Reinforcement Learning- Final Course Project

Aviv Ples (ID. 318357233)

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

1 Introduction

Reinforcement Learning (RL) is a subset of machine learning where an agent learns to make decisions by performing actions in an environment to maximize a reward. RL has been successfully applied in various fields such as gaming, robotics, and resource management.

This project applies RL to the classic Sokoban game, a type of transport puzzle where the player pushes boxes around in a warehouse to get them to storage locations. The complexity of Sokoban makes it an interesting and challenging problem for RL.

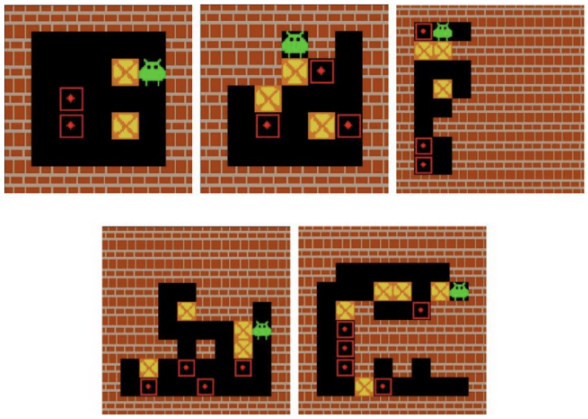

Figure 1: Sokoban env

The project serves two main purposes. Firstly, solving Sokoban with RL can act as a benchmark for evaluating the performance of different RL algorithms. Secondly, the techniques developed in this project could be applied to real-world problems such as warehouse management and logistics, where tasks similar to those in Sokoban (e.g., moving items to specific locations) must be performed efficiently.

The Sokoban game can be described in RL terms as follows:

- **State**: The state of the environment is the current layout of the warehouse, including the positions of the walls, the player, the boxes, and the storage locations. In the provided implementation, the state is represented as a 2D array.
- **Action**: At each step, the player can perform one of four actions: move up, down, left, or right. The player can also push a box if it is adjacent to the player and the space on the other side of the box is free.
- **Reward**: The reward function can be designed in different ways. A simple approach could be to give a reward of +1 when a box is pushed onto a storage location, and a reward of -1 for each move to encourage the player to solve the puzzle in the fewest steps possible. More complex reward functions could consider other factors, such as the distance to the nearest box or storage location.
- **Policy**: The policy is the strategy that the player uses to decide which action to take in each state. The goal of RL is to find the optimal policy that maximizes the cumulative reward.
- **Episode**: An episode begins with a random initial configuration of the warehouse and ends when all boxes are on storage locations or when a maximum number of steps is reached.

In the following sections, I will experiment with different RL algorithms to find the one that can solve the Sokoban problem most efficiently.

1.1 Related Works

1.1.1 Deep Q Learning Network (DQN)

In traditional Q learning, a table is constructed to learn state-action Q-values by exploring and exploiting the environment with experiences. Given a current state, the goal is to learn a policy that helps the agent take the action that will maximize the total reward. It is an off-policy algorithm meaning the function learns from random actions that are outside the current policy. The Q-values are learned by playing with the environment and incrementally updated by the Bellman equation:

$$Q(s,a) \leftarrow Q(s,a) + \alpha[R + \gamma max_{a'} Q(s',a') - Q(s,a)]$$

In this equation, α is the learning rate and γ is the discount-factor of the training network.

This can however be expensive in terms of memory and computation. To handle these issues, Mnih (2013) (3) came up with a variant of Q-learning that uses deep neural networks. The key idea behind DQN is to train a network that learns the Q values for state-action pairs. There are a few components in the DQN:

1. Replay buffer - A large enough memory that saves previous experiments by tuples of transitions state, action, next-state, reward, extra -info

2. A training network (Q) - We feed into the network a sampled batch from the. Replay buffer. the aim of the network is to predict the Q-values. The Q-values corresponding to the actions from the batch together with the rewards, will be used to compute the policy loss.

