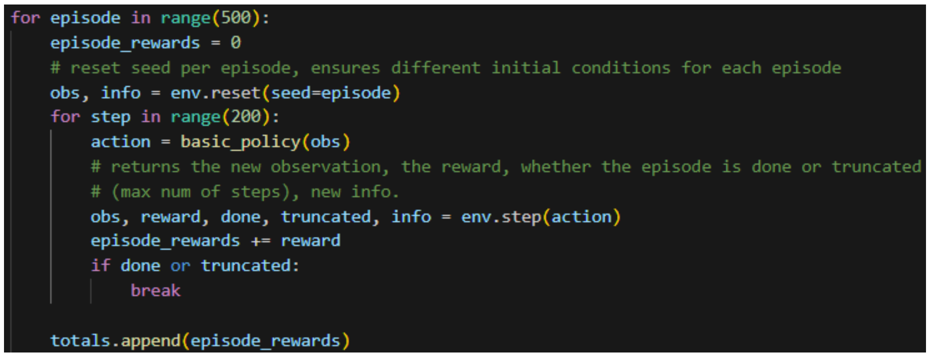
This first strategy of making a policy is simple, it consists of the following logic: if the pole is tilting to the left, then push the cart to the left, and vice versa. When put in terms for the environment to understand, if the angle (3rd element in the obs array) is positive then push the cart to the right and if its negative push to the left.



```
def basic_policy(obs):
    angle = obs[2]
    return 0 if angle < 0 else 1
```

Code 5 *Definition of our basic policy.*

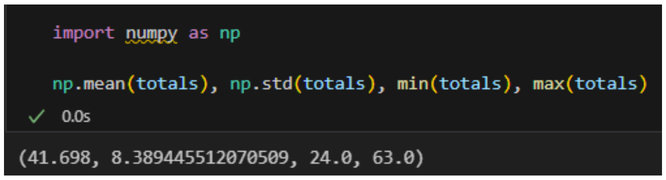
The code then runs 500 episodes using this policy and records the total rewards obtained in each episode, and for each episode it runs for a maximum of 200 steps. In each step the action of our agent is determined by the basic policy function, the environment is stepped forward using env.step(action), which returns the new observation, the reward obtained, whether the episode is done, whether the episode was truncated due to reaching the maximum number of steps, and additional information, then we update the episode rewards and if the episode is done or truncated, the step loop breaks.


```python
for episode in range(500):
    episode_rewards = 0
    # reset seed per episode, ensures different initial conditions for each episode
    obs, info = env.reset(seed=episode)
    for step in range(200):
        action = basic_policy(obs)
        # returns the new observation, the reward, whether the episode is done or truncated
        # (max num of steps), new info.
        obs, reward, done, truncated, info = env.step(action)
        episode_rewards += reward
        if done or truncated:
            break

    totals.append(episode_rewards)
```

Code 6 *Simulations of our agent following our basic policy.*

We can obtain some statistics for our hard coded policy to measure the performance of the agent, the mean gives an indication of the average performance, the standard deviation tells us about the variability or consistency of the performance, and the minimum and maximum values give us an idea of the range of performance observed across the episodes.


```python
import numpy as np

np.mean(totals), np.std(totals), min(totals), max(totals)
```
```
✓ 0.0s

(41.698, 8.389445512070509, 24.0, 63.0)
```

Code 7 *Result of basic statistics to measure the agent's performance.*

As these are the results, we can conclude that this strategy is still too basic, as the best we did was to keep the pole up for 63 steps and our objective is to keep it up for longer, at least 200 steps.

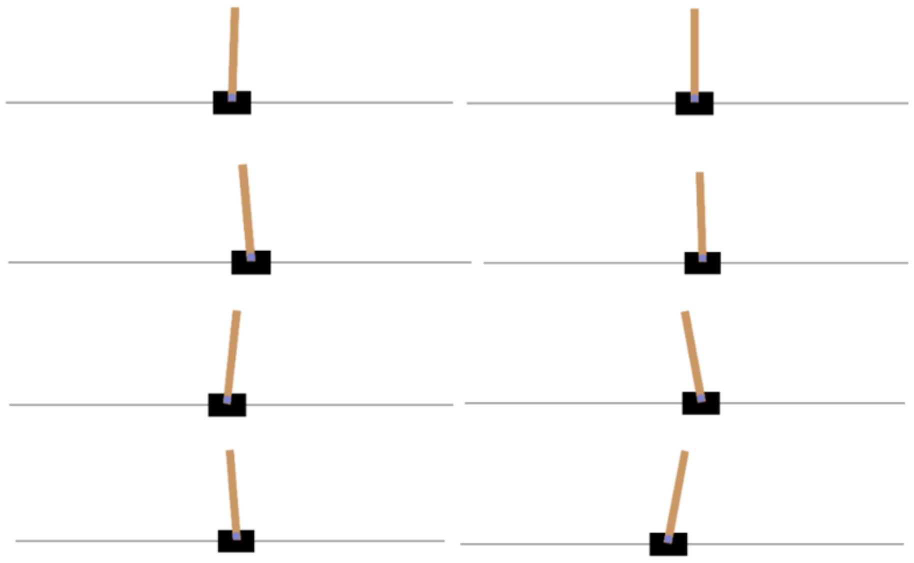


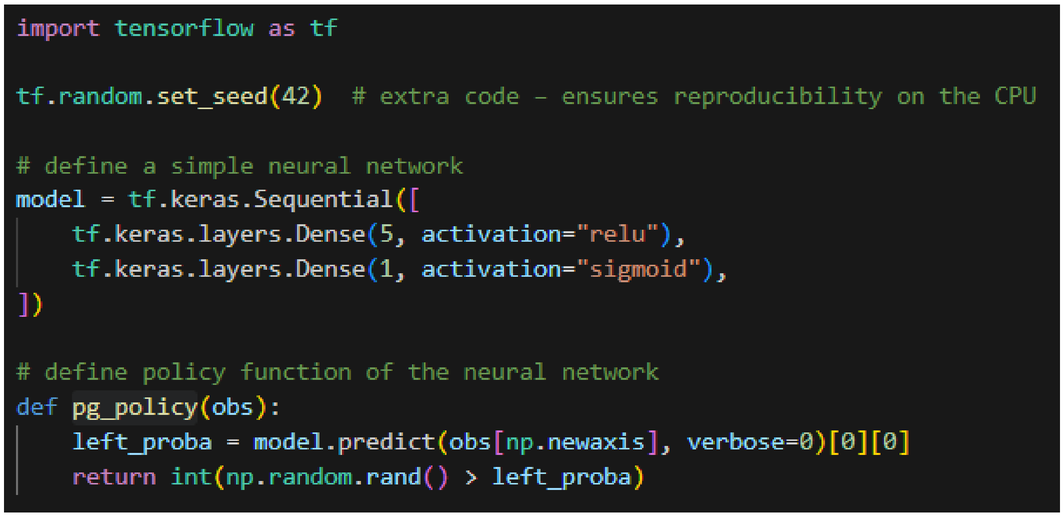
Figure 3 *Animated sequence of the agent trying to hold the pole.*

