Running Minion: Jump over Obstacles using Reinforcement Learning

Jiayu Li lijiayu1027@ucla.edu
Zhaoyu Lu zylu@ucla.edu
Siyuan Qi syqi@cs.ucla.edu
Yutong Zhang yzhang@cs.ucla.edu

Abstract

In this project, we designed a virtural game where a Minion is running and jumping in order to survive in the harsh living environment. In the game setting, a Minion is controlled by our algorithm to run as far as possible. Along the way, the minion is running at a constant speed, and it needs to jump at an appropriate position to avoid collision with obstacles. We implemented the game in Pygame and used a reinforcement learning algorithm (Q-learning) to learn the learn a decision policy that guides the Minion to take actions at an appropriate timing. More specifically, we adopted the deep Q-learning network to train the Minion. The experiments results show the superiority of our work and the learned agent could manage to survive in the cruel living environment for a long time.

1 Introduction

Learning game pattern for intelligent agent in a game can be a great challenge because it is difficult to figure out the relationship between high-dimenstional game input with game reward. Previous works [???] apply reinforcement learning to game policy learning.

Deep Reinforcement learning proposed in [1] gives the first convincing combination of deep neural network and reinforcement learning. It is able to learn policies for Atari 2600 games directly from high-dimenstional sensory input and create a competitve performace comparing to humans.

2 Related Work

3 Learning

3.1 Q-learning

In this project, we adopted a deep Q-learning network to train the agent. In traditional Q-learning, we consider tasks in which an agent interacts with an environment, in a sequence of actions, observations and rewards. Here the environment ε is the game environment, in which at each time step the agent selects an action from a set of legitimate game actions $A = \{a_1, a_2, \cdots, a_k\}$. At each step t, the agent observes a game state x_t and takes an action a_t, and then receives a reward r_t which represents the change of the game score. The goal of the training algorithm is to let the agent interact with the environment in a way that maximizes the future rewards.

To better compute the total reward, the standard assumption of reinforcement learning is that the future rewards are discounted by a factor of γ per time step. Hence the future accumulated return at time t can be computed as $R_t = \sum_{t'=t}^{T} \gamma^{t'-t} r_{t'}$, where T is the time when the game terminates.

Then the optimal action-value function $Q^*(s, a)$ is defined as the maximum expected return by choosing an optimal following strategy, after taking an action a at state s:

$$Q^*(s, a) = \max_{\pi} E[R_t | s_t = s, a_t = a, \pi]$$

where π is a policy of sequence-action mapping.

If we know the optimal value $Q^{s',a'}$ of the sequence s' at the next time-step for all possible actions a', then the optimal strategy is to select the action a' maximizing the expected value of $r + \gamma Q^*(s', a')$:

$$Q^*(s, a) = E[r + \gamma \max_{a'} Q^*(s', a') | s, a]$$

it is common to use a function approximator to estimate the action-value function: $Q(s, a; \theta) \approx Q^*(s, a)$. It is typically a linear function approximator, but non-linear functions such as neural networks are also adopted to estimate the action-value function. A neural network function approximator with weights θ can be trained by minimizing a sequence of loss functions $L_i(\theta_i)$ that changes at each iteration i:

$$L_i(\theta_i) = E_{s,a\sim\rho(s,a)}[(y_i - Q(s, a; \theta_i))^2]$$

where $y_i = E[r + \gamma \max_{a'} Q(s', a' | \theta_{i-1}) | s, a]$ is the target for iteration i and $\rho(s, a)$ is a probability distribution over sequences s and actions a that we refer to as the behaviour distribution. The parameters from the previous iteration θ_{i1} are held fixed when optimising the loss function $L_i(\theta_i)$.

3.2 Deep Q-Learning Network

4 Experiments

4.1 Game Environment

This project(game) is implemented by Pygame, a cross-platfom set of Python modules designed for creating games. By using Pygame, we can easily build up a game scenery for our intelligent agent to play with. Furthermore, Python has many open-source machine learning packages. Therefore we can simply focus on training our agent to run and jump through all the obstacles instead of spending much time on writing common learning algorithm from scratch. These set of modules allow us to write fully featured games and multimedia programs with small amount of code. It is very easy to use and truly portable. PyGame Learning Environment (PLE) is the learning environment we used in this work. PLE mimicks the Arcade Learning Environment[2] interface, allowing a quick start to reinforcement learning in python. The goal of PLE is allow practitioners to focus design of models and experiments instead of environment design. Through PLE, we can implement the whole learning agent without major modification of the game.