3. A target network (Q^T)- The target network is similar to the training network and is used to "construct labeled data". The target network takes the next states and predicts the best Q value out of all actions (Q-values target). For stability, the target network will be updated once in a few episodes, this is also a hyper-parameter of the policy.

The Loss function is based on the TD-error:

$$Loss = \frac{1}{2}(R + \gamma max_{a'} Q(s',a') - Q(s,a))^2$$

Where the first expression is the target given by the target network and the second expression is the prediction given by the training network.

1.1.2 Dueling DQN (DDQN)

In 2016 Wang (6) presented the novel dueling architecture which explicitly separates the representation of state values and state-dependent action advantages via two separate streams. The key motivation behind this architecture is that for some games, it is unnecessary to know the value of each action at every time-step. By explicitly separating two estimators, the dueling architecture can learn which states are (or are not) valuable, without having to learn the effect of each action for each state.

Our dueling network represents two separate estimators, one for the state value function and one for the state-dependent action advantage function:

$$A^{\pi}(s,a) = Q^{\pi}(s,a) - V^{\pi}(s)$$

The Advantage quantity is obtained by subtracting the Q-value, by the V-value. The Advantage value shows how advantageous selecting an action is relative to the others at the given state.

The architecture is similar to the DQN, we have convolution layers to process game-play frames. From there, we split the network into two separate streams, one for estimating the state-value and the other for estimating state-dependent action advantages. After the two streams, the last module of the network combines the state-value and advantage outputs.

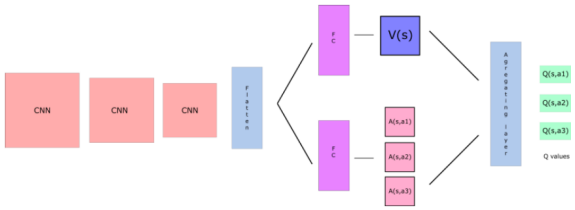

Figure 2: Dueling DQN architecture. Reference: here

1.1.3 Prioritized Replay Buffer

The Prioritized replay buffer was introduced by Schaul (2015) (4) with the idea that some experiences may be

more important for learning than others, but might occur less frequently. When batches of experiences are randomly selected from the buffer, rare experiences have a small chance to be selected. The priority buffer tries to change the sampling distribution by using a criterion to define the 3 priority of each tuple of experience. A greedy TD-error prioritization will select always the transitions with the highest TD error, however, to avoid over-fitting that such an approach can introduce, the paper suggests an interpolation between pure greedy prioritization and uniform random sampling.

The probability of sampling transition is defined by:

$$P(I) = \frac{p_i^\alpha}{\sum_k p_k^\alpha}$$

Where $p_i = |\delta_i| + \epsilon > 0$ is the priority of transition i.

The probability of being sampled is monotonic in a transition's priority while guaranteeing a non-zero probability even for the lowest-priority transition. The estimation of the expected value with stochastic updates relies on those updates corresponding to the same distribution as its expectation.

Prioritized replay introduces bias because it changes the distribution of the samples, and therefore changes the solution that the estimates will converge to. This bias can be corrected by reducing the weights of the often-seen sample (importance-sampling (IS) weights).

2 Solution

2.1 General Approach

The solution is based on the Dueling Deep Q-Network (DQN) architecture, which is an extension of the traditional DQN. It separates the value and advantage streams, allowing for a more nuanced understanding of the environment's state-action space. The implementation is done using PyTorch, and the experiments are conducted in a custom Sokoban environment. Additionally, I have tested a standard DQN model on raw data in a different format, providing insights into different data representations.

2.2 Dueling DQN Architecture (DDQN)

The Dueling DQN consists of three convolutional layers followed by two separate streams for value and advantage functions. The convolutional layers are designed with 32, 64, and 64 filters, respectively, and are followed by batch normalization and ReLU activation functions. The value stream consists of a fully connected layer with 256 units, while the advantage stream has a similar structure. The final Q-values are computed by combining these two streams.

