Reinforcement Learning Project

Yossi Gavriel (ID. 305498099) Arie Kfir Hayne (ID. 205601172)

Submitted as a final project report for Reinforcement Learning Course, IDC, Aug 2021

1 Introduction

In this work we attempt to implement and use different approaches that we learn at the class to solve two different environments at the open-AI-gym ,PushAndPull-Sokoban-v2 and LunarLander-v2.

Eventually we didn't succeeded to solve any of the environment by our self. We found notebook at the Internet that solve the LunarLander-v2 and we tried to implement the sokoban environment based on this notebook, but had some issues and no time to solved them.

1.1 Related Works

There are some git implementations for thus environments and we used them in order to solve the problems combining the class materials and notebooks.

2 Reinforcement Learning techniques

2.1 MDP (Markov decision processes)

MDP describe a set of problems that satisfy the Markov property - means, the behavior of the process depends on the current state, and is independent of all preceding states that the process has been in. Formally, an MDP is defined as a 5-tuple (S, A, P, R, γ),

1. S is a finite set of states.
2. A is a finite set of actions.
3. R(s, a) is a scalar reward function.
4. P(s, a, s_0) = Prb(s_0 | s, a) is a transition function that gives the probability of reaching state s_0 from s by taking action a.
5. $\gamma \in$ [0, 1] is a discount factor.

By using MDP our goal is to find the policy π(s, a) = Prb(a|s), that when followed, maximizes the expected discounted reward V π (s).

2.2 TD (Temporal Difference)

In thus kind of approaches the environment not have to be episodic, and it is model free same as MDP. We assume that the value of state s is known and we use it. instead of update End-to-End we update the reward from episode to episode.

We want to maximize the expected mean profit, so the input is the state and the output is the expected profit value for each action.

2.3 DQL

We are moving from tables approaches to approximation of q(s,a) using machine learning. In this approach we don't need the values of each (state,action) because the states that we already evaluate will give me good enough approximation for the rest, means it will generalize the unseen states.

We assuming that neural networks can learn any mathematical function as we want. We used thus techniques based on TD. the learning process divided to 3 sections:

1. froward - taken the data and stream it in to the network.
2. loss function - how far are we from the real value?
3. backpropagation - update the weight and run again.

2.4 DDQL - Double DQL

Double Q-Learning with experience replay - two estimators (neural networks) are introduced to separate the concerns of the expected value from the maximization of the action values, and showed that while this method may underestimate expected values, it generally converges to optimal policies more efficiently in comparison to standard Q-learning. After fixed number of episodes we'll copy our first neural network's weights to the second one.

Experience replay in our network updates in order to improve learning updates: Instead of sequentially learning at each time step in each episode with state s. We stored the (s, a, r, s') tuples in a sequence S of length M Before training, we randomly choose experiences of tuples (s, a, r, s'). In this case we use two separate estimators to map functions of max actions and expected rewards.

2.5 Actor-Critic

We learnt that in policy gradient there is high variance, one of the method to reduce it is using "reward to go" but we'll skip it and implement the second method which is reducing the baseline, means subtracting the average rewards from all the rewards.We will use the value function (the expected value of all the rewards) as the baseline.

So we want to combine Policy based and value-based approaches using Actor-Critic.

The "Actor" selects actions and for the action's estimations we'll use the "Critic" which criticizes the actions made by the actor. The learning process is done on-policy, the critic is a state-value function. After the action is selected, the critic evaluates if it's was good/bad action.

If the error is positive means we want to increase that action probability and vise-versa. We don't need to implement exploration/exploitation cause the model already behaving like that – give probability to each action. So we we'll build a trajectories instead of episodes, We want to maximize the probabilities such that will give me a good trajectory. We'll learn the advantage(value) using DQN that cause that method to use the policy based and the values based together. We can see that it's like TD means we don't need to run until the end of the episode cause we have the net to calculate the future expected value.

3 Sokoban

3.1 introduction

Before we started working on the SOKOBAN environment we first have to understand the game. SOKOBAN is Japanese for warehouse keeper and a traditional video game. The game is a transportation puzzle, where the player must push all boxes in the room on the storage locations/ targets. The possibility of making irreversible mistakes makes these puzzles so challenging especially for Reinforcement Learning algorithms, which mostly lack the ability to think ahead.

