Final Project

March 14, 2023

Abstract

In this paper we implement deep learning based reinforcement learning algorithms in order to solve a modified version of the *Sokoban* game. The task combines two challenges - solving for a fixed environment and solving for random environments. For the sake of both tasks, we used DQN (Deep Q-learning Network) and PPO (Proximal Policy Optimization). While both approaches managed to solve the first challenge in a reasonable amount of learning time, the second one raised some challenges due to the unstable nature of its random environment. We conducted several experiments using various hyper-parameters, different reward shapes and innovating ideas. We then compared the results between the different approaches. Finally, we managed to achieve above 40% success rate in solving new environments the agent wasn't trained on using a well calibrated version of PPO.

1 Introduction

1.1 Reinforcement Learning intro

Reinforcement Learning is a machine learning sub-field that teaches a computer program (often called an "agent") to choose the best action within his state, in order to maximize expected returns over time. The returns (and game rewards) are calculated based on the environment rules. There are two fundamental tasks in reinforcement learning: prediction and control. In prediction tasks, we are given a policy and our goal is to evaluate it by estimating the value or Q value of taking actions following this policy. In control tasks, we don't know the policy, and the goal is to find the optimal policy that allows us to collect most rewards. In our context, PPO is a policy based method, while DQN is a value based method and both of them can be used in control tasks as well as prediction tasks.

1.2 The environment

Sokoban (warehouse keeper in Japanese) is a puzzle video game in which the player pushes boxes around in a warehouse, trying to get them to storage locations. The game was designed by Hiroyuki Imabayashi and published in December 1982. The environment used in this article involves one box and one storage location, with 13 possible actions at each state. A graphical representation of the environment is provided in figure 1.

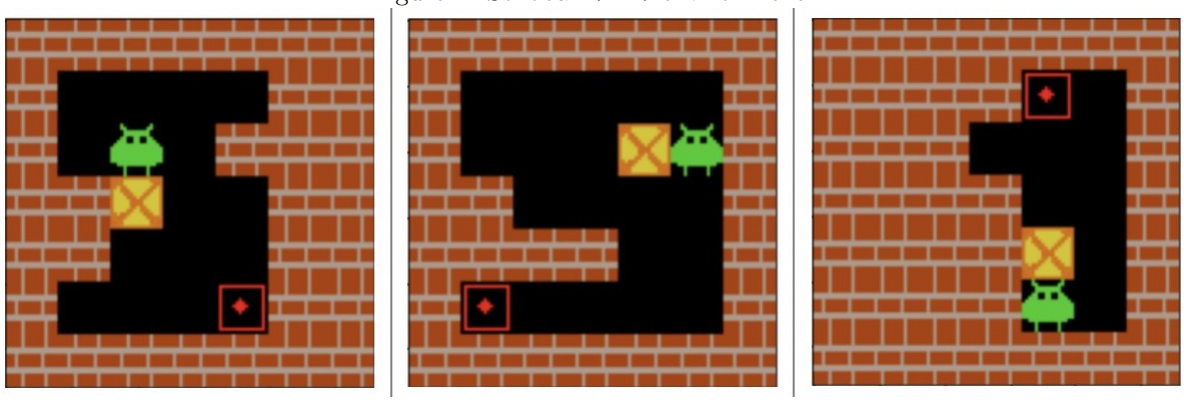
2 Methodology

2.1 Custom environment

In order to create the Sokoban environment, one has to import OpenAI's Gym library via supported coding language, where Python 3 was used for this article. The Sokoban environment we used differs from the original by the following:

- max steps parameter. (environment resets afterwards)
- 13 possible actions. (do nothing, 4 directions, 4 directions + push, 4 directions + pull)

Figure 1: Sokoban 7×7 environment



In addition to differences above, further modifications were made during our experiments (e.g custom reward system) using a custom wrapping class. Upon initialization, the following parameters are passed to the wrapping class:

1. Optional seed to get a specific environment after every reset.
2. Rendered image size. (e.g 24×24)
3. Whether the rendered image is colorized (3 channels) or not. (1 channel)
4. Allowed actions (all, only push and pull, only push)
5. Whether to use custom rewards or default rewards.
6. Box distance reward smoothing parameter λ.
7. Non-movement penalty amount ζ.

Due to the optional seed parameter, the wrapper is generic enough to handle both challenges. Inside the wrapper, we used custom rewards such that the agent will be rewarded according to the box distance from the storage, and will be punished if the the action he made resulted in the exact same state (e.g do nothing or trying to go through a wall). The reward system inherits from the original one, hence the agent will get a small negative reward for every step nonetheless. Formally, the reward at each step is defined as follows:

$$r_t = -0.1 + 15 \cdot \mathbb{I}[\text{is\_done}] + \frac{\lambda}{||p_b - p_s||_1} \cdot \mathbb{I}[\text{is\_box\_moved}] - \zeta \cdot \mathbb{I}[\text{is\_player\_moved}]$$

Where $||p_b - p_s||_1$ denotes the Manhattan distance between the box and the storage, λ is the reward smoothing parameter (typically $\lambda \in [1, 10]$) and $\mathbb{I}$ is the indicator function. Additionally, we customised the environment's `render` method, which will render an image of the current state according to the class parameters, provided on initialization. It was previously demonstrated, by Yang et al [1], that policy training by interacting with the environment using potential-based reward shaping improves the performance of the agent.

