

# Reinforcement Learning - Final Course Project

Ron Azuelos, Hadar Pur

Submitted as final project report for the RL course, IDC, 2023

## 1 Introduction

The goal of the final project in Reinforcement Learning (RL) course is to summarize the second part of the semester, in which we focused on Deep Reinforcement Learning (DRL).

We were requested to provide a DRL-based solution that attempts to solve the game known as “Sokoban” – a Japanese puzzle video game where the player’s goal is to bring the boxes to the targets, by pushing and pulling them. The game’s board consists of a grid of tiles, where each tile serves a different purpose and can be one of the following: wall, floor, box, target, and player. The tiles variations are demonstrated in Figure 1. A solvable game board has a player, an equal amount of boxes and targets, and a combination of floors and walls that allows the player to legally move the boxes to the targets. The green creature (i.e. the player) can perform either horizontal or vertical movements, which can be combined with pulling or pushing a box to the same direction. All 13 possible actions and their mapping are shown in Figure 2. Of course, all actions are limited by the walls, which means that the player cannot walk, push or pull through them. The terminal state is defined as the state where all the boxes are “on targets”. Reaching this state grants the player a reward of 10. Placing a box on a target rewards the player with 1, and removing it from one rewards it with -1. Any other action incurs a reward of -0.1. The final reward is the sum of all the rewards that the player has earned during the game, which of course means that the main goal is to “win” using a minimal number of steps.














Type	State	Graphic	TinyWorld
Wall	Static		
Floor	Empty		
Box Target	Empty		
Box	Off Target		
Box	On Target		
Player	Off Target		
Player	On Target		

Figure 1: Actions

Action	ID
No Operation	0
Push Up	1
Push Down	2
Push Left	3
Push Right	4
Move Up	5
Move Down	6
Move Left	7
Move Right	8
Pull Up	9
Pull Down	10
Pull Left	11
Pull Right	12

Figure 2: Elements

Both exercises of the project includes a grid of 112x112 pixels, that constructs a board of 7x7 tiles, whose borders contain walls, and each pixel is represented by 3 values: red, green and blue (RGB).

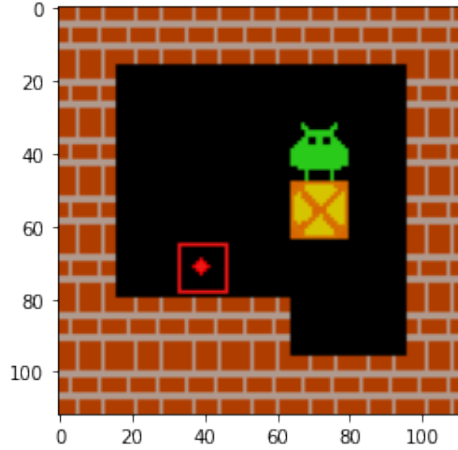


Figure 3: The fixed environment for exercise 1.

In the first exercise, we were asked to solve environments with a single box and target, where the training process was done for the same generated board for each episode. The “fixed” game board that was generated is displayed in Figure 3. In the second exercise, we were asked to solve environments with a single box and target as well, but during the training process the agent tries to solve different game boards in each episode.

## 2 Solution

### 2.1 General approach

As we were requested to solve the above mentioned exercises using DRL tools, we chose to focus on two DRL agents: Deep Q-Network (DQN), and Advantage Actor Critic (A2C). We implemented both agents using the same parameters, including a nearly-identical neural network structure, experience replay buffer (See Section 2.1.1), and with an optional optimization of using reward shaping (See Section 2.1.2). We were requested to experiment and compare different approaches to solve the exercises, so we decided to examine both agents for each exercise and compared their performances. In exercise 1, we trained the DQN and A2C agents with experience replay and without using reward shaping. In exercise 2, we tried to train both agents with the same configurations as in exercise 1. However, as the agents did not provide us with satisfying results, we trained them with reward shaping optimization technique, that was introduced in a study provided to us as a reference<sup>1</sup>. In all cases, we added a preprocessing stage for each state, that included cropping the edges and converting the RGB image into gray-scale. We ended up with four different models, two for each exercise.

#### 2.1.1 Experience Replay

Experience Replay is a technique that was presented to us during the course. Its main idea is to record the last X actions that the agent performed and to store them in a buffer, along with the state before the action was performed, the new state, and the reward that the action incurred. It allows the agent to randomly sample actions from the buffer throughout the training process, and train the agent’s network based on its entire experience, instead of relying on the last action only. The main benefit of this approach is that it enables the agent to learn from actions that does not occur frequently.