The repository implements the game Sokoban based on the rules presented DeepMind's paper "Imagination Augmented Agents for Deep Reinforcement Learning". The room generation is random and therefore, will allow to train Deep Neural Networks without overfitting on a set of predefined rooms.

In this work we are trying to solve PushAndPull-Sokoban-v2 means the player can push and pull the boxes. Therefore, no more irreversible moves exist, The conditions for winning is the same.

3.2 environment's features

As we know, for every environment there are features that defining here. sokoban is Grid game. the actions that the player can take is one of the list above : no operation, push up, push down, push left, push right, move up, move down, move left, move right, pull up, pull down, pull left, pull right. total there is 13 actions, divided into 3 sections:

1. Move - simply moves if there is a free field in the direction, which means no blocking box or wall.

2. Push - push tries to move an adjacent box if the next field behind the box is free. This means no chain pushing of boxes is possible. In case there is no box at the adjacent field, the push action is handled the same way as the move action into the same direction.

3. Pull - pull tries to move an adjacent box if the next field behind the box is free. This means no chain pushing of boxes is possible. In case there is no box at the adjacent field, the push action is handled the same way as the move action into the same direction.

Grid-Size	7X7
Pixels	112X112
Boxes	2
actions size	13

Figure 1: environment features

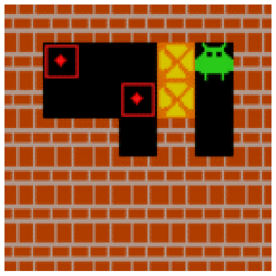


Figure 2: An image of a Random state of the game

3.3 MDP attempt

Sokoban can be characterized as an MDP because it is finite in its state and action space, discrete, and fully observable. Sokoban is also deterministic, which is not necessarily true for all MDPs. basically, Sokoban is an infinite MDP, but we'll stop the episode after 150 steps (if it wasn't finished). therefore we can consider the problem as a finite MDP.

We declared our reward function to this behavior ,Finishing the game by pushing all on the targets gives a reward of 1 in the last step. Also pushing a box on or off a target gives a reward of 0.1 respectively of -0.1. In addition a reward of -0.1 is given for every step, this penalizes solutions with many steps, in addition we add -0.1 for action that doesn't influence the state of the game (no operation, and go into the wall). Finally, we set the discount factor to 0.99.

When the state space is small, it is possible to solve MDPs using tabular methods such as value iteration or policy iteration However, for problems of practical interest, the state space is exponentially large, so it becomes necessary to use methods based on function approximation and reinforcement learning.

3.4 DQN based TD using CNN attempt

For this problem we'll use MSE loss cause we are trying to solve a regression problem which predict expected value for each action. We want the NN to learn the relevant features and classify the actions. We have also tried using CNN that learn direct from the frame pixels and we used the following pre-processing:

1. The input is a raw pixel, and the output is a value function estimating future rewards
2. Change the scale of the grid.
3. We needed to convert each "frame" to 1 dimension since that we don't care the colors to insert the state to our NN and getting the information from the frame.
4. We also cut the image to the relevant part means only the square parts that the agent can move.

The model is a CNN, and the output is at size like the actions space. each variable in the vector will be the probability to take this action in order to maximize the total rewards. We built a CNN with the following hyper parameters:

1. decade epsilon and without exploration rate
2. gamma – [0.99, 0.95, 0.9, 0.8,]
3. learning rate – [0.1, 0.01, 0.001]
4. exploration rate – [0.5, 0.4, 0.3, 0.2, 0.1]
5. exploration min - [0.1 ,0.01]
6. exploration decay – [0.999, 0.995, 0.9, 0.8]

our CNN Layers:

1. 2D Convolution - 64 filter size, activation relu, padding -same
2. 2D Convolution - 32 filter size, activation relu, padding -same
3. 2D Convolution - 16 filter size, activation relu, padding -same
4. Flatten layer - take the image and create 1D vector appending each row of the image
5. Dense - output action size (13), used 2 main activation functions for each attempt, linear and softmax.

We change the reward and the values above in order to figure out if there is any improvements in the result, without succeeded. When we added the reward of moving a box too much ,We think that the machine just ”playing” with the box and doesn’t finish the game. in order to ignore it we added the reward of finish the game to be huge and decrease the rewards of moving the box a bit again it doesn’t work for us. We tried to limit the length of each game such as MDP and we were unable to notice any benefit in the results. We added huge negative reward for actions that don’t affect on the state of the game, such as moving into wall, no operation etc.

