Udacity Capstone Project

Machine Learning Engineer Nanodegree

Kiyohito Kunii

March 2020

1 Definition

1.1 Project Overview

Reinforcement Learning (RL) has been applied and proven to be successful in many domains, and there has been a number of RL research using an simulation environment provided by OpenAI, an independent AI research institute. OpenAI provides a RL simulation environment called OpenAI Gym, a tool to help researchers develop RL algorithms.

One of the recent breakthroughs in the RL field is deep reinforcement learning (DRL), one of which applies a convolutional neural network to a variant of Q-learning to solve different Atari games [3]. Since then, there has been a number of innovation with DRL, including the famous AlgaGo [5]. My motivation is to explore the field of DRL and have a better understanding of some of the established algorithms through this capstone project.

1.2 Problem Statement

The problem is to solve a very simple CartPole environment (see Figure 1) by training a reinforcement learning agent.

In CartPole environment, a pole is attached by an un-actuated joint to a cart, which moves along a frictionless track. The pole starts upright, and the objective is to keep it upright to avoid failing over by moving the cart in either right or left direction.

While it is tempting to explore more complicated RL environments, such as Atari or MuJoCo Physics engine [1], because it is visually interesting, I chose CartPole for the following reasons:

1. CartPole is a very simple environment and therefore lets me iterate the experiments very quickly, allowing me to focus on learning fundamentals.
2. RL is completely new to me, so I decided to focus on building the basics which is crucial to explore more complex and realistic environments in the future.

[1] http://www.mujoco.org/

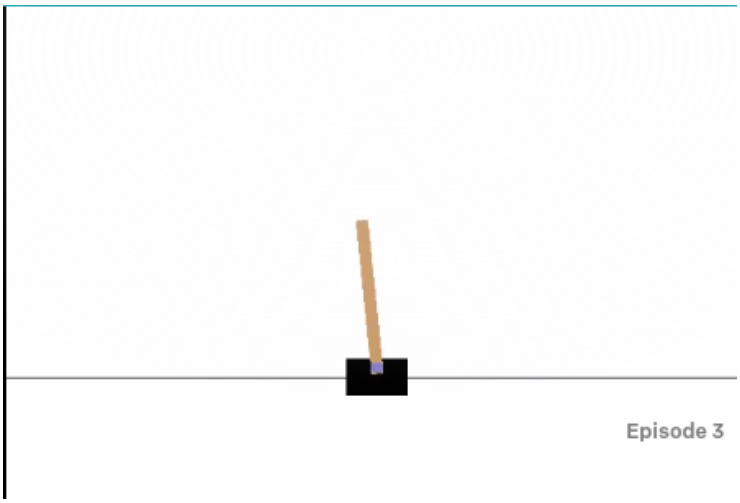

Figure 1: CartPole simulation example

Given the recent advancement in deep reinforcement learning (DRL), I chose to explore various foundations for DRL algorithms to solve this problem. While there are a number of algorithms, I decided to explore a vanilla policy gradient method called REINFORCE (Monte-Carlo policy gradient), and Actor-Critic method, since these are good starting points for state-of-art DRL algorithms such as DQN (Deep Q-learning) [3], A3C (Asynchronous Advantage Actor Critic) [2], PPO (Proximal Policy Optimization Algorithms)[**?**], and DDPG (Deep Deterministic Policy Gradients) [1]. The details of these algorithms are discussed in later section.

1.3 Evaluation Metrics

The metric used in the project is the convergence to the optimal policy measured by the average rewards over episodes. The RL agent is trained to maximise the total rewards in an environment, and therefore it is appropriate to measure the learning performance in terms of the rewards it earns over time.

2 Analysis

2.1 Data Exploration and Visualization

Unlike most of machine learning methods, reinforcement learning does not have a starting dataset. Instead, the agent's experience, in the form of rewards, actions and state space, is the dataset for training.

The inputs to the environment is the action to take in the environment. In the case of CartPole, the actions either 0 (push cart to the left) or 1 (push cart to the right).

