RL Cheatsheet (Work in progress)

Alexandre Thomas
Mines ParisTech & Sorbonne University
alexandre.thomas@mines-paristech.fr

June 5, 2021

Contents

1 Bandits

The *multi-armed bandits problem* is a simplified setting of RL where actions do not affect the world state. In other words, the current state does not depend on previous actions, and the reward is immediate. Like any RL problem with unknown MDP, a successful agent should solve the *exploration-exploitation dilemma*, i.e. find a balance between exploiting what it has already learned to improve the reward and exploring in order to find the best actions.

◇ **Settings** Bandits problems can be stationary or non-stationary, and the setting can be stochastic or adversarial. Agents typically learn in an online setting and, in the case of a non-associative task (no need to associate different actions with different situations), they try to find a single best action out of a finite number of actions (also called "arms"). For associative tasks, *contextual bandits* make use of additional information which can be global or individual context (i.e. per arm). Context can be fixed or variable.

◇ **ϵ-greedy** The ϵ-greedy strategy selects a random action with probability ϵ. TODO: add equations

◇ **Upper Confidence Bounds (UCB)** UCB [1] follows an optimistic strategy and selects the best arm in the best case scenario, i.e. according to the upper bounds $B_t(i)$ on the arms value estimates:

$$\pi_t = \arg\max_i B_t(i) \quad \text{with} \quad B_t(i) = \hat{\mu_{i,t}} + \sqrt{\frac{2\log t}{\sum_{s=0}^{t} \mathbb{1}_{\pi_s=i}}}$$

LinUCB [2] follows the UCB strategy but considers a linear and individual context $x_{i,t}$. We have $\mathbb{E}[r_{i,t}|x_{i,t}] = \theta_i^T x_{i,t}$ and parameters θ_i are estimated with Ridge Regression on previously observed contexts and rewards.

◇ **Thompson Sampling** Thompson Sampling [3] follows a Bayesian approach and considers a parametric model $P(\mathcal{D}|\theta)$ with a prior $P(\theta)$. For instance, in the linear case [4]: $P(r_{i,t}|\theta) = \mathcal{N}(\theta^T x_{i,t}, v^2)$ and $P(\theta) = \mathcal{N}(0, \sigma^2)$. Then, at each iteration t, we sample θ from $P(\theta|\mathcal{D}) \propto P(\mathcal{D}|\theta)P(\theta)$ and select the arm $\pi_t = \arg\max_i \mathbb{E}[r_{i,t}|x_{i,t}, \theta]$.

2 RL Framework

◇ **Markov Decision Process (MDP)** A Markov Decision Process is a tuple (S, A, R, P, ρ_0) where

- S is the set of all valid states
- A is the set of all valid actions
- $R : S \times A \times S \to \mathbb{R}$ is the *reward function*, such that $r_t = R(s_t, a_t, s_{t+1})$. In the case the reward is stochastic and r_t is a random variable, we have $R(s, a, s') = \mathbb{E}[r_t|s_t = s, a_t = a, s_{t+1} = s']$
- $P : S \times A \times S \to [0, 1]$ the *transition probability function*, such that $P(s'|s, a)$ is the probability of transitioning into state s' if you are in state s and take action a
- $\rho_0 : S \to [0, 1]$ is the starting state distribution

◇ **Markov property** Transitions only depend on the most recent state and action, and no prior history : $P(s_{t+1}|s_t, a_t, \ldots, s_1, a_0, s_0) = P(s_{t+1}|s_t, a_t)$. This assumption does not always hold, for instance when the observed state does not contain all necessary information (*Partially Observable Markov Decision Process*), or when P and R actually depend on t (*Non-Stationary Markov Decision Process*).

When the MDP is known (e.g. small tabular environments), optimal policies can be found offline without interacting with the environment, using Dynamic Programming (DP) algorithms. But this is generally not the case and RL algorithms have to do trial-and-error search (like bandits problems), and have to deal with *delayed rewards*. Actions may affect not only the immediate reward, but also the next state and therefore all subsequent rewards.

