Introduction to Deep Reinforcement Learning

Milan Straka

May 11, 2020

Charles University in Prague
Faculty of Mathematics and Physics
Institute of Formal and Applied Linguistics

EUROPEAN UNION
European Structural and Investment Fund
Operational Programme Research, Development and Education

Reinforcement Learning

History of Reinforcement Learning

Develop goal-seeking agent trained using reward signal.

- *Optimal control* in 1950s – Richard Bellman
- Trial and error learning – since 1850s
  - Law and effect – Edward Thorndike, 1911
    - Responses that produce a satisfying effect in a particular situation become more likely to occur again in that situation, and responses that produce a discomforting effect become less likely to occur again in that situation
  - Shannon, Minsky, Clark&Farley, … – 1950s and 1960s
  - Tsetlin, Holland, Klopf – 1970s
  - Sutton, Barto – since 1980s
- Arthur Samuel – first implementation of temporal difference methods for playing checkers
- Gerry Tesauro – 1992, human-level Backgammon playing program trained solely by self-play

Notable Successes of Reinforcement Learning

- IBM Watson in Jeopardy – 2011
- Human-level video game playing (DQN) – 2013 (2015 Nature), Mnih. et al, Deepmind
  - 29 games out of 49 comparable or better to professional game players
  - 8 days on GPU
  - human-normalized mean: 121.9%, median: 47.5% on 57 games
- A3C – 2016, Mnih. et al
  - 4 days on 16-threaded CPU
  - human-normalized mean: 623.0%, median: 112.6% on 57 games
- Rainbow – 2017
  - human-normalized median: 153%; ~39 days of game play experience
- Impala – Feb 2018
  - one network and set of parameters to rule them all
  - human-normalized mean: 176.9%, median: 59.7% on 57 games
- PopArt-Impala – Sep 2018
  - human-normalized median: 110.7% on 57 games; 57*38.6 days of experience

Notable Successes of Reinforcement Learning

- R2D2 – Jan 2019
  - human-normalized mean: 4024.9%, median: 1920.6% on 57 games
  - processes ~5.7B frames during a day of training
- MuZero – Nov 2019
  - planning with a learned model: 4999.2%, median: 2041.1%
- Data-efficient Rainbow – Jun 2019
  - learning from ~2 hours of game experience

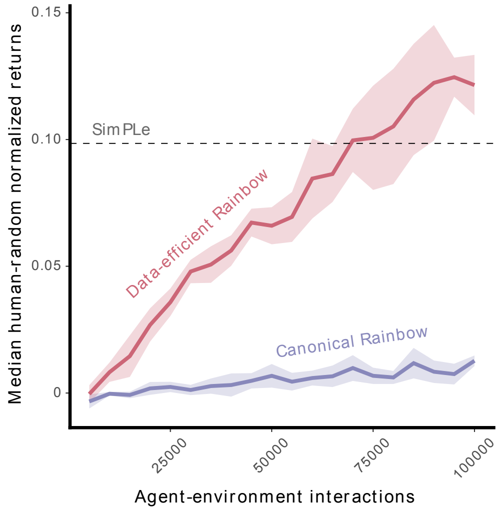


Figure 3 of the paper "When to use parametric models in reinforcement learning?" by Hado van Hasselt et al.

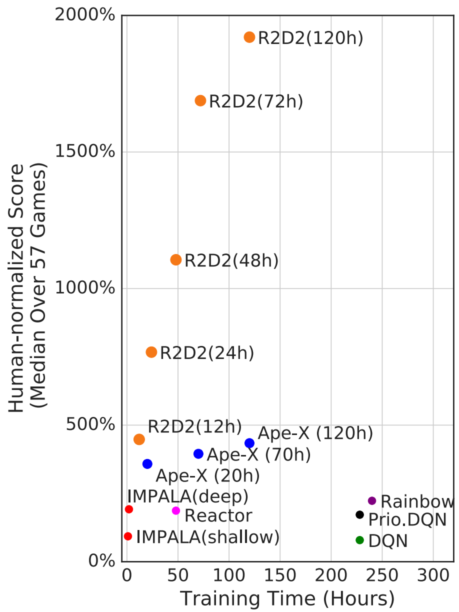


Figure 2 of the paper "Recurrent Experience Replay in Distributed Reinforcement Learning" by Steven Kapturowski et al.

Notable Successes of Reinforcement Learning

- AlphaGo
  - Mar 2016 – beat 9-dan professional player Lee Sedol
- AlphaGo Master – Dec 2016
  - beat 60 professionals, beat Ke Jie in May 2017
- AlphaGo Zero – 2017
  - trained only using self-play
  - surpassed all previous version after 40 days of training
- AlphaZero – Dec 2017 (Dec 2018 in Nature)
  - self-play only, defeated AlphaGo Zero after 30 hours of training
  - impressive chess and shogi performance after 9h and 12h, respectively

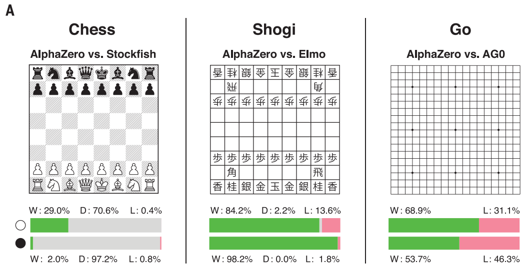


Figure 2 of the paper "A general reinforcement learning algorithm that masters chess, shogi, and Go through self-play" by David Silver et al.