2.2 Architecture

In this section we present our models. We used Adam [2] optimizer throughout the entire paper.

2.2.1 Deep Q-Learning Algorithm (DQN)

DQN [3] is a value-based method that directly learns the optimal action-value function that maps states to action values. It uses a neural network to approximate the action-value function and applies a variant of the classic Q-learning algorithm to update the network's weights. DQN also uses a replay

memory buffer to store past experiences and improve sample efficiency, and a separate target network to provide a more stable target for the Q-value updates. Generally, DQN uses the following hyper-parameters: 1. Learning rate α. 2. Replay memory buffer size N. 3. Discount factor γ. 4. Exploration rate ε. 5. Target net update frequency c.

Our implementation also uses the following:

- Exponential exploration decay rate β.
- Minimum exploration rate ε_{min}.

In addition, if ε_{min} has been reached and the agent fails to solve the current environment, we multiply the exploration rate back up by a factor of two. Finally, the exploration rate is set using the following decay formula: $\varepsilon = max(\varepsilon \cdot \beta, \varepsilon_{min})$, or $\varepsilon = 2\varepsilon$ if $\varepsilon < \varepsilon_{min}$ and the agent failed to solve the environment.

2.2.2 Proximal Policy Optimization (PPO)

PPO [4] is based on the idea of optimizing a surrogate objective function that approximates the true objective function used in our learning. The surrogate objective function is used to update the agent's policy. One of the key features of PPO is the use of a clipping mechanism to ensure that updates to the policy are not too large. This helps to prevent the agent from deviating too far from its previous behavior, which can be especially important in environments where the consequences of bad actions can be severe. Our PPO implementation uses the following hyper-parameters: 1. Learning rate α. 2. `num_steps` - trajectory length. 3. `PPO_epochs` - number of epochs to run over the collected trajectory data. 4. GAE parameter τ - a smoothing parameter used for reducing the variance in training to make it more stable.

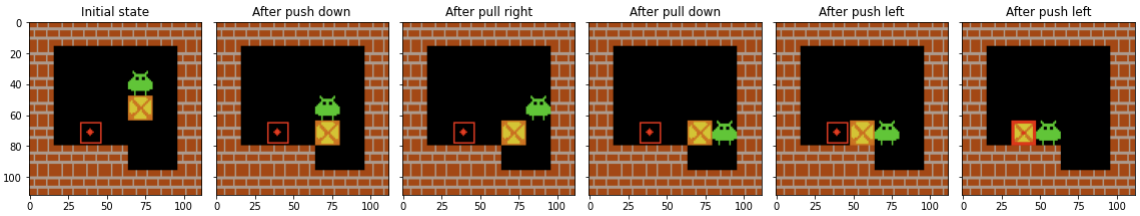
In general, PPO also uses policy clip range ε, an entropy coefficient c_1 and a value function coefficient c_2 which we have pre-set to commonly used values.

3 Results

3.1 Fixed Scenario

This challenge consists of one fixed environment, which the agent learns to solve by training solely on it. It is worth noting that the optimal solution involves 5 steps and can be achieved by more than one way. One of them is described in figure 2.

Figure 2: Fixed scenario optimal solution



3.1.1 Experiments

1. **Using DQN and PPO**:

   In order to solve the first challenge we use two different deep RL models. We try DQN and PPO on the fixed scenario and compare their results over 50 episodes. Figure 3 reveals that both our models work and converge to the optimal solution, with a slight advantage for DQN.

2. **Reward shaping**:

   We start by testing our custom reward system by comparing the learning procedure with the default reward system. We compare the average results of 5 DQN experiments to reduce variance. Indeed, figure 4, implies that our custom reward system accelerates learning. Consequently, we will continue using our custom reward system for the upcoming experiments.

