Project Machine Learning
Task 3: Reinforcement Learning
9 group

Tarelkin Evgenii
Steba Oxana

2022

.

1 Task 1

1.1 a

Reinforcement learning (RL) is a field of machine learning in which learning is carried out through interaction with the environment. There is a learner and a decision-maker called the agent, and the surrounding with which it interacts is called the environment. This is purposeful training, during which the trainee (agent) does not receive information about what actions should be performed, instead he learns about the consequences of his actions. Rewards can be positive or negative depending on their actions. Based on this, the agent has the intention to perform actions in which he receives a positive reward. Thus, learning turns into a trial and error process.

Let's try to draw an analogy where a pet represents an agent. In such a situation, a yummy received for a correctly executed command is a positive reward, and the absence of a yummy for non—fulfillment or incorrect execution of the command is a negative one.

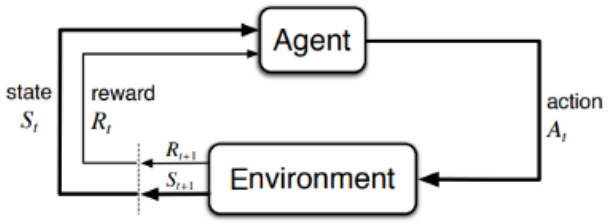

Figure 1: Agent-environment interaction loop.

Based on the above, we can conclude that the main characters of RL are the agent and the environment. The agent is a software program that makes intelligent decisions as we indicated above, in fact, it plays the role of a trainee in RL. As we can see in figure 1, these agents interact with the environment by actions and receive rewards depending on their actions. Everything an agent interacts with is called an environment. The environment is the demonstration of the problem to be solved. It includes everything that is external to the agent.

The agent interacts with the environment by perceiving a reward signal from the environment, a number that tells him how good or bad the current state of the world is.

$$r_t = R(s_t, a_t, s_{t+1})$$

Rewards are numerical values that an agent receives for performing some action in some state of the environment. The numeric value can be positive or negative depending on the actions of the agent. The agent's goal is to maximize

his total sum of rewards, called a return.
According to figure 1, the surrounding world with which the agent is in contact has a state. State s is the position of the agents at a specific time-step in the environment. As is customary in RL, we almost always represent states by a real vector, matrix, or higher-order tensor. For example, we can represent the state of a robot in the form of its connection angles and speeds.
It is also accepted in RL that different environments allow for different types of actions. In other words, the agent performs an action by moving from one state to another.
The value function determines how well an agent is in a particular state, in order to determine how well he will be in a particular state, it must depend on some actions that he will take. Its value is equal to the expected total result obtained by the agent, starting from the initial state. This is where policies comes in. The policy defines what actions to perform in a particular state s. At the same time, the way in which the agent chooses the action that it will perform depends on the policy. Imagine that you want to get from home to university. There are many possible route options - some paths are short, others are long. These paths are called "policies" because they represent a way to perform an action to achieve a goal.
Mathematically, a policy is defined as follows :

$$\pi(a \mid s) = P[A_t = a \mid A_t = s]$$

The Bellman equation helps us find optimal policies and value functions. Since the policy changes with experience, therefore the value functions depending on different policies will be different. The optimal value function is the function that yields the maximum value compared to all other value functions. The Bellman equation can be decomposed into two parts: the immediate reward $R_{[t+1]}$ and the discounted value of successor states. Mathematically, the Bellman equation defines as

$$q_*(s, a) = E\left[R_{t+1} + \gamma \max q_*(S_{t+1}, a) \mid S_t = s, A_t = a\right]$$

In a Markov Decision Process we have an actual agency that makes decisions or takes actions. The crucial difference between the Bellman equations for policy value functions and optimal value functions is the absence or presence of max over actions.
Since the policy depends on the actions of the agent, the policy may vary, here are the main ones:
1. The On-Policy Action-Value Function, $Q^\pi(s, a)$, that gives the expected return if start in state s, take an arbitrary action a (which may not have come from the policy), and then forever after act according to policy :

$$Q^\pi(s, a) = E_{\tau \to \pi}[R(\tau) \mid s_0 = s, a_0 = a]$$

2. The Optimal Value Function $V^*(s)$, which gives the expected return if you start in state s and always act according to the optimal policy in the environment:

$$V^*(s) = \max_\pi E_{\tau \to \pi}[R(\tau) \mid s_0 = s]$$

Optimal Policy, is one which results in optimal value function. There can be more than one optimal policy in an MDP. And all optimal policies achieve the same optimal value function and optimal state-action Value Function.
We find an optimal policy by maximizing over *q*(s, a)*.

$$\pi_*(a|s) = \begin{cases} 1 & \text{if } a = \underset{a \in \mathcal{A}}{\text{argmax}}\ q_*(s,a) \\ 0 & \textit{otherwise} \end{cases}$$

This shows that for a state s we pick the action a with probability 1, if it gives us the maximum *q*(s,a)*. So, if we know *q*(s,a)* we can get an optimal policy from it.

1.2 B, C, D

Since in this task γ is fixed (equal to 1) and it was decided to use $\epsilon = 0.1$. We have selected the best value for the α parameter for the SARSA and Q-learning algorithms:
Q-learning $\alpha = 0.5$ number of finish = 1,
SARSA $\alpha = 0.5$ number of finish = 1;
Q-learning $\alpha = 0.6$ number of finish = 1,
SARSA $\alpha = 0.6$ number of finish = 0;
Q-learning $\alpha = 0.7$ number of finish = 1,
SARSA $\alpha = 0.7$ number of finish = 1;
Q-learning $\alpha = 0.8$ number of finish = 1,
SARSA $\alpha = 0.8$ number of finish = 0;
Q-learning $\alpha = 0.9$ number of finish = 0,
SARSA $\alpha = 0.9$ number of finish = 1;
Q-learning $\alpha = 1.0$ number of finish = 1,
SARSA $\alpha = 1.0$ number of finish = 2.

