Reinforcement Learning Cheat Sheet

Terminology

a action, arm (multi-armed bandit problem)

A advantage

α step size

b bandit, baseline estimate, bias

c cost, context (= state)

d difference error

Δ difference

D replay buffer

E eligibility trace

ϵ learning rate (0 1)

ϕ state transition function

g gain (reward over time, target)

γ discount factor

∇ gradient (spacial derivative)

H horizon

J expected return (see loss)

L expected (squared) loss (see return)

λ trace decay (0 1)

m momentum

Pr transition distribution

π policy ($state \rightarrow action$)

q quality

r reward

R return

ρ regret

s state

τ trajectory

θ parameter weight (learning target)

θ_i^- target network

v value

vm variance momentum

w weight, winning probability

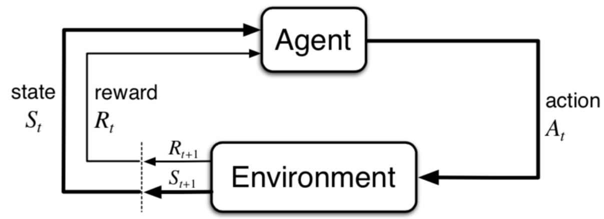
$_*$ optimal

$'$ next / derivative

$\hat{}$ estimation

vector norm

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*, $R_{t+1} \in R \in \mathbb{R}$.

Policy

A *policy* is a mapping from a state to an action

$$\pi_t(s|a) \tag{1}$$

That is the probability of select an action $A_t = a$ if $S_t = s$.

Reward

The total *reward* is expressed as:

$$G_t = \sum_{k=0}^{H} \gamma^k r_{t+k+1} \tag{2}$$

Where γ is the *discount factor* and H is the *horizon*, that can be infinite.

Bandits

A class of decision methods modeling how to pick an action in the absence of observable state influencing the outcome, signified by the multi-armed bandit problem. This reduces reinforcement learning to the sub-problem of exploration vs exploitation.
They estimate $\hat{r}_a \approx \mathbb{E}[r|a]$

Greedy

$$n_a = \sum_{t:a_t=a} 1; \hat{r}_a = \sum_{t:a_t=a} \frac{r_t}{n_a}; a_t = \operatorname*{argmax}_{a\in A} \hat{r}_a \tag{3}$$

Optimistic-Greedy

large initial $\hat{r}_a$, R

ϵ-Greedy

with probability ϵ, pick randomly

Upper Confidence Bound

try all for k rounds, then

$$a_t = \operatorname*{argmax}_{a\in A} \hat{r}_a + \sqrt{\frac{2logt}{n_a}} \tag{4}$$

Contextual Bandit (LinUCB)

$$x_{t,a} = \Phi(s_{t,a}); \mathbb{E}[r_{t,a}|x_{t,a}] = \theta_a \cdot x_{t,a}; \hat{\theta}_a = A^{-1}b \tag{5}$$

Posterior / Thompson sampling

choose by probability actions maximize expected reward

Greedy in the Limit with Infinite Exploration (GLIE)

infinitely explores, converges on greedy

Markov Decision Process

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

$$\begin{array}{l} \text{finite set of states:} \\ s \in S \\ \text{finite set of actions:} \\ a \in A \\ \text{state transition probabilities:} \\ p(s'|s,a) = Pr\{S_{t+1} = s' | S_t = s, A_t = a\} \\ \text{expected reward for state-action-nexstate:} \\ r(s', s, a) = \mathbb{E}[R_{t+1} | S_{t+1} = s', S_t = s, A_t = a] \end{array} \tag{6}$$

Value Function

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

$$V_\pi(s) = \mathbb{E}[G_t | S_t = s] \tag{7}$$

Informally, it is the expected return (expected cumulative discounted reward) when starting from s and following π.

Optimal

$$v_*(s) = \max_\pi v^\pi(s) \tag{8}$$

Action-Value (Q) Function

We can also denote the expected reward for state-action pairs.

$$q_\pi(s,a) = \mathbb{E}_\pi \left[G_t | S_t = s, A_t = a \right] \tag{9}$$

Optimal

The optimal value-action function is denoted as:

$$q_*(s,a) = \max_\pi q^\pi(s,a) \tag{10}$$

Clearly, using this new notation we can redefine v^*, equation 8, using $q^*(s,a)$, equation 10:

$$v_*(s) = \max_{a \in A(s)} q_{\pi*}(s,a) \tag{11}$$

Intuitively, the above equation expresses 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.

model-based methods (known MDP)

Exhaustive Search

Bellman Equation

An important recursive property emerges for both Value (7) and Q (9) functions if we expand them.

