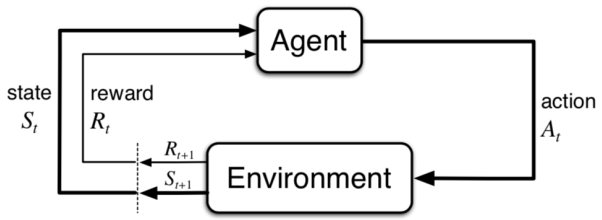
Reinforcement Learning Cheat Sheet

Agent-Environment Interface



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

Reward

The total expected cumulative *reward* is expressed as:

$$\mathcal{G}_t = \sum_{k=0}^{\mathcal{H}} \gamma^k r_{t+k+1} \tag{1}$$

Where γ is the *discount factor* ($\gamma \in [0, 1]$) and $\mathcal{H}$ is the *horizon*, that can be infinite.

Reward Hypothesis

All goals can be described by the maximisation of expected cumulative *rewards*.

Policy

A *policy* π is the behaviour function mapping a state to an action $\pi(s|a)$

History

The *history* is defined as what the agent has seen until time-step t

$$\mathcal{H}_t = \mathcal{S}_1, \mathcal{A}_1, \mathcal{R}_1, \cdots, \mathcal{S}_t, \mathcal{A}_t, \mathcal{R}_t \tag{2}$$

Model

A *model* is the representation of the environment and predicts what the environment will do next (model $\neq$ environment)

Transition model

A *transition model* predicts the next state

$$\mathcal{P}^a_{ss'} = \mathbb{P}[\mathcal{S}' = s' | \mathcal{S} = s, \mathcal{A} = a] \tag{3}$$

Reward model

A *reward model* predicts the next immediate rewards

$$\mathcal{R}^a_{ss'} = \mathbb{E}[\mathcal{R} | \mathcal{S} = s, \mathcal{A} = a] \tag{4}$$

Value Function

The *value function* informs the agent how good is a state based on the expected cumulative reward following policy π.

$$v_\pi(s) = \mathbb{E}_\pi[\mathcal{G}_t | \mathcal{S}_t = s] \tag{5}$$

In some state s and time-step t, the value function informs the agent of the expected sum of future rewards on a given policy π, so as to choose the right action from that state, that maximises that expected sum of rewards.

Markov Decision Process

Markov State

A state is *Markov* if and only if

$$\mathbb{P}[\mathcal{S}_{t+1} | \mathcal{S}_t] = \mathbb{P}[\mathcal{S}_{t+1} | \mathcal{S}_1, \cdots, \mathcal{S}_t] \tag{6}$$

A **Markov Decision Process** (MPD), is a tuple $(\mathcal{S}, \mathcal{A}, \mathcal{P}, \mathcal{R}, \gamma)$ where:

$$\begin{aligned}
&\mathcal{S} \text{ is a finite set of Markov states:} \\
&\mathcal{A} \text{ is a finite set of actions:} \\
&\mathcal{P} \text{ is a state transition probability matrix:} \\
&\mathcal{P}^a_{ss'} = \mathbb{P}\{\mathcal{S}_{t+1} = s' | \mathcal{S}_t = s, \mathcal{A}_t = a\} \\
&\mathcal{R} \text{ is the expected reward:} \\
&\mathcal{R}^a_{ss'} = \mathbb{E}[\mathcal{R}_{t+1} | \mathcal{S}_{t+1} = s', \mathcal{S}_t = s, \mathcal{A}_t = a]
\end{aligned} \tag{7}$$

State-Value Function - v

The *state-value* function of an MDP is the expected return starting from state s, following policy π:

$$v_\pi(s) = \mathbb{E}_\pi[\mathcal{G}_t | \mathcal{S}_t = s] \tag{8}$$

Action-Value Function - q

The *action-value* function of an MDP is the expected return starting from state s, taking action a, and following policy π:

$$q_\pi(s, a) = \mathbb{E}_\pi[\mathcal{G}_t | \mathcal{S}_t = s, \mathcal{A}_t = a] \tag{9}$$

