



香港中文大學  
The Chinese University of Hong Kong

# Reinforcement Learning: function approximation

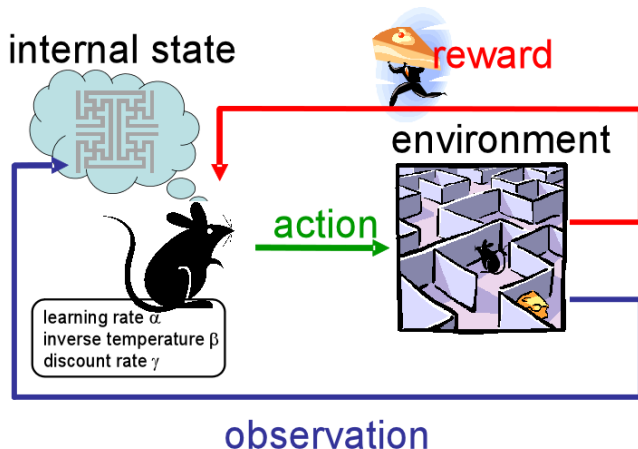
Zihao DENG  
Wenqian ZHAO

Department of Computer Science and Engineering  
The Chinese University of Hong Kong



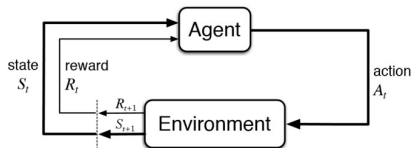
# What is Reinforcement Learning:

- ▶ different from either supervised or unsupervised learning



# What is Reinforcement Learning:

## ► Background



- Agent: take actions
- Action: S set of possible moves that Agent can make
- Environment: input: state and action taken, output: reward
- Policy: the strategy for agent to take actions
- Discount factor: dampen to make discounted future reward
- goal: obtain strategy to optimize:

$$\sum_{t=0}^{t=\infty} \gamma_t r(x(t), a(t)) \quad (1)$$



# Markov Decision Process

- ▶ The MDP gives us a precise formulation of the environment, given a state  $s_t$  we select an action  $a_t$  and observe  $s_{t+1}$  and  $r_t$  according to the transition probabilities  $P$ :
  - $S$  - set of possible states
  - $s_t \in S$  - state at step  $t$
  - $A$  - set of possible actions
  - $a_t \in A$  - selected action at step  $t$
  - $R$  - Reward function. The reward at step  $t$  is given by  $r_{t+1} = R(s_t, a_t, s_{t+1})$
  - $P$  - transition probabilities such that  $s_{t+1}P(s|s_t, a_t)$ , i.e.
  - $\rho$  - Initial state distribution such that  $s_0\rho(s)$
- ▶ Agent is defined with a policy function  $\pi(a|s)$ , mapping from states to actions and can be either deterministic or non-deterministic



# Markov Decision Process

- ▶ Given an MDP and a policy, an episode can be produced by repeating of:

- $a_t \pi(a|s_t)$
- $s_{t+1} P(s|s_t, a_t)$
- $r_{t+1} = r(s_t, a_t, s_{t+1})$

- ▶ which produce:

$$\text{episode} := s_0, a_0, r_1, s_1, a_1, r_1, \dots, s_{\tau-1}, a_{\tau-1}, r_{\tau-1}, s_{\tau}$$

- ▶ Optimal Solution gives:

$$\max_{\pi} E\left[\sum_{t=1}^{\tau} r_t\right] \quad (2)$$



# Markov Decision Process

- ▶ Value function defined as :

$$V^\pi(s) = E_\pi\left[\sum_{t=1}^{\tau} r_t | s_0 = s\right] \quad (3)$$

$$V^*(s) = \max_{\pi} E_\pi\left[\sum_{t=1}^{\tau} r_t | s_0 = s\right] \quad (4)$$

- ▶ Bellman equation: A recursive relation for value function:

$$V^*(s) = \max_{a \in A} E[r_{t+1} + V_{s_{t+1}}^* | s_t = s, a_t = t] \quad (5)$$