OpenAI Gym provides unified API for all of their environments, which makes it easier to transfer an algorithm developed in a simpler environment into more challenging environment with a few tweaks.

The example of interaction with an environment in Python code is described as follow:

Listing 1: Python code example of OpenAI Gym

```
import gym

env = gym.make("CartPole-v0")
env.reset()
for _ in range(1000):
    env.render()
    observation, reward, done, info = env.step(
        env.action_space.sample()
    ) # take a random action
env.close()
```

where step function trigger the agent to take an action, either 0 (push cart to the left) or 1 (push cart to the right), and the environment returns the following information:

1. observation: a state representation after an action is taken. It is a tuple object with 4 float numbers:
   - Cart Position between -2.4 and 2.4
   - Cart Velocity between -Inf and Inf
   - Pole Angle between -41.8 and 41.8 degrees
   - Pole Velocity At Tip between -Inf and Inf
2. reward: integer value indicating the reward achieved by the action taken. The reward is 1 for every step taken, including the termination step. As mentioned, the total reward in each episode is a key indicator for how well the agent is learning how to play the game.
3. done: a Boolean value to indicate whether the game is terminated. If the Pole angle is beyond the threshold or the cart is outside of the game window, the game automatically terminates. Since the agents gets more rewards by not terminating the game, the agent is trained to balance the pole to prevent the game from terminating.

The data from the environment can be visualised by printing the outputs as follow (the code is available in *explore.py*):

Listing 2: Data exploration of CartPole environment

```
obs: [-0.03072169 -0.79471598  0.04270241  1.10735587] action:
0 reward:  1.0 done:  False
```

```
obs: [-0.046616   -0.99037244  0.06484952  1.41312371] action:
0 reward: 1.0 done: False
obs: [-0.06642345 -0.79611136  0.093112    1.141397  ] action:
1 reward: 1.0 done: False
obs: [-0.08234568 -0.99231868  0.11593994  1.46176847] action:
0 reward: 1.0 done: False
obs: [-0.10219205 -0.79879296  0.14517531  1.20743877] action:
1 reward: 1.0 done: False
obs: [-0.11816791 -0.60581366  0.16932408  0.96354664] action:
1 reward: 1.0 done: False
obs: [-0.13028419 -0.41332125  0.18859502  0.72848105] action:
1 reward: 1.0 done: False
obs: [-0.13855061 -0.22123729  0.20316464  0.50058282] action:
1 reward: 1.0 done: False
obs: [-0.14297536 -0.02947142  0.21317629  0.27817044] action:
1 reward: 1.0 done: True
```

And the visual example of the CartPole is shown in Figure 1.

2.2 Algorithms and Techniques

In this section, I briefly discuss the basics of RL. RL studies how an agent behaves in an environment in order to maximise its total reward. As shown in Figure 2, the agent interacts with an environment, and at each time step t the agent takes an action and receives observation, which affects the environment state and the reward (or penalty) it receives as the action outcome.

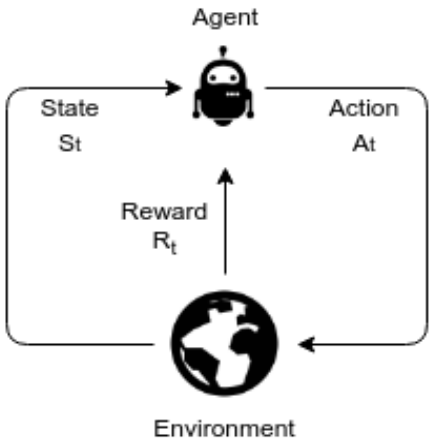

Figure 2: The interaction between an agent and an environment

2.2.1 Markov Decision Process (MDP)

An agent interacts with an environment in a sequence of discrete time steps, which is part of the sequential history of observations, actions and rewards. The sequential history is formalised as

$$H_{\mathrm{t}} = S_1, R_1, A_1, ..., A_{\mathrm{t\text{-}1}}, S_{\mathrm{t}}, R_{\mathrm{t}}. \tag{1}$$

where S is *states*, R is *rewards* and A is *actions*. A state S_{t} determines the next environment and has a *Markov property* if and only if

$$P[S_{\mathrm{t+1}}|S_{\mathrm{t}}] = P[S_{\mathrm{t+1}}|S_1, ..., S_{\mathrm{t}}]. \tag{2}$$

