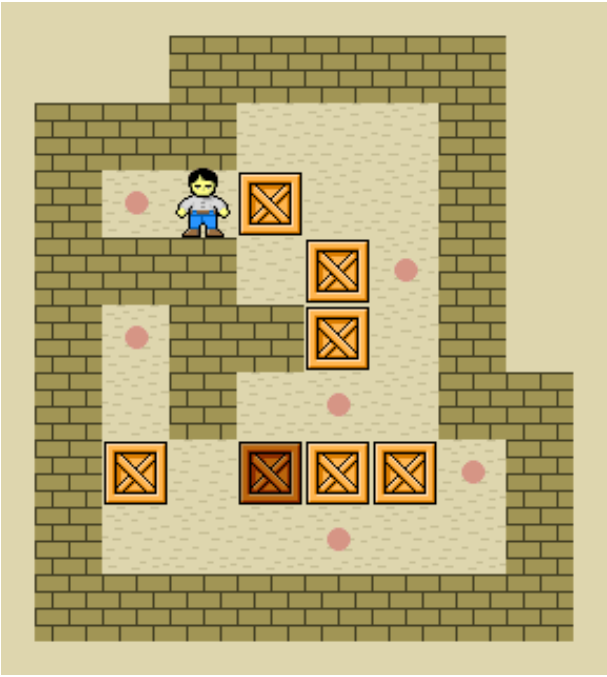
Sokoban

CS271 Final Report

Qi Lin	Ziying Yu	Wenbo Li
63447178	27083755	13642959

Dec 10, 2021



Github repo: `https://github.com/annyzy/Sokoban`

1 Problem Statement

1.1 Sokoban Overview

The game is played on a board of squares, where each square is a floor or a wall. Some floor squares contain boxes, and some floor squares are marked as storage locations.

The player is confined to the board and may move horizontally or vertically onto empty squares (never through walls or boxes). The player can move a box by walking up to it and pushing it to the square beyond. Boxes cannot be pulled or pushed to walls or other boxes. In our game, assume the number of boxes equals the number of storage locations. And there is no order assigned between the boxes and storage.

The goal is to move all boxes into all storage blocks.

1.2 Input/Output Data

According to the project requirements, the input data form is given as map size H and V, list of walls, list of initial boxes locations, list of storage locations, and the initial location of the player.

Output is a single line, beginning with nMoves followed by a sequence of letters (U, D, L, R) indicating the direction of the move, e.g., ”1 D”.

2 Solving Approach

We choose Deep Q learning as the reinforcement learning algorithm to solve the game. Considering tasks in which a player interacts with the Sokoban emulator in a sequence of actions, observations, and rewards, the player's goal is to interact with the emulator by selecting actions that maximize future rewards. The following class diagram shows the static structures of our solving approach:

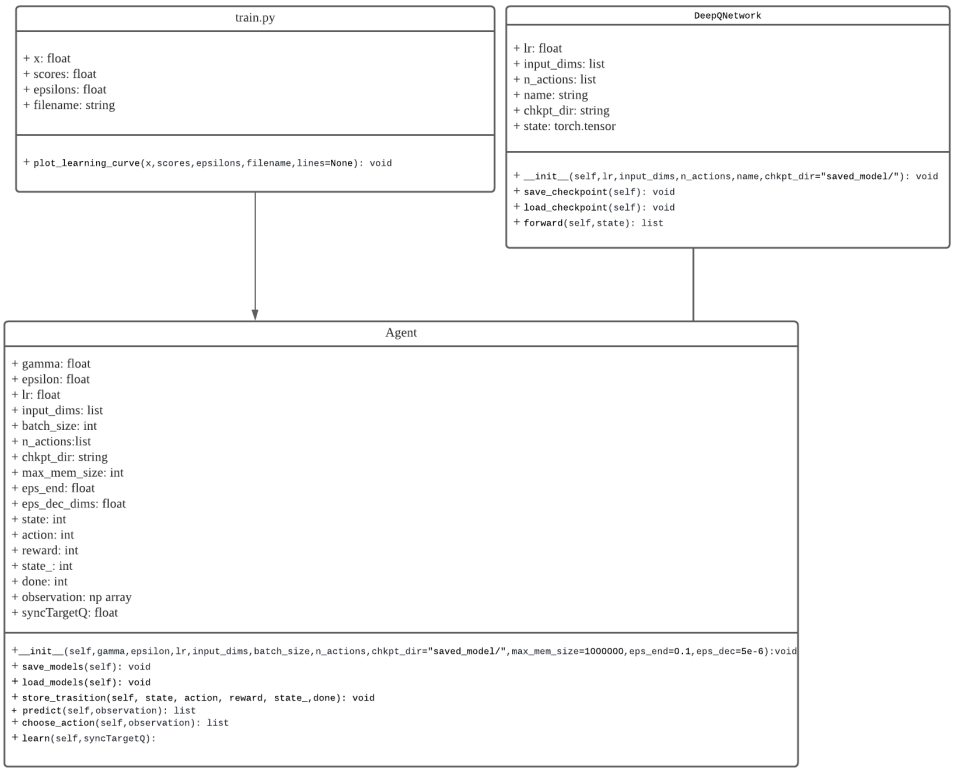


Figure 1: DQN Class Diagram

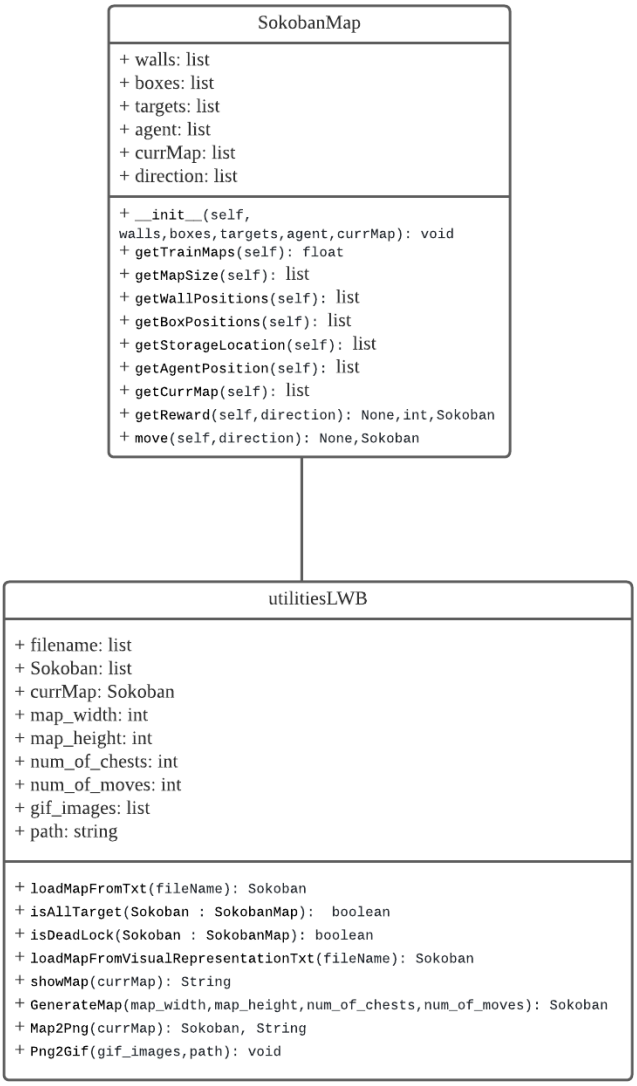


Figure 2: Helper Class Diagram

2.1 Data Structure

A neural network is the data structure that we use to keep track of the states, actions, and expected rewards. Rather than mapping a state-action pair to a q-value in Vanilla Q-leaning, it maps input states to (action, Q-value) pairs.

