PA230: Reinforcement Learning

Petr Novotný

"Good and evil, reward and punishment, are the only motives to a rational creature; these are the spur and reins whereby all mankind are set on work and guided."

John Locke, *Some Thoughts Concerning Education* (1693)

Organizational Information

General

- Lecture: Thursdays 2-3:40p.m.
- Homework: see the interactive syllabus in IS
  - mainly binary classification (accepted/not accepted)
  - all your homeworks need to be marked as passed to proceed to exam
  - can (but do not have to) be done in pairs (pairs can differ across the individual assignments)
  - for those who passed, the teacher will receive feedback on the general quality of the solutions for each student - can be taken into account when determining the final grade (typically in students' favor)
- Exam:
  - oral
  - each attempt counts ? (unlike the Brázdil system)
  - in general, knowledge of anything mentioned on the slides can be required, unless explicitly marked with "nex" (like the Brázdil system)

Team

- Lecturer: Petr Novotný

- HW team:

Martin Kurečka

Václav Nevyhoštěný

Vít Unčovský

Communication

Official discord server:

https://discord.gg/9mxTgYhcdB

- Official communication forum of the course: falls under the university ethical guidelines.
- Use your real name for posting (you can set-up an account under your IS email if necessary).

Reading

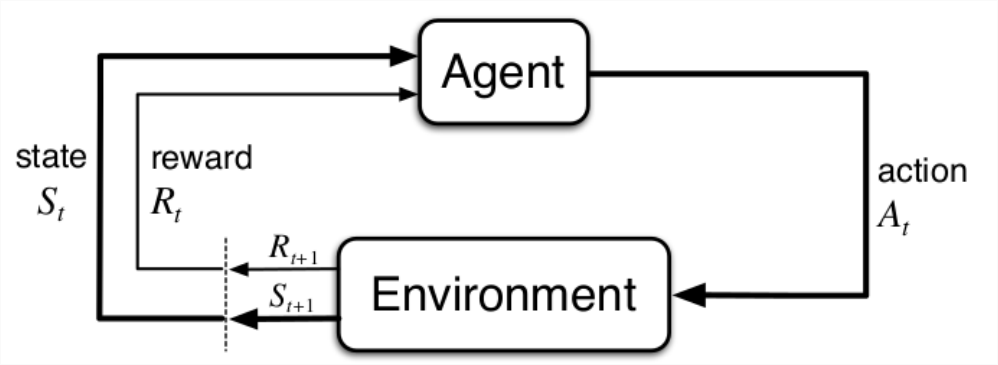
- Compulsory:
  - these slides,
  - material explicitly prescribed by these slides (not much).
- Recommended:
  - Sutton & Barto: Reinforcement Learning: An Introduction (2nd ed.), available at `http://incompleteideas.net/book/RLbook2020.pdf`
    - henceforth referenced as "S&B"
  - slides by David Silver `https://www.davidsilver.uk/teaching/`
  - CMU slides `https://www.andrew.cmu.edu/course/10-703/`
  - more specific literature recommendations will be given for each topic later

Reinforcement Learning: What, Why, When, How, & Other Questions

Types of machine learning

- unsupervised
  - spot "useful" patterns in data
- supervised
  - given labeled data, predict labels on unlabeled data
- reinforcement
  - agents and decision-making
  - agency = "the ability to take action or to choose what action to take" (Cambridge Dictionary)

General RL scheme



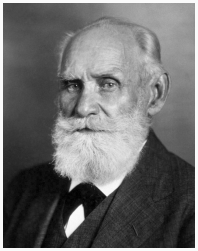
source: Sutton&Barto, p. 48

Keywords: sequential, dynamic, subject to uncertainty

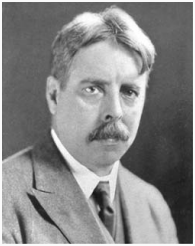
RL: Objective and approach

- Objective: Design a decision policy (= agent behavior) which prescribes to the agent how to act in different situations (states), typically so as to achieve some goal.
- Approach: Start with ($\pm$ random) behavior and adapt it based on past experience via the law of effect:
  - actions with good/bad consequences for the agent are more/less likely to be repeated by the agent (within the same context)

RL in psychology (nex)

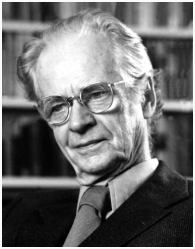


I.P. Pavlov
(1849-1936)
classical conditioning



E. Thorndike
(1874-1949)
law of effect

J.B. Watson
(1878-1958)
behaviorist manifesto



B.F. Skinner
(1904-1990)
radical behaviorism,
reinforcement,
rewards

Learning by trying & XX dilemma

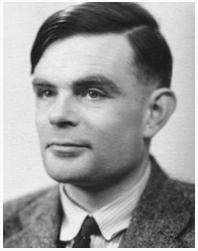
Underlying the RL approach is the idea of learning by trying:

- first, act more or less randomly (exploration)
  - integral part of early human development
- continually adapt behavior according to experience and feedback from the environment (exploitation)
  - strength of feedback $\approx$ strength of behavior adaptation

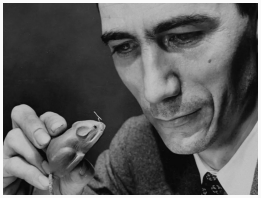
Balancing exploration and exploitation (XX) is a recurring theme in RL.

Incomplete history of RL in computer science I

"Learning by trying" machines and software, ad hoc approaches:



A. Turing
(1912-1954)
1948: theoretical "pleasure & pain" system
to train computers



C. Shannon
(1916-2001)
1950: Theseus maze-solving mouse

M. Minsky
(1927-2016)
1950s: analog neural net machines (SNARCS)

And many more...
Recommended: S&B: Sec. 1.7.

Incomplete history of RL in computer science II

Mathematical foundations of sequential decision making:

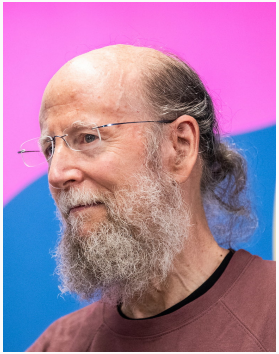
R. Bellman
(1920-1984)

R. Howard
(b. 1934)

- Formalization via Markov decision processes (MDPs)
- value iteration
  (attributed to Bellman, 1957)
- policy iteration
  (attributed to Howard, 1960)

Incomplete history of RL in computer science III

Since late 1980's: synthesis – learning by trial in MDPs



R. Sutton

A. Barto

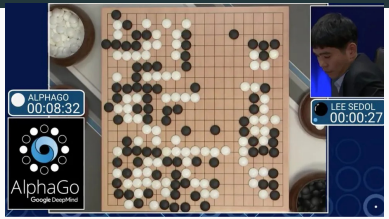
C. Watkins

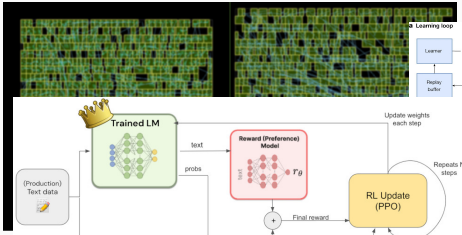
Temporal difference learning

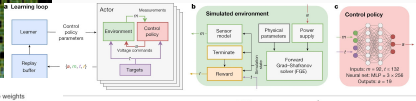
Q-learning

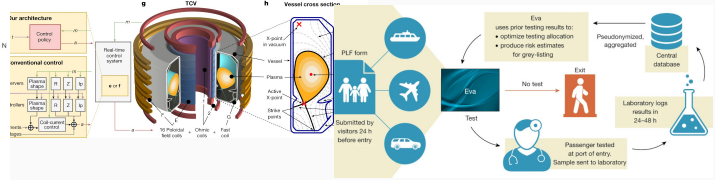
. . . and many more.

Successes of RL (nex)









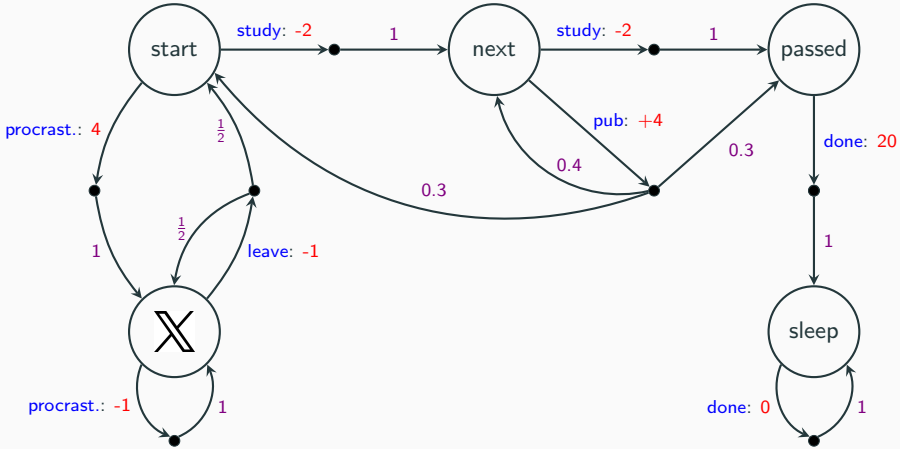
Words of caution (and controversy) (nex)

- **"Pure" Reinforcement Learning (cherry)**
  - The machine predicts a scalar reward given once in a while.
  - **A few bits for some samples**
- **Supervised Learning (icing)**
  - The machine predicts a category or a few numbers for each input
  - Predicting human-supplied data
  - **10→10,000 bits per sample**
- **Self-Supervised Learning (cake génoise)**
  - The machine predicts any part of its input for any observed part.
  - Predicts future frames in videos
  - **Millions of bits per sample**

Mathematical Foundations of Sequential Decision-Making

MDP Example

MDP with actions, rewards and transition probabilities.



Markov Decision Process

Given a set X, we denote $\mathcal{D}(X)$ the set of all probability distributions over X.

Definition 1

A Markov decision process (MDP) is a tuple $(\mathcal{S}, \mathcal{A}, p, r)$ where

- $\mathcal{S}$ is a set of states,
- $\mathcal{A}$ is a set of actions,
- $p\colon \mathcal{S} \times \mathcal{A} \to \mathcal{D}(\mathcal{S})$ is a probabilistic transition function,
- $r\colon \mathcal{S} \times \mathcal{A} \to \mathbb{R}$ is a reward function.

We will shorten $p(s,a)(s')$ to $p(s' \mid s,a)$.

The p, r can be partial functions: action a is enabled in state s if both $p(s,a)$ and $r(s,a)$ are defined. We denote by $\mathcal{A}(s)$ the set of all actions enabled in s.

Dynamics of MDPs

- start in some initial state s_0
- MDP evolves in discrete time steps $t = 0, 1, 2, 3, \ldots$
- in each time step t, let s_t be the current state; then:
  - agent selects action $a_t \in \mathcal{A}(s_t)$
  - the environment responds with next state $s_{t+1} \sim p(s_t, a_t)$ and with immediate reward $r_{t+1} = r(s_t, a_t)$
  - t is incremented and the process repeats in the same fashion forever

Thus, the agent produces a trajectory $\tau = s_0, a_0, r_1, s_1, a_1, r_2, s_2, a_2, r_3, s_3, \ldots$.

τ is produced randomly (due to p and possibly also agent choices being probabilistic): it is a random variable and so are its components: we define random variables

- S_t = state at time step t
- A_t = action at time step t
- R_t = reward received just before entering S_t

Policies

Definition 2

A history is a finite prefix of a trajectory ending in a state, i.e., an object of type

$$s_0, a_0, r_1, s_1, a_1, r_2, \ldots, a_{t-1}, r_t, s_t \in (\mathcal{S} \cdot \mathcal{A} \cdot \mathbb{R})^* \mathcal{S}.$$

we denote by $last(h)$ the last state of a history h.

Definition 3

A policy is a function $\pi \colon (\mathcal{S} \cdot \mathcal{A} \cdot \mathbb{R})^* \mathcal{S} \to \mathcal{D}(\mathcal{A})$ which to each history h assigns a probability distribution over $\mathcal{A}(last(h))$.

A policy is by definition an infinite object!

MD policies

Definition 4

A policy π is:

- memoryless if $\pi(h) = \pi(h')$ whenever $last(h) = last(h')$ (we can view memoryless policies as objects of type $\pi \colon \mathcal{S} \to \mathcal{D}(\mathcal{A})$);
- deterministic if $\pi(h)$ always assigns probability 1 to one action, and zero to all others (we can view det. policies of objects of type $\pi \colon (\mathcal{S} \cdot \mathcal{A} \cdot \mathbb{R})^* \mathcal{S} \to \mathcal{A}$).

Definition 5

A policy π is MD (memoryless deterministic) if it is both memoryless and deterministic.

Dynamics of MDPs (more precise)

Given a distribution $\mathcal{I}$ of initial states and a policy π

- start in some initial state $s_0 \sim \mathcal{I}$
- MDP evolves in discrete time steps $t = 0, 1, 2, 3, \ldots$
- in each time step t, let h_t be the history produced so far; then:
  - agent selects action $a_t \in \mathcal{A}(s_t)$ according to π, i.e. $a_t \sim \pi(h_t)$
  - the environment responds with next state $s_{t+1} \sim p(s_t, a_t)$ and with immediate reward $r_{t+1} = r(s_t, a_t)$, the history is extended by a_t, r_t, s_{t+1},
  - t is incremented and the process repeats in the same fashion forever

Probability space induced by a policy

In particular, each policy π together with a distribution $\mathcal{I}$ of initial states induce a probability measure $\mathbb{P}^\pi$ over the trajectories of the MDP.[1]

We denote by $\mathbb{E}^\pi$ the associated expected value (expectation) operator.

We denote by $\mathbb{P}^\pi[E \mid S_0 = s]$ the probability of event E provided that the initial state is fixed to s (and similarly for expectations).

Exercise 6

In the "study" MDP, consider an MD policy π s.t. $\pi(\mathit{start}) = \mathit{study}$ and $\pi(\mathit{next}) = \mathit{pub}$. Compute the following quantities:

- $\mathbb{P}^\pi[\text{visit pub at least twice}' \mid S_0 = \mathit{start}'']$
- $\mathbb{P}^\pi[\text{visit pub at exactly twice} \mid S_0 = \mathit{start}'']$
- $\mathbb{E}^\pi[R_1]$
- $\mathbb{E}^\pi[R_3]$

[1] $\mathcal{I}$ is typically known from the context and hence omitted from the notation

Memorylessness

In this course, we will almost exclusively focus on memoryless policies. Hence, from now on, policy = memoryless policy. General policies will be referred to as history-dependent policies should the need arise.

Why memoryless?

Intuition: Markov property of MDPs: next step depends only on the current state and on action performed in the current step. Hence, intuitively there is no need for a policy to remember the past so as to "play well".

The sufficiency of memoryless policies does not extended to more general/complex decision-making settings (not covered in this course), such as:

- partially observable MDPs
- non-stationary environments
- quantile/risk-aware MDPs, etc.

Returns (payoffs)

Definition 7

Let $\gamma \in [0, 1)$ be a discount factor.
For a trajectory $\tau = s_0, a_0, r_1, s_1, a_1, r_2, s_2, a_2, r_3, s_3, \ldots$ we define the discounted return (or payoff) of τ to be the quantity

$$G(\tau) = r_1 + \gamma \cdot r_2 + \gamma^2 \cdot r^3 + \cdots \gamma^3 \cdot r^4 = \sum_{i=0}^{\infty} \gamma^i \cdot r_{i+1}.$$

Equivalently

$$G = \sum_{i=0}^{\infty} \gamma^i \cdot R_{i+1}.$$

Returns (variants)

- Finite horizon (FH): additionally, we are given a finite decision horizon $H \in \mathbb{N} \cup \{\infty\}$. The return is that counted only up to step H:

$$G^H = \sum_{i=0}^{H-1} \gamma^i \cdot R_{i+1}$$

  For finite H, the discount factor can be 1. $H = \infty$ corresponds to the original definition.
- Episodic returns: In episodic tasks, there is a distinguished set $Term \subseteq \mathcal{S}$ of terminal states which is guaranteed to be reached with probability 1 under any policy. We denote by T a random variable denoting the first point in time when we hit a terminal state. We count rewards only up to that time:

$$G^T = \sum_{i=1}^{T-1} \gamma^i \cdot R_{i+1}$$

  Can be modeled under original definition by "sink" states.

Types of returns: discussion

- We will typically omit the superscripts since the type of task considered will be known from the context.
- We have $G^H \to G$ (pointwise) as $H \to \infty$. I.e., finite-horizon returns with high enough H approximate the standard (infinite-horizon) case.
- In real world, we typically deal with FH or episodic tasks: we cannot wait infinite time to learn something from a trajectory. However, the infinite-horizon case can be viewed as a neat mathematical abstraction of the FH&episodic tasks, and the classical sequential decision-making theory is most developed for the infinite horizon case.

Policy and state values

Definition 8

Let π be a policy and s a state. The value of π in state s is the quantity

$$v^\pi(s) = \mathbb{E}^\pi[G \mid S_0 = s].$$

Exercise 9

Discuss the values of MD policies in our running example.

Definition 10

The (optimal) value of state s is the quantity

$$v^*(s) = \sup_\pi v^\pi(s).$$

Definition 11

Let π be a policy and $\varepsilon > 0$. We say that π is ε-optimal in state s if

$$v^{\pi}(s) \geq v^{*}(s) - \varepsilon.$$

We say that π is optimal in s is it is 0-optimal in s, i.e. if

$$v^{\pi}(s) = v^{*}(s).$$

A policy is (ε-)optimal if it is (ε-)optimal in every state.

Existence of optimal policies

Theorem 12: (Classical result, not formally proven here)

Let $\mathcal{M}$ be a finite MDP (i.e., the state and action sets are finite) with infinite-horizon returns. Then there exists an optimal MD policy. Moreover, an optimal MD policy can be computed in polynomial time.

Agent control solved? NO! "Only" works if you can actually construct the MDP model of your environment and fit it into a computer. Otherwise, we use reinforcement learning.

Exact Planning with Known Model: Value & Policy Iteration

Goal of this lecture

Algorithms that compute the optimal value vector v^* and some optimal MD policy π^* given a full knowledge of an MDP $\mathcal{M}$.

Polynomial-time algorithm

MDPs can be solved by linear programming (LP)

$$\begin{aligned} &\text{maximize } \vec{c} \cdot \vec{x} \\ &\text{subject to } A \cdot \vec{x} \leq \vec{b} \end{aligned}$$

- LP can be solved in polynomial time by so-called interior-point algorithms.
- However, we typically use other, MDP-specific algorithms: value iteration (VI) and policy iteration (PI). These are not polynomial-time in general, but typically faster on practical instances.
- Moreover, most truly RL algorithms can be seen as approximate generalizations of VI or PI (or both).

Example: Policy evaluation

Exercise 13

Consider all four MD policies in our running "pub or study" example that always try to quit when in X state. Compute the values of these policies in the initial state *start*.

Policy evaluation equations

Theorem 14

For any memoryless policy π and any state s it holds:

$$v^{\pi}(s) = \sum_{a \in \mathcal{A}(s)} \pi(a|s) \cdot \underbrace{\left[r(s,a) + \gamma \cdot \sum_{s' \in \mathcal{S}} p(s'|s,a) \cdot v^{\pi}(s') \right]}_{\stackrel{\text{def}}{=} q^{\pi}(s,a)}.$$

Bellman optimality equations

Theorem 15

The following holds for any state s:

$$v^*(s) = \max_{a \in \mathcal{A}(s)} \underbrace{\left[r(s,a) + \gamma \cdot \sum_{s' \in \mathcal{S}} p(s'|s,a) \cdot v^*(s') \right]}_{\stackrel{\text{def}}{=} q^*(s,a)}$$

- Note: the policy evaluation equations are a special case of the Bellman ones: given a policy π, we can consider an MDP $\mathcal{M}^\pi$ in which there is a single action $*$ enabled in each state and the probability of transition $s \stackrel{*}{\to} s'$ equals $\sum_{a \in \mathcal{A}(s)} \pi(a|s) \cdot p(s'|s,a)$. Then $\mathcal{M}^\pi$ mimics the behavior of π in $\mathcal{M}$ and Bellman eq's in $\mathcal{M}^\pi$ = evaluation equations for π in $\mathcal{M}$.
- But these equations are no longer linear! How do we solve them? Is the solution even unique?

Bellman update operator

The right-hand-side (RHS) of the Bellman equations can be viewed as an operator $\Phi\colon \mathbb{R}^{\mathcal{S}} \to \mathbb{R}^{\mathcal{S}}$: for any $\vec{x} \in \mathbb{R}^{\mathcal{S}}$, $\Phi(\vec{x})$ is a vector such that for any state s:

$$\Phi(\vec{x})(s) \stackrel{\text{def}}{=} \max_{a \in \mathcal{A}(s)} \left[r(s,a) + \gamma \cdot \sum_{s' \in \mathcal{S}} p(s'|s,a) \cdot \vec{x}(s') \right]$$

Exercise 16

In our running example, compute $\Phi(\vec{0})$.

Theorem 15 says that the optimal value vector v^* is a fixed point of Φ:

$$v^* = \Phi(v^*).$$

Lemma 17: (not proven here)

For any discount factor $\gamma \in [0, 1)$, the Bellman operator Φ is a contraction, i.e. for any pair of vectors $\vec{x}, \vec{y}$ it holds

$$\|\vec{x} - \vec{y}\|_\infty \leq \gamma \cdot \|\Phi(\vec{x}) - \Phi(\vec{y})\|_\infty.$$

Theorem 18: Banach fixed point theorem (classical calculus, not proven here)

A contraction mapping from a complete metric space (in particular, $\mathbb{R}^n$) to itself has a unique fixed point.

Exact characterization of v^*

Corollary 19

The optimal value vector is a unique solution of the Bellman optimality equations.

In particular, also the policy evaluation equations have a unique solution, equal to v^π. Since the policy evaluation equations are linear, their solution can be computed by Gaussian elimination.

But the general Bellman equations are not linear. How can ve solve them?

Banach fixpoint theorem (full)

Theorem 20: Banach fixed point theorem (full version, not proven here)

A contraction mapping Φ from a complete metric space (in particular, $\mathbb{R}^n$) to itself has a unique fixed point $\vec{z}$.

Moreover, $\vec{z}$ is the limit of iterative applications of Φ on any initial vector. I.e., for any $\vec{x_0} \in \mathbb{R}^n$, the sequence $\vec{x_0}, \Phi(\vec{x_0}), \Phi(\Phi(\vec{x_0})), \Phi^{(3)}(\vec{x_0}), \ldots$ converges to $\vec{z}$:

$$z = \lim_{i \to \infty} \Phi^{(i)}(\vec{x_0})$$

Value iteration (VI; Bellman, 1957)

Algorithm 1: Value iteration

Input: MDP $\mathcal{M} = (\mathcal{S}, \mathcal{A}, p, r)$

Output: Approximation $\tilde{v}$ of v^*

$x \leftarrow$ any vector from $\mathbb{R}^{|\mathcal{S}|}$; `// typically` $\vec{0}$

$next \leftarrow x$;

repeat

  foreach $s \in \mathcal{S}$ **do**

    $next(s) \leftarrow \underbrace{\max_{a \in \mathcal{A}(s)} [r(s,a) + \gamma \cdot \sum_{s' \in \mathcal{S}} p(s'|s,a) \cdot \vec{x}(s')]}_{\Phi(x)(s)}$;

  $x \leftarrow next$

until *termination condition*;

Typical term. conditions:

- after a fixed no. of iterations (i.e., use for-loop with a fixed bound)
- after each component of x changes less then some given ε

How to use VI

By the Banach fixpoint theorem (and Lemma 17), the value of variable x VI converges to v^*. Can we recognize when is x "close enough" to $\vec{x}$?

In the following couple of theorems, let $\vec{x}_0, \vec{x}_1, \vec{x}_2, \ldots$ be the sequence of vectors computed by VI, i.e. $\vec{x}_0$ is arbitrary and $\vec{x}_{i+1} = \Phi(\vec{x}_i)$ for all $i \geq 0$.

