Reinforcement Learning - Final Report

Ron Darmon (ID. XXXXXXXXX), Roei Arpaly (ID. XXXXXXXXX)

Submitted as Final Assignment for the RL course, Reichman University, 2023

1 Introduction

Reinforcement learning is a machine learning technique that enables an agent to learn how to interact with its environment to maximize a reward. This approach has been successfully applied in numerous domains, such as robotics, gaming, finance, and healthcare, to name a few. It provides a powerful framework for enabling intelligent agents to make better decisions and optimize their performance in dynamic and uncertain environments.

In the Sokoban problem, the agent is analogous to a warehouse worker who must navigate and manipulate boxes in the environment to achieve a specific goal. The environment is the warehouse with its walls, obstacles, and the positions where the boxes should be moved to. The reward can be defined as the agent successfully moving all the boxes to their designated positions, or achieving some other objective, such as minimizing the number of moves required to complete the task.

Deep Q-Networks (DQN) and Double Deep Q-Networks (DDQN) are two advanced variants of Q-Learning that leverage deep neural networks to approximate the action-value function, which is a function that predicts the expected future reward for each possible action in a given state. Those algorithms are able to handle high-dimensional state spaces and can learn directly from raw sensory inputs, making it suitable for solving 2D games such as Sokoban.

The primary objective of this study is to solve the Sokoban problem, based on its pixel representation.

1.1 Related Works

Reinforcement Learning, An introduction, Second Edition, Richard S.Suttion, Andrew G. Barto, MIT press, cambridge, 2018.
Yang, Z., Preuss, M., Plaat, A. (2021). Potential-based Reward Shaping in Sokoban. arXiv preprint arXiv:2109.05022.

Yoshua Bengio, Jerome Louradour, Ronan Collobert, and Jason Weston. Curriculum learning. In International Conference on Machine Learning (ICML), pages 41–48, 2009.

2 Solution

2.1 General approach

The Sokoban problem is a classic challenge in the field of artificial intelligence that involves moving boxes to designated locations within a warehouse environment. There are several approaches to solving this problem, including the use of deep learning, convolutional neural networks (CNNs) and deep reinforcement learning algorithms such as DQN and DDQN.

1. DL: Deep learning is a technique that involves training a neural network with multiple layers to recognize patterns and make predictions. In the context of the Sokoban problem, deep learning can be used to train a model to predict the best move for a given state of the environment.

2. CNN: Convolutional neural networks (CNNs) are a type of deep learning model that is particularly well-suited to image recognition tasks. In the context of the Sokoban problem, a CNN can be trained to recognize the state of the environment and determine the best move to make based on that state. This approach potentially highly effective when the environment is represented as a 2D grid, as in the case of the Sokoban problem.

3. Deep RL: Deep reinforcement learning is a machine learning technique that combines deep learning with reinforcement learning. In the context of the Sokoban problem, deep reinforcement learning can be used to train an agent to learn how to navigate the environment and move the boxes to their designated locations. The agent receives rewards for successfully moving the boxes, and these rewards are used to update the agent's policy, or strategy, for selecting actions.

4. DQN: Deep Q-Networks are a type of deep reinforcement learning algorithm that uses a neural network to approximate the optimal action-value function. In the context of the Sokoban problem, DQN can be used to train an agent to learn the optimal policy for moving the boxes. The agent receives rewards for successfully moving the boxes, and these rewards are used to update the Q-values, which represent the expected future reward for each action in a given state. The agent then selects the action with the highest Q-value for each state, based on the current estimate of the action-value function.

5. DDQN: Dueling DQN is an improvement over DQN that addresses its tendency to overestimate Q-values. DDQN uses two separate networks to estimate the action-value function, with one network used to select the

action and the other used to evaluate the selected action. By decoupling the action selection and evaluation, DDQN is able to reduce overestimation of Q-values and achieve more stable and accurate performance.

We first tried to solve a fixed scenario of a Sokoban problem with a 7x7 grid, containing only one box and one target. The state was represented by an RGB array of size 128x128. After solving a specific scenario, we tried to train the model on different scenarios for it to be able to solve a new unseen scenarios. This means that the box and target positions could change from one scenario to another, increasing the complexity of the problem.