#### 2.1.2 Reward Shaping

To optimize the models so that they will be able to solve the second exercise, we tried using reward shaping technique, presented in the paper by Yang et al.<sup>1</sup> The motive is to change the reward that the agent receives

<sup>1</sup><https://arxiv.org/pdf/2109.05022v1.pdf>

for certain actions to encourage it to learn a specific behaviour. In our case, we wanted the reward to be in accordance to the distance of the box from the target, so it will encourage the agent to move the box to the target, and will “punish” it for longer distances. Similarly to the paper’s authors, we used the Manhattan distance metric to determine the distance between the box and the target. We define the potential of a state as the negative distance between the box and the target, divided by 10. If the player moved the box, we added the potential of the new state  $s'$  to the original reward  $R$ , and subtracted the potential of the previous state  $s$  from it. The reward shaping is formulated in Equation 1. This way, it is guaranteed that if the player did not move the box, the potential of the new and previous states will be equal, and therefore their subtraction will be zero and the reward will not be shaped. Furthermore, the agent will receive a “positive” reward if the box is moved closer to the target, and a “negative” reward if moved away from it.

$$R'(s, a, s') = R(s, a, s') + \phi(s') - \phi(s) \quad (1)$$

Also, we added another “punishment” that the article did not suggest, which is penalizing the agent for performing an action that keeps it at the same location. The purpose of this punishment was to encourage the agent to avoid from walking into walls.

### 2.1.3 DQN

DQN is a model-free and value-based RL algorithm, whose purpose is to estimate the Q-value of a state-action pair, using a deep neural network. The network update process is done according to the Bellman equation, the same as it is done in Q-Learning algorithm. The algorithm uses two identically initialized networks: the Q-network and the target network. The learning process is done with the Q-network, where the network is being update after a batch of actions. The target network is being updated using the weights of the Q-network, which can be done using a “copy” operation of the weights every preconfigured number of time steps, or using a “soft” update where the weights are updated after every time step, but with smaller steps. The use of the target network allows more stable training process, and provides faster and more efficient learning. At the end of the learning process, we own a model that can, hopefully, determine the best action to take based on a given state. Since we are using  $\varepsilon$ -greedy method to ensure exploration, the probability to perform the chosen action is  $1 - \varepsilon$ , and that probability decays (i.e. the  $\varepsilon$  value is decreased) during the learning process as the agent becomes more certain of its actions.

### 2.1.4 A2C

Actor-critic is a type of RL algorithm that combines elements of both policy-based and value-based methods, using two main components: the actor (the policy network) and the critic (the value network). The actor, on one hand, is responsible for selecting actions based on the current state of the environment. It learns a policy function that maps states to actions, and it tries to maximize the expected reward by choosing actions that lead to the highest expected return. The critic, on the other hand, learns a value function that estimates the expected return from a given state. It evaluates the actions chosen by the actor, and provides feedback in the form of a value signal that indicates how well was the chosen action. Actor-critic utilizes the two networks in a way which the actor uses the critic’s feedback to update its policy (using policy gradient method), and the critic uses the actor’s actions to update its value estimates (using the temporal difference error). As in any other algorithm, this process repeats until the policy and value estimates converge to optimal solutions.

Advantage Actor-Critic (A2C) is an extension to the Actor-Critic algorithm, that also utilizes advantage function to improve the learning process. In each time step, the agent selects an action based on the policy, receives a reward from the environment, and updates the policy and value function using the advantage function. A2C is computationally efficient, can learn from continuous action spaces and high-dimensional state spaces, and can achieve state-of-the-art results on a wide range of environments.

## 2.2 Design

We implemented our solution using PyTorch library for deep learning in Python programming language. We executed our experiments on a Google Colab notebook, with their free-tier GPU. Both algorithms were

implemented as fully-configurable classes, which allowed us to easily tune the various parameters of each algorithm, including whether to perform reward shaping or not. Therefore, we were able to perform two different training processes for the two exercise, using the same implementation.

