Learning Control Policies of Lunar Lander with Reinforced Signal

Alexander Shieh

Taipei Municipal Jianguo High School, Taiwan
teweishieh@gmail.com

Abstract

This research paper aims to solve the self-taught lunar lander task with a reinforcement learning agent with no prior knowlege of control. The goal for the agent is to learn the dynamics of the lunar lander and perform a soft landing on a designated laning zone. The agent's learning process is completely automated and unsupervised, only based on observable reward from the environment. We decomposed the task to two parts: the stable control task and the direct landing task and created simulated environments for each task. We then improvised the Q-Learning algorithm integrated with neural networks to train our agents. Our experiments showed it is possible to solve the self-taught lunar lander task with this approach. Furthermore, we want to investigate more general approaches to solve self-taught control policies in the future.

1 Introduction

The original lunar lander task requires an agent to perform soft landings on designated landing zones with controlable thrust and rotation. It was first introduced in 1979 in a game developed by Atari, Inc. and became a wide spread game concept. Recently the same concept was used on real world rockets, e.g. the Falcon 9 rocket by SpaceX, which elicited our interest in investigating this task. Instead of taking the traditional approach, we wanted to study whether there were methods other than explicitly programmed control algorithms.

We surveyed several methods and found reinforcement learning is capable of learning through unknown dynamics of an environment. In the reinforcement learning perspective, the lunar lander task has a continuous state space and the agent must learn accurate control policies to perform a successful landing. Most conventional reinforcement learning algorithms need to either iterate over states and store expected reward values for actions at each state or utilize linear regression or similar algorithms to approximate these values. These characteristics kept them from learning effectly from high dimensional and continuous inputs and complex policies[1].

However, recent breakthroughs in deep learning, which strengthened the capability of neural networks, has contributed significantly to reinforcement learning applications. Neural networks served as an regression model for learning expected reward values as well as a method of dimensionality reduction for observed features of states[2]. Thus we chose this approach to solve the lunar lander task.

After some preliminary tests we found the original task was too hard, so we decided to decompose the task into two fundamental tasks: the stable control task and the direct landing task. These two tasks are essential for the agent to perform a succesful landing and we want to solve each task with effective learning algorithms.

We implemented simulated environment for both tasks and conducted detaild experiments into various aspect of this approach. As a result, we found applicable ways to solve these tasks with considerable success rate. For experiment implementation, please visit http://ck.tp.edu.tw/~ckuco/tewei/LunarLander/.

2 Related Works

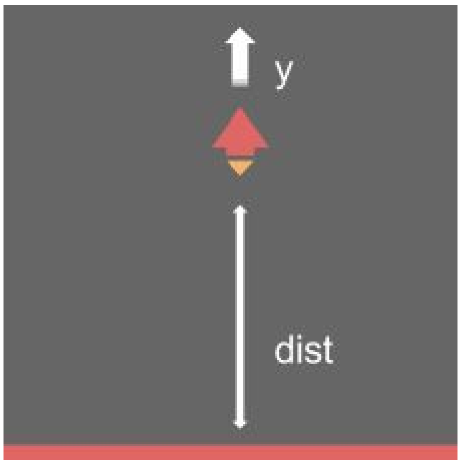
There are several reinforcement learning related approaches to the lunar lander problem. Genetic algorithms is one way to achieve a simplified version of the task[3]. Another more complex method is guided policy search, which is successful in learning mechanics of robots[4]. However, no past works using reinforcement learning can solve this task thoroughly and effectively.

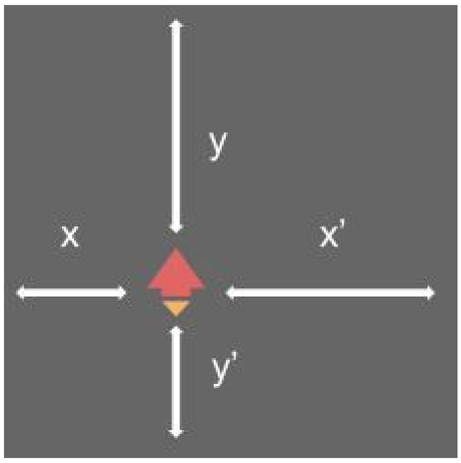
3 Experiment Constructions

3.1 Scene Construction

The environment for the stable control task is $500px \times 500px$ with a gravity of $50\,px/s^2$. The simulation ends if the agent touches the boundaries or runs out of fuel. The training cycle runs at 20 fps and the agent is limited to $10\,deg/s$ rotation speed and $100\,px/s^2$ acceleration.

As for the direct landing task, the environment is $500px \times 500px$ and has a gravity of $50\,px/s^2$. The agent needs to perform a soft landing on the ground in order to gain scores. The simulation will end if it crashes to the ground or runs out of bound. The training cycle runs at 20 fps and the successful landing criteria is set to vertical and horizontal veolcities $v_x, v_y \leq 30\,px/s$ and heading angle $cos\theta \leq 0.2$. The agent is also limited to $10\,deg/s$ rotation speed and $100\,px/s^2$ acceleration.





