Exploring DQN in DeepMind Lab ---- CS5180 Final Report

Pengkai Lyu, Xiaohu Li

Introduction

More and more robots appear in people's sight now. Most of them are designed to facilitate people's lives. For example, home robots can vacuum and mop. They contribute to significant time saving for individuals.

Safety and Efficiency are the major concerns of most households. We are curious about how home robot algorithms learn to pick up trash efficiently without knocking things over. Therefore, the primary objective in our project is to develop an algorithm that can significantly enhance the performance of these home robots.

Problem Statement

Our home robot will try its best to pick up more rewards and avoid punishments during finite steps (battery). Since a home robot will navigate based on what it sees with its on-board camera or lidar, our input will be images. Moreover, most home robots are driven by differential wheeled, so we decided to only put 3 actions into the action list, which are *Turn Left in place*, *Turn Right in place*, and *Move Forward*. With these actions, we believe it can adequately simulate the movement of real-world home robots.

Figure 1a Common Household Robots

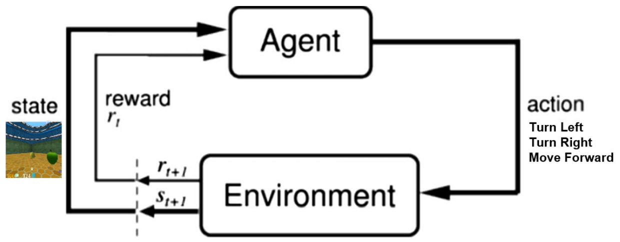


Figure 1b Problem Modeling

Simulator and dataset

To simulate and input the view of an onboard camera in a robot, we selected DeepMind Lab to serve as the simulator for our project because it can provide first-person views. Within this environment, we crafted our own reward system and developed several test maps to facilitate our research. To focus our efforts on refining the algorithms rather than on complex environment design, we simplified the home map design. We simulated an empty room where 15 positive rewards represent trash that needed to be cleaned, and 8 negative rewards stand for furniture that the agent should avoid. The customized map is shown as Figure 2, where the green spots stand for positive rewards where the yellow points represent negative rewards.

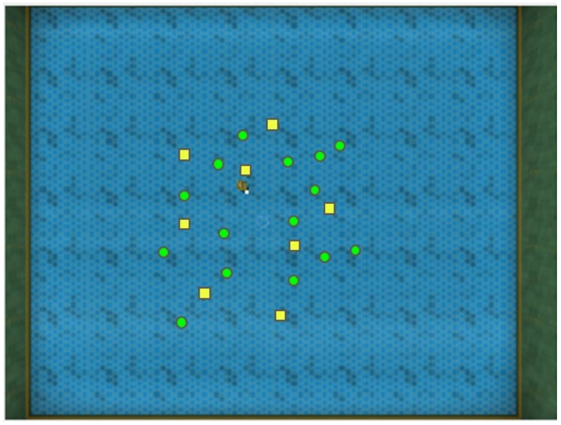


Figure 2

Methods

Because DeepMind Lab isn't as popular as Open AI Gym, we didn't find any existing implementation we can use directly. According to the algorithm we learned, we decided to start with a simple DQN first, writing from scratch so we will deeply understand DQN algorithm and see how far we can get. As we completed DQN algorithm in exercise 6, we built our algorithm based on that experience. The difference between our project and the exercise is that we add convolutional layers in our project as the increased complexity of environment. What's more, we changed the network structure according to LeNet-5, which is simple and a pioneering architecture in CNN, expecting it to bring some improvement for the DQN we implemented in exercise 6. Furthermore, we added an image buffer to store a short memory for the agent, which we proved to be of help for DQN algorithm.

Experiments

First attempt: make sure everything is working

After reviewing the DQN code in ex6, we adapted the network architecture by integrating two additional convolutional layers prior to the fully connected layers. We experimented with different sizes of input images to find an optimal balance with the replay memory size. Ultimately, we settled on an input image size of 84 x 84 and a replay memory size of 100,000. This configuration enables the agent to learn, as evidenced by a rise in the rewards curve (shown in Figure 4). But it didn't result in the best performance since the optimal reward is 15 if the agent collects all the positive rewards without touching the negative ones.

Then we input RGB color images instead of grayscale to seek an improvement on the agent as RGB giving more environmental information. However, it doesn't make the agent better but even worse (shown in Figure 4b) because RGB images bring redundant information that may become a burden to the agents. Thereafter, we convert the RGB first-person view to grayscale images as the input.