Theorem 21: Stopping condition (not proven here)

For any $\varepsilon > 0$: if

$$\|\vec{x}_{i+1} - \vec{x}_i\|_\infty \leq \varepsilon \cdot \frac{1-\gamma}{\gamma},$$

then

$$\|\vec{x}_{i+1} - v^*\|_\infty \leq \varepsilon$$

How to use VI (2)

How fast can we get to the point where we are close enough?

Theorem 22: Speed of convergence (not proven here)

For all $i \geq 0$ it holds

$$\|\vec{x}^n - v^*\|_\infty \leq \frac{\gamma^n}{1-\gamma} \cdot \|\vec{x}_1 - \vec{x}_0\|_\infty.$$

In particular, if we terminate VI after

$$i = \left\lceil \frac{\log(\varepsilon) + \log\left(\frac{1-\gamma}{\|\vec{x}_1 - \vec{x}_0\|_\infty}\right)}{\log(\gamma)} \right\rceil$$

steps, then its output x_i will be an ε-approximation of v^*.

How to use VI (3)

Can we actually get some optimal values instead of approximations? First, note that VI computes optimal finite-horizon values:

Let $v^i = \sup_\pi \mathbb{E}^\pi[\sum_{i=1}^H \gamma^{i-1} \cdot R_i]$. The supremum is over all (i.e., history dependent) policies, since in the FH problem an optimal policy needs to track the number of elapsed (and thus remaining) steps: memory is needed for that.

Theorem 23: (Easy but important exercise)

If $\vec{x}_0 = \vec{0}$, then $\vec{x}_H = v^H$ for all $H \geq 0$.
Moreover, let π^H be a deterministic history-dependent policy such that for all $1 \leq i \leq H$, whenever there are i steps remaining till the horizon, the policy π^H selects in state s an action a s.t.

$$a = \arg\max_{a \in \mathcal{A}(s)} [r(s,a) + \gamma \cdot \sum_{s' \in \mathcal{S}} p(s'|s,a) \cdot \vec{x}_{i-1}(s')]$$

(with ties broken arbitrarily). Then π^H is an optimal H-step policy.

Greedy policies

Can we actually get optimal policy for the inf. horizon problem?

Definition 24: $\vec{x}$-greedy policy (very important)

Let $\vec{x} \in \mathbb{R}^{\mathcal{S}}$ be any vector. A $\vec{x}$-greedy policy is an MD policy π such that in any state s:

$$\pi(s) = \arg\max_{a \in \mathcal{A}(s)} [r(s,a) + \gamma \cdot \sum_{s' \in \mathcal{S}} p(s'|s,a) \cdot \vec{x}(s')].$$

How to use VI (4)

Theorem 25: Optimal inf.-horizon policy from VI (not proven here)

There is a number N polynomial in size of the MDP and exponential in the binary encoding size of γ such that a policy π that is $\vec{x}_N$-greedy is optimal in every state, i.e. $v^\pi = v^*$.

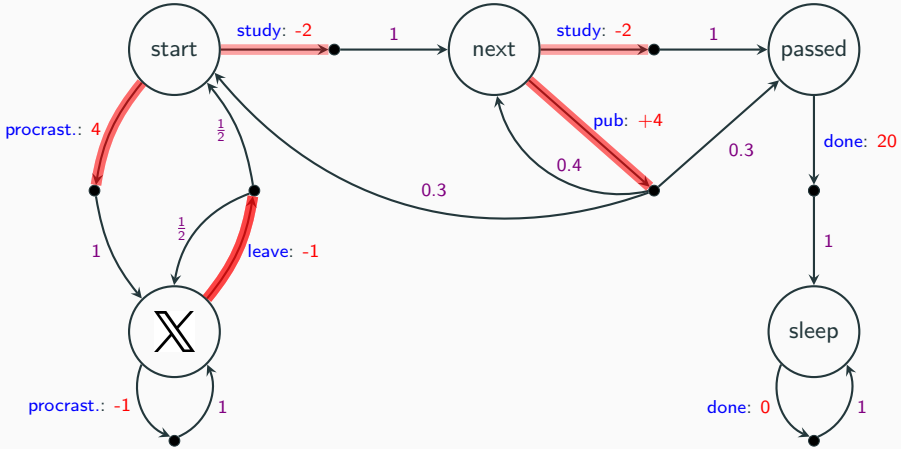
Note that once π is computed, v^π can be computed in polynomial time via policy evaluation equations.

Hence, VI can be said to solve MDPs in exponential time (and in polynomial time if the discount factor is assumed to be a fixed constant instead of an input parameter), though the approximate version is typically used in practice.

Note: the fact that some policy π is $\vec{x}$-greedy does not mean that $v^\pi \geq \vec{x}$! Homework: find a counterexample and post it to Discord.

However, for VI it can be shown that if $\|\vec{x}_{i+1} - \vec{x}_i\|_\infty \leq \varepsilon \cdot \frac{1-\gamma}{\gamma}$ (stopping condition from Theorem 21), then an $\vec{x}_{i+1}$-greedy policy is ε-optimal.

Policy improvement



$v^{\pi} = ($ $)$

let π' be v^{π}-greedy $v^{\pi'} = ($ $)$

let π'' be $v^{\pi'}$-greedy

Policy improvement theorem

Theorem 26: Policy improvement

Let π be a policy. If $\Phi(v^\pi) \geq v^\pi$, then any v^π-greedy policy π_g is at least as good as π, i.e. $\forall s \in \mathcal{S} : v^{\pi_g}(s) \geq v^\pi(s)$.
Moreover, if $\Phi(v^\pi)(s) > v^\pi(s)$ for some state s, then also $v^{\pi_g}(s') > v^\pi(s')$ for some state s'.

Returns from a given time step

For the proof of PIT and also many times later, we will need the following notation:

Definition 27: Important!

Let $\tau = s_0, a_0, r_1, s_1, a_1, r_2, s_2, a_2, r_3, s_3, \dots$ be a trajectory and $t \in \mathbb{N}$ a time step. We define

$$G_t(\tau) = \sum_{i=t}^{H-1} \gamma^{i-t} \cdot r_{i+1} = r_{t+1} + \gamma r_{t+2} + \gamma^2 r_{t+3} + \cdots,$$

where $H \in \mathbb{N} \cup \{\infty\}$ or $H = T$ for episodic tasks.
We similarly define, for any policy π:

$$G_t^\pi = \mathbb{E}^\pi[G_t] = \mathbb{E}^\pi[\sum_{i=t}^{H-1} \gamma^{i-t} \cdot R_{i+1}].$$

Proof of PIT (setup)

We will define a sequence of policies $\pi_0, \pi_1, \pi_2, \ldots$ s.t.:

- $\pi_0 = \pi$
- π_i behaves as π_g (i.e., selects the same actions in same states) for the first i steps, then "switches" back to behave as π:

- we also define $\pi_\infty = \pi_g$

We want: $v^{\pi_\infty}(s) \geq v^{\pi}(s)$ for all s.

Not hard to see: $v^{\pi_i} \to v^{\pi_\infty}$ as $i \to \infty$ (π_i behaves as π_∞ for longer and longer as i increases + discounting).

It suffices to show: $v^{\pi_i}(s) \geq v^{\pi}(s)$ for all $i \in \mathbb{N}$ and all $s \in \mathcal{S}$.

Proof of PIT (induction)

$v^{\pi_i}(s) \geq v^{\pi}(s)$ for all $i \in \mathbb{N}$ and all $s \in \mathcal{S}$

- $i = 0$: clear
- $i > 0$:

$$
\begin{aligned}
v^{\pi_i}(s) &= \mathbb{E}^{\pi_i}[R_1 + \gamma R_2 + \cdots + \gamma^{i-1} R_i + \gamma^i R_{i+1} + \cdots \mid S_0 = s] \\
&= \mathbb{E}^{\pi_i}[R_1 + \gamma R_2 + \cdots + \gamma^{i-2} R_{i-1} \mid S_0 = s] + \mathbb{E}^{\pi_i}[\gamma^{i-1} R_i + \gamma^i R_{i+1} \cdots \mid S_0 = s] \\
&= \mathbb{E}^{\pi_{i-1}}[R_1 + \gamma R_2 + \cdots + \gamma^{i-2} R_{i-1} \mid S_0 = s] + \mathbb{E}^{\pi_i}[\gamma^{i-1} R_i + \gamma^i R_{i+1} \cdots \mid S_0 = s] \\
&= \mathbb{E}^{\pi_{i-1}}[R_1 + \gamma R_2 + \cdots + \gamma^{i-2} R_{i-1} \mid S_0 = s] + \underbrace{\mathbb{E}^{\pi_i}[\gamma^{i-1} R_i + \gamma^i R_{i+1} \cdots \mid S_0 = s]}_{\text{Suppose we prove } \geq \mathbb{E}^{\pi_{i-1}}[\gamma^{i-1} R_i + \gamma^i R_{i+1} \cdots \mid S_0 = s]}
\end{aligned}
$$

$$
\begin{aligned}
&\geq \mathbb{E}^{\pi_{i-1}}[R_1 + \gamma R_2 + \cdots + \gamma^{i-2} R_{i-1} + \gamma^{i-1} R_i + \gamma^i R_{i+1} \cdots \mid S_0 = s] \\
&\overset{IH}{\geq} v^{\pi}(s)
\end{aligned}
$$

Proof of PIT (induction, behavior at "reset")

We need: $\mathbb{E}^{\pi_i}[\gamma^{i-1}R_i + \gamma^i R_{i+1} \cdots \mid S_0 = s] \geq \mathbb{E}^{\pi_{i-1}}[\gamma^{i-1}R_i + \gamma^i R_{i+1} \cdots \mid S_0 = s]$.

$$\mathbb{E}^{\pi_i}[\gamma^{i-1}R_i + \gamma^i R_{i+1} \cdots \mid S_0 = s] = \gamma^{i-1} \cdot \mathbb{E}^{\pi_i}[R_i + \gamma R_{i+1} \cdots \mid S_0 = s]$$

$$= \gamma^{i-1} \cdot \sum_{s' \in \mathcal{S}} \mathbb{P}^{\pi_i}[S_{i-1} = s' \mid S_0 = s] \cdot \Big(r(s', \pi_i(s')) + \gamma \cdot \sum_{s''} p(s'' \mid s', \pi_i(s')) \cdot \mathbb{E}^{\pi_i}[G_i \mid S_i = s''] \Big)$$

$$= \gamma^{i-1} \cdot \sum_{s' \in \mathcal{S}} \mathbb{P}^{\pi_g}[S_{i-1} = s' \mid S_0 = s] \cdot \Big(r(s', \pi_g(s')) + \gamma \cdot \sum_{s''} p(s'' \mid s', \pi_g(s')) \cdot \mathbb{E}^{\pi}[G_i \mid S_i = s''] \Big)$$

$$= \gamma^{i-1} \cdot \sum_{s' \in \mathcal{S}} \mathbb{P}^{\pi_g}[S_{i-1} = s' \mid S_0 = s] \cdot \Big(r(s', \pi_g(s')) + \gamma \cdot \sum_{s''} p(s'' \mid s', \pi_g(s')) \cdot \underbrace{\mathbb{E}^{\pi}[G_i \mid S_i = s'']}_{v^{\pi}(s'')} \Big)$$

$$= \gamma^{i-1} \cdot \sum_{s' \in \mathcal{S}} \mathbb{P}^{\pi_g}[S_{i-1} = s' \mid S_0 = s] \cdot \underbrace{\Big(r(s', \pi_g(s')) + \gamma \cdot \sum_{s''} p(s'' \mid s', \pi_g(s')) \cdot \underbrace{\mathbb{E}^{\pi}[G_i \mid S_i = s'']}_{v^{\pi}(s'')} \Big)}_{= \Phi(v^{\pi}), \text{ since } \pi_g \text{ is } v^{\pi}\text{-greedy}}$$

$$= \gamma^{i-1} \cdot \sum_{s' \in \mathcal{S}} \mathbb{P}^{\pi_g}[S_{i-1} = s' \mid S_0 = s] \cdot \underbrace{\Big(r(s', \pi_g(s')) + \gamma \cdot \sum_{s''} p(s'' \mid s', \pi_g(s')) \cdot \underbrace{\mathbb{E}^{\pi}[G_i \mid S_i = s'']}_{v^{\pi}(s'')} \Big)}_{\cdots}$$

Policy iteration (PI; Howard, 1960)

Algorithm 2: Policy iteration

Input: MDP $\mathcal{M} = (\mathcal{S}, \mathcal{A}, p, r)$

Output: Optimal MD policy π^* for $\mathcal{M}$, its value vector v^*

$\pi \leftarrow$ arbitrary MD policy ;

$v \leftarrow v^\pi$; `// e.g. by solving linear policy evaluation equations`

while $\Phi(v) \neq v$ **do**

  foreach $s \in \mathcal{S}$ **do**

    $\pi(s) \leftarrow \arg\max_{a \in \mathcal{A}(s)} [r(s,a) + \gamma \cdot \sum_{s' \in \mathcal{S}} p(s'|s,a) \cdot v(s')]$

  $v \leftarrow v^\pi$

return π, v

Theorem 28

Policy iteration terminates after at most exponentially many iterations. Upon termination, it returns an optimal MD policy.

Proof:

- Optimal upon termination: v^π is a fixpoint of Φ when terminating: optimality follows from Corollary 19.
- Terminates: π always stores an MD policy and there are finitely many of these. We will show that no single MD policy appears in more than one iteration of PI.
  Consider any iteration and let v, v' be the contents of variable v before and after the iteration. We will show that unless $\Phi(v) = v$, it holds $v' > v$, i.e. $v' \geq v$ componentwise with strict inequality in some component. Hence, $v = v^\pi$ strictly increases during PI, so no π can appear twice.

PI: correctness proof

$v' \geq v$:

We verify assumptions of PIT: $\Phi(v) \geq v$. Recall $v = v^\pi$. For all $s \in \mathcal{S}$:

$$\begin{aligned}\Phi(v)(s) &= \max_{a \in \mathcal{A}(s)} [r(s,a) + \gamma \cdot \sum_{s' \in \mathcal{S}} p(s' \mid s,a) \cdot v(s')] \\ &\geq r(s,\pi(s)) + \gamma \cdot \sum_{s' \in \mathcal{S}} p(s' \mid s,\pi(s)) \cdot v^\pi(s') \\ &= v^\pi(s) = v(s).\end{aligned}$$

By PIT, $v' = v^{\pi'} \geq v^\pi = v$ (here π' is the v-greedy policy).

PI: correctness proof II

It remains to prove that $v' > v$ or PI terminates. Assume that $v' = v$. Then for all $s \in \mathcal{S}$:

$$
\begin{aligned}
v'(s) &= r(s, \pi'(s)) + \gamma \cdot \sum_{s' \in \mathcal{S}} p(s' \mid s, \pi'(s)) \cdot v^{\pi'}(s') \\
&= r(s, \pi'(s)) + \gamma \cdot \sum_{s' \in \mathcal{S}} p(s' \mid s, \pi'(s)) \cdot v^{\pi}(s') \quad \text{(assumption)} \\
&= \max_{a \in \mathcal{A}(s)} [r(s, a) + \gamma \cdot \sum_{s' \in \mathcal{S}} p(s' \mid s, a) \cdot v^{\pi}(s')] \quad (\pi' \text{ is } v = v^{\pi}\text{-greedy}) \\
&= \max_{a \in \mathcal{A}(s)} [r(s, a) + \gamma \cdot \sum_{s' \in \mathcal{S}} p(s' \mid s, a) \cdot v^{\pi'}(s')] \quad \text{(assumption)} \\
&= \Phi(v')(s),
\end{aligned}
$$

so PI terminates at this point. Complexity?

PI&VI: discussion

- We know that MDPs have a linear programming (LP) formulation. PI is basically a variant of a simplex method for this LP, using a special pivoting rule.
- PI typicaly requires less iterations to converge than VI, though each iteration is more expensive (policy eval.)
- Both PI and VI typically work well in practice for MDPs whose explicit transition table fits inside a computer. Which of the two is faster is rather domain-specific.

PI variants

Can we get rid of the expensive policy evaluation by linear system solving?

Yes: we can approximate the value of the current policy π by applying VI on the MDP $\mathcal{M}^\pi$, for either fixed number of steps or until v does not change much. Often appearing in RL textbooks:

Policy iteration with approximate evaluation

$\pi \leftarrow$ arbitrary MD policy; $v \leftarrow$ arbitrary vector;

repeat

  $v \leftarrow \texttt{Eval}(\pi, v)$;

  foreach $s \in \mathcal{S}$ **do**

    $\pi(s) \leftarrow \arg\max_{a \in \mathcal{A}(s)} [r(s,a) + \gamma \cdot \sum_{s' \in \mathcal{S}} p(s'|s,a) \cdot v(s')]$

until π *has not changed*;

return π, v

Function $\texttt{Eval}(\pi, v)$**:**

  $v' \leftarrow v$;

  repeat

    foreach $s \in \mathcal{S}$ **do**

      $v(s) \leftarrow v'(s)$;

      $v'(s) \leftarrow r(s, \pi(s)) + \gamma \cdot \sum_{s' \in \mathcal{S}} p(s' \mid s, \pi'(s)) \cdot v(s')$

  until $\|v - v'\|_\infty \leq \varepsilon$;

return v'

Convergence of PI variants

- The algorithm on previous slide still converges to an optimal policy provided that ε is small enough.
- If we replaced the "π not changed condition" with the original "$\Phi(v) = v$" condition, the algorithm might not terminate, since the VI is only guaranteed to reach a true fixpoint in the limit. However, v would still converge to v^* and thus π would eventually become equal to an optimal policy.
- The previous point holds even in the very degenerate case when we do just one iteration of VI per policy evaluation! See next slide.

Curiously looking approximate PI

$v \leftarrow$ arbitrary vector;

$\pi \leftarrow v$-greedy MD policy ;

repeat

  foreach $s \in \mathcal{S}$ **do**

    $v'(s) \leftarrow r(s, \pi(s)) + \gamma \cdot \sum_{s' \in \mathcal{S}} p(s'|s, \pi(s)) \cdot v(s')$;

  $v \leftarrow v'$;

  foreach $s \in \mathcal{S}$ **do**

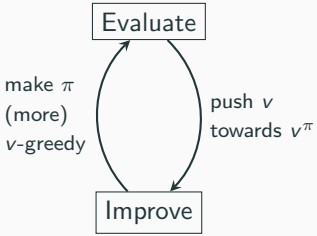
    $\pi(s) \leftarrow \arg\max_{a \in \mathcal{A}(s)} [r(s, a) + \gamma \cdot \sum_{s' \in \mathcal{S}} p(s'|s, a) \cdot v(s')]$

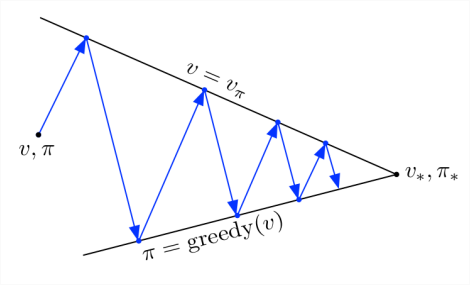
until $\Phi(v) = v$;

return π, v

This is just VI in disguise!

Generalized policy iteration





Source: Sutton&Barto, p. 87

Tabular Methods for Model-Free Reinforcement Learning

Model-free

We will still be working with MDPs. But for a bunch of the following lectures, we will not (necessarily) have access to, e.g.:

- a table containing explicit enumeration of all states/actions
- a table containing the description of p or r
- the ability to compute the probability vector $\delta(s, a)$ or the reward signal $r(s, a)$ given s and a (having this = gray-box model of the MDP)

Sampling from MDP

But the MDP is still there "behind the scene". In particular, we:

- know how the states of the MDP look like
  - (e.g. robot state = all possible output values of its sensors)
- know how the actions of the MDP look like
  - (e.g. robot = all possible signals that can be sent to the actuators)
- can, for any $s \in \mathcal{S}$, enumerate $\mathcal{A}(s)$
  - could be weakened, but simplifies things
- given $s \in \mathcal{S}$ and $a \in \mathcal{A}(s)$, we can sample the next state $s' \sim p(s, a)$ and receive the reward $r(s, a)$.

Sampling from a policy

Given an effective representation of a policy π, we can sample a trajectory $s_0, a_0, r_1, s_1, a_1, r_2, s_2, a_2, r_3, s_3, \ldots$ by performing, for each $t \in \{0, \ldots, T\}$:

- sample $a_t \sim \pi(s_t)$
- query the environment for $s_{t+1} \sim p(s_t, a_t)$ and $r_{t+1} = r(s_t, a_t)$
- increment t

Tabular = value estimates and policies represented as tables (e.g. $Q(s, a)$ for each state s and action a used in s – explicit representation might only be needed for states/actions actually encountered).

Basic classification of (tabular) RL algorithms

Three independent axes:

problem	on/off	updates
policy evaluation (value prediction)	on-policy	Monte Carlo
vs.	vs.	
control	off-policy	temporal difference

Assumptions: successor-dependent rewards & episodic tasks

Since we do no longer have the knowledge of the transition dynamics p, we cannot freely interchange MDPs with rewards functions of type $\mathcal{S} \times \mathcal{A} \times \mathcal{S} \to \mathbb{R}$ and $\mathcal{S} \times \mathcal{A} \to \mathbb{R}$ via the equation $r(s, a) = \sum_{s' \in \mathcal{S}} p(s' \mid s, a) \cdot r(s, a, s')$.
Hence, to maintain generality (and correspondence to e.g. Gymnasium environments) we will assume reward functions of type $\mathcal{S} \times \mathcal{A} \times \mathcal{S} \to \mathbb{R}$.

We will assume episodic returns: each trajectory terminates with probability 1 at some (possibly random) time step T. Termination can be defined e.g. by reaching some terminal state or by running out of some fixed decision horizon (in Gymnasium, this is sometimes called truncation):

$$G = \sum_{i=0}^{T-1} \gamma^i \cdot R_{i+1}.$$

Episode = one high-level iteration of an RL algorithm, corresponding of sampling a single trajectory from some policy.

Monte Carlo Methods

MC evaluation

Policy evaluation: given an effective representation of a policy π, estimate v^π (or q^π).

Naive Monte Carlo: Sample from π: if $\{\tau_1, \tau_2, \ldots, \tau_n\}$ are trajectories (episodes) independently sampled under π from the same initial state s, then $\frac{1}{n}\sum_{i=1}^{n} G(\tau_i) \to v^\pi(s)$ as $n \to \infty$ due to law of large numbers (LLN).

But this throws away a lot of valuable information! E.g. what if we want to estimate the whole v^π?

First-visit MC

For each s, we estimate $v^\pi(s)$ as an average of sample returns $Ret(s)$ which is formed as follows:

- initially, $Ret(s) = \emptyset$ for all s
- we then sample trajectories until timeout:
  - for each sampled trajectory τ and each state s, we identify the first occurrence of s on τ: let this be at timestep t; we add $G_t(\tau)$ to $Ret(s)$

Sub-trajectory starting at the first appearance of s can be seen as a trajectory sampled from π when s is the initial state! (Since we consider memoryless π.)

Theorem 29

As $|Ret(s)| \to \infty$, the average of $Ret(s)$ converges to $v^\pi(s)$. Moreover, the average of $Ret(s)$ is an unbiased estimate of $v^\pi(s)$ (as long as $Ret(s) \neq \emptyset$).