Notable Successes of Reinforcement Learning

- Dota2 – Aug 2017
  - won 1v1 matches against a professional player
- MERLIN – Mar 2018
  - unsupervised representation of states using external memory
  - beat human in unknown maze navigation
- FTW – Jul 2018
  - beat professional players in two-player-team Capture the flag FPS
  - solely by self-play, trained on 450k games
    - each 5 minutes, 4500 agent steps (15 per second)
- OpenAI Five – Aug 2018
  - won 5v5 best-of-three match against professional team
  - 256 GPUs, 128k CPUs; 180 years of experience per day
- AlphaStar – 2019
  - Jan: won 10 out of 11 StarCraft II games against two professional players
  - Oct: in the full game of StarCraft II, got above 99.8% of oficially ranked players

Notable Successes of Reinforcement Learning

- Neural Architecture Search – since 2017
  - automatically designing CNN image recognition networks surpassing state-of-the-art performance
  - AutoML: automatically discovering
    - architectures (CNN, RNN, overall topology)
    - activation functions
    - optimizers
    - ...
- System for automatic control of data-center cooling – 2017
- AlphaFold won CASP13, a global competition in protein structure prediction

Multi-armed Bandits

http://www.infoslotmachine.com/img/one-armed-bandit.jpg

Multi-armed Bandits

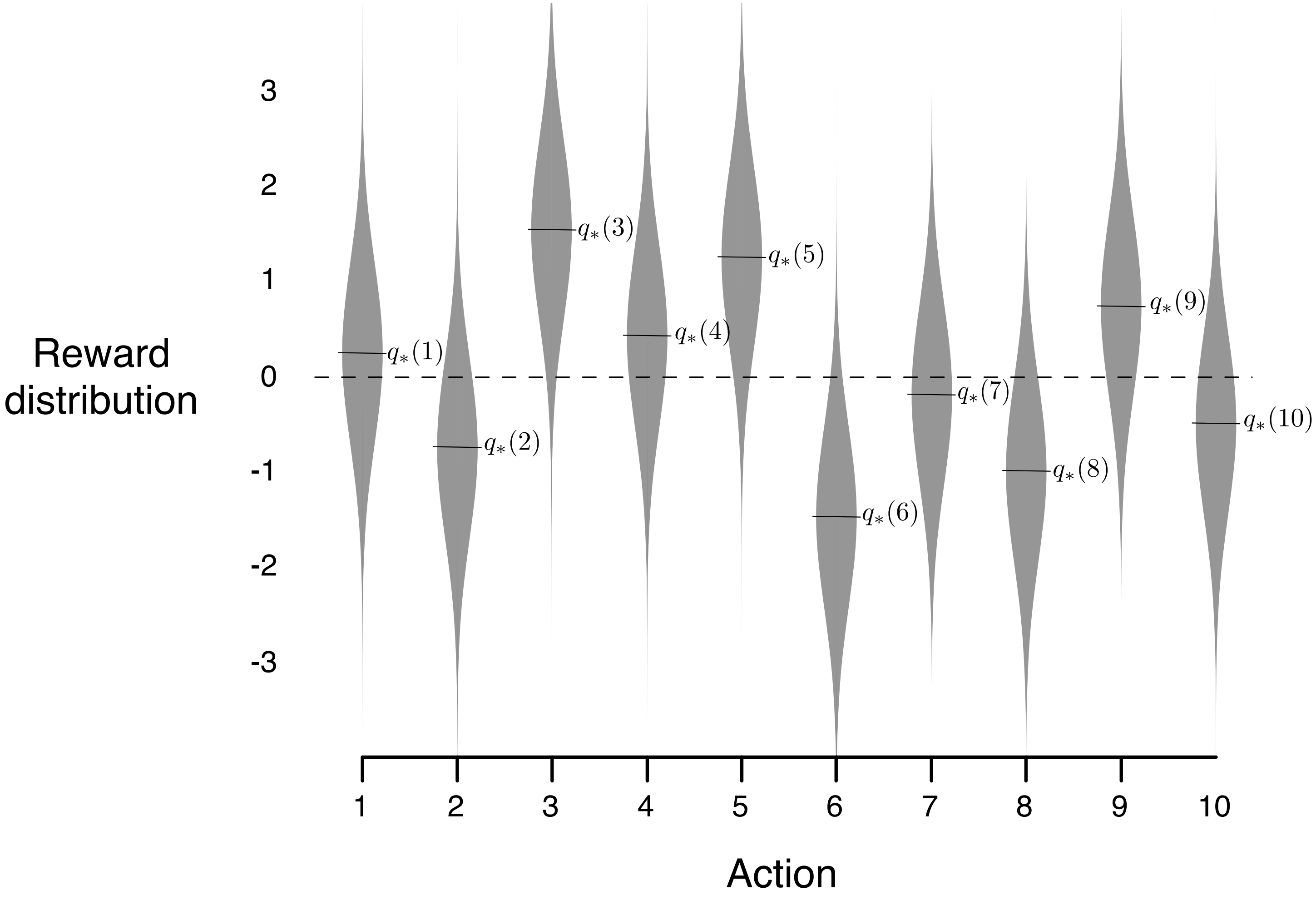


Figure 2.1 of "Reinforcement Learning: An Introduction, Second Edition".