Our DQN constructs two neural networks that process learning. They have the same architecture but different weights. Each N time-step, the weights from the Current Network are copied to the Target Network. And Experience Replay interacts with the Environment to generate data to train the Current Network. To be specific:

- Current Network: takes the current state and actions from each Sokoban data sample and predicts a action's Q value.
- Q Network: take the following state from each data sample and predicts the max Q value out of all actions.
- Experience Replay: randomly selects a batch of sample after shuffling the training data. We used a deque in Python's built-in collections library to store the Agent's experiences. It's a list with a maximum size. If you try to append something to a full list, it will remove the first item and add the new item to the end of the list. The experiences themselves are tuples of [observation, action, reward, done flag, next state] to keep the transitions obtained from the environment.

3 Algorithm

3.1 Sequence Flow

We design the sequence flow of our DQN algorithm to be the following:

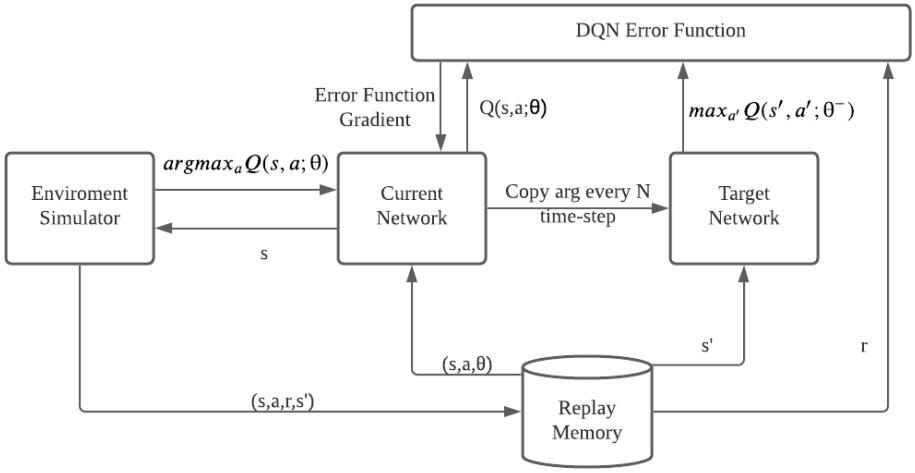


Figure 3: Sequence Flow

3.2 Pseudo-code

The pseudo-code for our expected algorithm is the following(adapted part of our code implementation from *Playing Atari with Deep Reinforcement Learning*[1].):

Initialize replay memory D to capacity N
Initialize action-value function Q with random weights θ
Initialize target action-value function $\hat{Q}$ with weight $\theta^- = \theta$
For episode = 1, M do
  Initialize sequence $s_1 = \{x_1\}$ and preprocessed sequence $\phi_1 = \phi(s_1)$
  For t = 1, T do
    With probability ϵ select a random action a_t
    else select $a_t = argmax_a\ Q(\phi(s_t), a; \theta)$
    Execute action a_t in emulator and observe reward r_t and image x_{t+1}
    Set $s_{t+1} = s_t, a_t, x_{t+1}$ and preprocess $\phi_{t+1} = \phi(s_{t+1})$
    Store transition $(\phi_t, a_t, r_t, \phi_{t+1})$ in D
    Sample random minibatch of transitions $(\phi_t, a_t, r_t, \phi_{t+1})$ from D
    Set $y_j = \begin{cases} r_j \text{ , if episode terminates at step } j+1 \\ r_j + \gamma\ max_{a'} \hat{Q}(\phi_{t+1}, a'; \theta^-) \text{ , else} \end{cases}$
    Perform a gradient descent step on $(y_j - Q(\phi_j, a_j; \theta))^2$ with respect to the network parameters θ
    Every C steps reset $\hat{Q} = Q$
  End For
End for

Figure 4: DQN with Experience Replay

3.3 Explanation

3.3.1 Q-Learning

To get the optimized policy, we should consider the values of every situation and every selection. Then we found, through analysis, that the Q(s, a) of each time image is related to the current Reward and Q(s, a) of the next time image. It may be a little confusing here since we can only know the Q value of the current image in one experiment, but how to get the Q value of the next image? The

Q values contribute this from previous experiments since Q-learning works based on infinitely many experiments in a virtual environment. In this way, we can update our current Q value based on the current reward and the Q value from the following image of the previous experiment.

