Reinforcement learning

Assignment

Table of Contents

1. Intro

Reinforcement learning (RL) has emerged as a powerful tool for training agents to make decisions in a variety of environments. In this document, we will explore the use of RL in three different environments: Breakout, Racing Car, and a Custom Environment.

The Breakout environment, a classic Atari game, has been widely used as a benchmark for testing the performance of RL algorithms. Its simple yet challenging nature makes it an ideal testbed for developing and evaluating new RL algorithms.

The Racing Car environment provides a more complex and dynamic setting, where the agent must navigate a racetrack while avoiding obstacles. This environment requires the agent to learn how to balance speed and safety, making it an interesting challenge for RL algorithms.

The Custom Environment is a very basic environment that I have created. It includes an observation space consisting of temperature readings between 0 and 100 degrees. The RL agent receives a reward of 1 if the temperature falls within the range of 37 to 39 degrees; otherwise, the reward is -1. Additionally, the shower length is set to 60 seconds, and each time an action is taken, it decreases by one second. The initial state is represented by a temperature reading of 38 degrees, with a variance of plus or minus 3 degrees.

In this document, we will describe the three environments in detail, outline the RL algorithms used to train agents in each environment, and report on the experimental results obtained. Through this exploration, we hope to gain insights into the strengths and limitations of different RL algorithms and shed light on the potential of RL for solving real-world problems.

2. Breakout

I am stating this exercise by analyzing the action space and observation space. The action space consists of four discrete numbers, meaning that there are different actions that can be taken.

The observation space in the classic Atari game Breakout, as implemented in the OpenAI Gym environment, is a Box observation space with shape (210, 160, 3). This means that each observation is a 3-dimensional array of size 210x160x3, representing an RGB image of the game screen.

The first two dimensions of the observation correspond to the width and height of the game screen, while the third dimension represents the RGB values of each pixel. The RGB values are typically represented as integers ranging from 0 to 255. Using an RGB image as the observation space allows an agent to directly observe the visual state of the game.

The next step I take is to play five episodes, where I randomly choose the action. The result is this:

```
Episode:1 Score:0.0
Episode:2 Score:1.0
Episode:3 Score:0.0
Episode:4 Score:2.0
Episode:5 Score:0.0
```

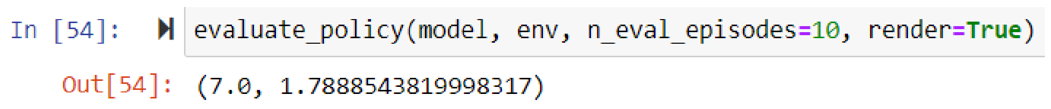
As seen above, it is very low. Now it would be interesting to play the game using a trained agent.

For training the agent, I decided to use four parallel environments to speed up the training process and increase sample efficiency. Running multiple environments in parallel allows an agent to experience more transitions from the environment in a given amount of time, which can accelerate learning.

To be able to train multiple instances of the same environment in parallel, I had to vectorize the environment. Vectorizing the environment allows for efficient parallelization of the RL training process by distributing the workload across multiple CPU cores or even multiple machines.

I trained the agent using the “A2C” model and “CnnPolicy” policy. I chose A2C because certain algorithms will perform better for certain environments. For discrete action space and multiple instances of the environment, I found that A2C/A3C and PPO perform well. This being said, I chose A2C.

Training the agent for 100000 steps, yields an average reward per episode of 7 and SD 1.788.



```
In [54]:    evaluate_policy(model, env, n_eval_episodes=10, render=True)

   Out[54]: (7.0, 1.7888543819998317)
```

Based on the results obtained, it can be concluded that using a trained agent for playing the game of Breakout yields significantly better scores than doing random actions. With an average score of 0.6 for random actions and an average score of 7 for the trained agent, the difference in performance is clear.

3. Racing Car

I am stating this exercise by analyzing the action space and observation space. The observation space is represented as a box, which means that it is a continuous space with a finite range of values for each dimension.