◇ **Definitions**

- The *policy* π determines the behavior of our agent, who will take actions $a_t \sim \pi(\cdot|s_t)$. Policies can be derived from an action-value function or can be explicitly parameterized and denoted by π_θ. They can also be deterministic, in which case they are sometimes denoted by μ_θ, with $a_t = \mu_\theta(s_t)$.
- A *trajectory* $\tau = (s_0, a_0, s_1, \dots)$ is a sequence of states and actions in the world, with $s_0 \sim \rho_0$ and $s_{t+1} \sim P(\cdot|s_t, a_t)$. It is sampled from π if $a_t \sim \pi(\cdot|s_t)$ for each t. Trajectories are also called *episodes*.
- The return $R(\tau)$ is the cumulative reward over a trajectory and is the quantity to be maximized by our agent. It can refer to the *finite-horizon undiscounted return* $R(\tau) = \sum_{t=0}^{T} r_t$ or the *infinite-horizon discounted return* $R(\tau) = \sum_{t=0}^{\infty} \gamma^t r_t$ for instance. Parameter γ is called the *discount factor*.
- The *on-policy value function*: $V^\pi(s) = \mathbb{E}_{\tau\sim\pi}[R(\tau)|s_0 = s]$
- The *on-policy action-value function*: $Q^\pi(s,a) = \mathbb{E}_{\tau\sim\pi}[R(\tau)|s_0 = s, a_0 = a]$. We have $V^\pi(s) = \mathbb{E}_{a\sim\pi(\cdot|s)}[Q^\pi(s,a)]$.
- The *advantage function*: $A^\pi(s,a) = Q^\pi(s,a) - V^\pi(s)$
- The optimal value and action-value functions are obtained by acting according to an optimal policy π^*: $V^*(s) = \max_\pi V^\pi(s)$, $Q^*(s,a) = \max_\pi Q^\pi(s,a)$. We have $V^*(s) = \max_a Q^*(s,a)$, and a deterministic optimal policy can be obtained with $\pi^*(s) = \arg\max_a Q^*(s,a)$.

◇ **Bellman Equations**

$$V^\pi(s) = \mathop{\mathbb{E}}_{\substack{a\sim\pi(\cdot|s)\\ s'\sim P}}[R(s,a,s') + \gamma V^\pi(s')] \tag{1}$$

$$Q^\pi(s,a) = \mathop{\mathbb{E}}_{s'\sim P}\left[R(s,a,s') + \gamma \mathop{\mathbb{E}}_{a'\sim\pi(\cdot|s')}[Q^\pi(s',a')]\right] \tag{2}$$

$$V^*(s) = \max_a \mathop{\mathbb{E}}_{s'\sim P}[R(s,a,s') + \gamma V^*(s')] \tag{3}$$

$$Q^*(s,a) = \mathop{\mathbb{E}}_{s'\sim P}\left[R(s,a,s') + \gamma \max_{a'} Q^*(s',a')\right] \tag{4}$$

Where $s' \sim P$ is a shorthand for $s' \sim P(\cdot|s,a)$

3 Dynamic Programming

◇ **Policy Evaluation Algorithm** For (small) tabular environments with known MDP, Bellman equations can be computed exactly and one can converge on V^π by applying equation 1 repeatedly:

$$V_{i+1}(s) = \sum_{a\in A}\sum_{s'\in S} \pi(a|s)P(s'|s,a)[R(s,a,s') + \gamma V_i(s')]$$

◇ **Policy Iteration Algorithm** Being *greedy* with respect to current value function V^{π_k} makes it possible to define a better (deterministic) policy:

$$\pi_{k+1}(s) = \arg\max_a \sum_{s'\in S} P(s'|s,a)[R(s,a,s') + \gamma V^{\pi_k}(s')] \qquad \textit{policy improvement}$$

In the policy iteration algorithm, the policy evaluation algorithm is applied until convergence to V^{π_k} and followed by one policy improvement step. This is repeated until convergence to π^* (characterized by stationarity). The idea of having these two policy evaluation and improvement processes interact is found in many RL algorithms (not necessarily as separate steps, but also simultaneously) and is called *general policy iteration*.

◇ **Value Iteration Algorithm** Equation 3 is applied repeatedly to converge directly on V^*:

$$V_{i+1}(s) = \max_{a \in A} \sum_{s' \in S} P(s'|s,a)[R(s,a,s') + \gamma V_i(s')]$$

And π^* is obtained by being greedy to V^*, which is possible since we know the MDP. Q-Value iteration (with equation 4) requires storing more values but is possible as well, and makes model-free approaches possible when the MDP is unknown (Value-Based section).

