EEC 289A Final Project Report
Attempting to Solve *Sokoban* using DQN

Kolin Guo[*]
University of California, Davis
`kolguo@ucdavis.edu`

Daniel Vallejo[*]
University of California, Davis
`davallejo@ucdavis.edu`

Fengqiao Yang[*]
University of California, Davis
`yfqyang@ucdavis.edu`

Abstract

In this project, we are working on **Option 2**: applying existing algorithms to an application involving significant programming. Specifically, we employed the Deep Q-learning Network (DQN) to solve the *Sokoban* game. The game environment we used is an open-sounced GitHub repository `gym-sokoban` which is based on OpenAI Gym Environment. The DQN is designed using three 3D convolutional layers followed by two fully-connected layers in Tensorflow 2.1.0. After training, the network is evaluated on a separate validation dataset and game-play visualizations.

1 Introduction

Sokoban (a.k.a., warehouse keeper) is a Japanese puzzle video game invented in 1982. In this game, the player pushes boxes in a warehouse, trying to get them to target locations. A screenshot is shown in Figure 1. The goal is to explore the optimal strategy for pushing all boxes into targets with fewest moves.

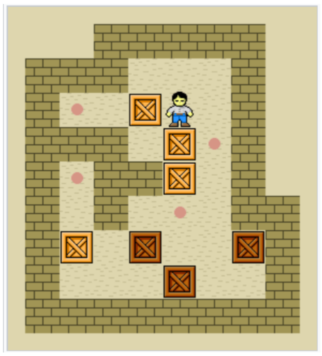


Figure 1: Screenshot of the *Sokoban* Game from Wikipedia (Destinations of the boxes are marked with red dots)

[*]Alphabetically ordered by last names. All three authors belong to the Department of Electrical and Computer Engineering.

However, the possibility of making irreversible mistakes (deadlocks) makes *Sokoban* very challenging. In a *Sokoban* level, a location is call *dead* if a box placed on it can never be pushed to a goal. Therefore, pushing a box on a dead location will result in a deadlock. The dead locations in an example *Sokoban* level are shown in Figure 2.

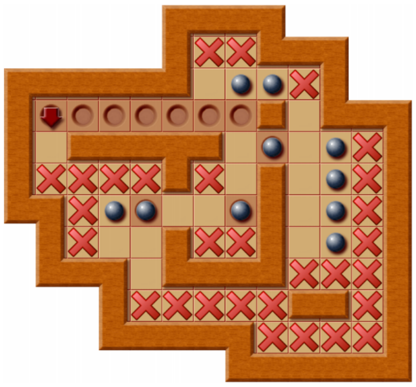


Figure 2: Example deadlock in **Sokoban**. All dead locations are marked with a red X.

Because of the deadlock possibility, *Sokoban* is been proved to be a PSPACE-complete decision problem [1]. In computational complexity theory, a PSPACE-complete decision problem is the hardest problem that can be solved by a Turing machine using a polynomial amount of space. Also, it has been shown that in a Sokoban level of 20x20 grid size, the average branching factor is 12 [2]. This means that on average, a player has 12 legal moves at each game state. In comparison, Rubik's cube has an average branching factor of 13.35, chess has a 35 branching factor and the Go game has an estimated 250 branching factor [2].

Attempting to solve the *Sokoban* game, we employed the Deep Q-learning Network (DQN) first introduced in DeepMind's Atari paper [3]. The open-source *Sokoban* game interface we used is described in Section 2.1 and the DQN algorithm is described in Section 2.2. The training, evaluating, and game-play results are discussed in Section 3.1 3.2, 3.3.

Individual Contribution Throughout this project, Kolin Guo is responsible for implementing the training code (`train.py`) and all utility functions (`experience_replay.py`, `network.py`, `state_buffer.py`, and `utils.py`), setting up training environment using `docker`, initializing training, and collecting training results using `TensorBoard` visualization tool.

Daniel Vallejo is responsible for implementing the evaluation code (`test.py`), generating evaluation results for the trained DQNs, and comparing the results for different training configurations.

Fengqiao Yang is responsible for implementing the game-play visualization code (`play.py`), generating game-play episodes using the trained DQNs, and analyzing the game-play results.

2 Implementation Details

2.1 Open-Source Sokoban Environment Interface

We built our project based on an open-source *Sokoban* environment interface found on GitHub [4]. The interface is implemented based on the OpenAI Gym Environment [5]. This interface enables a simple one-line code to interact with the *Sokoban* environment:

```
observation, reward, terminal, info = env.step(action)
```

Puzzle Generation The original `gym-sokoban` environment generates a *Sokoban* puzzle using a three-step depth-first-search (DFS) graph searching technique: topology generation, placement of targets and players, and reverse playing. However, this means that on average each puzzle takes about 10 seconds to generate.