Multi-armed Bandits

We start by selecting action A_1, which is the index of the arm to use, and we get a reward of R_1. We then repeat the process by selecting actions A_2, A_3, ...

Let $q_*(a)$ be the real *value* of an action a:

$$q_*(a) = \mathbb{E}[R_t | A_t = a].$$

Denoting $Q_t(a)$ our estimated value of action a at time t (before taking trial t), we would like $Q_t(a)$ to converge to $q_*(a)$. A natural way to estimate $Q_t(a)$ is

$$Q_t(a) \stackrel{\text{def}}{=} \frac{\text{sum of rewards when action } a \text{ is taken}}{\text{number of times action } a \text{ was taken}}.$$

Following the definition of $Q_t(a)$, we could choose a *greedy action* A_t as

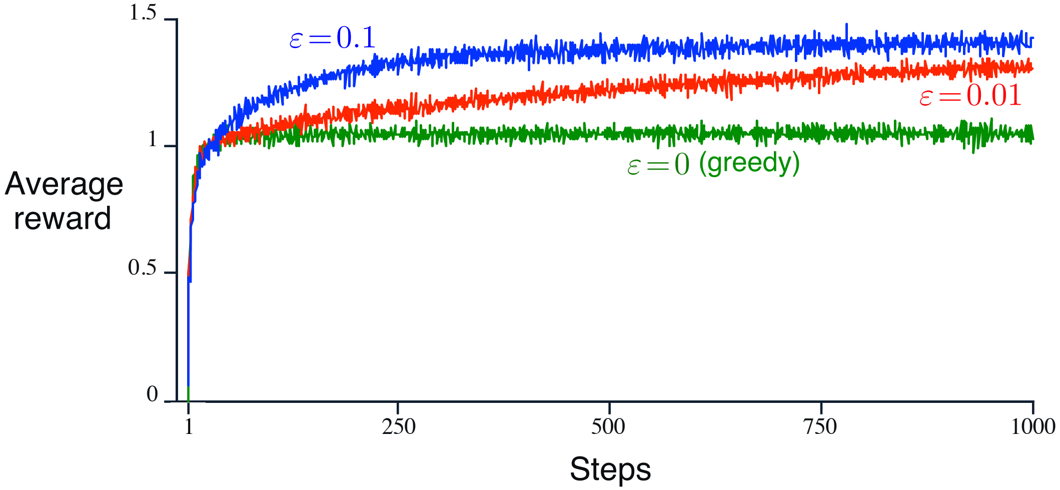
$$A_t \stackrel{\text{def}}{=} \arg\max_a Q_t(a).$$

Exploitation versus Exploration

Choosing a greedy action is *exploitation* of current estimates. We however also need to *explore* the space of actions to improve our estimates.

An ε-*greedy* method follows the greedy action with probability $1 - \varepsilon$, and chooses a uniformly random action with probability ε.

ε-greedy Method



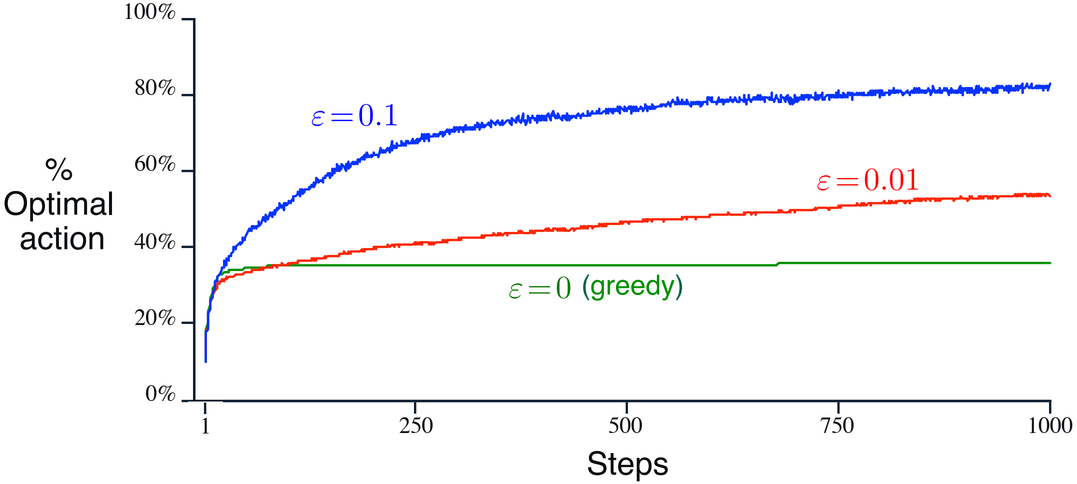


Figure 2.2 of "Reinforcement Learning: An Introduction, Second Edition".