It is obvious from the results that it is not advisable to take α greater than one, since the float64 field quickly overflows, but for the sake of curiosity, the following experiment was carried out:
Q-learning $\alpha = 1.1$ number of finish = 1
SARSA $\alpha = 1.1$ number of finish = 0;
Q-learning $\alpha = 1.2$ number of finish = 0,
SARSA $\alpha = 1.2$ number of finish = 1;

Based on this, it was decided to choose $\alpha = 1$

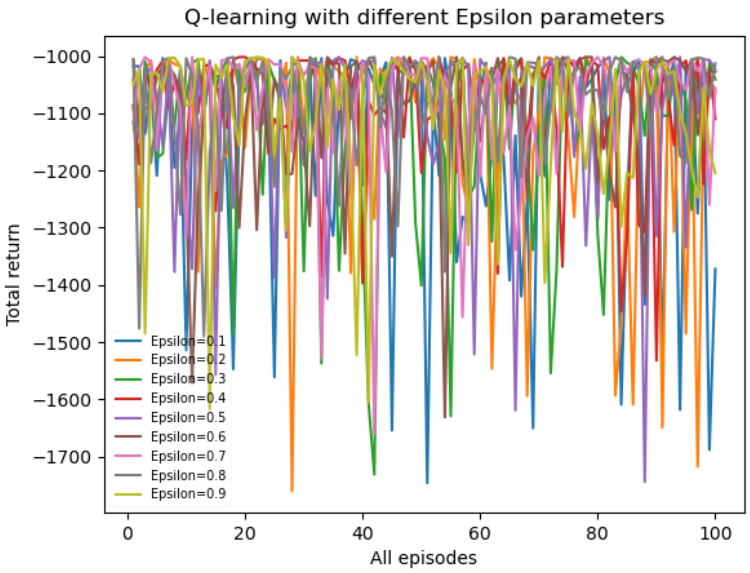

Figure 2: Q-learning with different ϵ (100 experiments)

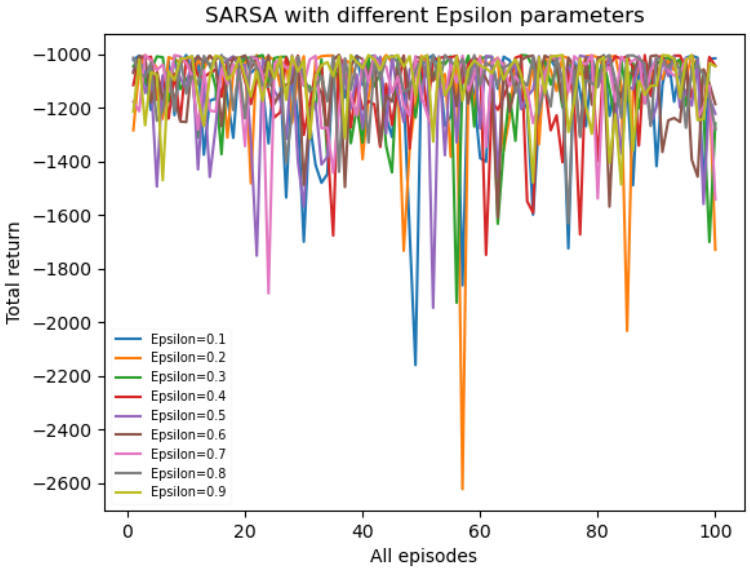

Figure 3: SARSA with different ϵ (100 experiments)

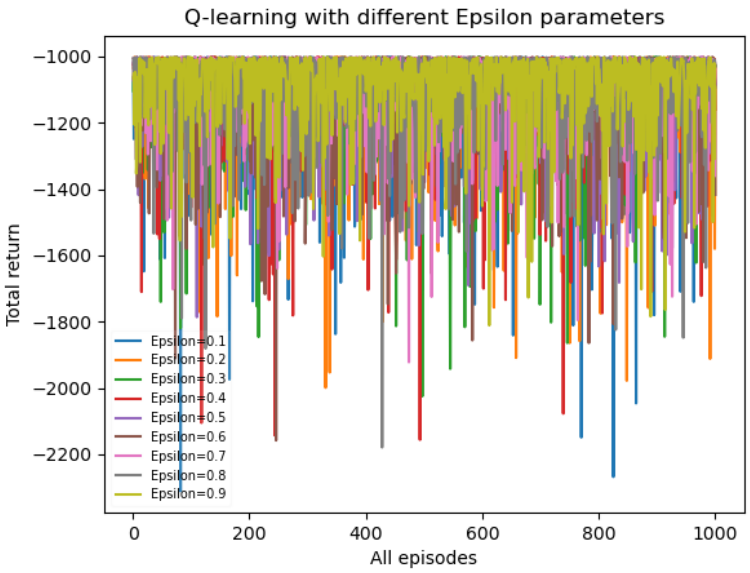

Figure 4: Q-learning with different ϵ (1000 experiments)

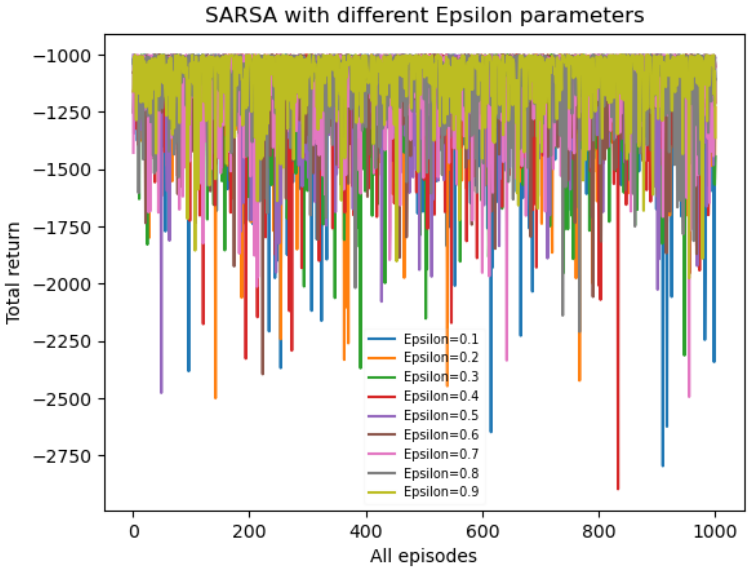

Figure 5: SARSA with different ϵ (1000 experiments)

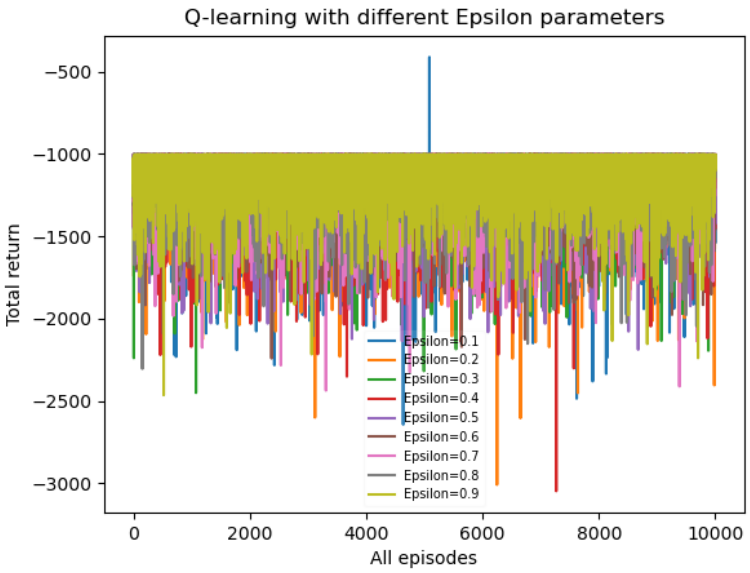

Figure 6: Q-learning with different ϵ (10000 experiments)