4.1.1 Graphics(User interface)

The Minion has three image status when running. We set these three status chaning repeatedly with certain pattern. When jumping, the Minion has another image status. The obstacle we used here is a pipe shown in figure 1.

4.1.2 Game setting

(1) Input parameters For every step in the game process, we keep track of three game states: Horizontal distance between obstacle and agent, Height of obstacle and life status of the agent. (2) Actions For every step, the agent can choose to take an action and

Figure 1: *(a) Three patterns of the running minion. (b) Image of a jumping minion.*

Figure 2: *Obstacles in the game.*

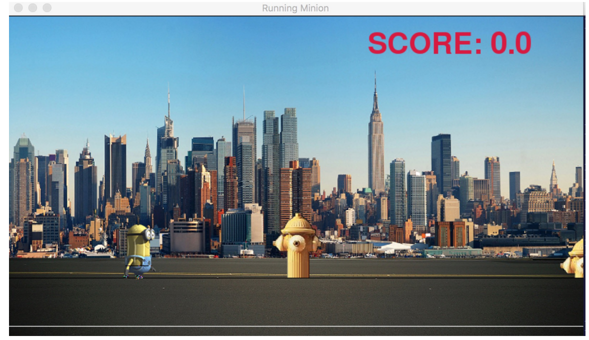

Figure 3: *The game interface.*

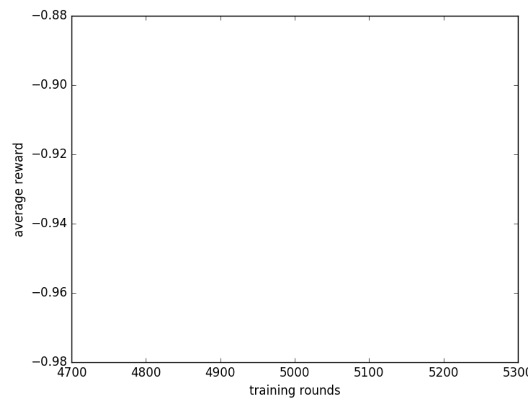

Figure 4: *Testing result."*

jump or do nothing. (3) Rewards In our game setting, if the Minion passes through one obstacle, it will gain 1 reward but will lose 5 rewards if hit on the pipe. Besides, in order to combine the appropriate rate of jumping, we set a penalty to the score whenever the Minion jump, which means the agent will lose 0.2 every time it jumps.

After one game, the more rewards the agent gained, the better model we have learned.

4.1.3 Game Learning Procedure

Step 1: Observe what state the Minion is in and perform the action (jump or not jump) that maximizes expected reward. Let the game engine perform its tick. Now the Minion is in a next state s. Step 2: Observe the new state s, and the reward associated with it. Step 3: Update the Q array according to the Q-Learning rule and pass the array to the learning procedure.

4.1.4 Experiment results

(1) Stability (Image) In reinforcement learning, the evaluation of an agent is difficult to compute during training. Instead, we compute the total reward the agent collects in one episode. From the figure, we can find that the average total reward is very noisy. This is because a small change to the weight of stragety will largely change the state of policy and thus greatly affect the reward of that policy.

(2) Different reward strategies for agent learning

5 Conclusion & Future Work

In this project, we created a game where a Minion is running and supposed to jump at certain position to avoid hitting the obstacles. We used Q-Learning to train the Minion agent and achieved Minion-self-play. The experiments results displayed demonstrate that this learned agent can evolve and gain better performance through generation. For future work, we would like to modifify the game and create a much more realistic 3D game with complex settings. What is more, we can add more forms of obstacles (gaps, different shapes of obstalces, etc) in the future to increase the difficulty level of the game. Or even we can try more learning algorithms (NEAT) and do some compasiron with Q-learning.

Acknowledgements

To Robert, for all the bagels.