3.3.2 Value Function Approximation

To achieve value function approximation, instead of a single value, we use a function to represent $Q(s, a)$ i.e.

$$Q(s, a) = f(s, a)$$

f can be any kind of functions, for example, a linear one:

$$Q(s, a) = w_1 s + w_2 a + b$$

where w_1, w_2, b are parameters. You can always get a single value Q from whatever number of dimensions. This is the basic idea of value function approximation. We use a general format to write down f as

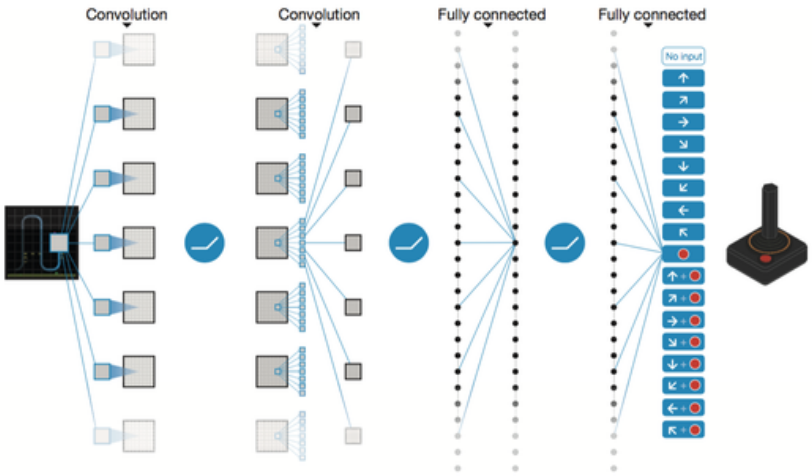
$$Q(s, a) = f(s, a, \theta)$$

where θ are the parameters. Since we don't know the actual distribution of Q values, we basically using this function to approximate it.

$$Q(s, a) \approx f(s, a, \theta)$$

3.3.3 Q Neural Network

Now, we can finally combine this method with deep learning. And to do this, we use a simple way: using a NN to replace the function f.



We use Atari's DQN as an example to explain here. The inputs are 4 continuous 84×84 images. Then 2 convolution layers, 2 fully connected layers and one output layer.

3.3.4 DQN

The training of a neural network is an optimization problem. We want an optimized loss function, the deviation between the ground truth and the network output. The goal is to minimize the loss. For this, we need a lot of samples, a massive amount of labeled data, and then use gradient descent to update the parameters of the neural network through back propagation. So we need these samples to train the DQN.

We use Q-learning to provide these labeled samples. Recall this is what based on to update the Q values,

$$R_{t+1} + \lambda maxQ(S_{t+1}, a)$$

So, we use the target Q values as the lables since our goal is getting Q values $\rightarrow$ target Q values. Thus, the loss function used to train DQN is,

$$L(\theta) = \mathbb{E}[(r + \gamma maxQ(s', a', \theta) - Q(s, a, \theta))^2]$$

3.3.5 Rewards

We introduce the fixed reward value method to implement the reward function of various actions, which are in the following categories:

Fixed Rewards List			
Actions	Rewards Value	Actions	Rewards Value
Walking	-1	Hit wall or box	-50
Move off-target	-10	Move deadlock	-100
Move on-target	10	Move all-target	100

For the Walking Rewards, we assign the reward of each plain movement, defined as the reward value of walking without pushing a box, as -1. From this penalty reward, our algorithm can avoid excessively long or unnecessary operations, which may increase the algorithm's resource consumption.