### 2.2.1 Preprocessing

To decrease the amount of data that the network needs to learn from, we decided to avoid using the original colors of the images and to convert it to gray-scale. This means, we are transforming our input images from three channels of 112x112 pixels, corresponding to red (R), green (G) and blue (B), to a single channel, corresponding to different shades of black. We did so using the Python’s package for image processing named *openCV*, which defines a formula (*CV\_RGB2GRAY*) for translating BGR (or RGB) image into a gray-scale image, presented in Equation 2.

$$Grayscale(R, G, B) = 0.299 \cdot R + 0.587 \cdot G + 0.144 \cdot B \quad (2)$$

In addition, we cropped the edges of each state, as it did not hold any useful information which made them irrelevant to the learning process. To summary, we converted each state tensor from a RGB image of shape 112x112x3 to a cropped gray-scale image of shape 80x80x1, which was used to train our model.

### 2.2.2 Experimental Settings

For both algorithms, we implemented a convolutional neural network (CNN) as the model. The input layer expects a tensor input of size 80x80, representing state’s pixels, after the preprocessing stage. The input layer is followed by three convolutional layers the outputs 32, 64 and 64 channels respectively, with kernel sizes of 8, 4, and 3, and strides of 4, 2 and 1. Each layer is followed by a ReLU activation. After the last convolutional layer and its activation, the input is flattened and then send to the output layer. In the case of the DQN and the actor of the A2C, the output layer is a fully connected layer that accepts an input of size 2304 and outputs 13 values, one for each possible action. In the case of the critic of the A2C, the output layer is a fully connected layer that accepts an input of the same size, but outputs a single value, corresponding to the state-value. Also, the actor’s output is applied with softmax activation, to ensure a normalized distribution of the action probabilities. We used *AdamW* optimizer, with a learning rate of 0.0001, to ensure a more stable and safe optimization process of the networks. We used smooth L1 loss, which is closely related to Huber loss, as the loss function of the DQN update process, and the calculation of the critic loss in the case of A2C. Both of them used a discount factor equal to 0.99. The actor loss is calculated using the advantages, and the combined loss of the A2C is calculated as seen in Equation 3, where *entropy* is the mean entropy of the action distribution of the batch.

$$Loss(A2C) = Loss(actor) + 0.5 * Loss(critic) - 0.1 * entropy \quad (3)$$

The parameters we used for both algorithms were identical, and were chosen after experimenting with a large amount of models with different configurations that converged too slow, or even did not converge. We used an experience replay buffer of 10,000 records, and sampled a batch of 128 records from it at each training step. The action selection policy that we used was  $\varepsilon$ -greedy, with decaying  $\varepsilon$  value, starting from 0.9 and decaying up to 0.05 after every step, using the formula presented in Equation 4 (steps is the total amount of steps completed). Also, in the case of DQN, we chose a soft-update approach for the target model, and updated its weights after each step, using  $\tau = 0.005$ .

$$\varepsilon = end_{\varepsilon} + (start_{\varepsilon} - end_{\varepsilon}) \cdot e^{-\frac{steps}{100000}} \quad (4)$$

The models were trained for a total of 200 and 500 episodes, in exercise 1 and 2 respectively, this due to their performance and ability to converge. The agent was allowed to perform a maximum of 500 steps per episode.

### 3 Experimental results

#### 3.1 Exercise 1: Fixed board configuration

As mentioned before, in the first exercise we were requested to train a model that attempts to solve a game with a fixed board setup. This means that the entire training process is done for the same specific known board. To solve this exercise, we used the two algorithms we implemented: DQN and A2C. All the detailed results are reproducible.

##### 3.1.1 DQN

As requested, we provided two videos that shows the solution that the agent was able to offer, one right at the half of the training process, and the second one after it is completed. The videos show that the agent provides the exact same solution in the two cases, with both reaching an optimal solution with a minimal number of steps. Also, we were requested to supply graphs describing the convergence of the model, including the number of steps performed by the agent in each episode shown in Figure 4, and the cumulative reward for each episode shown in Figure 5. Both graphs indicate that the agent was managed to solve the game even before it completed 100 episodes of training. To better understand the trends, and to emphasize the convergence of the models, we supplied two additional graphs showing the mean number of steps and mean cumulative reward over the last 100, or less, episodes (Figures 6 and 7 respectively). It is noticeable that the total number of steps decreases over time, and the total reward increases. We can see that after about 120 episodes, the mean steps and mean rewards stabilized around their optimal values, which indicates that our model converged.

