UNIVERSITY
SCHOOL OF SCIENCE & TECHNOLOGY

Lecture 6
LAB 2
Dynamic programming
Group practice

jmanero@faculty.ie.edu

MBD-EN2024ELECTIVOS-MBDMCSBT_37E89_467614

Contents

- Practice 2 – Dynamic Programming
- Objectives of the Group Practice
- Groups?
- What to do
- Presenting the Group Practice to the Class

Practice 2
Dynamic Programming

Practice 2

Value iteration vs policy iteration

Advantages of Value Iteration:

1. **No need to explicitly manage the policy during iterations**: Since value iteration focuses directly on the value function and not on the policy, the algorithm is often simpler and easier to implement. You don’t need to worry about separate policy evaluation and improvement steps, as you do with policy iteration.
2. **Optimal Policy at the End**: As you noted, **value iteration** provides the optimal policy once the value function has converged. This means that, while you might need more iterations to reach convergence, once you're done, you can immediately extract the optimal policy.
3. **Simplicity**: The algorithm's structure is relatively straightforward: you update the value function and repeat until it converges, then use that value function to determine the policy. This can be easier to understand and implement compared to policy iteration, which involves both policy evaluation and policy improvement steps.
4. **Efficient when state space is small**: For problems where the state space is not huge, value iteration can often converge quickly and is straightforward to implement. It's a good choice if you're looking for a quick and direct solution for relatively small problems.

Disadvantages or When It Might Not Be as Fast:

1. **Slower convergence in larger state spaces**: In cases where the state space is large or has many possible actions, value iteration can take a long time to converge. This is because the value function is updated for every state in each iteration, and it might require many iterations to converge to a sufficiently accurate value.
2. **Computational Cost**: Although the policy is derived at the end, each iteration requires computing the value for all states, which can be computationally expensive in larger state spaces. Policy iteration may, in some cases, converge more quickly in terms of the number of iterations, as it often requires fewer iterations to find the optimal policy.

The document

Practice 2
Dynamic Programming
Policy Iteration and Value Iteration

Learning Objectives

The objective of this Practice is to develop a Policy iteration and a Value iteration in the Russell's Grid Universe that we developed in Practice 1 ([RN10]). In this Practice we will learn how to apply the two basic reinforcement loops for Dynamic Programming in this world. These Loops can be applied to any other world (Activity 3).

I provide with the Practice_2_solved.ipynb file. You don't need to do activity 1 and 2, but you must run the file Practice_2_solved.ipynb in your environment. You need to answer the questions in Activity 3 and solve the FrozenLake using dynamic programming (follow the hint).

1 Files for this Practice

```
015_DProgramming_Russells_World-BASELINE.ipynb
015_DProgramming_Taxi.ipynb
```

Background on Rusell's Grid World

Russell's World is a grid-based environment where an agent must navigate from a start position to a goal position while avoiding obstacles and possibly collecting rewards. The environment is defined by:

- **Observation space (S)**: The set of all possible positions in the grid.
- **Action space (A)**: The set of possible moves (e.g., left, right, up, down).
- **Rewards (R)**: Values received after moving from one state to another.
- **Transitions (T)**: The probabilities of moving from one state to another given an action.
- **Agent** starts in position (1,1)

This environment is based on a universe of a 3x4 board with a set of cells and two terminal states. Beginning in the start state, it must choose an action at each time step. The interaction terminates when the agent reaches one of the goal states (+1 or −1), one is a positive end the other a negative one.

3 Objectives

1. Finish a complete policy iteration code with Russell's World environment
2. Complete a value iteration with Russell's World environment
3. Develop a Dynamic Programming approach for Frozen Lake environment

Starting Point

```
v = zeros

Repeat

    v = policy_evaluation
    new_policy = policy improvement(v)
    Delta = new_policy  - policy
    policy = new_policy

until Delta < theta
```

Complete Policy Iteration (Sutton)

1. Initialization

$V(s) \in \mathrm{R}$ and $\pi(s) \in \mathcal{A}(s)$ arbitrarily for all $s \in \mathcal{S}$

2. Policy Evaluation

Given:

- A Markov Decision Process (MDP) with states S, actions A, state transition probabilities $P(s'|s,a)$, and reward function $R(s,a)$.
- A policy $\pi(a|s)$ that specifies the probability of taking action a in state s.