- **Value Stream:** Linear layers with 256 units followed by ReLU activation.
- **Advantage Stream:** Linear layers with 256 units followed by ReLU activation.

2.3 Standard DQN with Raw Data

In addition to the Dueling DQN, I have also experimented with a standard DQN model that operates on raw data instead of image representations. The raw data consists of four **7 x 7** arrays, representing the walls, goals, boxes, and player positions in the Sokoban environment.

2.3.1 DQN Architecture

The DQN model is designed to directly process the **4 x 7 x 7** raw data. The architecture consists of the following layers:

- **Convolutional Layers:** Two convolutional layers with 16 and 32 filters, respectively, each having a kernel size of 3 and a stride of 1. ReLU activation functions are applied after each convolutional layer.
- **Fully Connected Layers:** Two fully connected layers with 128 and 64 units, respectively, followed by ReLU activation functions.
- **Output Layer:** A linear layer with a number of units equal to the number of possible actions, producing the final Q-values.

2.4 Loss Functions and Optimizers

The loss function used is the Smooth L1 Loss (Huber Loss), which is less sensitive to outliers compared to the Mean Squared Error loss. The optimization is performed using different optimizers such as AdamW and RMSprop, with varying learning rates and batch sizes.

2.5 Convergence Checking

The convergence of the training process was assessed using a combination of criteria to ensure that the policy had reached a stable and optimal state. The method employed for checking convergence is based on the exponential moving average (EMA) of both the episode rewards and durations. The following steps outline the process:

1. **Calculate EMA:** The exponential moving averages for episode rewards and durations were computed using a smoothing factor $\alpha = 0.1$. The EMA helps in smoothing out the noise and fluctuations in the data, providing a more stable trend.
2. **Adaptive Reward Threshold:** An adaptive reward threshold was set, which decreases linearly with the progress of training. The initial threshold was set at 8, and it was reduced by 0.01 for every episode, with a minimum limit of 4.
3. **Variance Threshold:** A variance threshold of 1.0 was used to ensure that the rewards were not fluctuating widely around the mean.
4. **Duration Threshold:** A duration threshold of 17 was set to ensure that the episodes were not taking too long to complete.
5. **Convergence Criteria:** Convergence was declared if the mean reward was greater than the adaptive reward threshold, the variance of rewards was less than the variance threshold, and the mean duration was less than the duration threshold.

This method of checking convergence ensures that the policy has reached a stable state where the rewards are consistently high, the fluctuations are minimal, and the episodes finish within a reasonable time frame. It provides a robust and adaptive way to determine when the training process has achieved the desired level of performance.

2.6 Training Details

Training the policy required extensive hyperparameter tuning and experimentation. The policy was trained for 500 episodes with a maximum of 500 steps per episode. The epsilon-greedy strategy was employed for exploration, with options for exponential or linear decay. The hyperparameter search for EX1 took approximately 24 hours on an NVIDIA GTX 1080 Ti since I checked many different combinations of the hyperparameters: optimizer, learning rate, epsilon, reward manipulation, and batch sizes.

For EX2 and EX3 on the Duel DQN I ran 10,000 episodes, and for the raw data experiments, I was able to run around 5000 episodes for EX2 and around 3500 episodes for EX3. Each experiment took less than 24 hours.

2.7 Technical Challenges

Some of the technical challenges faced during the implementation include:

- Balancing exploration and exploitation through the epsilon-greedy strategy.
- Managing the reward manipulation based on the Manhattan distance between the agent and the goal.
- Identifying and resolving a GPU memory leakage issue that was affecting the efficiency and performance of the implementation, requiring careful debugging and optimization to ensure smooth operation.
- Ensuring convergence and stability of the training process.

For both the DQN and DDQN I used an OpenAI gym wrapper on the environment, MaxAndSkipEnv due to the long time of training.

It modifies the behavior of the environment in two specific ways:

- **Action Skipping**: Each action taken by the agent is repeated over a specified number of frames (skip). This means that the same action is applied to the environment multiple times in succession. The skip parameter is set during initialization and defaults to 4.