Incremental Implementation

Let Q_{n+1} be an estimate using n rewards $R_1, \ldots, R_n$.

$$
\begin{aligned}
Q_{n+1} &= \frac{1}{n}\sum_{i=1}^{n} R_i \\
&= \frac{1}{n}(R_n + \frac{n-1}{n-1}\sum_{i=1}^{n-1} R_i) \\
&= \frac{1}{n}(R_n + (n-1)Q_n) \\
&= \frac{1}{n}(R_n + nQ_n - Q_n) \\
&= Q_n + \frac{1}{n}\Big(R_n - Q_n\Big)
\end{aligned}
$$

ε-greedy Method Algorithm

A simple bandit algorithm

Initialize, for $a = 1$ to k:

$Q(a) \leftarrow 0$

$N(a) \leftarrow 0$

Loop forever:

$$A \leftarrow \begin{cases} \arg\max_a Q(a) & \text{with probability } 1 - \varepsilon \quad \text{(breaking ties randomly)} \\ \text{a random action} & \text{with probability } \varepsilon \end{cases}$$

$R \leftarrow bandit(A)$

$N(A) \leftarrow N(A) + 1$

$Q(A) \leftarrow Q(A) + \frac{1}{N(A)}\big[R - Q(A)\big]$

Algorithm 2.4 of "Reinforcement Learning: An Introduction, Second Edition".

Markov Decision Process

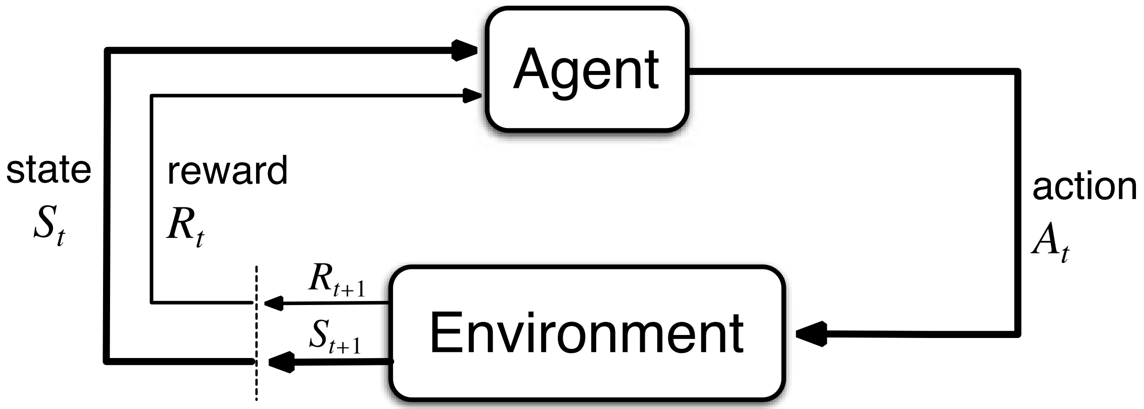


Figure 3.1 of "Reinforcement Learning: An Introduction, Second Edition".

A *Markov decision process* (MDP) is a quadruple $(\mathcal{S}, \mathcal{A}, p, \gamma)$, where:

- $\mathcal{S}$ is a set of states,
- $\mathcal{A}$ is a set of actions,
- $p(S_{t+1} = s', R_{t+1} = r | S_t = s, A_t = a)$ is a probability that action $a \in \mathcal{A}$ will lead from state $s \in \mathcal{S}$ to $s' \in \mathcal{S}$, producing a *reward* $r \in \mathbb{R}$,
- $\gamma \in [0, 1]$ is a *discount factor* (we will always use $\gamma = 1$ and finite episodes in this course).

Let a *return* G_t be $G_t \stackrel{\text{def}}{=} \sum_{k=0}^{\infty} \gamma^k R_{t+1+k}$. The goal is to optimize $\mathbb{E}[G_0]$.

Multi-armed Bandits as MDP

To formulate n-armed bandits problem as MDP, we do not need states. Therefore, we could formulate it as:

- one-element set of states, $\mathcal{S} = \{S\}$;
- an action for every arm, $\mathcal{A} = \{a_1, a_2, \ldots, a_n\}$;
- assuming every arm produces rewards with a distribution of $\mathcal{N}(\mu_i, \sigma_i^2)$, the MDP dynamics function p is defined as

$$p(S, r | S, a_i) = \mathcal{N}(r | \mu_i, \sigma_i^2).$$

One possibility to introduce states in multi-armed bandits problem is to have separate reward distribution for every state. Such generalization is usually called *Contextualized Bandits* problem. Assuming that state transitions are independent on rewards and given by a distribution $\mathit{next}(s)$, the MDP dynamics function for contextualized bandits problem is given by

$$p(s', r | s, a_i) = \mathcal{N}(r | \mu_{i,s}, \sigma_{i,s}^2) \cdot \mathit{next}(s' | s).$$

Episodic and Continuing Tasks

If the agent-environment interaction naturally breaks into independent subsequences, usually called *episodes*, we talk about **episodic tasks**.

