Introduction to Various Reinforcement Learning Algorithms. Part I (Q-Learning, SARSA, DQN, DDPG)

Kung-Hsiang, Huang (Steeve) Jan 12, 2018 · 9 min read ★

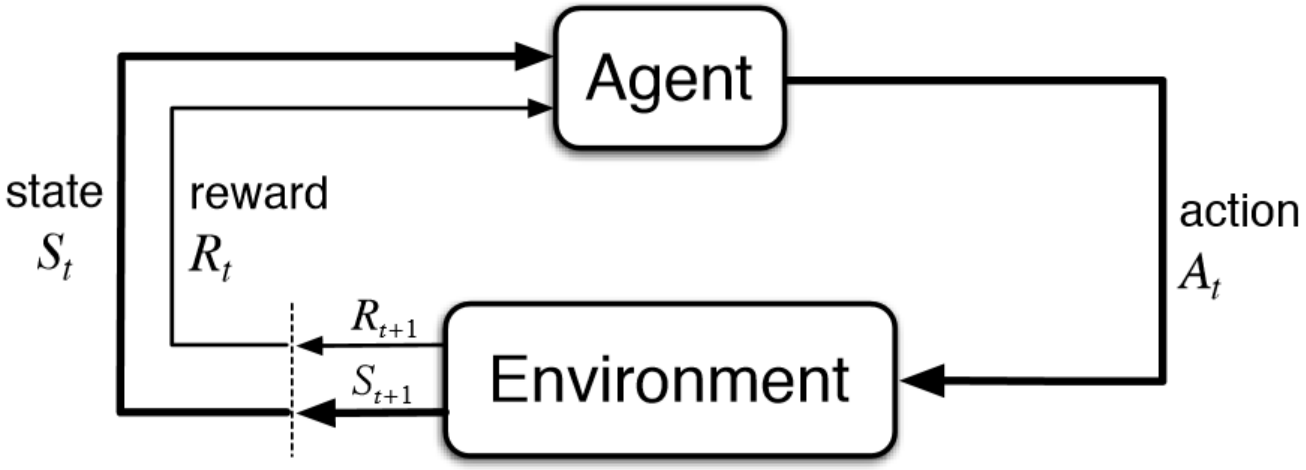
Reinforcement Learning (RL) refers to a kind of Machine Learning method in which the agent receives a delayed reward in the next time step to evaluate its previous action. It was mostly used in games (e.g. Atari, Mario), with performance on par with or even exceeding humans. Recently, as the algorithm evolves with the combination of Neural Networks, it is capable of solving more complex tasks, such as the pendulum problem:

Deep Deterministic Policy Gradient (DDPG) Pendulum OpenAI Gym using Tensorflow

Although there are a great number of RL algorithms, there does not seem to be a comprehensive comparison between each of them. It gave me a hard time when deciding which algorithms to be applied to a specific task. This article aims to solve this problem by briefly discussing the RL setup, and providing an introduction for some of the well-known algorithms.

1. Reinforcement Learning 101

Typically, a RL setup is composed of two components, an agent and an environment.



Reinforcement Learning Illustration (https://i.stack.imgur.com/eoeSq.png)

sending a state to the agent, which then based on its knowledge to take an action in response to that state. After that, the environment send a pair of next state and reward back to the agent. The agent will update its knowledge with the reward returned by the environment to evaluate its last action. The loop keeps going on until the environment sends a terminal state, which ends to episode.

Most of the RL algorithms follow this pattern. In the following paragraphs, I will briefly talk about some terms used in RL to facilitate our discussion in the next section.

Definition

1. Action (A): All the possible moves that the agent can take
2. State (S): Current situation returned by the environment.
3. Reward (R): An immediate return send back from the environment to evaluate the last action.
4. Policy (π): The strategy that the agent employs to determine next action based on the current state.
5. Value (V): The expected long-term return with discount, as opposed to the short-term reward R. $V\pi(s)$ is defined as the expected long-term return of the current state sunder policy π.
6. Q-value or action-value (Q): Q-value is similar to Value, except that it takes an extra parameter, the current action *a*. $Q\pi(s, a)$ refers to the long-term return of the current state *s*, taking action *a* under policy π.