First-visit MC (pseudocode)

First-visit MC prediction, for estimating $V \approx v_\pi$

Input: a policy π to be evaluated

Initialize:

$V(s) \in \mathbb{R}$, arbitrarily, for all $s \in \mathcal{S}$

$Returns(s) \leftarrow$ an empty list, for all $s \in \mathcal{S}$

Loop forever (for each episode):

Generate an episode following π: $S_0, A_0, R_1, S_1, A_1, R_2, \ldots, S_{T-1}, A_{T-1}, R_T$

$G \leftarrow 0$

Loop for each step of episode, $t = T-1, T-2, \ldots, 0$:

$G \leftarrow \gamma G + R_{t+1}$

Unless S_t appears in $S_0, S_1, \ldots, S_{t-1}$:

Append G to $Returns(S_t)$

$V(S_t) \leftarrow \text{average}(Returns(S_t))$

Source: Sutton&Barto, p.92

Every-visit MC

For each s, we estimate $v^\pi(s)$ as an average of sample returns $Ret(s)$ which is formed as follows:

- initially, $Ret(s) = \emptyset$ for all s
- we then sample trajectories until timeout:
  - for each sampled trajectory τ and each state s, and each t such that $S_t(\tau) = s$ we add $G_t(\tau)$ to $Ret(s)$

The sample returns added to $Ret(s)$ within the same episode are not independent! Hence, the estimate is biased, though the bias vanishes in the limit:

Theorem 30

As $|Ret(s)| \to \infty$, the average of $Ret(s)$ converges to $v^\pi(s)$.

MC convergence

Optional reading: More on MC estimate bias, variance, and convergence in:

Singh, S.P. and Sutton, R.S.: Reinforcement Learning with Replacing Eligibility Traces. In *Machine Learning* 22:123–158. Kluwer, 1996.
(Section 3, particularly 3.3 and onwards, you can skip Theorem 4.)

Towards MC control

Control = computation of "good" policy for a given environment. (Ideally, the policy should get closer to the optimal policy the more episodes we sample.)

We know (PIT): given a policy π a v^π-greedy policy is at least as good as π:

$$\pi_g(s) = \arg\max_{a \in \mathcal{A}(s)} \Big[\sum_{s' \in \mathcal{S}} p(s' \mid s, a) \cdot \big(r(s, a, s') + \gamma \cdot v^\pi(s') \big) \Big]$$

Do we have an algo? There is an issue:

MC control with q-values

Recall:

$$q^{\pi}(s,a) \stackrel{\text{def}}{=} \sum_{s' \in \mathcal{S}} p(s' \mid s,a) \cdot \big(r(s,a,s') + \gamma \cdot v^{\pi}(s').\big)$$

Thus, the v^{π}-greedy policy π_g can be defined as:

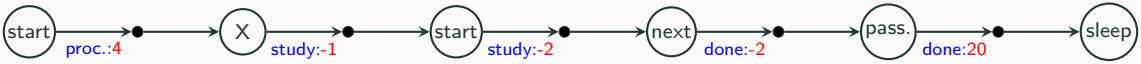
$$\pi_g(s) = \arg\max_{a \in \mathcal{A}(s)} \underbrace{q^{\pi}(s,a)}_{\text{Estimate by MC.}}$$

MC for q-value estimation

Analogous to value estimation, e.g. first-visit:

For each s, a, we estimate $q^\pi(s, a)$ as an average of sample returns $Ret(s, a)$ which is formed as follows:

- initially, $Ret(s, a) = \emptyset$ for all s
- we then sample trajectories until timeout:
  - for each sampled trajectory τ and each state-action pair (s, a), we identify the first t such that $S_t(\tau) = s \wedge A_t(\tau) = a$; we add $G_t(\tau)$ to $Ret(s)$



Similarly for every visit. Convergence guarantees the same as for state values.

Infinite exploration and exploring starts

Issue: MC only estimates $q^\pi(s, a)$ if:

- s guaranteed to be visited with positive probability in each episode
- $\pi(a \mid s) > 0$.

Definition 31: Infinite exploration

A RL algorithm has infinite exploration (IE) if, during the infinite execution of the algorithm, each state-action pair (s, a) is visited infinitely often with probability 1.

One way of achieving IE is through exploring starts (ES): each episode begins with (typically uniformly) randomly selected s_0 and a_0. This is achievable when training, e.g., in simulated environments but might be difficult/impossible in real-world environments.

MC control with exploring starts

Monte Carlo ES (Exploring Starts), for estimating $\pi \approx \pi_*$

Initialize:

$\pi(s) \in \mathcal{A}(s)$ (arbitrarily), for all $s \in \mathcal{S}$

$Q(s, a) \in \mathbb{R}$ (arbitrarily), for all $s \in \mathcal{S}$, $a \in \mathcal{A}(s)$

$Returns(s, a) \leftarrow$ empty list, for all $s \in \mathcal{S}$, $a \in \mathcal{A}(s)$

Loop forever (for each episode):

Choose $S_0 \in \mathcal{S}$, $A_0 \in \mathcal{A}(S_0)$ randomly such that all pairs have probability > 0

Generate an episode from S_0, A_0, following π: $S_0, A_0, R_1, \ldots, S_{T-1}, A_{T-1}, R_T$

$G \leftarrow 0$

Loop for each step of episode, $t = T-1, T-2, \ldots, 0$:

$G \leftarrow \gamma G + R_{t+1}$

Unless the pair S_t, A_t appears in $S_0, A_0, S_1, A_1 \ldots, S_{t-1}, A_{t-1}$:

Append G to $Returns(S_t, A_t)$

$Q(S_t, A_t) \leftarrow \text{average}(Returns(S_t, A_t))$

$\pi(S_t) \leftarrow \arg\max_a Q(S_t, a)$

Source: Sutton&Barto, p.99

IE through ε-soft policies

Exploring starts are not always feasible. Alternative: make the sampled policy itself exploratory.

Definition 32: ε-soft policy

A policy π is ε-soft if for every $s \in \mathcal{S}$ and every $a \in \mathcal{A}(s)$ it holds $\pi(a|s) \geq \frac{\varepsilon}{|\mathcal{A}(s)|}$.

Definition 33: ε-greedy policy

Let $v \in \mathbb{R}^{\mathcal{S}}$ be a value vector. A policy π is v-ε-greedy if for every state $s \in \mathcal{S}$ there is action $a^* = \arg\max_{a \in \mathcal{A}(s)} \sum_{s' \in \mathcal{S}} p(s' \mid s, a) \cdot \big(r(s, a, s') + \gamma \cdot v(s')\big)$ such that for any action $a \in \mathcal{A}(s)$ it holds:

$$\pi(a|s) = \begin{cases} \frac{\varepsilon}{\mathcal{A}(s)} & \text{if } a \neq a^* \\ 1 - \varepsilon + \frac{\varepsilon}{\mathcal{A}(s)} & \text{if } a = a^*. \end{cases}$$

Interpretation: with prob. ε: play uniformly at random; with prob. $1 - \varepsilon$: play greedily.

ε-softing

Definition 34

Let π be a policy. An ε-softing of π is a policy π_ε defined as follows: in each state s

- with probability ε, π_ε selects an action uniformly at random;
- with probability $1-\varepsilon$, π_ε selects $a \sim \pi(s)$.

I.e., an ε-greedy policy can be alternatively defined as ε-softing of a greedy policy.

MC control with ε-greedy policies

Algorithm parameter: small $\varepsilon > 0$

Initialize:

$\pi \leftarrow$ an arbitrary ε-soft policy

$Q(s,a) \in \mathbb{R}$ (arbitrarily), for all $s \in \mathcal{S}$, $a \in \mathcal{A}(s)$

$Returns(s,a) \leftarrow$ empty list, for all $s \in \mathcal{S}$, $a \in \mathcal{A}(s)$

Repeat forever (for each episode):

Generate an episode following π: $S_0, A_0, R_1, \ldots, S_{T-1}, A_{T-1}, R_T$

$G \leftarrow 0$

Loop for each step of episode, $t = T-1, T-2, \ldots, 0$:

$G \leftarrow \gamma G + R_{t+1}$

Unless the pair S_t, A_t appears in $S_0, A_0, S_1, A_1 \ldots, S_{t-1}, A_{t-1}$:

Append G to $Returns(S_t, A_t)$

$Q(S_t, A_t) \leftarrow \text{average}(Returns(S_t, A_t))$

$A^* \leftarrow \arg\max_a Q(S_t, a)$ (with ties broken arbitrarily)

For all $a \in \mathcal{A}(S_t)$:

$$\pi(a|S_t) \leftarrow \begin{cases} 1 - \varepsilon + \varepsilon/|\mathcal{A}(S_t)| & \text{if } a = A^* \\ \varepsilon/|\mathcal{A}(S_t)| & \text{if } a \neq A^* \end{cases}$$

source: Sutton&Barto, p. 101

Policy iteration for ε-soft policies

Theorem 35

Let π be an ε-soft policy and let π' be a v^{π}-ε-greedy policy. Than $v^{\pi'} \geq v^{\pi}$ (componentwise). Moreover, the two value vectors are equal if and only if bot π and π' are optimal among all ε-soft policies; i.e. if, for every state s:

$$v^{\pi}(s) = \sup_{\bar{\pi} \text{ that is } \varepsilon\text{-soft}} v^{\bar{\pi}}(s).$$

Proof: Required reading: Sutton&Barto, p.101-103.

Incremental computing of averages

Given a sample $\{n_1, n_2, \ldots, n_{k+1}\}$ and average $A = avg(\{n_1, n_2, \ldots, n_k\})$, how to compute $A' = avg(\{n_1, n_2, \ldots, n_k, n_{k+1}\})$ without recomputing the average of the whole sample?

$$A' = \frac{k}{k+1} \cdot A + \frac{n_{k+1}}{k+1}.$$

On-policy vs. off-policy

- On-policy algorithms: track one "policy variable" π; the policy stored in π is used to interact with the environment (i.e., to sample episodes) and at the same time we learn something about it (e.g. its value vector).
  - Corresponds to the generalized policy iteration scheme.
  - All the MC algos we have seen so far.
- Off-policy algorithms: track more (typically two) different policy variables:
  - behavior policy: used to sample episodes
  - target policy: which we want to learn about

Off-policy evaluation

We are given effective representations of:

- a behavior policy β,
- a target policy π.

The task is to estimate v^π by sampling episodes from β . We cannot sample from π! (E.g. π too risky or expensive to sample from.)

Assumptions:

- given (s, a), we can effectively compute $\pi(a|s)$ and $\beta(a|s)$ (or at least estimate via sampling)
- coverage: $\forall s \in \mathcal{S}, a \in \mathcal{A}(s)$: if $\pi(a|s) > 0$, then also $\beta(a|s) > 0$

Definition 36: Importance ratio

Let $\tau = s_0, a_0, r_1, s_1, a_1, r_2, s_2, a_2, r_3, s_3, \dots$ be a trajectory. The importance-sampling ratio of τ is the quantity

$$\begin{aligned}\rho(\tau) &\stackrel{def}{=} \frac{\mathbb{P}^{\pi}[\tau \mid S_0 = s_0]}{\mathbb{P}^{\beta}[\tau \mid S_0 = s_0]} \\ &= \frac{\mathbb{P}^{\pi}[A_0 = a_0, S_1 = s_1, A_1 = a_1, \dots, A_{T-1} = a_{T-1}, S_T = s_T \mid S_0 = s_0]}{\mathbb{P}^{\beta}[A_0 = a_0, S_1 = s_1, A_1 = a_1, \dots, A_{T-1} = a_{T-1}, S_T = s_T \mid S_0 = s_0]}.\end{aligned}$$

$\rho(\tau)$ can be computed without the knowledge of MDP transition probabilities!

$$\rho(\tau) = \frac{\pi(a_0 \mid s_0) \cdot p(s_1 \mid s_0, a_0) \cdot \pi(a_1 \mid s_1) \cdot p(s_2 \mid s_1, a_1) \cdots}{\beta(a_0 \mid s_0) \cdot p(s_1 \mid s_0, a_0) \cdot \beta(a_1 \mid s_1) \cdot p(s_2 \mid s_1, a_1) \cdots}$$

Importance ratio from time t

Definition 37

Let $\tau = s_0, a_0, r_1, s_1, a_1, r_2, s_2, a_2, r_3, s_3, \ldots$ be a trajectory. By $\tau_{i..j}$ we denote the sub-trajectory of τ starting in time step i and ending in timestep j. By $\tau_{i..}$ we denote the suffix of $s_i, a_i, r_{i+1}, s_{i+1}, a_{i+1}, \ldots$.

Definition 38: Importance ratio from time t

Let $\tau = s_0, a_0, r_1, s_1, a_1, r_2, s_2, a_2, r_3, s_3, \ldots$ be a trajectory and t a time step. The importance-sampling ratio of τ from t is the quantity

$$\begin{aligned}\rho_t(\tau) &\stackrel{def}{=} \frac{\mathbb{P}^{\pi}[\tau_{t..} \mid S_0 = s_t]}{\mathbb{P}^{\beta}[\tau_{t..} \mid S_0 = s_t]} \\ &= \frac{\mathbb{P}^{\pi}[A_0 = a_t, S_1 = s_{t+1}, A_1 = a_{t+1}, \ldots, A_{T-1-t} = a_{T-1}, S_{T-t} = s_T \mid S_0 = s_t]}{\mathbb{P}^{\beta}[A_0 = a_t, S_1 = s_{t+1}, A_1 = a_{t+1}, \ldots, A_{T-1-t} = a_{T-1}, S_{T-t} = s_T \mid S_0 = s_t]}.\end{aligned}$$

Off-policy evaluation with importance sampling

Theorem 39

For any $s \in \mathcal{S}$ it holds:

$$\mathbb{E}^{\beta}[\rho \cdot G \mid S_0 = s] = v^{\pi}(s).$$

Proof:

$$\begin{aligned}
\mathbb{E}^{\beta}[\rho \cdot G \mid S_0 = s] &= \sum_{\tau} \mathbb{P}^{\beta}[\tau \mid S_0 = s] \cdot \rho(\tau) \cdot G(\tau) \\
&= \sum_{\tau} \mathbb{P}^{\beta}[\tau \mid S_0 = s] \cdot \frac{\mathbb{P}^{\pi}[\tau \mid S_0 = s]}{\mathbb{P}^{\beta}[\tau \mid S_0 = s]} \cdot G(\tau) \\
&= \sum_{\tau} \mathbb{P}^{\pi}[\tau \mid S_0 = s] \cdot G(\tau) = \mathbb{E}^{\pi}[G \mid S_0 = s] = v^{\pi}(s).
\end{aligned}$$

Easily integrates into both first-visit and every visit MC: sample from β and store $\rho_t(\tau) \cdot G_t(\tau)$ in $Ret(s_t)$.

Weighted importance sampling

First-visit variant: for each state s, we keep a set of samples $Sam(s)$. Each sample is a tuple (τ, t) – trajectory and time step.

- initially, $Sam(s) = \emptyset$ for all s
- we then sample trajectories until timeout:
  - for each sampled trajectory τ and each state s, and the smallest t such that $S_t(\tau) = s$ we add (τ, t) to $Sam(s)$

Throughout the algorithm, the value of state s is estimated as

$$WIS(s) = \frac{\sum_{(\tau,t)\in Sam(s)} \rho_t(\tau) \cdot G_t(\tau)}{\sum_{(\tau,t)\in Sam(s)} \rho_t(\tau)}$$

Exercise 40

Compare ordinary/weighted importance sampling after single sample.

Weighted importance sampling – correctness

The weighted sampling is clearly a biased estimator. However, the bias vanishes in the limit:

Theorem 41

With probability 1: as $|Sam(s)| \to \infty$, we have that $WIS(s) \to v^{\pi}(s)$.

Proof:

Ordinary vs. weighted sampling

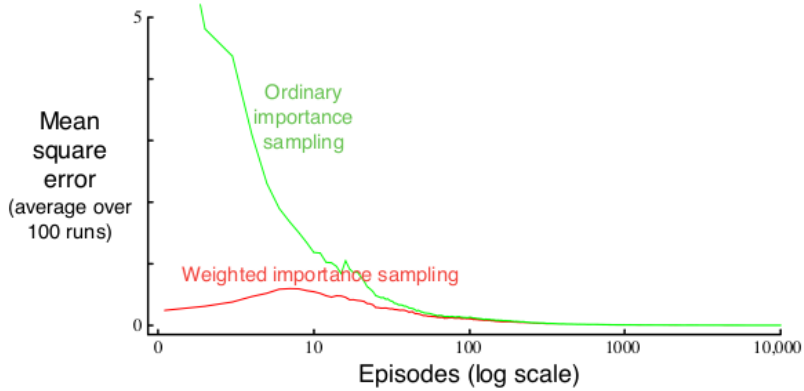


Figure 5.3: Weighted importance sampling produces lower error estimates of the value of a single blackjack state from off-policy episodes. ■

source: Sutton&Barto, p. 106

Importance sampling: summary

But ordinary and weighted importance sampling can be adapted to every-visit MC.

Bias & Convergence:

- First visit:
  - ordinary IS: unbiased, i.e. also converges
  - weighted IS: biased, but converges in the limit
- Every visit:
  - both ordinary and weighted: biased (due to EV), but converges in the limit

Weighted IS: incremental implementation

Instead of recomputing the weighted average for each new sample, $WIS(s)$ can be updated by keeping keep just two variables:

- V – current value of $WIS(s)$, initially arbitrary
- C – the sum of importance ratios, initially 0

Upon arrival of new sample (τ', t'), we update V, C into new values V', C' by setting:

$$C' = C + \rho_{t'}(\tau')$$
$$V' = V + \frac{\rho_{t'}(\tau')}{C'} \cdot (G_{t'}(\tau') - V)\,.$$

Off-policy evaluation with weighted IS

Off-policy MC prediction (policy evaluation) for estimating $Q \approx q_\pi$

Input: an arbitrary target policy π
Initialize, for all $s \in \mathcal{S}$, $a \in \mathcal{A}(s)$:
    $Q(s,a) \in \mathbb{R}$ (arbitrarily)
    $C(s,a) \leftarrow 0$

Loop forever (for each episode):
    $b \leftarrow$ any policy with coverage of π
    Generate an episode following b: $S_0, A_0, R_1, \ldots, S_{T-1}, A_{T-1}, R_T$
    $G \leftarrow 0$
    $W \leftarrow 1$
    Loop for each step of episode, $t = T-1, T-2, \ldots, 0$, while $W \neq 0$:
        $G \leftarrow \gamma G + R_{t+1}$
        $C(S_t, A_t) \leftarrow C(S_t, A_t) + W$
        $Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \frac{W}{C(S_t, A_t)} [G - Q(S_t, A_t)]$
        $W \leftarrow W \frac{\pi(A_t|S_t)}{b(A_t|S_t)}$

Off-policy control with weighted IS

Required reading: Sutton&Barto, Section 5.7.

Temporal Difference Methods

TD: Motivation

Let us first focus on policy evaluation.

MC: zero bias (at least in the limit), but potentially high variance: many samples needed to converge. Also, to update estimates, it must wait till the end of each episode.

TD methods retain the focus on sampling but combine it with bootstrapping.

Incremental update notation

Definition 42: Notation for updates

In the context of RL algorithms will denote by $V^n(s)$ (resp. $Q^n(s,a)$) the algorithm's estimate of $v^\pi(s)$ (resp. $q^\pi(s,a)$) after n-th update of this estimate.

MC vs. TD(0) update

On-policy MC (incremental) update using sampled trajectory
$\tau = s_0, a_0, r_1, s_1, a_1, r_2, s_2, a_2, r_3, s_3, \ldots$:

$$V^{n+1}(s_t) \leftarrow (1-\alpha_n)V^n(s_t) + \alpha_n G_t(\tau) = V^n(s_t) + \alpha_n \cdot \big[\underbrace{G_t(\tau)}_{\text{update target}} - V^n(s_t)\big]}_{\text{update error}},$$

where $\alpha_n = n/(n+1)$.

TD(0) update in the same situation, with α_n "suitably chosen" (possibly constant):

$$V^{n+1}(s_t) \leftarrow V^n(s_t) + \alpha_n \cdot \big[R_{t+1}(\tau) + \gamma \cdot \underbrace{V^n(S_{t+1}(\tau))}_{\text{bootstrap}} - V^n(s_t)\big]$$

Policy evaluation with TD(0)

Tabular TD(0) for estimating v_π

Input: the policy π to be evaluated
Algorithm parameter: step size $\alpha \in (0, 1]$
Initialize $V(s)$, for all $s \in \mathcal{S}^+$, arbitrarily except that $V(\mathit{terminal}) = 0$

Loop for each episode:
  Initialize S
  Loop for each step of episode:
    $A \leftarrow$ action given by π for S
    Take action A, observe R, S'
    $V(S) \leftarrow V(S) + \alpha\big[R + \gamma V(S') - V(S)\big]$
    $S \leftarrow S'$
  until S is terminal

source: Sutton&Barto, p. 120

How can it even work?

Really "just" a very asynchronous, sample-based, and "α-dampened" version of value iteration.

$$\mathbb{E}^\pi[G_t|S_t = s] = \mathbb{E}^\pi[R_{t+1} + \gamma \cdot G_{t+1} \mid S_t = s] = \mathbb{E}^\pi[R_{t+1} \mid S_t = s] + \gamma \cdot \underbrace{\mathbb{E}^\pi[G_{t+1} \mid S_t = s]}_{v^\pi(S_{t+1})}.$$

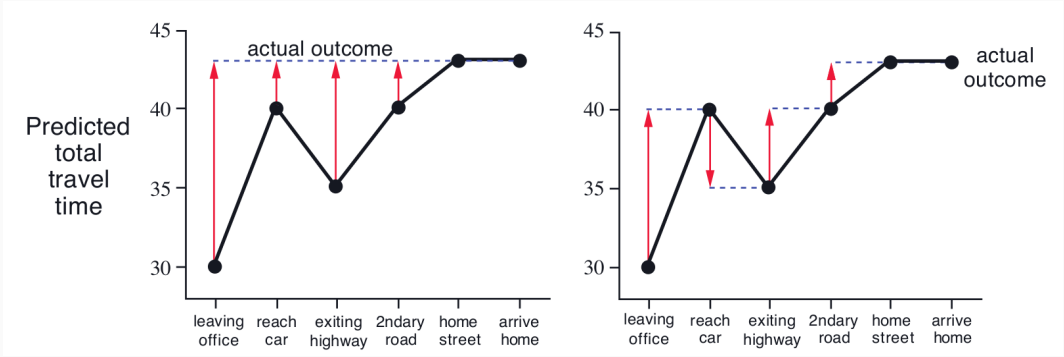
In expectation, the TD(0) update is the same as VI update in $\mathcal{M}^\pi$. Thanks to the contractivity of the Bellman operator, VI possesses an error reduction property: after each update, the error of the estimate decreases. Hence, in expectation, the same is true for the TD(0) update.

Formal proof of correctness in optional reading:

> Sutton, R.S.: Learning to Predict by Methods of Temporal Differences. In *Machine Learning* 3:9–44. Kluwer, 1988. (For MDPs with function approximation.)

Why TD is natural (Sutton&Barto, p. 122-123)

State	*Elapsed Time (minutes)*	*Predicted Time to Go*	*Predicted Total Time*
leaving office, friday at 6	0	30	30
reach car, raining	5	35	40
exiting highway	20	15	35
2ndary road, behind truck	30	10	40
entering home street	40	3	43
arrive home	43	0	43



Left: MC. Right TD(0).

On-policy TD control

Recall:

- In control setting, we need to estimate q-values of a policy.
- On-policy: we sample trajectories according to some policy π and then push value estimates towards q^π.

To maintain exploration, the policy π will typically be the ε-Q-greedy policy for some $\varepsilon > 0$, where Q are the current Q values estimates. I.e., throughout the algorithm

$$\pi(a|s) = \begin{cases} 1 - \varepsilon + \frac{\varepsilon}{\mathcal{A}(s)} & \text{if } a = \arg\max_{a' \in \mathcal{A}(s)} Q(s, a') \\ \frac{\varepsilon}{\mathcal{A}(s)} & \text{otherwise.} \end{cases}$$

SARSA

State-Action-Reward-State-Action. Introduced in Rummery, Niranjan: *On-Line Q-Learning Using Connectionist Systems* (1994).

