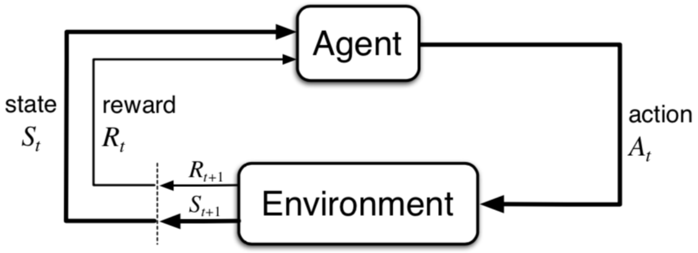
Reinforcement Learning Cheat Sheet

Agent-Environment Interface



The Agent at each step t receives a representation of the environment's *state*, $S_t \in S$ and it selects an action $A_t \in A(s)$. Then, as a consequence of its action the agent receives a *reward*, $R_{t+1} \in R \in \mathbb{R}$.

Markov Decision Process

A **Markov Decision Process**, MPD, is a 5-tuple (S, A, P, R, γ) where:

finite set of states: $s \in S$
finite set of actions: $a \in A$
state transition probabilities:
$p(s'|s,a) = Pr\{S_{t+1} = s'|S_t = s, A_t = a\}$ or
$p(s',r|s,a) = Pr\{S_{t+1} = s', R_{t+1} = r|S_t = s, A_t = a\}$
expected reward for state-action-next state:
$r(s,a,s') = \mathbb{E}[R_{t+1}|S_{t+1} = s', S_t = s, A_t = a]$

Reward

The total *reward* is expressed as $G_t = \sum_{k=0}^{H} \gamma^k r_{t+k+1}$, where γ is the *discount factor* and H is the *horizon*, that can be infinite.

Policy

A *policy* is a mapping from a state to an action $\pi_t(a|s)$. That is the probability of select an action $A_t = a$ if $S_t = s$.

Bellman Equation

Value Function

Value function describes *how good* is to be in a specific state s under a certain policy π.
Policy evaluation:

$$\begin{aligned} v_\pi(s) &= \mathbb{E}_\pi\left[G_t|S_t = s\right]\\ &= \mathbb{E}_\pi\left[\sum_{k=0}^{\infty}\gamma^k R_{t+k+1}|S_t = s\right]\\ &= \mathbb{E}_\pi\left[R_{t+1} + \gamma\sum_{k=0}^{\infty}\gamma^k R_{t+k+2}|S_t = s\right]\\ &= \sum_a \pi(a|s)\sum_{s',r} p(s',r|s,a)\left[r + \gamma\,\underbrace{\mathbb{E}_\pi\left[\sum_{k=0}^{\infty}\gamma^k R_{t+k+2}|S_{t+1} = s'\right]}_{\text{Expected reward from } s_{t+1}}\right]\\ &= \sum_a \pi(a|s)\sum_{s',r} p(s',r|s,a)\left[r + \gamma v_\pi(s')\right] \end{aligned}$$

Q Function

The expected return starting from s, taking the action a, and thereafter following policy π:

$$\begin{aligned} q_\pi(s,a) &= \mathbb{E}_\pi\left[G_t|S_t = s, A_t = a\right]\\ &= \mathbb{E}_\pi\left[\sum_{k=0}^{\infty}\gamma^k R_{t+k+1}|S_t = s, A_t = a\right]\\ &= \mathbb{E}_\pi\left[R_{t+1} + \gamma\sum_{k=0}^{\infty}\gamma^k R_{t+k+2}|S_t = s, A_t = a\right]\\ &= \sum_{s',r} p(s',r|s,a)\left[r + \gamma\mathbb{E}_\pi\left[\sum_{k=0}^{\infty}\gamma^k R_{t+k+2}|S_{t+1} = s'\right]\right]\\ &= \sum_{s',r} p(s',r|s,a)\left[r + \gamma v_\pi(s')\right]\\ &= \sum_{s',r} p(s',r|s,a)\left[r + \gamma\sum_{a'}\pi(a'|s')q_\pi(s',a')\right] \end{aligned}$$