Model-free v.s. Model-based

The model stands for the simulation of the dynamics of the environment. That is, the model learns the transition probability $T(s1 | (s0, a))$ from the pair of current state *s0* and action a to the next state *s1*. If the transition probability is successfully learned, the agent will know how likely to enter a specific state given current state and action. However, model-based algorithms become impractical as the state space and action space grows (S * S * A, for a tabular setup).

and actions. All the algorithms discussed in the next section fall into this category.

On-policy v.s. Off-policy

An on-policy agent learns the value based on its current action a derived from the current policy, whereas its off-policy counter part learns it based on the action a* obtained from another policy. In Q-learning, such policy is the greedy policy. (We will talk more on that in Q-learning and SARSA)

2. Illustration of Various Algorithms

2.1 Q-Learning

Q-Learning is an off-policy, model-free RL algorithm based on the well-known Bellman Equation:

Bellman Equation (https://zhuanlan.zhihu.com/p/21378532?refer=intelligentunit)

E in the above equation refers to the expectation, while ƛ refers to the discount factor. We can re-write it in the form of Q-value:

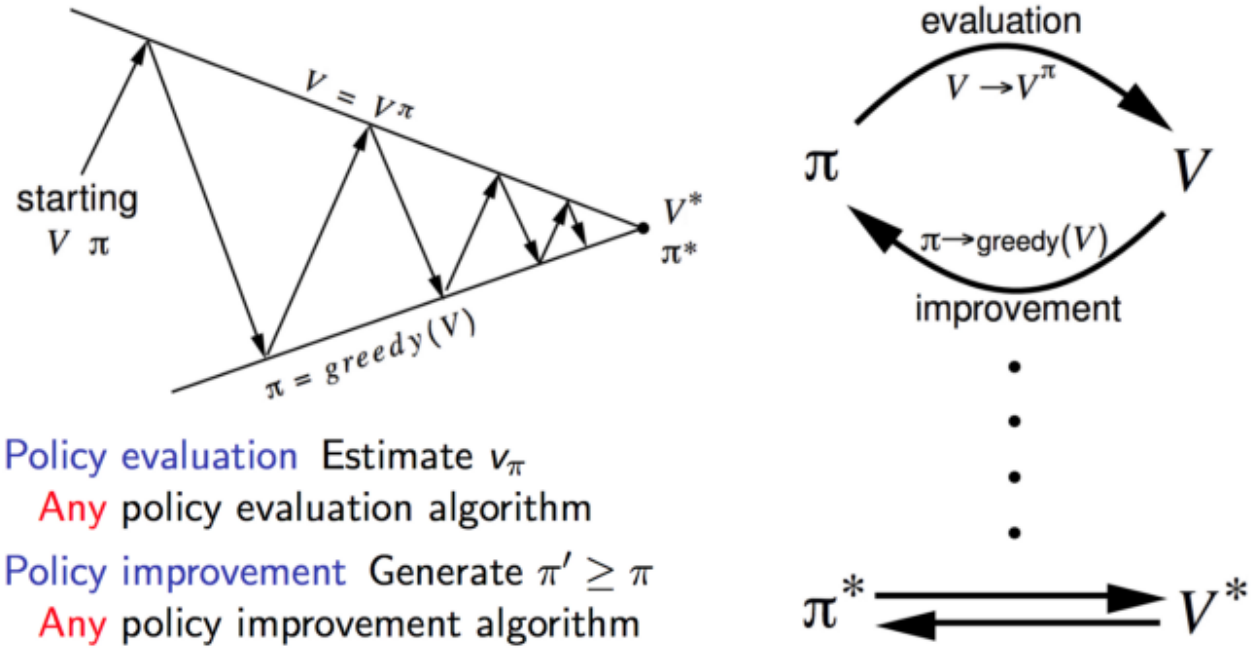
Bellman Equation In Q-value Form (https://zhuanlan.zhihu.com/p/21378532?refer=intelligentunit)