To improve the efficiency, we used the `Boxoban` variant included in the `gym-sokoban` repository. This variant randomly samples DeepMind's pregenerated *Sokoban* puzzles which are divided into training (900k puzzles), validation (100k puzzles), and testing (1k puzzles) datasets. The difficulties of these puzzles are *unfiltered*: some of the puzzles are very difficult while some are relatively easy.

Each of these puzzles has a 10-by-10 grid size with 4 boxes. Because the outer boundary must be walls, each puzzle has a 8-by-8 non-deterministic grid space. According to the rendering configuration, there are 7 rendering textures: wall, floor, target, two for boxes (on-target and off-target), and two for players (on-target and off-target). A quick calculation shows that the game setup contains at most $7^{8\times 8} = 7^{64} \approx 10^{54}$ number of possible states. Given this huge state-space size, it is sensible to employ function-approximation based reinforcement learning algorithms.

Actions The `gym-sokoban` interface provides 9 actions to interact with the environment: no operation, 4 pushes (up, down, left, right), and 4 moves (up, down left, right). According to its implementation, *"move"* simply moves the player if there is a unoccupied location (no blocking box or wall) in the direction while *"push"* tries to move an adjacent box if the next location behind the box is unoccupied. In case there is no box at the adjacent location, the *push* action is handled the same way as the *move* action into the same direction.

Because the *push* actions overlaps the *move* actions and the original *Sokoban* game only includes *push* actions for human playing, we only used the 4 *push* actions as our action space during implementation.

Rewards The original rewards in the `gym-sokoban` interface are summarized in the "Original Rewards" column of Table 1 below. Because the reward of pushing a box on target is only 10 times the penalty of performing a step, 10 steps of walking (i.e. moving towards a box, pushing a box towards a target, and so on) will overturn the reward of pushing a box on target. This might reduce our DQN's ability to learn from pushing a box on target. Therefore, we modified the original `gym-sokoban` implementation and added a function for customizing reward values. In our training and evaluation steps, we separated this two setups of rewards and compared their performances.

Table 1: Rewards

Action	Original Rewards	$10\times$ Rewards
Perform a Step	-0.1	-0.1
Push Box on Target	1.0	10.0
Push Box off Target	-1.0	-10.0
Pushed All Boxes on Targets	10.0	100.0

2.2 Deep Q-Learning Algorithm

We chose the deep Q-learning algorithm originally presented in DeepMind's Atari paper as our reinforcement learning algorithm [3] [6]. The pseudocode of DQN is described in Algorithm 1. We adapted part of our code implementation from the `DQN_Atari` GitHub repository (`experience_replay.py`, `state_buffer.py`) [7]. The remaining code implementation are mostly customized to our project. Also, the `DQN_Atari` repository is implemented in `Tensorflow 1.5` while we switched to the more efficient `Tensorflow 2.1.0` for faster training, simpler network definitions using `tf.keras` and better visualization via `TensorBoard 2.1.1`.

Line-By-Line Walkthrough of Deep Q-Learning Algorithm At the start of our `train.py`, we initialized a replay memory to size of 1 million steps, an action-value function Q with random weights and another target action-value function $\hat{Q}$ with same weights as Q.

Then for all number of training steps T, we used ε-greedy to select an action a_t, executed the action and observed reward r_t and grid state x_{t+1}. The state x_{t+1} was then preprocessed and placed into a state buffer of size 4. The grid transition $(\phi_t, a_t, r_t, \phi_{t+1})$ was added to the replay memory. If next grid x_{t+1} reached terminal, the environment was reset and the initial grid was preprocessed and set as initial state.

During training, we sampled a random minibatch of 32 state transitions (s_t, a_t, r_t, s_{t+1}) from the replay memory. Using the target action-value function $\hat{Q}$, we computed the target return y_j and performed a gradient descent step on $[y_j - Q(s_j, a_j; \theta)]^2$ with respect to the weights θ. Also, we updated the target action-value function $\hat{Q}$ using Q every $10,000$ steps.