In each episode, sample a trajectory $\tau = s_0, a_0, r_1, s_1, a_1, r_2, s_2, a_2, r_3, s_3, \ldots, s_T$ according to current policy π; for each time step $0 \leq t \leq T-1$, perform the following update:

$$Q^{n+1}(s_t, a_t) = Q^n(s_t, a_t) + \alpha_n \cdot \Big[r_{t+1} + \gamma Q^n(s_{t+1}, a_{t+1}) - Q^n(s_t, a_t) \Big]$$

The update can be performed immediately when s_{t+1} and a_{t+1} is known (no need to wait for the episode to terminate).

After the episode ends, make π ε-Q-greedy.

Conforms to the GVI scheme.

SARSA pseudocode

Sarsa (on-policy TD control) for estimating $Q \approx q_*$

Algorithm parameters: step size $\alpha \in (0, 1]$, small $\varepsilon > 0$
Initialize $Q(s, a)$, for all $s \in \mathcal{S}^+, a \in \mathcal{A}(s)$, arbitrarily except that $Q(\mathit{terminal}, \cdot) = 0$

Loop for each episode:
  Initialize S
  Choose A from S using policy derived from Q (e.g., ε-greedy)
  Loop for each step of episode:
    Take action A, observe R, S'
    Choose A' from S' using policy derived from Q (e.g., ε-greedy)
    $Q(S, A) \leftarrow Q(S, A) + \alpha\big[R + \gamma Q(S', A') - Q(S, A)\big]$
    $S \leftarrow S'$; $A \leftarrow A'$;
  until S is terminal

source: Sutton&Barto, p. 130

GLIE policies

Definition 43: GLIE condition

A RL algorithm is greedy in the limit (GL) if its behavior policy (=target policy in on-policy algorithms) converges to a 0-greedy policy with increasing number of episodes.
A RL algorithm is GLIE if it is GL and IE (infinitely exploring).

Typical ways of ensuring GLIE:

- Dynamically adjust ε in ε-greedy policy selection" When selecting action in state s, behave ε-greedily with $\varepsilon = \frac{c}{n(s)}$, where $0 < c < 1$ is a constant and $n(s)$ is a number of visits to state s over all the episodes so far.
- Use Boltzmann (softmax) exploration:

$$\pi(a \mid s) = \frac{e^{\frac{Q(s,a)}{\eta(s)}}}{\sum_{b \in \mathcal{A}(s)} e^{\frac{Q(s,b)}{\eta(s)}}},$$

where η is a state-dependent and time-varying temperature parameter. We need η to converge to 0 over time, but not too fast (often, $\eta(s)$ proportional to $\frac{1}{\log(n(s))}$).

Convergence of SARSA

Theorem 44

Consider a GLIE instantiation of SARSA. Moreover, assume that the sequence of learning rates $(\alpha_n)_{n\in\mathbb{N}}$ satisfies $\sum_n \alpha_n = \infty$ and $\sum_n \alpha_n^2 < \infty$. In this setting, Q converges to q^* and the behavior policy of SARSA converges to some optimal policy π^*.

For the proof, see optional reading: Singh, Jaakkola, Littman, Szepesvári: Convergence Results for Single-Step On-Policy Reinforcement-Learning Algorithms. In *Machine Learning* 39:287-308. Kluwer, 2000.

Note: learning rate can itself be state/action dependent (omitted for conciseness, constant learning rates preferred in practice).

Off-policy TD control

Surprise: no importance sampling!

Recall the SARSA update:

$$Q^{n+1}(s_t, a_t) = Q^n(s_t, a_t) + \alpha_n \cdot \Big[r_{t+1} + \gamma Q^n(s_{t+1}, a_{t+1}) - Q^n(s_t, a_t) \Big]$$

It pushes Q towards q^π, where π is the current policy.

Idea: push Q directly towards q^*.

We could do this e.g. by a VI-like update:

$$Q^{n+1}(s_t, a_t) = Q^n(s_t, a_t) + \alpha_n \cdot \Big[\max_{a \in \mathcal{A}(s)} \sum_{s' \in \mathcal{S}} p(s' \mid s, a) \big(r(s, a, s') + \gamma V(s') \big) - Q^n(s_t, a_t) \Big].$$

Two problems:

- We do not calculate v-estimates. (Must somehow replace with Q)
- We must get rid of transition probabilities and instead use the sampled a_t and r_{t+1}.

Q-learning update

Solution: push the max towards the bootstrap.

Q-learning (Watkins, 1989): given a sampled trajectory $s_0, a_0, r_1, s_1, a_1, r_2, s_2, a_2, r_3, s_3, \ldots$, for every t we update:

$$Q^{n+1}(s_t, a_t) = Q^n(s_t, a_t) + \alpha_n \cdot \left[r_{t+1} + \gamma \cdot \left(\max_{a \in \mathcal{A}(s_{t+1})} Q^n(s_{t+1}, a) \right) - Q^n(s_t, a_t) \right]$$

Q-learning pseudocode

Q-learning (off-policy TD control) for estimating $\pi \approx \pi_*$

Algorithm parameters: step size $\alpha \in (0, 1]$, small $\varepsilon > 0$
Initialize $Q(s, a)$, for all $s \in \mathcal{S}^+, a \in \mathcal{A}(s)$, arbitrarily except that $Q(\mathit{terminal}, \cdot) = 0$

Loop for each episode:
  Initialize S
  Loop for each step of episode:
    Choose A from S using policy derived from Q (e.g., ε-greedy)
    Take action A, observe R, S'
    $Q(S, A) \leftarrow Q(S, A) + \alpha\big[R + \gamma \max_a Q(S', a) - Q(S, A)\big]$
    $S \leftarrow S'$
  until S is terminal

source: Sutton&Bato, p. 131

Off-policy: where is the second policy?

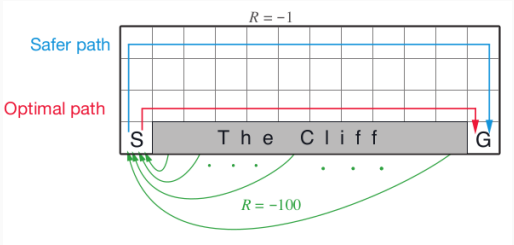
Q-learning convergence

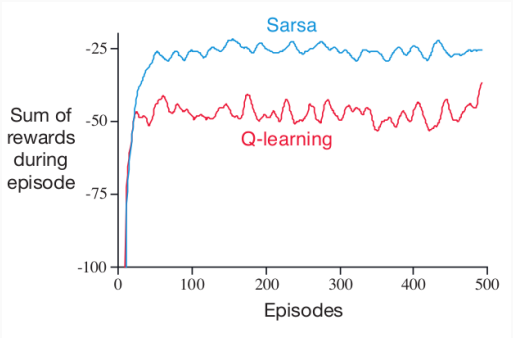
Theorem 45

Consider any Q-learning instantiation with infinite exploration. Assume that the sequence of learning rates $(\alpha_n)_{n\in\mathbb{N}}$ satisfies $\sum_n \alpha_n = \infty$ and $\sum_n \alpha_n^2 < \infty$. In this setting, Q converges to q^*. Moreover, if the behavior policy is GL, then it converges to an optimal policy π^*.

Proof in optional reading: Watkins, Dayan: Q-Learning. In *Machine Learning* 8:279-292. Kluwer, 1992.

SARSA vs. Q-learning (SB: p. 132)





Left: greedy policies learned by SARSA and Q-learning.

Right: in-training performance with a 0.1-greedy behavior policy.

(Rough) takeaway: Q-learning more aggressive in finding optimal policy, can lead to risky behavior. Possibly advantageous when environment not too stochastic or if in-training performance has less importance (simulator vs. real world).

Maximization bias in Q-learning

Q-learning is "risky" not only due to exploration, but also because it is optimistic in the face of uncertainty. TBD

The positive bias only disappears in the limit.

Double Q-learning

Idea: use two independent value estimates Q_1, Q_2 and decouple action selection from evaluation in the bootstrap. During each update, we randomly select one of these for update, which is also used to select the maximizing action in bootstrap. The other is used as the bootstrap estimate.

I.e., in each time step t we perform one of these updates, each with probability $\frac{1}{2}$: either

$$Q_1(s_t, a_t) = Q_1(s_t, a_t) + \alpha_n \cdot \Big[r_{t+1} + \gamma \cdot Q_2\big(s_{t+1}, (\max_{a \in \mathcal{A}(s_{t+1})} Q_1(s_{t+1}, a))\big) - Q_1(s_t, a_t) \Big]$$

or

$$Q_2(s_t, a_t) = Q_2(s_t, a_t) + \alpha_n \cdot \Big[r_{t+1} + \gamma \cdot Q_1\big(s_{t+1}, (\max_{a \in \mathcal{A}(s_{t+1})} Q_2(s_{t+1}, a))\big) - Q_2(s_t, a_t) \Big].$$

Behavior policy = e.g. ε-greedy w.r.t. $Q_1 + Q_2$.

Double Q-learning pseudocode

Double Q-learning, for estimating $Q_1 \approx Q_2 \approx q_*$

Algorithm parameters: step size $\alpha \in (0, 1]$, small $\varepsilon > 0$
Initialize $Q_1(s, a)$ and $Q_2(s, a)$, for all $s \in \mathcal{S}^+, a \in \mathcal{A}(s)$, such that $Q(\mathit{terminal}, \cdot) = 0$

Loop for each episode:
  Initialize S
  Loop for each step of episode:
    Choose A from S using the policy ε-greedy in $Q_1 + Q_2$
    Take action A, observe R, S'
    With 0.5 probabilility:

$$Q_1(S, A) \leftarrow Q_1(S, A) + \alpha\Big(R + \gamma Q_2\big(S', \operatorname{argmax}_a Q_1(S', a)\big) - Q_1(S, A)\Big)$$

    else:

$$Q_2(S, A) \leftarrow Q_2(S, A) + \alpha\Big(R + \gamma Q_1\big(S', \operatorname{argmax}_a Q_2(S', a)\big) - Q_2(S, A)\Big)$$

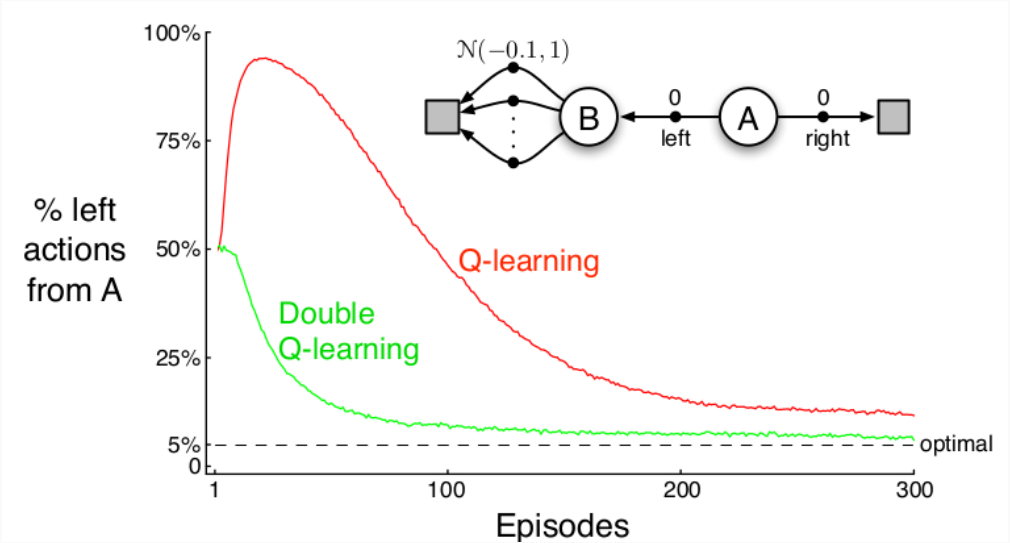
    $S \leftarrow S'$
  until S is terminal

source: Sutton&Barto, p. 136

Why Double Q-learning helps

TBD

Double Q-learning: experiment



source: Sutton&Barto, p. 135

Between Monte Carlo and TD: n-Step and λ-Returns

MC vs TD(0) update targets

Given a trajectory $s_0, a_0, r_1, s_1, a_1, r_2, s_2, a_2, r_3, s_3, \ldots$:

Update for time step t in:

- MC = discounted return from t till the end of trajectory, e.g. for $t = 1$:

$$r_2 + \gamma r_3 + \gamma^2 r_4 + \cdots + \gamma^{T-2} r_T$$

  unbiased, but high variance + need the whole trajectory

- TD(0) = 1-step reward and then (discounted) bootstrap:

$$r_2 + \gamma V(s_2)$$

n-step return

Idea: use n-step discounted return and then bootstrap

Definition 46

Let $\tau = s_0, a_0, r_1, s_1, a_1, r_2, s_2, a_2, r_3, s_3, \dots$ be a trajectory and $n \in \mathbb{N} \setminus \{0\}$.
An *n-step return* of τ from time step t is the quantity

$$G_{t:t+n}(\tau) = r_{t+1} + \gamma \cdot r_{t+2} + \gamma^2 \cdot r_{t+3} + \cdots \gamma^{n-1} r_{t+n} + \gamma^n \cdot V(s_{t+n}).$$

We also define a Q-estimate-based version:

$$G_{t:t+n}(\tau) = r_{t+1} + \gamma \cdot r_{t+2} + \gamma^2 \cdot r_{t+3} + \cdots \gamma^{n-1} r_{t+n} + \gamma^n \cdot Q(s_{t+n}, a_{t+n}).$$

(Which of the two is used will be clear from the context.)

n-step TD policy evaluation

Similar to TD(0), but using *n*-step return targets:

given a trajectory $\tau = s_0, a_0, r_1, s_1, a_1, r_2, s_2, a_2, r_3, s_3, \ldots$, for each $0 \leq t < T$ we perform an update

$$V(s_t) \leftarrow V(s_t) + \alpha[G_{t:t+n}(\tau) - V(s_t)].$$

Note 1: for $n = 1$ we get exactly TD(0).

Note 2: for $n > 1$, we cannot update $V(s_t)$ directly at step $t + 1$. We need to obtain $r_{t+1}, \ldots, r_{t+n}, s_{t+n}$ first, i.e. we can perform the update after step $t + n$.

Note 3: if $t + n > T$, we truncate the sum in $G_{t:t+n}$ at r_T, i.e. in such a case $G_{t:t+n} = G_t$.

n-step TD policy evaluation: pseudocode

n-step TD for estimating $V \approx v_\pi$

Input: a policy π
Algorithm parameters: step size $\alpha \in (0, 1]$, a positive integer n
Initialize $V(s)$ arbitrarily, for all $s \in \mathcal{S}$
All store and access operations (for S_t and R_t) can take their index mod $n + 1$

Loop for each episode:
  Initialize and store $S_0 \neq$ terminal
  $T \leftarrow \infty$
  Loop for $t = 0, 1, 2, \ldots$:
  | If $t < T$, then:
  | Take an action according to $\pi(\cdot|S_t)$
  | Observe and store the next reward as R_{t+1} and the next state as S_{t+1}
  | If S_{t+1} is terminal, then $T \leftarrow t + 1$
  | $\tau \leftarrow t - n + 1$ (τ is the time whose state's estimate is being updated)
  | If $\tau \geq 0$:
  | $G \leftarrow \sum_{i=\tau+1}^{\min(\tau+n,T)} \gamma^{i-\tau-1} R_i$
  | If $\tau + n < T$, then: $G \leftarrow G + \gamma^n V(S_{\tau+n})$ $(G_{\tau:\tau+n})$
  | $V(S_\tau) \leftarrow V(S_\tau) + \alpha \left[G - V(S_\tau)\right]$
  Until $\tau = T - 1$

source: Sutton&Barto, p.144

n-step TD policy evaluation: performance

19-state symmetric random walk:

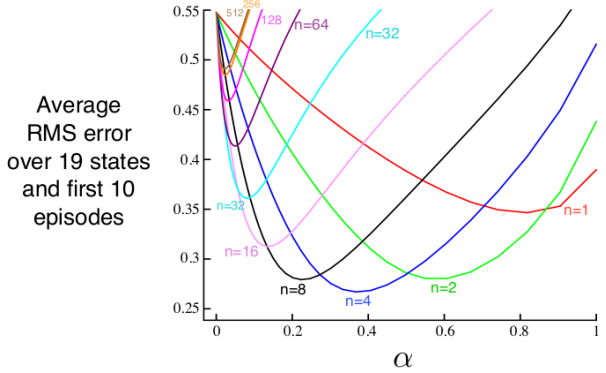


Figure 7.2: Performance of n-step TD methods as a function of α, for various values of n, on a 19-state random walk task (Example 7.1). ■

source: Sutton&Barto, p.145

n-step SARSA (on-policy control)

Uses Q-value-bootstrapped n-step returns.

For a sampled trajectory $\tau = s_0, a_0, r_1, s_1, a_1, r_2, s_2, a_2, r_3, s_3, \ldots$ and for all time steps $0 \leq t < T$ we perform an update

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha[G_{t:t+n} - Q(s_t, a_t)].$$

We sample trajectories according to a policy π that is ε-greedy w.r.t. current Q-estimates:

$$\pi(a|s) = \begin{cases} 1 - \varepsilon + \frac{\varepsilon}{|\mathcal{A}(s)|} & \text{if } a = \arg\max_{a' \in \mathcal{A}(s)} Q(s, a') \text{ (ties broken in principled way)} \\ \frac{\varepsilon}{|\mathcal{A}(s)|} & \text{otherwise.} \end{cases}$$

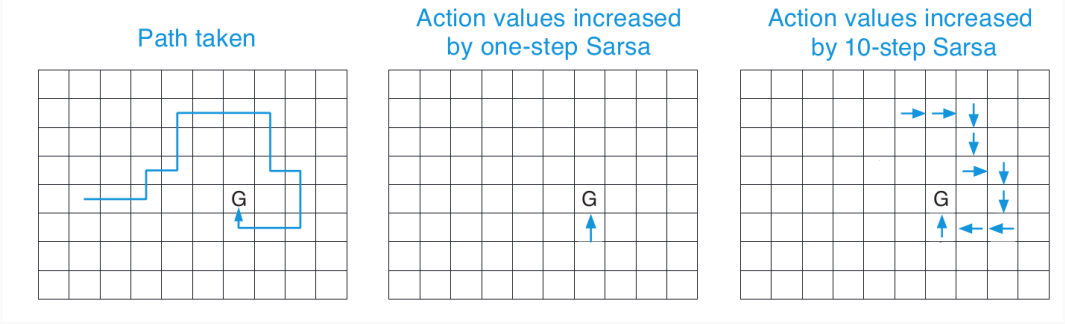
π is redefined in this way after each episode

n-step SARSA (pseudocode)

Loop for each episode:
  Initialize and store $S_0 \neq$ terminal
  Select and store an action $A_0 \sim \pi(\cdot|S_0)$
  $T \leftarrow \infty$
  Loop for $t = 0, 1, 2, \ldots$:
  | If $t < T$, then:
  | Take action A_t
  | Observe and store the next reward as R_{t+1} and the next state as S_{t+1}
  | If S_{t+1} is terminal, then:
  | $T \leftarrow t + 1$
  | else:
  | Select and store an action $A_{t+1} \sim \pi(\cdot|S_{t+1})$
  | $\tau \leftarrow t - n + 1$ (τ is the time whose estimate is being updated)
  | If $\tau \geq 0$:
  | $G \leftarrow \sum_{i=\tau+1}^{\min(\tau+n,T)} \gamma^{i-\tau-1} R_i$
  | If $\tau + n < T$, then $G \leftarrow G + \gamma^n Q(S_{\tau+n}, A_{\tau+n})$ $(G_{\tau:\tau+n})$
  | $Q(S_\tau, A_\tau) \leftarrow Q(S_\tau, A_\tau) + \alpha \left[G - Q(S_\tau, A_\tau)\right]$
  | If π is being learned, then ensure that $\pi(\cdot|S_\tau)$ is ε-greedy wrt Q
  Until $\tau = T - 1$

source: Sutton&Barto, p. 147

n-step SARSA (speed of signal propagation)



source: Sutton&Barto, p. 147

n-step Q-learning

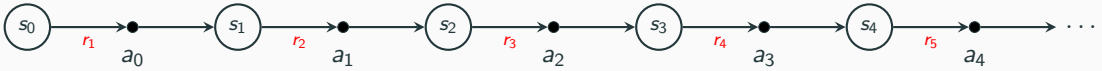
The Q-learning update for a trajectory $\tau = s_0, a_0, r_1, s_1, a_1, r_2, s_2, a_2, r_3, s_3, \ldots$ and time step t:

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha \cdot [r_{t+1} + \gamma \cdot \max_{a \in \mathcal{A}(s_{t+1})} Q(s_{t+1}, a) - Q(s_t, a_t)].$$

Naive extension to n-step returns

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha \cdot [r_{t+1} + \gamma r_{t+2} + \cdots + \gamma^{t+n-1} \cdot r_{t+n} + \gamma^{t+n} \cdot \max_{a \in \mathcal{A}(s_{t+1})} Q(s_{t+n+1}, a) - Q(s_t, a_t)]$$

does not really correspond to Q-learning, since some of the actions $a_{t+1}, \ldots, a_{t+n}$ might not be Q-greedy (the behavior policy is ε-greedy, so some actions might be exploratory). Hence, we are no longer pushing Q towards the Q-value of an optimal policy.



n-step Q-learning: correct

Idea: apply the Q-learning bootstrap at the first occurrence of a non-Q-greedy action.

I.e., for each episode:

- make π an ε-Q-greedy policy
- sample a trajectory $\tau = s_0, a_0, r_1, s_1, a_1, r_2, s_2, a_2, r_3, s_3, \ldots$ from π
- for each time step $0 \leq t < T$:
  - identify the smallest $n' \in \{t+1, t+2, \ldots, t+n\}$ such that $a_{n'}$ is not a Q-greedy action
  - perform the update

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha \cdot [r_{t+1} + \gamma r_{t+2} + \cdots + \gamma^{n'-1} r_{n'} + \gamma^{n'} \cdot \max_{a \in \mathcal{A}(s_{n'})} Q(s_{n'}, a) - Q(s_t, a_t)]$$

(if $n' > T$, do the standard MC update).

λ-returns: idea

By varying n, the n-step returns provide a nice tradeoff between bias and variance (and update speed). But the choice of optimal n is mostly a guesswork.

Idea: find a notion of return which combines n-step returns for multiple n's. E.g., a suitable convex combination of individual n-step returns. This leads to the notion of λ-returns.

We will focus only on policy evaluation, though λ-returns can be used also in control.

λ-returns: definition

Recall: $G_{t:t+n}$ is the n-step return from timestep t.

Definition 47: λ-return

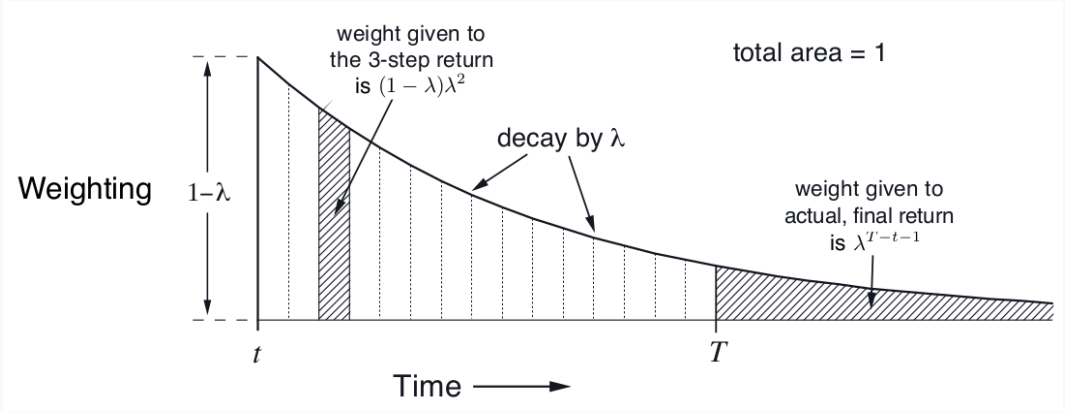
Let $\lambda \in [0,1]$. A λ-return from timestep t is the random variable

$$G_t^\lambda = (1-\lambda)\sum_{n=1}^{\infty} \lambda^{n-1} G_{t:t+n}.$$

Note that due to truncation at $t+n \geq T$, the λ-return can be more explicitly written as

$$G_t^\lambda = (1-\lambda)\sum_{n=1}^{T-t-1} \lambda^{n-1} \cdot G_{t:t+n} + \lambda^{T-t-1} \cdot G_t.$$

λ-return as discounting of n-step returns



source: Sutton&Barto, p. 290

(Forward-view) TD(λ)

Like TD(0), but uses λ-returns.