In other words, the probability of reaching $S_{\mathrm{t+1}}$ depends only on S_{t}, which captures all the relevant information from the earlier history [4]. When an agent must make a sequence of decision, the sequential decision problem can be formalised using the *Markov decision process (MDP)*. MDP formally represents a fully observable environment of an agent for RL.

Markov decision process (MDP) is defined in the form of a tuple $\langle S, A, T, R\rangle$ where:

- S is the set of finite states that is observable in the environment.
- A is the set of finite actions taken by the agent.
- T is *a state transition function*: $S \times A \times S \rightarrow [0, 1]$, which is a probability of reaching a future state $s' \in S$ by taking an action $a \in A$ in the current state $s \in S$.
- R is a reward function $R_{\mathrm{a}}(s, s') = P[R_{\mathrm{t+1}}|S_{\mathrm{t+1}} = s', A_{\mathrm{t}} = a, S_{\mathrm{t}} = s]$, the expected immediate reward that action a in state s at time t will return.

The objective of an agent is to solve an MDP by taking a sequence of actions to maximise the total cumulative reward it receives.

Formally, the objective of a RL agent is to maximise *expected discounted return* over time steps k,which is defined as:

$$G_{\mathrm{t}} = R_{\mathrm{t+1}} + \gamma R_{\mathrm{t+2}} + ... = \sum_{k=0}^{\infty} \gamma^{\mathrm{k}} R_{\mathrm{t+k+1}} \tag{3}$$

where γ is a discount rate $\gamma \in$ [0,1], a parameter which represents the preference of the agent for the present reward over future rewards. If γ is low, the agent is more interested in maximising immediate rewards.

2.2.2 Policies and Value Functions

Most RL algorithms are concerned with estimating *Value functions*. Value functions estimate the expected return, or discounted cumulative future reward, for a given action in a given state. The expected reward for an agent is dependent

on the agent's action. The state value function $\mathrm{v}_\pi(\mathrm{s})$ of an MDP under a *policy* π is the expected return starting from state s, which is of the form:

$$v_\pi(s) = [G_\mathrm{t}|S_\mathrm{t} = s] \text{ for all } s \in \mathcal{S} \tag{4}$$

The optimal state-value function $\mathrm{v}^*(\mathrm{s})$ maximises the value function over all policies in the MDP, which is of the form:

$$v^*(s) = \pi\mathrm{max}\ v_\pi(s) \tag{5}$$

Value functions are defined by *policies*, which map from states to probabilities of choosing each action. A policy π is defined as follows:

$$\pi(a|s) = P[A_\mathrm{t} = a|S_\mathrm{t} = s] \tag{6}$$

An optimal policy achieves the optimal state-action value functions, and it can be computed by maximising over the optimal state-action value functions. The optimal state-action-value function $q^*(s, a)$ maximises the state-action value functions over all policies in the MDP, which is of the form:

$$q^*(s, a) = \pi\mathrm{max}\ q_\pi(s, a) \tag{7}$$

2.2.3 Function Approximation and Policy Gradient Methods

RL with a tabular representation works when every state-action value can be represented. In case of very large MDPs, however, it may not be possible to represent all state-action values with a tabular representation. This problem motivates the use of *function approximation*, which estimates value function with function approximation. Unlike tabular representation, changing one weight updates the estimated value of not only one state, but many states, and this generalisation makes it more flexible to apply to different scenarios where the tabular approach could not be applied. *Policy gradient method* is a special type of function approximation in that instead of parameterised value function, the agent learns to directly estimate a parameterised policy.