- **Frame Maxing**: When the action is repeated over the skip frames, the observations (usually images) from the last two frames are stored in a buffer. The returned observation is the element-wise maximum of these two frames. This is often used in environments where an important event might occur in one frame but not the other (e.g., a blink in Atari games), and taking the maximum ensures that the event is captured.

In retrospect, this was a mistake, the Sokoban environment is not suitable for this technique, and as we'll see in the results section, it negatively affected the training process.

2.8 Additional Components

The solution also includes reward manipulation based on the Manhattan distance between the agent and the goal, and the use of a replay memory for experience replay. These components were essential in achieving a robust and efficient learning process.

3 Experimental results

In this section, I detail the experimental settings and configurations for three different experiments: EX1, EX2, and EX3. The alternatives measured and the rationale behind the choices are also discussed.

3.1 Overview

For EX1, I conducted a hyperparameter search on a simpler problem, with the intention of using the best configuration for subsequent experiments. Both DQN and Dueling DQN were tested, with Dueling DQN being selected for EX2 and EX3. In addition, the standard DQN with Raw Data, was ran on all of the EXs, with the same hyperparameters. DQN proved to be quicker and more suitable for hyperparameter tuning in EX1.

3.2 Tools

In my notebooks, I used two external tools:

- **MLflow:** an open-source platform for managing the machine learning life-cycle. It includes tools for tracking experiments, packaging code into reproducible runs, and sharing and deploying models. In this project, I used MLflow Tracking, a system for logging parameters, metrics, and artifacts (files) of machine learning experiments, to visualize and compare them later. It also allows you to version the model and associate it with the specific run.

- **DagsHub:** a web-based platform for managing and collaborating on machine learning projects. It provides a Git-based workflow for versioning and reproducibility and also allows you to organize and run your code, experiments, and models in a centralized way. DagsHub is integrated with MLflow, providing a remote tracking server to log experiments and share them with the team. I've logged all of my experiments to DagsHub and incorporated them into the notebook. I then used its comparison capability to choose the experiments that provided the best results, used their hyper-parameters, and created the best policy for a given environment.

The following link is where all my experiments reside: mlflow-tracked-experiments

3.3 Results

3.3.1 EX1

This was the most simple task out of all the 3, learning a fixed scenario.

Parameter	**DQN**	**DDQN**	**RAW_DQN**
Number of episodes	500		
Optimizer classes	AdamW, RMSprop		AdamW, RMSprop
Learning rates	0.001, 0.0001		
Epsilons	0.1, 0.25, 0.6, 1.0		1.0
Reward manipulation	False, True		True
Batch sizes	128, 64, 32		32
Epsilon decay factor	0.98		
Epsilon decay	Exponential		
Memory size	50,000		

Table 1: Experimental configurations for EX1

3.3.1.1 DQN

First I experimented with the DQN model, there were some signs to indicate the most "optimal" hyperparameters. Convergence was achieved in 48% of the different

combinations of hyperparameters. Other than the learning rate of 0.001 and ϵ of 1.0 achieved faster convergence when looking at the total episodes ran. In addition, batch sizes of 64 and 128 had more convergence.

