Reinforcement Learning – Final Project

Sahar Millis 300420379
Tomer Segall 301833836

1. Introduction

In this project we have tried to use various recommendation system algorithms and solve Lunar Lander game in its continuous action space version and the Sokoban environment (PushAndPull-sokoban-v0). We have tried to use three different algorithms (two for Lunar Lander and one for Sokoban) to solve these environments:

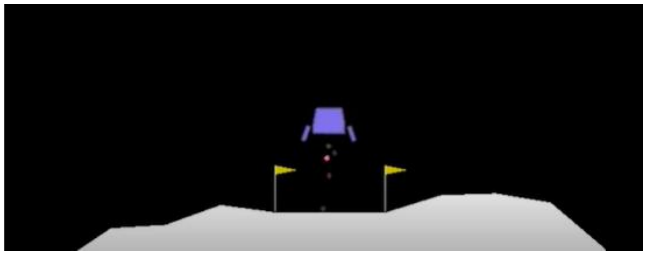
1. Deep q network (DQN) – used for Lunar Lander
2. Double deep Q network (DDQN) - used for Lunar Lander
3. Actor critic – used for Sokoban

We haven't succeeded to solve the Sokoban environment and we felt that we need much more computational power to solve it. We have seen in other papers that solve the environment that the agent needs to run thousands of episodes. We will show our approach for trying to solve the Sokoban, but our focus will be more on the Lunar Lander environment, which we succeeded to solve. We will also show a convergence comparison between different hyper parameters configurations in each one of the algorithms.

2. Background

Lunar Lander

The goal in Lunar Lander is to land a spaceship between two different flag poles:



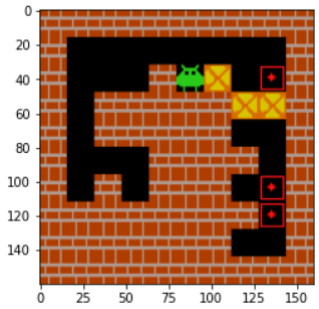
The game has two versions – discreate and continuous action space. We first solved the discrete version, which is easier since there are only 4 possible actions. Then we solved the continuous action space version. In the continuous version the action consists of two continuous values:

1. Value 1 – range of [-1,1] of the main engine where [-1,0] is off and (0,+1) is on.
2. Value 2 – range of [-1,1] of the side engine where the left engine works at [-1,-0.5], the right engine works at [0.5,1] and in the range (-0.5, 0.5) the side engine is off.

We discretized the continuous action space to a discrete one with 9 different possible actions. We have figured that the more actions the agent will have to choose from, the more episodes the agent will need to solve the environment. This is the reason we have decided to discretize the environment to 9 different possible actions, which is more than two times the Lunar Lander's discrete version.

Sokoban

The goal here is to solve a puzzle where the player needs to push the different boxes in the room to the target locations:



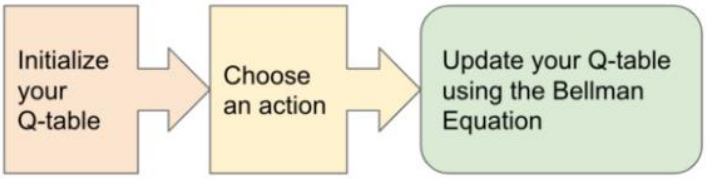
3. Methods

Lunar Lander

we used DQN and DDQN to solve the Lunar Lander environments. We will give a short explanation starting from q learning through the improvements of DQN and finally DDQN.

Q- Learning

The algorithm Q-learning consists of 3 main steps:



The Q-table is a simple table that maps between a pair of a state and an action to their expected reward, and more specifically it maps the state-action pair to the estimated optimal future value (Q value). The agent's goal is to learn this Q-table. The agent performs one of two operations:

a. **Exploration –** The agent tries different actions at different states and updates the Q-table according to the actual reward received in the different state-action pairs.
b. **Exploitation –** The Q value is the maximum expected reward an agent can reach performing action A in a state S. After learning the Q-table, the agent can choose the action A that will give the best expected reward. Choosing this best action is called exploitation.