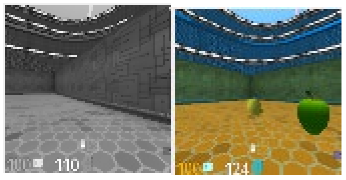


Figure 3a. Gray scaled/RGB input

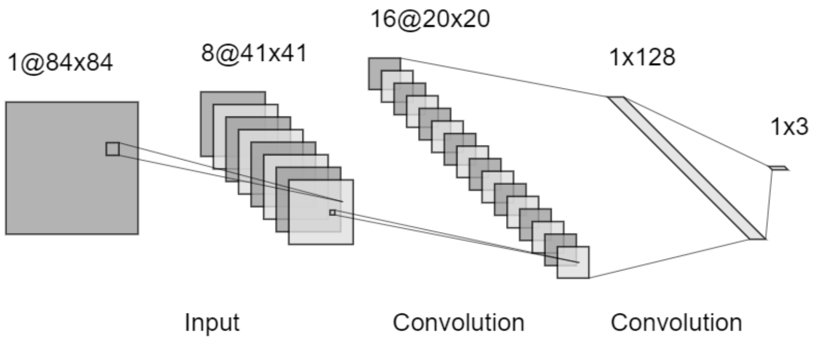


Figure 3b. Custom deep network structure

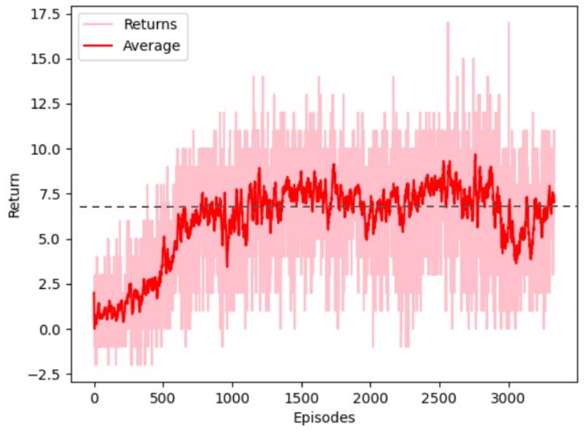


Figure 4a. trained result of grayscale input

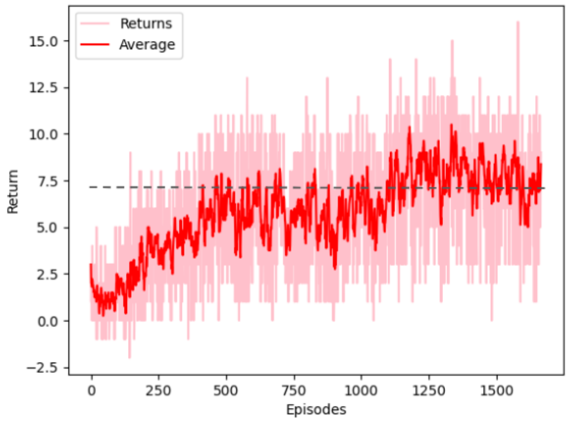


Figure 4b. trained result of RGB input

Problem

After training a few models, we tested them with agents in the environment. On the plus side, we saw that the agent picked up the rewards in the arena and tried to accumulate the rewards. However, we found that there would always be some states where the agent was “shaking its head”, not moving forward, turning left and right in place. Firstly, we tried to add more actions into the action list such as “go straight and turn right” and “go straight and turn left” so the agent will have more options to choose rather than only choose between turn left and right in place. As a result, it took more time to train. We set the training steps three times more than before, but it still couldn’t converge in the end. Then we decided to try different approaches to address this problem.

Further research, we consider that happened because two states are similar to each other so the agent couldn’t extract the features properly. With that in mind, we come up with two improvements which help the agent when it is stuck somewhere and doesn’t move forward.

Improvement 1: Change network structure

Our network is rudimentary as our first attempt is to make sure everything is working. So we did some research on CNN to seek potential network improvement. We found that LeNet-5 is simple and a pioneering architecture in CNN, which fits our project. Then according to LeNet-5, we adapted our structure by adding two subsampling layers, shown as Figure 5.