The optimal Q-value, denoted as Q* can be expressed as:

Optimal Q-value (https://zhuanlan.zhihu.com/p/21378532?refer=intelligentunit)

The goal is to maximize the Q-value. Before diving into the method to optimize Q-value, I would like to discuss two value update methods that are closely related to Q-learning.

Policy Iteration



Policy Iteration (http://blog.csdn.net/songrotek/article/details/51378582)

Policy evaluation estimates the value function V with the greedy policy obtained from the last policy improvement. Policy improvement, on the other hand, updates the policy with the action that maximizes V for each of the state. The update equations are based on Bellman Equation. It keeps iterating till convergence.

1. Initialization
   $V(s) \in \mathbb{R}$ and $\pi(s) \in \mathcal{A}(s)$ arbitrarily for all $s \in \mathcal{S}$

2. Policy Evaluation
   Repeat
   $\Delta \leftarrow 0$
   For each $s \in \mathcal{S}$:
   $v \leftarrow V(s)$
   $V(s) \leftarrow \sum_{s',r} p(s', r|s, \pi(s))\left[r + \gamma V(s')\right]$
   $\Delta \leftarrow \max(\Delta, |v - V(s)|)$
   until $\Delta < \theta$ (a small positive number)

3. Policy Improvement

policy-stable $\leftarrow$ *true*

For each $s \in \mathcal{S}$:

$a \leftarrow \pi(s)$

$\pi(s) \leftarrow \arg\max_a \sum_{s',r} p(s', r|s, a)\left[r + \gamma V(s')\right]$

If $a \neq \pi(s)$, then *policy-stable* $\leftarrow$ *false*

If *policy-stable*, then stop and return V and π; else go to 2

Pseudo Code For Policy Iteration (http://blog.csdn.net/songrotek/article/details/51378582)

Value Iteration

Value Iteration only contains one component. It updates the value function V based on the Optimal Bellman Equation.

$$
\begin{aligned}
v_*(s) &= \max_a \mathbb{E}[R_{t+1} + \gamma v_*(S_{t+1}) \mid S_t = s, A_t = a] \\
&= \max_a \sum_{s',r} p(s', r|s, a)\left[r + \gamma v_*(s')\right]
\end{aligned}
$$

Optimal Bellman Equation (http://blog.csdn.net/songrotek/article/details/51378582)

Initialize array V arbitrarily (e.g., $V(s) = 0$ for all $s \in \mathcal{S}^+$)

Repeat

$\Delta \leftarrow 0$

For each $s \in \mathcal{S}$:

$v \leftarrow V(s)$

$V(s) \leftarrow \max_a \sum_{s',r} p(s', r|s, a)\left[r + \gamma V(s')\right]$

$\Delta \leftarrow \max(\Delta, |v - V(s)|)$

until $\Delta < \theta$ (a small positive number)

$$\pi(s) = \arg\max_a \sum_{s',r} p(s', r|s, a)\left[r + \gamma V(s')\right]$$

Pseudo Code For Value Iteration (http://blog.csdn.net/songrotek/article/details/51378582)

After the iteration converges, the optimal policy is straight-forwardly derived by applying an argument-max function for all of the states.

Note that these two methods require the knowledge of the transition probability *p*, indicating that it is a model-based algorithm. However, as I mentioned earlier, model-based algorithm suffers from scalability problem. So how does Q-learning solves this problem?