Given a trajectory $\tau = s_0, a_0, r_1, s_1, a_1, r_2, s_2, a_2, r_3, s_3, \ldots$ sampled from the evaluated policy π, we perform, for each time step t an update:

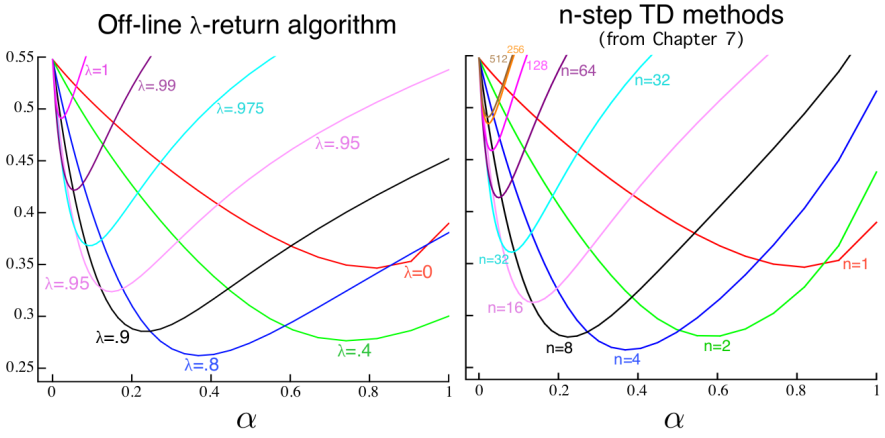
$$V(s_t) \leftarrow V(s_t) + \alpha(G_t^\lambda(\tau) - V(s_t)).$$

Note:

- for $\lambda = 0$, this is exactly the TD(0) update,
- for $\lambda = 1$, this is exactly the MC update,
- $G_t^\lambda(\tau)$ depends on the whole suffix of $\tau_{t..}$, hence the update can be only performed at the end of the episode. (We will show a workaround later.)

TD(λ) vs. n-step TD on 19-state random walk

RMS error at the end of the episode over the first 10 episodes



source: Sutton&Barto, p. 291

Forward vs. backward view TD(λ)

Backward-view TD(λ) = an algorithm performing roughly the same updates as Forward-view TD(λ) in an online fashion ($V(s_t)$ can be updated by time $t+1$).

Implemented using eligibility traces: state-wise signals that indicate how much is the current state eligible for an update (sort of state-wise modulation of the learning rate).

We are more keen to update states that:

- appear often along the trajectory (frequency heuristic)
- were visited in the recent past (recency heuristic)

Accumulating eligibility trace

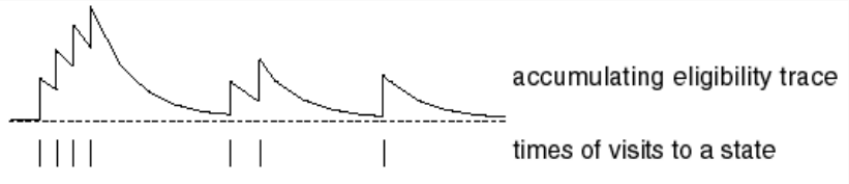
Definition 48: (Accumulating) eligibility trace

For a trajectory $\tau = s_0, a_0, r_1, s_1, a_1, r_2, s_2, a_2, r_3, s_3, \ldots$, $\lambda \in [0, 1]$, and a state $s \in \mathcal{S}$, an (accumulating) eligibility trace is a sequence of values $E_0(s), E_1(s), E_2(s), \ldots$ defined inductively as follows:

$$E_0(s) = 0$$

$$\text{and for } t > 0 \qquad E_t(s) = \gamma \cdot \lambda \cdot E_{t-1}(s) + \mathbb{I}(S_t(\tau) = s),$$

where $\mathbb{I}(S_t(\tau) = s)$ is the indicator of the t-th state of τ being s, i.e. $\mathbb{I}(S_t(\tau) = s) = 1$ if $s_t = s$ and $\mathbb{I}(S_t(\tau) = s) = 0$ otherwise.



source: slides of D. Silver (Model-free prediction)

Backward-view TD(λ): idea

- $E_t(s)$ denotes how much is s eligible for an update after playing t-th action along the run (i.e., action a_{t-1}).
- In time step t, all states with non-zero eligibility signal will have their estimates updated in proportion to the learning rate and the strength of the eligibility signal.
- The update target is the standard TD(0) target for time t. I.e., for each timestep t and each state q, we perform the update

$$V(q) \leftarrow V(q) + \alpha \cdot E_t(q) \cdot \left[r_{t+1} + \gamma \cdot V(s_{t+1}) - V(s_t)\right].$$

Backward-view TD(λ): pseudocode

Input: policy π to evaluate

Output: Estimate V of v^π

initialize V arbitrarily

repeat

- $s \leftarrow$ sample uniformly (ES) or according to init. distr.
- initialize E to be uniformly zero
- **while** s *not terminal* **do**
  - $a \leftarrow$ sample from $\pi(s)$
  - $s' \leftarrow$ sample from $p(s, a)$
  - $r \leftarrow r(s, a, s')$
  - **foreach** $q \in \mathcal{S}$ *(Only q's visited so far)* **do**
    - $E(q) = \gamma \cdot \lambda \cdot E(q) + \mathbb{I}(q = s)$
    - $V(q) \leftarrow V(q) + \alpha \cdot E(q) \cdot \big[r + \gamma \cdot V(s') - V(s)\big]$
  - $s \leftarrow s'$

until *timeout*

Forward vs. backward view

If $\lambda = 0$, then $E_t(q) = \mathbb{I}(S_t(\tau) = q)$, i.e. the backward-view update at time point t is

$$V(s_t) \leftarrow V(s_t) + \alpha \cdot \big[r + \gamma \cdot V(s_{t+1}) - V(s_t)\big],$$

while for all states other than s_t, no update is performed. I.e., backward TD(0) is exactly the same thing as forward TD(0).

For general λ the correspondence is more subtle:

Theorem 49: Forward-backward view correspondence

Assume that in the backward view, all the updates along the trajectory are performed offline, i.e. only after the end of the episode, and in a batch, i.e. concurrently, using the pre-episode estimates in right-hand sides.
Then, for any $\lambda \in (0, 1)$, this offline backward TD(λ) performs the same updates as forward TD(λ).

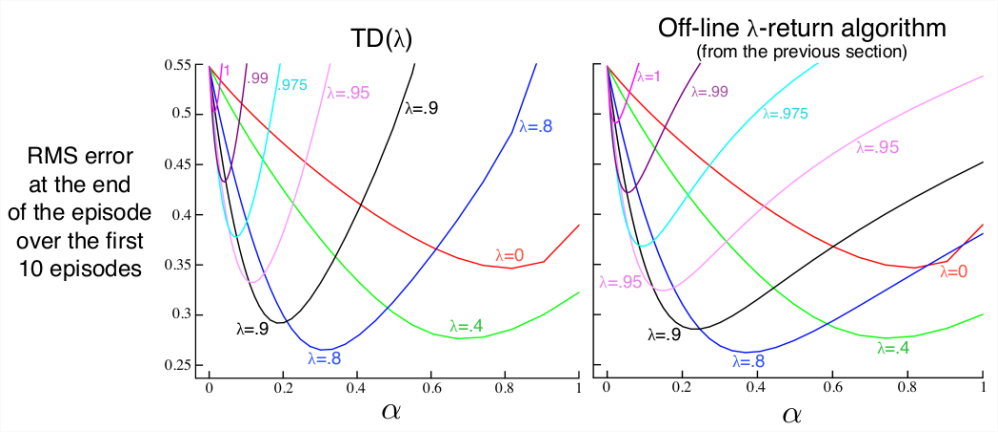
Offline backward and forward view (source: D. Silver slides)

Given the batch nature of updates, it suffices to show that the forward update target at time t equals the sum of all updates "triggered" by a visit to state s_t.

$$
\begin{aligned}
G_t^\lambda - V(S_t) &= -V(S_t) &&+ (1-\lambda)\lambda^0 \left(R_{t+1} + \gamma V(S_{t+1})\right) \\
&&&+ (1-\lambda)\lambda^1 \left(R_{t+1} + \gamma R_{t+2} + \gamma^2 V(S_{t+2})\right) \\
&&&+ (1-\lambda)\lambda^2 \left(R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \gamma^3 V(S_{t+3})\right) \\
&&&+ \ldots \\
&= -V(S_t) &&+ (\gamma\lambda)^0 \left(R_{t+1} + \gamma V(S_{t+1}) - \gamma\lambda V(S_{t+1})\right) \\
&&&+ (\gamma\lambda)^1 \left(R_{t+2} + \gamma V(S_{t+2}) - \gamma\lambda V(S_{t+2})\right) \\
&&&+ (\gamma\lambda)^2 \left(R_{t+3} + \gamma V(S_{t+3}) - \gamma\lambda V(S_{t+3})\right) \\
&&&+ \ldots \\
&= &&(\gamma\lambda)^0 \left(R_{t+1} + \gamma V(S_{t+1}) - V(S_t)\right) \\
&&&+ (\gamma\lambda)^1 \left(R_{t+2} + \gamma V(S_{t+2}) - V(S_{t+1})\right) \\
&&&+ (\gamma\lambda)^2 \left(R_{t+3} + \gamma V(S_{t+3}) - V(S_{t+2})\right) \\
&&&+ \ldots \\
&= \delta_t + \gamma\lambda\delta_{t+1} + (\gamma\lambda)^2\delta_{t+2} + \ldots
\end{aligned}
$$

Online vs. offline backward view on 19-state RW

In practice, we want to use the online backward algorithm, which only approximates the forward view. Nevertheless, it performs acceptably:



source: Sutton&Barto, p. 295, LEFT: online backward TD(λ), RIGHT: offline forward TD(λ)

Concluding remarks on λ-returns

- There are other types of eligibility traces (replacing, dutch, ...), yielding different algorithms.
- Eligibility traces neatly generalize to deep learning, where they are not state-wise but parameter-wise signals; optional reading: Sutton&Barto, Sec. 12.1-12.2
- λ-returns and eligibility traces can be generalized to control setting SARSA(λ), Q(λ); optional reading: Sutton&Barto, Sec. 12.7-12.10.
- There is a true online backward TD(λ) version. Here, true online=having perfect equivalence with the forward view. However, the equivalence is w.r.t. a more complex notion of λ-return (truncated λ-return) and uses more complex version of eligibility traces than presented here. Outperforms both forward and backward algorithms presented here. Optional reading: van Seijen, Sutton: *True Online TD(λ)*. In *Proceedings of ICML'14*.

First Steps Towards Deep RL: Value-Based On-Policy Methods

Working with huge MDPs

E.g. original Atari games have 160x192 resolution with 128 colors: observable state space of size $2^{7 \cdot 160 \cdot 192} = 2^{215040}$ (though only a fraction reachable and resolution typically scaled down in benchmarks – however, state typically encompass last 3 frames so as to provide some info on movement).

State space can be even continuous (position, velocity,...).

Most states will not be seen - we need the ability to generalize from experience to unseen/rarely seen states.

Huge MDP representation

From now on, states of the MDP will be represented by vectors from $\mathbb{R}^n$. The vectorized representation is chosen in a domain-specific manner, e.g.:

- Atari = one component per pixel per frame
- continuous navigation = agent coordinates, velocity, etc.
- small discrete MDPs can be represented by one-hot encoding

For simplicity, we will still assume that the action space is discrete, and reasonably small, though many algorithms can be adapted for continuous actions (acceleration, etc).

Function approximators

The value functions have types:

$$v^\pi, v^* : \mathbb{R}^n \to \mathbb{R} \qquad q^\pi, q^* : \mathbb{R}^n \times \mathcal{A} \to \mathbb{R}.$$

In RL, we need to approximate these functions.

Definition 50

A function approximator (FA) for functions of type $X \to Y$ is a class of functions $f \subseteq Y^X$ parameterized by a some set of parameter vectors $\Theta \subseteq \mathbb{R}^n$.

Each concrete parameter vector $\theta \in \Theta$ defines a concrete function $f_\theta \in f$, i.e. $f = \{f_\theta \mid \theta \in \Theta\}$.

For FA f, we often write $f_\theta(x) = f(x, \theta)$ to stress the fact that the output of f_θ depends on both the input x and on θ. Hence, FA for type $X \to y$ can be itself seen as a function of type $X \times \Theta \to Y$.

Function approximators in RL

Our algorithms will use mainly these types of function approximators:

- $V\colon \mathbb{R}^n \times \Theta \to \mathbb{R}$ to approximate v^π or v^θ
- $Q\colon (\mathbb{R}^n \times \mathcal{A}) \times \Theta \to \mathbb{R}$ to approximate q^π or q^θ

The typical task is to find $\theta \in \Theta$ such that $V_\theta = V(\cdot, \theta)$ is a "good" approximation for v^π or v^θ, and similarly for Q_θ.

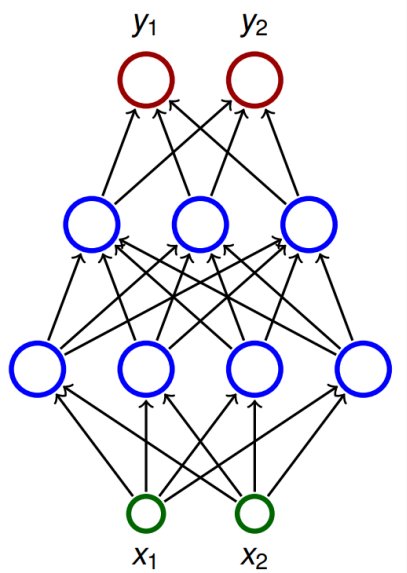
The parametrization Θ will depend on the concrete form of function approximator used.

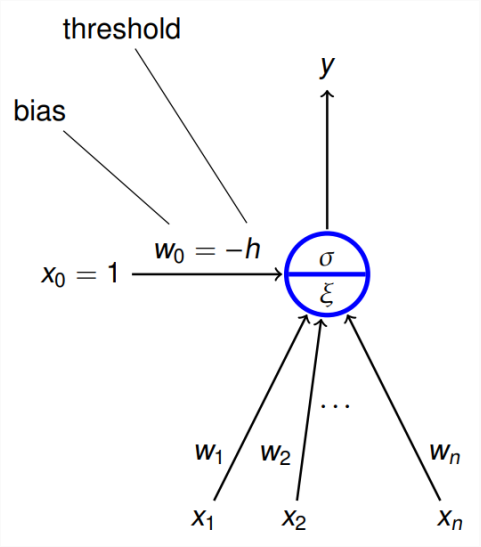
Forms of function approximators

- tabular
  - $\theta =$ the vector containing the contents of the table
- linear
  - $\Theta = \mathcal{S} = \mathbb{R}^n$ and e.g. $V_\theta(s) = \theta^\top \cdot s$
- neural nets
  - $\theta =$ NN weights and biases
- decision trees
- ...

We require the approximators to be differentiable and to admit a training method suitable for non-stationary data.

Neural nets (source: slides by T. Brázdil)





Policy evaluation with FAs

Task: given a policy π and FA $V\colon \mathbb{R}^n \times \Theta \to \mathbb{R}$, find θ s.t. V_θ is "close" to v^π.

"Closeness" can be expressed using various loss functions. Typically, we want to minimize the mean squared error (MSE):

$$MSE(v^\pi, V_\theta) = \frac{1}{2}\mathbb{E}_{s\sim\mu}\left[(v^\pi(s) - V_\theta(s))^2\right] = \frac{1}{2}\sum_{s\in\mathcal{S}} \mu(s) \cdot \left[(v^\pi(s) - V_\theta(s))^2\right],$$

where μ is some distribution over states expressing how much do we care about errors in particular states.

A local minimum of MSE can be found gradient descent: making successive step in the direction opposite to the gradient of MSE.

Recall: gradients

Definition 51

Given a scalar function $f(x_1, \ldots, x_n, \theta_1, \ldots, \theta_m)$: $\mathbb{R}^n \times \Theta \to \mathbb{R}$ (where $\Theta \subseteq \mathbb{R}^m$), the gradient of f w.r.t. parameters $\theta = (\theta_1, \ldots, \theta_m)$ is the vector function

$$\nabla_\theta f = \left(\frac{\partial f}{\partial \theta_1}, \ldots, \frac{\partial f}{\partial \theta_m}\right) \text{ of type } \mathbb{R}^n \times \Theta \to \mathbb{R}^m$$

When f is a function approximator defined by a neural net, the value of the gradient $\nabla_\theta f(x, \theta)$ at a given point $(x, \theta) = (x_1, \ldots, x_n, \theta_1, \ldots, \theta_m)$ can be computed by backpropagation (under some usual conditions like smoothness, etc.).

Gradient descent for policy evaluation

To (locally) minimize $MSE(v^\pi, V_\theta)$, it suffices to perform (sufficiently small) steps in the negative direction of the current gradient, i.e., repeatedly perform updates:

$$\begin{aligned}\theta &\leftarrow \theta - \alpha \cdot \nabla_\theta MSE(v^\pi, V_\theta) = \theta - \alpha \cdot \nabla_\theta \frac{1}{2} \cdot \mathbb{E}_{s\sim\mu}\left[\left(v^\pi(s) - V_\theta(s)\right)^2\right] \\ &= \theta - \frac{\alpha}{2} \cdot \mathbb{E}_{s\sim\mu}\left[\nabla_\theta \left(v^\pi(s) - V_\theta(s)\right)^2\right] \\ &= \theta + \alpha \cdot \mathbb{E}_{s\sim\mu}\left[\left(v^\pi(s) - V_\theta(s)\right) \cdot \nabla_\theta V_\theta(s)\right]\end{aligned}$$

The expected value above is typically impossible to evaluate in practice. Instead we estimate it by samples $\Rightarrow$ stochastic gradient descent.

We typically take $\mu(s)$ representing the overall fraction of time spent in s when behaving according to μ. Hence, $\mathbb{E}_{s\sim\mu}$ can be estimated by sampling a trajectory from μ and performing the update for each s on the trajectory in an every-visit fashion.

Stochastic gradient policy evaluation + MC instantiation

We keep sampling trajectories τ from π:

For each timestep t we perform the update of parameters

$$\theta \leftarrow \theta + \alpha \cdot \left[\left(v^{\pi}(s_t) - V_\theta(s_t) \right) \cdot \nabla_\theta V_\theta(s_t) \right].$$

Problem: in policy evaluation setting, we do not know $v^{\pi}(s_t)$. Hence, we estimate it using RL targets.

The simplest is the Monte Carlo target: estimate s_t by the discounted return of the sampled trajectory from s_t, i.e. perform updates of the form

$$\theta \leftarrow \theta + \alpha \cdot \left[\left(G_t(s_t) - V_\theta(s_t) \right) \cdot \nabla_\theta V_\theta(s_t) \right].$$

Gradient Monte Carlo policy evaluation: pseudocode

Algorithm 3: Gradient MC evaluation

Input: Policy π, FA $V : \mathcal{S} \times \Theta \to \mathbb{R}$, step size α

Output: Approximation V_θ of v^π

initialize θ arbitrarily;

repeat

  sample trajectory $\tau = s_0, a_0, r_1, s_1, a_1, r_2, s_2, a_2, r_3, s_3, \ldots$ from π;

  foreach $t \in \{0, \ldots, T-1\}$ **do**

    $\theta \leftarrow \theta + \alpha \cdot [G_t(\tau) - V(s_t, \theta)] \cdot \nabla_\theta V(s_t, \theta)$

until *timeout*;

Semi-gradient TD(0)

In the gradient update formula

$$\theta \leftarrow \theta + \alpha \cdot \big[\big(v^{\pi}(s_t) - V_{\theta}(s_t)\big) \cdot \nabla_{\theta} V_{\theta}(s_t)\big].$$

we can also estimate $v^{\pi}(s_t)$ with the TD(0) target:

$$\theta \leftarrow \theta + \alpha \cdot \big[\big(r_{t+1} + \gamma \cdot V_{\theta}(s_{t+1}) - V_{\theta}(s_t)\big) \cdot \nabla_{\theta} V_{\theta}(s_t)\big].$$

This yield the semi-gradient TD(0) policy evaluation algorithm.

Why semi-gradient?

Gradient vs. semi-gradient TD(0)

Recall that our ultimate goal is to minimize

$$MSE(v^\pi, V_\theta) = \frac{1}{2}\mathbb{E}_{s\sim\mu}\big[(v^\pi(s) - V_\theta(s))^2\big].$$

The gradient of this loss is

$$\nabla_\theta \frac{1}{2}\mathbb{E}_{s\sim\mu}\big[(v^\pi(s) - V_\theta(s))^2\big] = \frac{1}{2}\mathbb{E}_{s\sim\mu}\big[\nabla_\theta(v^\pi(s) - V_\theta(s))^2\big],$$

Estimation with sample trajectory $\tau = s_0, a_0, r_1, s_1, a_1, r_2, s_2, a_2, r_3, s_3, \ldots$ and substituting $v^\pi(s)$ with the TD(0) target would yield

$$\nabla_\theta MSE(v^\pi, V_\theta) \approx \frac{1}{2}\nabla_\theta\big(r_{t+1}+\gamma V_\theta(s_{t+1})-V_\theta(s_t)\big)^2 = \big(r_{t+1}+\gamma V_\theta(s_{t+1})-V_\theta(s_t)\big)\cdot\big(\gamma\nabla_\theta V_\theta(s_{t+1})-\nabla_\theta V_\theta(s_t)\big)$$

different update then semi-gradient TD(0)! However, this full gradient:

- is more expensive to compute (2 backpropagations per update);
- does not really express TD(0) idea (the update target is not fixed).

Semi-gradient TD(0): pseudocode

Algorithm 4: Semi-gradient TD(0) evaluation

Input: Policy π, FA $V : \mathcal{S} \times \Theta \to \mathbb{R}$, step size α

Output: Approximation V_θ of v^π

initialize θ arbitrarily;

repeat

  $s \leftarrow$ initial state;

  $a \sim \pi(s)$;

  while *s not terminal* **do**

    $s' \sim p(s, a)$;

    $r \leftarrow r(s, a, s')$;

    $a' \sim \pi(s')$;

    $\theta \leftarrow \theta + \alpha \cdot [r + \gamma \cdot V(s', \theta) - V(s, \theta)] \cdot \nabla_\theta V(s, \theta)$;

    $s \leftarrow s'$; $a \leftarrow a'$

until *timeout*;

On-policy control with function approximation

Semi-gradient SARSA uses the same idea as TD(0), but with Q-approximator, i.e. $Q\colon \mathbb{R}^n \times \mathcal{A} \times \Theta \to \mathbb{R}$.

Behavior policy = e.g. ε-greedy with respect to the current Q. For a sampled trajectory $\tau =$

we perform, in each timestep t, an update

$$\theta \leftarrow \theta + \alpha \cdot \big[\big(r_{t+1} + \gamma \cdot Q_\theta(s_{t+1}, a_{t+1}) - Q_\theta(s_t, a_t)\big) \cdot \nabla_\theta Q_\theta(s_t, a_t)\big].$$

Semi-gradient SARSA: pseudocode

Algorithm 5: Semi-gradient SARSA

Input: FA $Q\colon \mathcal{S} \times \mathcal{A} \times \Theta \to \mathbb{R}$, step size α

Output: Approximation Q_θ of q^*

initialize θ arbitrarily;

repeat

  $s \leftarrow$ initial state;

  $\pi \leftarrow$ policy ε-greedy w.r.t. Q_θ;

  $a \sim \pi(s)$;

  while s *not terminal* **do**

    $s' \sim p(s, a)$;

    $r \leftarrow r(s, a, s')$;

    $a' \sim \pi(s')$;

    $\theta \leftarrow \theta + \alpha \cdot [r + \gamma \cdot Q_\theta(s', a') - Q_\theta(s, a)] \cdot \nabla_\theta Q_\theta(s, a)$;

    $s \leftarrow s'$; $a \leftarrow a'$

until *timeout*;

Representing actions in DNNs

How to represent actions in the (say, DNN) function approximator Q is largely a domain-dependent engineering choice.