These images try to illustrate how the pole moves, and consequently where the agent reacts based on the policy we wrote. We can appreciate how the pole begins to move more erratically to the sides, to the point where the agent can no longer hold it up right and fails. We need to smarten up the agent, so the wobbles of the pole are not as unstable, which leads to an early game over.

Neural network policies

To better up our agent, we work on creating a neural network to dictate our policy, in general, neural networks take observations as inputs, and output the probabilities of actions to take for each observation. In this case, as there are two possible actions (move left or right) we only need one output neuron: it will output the probability 'p' of the action 0 (left), and of course the probability of action 1 (right) will be '1 – p'.

A benefit of working with this environment is that it allowed us to make this simple model and work with these probabilities is that actions can be independent from one another, as our observation output considers the full state of the Pole (the cart's position, its velocity, the angle of the pole and the angular velocity).



```python
import tensorflow as tf

tf.random.set_seed(42)  # extra code – ensures reproducibility on the CPU

# define a simple neural network
model = tf.keras.Sequential([
    tf.keras.layers.Dense(5, activation="relu"),
    tf.keras.layers.Dense(1, activation="sigmoid"),
])

# define policy function of the neural network
def pg_policy(obs):
    left_proba = model.predict(obs[np.newaxis], verbose=0)[0][0]
    return int(np.random.rand() > left_proba)
```

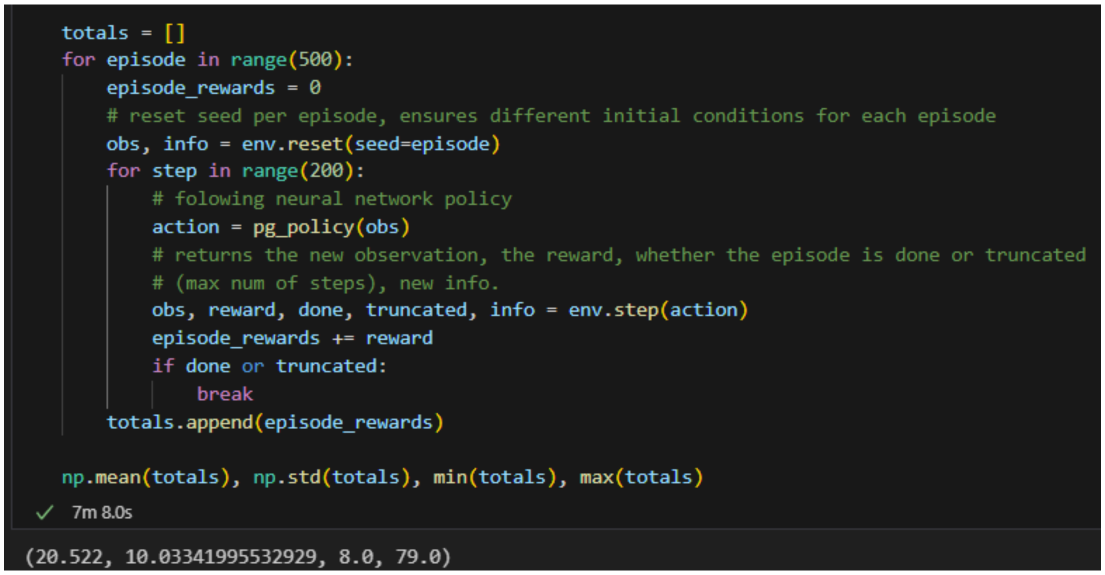
Code 8 *Defining the neural network model and its policy.*

Using function in the TensorFlow library we define our neural network, this model, according to the Tenser Flow wiki guide, is apparently used most in simple neural networks for tasks such as binary classification, in our case, to decide to move left or right.

Once we have the model, we use it to define our policy, the function uses the neural network model to predict the probability of taking the left action given the current observation. This prediction outputs a probability value for the left action. Once we calculate this probability, the function generates a random number between 0 and 1, if this random number is greater than the predicted probability of taking the left action, it selects the right action; otherwise, it selects the left action.

This process of selecting a random action introduces stochasticity in action selection, which is the randomness needed to balance the exploitation and exploration of our agent., which as we talked in class is the process of relaying in the positive actions, we made in the past but without closing the possibility of something new.

Using the same code to iterate episodes and steps, only modifying that the actions are based on the new policy using neural network, we can measure the performance of this model. The result is that by itself, as it is now, the agent is bad, the randomness of the actions leads to a major gap in between the minimum and maximum of steps, and the mean is a lot lower. Meaning that we can have these jumps in between a very good run of the agent and something abysmal, decreasing the reliability compared to the hard coded policy.



```
totals = []
for episode in range(500):
    episode_rewards = 0
    # reset seed per episode, ensures different initial conditions for each episode
    obs, info = env.reset(seed=episode)
    for step in range(200):
        # folowing neural network policy
        action = pg_policy(obs)
        # returns the new observation, the reward, whether the episode is done or truncated
        # (max num of steps), new info.
        obs, reward, done, truncated, info = env.step(action)
        episode_rewards += reward
        if done or truncated:
            break
    totals.append(episode_rewards)

np.mean(totals), np.std(totals), min(totals), max(totals)
```