3.5 Sokoban results summery

Each attempt we run for 150k episodes During our attempts to solve this environments, we tried applying Actor-Critic architecture based on github project that we found but we don’t succeed to make it run and works, so we don’t represented thus attempts here. Eventually we moved to the LunarLander-v2 environment and tried to use the functionality and the conclusions that we had during our attempts.

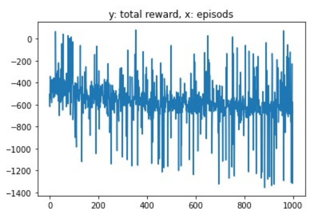
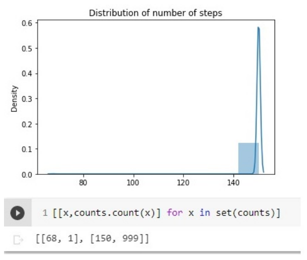


Figure 3: one reward graph of our runs



```
[[x,counts.count(x)] for x in set(counts)]
```

```
[[68, 1], [150, 999]]
```

Figure 4: distribution of steps for one of our runs

It can be seen that the vast majority of games did not converge after 150 steps and out of 1000 games only one finished. we gently decrees the number of steps for each game cause we got enormous running time from 500 to 150, Consult with the other students in the course about this optimal number and we came to the decision that 150 is good enough.

4 LunarLander-v2

4.1 introduction

There is one spaceship. The task is to land the spaceship between the flags softly and fuel-efficiently as possible. The ship has 3 throttles in it. One throttle points downward and the other 2 points in the left and right directions. With the help of these, you must control the Ship. There are 2 different Lunar Lander Environment in OpenAiGym. One has continuous action space and the other discrete action space, we'll solve the continues one.

4.2 environment's features

Landing pad is always at coordinates (0,0). Coordinates are the first two numbers in state vector. Reward for moving from the top of the screen to landing pad and zero speed is about 100-140 points. If lander moves away from landing pad it loses reward back. Episode finishes if the lander crashes or comes to rest, receiving additional -100 or +100 points. Each leg ground contact is +10. Firing main engine is -0.3 points each frame. Solved is 200 points. Landing outside landing pad is possible. Fuel is infinite, so an agent can learn to fly and then land on its first attempt. Action is two real values vector from -1 to +1. First controls main engine, [-1, 0] off, [0, 1] throttle from 50% to 100% power. Engine can't work with less than 50% power. Second value [-1, -0.5] fire left engine, [0.5, 1] fire right engine, [-0.5, 0.5] off.

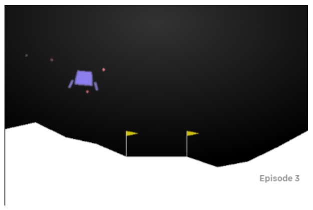


Figure 5: An image of a Random state of the game

For this task we will try to make the action space discreet and solve the problem for discrete action space, in the discrete case there are 4 discrete actions: do nothing, fire left orientation engine, fire main engine, fire right orientation engine.

So we'll model the continuous action space for 10 discrete actions as follows:

1. very left half power - [0.5, -1]
2. very left full power - [1, -1]
3. left full power - [1, -0.75]
4. left half power - [0.5, -0.75]

5. off - [0, 0]
6. full power down - [1, 0]
7. right half power - [0.5, 0.75]
8. right full power - [1, 0.75]
9. very right half power - [0.5, 1]
10. very right full power - [1, 1]

while the first element represents the action space's range of the direction's engine and the second element represents the main engine action space.

Our model will predict one of the 10 discrete actions and the agent will act according the actions above.

We saw that the agent learning that it preferred that the spaceship will not land at all cause it not crashed and then not getting the reward of -100 but we want it to perform less steps and also to land on the pad so we increase that situation penalty to -100/-500.

4.3 DQL - attempt

We implemented a DQN with replay buffer

Network architectures:

1. Hidden layers: combination of 24, 64, 128, 256, 512
2. Hidden_1 – fully connected, activation: relu
3. Hidden_2 – fully connected, activation: relu
4. Output layer – 9 - linear activation of size 9
5. Loss – linear

4.4 DDQL - attempt

Double Deep Q-Learning with experience replay.

