Hand Cancer Killer
Using Deep Q-Network to Learn Game Strategies

Haoming Jiang

Abstract—Reinforcement learning plays an important roles in strategies setting in order to get a better performance. In this project, we show that deep reinforcement learning is very effective at learning how to play the game Radian which requires player to control the spaceship and dodge the coming bullets and survive as long as possible. The agent is not given any information other than the raw picture which is also the only input for human beings. Also the agent is aware of how long it have survived which is the score of the game. Our agent uses a convolutional neural network to evaluate the Q-funtion for a variant of Q-learning, and we show that it is able to achieve good performance. Furthermore, we discuss difficulties and potential improvements with deep reinforcement learning.

Index Terms—Reinforcement Learning, Artificial Intelligence, Q-Network, Deep Learning, Convolutional Neural Network

1 Introduction

Artificial intelligence always attracts human beings. They appears in movies, novels and human's imagination. By researching the way that human do things, we can design the AI to do the same thing. We obtain information from the environment, react and then change the environment. Finally we achieve our goal. However, it is not easy for AI. The biggest challenge is to learn the optimal policy to choose a right reaction based on current information. Reinforcement learning is one of the most powerful tools which can be applied in the above situation. Game , which has will designed rules and regular movements or reactions, is the best platform to validate this idea. Inspired by the success of AlphaGo, we decided to explore the possibility of applying this algorithm to the game Radian which always makes player thinks they have "Hand Cancer" which means they always fails by little wrong operation.

The goal of this project is to let the agent learn how to play the game Radian. Typical Radian aims at surviving as long as possible and getting scores as much as possible. However the game we discuss here is a variance of Radian which has a simpler rule than the typical one.(http://baike.baidu.com/view/3022396.htm) It is also require player to survive as long as possible. Bullets will appear semi-randomly on the screen. The player can control the space ship to dodge those bullets with movement in eight direction (up, down, right, left, up left, up right etc.) The game score is measured by how the space ship survive.

Training the agent is quite challenge because our goal is to provide the agent with only pixel information an the score. The agent is provided with information regarding neither what the space ship and bullets look like nor the speed and the locations of them. Instead it have to learn those characteristics and be able to generalize due to the large state space.

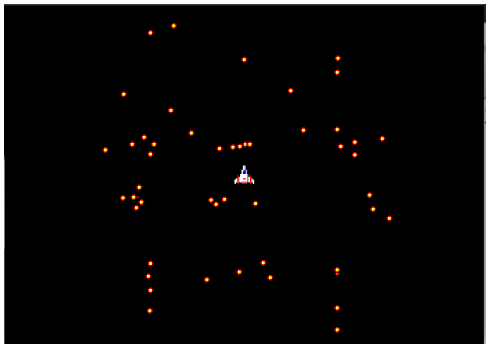

Fig. 1. Screenshot of the game

2 Related Work

The related work in this area is primarily by Google Deepmind. Mnih et al. are able to successfully train agents to play the Atari 2600 games using deep reinforcement learning, surpassing human expert-levelon multiple games [2], [1]. These works inspired this project, which is heavily modeled after their approach. They use a Deep Q-Network (DQN) to represent the Q-function for Q-learning and also use experience replay to de-correlate experiences. Their approach is essentially state-of-the-art and was the main catalyst for deep reinforcement learning, after which many papers tried to make improvements. The main strength is that they were able to train an agent despite extremely high dimensional input (pixels) and no specification about intrinsic game parameters. In fact, they are able to outperform a human expert on three out of seven Atari 2600 games. Furthermore, in march 2016, the AlphaGo which take DQN as an essential part of the whole large program, win Lee Sedol who is the best Go player in the world. However, further improvements involve prioritizing experience replay, more efficient training , and better stability when training. [1] tried

to address the stability issues by clipping the loss to +1 or -1, and by updating the target network once in every C updates to the DQN rather than updating the target network every iteration.

3 METHOD

In this section, we describe how the model is established and the algorithm framework.