4 Value-Based

Value-based methods typically aim at finding Q^* first, which then gives the optimal policy $\pi^*(s) = \arg\max_a Q^*(s,a)$. For this, they require finite action spaces.

4.1 Tabular environments

In the case of tabular environments (i.e. S is finite as well), Q^π can actually be represented by a matrix.

◇ **Tabular Q-learning** Starting from a random value-action function Q, tabular Q-learning consists in interacting with the environment and applying equation 4 to converge to Q^* (algorithm 1). Q-learning is an *off-policy* method, i.e. learns the value of the optimal policy independently of the agent's actions, and π can be any policy as long as all state-action pairs are visited enough.

Algorithm 1: Tabular Q-learning

for *a number of episodes* **do**
  initialize s_0
  while *episode not done* **do**
    take action $a_t \sim \pi(s_t)$, observe r_t and s_{t+1}
    $Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha \left[r_t + \gamma \max_a Q(s_{t+1}, a) \right]$

◇ **SARSA** The SARSA algorithm is the *on-policy* equivalent of Q-learning as it learns the value of the current agent's policy π. Instead of using Bellman equation 4, it uses equation 2 for the update step, and converges to the optimal policy π^* as long as policy π becomes greedy in the limit.

◇ **Exploration strategies** Used by value-based algorithms (SARSA, Q-learning) to balance between exploration and exploitation when interacting with the environment.

- *ϵ-greedy strategy*

$$\pi(s) = \begin{cases} \arg\max_a Q(s,a) & \text{be greedy with probability } \epsilon \\ a \textit{ random action}, & \text{otherwise} \end{cases}$$

- *Boltzmann selection*

$$\pi(a|s) = \frac{\exp(Q(s,a)/\tau)}{\sum_{a'} \exp(Q(s,a')/\tau)}$$

- Others: UCB-1, pursuit strategy. Comparison in [5] (for stochastic mazes).

◇ **Forward view: TD learning** Temporal Difference methods combine Monte-Carlo (MC) sampling [1] with the *bootstrapping* of DP methods [2]. They can be used to learn V^π or Q^π. The following quantity

$$\delta_t = r_{t+1} + \gamma V(s_{t+1}) - V(s_t)$$

is called the *TD-error*.

[1] Since we don't know the MDP, we have to approximate the expectation using trajectory samples
[2] Bootstrapping: using our current approximation of $V^\pi(s')$ to estimate $V^\pi(s)$

- *TD(0) learning* is the most straightforward method. After acting according to π, the value function is updated with:
$$V(s_t) \leftarrow V(s_t) + \alpha[r_t + \gamma V(s_{t+1}) - V(s_t)]$$
Since $\mathbb{E}[r_t + \gamma V^\pi(s_{t+1})] = V^\pi(s_t)$, V will eventually converge to V^π.

- *Multi-steps learning.* When the reward is delayed and is observed several steps after the decisive action, TD(0) is slow since values only propagate from one state to the previous one. With:
$$G_t^{(n)} = r_t + \gamma r_{t+1} + \cdots + \gamma^{n-1} r_{t+n} + \gamma^n V^\pi(s_{t+n})$$
the *n-step return*, it is also true that $\mathbb{E}[G_t^{(n)}] = V^\pi(s_t)$. Therefore, the update steps becomes:
$$V(s_t) \leftarrow V(s_t) + \alpha[G_t^{(n)} - V(s_t)]$$
Multi-steps learning can converge faster depending on the problem. As $n \to \infty$, this update approaches the unbiased MC estimate of V^π, at the cost of higher variance.

- *TD(λ) learning.* Instead of choosing the right n, a better way to navigate between TD(0) and MC updates is to average all n-step returns with a decay λ. For $\lambda \in [0, 1)$, the *λ-return* is defined as
$$G_t^\lambda = (1-\lambda) \sum_{n=1}^{\infty} \lambda^{n-1} G_t^{(n)}$$
and also verifies $\mathbb{E}[G_t^\lambda] = V^\pi(s_t)$, therefore can be used in the update step. Choosing $\lambda = 0$ is equivalent to TD(0) updates, and $\lambda \to 1$ approaches MC updates.