✓ 7m 8.0s

```
(20.522, 10.03341995532929, 8.0, 79.0)
```

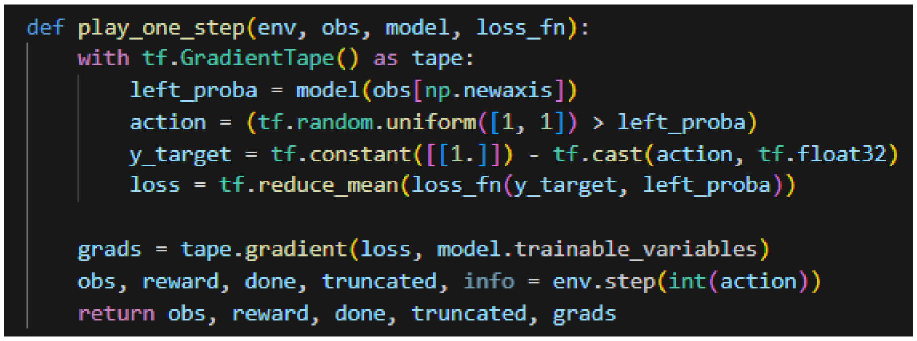
Code 9 *Run of the agent with neural network policy.*

We need to upgrade this policy to make it more consistent and preform better, as even if we had some good runs and keep the pole up to 79 steps, it is still far from our objective of 200 steps. The next objective is to see if the agent can learn a better policy on its own that leads to less wobbles of the pole and a lot longer run.

Policy gradients

Policy gradients is a class of reinforcement learning algorithms used to train an agent to learn a policy that maximizes cumulative rewards in an environment. In our case, we will need to define the target probabilities **y**. If an action is good, we should increase its probability, and conversely if it is bad we should reduce it. The problem lays in knowing which actions are positive, as the effect may not be immediate, this issue is known as the *credit assignment problem.*

The algorithm tackles this problem by first playing multiple episodes, then making the actions near positive rewards slightly more likely, while actions near negative rewards are made slightly less likely. (Géron, 2022). To accomplish this solution, we need a function that assumes the action it is taking now is the best one based on our neural network model we had. As this action is the right one, we compute the loss and its gradient and save it, to modify and determine later how good or bad it really was.


```python
def play_one_step(env, obs, model, loss_fn):
    with tf.GradientTape() as tape:
        left_proba = model(obs[np.newaxis])
        action = (tf.random.uniform([1, 1]) > left_proba)
        y_target = tf.constant([[1.]]) - tf.cast(action, tf.float32)
        loss = tf.reduce_mean(loss_fn(y_target, left_proba))

    grads = tape.gradient(loss, model.trainable_variables)
    obs, reward, done, truncated, info = env.step(int(action))
    return obs, reward, done, truncated, grads
```

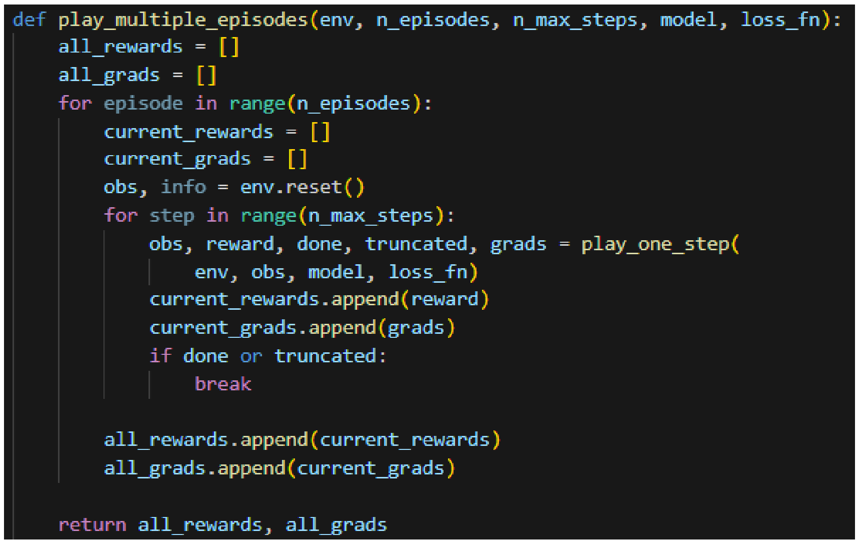
Code 10 *Function that makes a single step for agent and calculates the consequences.*

The function implements a step of interaction between an agent and the environment using the model to predict actions. Based on the input, the neural network model is used to predict the probability of taking the left action given the current observation. A binary action is then sampled stochastically based on the predicted probability. If a random number generated from a uniform distribution is greater than the predicted probability, the action is set to 1 (indicating the right action); otherwise, it is set to 0 (indicating the left action), like the policy made earlier in the neural network policy.

The novelty is that a target value **y** is computed based on the sampled action. If the sampled action is 1, the target value is 0; otherwise, it is 1. This target value is then compared to the predicted probability of taking the left action. The loss between the target value and the predicted probability is computed using the specified loss function. This loss is used to compute gradients of the model's trainable variables with respect to the loss using backpropagation.

The sampled action is applied to the environment and the environment returns the new observation, reward, doneness, the truncated status, and additional information about the step, basically this function calculates a single step of interaction between the agent and the environment.

Using this function that operates on a single step of our agent, we start a new function that operates on multiple episodes, and multiple steps, where we record all the rewards and gradients in each episode and in each step, for each of these we apply the logic explain before. The policy gradient algorithm looks back at the rewards we saved up, and normalizes them, as to facilitated choosing the best actions.