In addition to the deep learning and reinforcement learning approaches mentioned earlier, there are other methods to solve the Sokoban problem. These include naive random walk, depth-first search (DFS), breadth-first search (BFS), A* ("A-star") and others. These methods can be effective in finding solutions for smaller Sokoban problems but may struggle to scale up to larger, more complex scenarios.

2.2 Design

The presented work employs Python as the programming language and Google Colab as the platform for conducting the experiments.

2.2.1 Reinforcement Learning Techniques

In this study, reinforcement learning approaches were employed, including:

1. **Reward Shaping:** Manhattan distance was measured between the target and the box in our Sokoban problem in order to help the agent learn that moving the box towards target will increases the reward. Additionally, we incorporated the distance between the agent and the box minus one distance unit (to make sure we don't penalize the agent for being adjacent to the box). By doing so, we aimed to ensure that the agent is always in close proximity to the box, which leads to a more efficient and effective solution. both of the approaches not only enhances the performance of the agent but improving the convergence rate.

2. **Action Shaping:** To simplify the model's output space and reduce the complexity, we used only push and pull actions, which cover the functionalities of the remaining actions. This resulted in a total of eight actions being used (instead of the original 13).

3. **Curriculum Learning:** Curriculum learning is a powerful technique that involves breaking down challenging tasks into smaller, more achievable steps. We applied this approach to solve the Sokoban problem with dynamic environments by gradually increasing the game's complexity. Starting with simpler environments, such as those with a Manhattan distance

of one between the target and box, enabled the agent to learn the puzzle-solving process incrementally.

4. **Preprocessing:**

1. **Cropping:** To decrease the complexity of the input array, we tried implementing two techniques. Firstly, we have removed the irrelevant pixels by cropping the edges of the environment, resulting in a reduction of the RGB array size from 112x112x3 to 84x84x3.
2. **Image scaling:** We have experimented with different array sizes to further minimize the array dimensions, utilizing the "tiny RGB array" with dimensions of 5x5x3 after cropping the edges.
3. **Gray-scale:** Additionally, we have experimented with gray-scale, which has further reduced the array size to 5x5x1, approximately 1,600 times smaller than the original 112x112x3 space.

The algorithm's hyperparameters are adjustable parameters that play a crucial role in the performance of the DQN and DDQN models. The hyperparameters used in those algorithms include $MaxMemoryLength$, which is the maximum number of experience tuples stored in the memory replay buffer, $BatchSize$, the number of experience tuples used in each training iteration (which affect how quickly the agent can learn from past experience and generalize its knowledge to new situations), and $TargetNetworkTrainFrequency$, which is the frequency of updating the target network weights with the main network. Additionally, the hyperparameters including the $InitialLearningRate$ for the optimizer, $LearningRateDecay$ which is the decay of the learning rate over time, and $LearningRateDecaySteps$ that determines the frequency of the decay.

Other hyperparameters include $Clipnorm$, which is the maximum norm of the gradients used in gradient clipping during training (which helps prevent gradient explosions), $InitialEpsilon$, which is the initial value of the epsilon parameter used in the epsilon-greedy exploration strategy, $EpsilonDecay$, the rate at which the epsilon parameter decays over time, and $EpsilonDecayThreshold$, the threshold value below which the epsilon parameter is not decayed any further.

These hyperparameters are critical to the performance of the algorithms as they control the learning rate, exploration-exploitation trade-off, training batch size, and model architecture. These parameters determines how quickly the agent learns and how it explores the environment. Therefore, adjusting it had a significant impact on the result of the models and a thoughtful choice of parameters led to better learning and faster convergence of the model.

2.2.2 Challenges

During the experimentation process, we encountered various technical challenges within the environment. To address these challenges, we developed specific

solutions:

1. **Environment reset:** One challenge involved a problem with the seed when resetting the environment. To overcome this issue, we created an environment wrapper with a user defined seed (later on we had to make sure each episode has a different random seed).