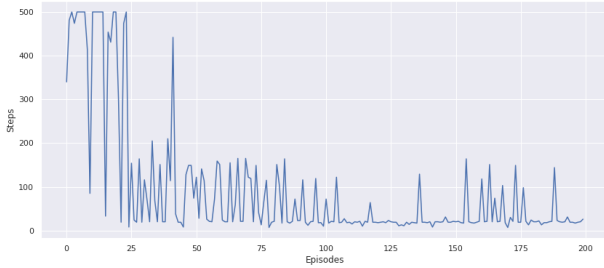


Figure 4: Steps vs. Episodes

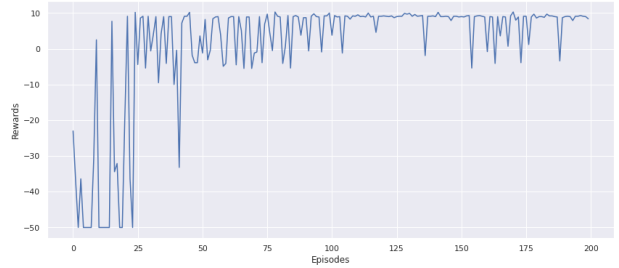


Figure 5: Rewards vs. Episodes

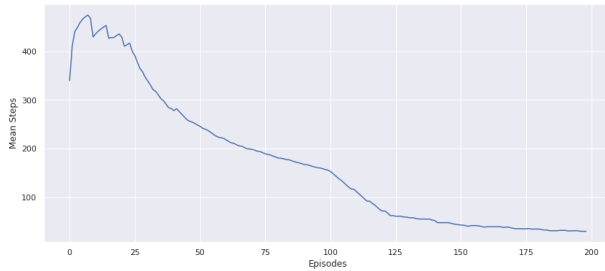


Figure 6: Mean Steps vs. Episodes

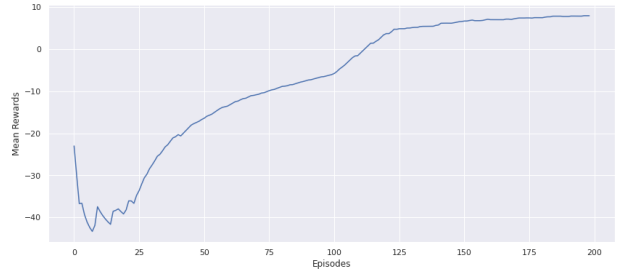


Figure 7: Mean Rewards vs. Episodes

### 3.1.2 A2C

In this experiment, we also provided the requested videos, and similar graphs as was generated for the DQN. The videos show that the A2C model provides an identical solution for the game, both in the middle and at the end of the learning process. Figures 8 and 9 show the total amount of steps and total reward for each episode. We can see that the A2C agent behaves very similar to the DQN agent, but it seems to be able to converge slightly faster than the DQN agent, and seem to be more stable than it. This can be supported by the faster decrease of the mean number of steps at an earlier stage, seen in Figure 10, and the faster increase in mean reward, seen in Figure 11, comparing to the ones shown in the case of the DQN agent.

