# Sokoban

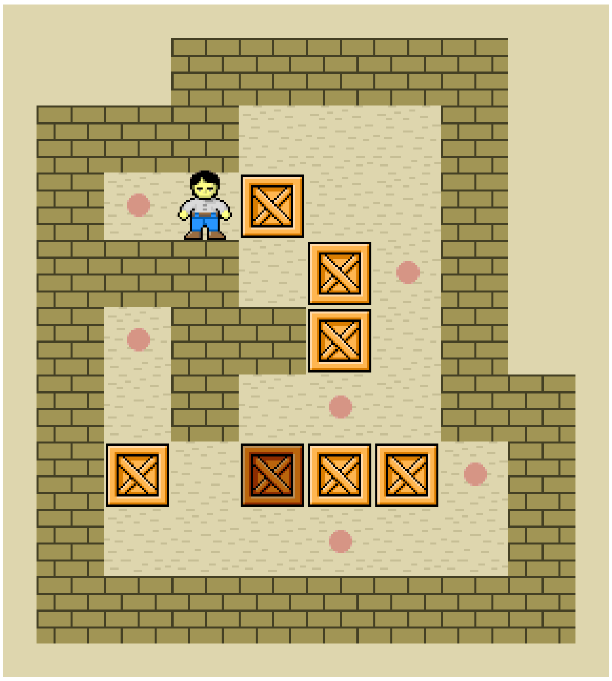
CS271 Draft Design

Qi Lin
63447178

Ziying Yu
27083755

Wenbo Li
13642959

Nov 18, 2021



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

# 1 Design

## 1.1 Algorithm Sequence Flow

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

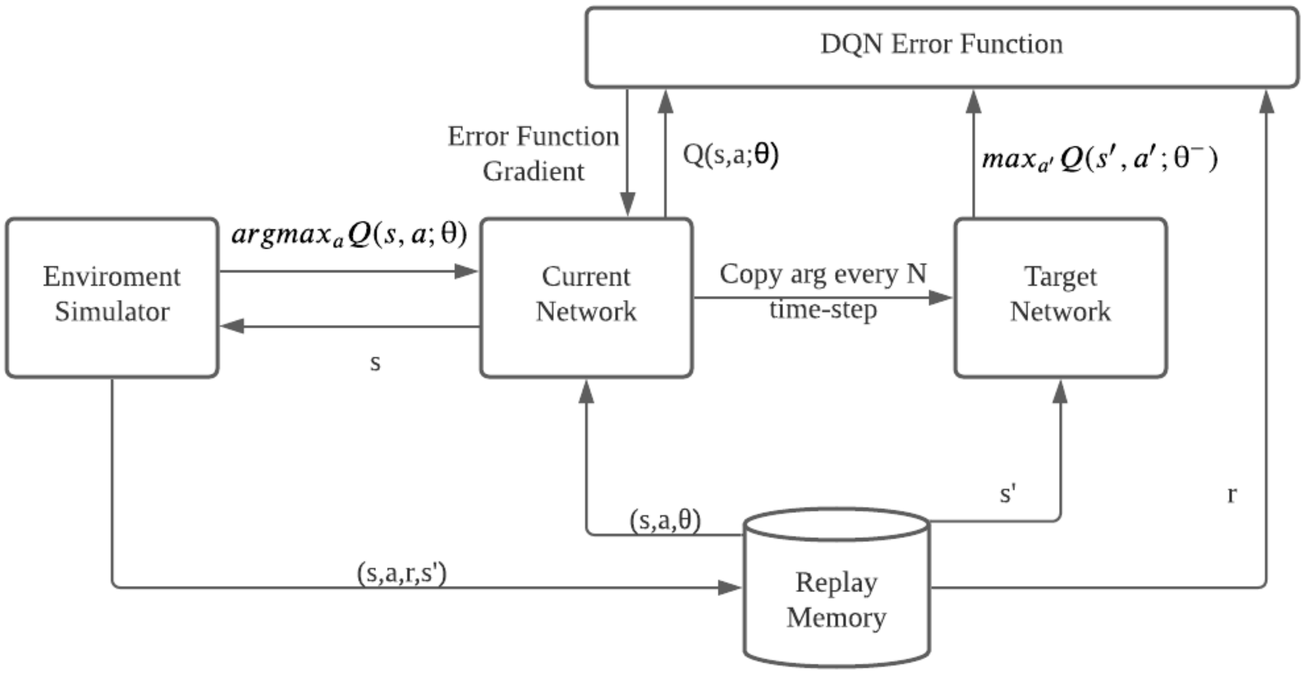


Figure 1: Sequence Flow

## 1.2 Algorithm

We choose the Deep Q learning algorithm as our reinforcement learning algorithm. We consider tasks in which a player interacts with the Sokoban emulator in a sequence of actions, observations, and rewards. The goal of the player is to interact with the emulator by selecting actions that maximize future rewards. We adapted part of our code implementation from *Playing Atari with Deep Reinforcement Learning*[1]. Our DQN architecture has two two neural nets, the Current Network and the Target Network, and a component called Experience Replay.