Value Function

$$
\begin{aligned}
v_\pi(s) &= \mathbb{E}_\pi\left[G_t|S_t = s\right] \\
&= \mathbb{E}_\pi\left[\sum_{k=0}^{\infty}\gamma^k R_{t+k+1}|S_t = s\right] \\
&= \mathbb{E}_\pi\left[R_{t+1} + \sum_{k=0}^{\infty}\gamma^k R_{t+k+2}|S_t = s\right] \\
&= \underbrace{\sum_a \pi(a|s)\sum_{s'}\sum_r p(s',r|s,a)}_{\text{Sum of all probabilities } \forall \text{ possible } r} \\
&\quad \left[r + \gamma\underbrace{\mathbb{E}_\pi\left[\sum_{k=0}^{\infty}\gamma^k R_{t+k+2}|S_{t+1} = s'\right]}_{\text{Expected reward from } s_{t+1}}\right] \\
&= \sum_a \pi(a|s)\sum_{s'}\sum_r p(s',r|s,a)\left[r + \gamma v_\pi(s')\right]
\end{aligned} \tag{12}
$$

Similarly, we can do the same for the Q function:

$$
\begin{aligned}
q_\pi(s,a) &= \mathbb{E}_\pi\left[G_t|S_t = s, A_t = a\right] \\
&= \mathbb{E}_\pi\left[\sum_{k=0}^{\infty}\gamma^k R_{t+k+1}|S_t = s, A_t = a\right] \\
&= \mathbb{E}_\pi\left[R_{t+1} + \sum_{k=0}^{\infty}\gamma^k R_{t+k+2}|S_t = s, A_t = a\right] \\
&= \sum_{s',r} p(s',r|s,a)\left[r + \mathbb{E}_\pi\left[\sum_{k=0}^{\infty}\gamma^k R_{t+k+2}|S_{t+1} = s'\right]\right] \\
&= \sum_{s',r} p(s',r|s,a)\left[r + \gamma V_\pi(s')\right]
\end{aligned} \tag{13}
$$

Contraction Mapping

Let (X,d) be a metric space and $f: X \to X$. We say that f is a *contraction* if there is a *Lipschitz coefficient* $k \in [0,1)$ such that for all x and y in X:

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

Contraction Mapping theorem

For complete metric space (X,d) and contraction $f: X \to X$, there is 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 $\pi(s) \in A$ for all $s \in S$,
$\Delta \leftarrow 0$
2. Policy Evaluation
while $\Delta < \theta$ *(a small positive number)* **do**
  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
end
3. Policy Improvement
policy-stable $\leftarrow$ *true*
while *not policy-stable* **do**
  foreach $s \in S$ **do**
    old-action $\leftarrow \pi(s)$
    $\pi(s) \leftarrow \operatorname{argmax}_a \sum_{s',r} p(s',r|s,a)\left[r + \gamma V(s')\right]$
    policy-stable $\leftarrow$ *old-action* $\neq \pi(s)$
  end
end

Policy iteration methods:

- gradient-based (policy gradient methods): gradient ascent
- gradient-free: simulated annealing, cross-entropy search or methods of evolutionary computation
- value search/iteration: stop after 1 state sweep
- async DP: update iteratively, no full sweeps
- generated policy iteration (GPI)

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 $\Delta < \theta$ *(a small positive number)* **do**
  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
end
ouput: Deterministic policy $\pi \approx \pi_*$ such that
$\pi(s) = \operatorname{argmax}_a \sum_{s',r} p(s',r|s,a)\left[r + \gamma V(s')\right]$

Model-free methods

Monte Carlo (MC) Methods

episodic, based on **averaging sample returns** for each state-action pair. Implementation:

Initialise 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 π
  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

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

$$V(S_t) \leftarrow V(S_t) + \alpha\left[G_t - V(S_t)\right] \tag{14}$$

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

Temporal Difference (TD)

combines DP's bootstrap (learn mid-episode) w MC's sampling. substitutes expected discounted reward G_t from the episode with an estimation:

$$V(S_t) \leftarrow V(S_t) + \alpha\left[R_{t+1} + \gamma V(S_{t+1} - V(S_t)\right] \tag{15}$$

See Sarsa below for a sample implementation.

Sarsa

Sarsa (State-action-reward-state-action) is a on-policy TD control. Can use priors. Update rule:

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

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

n-step Sarsa (n-step TD)

Define the n-step Q-Return

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

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

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

Forward View Sarsa(λ) (TD(λ))

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

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

Backward View Sarsa(λ) (TD(λ))