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha[r_{t+1} + \gamma \max_a Q(s_{t+1}, a) - Q(s_t, a_t)]$$

Q-Learning Update Equation (https://www.quora.com/What-is-the-difference-between-Q-learning-and-SARSA-learning)

α refers to the learning rate (i.e. how fast are we approaching the goal). The idea behind Q-learning is highly relied on value iteration. However, the update equation is replaced with the above formula. As a result, we do not need to worry about the transition probability anymore.

Q-learning: Learn function $Q : \mathcal{X} \times \mathcal{A} \to \mathbb{R}$

Require:
Sates $\mathcal{X} = \{1, \ldots, n_x\}$
Actions $\mathcal{A} = \{1, \ldots, n_a\}$, $A : \mathcal{X} \Rightarrow \mathcal{A}$
Reward function $R : \mathcal{X} \times \mathcal{A} \to \mathbb{R}$
Black-box (probabilistic) transition function $T : \mathcal{X} \times \mathcal{A} \to \mathcal{X}$
Learning rate $\alpha \in [0, 1]$, typically $\alpha = 0.1$
Discounting factor $\gamma \in [0, 1]$
procedure QLEARNING($\mathcal{X}$, A, R, T, α, γ)
  Initialize $Q : \mathcal{X} \times \mathcal{A} \to \mathbb{R}$ arbitrarily
  while Q is not converged **do**
    Start in state $s \in \mathcal{X}$
    while s is not terminal **do**
      Calculate π according to Q and exploration strategy (e.g. $\pi(x) \leftarrow \arg\max_a Q(x, a)$)
      $a \leftarrow \pi(s)$
      $r \leftarrow R(s, a)$ ▷ Receive the reward
      $s' \leftarrow T(s, a)$ ▷ Receive the new state

Q-learning Pseudo Code (https://martin-thoma.com/images/2016/07/q-learning.png)

Note that the next action *a'* is chosen to maximize the next state's Q-value instead of following the current policy. As a result, Q-learning belongs to the off-policy category.

2.2 State-Action-Reward-State-Action (SARSA)

SARSA very much resembles Q-learning. The key difference between SARSA and Q-learning is that SARSA is an on-policy algorithm. It implies that SARSA learns the Q-value based on the action performed by the current policy instead of the greedy policy.

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha[r_{t+1} + \gamma Q(s_{t+1}, a_{t+1}) - Q(s_t, a_t)]$$

SARSA Update Equation (https://www.quora.com/What-is-the-difference-between-Q-learning-and-SARSA-learning)

The action a_(t+1) is the action performed in the next state s_(t+1) under current policy.

SARSA(λ): Learn function $Q : \mathcal{X} \times \mathcal{A} \to \mathbb{R}$

Require:
- Sates $\mathcal{X} = \{1, \ldots, n_x\}$
- Actions $\mathcal{A} = \{1, \ldots, n_a\}$, $A : \mathcal{X} \Rightarrow \mathcal{A}$
- Reward function $R : \mathcal{X} \times \mathcal{A} \to \mathbb{R}$
- Black-box (probabilistic) transition function $T : \mathcal{X} \times \mathcal{A} \to \mathcal{X}$
- Learning rate $\alpha \in [0, 1]$, typically $\alpha = 0.1$
- Discounting factor $\gamma \in [0, 1]$
- $\lambda \in [0, 1]$: Trade-off between TD and MC

procedure QLEARNING($\mathcal{X}$, A, R, T, α, γ, λ)
- Initialize $Q : \mathcal{X} \times \mathcal{A} \to \mathbb{R}$ arbitrarily
- Initialize $e : \mathcal{X} \times \mathcal{A} \to \mathbb{R}$ with 0. ▷ eligibility trace
- **while** Q is not converged **do**
  - Select $(s, a) \in \mathcal{X} \times \mathcal{A}$ arbitrarily
  - **while** s is not terminal **do**
    - $r \leftarrow R(s, a)$
    - $s' \leftarrow T(s, a)$ ▷ Receive the new state
    - Calculate π based on Q (e.g. epsilon-greedy)
    - $a' \leftarrow \pi(s')$
    - $e(s, a) \leftarrow e(s, a) + 1$
    - $\delta \leftarrow r + \gamma \cdot Q(s', a') - Q(s, a)$
    - **for** $(\tilde{s}, \tilde{a}) \in \mathcal{X} \times \mathcal{A}$ **do**
      - $Q(\tilde{s}, \tilde{a}) \leftarrow Q(\tilde{s}, \tilde{a}) + \alpha \cdot \delta \cdot e(\tilde{s}, \tilde{a})$
      - $e(\tilde{s}, \tilde{a}) \leftarrow \gamma \cdot \lambda \cdot e(\tilde{s}, \tilde{a})$
    - $s \leftarrow s'$

