Notes In Deep Reinforcement Learning

Concepts in Reinforcement Learning

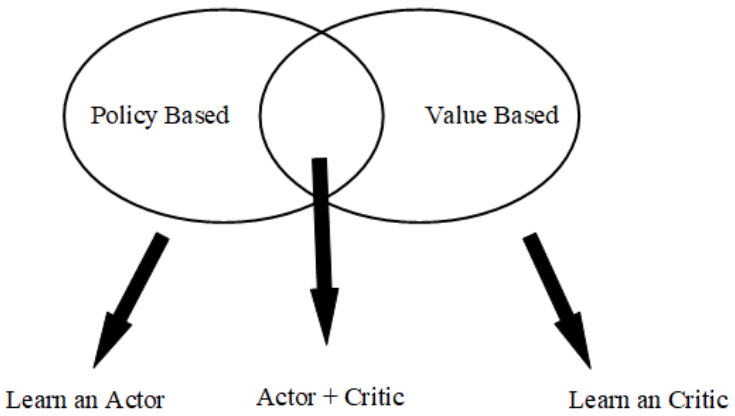
1. The main goal of Reinforcement Learning is to maximum the `Total Reward`.
2. `Total Reward` is the sum of all reward in `One Eposide`, so the model doesn't know which steps in this episode are good and which are bad.
3. Only few actions can get the positive reward (ex: fire and killing the enemy in Space War game gets positive reward but moving gets no reward), so how to let the model find these right actions is very important.

Difficulties in RL

1. Reward Delay
    - Only "Fire" can obtain rewards, but moving before fire is also important (moving has no reward), how to let the model learn to move properly?
    - In chess game, it may be better to sacrifice immediate reward to gain more long-term reward.
2. Agent's actions may affect the environment
    - How to explore the world (`observation`) as more as possible.
    - How to explore the `action-combination` as more as possible.

A3C Method Brief Introduction

The A3C method is the most popular model which combines policy-based method and value-based method, the structure is shown as below. To learn A3C model, we need to know the concepts of `policy-based` and `value-based`. The details of A3C are shown here.



Policy-based Approach - Learn an Actor (Policy Gradient Method)

This approach try to learn a policy(also called actor). It accepts the observation as input, and output an action. The policy(actor) can be any model. If you use an Neural Network to as your actor, then you are doing Deep Reinforcement Learning.

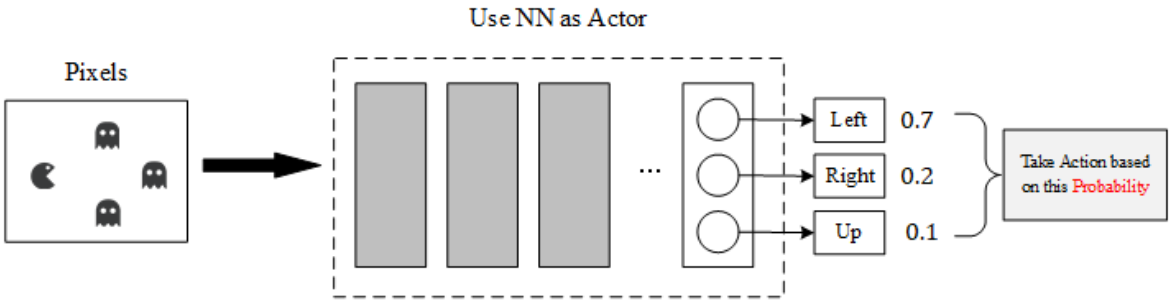
$$Input(Observation) \rightarrow Actor/Policy \rightarrow Output(Action)$$

There are **three steps** to build DRL:

1. Decide Function of Actor Model (NN? ...)

Here we use the NN as our Actor, so:

- The Input of this NN is the observation of machine represented as Vector or Matrix. (Ex: Image Pixels to Matrix)
- The Output of this NN is Action `Probability`. The most important point is that we shouldn't always choose the action which has the highest probability, it should be a stochastic decisions according to the probability distribution.
- The Advantage of NN to Q-table is: we can't enumerate all observations (such as we can't list all pixels' combinations of a game) in some complex scenes, then we can use Neural Network to promise that we can always obtain an output even if this observation didn't appear in the previous train set.



2. Decide Goodness of this Function

Since we use the Neural Network as our function model, we need to decide what is the goodness of this model (a standard to judge the performance of current model). We use $\overline{R(\theta)}$ to express this standard, which θ is the parameters of current model.

- Given an actor $\pi_\theta(t)$ with Network-Parameters θ, t is the observation (input).
- Use the actor $\pi_\theta(t)$ to play the video game until this game finished.
- Sum all rewards in this episode and marked as $R(\theta) \rightarrow R(\theta) = \sum_{t=1}^{T} r_t$.
  `Note`: $R(\theta)$ is a variable, cause even if we use the same actor $\pi_\theta(t)$ to play the same game many times, we can get the different $R(\theta)$ (*random mechanism in game and action chosen*). So we want to maximum the $\overline{R(\theta)}$ which expresses the expect of $R(\theta)$.
- Use the $\overline{R(\theta)}$ to expresses the goodness of $\pi_\theta(t)$.

How to Calculate the $R(\theta)$?

- An episode is considered as a trajectory τ
  - $\tau = \{s_1, a_1, r_1, s_2, a_2, r_2, \ldots, s_T, a_T, r_T\} \rightarrow$ *all the history in this episode*
  - $R(\tau) = \sum_{t=1}^{T} r_t$
- Different τ has different probability to appear, the probability of τ is depending on the parameter θ of actor $\pi_\theta(t)$. So we define the probability of τ as $P(\tau|\theta)$.

$$\overline{R(\theta)} = \sum_{\tau} P(\tau|\theta) R(\tau)$$

- We use actor $\pi_\theta(t)$ to play N times game, obtain the list $\{\tau^1, \tau^2, \ldots, \tau^N\}$. Each τ^n has a reward $R(\tau^n)$, the mean of these $R(\tau^n)$ approximate equals to the expect $\overline{R(\theta)}$.

$$\overline{R(\theta)} \approx \frac{1}{N}\sum_{n=1}^{N} R(\tau^n)$$

3. Choose the best function

Now we need to know how to calculate the θ, here we use the `Gradient Ascend` method.

- problem statements:

$$\theta^* = argmax_\theta \overline{R(\theta)} \rightarrow \overline{R(\theta)} = \sum_\tau P(\tau|\theta)R(\tau)$$

- gradient ascent:
  - Start with θ^0.
  - $\theta^1 = \theta^0 + \eta \bigtriangledown \overline{R(\theta^0)}$
  - $\theta^2 = \theta^1 + \eta \bigtriangledown \overline{R(\theta^1)}$
  - ...
- The θ includes the parameters in the current Neural Network, $\theta = \{ w_1, w_2, w_3, \ldots, b_1, b_2, b_3, \ldots \}$, which the $\bigtriangledown R(\theta) = \begin{bmatrix} \frac{\partial R(\theta)}{\partial w_1} \\ \frac{\partial R(\theta)}{\partial w_2} \\ \cdots \\ \frac{\partial R(\theta)}{\partial b_1} \\ \frac{\partial R(\theta)}{\partial b_2} \\ \cdots \end{bmatrix}$.