Algorithm 1: Deep Q-Learning with Experience Replay

```
1  Initialize replay memory $D$ to capacity $N$
2  Initialize action-value function $Q$ with random weights $\theta$
3  Initialize target action-value function $\hat{Q}$ with weights $\theta^- = \theta$
4  for $t$ = 1, $T$ do
5      With probability $\epsilon$ select a random action $a_t$
6      otherwise select $a_t = \arg\max_a Q(s_t, a; \theta)$
7      Execute action $a_t$ and observe reward $r_t$ and grid $x_{t+1}$
8      Preprocess $\phi_{t+1} = \phi(x_{t+1})$ and set state $s_{t+1} = \{s_t, \phi_{t+1}\}$
9      Store grid transition $(\phi_t, a_t, r_t, \phi_{t+1})$ in $D$
10     if $x_{t+1}$ is terminal state then
11         Reset environment
12         Preprocess initial grid $\phi_1 = \phi(x_1)$ and set initial state $s_1 = \{\phi_1\}$
13     Sample random minibatch of state transitions $(s_j, a_j, r_j, s_{j+1})$ from D
14     Set $y_j = \begin{cases} r_j & \text{if episode terminates at step } j+1 \\ r_j + \gamma \max_{a'} \hat{Q}(s_{j+1}, a'; \theta^-) & \text{otherwise} \end{cases}$
15     Perform a gradient descent step on $[y_j - Q(s_j, a_j; \theta)]^2$ with respect to the weights $\theta$
16     Every $C$ steps update $\hat{Q} = Q$
17 end
```

Experience Replay The deep Q-learning algorithm described in Algorithm 1 used a technique known as experience replay [8] in which we stored the agent's experiences at each time-step, $e_t = (s_t, a_t, r_t, s_{t+1})$, in a replay memory $D_t = \{e_1, \ldots, e_t\}$. At training time, we sampled a random minibatch of state transitions and apply gradient descent. This mainly has three advantages over standard online Q-leaning. First, each step of experience is potentially used in many network weight updates, allowing for greater data efficiency. Second, when learning on-policy, the current learned parameters θ determine the next data sample that the parameters are trained on, creating undesired feedback training loops. Third, learning directly from consecutive data samples is inefficient due to the strong correlations between the samples; randomizing the samples breaks these correlations and therefore reduces the loss variance.

Target Action-Value Function $\hat{Q}$ The deep Q-learning algorithm also used a separate network $\hat{Q}$ for generating targets y_j in the Q-learning update step. Every C steps, we cloned the training network Q to obtain a target network $\hat{Q}$, and use $\hat{Q}$ for generating the Q-learning targets y_j for the following C updates to Q. This technique makes the learning algorithm more stable compared to standard online Q-learning. In standard online Q-learning, an update that increases $Q(s_t, a_t)$ often also increases $Q(s_{t+1}, a)$ for all a and thus also increases the target y_j, possibly leading to oscillations or divergence of the function. Generating the targets y_j using an older set of weights adds a C-step delay between the time an update to Q is made and the time the update affects the target y_j, making divergence or oscillations much more unlikely.

2.3 Action-Value Function Approximation Network

For our action-value function $Q(s_t, a; \theta)$, we chose a convolutional neural network (CNN). The network architecture is shown in Figure 3. Each grid state x_t is encoded by seven 10-by-10 feature planes of the 7 rendering textures and the input state s_t contains 4 consecutive grid states. Thus, the input for our CNN has dimension of $(10, 10, 7, 4)$. To extract features from this sequential image-like data, we used three 3D convolutional layers: 32 $(7, 7, 5)$ kernels, 64 $(5, 5, 5)$ kernels, and 64 $(3, 3, 3)$ kernels. After the three convolutional layers, we connected two fully-connected layers and outputted 4 action-values.

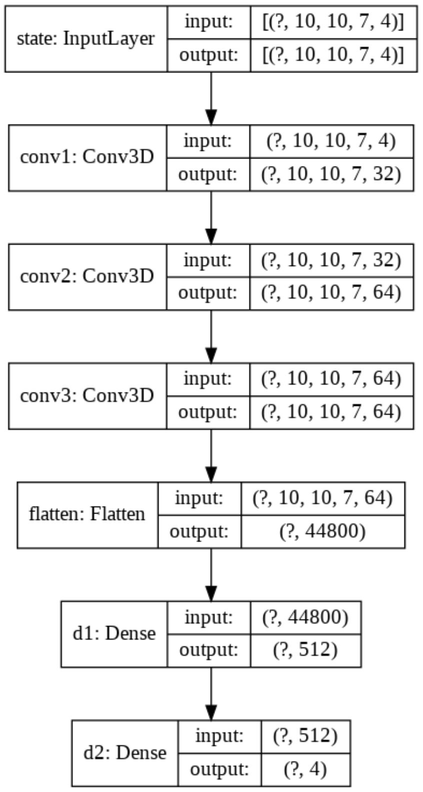


Figure 3: Action-Value Function Approximation Network: Three 3D convolutional layers followed by two fully-connected layers

During training, we used the Huber loss function (Equation 1) with $\delta = 1.0$ instead of the squared-error loss function. The Huber loss function is quadratic for small error $|y - f(x)|$, and linear for large error, with equal values and slopes at $|y - f(x)| = \delta$. Choosing the Huber loss function over the squared-error loss function is because the Huber loss function is less sensitive to outliers in data and thus produces smaller loss variance.