Bellman Expectation Equation

$$\begin{aligned}
v_\pi(s) &= \sum_{a \in \mathcal{A}} \pi(a|s) \Big(\mathcal{R}^a_s + \gamma \sum_{s' \in \mathcal{S}} \mathcal{P}^a_{ss'} v_\pi(s') \Big) \\
q_\pi(s, a) &= \mathcal{R}^a_s + \gamma \sum_{s' \in \mathcal{S}} \mathcal{P}^a_{ss'} \sum_{a' \in \mathcal{A}} \pi(s'|a') q_\pi(s', a')
\end{aligned} \tag{10}$$

Equivalence v_π - q_π

$$\begin{aligned}
v_\pi(s) &= \sum_{a \in \mathcal{A}} \pi(s|a) q_\pi(s, a) \\
q_\pi(s, a) &= \mathcal{R}^a_s + \gamma \sum_{s' \in \mathcal{S}} \mathcal{P}^a_{ss'} v_\pi(s')
\end{aligned} \tag{11}$$

Optimality

Theorem of Optimality

A policy $\pi(s|a)$ achieves the optimal value from state s, $v_\pi(s) = v_*(s)$ if and only if for any state s' reachable from s, π achieves the optimal value from state s', $v_\pi(s') = v_*(s')$.

$$\pi_*(s|a) = \begin{cases} 1, \text{ if } a = \underset{a \in \mathcal{A}}{\operatorname{argmax}}\, q_*(s, a) \\ 0 \end{cases} \tag{12}$$

Optimal Value Function

$$\begin{aligned}
v_*(s) &= \max_\pi v_\pi(s) \\
q_*(s, a) &= \max_\pi q_\pi(s)
\end{aligned} \tag{13}$$

Bellman Optimality Equation

$$\begin{aligned}
v_*(s) &= \max_a \Big(\mathcal{R}^a_s + \gamma \sum_{s' \in S} \mathcal{P}^a_{ss'} v_*(s') \Big) \\
q_*(s, a) &= \mathcal{R}^a_s + \gamma \sum_{s' \in \mathcal{S}} \mathcal{P}^a_{ss'} \max_{a'} q_*(s', a')
\end{aligned} \tag{14}$$

Contraction Mapping

Definition

Let (X, d) be a metric space and $f : X \to X$. We say that f is a *contraction* if there is a real number $k \in [0, 1)$ such that

$$d(f(x), f(y)) \leq k d(x, y)$$

for all x and y in X, where the term k is called a *Lipschitz coefficent* for f.

Contraction Mapping theorem

Let (X, d) be a complete metric space and let $f : X \to X$ be a contraction. Then there is one and only one fixed point x^* such that

$$f(x^*) = x^*$$

Moreover, if x is any point in X and $f^n(x)$ is inductively defined by $f^2(x) = f(f(x))$, $f^3(x) = f(f^2(x))$, ..., $f^n(x) = f(f^{n1}(x))$, then $f^n(x) \to x^*$ as $n \to \infty$. This theorem guarantees a unique optimal solution for the dynamic programming algorithms detailed below.

Dynamic Programming

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

Policy Iteration

We can now find the optimal policy

1. Initialisation
$v(s) \in \mathbb{R}$, (e.g. $V(s) = 0$) and $\pi(s) \in \mathcal{A}$ for all $s \in \mathcal{S}$,
$\Delta \leftarrow 0$
2. Policy Evaluation
while $\Delta \geq \theta$ *(a small positive number)* **do**
  foreach $s \in \mathcal{S}$ **do**
    $v \leftarrow v(s)$
    $v(s) \leftarrow \sum_a \pi(a|s) \sum_{s' \in \mathcal{S}} \mathcal{P}^a_{ss'} [\mathcal{R}^a_s + \gamma v(s')]$
    $\Delta \leftarrow \max(\Delta, |v - v(s)|)$
  end
end
3. Policy Improvement
policy-stable $\leftarrow$ *true*
foreach $s \in \mathcal{S}$ **do**
  old-action $\leftarrow \pi(s)$
  $\pi(s) \leftarrow \underset{a}{\operatorname{argmax}} \sum_{s' \in \mathcal{S}} \mathcal{P}^a_{ss'} [r + \gamma v(s')]$
  policy-stable $\leftarrow$ *old-action* $= \pi(s)$
end
if *policy-stable* return $v \approx v_*$ and $\pi \approx \pi_*$, 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