For the Moving Rewards, we assume that different positions will have different rewards. Moving the box to the target position should give a larger positive reward, while other off-target positions are negative. From this assumption, we can reduce the intermediate process of moving the box by providing a negative reward and getting a positive reward by moving the box into the correct positions. In addition, deadlock situations, such as the corner of the wall, should be punished with a larger negative reward to avoid such moving, thus, to prevent the algorithm from falling into an infinite loop or deadlock. Therefore, we divide moving reward into the following five categories:

- Move off-target reward: the reward value when the box moves to a non-target, non-deadlock position. It is an intermediate state in the algorithm solving process, and we set it to -10.
- Move on-target reward: the reward value when the box moves to the target position. It is the expected state of the algorithm, and we set it to 10.
- Hit wall or box reward: the reward value when hit boxes or wall. It is the deadlock or infinite loop state in the algorithm solving process, and we set it to -50.
- Deadlock reward: the reward value when the box moves to the deadlock position. It is the deadlock or infinite loop state in the algorithm solving process, and we set it to -100.
- Move all-target reward: the reward value when all boxes are moved to the target position. It is the final stage in the algorithm solving process, and we set it to 100.

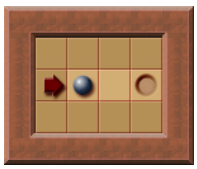
4 Properties

4.1 Deadlock heuristic

Due to the limitation of just pushing a box but never pulling it, a level can get "deadlocked." That means the level isn't solvable anymore, no matter what the user does. We call these locations a deadlock location, and the only way to solve the level is to undo a movement or restart the level. To avoid pushing the boxes into those locations, we first need to determine whether some locations are deadlocked. Here are some common deadlock types:

4.1.1 Dead square deadlocks

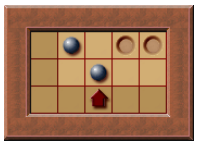
Dead square deadlocks are squares in a level that immediately create a deadlock situation when pushing a box to them[2].



The player can push the box in every direction. But moving the box to a darker shaded square will result in a deadlock. If the player pushed the box one square up, the box would still be pushable (to the left and right), but no matter what the player does, it won't be possible to push the box to the goal anymore.

4.1.2 Freeze deadlocks

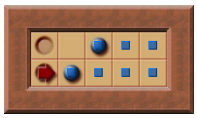
Sometimes boxes become immovable. Suppose a box becomes immovable while not located on a goal; the whole level is deadlocked. The box is "frozen" on that square and can never be pushed again[2].



Pushing the box above the player one square up results in a freeze deadlock. The box becomes immovable without being located on a goal.

4.1.3 Corral deadlocks

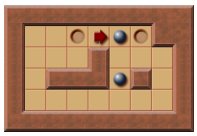
A corral is an area the player can't reach[2].



The right area (marked with little blue quadrats) isn't reachable for the player. Pushing the lower box to the right results in a deadlock situation. Even though both boxes are still pushable, they cannot reach a goal anymore.

4.1.4 Deadlocks due to frozen boxes

Frozen boxes don't create a Freeze deadlock when located on a goal. Nevertheless, they may influence the reachable area of other boxes[2].



Pushing the box next to the player to the right results in a deadlock. Since the box gets frozen, the other can't be pushed to a goal anymore.

4.2 Avoid repeating heuristic

We punish the player for going to the position he was at the last step unless this is the only feasible point.

4.3 Explore heuristic

We adapt the explore heuristic to reduce the search steps to avoid too many repetitive movements. The player only picks the direction towards the location where he never reached before or reached before for the least time. This heuristic can make players jump out of the meaningless loop as soon as possible, thus reducing the movements, leading to the optimal solution of Sokoban.