## 1.3 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.

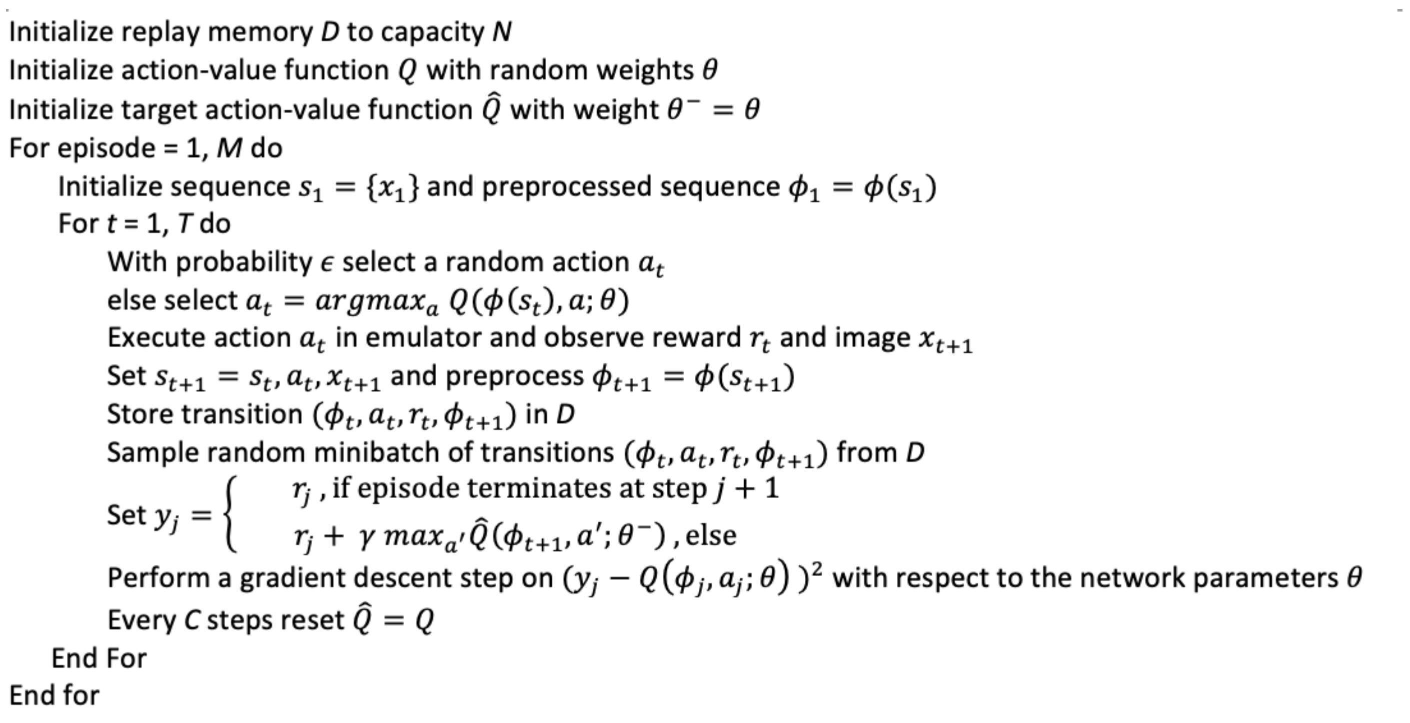
Our DQN constructs two neural networks that process learning. They have the same architecture but different weights. Each N time-step, the weights from the Current Network are copied to the Target Network. And Experience Replay interacts with the Environment to generate data to train the Current Network.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

## 2 Pseudo-code

The pseudo-code for our expected algorithm is the following:



```
Initialize replay memory $D$ to capacity $N$
Initialize action-value function $Q$ with random weights $\theta$
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 $\epsilon$ 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 $\theta$
        Every $C$ steps reset $\hat{Q} = Q$
    End For
End for
```

Figure 2: DQN with Experience Replay

## 3 Explanation

Here's a detailed explanation of the algorithm.