The network architectures is same for both neural networks:

1. Hidden layers: combination of 24, 64, 128, 256, 512
2. Hidden_1 – fully connected, activation: Relu
3. Hidden_2 – fully connected, activation: Relu
4. output layer – 9 - linear activation of size 9
5. Loss – MSE
6. Optimizer – Adam

We have also tried to give higher probabilities for some actions when performing exploration step, i.e we gave 0.35 probability for the action "full power down" in order to avoid states where the spaceship turn to the sides.

4.5 Actor-Critic - attempt

We'll use discount factor of 0.99/0.9

Actor architecture-

1. Hidden_1 – fully connected, combination of [400, 256], activation: relu
2. Output layer – 9 - linear activation of size 9, activation: softmax
3. Loss – categorical-cross-entropy
4. Optimizer – Adam
5. Learning rate – 0.005/0.001

We are using here categorical-cross-entropy loss function since we want it to output the distribution for each action. The hidden layers also contains batch normalization.

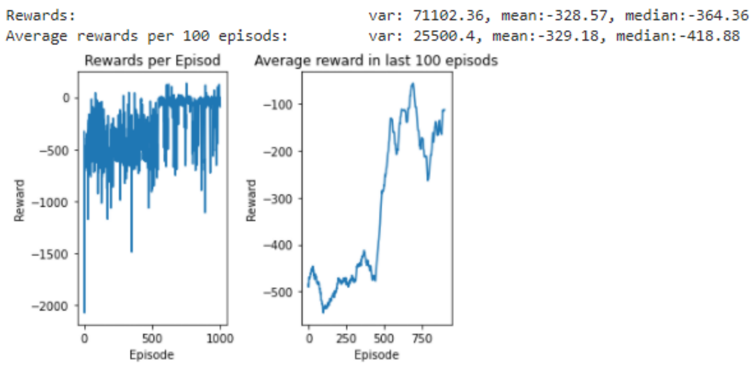
Critic architecture–

1. Hidden_1 (for state) – fully connected, combination of [400, 200], activation: relu
2. Hidden_2 (for state) – fully connected, combination of [400, 200], activation: relu
3. Hidden_3 (for action) – fully connected, combination of [400, 200], activation: relu
4. Loss – MSE
5. Optimizer – Adam
6. Learning rate – 0.005/0.001

Hidden layers of action and state are added together with RELU activation on it. The loss function will be MSE because we want to learn the value.

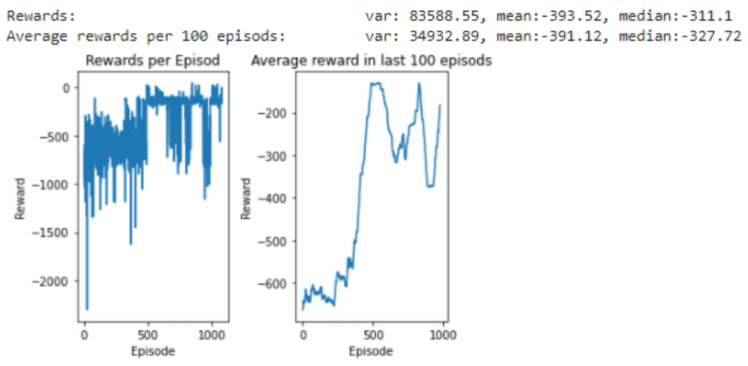
4.6 Lunar results summery

The results of our attempts are presented in the next four figures.



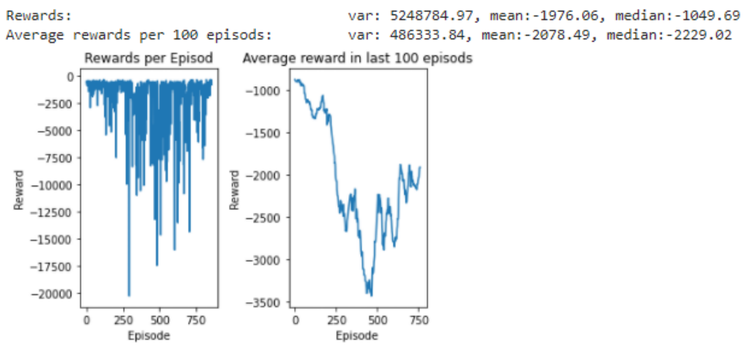
```
Rewards:                                  var: 71102.36, mean:-328.57, median:-364.36
Average rewards per 100 episodes:         var: 25500.4, mean:-329.18, median:-418.88
```