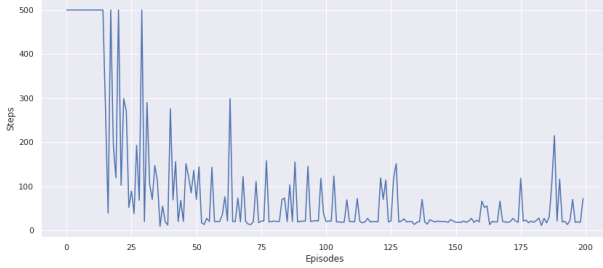


Figure 8: Steps vs. Episodes

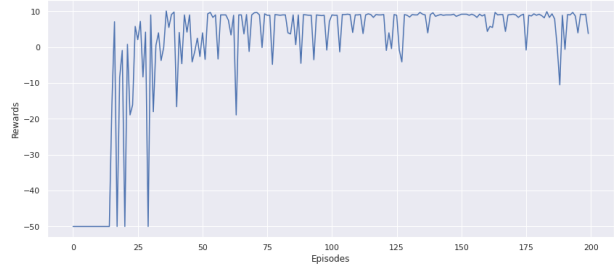


Figure 9: Rewards vs. Episodes

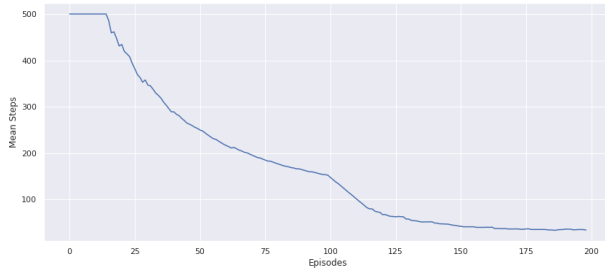


Figure 10: Mean Steps vs. Episodes

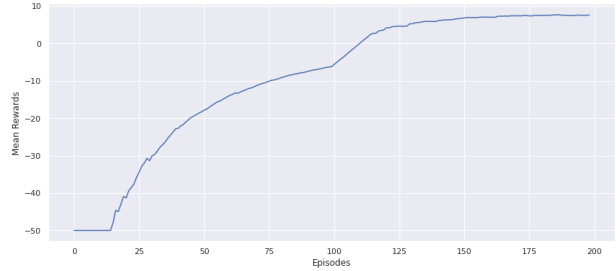


Figure 11: Mean Rewards vs. Episodes

## 3.2 Exercise 2: Random board configuration

As mentioned, in the second exercise we were requested to train a model that solves a game that its board configuration is randomly generated, and each episode is trained over a different game boards. In the following experiments of the exercise, we used the same two algorithms to try and solve the environments, and performed similar analyses that include the generation of the requested videos, and supplying the same four graphs for each agent. Since the models are being trained using different game boards, we evaluated the performance of each model over 100 newly generated boards, to see how well they were able to “generalize” and to learn the right approach to solve each and every environment. Here as well, all of the detailed results are reproducible.

### 3.2.1 DQN

In this experiment, the agent has not been able to converge, and did not provide any interesting results. Figure 12 shows that even after 500 episodes of learning, the agent still cannot solve most of the different variants of the game board, the episodes ends after the maximum number steps allowed without reaching the goal. Consequently, the final reward of each episode did not increase during the episodes, as can be seen

in Figure 13. As before, Figures 14 and 15 better show these trends, and we can see that the average reward and the average steps for 100 episodes does not improve as well, which leads us to the understanding that the model did not perform well. The unsatisfying results are also reflected in the evaluation part, provided in the Colab notebook, where the agent only managed to solve 2 out of 100 the randomly generated games. Due to all of the above, we assume that even those two successes were accidental, in much simpler environments. We then decided to test the A2C model and compare the results.