Initialize the value function $V(s) = 0$ for all $s \in S$.
repeat
  for each $s \in S$
    $v \leftarrow V(s)$
    $V(s) \leftarrow \sum_{a \in A} \pi(a|s) \sum_{s' \in S} P(s'|s,a) \left[R(s,a) + \gamma V(s')\right]$
    $\Delta \leftarrow \max(\Delta, |v - V(s)|)$
until $\Delta < \theta$ {where θ is a small positive threshold}
Output: The value function $V(s)$ for the policy π.

3. Policy Improvement

Input: A policy π and value function V.
policy_stable $\leftarrow$ *True*
for each $s \in \mathcal{S}$
  $a \leftarrow \pi(s)$
  $\pi(s) \leftarrow \arg\max_a \sum_{s',r} p(s', r \mid s, a) \left[r + \gamma V(s')\right]$
  if $a \neq \pi(s)$ **then**
    policy_stable $\leftarrow$ *False*
Output: Improved policy π

Practice 2

Activity 1 – Policy Loops

```python
def policy_evaluation(env, policy, theta=1e-8, gamma=1.0):
    """
    Evaluates policy

    Arguments: env, policy, theta and Gamma
    Returns:
          V (value vector)

    """
    V = np.zeros(env.observation_space.n)  # Initialize value function
    while True:
        delta = 0
        for s in range(env.observation_space.n):
            if s == env.invalid_position[0] * env.ncol + env.invalid_position[1]:
                continue
            v = 0
            for a, action_prob in enumerate(policy[s]):
                for prob, next_state, reward, done in env.P[s][a]:
                    v += action_prob * prob * (reward + gamma * V[next_state] * (not done))
            delta = max(delta, np.abs(v - V[s]))
            V[s] = v
        if delta < theta:
            break
    return V

def policy_improvement(env, V, gamma=1.0):
    """
    Computes greedy policy w.r.t a given env and value function.

    Arguments: env, value function, discount factor(gamma)
    Returns:   policy
    """
    policy = np.zeros([env.observation_space.n, env.action_space.n])
    for s in range(env.observation_space.n):
        if s == env.invalid_position[0] * env.ncol + env.invalid_position[1]:
            continue
        q_values = np.zeros(env.action_space.n)
        for a in range(env.action_space.n):
            for prob, next_state, reward, done in env.P[s][a]:
                q_values[a] += prob * (reward + gamma * V[next_state] * (not done))
        best_a = np.argmax(q_values)
        policy[s] = np.eye(env.action_space.n)[best_a]
    return policy
```

Activity 1 – Policy iteration code

```
46
47 def policy_iteration(env, theta=1e-8, gamma=1.0):
48     policy = np.ones([env.observation_space.n, env.action_space.n]) / env.action_space.n
49     i=0
50     while True:
51
52         # [CODE FOR ACTIVITY 1]
53
54     return policy, V
```

Activity 1 – Policy iteration code

Policy Iteration

POLICY ITERATION CODING PROPOSAL

Input: env, theta, gamma.
policy ← np.ones([obs.space, env.action.space])/env.action.space
i = 0
repeat
  $v \leftarrow policy\_evaluation(env, policy, theta, gamma)$
  $new\_policy = policy\_improvement(env, v, gamma)$
  $\Delta \leftarrow \max(\Delta, |v - V(s)|)$
  $policy = new\_policy$
  $i = i + 1$
until $\Delta < \theta$
Output: policy π, V

HINT

$\Delta \leftarrow np.all(np.abs(policy - new\_policy))$

Stochastic Environment

Definition

- A stochastic environment in Reinforcement Learning (RL) is one in which the outcome of an agent's action is not deterministic – that is, taking the same action in the same state may lead to different results with some probability.

Stochastic Effect: If you try to move **right**, there's a chance you'll slip **down** or **stay in place**.

If **is_slippery=True** the player will move in intended direction with probability of 1/3 else will move in either perpendicular direction with equal probability of 1/3 in both directions.

For example, if action is left and is_slippery is True, then:

- P(move left)=1/3
- P(move up)=1/3
- P(move down)=1/3

Solving Stochastic environments

- Agents must **learn expectations**, not absolute rules.
- Algorithms like SARSA or Q-learning estimate **expected value functions**

$$Q(s,a) = \mathrm{E}[\mathrm{R}]$$

- **Exploration** is crucial to properly model the stochasticity.
- In stochastic environments, we don't aim for perfect prediction — we aim for optimal expectation.