If the set of actions $\mathcal{A} = \{a^1, \ldots, a^k\}$ is discrete and reasonably small, we can consider a net which inputs a state (i.e., n input neurons when $\mathcal{S} = \mathbb{R}^n$) and outputs an $|\mathcal{A}|$-dimensional vector (i.e., one output neuron per action), so that the output of the i-th neuron on input s is interpreted as $Q(s, a^i)$.

I.e., in such a case we consider Q to be function of type $Q \colon \mathcal{S} \times \Theta \to \mathbb{R}^{|\mathcal{A}|}$.

On-policy semi-gradient methods: concluding remarks

The presented algorithms can be instantiated also with other types of returns, such as:

- n-step returns
  - n-step SARSA update:
    $\theta \leftarrow \theta + \alpha \cdot \big[\big(\underbrace{r_{t+1} + \gamma \cdot r_{t+2} + \cdots + \gamma^{n-1} r_{t+n} + \gamma^n \cdot Q_\theta(s_{t+n}, a_{t+n})}_{G_{t:t+n,\theta}} - Q_\theta(s_t, a_t)\big) \cdot \nabla_\theta Q_\theta(s_t, a_t)\big]$
- forward-view λ-returns
  - SARSA(λ) update: $\theta \leftarrow \theta + \alpha \cdot \big[\big((1-\lambda) \sum_{n=1}^{\infty} \lambda^{n-1} G_{t:t+n,\theta} - Q_\theta(s_t, a_t)\big) \cdot \nabla_\theta Q_\theta(s_t, a_t)\big]$
- backward-view λ-returns (optional reading: Sutton&Barto, sections 12.2 and 12.7)

Value-Based Off-Policy Control with Approximators: DQNs and Friends

Off-policy methods with function approximation

...are tricky to get right, already in the case of policy evaluation. The training can become very unstable.

For on-policy (semi)-gradient methods, one can typically prove convergence to correct/optimal values at least in the case of linear function approximation (though not in the more general case of NN approximators).

Off-policy semi-gradient methods, such as:

- TD with importance sampling (not covered here), or
- Q-learning with function approximators (will be covered a bit later),

can diverge already with linear function approximators.

Divergence examples (high-level)

- Baird's counterexample: semi-gradient TD with importance sampling can diverge in presence of linear FAs
- Moreover, the divergence is not due to the instability of (semi)-gradient descent. Tsitsiklis and Van Roy's counterexample shows divergence even in the case where each update completely replaces the current θ with the optimal θ^* which minimizes the MSE between V_θ and the TD(0) update target. The problem lies in the off-policy distribution of updates.
- Counterexamples explained in optional reading: Sutton&Barto, Sec. 11.2.

Deadly triad

Identified by Sutton&Barto: risk of training instability and divergence steeply rises when combining:

- function approximation,
- bootstrapping, and
- off-policy training.

But often we want to do just that. :)

Practical solution: Happily do the deadly triad, but use insights from supervised learning to develop additional techniques that help stabilize the training.

Deep Q-Networks (DQN)

2013 arXiv tech. report, there is also follow-up 2015 Nature paper

Playing Atari with Deep Reinforcement Learning

Volodymyr Mnih **Koray Kavukcuoglu** **David Silver** **Alex Graves** **Ioannis Antonoglou**

Daan Wierstra **Martin Riedmiller**

DeepMind Technologies

{vlad,koray,david,alex.graves,ioannis,daan,martin.riedmiller} @ deepmind.com

Abstract

We present the first deep learning model to successfully learn control policies directly from high-dimensional sensory input using reinforcement learning. The model is a convolutional neural network, trained with a variant of Q-learning, whose input is raw pixels and whose output is a value function estimating future rewards. We apply our method to seven Atari 2600 games from the Arcade Learning Environment, with no adjustment of the architecture or learning algorithm. We find that it outperforms all previous approaches on six of the games and surpasses a human expert on three of them.

Q-learning with function approximators

Same semi-gradient idea as in TD(0), SARSA: adjust θ to bring $Q_\theta(s, a)$ closer to the fixed Q-learning update target.

I.e., for a sampled trajectory

and its timestep t, the update is

$$\theta \leftarrow \theta + \alpha \cdot \left[\left(r_{t+1} + \gamma \cdot \max_{a \in \mathcal{A}(s_{t+1})} Q_\theta(s_{t+1}, a) - Q_\theta(s_t, a_t)\right) \cdot \nabla_\theta Q_\theta(s_t, a_t)\right].$$

But performing the updates based solely on the current step would be susceptible to instability due to the presence of the deadly triad.

Deep Q-learning challenges

From Mnih et al. *Playing Atari with Deep Reinforcement Learning*:

> However reinforcement learning presents several challenges from a deep learning perspective. Firstly, most successful deep learning applications to date have required large amounts of hand-labelled training data. RL algorithms, on the other hand, must be able to learn from a scalar reward signal that is frequently sparse, noisy and delayed. The delay between actions and resulting rewards, which can be thousands of timesteps long, seems particularly daunting when compared to the direct association between inputs and targets found in supervised learning. Another issue is that most deep learning algorithms assume the data samples to be independent, while in reinforcement learning one typically encounters sequences of highly correlated states. Furthermore, in RL the data distribution changes as the algorithm learns new behaviours, which can be problematic for deep learning methods that assume a fixed underlying distribution.

Experience replay

Originated in the work of Long-Ji Lin, e.g.: *Reinforcement Learning for Robots Using Neural Networks*, dissertation, 1993.

Definition 52: Experience

An experience is a 4-tuple $(s, a, r, s') \in \mathcal{S} \times \mathcal{A} \times \mathcal{S} \times \mathbb{R}$ interpreted as ("state", "action played in it", "reward obtained", "next state observed").

- DQN does not perform update based only on the current step. Instead, for each sampled trajectory $\tau = s_0, a_0, r_1, s_1, a_1, r_2, s_2, a_2, r_3, s_3, \ldots$ and each timestep t it:
  - first stores the one-step experience $(s_t, a_t, r_{t+1}, s_{t+1})$ in a data structure $\mathcal{B}$ called replay buffer;
  - then, sample a random minibatch of experiences $B \subseteq \mathcal{B}$ of a given minibatch size *Bsize*
  - perform a minibatch-gradient-descent update w.r.t. B: compute the gradient of the Q-learning loss for each $e \in B$ and then update θ in the direction of an average gradient over the whole B.

Minibatch update

Fix a minibatch B.

For each $e = (s, a, r, s') \in B$ we compute the gradient of the Q-learning loss $\nabla_\theta \mathcal{L}(\theta, e)$ at point e:

$$\nabla_\theta \mathcal{L}(\theta, e) = \nabla_\theta \frac{1}{2} \Big(\underbrace{r + \gamma \cdot \max_{b \in \mathcal{A}(s')} Q_\theta(s', b)}_{\text{fixed target}} - Q_\theta(s, a) \Big)^2$$

$$= \Big[\underbrace{\Big(r + \gamma \cdot \max_{b \in \mathcal{A}(s')} Q_\theta(s', b) \Big)}_{=0 \text{ if } s \text{ terminal}} - Q_\theta(s, a) \Big) \cdot \nabla_\theta Q_\theta(s, a) \Big]$$

We then perform an update in the direction of average gradient:

$$\theta \leftarrow \theta + \alpha \cdot \frac{1}{|B|} \sum_{e \in B} \nabla_\theta \mathcal{L}(\theta, e).$$

Experience replay rationale

- helps decorrelate the DNN training data
- helps to prevent catastrophic forgetting
- improves data efficiency via experience re-use

Why good match for deep Q-learning? Experience replay is by design off-policy since we train on old data, which were sampled from different policy than the current one.

Replay buffer implementation

The replay buffer $\mathcal{B}$ is typically not unbounded, but has a fixed capacity $\mathcal{B}size$. Replacement is eventually needed. If $\mathcal{B}$ is full, the oldest experience if removed ($\mathcal{B}$ = queue).

How to sample the minibatches?

- Original DQN: uniformly from $\mathcal{B}$.
- Alternative: prioritized experience replay: each experience is assigned a priority (several heuristics exist). An experience is sampled into a minibatch with probability proportional to its priority.

DQN: 2013 pseudocode

Algorithm 6: DQN with replay buffer

Input: Black-box MDP $\mathcal{M} = (\mathcal{S}, \mathcal{A}, p, r)$, approximator Q; hyperparam's $\varepsilon, \mathcal{B}size, Bsize, \ldots$

Output: Approximation Q_θ of q^*

initialize θ arbitrarily; initialize empty replay buffer $\mathcal{B}$ of capacity $\mathcal{B}size$;

repeat

  $s \leftarrow$ initial state;

  while *s not terminal* **do**

    $\pi \leftarrow$ policy ε-greedy w.r.t. Q_θ;

    $a \sim \pi(s)$;

    $s' \sim p(s, a)$;

    $r \leftarrow r(s, a, s')$;

    store (s, a, r, s') in $\mathcal{B}$;

    sample a minibatch B of size $Bsize$ from replay buffer $\mathcal{B}$;

    perform the minibatch update $\theta \leftarrow \theta + \alpha \cdot \frac{1}{Bsize} \sum_{e \in B} \nabla_\theta \mathcal{L}(\theta, e)$ (see this slide);

    $s \leftarrow s'$;

until *timeout*;

DQN: 2013 results

	B. Rider	Breakout	Enduro	Pong	Q*bert	Seaquest	S. Invaders
Random	354	1.2	0	−20.4	157	110	179
Sarsa [3]	996	5.2	129	−19	614	665	271
Contingency [4]	1743	6	159	−17	960	723	268
DQN	**4092**	**168**	**470**	**20**	**1952**	**1705**	**581**
Human	7456	31	368	−3	18900	28010	3690
HNeat Best [8]	3616	52	106	19	1800	920	**1720**
HNeat Pixel [8]	1332	4	91	−16	1325	800	1145
DQN Best	**5184**	**225**	**661**	**21**	**4500**	**1740**	1075

Table 1: The upper table compares average total reward for various learning methods by running an ϵ-greedy policy with $\epsilon = 0.05$ for a fixed number of steps. The lower table reports results of the single best performing episode for HNeat and DQN. HNeat produces deterministic policies that always get the same score while DQN used an ϵ-greedy policy with $\epsilon = 0.05$.

Mnih et al. (2013, arXiv)

Target networks: another stabilizing factor in DQNs

Introduced in the reviewed version of DQN paper:

> Mnih et al.: Human-level control through deep reinforcement learning. *Nature*, 518 (2015).

Performing the (minibatch) Q-update looks like supervised learning:

$$\text{change} \quad \theta \quad \text{so that} \quad Q_\theta(s,a) \quad \text{gets closer to the fixed target} \quad r + \gamma \cdot \max_{b \in \mathcal{A}(s')} Q_\theta(s', b),$$

where (s, a, r, s') is the processed experience.

Performing the (minibatch) Q-update looks like supervised learning, but:

$$\text{change} \quad \theta \quad \text{so that} \quad Q_\theta(s,a) \quad \text{gets closer to the fixed target} \quad r + \gamma \cdot \max_{b \in \mathcal{A}(s')} \underbrace{Q_\theta(s', b)}_{\text{the "label" changes with each update!}},$$

Target networks: idea

To stabilize learning, we use two networks: the main network, and the target network. They have the same architecture (denote it by Q), but their weights may differ during the execution of the algorithm.

We denote:

- θ - weights of main network
- $\hat{\theta}$ - weights of target network

Usage:

- The target network is used only to compute TD targets when computing losses:

$$\nabla_\theta \mathcal{L}(\theta, e) = \left(r + \gamma \cdot \max_{b \in \mathcal{A}(s')} Q(s', b, \hat{\theta}) - Q(s, a, \theta)\right) \cdot \nabla_\theta Q(s, a, \theta)$$

- At the start, and also in periodic intervals (but not after each update!) the two networks are synchronized by performing $\hat{\theta} \leftarrow \theta$. Other than this, $\hat{\theta}$ stays fixed, the gradient steps are only used to update θ (i.e., the main network).

DQN: 2015 pseudocode

Algorithm 7: DQN with replay buffer and target network

Input: Black-box MDP $\mathcal{M} = (\mathcal{S}, \mathcal{A}, p, r)$, approximator Q; hyperparam's $\varepsilon, \mathcal{B}size, Bsize, C, \ldots$

Output: Approximation Q_θ of q^*

initialize θ arbitrarily; $\hat{\theta} \leftarrow \theta$; $counter \leftarrow C$;

initialize empty replay buffer $\mathcal{B}$ of capacity $\mathcal{B}size$;

repeat

  $s \leftarrow$ initial state;

  while s *not terminal* **do**

    if $counter = 0$ **then** $\hat{\theta} \leftarrow \theta$; $counter \leftarrow C$ **else** $counter \leftarrow counter - 1$;

    $\pi \leftarrow$ policy ε-greedy w.r.t. Q_θ;

    $a \sim \pi(s)$;

    $s' \sim p(s, a)$;

    $r \leftarrow r(s, a, s')$;

    store (s, a, r, s') in $\mathcal{B}$;

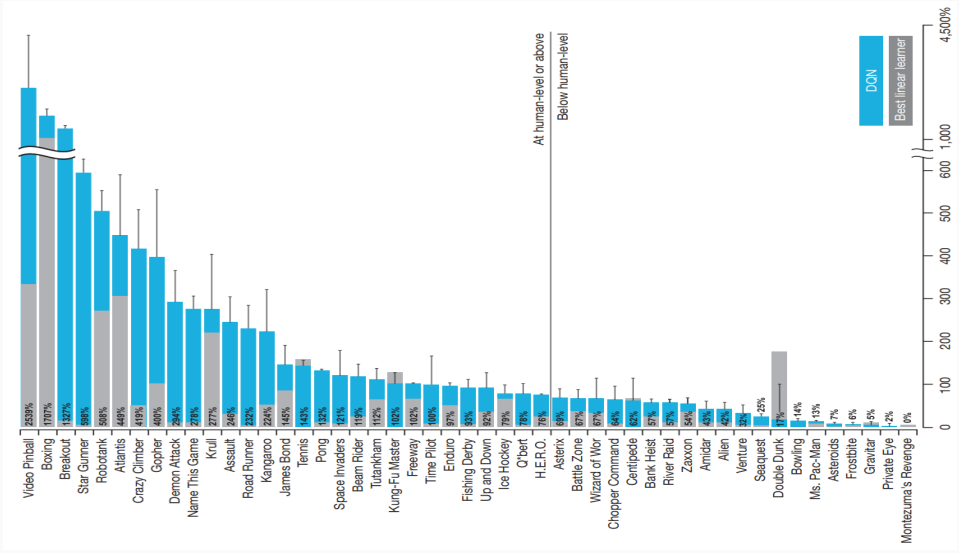
    sample a minibatch B of size $Bsize$ from replay buffer $\mathcal{B}$;

    perform the minibatch update $\theta \leftarrow \theta + \alpha \cdot \frac{1}{Bsize} \sum_{e \in B} \nabla_\theta \mathcal{L}(\theta, e)$, where $\nabla_\theta \mathcal{L}(\theta, e) = \left(r + \gamma \cdot \max_{b \in \mathcal{A}(s')} Q(s', b, \hat{\theta}) - Q(s, a, \theta)\right) \cdot \nabla_\theta Q(s, a, \theta)$;

    $s \leftarrow s'$;

until *timeout*;

DQN: 2015 results



Engineering behind DQN for Atari: partial observability

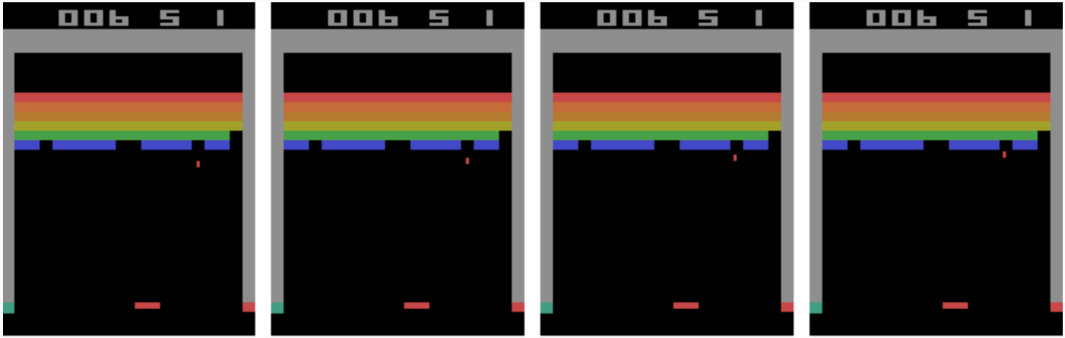
True state = current state (program counter + variable values) of the game program.

We do not see this – only the frames rendered on screen.

Solving partially observable environments requires (per some POMDP theory) making decisions based on the whole history of observations. This is computationally demanding (recurrent NNs...).

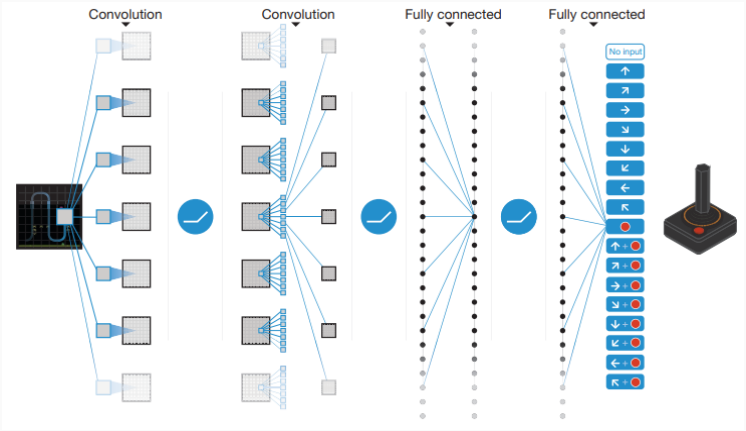
DQN for Atari solves this by feeding the last 4 observed frames into the NN. This is typically enough to deduce the dynamics of the current play.

DQN: dynamics from limited frame history



Engineering behind DQN for Atari: the network

Inputs four 84x84px images, then 3 convolutional layers, then two fully connected layers. all with ReLU activations. Outputs Q-estimate for each action.



source: Mnih et al. (Nature, 2015), details in appendix "Model architecture"

DQN for Atari: dirty engineering tricks

- Preprocessing: 210x160 RGB color images are converted to grayscale and resized to 84x84 resolution.
- Frame skipping: the agent only observes and acts in every K-th frame, for the frames in between, the last selected action is repeated without providing the frame to the agent. (In the paper, $K = 4$.)
- Reward clipping: all positive one-step Atari rewards are clipped to $+1$, all negative ones are clipped to -1 (Atari gives integer rewards).
- TD error clipping: for each update, the Q-learning error $r + \gamma \cdot \max b \in \mathcal{A}(s')Q(s', b) - Q(s, a)$ is clipped to $[-1, 1]$.

DQN for Atari: selected hyperparameters (nex)

minibatch size $Bsize$	32
replay buffer size $\mathcal{B}size$	1,000,000
target network update freqeuency C	10,000
discount factor γ	0.99
update frequency (steps between two minibatch updates)	4
learning rate	0.00025
initial ε	1
final ε (linear decay)	0.1
final decay frame	1,000,000
random policy played for init.	50,000 frames
max. do-nothing actions at episode start	30

RAINBOW of heuristics

Multitude of heuristics for the improvement of DQN were developed over time. Some of them make sense also in the context of other deep RL algorithms.

The RAINBOW agent combines six such heuristics to further improve the DQN performance on Atari games.

Rainbow: Combining Improvements in Deep Reinforcement Learning

Matteo Hessel DeepMind
Joseph Modayil DeepMind
Hado van Hasselt DeepMind
Tom Schaul DeepMind
Georg Ostrovski DeepMind
Will Dabney DeepMind
Dan Horgan DeepMind
Bilal Piot DeepMind
Mohammad Azar DeepMind
David Silver DeepMind

(In proceedings of AAAI 2018.)

Rainbow heuristics

- dueling networks architecture
- double DQN
- prioritized experience replay
- n-step rewards
- distributional learning
- noisy networks

Action advantage

Idea: imagine that for some state s, the Q-values of all actions are high. Then s should be in some sense valuable in itself.

Definition 53: Advantage

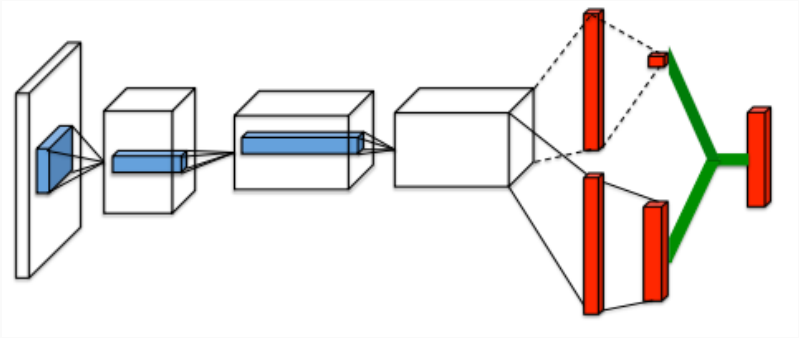
Let π be a policy. An advantage function $adv^\pi : \mathcal{S} \times \mathcal{A} \to \mathbb{R}$ is defined as

$$adv^\pi(s, a) = q^\pi(s, a) - v^\pi(s).$$

Dueling architecture splits certain layers of the neural network into two "streams", one estimating (somethign like) $v^\pi(s)$ and one estimating (something like) $adv^\pi(s, a)$. The final layer combines these estimates to produce an estimate of $q^\pi(s, a)$.

Dueling architecture

Wang et al.: *Dueling Network Architectures for Deep Reinforcement Learning*. In proceedings of ICML'16.



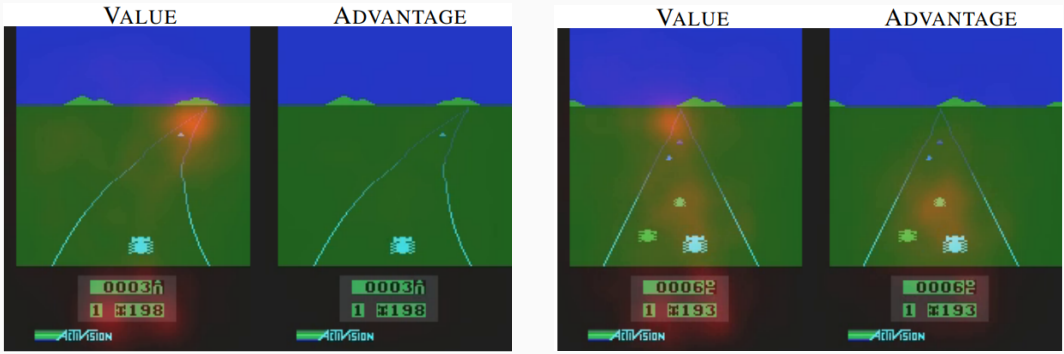
Dueling architecture: theory and training

We have $Q_{\theta,\alpha,\beta}(s,a) = aggregate(V_{\theta,\alpha}(s), A_{\theta,\beta}(s,a))$, where

- θ - convolutional (or other feature extraction) layer parameters
- α - value channel parameters
- β - advantage channel parameters

The whole network Q is trained to estimate q^π (where π is the target policy) using any deep RL algorithm (e.g. DQN, in which case π is the optimal policy). There is nothing new from RL perspective here, all the novelty is inside the network. The factorization into state value and advantage is supposed to help the network "focus" on features that are important to recognize valuable states and features that help us rank actions.