Figure(1): (Left) Environment construction for direct landing taks. The reward function we used mainly depends on the velocity along y-axis and the distance to ground.
Figure(2): (Right) Scene construction for the stable control task. The reward function keeps track of the max distance to boundary.

3.2 Learning Control Policies with Deep Q-Learning

Q-Learning is a common algorithm for temporal difference reinforcement learning. It learns the optimal expected reward for each state-action pair, denoted as $Q^*(s,a)$. As a result the optimal policy will be choosing the action with the highest Q-value at each state.[5] The following formulas are the definitions of Q-Learning. (1) is the definition of optimal Q-values for a state-action pair (s,a), and (2) is the optimal policy $\pi^*(s)$ at state s with learned Q-values.

$$Q^*(s,a) = \sum_{s'} P(s,a,s')[R(s,a,s') + \gamma max_{a'} Q^*(s',a')]...(1)$$
$$\pi^*(s) = argmax_a Q^*(s,a)...(2)$$

Where $P(s,a,s')$ denotes the probability of transition from state s to s' via action a, and $R(s,a,s')$ denotes the reward for such transition. γ is a constant responsible for reward discounting over a sequence of states. We used 0.9 as the value of γ.

The actual Q-Learning algorithm is a model free method, which means it doesn't learn the probabilities of state transitions directly. Instead, it learns through exploring the environment and accumulate experiences. The conventional Q-Learning algorithm is as follows:

$$Q_i^*(s,a) = Q_{i-1}^*(s,a) + \alpha[R(s,a,s') + \gamma max_{a'} Q_{i-1}^*(s',a') - Q_{i-1}^*(s,a)]...(3)$$

Where α is a small constant close to 0 that acts as averaging a newly observed reward into the Q-vlaue in the previous iteration. After many iterations, the Q-values will converge to optimal.

However, with continuous states like those described in our task, it is impossible to store Q-values for each state. Therefore, we used a method similar to [6], which utilized a neural network to approximate Q-values for each action of a state. It uses the epsilon-greedy method to explore new policies. For every sample reward the agent experienced, the algorithm performs a gradient descent step on the weights w of the neural network with loast function $L(w)$ to fit the observed sample.

$$L(w) = [R(s,a,s') + \gamma max_{a'} Q^*(s',a') - Q^*(s,a)]^2$$

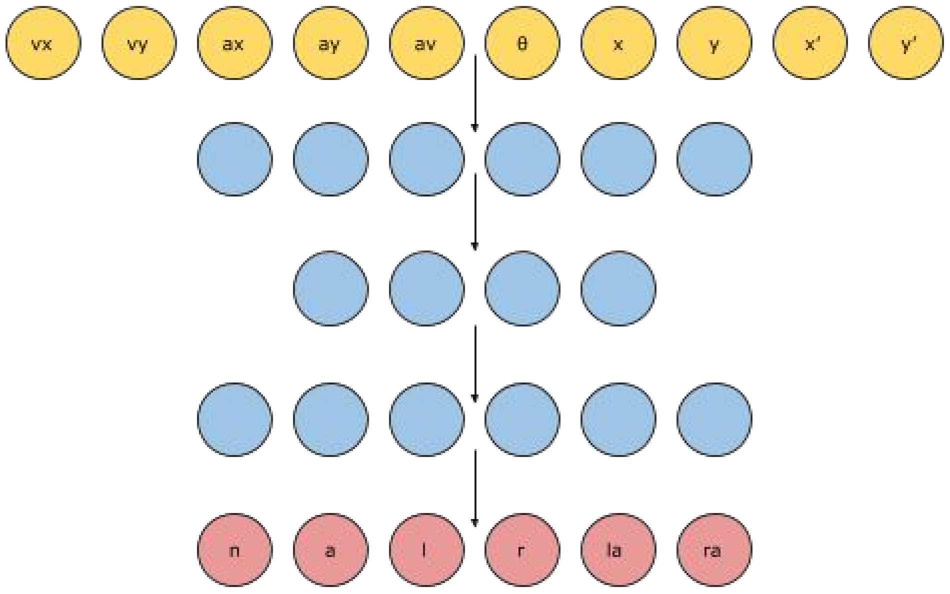
This algorithm is often refered to as Deep Q-Networks (DQN). The complete algorithm we used is as follows:

Algorithm 1: Deep Q-Learning

```
net.init()
for each epoch e:
    game.init()
    for each timestep t:
        $R_t$ = getReward($s_t$)
        applyAction($a_t$)
        if random() < eps:
            $a_t$ = random action
        else:
            $a_t$ = argmax net.forward($s_t$)
        store [$s_{t-1}$, $a_{t-1}$, ($R_t$ - $R_{t-1}$), $s_t$] into experience stack s
        end if
        if game.end():
            while !s.empty():
                pop sample [$s_{r-1}$, $a_{r-1}$, ($R_r$ - $R_{r-1}$), $s_r$] from s
                X = $s_{r-1}$
                Y = net.forward($s_{r-1}$)
                Y[$a_{r-1}$] = ($R_r$ - $R_{r-1}$) + $\gamma$ max Q($s_r$, a')
                net.train(X,Y)
            end while
        end if
    end for
end for
```