- ▶  $(TV)(s)$  is the Bellman operator and we can recursively calculate it and update  $V(s)$  (value iteration) to reach optimal (Monotonicity and Contraction mapping)

$$(TV)(s) = \max_{a \in A} E[r_{t+1} + V_{s_{t+1}}^* | s_t = s, a_t = t] \quad (6)$$

$$V_{k+1} = TV_k \quad (7)$$

- ▶ policy iteration use the same idea, but instead of updating value function, it update Policy  $\pi$



# Markov Decision Process

- Another approach: State-Value Function: define a quantity  $Q : S \times A \rightarrow \mathbb{R}$ :

$$Q^\pi(s, a) = \bar{R}(s, a) + \sum_{s' \in S} P_{s,a}(s') V^\pi(s') \quad (8)$$

- Recursively Calculate optimal by using Bellman Operator:

- $FQ(s, a) = \bar{R}(s, a) + \sum_{s' \in S} P_{s,a}(s') \max_{a' \in A} Q(s', a')$
- $Q(s, a) = FQ(s, a)$

- Greedy action selection is simple:

$$\pi(s) = \arg \max_{a' \in A} Q(s_{t+1}, a') \quad (9)$$



# MDP vs RL

- ▶ Difference between Markov Decision Process and Reinforcement learning:
  - MDP: the transition matrix  $P(s'|s, a)$  is known  $\rightarrow$  used to find the optimal agent
  - RL:  $P(s'|s, a)$  unknown and need to be learned :
    - i. Interacting with environment
    - ii. Requiring explicit knowledge of  $P$
- ▶ But the same idea of state-action value function can be used for Reinforcement learning

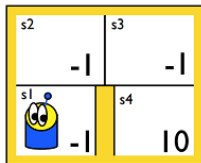




# Q-learning and SARSA

► Q-learning:

i. Build a Q-table which stores  $Q(s, a)$  for each  $s$  and  $a$  (randomly initialized). i.e.



$$\alpha = .7$$

	↑	↓	←	→
<b>s<sub>1</sub></b>	0	0	0	0
<b>s<sub>2</sub></b>	0	0	0	0
<b>s<sub>3</sub></b>	0	0	0	0
<b>s<sub>4</sub></b>	0	0	0	0

Q-Table



# Q-learning and SARSA

- ▶ Q-learning:

- ii. update  $Q(s, a)$  with:

$$Q_{k+1} := (1 - \gamma_k)Q_k + \gamma_k(r + \max_{a' \in A} Q_k(s', a')) \quad (10)$$

where  $\gamma_k$  is the learning rate, with  $\sum_{k=1}^{\infty} \gamma_k = \infty$  and  $\sum_{k=1}^{\infty} \gamma_k^2 < \infty$  :

$$Q_{k+1} = Q_k + \gamma_k(r + \max_{a' \in A} Q_k(s', a') - Q_k(s, a)) \quad (11)$$

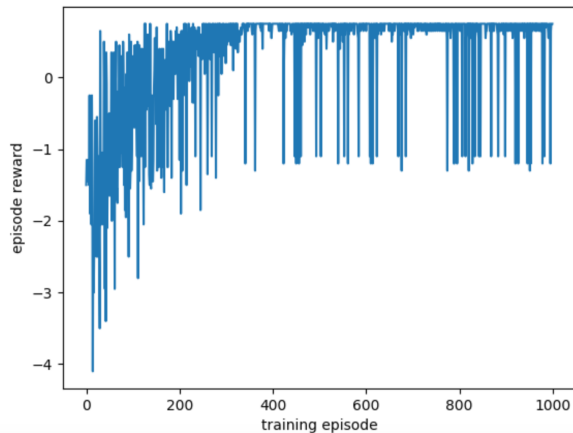
- ▶ Q-learning Demo: [maze 4\\*4](#)