Figure 3: DQN and PPO sanity check

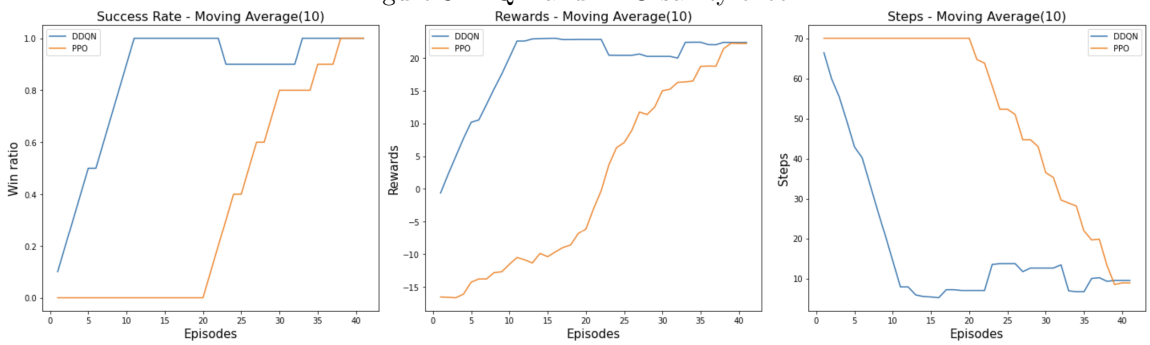
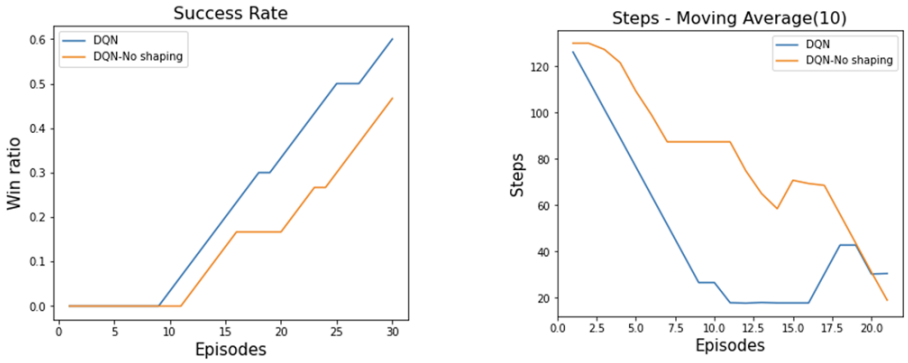


Figure 4: Reward System Comparison



3.2 Random Scenario

This challenge consists of a random environment after each reset. In this scenario, the agent needs to learn how to solve a general one box 7×7 Sokoban game.

3.2.1 Experiments - setup

We design our experiments in the following method to insure consistency and equal testing conditions among all experiments:

- We fix 400 random environments to be used during training in all experiments.
- During training we document steps per episode, rewards per episode and the total number of wins[1] per training phase.
- We Set additional 300 random environments, not overlapping with the training environments, to be used during the test phase to evaluate the agents.
- When evaluating the agent, we calculate win ratio and we calculate the average steps in took the agent to finish the games, considering only win games.
- Each experiment will run over 400 episodes, as one of the project limitations was not to exceed 500 iterations.

We run several experiments on DQN and on PPO and finally compare between the two models.

3.2.2 Experiments - DQN

1. **Different environment max steps**:

[1] A win occurs when the number of steps it took the agent to finish the game is lower than the max steps allowed.

In this experiment we try different max steps allowed per game for the Sokoban environment. That is, the game ends if the goal has been achieved or `max_steps` has been reached. We split this experiment into two - the first one evaluates general `max_steps` sizes and will be denoted as the *macro* experiment, while the second one will evaluate more specific `max_steps` range, based on the macro results and will be denoted as the *micro* experiment. We start our macro experiment by testing the following `max_steps`:

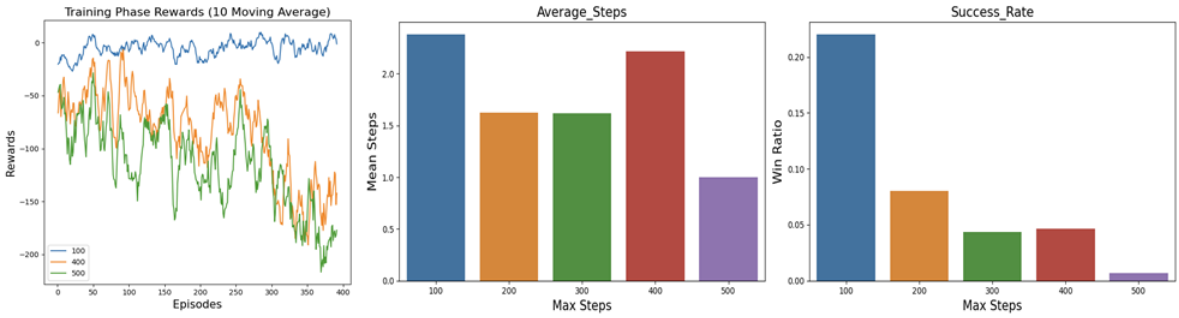
$$\texttt{max\_steps}_{mac} \in \{100, 200, 300, 400, 500\}$$

As implied from figure 5, using higher max steps didn't result in better win ratio, but the opposite. It is worth noting that `max_steps` of 500 had lower average steps because it tends to solve simpler environment scenarios, which often takes one step to solve. To be fair, the reward graph is biased because of the nature of the reward system, punishing for more steps, hence the reward is negatively correlated with our tested parameter, `max_steps`. But even after considering the bias, we can observe overall decreasing rewards for higher `max_steps` as opposed to `max_steps`=100 which seems to be increasing, which might indicate a healthy training process.