Figure 6: Lunar with Actor-Critic



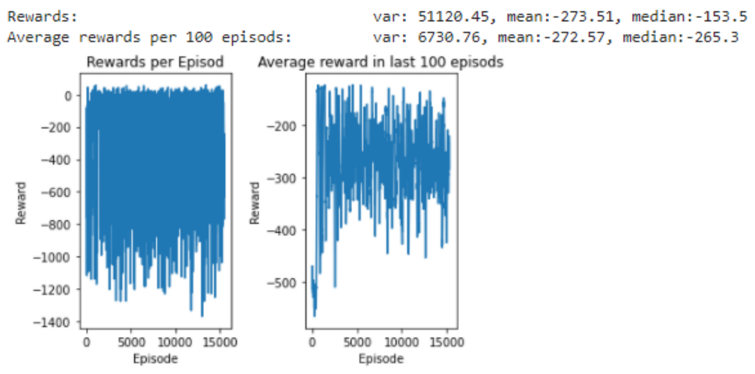
```
Rewards:                                  var: 83588.55, mean:-393.52, median:-311.1
Average rewards per 100 episodes:         var: 34932.89, mean:-391.12, median:-327.72
```

Figure 7: Lunar with DQN



```
Rewards:                                  var: 5248784.97, mean:-1976.06, median:-1049.69
Average rewards per 100 episodes:         var: 486333.84, mean:-2078.49, median:-2229.02
```

Figure 8: Lunar with Double-DQN



```
Rewards:                                var: 51120.45, mean:-273.51, median:-153.5
Average rewards per 100 episods:        var: 6730.76, mean:-272.57, median:-265.3
```

Figure 9: Lunar with Actor-Critic 15k episodes

5 Conclusion

We have tried various reinforcement learning approaches such as DQN, Double-DQN, Actor-Critic, with and without experience replay and epsilon decay. In addition we have tried to change the hyper parameters such as the exploration rate, learning rates, optimizers, losses, and the structure of the layers in the NN . Also we have tried to change the rewards according the problem and the result that we achieve from our previous approaches but none of this approaches solved the environment.

In this project we have tried to implement all what we learn in the class, we briefly moved over thus techniques at chapter 2 and dive into the problem that we have tried to solved using them. Hope that you enjoy to read our work, unfortunately we didn't succeeds to solve any of the environment by our self.

We can see that our attempt with actor critic solving Lunar environment indeed reduced the variance but unfortunately, didn't managed to solve the game.

Note: We worked with the computing power at our disposal, which is our personal computers and COLAB. Every run on our PCs took days until it finished, at COLAB we mostly checked the feasibility of things because the environment there is unstable.

6 Code & References

- LunarLander-v2 open-AI -
  `https://gym.openai.com/envs/LunarLander-v2/`
- LunarLander-v2 found solution -
  `https://colab.research.google.com/drive/1NdKmC7fR-7JmSju7hQRcepJmyBirok0C?usp=sharing`

- Sokoban v1 Colab notebooks -
    1. https://colab.research.google.com/drive/1fsb5N7mFV73Rdar2VHkNfoDjJXMp3G_h?usp=sharing
- Sokoban Push and pull Colab notebooks -
    1. https://colab.research.google.com/drive/1pAFvhHnqR1MsOiQb96T7iF17AnM7mHcr?usp=sharing
    2. https://colab.research.google.com/drive/1qTK833S9ywKKz3B-5lcfhgjpQ2ppzAwI?usp=sharing
- Lunar Colab notebooks -
    1. Double-DQN: https://colab.research.google.com/drive/1iGrWTJDHJo-FD6UIZqaM_vX7GIZ3JSJM?usp=sharing
    2. DQN: https://colab.research.google.com/drive/13k_7-JnfjBT_lKlcBP1Zh3oHEJwxyOht?usp=sharing
    3. Actor-Critic: https://colab.research.google.com/drive/1KMVrfC22SdK-9-XYTtYqm9jOXSKNhv
       usp=sharing
- used Github Repository that we used -

  https://github.com/mpSchrader/gym-sokoban