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

Recently, many people are fond of playing some causual games in their leisure time. Usually, these games will be very easy to learn and play, such as with only jump action in our game setting. Although the rule of the game is simple, it is not easy to play the game by hand as the setting becomes much complicated and the obstacles generating more randomly. There has been many related works about self-palying games. General Game Playing (GGP)[Genesereth et al. 2005] was introduced to design game-playing systems with applicability to more than one specific game.P Spronck designed an adaptive game AI with dynamic scripting [Spronck et al. 2006].

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. A bunch of previous works apply machine learning algorithms to game policy learning. G.Chaslot, etc used Monte-Carlo Tree Search to implemente a frame work for game AI [Chaslot et al. 2008]. M Ponsen improved adaptive game AI with evolutionary learning [Ponsen 2004]. Despite these achievements, there have been some working using reinforcement learning to implement great game agents. M Bowling gave an An analysis of stochastic game theory for multiagent reinforcement learning [Bowling and Veloso 2002] and deep Reinforcement learning proposed in [Mnih et al. 2013] 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. Arcade Learning Environment (ALE) [Bellemare et al. 2013] is a a platform and methodology for evaluating the development of general, domain-independent AI technology. ALE is a framework that designed to make it easy to develop self-play game agents and is also an experimental methodology that can be used to evaluate the performance of agents.

In the following sections of this report, we will first briefly describe the learning methods we adapt in section4. Then the details of the game will be talked in section 5. Next, we would like to display our experiments and analyze the experimental results of our work. Finally, we will draw a conclusion and discuess about the possible future works.

2 Related Work

Reinforcement learning has been successfully applied to real-life environment for robotics controlling [Kober et al. 2013; Abbeel et al. 2007]. To better cope with real-life situations, deep learning is a heated choice [Chen et al. 2014]. Similarly, the researchers applied the algorithm to virtual environment and help the agents to finish tasks such as walking in the maze or chasing the pray [Manteghi et al. 2015; Olsen and Fraczkowski 2015]. However, in those research in virtual setting, only linear function is used in reinforcement learning due to the chance of causing not convergence of the system.

Previously, some researchers manage to combine deep learning and reinforcement learning [Mnih et al. 2013; Guo et al. 2014]. [] develops an agent with Deep Q-learning (DQN) and make it learn to play video games in Atari 2600 emulator. They apply convolution neural network to store all Q values and the input to the network is only raw pixel of the game image. The experiment result is excellent and even outperform human-level control. The paper also applies experience-replace to break the dependency of sequential frame of the game procedure. The research is a good start to extent to more games. In this paper, DQN is adopted with Pygame environment [Tasfi 2016] in our experiment to make the agent learn to survive in new game environment.

3 Learning

In this section, we briefly introduce the learning algorithm we adopted for this project. We first introduce the essential idea of Q-learning, and then introduce how to connect reinforcement learning to deep network.

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

We apply the deep Q-learning network to train our agent [Mnih et al. 2013]. Recent breakthroughs in computer vision and speech recognition have relied on efficiently training deep neural networks on very large training sets. The most successful approaches are trained directly from the raw inputs, using lightweight updates based on stochastic gradient descent. In this project, we use a reinforcement learning algorithm that takes the internal state as input.

The original architecture of connecting a neural network to reinforcement learning updates the parameters of a network that estimates the value function, directly from on-policy samples of experience, $s_t, a_t, r_t, s_{t+1}, a_{t+1}$, drawn from the algorithms interactions with the environment (or by self-play, in the case of backgammon). Recently, a new architecture exploiting the deep neural network and scalable RL algorithms was proposed. In this approach, a technique known as experience-replay is utilized. The agent's experiences $e_t = s_t, a_t, r_t, s_{t+1}$ at each time step is stored in a dataset $D = e_1, \cdots, e_N$. The experiences are pooled over many episodes into a replay memory. In the inner loop of the algorithm, the Q-learning updates are applied to samples of experience, $e \sim D$, drawn from the pool of stored samples (replay memory). After performing experience replay, the agent selects and executes an action according to an ϵ-greedy policy. Since using histories of arbitrary length as inputs to a neural network can be difficult, the Q-function instead works on fixed length representation of histories produced by a function phi.

In practice, this algorithm only stores the last N experience tuples in the replay memory, and samples uniformly at random from D when performing updates. This approach is in some respects limited since the memory buffer does not differentiate important transitions and always overwrites with recent transitions due to the finite memory size N. Similarly, the uniform sampling gives equal importance to all transitions in the replay memory.

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 [Bellemare et al. 2013] 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.

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

Figure 2: *Obstacles in the game.*

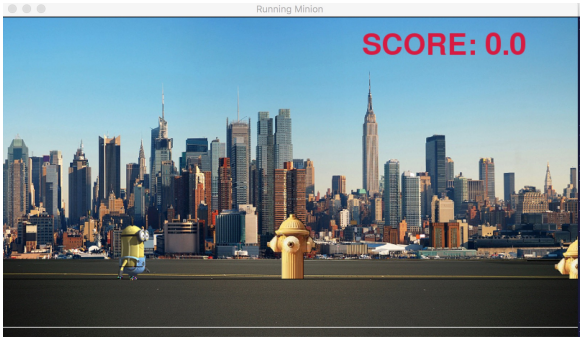

Figure 3: *The game interface.*

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 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**

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**

In the experimental part, we have tried Tried different combinations of rewards setting. The basic idea of the rewards of the game is that the Minion will gain rewards after jumping over each obstacle and will lose rewards whenver he jumps or hitted. It is obivous that the objective score should be deducted when the Minion died so the trained model would try to avoid hitting the obstacle. So our initial setting was gaining rewards after over obstacles and losing rewards when dead. However, in the training process, we found that this setting would cause the Minion to jump constantly and barely walk. So in order to let the Minion play more like the way human does, we set the rewards be deduceted whenever the Minion jumps. We call this consumes physical strength. After adding this, the trained Minion combines walking and jumping reasonably. Finally, we chose the reawards setting as gaining 1 reward after jumping over each obstacle, and losing 0.2 and 5 points respectively when hitted and jumping.