Optimal

$$\begin{aligned} v_*(s) &= \max_\pi v_\pi(s)\\ &= \max_a \sum_{s'}\sum_r p(s',r|s,a)\left[r + \gamma v_*(s')\right] \end{aligned} \tag{1}$$

The optimal value-action function:

$$\begin{aligned} q_*(s,a) &= \max_\pi q_\pi(s,a)\\ &= \sum_{s',r} p(s',r|s,a)\left[r + \gamma v_*(s')\right] \end{aligned} \tag{2}$$

Clearly, we can redefine v_*, (1), using $q_*(s,a)$, (2):

$$v_*(s) = \max_{a \in A(s)} q_*(s,a)$$

Intuitively, the above equation express the fact that the value of a state under the optimal policy **must be equal** to the expected return from the best action from that state.

Dynamic Programming Algorighms

Taking advantages of the subproblem structure of the V and Q function we can find the optimal policy by just *planning*

Policy Iteration (iterative policy evaluation)

```
1. Initialisation
$V(s) \in \mathbb{R}$, (e.g $V(s) = 0$) and $\pi(s) \in A$ for all $s \in S$,
2. Policy Evaluation
repeat
    $\Delta \leftarrow 0$
    foreach $s \in S$ do
        $v \leftarrow V(s)$
        $V(s) \leftarrow \sum_a \pi(a|s) \sum_{s',r} p(s',r|s,a)\left[r + \gamma V(s')\right]$
        $\Delta \leftarrow \max(\Delta, |v - V(s)|)$
    end
until $\Delta < \theta$ (a small positive number);
3. Policy Improvement
policy-stable $\leftarrow$ true
foreach $s \in S$ do
    old-action $\leftarrow \pi(s)$
    $\pi(s) \leftarrow \mathrm{argmax}_a \sum_{s',r} p(s',r|s,a)\left[r + \gamma V(s')\right]$
    If old-action $\neq \pi(s)$, policy-stable $\leftarrow$ false
end
If policy-stable, stop; else go to 2.
```

Algorithm 1: Policy Iteration

Value Iteration

We can avoid to wait until $V(s)$ has converged and instead do policy improvement and truncated policy evaluation step in one operation

```
Initialize $V(s) \in \mathbb{R}$, e.g $V(s) = 0$
repeat
    $\Delta \leftarrow 0$
    foreach $s \in S$ do
        $v \leftarrow V(s)$
        $V(s) \leftarrow \max_a \sum_{s',r} p(s',r|s,a)\left[r + \gamma V(s')\right]$
        $\Delta \leftarrow \max(\Delta, |v - V(s)|)$
    end
until $\Delta < \theta$ (a small positive number);
ouput: Deterministic policy $\pi \approx \pi_*$ such that
$\pi(s) = \mathrm{argmax}_a \sum_{s',r} p(s',r|s,a)\left[r + \gamma V(s')\right]$
```

Algorithm 2: Value Iteration

Asynchronous DP

The values of states are updated in any order whatsoever, using values of other states happen to be available. Not necessarily sweep all states in one iteration.

Monte Carlo Methods

Monte Carlo (MC) is a *Model Free* method, It does not require complete knowledge of the environment. It is based on **averaging sample returns** for each state-action pair. The following algorithm gives the basic implementation

```
Initialize for all $s \in S, a \in A(s)$ :
  $Q(s,a) \leftarrow$ arbitrary
  $\pi(s) \leftarrow$ arbitrary
  $Returns(s,a) \leftarrow$ empty list
while forever do
    Choose $S_0 \in S$ and $A_0 \in A(S_0)$, all pairs have probability $> 0$
    Generate an episode starting at $S_0, A_0$ following $\pi$
    foreach pair $s, a$ appearing in the episode do
        $G \leftarrow$ return following the first occurrence of $s, a$
        Append $G$ to $Returns(s,a)$
        $Q(s,a) \leftarrow average(Returns(s,a))$
    end
    foreach $s$ in the episode do
        $\pi(s) \leftarrow \operatorname{argmax}_a Q(s,a)$
    end
end
```

Algorithm 3: On-policy Monte Carlo

Off-policy Monte Carlo: use μ to generate episodes.