4.4 Time Complexity

Let's assign a time for each line in our pseudo-code, we get:

Initialize replay memory D to capacity N	(need t_0)
Initialize action-value function Q with random weights θ	(need t_1)
Initialize target action-value function $\hat{Q}$ with weight $\theta^- = \theta$	(need t_2)
For episode = 1, M do	(need t_3, repeat for M times)
Initialize sequence $s_1 = \{x_1\}$ and preprocessed sequence $\phi_1 = \phi(s_1)$	(need $t_{3.1}$)
For t = 1, T do	(need $t_{3.2}$, repeat for T times)
With probability ϵ select a random action a_t	(need $t_{3.2.1}$)
else select $a_t = argmax_a\, Q(\phi(s_t), a; \theta)$	(need $t_{3.2.1}$)
Execute action a_t in emulator and observe reward r_t and image x_{t+1}	(need $t_{3.2.2}$)
Set $s_{t+1} = s_t, a_t, x_{t+1}$ and preprocess $\phi_{t+1} = \phi(s_{t+1})$	(need $t_{3.2.3}$)
Store transition $(\phi_t, a_t, r_t, \phi_{t+1})$ in D	(need $t_{3.2.4}$)
Sample random minibatch of transitions $(\phi_t, a_t, r_t, \phi_{t+1})$ from D	(need $t_{3.2.5}$)
Set $y_j = \begin{cases} r_j \text{ , if episode terminates at step } j+1 \\ r_j + \gamma\, max_{a'} \hat{Q}(\phi_{t+1}, a'; \theta^-) \text{ , else} \end{cases}$	(need $t_{3.2.6}$)
Perform a gradient descent step on $(y_j - Q(\phi_j, a_j; \theta))^2$ with respect to the network parameters θ	(need $t_{3.2.7}$)
Every C steps reset $\hat{Q} = Q$	(need $t_{3.2.8}$)
End For	
End for	

Figure 5: DQN Time

In total, we get overall time:

T(episode,t)
$= t_0 + t_1 + t_2 + (t_3 + t_{3,1} + (t_{3,2} + t_{3,2,1} + t_{3,2,2} + t_{3,2,3} + t_{3,2,4} + t_{3,2,5} + t_{3,2,6} + t_{3,2,7} + t_{3,2,8}) * T) * M$
$= t_{c_1} + (t_{c_2} + t_{c_3} * T) * M$ $(Combine\ constant\ terms)$
$= t_{c_1} + (t_{c_2} * M + t_{c_3} * T * M)$
$= t_{c_1} + t_{c_2} * M + t_{c_3} * T * M$
$= t_{c_3} * T * M$
$= T * M$

In the end, we get time complexity as:
T(episode,t)
$= O(N_t N_m)$, $(N_t\ is\ the\ numbers\ of\ time - step\ in\ each\ episode,\ N_m\ is\ the\ numbers\ of\ episode)$

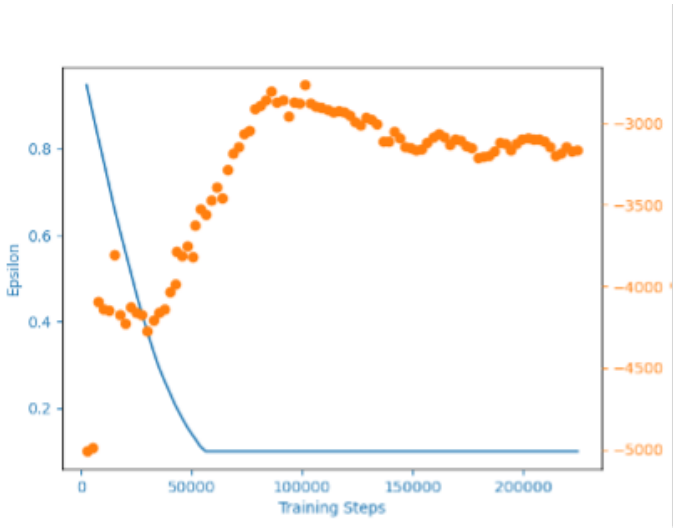
4.5 Space Complexity

The space complexity for a model-free algorithm is always sub-linear (for any T) relative to the space required to store an MDP. However, we estimated our DQN has a space complexity that is asymptotically less than the one necessary to hold an MDP.