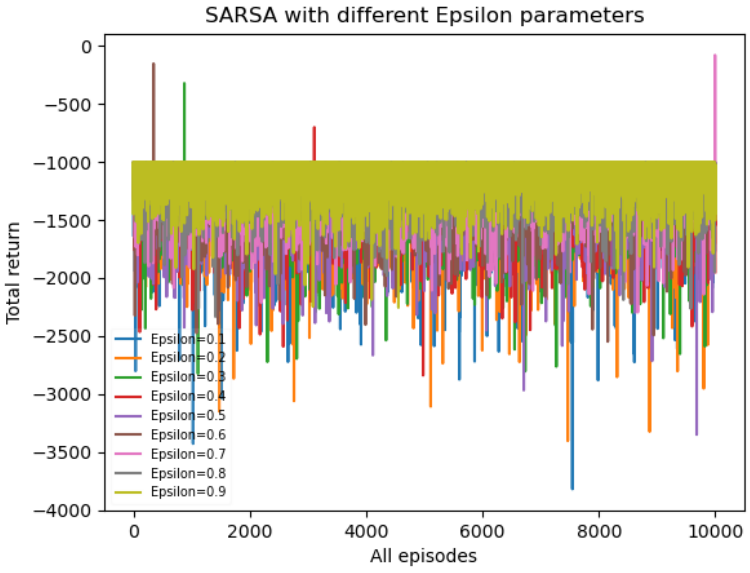

Figure 7: SARSA with different ϵ (10000 experiments)

In figures 8 and 9, the path is drawn based on the fact that $\epsilon = 0$, since if ϵ is not equal to 0, then the path will be partially random.

If $\epsilon = 1$, then the path is chosen absolutely randomly.

And the results for $0 \leq \epsilon \leq 1$ are not much different for both algorithms.

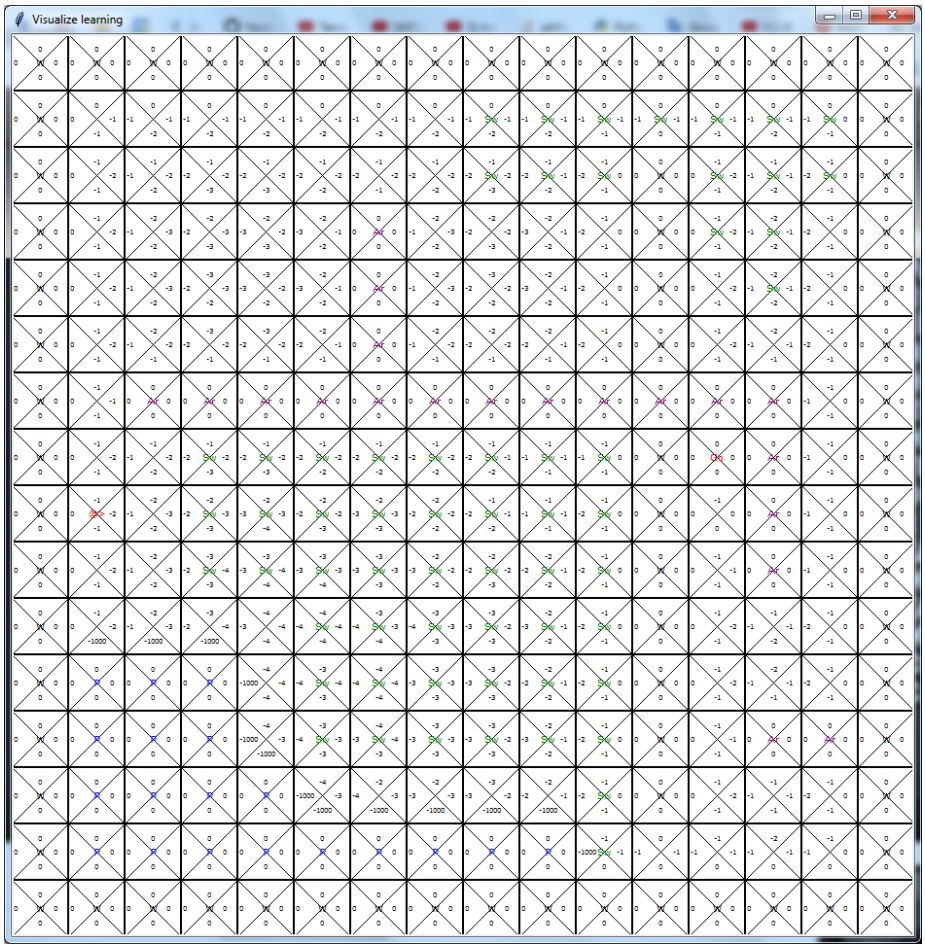

Figure 8: Q-learning experiments to draw the agents path (10000 experiments)

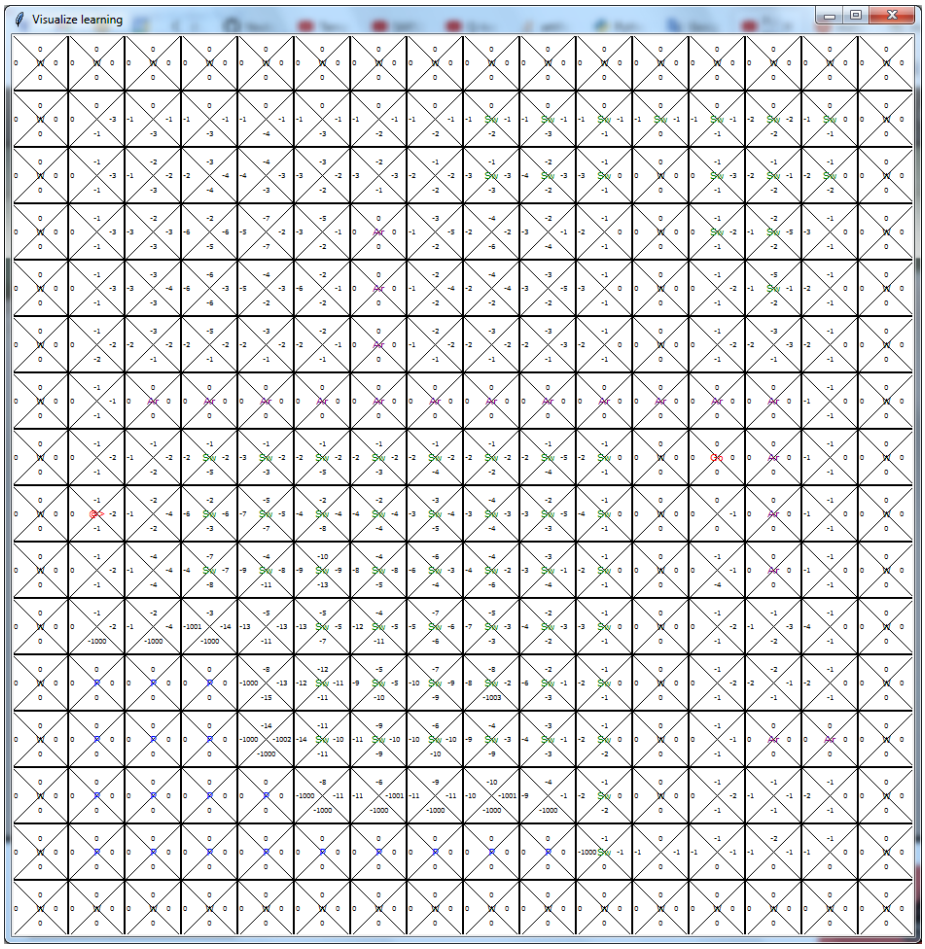

Figure 9: SARSA experiments to draw the agents path (10000 experiments)