The first algorithm I explore in the project is called Monte Carlo Policy Gradient (also called REINFORCE), which directly update the parameterised policy using generic stochastic gradient ascent (not descent) as follow:

$$\theta_{\mathrm{t}+1} \doteq \theta_\mathrm{t} + \alpha G_\mathrm{t} \frac{\nabla_\theta \pi(A_t|S_t, \theta)}{\pi(A_t|S_t, \theta_t)} \tag{8}$$

An extension to REINFORCE algorithm is to use state-value function (as a critic) to stabilize the high variability of policy distribution, and faster the convergence of vanilla policy gradient methods. This method of using actor (policy) and critic (state-value function) is called *actor-critic method*, which is formalized as follow:

$$\theta_{\mathrm{t}+1} \doteq \theta_\mathrm{t} + \alpha(R_{t+1} + \gamma\hat{v}(S_{t+1}, w) - \hat{v}(S_t, w)) \frac{\nabla_\theta \pi(A_t|S_t, \theta)}{\pi(A_t|S_t, \theta_t)} \tag{9}$$

Where G in REINFORCE was replaced with learnt state-value function.

Since CartPole environment is continuous states, policy gradient methods are perfect fit to solve the problem.

2.3 Benchmark

As specified in the Github repository for OpenAI Gym[2], the CartPole is considered solved when the average reward is greater than or equal to 195.0 over 100 consecutive trials.

3 Methodology

3.1 Data Preprocessing

Unlike other machine learning dataset, OpenAI platform is an experimental simulation environment, and the observations of states and rewards from the environment are clearly defined. Therefore no data preprocessing was necessary. In real RL problems, however, designing a simulation environment as well as rewards is often considered the necessary preprocessing step.

3.2 Implementation

The experiments were conducted using the following technology:

1. Python 3.6 and Anaconda
2. OpenAI gym library for simulation environment
3. PyTorch and Numpy for building non-linear function approximation
4. Matplotlib for visualising and reporting the experiment
5. The hardware used is MacBook 2.7 GHz Intel Core i7 (16GB memory), and no GPU was used to train the agent.

All dependencies are specified in the file *requirements.txt*.

As a starting point, I implemented a vanilla policy gradient method REINFORCE. The policy model is a neural network model with 2 fully-connected layers, which is updated after every episode.

Unlike the traditional software engineering where unit tests ensure the correctness of the code, debugging reinforcement learning algorithm was not straightforward because of multiple moving variables and non-deterministic nature of the problem. The two types of visualisation helped debugging process. The first visualization is Matplotlib to visualize the learning process. If the total rewards in each episode is not improving, the reason is either that there is a bug in the code, or the learning algorithm is not stable. Since the policy gradient

[2]https://github.com/openai/gym/wiki/CartPole-v0

methods algorithm was not something I invented, the unstable learning process was mostly due to a bug in the code.

Another technique to test the code is to use OpenAI's build-in monitor wrapper functionality, which records the learning simulation and stores it in JSON format and MP4 video. MP4 video is especially useful to compare the actual performance of the agent between early stage and later part of the training in human-readable way. The monitor wrapper was easily enabled by extending the environment variable as follow:

Listing 3: Python example

```
import gym
from gym import wrappers
env = gym.make("CartPole-v0")
env = wrappers.Monitor(env, "actor_critic", force=True)
...
```

The initial result of the REINFORCE algorithm is showed in Figure 3.

