# Sokoban RL

**Deval Patel** DEVALM.PATEL@MAIL.UTORONTO.CA
*1005468655*

**Kyrel Jerome** KYREL.JEROME@MAIL.UTORONTO.CA
*1004186169*

## Abstract

We proceed to implement a custom Sokoban environment using OpenAI Gym's custom environment feature. To test our environment, we use Stable Baselines 3 algorithms, particularly A2C, DQN and PPO. We experimented with different levels as well as reward schemes.

## 1. Introduction

Our project creates a custom RL environment based off of the puzzle game Sokoban. Then we proceed to test our environment on 3 state of the art algorithms; A2C, DQN and PPO. We will talk more about them later in this report. Our project's repository can be found at `https://github.com/deval-patel/sokobanRL`.

Our original idea for the project was to have 2 RL agents, where one would learn how to solve the puzzle, and the other would be able to learn how to *create* new puzzles. However, it was deemed out of scope, so we did not end up doing PCGRL based on the paper by Khalifa et al. (2020)

Instead, as a TA suggested we make our own environment and analyze it with different algorithms.

### 1.1 What is Sokoban?

Sokoban is a grid-based single player puzzle game of Japanese origins where the player must make movements in one of the 4 cardinal directions throughout the game to cover checkpoints with boxes by pushing the boxes onto the checkpoints Wikipedia (2021). This is the basics of the game, however there is often a turn limit associated with a level. This means that in a certain number of turns, the player must solve the level otherwise they lose. Another alternative approach is a push limit. This limits the number of times they are allowed to push a box. Both of these constraints adds an extra difficulty to the simple puzzle solving game, making the problem of solving Sokoban Levels an NP Hard problem.

## 2. Environment

As mentioned previously, we made our own custom RL environment. In order to do this, we had to first make the game itself. The code is quite modular following good software design principals and can be found at `sokobanRL/gym\_498\_sokoban`. The main game logic is found in `sokobanRL/gym_498_sokoban/SokobanGame.py`. To run the game in text format, you can run `sokobanRL/gym_498_sokoban/SokobanController.py`.

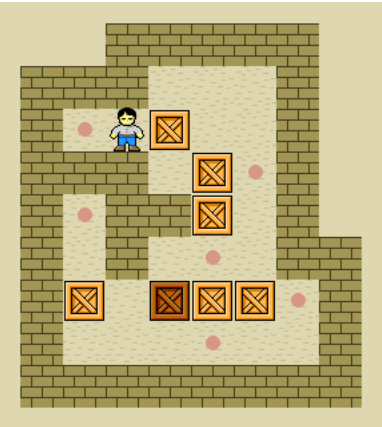

Figure 1: Sample Sokoban Map

### 2.1 Creating Custom Gym Environment

We used OpenAI Gym's environment creation guide in order to create our custom gym environment Openai (2021). We chose to do this as OpenAI's gym provides a good template for environment code and it was easy to pick up. Furthermore, we ended up using Raffin et al. (2021) which contains implementations of state of the art algorithms in order to test our environment. The good thing about stable baselines3 is its support for gym environments. Our custom gym environments can be found under `sokobanRL/gym_498_sokoban/envs/`.

#### 2.1.1 Observation Space

Our observation space is a Box of type `np.uint8`. We pass in the state of the grid to the agent as an observation. Therefore, we used a Box with the shape of the grid (`shape=(height, width)`). We also set $low = 0$ and $high = 6$ limits on the values of our box since that represents the unique pieces we have for our game.

**Pieces Mapping:**

```
EMPTY: int = 0
WALL: int = 1
TARGET: int = 2
BOX: int = 3
BOX_ON_TARGET: int = 4
PLAYER: int = 5
PLAYER_ON_TARGET: int = 6
```

#### 2.1.2 Action Space

Our action space is discrete and of size 4. The agent is able to move the player in one of the 4 cardinal directions, UP, DOWN, LEFT or RIGHT. Each of these actions are mapped to integers, which resulted in our discrete action space.