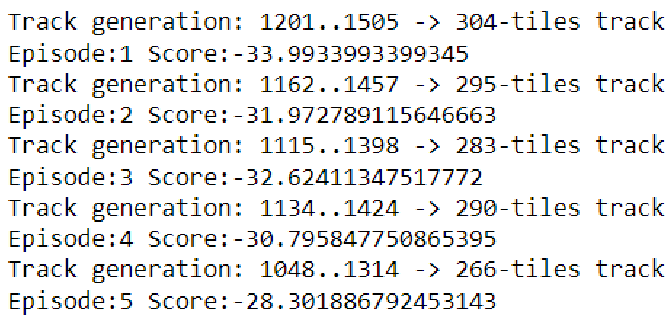
The observation space is an image, representing the current state of the game screen. Specifically, the observation consists of a 3-dimensional array of shape (96, 96, 3), where the last dimension represents the red, green, and blue color channels of the image. This means that the observation is a continuous space with a total of 27,648 dimensions. In this case, each dimension of the observation space represents a pixel value in the image, which can take on a value between 0 and 255.

In addition to the image observation, the environment also provides a vector observation consisting of 10 elements, which include the current car velocity, angular velocity, and a boolean indicating whether the car is on the road or not.

The action space for the CarRacing-v0 environment is a continuous space consisting of three values representing the steering wheel position, gas pedal, and brake pedal, respectively. Specifically, the action is a 1-dimensional array of shape (3,) with each element representing a continuous value between -1 and 1.

The first element of the action array controls the steering wheel position, where a value of -1 corresponds to full left turn, a value of 0 corresponds to a straight position, and a value of 1 corresponds to full right turn. The second element of the array controls the gas pedal, where a value of -1 corresponds to no gas, and a value of 1 corresponds to full throttle. The third element of the array controls the brake pedal, where a value of -1 corresponds to no brake, and a value of 1 corresponds to full braking force.

The next step I take is to play five episodes, where I randomly choose the action. The result is this:

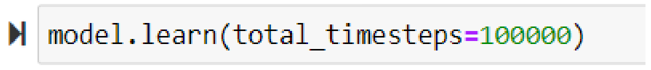


```
Track generation: 1201..1505 -> 304-tiles track
Episode:1 Score:-33.9933993399345
Track generation: 1162..1457 -> 295-tiles track
Episode:2 Score:-31.972789115646663
Track generation: 1115..1398 -> 283-tiles track
Episode:3 Score:-32.62411347517772
Track generation: 1134..1424 -> 290-tiles track
Episode:4 Score:-30.795847750865395
Track generation: 1048..1314 -> 266-tiles track
Episode:5 Score:-28.301886792453143
```

In the snip above when we play the game randomly, the scores of each episode are very low, they are all negative.

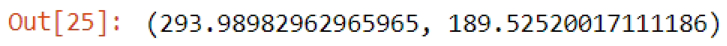
I trained the agent using the "PPO" model and "CnnPolicy" policy. I chose the deep reinforcement learning algorithm. For discrete action space, I found that PPO can perform well.

Because of my low computational resources, chose to train my agent for 100000 steps.



```
model.learn(total_timesteps=100000)
```

When testing the agent, the result is the following:



```
Out[25]: (293.98982962965965, 189.52520017111186)
```

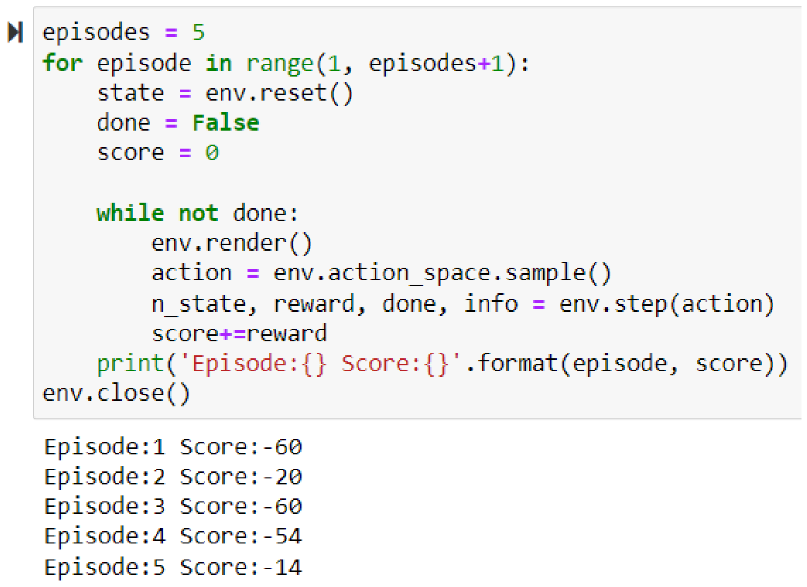
The first number represents the average score for ten episodes and the second number represents the standard deviation. When rendering the environment, it can be clearly seen that 100000 steps is not enough. The car moves much better compared to the randomly selected actions, but there is definitely room for improvement.