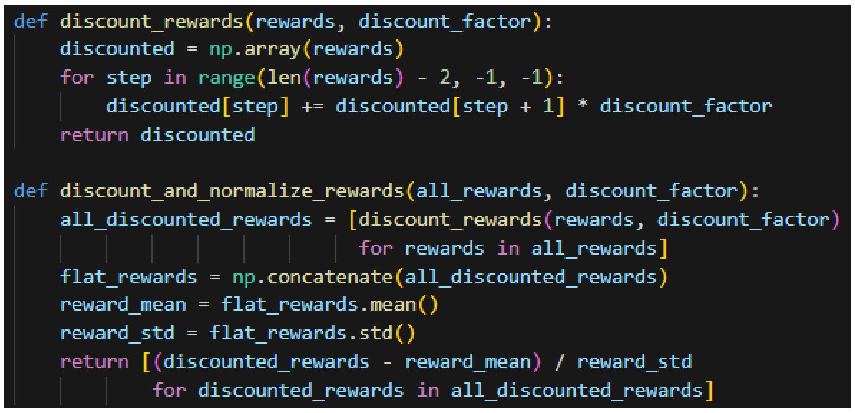


```
def play_multiple_episodes(env, n_episodes, n_max_steps, model, loss_fn):
    all_rewards = []
    all_grads = []
    for episode in range(n_episodes):
        current_rewards = []
        current_grads = []
        obs, info = env.reset()
        for step in range(n_max_steps):
            obs, reward, done, truncated, grads = play_one_step(
                env, obs, model, loss_fn)
            current_rewards.append(reward)
            current_grads.append(grads)
            if done or truncated:
                break

        all_rewards.append(current_rewards)
        all_grads.append(current_grads)

    return all_rewards, all_grads
```

Code 11 *Code that runs and records multiple episodes.*

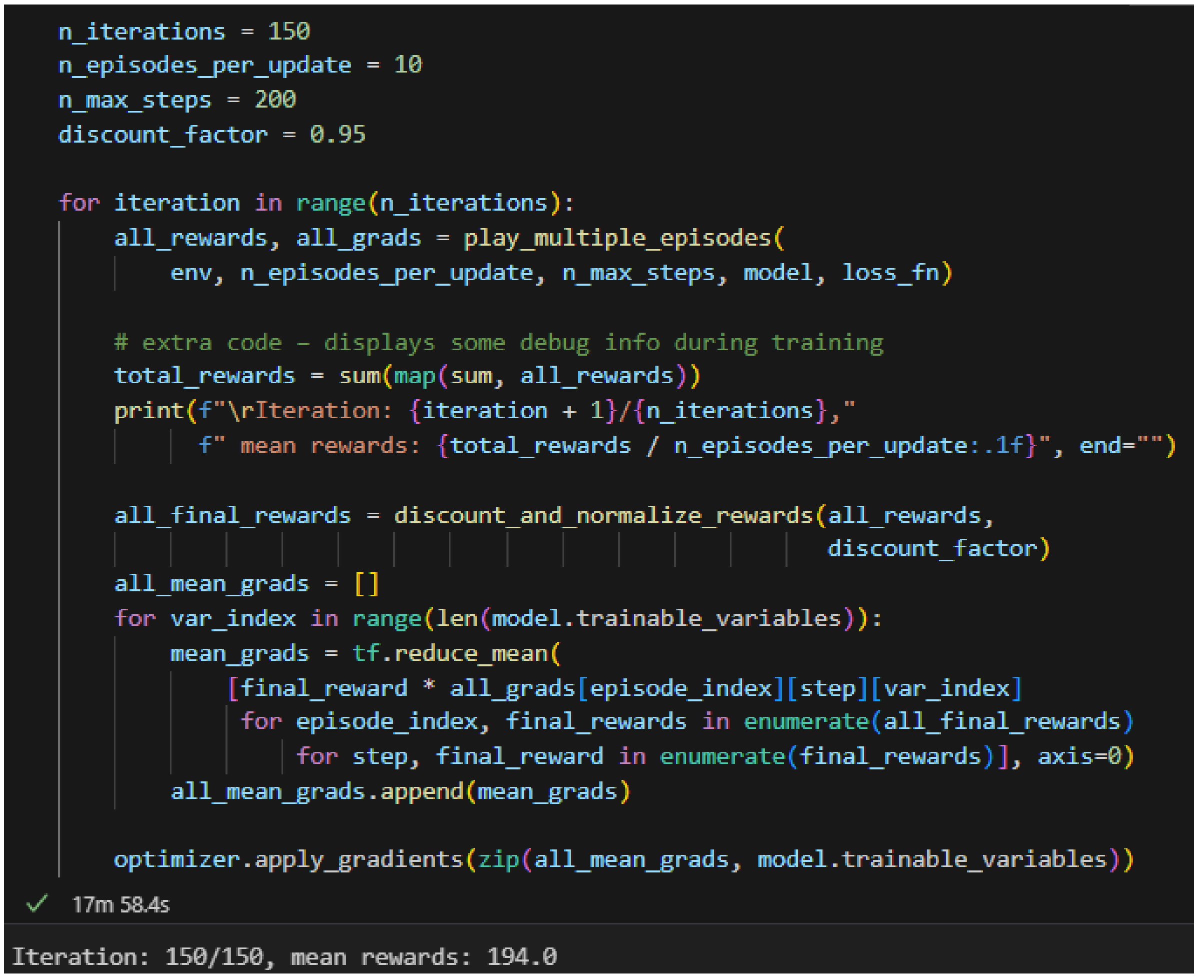


```
def discount_rewards(rewards, discount_factor):
    discounted = np.array(rewards)
    for step in range(len(rewards) - 2, -1, -1):
        discounted[step] += discounted[step + 1] * discount_factor
    return discounted

def discount_and_normalize_rewards(all_rewards, discount_factor):
    all_discounted_rewards = [discount_rewards(rewards, discount_factor)
                              for rewards in all_rewards]
    flat_rewards = np.concatenate(all_discounted_rewards)
    reward_mean = flat_rewards.mean()
    reward_std = flat_rewards.std()
    return [(discounted_rewards - reward_mean) / reward_std
            for discounted_rewards in all_discounted_rewards]
```

Code 12 *Functions that operate the rewards.*

Once we have all the elements needed for our agent to learn and discern the best policy to follow based on different actions, we start training by making iterations of different episodes and recording the different results and use each of the iterations as a background to modify the next run and maximize the rewards by adjusting its policy based on the feedback obtained from interactions.



```
n_iterations = 150
n_episodes_per_update = 10
n_max_steps = 200
discount_factor = 0.95

for iteration in range(n_iterations):
    all_rewards, all_grads = play_multiple_episodes(
        env, n_episodes_per_update, n_max_steps, model, loss_fn)

    # extra code – displays some debug info during training
    total_rewards = sum(map(sum, all_rewards))
    print(f"\rIteration: {iteration + 1}/{n_iterations},"
          f" mean rewards: {total_rewards / n_episodes_per_update:.1f}", end="")

    all_final_rewards = discount_and_normalize_rewards(all_rewards,
                                                       discount_factor)
    all_mean_grads = []
    for var_index in range(len(model.trainable_variables)):
        mean_grads = tf.reduce_mean(
            [final_reward * all_grads[episode_index][step][var_index]
             for episode_index, final_rewards in enumerate(all_final_rewards)
                 for step, final_reward in enumerate(final_rewards)], axis=0)
        all_mean_grads.append(mean_grads)

    optimizer.apply_gradients(zip(all_mean_grads, model.trainable_variables))
```