◇ **Backward view: eligibility traces** Eligibility traces are an equivalent but more convenient way of implementing TD(λ) learning. Updates are done online instead of having to wait the end of the trajectory.

Algorithm 2: Eligibility traces, also called online TD(λ), tabular version

for *a number of episodes* **do**
  initialize s_0 and set $e_0(s) = 0, \forall s$
  while *episode not done* **do**
    take action $a_t = \pi(s_t)$, observe r_t and s_{t+1}
    for *all s* **do**
      $e_t(s) \leftarrow \lambda\gamma e_{t-1}(s) + \mathbb{1}_{s_t=s}$ `// we have` $e_t(s) = \sum_{k=0}^{t} (\lambda\gamma)^{t-k} \mathbb{1}_{s_k=s}$
      $V(s_t) \leftarrow V(s_t) + \alpha e_t(s) \delta_t$

TODO: version with weights

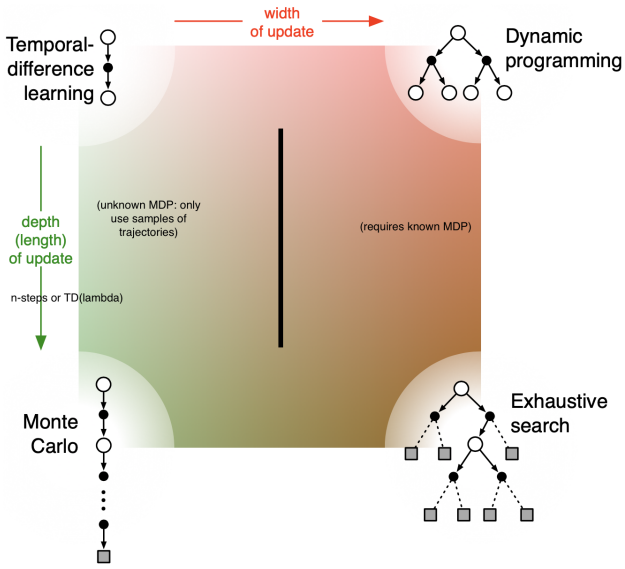

Figure 1: Taken from [6]

4.2 Approximate Q-learning

Tabular learning approaches have trouble learning in large environments since there is no generalization between similar situations, and storing Q or V can even be an issue. Approximate approaches allow working with large finite environments and even with continuous state space S, by considering

$$Q(s,a) \approx Q_\theta(\phi(s),a)$$

with $\phi(s) \in \mathbb{R}^d$ a vector of features. Features ϕ can be hand-crafted, but in Deep Reinforcement Learning (DRL) they are typically learned using neural networks and we simply note $Q(s,a) \approx Q_\theta(s,a)$.

◇ **Deep Q-Network (DQN) [7][8]** DQN was the first successful attempt at applying DRL on high-dimensional state spaces, and uses 2D convolutions with an MLP to extract features. It overcomes stability issues thanks to:

- *Experience-replay.* Within a trajectory, transitions are strongly correlated but gradient descent algorithms typically assume independent samples, otherwise gradient estimates might be biased. Storing transitions and sampling from a memory buffer D reduces correlation, and even allows mini-batch optimization to speed up training. Re-using past transitions also limits the risk of catastrophic forgetting.
- *Target network.* Based on eq. 4, values $Q_\theta(s,a)$ can be learned by minimizing the error[3] to the target $r + \gamma \max_{a'} Q_\theta(s',a')$. In this case, θ is continuously updated and we are chasing a non-stationary target. Learning can be stabilized by using a separate network Q_{θ^-} called the target network, with weights θ^- updated every k steps to match θ.