As the upper `max_steps` range introduced bad results, the lower range is worth further investigation. Thus, we run the micro experiment by testing the following `max_steps`:

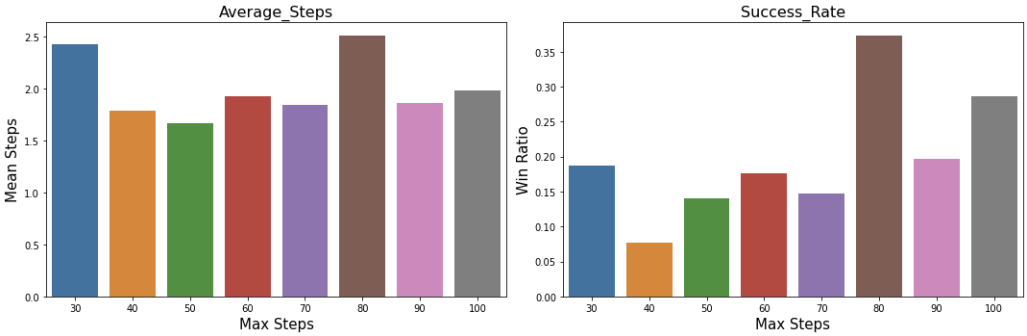
$$\texttt{max\_steps}_{mic} \in \{30, 40, 50, 60, 70, 80, 90, 100\}$$

Figure 5: Different Max Steps - Macro



Following the results from figure 6, limiting the agent with 80 steps seems to achieve the highest winning ratio and will be used in the upcoming DQN experiments.

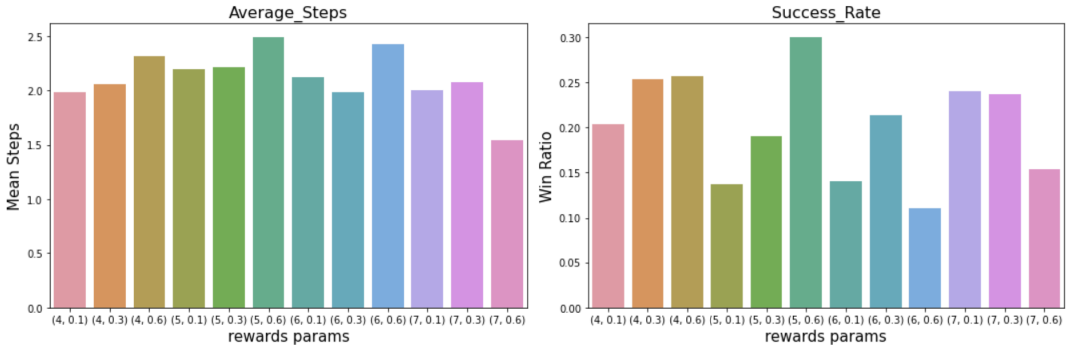
Figure 6: Different Max Steps - Micro



2. **Different Reward Shaping**:

   In this experiment we research the affect in tuning the custom reward system hyper-parameters. We set the custom reward smoothing parameter $\lambda \in \{4, 5, 6, 7\}$ and the non-movement penalty parameter $\zeta \in \{0.1, 0.3, 0.6\}$. $\lambda = 5$ and $\zeta = 0.6$ produced the best results as displayed in figure 7. These values were used in the subsequent experiments.

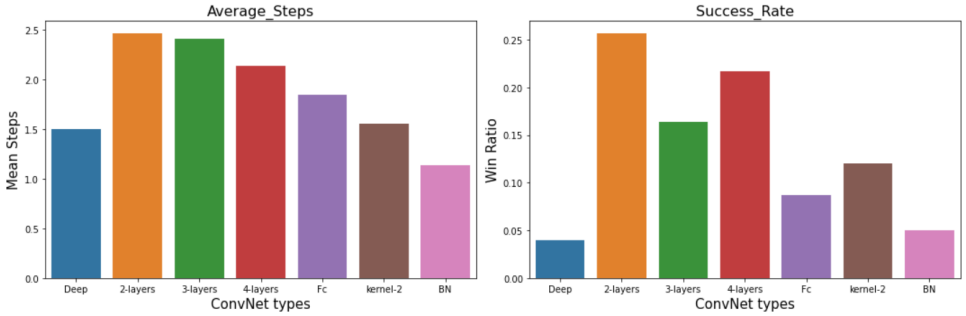
Figure 7: Different Reward Shaping



3. **Different Net Architectures**:

   In order to explore the influence of different neural network designs on the DQN agent, we investigate various architectures, each differs in number of layers, kernel sizes, channel depth, type and normalization. Figure 8 reveals an advantage for 2-layer conv layers backbone.