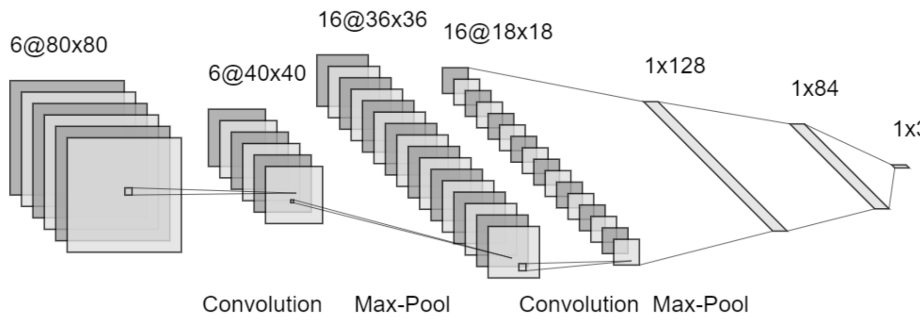


Figure 5 Modified network structure

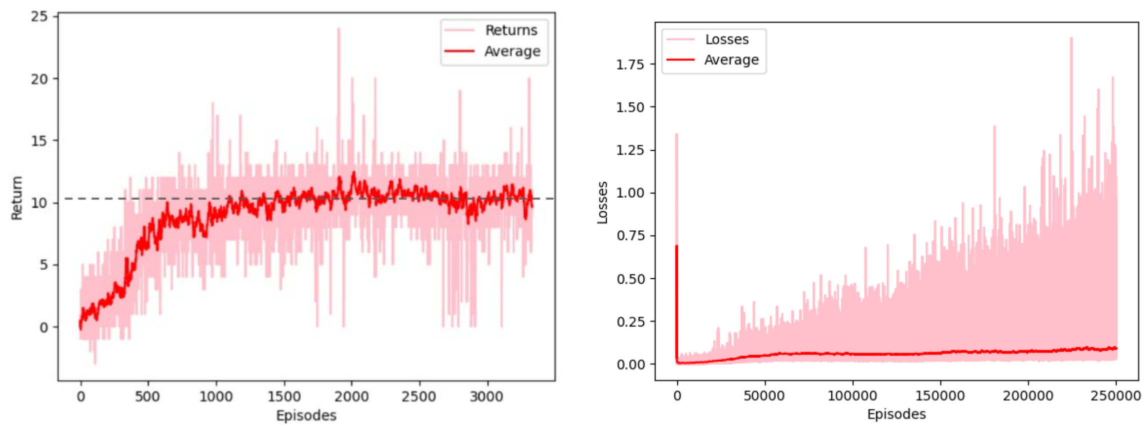


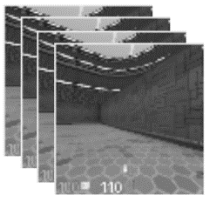
Figure 6 Trained result after first improvement

As we expected, updating the network structure brings improvement to the agent's performance. The optimal reward is increased to 10 (shown as Figure 6).

Improvement 2: Add an image buffer

In order to make the agent realize itself stuck somewhere, we believe that giving it an image buffer to help storing the scenario would be a method. Therefore, we preprocess the images before inputting them to CNN. Images are inserted at the rear and pop from the front. Buffer's length is 4 images. Hence, the agent will have the memory of the last 4 steps.

As we expected, adding an image buffer brings improvement to the agent's performance. The optimal reward is increased to 11.8 (shown as Figure 7b).