[3]This can be the mean square error, or Huber loss for more stability

Algorithm 3: Deep Q-learning with experience replay and target network (DQN)

Init: replay memory D with capacity M, Q_θ with random weights, $\theta^- = \theta$
for *a number of episodes* **do**
  initialize s_0
  while *episode not done* **do**
    take action $a_t \sim \pi(s_t)$, observe r_t and s_{t+1} `// π can be ϵ-greedy for instance`
    store transition (s_t, a_t, r_t, s_{t+1}) in D
    sample random mini-batch of transitions $((s_j, a_j, r_j, s_{j+1}))_{j=1,\dots,N}$ from D
    set $y_j = \begin{cases} r_j & \text{if } s_{j+1} \text{ is a terminal state} \\ r_j + \gamma \max_{a'} Q_{\theta^-}(s_{j+1}, a') & \text{otherwise} \end{cases}$
    $\theta \leftarrow \theta - \alpha \nabla_\theta \frac{1}{N} \sum_{j=1}^{N} (y_j - Q_\theta(s_j, a_j))^2]$
    every k steps, update $\theta^- = \theta$

◇ **Prioritized Experience Replay (PER) [9]** Improve experience replay: rather than sampling transitions t_i uniformly from the memory buffer, prioritize the ones that are the most informative, i.e. with the largest error.

$$i \sim P(i) = \frac{p_i^\alpha}{\sum_k p_k^\alpha} \quad \text{with} \quad p_i = |\delta_i| + \epsilon$$

And $\delta_t = r_t + \gamma \max_a Q(s_{t+1}, a) - Q(s_t, a_t)$ the TD-error again. In practice when implementing PER, errors δ_i are stored in memory with their associated transition t_i and are only updated with current Q_θ when t_i is sampled. A SumTree structure can be used to efficiently sample from $P(i)$, in $O(\log |D|)$. Since PER introduces a bias[4], authors use *importance sampling* and correct the loss with a term $w_i = 1/(NP(i))^\beta$.

◇ **Double DQN [10]** Because it is using the same value function Q_θ for both action selection and evaluation in the target $y_t = r_t + \gamma \max_a Q_\theta(s_{t+1}, a)$, Q-learning tends to overestimate action values as soon as there is any estimation error. This is particularly the case when the action space is large (figure 2). Double Q-learning [11] decouples action selection and evaluation using 2 parallel networks Q_θ and $Q_{\theta'}$. Double DQN [10] improves DQN performance and stability by doing it simply with the online network Q_θ and target network Q_{θ^-}:

$$y_t = r_t + \gamma Q_{\theta^-}(s_{t+1}, \arg\max_a Q_\theta(s_{t+1}, a))$$

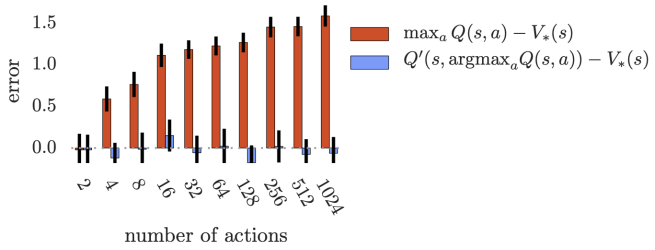

Figure 2: Taken from [10]

◇ **Rainbow [12]** Rainbow studies and combines a number of improvements to the DQN algorithm: multi-steps returns, prioritized experience replay, Double Q-learning, as well as dueling networks (using a value function $V_\theta(s)$ and an advantage function $A_\theta(s, a)$ to estimate $Q(s, a)$), noisy nets (adaptive exploration by adding noise to the parameters of the last layer, instead of being ϵ-greedy) and distributional RL (approximate the distribution of returns instead of the expected return).

◇ **Deep Recurrent Q-Network [13]** By using a RNN to learn the features (e.g. after the convolutional layers), DQRN keeps in memory a representation of the world according to previous observations. This makes it possible to go beyond the Markov property and work with POMDPs, but it can be harder to train.

[4] This blog post goes into more details

5 Policy Gradients

Policy gradient algorithms directly optimize the policy $\pi_\theta(\cdot|s)$. The goal is to maximize the expected return under π_θ:

$$\max_\theta J(\theta) = \mathop{\mathbb{E}}_{\tau\sim\pi_\theta}[R(\tau)]$$

Using the log-derivative trick ($\nabla f = f\nabla \log f$) and the likelihood of a trajectory τ:

$$\pi_\theta(\tau) = \rho_0(s_0)\prod_{t=0}^{T} P(s_{t+1}|s_t, a_t)\pi_\theta(a_t|s_t)$$

the *policy gradient* can be derived:

$$\begin{aligned}\nabla_\theta J(\theta) &= \mathop{\mathbb{E}}_{\tau\sim\pi_\theta}[\nabla_\theta \log \pi_\theta(\tau)R(\tau)] \\ &= \mathop{\mathbb{E}}_{\tau\sim\pi_\theta}\left[\sum_{t=0}^{T} R(\tau)\nabla_\theta \log \pi_\theta(a_t|s_t)\right] \end{aligned} \tag{5}$$