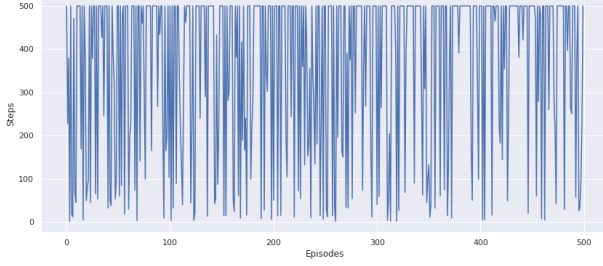


Figure 12: Steps vs. Episodes

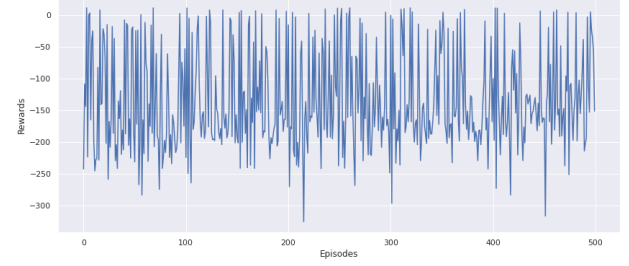


Figure 13: Rewards vs. Episodes

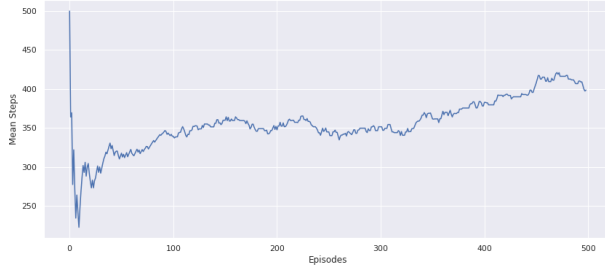


Figure 14: Mean Steps vs. Episodes

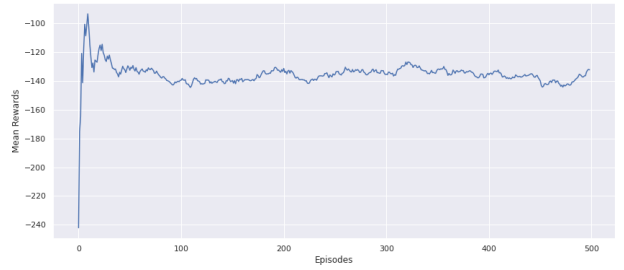


Figure 15: Mean Rewards vs. Episodes

### 3.2.2 A2C

Regarding the number of steps per episodes, the A2C agent training process is looking very similar to the DQN training process, as reflected in Figure 16. Unsurprisingly, Figure 18 shows almost no improvement regarding the mean number of steps for 100 episodes comparing to the equivalent graph in DQN. Regarding the final reward, Figure 17 presents some improvement over time, comparing to the beginning of the training process, although the increase does not look significant enough. Figure 19, on the other hand, shows a much more noticeable and somewhat steady improvement regarding the mean reward for 100 episodes, which led us to think it might perform better. To evaluate the model, we examined its performance over the same 100 randomly generated games we tested the DQN agent with. As we suspected, the agent was managed to solve 26 of them, comparing to just 2 in the case of the DQN. This means that those results are 13 times better than the results of the equivalent DQN agent presented, and the best results we were able to achieved.

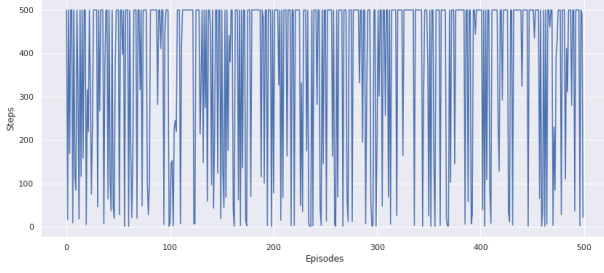


Figure 16: Steps vs. Episodes

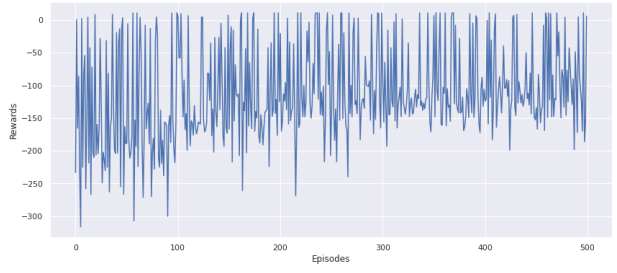


Figure 17: Rewards vs. Episodes

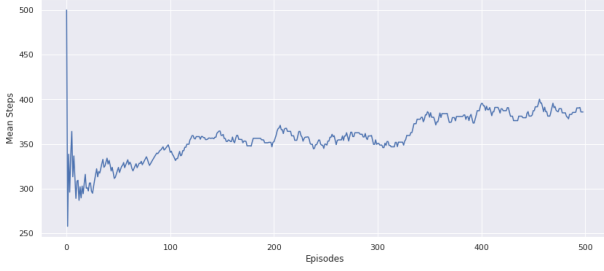


Figure 18: Mean Steps vs. Episodes

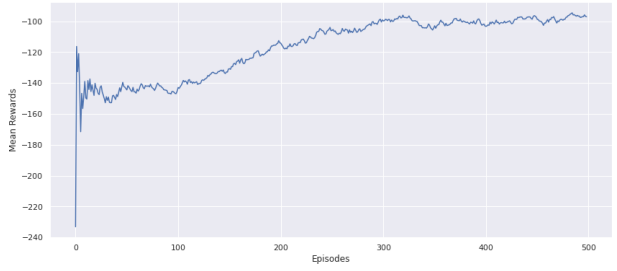


Figure 19: Mean Rewards vs. Episodes

We assumed that the performance of our model is affected by the amount of episodes that we let it train, and with additional training it might improve. Therefore, we conducted an extra experiment to test this assumption, where we tried to continue training the A2C model we already trained for another 500 episodes. We used the same randomness seed for each generated board, which means that the training process will use the same 500 randomly generated boards it already experienced, to try and improve its performance.