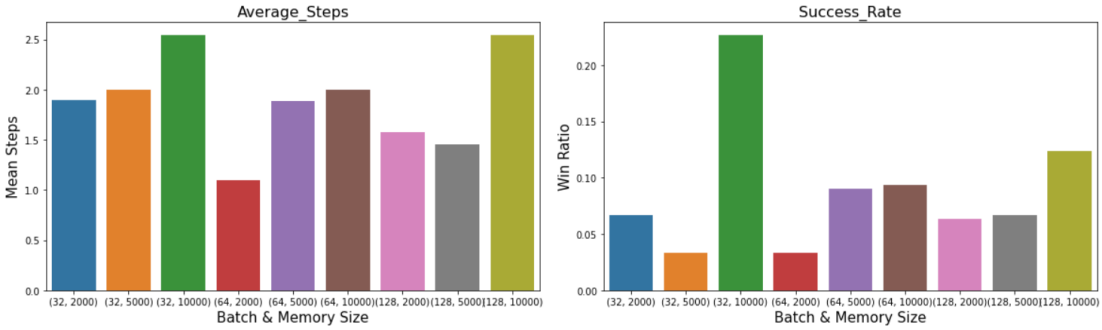
Figure 8: Different Net Architectures



4. **Different batch size and memory buffer size**:

   In this experiment we analyze the impact of different batch sizes used during training and various memory buffer sizes. We use value combinations of batch size $\in \{32, 64, 128\}$ and buffer size $\in \{2000, 5000, 10,000\}$. The results are depicted in figure 9. It seems like a buffer size of $10,000$ works best when combined with a batch size of 32.

Figure 9: Different batch size and memory buffer size

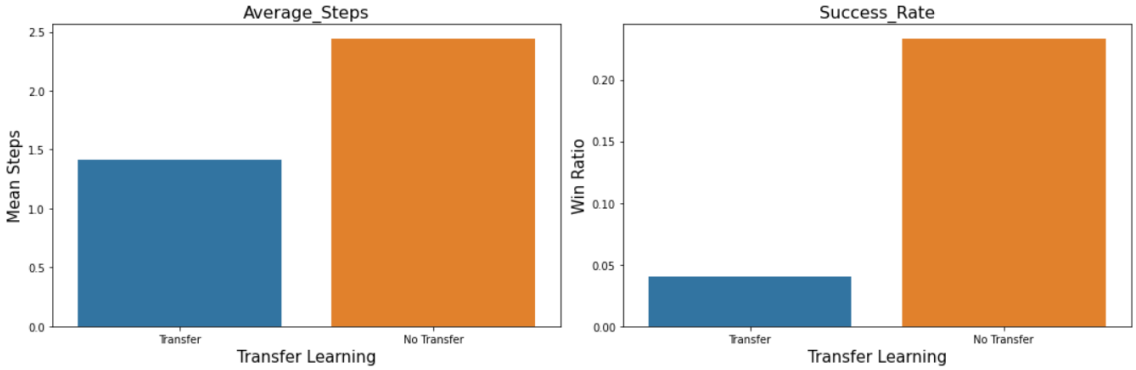


5. **Transfer Learning**:

   In machine learning, transfer learning often speeds up training, and it was shown that prior knowledge improves learning in Sokoban [5]. As a DQN model was able to solve the fixed scenario challenge, we were curious to explore whether an agent, trained on 50 pre-chosen environments,

each of which is fixed for 10 episodes, was able to generalize its behavior to handle different environments after the training phase is over. Unsurprisingly, figure 10 reveals it's not the case. This approach might have introduced *overfitting* to the trained environments, implied by introducing poor success rate of less than 5%.

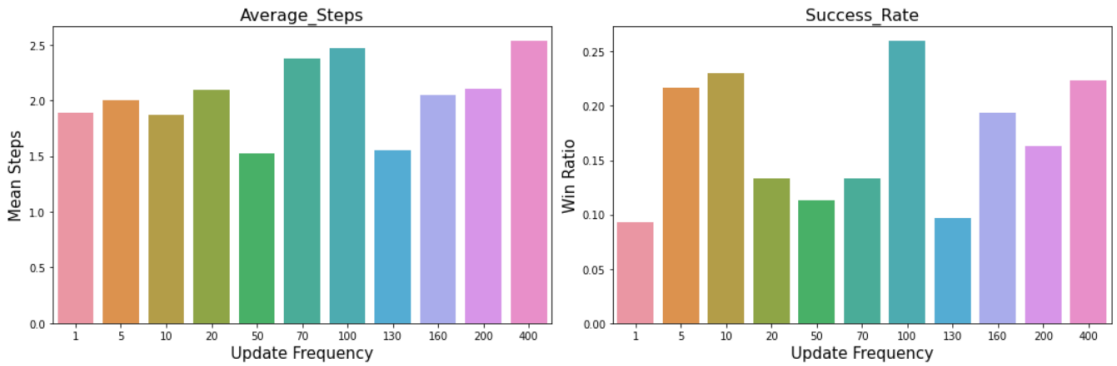
Figure 10: DQN Transfer Learning



6. **Network Update frequency**:

   In this experiment we investigate how the frequency of updating the target net with the weights of the policy might affect the agent's performance. Each such update consists of setting the target net weights to equal the policy net weights. We set the values as follows: frequency $\in \{1, 5, 10, 20, 50, 70, 100, 130, 160, 200, 400\}$. frequency=100 produced the best results as displayed in figure 11.