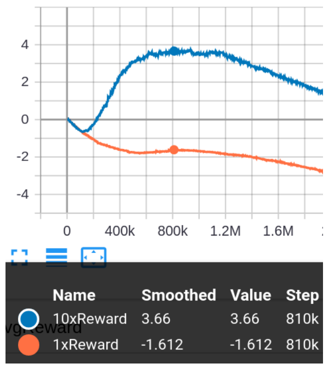
$$L_\delta(y, f(x)) \leftarrow \begin{cases} \frac{1}{2}\left[y - f(x)\right]^2 & \text{for } |y - f(x)| \leq \delta \\ \delta|y - f(x)| - \frac{1}{2}\delta^2 & \text{otherwise} \end{cases} \tag{1}$$

We also employed RMSprop optimizer for reducing variance of network updates. Our DQN is trained with one Nvidia Titan Xp GPU (12 GB VRAM).

3 Results

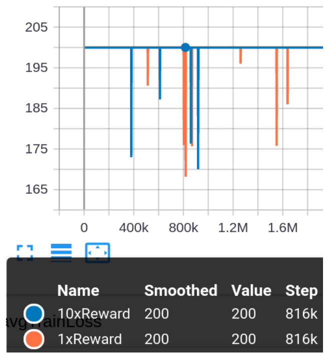
3.1 Training Results

The training results for original reward setup and $10\times$ reward setup are shown in Figure 4. The average Q values for $10\times$ reward setup are greater than those for original reward setup. It is possible that the $10\times$ reward setup is better than the original reward setup; however, this could also be a result of higher reward values. In the average step per episode plots, since the maximum number of allowed steps per episode is 200, any line drops below 200 means that our network successfully pushes all four boxes to the targets. From the Figure 4 (b), we can tell that around 800k training step our network have most frequent successes in completing the episodes. Thus, we restart training from 800k step for both setup and the average Q-value curves are shown in Figure 5.



Name	Smoothed	Value	Step
10xReward	3.66	3.66	810k
1xReward	-1.612	-1.612	810k

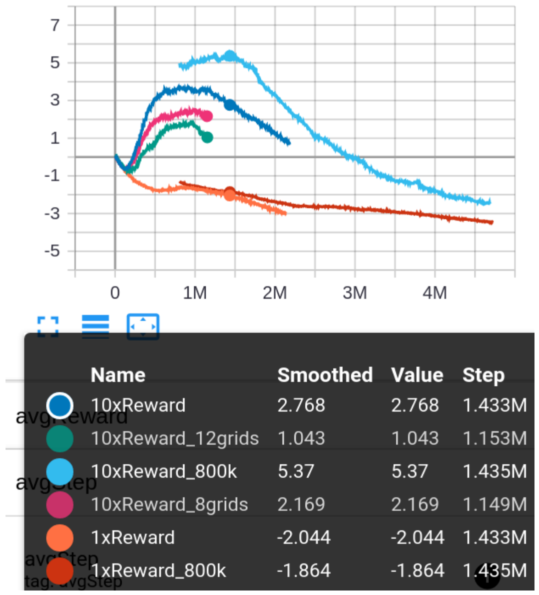
(a) Average Q-Values vs Training Steps



Name	Smoothed	Value	Step
10xReward	200	200	816k
1xReward	200	200	816k

(b) Average Step per Episode vs Training Steps

Figure 4: Training results for original and $10\times$ rewards



Name	Smoothed	Value	Step
10xReward	2.768	2.768	1.433M
10xReward_12grids	1.043	1.043	1.153M
10xReward_800k	5.37	5.37	1.435M
10xReward_8grids	2.169	2.169	1.149M
1xReward	-2.044	-2.044	1.433M
1xReward_800k	-1.864	-1.864	1.435M

Figure 5: Average Q-values for all training setups: restart training from 800k boosts the average Q values for $10\times$ rewards a bit higher but has no significane improvements on the $1\times$ reward setup.