There is a tradeoff between exploration and exploitation. At the beginning, the agent doesn't know which action is best for all given states, meaning it must explore the environment. After the agent explores the environment and learns the expected reward for some state-action pairs, it wouldn't be optimal for the agent to always choose the best action since it won't learn new possible actions that might be better than the action it chose. The **epsilon greedy exploration strategy** helps to address this issue. At every step where the agent needs to choose an action, the agent chooses a random action at

probability ϵ. At probability $1-\epsilon$ the agent chooses the action that optimizes the future reward according to the Q-table.
The final step of the Q-learning algorithm is to update the Q-table using the Bellman equation:

$$Q(S_t, A_t) = (1-\alpha)\, Q(S_t, A_t) + \alpha * (R_t + \gamma * \max_{a} Q(S_{t+1}, a))$$

Where:
$S - the\ state$
$A - the\ action$
$R - the\ reward\ given\ by\ taking\ the\ action$
$t - current\ time\ step$
$\alpha - the\ learning\ rate$
$\gamma - the\ discount\ factor. It\ causes\ the\ rewards\ to\ lose\ thier\ value\ over\ time$

DQN

The only difference in DQN compared to Q-learning is the agent's brain – in Q-learning the agent's brain is the Q table, whereas in DQN the agent's brain is a deep neural network. The input to the network is a state. The number of neurons at the output is the number of possible actions, where the value of each neuron represents the Q value. The deep neural network in DQN maps between a state to an action-q value pair. The target ("real values") of the network is estimated as:

$$Q'(S, A) = R(S, A) + \gamma * \max_{a} Q(S', A)$$

Where $Q(S', A)$ is the Q value of the next state given by the action taken in the current state. The DQN loss is therefore:

$$Q(S, A) = Q(S, A) + \alpha(Q'(S, A) - Q(S, A))$$

A concept that helps the deep neural network to converge is called **experience replay.** This is the act of storing game states – state, action, reward and next state that the agent has seen. DQN is using experience replay to store seen game states, sample from it randomly when training to help and speed up the network's convergence.

DDQN

The improvement of DDQN over DQN is that the target value is calculated based on a secondary network called the **target network**. Instead of calculating the targeted Q-value with the reward added to the next state maximum Q-value, the target network calculates the targeted Q-value. The weights of this network is copied from the DQN network's weight every X steps. The usage of a secondary network leads to a more stabilized learning process.

<u>Sokoban</u>

In the Sokoban we have tried to solve the environment using the **actor critic algorithm**.

In this algorithm there are also two networks – the actor and the critic. The actor network acts according to some policy – it takes in states and produces actions. The critic network learns how well the current policy (the actor) is performing and uses this information to improve the policy. Unlike in DQN and DDQN, in this algorithm the agent learns the policy directly. This is referred as **policy gradient**.
In our first trials of solving Sokoban, we have seen that the agent gets stuck learning after a few episodes since it picked the same action over and over. We decided to change the rewards in a way that will make the agent avoid getting stuck in one action:

1. Penalize more if the action taken didn't change the image of the new state compared to the previous state (-0.3 instead of -0.1)
2. Penalize even more if a certain action is taken consecutively more than 5 times in a row (-1.0 instead of -0.1).

This approach prevented the agent from getting stuck and choosing a single action.

3.4 Network Architecture

We coded our deep neural networks using the keras framework.
Lunar Lander
The architecture of DQN and DDQN (both networks are the same) is:

1. Input layer of the state size (8 neurons)
2. Fully connected layer (256 neurons) with a relu activation function
3. Fully connected layer (256 neurons) with a relu activation function
4. Output layer of the action size (9 neurons)
5. Adam is the optimizer chosen of the neural network
6. MSE is the loss function of the neural network

Sokoban
The architecture of the actor and the critic networks is the same except for the output layer:

1. Input layer of the image RGB (160,160,3)
2. Convolutional layer of 64 neurons with a kernel size of 3X3 with max pooling and batch normalization
3. Convolutional layer of 128 neurons with a kernel size of 3X3 with max pooling and batch normalization
4. Fully connected layer (24 neurons) with a relu activation function

The actor network's output layer consists of 13 neurons (the number of possible actions) with a categorical cross-entropy loss and an Adam optimizer. The critic network's output layer consists of a single neuron (represents the value) with MSE loss and an Adam optimizer.

3.5 Hyper Parameters Tuning

Lunar Lander

We did hyper parameter tuning on three parameters, running 100 episodes for each permutation:

1. $\gamma - discount\ factor$
2. $\alpha - learning\ rate$
3. $\epsilon - exploration\ rate$

Here is the table of the different hyper parameter combinations:

Discount factor	Learning rate	Exploration rate
0.9	0.0005	0
0.9	0.0005	0.1
0.9	0.001	0
0.9	0.001	0.1
0.99	0.0005	0
0.99	0.0005	0.1
0.99	0.001	0
0.99	0.001	0.1

Sokoban

Here we performed the hyper parameter tuning on two parameters, running 100 episodes for each permutation:

Discount factor	Actor network learning rate
0.9	0.001
0.9	0.005
0.99	0.001
0.99	0.005

3.6 Run with best hyper parameter configuration

We chose the best hyper parameter configuration that converges the fastest in the Lunar Lander environment and ran 300 episodes to check that the environment is solved and that the learning curve is stable.