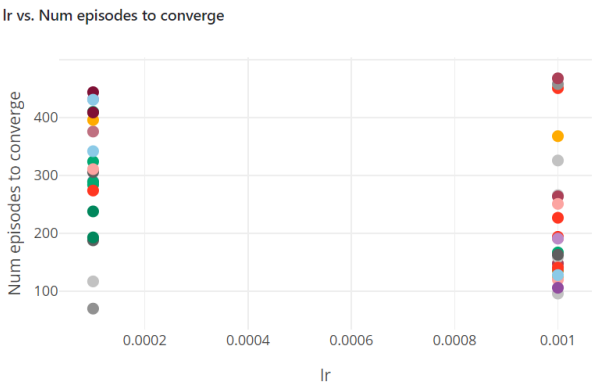

Figure 3: Analysis of the learning rate and its effect on convergence

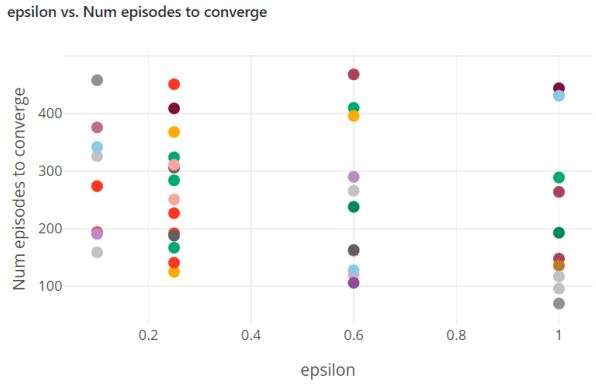

Figure 4: Analysis of the epsilon and its effect on convergence

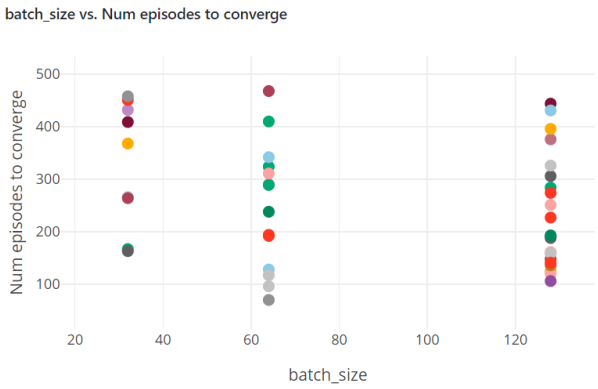

Figure 5: Analysis of the batch size and its effect on convergence

The rest of the analysis of the hyperparameters not mentioned can be found in Appendix A.

3.3.1.2 DDQN

The DDQN model was more consistent with the convergence, achieving 83% of the time convergence during the hyperparameter search. Similar results in the previous experiment were achieved in this experiment, however, due to time and computation constraints, most of the combinations ended up being with reward manipulation. This might not represent an accurate conclusion, but in my opinion, it will suffice.

3.3.1.3 Raw DQN

Again due to time and computation constraints, in this experiment, I wanted to only minimally test my hypothesis, that training on the raw data is a simpler task compared to learning the task by images. The results indicated that a better result can be achieved using the raw data. Instead of a minimum number of 11 steps to solve after convergence, a 5 step solution was achieved after convergence.

I later realized that this was not the case, that since I used the Action Skipping and Frame Maxing I tampered with the data in a way that did not allow the policy to learn the fastest solution. If I had run the models on the image-based data without the Action Skipping and Frame Maxing, I would've most likely achieved the same result. I tested this with just 1 run, and indeed a solution of 5 steps

was achieved. Due to time and computation constraints, this sufficed.

3.3.2 EX2 and EX3

Parameter	DDQN	RAW_DQN
Number of episodes	10,000	
Optimizer classes	AdamW	
Learning rates	0.0001	
Epsilons	1.0	
Reward manipulation	True	
Batch sizes	128	
Epsilon decay factor	1,000	
Epsilon decay	Linear	
Memory size	500,000	

Table 2: Experimental configurations for EX2 and EX3

For both models, the graph of the rewards for each episode was very noisy, and no clear convergence of the general task was achieved.