2. **Box position:** To obtain the position of the box within the environment, it is only possible immediately after the box has been moved. In order to address this limitation, we have implemented a solution whereby the most recent known location of the box is stored within the environment wrapper.

3. **Action Zero:** Another challenge arose when it was discovered that action 0 ("No Operation") was identical to step 9 ("Move Right"). As a result, we decided not to utilize this particular action.

2.2.3 Additional challenges:

1. **Workspace and workflow:** During the experimentation process, the notebook utilized was extensive and necessitated numerous imports for setup, it was hard to debug, track and log our experiments.

2. **Availability of resources:** The availability of the GPU and memory was intermittent and there were multiple session crashes due to timeout or inadequate memory. We addressed this issue through housekeeping using Garbage Collector and Clear Session (Keras backend management).

3. **Computational power:** The testing of various parameters proved to be challenging due to the given computational power, so we terminated the agent based on the loss and reward curve if convergence was deemed unachievable. We monitored the agent's progress by printing all relevant information and occasionally collecting snapshots of the training process to evaluate its progression in a video. (Note: video snapshots were collected at the quarter mark of the entire training process).

3 Experimental results

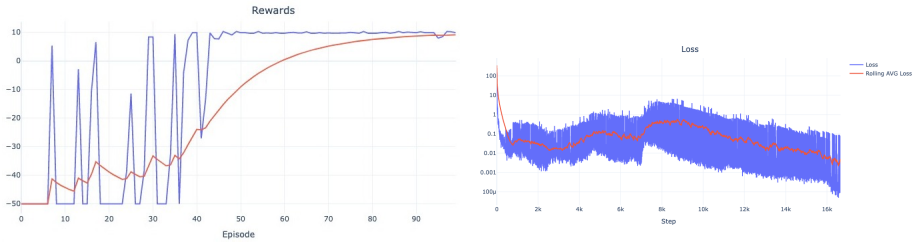
In this study, we conducted several experiments to investigate the impact of different hyperparameters on the performance of a DQN and DDQN models. For training the DQN model on a GPU, we observed that each episode of up to 60 steps took approximately 10 seconds, with additional time required for the DDQN architecture.

Our research has shown that in the case of reward shaping, it is more effective to terminate the environment after 60 steps as opposed to 120. This is due to the fact that the agent is able to learn more efficiently within this shorter

time frame. However, in the absence of reward shaping, we found that it is necessary to increase the number of steps to 500 as the agent can only learn through random completion of the environment. Furthermore, clipping the norm to 1 enhanced the stability of the model. Additionally, we observed that a batch size of 32 led to spikes in loss for certain problematic batch samples, while batch size of 64 was less prone to this issue.

We made adjustments to the epsilon value and conducted observations which revealed that it typically takes a certain number of steps to randomly complete the environment. Following this, it then takes a subsequent number of steps to achieve convergence. In response to this, we implemented an epsilon decay function that would gradually reduce the value to a threshold of 0.2 after roughly 3200 steps in the fixed environment. By doing so, we ensured that the agent would continue to explore new scenarios adjacent to the chosen path, even after achieving convergence.

In our study, we were able to achieve a successful solution without using the Manhattan reward shaping technique, although the model converged much slower.



(a) Rewards. (b) Loss.

Figure 1: Convergence Without Reward Shaping (max steps of 500)

After it, we attempted to analyze the performance of a cropped "tiny RGB array" 5x5x3 through the implementation of a simpler convolution network with less parameters. Our findings indicate successful convergence in this case. Conversely, we also explored the effectiveness of gray-scale 5x5x1, which unfortunately failed to converge. We also attempted to tackle the Sokoban problem through the utilization of three fully connected layers without convolution, but unfortunately, we encountered convergence issues and were unable to solve the Sokoban problem using this approach.