In episodic tasks, it is often the case that every episode ends in at most H steps. These *finite-horizont tasks* then can use discount factor $\gamma = 1$, because the return $G \stackrel{\text{def}}{=} \sum_{t=0}^{H} \gamma^t R_{t+1}$ is well defined.

If the agent-environment interaction goes on and on without a limit, we instead talk about **continuing tasks**. In this case, the discount factor γ needs to be sharply smaller than 1.

(State-)Value and Action-Value Functions

A *policy* π computes a distribution of actions in a given state, i.e., $\pi(a|s)$ corresponds to a probability of performing an action a in state s.

To evaluate a quality of a policy, we define *value function* $v_\pi(s)$, or *state-value function*, as

$$v_\pi(s) \stackrel{\text{def}}{=} \mathbb{E}_\pi\left[G_t \middle| S_t = s\right] = \mathbb{E}_\pi\left[\sum\nolimits_{k=0}^\infty \gamma^k R_{t+k+1} \middle| S_t = s\right].$$

An *action-value function* for a policy π is defined analogously as

$$q_\pi(s, a) \stackrel{\text{def}}{=} \mathbb{E}_\pi\left[G_t \middle| S_t = s, A_t = a\right] = \mathbb{E}_\pi\left[\sum\nolimits_{k=0}^\infty \gamma^k R_{t+k+1} \middle| S_t = s, A_t = a\right].$$

Evidently,

$$\begin{aligned} v_\pi(s) &= \mathbb{E}_\pi[q_\pi(s, a)], \\ q_\pi(s, a) &= \mathbb{E}_\pi[R_{t+1} + \gamma v_\pi(S_{t+1}) | S_t = s, A_t = a]. \end{aligned}$$

Optimal state-value function is defined as

$$v_*(s) \stackrel{\text{def}}{=} \max_\pi v_\pi(s),$$

analogously

$$q_*(s,a) \stackrel{\text{def}}{=} \max_\pi q_\pi(s,a).$$

Any policy π_* with $v_{\pi_*} = v_*$ is called an *optimal policy*. Such policy can be defined as $\pi_*(s) \stackrel{\text{def}}{=} \arg\max_a q_*(s,a) = \arg\max_a \mathbb{E}[R_{t+1} + \gamma v_*(S_{t+1}) | S_t = s, A_t = a]$. When multiple actions maximize $q_*(s,a)$, the optimal policy can stochastically choose any of them.

Existence

In finite-horizon tasks or if $\gamma < 1$, there always exists a unique optimal state-value function, unique optimal action-value function, and (not necessarily unique) optimal policy.

Monte Carlo Methods

We now present the first algorithm for computing optimal policies without assuming a knowledge of the environment dynamics.

However, we still assume there are finitely many states $\mathcal{S}$, finitely many actions $\mathcal{A}$ and we will store estimates for every possible state-action pair.

Monte Carlo methods are based on estimating returns from complete episodes. Furthermore, if the model (of the environment) is not known, we need to estimate returns for the action-value function q instead of v.

Monte Carlo and ε-soft Policies

For the estimates to converge to the real values, every reachable state must be visited infinitely many times and every action in such a state must be selected infinitely many times in limit.

A policy is called ε-soft, if

$$\pi(a|s) \geq \frac{\varepsilon}{|\mathcal{A}(s)|}.$$

We call a policy ε-greedy, if one action has maximum probability of $1 - \varepsilon + \frac{\varepsilon}{|A(s)|}$.

It can be shown that when considering the class of ε-soft policies, one of the optimal policies is always ε-greedy – we will therefore search among the ε-greedy policies only.

On-policy every-visit Monte Carlo for ε-soft Policies

Algorithm parameter: small $\varepsilon > 0$

Initialize $Q(s, a) \in \mathbb{R}$ arbitrarily (usually to 0), for all $s \in \mathcal{S}, a \in \mathcal{A}$
Initialize $C(s, a) \in \mathbb{Z}$ to 0, for all $s \in \mathcal{S}, a \in \mathcal{A}$

Repeat forever (for each episode):

- Generate an episode $S_0, A_0, R_1, \ldots, S_{T-1}, A_{T-1}, R_T$, by generating actions as follows:
  - With probability ε, generate a random uniform action
  - Otherwise, set $A_t \stackrel{\text{def}}{=} \arg\max_a Q(S_t, a)$
- $G \leftarrow 0$
- For each $t = T-1, T-2, \ldots, 0$:
  - $G \leftarrow \gamma G + R_{t+1}$
  - $C(S_t, A_t) \leftarrow C(S_t, A_t) + 1$
  - $Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \frac{1}{C(S_t, A_t)}(G - Q(S_t, A_t))$

Policy Gradient Methods

Instead of predicting expected returns, we could train the method to directly predict the policy

$$\pi(a|s;\boldsymbol\theta).$$

Obtaining the full distribution over all actions would also allow us to sample the actions according to the distribution π instead of just ε-greedy sampling.

However, to train the network, we maximize the expected return $v_\pi(s)$ and to that account we need to compute its *gradient* $\nabla_{\boldsymbol\theta} v_\pi(s)$.

