Solve Sokoban with Deep Reinforcement Learning

Achia Rosin

August, 2023

1 Introduction

The purpose of this project is to showcase how the Sokoban game, a difficult transportation puzzle where boxes are moved to storage locations, can be solved using Reinforcement Learning approach. We chose to solve using Deep Q Learning. The report introduces Deep Q Learning and the Sokoban game, and explains how the game was designed and formulated as a problem. The report also discusses the reward function used in solving the requested exercises with the Deep Q Learning algorithm. The results of the experiments are then presented and analyzed, demonstrating the algorithm's success in solving the Sokoban game for a few tasks.

2 Solution

2.1 General approach

We focus on Deep Q Learning to solve the Sokoban transportation puzzle:

deep Q-network algorithm:
deep Q-network (DQN) is a type of artificial neural network that learns to approximate the optimal Q-value function in a reinforcement learning problem. The Q-value function maps states to action values, indicating the expected reward that an agent can achieve by taking a particular action in a given state. In DQN, the network receives a state as input and outputs the Q-values for all possible actions. The network is trained using a combination of supervised and reinforcement learning methods, where the loss function is defined as the difference between the predicted Q-values and the actual rewards obtained from the environment.

The DQN algorithm involves iteratively updating the network parameters using batches of experiences sampled from the replay buffer. The update rule is based on the Bellman equation:

$$Q(s,a) = E[R + \gamma \max_{a'} Q(s',a') \mid s,a] \tag{1}$$

where Q(s, a) is the Q-value of taking action a in state s, R is the immediate reward obtained from the environment, s' is the next state, a' is the next action, is the discount factor that determines the importance of future rewards, and max Q(s', a') is the maximum Q-value over all possible actions in the next state s'. The expectation is taken over all possible next states s' and actions a'.

During training, the DQN updates its Q-value estimates by minimizing the mean squared error loss between the predicted Q-values and the target Q-values, which are calculated using the Bellman equation. Specifically, the loss function is defined as:

$$L(\theta) = E\left[(r + \gamma \max_{a'} Q'(s', a'; \theta') - Q(s, a; \theta))^2\right] \tag{2}$$

where and ' are the parameters of the DQN and the target network, respectively, r is the immediate reward obtained from the environment, and Q' is the target Q-value function that is used to compute the target Q-values. The target Q-values are updated periodically by copying the parameters of the DQN to the target network, which helps to stabilize the training process and prevent the network from overfitting to the current Q-values.

Replay Buffer:
The replay buffer is a fixed-size queue used to store past experiences of the agent. The buffer stores the agent's state, action, reward, and next state tuples, collected during the training process. The buffer is used to sample batches of experiences randomly during training, enabling the agent to learn from a diverse set of experiences and reducing the correlation between consecutive experiences.

Epsilon-Greedy Strategy:
The epsilon-greedy exploration strategy is a method used to balance exploration and exploitation during the training process. Exploration is necessary to discover new strategies and improve the agent's performance, while exploitation involves using the current policy to maximize the expected reward. The epsilon-greedy strategy selects an action randomly with a probability of epsilon, and selects the action with the highest Q-value with a probability of 1-epsilon. The value of epsilon is gradually decreased over time, encouraging the agent to explore less as it becomes more confident in its policy.

2.2 Design

The design consist of 3 main parts:

- **QNN** - the neural network predicts Q-values of state-action pairs.
- **Replay Memory** - the replay buffer which stores the experience.
- **DQN Algorithm** - the implementation of the algorithm as described.

Deep Q Neural-Network:
The neural network is trained to predict the Q-value of an action in a particular state. The underlying principle of the Deep Q Learning algorithm is the Bellman equation, which states that the optimal Q-value is the highest expected reward from all potential actions in a given state. The CNN is optimized by computing the mean squared error between the predicted and actual Q-values obtained from the Bellman equation. We used the following CNN architecture to extract features from RGB images.