Figure 11: Network Update frequency and replay buffer size



7. **Different action spaces**:

   Our environment consists of 13 actions. Even without playing Sokoban before, one shall question the variety of this action space. It is fairly easy to conclude that some of them are redundant. For this reason, our next experiment tests the following action spaces:

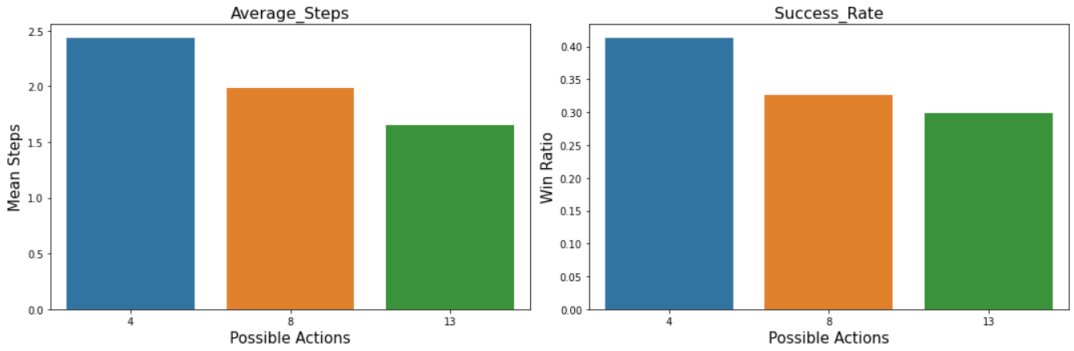
$$A = \{\text{do nothing}, \text{move all sides}, \text{push all sides}, \text{pull all sides}\}$$

$$A_{p\&p} = \{\text{push all sides}, \text{pull all sides}\}$$

$$A_p = \{\text{push all sides}\}$$

   Where $|A| = 13$, $|A_{p\&p}| = 8$ and $|A_p| = 4$.
   Figure 12 reveals negative correlation between $|A_i|$ and the model's success rate, with $|A_p|$ achieving above 40% winning rate.

Figure 12: DQN - Different Action Spaces

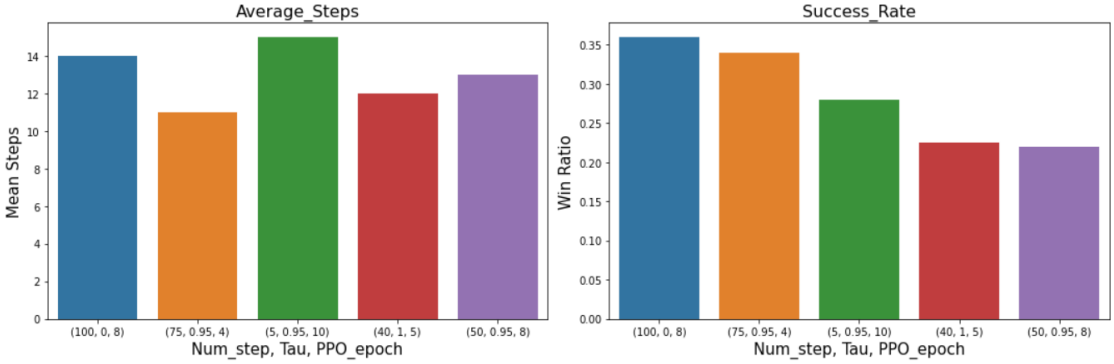


3.2.3 Experiments - PPO

1. **Core parameters grid search:**

   In this experiment we use grid search in order to find the best combination between the following hyper-parameters: 1. τ, `num_steps` and `ppo_epochs`. The best result was achieved when using `num_steps` $= 100$, $\tau = 0.95$ and `ppo_epochs` $= 8$ as shown in figure 13, however we will continue using 4 `ppo_epochs` and 75 `num_steps` for running time advantage.