Learning the optimal policy can be easier than learning all the action values $Q(s, a)$, and directly optimizing π_θ makes it possible to have smoother updates and more stable convergence than Q-learning. In addition, all policy gradient algorithms work with continuous/infinite action spaces, and can encourage exploration with additional rewards (entropy). However, they can be less adapted to tabular environments and are often less sample-efficient than value-based approaches, which can use memory buffers.

5.1 On-Policy

◇ **REINFORCE** Based on eq. 5, the REINFORCE algorithm simply uses a Monte-Carlo estimate of $\nabla_\theta J(\theta)$ to optimize π_θ, and increase the likelihood of trajectories with high rewards.

Algorithm 4: Vanilla REINFORCE

for *a number of epochs* **do**
  while *not enough samples* **do**
    collect trajectory $\tau_i = (s_0^i, a_0^i, \dots, s_{T_i}^i)$ by running active policy π_θ
  $\hat{g} \leftarrow \sum_i \sum_t R(\tau^i)\nabla_\theta \log \pi_\theta(a_t^i|s_t^i)$ `// compute policy gradient estimate`
  $\theta \leftarrow \theta + \alpha\hat{g}$

It is unbiased but typically has very high variance and is slow to converge. Variance can be reduced using alternatives expressions of the gradient (all three can be combined):

- *Causality (don't let the past distract you)* Intuitively, rewards obtained before taking an action do not bring any information, and are just noise. And indeed we can use the *reward-to-go* $R_t(\tau) = \sum_{t'=t}^{T} r'_t$ instead of $R(\tau)$ in the policy gradient expression.
- *Baseline* For any function $b : \mathcal{S} \to \mathbb{R}$, we have:

  $$\mathop{\mathbb{E}}_{\tau\sim\pi_\theta}\left[\sum_{t=0}^{T} b(s_t)\nabla_\theta \log \pi_\theta(a_t|s_t)\right] = 0$$

  Hence we can use $(R(\tau) - b(s_t))$ instead of $R(\tau)$, and the variance is minimal when choosing $b(s) = V^\pi(s)$ (e.g. by fitting $b(s_t)$ to $R_t(\tau)$).
- *Discount* Using a discounted return $R(\tau) = \sum_{t=0}^{T} \gamma r_t$ can also reduce the variance but, unlike the 2 previous techniques, it makes the gradient biased.

More generally, the policy gradient can be expressed as:

$$\nabla_\theta J(\theta) = \mathop{\mathbb{E}}_{\tau\sim\pi_\theta}\left[\sum_{t=0}^{T} \Psi_t \nabla_\theta \log \pi_\theta(a_t|s_t)\right] \tag{6}$$

Where Ψ_t may be the reward $R(\tau)$, the reward-to-go $R_t(\tau)$, with or without a baseline (e.g. $R_t(\tau) - b(s_t)$).

◇ **Actor-Critic** The policy gradient (eq. 6) can be equivalently expressed with:

- $\Psi_t = Q^\pi(s_t, a_t)$: state-action value function
- $\Psi_t = A^\pi(s_t, a_t)$: advantage function
- $\Psi_t = r_t + \gamma V^\pi(s_{t+1}) - V^\pi(s_t)$: TD error

Actor-critic methods combine the strengths of policy gradient and value-based methods by estimating a critic (Q^π, V^π and/or A^π) in order to better optimize the actor (policy π_θ). Adding a critic reduces the variance, but estimation errors introduce a bias as soon as the critic is not perfect.