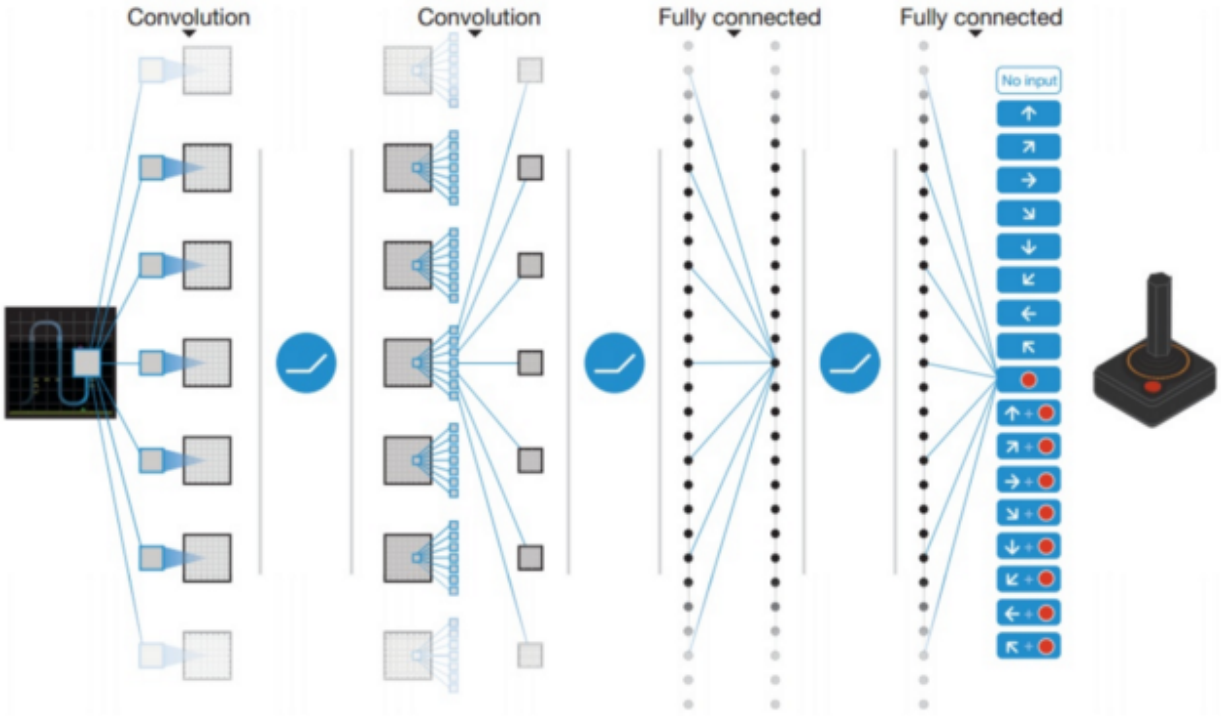
SARSA Pseudo Code (https://martin-thoma.com/images/2016/01/sarsa-lambda.png)

From the pseudo code above you may notice two action selection are performed, which always follows the current policy. By contrast, Q-learning has no constraint over the next action, as long as it maximizes the Q-value for the next state. Therefore, SARSA is an on-policy algorithm.