(3) **Different states chosen for agent learning**

For every step in the game process, we need to keep track of the game states as the input parameters of the training agent model. Intuitively, we could see that some state parameters such as the distance between the next obstacle and agent, height of the next obstacle and life status of the agent, the distance between the one after the next obstacle and agent, height of the one after the next obstacle and life status of the agent will matter. After trying and comparing different states, we decided to use the state parameters that achieve the best experimental results. The state contains three parameters: Horizontal distance between obstacle and agent, Height of obstacle and life status of the agent.

(4) **Different input parameters for the training of deep Q-learning network**

In the experiments, we found that the parameters for the training of deep Q-learning network need to be tuned to produce satisfying results. Since the agent is choosing an operation each frame, it is very likely that it jumps every several frames and cannot explore the effect of taking no operation for a long period of time. Hence it is important to appropriately set the ϵ, which percentage of time we perform a random action that helps exploration. In the experiments, we initialize ϵ to 0.15 and decay to 0.1 in 1000 steps.

The minimum memory size before the training start needs to be small enough for the algorithm to be efficient. Otherwise the agent will not succeed to pass a single obstacle before the training starts. Since the correlation between different frames is strong, we set the number of frames in a state to be 1 and set the number of skipped frames to be 2. The training batch size is set to be 20, which is smaller than the minimum memory size required for the replay memory of the training algorithm.

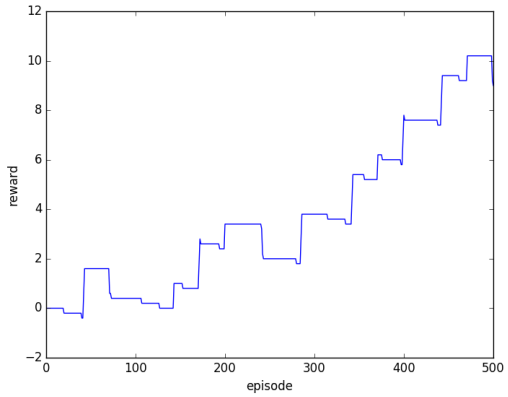

Figure 4: *Testing result of score versus number of traning epoches. As the number of training epoches increases, the score gets higher.*

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 difficulity level of the game. Or even we can try more learning algorithms (NEAT) and do some compasiron with Q-learning.

Acknowledgements

We want to give special thanks to Professor Terzopoulos for giving the great course about artificial life and overview of related techniques. We shose the project topic based on our interest and enjoyed the process.

References

ABBEEL, P., COATES, A., QUIGLEY, M., AND NG., A. Y. 2007. Reinforcement learning in robotics: A survey. *Advances in neural information processing systems*.

BELLEMARE, M. G., NADDAF, Y., VENESS, J., AND BOWLING, M. 2013. The arcade learning environment: An evaluation platform for general agents. *Journal of Artificial Intelligence Research 47* (06), 253–279.

BOWLING, M., AND VELOSO, M. 2002. Multiagent learning using a variable learning rate. *Artificial Intelligence 136*, 2, 215–250.

CHASLOT, G., BAKKES, S., SZITA, I., AND SPRONCK, P. 2008. Monte-carlo tree search: A new framework for game ai. In *AIIDE*.

CHEN, W., QU, T., ZHOU, Y., WENG, K., WANG, G., AND FU, G. 2014. Door recognition and deep learning algorithm for visual based robot navigation. *Robotics and Biomimetics*, 1793–1798.

GENESERETH, M., LOVE, N., AND PELL, B. 2005. General game playing: Overview of the aaai competition. *AI magazine 26*, 2, 62.

GUO, X., SINGH, S., LEE, H., L. LEWIS, R., AND WANG, X. 2014. Deep learning for real-time atari game play using offline monte-carlo tree search planning. *Advances in Neural Information Processing Systems*, 3338–3346.

KOBER, J., BAGNELL, J. A., AND PETERS, J. 2013. Reinforcement learning in robotics: A survey. *The International Journal of Robotics Research*.

MANTEGHI, S., PARVIN, H., HEIDARZADEGAN, A., AND NEMATI, Y. 2015. Multitask reinforcement learning in nondeterministic environments: Maze problem case. *Pattern Recognition*, 64–73.

MNIH, V., KAVUKCUOGLU, K., SILVER, D., GRAVES, A., ANTONOGLOU, I., WIERSTRA, D., AND RIEDMILLER, M. 2013. Playing atari with deep reinforcement learning. *arXiv preprint arXiv:1312.5602*.

OLSEN, M. M., AND FRACZKOWSKI, R. 2015. Co-evolution in predator prey through reinforcement learning. *Journal of Computational Science*, 118–124.

PONSEN, M. 2004. *Improving adaptive game AI with evolutionary learning*. PhD thesis, Citeseer.

SPRONCK, P., PONSEN, M., SPRINKHUIZEN-KUYPER, I., AND POSTMA, E. 2006. Adaptive game ai with dynamic scripting. *Machine Learning 63*, 3, 217–248.

TASFI, N., 2016. Pygame learning environment. https://github.com/ntasfi/PyGame-Learning-Environment.