### 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.

For Q-learning, firstly, we need to determine how to save the Q values. The simplest way is using a matrix. List all states and actions and assign Q values for each pair of them. Like this,

|  | a1 | a2 | a3 | a4 |
|---|---|---|---|---|
| s1 | Q(1,1) | Q(1,2) | Q(1,3) | Q(1,4) |
| s2 | Q(2,1) | Q(2,2) | Q(2,3) | Q(2,4) |
| s3 | Q(3,1) | Q(3,2) | Q(3,3) | Q(3,4) |
| s4 | Q(4,1) | Q(4,2) | Q(4,3) | Q(4,4) |

Step 1: Initialize the matrix. Assign with random values. For example, all zeros.

Step 2: Start the experiment. Select action based on the current values and $\epsilon$-*greedy* method. For instance, assume we are now at state 1 but the values of all the actions are zero so we can choose any action at this moment.

|  | a1 | a2 | a3 | a4 |
|---|---|---|---|---|
| s1 | 0 | 0 | 0 | 0 |
| s2 | 0 | 0 | 0 | 0 |
| s3 | 0 | 0 | 0 | 0 |
| s4 | 0 | 0 | 0 | 0 |

Say we choose action 2, then we got a reward equals to 1 and then enters state 3. Then we update the Q value according to

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha(R_{t+1} + \lambda maxQ(S_{t+1}, a) - Q(S_b, A_t))$$

Assume we have $\alpha = 1$ and $\lambda = 1$, then we can simplify this equation to

$$Q(S_t, A_t) = R_{t+1} + maxQ(S_{t+1}, a)$$

and in this situation, it is

$$Q(s1, a2) = 1 + maxQ(s3, a)$$

which is,

$$Q(s1, a2) = 1 + 0 = 1$$

And we have a updated matrix,

|  | a1 | a2 | a3 | a4 |
|---|---|---|---|---|
| s1 | 0 | 1 | 0 | 0 |
| s2 | 0 | 0 | 0 | 0 |
| s3 | 0 | 0 | 0 | 0 |
| s4 | 0 | 0 | 0 | 0 |

Step 3: The current state is 3, assume we chose a3 then get reward 1 and go to s1. Similarly,

$$Q(s_3, a_3) = 2 + maxQ(s_1, a) = 2 + 1 = 3$$

And we have a updated matrix,

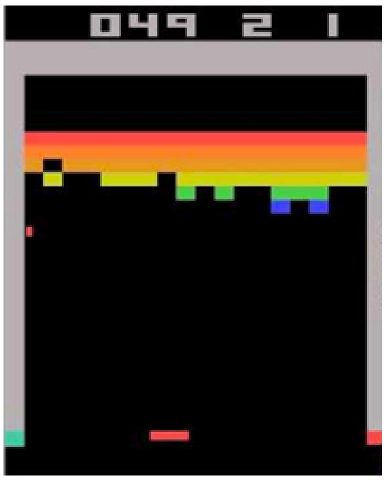
|  | a1 | a2 | a3 | a4 |
|---|---|---|---|---|
| s1 | 0 | 1 | 0 | 0 |
| s2 | 0 | 0 | 0 | 0 |
| s3 | 0 | 0 | 3 | 0 |
| s4 | 0 | 0 | 0 | 0 |

Step 4: Repeat until Q values converge.

## 3.2 Curse of Dimensionality

We used a matrix to represent $Q(s, a)$ in the previous section. However, this will not work for almost all real-world applications due to way too many states and actions. A matrix won't be enough to store them.

For example, the Atari game,



In this example, we take all the $210 \times 160$pixels as input, which is obviously banned in this project. But what I want to say is this "matrix" will have $256^{210\times160}$ variables in it, which is obviously not a good way to put them into a matrix. One way to solve this is using value function approximation.