Replay Memory:
Implemented as a First-In-First-Out (FIFO) queue to store past experiences. the method push adds a new experience to the end of the queue, and sample retrieves a random batch of experiences from the queue. This allows the agent to learn from a diverse set of experiences and improve its performance over time.

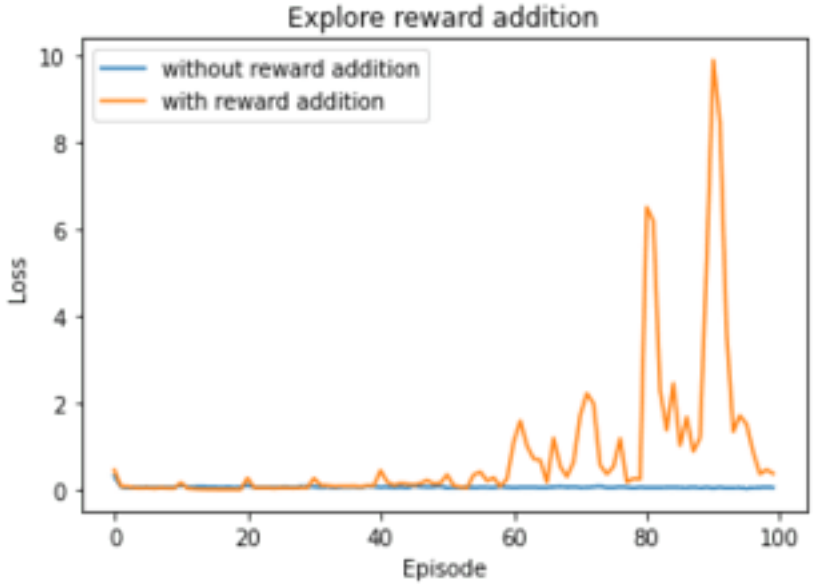
DQN Algorithm:
The DQN class has several methods, each of which performs a specific task in the DQN algorithm.
init - initializes the DQN object with the environment, hyperparameters, Q and target neural networks, a replay buffer, and an optimizer.
update target - updates the target network with the parameters of the Q network.
get state tensor - converts the state of the environment into a tensor that can be input to the neural network.
epsilon greedy action - selects an action based on an epsilon-greedy policy, which balances exploration and exploitation during training.
calculate reward - calculates the reward for a given state, based on the distance between boxes and targets.
update model - updates the Q network using a minibatch of experiences sampled from the replay buffer.
train - trains the Q network for a specified number of episodes, while **test** tests the trained model on a given environment and saves a video of the gameplay.

3 Experimental Results

3.1 Explore Parameters

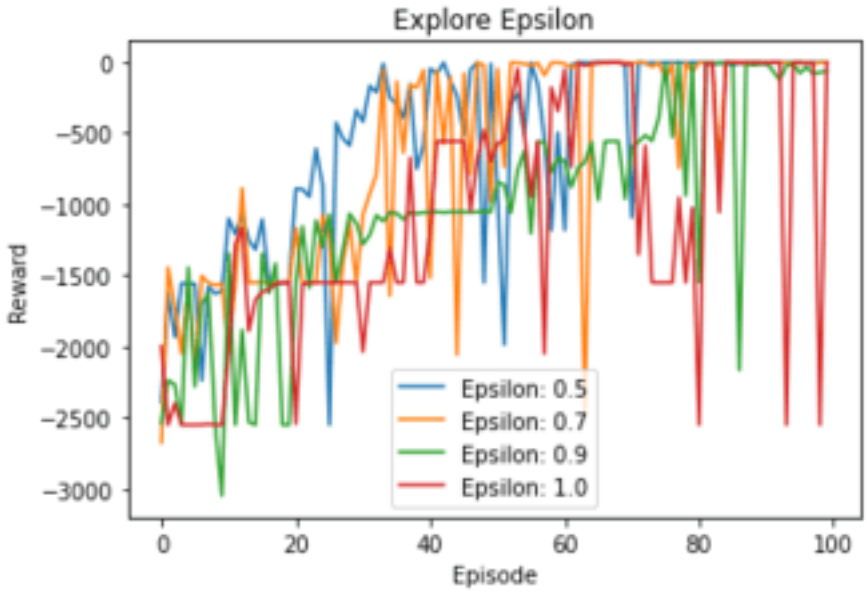
Since the fix scenario is the easiest, we will use it to explore the best parameters for the algorithm:

- **Reward**: First, we'll show results with and without our addition to reward. As mentioned above we added a function "calculate reward", that adds a reward in shape of the distance between targets and boxes - the closer a box to a target - the bigger the reward. Here is a graph of the loss with and without reward addition:



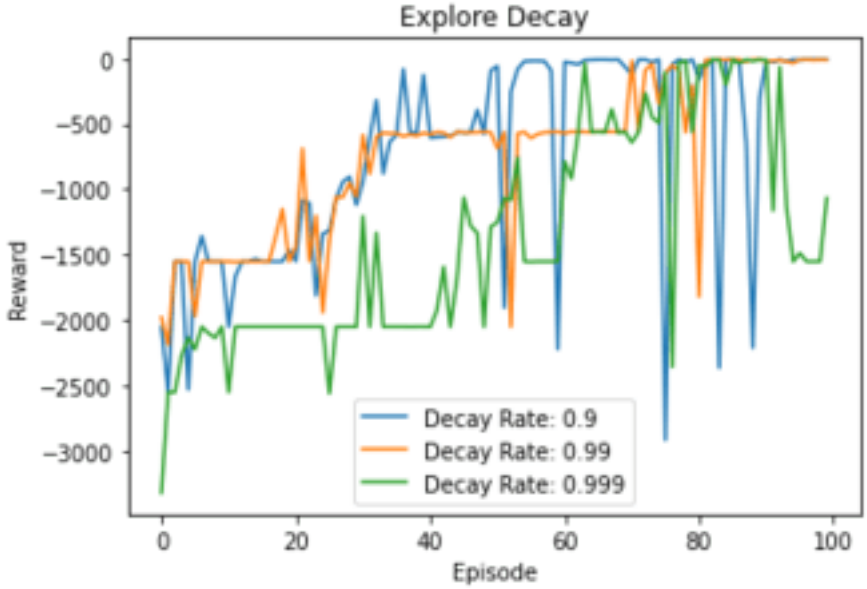
  The loss without the reward addition remains constant - thee is no learning process, whereas with the reward addition, since the target (the supervisor) is updated once in 10 episodes - the loss is growing every time the target been changed, and then decreasing for the next 10 episodes, and there is a learning process going on.

- **Epsilon**: We'll try a few values of epsilon to get the one which brings to better results:



We can see clearly that the bigger the epsilon it explore more, so the reward has "leaks" even towards the end of the learning.

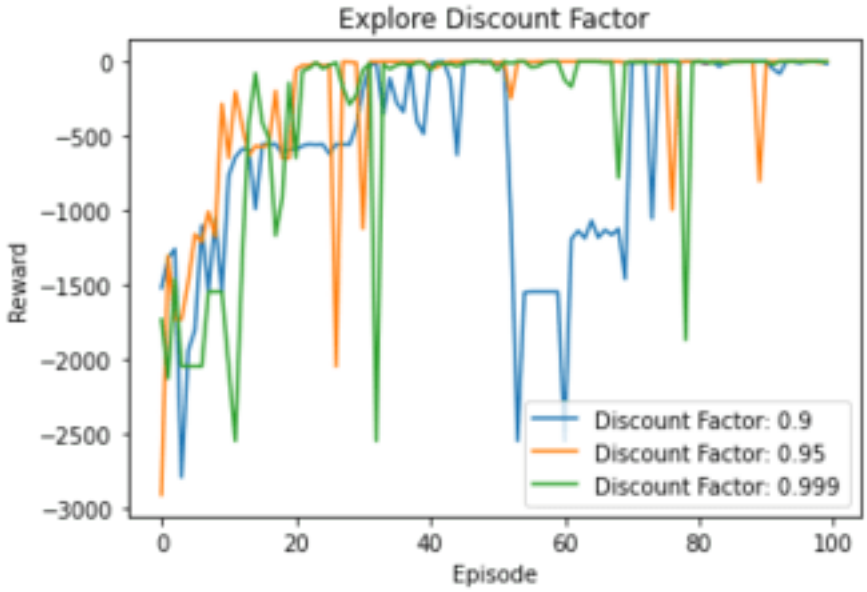
- **Decay Rate**: We'll try different rates of decay for the epsilon greedy strategy:



It seem that 0.99 is the best option of them, even though it might be just arbitrary, but maybe if the exploration isn't enough, the algorithm won't

get the best option but will stuck in a local minima.

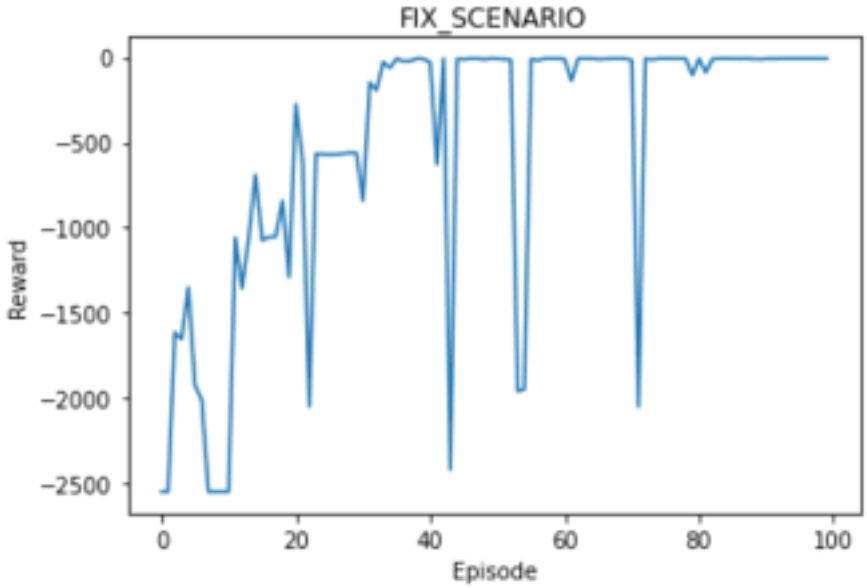
- **Discount Factor**: We'll next look at discount factor, and try a few values to see what works best:



We see that 0.9 gives worse results than the other 2. we'll continue with the best parameters we found to solve the requested exercises.

3.2 Fix Scenario - one box

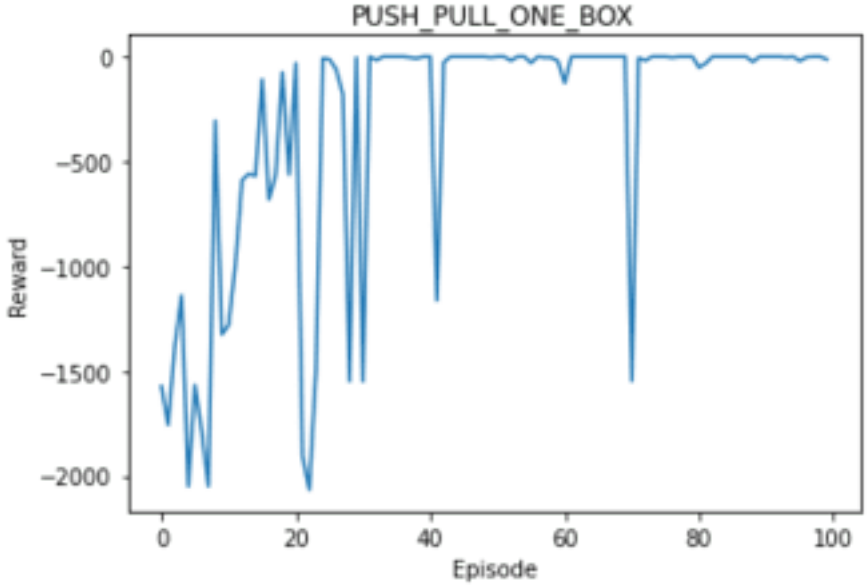
One box, One Target, Grid size: 112 X 112 (RGB), i.e. Every episode you get the same configuration:



Using the parameters we found best, solved Ex1 - the fix scenario. We did not solve it using the original reward merely though.

3.3 Random scenario - One Box

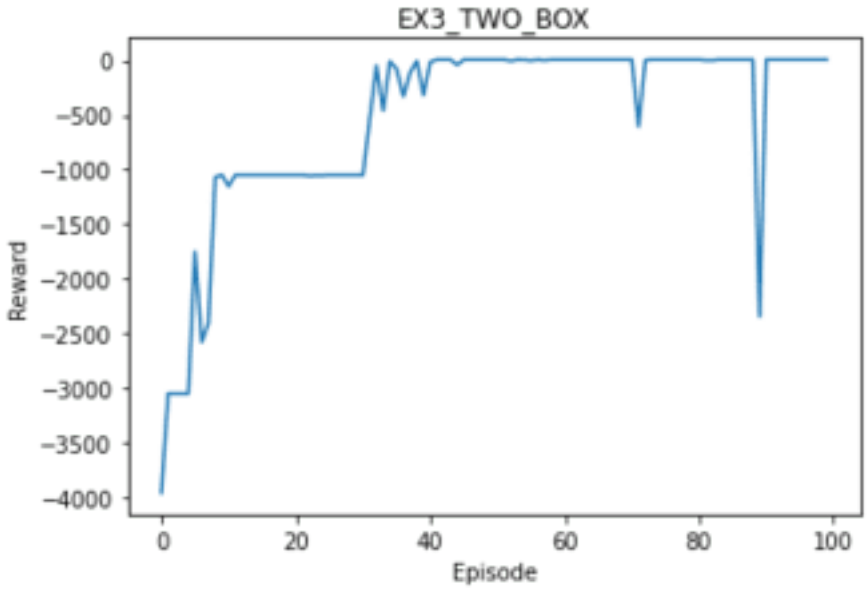
One box, One Target, Grid size: 112 X 112 (RGB), Random generated scenario:



Again, we see that the DQN algorithm is good enough to solve the random scenario.

3.4 Random scenario - Two Boxes

To check the strength of the algorithm we wrote we tried it on a random generated scenario with 2 boxes:



And it did manage to solve it. It even worked well, which shows that the DQN algorithm we wrote is a good one.

4 Discussion

The main focus of this project was to apply Deep Q Learning algorithm to solve the Sokoban game. Through our experiments and analysis, we have demonstrated the success of the algorithm in solving the game for the given exercises. The algorithm combines the advantages of both supervised and reinforcement learning methods, enabling the agent to learn from a diverse set of experiences and improving its performance over time.

One of the challenges we faced during the implementation of the algorithm was the design of the reward function. We experimented with different reward functions, including sparse and dense rewards, and found that a combination of both was most effective. By providing a sparse reward for reaching the goal state and a dense reward for moving boxes towards their goal locations, we were able to guide the agent towards the optimal solution.

Another challenge we faced was the selection of hyperparameters. Hyperparameters such as the learning rate, discount factor, and exploration rate can have a significant impact on the performance of the algorithm. We experimented with different values for these hyperparameters and found that the most effective ones for solving the Sokoban game.

In conclusion, our experiments have demonstrated the success of the Deep Q Learning algorithm in solving the Sokoban game.

5 Code

google colab notebook with the code