RL211 - HW3

Omri Himelbrand - 200863843
Nitzan Cohen - 203980750

December 20, 2020

Contents

1 Introduction

The solution is implemented in a way that makes it possible to run it as a script or as a module.
The code will explore the environment using SARSA(λ) with ϵ-greedy, using a decaying ϵ.
Using LFA for approximating Q, this is done by using separate weights for each possible action, 3 overall, the weights are saved as a 2D matrix.
We used a 200 maximum steps per episode, and there is an option for a relaxed version with 500 steps limit for episode.
In order to run this code you must have: **numpy** installed, and have an out directory for the output files.

2 Running the solution

2.1 Running as a script

```
usage: hw3.py [-h] [-human] [-gamma G] [-d] [-ms MAX_STEPS]
              [-es EVAL_STEPS] [-png PNG_SUFFIX] [-relax]

AI agent using SARSA lambda and LFA for AI-Gym Mountain Car.

optional arguments:
  -h, --help       show this help message and exit
  -human           use this flag to run human agent
  -gamma G         a float for gamma in [0,1] (default:
                   0.95).
  -d               use this flag to get debug prints
  -ms MAX_STEPS    a int for number of maximum steps for
                   learning.
  -es EVAL_STEPS   a int for number of steps between
                   evaluations.
```

```
  -png PNG_SUFFIX  a suffix for png out file
  -relax           use this flag to use 500 steps episodes
```

2.2 Running as module

```
import hw3

hw3.main(gamma=1)
```

3 Code details

3.1 Global variables

DEBUG: A boolean which is initialized to False.
MAX_STEPS: An integer to indicate the number of steps for the entire learning process, initialized to 120000.
EVAL_STEPS: An integer to indicate the number of steps between running evaluations simulations, initialized to 1000.
RELAXED: A boolean value, indicating whether or not to run the relaxed version (500 steps limit per episode).
P_CENTERS: A list containing the centers of the positions' Gaussians.
V_CENTERS: A list containing the centers of the velocities' Gaussians.
CENTER_PRODUCTS: $P\_CENTERS \times V\_CENTERS$.
SIGMA_P: variance/std of position.
SIGMA_V: variance/std of velocity.
COV: the co-variance matrix.
INV_COV: COV^{-1}
P_I: float value of interval size for centers of position.
V_I: float value of interval size for centers of velocity.

```
#initializing the last few globals can be done using these functions
def init_intervals(Ip=0.18,Iv=0.014):
    global P_I,V_I
    P_I = Ip
    V_I = Iv

def init_covariance(sigma_p=0.04,sigma_v=0.0004):
    global SIGMA_P,SIGMA_V,COV,INV_COV
    SIGMA_P = sigma_p
    SIGMA_V = sigma_v
    COV = np.diag([SIGMA_P,SIGMA_V])
    INV_COV = np.linalg.inv(COV)

def init_centers(p_half=4,v_half=4):
    global P_CENTERS,V_CENTERS,CENTER_PRODUCTS
    P_CENTERS = [(i)*P_I for i in range(-p_half,p_half)]
    V_CENTERS = [(i)*V_I for i in range(-v_half,v_half)]
    CENTER_PRODUCTS = np.array(list(product(P_CENTERS,V_CENTERS)))
```

3.2 The functions

```
def init_env(max_steps=200):
```

Calls gym.make, sets the max episode steps for the created environment and returns it.

```
def set_debug(value):
```

Sets the global variable DEBUG to value.

```
def set_max_steps(value):
```

Sets the global variable MAX_STEPS to value.

```
def human_agent(env):
```

Prompts user to pick action for the next step of simulation, used for mostly for debugging.
Return value is a action (int).

```
def evaluate(env,w,gamma,episodes_num=100,show=False):
```

Given the current weights (w), and the rest of the arguments seen in the signature, evaluates v_0^π, using a MC-like evaluation.
This function returns value v_0^π.

```
def apply_policy(Qhat,actions,eps=0):
```

This function apply the policy defined by the weights using ϵ-greedy scheme on $\hat{Q}$ (in case *eps¿0*), generating random number, if it is less than ϵ uniform random choice of action from action space, else argmax.

```
def sarsa(env,w,gamma,Lambda,alpha,actions,eps,max_step=5000,iters=0,epsilon_decay=0.99,min_eps=0.1):
```

This function does SARSA(λ) with LFA, computing $\hat{Q}_a$ given the current state and weights, the weights are updated in-place, using tiled elegibility trace and gradient decent, using decaying ϵ, with extra decay according to RBED (Reward Based Epsilon Decay). The decay is stopped at a minimal value for ϵ in order to enable some exploration after a long time.
The

```
def run_simulation(env,w=None,human=False,show=True):
```

resets env to initial state, then runs simulation either using the given policy (by weights) or using a human agent.

```
def centers_distance(s:np.ndarray):
```

given a state returns vector of distances from the CENTER_PRODUCTS vector.

```
def theta(x:np.ndarray):
```

given the result of *centers_distance* as x, computes the features of the state given to *centers_distance*.

```
def tiles(x:np.ndarray):
```

returns an indicator vector indicating which of the elements of CENTER_PRODUCTS have a minimal distance to the state (x is given by the return value of *centers_distance*), this function is used for the eligibility trace since the states are continuous.

```
def init_weights(nA=3,seed=27021990):
```

Initializes the weights randomly, seed is used only in debug mode.

```
def QApproximation(theta:np.ndarray,w:np.ndarray):
```

given θ and w, computes $\hat{Q}$.

```
def learn_policy(env,actions,gamma,Lambda,alpha):
```

This function runs the whole learning process, running sarsa for EVAL_STEPS, then running evaluation.