2.3 Deep Q Network (DQN)

Although Q-learning is a very powerful algorithm, its main weakness is lack of generality. If you view Q-learning as updating numbers in a two-dimensional array (Action Space * State Space), it, in fact, resembles dynamic programming. This indicates that for states that the Q-learning agent has not seen before, it has no clue which action to take. In other words, Q-learning agent does not have the ability to estimate value for unseen states. To deal with this problem, DQN get rid of the two-dimensional array by introducing Neural Network.

DQN leverages a Neural Network to estimate the Q-value function. The input for the network is the current, while the output is the corresponding Q-value for each of the action.



In 2013, DeepMind applied DQN to Atari game, as illustrated in the above figure. The input is the raw image of the current game situation. It went through several layers including convolutional layer as well as fully connected layer. The output is the Q-value for each of the actions that the agent can take.

The question boils down to: **How do we train the network?**

The answer is that we train the network based on the Q-learning update equation. Recall that the target Q-value for Q-learning is:

$$r_j + \gamma \max_{a'} Q\left(\phi_{j+1}, a'; \theta^-\right)$$

Target Q-value (https://storage.googleapis.com/deepmind-media/dqn/DQNNaturePaper.pdf)

The ϕ is equivalent to the state s, while the $\boldsymbol{\theta}$ stands for the parameters in the Neural Network, which is not in the domain of our discussion. Thus, the loss function for the network is defined as the Squared Error between target Q-value and the Q-value output from the network.

Algorithm 1: deep Q-learning with experience replay.

Initialize replay memory D to capacity N
Initialize action-value function Q with random weights θ
Initialize target action-value function $\hat{Q}$ with weights $\theta^- = \theta$
For episode = 1, M **do**
  Initialize sequence $s_1 = \{x_1\}$ and preprocessed sequence $\phi_1 = \phi(s_1)$
  For t = 1,T **do**
    With probability ε select a random action a_t
    otherwise select $a_t = \text{argmax}_a Q(\phi(s_t), a; \theta)$
    Execute action a_t in emulator and observe reward r_t and image x_{t+1}
    Set $s_{t+1} = s_t, a_t, x_{t+1}$ and preprocess $\phi_{t+1} = \phi(s_{t+1})$
    Store transition $\left(\phi_t, a_t, r_t, \phi_{t+1}\right)$ in D
    Sample random minibatch of transitions $\left(\phi_j, a_j, r_j, \phi_{j+1}\right)$ from D
    Set $y_j = \begin{cases} r_j & \text{if episode terminates at step j+1} \\ \end{cases}$

network parameters θ
Every C steps reset $\hat{Q} = Q$
End For
End For

DQN Pseudo Code (https://storage.googleapis.com/deepmind-media/dqn/DQNNaturePaper.pdf)

Another two techniques are also essential for training DQN:

1. **Experience Replay**: Since training samples in typical RL setup are highly correlated, and less data-efficient, it will leads to harder convergence for the network. A way to solve the sample distribution problem is adopting experience replay. Essentially, the sample transitions are stored, which will then be randomly selected from the "transition pool" to update the knowledge.

2. **Separate Target Network**: The target Q Network has the same structure as the one that estimates value. Every C steps, according to the above pseudo code, the target network is reset to another one. Therefore, the fluctuation becomes less severe, resulting in more stable trainings.

2.4 Deep Deterministic Policy Gradient (DDPG)

Although DQN achieved huge success in higher dimensional problem, such as the Atari game, the action space is still discrete. However, many tasks of interest, especially physical control tasks, the action space is continuous. If you discretize the action space too finely, you wind up having an action space that is too large. For instance, assume the degree of free random system is 10. For each of the degree, you divide the space into 4 parts. You wind up having $4^{10} = 1048576$ actions. It is also extremely hard to converge for such a large action space.

DDPG relies on the actor-critic architecture with two eponymous elements, actor and critic. An actor is used to tune the parameter $\boldsymbol{\theta}$ for the policy function, i.e. decide the best action for a specific state.

