Lunar Lander using Reinforcement Learning

Krishan Subudhi*
Department of Computer Science
University of Washingtom
Seattle, WA
ksubudhi@uw.edu

Abstract

Reinforcement Learning is a techniques in Artificial Intelligence where an agent interacts with the real world and periodically receives rewards that reflect how well it is doing. In contrast to solving an MDP, where the agent has knowledge of how the world behaves, Reinforcement Learning learns the rules of the world by actually acting in the world. Lunar Lander is an environment provided by the OpenAI gym toolkit. Here the agent is a robot which tries to land on the moon surface. The agent has to learn the rules and land on the surface as smoothly and fuel efficiently as possible. In this project I train the robot/lander by using a technique called Deep Q Learning. I start with simple algorithms first and slowly move to more complex ones.

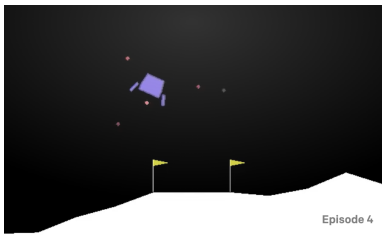


Figure 1: Visualization of the lunar lander problem

1 Introduction

In this project I have a lander which attempts to land on the moon surface. The moon has some gravity. An initial force is applied on the lander. The lander in purple is the agent and the moon is the environment. The goal is to come up with a Reinforcement Learning based algorithm such that the agent achieves the maximum reward at the end of one episode.

Landing pad is always at coordinates (0,0). Coordinates are the first two numbers in state vector. Reward for moving from the top of the screen to landing pad and zero speed is about 100..140 points. If lander moves away from landing pad it loses reward back. Episode finishes if the lander crashes or comes to rest, receiving additional -100 or +100 points. Each leg ground contact is +10. Firing main engine is -0.3 points each frame. Solved is 200 points. Landing outside landing pad is possible. Fuel is infinite, so an agent can learn to fly and then land on its first attempt. Four discrete actions available: do nothing, fire left orientation engine, fire main engine, fire right orientation engine.

*UW CSE PMP student, 2040900

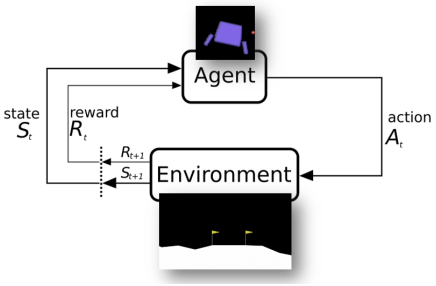


Figure 2: Agent Environment interaction state diagram

I treat the environment as a blackbox. The agent doesn't know the transition probabilities $T(s, a, s')$ and reward function $R(s, a, s')$. I tried four algorithms: Random baseline, Q Learning, Approximate Q Learning and Deep Q learning.

2 Models

2.1 Random Baseline

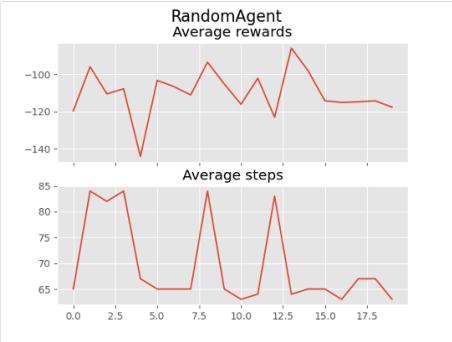


Figure 3: Random baseline

I start with an agent which takes actions randomly. As it can be seen, the agent mostly crashes and gets negative rewards every episode.

2.2 Q Learning

Q learning maintains a table of Q values for each state, action pair. Since the states are continuous , I discretized the state space. The Q values are updated using Bellman's equation

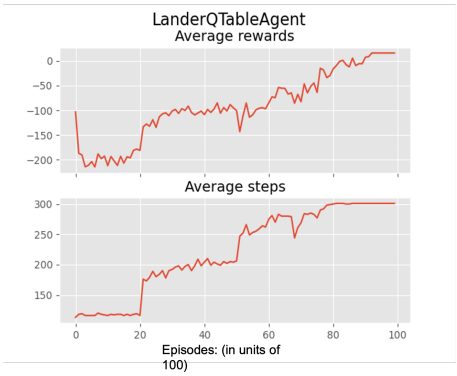
$$Q_{sample} = R(s, a, s') + \gamma \max_{a'} Q(s', a')$$
$$Q_{t+1} = (1 - \alpha)Q_t + \alpha Q_{sample}$$

I train for 10000 episodes with 300 as the maximum steps per episode. I use a learning rate α of 0.3 and gamma of 0.95. Exploration parameter ϵ is decayed step wise starting from 0.5 ending at 0.

Although Q Learning achieved positive reward (figure: 4), the high dimensionality of the discrete state space need a lot of training episodes. Also, discretization removes some details of the agent state.