It's time to calculate the gradient of $R(\theta) = \sum_\tau P(\tau|\theta)R(\tau)$, since $R(\tau)$ has nothing to do with θ, the gradient can be expressed as:

$$\bigtriangledown R(\theta) = \sum_\tau R(\tau) \bigtriangledown P(\tau|\theta) = \sum_\tau R(\tau)P(\tau|\theta)\frac{\bigtriangledown P(\tau|\theta)}{P(\tau|\theta)} = \sum_\tau R(\tau)P(\tau|\theta) \bigtriangledown logP(\tau|\theta)$$

`Note`: $\frac{dlog(f(x))}{dx} = \frac{1}{f(x)}\frac{df(x)}{dx}$

Use θ policy play the game N times, obtain $\{\tau_1, \tau_2, \tau_3, \ldots\}$:

$$\bigtriangledown R(\theta) \approx \frac{1}{N}\sum_{n=1}^{N} R(\tau^n) \bigtriangledown logP(\tau|\theta)$$

How to Calculate the $\bigtriangledown logP(\tau|\theta)$?

Since τ is the history of one episode, so:

$$\begin{aligned} P(\tau|\theta) &= P(s_1)P(a_1|s_1,\theta)P(r_1,s_2|s_1,a_1)P(a_2|s_2,\theta)\ldots \\ &= P(s_1)\prod_{t=1}^{T} P(a_t|s_t,\theta)P(r_t,s_{t+1}|s_t,a_t)logP(\tau|\theta) \\ &= logP(s_1) + \sum_{t=1}^{T} logP(a_t|s_t,\theta) + logP(r_t,s_{t+1}|s_t,a_t) \end{aligned}$$

Ignore the terms which not related to θ:

$$\bigtriangledown logP(\tau|\theta) = \sum_{t=1}^{T} \bigtriangledown logP(a_t|s_t,\theta)$$

So the final result of $\bigtriangledown\overline{R(\theta)}$ is :

$$\nabla\overline{R(\theta)} = \frac{1}{N}\sum_{n=1}^{N}\sum_{t=1}^{T} R(\tau^n) \nabla logP(a_t^n|s_t^n,\theta)$$

The meaning of this equation is very clear:

- if $R(\tau^n)$ is positive $\rightarrow$ tune θ to increase the $P(a_t^n|s_t^n)$.
- if $R(\tau^n)$ is negative $\rightarrow$ tune θ to decrease the $P(a_t^n|s_t^n)$

Use this method can resolve the Reward Delay Problem in **Difficulties in RL** chapter, because here we use the `cumulative reward` of one entire episode $R(\tau^n)$, not just the immediate reward after taking one action.

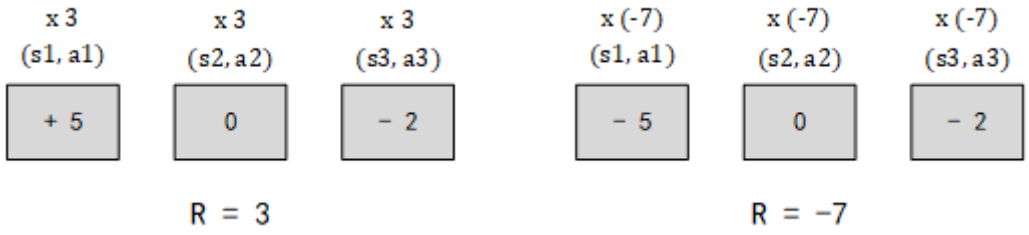
Add a Baseline - b

To avoid all of $R(\tau^n)$ is positive (*there should be some negative reward to tell model don't take this action at this state*), we can add a baseline. So the equation changes to:

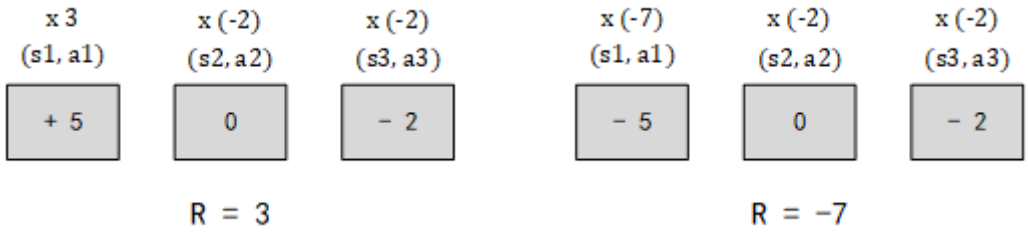
$$\nabla\overline{R(\theta)} = \frac{1}{N}\sum_{n=1}^{N}\sum_{t=1}^{T} (R(\tau^n) - b) \nabla logP(a_t^n|s_t^n,\theta)$$

Assign Suitable Weight of each Action

Use the total reward $R(\tau)$ to tune the all actions' probability in this episode also has some disadvantage, show as below:



The left picture show one episode whose total reward R is 5, so the probabilities of all actions in this episode will be increased (such as x5), but the main positive reward obtained from the a_1, while a_2 and a_3 didn't give any positive reward, but the probability of a_2 and a_3 also be increased in this example. Same as right picture, a_1 is a bad action, but a_2 may not be a bad action, so probability of a_2 shouldn't be decreased.



To avoid this problem, we assign different R to each a_t, the R is the cumulation of r_t which is the sum of all rewards obtained after a_t, now the equation becomes:

$$\nabla\overline{R(\theta)} = \frac{1}{N}\sum_{n=1}^{N}\sum_{t=1}^{T} (\sum_{t'=t}^{T}\gamma^{t'-t}r_{t'}^n - b) \nabla logP(a_t^n|s_t^n,\theta)$$

`Note`: γ called discount factor, $\gamma < 1$.

We can use $A^{\theta}(s_t, a_t)$ to express the $(\sum_{t'=t}^{T} \gamma^{t'-t} r_{t'}^{n} - b)$ in above equation, which called `Advantage Function`. This function evaluate how good it is if we take a_t at this state s_t rather than other actions.

On-Policy v.s. Off-Policy

On-Policy and Off-Policy are two different modes of learning:

- On-Policy: The agent learn the rules by `interacting` with environment. (*learn from itself*)
- Off-Policy: The agent learn the rules by `watching` others' interacting with environment. (*learn from others*)

Our Policy Gradient Method is an On-Policy learning mode, so why we need Off-Policy mode? This is because we use sampling N times and get the mean value to approximate the expect $\overline{R(\theta)} = \sum_{\tau} P(\tau|\theta)R(\tau)$. But when we update the θ, the $P(\tau|\theta)$ changed, so we need to do N sampling again and get the mean value. This will take a lot of time to do sampling after we update θ. The resolution is, we build a model π_{θ}, this model accept the training data from the other model $\pi_{\theta'}$. Use $\pi_{\theta'}$ to collect data, and train the θ with θ', since don't change θ', the sampling data can be reused.