5 Access Performance

5.1 Experimental Work

Our DQN model is trained with one Nvidia RTX2080ti GPU (11 GB), the version of our software is cuda 10.1 & pytorch 1.7.0. The sample of our training is shown as below:



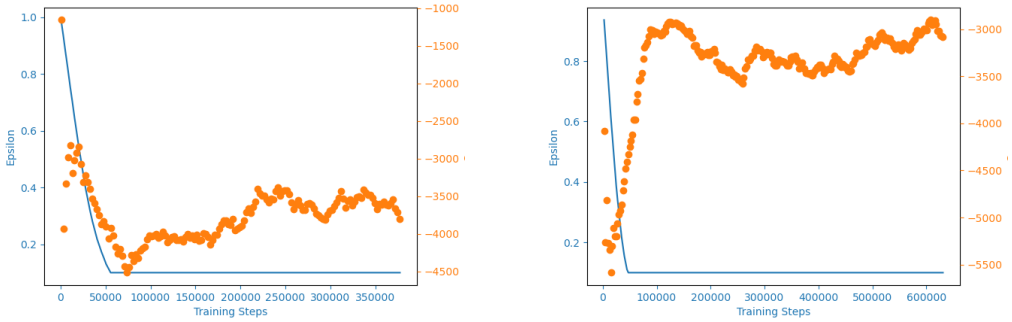
From this training result, we can see, it cost so many times to learn feasible action; this will make our algorithm avoid deadlock and infeasible movement. After that, our algorithm will learn the optimal solution of the Sokoban problem. Here are two critical processes in our training:

1. Training effective action: Our algorithm learns what is effective action. It will leverage the deadlock heuristic method to speed up its training efficiency.

2. Training optimal action: Our algorithm learns what optimal action is. It will use the avoid repeating heuristic and explore heuristic to find the shortest path to solve the problem.

5.2 Negative Result

The input-04 and input-05b show the sample of negative result:



(a) -04 (b) -05b

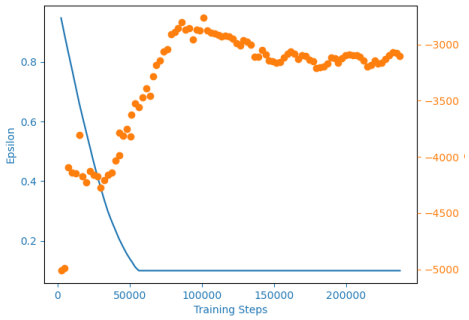
They look similar to the positive result curves, which means the training did result in some converge, which didn't lead to a satisfactory solution. The reason behind this is the critical weakness of our DQN. We lack samples with a positive reward. As we know, DQN randomly picked

$$samples\ of\ amount = batch\ size$$

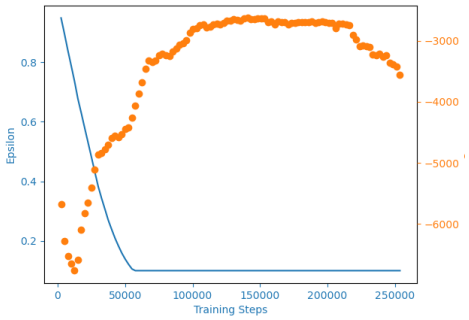
from its memory to retrain the network. But since we have way too many samples with negative rewards and only a few samples that reached the target(positive reward), the data are not balanced. That's why our agent ended up stepping back and forth in the corner to avoid that -50(hit wall) and -100(deadlock) penalty. If we have more positive samples, the agent would "know" to go for targets instead of stepping back and forth in the same location.