4. Custom Environment

Description of custom environment:

- observation space: temperature readings between 0 and 100 degrees.
- Action space: Discrete(2): increase temperature by 1 degree or decrease temperature by 1 degree.
- The RL agent receives a reward of 1 if the temperature falls within the range of 37 to 39 degrees; otherwise, the reward is -1.
- The shower length is set to 60 seconds, and each time an action is taken, it decreases by one second.
- The initial state is represented by a temperature reading of 38 degrees, with a variance of plus or minus 3 degrees.

When randomly playing the game, the results are quite low:



```
episodes = 5
for episode in range(1, episodes+1):
    state = env.reset()
    done = False
    score = 0

    while not done:
        env.render()
        action = env.action_space.sample()
        n_state, reward, done, info = env.step(action)
        score+=reward
    print('Episode:{} Score:{}'.format(episode, score))
env.close()
```

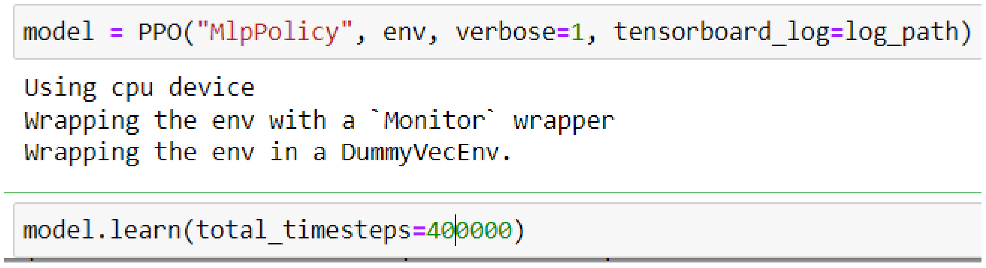
```
Episode:1 Score:-60
Episode:2 Score:-20
Episode:3 Score:-60
Episode:4 Score:-54
Episode:5 Score:-14
```

All scores are below zero.

Let’s see if it improves when training an agent to play the game.

Because the environment is very simple, I am going to use the MlpPolicy and PPO model.

I am training the agent for 40000 steps and creating a tensorboard for visualization purposes.

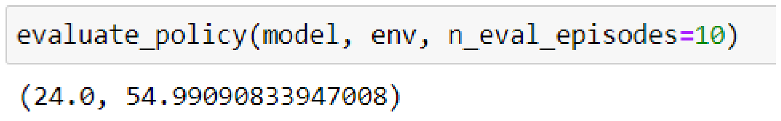


```
model = PPO("MlpPolicy", env, verbose=1, tensorboard_log=log_path)
```

```
Using cpu device
Wrapping the env with a `Monitor` wrapper
Wrapping the env in a DummyVecEnv.
```

```
model.learn(total_timesteps=400000)
```

Upon evaluation, the results are much better when using the agent compared to when playing the game by performing random actions. Now, the score is 24 on average.



```
evaluate_policy(model, env, n_eval_episodes=10)
```

```
(24.0, 54.99090833947008)
```

5. Conclusion

By doing this assignment I interacted with two gym environments: Breakout and Racing Car. First, I played the games by taking random actions and then by using an agent. This allowed me to see that Reinforcement Learning works, by seeing impressive improvements. Additionally, I experimented with two Reinforcement Learning models: A2C and PPO and two policies: CnnPolicy and MlpPolicy.

I also learned about different types of spaces: Box, Tuple, Dict, Multibinary, MultiDiscrete and when to use them.

A next step I can take regarding Reinforcement Learning is to compare the scores on one environment when using different models and/or policies, to see the differences. Then, to learn more, I could try hyperparameter tunning and building a more detailed environment, for example using PyGame.