$$\begin{aligned} V(s) &= \mathbb{E}_\pi[G_t|S_t = s] = \sum_t \mathbb{E}_\pi[G_t|S_t = s] \\ &= \sum_t \mathbb{E}_\mu[L_t G_t|S_t = s] \approx \frac{\sum\limits_{t\in\mathcal{J}(s)} L_t G_t}{\sum\limits_{t\in\mathcal{J}(s)} L_t}, \end{aligned}$$

where

$$L_t = \prod_{k=t} \frac{\pi(a_k|s_k)}{\mu(a_k|s_k)}.$$

Rollout Algorithm

Given s and π, MC estimate $q_\pi(s,a)$. π is called the *rollout policy.* The aim of rollout algorithm is to improve upon the default policy, not find an optimal policy.

Temporal Difference - Q Learning

Temporal Difference (TD) methods learn directly from raw experience without a model of the environment's dynamics. TD substitutes the expected discounted reward G_t from the episode with an estimation:

$$V(S_t) \leftarrow V(S_t) + \alpha\left[R_{t+1} + \gamma V(S_{t+1}) - V(S_t)\right] \tag{3}$$

Sarsa (on-policy TD)

```
Initialize $Q(s,a)$ arbitrarily and $Q(terminal, \cdot) = 0$
foreach episode do
    Choose $a$ from $s$ using policy derived from $Q$ (e.g., $\epsilon$-greedy)
    while $s$ is not terminal do
        Take action $a$, observer $r, s'$
        Choose $a'$ from $s'$ using policy derived from $Q$ (e.g., $\epsilon$-greedy)
        $Q(s,a) \leftarrow Q(s,a) + \alpha\left[r + \gamma Q(s',a') - Q(s,a)\right]$
        $s \leftarrow s'$, $a \leftarrow a'$
    end
end
```

Algorithm 4: Sarsa

Q-learning (Off-policy TD)

```
Initialize $Q(s,a)$ arbitrarily and $Q(terminal, \cdot) = 0$
foreach episode do
    while $s$ is not terminal do
        Choose $a$ from $s$ using policy derived from $Q$ (e.g., $\epsilon$-greedy)
        Take action $a$, observer $r, s'$
        $Q(s,a) \leftarrow Q(s,a) + \alpha\left[r + \gamma \max_{a'} Q(s',a') - Q(s,a)\right]$
        $s \leftarrow s'$
    end
end
```

Algorithm 5: Q-Learning

Dyna-Q: Indirect RL; also learns the model.

```
Initialize $Model(s,a)$ and $Q(s,a)$, $Q(terminal, \cdot) = 0$
foreach episode do
    while $s$ is not terminal do
        Choose $a$ from $s$ using policy derived from $Q$ (e.g., $\epsilon$-greedy)
        Take action $a$, observer $r, s'$
        $Q(s,a) \leftarrow Q(s,a) + \alpha\left[r + \gamma \max_{a'} Q(s',a') - Q(s,a)\right]$
        $Model(s,a) \leftarrow (s', a')$, $s \leftarrow s'$
        repeat
            $s \leftarrow$ random previously observed sate
            $a \leftarrow$ random action previously taken at $s$
            $r, s' \leftarrow Model(s,a)$
            $Q(s,a) \leftarrow$
              $Q(s,a) + \alpha\left[r + \gamma \max_{a'} Q(s',a') - Q(s,a)\right]$
        until $n$ times;
    end
end
```

Algorithm 6: Dyna-Q

$Model(s,a)$ denotes the contents of the predicted next state and reward for state-action pair (s,a).