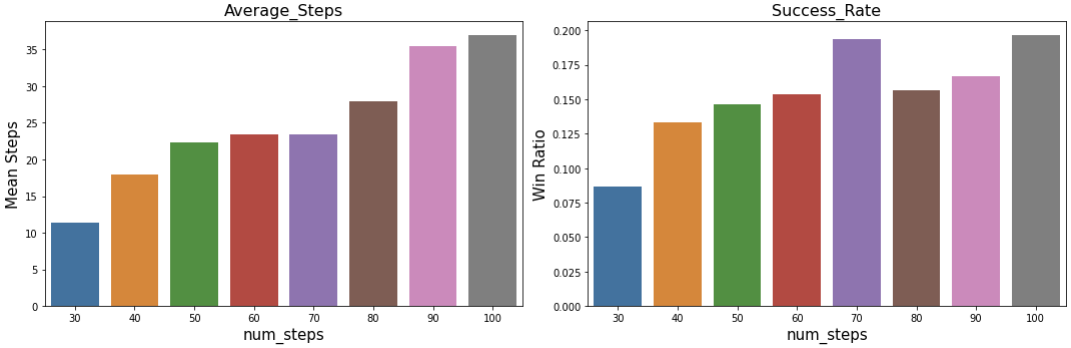
Figure 13: PPO - Core Parameters Grid Search



2. **Different environment max steps:**

   We wanted to test whether PPO is sensitive to the environment's maximum steps like DQN is. In this experiment, we only tested the smaller range of steps, corresponding to the DQN micro experiment. The results in figure 14 show overall good results for steps greater than 30, with preferred values of 70 and 100. For subsequent experiments we will use `max_steps` of 70.

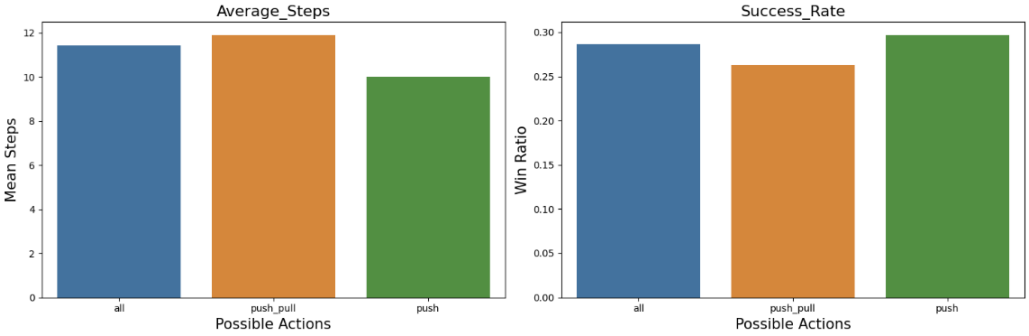
Figure 14: PPO - Environment max steps



3. **Different Action space:**

   Similar to Action space DQN experiment, different action spaces were tested on the PPO model as well. Interestingly, figure 15 reveals that PPO does not improve on smaller action spaces. In fact, the model doesn't seem to care. This might be due to the nature of PPO, giving the redundant actions a very low overall probability, as opposed to DQN which gives all action equal probability during exploration.

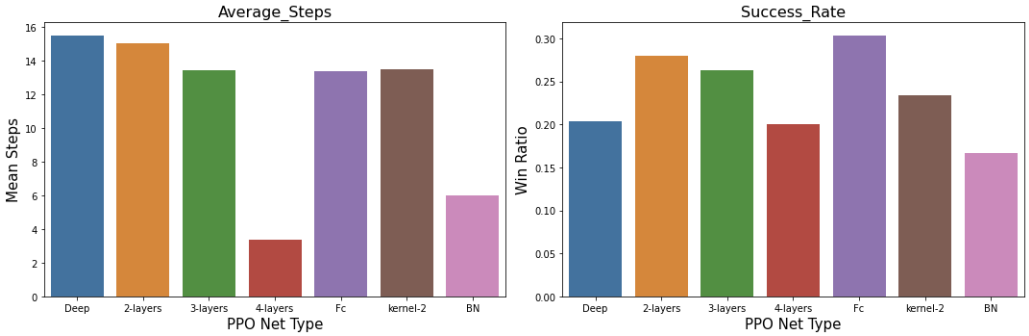
Figure 15: PPO - Different Action space



4. **Different net architectures:**

   Similar to the DQN Different net architectures experiment, we use the same neural network architectures to detect the optimal network when using the PPO model. As can be depicted from figure 16, using only FC layers (without convolutions) produced the best results. The low average steps for 4 conv-layers model suggests that this model could only solve easy environments.

Figure 16: PPO - Different net architectures



3.2.4 Final experiment - DQN vs PPO

Both DQN and PPO models used in this experiment are calibrated with the leading Hyper-parameters found in the subsequent experiments for each model. We run the each agent over 500 episodes and then test it over 300 different environments. As depicted in figure 17, PPO outperforms DQN in terms of winning ratio, scoring 46.6% and 34.3% respectively. In terms of average steps, PPO's average of nearly 7 steps implies solving more difficult levels and/or providing less determinism compared to DQN's average of just above 2 steps. As for the training process, figure 18 reveals similar trends between the models. Surprisingly, DQN presents overall higher reward levels, considering the fact that it scored less than PPO and used `max_steps` of 80 compared to PPO's `max_steps` of 70, which as we mentioned before, should lead to lower reward levels because of the step penalty.