Importance Sampling (On-Policy $\rightarrow$ Off-Policy)

Importance Sampling is a method to get the expect of one function $E_{x\sim p}(p(x))$ by sampling another function $q(x)$. Since we have already known:

$$E_{x\sim p}[f(x)] \approx \frac{1}{N}\sum_{i=1}^{N} f(x^i)$$

But if we only have $\{x^i\}$ sampled from $q(x)$, how to use this samples to calculate the $E[p(x)]$? We can change equation above:

$$E_{x\sim p}[f(x)] = \int p(x)f(x)dx = \int f(x)\frac{p(x)}{q(x)}q(x)dx = E_{x\sim q}[f(x)\frac{p(x)}{q(x)}]$$

That means we can get the expect of distribution $p(x)$ by sampling the $\{x^i\}$ from another distribution $q(x)$, only need to do some rectification, $\frac{p(x)}{q(x)}$ called rectification term. Now we can consider our π_{θ} model as $p(x)$, the $\pi_{\theta'}$ as $q(x)$, use the $q(x)$ to sample data to tune $p(x)$.

$$\bigtriangledown\overline{R(\theta)} = E_{\tau\sim p_{\theta}(\tau)}[R(\tau) \bigtriangledown logp_{\theta}(\tau)] = E_{\tau\sim p_{\theta'(\tau)}}[\frac{p_{\theta}(\tau)}{p_{\theta'}(\tau)}R(\tau) \bigtriangledown logp_{\theta}(\tau)]$$

then we can use θ' to sample many times and train θ many times. After many iterations, we update θ'. Continue to transform the equation:

$$E_{(s_t,a_t)\sim\pi_{\theta}}[A^{\theta}(s_t,a_t) \bigtriangledown logp_{\theta}(a_t^n|s_t^n)] = E_{(s_t,a_t)\sim\pi_{\theta'}}[\frac{P_{\theta}(s_t,a_t)}{P_{\theta'}(s_t,a_t)}A^{\theta'}(s_t,a_t) \bigtriangledown logp_{\theta}(a_t^n|s_t^n)]$$

Let the $P_{\theta'}(s_t,a_t) = P_{\theta'}(a_t|s_t)P_{\theta'}(s_t)$, and $P_{\theta}(s_t,a_t) = P_{\theta}(a_t|s_t)P_{\theta}(s_t)$. We consider the environment observation s is not related to actor θ (*ignore the environment changing by action*), then $P_{\theta}(s_t) = P_{\theta'}(s_t)$, equation becomes:

$$E_{(s_t,a_t)\sim\pi_{\theta'}}[\frac{P_{\theta}(a_t|s_t)}{P_{\theta'}(a_t|s_t)}A^{\theta'}(s_t,a_t) \bigtriangledown logp_{\theta}(a_t^n|s_t^n)]$$

Here defines:

$$J^{\theta'}(\theta) = E_{(s_t,a_t)\sim\pi_{\theta'}}[\frac{P_\theta(a_t|s_t)}{P_{\theta'}(a_t|s_t)}A^{\theta'}(s_t,a_t)]$$

`Note`: Since we use θ' to sample data for θ, the distribution of θ can't be very different from θ', how to determine the difference between two distribution and end the model training if θ' is distinct from θ? Now let's start to learn PPO Algorithm.

PPO Algorithm —— Proximal Policy Optimization

PPO is the resolution of above question, it can avoid the problem which raised from the difference between θ and θ' . The target function shows as below:

$$J^{\theta'}_{PPO}(\theta) = J^{\theta'}(\theta) - \beta KL(\theta,\theta')$$

which the $KL(\theta,\theta')$ is the divergence of output action from policy θ and policy θ'. The algorithm flow is:

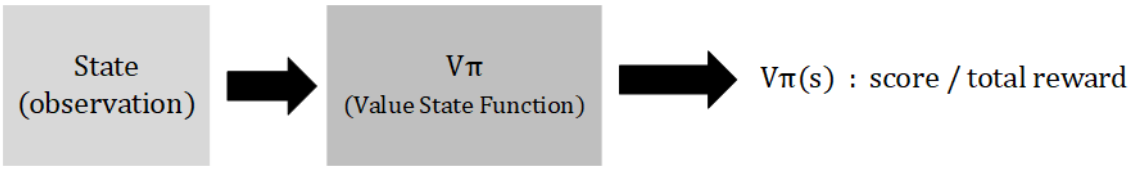
- Initial Policy parameters θ
- In each iteration:
  - Using θ^k to interact with the environment, and collect $\{s_t, a_t\}$ to calculate the $A^{\theta^k}(s_t,a_t)$
- Update the $J^{\theta'}_{PPO}(\theta)$ **several** times: $J^{\theta^k}_{PPO}(\theta) = J^{\theta^k}(\theta) - \beta KL(\theta,\theta^k)$
  - If $KL(\theta,\theta^k) > KL_{max}$, that means KL part takes too big importance of this equation, increase β
  - If $KL(\theta,\theta^k) < KL_{min}$, that means KL part takes lower importance of this equation, decrease β

Value-based Approach - Learn an Critic

A critic doesn't choose an action (*it's different from actor*), it `evaluates the performance` of a given actor. So an actor can be found from a critic.

Q-learning

Q-Learning is a classical value-based method, it evaluates the score of an observation under an actor π, this function is called `state value function` $V^\pi(s)$. The score is calculated as the total reward from current observation to the end of this episode.



How to estimate $V^\pi(s)$?

We know we need to calculate the total reward to express the performance of current actor π_θ, but how to get this value?

- Monte-Carlo based approach

In the current state S_a (observation), until the end of this episode, the cumulated reward is G_a; In the current state S_b (observation), until the end of this episode, the cumulated reward is G_b. That means we can estimate the value of an observation s_a under an actor π_θ, the low value could be explain as two possibilities:

a) the current observation is bad, even if a good actor can not get a high value.

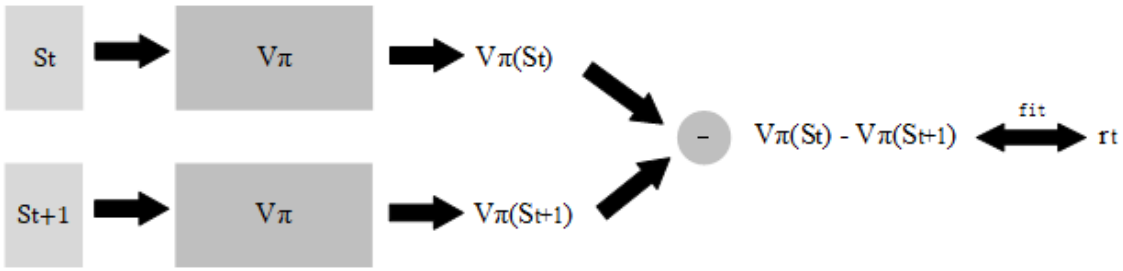
b) the actor has a bad performance.