3.1 MDP Formulation

The **agent** situated in an **environment** which means the game here. The environment is in a certain **state** which contains the attributes of objects in the game including locations velocity and size. Based on the state of the environment, the agent can perform certain **actions** in the environment which means the movement in eight directions. These actions sometime result in a **reward** which can be the increase of the score and penalty of the death. The environment is changed according to the previous state and what the agent reacts. At a new state, the agent can react again. The environment is stochastic, which means the next stage may to be totally predictable. **Markov decision process, MDP** is a sequence of states, actions an rewards. One **episode** is illustrated as bellow.

$$s_0, a_0, r_1, s_1, a_1, r_2, ..., s_{n-1}, a_{n-1}, r_n, s_n, r_{n+1}$$

s_n is the terminal state of the process. A MDP is based on Markov assumption, that the probability of the next state only depends on current state and action. However the Markov assumption is compromised at the current situation due to the agent cannot deduce the velocity based on single screen. In order to fix that, the current state is consist of a set of latest screens and actions.

$$s_t = \{x_{t-histLen+1}, a_{t-histLen+1}, ..., x_{t-1}, a_{t-1}, x_t\}$$

histLen which means history length is a hyperparameter.

3.2 Discounted Future Reward

In order to perform well in long-term, we need consider both the current rewards and future rewards.

For a given episode of MDP, **total reward** is:

$$R = r_1 + r_2 + ... + r_{n+1}$$

Total future reward from time point t onward can be expressed as:

$$R_t = r_t + r_{t+1} + ... + r_{n+1}$$

Due to the stochastic environment, future reward may diverge. For that reason it is common to use **discounted future reward** instead. Here we set $\gamma = 0.99$

$$R_t = r_t + \gamma r_{t+1} + \gamma^2 r_{t+2} + ... + \gamma^{n+1-t} r_{n+1} = r_t + \gamma R_{t+1}$$

A basic strategy is choosing the action which **maximizes the (discounted) future reward**. In this game only two rewards are defined:*surviving reward* and *death reward*. The meanings of these two rewards are quite straight forward. The agent will be given 0.1 *surviving reward* each step. In order to emphasis the penalty of death, The agent will be given -10 *death reward* when it collides with bullets.

3.3 Q-Learning

We define a **Q-function** *Q(s,a)* represent **the maximum discount future reward when we do action "a" in state "s"**. In order to get a higher final score, it is obvious that choosing the action with the highest Q-value once the accurate Q-function have been obtained.

Giving a close look at the relation between $Q(s, a)$ and $Q(s, a)$. The **Bellman Equation** is easy to obtained.

$$Q(s, a) = r + max_{a'} Q(s', a')$$

The main idea of Q-learning is that we can iteratively approximate the Q-function using Bellman Equation. Details will be demonstrated in the next section.

3.4 Deep Q-Network

Deep reinforcement learning is a general framework for AI to learn how to play games without giving specified game features. It take raw pixel as raw input of the algorithm. Since the state space is too large for conventional reinforcement learning, we introduce **Convolutional Neural Network, CNN**, which is quite suitable for image processing, into reinforcement learning. The optimized architecture of Q-network, is used in DeepMind paper [1], is illustrated below Fig 2.

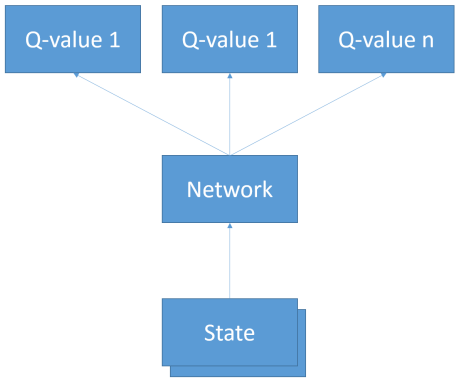

Fig. 2. Deep Q-Network

The inputs are histLen*80*80 grayscale game screen, while the outputs are Q-values for nine possible actions. A typical network architecture is presented in Fig 3

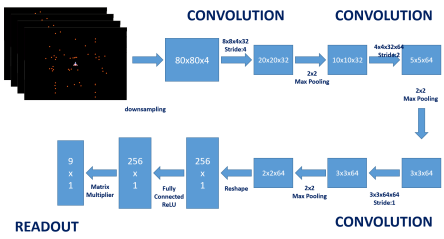

Fig. 3. Deep Q-Network

Since Q-value can be any real value, so it is a regression task, which can be optimized with simple squared error loss.