2.3 Approxomate Q Learning

Q table had a lot of discrete states. It was very difficult to efficiently train the high dimensional discrete state space. Hence I needed to generalize. I used approximate Q Learning to learn a linear



(a) Q learning

(b) Approximate Q learning

Figure 4: Q learning vs approximate Q learning

representation of the Q values for each state action pair. I used a multi class perceptron algorithm for this.

$$Q(s,a) = \sum_i w_i f_i(s,a) + bias$$

However approximate Q learning converges to a lower average reward. It was difficult to understand whether there is a flaw in code or the algorithm itself. To test my code, I develped a simple game called 1DTarget.

```
[ 0 0 0 0 0 100 0 0 1 0 0]
```

Given a target(the point 100) the agent (point 1) has to take actions to efficiently reach the target. Agent can take either left or right at each state except the leftmost and rightmost states where it can take right and left respectively. When agent reaches target, it gets a reward of 1.

Approximate Q Learning Agent learned the Q value function correctly most of the times , but sometimes struggled at the edges.

```
[ 0 0 1 0 0 100 0 0 0 0 0]
[ 0 0 0 1 0 100 0 0 0 0 0]
[ 0 0 0 0 1 100 0 0 0 0 0]
[ 0 0 0 0 0 100 0 0 0 0 0]
```

Struggle at the edges

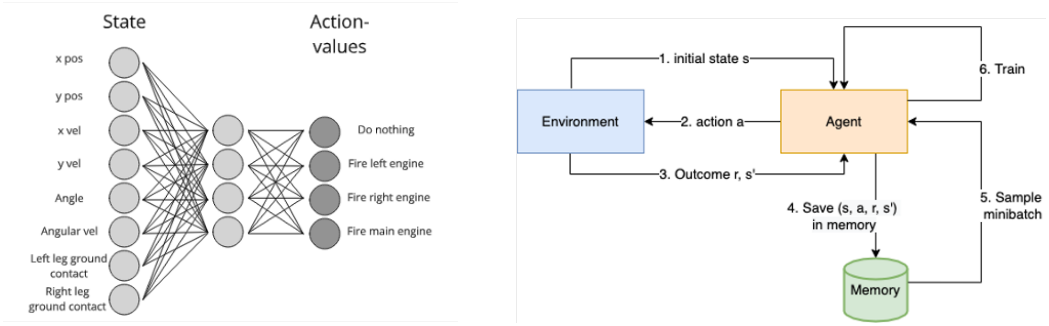
```
[ 0 0 0 0 1 100 0 1 0 0 0]
[ 0 0 0 0 0 100 1 0 0 0 0]
[ 0 0 0 0 1 100 0 1 0 0 0]
```

This proved that the code is correct.

2.4 Deep Q Learning

Although approximate Q learning works for simple games, it probably struggles with underfitting issues for complex task like the lunar lander.

DQL uses a neural network instead of a linear function approximation. But there are certain challenges of applying deep learning in RL. Conventional Deep Learning assumes iid for all the training samples. But RL training samples are generated from experience and they are highly correlated.



(a) Neural Network (b) Experience replay

Figure 5: DQL design

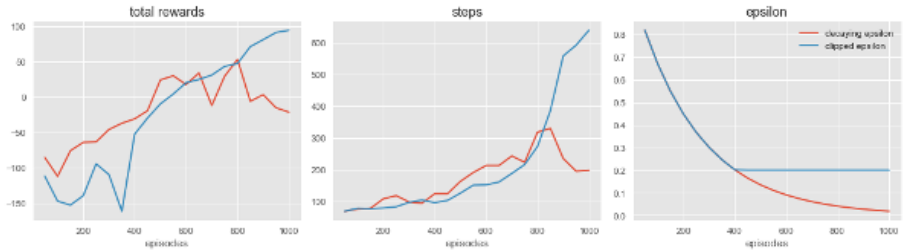


Figure 6: DQL results

To solve this DQL uses a mechanism called experience replay where the experiences are stored in memory and examples are sampled randomly during training.

I was able to achieve highly positive rewards after using DQL. I used an exponentially decaying epsilon. I had to tune the hyper-parameters extensively. For example, clipping the exploration parameter epsilon to 0.2 improved the performance.

I used a learning rate of 0.001, gamma 0.99 and an exponentially decaying epsilon. I used a neural network with two hidden layers of size 64. Replay memory was 1000000 previous samples. Batch size was 128. Maximum steps per episode was set to 1000. The agent was trained for total 1000 episodes.

3 Conclusion

I was able to successfully train the lunar lander using DQL. Through the process, I also gained numerous insights on how to approach a reinforcement learning problem. First, always start with simple algorithms and simple settings. Algorithms like Q tables and games like 1DTarget are simple enought to start with. RL needs multiple hyper parameter tuning- especially the exploration parameter ϵ. I also needed to periodically monitor the renders and graphs during training to understand what's going wrong. Using GPUs didn't speed up deep q learning.

The source code for this project is available in my github repo.

4 References

References

[1] Project Source code *https://github.com/krishansubudhi/lunarlander*, Krishan Subudhi

[2] Gadgil, Soham et al. "Solving The Lunar Lander Problem under Uncertainty using Reinforcement Learning." 2020 SoutheastCon 2 (2020): 1-8.

[3] Environment: OpenAI Gym LunarLanderV2 *https://gym.openai.com/envs/LunarLander-v2/*