Approximate DP

TD(0)

$$w \leftarrow w + \alpha\left[R_{t+1} + \gamma\hat{V}(S_{t+1}, w) - \hat{V}(S_t, w)\right]\nabla\hat{V}(S_t, w)$$

TD(λ)

$$\begin{aligned} &z_t = \gamma\lambda z_{t-1} + \nabla\hat{V}(S_t, w_t), \quad z_{-1} = 0 \\ &\delta_t = R_{t+1} + \gamma\hat{V}(S_{t+1}, w_t) - \hat{V}(S_t, w_t) \\ &w_{t+1} = w_t + \alpha_t\delta_t z_t \end{aligned}$$

Derivation of TD(1)

$$\text{Cost :} \qquad J = \frac{1}{2}\sum_{t=0}^{\infty}\left(\sum_{k=t}^{\infty}\gamma^{k-t}R_{k+1} - \hat{V}(s_t, w)\right)^2$$

Gradient descent (batched)

$$\begin{aligned} w &\leftarrow w + \alpha\partial_w J \\ &\leftarrow w + \alpha\textstyle\sum_{t=0}^{\infty}\left(\sum_{k=t}^{\infty}\gamma^{k-t}R_{k+1} - \hat{V}(s_t, w)\right)\partial_w\hat{V}(s_t, w) \\ &\leftarrow w + \alpha\textstyle\sum_{t=0}^{\infty}\left(\sum_{k=t}^{\infty}\gamma^{k-t}\delta_k\right)\partial_w\hat{V}(s_t, w) \\ &\leftarrow w + \alpha\textstyle\sum_{k=0}^{\infty}\delta_k\sum_{t=0}^{k}\gamma^{k-t}\partial_w\hat{V}(s_t, w) \end{aligned}$$

Sarsa(λ) and Q(λ)

Define the n-step Q-Return

$$q^{(n)} = R_{t+1} + \gamma R_{t+2} + \ldots + \gamma^{n-1}R_{t+n} + \gamma^n Q(s_{t+n}, a_{t+n})$$

n-step Sarsa update Q towards the n-step Q-return

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha\left[q_t^{(n)} - Q(s_t, a_t)\right]$$

Forward-view Sarsa(λ):

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha\left[q_t^{\lambda} - Q(s_t, a_t)\right]$$

$$q_t^\lambda = (1-\lambda)\sum_{n=1}^{\infty}\lambda^{n-1}q_t^{(n)}$$

Backward-view Sarsa(λ): TD(λ) with $\hat{V} \Rightarrow \hat{Q}$.
Backward-view Q(λ): Sarsa(λ) with
$\hat{Q}(S_{t+1}, a_{t+1}, w_t) \Rightarrow \inf_{a'} \hat{Q}(S_{t+1}, a', w_t)$.

Deep Q Learning

Deep Q Learning substitutes the Q function with a deep NN called *Q-network*. It also keep track of some observation in a *memory* in order to use them to train the network.

$$L_i(\theta_i) = \mathbb{E}_{(s,a,r,s')\sim U(D)}\left[\underbrace{(r + \gamma \max_a Q(s', a'; \theta_{i-1})}_{\text{target}} - \underbrace{Q(s, a; \theta_i))}_{\text{prediction}}^2\right]$$

where θ are the weights of the network and $U(D)$ is the experience replay history.

```
Initialize replay memory $D$ with capacity $N$
Initialize $Q(s,a)$ arbitrarily
foreach episode $\in$ episodes do
    while $s$ is not terminal do
        With probability $\epsilon$ select a random action $a \in A(s)$
        otherwise select $a = \max_a Q(s, a; \theta)$
        Take action $a$, observer $r, s'$
        Store transition $(s, a, r, s')$ in $D$
        Sample random minibatch of transitions
          $(s_j, a_j, r_j, s'_j)$ from $D$
        Set $y_j \leftarrow$
        $\begin{cases} r_j & \text{for terminal } s'_j \\ r_j + \gamma \max_a Q(s', a'; \theta) & \text{for non-terminal } s'_j \end{cases}$
        Gradient descent on $(y_j - Q(s_j, a_j; \theta))^2$
        $s \leftarrow s'$
    end
end
```

Algorithm 7: Deep Q Learning

Policy gradient method

Policy gradient (gradient ascent):

$$\theta_{t+1} = \theta_t + \alpha\partial_\theta\rho(\pi_\theta)$$

Policy gradient theorem

$$\partial_\theta\rho(\pi_\theta) \propto \sum_s \mu_s \sum_a q_\pi(s, a)\partial_\theta\pi(a|s, \theta)$$

μ_s is the on-policy distribution under π (the fraction of time spent in each state). For average reward, μ is steady state distribution.