Algorithm 1 Updating the network

Input: a transition $< s, a, r, s' >$
Do a feedforward pass for the current state s to get predicted Q-values for all actions.

Do a feedforward pass for the next state s and calculate maximum overall network outputs $\max_a Q(s, a)$.

Set Q-value target for action a to $r+\lambda \max_a Q(s, a)$. For all other actions, set the Q-value target to the same as originally returned from step 1, making the error 0 for those outputs.

Update the weights using backpropagation.

$$L = \frac{1}{2}[\underbrace{r + \max_{a'} Q(s', a')}_{target} - \underbrace{Q(s, a)}_{prediction}]^2$$

Given a transition $< s, a, r, s' >$,the network is updated by the following steps (Algorithm 1).

3.5 Experience Replay

There is a large amount of tricks have to be applied, in order to make the network converge. The most trick is **experience replay** which is used for overcoming the correlation of the experiences sequence.

During gameplay all the experiences $< s, a, r, s >$ are stored in a replay memory, which has a certain size *replayMemorySize*. When training the network, random minibatches from the replay memory are used instead of the most recent transition. This breaks the similarity of subsequent training samples, which otherwise might drive the network into a local minimum. Also experience replay makes the training task more similar to usual supervised learning, which simplifies debugging and testing the algorithm. One could actually collect all those experiences from human gameplay and then train network on these.

3.6 Exploration-Exploitation

Since the Q-network is randomly initialized, its outputs are meaningless. As the Q-network converges, it returns more consist Q-values. As a result, the Q-network incorporates the exploration as a part of the algorithm. But this exploration is greedy, it settles with the first effective strategy it finds.

A simple trick to deal with this problem is $\epsilon - greedyexploration$ - with probability ϵ choose a random action, otherwise go with the "greedy" action with the highest Q-value. In our project ϵ decrease from 1 to 0.05 in 300000 steps.

3.7 Pre-processing

The state space is too large owing to large screen (680*480 pixels with three color channels). We preprocess the image before using the CNN. First, we down sample the original picture and resize it to a smaller scale. Here we test both input scales as 80*80 and 160*160. Next, we turn the screen into gray scale. Finally, we do binarization, which can let the AI be sensitive to the brim of the screen, on the gray screen. The above **pre-processing** denoted as $\phi(s)$

Algorithm 2 Deep Q-learning for Radian

initialize replay memory
initialize QDN with random weights
initialize ϵ
Loop:
  start a new episode (a new game)
  initialize state s_0
  Repeat
    extract x_t from raw pixel data update state s_t with x_t
    add experience $e_t = (\phi(s_{t-1}), a_{t-1}, r_{t-1}, \phi(s_t))$ to replay memory
    generate uniformly a random real number u between 0 and 1
    If $u < \epsilon$
      take a random action
    Else
      take the best action with highest Q-value
    scale down ϵ
    update state s_t with a_t
    update current reward r_t and total reward
    update game and refresh screen
    uniformly sample a batch of experiences from the replay memory
    backpropagate and update DQN with minibatch
  Until the aircraft crash
  restart the game
End Loop

3.8 Pipeline

The whole process of the project is illustrated in the Algorithm 2.

4 Results

Because the limitation of the time we only do some simple experiment without fine calibration. Through the experiments, we come to partial conclusion that the deep learning indeed can improve the performance. However, we also show the limitation of it. A 30s demo is available online.

4.1 Testing parameters

The game was run at 30 frames per second and $historyLength = 4$. The wards were: $survivingReward = 0.1, deathReward = -10$. The discount factor γ is set to 0.99. The ϵ used in exploration decrease from 1 to 0.05 in 300000 steps. The replay memory size is 30000. The size of minibatch is set to 32. The CNN is implemented with TensorFlow. We only bagan training after we have collect 30000 experiences.

In addition, we compared two network architectures' performance. Also, we explored the performance of the network in the easy mode (30 bullets) compared to the hard mode (50 bullets).

4.2 Two network structure

Here we proposed two sets of network in Fig 2 and Fig 5.