Dueling architecture: Atari example



From Wang et al.: *Dueling Network Architectures for Deep Reinforcement Learning*. In proceedings of ICML'16.

Learning state values and advantages

- The whole net is trained end-to-end to predict q^π.
- How do we ensure the value/advantage channels are trained to predict state values/advantages?
- By a suitable choice of aggregator:
  - $Q_{\theta,\alpha,\beta}(s,a) = V_{\theta,\alpha}(s) + A_{\theta,\beta}(s,a)$ does not work: e.g. $V_{\theta,\alpha}$ could converge to constant 0 and $A_{\theta,\beta}$ to q^π.
  - $$Q_{\theta,\alpha,\beta}(s,a) = V_{\theta,\alpha}(s) + A_{\theta,\beta}(s,a) - \max_{b \in \mathcal{A}(s)} A_{\theta,\beta}(s,b),$$

    the training then indeed pushes $V_{\theta,\alpha}$ to v^π and $A_{\theta,\beta}$ to $adv^\pi + c$ where c is some constant. Issues: not differentiable, update sensitive to the value of maximizing action changes.

Aggregation in Rainbow

The point: aggregating layer should anchor the sum of the channels to same baseline value derived non-trivially from the advantages (if all advantages shift up/down, so should the baseline). Rainbow uses mean advantage baseline:

$$Q_{\theta,\alpha,\beta}(s,a) = V_{\theta,\alpha}(s) + A_{\theta,\beta}(s,a) - \frac{1}{|\mathcal{A}(s)|} \sum_{b \in \mathcal{A}(s)} A_{\theta,\beta}(s,b)$$

pushes the value channel to predict $\frac{1}{|\mathcal{A}(s)|} \sum_{a \in \mathcal{A}(s)} q^{\pi}(s,a)$.

Double DQN

van Hasselt, Guez, Silver: *Deep Reinforcement Learning with Double Q-learning.* In proceedings of AAAI 2016.

Similar idea to tabular Double Q-learning (use different estimates for selecting maximizing action in bootstrap and for evaluating the bootstrap), but instead of independently updated networks uses main and target networks. I.e., for experience (s, a, r, s'), the update is:

$$\theta \leftarrow \theta + \alpha \cdot \left[r + \gamma \cdot Q_{\hat{\theta}}\left(s', \operatorname*{arg\,max}_{b \in \mathcal{A}(s')} Q_\theta(s', b)\right) - Q_\theta(s, a) \right] \nabla_\theta Q_\theta(s, a),$$

where $\hat{\theta}$ is the parameter vector of the target network.

Prioritized experience replay (Schaul et al., ICLR'16)

Each experience $e = (s, a, r, s')$ in the replay buffer is assigned a priority according to its TD-error

$$p_e = |r + \gamma \cdot \max_{b \in \mathcal{A}(s')} Q_{\hat{\theta}}(s', b) - Q_\theta(s, a)| + \varepsilon$$

($\varepsilon > 0$ ensures all priorities are positive).

The probability of sampling an experience e from the buffer is set to $\frac{p_e^\alpha}{\sum_{e' \in \mathcal{B}} p_{e'}^\alpha}$, where $\alpha > 0$ is a hyperparameter controlling the degree of prioritization.

Prioritization induces bias: the sampled experiences no longer follow the same distribution as sampled trajectories. We can correct this by using importance sampling during updates:

$$\theta \leftarrow \theta + \alpha \cdot \left(\frac{1}{|\mathcal{B}|} \cdot \frac{1}{p_e^\alpha} \right)^\beta \cdot \left[r + \gamma \cdot Q_{\hat{\theta}}\left(s', \arg\max_{b \in \mathcal{A}(s')} Q_\theta(s', b)\right) - Q_\theta(s, a) \right] \cdot \nabla_\theta Q_\theta(s, a),$$

where $\beta > 0$ determines the degree of IS correction (annealed to 1 during training).

n-step returns

Self-explanatory, use n-step return with Q-learning bootstrap when computing TD target.

How to combine with replay buffer? Each experience stores a single step.

Solutions:

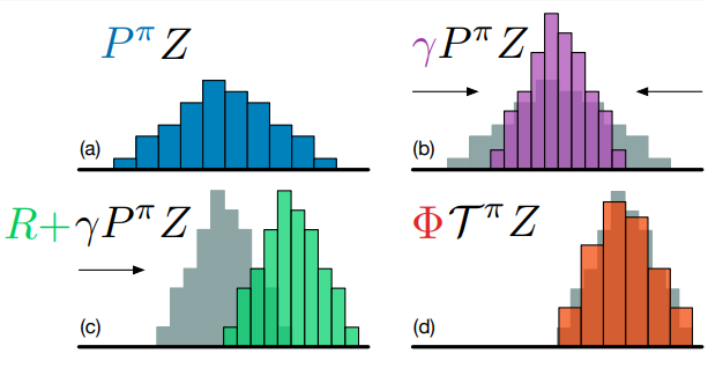
- Store experiences in $\mathcal{B}$ sequentially, with each sampled experience, retrieve also the next $n-1$ ones (up to episode termination). Requires careful implementation.
- Naive: each element of $\mathcal{B}$ consists of n consecutive experiences (space inefficient).

Given consecutive experiences
$(s_t, a_t, r_{t+1}, s_{t+1}), (s_{t+1}, a_{t+1}, r_{t+2}, s_{t+2}), \ldots, ((s_{t+n-1}, a_{t+n-1}, r_{t+n}, s_{t+n}))$, perform update

$$\theta \leftarrow \theta + \alpha \cdot \left[r_{t+1} + \gamma r_{t+2} + \cdots + \cdot\gamma^{n-1} r_{t+n} + \gamma^n \max_{b \in \mathcal{A}(s_{t+n})} Q_{\hat{\theta}}(s_{t+n}, b) - Q_\theta(s_t, a_t)\right] \nabla_\theta Q_\theta(s_t, a_t).$$

Distributional learning

Very rough idea: instead of expected returns, predict (discretized) distribution of returns.



Source: Bellemare, Dabney, Munos: *A Distributional Perspective on Reinforcement Learning*. In proceedings of ICML'17.

We still optimize the expected value of the distribution, but the NN processes richer information (neurological inspiration).

Noisy nets

Fortunato et al.: *Noisy Networks for Exploration* . In proceedings of ICRL'17.

An alternative way of achieving exploration (without ε-greedy policies). Replaces linear layers $y = W \cdot x + b$ with noisy layers of the form

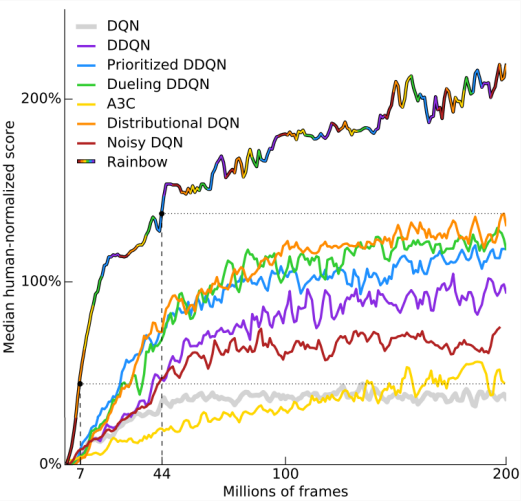
$$y = (\mu_w + \sigma_w \odot \varepsilon_w) \cdot x + \mu_b + \sigma_b \odot \varepsilon_b,$$

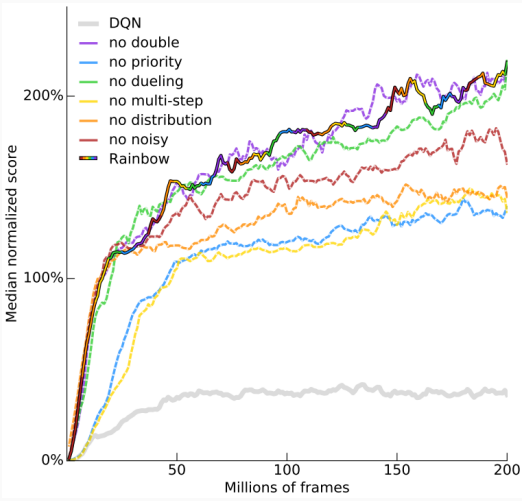
where matrices μ_w, σ_w and vectors μ_b, σ_b are learnable, matrix ε_w and vector ε_b consist of random noise, and $\odot$ represents component-wise multiplication.

The loss function of the DQN training is then encapsulated in expectation over the noise.

Interesting point: the net can learn to adjust σ's and thus the degree of exploration over time.

Rainbow: evaluation and ablations (Hessel et al., 2017)





Policy Gradient Methods

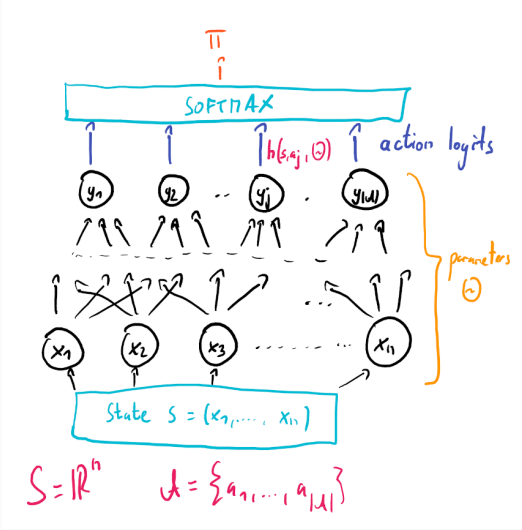
Value-based vs. policy-based methods

So far: focus on approximating q^* via some parameterized estimate Q_θ, policy the defined by Q_θ, e.g. (ε-)-greedy...

In policy gradient methods we work directly with some parameterized representation of a policy π_θ, and update θ so as to improve some performance characteristic of π_θ (e.g., expected return).

In particular, the policy π can be represented by a function approximator $\pi_\theta \colon \mathcal{S} \times \Theta \to \mathcal{D}(\mathcal{A})$.

Standard NN + softmax representation



$$\pi_\theta(a|s) = \frac{e^{h(s,a,\theta)}}{\sum_{b\in\mathcal{A}} e^{h(s,b,\theta)}},$$

where $h(s, a, \theta)$ is the logit ("preference") of action a

General policy gradient scheme

We want to find θ that maximizes some performance measure (or objective function) $J(\theta)$ of π_θ. The obvious choice for J is the expected return:

$$J(\theta) = v^{\pi_\theta} = \mathbb{E}^{\pi_\theta}[G],$$

thought some algorithms use a different surrogate objective function.

The optimization problem

$$\max_\theta J(\theta)$$

can be (locally) solved using gradient ascent: repeatedly perform updates

$$\theta \leftarrow \theta + \alpha \cdot \nabla_\theta J(\theta).$$

It is thus necessary to compute or approximate $\nabla_\theta J(\theta)$: this is the scope of various policy gradient theorems.

Gradient of expected return: possible version

$$\nabla_\theta J(\theta) = \nabla_\theta \mathbb{E}^{\pi_\theta}[G] =$$

Gradient of expected return (cont'd)

Vanilla MC policy gradient

Algorithm 8: Vanilla MC policy gradient

Input: Black-box MDP $\mathcal{M} = (\mathcal{S}, \mathcal{A}, p, r)$, policy parametrization π_θ, learning rate α

Output: Approximation π_θ of π^*

initialize θ arbitrarily;

repeat

  $s_0 \sim$ initial distribution;

  generate episode $\tau = s_0, a_0, r_1, s_1, a_1, r_2, s_2, a_2, r_3, s_3, \ldots, r_T, s_T$ using π_θ;

  $\theta \leftarrow \theta + \alpha \cdot G(\tau) \cdot \sum_{t=0}^{i} \nabla_\theta \log \pi_\theta(a_j | s_j)$

until *timeout*;

Score function form

Step-wise gradient

$$\nabla_\theta J(\theta) = \nabla_\theta \mathbb{E}^{\pi_\theta}[G] = \nabla_\theta \mathbb{E}^{\pi_\theta}[\sum_{t=0}^{\infty} \gamma^t R_{t+1}]$$

Step-wise gradient (cont'd)

REINFORCE: better MC policy gradient

Algorithm 9: REINFORCE (Williams, 1992)

Input: Black-box MDP $\mathcal{M} = (\mathcal{S}, \mathcal{A}, p, r)$, policy parametrization π_θ, learning rate α

Output: Approximation π_θ of π^*

initialize θ arbitrarily;

repeat

  $s_0 \sim$ initial distribution;

  generate episode $\tau = s_0, a_0, r_1, s_1, a_1, r_2, s_2, a_2, r_3, s_3, \ldots, r_T, s_T$ using π_θ;

  $G \leftarrow 0$;

  for $t = T - 1$ **to** 0 **do**

    $G \leftarrow r_{t+1} + \gamma \cdot G$;

    $\theta \leftarrow \theta + \alpha \cdot \gamma^t \cdot G \cdot \nabla_\theta \log \pi_\theta(a_t|s_t)$

until *timeout*;

Baseline in policy gradient

Theorem 54: Baseline theorem

Let $b_\theta(s)\colon \mathcal{S} \times \theta \to \mathbb{R}$ be any function. Then for any t:

$$\mathbb{E}^{\pi_\theta}\left[b_\theta(s_t) \cdot \nabla_\theta log \pi_\theta(a_t|s_t)\right] = 0.$$

As a consequence

$$\nabla_\theta J(\theta) = \mathbb{E}^{\pi_\theta}\Big[\sum_{t=0}^{T} \gamma^t \cdot G_t \cdot \nabla_\theta \log \pi_\theta(a_t|s_t)\Big] =$$

$$= \mathbb{E}^{\pi_\theta}\Big[\sum_{t=0}^{T} \gamma^t \cdot \big(G_t - b_\theta(s_t)\big) \cdot \nabla_\theta \log \pi_\theta(a_t|s_t)\Big]$$

The gradient estimates using baseline have the same expectations as the standard REINFORCE estimate, but might have a lower variance if baseline selected correctly.

State-value baseline

Good choice is

$$b(s) := v^{\pi_\theta}(s),$$

reducing the estimate variance by correcting the return for a bias caused by being in a certain state.

Problem: we do not know v^{π_θ}.

Solution: Learn v^{π_θ} online using a separate function approximator V, e.g. via gradient (every-visit) Monte Carlo.

REINFORCE with a state-value baseline

Algorithm 10: REINFORCE with baseline

Input: Black-box MDP $\mathcal{M} = (\mathcal{S}, \mathcal{A}, p, r)$, policy parametrization π_θ, value parametrization V_η, learning rates α_π, α_V for the two approximators

Output: Approximation π_θ of π^*

initialize θ and η arbitrarily;

repeat

  $s_0 \sim$ initial distribution;

  generate episode $\tau = s_0, a_0, r_1, s_1, a_1, r_2, s_2, a_2, r_3, s_3, \ldots, r_T, s_T$ using π_θ;

  $G \leftarrow 0$;

  for $t = T - 1$ **to** 0 **do**

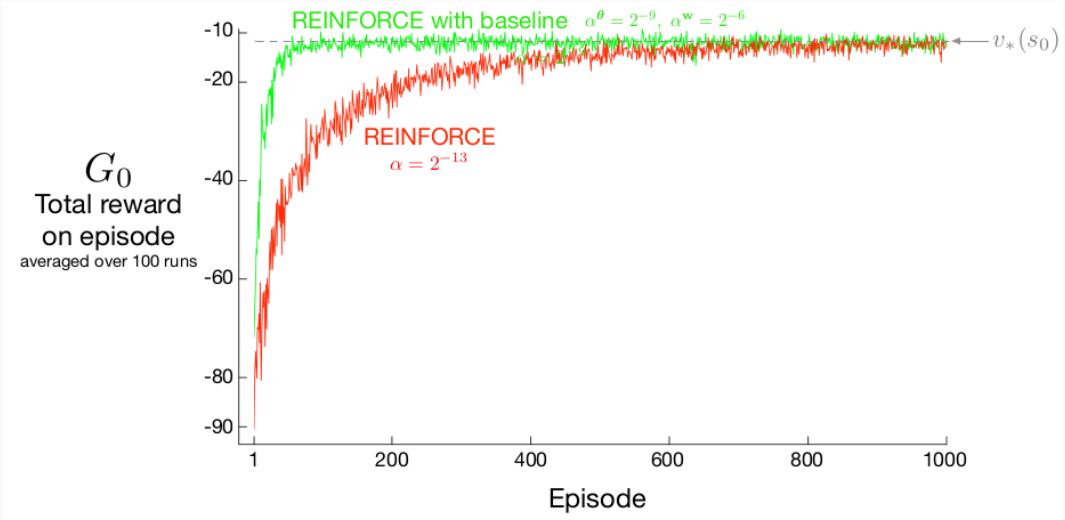
    $G \leftarrow r_{t+1} + \gamma \cdot G$;

    $\eta \leftarrow \eta + \alpha_V \cdot \big[G - V_\eta(s_t)\big] \cdot \nabla_\eta V_\eta(s_t)$;

    $\theta \leftarrow \theta + \alpha_\pi \cdot \gamma^t \cdot \big[G - V_\eta(s_t)\big] \cdot \nabla_\theta \log \pi_\theta(a_t|s_t)$

until *timeout*;

REINFORCE: baseline effect experiment



source: Sutton&Barto, p. 330

Advantage estimation in policy gradient

Comparing returns to some state-dependent baseline is reminiscent of what happened in the dueling network architecture.

In particular, one can derive another form of the policy gradient, showing that

$$\nabla_\theta J(\theta) \propto \mathbb{E}_{s\sim\pi, a\sim\pi_\theta}[q^{\pi_\theta}(s,a) \cdot \nabla_\theta \log \pi(a|s)].$$

Inputting a state-value baseline yields

$$\nabla_\theta J(\theta) \propto \mathbb{E}_{s\sim\pi, a\sim\pi_\theta}[\underbrace{(q^{\pi_\theta}(s,a) - v^{\pi}(s))}_{adv^{\pi}(s,a)} \cdot \nabla_\theta \log \pi(a|s)].$$

Hence, policy-gradient-type algorithms often formulate the update of policy parameters in the form

$$\theta \leftarrow \theta + \alpha \cdot A^t_{\theta,\eta}(s_t, a_t) \cdot \nabla_\theta \log \pi_\theta(a_t|s_t),$$

where $A^t_{\theta,\eta}$ is some approximator called advantage estimate. E.g. in PG with baseline, $A^t_{(\theta,\eta)} = \gamma^t \cdot (G_t - V_\eta(s_t))$.

Proof of Baseline theorem

Actor-critic: Policy gradient with bootstrapping

Recall the REINFORCE-with-baseline update:

$$\theta \leftarrow \theta + \alpha \cdot \gamma^t \cdot (G_t - V_\eta(s_t)) \cdot \nabla_\theta \log \pi_\theta(a_t|s_t)$$

G_t is a possible source of variance: let's remove it (at the cost of introducing bias) via bootstrapping! E.g. TD(0):

$$\theta \leftarrow \theta + \alpha \cdot \gamma^t \cdot (r_t + \gamma \cdot V_\eta(s_{t+1}) - V_\eta(s_t)) \cdot \nabla_\theta \log \pi_\theta(a_t|s_t).$$

Here, the value network V_η both estimates the baseline value of the current state, and (via boostrap) the quality of the played action: we call it a critic, while the policy network π_θ is called an actor.

Note: we can use the same bootstrap to update critic parameters.

Basic Actor-Critic (AC) algorithm: pseudocode

Algorithm 11: One-step AC

Input: Black-box MDP $\mathcal{M} = (\mathcal{S}, \mathcal{A}, p, r)$, policy parametrization π_θ, value parametrization V_η, learning rates α_π, α_V for the two approximators

Output: Approximation π_θ of π^*

initialize θ and η arbitrarily;

repeat

  $s \sim$ initial distribution;

  $D \leftarrow 1$;

  while *s is not terminal* **do**

    $a \sim \pi_\theta(s)$;

    $s' \sim p(s, a)$;

    $\delta \leftarrow r(s, a, s') + \gamma \cdot V_\eta(s') - V_\eta(s)$;

    $\eta \leftarrow \eta + \alpha_V \cdot \delta \cdot \nabla_\eta V_\eta(s_t)$;

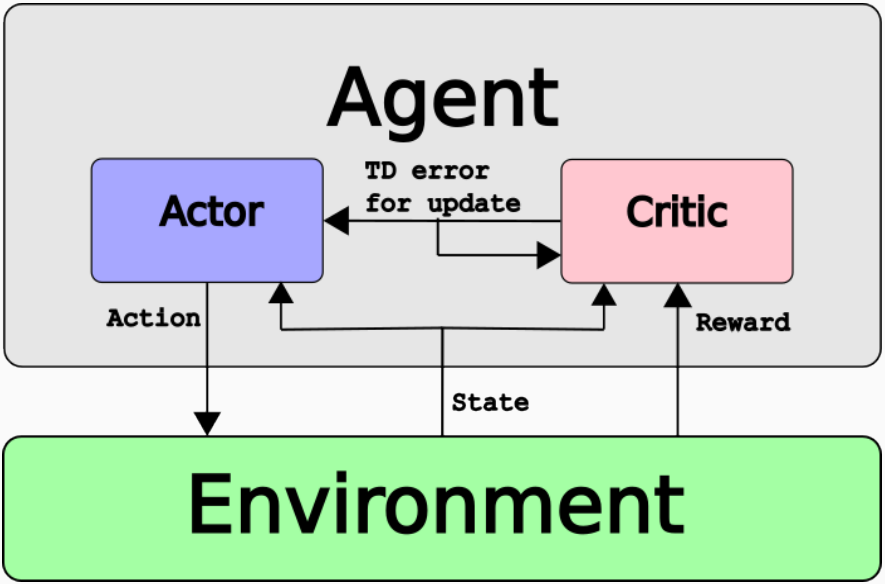
    $\theta \leftarrow \theta + \alpha_\pi \cdot D \cdot \delta \cdot \nabla_\theta \log \pi_\theta(a_t|s_t)$;

    $D \leftarrow \gamma \cdot D$;

    $s \leftarrow s'$

until *timeout*;

AC: picture


Agent
Actor
TD error
for update
Critic
Action
Reward
State
Environment

Comments on AC

- The algorithm presented on previous slide is just a basic variant: actor-critic framework covers a wide range of algorithms: soft Actor-Critic (SAC), A2C, A3C, deep deterministic policy gradient (DDPG), twin-delayed DDPG (TD3), PPO (next time),...

Varieties of policy gradient heuristics

- Entropy regularization: add to the objective function a term representing the entropy of the policy: we prefer "more" randomized policies to encourage exploration (used e.g. in SAC).
  - $J_{ENTR}(\theta) = \mathbb{E}^{\pi}[G] + \beta \cdot \mathbb{E}_{s \sim \pi_\theta}[H(\pi(s))]$
- Off-policy training by using replay buffer (e.g. in SAC, DDPG, TD3).
- Using parallel agents whose gradients are averaged for each update (A2C).
- Using n-step (e.g. A2C) or λ-returns in bootstrap (PPO).
- Using different $J(\theta)$ than just expected return (SAC, TRPO, PPO).

Note that the above algorithms typically differ from "vanilla" AC in more aspects then presented above. We shall see soon on the case of PPO.