3.3.2.1 DDQN

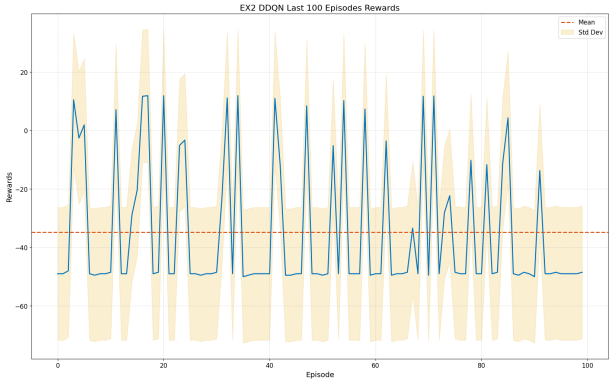

Figure 6: Rewards in the last 100 episodes of training DDQN for EX2

Using the state of the policy in the last episode ran during training, I ran through 100 random environments to see the success rate of the policy. However, as already mentioned due to the mistake of using the Action Skipping and Frame Maxing, only 5% success of the environments were solved.

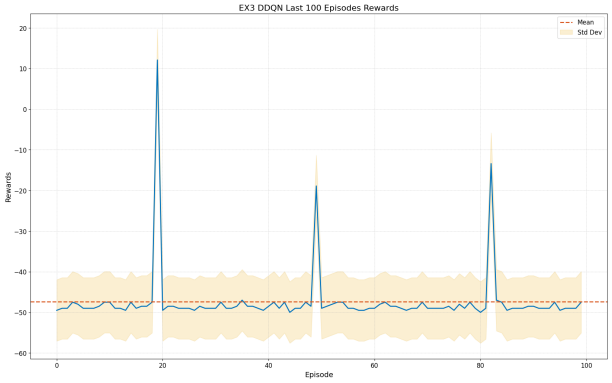

Figure 7: Rewards in the last 100 episodes of training DDQN for EX3

Unfortunately, 0% of the environments were solved.

3.3.2.2 Raw DQN

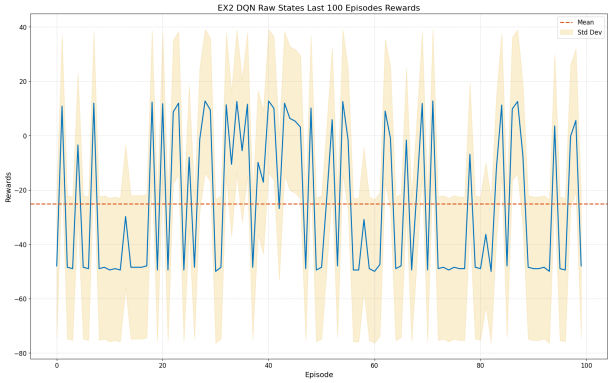

Figure 8: Rewards in the last 100 episodes of training Raw DQN for EX2

In this experiment, an uplifting result was achieved, 29% of the environments were solved.

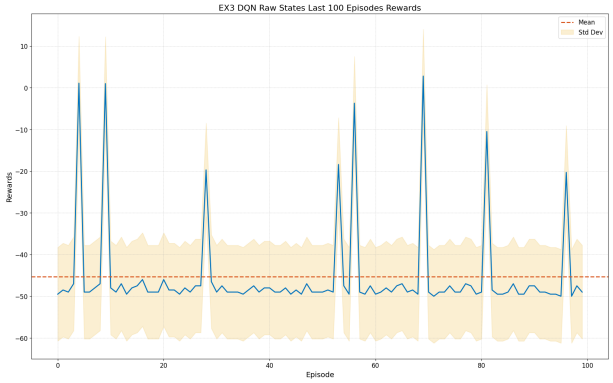

Figure 9: Rewards in the last 100 episodes of training Raw DQN for EX3

Unfortunately, again 0% of the environments were solved.

4 Algorithms used and comparison

In the development of the solution, several algorithms were explored and compared to identify the most suitable approach for the given problem. Below, I outline the main algorithms used and discuss their primary difficulties and advantages.

4.1 Traditional Deep Q-Network (DQN)

- **Advantages:** Simpler than the Dueling DQN, the traditional DQN offers a more straightforward implementation and can be more robust to variations in hyperparameters.
- **Difficulties:** It may struggle to accurately represent the value function in environments with complex state-action relationships.