With ϵ=0, the algorithms will get stuck next to the wall, since the modification of Q value and V value occurs only after the move.

Figure 10: Example of an agent getting stuck next to a wall

An example of getting stuck in SARSA: since the agent walks into the wall at the beginning, then he gets a "new state" (it does not change when walking into the wall). And the future step is calculated based on the "new state",

therefore, he tries to always walk to the left.
An example of getting stuck in Q-learning: Q-learning always chooses the maximum Q value (direction) available in the current state. Therefore, the agent will try to walk into the wall constantly in this case.
In both algorithms: Since the state of the agent does not change, it does not go to the wall and the modification of the state near the wall does not occur, therefore, stucks occur.

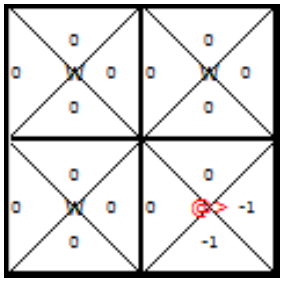

Figure 11: An example of an agent stucks in a corner, working for SARSA and Q-learning with ϵ=0

1.3 E

1 policy:
Assumption: Since the finishing cell does not give any positive reward and its reward is the same as that of a regular cell, the agent rarely reaches the finish cell . The experiment confirmed our hypothesis (experiments given above).
Since the finish is the same cell as a regular cell with the same reward - 1, there is no training (or very weak due to pit cells).
This can be solved by adding a positive reward when standing at the finish cell , as done in policy 2.

2 policy:
Assumption: with this policy, the agent will come to the finish very often, since the finish has a positive payoff, this is very important.
But this policy can be improved by making a reward of 1 at each step, and a reward of 1000 at the finish, the opposite of the Pit cell reward (-1000).
Adding a reward of 1 on each step will allow the agent to see where he is walking, and therefore a big finish is needed more reward per step, which will allow him to pave the path to the finish.

Algorithm	epsilon	gamma	alpha	Reached the finish cell	Figure
Q-learning	0.1	1	1	613	12
Q-learning	0.1	0.9	1	8255	
Q-learning	0.1	1	0.9	1168	
Q-learning	0.1	0.9	0.9	7987	
Q-learning	0	0.9	0.9	9987	13
SARSA	0.1	1	1	631	14
SARSA	0.1	1	1	8184	
SARSA	0.1	1	0.9	1149	
SARSA	0.1	0.9	0.9	7812	
SARSA	0	0.9	0.9	9987	
SARSA	0	0.9	1	9987	

Table 1: 2 policy results.

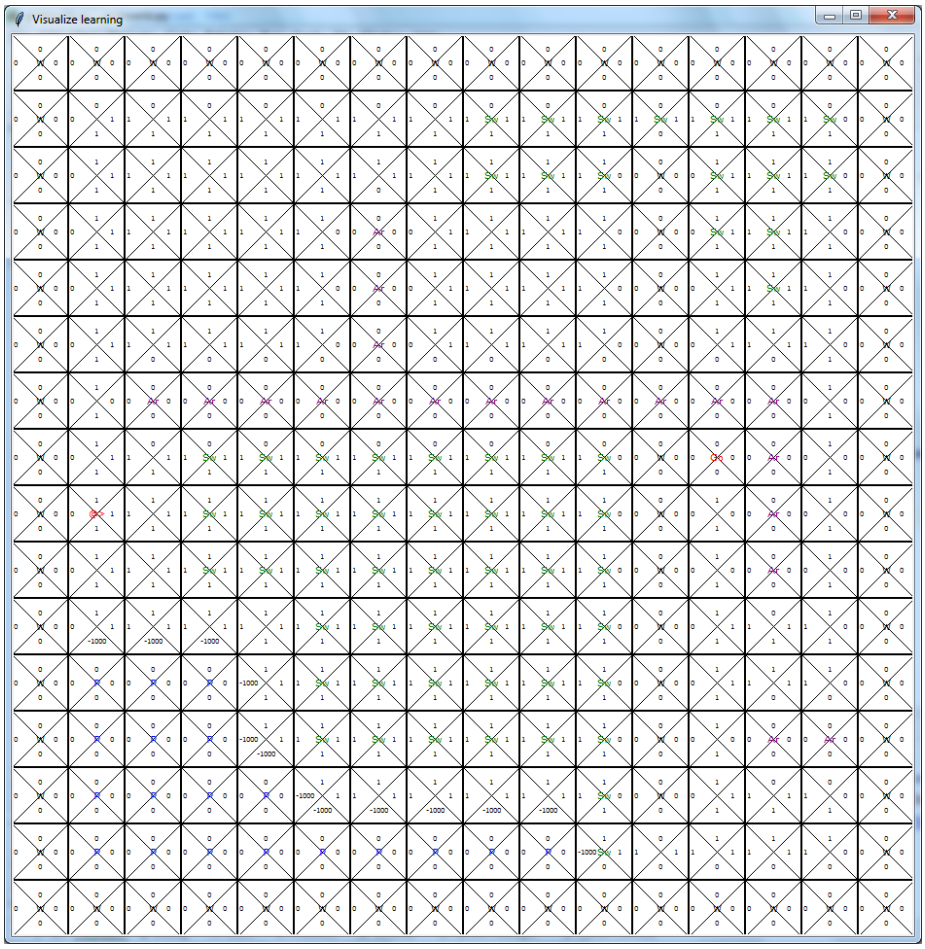

Figure 12: Q-learning (reached the finish cell 613 times)