## 3.3 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

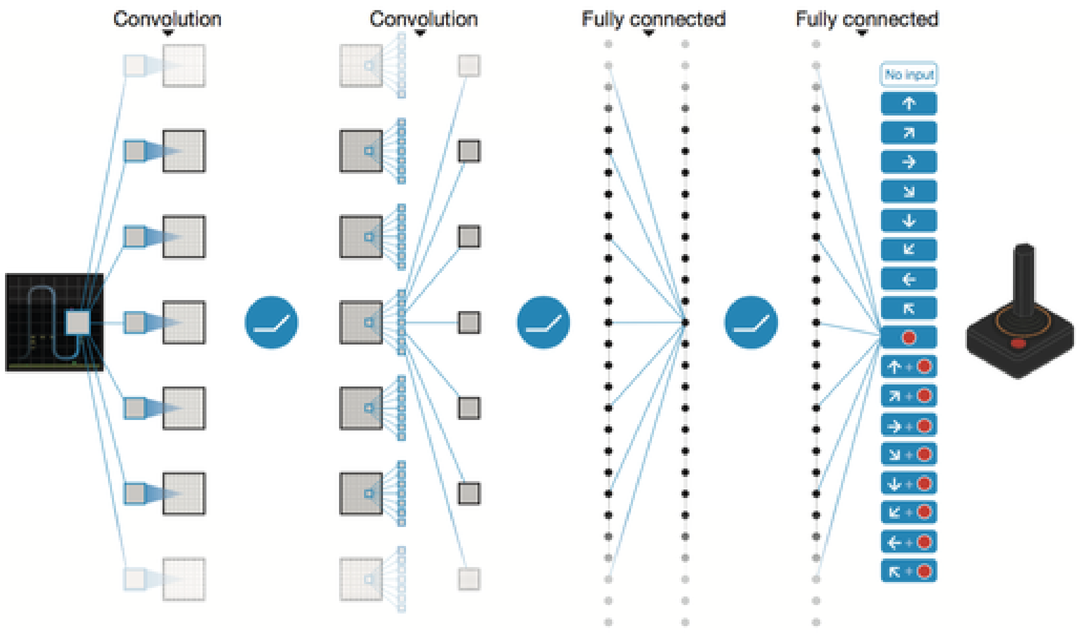
$$Q(s, a) = f(s, a, \theta)$$

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

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

## 3.4 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 $84 \times 84$ images. Then 2 convolution layers, 2 fully connected layers and one output layer.

## 3.5 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,

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

## 3.6 Training

The pseudo-code may seem complicated. But it is doing experiments and then storing the data. When we have a certain amount of data, we randomly use them and perform the gradient descent.

## 3.7 Reward Function

Designing our reward function is the core problem of our algorithm; the reward module is the core module. In the design, we first introduce the fixed reward value method to implement the reward function of various actions, which are specifically divided into the following categories:

### 3.7.1 Walking Reward

Our DQN algorithm aims to efficiently solve the Sokoban puzzle problem, so we need to minimize unnecessary movements. 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.

### 3.7.2 Moving Reward

When we push the box, it will update its position, and we define this action as moving. In this Sokoban puzzle problem, 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 four 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.
- 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.