◇ **Actor-Critic with compatible functions**

◇ **Actor-Critic with Generalized Advantage Estimation (GAE) [14]** This method only requires estimating V^π (easier to learn than $Q^\pi(s,a) \forall s,a$, especially in high dimension) to obtain an estimate of A^π. Similar to multi-steps learning, we can define the *n-steps advantage estimate*:

$$A_t^{(n)} = r_t + \gamma r_{t+1} + \cdots + \gamma^{n-1} r_{t+n-1} + \gamma^n V^\pi(s_{t+n}) - V^\pi(s_t)$$

and control the bias-variance trade-off with n ($n = 1$: lower variance but higher bias when the estimate of V^π is wrong, $n \to \infty$: Monte-Carlo estimate, no bias but higher variance). Like TD(λ) learning, GAE averages the estimates and uses $\lambda \in [0, 1]$ to control the trade-off:

$$A_t^{GAE(\gamma,\lambda)} = (1-\lambda) \sum_{n=0}^{\infty} \lambda^n A_t^{(n)} = \sum_{n=0}^{\infty} (\lambda\gamma)^n \delta_{t+n}$$

Here as well, eligibility traces are a way to implement GAE(γ, λ)

◇ **A2C / A3C**

◇ **Entropy**

◇ **Policy Performance Bounds**

◇ **Trust Region Policy Optimization (TRPO)**

◇ **Proximal Policy Optimization (PPO)**

◇ **Natural Gradient with compatible functions**

◇ **ACKTR**

5.2 Off-Policy

◇ **Off-policy policy gradient theorem**

◇ **Estimating Q^π**

◇ **Actor-Critic with Experience Replay (ACER)**

◇ **Trajectory-based off-policy: weighted importance sampling**

◇ **Importance Sampling on the stationary state distribution**

◇ **Deterministic Policy Gradients (DPG/DDPG)**

◇ **Policy gradient with an off-policy critic (Q-Prop)**

References

[1] Peter Auer. Using confidence bounds for exploitation-exploration trade-offs. *Journal of Machine Learning Research*, 3(Nov):397–422, 2002.

[2] Lihong Li, Wei Chu, John Langford, and Robert E Schapire. A contextual-bandit approach to personalized news article recommendation. In *Proceedings of the 19th international conference on World wide web*, pages 661–670, 2010.

[3] Emilie Kaufmann, Nathaniel Korda, and Rémi Munos. Thompson sampling: An asymptotically optimal finite-time analysis. In *International conference on algorithmic learning theory*, pages 199–213. Springer, 2012.

[4] Shipra Agrawal and Navin Goyal. Thompson sampling for contextual bandits with linear payoffs. In *International Conference on Machine Learning*, pages 127–135. PMLR, 2013.

[5] Arryon D Tijsma, Madalina M Drugan, and Marco A Wiering. Comparing exploration strategies for q-learning in random stochastic mazes. In *2016 IEEE Symposium Series on Computational Intelligence (SSCI)*, pages 1–8. IEEE, 2016.

[6] Richard S Sutton and Andrew G Barto. *Reinforcement learning: An introduction*. MIT press, 2018.

[7] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin Riedmiller. Playing atari with deep reinforcement learning. *arXiv preprint arXiv:1312.5602*, 2013.

[8] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A Rusu, Joel Veness, Marc G Bellemare, Alex Graves, Martin Riedmiller, Andreas K Fidjeland, Georg Ostrovski, et al. Human-level control through deep reinforcement learning. *nature*, 518(7540):529–533, 2015.

[9] Tom Schaul, John Quan, Ioannis Antonoglou, and David Silver. Prioritized experience replay. *arXiv preprint arXiv:1511.05952*, 2015.

[10] Hado Van Hasselt, Arthur Guez, and David Silver. Deep reinforcement learning with double q-learning. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 30, 2016.

[11] Hado Hasselt. Double q-learning. *Advances in neural information processing systems*, 23:2613–2621, 2010.

[12] Matteo Hessel, Joseph Modayil, Hado Van Hasselt, Tom Schaul, Georg Ostrovski, Will Dabney, Dan Horgan, Bilal Piot, Mohammad Azar, and David Silver. Rainbow: Combining improvements in deep reinforcement learning. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 32, 2018.

[13] Matthew Hausknecht and Peter Stone. Deep recurrent q-learning for partially observable mdps. *arXiv preprint arXiv:1507.06527*, 2015.

[14] John Schulman, Philipp Moritz, Sergey Levine, Michael Jordan, and Pieter Abbeel. High-dimensional continuous control using generalized advantage estimation. *arXiv preprint arXiv:1506.02438*, 2015.