Real life examples of stochastic environments

🚙 Real-world Examples:

Scenario	Why It's Stochastic
Robot on slippery floor	Wheels may slip; direction changes unpredictably
Stock trading agent	Market responds probabilistically to actions
Game with dice (e.g., Monopoly)	Player's move depends on dice outcome
Medical treatment agent	Same treatment may affect patients differently

Practice 2
Questions

- What happens if you use is_slippery = True?
- How is the optimal policy with is_slippery=True? Why?
- What is faster Policy iteration or Value Iteration

Questions

- ¿What is best policy of Value iteration?

Comparison. Speed, accuracy, …

Value vs Policy

Method	Iteration Cost	Total Iterations	Convergence speed
Policy iteration	High (each step evaluates full policy)	Fewer	Faster overall
Value iteration	Low (only one calculation of Bellman equation per each state)	Larger	Slower overall but cheaper steps

Questions

- ¿How does Gamma influence results?

Practice 2

Does Gamma influences result?

Gamma $\gamma \in [0,1]$ controls how much future rewards are valued compared to immediate rewards.

Value of Gamma	Performance	Behavior
0	Blind agent	Only cares about immediate reward ignores future
Close to 0	Short-sighted	Prefers quick rewards
0.9-0.99	Far-sighted	Values long-term rewards good for planning
1	Infinite horizon	May not converge, treats all rewards as equivalent

Group Practice

Video in COLAB

- https://github.com/castorgit/RL_course/blob/main/00_LunarLander-COLAB_render.ipynb

Contents

- Objectives of the Group Practice
- Groups?
- What to do
- Presenting the Group Practice to the Class

Objectives

- To perform a complex Reinforcement Learning project in a group setting
- The objective is to understand and then communicate to the class
- All your work must be distilled into a presentation (visuals!!)
- Apply previously learned techniques
- It is a small project that will prepare you for the “real Capstone”

Groups

- You must create the groups
- Everybody works – no tailgating
- Use the different skills wisely
- You must communicate, focus in understanding the problem and the solution

Group Practice / Project Evaluation

4 scores – group scores –

E1: Presentation material, delivery, organization

E2: Background analysis

E3: Methods and algorithms

E4: Conclusions

E5: individual score (Mostly presentation)

Final individual score = E1+E2+E3+E4+E5 < this is what it will go into your grade

Group Score = E1+E2+E3+E4

Group Practice / Project Evaluation

E1: Presentation material, delivery, organization

- How well has been the presentation organized, storytelling, goals, results, conclusions, …

E2: Background analysis

- Data Analysis, findings and structure of analysis

E3: Methods and algorithms

- Tools used for E1 and E2 (algorithms, …)

E4: Conclusions

- Are your findings and conclusions convincing?

E5: Individual Score

- Your presentation Delivery

The Group Project Description

Let's review the complete description
We will land in the moon!!!
or race!

Group Project
Training an agent with DQN & DDQN
Lunar Landing & Car Racing

Learning Objectives

In the Group project, you will implement the famous Deep Q-Network (DQN) and its successor Double DQN (DDQN) on an environment of your choice. You can choose between the Lunar Landing and the Car Racing environment defined in the package Gymnasium. The goals of this assignment are to (1) understand how deep reinforcement learning works when interacting with the pixel-level information of an environment and (2) implement a recurrent state to encode and maintain history.

There is a good resource in the internet that can be used as explanation of this practice. See [Rai23].

1 Files for this Practice

```
00_Lunar_lander.ipynb
030_DQN_CARTPOLE_KERAS.ipynb
031_DDQN_CARTPOLE_KERAS.ipynb
```

2 Background on Lunar Lander

This environment is a classic rocket trajectory optimization problem. According to Pontryagin's maximum principle, it is optimal to fire the engine at full throttle or turn it off. This is the reason why this environment has discrete actions: engine on or off.

There are two environment versions: discrete or continuous. The landing pad is always at coordinates (0,0). The coordinates are the first two numbers in the state vector. Landing outside of the landing pad is possible. Fuel is infinite, so an agent can learn to fly and then land on its first attempt.