3.2 Evaluation Results

The way we tested the results were by having a script run through 20 episodes at each checkpoint for both the 1x and 10x reward models. Checkpoints were captured at every 100,000 episodes of training for the purpose of diagnostics. The tests started at 800,000 where we first saw more significant results and more game completions. From there each of these large files would be loaded and ran as if at that current state in training.

We observed the average reward, the times pushed on a target, the times pushed off a target, the average step at completion, and the number of wins for the 20 episodes ran at each checkpoint.

The results were sent to an easily digestible csv file and graphed. Wins were rare as well as pushing on and off targets appeared close to random.

3.3 Game-Play Results

To generate gifs for checking out the overall performance, we first need several images for each step and store them into a folder. We have a folder generation processes before running the whole episode. Screenshots for each step will be store into a folder under 'image', called 'steps', and the overall gif for the episode is under the folder named 'gif'. The general process of Game-Play is to generate episodes by loading the trained network checkpoint file and record the whole episode in gif format.

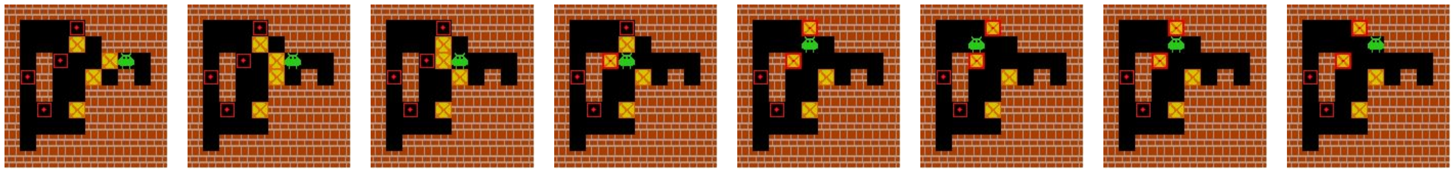


Figure 6: A sequence of game-play moves

The result showing in Figure 6 is one of the examples for demonstration. Character successfully pushed the first two boxes on to the destination. However, the network structure was only designed to store 4 past actions and resulted in a lack of knowledge of what the character has done before. The consequence of such limitation is that the character will move randomly and stuck at a single corner. This limitation can be alleviated by setting the network structure to have larger action register rather than size of 4. The trade-off would be a huge amount of extra time required for training processes and extra computing power.

Appendix A How To Run Our Program

The source code of our project can be found on GitHub. A zip file is also submitted via Canvas. The training logs are uploaded to our GitHub repo under `logs/tf_train`. Please use `TensorBoard` to visualize the training results. However, due to file size concerns, we didn't include the training checkpoints when submitting. We are not sure how to include the checkpoint files since each of them is around 200 MB.

To set up the runtime environment, a `Dockerfile` is provided. Also, Google Colab can be used to run our code.

References

[1] J. Culberson, "Sokoban is PSPACE-complete," *Department of Computing Science, University of Alberta*, 1997. [Online]. Available: https://webdocs.cs.ualberta.ca/~joe/Preprints/Sokoban/

[2] A. Junghanns and J. Schaeffer, "Sokoban: Enhancing general single-agent search methods using domain knowledge," *Artificial Intelligence*, vol. 129, no. 1-2, p. 219–251, 2001.

[3] V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra, and M. A. Riedmiller, "Playing atari with deep reinforcement learning," *CoRR*, vol. abs/1312.5602, 2013. [Online]. Available: http://arxiv.org/abs/1312.5602

[4] M.-P. B. Schrader, "gym-sokoban," https://github.com/mpSchrader/gym-sokoban, 2018.

[5] G. Brockman, V. Cheung, L. Pettersson, J. Schneider, J. Schulman, J. Tang, and W. Zaremba, "OpenAI Gym," 2016.

[6] V. Mnih, K. Kavukcuoglu, D. Silver, A. Rusu, J. Veness, M. Bellemare, A. Graves, M. Riedmiller, A. Fidjeland, G. Ostrovski, S. Petersen, C. Beattie, A. Sadik, I. Antonoglou, H. King, D. Kumaran, D. Wierstra, S. Legg, and D. Hassabis, "Human-level control through deep reinforcement learning," *Nature*, vol. 518, pp. 529–33, 02 2015.

[7] M. Sinton, "DQN_Atari," https://github.com/msinto93/DQN_Atari, 2018.

[8] L. ji Lin, "Reinforcement learning for robots using neural networks," 1992.