# Q-learning and SARSA

## ► Q-learning result:

Reward of episodes of experiment on Maze game



# Q-learning and SARSA

- Exploration:  $\epsilon$  – *greedy*

$$a_t = \begin{cases} \arg \max_{a \in A} Q_{s_t, a}, & \text{w.p. } 1 - \epsilon. \\ \text{unif}(A), & \text{w.p. } \epsilon. \end{cases} \quad (12)$$

- SARSA:

- update based on the current play  $(s, a, r, s', a')$

$$Q_{k+1} = Q_k + \gamma_k(r + Q_k(s', a') - Q_k(s, a)) \quad (13)$$

- Similar to Q-learning but is On-policy



# Deep Q-Networks

- ▶ Drawback of Q-learning and SARSA:

- Q-table can be too big if environment is complicate i.e.  $10^6 \times 10^3$  maze

- ▶ Alternative Algorithm: DQN:

- Use a function approximator to estimate action-value function with Q-Network

- ▶ Steps:

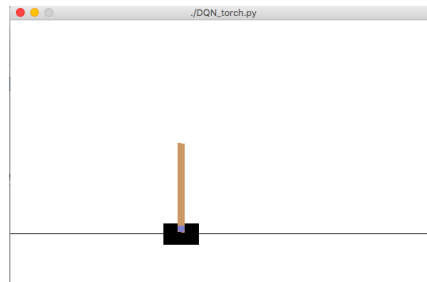
- store transition  $(s_t, a_t, r_{t+1}, s_{t+1})$  in memory
- sample mini-batch of transitions, optimise MSE between Q-network and Q-learning targets:

$$\text{minimize } L_w = E_{s,a,r,s'}[(r + \gamma \max_{a'} Q(s', a'; w^-) - Q(s, a; w))^2] \quad (14)$$

- Important tricks: experience replay and fixed target



# Deep Q-Networks(DQN): play games in OpenAI gym

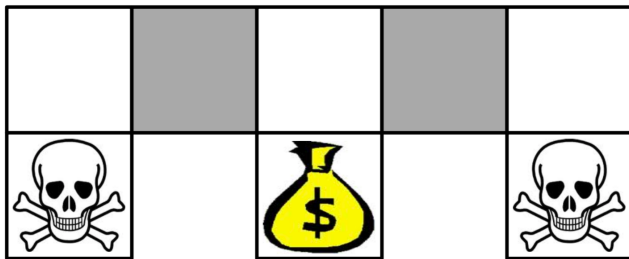


- ▶ States are represented by 4-element tuples (position, cart velocity, angle, tip velocity)
- ▶ Actions can be either moving left or right
- ▶ Function approximator is a feed forward neural network
- ▶ 1 hidden layer with 10 neurons, 2 output neurons representing value estimation for two actions
- ▶ Implemented using torch and tensorflow, can stay alive for 1 minute



# Policy based method

What's wrong with value based methods?



- ▶ Main problem: deterministic policy
- ▶ No good for partially observable environment
- ▶ The agent cannot differentiate the grey states (Horizontally symmetric)
- ▶ An optimal deterministic will either go left or right
- ▶ In this case, if change starting point, the agent may not reach optimal:run



# Policy Gradient: problem formulation

- ▶ Policy is a function of observation:  $\pi_{\theta}(\cdot)$
- ▶ Trajectory  $\tau$ :  $\{s_0, a_0, r_0, s_1, a_1, r_1, \dots\}$  is treated as random variable
- ▶ Distribution of  $\tau$  is determined by policy  $\pi_{\theta}$
- ▶ For each trajectory, total reward is defined as  $R(\tau)$
- ▶ Ultimate goal: optimize expectation  $E_{\pi_{\theta}}[R(\tau)]$  w.r.t  $\theta$





# Policy Gradient: approximate the gradient

- ▶ What does the gradient look like?