In many cases, we can't enumerate all observations to calculate the all rewards G_i. The resolution is using a Neural-Network to fit the function from observation to value G.

Fit the NN with $(S_a, G(a))$, try to minimize the difference between the NN output $V_\pi(S_a)$ and Monte-Carlo reward $G(a)$.

- Temporal-Difference approach

MC approach is worked, but the problem is you must get the total reward in the end of one episode. It may be a very long way to reach the end state in some cases, Temporal-Difference approach could address this problem.



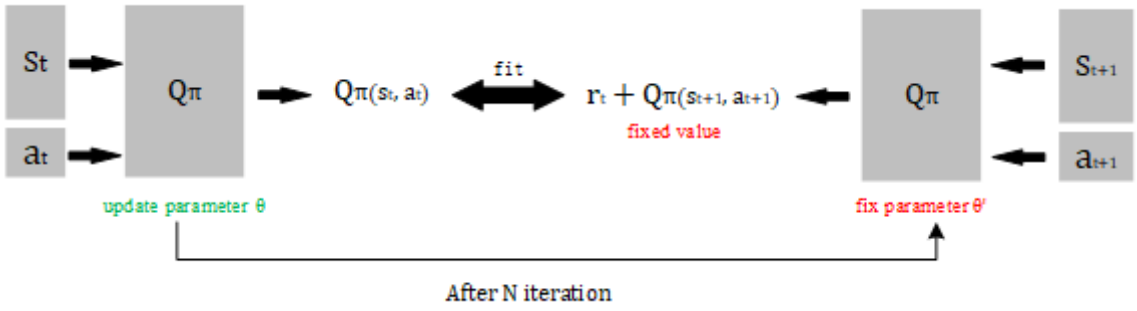
here is a trajectory $\{\ldots, s_t, a_t, r_t, s_{t+1}, \ldots\}$, there should be:

$$V^\pi(s_t) = V^\pi(s_{t+1}) + r_t$$

so we can fit the NN by minimize the difference between $V^\pi(s_t) - V^\pi(s_{t+1})$ and r_t.

Here is a tip in practice: we are training the same model V^π, so the two outputs $V_\pi(s_t)$ and $V_\pi(s_{t+1})$ are all generate from one parameter group θ. When we update the θ after one iteration, both $V_\pi(s_t)$ and $V_\pi(s_{t+1})$ are changed in next iteration, which makes the model unstable.

The tip is: fix the parameter group θ' to generate the $V_\pi(S_{t+1})$, and update the θ for $V_\pi(S_t)$. After N iterations, let the θ' equal to θ. Fixed parameter Network (right) is called Target Network.



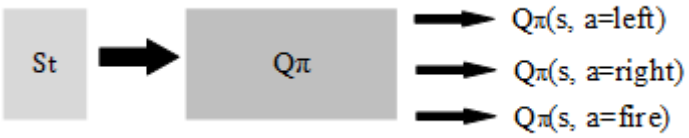
- MC v.s. TD

- Monte-Carlo has larger `variance`. This is caused by the randomness of $G(a)$, since $G(a)$ is the sum of all reward r_t, each r_t is a random variable, the sum of these variable must have a larger variance. Playing N times of one game with the same policy, the reward set $\{G(a), G(b), G(c), \ldots\}$ has a large variance.
- Temporal-Difference also has a problem, which is $V^{\pi}(s_{t+1})$ may estimate `incorrectly` (cause it's not like Monte-Carlo approach to cumulative the reward until the end of this episode), so even the r_t is correct, the $V^{\pi}(s_t) - V^{\pi}(s_{t+1})$ may not correct.

In the practice, people prefer to use TD method.

- Q-value approach $\rightarrow Q^{\pi}(s, a)$

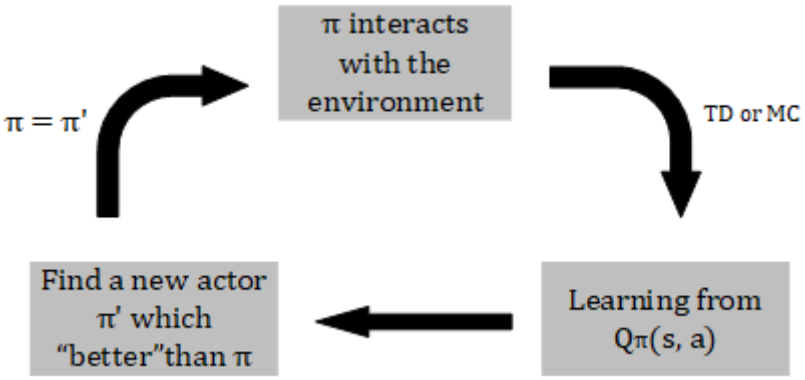
In current state (observation), enumerate all valid actions and calculate the Q-value of each action.



`note`: In current state we force the actor to take the specific action to calculate the value this action, but random choose actions according to the π_θ actor in next steps until the end of episode.

Use Q-value to learn an actor

We can learn an actor π with the Q-value function, here is the algorithm flow:



the question is: how to estimate the π' is better than π?

If π' is better than π, then:

$$V^{\pi'}(s_i) \geqslant V^{\pi}(s_i), \qquad \forall s_i \in S$$

We can use equation below to calculate the π' from π:

$$\pi'(s) = argmax_a Q^{\pi}(s, a)$$

`Note`: This approach not suitable for continuous action, only for **discrete action**.

But if we always choose the best action according to the Q^{π}, some other better actions we can never detect. So we infer use `Exploration` method when we choose action to do.