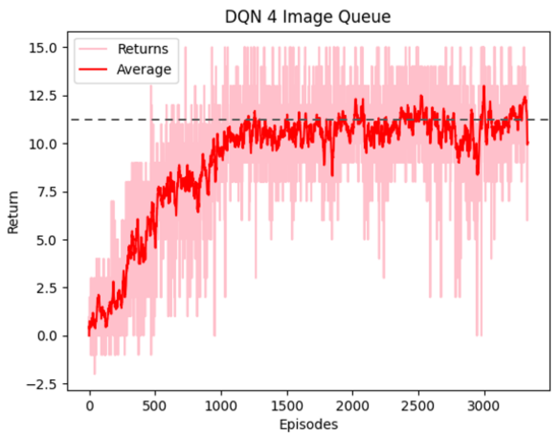


Figure 7a. Image queue

Figure 7b. Trained result

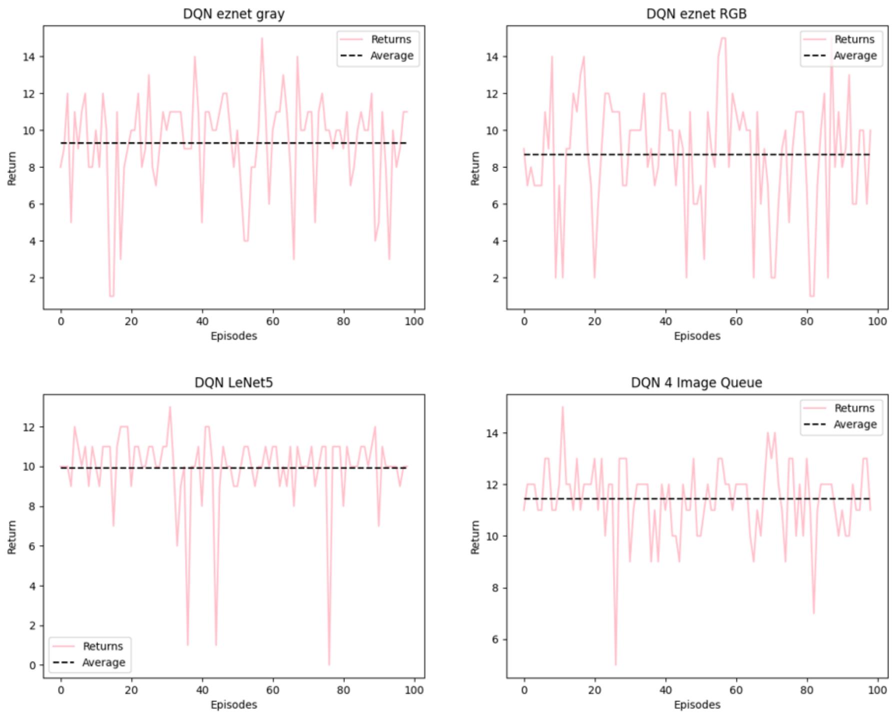


Figure 8

Testing Result and Analysis

Based on the results (Figure 8) from running 100 episodes for each of the trained models and calculating their respective average performances, in terms of effectiveness, we can deduce that the agent works better when its input are gray scaled images enqueued in an image buffer. The model utilizing an image buffer (DQN 4 Image Queue) yielded the highest performance, suggesting that incorporating a memory cell enhances the learning process. Following this, the DQN model, modified based on the LeNet-5 architecture, showed promising results, outperforming the simpler DQN models. Notably, the model with gray-scaled input did better than its RGB counterpart, indicating that simplifying the input data can be beneficial, possibly due to reduced complexity and computational demands.

In summary, the architecture of the network is crucial, and integrating a memory cell into the algorithm can significantly enhance the performance of home robot, particularly in tasks involving learning and decision-making based on first-person view input.

Discussion and Future Directions

Observing the improvement brought by the implementation of an image buffer, we are inclined to posit that the integration of Long Short-Term Memory (LSTM) networks within the DQN framework could further benefit the agent's learning process. We have come across a research paper that has a similar approach[1]. Due to time constraints, we were unable to integrate this method in our project. However, we are planning to explore the implementation of LSTM in our DQN model as a part of our future research.

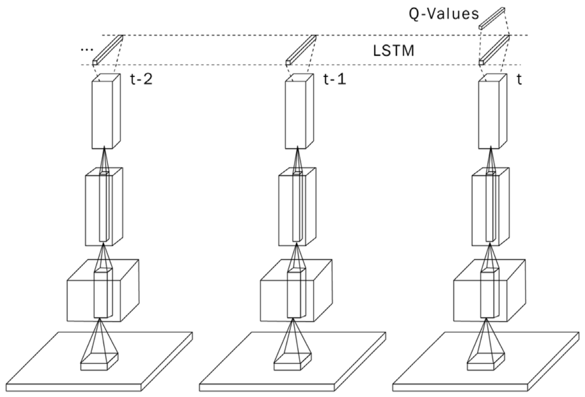


Figure 9

The challenges we encountered during our initial stage is the setup of DeepMind Lab environment. Due to inexperience with DeepMind Lab, we invest more time to configure and utilize its features. But after getting to know more about DeepMind Lab, we recognized its potential as a powerful tool for AR and MR applications. By allowing AI agents to interact with the environment in ways similar to how a human would in a virtual space, DeepMind Lab provides a robust testing ground for algorithms that could support real-world AR and MR applications.

Looking ahead, with the rise of Augmented Reality (AR) and Mixed Reality (MR), the development of applications with first-person view input will become more and more important. So we should utilized DeepMind Lab or comparable simulators that provide the first-person view to advance the development of applications that can deliver meaningful benefits to society.

[1] Hausknecht, M. (2015, July 23). Deep recurrent Q-Learning for partially observable MDPs. arXiv.org. https://arxiv.org/abs/1507.06527