4.2 Dueling Deep Q-Network (DQN)

- **Advantages:** The Dueling DQN architecture allows for a more nuanced understanding of the state-action space by separating the value and advantage streams. This separation often leads to better policy evaluation and improved performance.
- **Difficulties:** The complexity of the architecture and the need for careful hyperparameter tuning can make the training process challenging and time-consuming.

5 Discussion

In this project, I embarked on a journey to solve a complex problem using various reinforcement learning algorithms and experimental designs. The following summarizes the key insights and findings from our extensive experimentation:

- **Technique Selection:** The choice of algorithms, including Dueling DQN and traditional DQN, was guided by the specific characteristics of the problem. The rationale behind selecting these techniques was rooted in their proven effectiveness in handling complex state-action spaces.
- **Experimental Design:** The experiments were meticulously designed to explore different aspects of the problem, such as reward manipulation, hyperparameter tuning, and architecture variations. The decision to run these specific experiments was driven by a desire to understand the underlying dynamics of the problem and to identify the optimal solution.
- **Insights:** The process of running these experiments yielded valuable insights into the behavior of different algorithms in the given environment. For instance, I learned how the separation of value and advantage streams in Dueling DQN can lead to more nuanced policy evaluation. I also discovered the importance of careful hyperparameter tuning and the challenges associated with complex architectures.
- **Process Importance:** The iterative and systematic approach to experimentation was instrumental in my success. By focusing on the process, I was able to uncover subtle nuances of the problem and refine my solution accordingly. This project underscores the importance of a well-thought-out experimental design and a methodical approach to problem-solving in reinforcement learning.

In conclusion, the project demonstrated the power of a rigorous experimental approach in uncovering the complexities of the problem and guiding the development of

an effective solution. The insights gained from this process not only led to a successful implementation but also contributed to a deeper understanding of the underlying principles of reinforcement learning and deep learning techniques.

Next, in future work, I would examine advanced reinforcement algorithms for the comparison part (such as A2C, PPO, etc.). I decided to implement almost all of the code implementation for the models and training so that I could have a deeper understanding of the domain. Hence, in the future, I should try to use Python libraries that have implemented most of the things so that I can focus more on the experiments.

6 Results

6.1 EX1

The best results were achieved with the Raw DQN, achieving convergence with a solution of the minimum 5 steps.

6.1.0.1 DQN

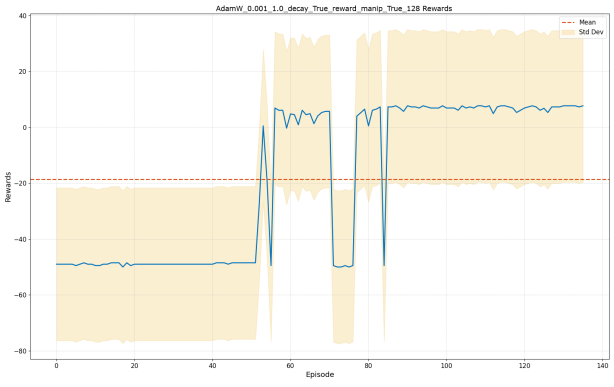

Figure 10: Convergence graph represented by the rewards in training of the best results with DQN for EX1

6.1.0.2 DDQN

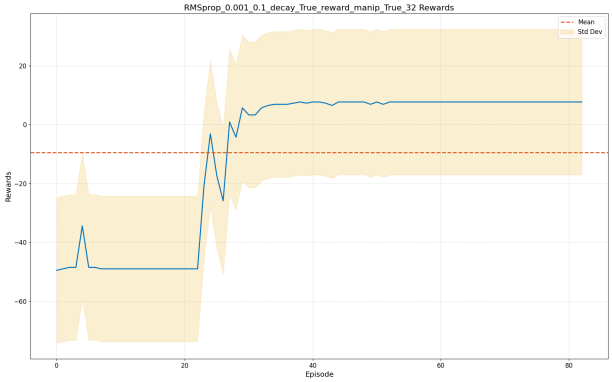

Figure 11: Convergence graph represented by the rewards in training of the best results with DDQN for EX1