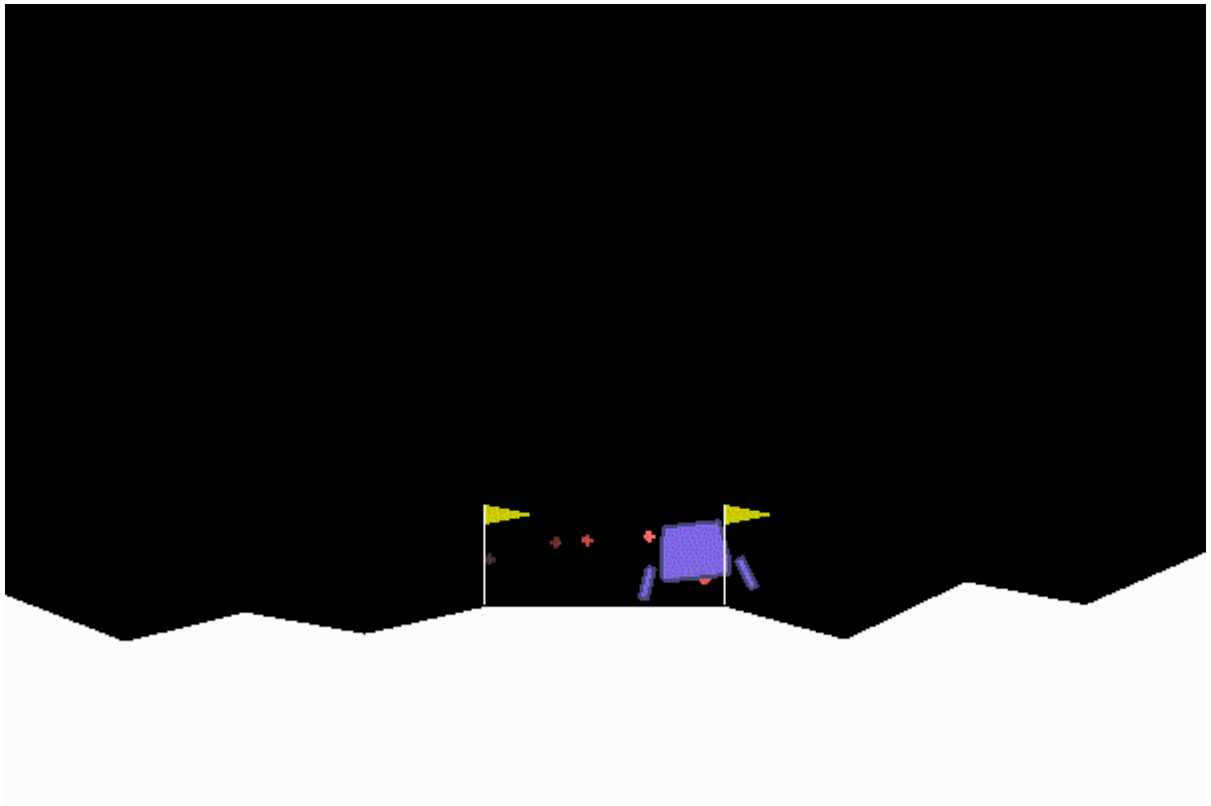
To observe a heuristic landing, see the notebook

```
00_Lunar_lander.ipynb
```

3 Description of the Group Project

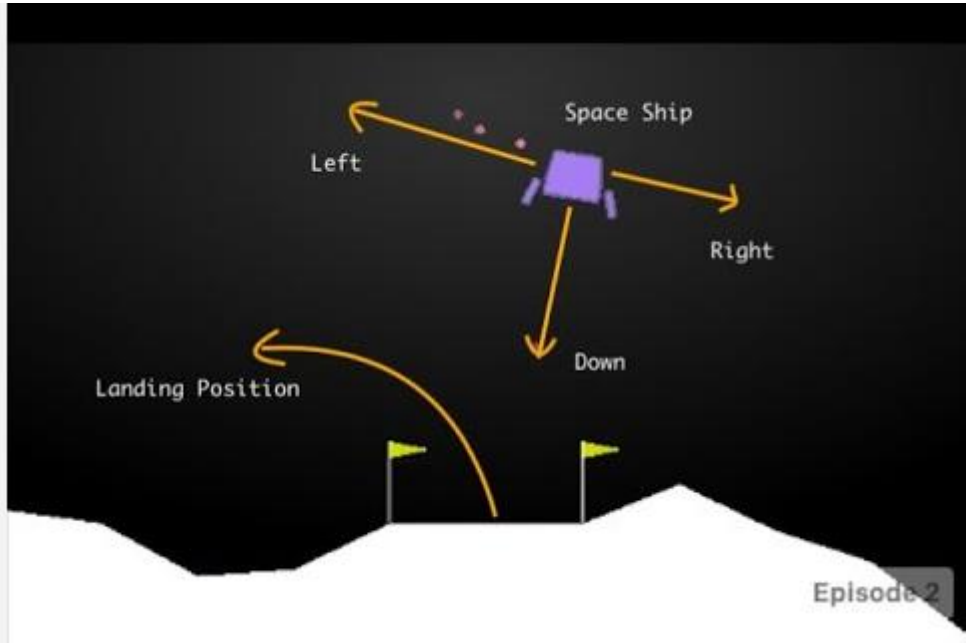
You will implement the Deep Q-Network (DQN) on the game of Lunar Landing using the the Gymnasium environment. The goals of this project are