✓ 17m 58.4s

```
Iteration: 150/150, mean rewards: 194.0
```

Code 13 *Policy gradient training loop.*

This output suggests that the training process has progressed well, with the mean rewards steadily increasing over the course of the iterations. Each iteration represents a cycle of collecting new input from the environment and updating the model's parameters. In this case, there are 150 iterations in total. A mean reward of 194 indicates that the agent is achieving a high level of performance in the task, indicating that, on average, the agent is receiving a reward of 194.0 across episodes in the current iteration.

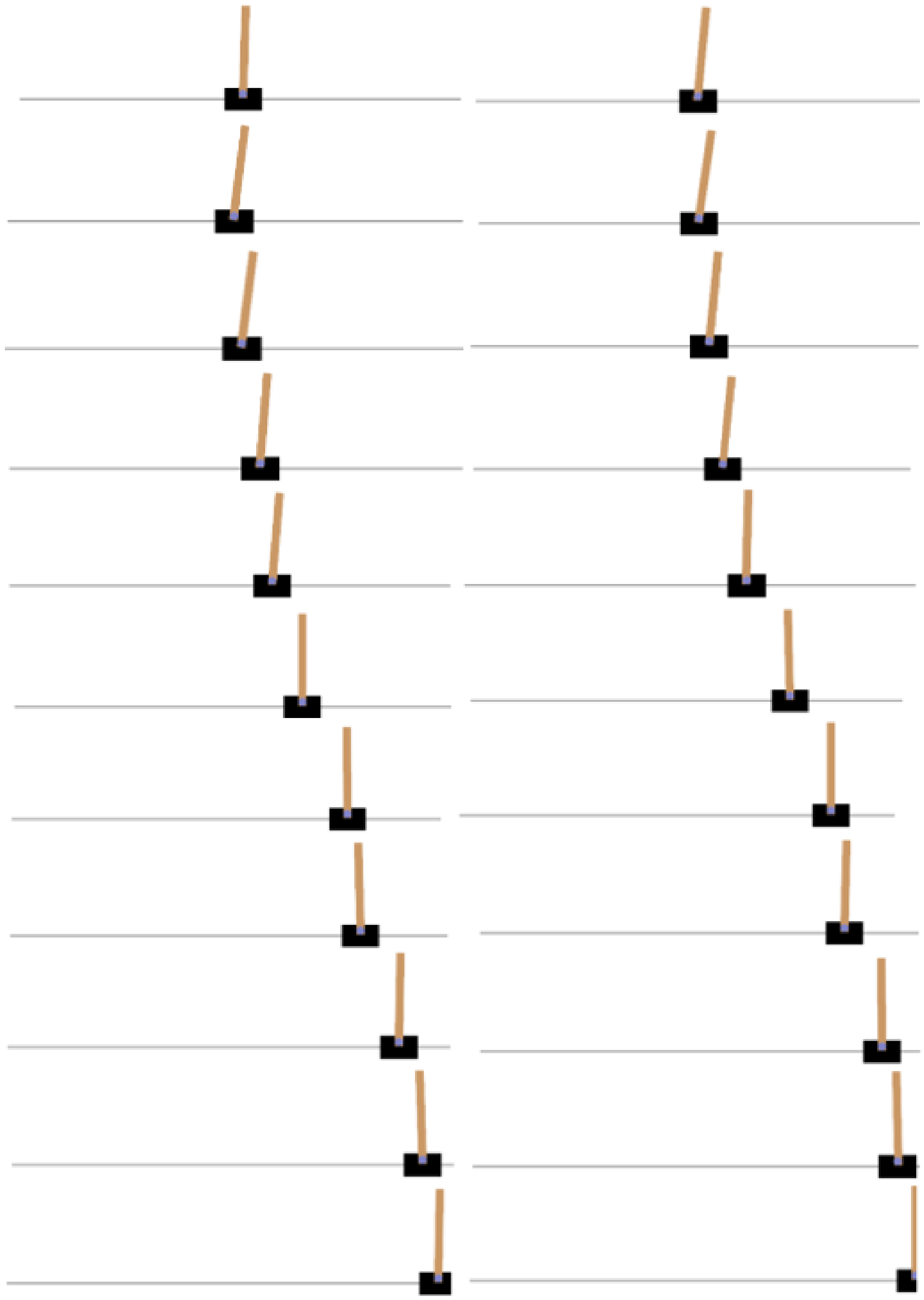


Figure 4 *Animation of the cart following the gradient policy to hold the pole.*

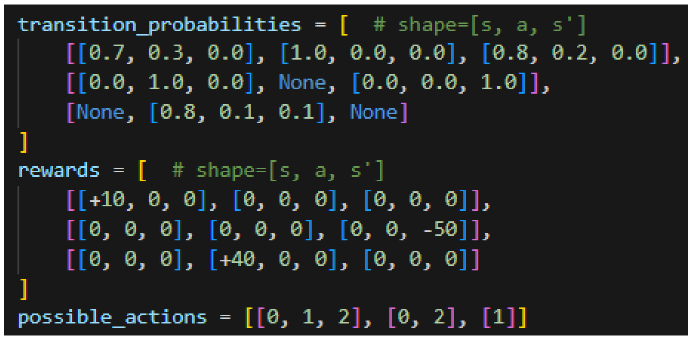
Q-learning

This method has a systematic approach for agents to learn optimal strategies through trial and error, as it revolves around the notion of Q-values, which represent the expected cumulative rewards an agent can attain by taking a particular action in a specific state. This algorithm operates in an iterative fashion, where the agent explores the environment, receives rewards, and updates its Q-values accordingly.