Epsilon Greedy

Set a probability ε, take max Q-value action or take random action show as below. Typically , ε decreases as time goes by.

$$a = \begin{cases} argmax Q(s,a), & \text{with probability 1 - } \varepsilon \\ random, & \text{with probability } \varepsilon \end{cases}$$

Boltzmann Exploration

Since the Q^{π} is an Neural Network, the output of this Network is the probability of each action. Use this probability to decide which action should take, show as below:

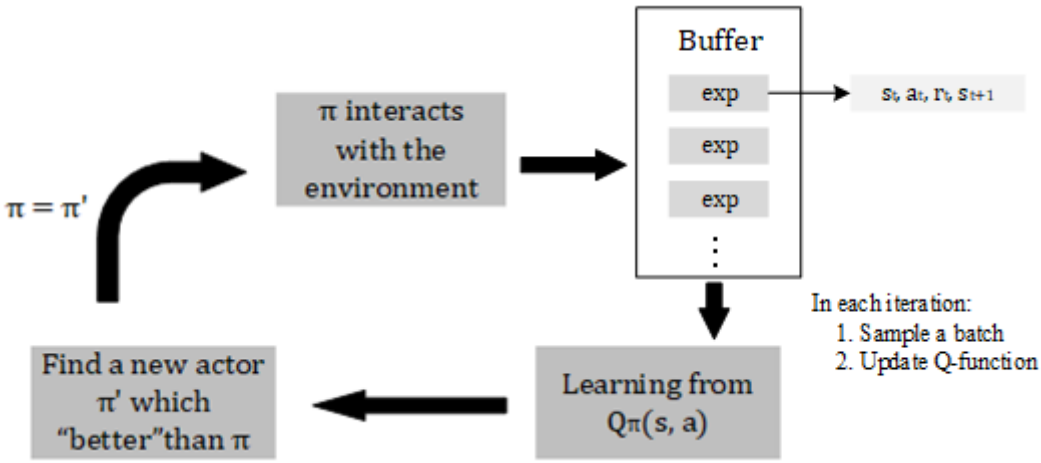
$$P(a_i|s) = \frac{exp(Q(s,a_i))}{\sum_a exp(Q(s,a))}$$

Q-value may be negative, so we take exp-function to let them be positive.

Replay Buffer

Replay buffer is a buffer which stores a lot of *experience* data. When you train your Q-Network, random choose a batch from buffer to fit it.

- An experience is a set which looks like $\{s_t, a_t, r_t, s_{t+1}\}$.
- The experience in buffer may comes from different policy $\{\pi_{\theta_1}, \pi_{\theta_2}, \pi_{\theta_3}, \ldots\}$.
- Drop the old experience when buffer is full.



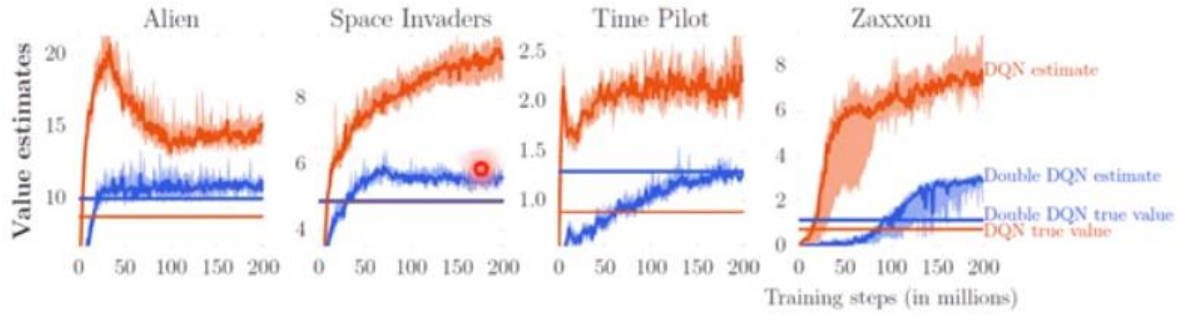
Typical Q-Learning Algorithm

Here is the main algorithm flow of Q-learning:

- Initialize Q-function Q, Initialize target Q-function $\hat{Q} = Q$
- in each episode
  - for each step t
    - Given state s_t, take an action a_t based on Q (ε-greedy exploration)
    - Obtain the reward r_t and next state s_{t+1}
    - Store this experience $\{s_t, a_t, r_t, s_{t+1}\}$ into the replay buffer
    - Sample a batch of experience $\{(s_i, a_i, r_i, s_{i+1}), (s_j, a_j, r_j, s_{j+1}), \ldots\}$ from buffer
    - Compute target $y = r_i + max_a \hat{Q}(s_{i+1}, a_)$
    - Update the parameters in Q to make $Q(s_i, a_i)$ close to y.
    - After N steps set $\hat{Q} = Q$

Double DQN

Double DQN is designed to solve the problem of DQN. Problem of DQN show as below:



Q-value are always over estimate in DQN training (Orange curve is DQN Neural Network output reward, Blue curve is Double DQN Neural Network output reward; Orange line is the real cumulative reward of DQN, Blue line is the real cumulative reward of Double DQN). Notes that Blue lines are over than Orange lines which means Double DQN has a greater true value than DQN.

Why DQN always over-estimate Q-value?

This because when we calculate the target y which equals $r_t + max_a Q_\pi(s_{t+1}, a)$, we always choose the best action and compute the highest Q-value. This may over-estimate the target value, so the real cumulative reward may lower than that target value. While Q function is try to close the target value, this results the output of Q-Network is higher than the actual cumulative reward.

$$Q(s_t, a_t) \qquad \Longleftrightarrow \qquad r_t + max_a Q(s_{t+1}, a)$$

Double DQN resolution

To avoid above problem, we use two Q-Network in training, one is in charge of choose the best action and the other is to estimate Q-value.

$$Q(s_t, a_t) \qquad \Longleftrightarrow \qquad r_t + Q'(s_{t+1}, argmax_a Q(s_{t+1}, a))$$

Here use Q to select the best action in each state but use Q' to estimate the Q-value of this action. This method has two advantages:

- If Q over-estimate the Q-value of action a, although this action is selected, the final Q-value of this action won't be over estimated (because we use Q' to estimate the Q-value of this action).
- If Q' over-estimate one action a, it's also safe. Because the Q policy won't select the action a (because a is not the best action in Policy Q).

In DQN algorithm, we already have two Network: `origin Network` θ and `target Network` θ' (need to be fixed). So here use `origin Network` θ to select the action, and `target Network` θ' to estimate the Q-value.

Other Advanced Structure of Q-Learning

- Dueling DQN

Change the output as two parts: $Q^\pi(s_t, a_t) = V(s_t) + A(s_t, a_t)$, which means the final Q-value is the sum of environment value and action value.

- Prioritized Replay

When we sample a batch of experience from replay buffer, we don't use random select. Prioritized Replay marked those experience which has a high loss after one iteration, and increase the probability of selecting those experience in the next batch.

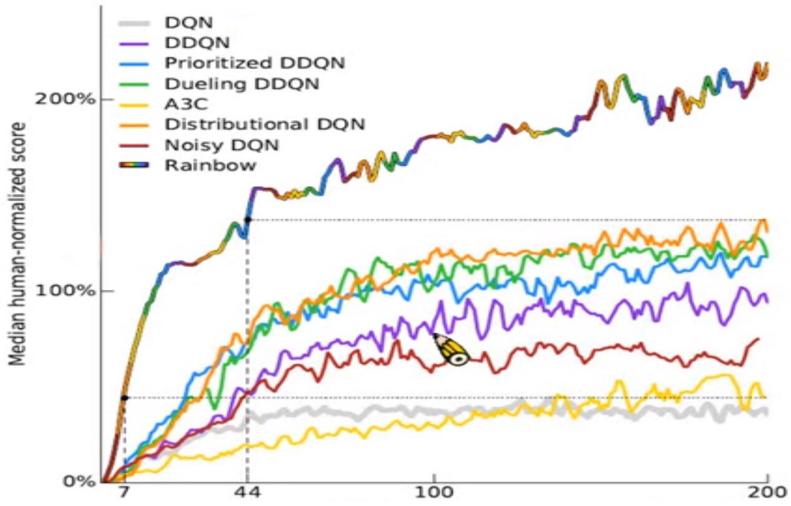
- Multi-Step

Change the experience format in the Replay Buffer, not only store one step $\{s_t, a_t, r_t, s_{t+1}\}$, store N steps { $s_t, a_t, r_t, s_{t+1}, \ldots, s_{t+N}, a_{t+N}, r_{t+N}, s_{t+N+1}$ }.