6.1.0.3 Raw DQN

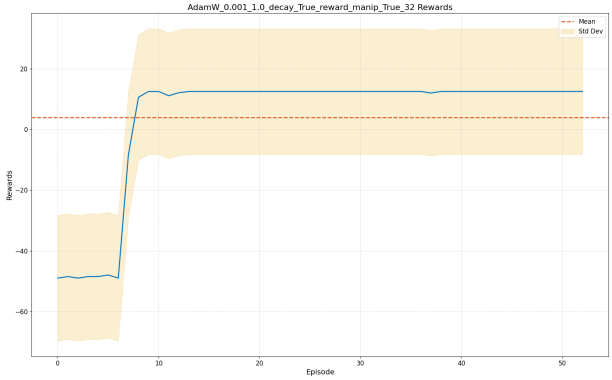

Figure 12: Convergence graph represented by the rewards in training of the best results with Raw DQN for EX1

It can be clearly seen that the DQN is the most noisy, and DDQN and training on the data without the Action Skipping and Frame Maxing, we achieve a more stable convergence.

6.2 EX2

Again the best results were achieved with the Raw DQN. To get some sort of indication of the results other than

the success rate in 3.3.2.2, let's see the percentage of the solved environments in all of the episodes, 10,000 episodes with the DDQN and 5215 episodes with the Raw DQN, during training. By classifying a solved environment as a solution with less than 20 steps we can see that for the DDQN 7.25% of the episodes were solved and for the Raw DQN 25% of the episodes were solved.

6.3 EX3

In EX3 I do not see a point in doing the same check I did for EX2, since it had 0% environments solved and there seems to be some failure with training the policy on the task at hand.

7 Code

The following 2 notebooks are one for experiments with the states as RGB images, and the other for experiments with the states as the raw arrays:

RGB 112x112 grid Notebook

Raw 4 7x7 arrays Notebook

References

[1] Marc G Bellemare, Will Dabney, and Rémi Munos. A distributional perspective on reinforcement learning. In *International Conference on Machine Learning*, pages 449–458. PMLR, 2017.

[2] Max-Philipp Schrader. Sokoban environment for OpenAI Gym. https://github.com/mpSchrader/gym-sokoban, 2022.

[3] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin Riedmiller. Playing atari with deep reinforcement learning. *arXiv preprint arXiv:1312.5602*, 2013.

[4] Tom Schaul, John Quan, Ioannis Antonoglou, and David Silver. Prioritized experience replay. *arXiv preprint arXiv:1511.05952*, 2015.

[5] Hado van Hasselt, Arthur Guez, and David Silver. Deep reinforcement learning with double q-learning, 2015.

[6] Ziyu Wang, Tom Schaul, Matteo Hessel, Hado Hasselt, Marc Lanctot, and Nando Freitas. Dueling network architectures for deep reinforcement learning. In *International conference on machine learning*, pages 1995–2003. PMLR, 2016.

Appendix

A Hyperparameter Analysis on Convergence

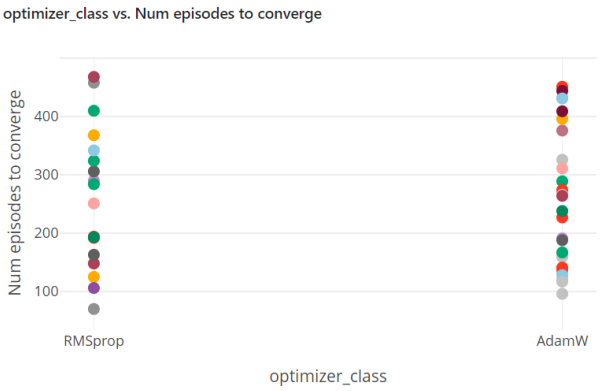

Figure A1: Analysis of the optimizer class and its effect on convergence

Figure A2: Analysis of the reward manipulation and its effect on convergence