Figure 17: DQN vs PPO - Test results

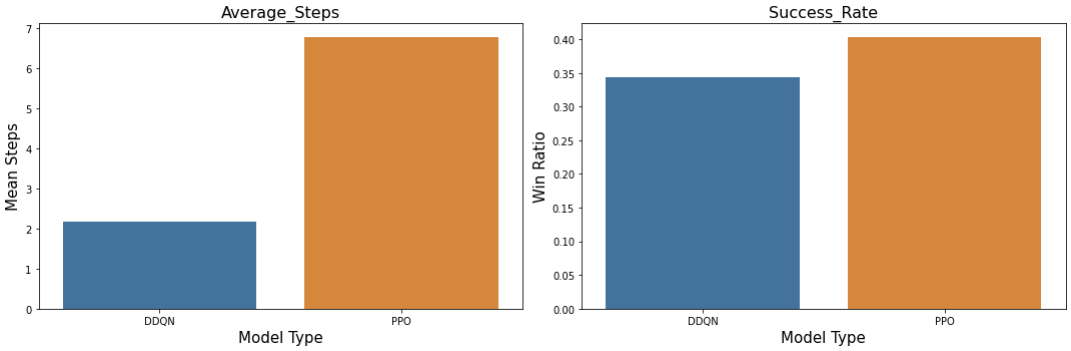
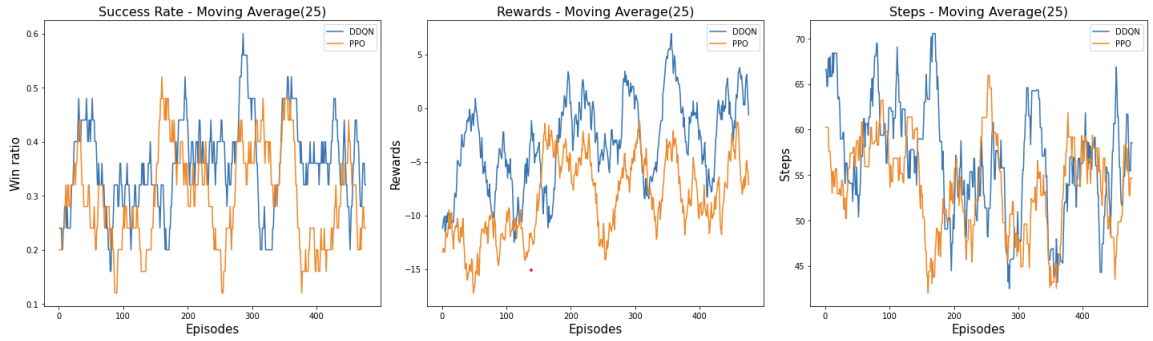


Figure 18: DQN vs PPO - Training Phase



4 Conclusions

In this project we implemented DQN and PPO algorithms in order to solve the Sokoban challenge. After conducting various experiments, we observed drastic accuracy changes when modifying certain hyper-parameters. It is especially relevant to PPO as it consists of more hyper-parameters than DQN. The second exercise, solving random environments, was challenging especially due to the 500 iterations limitation requirement. In a former study[6], PPO, when used as a benchmark, was able to reach around 60% success rate. It will be interesting to examine further the performance of the models without this constraint. Comparing the top models, We conclude that PPO provides overall better results than DQN in terms of test set accuracy, determined by success ratio over 300 unseen environments, which might suggest that stochastic policy based models are more suitable to this challenge than deterministic value-based models.

References

[1] Zhao Yang, Mike Preuss, and Aske Plaat. *Potential-based Reward Shaping in Sokoban*. 2021. DOI: 10.48550/ARXIV.2109.05022. URL: https://arxiv.org/abs/2109.05022.

[2] Diederik P. Kingma and Jimmy Ba. *Adam: A Method for Stochastic Optimization*. 2014. DOI: 10.48550/ARXIV.1412.6980. URL: https://arxiv.org/abs/1412.6980.

[3] Volodymyr Mnih et al. *Playing Atari with Deep Reinforcement Learning*. 2013. DOI: 10.48550/ARXIV.1312.5602. URL: https://arxiv.org/abs/1312.5602.

[4] John Schulman et al. *Proximal Policy Optimization Algorithms*. 2017. DOI: 10.48550/ARXIV.1707.06347. URL: https://arxiv.org/abs/1707.06347.

[5] Zhao Yang, Mike Preuss, and Aske Plaat. *Transfer Learning and Curriculum Learning in Sokoban*. 2021. DOI: 10.48550/ARXIV.2105.11702. URL: https://arxiv.org/abs/2105.11702.

[6] Xie Yiheng et al. *Learning Generalizable Behavior via Visual Rewrite Rules*. 2021. DOI: 10.48550/arXiv.2112.05218. URL: https://arxiv.org/abs/2112.05218.