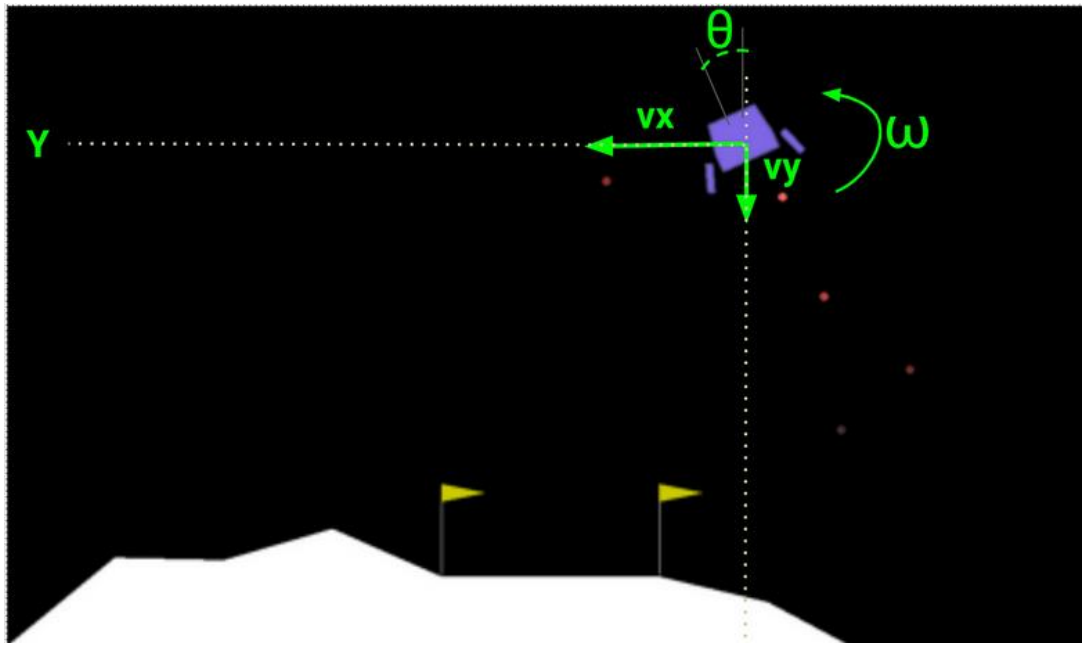
Lunar Lander



https://github.com/castorgit/RL_course/blob/main/00_LunarLander.ipynb

Lunar Lander – Landing in the moon





This environment is a classic rocket trajectory optimization problem. According to Pontryagin's maximum principle, it is optimal to fire the engine at full throttle or turn it off. This is the reason why this environment has discrete actions: engine on or off.

There are two environment versions: discrete or continuous. The landing pad is always at coordinates (0,0). The coordinates are the first two numbers in the state vector. Landing outside of the landing pad is possible. Fuel is infinite, so an agent can learn to fly and then land on its first attempt

Lunar Lander – env characteristics

Action Space

There are four discrete actions available: do nothing, fire left orientation engine, fire main engine, fire right orientation engine.