Policy Gradient Theorem

Assume that $\mathcal{S}$ and $\mathcal{A}$ are finite and that maximum episode length H is also finite.

Let $\pi(a|s;\boldsymbol\theta)$ be a parametrized policy. We denote the initial state distribution as $h(s)$ and the on-policy distribution under π as $\mu(s)$. Let $J(\boldsymbol\theta) \stackrel{\text{def}}{=} \mathbb{E}_{h,\pi} v_\pi(s)$.

Then

$$\nabla_{\boldsymbol\theta} v_\pi(s) \propto \sum_{s'\in\mathcal{S}} P(s \rightarrow \ldots \rightarrow s'|\pi) \sum_{a\in\mathcal{A}} q_\pi(s', a) \nabla_{\boldsymbol\theta} \pi(a|s';\boldsymbol\theta)$$

and

$$\nabla_{\boldsymbol\theta} J(\boldsymbol\theta) \propto \sum_{s\in\mathcal{S}} \mu(s) \sum_{a\in\mathcal{A}} q_\pi(s, a) \nabla_{\boldsymbol\theta} \pi(a|s;\boldsymbol\theta),$$

where $P(s \rightarrow \ldots \rightarrow s'|\pi)$ is probability of being in state s' when starting from state s, after any number of 0, 1, … steps.

Proof of Policy Gradient Theorem

$$
\begin{aligned}
\nabla v_\pi(s) &= \nabla \Big[\sum\nolimits_a \pi(a|s;\boldsymbol\theta) q_\pi(s,a)\Big] \\
&= \sum\nolimits_a \Big[\nabla \pi(a|s;\boldsymbol\theta) q_\pi(s,a) + \pi(a|s;\boldsymbol\theta) \nabla q_\pi(s,a)\Big] \\
&= \sum\nolimits_a \Big[\nabla \pi(a|s;\boldsymbol\theta) q_\pi(s,a) + \pi(a|s;\boldsymbol\theta) \nabla \big(\sum\nolimits_{s'} p(s'|s,a)(r + v_\pi(s'))\big)\Big] \\
&= \sum\nolimits_a \Big[\nabla \pi(a|s;\boldsymbol\theta) q_\pi(s,a) + \pi(a|s;\boldsymbol\theta) \big(\sum\nolimits_{s'} p(s'|s,a) \nabla v_\pi(s')\big)\Big]
\end{aligned}
$$

We now expand $v_\pi(s')$.

$$
\begin{aligned}
&= \sum\nolimits_a \Big[\nabla \pi(a|s;\boldsymbol\theta) q_\pi(s,a) + \pi(a|s;\boldsymbol\theta) \Big(\sum\nolimits_{s'} p(s'|s,a) \Big(\\
&\qquad \sum\nolimits_{a'} \Big[\nabla \pi(a'|s';\boldsymbol\theta) q_\pi(s',a') + \pi(a'|s';\boldsymbol\theta) \Big(\sum\nolimits_{s''} p(s''|s',a') \nabla v_\pi(s'')\Big)\Big)\Big)\Big]
\end{aligned}
$$

Continuing to expand all $v_\pi(s'')$, we obtain the following:

$$
\nabla v_\pi(s) = \sum_{s' \in \mathcal{S}} \sum_{k=0}^{H} P(s \rightarrow s' \text{ in } k \text{ steps } |\pi) \sum_{a \in \mathcal{A}} q_\pi(s',a) \nabla_{\boldsymbol\theta} \pi(a|s';\boldsymbol\theta).
$$

Proof of Policy Gradient Theorem

To finish the proof of the first part, it is enough to realize that

$$\textstyle\sum_{k=0}^{H} P(s \rightarrow s' \text{ in } k \text{ steps } |\pi) \propto P(s \rightarrow \ldots \rightarrow s'|\pi).$$

For the second part, we know that

$$\nabla_{\boldsymbol\theta} J(\boldsymbol\theta) = \mathbb{E}_{s\sim h} \nabla_{\boldsymbol\theta} v_\pi(s) \propto \mathbb{E}_{s\sim h} \sum_{s'\in\mathcal{S}} P(s \rightarrow \ldots \rightarrow s'|\pi) \sum_{a\in\mathcal{A}} q_\pi(s', a) \nabla_{\boldsymbol\theta} \pi(a|s'; \boldsymbol\theta),$$

therefore using the fact that $\mu(s') = \mathbb{E}_{s\sim h} P(s \rightarrow \ldots \rightarrow s'|\pi)$ we get

$$\nabla_{\boldsymbol\theta} J(\boldsymbol\theta) \propto \sum_{s\in\mathcal{S}} \mu(s) \sum_{a\in\mathcal{A}} q_\pi(s, a) \nabla_{\boldsymbol\theta} \pi(a|s; \boldsymbol\theta).$$

Finally, note that the theorem can be proved with infinite $\mathcal{S}$ and $\mathcal{A}$; and also for infinite episodes when discount factor $\gamma < 1$.