After each evaluation the V_{init}^π is saved with the number of total steps taken so far, and we keep track of the best w according to the evaluations so far, keeping it for return.

The return value of this function is: x - array of step counts, y - array of V_0^π collected, and the best w.

```
def main(gamma=1,human=False):
```

This function is called when running the code as a script, but can be used as seen above, this function does the following:

- calls "init_env","init_covariance","init_intervals", and "init_centers"
- if human flag is set, runs a single simulation with a human agent.
- else calls "learn_policy", after running a simulation using a previous learned weights (if such exists).
- for each run, calls "run_simulation" using the returned w
- Also for each run, saves the x and y values returned with key-label
- After running all runs and collecting all the x and y values, calls plot_results function

4 Plots

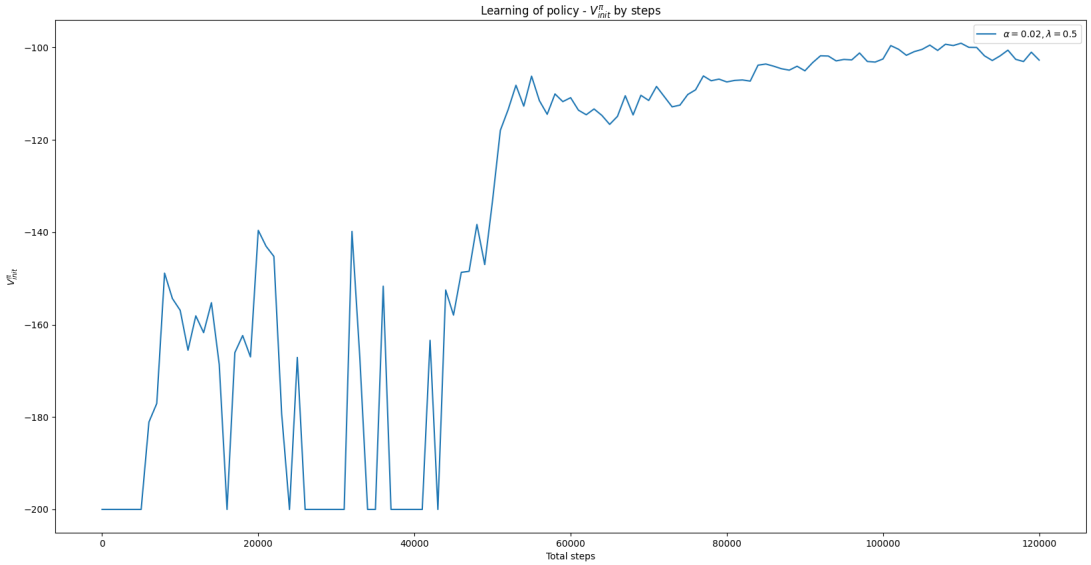

Figure 1: Original problem 200 steps limit per episode

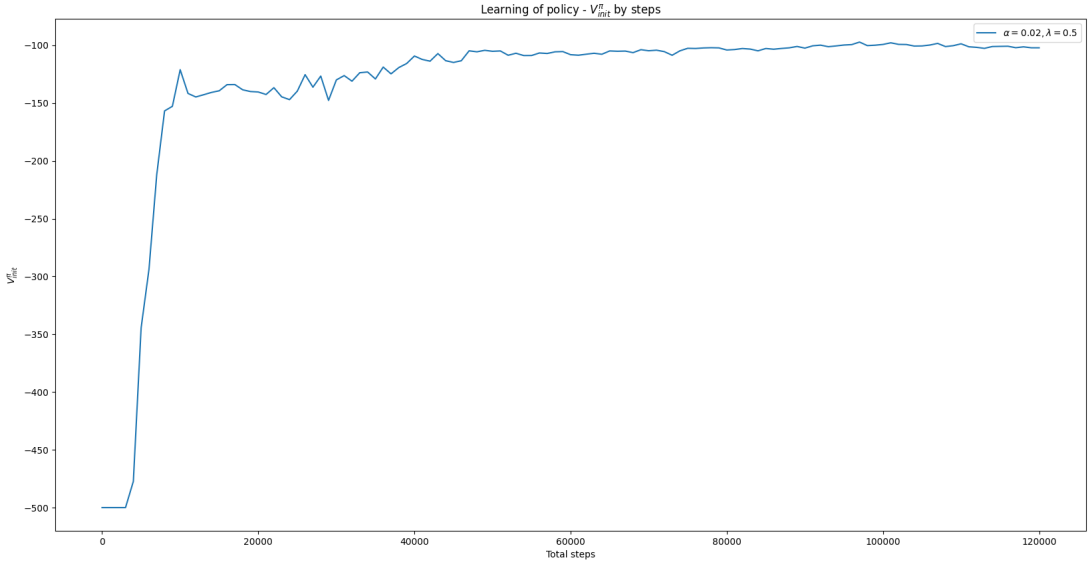

Figure 2: Relaxed problem 500 steps limit per episode