#### 2.1.3 Transitions

Depending on the action that the agent chooses, a move corresponding to that action will be made (moving the player in the direction of the action, if valid). The algorithm can be summarized in the following steps (can be found under `sokobanRL/gym_498_sokoban/SokobanGame.py:132`).

1. Check for a valid move given an action.
2. Store current state in stack
3. Check if the action requires a box to be pushed. If so, move the box.
4. Move the player to their new position.
5. Decrement the turn limit by 1 since a succesful turn has been made.
6. Return the reward based on the move.

#### 2.1.4 Rewards

Reinforcement Learning is based on the idea of an agent maximizing the reward it can achieve. As a result, the agent needs to be awarded for making actions in our environment. We define the following reward system (`sokobanRL/gym_498_sokoban/RewardSystem.py`):

**Reward for Move (no effect):** $-0.05$
This is a slight negative reward for making any move. We provide a slight negative reward so that the agent does not make unnecessary moves, since turn limit is an optional factor in the game as well. We want the agent to win in the optimal number of turns.

**Reward for Invalid Move:** $-3.0$
If the agent takes an action which makes them move the player into the wall or into a box which cannot be pushed, they receive a slightly higher negative reward since this move does not do anything productive.

**Reward for Moving Box on Target:** 8.0 We reward the agent positively for moving the boxes onto the target, as this is a part of the win condition. Remember, to win this puzzle game, the agent is required to move all boxes onto the targets.

**Reward for Moving Box off Target:** $-8.0$
On the other hand, if the agent moves a box off of a target, then this is likely going against the win condition, so we penalize the agent to avoid this mistake.

**Reward for Victory:** 30.0
Finally, if the agent completes the level, they are rewarded with a largely positive reward.

**Reward for Loss:** $-30.0$
If the agent is to lose the game, they are penalized with a largely negative reward.

## 3. Algorithms

For our algorithms, we used Stable Baselines3 as it contains stable implementations of state of the art algorithms. Although SB3 has 7 RL algorithms at its disposal (A2C, DDPG, DQN, HER, PPO, SAC, TD3) we were limited to using A2C, DQN and PPO. This is because the algorithms use case is limited on the action and observation spaces. Since we have a discrete action space and a box observation space, we could not use DDPG, SAC and TD3. A2C and DQN were covered in this courses' material, however PPO was not and we were excited to use a new algorithm and explore its performance.

| Space | Action | Observation |
|---|---|---|
| Discrete | ✔ | ✔ |
| Box | ✔ | ✔ |
| MultiDiscrete | ✔ | ✔ |
| MultiBinary | ✔ | ✔ |
| Dict | ✘ | ✔ |

Figure 2: A2C Use Cases

### 3.1 A2C

Advantage Actor Critic as learned in lecture uses 2 models (the actor and critic). The actor learns the policy, whereas the critic evaluates our actions and acts as our value function. The implementation that SB3 uses (ACKTR: Actor Critic using Kronecker-factored Trust Region) combines three techniques: actor-critic methods, trust region optmization and distributed Kronecker factorization Wu (2020). ACKTR takes a step in the *natural gradient* instead of the gradient direction. As we learned in class, limiting the KL divergence helps with consistency, so that our policy does not drastically change on an update (this could lead to worse performance).

### 3.2 DQN

Deep Q-Learning learns Q values with a neural network. As we learned in class, there are several hyperparameters which can be configured for DQN and there are also algorithmic variants which build on the fundamentals that DQN has set. Since DQN can succesfully perform with high dimension inputs, it was a good option for our environment Mnih et al. (2015). SB3's implementation of DQN builds on Fitted Q-Iteration (FQI) and uses tricks for stable learning with the neural network. It uses a replay buffer, a target network as well as gradient clipping Raffin et al. (2021). The implementation is just a vanilla DQN, there is

no further support for Double-DQN, Dueling-DQN or Prioritized Experience Replay. This was unfortunate as we wished to experiment with further optimizations of DQN that we learned and implemented in the course.