The updating process is guided by the Bellman equation, which balances the immediate reward with the expected future rewards, incorporating a learning rate and discount factor to control the impact of new information and the importance of future outcomes. By continually refining its Q-values through experience, the agent gradually converges towards an optimal policy, enabling it to make informed decisions and navigate complex environments more effectively.

$$q^{new}(s,a) = (1-\alpha)\underbrace{q(s,a)}_{\text{old value}} + \alpha\overbrace{\left(R_{t+1} + \gamma \max_{a'} q(s',a')\right)}^{\text{learned value}}$$

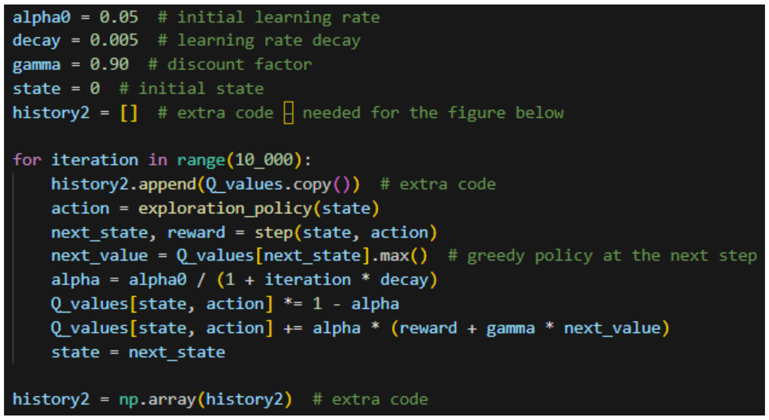
If we use the matrix of actions and probabilities defined by the user, we can run the code without much problem as it uses discrete values, which works perfectly for the equation above. This will result in computing the optimal q-values to follow in training our agent where it moves around the values stablished in the matrix below.



```
transition_probabilities = [  # shape=[s, a, s']
    [[0.7, 0.3, 0.0], [1.0, 0.0, 0.0], [0.8, 0.2, 0.0]],
    [[0.0, 1.0, 0.0], None, [0.0, 0.0, 1.0]],
    [None, [0.8, 0.1, 0.1], None]
]
rewards = [  # shape=[s, a, s']
    [[+10, 0, 0], [0, 0, 0], [0, 0, 0]],
    [[0, 0, 0], [0, 0, 0], [0, 0, -50]],
    [[0, 0, 0], [+40, 0, 0], [0, 0, 0]]
]
possible_actions = [[0, 1, 2], [0, 2], [1]]
```

Code 14 *Matrix to work the q-values based on the equation.*

As we establish the possible behavior the agent can have, as well as the reward and the possibilities of each path, we can begin training our agent by having it calculate the modified q-values every iteration and improving itself.



```
alpha0 = 0.05  # initial learning rate
decay = 0.005  # learning rate decay
gamma = 0.90  # discount factor
state = 0  # initial state
history2 = []  # extra code ▯ needed for the figure below

for iteration in range(10_000):
    history2.append(Q_values.copy())  # extra code
    action = exploration_policy(state)
    next_state, reward = step(state, action)
    next_value = Q_values[next_state].max()  # greedy policy at the next step
    alpha = alpha0 / (1 + iteration * decay)
    Q_values[state, action] *= 1 - alpha
    Q_values[state, action] += alpha * (reward + gamma * next_value)
    state = next_state

history2 = np.array(history2)  # extra code
```

Code 15 *Training the agent by following a greedy policy and updating q-values.*

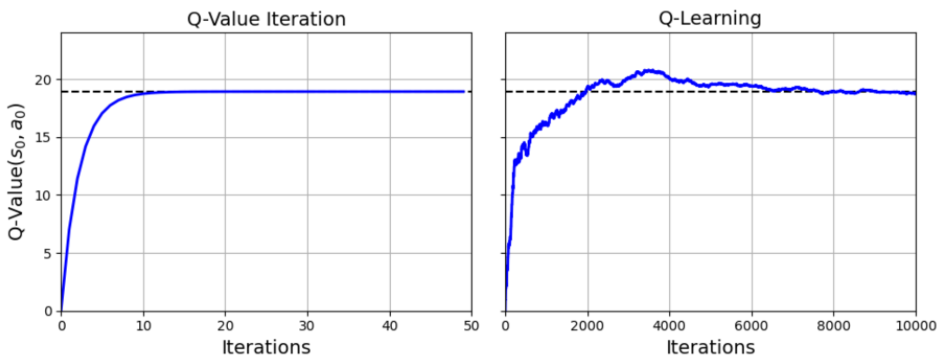


Figure 5 *Representation of increased rewards as the agent improves.*

As we can see in the Figure above, our agent around the 3000 iterations, starts learning the best way to obtain the rewards, and it stabilizes further down the iterations at around 6000. Meaning the cart stats holding the pole upright for longer time and knows the best policy to follow based on the last episodes to continue holding the pole.

But as you can see, we are not really applying the q-learning algorithm as directly as we could, as the matrix in which we base our calculations are user input, as such I followed a tutorial by Haber to perfect this which resulted in the following. First, when we apply the q-learning method to our Cart Pole problem we can synthesize our necessary elements into the following:

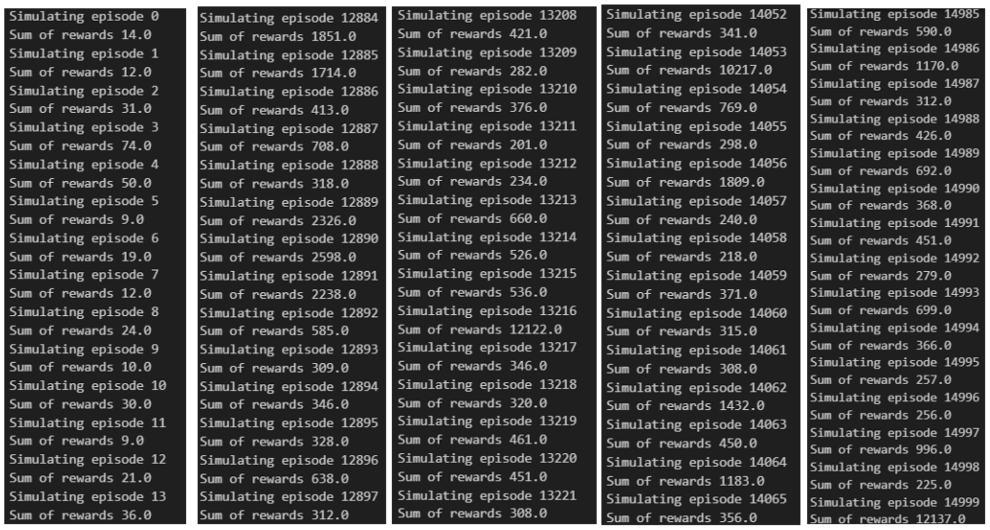
- The action space consists of two actions: Push the cart left (0) or right (1).
- The states are the cart position, the cart velocity, the pole angle, and the pole angular velocity,
- An episode terminates under the following conditions: the pole angle becomes greater than |0.2095| radians, the cart position is greater than |2.4| and if the number of steps in an episode is greater than 500.