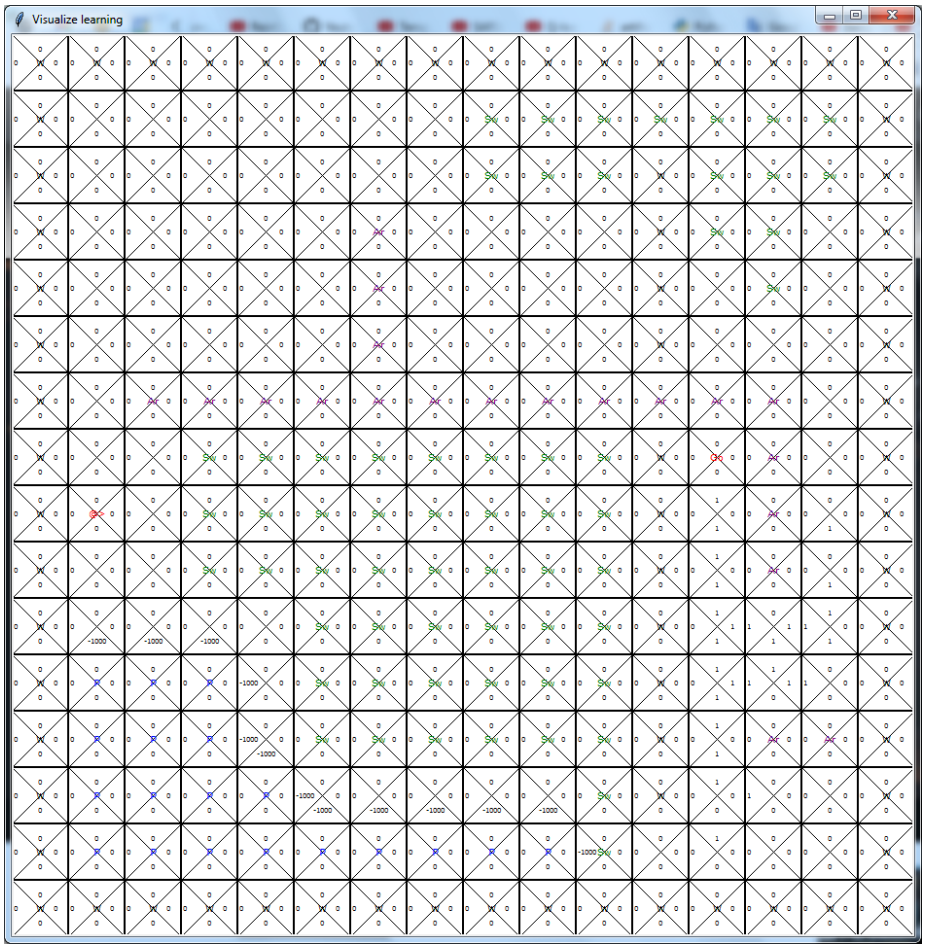

Figure 13: Q-learning (reached the finish cell 9987 times)

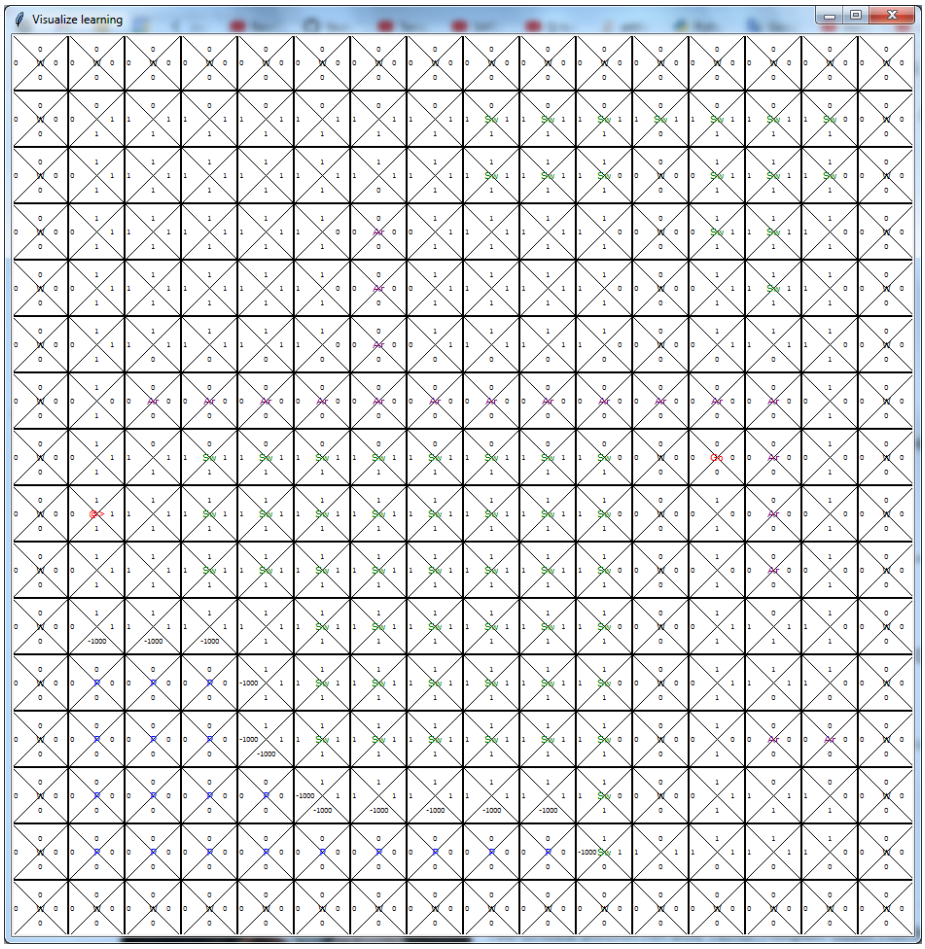

Figure 14: SARSA (reached the finish cell 631 times)

The assumption about the second policy turned out to be correct.

3 policy:

Assumption: Since the Manhattan distance is negative, then most likely this policy will not encourage the agent moving away from GoalCell in the other direction.

The assumption regarding the 3rd policy was confirmed, but we managed to find parameters that increase the number of agent arrivals at the finish line.

It was also confirmed that this policy makes it so that the agent constantly walked away from the target and the agent constantly fell into Pit along the coordinates x=1, y =11 with α tending to 1 and γ tending to 1.

But with α=1 and γ=0.1, the agent began to reach the finish cell better, but

Algorithm	epsilon	gamma	alpha	Reached the finish cell	Figure
SARSA	0.1	0.9	0.9	0	15
Q-learning	0.1	0.9	0.9	0	16
SARSA	0.1	1	1	0	
Q-learning	0.1	1	1	0	
SARSA	0.1	0.1	0.9	313	
Q-learning	0.1	0.1	0.9	321	

Table 2: 3 policy results.

it was difficult for him to overcome the wall, since walking down the wall has lower Q values than walking up the wall, since the target coordinate suggests walking higher.