### 3.7.3 Summarize

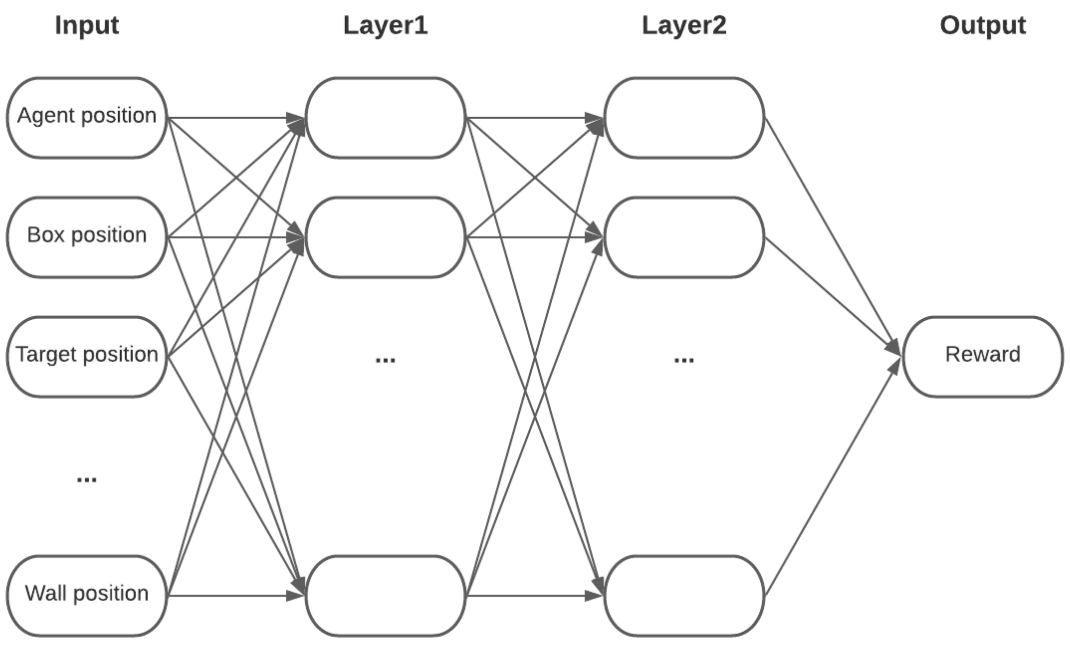
We can summarize our fixed reward value as the following table:

| Fixed Rewards List | |
|---|---|
| Actions | Rewards Value |
| Walking | -1 |
| Move off-target | -10 |
| Move on-target | 10 |
| Move deadlock | -100 |
| Move all-target | 100 |

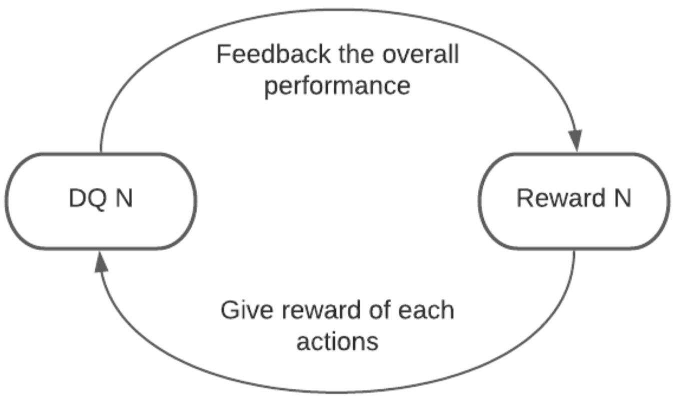
### 3.7.4 Dynamic Rewards Module

We found that sometimes the agent has to go the opposite direction just to clear the boxes on the pathway. So we believe"each step reward -1" won't be enough to describe whether a step is effective, which means that other factors also matter; for example, if there's any deadlock spot around the agent, is there any wall block?

So we came up with another interesting idea that caught our attention. In Deep Q learning, the Q function is no longer a function but a Q network instead. So why don't we use a reword network to replace our reward function here? We want to build a neural network like this,



We input some critical elements as inputs and a single output as the reward. We give some initial weights to each node then train them according to the overall performance of the Deep Q learning—specifically, all the steps needed to complete the mission. So, in other words, the overall performance of the whole Deep Q learning process is also a reward function to the reward function. Their relations are like the picture shown below,