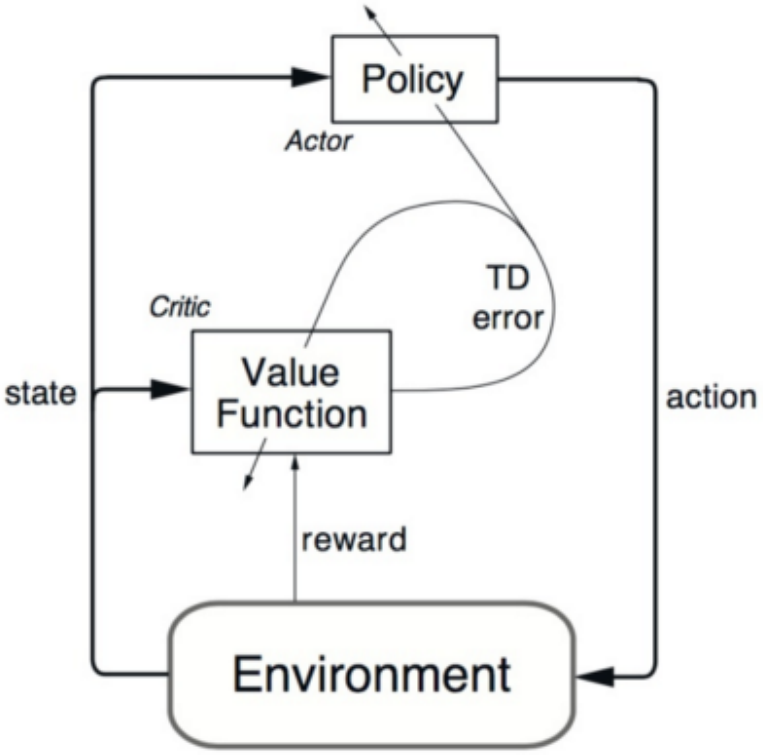
$$\pi_\theta(s, a) = \mathbb{P}[a \mid s, \theta]$$

A critic is used for evaluating the policy function estimated by the actor according to the temporal difference (TD) error.

$$r_{t+1} + \gamma V^v(s_{t+1}) - V^v(s_t)$$

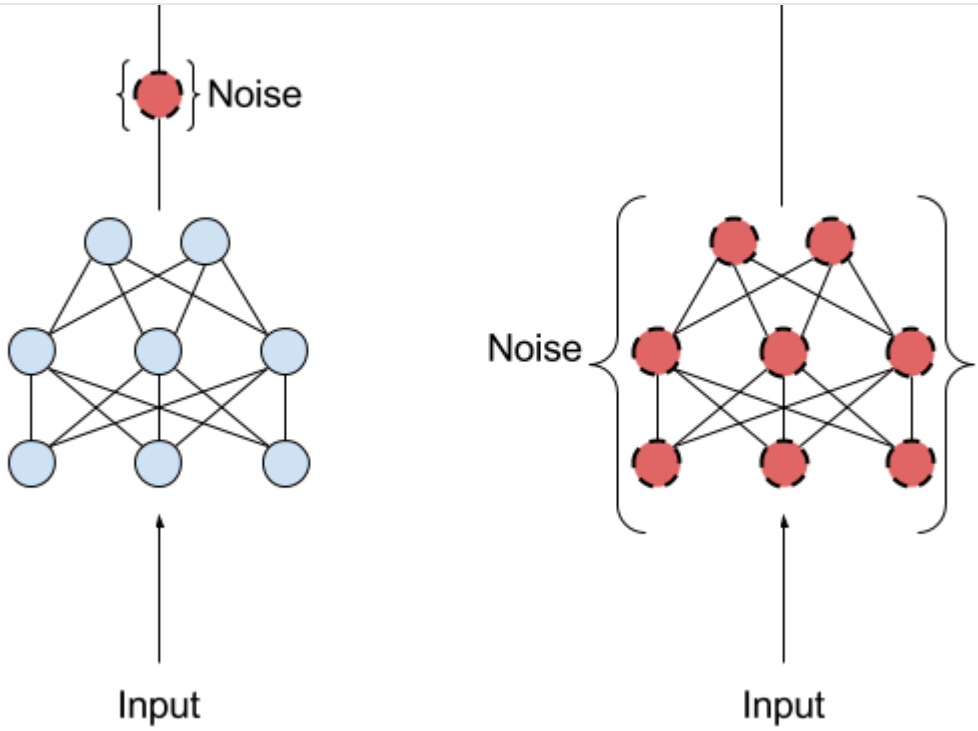
Temporal Difference Error (http://proceedings.mlr.press/v32/silver14.pdf)