Additionally, the reward is obtained every time a step is taken within an episode. This is because the objective is to keep the pole in the upright position. (Haber, 2023).

Another important thing to remember, and it caused me mayor issues, is that the cart pole problem has a non-discrete state matrix, this means that any of the state variables in our vector can take any value in the intervals determined by the limits of the space and since the action value function Q(S,A) is a function of the state, this means that we have an infinite number of action-value functions, which means it is practically impossible to store and update the value of Q(S,A).

This problem with the continuity of our values, as I found out, can be managed by discretizing each state variable, that basically means dividing the continuous range of values in of our vector into a finite number of intervals. This way we create the Q-table, and this is what we use to run the algorithm as intended.

The code made by Haber is quite big, the full code and commentary will be all in the Jupiter notebook as to not make this report any longer, but as we need to calculate several elements based on what we have, the main problem of the code is the sheer number of simulations made to really arrive to a conclusion, but in the end following the basic hard coded policy, of moving the cart right or left, we can appreciate in the figure below that around the episode 8000 the agent starts earning more rewards, which means it holds up the pole the longest. The code will also give an animation, but I don't recommend running it just for it.



```
Simulating episode 0
Sum of rewards 14.0
Simulating episode 1
Sum of rewards 12.0
Simulating episode 2
Sum of rewards 31.0
Simulating episode 3
Sum of rewards 74.0
Simulating episode 4
Sum of rewards 50.0
Simulating episode 5
Sum of rewards 9.0
Simulating episode 6
Sum of rewards 19.0
Simulating episode 7
Sum of rewards 12.0
Simulating episode 8
Sum of rewards 24.0
Simulating episode 9
Sum of rewards 10.0
Simulating episode 10
Sum of rewards 30.0
Simulating episode 11
Sum of rewards 9.0
Simulating episode 12
Sum of rewards 21.0
Simulating episode 13
Sum of rewards 36.0
```

```
Simulating episode 12884
Sum of rewards 1851.0
Simulating episode 12885
Sum of rewards 1714.0
Simulating episode 12886
Sum of rewards 413.0
Simulating episode 12887
Sum of rewards 708.0
Simulating episode 12888
Sum of rewards 318.0
Simulating episode 12889
Sum of rewards 2326.0
Simulating episode 12890
Sum of rewards 2598.0
Simulating episode 12891
Sum of rewards 2238.0
Simulating episode 12892
Sum of rewards 585.0
Simulating episode 12893
Sum of rewards 309.0
Simulating episode 12894
Sum of rewards 346.0
Simulating episode 12895
Sum of rewards 328.0
Simulating episode 12896
Sum of rewards 638.0
Simulating episode 12897
Sum of rewards 312.0
```

```
Simulating episode 13208
Sum of rewards 421.0
Simulating episode 13209
Sum of rewards 282.0
Simulating episode 13210
Sum of rewards 376.0
Simulating episode 13211
Sum of rewards 201.0
Simulating episode 13212
Sum of rewards 234.0
Simulating episode 13213
Sum of rewards 660.0
Simulating episode 13214
Sum of rewards 526.0
Simulating episode 13215
Sum of rewards 536.0
Simulating episode 13216
Sum of rewards 12122.0
Simulating episode 13217
Sum of rewards 346.0
Simulating episode 13218
Sum of rewards 320.0
Simulating episode 13219
Sum of rewards 461.0
Simulating episode 13220
Sum of rewards 451.0
Simulating episode 13221
Sum of rewards 308.0
```

```
Simulating episode 14052
Sum of rewards 341.0
Simulating episode 14053
Sum of rewards 10217.0
Simulating episode 14054
Sum of rewards 769.0
Simulating episode 14055
Sum of rewards 298.0
Simulating episode 14056
Sum of rewards 1809.0
Simulating episode 14057
Sum of rewards 240.0
Simulating episode 14058
Sum of rewards 218.0
Simulating episode 14059
Sum of rewards 371.0
Simulating episode 14060
Sum of rewards 315.0
Simulating episode 14061
Sum of rewards 308.0
Simulating episode 14062
Sum of rewards 1432.0
Simulating episode 14063
Sum of rewards 450.0
Simulating episode 14064
Sum of rewards 1183.0
Simulating episode 14065
Sum of rewards 356.0
```

```
Simulating episode 14985
Sum of rewards 590.0
Simulating episode 14986
Sum of rewards 1170.0
Simulating episode 14987
Sum of rewards 312.0
Simulating episode 14988
Sum of rewards 426.0
Simulating episode 14989
Sum of rewards 692.0
Simulating episode 14990
Sum of rewards 368.0
Simulating episode 14991
Sum of rewards 451.0
Simulating episode 14992
Sum of rewards 279.0
Simulating episode 14993
Sum of rewards 699.0
Simulating episode 14994
Sum of rewards 366.0
Simulating episode 14995
Sum of rewards 257.0
Simulating episode 14996
Sum of rewards 256.0
Simulating episode 14997
Sum of rewards 996.0
Simulating episode 14998
Sum of rewards 225.0
Simulating episode 14999
Sum of rewards 12137.0
```