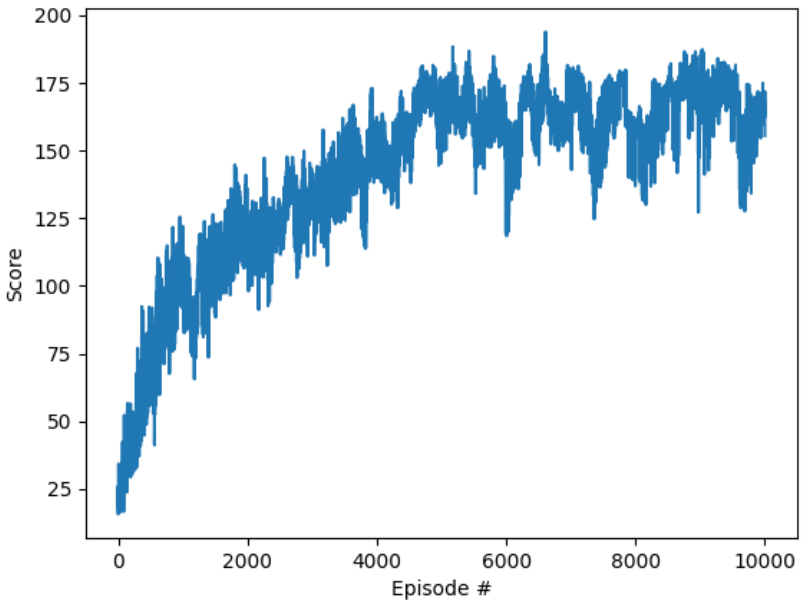

Figure 3: Training of REINFORCE algorithm

The training experiments were conducted 10 times and average the result to smooth the learning result. As can be seen, the learning curve is upward until around 5000 episodes, and stays for the rest of the episodes. Figure 4 shows one

of 10 experiments in the training process. It shows variability of convergence in different phase of episodes.

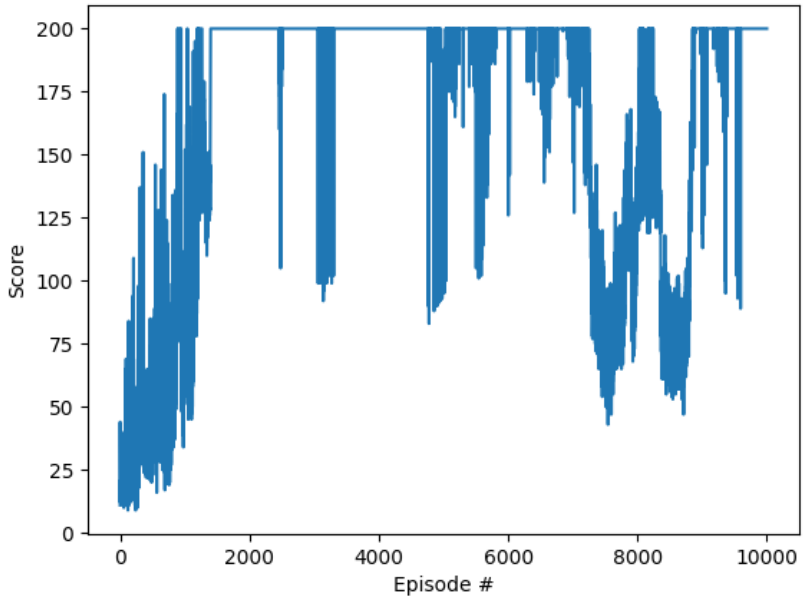

Figure 4: Example of 1 REINFORCE training

In the next section, I attemp to stabilise the learning process with actor-critic method.

3.3 Refinement

As discussed in 2.2.3, one of the extension to a vanilla policy gradient method is to use state-value function as a critic. So I extended the REINFORCE codebase by adding critic model, which is another neural network mode with 3 fully-connected layers. The presence of critic model improves the learning speed and stability, which shows the result in Figure 5.

Compared to the vanilla policy gradient method, the learning was converging faster than REINFORCE algorithm. The result is consistent with the theoretical understanding of faster and stable learning with actor-critic method.