$$\begin{aligned}\nabla_{\theta} E_{\pi_{\theta}}[R(\tau)] &= \nabla_{\theta} \sum_{\tau} P_{\theta}(\tau) R(\tau) \\ &= \sum_{\tau} \nabla_{\theta} P_{\theta}(\tau) R(\tau) \\ &= \sum_{\tau} P_{\theta}(\tau) \frac{\nabla_{\theta} P_{\theta}(\tau)}{P_{\theta}(\tau)} R(\tau) = E_{\pi_{\theta}}[\nabla_{\theta} \ln P_{\theta}(\tau) R(\tau)]\end{aligned}\tag{15}$$

- ▶ Equation 15 tell us: gradient can be represented as an expectation
- ▶ Why it is important: expectation can be approximated by sampling



# Policy Gradient: approximate the gradient

- Why the gradient even exists?

$$P(\tau) = P(s_0) \prod_{i=0}^{\infty} \pi_{\theta}(a_i, s_i) P(s_{i+1} | s_i, a_i) \quad (16)$$

- Assumption: there is an underlying MDP specifying  $P(s_{i+1} | s_i, a_i)$  and  $P(s_0)$

$$\begin{aligned} \nabla_{\theta} \ln P(\tau) &= \nabla_{\theta} \ln [P(s_0) \prod_{i=0}^{\infty} \pi_{\theta}(a_i, s_i) P(s_{i+1} | s_i, a_i)] \\ &= \nabla_{\theta} \ln P(s_0) + \nabla_{\theta} \sum_{i=0}^{\infty} [\ln \pi_{\theta}(a_i, s_i) + \ln P(s_{i+1} | s_i, a_i)] \\ &= \nabla_{\theta} \sum_{i=0}^{\infty} \ln \pi_{\theta}(a_i, s_i) \end{aligned} \quad (17)$$



# Policy Gradient: understanding the formula

Combine all equation in previous slides, one important formula:

$$\nabla_{\theta} E_{\pi_{\theta}}[R(\tau)] = E_{\pi_{\theta}}[R(\tau) \nabla_{\theta} \sum_{s_i, a_i \in \tau} \ln \pi_{\theta}(a_i, s_i)] \quad (18)$$

Intuition from equation 18, adjustment magnitude of policy on  $\pi_{\theta}(a, s)$ :

- ▶ In proportion to the total reward gained from trajectories containing  $(a, s)$ 
  - ▶ Rationale: good actions lead to good trajectories, while bad actions lead to bad ones
  - ▶ What about good actions in trajectories with bad overall performance? work on it later
- ▶ In inverse proportion to the probability of performing action  $a$  on state  $s$ 
  - ▶ consider actions sampled frequently but with small positive effect
  - ▶ mitigate case where 'not-so-good' actions are rewarded frequently

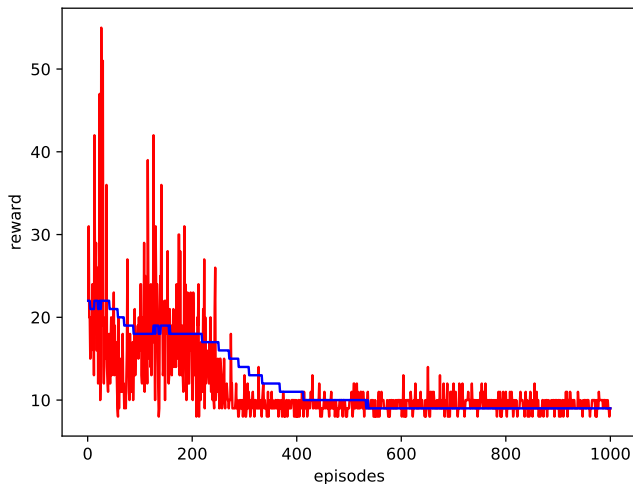


# Vanilla Policy Gradient: **REINFORCE**

So far, we obtain the first policy gradient algorithm called **REINFORCE** [Williams, R. J.]