+ more efficient
+ can update at every time-step

- eligibility traces (per s/a pair): find cause in frequency vs recency

$$E_0(s,a) = 0$$

$$E_0(s,a) = \gamma\lambda E_{t-1}(s,a) + \not{\Vdash}(S_t = t, A_t = a)$$

$$\delta_t = R_{t+1} + \gamma Q(S_{t+1}, A_{t+1}) - Q(S_t, A_t)$$

$$Q(s,a) \Leftarrow Q(s,a) + \alpha \delta_t E_t(s,a)$$

Linear Function Approximation

general + efficient learning

update temporal difference error, minimize squared loss:

$$\delta \leftarrow r_t + \gamma \theta^T \phi(s_{t+1}) - \theta^T \phi(s_t) \theta \leftarrow \theta + \alpha\delta\phi(s_t) J(\theta) = ||\delta||^2$$

Q Learning

$$\delta \leftarrow r_t + \gamma \operatorname*{argmax}_{a\in A} Q(\phi(s_{t+1},a);\theta_i^-) - Q(\phi(s_t,a);\theta_i)$$

$$J(\theta) = ||\delta||^2$$

Update rule:

$$Q(s,a) \leftarrow Q(s,a) + \alpha\left[r + \gamma \max_{a'} Q(s',a') - Q(s,a)\right]$$

Replay Memory (update θ w SGD):

$$\nabla_{\theta_i} J(\theta_i) = \mathbb{E}_{s_t,a_t,r_t,s_{t+1}\sim D}\, \delta \nabla_{\theta_i} Q(\phi(s_t,a_t);\theta)$$

Deep Q Learning

Created by *DeepMind*, Deep Q Learning, DQL, substitutes the Q function with a deep neural network called *Q-network*. It keeps some observation in a *memory* to train the network on.

$$Y_{DQN} = r_t + \gamma \max_{a\in A} Q(\Phi(s_{t+1},a);\theta_i^-)$$

$$\nabla_i(\theta_i) = \mathbb{E}_{(s,a,r,s')\sim U(D)}\left[\underbrace{(r + \gamma \max_a Q(s',a';\theta_{i-1}) -}_{\text{target}} \underbrace{Q(s,a;\theta_i))^2}_{\text{prediction}}\right] \tag{16}$$

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 $y_i \leftarrow$
    $\begin{cases} r_j & \text{for terminal } s'_j \\ r_j + \gamma \max_a Q(s',a';\theta) & \text{for non-terminal } s'_j \end{cases}$
    Perform gradient descent step on $(y_j - Q(s_j,a_j;\Theta))^2$
    $s \leftarrow s'$
  end
end

Prioritized Replay Memory

learn esp. from high loss (traumas)

$$p(s_t,a_t,r_t,s_{t+1}) \propto r_t + \gamma \max_{a\in A} Q(\Phi(s_{t+1},a);\theta_i^-)$$

Double DQN

solve bias from reused θ, faster

$$Y_{DQN} = r_t + \gamma Q(\Phi(s_{t+1}, \operatorname*{argmax}_{a\in A} Q(\Phi(s_{t+1},a);\theta_i));\theta_i^-)$$

Policy Gradient

learn π.

+ simpler than Q or V
+ allows stochastic policy (rock-paper-scissors)
− local optimum
− less sample efficient

$$L = \mu(logp(a|s) \cdot R(s))$$

Likelihood Ratio

return state-action trajectory:

$$R(\tau) = \sum_{t=0}^{T} R(s_t,a_t)$$

expected return:

$$\begin{aligned} J(\theta) &= \mathbb{E}\left[\textstyle\sum_{t=0}^{T} R(s_t,a_t);\pi_\theta\right] \\ &= \textstyle\sum_{\tau}^{T} R(s_t,a_t);\pi_\theta \end{aligned}$$

find θ to max:

$$\max_\theta J(\theta) = \max_\theta \sum_\tau P(\tau;\theta) R(\tau)$$

REINFORCE

- Uses a policy during an episode to collect information on states, actions and rewards.
- Computes the return for each episode using the rewards collected.
- Updates the model parameters in the direction of the policy gradient.

$$\nabla_\theta J(\theta) = \sum_{t=0}^{T} \nabla_\theta \log \pi_\theta(a_t|s_t) R$$

desired loss function:

$$\frac{1}{m}\sum_{1}^{m} \nabla_\theta \log \pi_\theta(a_t|s_t) R$$

Baselined REINFORCE

use baselined rewards

$$\nabla_\theta J(\theta) = \sum_{t=0}^{T} \nabla_\theta \log \pi_\theta(a_t|s_t)(R - V_{\phi(s_t)})$$

Actor-critic

+ less variance than Baselined REINFORCE

- actor (makes policy): policy gradient
- critic: policy iteration

advantage:

$$A_\pi(s,a) = Q(s,a) - V(s) \tag{17}$$

Asynchronous Advantage Actor Critic (A3C)

- 5-step Q-Value estimation
- shares params between actor/critic

+ run parallel, one policy
+ no more need for DQN's replay policy

$$Q(s_t,a_t) = E[r_t + \gamma r_{t+1} + \gamma^2 r_{t+2} + \cdots + \gamma^n V(s_{t+n})]$$