### 3.3 PPO

Proximal Policy Optimization is an algorithm that was not covered in this course. The algorithm combines ideas from A2C such as having multiple workers and it also uses a trust region to improve the actor. When PPO makes an update for its policy, the new policy should not be *too far* from the old policy. Similarly to ACKTR, PPO uses clipping in order to avoid too large of an update. Furthermore, the advantages computed are normalized and the critic can also be clipped. These are some further modifications made by OpenAI compared to the original paper on PPO by Schulman et al. (2017).

Something that we found interesting is that SB3's new PPO algorithm does not use an adaptive KL penalty. It uses a different objective function which is not common in other algorithms. This makes the algorithm simpler and removes adaptive updates. However, it matches ACER's (Actor Critic with Experience Replay) performance despite being simpler to implement.

$$L^{CLIP}(\theta) = \hat{E}_t[min(r_t(\theta)\hat{A}_t, clip(r_t(\theta), 1-\varepsilon, 1+\varepsilon)\hat{A}_t)\,]$$

- $\theta$ is the policy parameter
- $\hat{E}_t$ denotes the empirical expectation over timesteps
- $r_t$ is the ratio of the probability under the new and old policies, respectively
- $\hat{A}_t$is the estimated advantage at time $t$
- $\varepsilon$ is a hyperparameter, usually 0.1 or 0.2

Figure 3: Trust Region Update Schulman (2020)

## 4. Experiments

To enable the use of the above algorithms we implemented sets of generalized classes around the Stable Baselines engine. The classes enabled us to interchangeably interact with and design constraints for the individual agents. To enable the Stable Baselines framework to properly interact with the sokoban environment, we created a Solid Baselines Custom Environment such that the agents and pre-built algorithms can properly interact with the environment.

Due to time constraints along with issues interacting with the Stable Baselines logging interface, we were unable to complete our assessment of which algorithms perform best on a multitude of maps. Our goal for evaluation was to first generate and train a learning agent on the training maps, with the hope that the agent would diversify to function on the validation and test maps.

To further test the agents, our goal was to test using differing sets of hyper-parameters along with neural network sizes. In the future, we would choose to select the bot during training that best functions on the validation test set, allowing us to ensure that the model is not over-fitting to the training maps.

## 5. Conclusion

By implementing a custom RL environment for Sokoban, our work provides the required base for multiple further environment projects. Specifically, in later projects this implementation may be used to explore the use of machine learning agents to procedurally generate levels, using a pre-trained agent to determine the viability or difficulty of the output levels. In its current state, our project presents a platform and tooling to assess the functionality of discrete action machine learning algorithms on the Sokoban game.

## References

Ahmed Khalifa, Philip Bontrager, Sam Earle, and Julian Togelius. Pcgrl: Procedural content generation via reinforcement learning. In *Proceedings of the AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment*, volume 16, pages 95–101, 2020.

Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A Rusu, Joel Veness, Marc G Bellemare, Alex Graves, Martin Riedmiller, Andreas K Fidjeland, Georg Ostrovski, et al. Human-level control through deep reinforcement learning. *nature*, 518(7540):529–533, 2015.

Openai. Creating environments, openai gym, Dec 2021. URL `https://github.com/openai/gym/blob/master/docs/creating_environments.md`.

Antonin Raffin, Ashley Hill, Adam Gleave, Anssi Kanervisto, Maximilian Ernestus, and Noah Dormann. Stable-baselines3: Reliable reinforcement learning implementations. *Journal of Machine Learning Research*, 22(268):1–8, 2021. URL `http://jmlr.org/papers/v22/20-1364.html`.

John Schulman. Proximal policy optimization, Sep 2020. URL `https://openai.com/blog/openai-baselines-ppo/`.

John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal policy optimization algorithms. *arXiv preprint arXiv:1707.06347*, 2017.

Wikipedia. Sokoban, Nov 2021. URL `https://en.wikipedia.org/wiki/Sokoban`.

Yuhuai Wu. Openai baselines: Acktr and a2c, Jun 2020. URL `https://openai.com/blog/baselines-acktr-a2c/`.