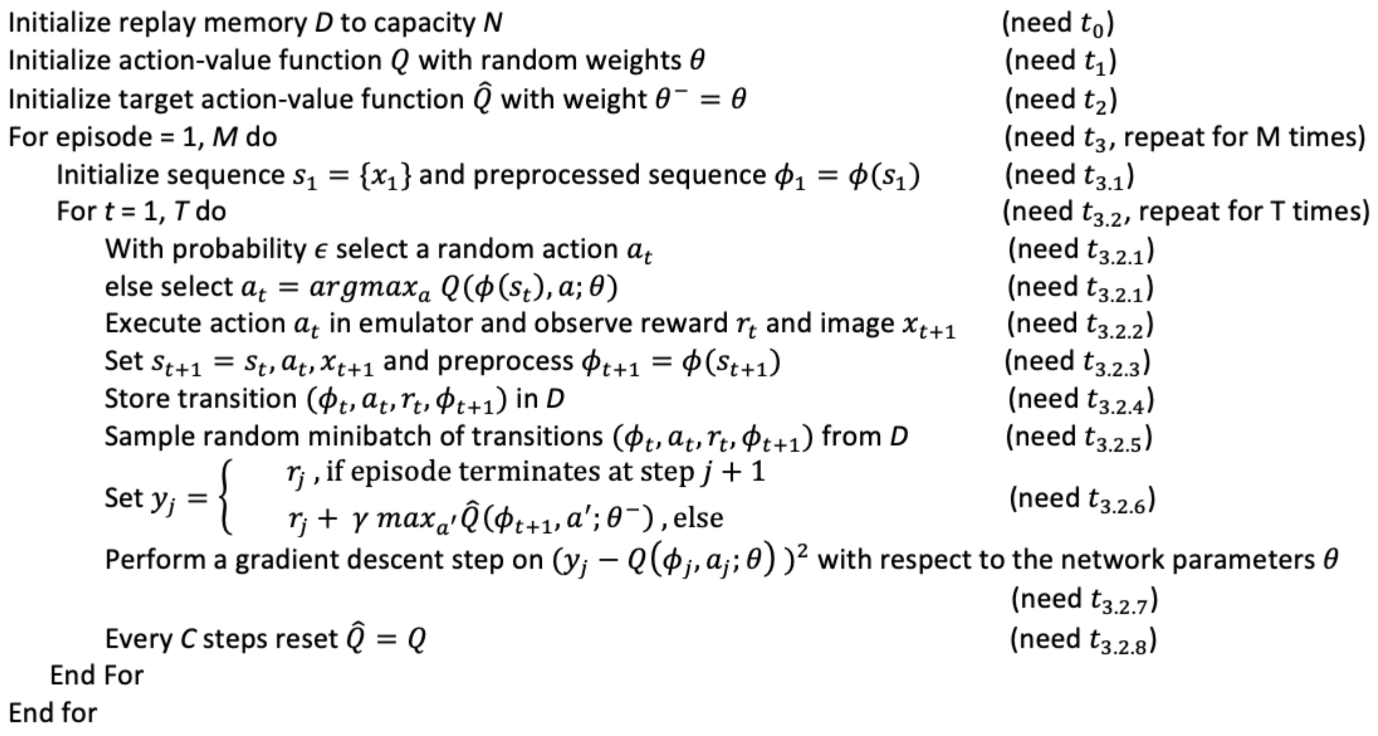


Note that this idea may not have an acceptable result (the weights of the reward network may not converge eventually). Still, we consider this an exciting way to calculate the reward, so we think it is worth trying because using ordinary ways to solve this problem is a waste of life and our valuable brain cells.

# 4 Evaluation

## 4.1 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 $\theta$                   (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 $\epsilon$ 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 $\theta$
                                                                                    (need $t_{3.2.7}$)
        Every $C$ steps reset $\hat{Q} = Q$                                         (need $t_{3.2.8}$)
    End For
End for
```

Figure 3: DQN Time

In total, we get overall time:

$$
\begin{aligned}
&\text{T(episode,t)} \\
&= \mathrm{t}_0 + t_1 + t_2 + (t_3 + t_{3,1} + (t_{3,2} + t_{3,2,1} + t_{3,2,2} + t_{3,2,3} + t_{3,2,4} + t_{3,2,5} + t_{3,2,6} + t_{3,2,7} + t_{3,2,8}) * T) * M \\
&= t_{c_1} + (t_{c_2} + t_{c_3} * T) * M \ (Combine\ constant\ terms) \\
&= t_{c_1} + (t_{c_2} * M + t_{c_3} * T * M) \\
&= t_{c_1} + t_{c_2} * M + t_{c_3} * T * M \\
&= t_{c_3} * T * M \\
&= T * M
\end{aligned}
$$

In the end, we get time complexity as:

$$
\begin{aligned}
&\text{T(episode,t)} \\
&= \mathrm{O}(\mathrm{N}_t N_m),\ (N_t\ is\ the\ numbers\ of\ time-step\ in\ each\ episode,\ N_m\ is\ the\ numbers\ of\ episode)
\end{aligned}
$$

## 4.2 Space Complexity

From the RL theory, 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 expect our DQN has a space complexity that is asymptotically less than the one necessary to hold an MDP. And we are going to explore an estimated space complexity for our algorithm along the way of implementing the game.

## References

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