Actually, in the second network, the last max pooling layer should be removed. But we do not have the time to

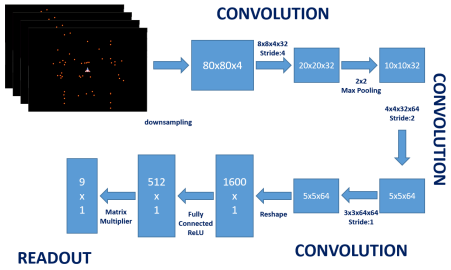

Fig. 4. Deep Q-Network 1

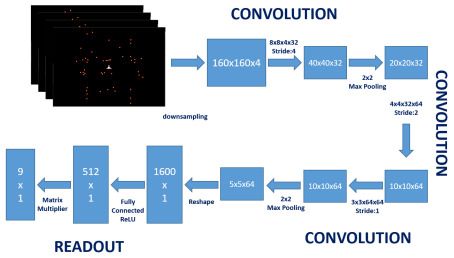

Fig. 5. Deep Q-Network 2 with bigger raw input image

revise it. The original idea of this layer is reduce the parameters in the fully connected layer and reduce the computation load, owing to the limitation of our computational resource.

We compare two trained networks' average performance in 200 rounds games in hard mode with 50 bullets. For the simple network which was trained for 3,540,000 steps, the average performance is 2.26s. However for the complex network which was trained for 1,020,000 steps, the average performance is 2.5s. Although, the complex network trained for less steps, it's performance is better than the simple network.

4.3 Less bullets

We also give a deeper insight on the average surviving time which generally grow with the rounds of games that AI have learnt. The same research is also be done in the easy mode. Form the plot we can conclude that when the AI palt more games, it can be trained better. As the figures show, the performance is still improving. Be given more time, I may reach a better performance.

5 Conclusion

Although we only do partial research, we can still obtained some useful conclusions. First, the deep reinforcement learning can really improve the AI performance, however it can not reach a high level performance with simple calibration. However, there are serval ways to improve the result. A more complex network seem like a better choice. Owing to that it have to discern the little objects (lots of bullets), the max pooling layer, which may give some computation benefits, seems like a destructor of the network. In this circumstance, the large quantity of the bullets may make the task hard. A small mistake may lead to death, which means it require the AI have to deal with low error-tolerant rate.

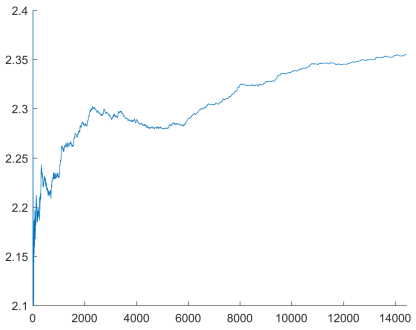

Fig. 6. The average surviving time in hard mode.

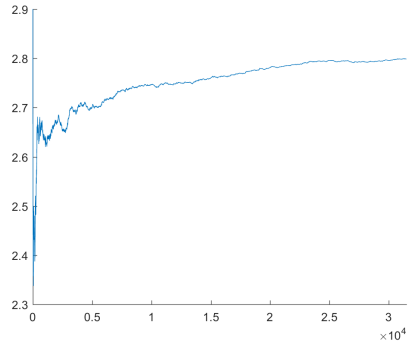

Fig. 7. The average surviving time in easy mode.

References

[1] Mnih Volodymyr, Koray Kavukcuoglu, David Silver, Andrei A. Rusu, Joel Veness, Marc G. Bellemare, Alex Graves, Martin Riedmiller, Andreas K. Fidjeland, Georg Ostrovski, Stig Petersen, Charles Beattie, Amir Sadik, Ioannis Antonoglou, Helen King, Dharshan Kumaran, Daan Wierstra, Shane Legg, and Demis Hassabis. Human-level Control through Deep Reinforcement Learning. Nature, 529-33, 2015.

[2] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin Riedmiller. Playing Atari with Deep Reinforcement Learning. NIPS, Deep Learning workshop

[3] Kevin Chen. Deep Reinforcement Learning for Flappy Bird