The final model architecture consisted of Conv2D layers of 32, followed by maxpooling of 2D, Conv2D layers of 64 and maxpooling of 2D, Conv2D layers of 128 and maxpooling of 2D, a flatten layer, a dense layer of 512, and output of action space size. Each convolution was performed using a kernel size of 5. The mode had a linear activation function and the optimizer chosen was Adam while

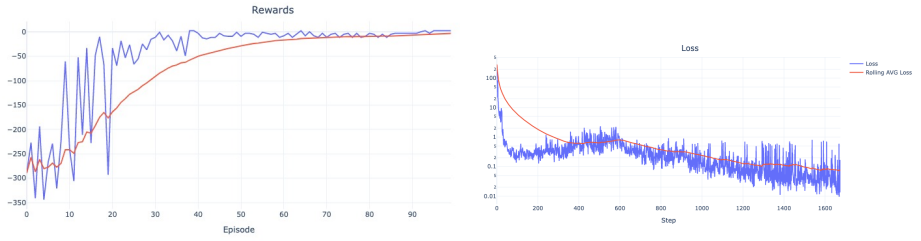
using mean squared error (MSE) as the loss.

```
Model: "sequential"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 conv2d (Conv2D)             (None, 84, 84, 32)        2432

 max_pooling2d (MaxPooling2D  (None, 42, 42, 32)       0
 )

 re_lu (ReLU)                (None, 42, 42, 32)        0

 conv2d_1 (Conv2D)           (None, 42, 42, 64)        51264

 max_pooling2d_1 (MaxPooling  (None, 21, 21, 64)       0
 2D)

 re_lu_1 (ReLU)              (None, 21, 21, 64)        0

 conv2d_2 (Conv2D)           (None, 21, 21, 128)       204928

 max_pooling2d_2 (MaxPooling  (None, 11, 11, 128)      0
 2D)

 re_lu_2 (ReLU)              (None, 11, 11, 128)       0

 flatten (Flatten)           (None, 15488)             0

 dense (Dense)               (None, 512)               7930368

 dense_1 (Dense)             (None, 8)                 4104

=================================================================
Total params: 8,193,096
Trainable params: 8,193,096
Non-trainable params: 0
_________________________________________________________________
```

Figure 2: Model Architecture

4 Discussion

After testing various networks, examining different parameters and validating the implementation of the algorithms DQN and DDQN, we managed to solve a single Sokoban environment by reaching to convergence after a few episodes.
Throughout the course of our experimentation, we endeavored to tackle changing Sokoban environments, which involved modifying the environment, in a multitude of configurations and with diverse parameter settings. Despite our extensive efforts, the agent did not converge, likely due to not optimal architecture or parameters setup.
Curriculum learning significantly enhanced the agent's learning ability, enabling



(a) Rewards. (b) Loss.

Figure 3: Final model convergence with reward-shaping (max steps of 60)

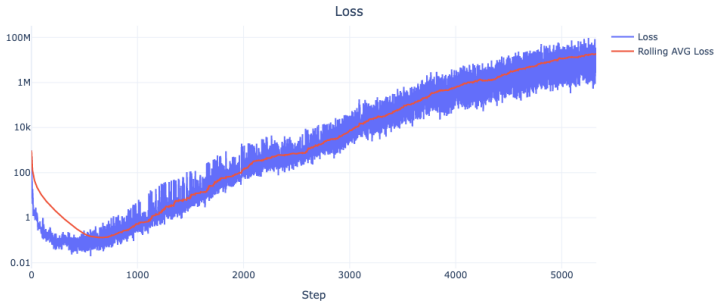


Figure 4: Loss steadily increasing dynamic environments

us to train it for more steps with lower loss and higher rewards. As illustrated in Figure 6, we progressively elevated the curriculum level from easy (starting with one distance step from the target) to more difficult ones every 100 episodes, resulting in a step-like function. It is also apparent that the agent commenced with a low reward and gradually elevated it towards the end of the curriculum, suggesting that it would require more episodes in each curriculum to solve it and reach a form of convergence and maximum rewards in each curriculum. It is worth noting that the rate of improvement diminishes from one curriculum to the next as the environments become increasingly challenging to solve in an exponential manner.