- Noise Net

This method used to explore more action. Add some noise in current Network Q at the beginning of one episode.

Here is the comparison of different algorithms:

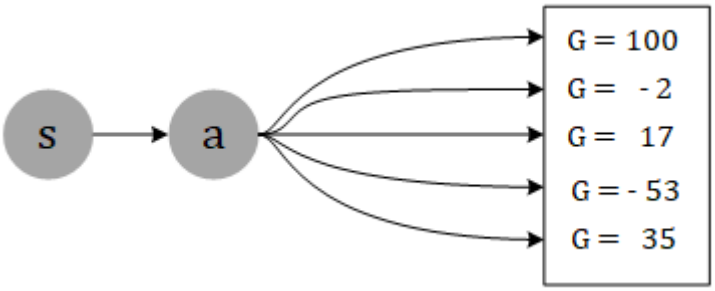


A3C Method - Asynchronous Advantage Actor-Critic

Why we need A3C method? This is designed to solve the variance problem of Policy Gradient. In Policy Gradient method, even in the same state s_t and take the same action a_t N times, we may get very different result total reward G. This because randomness existed when we calculate the cumulative reward in below equation:

$$\bigtriangledown \overline{R(\theta)} = \frac{1}{N}\sum_{n=1}^{N}\sum_{t=1}^{T}(\sum_{t'=t}^{T}\gamma^{t'-t}r_{t'}^{n} - b) \bigtriangledown logP(a_t^n|s_t^n, \theta)$$

This part $(\sum_{t'=t}^{T}\gamma^{t'-t}r_{t'}^{n} - b)$ could be very different for r_t is a random variable with big variance, the result may be like this:



unless we sample enough times to cover all possible rewards, the model could be stable —— but it's hard to do this. If we can replace $\sum_{t'=t}^{T}\gamma^{t'-t}r_{t'}^{n}$ with the expect $E(r_{t'}^{n})$, then we can solve this problem.

Advantage Actor-Critic (A2C Method)

We have already introduced value-based method, the definition of $Q^{\pi_\theta}(s_t, a_t)$ is the expect of total reward of taking action a_t at current state s_t. The definition of $V^{\pi_\theta}(s_t)$ is the expect reward of current state s_t (just the state value without specific which action should take). Now we change the equation:

$$\bigtriangledown\overline{R(\theta)} = \frac{1}{N}\sum_{n=1}^{N}\sum_{t=1}^{T}(Q^{\pi_\theta}(s_t^n, a_t^n) - V^{\pi_\theta}(s_t^n)) \bigtriangledown logP(a_t^n|s_t^n, \theta)$$

`note`: Here we use state value to replace the baseline b.

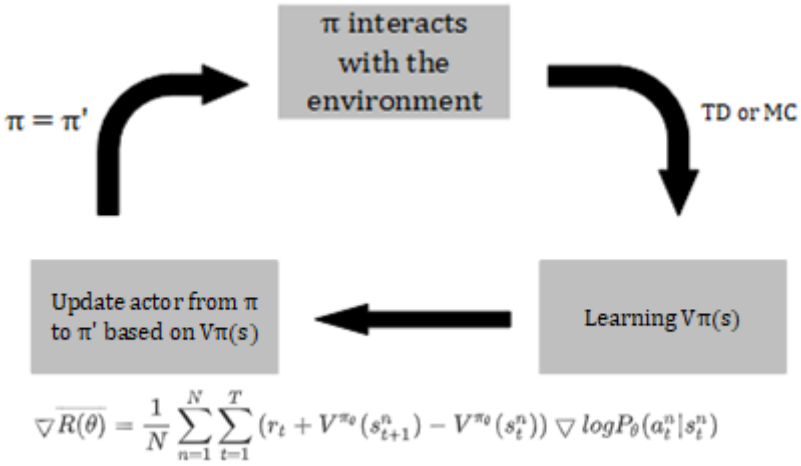
We can infer the value of $Q^{\pi_\theta}(s_t, a_t)$ from $V^{\pi_\theta}(s_t)$:

$$Q^{\pi}(s_t, a_t) \quad = \quad E[r_t + V^{\pi}(s_{t+1})] \quad \rightarrow \quad r_t + V^{\pi}(s_{t+1})$$

We should use the expect because r_t is a random variable, but it's hard to calculate, so we take off the expect. Now the equation becomes:

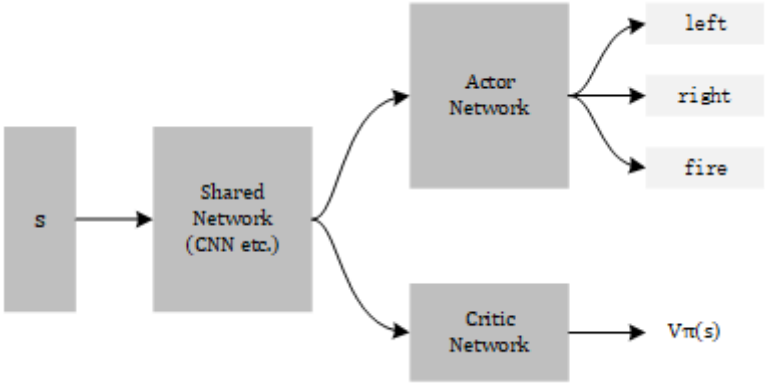
$$\bigtriangledown\overline{R(\theta)} = \frac{1}{N}\sum_{n=1}^{N}\sum_{t=1}^{T}(r_t + V^{\pi_\theta}(s_{t+1}^n) - V^{\pi_\theta}(s_t^n)) \bigtriangledown logP_\theta(a_t^n|s_t^n)$$

Algorithm flow of Advantage Actor-Critic method show as below:



Tips

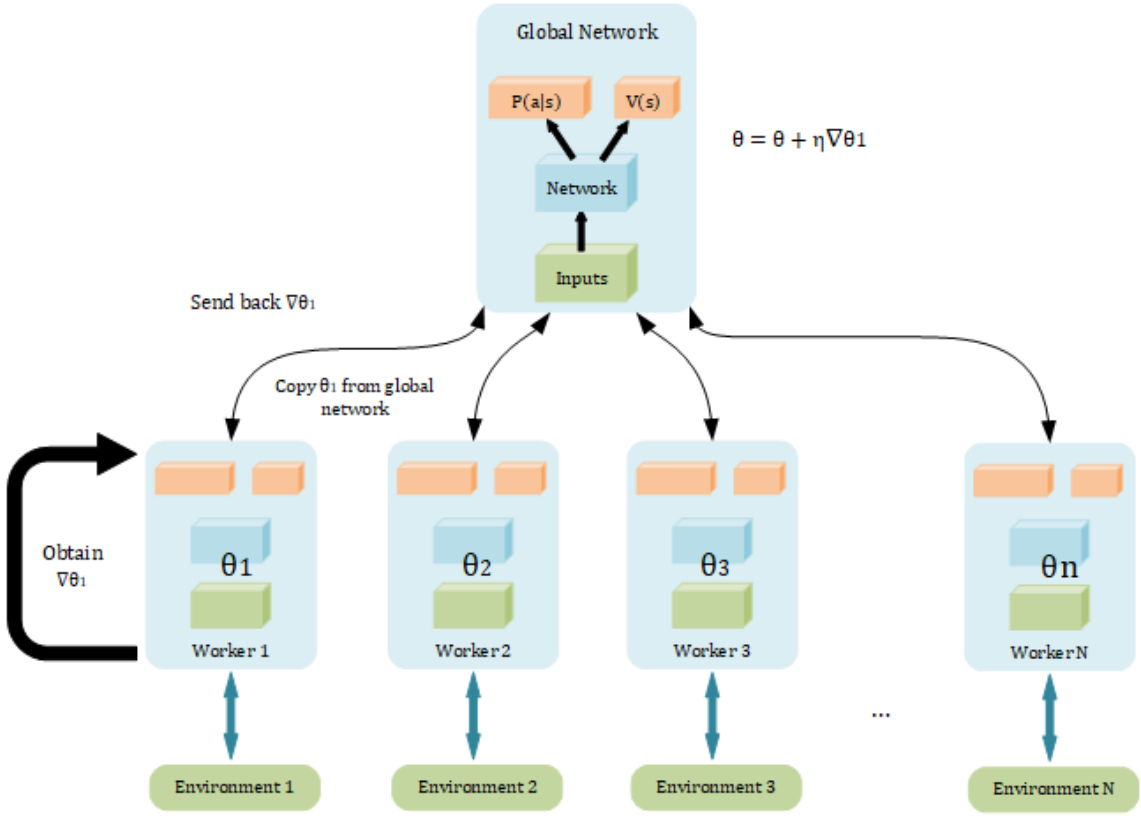
1. There are two Networks to train in this algorithm: Actor π_θ and Critic $V^{\pi}(s)$. But two networks accept the same input s, only different in output —— scaler $V(s)$ for Critic Network and Probability Distribution $P(a|s)$ for Actor Network. So two networks can share some layers in the front of structure, looks like this:



2. Use output entropy as regularization for $\pi(s)$, this could make the probability of each action more even so that the model can do more exploration.

Asynchronous Advantage Actor-Critic (A3C Method)

A3C is designed to speed up A2C. It maintain one global Network and create N workers, each worker interact with different environment, calculate the gradient and update the global Network.



- Copy global parameters θ_1
- Sampling some data
- Compute gradients
- Update global model

`note`: All workers are parallelized, which means when $\bigtriangledown\theta_1$ finish compute and send back to global model, θ may changed (updated by other worker, so it may not remain θ_1). But we still use the $\bigtriangledown\theta_1$ to update the current parameters $\theta \rightarrow \theta + \eta \bigtriangledown \theta_1$.

Sparse Reward

In reinforcement learning, reward is very important for agent to know which actions are good. But only few action could obtain a positive reward (ex. only fire and destroy the enemy could obtain a positive reward in Spaceship game), most of actions have no reward (ex. move left or move right). This phenomenon is called Sparse Reward.

Reward Shaping

Typically, few state could get the positive reward in training, thus we can create some extra rewards to guide the agent do some action in current state. For example, if we want to train an plane agent to destroy the enemy plane, the actual reward should be obtained from "fire and destroy the enemy". But in the start of the game, our plane don't know how to find enemy, so we can create an extra positive reward if our plane is fly toward enemy plane.

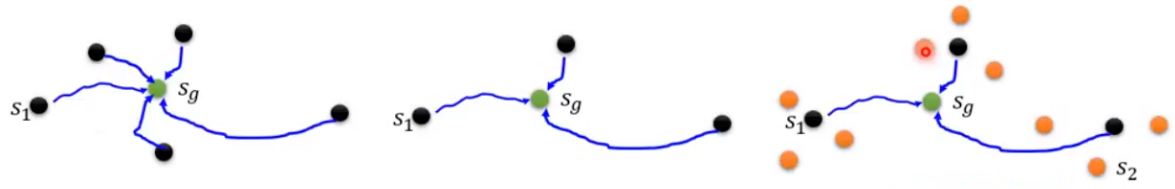
`note`: This method needs domain knowledge to design which rules desire positive reward and how much reward should be assigned.

Curriculum Learning

Typically, a hard task could be split into many simple tasks. Curriculum Learning Algorithm is starting from simple training examples, and then becoming harder and harder.

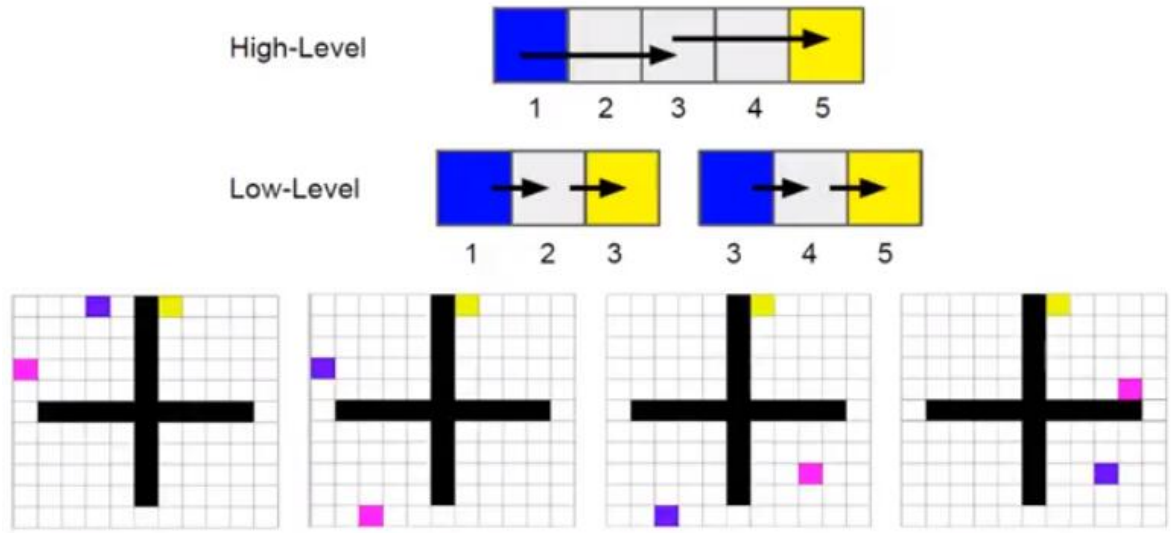
The most common technique is **Reverse Curriculum Learning**, explain as below:

- Given a goal state s_g.
- Sample some states $\{s_1, s_2, \ldots\}$ close to s_g.
- Each state has a reward to goal state s_g, compute $\{R(s_1), R(s_2), \ldots\}$.
- Delete those state whose reward is too large (it's too easy from this state to goal state) or too small (it's too difficult from this state to goal state).
- Sample more states near the $\{s_1, s_2, \ldots\}$.