Here, the lower-case *v* denotes the policy that the actor has decided. Does it look familiar? Yes! It looks just like the Q-learning update equation! TD learning is a way to learn how to predict a value depending on future values of a given state. Q-learning is a specific type of TD learning for learning Q-value.



Actor-critic Architecture (https://arxiv.org/pdf/1509.02971.pdf)

DDPG also borrows the ideas of **experience replay** and **separate target network** from DQN **.** Another issue for DDPG is that it seldom performs exploration for actions. A solution for this is adding noise on the parameter space or the action space.



Action Noise (left), Parameter Noise (right) (https://blog.openai.com/better-exploration-with-parameter-noise/)

It is claimed that adding on parameter space is better than on action space, according to this article written by OpenAI. One commonly used noise is Ornstein-Uhlenbeck Random Process.

Algorithm 1 DDPG algorithm

Randomly initialize critic network $Q(s, a|\theta^Q)$ and actor $\mu(s|\theta^\mu)$ with weights θ^Q and θ^μ.
Initialize target network Q' and μ' with weights $\theta^{Q'} \leftarrow \theta^Q$, $\theta^{\mu'} \leftarrow \theta^\mu$
Initialize replay buffer R
for episode = 1, M **do**
  Initialize a random process $\mathcal{N}$ for action exploration
  Receive initial observation state s_1
  for t = 1, T **do**
    Select action $a_t = \mu(s_t|\theta^\mu) + \mathcal{N}_t$ according to the current policy and exploration noise
    Execute action a_t and observe reward r_t and observe new state s_{t+1}
    Store transition (s_t, a_t, r_t, s_{t+1}) in R
    Sample a random minibatch of N transitions (s_i, a_i, r_i, s_{i+1}) from R
    Set $y_i = r_i + \gamma Q'(s_{i+1}, \mu'(s_{i+1}|\theta^{\mu'})|\theta^{Q'})$
    Update critic by minimizing the loss: $L = \frac{1}{N}\sum_i (y_i - Q(s_i, a_i|\theta^Q))^2$
    Update the actor policy using the sampled policy gradient:

$$\nabla_{\theta^\mu} J \approx \frac{1}{N} \sum \nabla_a Q(s, a|\theta^Q)|_{s=s_i, a=\mu(s_i)} \nabla_{\theta^\mu} \mu(s|\theta^\mu)|_{s_i}$$

$$\theta^{\mu'} \leftarrow \tau\theta^{\mu} + (1-\tau)\theta^{\mu'}$$

end for
end for

DDPG Pseudo Code (https://arxiv.org/pdf/1509.02971.pdf)

3. Conclusion

I have discussed some basic concepts of Q-learning, SARSA, DQN , and DDPG. In the next article, I will continue to discuss other state-of-the-art Reinforcement Learning algorithms, including NAF, A3C… etc. In the end, I will briefly compare each of the algorithms that I have discussed. Should you have any problem or question regarding to this article, please do not hesitate to leave a comment below or follow me on twitter.

Machine Learning Reinforcement Learning Ddpg Deep Learning Towards Data Science