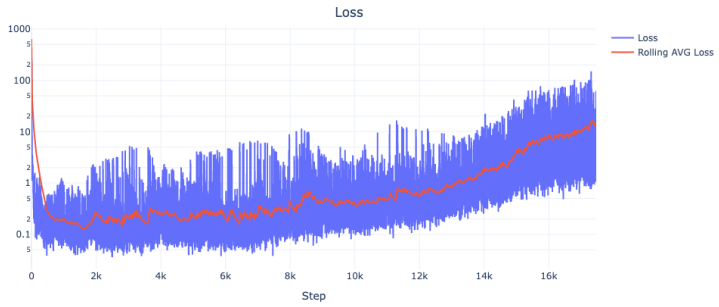


Figure 5: Loss using Curriculum learning in dynamic environment

After observing the exponential increase in difficulty to solve each curriculum, we decided to restart the agent with a simpler curriculum. This new curriculum consists exclusively of environments where the initial distance between the box and target is either 1 or 2. By implementing this strategy, we aimed to provide the agent with a more manageable learning environment and enable it to build

a stronger foundation before encountering more complex challenges. As can be seen in Figure 7, the loss is steadily decreasing.

Regrettably, even though we tried employing various reinforcement learning methodologies, such as Curriculum Learning, our agents were unable to achieve full proficiency in the designated environments. However, Curriculum Learning has enabled us solving about 20 percent of new unseen environments. This implies that additional computational power or advanced techniques that can accelerate the convergence rate might be necessary.

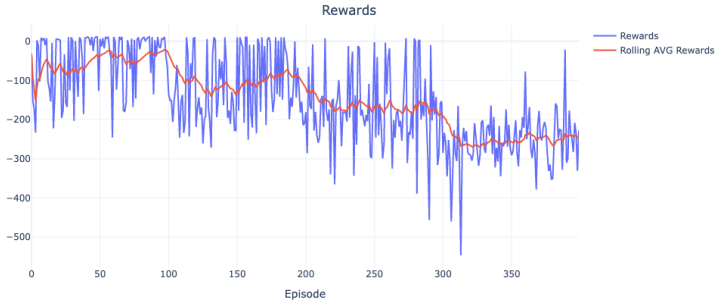


Figure 6: Rewards using Curriculum learning in dynamic environment

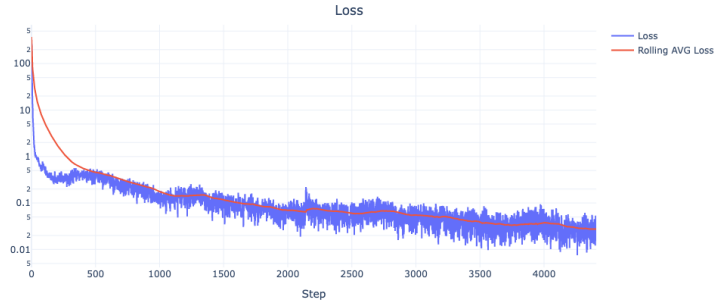


Figure 7: Easier Curriculum Learning Loss (initial distance of 1 and 2)

Our results suggest that Deep RL approaches such as DQN and DDQN are effective in solving fixed scenarios in the Sokoban problem. However, the non-fixed scenario presents a greater challenge, highlighting the need for further research and development in solving such problems with limited computational resources. The inability to solve the non-fixed scenario using DQN and DDQN

may suggest that the models are experiencing a significant level of forgetting of previously learned information, which could be due to catastrophic forgetting. As the non-fixed scenario presents new and varying challenges, the models must adapt and learn new strategies to solve the problem.

However, if the models are experiencing catastrophic forgetting, they may be overwriting previously learned information, making it difficult to adapt to new challenges while still maintaining previously learned strategies. This may lead to a failure to converge and solve the problem. Hence, more advanced RL algorithms or model architectures may be necessary for solving complex, non-fixed Sokoban problems.

5 Code

Experimentation Notebook:
`https://colab.research.google.com/drive/14KxJmJG5gEpMVCfMAzTSBM97JECi-sDM?usp=sharing`

Test Notebook:
`https://colab.research.google.com/drive/1JX5H869X0WHaNbvgxJJMlyO5QOv_YRMw?usp=sharing`