The results of this experiment are presented in Figures 20, 22, 21 and 23, which shows that the extra training did not improve neither the steps number nor the reward that the agent got. It is worth to mention that all the supplied graphs of this experiment show the complete learning process, meaning they present the information for the entire 1000 episodes of learning. Furthermore, we can notice a sudden degradation in the performance around the 500th episode, which seems to be recovered later on. This model was managed to solve 13 out of the same 100 randomly generated games as in the previous experiments, which is exactly half of the environments solved by the best performing model. Unfortunately, we were not able to prove that the caveat of our poor performance was the number of episodes. However, we also cannot decisively conclude that it was not affecting the performance, since we did not examine the model over 1000 different game boards, which may have led to much different results. We were unable to perform this kind of experiment, due to a lack of compute resources and time.

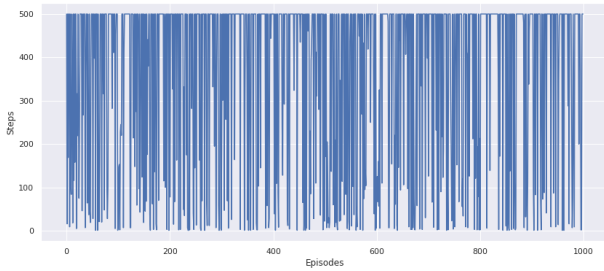


Figure 20: Steps vs. Episodes

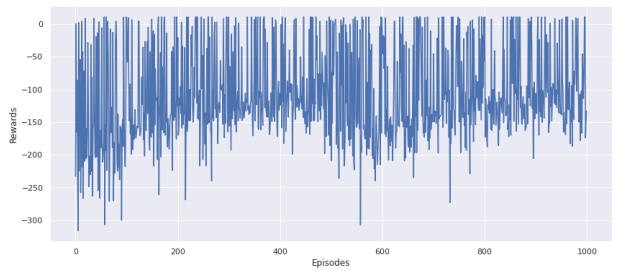


Figure 21: Rewards vs. Episodes



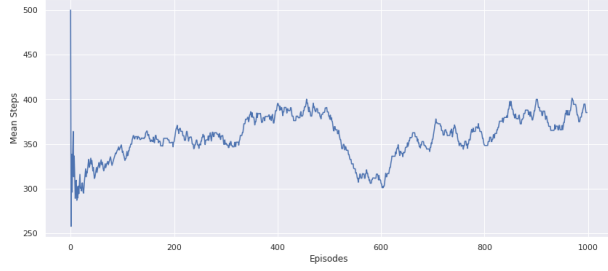


Figure 22: Mean Steps vs. Episodes

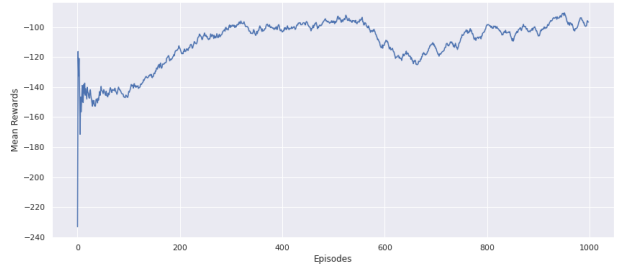


Figure 23: Mean Rewards vs. Episodes

## 4 Discussion

Our work compared two different approaches to solve the same problems. However, it was difficult to compare the same approach using different parameters for mainly two reasons: (a) it was very challenging to find a set of parameters that was performing well enough to solve the environment, in which almost none of the parameters and models configurations we tried were unsuccessful; and (b) we had a strict time constraint and very limited resources to conduct a large amount different experiments, which unfortunately prevented us from testing another parameters. For the same reasons, the results of the second exercise were not satisfying, and it is very likely that we might been able to achieve better models if we had access to a better computational power, and of course more time to extend our research.

In conclusion, in the first exercise both approaches provided satisfying results, and both of them managed to solve the fixed game board and reached the same solution. Also, both agents succeeded to do so even halfway through the learning process (i.e., after 100 episodes). Overall, the A2C agent did provide slightly better results, that are mostly reflected in the training process, where it managed to converge faster than the DQN agent, and seem to be more stable while doing so. As for the second exercise, the A2C agent was able to achieve significantly better results than the DQN agent, and solved more than 25% of the tested environments.

## 5 Code

You can access our public Colab notebook in the following link: <https://colab.research.google.com/drive/1rQ0ZhH1dY80VecVCqkV82tElf2iPepaF?usp=sharing>

Please refer to the notes provided in the *explainer.md* file attached to the submission box.