Hierarchical Reinforcement Learning

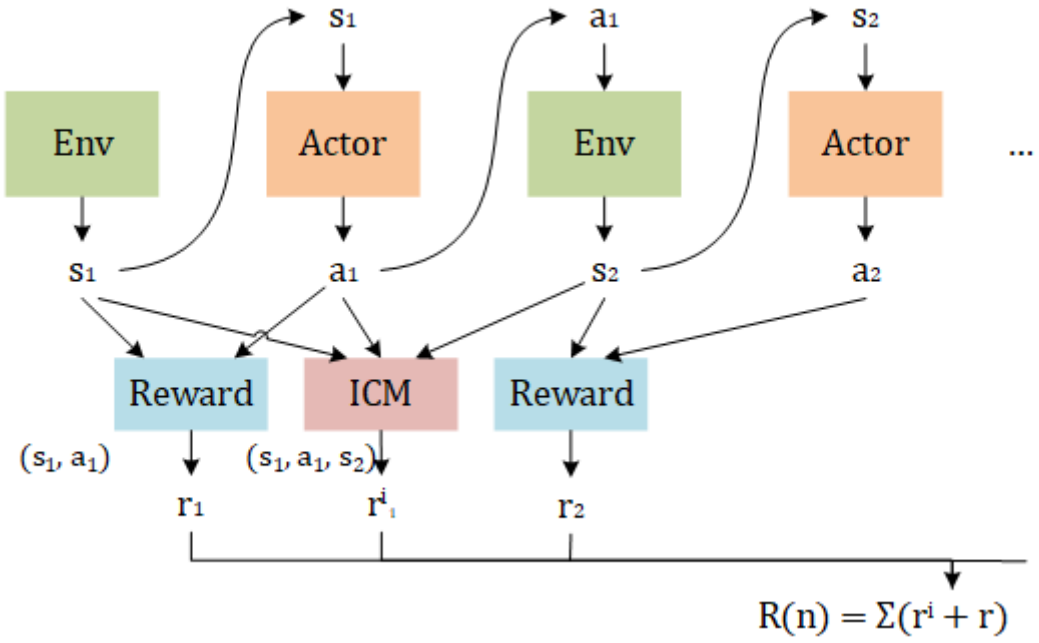
A entire model could be split into different hierarchies, top-level model only give the top-level order and low-level model choose the actual actions. For example, if we wanna train a plane agent, top-level model only give the way point of next target while the low-level model control the plane to fly to that target (turn left or turn right).



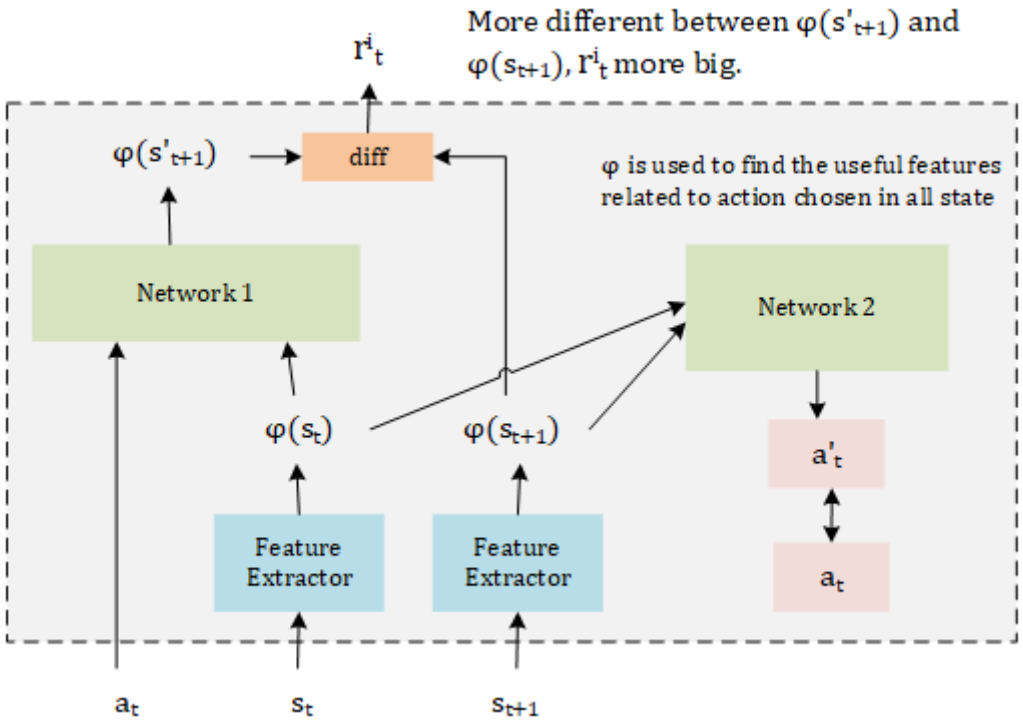
Here is a game example, blue point is the agent which is asked to reach the yellow point. Pink point is the temporary target given by high-level model while the low-level mode follow this instruction and control the agent to reach pink point.

ICM —— Intrinsic Curiosity Module

ICM Algorithm can let model to do more exploration. It adds an extra Reward function r_t^i which accept three parameters (s_t, a_t, s_{t+1}). The Network need to maximize the sum value $\sum_{t=1}^{N}(r_t^i + r_t)$.



Now let's see how ICM calculate reward r_t^i in each step t :



Here are two Networks in ICM module:

- Network 1: This Network is used to predict the next state s_{t+1} after taking action a, if s_{t+1} is hard to predict, then the reward r^i is high.
- Network 2: There may be a lot of features are not related to do actions (ex. the position of sun in the Spaceship War game). If we just maximize the reward of state which hard to predict, then the agent will stay and watch the sun moving. So we need to find the useful features in action chosen, this is the work of Network 2. It predict the action a_t according to s_t and s_{t+1}.