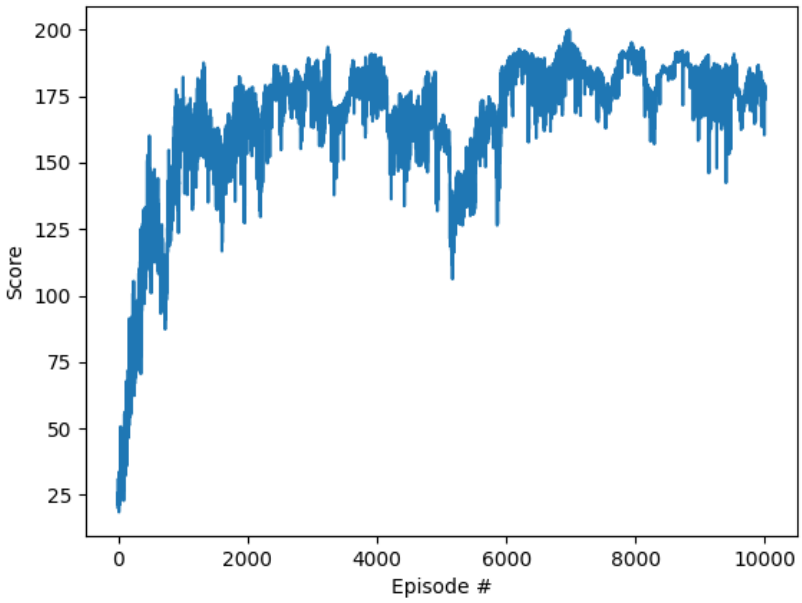

Figure 5: Training of actor-critic method

4 Results

4.1 Model Evaluation and Validation

In addition to the performance result and robustness analysis discussed in the previious sections, all the parameters used in the evaluations are summarised in Table 1.

Parameter	REINFORCE	Actor-critic Method
The number of experiment per evaluation	10	10
The number of episode per evaluation	10000	10000
Maximum time steps per episode	1000	1000
Discount rate	1.0	1.0
α (learning rate)	1e-2	1e-2
Reward for any states including a terminal state	1	1

Table 1: List of parameters used in the evaluations

We trained the agents with maximum 1000 time step, 10000 episodes, and conduct the same experiment 10 times to get the average scores in each episode.The number of time steps should be sufficient for the both algorithms to reach the

maximum score of 200. Every episode starts from a starting state. The learning rate α, determines how much parameterised policy and value function is updated each time. The rewards were assigned 1 for all states including the terminal state.

4.2 Justification and reflection

The final model of actor-critic method was used to evaluate the performance without any exploration, and succefully reached the threshold of over 195 scores over 100 consecutive trials. This is significant enough to have adequately solved the problem. Further extension to this experiment would be to

- Apply the same algorithm to harder environments
- Try the agent in longer iteration or use GPU to accelerate the learning process
- Try different types of policy gradient-based algorithms, such as A3C or PPO
- Hyper-parameter tuning of the current actor-critic method

Throughout this capstone project, I learnt the basics of RL and policy gradient methods. These are fundamental knowledge for me to further explore advanced RL algorithms, especially deep reinforcement learning, and I am confident that the experience in this capstone project prepared me for further learning journey in RL field.

References

[1] Timothy P Lillicrap, Jonathan J Hunt, Alexander Pritzel, Nicolas Heess, Tom Erez, Yuval Tassa, David Silver, and Daan Wierstra. Continuous control with deep reinforcement learning. *arXiv preprint arXiv:1509.02971*, 2015.

[2] Volodymyr Mnih, Adria Puigdomenech Badia, Mehdi Mirza, Alex Graves, Timothy Lillicrap, Tim Harley, David Silver, and Koray Kavukcuoglu. Asynchronous methods for deep reinforcement learning. In *International conference on machine learning*, pages 1928–1937, 2016.

[3] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin Riedmiller. Playing atari with deep reinforcement learning. *arXiv preprint arXiv:1312.5602*, 2013.

[4] Martin L Puterman. *Markov decision processes: discrete stochastic dynamic programming*. John Wiley & Sons, 2014.

[5] David Silver, Julian Schrittwieser, Karen Simonyan, Ioannis Antonoglou, Aja Huang, Arthur Guez, Thomas Hubert, Lucas Baker, Matthew Lai, Adrian Bolton, et al. Mastering the game of go without human knowledge. *Nature*, 550(7676):354–359, 2017.