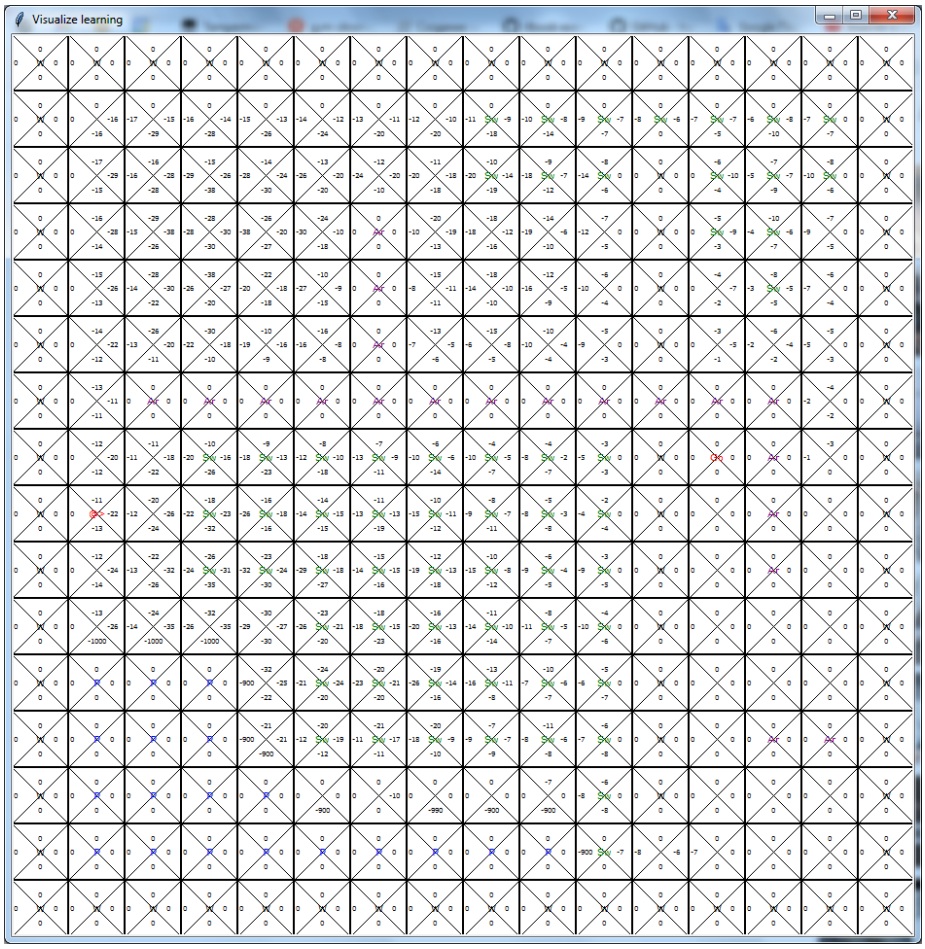

Figure 15: SARSA (reached the finish cell 0 times)

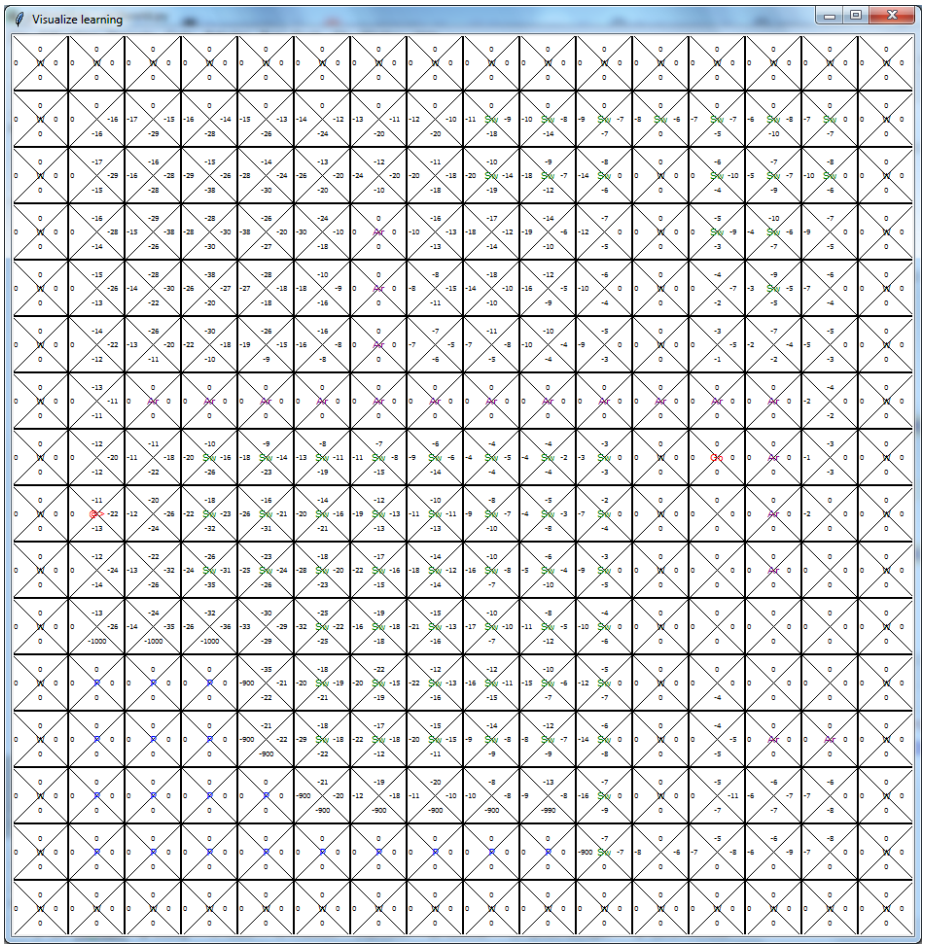

Figure 16: Q-learning (reached the finish cell 0 times)

1.4 F

According to the Bellman theorem, each optimal solution (optimal trajectory τ^*) is composed of optimal partial solutions as well, so the optimal path was chosen and the Bellman theorem was calculated from finish to start.
For task 1.e.1 with optimal policy, the return is equal to -1.0 with γ equal to 0.0,
For task 1.e.2 with optimal policy, the return is equal to 1.0 with γ equal to 1.0,
For task 1.e.3 with optimal policy, the return is equal to -11.0 with γ equal to 0.0.

Testing optimal policies:
For all tests α=1 and $\epsilon = 0.1$
For the standard policy, 50,000 cycles were taken to obtain a result comparable

to task B, C, D.

Standard policy (1.e.1)
SARSA - Reached Finish 23
Q-learning - Reached Finish 28

For policies E2 and E3, 10,000 cycles were taken to make the result comparable to task E.

Changing policy to 1.e.2
SARSA - Reached Finish 658
Q-learning - Reached Finish 661

Changing policy to 1.e.3
SARSA - Reached Finish 88
Q-learning - Reached Finish 85

Applying a gamma equal to the optimal policy for various reward functions showed:
- in some cases, the result is better when the gamma was equal to the optimal policy (in the case of the e1 policy);
- in some cases, the result is better when the gamma tends to the optimal policy (in the case of policies e2, e3).
This was proven in task E when selecting parameters for different policies.
It was also concluded that the value of the Alpha - learning rate parameter affects the result less than the value of the Gamma - Discount factor parameter.