Experiment on CartPole game:

Bad performance, worse than random play after 1000 episodes of training



# Improvement for **REINFORCE**: baseline

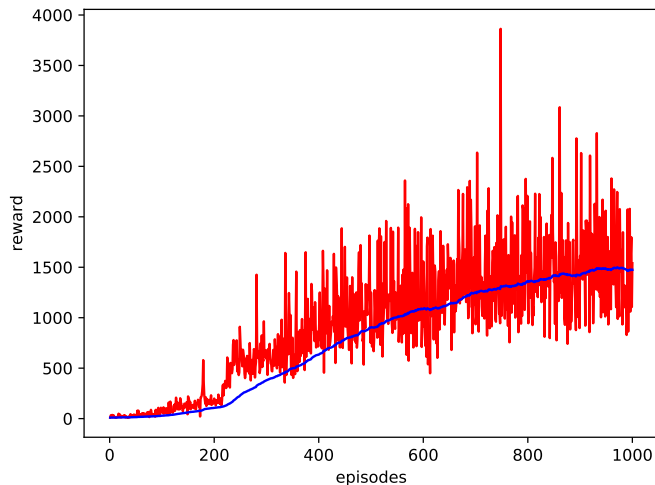
- ▶ One natural question: what if reward is always positive?
- ▶ Do we have to always increase  $\pi_{\theta}(a, s)$  because  $R(\tau)$  is positive (as in formula 18)?
- ▶ Actually, we only care about the relative performance of trajectories
- ▶ Observation from formula 19: we can remove any constant term  $A$  from the expectation without introducing bias.
- ▶  $A$  can be the average performance for all trajectories, it is referenced as a baseline

$$\begin{aligned} E_{\pi_{\theta}} \left[ \sum_a A \cdot \nabla \ln \pi(a, S) \right] &= \sum_a \pi_{\theta}(a, S) A \frac{\nabla_{\theta} \pi_{\theta}(a, S)}{\pi_{\theta}(a, S)} \\ &= A \sum_a \nabla_{\theta} \pi(a, S) \\ &= A \cdot \nabla_{\theta} \sum_a \pi_{\theta}(a, S) = A \nabla_{\theta} (1) = 0 \end{aligned} \tag{19}$$



# Improvement for **REINFORCE**: baseline

Experiment on CartPole game:  
Better than before: an upgoing trend of rewards



# Improvement for **REINFORCE**: advantage function

- ▶ Why we award/punish an action  $(s,a)$  based on the entire trajectory reward?
- ▶ Markov property: the action  $a_t$  only affects rewards after time  $t$ .

$$E_{\pi_{\theta}}[R_{0:i-1}(\tau) \nabla_{\theta} \ln \pi_{\theta}(a_i, s_i)] = 0 \quad (20)$$

- ▶ Actually, we can exploit the markov property to refine formula 18

$$\begin{aligned} \nabla_{\theta} E_{\pi_{\theta}}[R(\tau)] &= E_{\pi_{\theta}}[\sum (\nabla_{\theta} \ln \pi_{\theta}(a_i, s_i)(R_{0:i-1}(\tau) + R_{i:\infty}(\tau)))] \\ &= E_{\pi_{\theta}}[\sum (\nabla_{\theta} \ln \pi_{\theta}(a_i, s_i) R_{i:\infty}(\tau))] \end{aligned} \quad (21)$$

- ▶ Recall that subtraction of baseline doesn't change the expectation

$$\nabla_{\theta} E_{\pi_{\theta}}[R(\tau)] = E_{\pi_{\theta}}[\sum \nabla_{\theta} \ln \pi_{\theta}(a_i, s_i)(R_{i:\infty}(\tau) - V_{\pi_{\theta}}(s_i))] \quad (22)$$