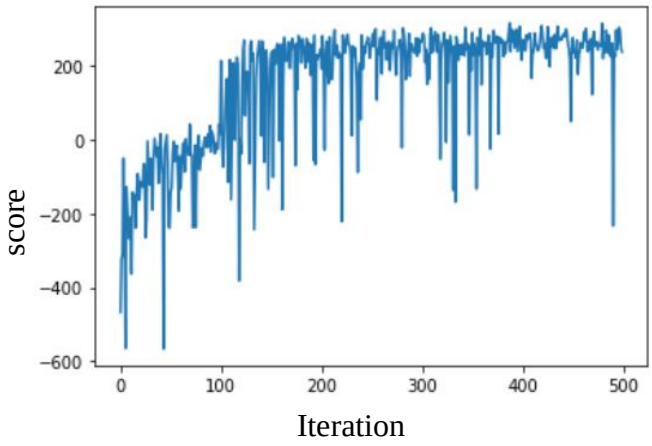
5. Results

Lunar Lander

In this section we will show the different results we have got for the different algorithms.

5.1 Lunar Lander – Discrete version

We solved this environment using DQN. Here is the learning convergence:



Here is a link to the relevant notebook

5.2 Lunar Lander – Continuous version DQN

The following are the results of the hyper parameter tuning of DQN on Lunar Lander's continuous version:

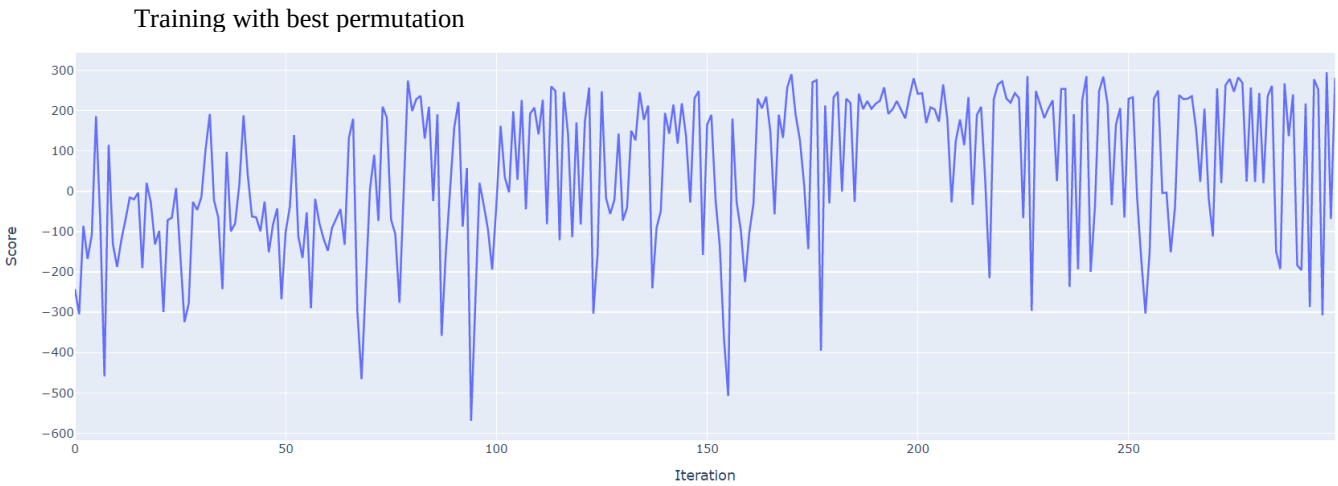
Here is a link to the relevant notebook

5.3 Lunar Lander – Continuous version DDQN

The following are the results of the hyper parameter tuning of DDQN on Lunar Lander's continuous version:

Here is a link to the relevant notebook

We chose the permutation of gamma= 0.99, learning rate = 0.001 and epsilon = 0.1 as the best permutation and ran 300 episodes:



Here is a link to the relevant notebook

Sokoban

5.4 Sokoban (PushAndPull-sokoban-v0) – Actor Critic

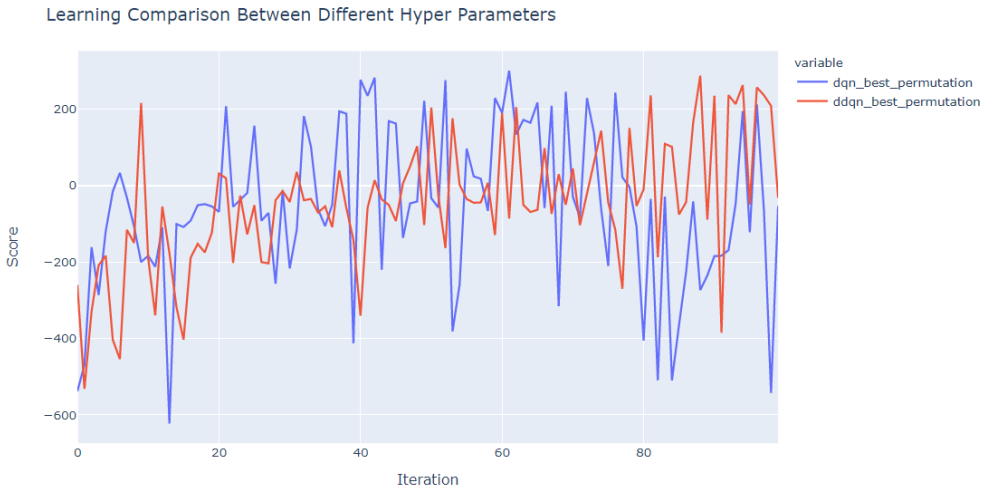
The following are the results of the hyper parameter tuning of the actor critic on Sokoban:

Here is a link to the relevant notebook

6. Discussion

Lunar Lander

We have managed to solve the lunar lander in both of its version – discrete and continuous. We first solved the discrete version, as it is considered easier than the continuous one, with DQN to have confident that our algorithm works properly. Then we have moved to solve the continuous version, where we compared two different algorithms (DQN and DDQN) and for each one of them compared 8 different hyper parameter permutations of the parameters discount factor (γ), learning rate and exploration rate (ϵ). A first interesting observation is that the DDQN after 80 iterations has succeeded to solve the environment with several different permutations, while the DQN convergence curve is not as stable after 80 iterations in all the different permutations. The following is a comparison between the best permutation of each algorithm:



As mentioned above, DDQN has a more stabilized learning curve, which we could also see in our results. In DQN, the target is updated by:

$$Q'(S,A) = R(S,A) + \gamma * \max_{a} Q(S',A)$$

In case the DQN network slightly overestimates the target value, then this error gets compounded in the target values which can obviously affect the stability of the learning curve – This phenomenon is known as **maximization bias**. We can see that after 50 iterations the DQN algorithm is on the way of solving the environment but then at around 80 iterations we can see the drop in the score confirming the instability of the learning curve. In DDQN, on the other hand, we have a second network (called target network) that calculates the target value. Its parameters are updated every X steps to the weights of the DQN's network. The idea is to control the over estimation of the target value using a second network to help and stabilize the learning curve. As we can see, the DDQN's learning curve is indeed more stable.

Another interesting observation that we can see in both algorithms is that the discount factor has the most influence on the learning curves – all permutations in both algorithms

(DQN and DDQN) with $\gamma = 0.9$ has not succeeded to solve the environment in 100 episodes, while the permutations with $\gamma = 0.99$ certainly have better convergence and some have even solved the environment. A lower discount factor means that the agent gives more weight to immediate rewards rather than to future rewards. In the Lunar Lander game, the high rewards are given in the future: +100 when the ship comes to rest, between +100 to +140 if the ship lands on the landing pad or -100 if the ship crashes. Therefore, it makes sense to give high weights for future rewards and this is the reason that higher discount factors produce better results.

Sokoban

We can also see that in the Sokoban, the higher discount factor produced better learning curves probably due to the same reason we mentioned in the Lunar Lander case – the high positive reward is given in case the player succeeds to solve the puzzle and therefore future rewards should have high weights. We can also observe that after 80 episodes the different agents got stuck in learning. We think that this happens since the agent barely obtains positive rewards. We have a few ideas for future work:

1. Preprocess the image – convert it to grey scale, reduce the number of pixels
2. Give the agent additional positive rewards – for example, in case the average distance between the boxes and the target locations is smaller in the new state compared to the previous state then the agent should receive a positive reward.
3. Try different network architecture – deeper network with more parameters.
4. Run the algorithm on a more scalable machine than our personal computers for thousands of episodes.

7. Code

Here are the links to the different colab notebooks:

1. Lunar Lander Discrete Version Using DQN
2. Lunar Lander Continuous Version Using DQN
3. Lunar Lander Continuous Version Using DDQN
4. Lunar Lander Using DDQN With Best Hyper Parameter Permutation
5. Sokoban Using Actor Critic

8. References (links)

1. Solving the lunar lander problem under uncertainty using reinforcement learning
2. Supervised Actor-Critic Reinforcement Learning
3. Deep Reinforcement Learning: Pong from pixels