5.3 Positive Result

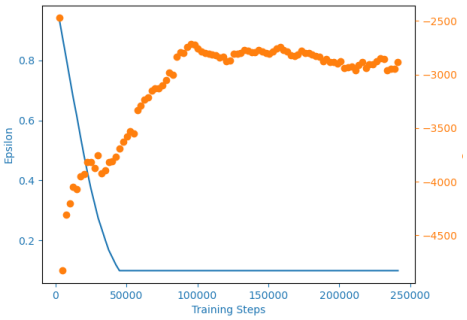
The input-01, input-02 and input-04 and input-05a shown the sample of positive result:



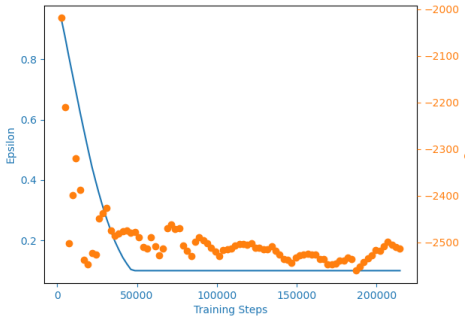
(c) -01



(d) -02



(e) -04



(f) -05a

We can see the reward (orange point) gradually increase as train steps rise and finally convergence, the epsilon (blue line) decreases as train steps increase.

We find that the reward firstly increases rapidly and can eventually convergence. Because our model learns the feasible action first, the reward in this state can grow very fast. When learning optimal movement, the speed of reward increase becomes smaller and finally convergence at some value.

The epsilon decreased as train steps increased. The epsilon decrease is that we want to get enough random samples and let the agent have the overall knowledge of the map, so we set the epsilon very large in the beginning. However, when train steps increase, to make more decisions come from the DQN model instead of randomly generated, the epsilon becomes smaller, making the training more efficient.

5.4 System Performance

The overall performance is not perfect. We believe the main reason for this is the imbalance of the sample data. If we keep on going for 1 another hour, 04 and 05b can also get solved since we would have more samples with positive rewards during the second hour. If we balance the data before each training process, the whole process will become much more effective. Please refer to the statistic of our performance on benchmark on Appendix A.

6 Observation

From the experiment results, we can find that Deep Reinforced learning cannot solve the Sokoban problem very well; there are several reasons for this poor performance:

6.1 Size and Environment

The size and environment of the problem changed so rapidly and rigidly.

We can see that the reinforcement learning method is more suitable for a single scene, like lunar landing problem, flappy bird game, or Atari game. These scenes have limited input and output, and their environments are fixed or rarely changes. But in our Sokoban problem, the action space is too large to cover. The size of each problem is different; we cannot efficiently learn lots of beneficial common experiences for other Sokoban problems in a short period. This causes the poor result of our algorithm.

6.2 Core Sampling Method

The core sampling technology has failed.

As we all know, the core sampling method in Deep Q-Learning is the memory reply scheme; it helps us train our model more efficiently and lead better performance in the final test. However, due to the hardware and training time limit, we cannot set the minibatch of memory reply too large, which makes our training inefficient.

Besides, most memory samples are deadlock movements and infeasible movements; thus, the probability of reasonable movements in-memory storage being sampled is too low. All of these make our memory reply scheme work poorly and lead our algorithm to not learn from useful experiences.

Here is an improvement for the memory reply scheme. In each time, we can throw the current action and the previous 3 consecutive actions into the training together for 4 successive steps to improve the memory ability of the network and reduce useless action memory.

6.3 Deadlock Detect

We cannot detect all deadlock positions or maps.

Even we use the deadlock heuristic to eliminate the deadlock movements in action space, and we still cannot detect all deadlock positions or maps in a single level of the Sokoban game. It makes our learning process more inefficient and even stuck in an endless loop. That's a critical reason for the poor result of the experiment.

Appendix A Performance Statistics

System Performance			
Test Case	Completion	Within Time Limit	Total Steps
01	3 out of 3	Y	55
02	3 out of 3	Y	1083
03	4 out of 4	Y	1109
04	2 out of 4	N	248
05a	2 out of 2	Y	727
05b	3 out of 4	N	354

References

[1] Volodymyr Mnih. Playing Atari with Deep Reinforcement Learning. *DeepMind Technologies*, page 5, 2013.

[2] Deadlocks - sokoban wiki. *Sokoban Wiki*, n.d.