Figure 6 *Printed results of each episode and the reward.*

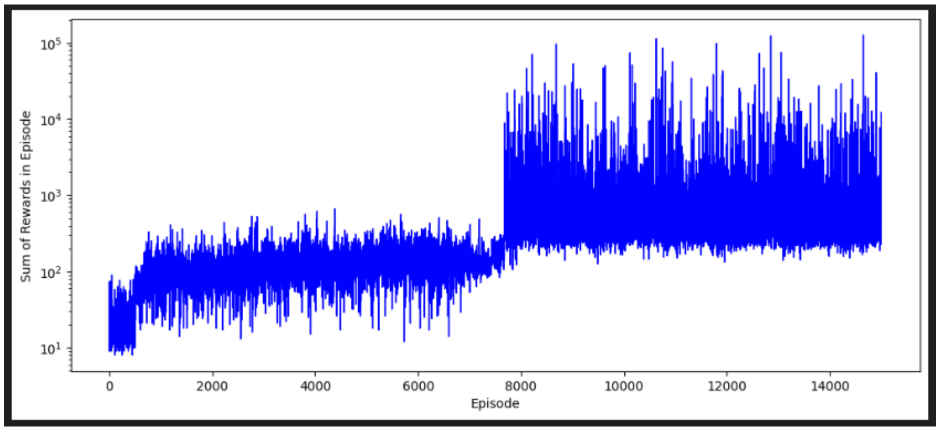


Figure 7 *Graph illustrating the progress of the agent obtaining rewards with Q Learning.*

Conclusions

There are several things we can take from experimenting with the Cart Pole environment with different policies and learning methods, each with setbacks and challenges to overcome that clearly illustrate the different ways reinforced learning works.

As we started very simple, just following along the policy of moving the cart to the right if the pole is falling to that side and vice versa, but soon it wasn't enough, as following this didn't allow our agent to last long holding the pole, as 63 steps was the maximum continuity it could hold.

To up our game we moved to neural network policies, that operate with probabilities and enough randomness to make our agent explore more possibilities in movement to fulfill its goal, but with this new way of learning and exploring our agent had too much room to explore and had no way to guide itself into improving, as it mostly operated by comparing probabilities, which one was random. But the idea had merit, so we improved this by making the policy gradient.

This new reinforcement learning method has the base of the last one, were we move the agent based on probabilities, but in this case, the probabilities of good courses of actions are elevated slightly more that the bad ones in each iteration, improving the policy to follow by itself. With this logic, our agent little by little could improve itself by only following the best probability as it indicates a positive outcome, but it creates a new problem, how do we know the good actions from the bad ones.

The approach to solve this is by introducing the concept of the reward, with this reward we can guide our agent into the best course of action by running different episodes, comparing the actions and the reward obtain in each to use it as feedback for the next one, so that by the end of our iterations our agent will have explored enough to know some of the best courses of action based on the reward it achieved. This method had by far the best result and had the longest holding time, but it started to height heavily on the hardware and it took a long time to finish processing.

Finally, we applied the q-learning algorithm that operates around rewards completely and has an expected sum of reward to obtain in each run, which we calculate using the Bellman equation. For this specific environment, were we had continuous values to operate the equation, it led to having to work around it to make them discrete, even if that can have some negative effects and the efficiency and learning rate.

In the end this method had the best result, as the mean of the rewards obtained at the end of iteration far surpasses the policy gradient, but this may be because the number of iterations is a lot bigger. But in the same note, the time spend processing this method was a lot longer and put some heavy work on the computer.

Personally, some of the challenges in this project where more around the particularities of setting the environment correctly and at some point, I gave up on relaying in Google Collab, so installing and correcting everything for the gymnasium to work was more complex than I thought. The trade off was worth it as my computer has the capability to run the code in a decent amount of time.

The last major issue I faced was arraying the q learning policy to the cart pole problem, as the one included in the code wasn't about it, it worked around the table of probabilities fixed by the user and even if that worked fine as an example, it wasn't a really a q learning curve based on the possible actions for our cart. The actions weren't reflected correctly, and the possible states weren't either.

I tried to just modify a little the one already there, but I ran into the already mentioned problem of the continuity of the variables, I got external help linked in the references and the Jupiter Notebook to achieve the desired result, even if the hardware of the computer took a bit of a hit.

At the end, I could really appreciate the difference between all the reinforcement learning methods, by starting with simple follow along policies of the hard coded algorithm to the more complicated self-updating policies of the gradient and q-learning. It really shows the contrast of how even a simple problem of holding a pole upright by a cart can be perfected to a point the cart can hold up independently, as long the computer allows it.

GitHub Link

For the full code:

https://github.com/SofiaGC13/Artificial_Intelligence_Repository/tree/ab8bb4abfb4622e9678ed6a4bf41734b4c094a35/FC2%3A%20Reinforcement%20Learning

References

-Géron, A. (2022). "Hands-On Machine Learning with Scikit-Learn, Keras & Tensorflow". 3rd Edition. O'Reilly. https://github.com/ageron/handson-ml3/blob/main/18_reinforcement_learning.ipynb

-Sutton, R. S., & Barto, A. G. (2018). Reinforcement Learning: An Introduction (2nd ed.) [PDF]. Stanford University. Retrieved from https://web.stanford.edu/class/psych209/Readings/SuttonBartoIPRLBook2ndEd.pdf

-Gymnasium. (n.d.). Basic Usage. Retrieved from https://gymnasium.farama.org/content/basic_usage/

-Sonawane, B. (2023). OpenAI Gym: A Toolkit for Developing and Comparing Reinforcement Learning Algorithms. Built In. Retrieved from https://builtin.com/software-engineering-perspectives/openai-gym

-Sahoo, S. (2021). Reinforcement Learning Concept on Cart Pole with DQN. Towards Data Science. https://towardsdatascience.com/reinforcement-learning-concept-on-cart-pole-with-dqn-799105ca670

-TensorFlow Quickstart for Beginners. (n.d.). TensorFlow. Retrieved from https://www.tensorflow.org/tutorials/quickstart/beginner?hl=es-419

- Haber, A. (2023). Q-Learning in Python with Tests in Cart Pole OpenAI Gym Environment Reinforcement Learning Tutorial. Aleksandar Haber. Retrieved from https://aleksandarhaber.com/q-learning-in-python-with-tests-in-cart-pole-openai-gym-environment-reinforcement-learning-tutorial/