Observation Space

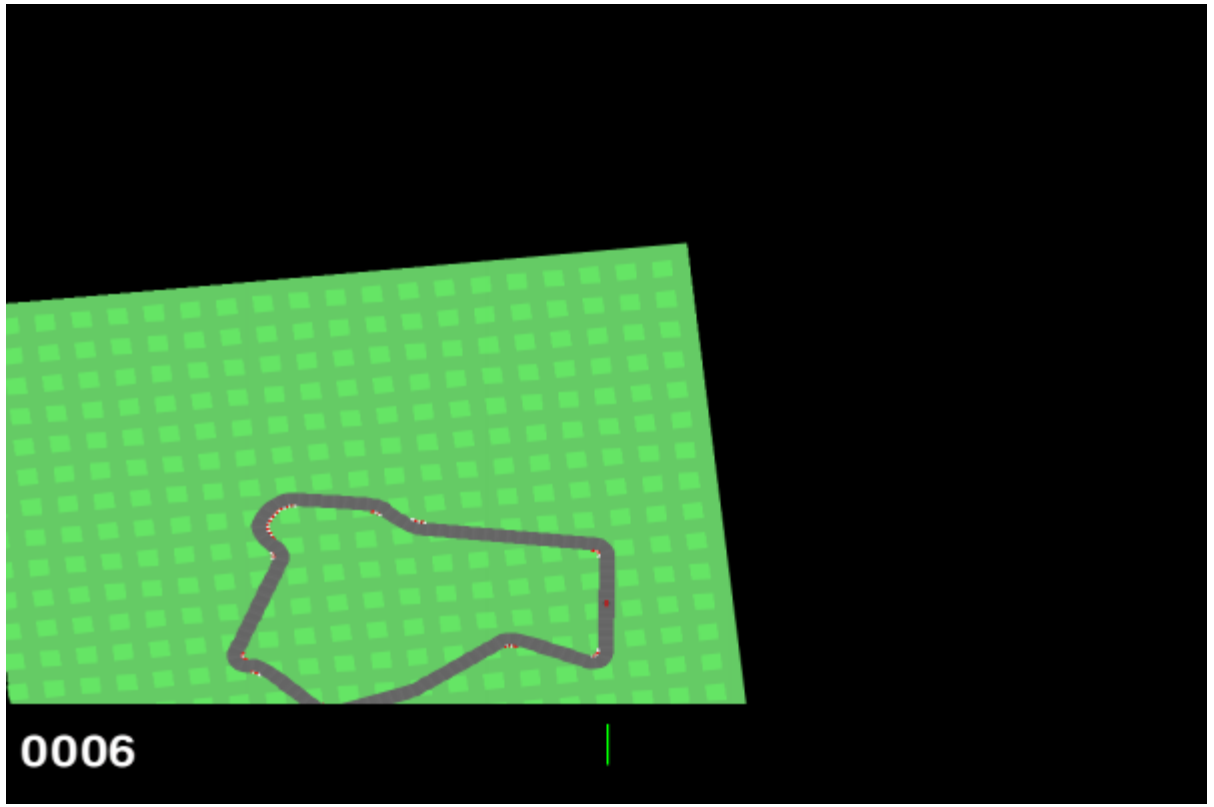
The state is an 8-dimensional vector: the coordinates of the lander in x & y, its linear velocities in x & y, its angle, its angular velocity, and two booleans that represent whether each leg is in contact with the ground or not.

Rewards

Reward for moving from the top of the screen to the landing pad and coming to rest is about 100-140 points. If the lander moves away from the landing pad, it loses reward. If the lander crashes, it receives an additional -100 points. If it comes to rest, it receives an additional +100 points. Each leg with ground contact is +10 points. Firing the main engine is -0.3 points each frame. Firing the side engine is -0.03 points each frame. Solved is 200 points.

Car Racing

Action Space	`Box([-1. 0. 0.], 1.0, (3,), float32)`
Observation Space	`Box(0, 255, (96, 96, 3), uint8)`
import	`gymnasium.make("CarRacing-v3")`



Car Racing

Action Space

If continuous there are 3 actions :

- 0: steering, -1 is full left, +1 is full right
- 1: gas
- 2: braking

If discrete there are 5 actions:

- 0: do nothing
- 1: steer left
- 2: steer right
- 3: gas
- 4: brake

Observation Space

A top-down 96x96 RGB image of the car and race track.

Rewards ¶

The reward is -0.1 every frame and +1000/N for every track tile visited, where N is the total number of tiles visited in the track. For example, if you have finished in 732 frames, your reward is 1000 - 0.1*732 = 926.8 points.

Car Racing

- You need to use a convolutional network as you are interpreting each frame
  - A state/observation is a frame of (96,96,3)
  - It is very similar to Atari Games
- Skip the first 15 frames (is the image of whole racetrack) as they don't add information to the agent
- It will take longer and more complex fine tunning than with Lunar Lander
- Use Keras

END
Session 6