It has been suggested that if we use γ=0.1 for policy 1, then maybe we will have a good result, but after 50000 cycles of SARSA and Q-learning with the following parameters' ϵ=0.1, γ=0.1, α=1 were obtained, SARSA reached the finish cell 1 times, and Q-learning reached the finish cell 2 times.

2 Task 2

2.1 A

Q-learning.
Q-learning ran 50000 episodes. The Q value matrix was taken, which had the largest mean value among the total_return of a specific and previous episodes (marked in figure 17 with a red dot).

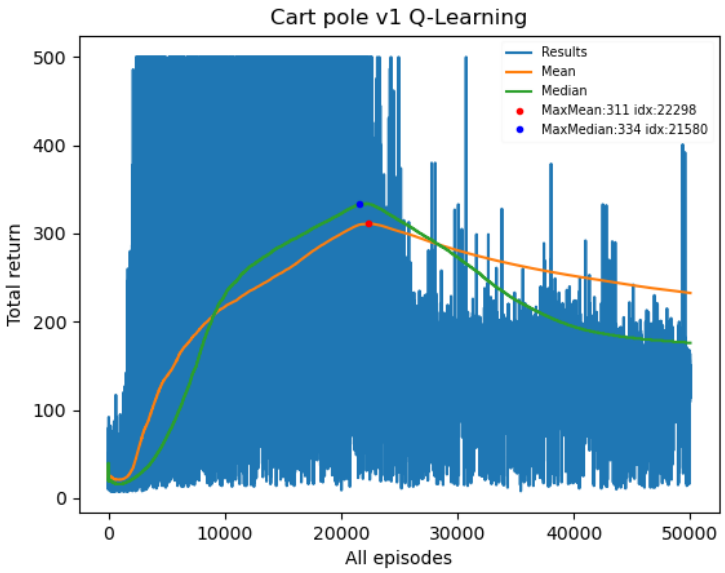

Figure 17: Cart Pole Q-Learning (50000 experiments)

Then this matrix was used for 100 episodes and the following figure 18 was obtained.
Conclusion: more than or equal to 195 in 100 experiments in a row can be obtained using Q-learning. Although it is clear that Q-learning worsens its results after a certain point.

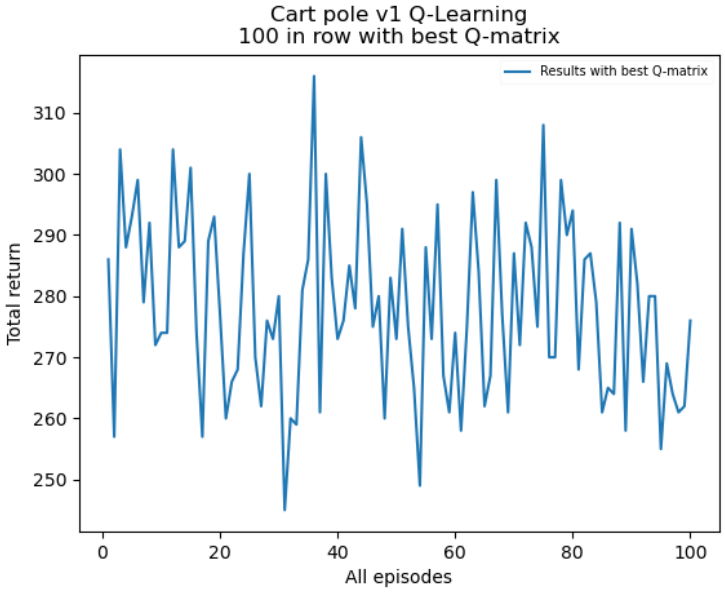

Figure 18: Cart Pole Q-Learning (100 experiments in row)

SARSA.

For SARSA, the V matrix with the largest mean value was also taken and used to obtain 100 episodes. The result can be seen in figures 19 and 20. Conclusion: more than or equal to 195 in 100 experiments in a row can be obtained using SARSA.

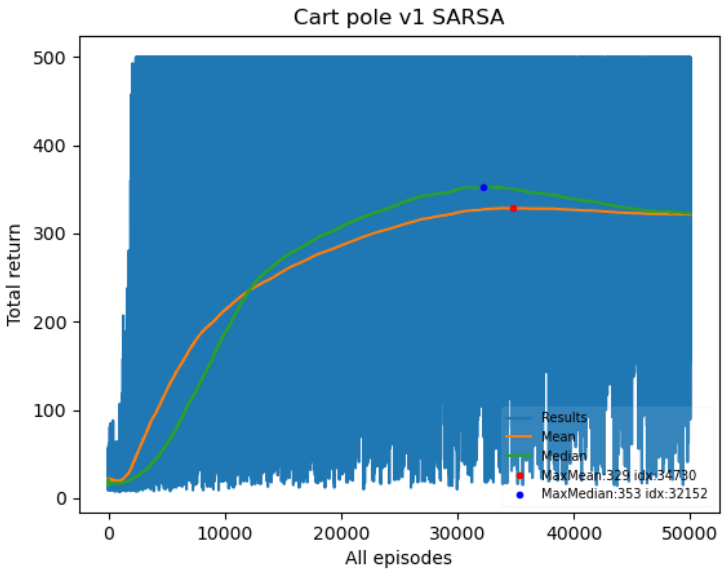

Figure 19: Cart Pole SARSA (50000 experiments)

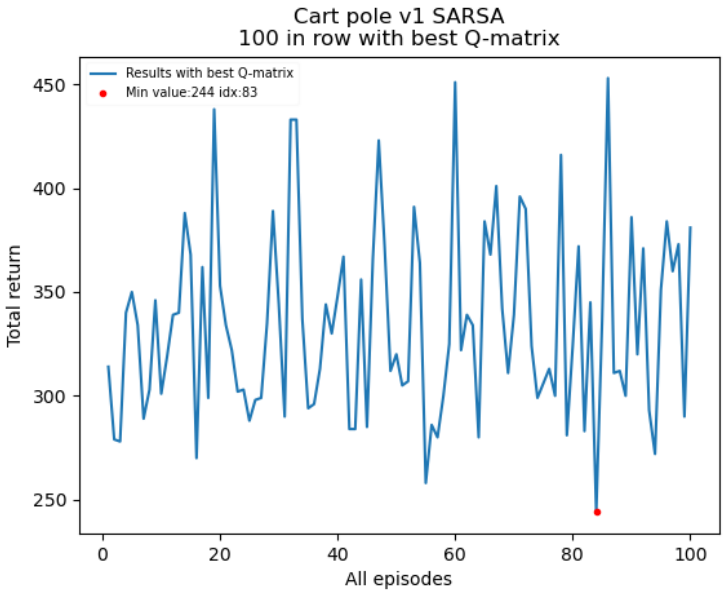

Figure 20: Cart Pole SARSA (100 experiments in row)

General conclusion:
The dispersion of Q-learning is less than SARSA's dispersion based on the figures 18 and 20. Also, Q-learning reaches the peaks of the mean and median values faster than SARSA.
The peaks of these parameters are lower for Q-leaning than for SARSA.
In the Q-learning algorithm, after peaks, these parameters begin to drop rapidly, which is not observed in SARSA.