3.3 Neural Network Architecture

We used a four layer fully connected neural network to approximate Q values. The inputs are horizontal velocity(v_x), vertical velocity(v_y), horizontal acceleration(a_x), vertical acceleration(a_y), angular velocity(av), heading angle (θ), and position($(x, y), (x', y')$) of the agent and the outputs are expected reward of the six actions, namely neutral(n), accelerate(a), left(l), right(r), left accelerate(la) and right accelerate(ra). The number of hidden units are {6, 4, 6} for the three hidden layors respectively. We also used the adadelta algorithm for neural network optimization.



Figure(3): Neural network construction used in the experiments.

4 Experiment Results

4.1 Direct Landing Task

The Direct Landing Task reaches a dead end after 100 epochs of training at first. We discovered that the margin between the expected rewards of action “acceleration” and “neutral” is less than 0.02. which might be a result of the states being too indistinguishable for the agent to decide which action is better.

Therefore, we changed the reward to the delta of rewards between states and the sampling cycle from 60fps to 20 fps. The agent accomplished an impressive 49% successful landing rate during training phase after given a more stable initial state and smaller angular velocity, It eventually achieved 75% successful landings during testing.

The reward function we used combined distance to ground, rotation and velocity: $0.5log\frac{1}{dist} + 3sin\theta + log\frac{1}{V_y}$. This reward function regulates the agent from going far from ground and helps it remain in the correct heading while controling itself at a slow speed.

4.2 Stable Control Task

This task was significantly harder than the direct landing task. We tried numerous optimizations on our training algorithm, including adjusting sampling cycle, thrust and speed constraints but resulted in vain.

We then examined different kinds of reward functions and found the balance of penalties could affect greatly on training performance. The best result we got used the following reward function: $log(\frac{1}{10}^{k}) + 3sin\theta + log\frac{1}{V_y}$, where $k = \frac{max\ \{x,y,x',y'\}}{10} - 25$. This reward function penalizes the agent greatly when it goes too far from the center, turns upside down, or gains too much speed.

In some of our training sessions, the agent learned near optimal control policies and was able to correct its own heading and altitude. It is able to perform delicate control on its thrust and rotation. However, the training process is highly unstable. We suspect it might result from the agent not being able to distinguish similar consecutive states and thus unable to learn the correct frequence of thrust control that is vital for controling its altitude and speed. The training needed to achieve such results varied from 30 to 150+ epochs. It is also hard to evaluate the fitness of the agent, so we provided the following recording to show its finest capability.

Experiment Result Recording: https://youtu.be/RHi8iWSg8-s

5 Conclusion

We successfully demonstrated a possible way to solve the self-taught lunar lander task with Deep Q-learning. The agents can learn to achive considerable success rate on both stable control and direct landing tasks with great efficency.

We also discovered that in these scenarios, a good reward function is essential for learning correct policies and using the difference of rewards at each states to train the agent is more effective than using original rewards. This change allows the agent to distinguish each state-action pairs better and learn subtle controls.

6 Future Works

We want to investigate deeper into the mechanics of reinforcement learning methods and develop a more general approach to this kind of problem without relying on handbuilt features and reward functions.

In addition, we want to try different methods on learning state representations that can more accurately describe the state. For example, recurrent neural networks is a possible way to combine past actions and states together to predict at the current state[7].

We also want to study whether enlarging our action sets with methods that can output a sequence of actions will improve the control policies. Furthermore, we want to implement our algorithms using more efficient optimization libraries and incorporate GPUs for better training speed in the future.

References

[1] Cs. Szepesvári. Algorithms for Reinforcement Learning. In *Synthesis Lectures on Artificial Intelligence and Machine Learning.* Morgan Claypool Publisher, 2010.

[2] G. E. Hinton and R. R. Salakhutdinov. Reducing the Dimensionality of Data with Neural Networks. *Science 28 July 2006: Vol. 313 no. 5786 pp. 504-507.*

[3] Z. Liu. A Guided Genetic Algorithm for the Planning in Lunar Lander Games. In *GAME-ON'2006.*

[4] S. Levine and P. Abbeel. Learning Neural Network Policies with Guided Policy Search under Unknown Dynamics. In *NIPS 2014.*

[5] A. Defazio and T. Graepel. A Comparison of Learning Algorithms on the Arcade Learning Environment. *Tech Report, Australian National University.*

[6] V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra, M. Riedmiller. Playing Atari with Deep Reinforcement Learning. In *NIPS 2014 Workshop.*

[7] K. Narasimhan, T. Kulkarni, R. Barzilay. Language Understanding for Text-based Games using Deep Reinforcement Learning. In *EMNLP 2015.*