The REINFORCE algorithm (Williams, 1992) uses directly the policy gradient theorem, minimizing $-J(\boldsymbol\theta) \stackrel{\text{def}}{=} -\mathbb{E}_{h,\pi} v_\pi(s)$. The loss gradient is then

$$\nabla_{\boldsymbol\theta} -J(\boldsymbol\theta) \propto -\sum_{s\in\mathcal{S}} \mu(s) \sum_{a\in\mathcal{A}} q_\pi(s,a) \nabla_{\boldsymbol\theta} \pi(a|s;\boldsymbol\theta) = -\mathbb{E}_{s\sim\mu} \sum_{a\in\mathcal{A}} q_\pi(s,a) \nabla_{\boldsymbol\theta} \pi(a|s;\boldsymbol\theta).$$

However, the sum over all actions is problematic. Instead, we rewrite it to an expectation which we can estimate by sampling:

$$\nabla_{\boldsymbol\theta} -J(\boldsymbol\theta) \propto \mathbb{E}_{s\sim\mu} \mathbb{E}_{a\sim\pi} q_\pi(s,a) \nabla_{\boldsymbol\theta} -\ln \pi(a|s;\boldsymbol\theta),$$

where we used the fact that

$$\nabla_{\boldsymbol\theta} \ln \pi(a|s;\boldsymbol\theta) = \frac{1}{\pi(a|s;\boldsymbol\theta)} \nabla_{\boldsymbol\theta} \pi(a|s;\boldsymbol\theta).$$

REINFORCE therefore minimizes the loss

$$\mathbb{E}_{s\sim\mu}\mathbb{E}_{a\sim\pi}q_\pi(s,a)\nabla_{\boldsymbol\theta} - \ln\pi(a|s;\boldsymbol\theta),$$

estimating the $q_\pi(s,a)$ by a single sample.

Note that the loss is just a weighted variant of negative log likelihood (NLL), where the sampled actions play a role of gold labels and are weighted according to their return.

REINFORCE: Monte-Carlo Policy-Gradient Control (episodic) for π_*

Input: a differentiable policy parameterization $\pi(a|s,\boldsymbol\theta)$
Algorithm parameter: step size $\alpha > 0$
Initialize policy parameter $\boldsymbol\theta \in \mathbb{R}^{d'}$ (e.g., to $\mathbf{0}$)

Loop forever (for each episode):
  Generate an episode $S_0, A_0, R_1, \ldots, S_{T-1}, A_{T-1}, R_T$, following $\pi(\cdot|\cdot,\boldsymbol\theta)$
  Loop for each step of the episode $t = 0, 1, \ldots, T-1$:
    $G \leftarrow \sum_{k=t+1}^{T} \gamma^{k-t-1} R_k$ (G_t)
    $\boldsymbol\theta \leftarrow \boldsymbol\theta + \alpha G \nabla \ln\pi(A_t|S_t,\boldsymbol\theta)$

Modification of Algorithm 13.3 of "Reinforcement Learning: An Introduction, Second Edition".

REINFORCE with Baseline

The returns can be arbitrary – better-than-average and worse-than-average returns cannot be recognized from the absolute value of the return.

Hopefully, we can generalize the policy gradient theorem using a baseline $b(s)$ to

$$\nabla_{\boldsymbol\theta} J(\boldsymbol\theta) \propto \sum_{s\in\mathcal{S}} \mu(s) \sum_{a\in\mathcal{A}} \big(q_\pi(s,a) - b(s)\big) \nabla_{\boldsymbol\theta} \pi(a|s;\boldsymbol\theta).$$

The baseline $b(s)$ can be a function or even a random variable, as long as it does not depend on a, because

$$\sum_a b(s) \nabla_{\boldsymbol\theta} \pi(a|s;\boldsymbol\theta) = b(s) \sum_a \nabla_{\boldsymbol\theta} \pi(a|s;\boldsymbol\theta) = b(s)\nabla 1 = 0.$$

REINFORCE with Baseline

A good choice for $b(s)$ is $v_\pi(s)$, which can be shown to minimize variance of the estimator. Such baseline reminds centering of returns, given that

$$v_\pi(s) = \mathbb{E}_{a\sim\pi} q_\pi(s, a).$$

Then, better-than-average returns are positive and worse-than-average returns are negative. The resulting $q_\pi(s, a) - v_\pi(s)$ function is also called an *advantage function*

$$a_\pi(s, a) \stackrel{\text{def}}{=} q_\pi(s, a) - v_\pi(s).$$

Of course, the $v_\pi(s)$ baseline can be only approximated. If neural networks are used to estimate $\pi(a|s; \boldsymbol\theta)$, then some part of the network is usually shared between the policy and value function estimation, which is trained using mean square error of the predicted and observed return.