2.2 B

A fully connected neural network has been created and it can solve the cart pole v1 problem. Several versions of fully connected neural networks have been tested, and the neural network with one hidden layer shows the best quality in terms of speed and returns obtained (figures 21 and 22).

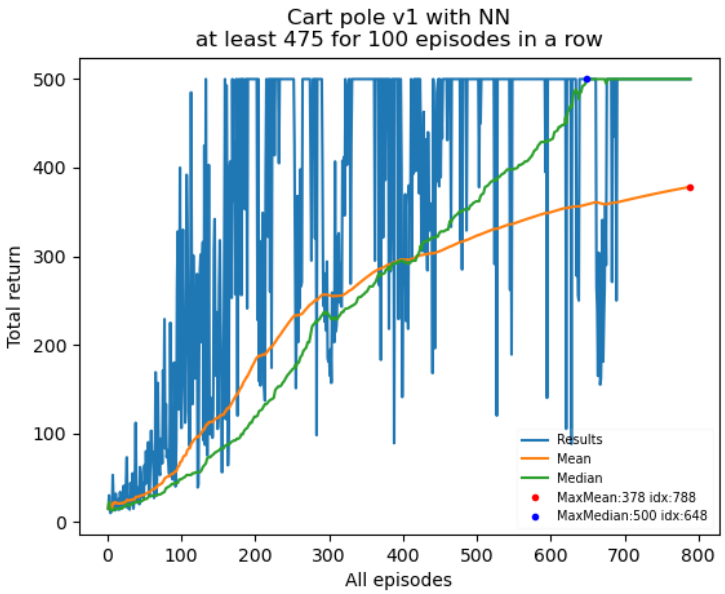

Figure 21: Cart Pole NN (789 experiments)

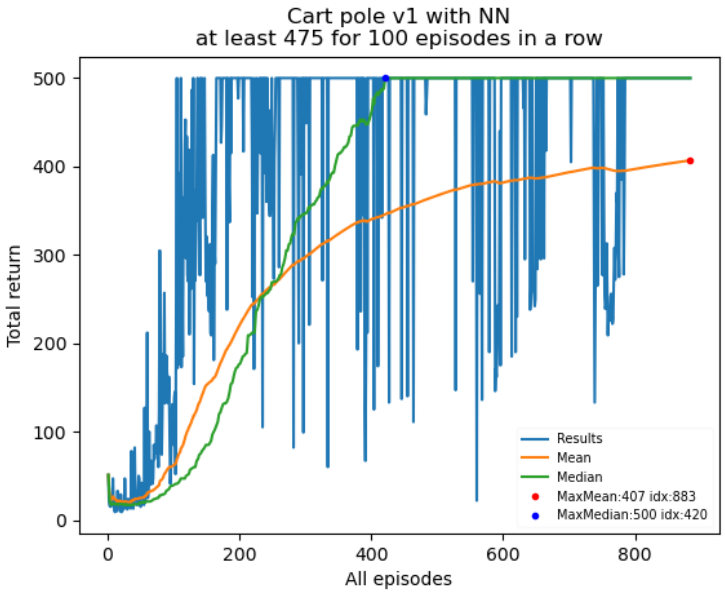

Figure 22: Cart Pole NN (884 experiments)

A neural network with two hidden layers has a large variance of the return parameter and a lower speed (an example of a neural network with two hidden layers in figure 23), so a fully connected neural network with one hidden layer was chosen to work.

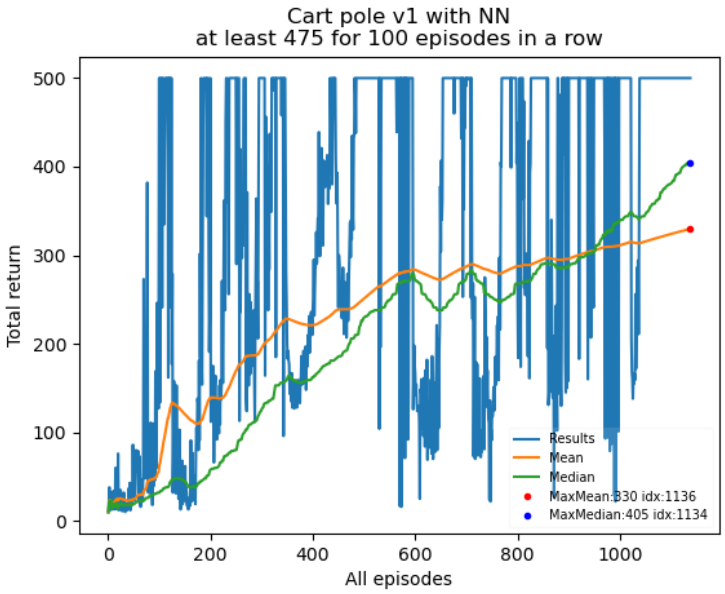

Figure 23: Cart Pole NN with 2 hidden layers (1137 experiments)

2.3 C

Unfortunately, multiple attempts to make this task 2.C were unsuccessful.

The main error appeared when we tried to use the states that have already been used in the neural network in previous experiments with this neural network.

The main error is the following:

Trying to backward through the graph a second time, but the saved intermediate results have already been freed. Specify retain_graph=True when calling .backward() or autograd.grad() the first time.

Many strategies have been used to solve this problem and helped only: retain_graph=True when calling loss.backward().

But the next error occurred:

RuntimeError: one of the variables needed for gradient computation has been modified by an inplace operation: [torch.FloatTensor [100, 2]], which is output 0 of TBackward, is at version 2; expected version 1 instead. Hint: enable anomaly detection to find the operation that failed to compute its gradient, with torch.autograd.set_detect_anomaly(True).

We used to use torch.autograd.set_detect_anomaly(True), but we did not succeed to understand why this error occurs.

In this task, is not yet clear how to write loss function for the values taken from the Experience Replay Buffer, as it differs from the usual loss function

in the task 2B. Cause the reward of random values taken from the Experience Replay Buffer is different.

The code that issues the above errors is posted in the GitLab. This part of code was specially commented.

References

[1] Reinforcement Learning: Markov-Decision Process https://towardsdatascience.com 2019. [Online; accessed 12-January-2022]

[2] Part 1: Key Concepts in RL https://spinningup.openai.com 2018. [Online; accessed 12-January-2022]