# Improvement for **REINFORCE**: advantage function

- ▶ As shown in equation 22,  $R_{i:\infty}(\tau) - V_{\pi_\theta}(s_i)$  is the actual term that determines the magnitude we adjust probability  $\pi_\theta(a_i, s_i)$
- ▶  $R_{i:\infty}(\tau) - V_{\pi_\theta}(s_i)$  is also known as the advantage function
- ▶ Rationale: extra reward gained when performing certain action  $a_i$  on state  $s_i$  compared to average reward from that state under policy  $\pi_\theta$

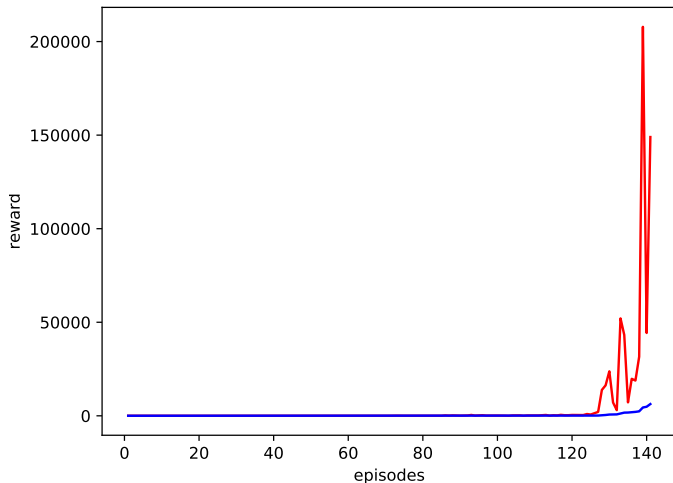




# Improvement for **REINFORCE**: advantage function

Experiment on CartPole game:

Only after 141 episodes of training, surviving time boosted to 20k!



# Actor-Critic: a combination

So far, we mainly focused on pure value-based and pure policy-based methods ...

- ▶ Value-based: problem of deterministic policy in partially observed environments
- ▶ Policy-based: credit assignment problem (delay between action and reward)
- ▶ Why not combine them?
- ▶ Still use policy function
- ▶ Also adopt an estimator for state values to approximate advantage function
- ▶ Policy updates without delay!



## Further improvement: clipped objective function

- ▶ In previous section, policy gradient method works by computing gradient estimator in form

$$\hat{g} = \hat{E}_t[\nabla_{\theta} \ln \pi_{\theta}(a_t, s_t) \hat{A}_t] \quad (23)$$

- ▶ The estimator  $\hat{g}$  can be obtained by differentiating the objective

$$L^{PG}(\theta) = \hat{E}_t[\ln \pi_{\theta}(a_t, s_t) \hat{A}_t] \quad (24)$$

- ▶ Multiple steps to optimize  $L^{PG}$  on same trajectory: destructively large policy updates [Schulman, John et al.].
- ▶ To improve sample efficiency, they adopt strategy of clipping the surrogate objective function in form  $L^{CPI}(\theta)$ :



## Further improvement: clipped objective function

$$L^{CPI}(\theta) = \hat{E}_t \left[ \frac{\pi_{\theta}(a_t, s_t)}{\pi_{\theta_{old}}(a_t, s_t)} \hat{A}_t \right] \quad (25)$$

- ▶  $\pi_{\theta_{old}}$ : fixed term generated by old policy
- ▶  $\pi_{\theta}$ : current policy being optimized.
- ▶ The ratio  $\frac{\pi_{\theta}(a_t, s_t)}{\pi_{\theta_{old}}(a_t, s_t)}$  is denoted as  $r_t(\theta)$
- ▶  $r_t(\theta)$  measures the difference between current policy and old policy
- ▶ we don't want too big a update step, hence some constraint based on  $r_t(\theta)$
- ▶ In practise we use the gradient of following objective function

$$L^{CLIP}(\theta) = \hat{E}_t [\min(r_t(\theta)\hat{A}_t, \text{clip}(r_t(\theta), 1 - \epsilon, 1 + \epsilon)\hat{A}_t)] \quad (26)$$