Initialise $v(s) \in \mathbb{R}$, e.g $v(s) = 0$
$\Delta \leftarrow 0$
while $\Delta \geq \theta$ *(a small positive number)* **do**
  foreach $s \in \mathcal{S}$ **do**
    $v \leftarrow v(s)$
    $v(s) \leftarrow \max_a \sum_{s' \in \mathcal{S}} \mathcal{P}^a_{ss'}[\mathcal{R}^a_s + \gamma v(s')]$
    $\Delta \leftarrow \max(\Delta, |v - v(s)|)$
  end
end
ouput: Deterministic policy $\pi \approx \pi_*$ such that $\pi(s) = \operatorname{argmax}_a \sum_{s' \in \mathcal{S}} \mathcal{P}^a_{ss'}[\mathcal{R}^a_s + \gamma v(s')]$

Algorithm 2: Value Iteration

Monte-Carlo Methods

Monte-Carlo (MC) are *Model-Free* methods, that do not require complete knowledge of the environment but learn from **complete** episodes. It is based on **averaging sample returns** for each state-action pair.

Initialise for all $s \in \mathcal{S}, a \in \mathcal{A}$:
  $q(s, a) \leftarrow$ arbitrary
  $\pi(s) \leftarrow$ arbitrary
  $Returns(s, a) \leftarrow$ empty list
while *forever* **do**
  Choose $\mathcal{S}_0 \in \mathcal{S}$ and $\mathcal{A}_0 \in \mathcal{A}$, all pairs have probability > 0
  Generate an episode starting at $\mathcal{S}_0, \mathcal{A}_0$ following π
  foreach *pair* s, a *appearing in the episode* **do**
    $\mathcal{G} \leftarrow$ return following the first occurrence of s, a
    Append $\mathcal{G}$ 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: Monte Carlo first-visit

For non-stationary problems, the Monte Carlo estimate for, e.g, v is:

$$V(S_t) \leftarrow V(S_t) + \alpha \left[G_t - V(S_t)\right] \tag{15}$$

Where α is the learning rate, how much we want to forget about past experiences.

Sarsa

Sarsa (State-action-reward-state-action) is a on-policy TD control. The update rule:

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

n-step Sarsa

Define the n-step Q-Return

$$q^{(n)} = R_{t+1} + \gamma Rt + 2 + \ldots + \gamma^{n-1} R_{t+n} + \gamma^n Q(S_{t+n})$$

n-step Sarsa update $Q(S, a)$ towards the n-step Q-return

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

Forward View Sarsa(λ)

$$q^\lambda_t = (1 - \lambda) \sum_{n=1}^{\infty} \lambda^{n-1} q^{(n)}_t$$

Forward-view Sarsa(λ):

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha \left[q^\lambda_t - Q(s_t, a_t)\right]$$

Initialise $Q(s, a)$ arbitrarily and
  $Q(terminal - state,) = 0$
foreach *episode* $\in$ *episodes* **do**
  Choose a from s using policy derived from Q (e.g., ϵ-greedy)
  while s *is not terminal* **do**
    Take action a, observer r, s'
    Choose a' from s' using policy derived from Q (e.g., ϵ-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(λ)

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:

$$V(S_t) \leftarrow V(S_t) + \alpha \left[R_{t+1} + \gamma V(S_{t+1} - V(S_t)\right] \tag{16}$$

The following algorithm gives a generic implementation.

Initialise $Q(s, a)$ arbitrarily and
  $Q(terminal - state,) = 0$
foreach *episode* $\in$ *episodes* **do**
  while s *is not terminal* **do**
    Choose a from s using policy derived from Q (e.g., ϵ-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

Deep Q Learning

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

$$L_i(\theta_i) = \mathbb{E}_{(s,a,r,s') \sim U(D)} \left[(\underbrace{r + \gamma \max_a Q(s', a'; \theta_{i-1})}_{\text{target}} - \underbrace{Q(s, a; \theta_i)}_{\text{prediction}})^2 \right] \tag{17}$$

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

Initialise replay memory D with capacity N
Initialise $Q(s, a)$ arbitrarily
foreach *episode* $\in$ *episodes* **do**
  while s *is not terminal* **do**
    With probability ϵ 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 $y_j \leftarrow \begin{cases} r_j & \text{for terminal } s'_j \\ r_j + \gamma \max_a Q(s', a'; \theta) & \text{for non-terminal } s'_j \end{cases}$
    Perform gradient descent step on $(y_j - Q(s_j, a_j; \Theta))^2$
    $s \leftarrow s'$
  end
end

Algorithm 6: Deep Q Learning