REINFORCE with Baseline

REINFORCE with Baseline (episodic), for estimating $\pi_{\boldsymbol\theta} \approx \pi_*$

Input: a differentiable policy parameterization $\pi(a|s, \boldsymbol\theta)$
Input: a differentiable state-value function parameterization $\hat v(s,\mathbf{w})$
Algorithm parameters: step sizes $\alpha^{\boldsymbol\theta} > 0$, $\alpha^{\mathbf{w}} > 0$
Initialize policy parameter $\boldsymbol\theta \in \mathbb{R}^{d'}$ and state-value weights $\mathbf{w} \in \mathbb{R}^d$ (e.g., to $\mathbf{0}$)

Loop forever (for each episode):
  Generate an episode $S_0, A_0, R_1, \ldots, S_{T-1}, A_{T-1}, R_T$, following $\pi(\cdot|\cdot, \boldsymbol\theta)$
  Loop for each step of the episode $t = 0, 1, \ldots, T-1$:
    $G \leftarrow \sum_{k=t+1}^T \gamma^{k-t-1} R_k$ (G_t)
    $\delta \leftarrow G - \hat v(S_t,\mathbf{w})$
    $\mathbf{w} \leftarrow \mathbf{w} + \alpha^{\mathbf{w}} \delta \nabla \hat v(S_t,\mathbf{w})$
    $\boldsymbol\theta \leftarrow \boldsymbol\theta + \alpha^{\boldsymbol\theta} \delta \nabla \ln \pi(A_t|S_t, \boldsymbol\theta)$

Modification of Algorithm 13.4 of "Reinforcement Learning: An Introduction, Second Edition".

REINFORCE with Baseline

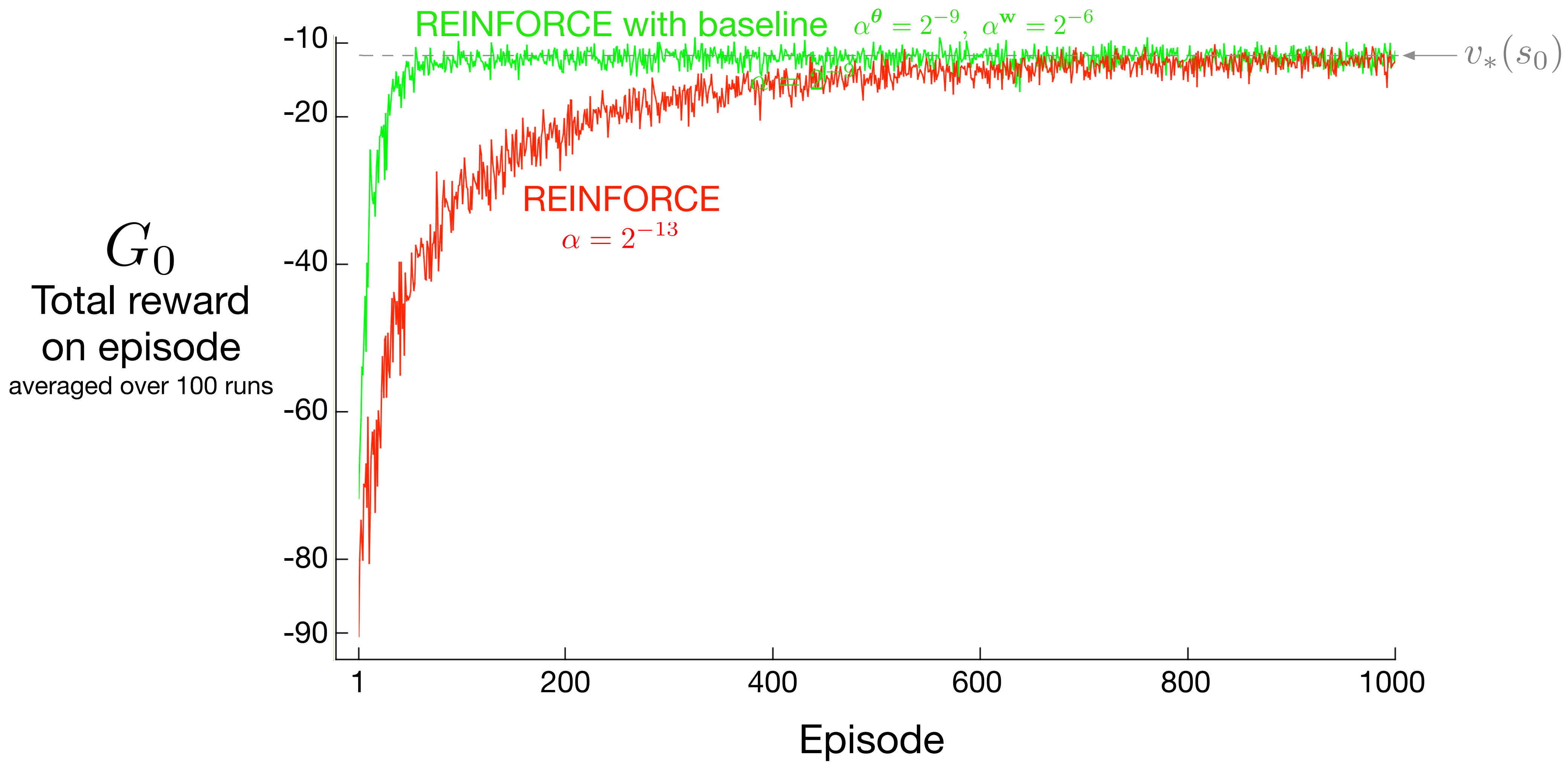


Figure 13.2 of "Reinforcement Learning: An Introduction, Second Edition".