# Further improvement: clipped objective function

- ▶ Why  $\hat{E}_t[\frac{\pi_\theta(a_t, s_t)}{\pi_{\theta_{old}}(a_t, s_t)} \hat{A}_t]$  in equation 25?
- ▶ Well justified in "Trust Region Policy Optimization"[Schulman, John et al.]
- ▶ My understanding: a case of importance sampling
- ▶ Importance sampling: adjusted rewards, learn from different policy
- ▶ Trajectories generated from  $\pi_{old}$  are learned multiple times to update a different policy  $\pi$ , through importance sampling
- ▶ More on importance sampling: On a Connection between Importance Sampling and the Likelihood Ratio Policy Gradient [Tang Jie and Pieter Abbeel.]



# Proximal Policy Optimization(PPO): some demo

Test on [OpenAI gym](#) Agents implemented and trained using [Pytorch](#)

For detailed information about task environment, check [this list](#)

- ▶ CartPole-v0: [no training](#) and [trained](#)
- ▶ MountainCar-v0: [no training](#) and [trained](#)
- ▶ LunarLander-v2: [no training](#) and [trained](#)
- ▶ Pendulum-v0: [no training](#) and [trained](#)

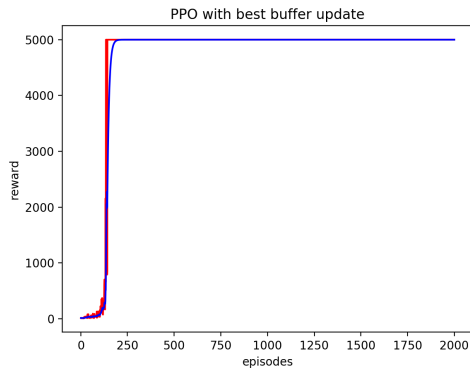
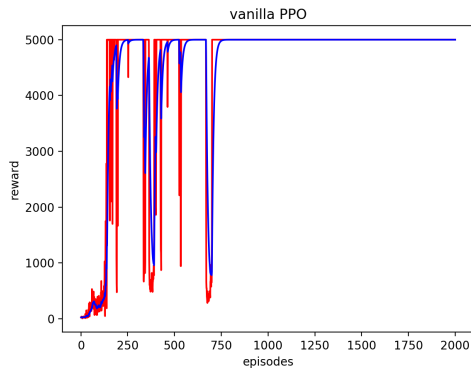
Some strategies in our training:

- ▶ For continous action space (like Pendulum-v0): discretize it
- ▶ Set a maximum number (5000) of steps for each episode during training
- ▶ Use a large batch size (512) to perform gradient descent
- ▶ Adopt different step size for Actor and Critic updates
- ▶ Have a look at [our code](#) on github



# Further improvement: high score buffer replay

- The learning curve is like:



# Reference

- Williams, Ronald J. *Simple statistical gradient-following algorithms for connectionist reinforcement learning*. *Machine learning*, 8(3-4):229-256, 1992.
- Mnih, V., Kavukcuoglu, V., Silver, D., Graves, A., Antonoglou, I., Wierstra, D., Riedmiller, M. (2013). *Playing Atari with Deep Reinforcement Learning*. *NIPS*, arXiv:1312.5602
- Schulman, J., Levine, S., Moritz, P., Jordan, M.I., & Abbeel, P. (2015). *Trust Region Policy Optimization*. *ICML*.
- Tang, J., & Abbeel, P. (2010). *On a Connection between Importance Sampling and the Likelihood Ratio Policy Gradient*. *NIPS*.
- Schulman, J., Wolski, F., Dhariwal, P., Radford, A., & Klimov, O. (2017). *Proximal Policy Optimization Algorithms*. *CoRR*, abs/1707.06347.

Github link for the whole project: [https://github.com/JamesTuna/RL\\_collects](https://github.com/JamesTuna/RL_collects)