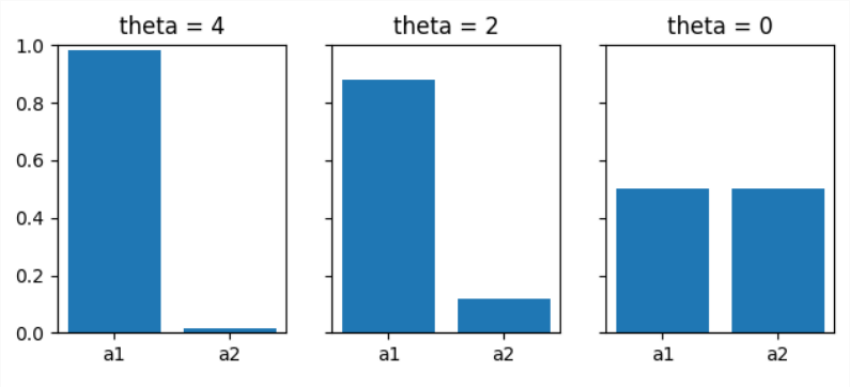
Taming Unstable Gradients with Trust Regions: TRPO, PPO

Limitations of policy gradient methods

- Basic policy gradient methods are prone to large variance of gradient estimates.
- These can be mitigated to some degree, e.g. by using n-step or λ-returns in advantage estimation.
- Even then, the methods can yield large updates which can destabilize the training. We are never sure the parameter updates will actually lead to an improvement of a policy:

Sensitivity of policy to parameters

For two actions and $\pi(a|s) = \begin{cases} \frac{1}{1-e^{-\theta}} & a = a_1 \\ 1 - \frac{1}{1-e^{-\theta}} & a = a_2, \end{cases}$ we get



Source: slides by Emma Brunskill, Lecture 6,
https://web.stanford.edu/class/cs234/modules.html

Direct policy improvement via surrogate objectives

- We will present algorithms that use different performance metric so as increase the chance of policy improvement on update.
- Moreover, we will design the performance metric so that its gradient can be easily computed by an automated differentiation tool without the need to derive the formulas manually.

Overall structure of presented algorithms

- The algorithms will generate a sequence of policies π_1, π_2, π_3 such that each policy will be (likely) an improvement over the previous one.
- Each step from π_i to π_{i+1} entails finding a policy π_{i+1} that optimizes some performance metric $\mathcal{L}_{\pi_i}$ that is dependent on the previous policy π_i! (Cf. standard policy gradient: the performance metric J evaluated only the current policy).
- I.e. a run of such an algorithm entails solving (using gradient-based methods) multiple optimization problems: one per each policy update.

- We will now focus on the single improvement step: we will denote by
  - $\pi = \pi_\theta$ the previous policy (π_i)
  - $\pi' = \pi_{\theta'}$ the new policy (π_{i+1}) we seek.

  θ is treated as a constant, θ' as variables!

Roles of advantages in policy improvement

Theorem 55

Let θ, θ' be two parameter vectors and $\pi = \pi_\theta, \pi' = \pi_{\theta'}$.
Then

$$J(\theta') - J(\theta) = \underbrace{\mathbb{E}_{\tau \sim \pi'}\left[\sum_{t=0}^{\infty} \gamma^t \cdot adv^{\pi}(s_t, a_t)\right]}_{\stackrel{\text{def}}{=} \mathcal{L}_{\pi}(\pi')}.$$

Another loss surrogate

To ensure that update from θ to θ' is an improvement, we want to maximize

$$\mathcal{L}_\pi(\pi') = \mathbb{E}_{\tau\sim\pi'}\Big[\sum_{t=0}^{\infty}\gamma^t \cdot adv^\pi(s_t, a_t)\Big]$$

But we cannot sample from π', neither can we easily compute the gradient of the loss by automated differentiation.

Trick: the loss function $\mathcal{L}_\pi$ behaves similarly to the following loss function $\widetilde{\mathcal{L}}_\pi$ for all points π' that are "close enough" to π:

$$\widetilde{\mathcal{L}}_\pi(\pi') = \mathbb{E}_{\tau\sim\pi}\Big[\sum_{t=0}^{\infty}\gamma^t \cdot adv^\pi(s_t, a_t) \cdot \frac{\pi'(a_t|s_t)}{\pi(a_t|s_t)}\Big]$$

.

Closeness of $\mathcal{L}_\pi$ and $\widetilde{\mathcal{L}}_\pi$

$$\mathcal{L}_\pi(\pi) = \mathbb{E}_{\tau\sim\pi'}\Big[\sum_{t=0}^{\infty}\gamma^t \cdot adv^\pi(s_t, a_t)\Big] \qquad \widetilde{\mathcal{L}}_\pi = \mathbb{E}_{\tau\sim\pi}\Big[\sum_{t=0}^{\infty}\gamma^t \cdot adv^\pi(s_t, a_t) \cdot \frac{\pi'(a_t|s_t)}{\pi(a_t|s_t)}\Big]$$

"Behaves similarly" can be formalized as follows:

Theorem 56

It holds $\mathcal{L}_\pi(\pi) = \widetilde{\mathcal{L}}_\pi(\pi)$.
Moreover the gradients $\nabla_{\theta'}\mathcal{L}_\pi$ and $\nabla_{\theta'}\widehat{\mathcal{L}}_\pi$ are equal at point π.

Proof: the first part is trivial. The second part is technical and requires converting the expectation into expectation-over-states form, see

Optional reading: Schulman, Levine, Abbeel, Jordan, Moritz: *Trust Region Policy Optimization*. In Proceedings of ICML'15.

Hence, the constrained optimization problems

$$\text{maximize } \mathcal{L}_\pi(\pi') \text{ subject to } \pi' \text{ close to } \pi$$
$$\text{maximize } \widetilde{\mathcal{L}}_\pi(\pi') \text{ subject to } \pi' \text{ close to } \pi$$

have approximately the same optimal solutions. We want to solve the first, and will proceed by solving the second.

The set of π' that are "close enough" to π is called a trust region. What "close enough" means differs among algorithms. Exact bounds on the error of the approximation in terms of KL divergence between π and π' was given in

Optional reading: Achiam, Held, Tamar, Abbeel: *Constrained Policy Optimization*. In Proceedings of ICML'17.

Optimizing the loss by sampling and automated differentiation

$$\widetilde{\mathcal{L}}_\pi = \mathbb{E}_{\tau\sim\pi}\Big[\sum_{t=0}^{\infty}\gamma^t \cdot adv^\pi(s_t,a_t)\cdot\frac{\pi'(a_t|s_t)}{\pi(a_t|s_t)}\Big]$$

We replace the true advantage adv^π with an advantage estimator A_η (neural net, more on that later).

Instead of optimizing the true loss $\widetilde{\mathcal{L}}_\pi$, we optimize a sample loss $\widehat{\mathcal{L}}_\pi$: for a trajectory $\tau = s_0, a_0, r_1, s_1, a_1, r_2, s_2, a_2, r_3, s_3, \ldots$:

$$\widetilde{\mathcal{L}}_\pi(\pi') \approx \widehat{\mathcal{L}}_\pi(\pi') = \sum_{t=0}^{\infty}\gamma^t \cdot A_\eta(\tau,t)\cdot\frac{\pi'(a_t|s_t)}{\pi(a_t|s_t)}$$

(More trajectories can be sampled, in which case we optimize the average sample loss over the trajectories.) We then let an automated gradient-based optimizer find $\pi' = \pi_{\theta'}$ maximizing $\widehat{\mathcal{L}}_\pi(\pi')$. Note that the only term in $\widehat{\mathcal{L}}_\pi$ that depends on the optimized parameters θ' are the likelihood ratios! Hence, the gradient can be computed easily.

Trust region policy optimization (TRPO)

Schulman, Levine, Abbeel, Jordan, Moritz: *Trust Region Policy Optimization.* In Proceedings of ICML'15.

To perform update from π to π', theoretical TRPO:

- samples a trajectory τ (or a batch of trajectories) from π
- uses the trajectory to:
  - update the advantage estimator A_η (i.e., update its parameters η)
  - construct the loss $\widehat{\mathcal{L}}_\pi$
- solves the optimization problem

$$\text{maximize } \underbrace{\widehat{\mathcal{L}}_\pi(\pi') - \beta \cdot D_{KL}(\pi, \pi')}_{J_{TRPO}}$$

(D_{KL} - KL divergence, β - suitable constant) via a black-box gradient-based optimizer.

This update loop is performed until timeout. Practical TRPO makes several changes to individual steps of the above scheme.

Proximal Policy Optimization (PPO)

Proximal Policy Optimization Algorithms

John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, Oleg Klimov
OpenAI
{joschu, filip, prafulla, alec, oleg}@openai.com

Abstract

We propose a new family of policy gradient methods for reinforcement learning, which alternate between sampling data through interaction with the environment, and optimizing a "surrogate" objective function using stochastic gradient ascent. Whereas standard policy gradient methods perform one gradient update per data sample, we propose a novel objective function that enables multiple epochs of minibatch updates. The new methods, which we call proximal policy optimization (PPO), have some of the benefits of trust region policy optimization (TRPO), but they are much simpler to implement, more general, and have better sample complexity (empirically). Our experiments test PPO on a collection of benchmark tasks, including simulated robotic locomotion and Atari game playing, and we show that PPO outperforms other online policy gradient methods, and overall strikes a favorable balance between sample complexity, simplicity, and wall-time.

PPO vs TRPO

PPO follows the same general scheme as TRPO with the following tweaks:

- Explicitly specifies how the advantage should be estimated: generalized advantage estimation (essentialy, truncated offline λ-returns, see later).
- Tweaks the loss function a bit - uses different way of ensuring that π' is in the proximity of π.

PPO comes in two variants depending on how it tweaks the loss function:

- Adaptive KL divergence penalty coefficient: like TRPO, but the coefficient β changes adaptively. Empirically does not perform as well as the second method:
- Clipped likelihood ratios.

Definition 57: Clipping

The function *CLIP* is defined as follows:

$$CLIP(x, a, b) = \begin{cases} x & \text{if } a \leq x \leq b \\ a & \text{if } x < a \\ b & \text{if } b < x \end{cases}$$

Basic PPO empirical loss

For a trajectory $\tau = s_0, a_0, r_1, s_1, a_1, r_2, s_2, a_2, r_3, s_3, \ldots$ denote

$$r_t(\pi, \pi') = \frac{\pi'(a_t|s_t)}{\pi(a_t|s_t)}$$

The main component of PPO loss/performance metric is:

$$\mathcal{L}_{\pi}^{CLIP}(\pi') = \sum_{t=0}^{\infty} \min\left(A_\eta(\tau, t) \cdot r_t(\pi, \pi'),\ A_\eta(\tau, t) \cdot CLIP\Big(r_t(\pi, \pi'), 1-\varepsilon, 1+\varepsilon\Big) \right)$$

(Note that no γ^t is included in the formula: discounting to some degree implicitly encompassed in the advantage estimation.)

Constraining improvements but not damages

$$\mathcal{L}_{\pi}^{CLIP}(\pi') = \sum_{t=0}^{\infty} \min\left(A_\eta(\tau, t) \cdot r_t(\pi, \pi'),\ A_\eta(\tau, t) \cdot CLIP\Big(r_t(\pi, \pi'), 1-\varepsilon, 1+\varepsilon\Big)\right)$$

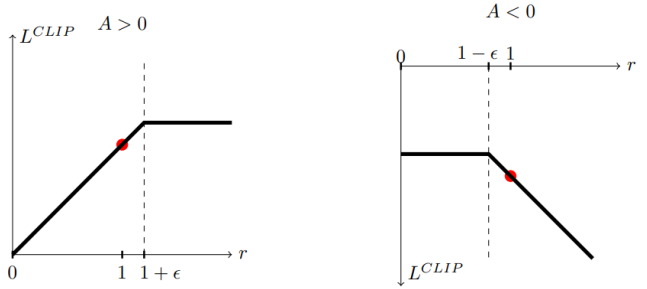


Figure 1: Plots showing one term (i.e., a single timestep) of the surrogate function L^{CLIP} as a function of the probability ratio r, for positive advantages (left) and negative advantages (right). The red circle on each plot shows the starting point for the optimization, i.e., $r = 1$. Note that L^{CLIP} sums many of these terms.

source: Schulman et al.: Proximal Policy Optimization Algorithms

Advantage estimation in PPO

PPO uses generalized advantage estimation = an actor-critic framework using λ-return to estimate the q-value.

Formally, the advantage estimator is built on a value network $V_\eta \colon \mathcal{S} \times \Theta \to \mathbb{R}$.

Let $\tau = s_0, a_0, r_1, s_1, a_1, r_2, s_2, a_2, r_3, s_3, \dots$ be a trajectory. An n-step advantage from time step t is the quantity

$$A_\eta^{t:t+n}(\tau) = r_{t+1} + \gamma r_{t+2} + \cdots + \gamma^{n-1} r_{t+n} + \gamma^n V_\eta(s_{t+n}) - V_\eta(s_t).$$

For a parameter $\lambda \in [0, 1]$, the generalized advantage estimate at time t is

$$GAE_\eta^{(\lambda,t)}(\tau) = \sum_{n=1}^{T-t} \lambda^n \cdot A_\eta^{t:t+n}(\tau).$$

For a trajectory τ, PPO puts $A_\eta(\tau, t) = GAE_\eta^{\lambda,t}(\tau)$.

PPO loss extension

Apart from $\mathcal{L}_\pi^{CLIP}$, PPO loss consist of two additional terms:

1. The value network V_η and policy network π_θ might share parameters (e.g. the same feature extraction layers). In this case, η and θ should be trained together: for a sampled trajectory $\tau = s_0, a_0, r_1, s_1, a_1, r_2, s_2, a_2, r_3, s_3, \ldots$, the PPO loss will incorporate the empirical value loss

$$\mathcal{L}^V(\eta') = \sum_{t=0}^{T} (V_{\eta'}(s_t) - \underbrace{G_t(\tau)}_{\text{target}})^2$$

(instead of sample return, the training target might be e.g. TD(0), or $A_\eta(\tau, t) + V_\eta(s_t)$: note that η in the target (current value of the parameter) is a constant).

2. Original PPO also used entropy regularization, adding sample entropy loss:

$$\mathcal{L}^{entropy}(\pi') = -\frac{1}{T} \sum_{t=0}^{T-1} \sum_a \pi'(a|s_t) \cdot \log \pi'(a|s_t)$$

Total PPO loss

Given a sampled trajectory $\tau = s_0, a_0, r_1, s_1, a_1, r_2, s_2, a_2, r_3, s_3, \ldots$, the current reference policy π, and hyperparameters $\beta_1, \beta_2 \in \mathbb{R}^+$, the canonical PPO loss is:

$$\begin{aligned}\mathcal{L}_\tau^{PPO}(\theta', \eta') &= \mathcal{L}_\pi^{CLIP}(\pi') - \beta_1 \cdot \mathcal{L}^V(\eta') + \beta_2 \cdot \mathcal{L}^{entropy}(\pi') \\ &= \sum_{t=0}^{T} \min\left(A_\eta(\tau, t) \cdot r_t(\pi, \pi'),\ A_\eta(\tau, t) \cdot CLIP\Big(r_t(\pi, \pi'), 1-\varepsilon, 1+\varepsilon\Big)\right) \\ &\quad - \beta_1 \cdot \sum_{t=0}^{T} (V_{\eta'}(s_t) - \underbrace{G_t(\tau)}_{\text{target}})^2 + \beta_2 \cdot -\frac{1}{T} \sum_{t=0}^{T-1} \sum_a \pi'(a|s_t) \cdot \log \pi'(a|s_t).\end{aligned}$$

Important note: the advantage estimates A_η in $\mathcal{L}^{CLIP}$ are evaluated before the loss is constructed and are treated as constants inside $\mathcal{L}^{CLIP}$! The value network parameters are only considered variable inside the value loss.

Usually, a batch of τ is sampled and average loss $\frac{1}{|\text{batch}|} \cdot \sum_{\tau \in \text{batch}} \mathcal{L}_\tau^{PPO}(\theta', \eta')$ is optimized using gradient-based optimizer (eg., ADAM) to yield a policy update.

PPO high-level pseudocode

Algorithm 12: PPO

Input: policy network π_θ, value network V_η, hyperparameters λ, $|B|$, β_1, β_2,...

initialize θ, η;

repeat

  sample a batch B of trajectories from policy $\pi = \pi_\theta$;

  foreach $\tau \in B$ **do**

    foreach $0 \leq t \leq T-1$ **do**

      $a_t(\tau) \leftarrow GAE_\eta^{\lambda,t}(\tau)$;

    define $\mathcal{L}_\tau^{CLIP}(\pi') = \sum_{t=0}^{T} \min\Big(a_t(\tau) \cdot r_t(\pi,\pi')\,,\; a_t(\tau) \cdot CLIP(r_t(\pi,\pi'), 1-\varepsilon, 1+\varepsilon)\Big)$;

    define $\mathcal{L}_\tau^{PPO}(\theta',\eta') = \mathcal{L}_{\pi,\tau}^{CLIP}(\pi') - \beta_1 \cdot \mathcal{L}^V(\eta') + \beta_2 \cdot \mathcal{L}^{entropy}(\pi')$

  define $\mathcal{L}^{PPO}(\theta',\eta') = \frac{1}{|B|} \cdot \sum_{\tau \in B} \mathcal{L}_\tau^{PPO}(\theta',\eta')$;

  use ADAM or other optimizer to find θ', η' approximately maximizing $\mathcal{L}^{PPO}(\theta',\eta')$;

  $\theta \leftarrow \theta'$; $\eta \leftarrow \eta'$

until *timeout*;

Exploration vs. Exploitation: A Systematic Approach

EvE dilemma

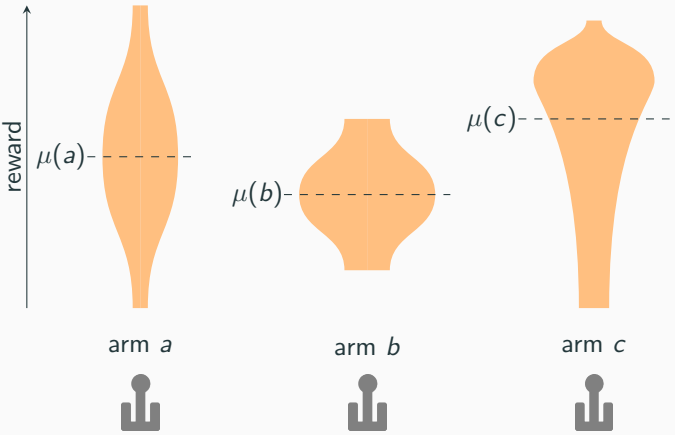
When you see a good move, look for a better one.

Emanuel Lasker (1868-1941), 2nd World Chess Champion

- In RL, the algorithms often need to balance exploration of the MDP state space with exploitation: focusing on behavior that worked well in the past.
- Typical RL algorithms ensure exploration by ε-greediness, with ε-possibly annealed towards zero over the training. (Other options: noisy nets, entropy regularization,...)
- These are rather ad-hoc approaches: the setting of hyperparameters (annealing rate, entropy coefficient) is often highly domain-dependent.
- EvE dilemma also appears in domains that are not typically modeled as MDPs: recommender systems, disease treatment plans, or games.
- The EvE dilemma is systematically studied using the formalism of multi-armed bandits (MABs).

One-armed bandit

Multi-armed bandit (MAB)



A MAB is given by:

- a finite set of arms A;
- for each arm a a reward distribution D_a with mean $\mu(a)$
- the D_a and $\mu(a)$ are unknown to the player!

For simplicity, we will assume that rewards are from the interval [0,1].

Dynamics of multi-armed bandits

- Interaction proceeds in discrete time steps $1, 2, 3, \ldots, T$, where T is a termination time which might or might not be known to the player.
- In each time step t, we choose some arm $a_t \in A$ to pull and receive reward $r_t \sim D_{a_t}$.

= like a one-state MDP with stochastic rewards, but we do not known the reward distributions D_i in advance.

Our goal is to maximize the expected accumulated reward $\mathbb{E}[\sum_{t=1}^{T} r_t] = \sum_{t=1}^{T} \mathbb{E}[r_t]$.

Clearly, the expected reward is maximized by pulling, in each step t, the arm a^* with maximal mean reward μ^* (we assume all arms have different mean rewards):

$$\mu^* = \max_{a \in A} \mu(a)$$

$$a^* = \arg\max_{a \in A} \mu(a)$$

However, since we do not know D_a's and $\mu(a)$'s, we cannot a priori determine which arm is optimal! We need to learn something about the reward distributions by exploring individual arms, while trying to maximize the accumulated returns.

Definition 58: MAB policy

A policy in a multi-armed bandit problem is a function π which to each history $a_1, r_1, a_2, r_2, \ldots, a_t, r_t$ of actions and resulting rewards assigns a distribution over arms.

The policies we will work with typically based their decisions on the following statistics of the history: for each arm a we keep:

- $N_t(a)$ – the number of times the arm a was pulled by time t
- $R_t(a) = \sum_{1 \leq i \leq t} r_t \cdot \mathbb{I}(a_i = a)$ – the total reward accumulated by pulling arm a up to time t
- $\hat{\mu}_t(a) = \frac{R_t(a)}{N_t(a)}$ – the empirical mean return of arm a

Node that if $N_t(a) \to \infty$ as $t \to \infty$, then $\hat{\mu}_t(a) \to \mu(a)$.

Example: ε-greedy policy

Given $\varepsilon \in [0, 1]$, an ε-greedy policy selects arm a_{t+1}, for any $t \geq 0$, as follows:

- with probability ε it selects an arm uniformly at random
- with probability $1 - \varepsilon$ it selects an arm a such that

$$a = \arg\max_{b \in A} \hat{\mu}_t(b)$$

Intuitively, this is sub-optimal policy: even for large t, when all $\hat{\mu}_t(b)$ should be relatively good approximations of true mean rewards, the policy still selects sub-optimal arms at a constant rate. How to formalize this issue?

Regret

The regret of a policy at time T is the difference between the expected return of the policy and the return we would get by always pulling the optimal arm. Formally

Definition 59: Regret

The regret of a policy π at time T is the quantity

$$Regret_{\pi}(T) = T \cdot \mu^* - \mathbb{E}^{\pi}[\sum_{t=1}^{T} r_t].$$

Another form of writing the regret:

$$Regret_{\pi}(T) = T \cdot \mu^* - \sum_{a \in A} \mu(a) \cdot \mathbb{E}^{\pi}[N_T(a)].$$

Lower bound on the regret of ε-greedy policy

We will show that ε-greedy policy has regret linear in T, i.e. asymptotically the worst possible one.

Let b be any sub-optimal action. Denote $\Delta_b = \mu^* - \mu_b > 0$.

Since b has probability at least ε of being pulled in every step, the expected number of times b is pulled in T steps is at least $T \cdot \varepsilon$.

Hence,

$$\begin{aligned} \mathit{Regret}_{\varepsilon\text{-greedy}}(T) &= T \cdot \mu^* - \sum_{a \in A} \mu(a) \cdot \mathbb{E}^{\varepsilon\text{-greedy}}[N_T(a)] \\ &= \sum_{a \in A} \mathbb{E}^{\varepsilon\text{-greedy}}[N_T(a)] \cdot \mu^* - \sum_{a \in A} \mu(a) \cdot \mathbb{E}^{\varepsilon\text{-greedy}}[N_T(a)] \\ &= \sum_{a \in A} \mathbb{E}^{\varepsilon\text{-greedy}}[N_T(a)] \cdot (\mu^* - \mu(a)) \geq \mathbb{E}^{\varepsilon\text{-greedy}}[N_T(a)] \cdot \Delta_b \geq T \cdot \varepsilon \cdot \Delta_b. \end{aligned}$$

Logarithmic regret

We will now demonstrate a simple policy achieving logarithmic regret.

The policy π we will construct requires the advance knowledge of the termination time T and of the the gap between the optimal and second-best arm:

$$\Delta_{\min} = \min_{b \neq a^*} \Big(\mu^* - \mu(b) \Big)$$

Policy π proceeds in two phases:

- Phase 1 (exploration): in the first $T_1 = \lceil \frac{\log(T) \cdot |A| \cdot 4}{\Delta_{\min}^2} \rceil$ steps it deterministically cycles through all arms. I.e., if $A = \{a^1, a^2, \ldots, a^k\}$, then in step i it plays arm a^j where $j = i \pmod k + 1$.
- Phase 2 (exploitation) at timestep $T_1 + 1$, π identifies action a with maximal empirical mean $\hat{\mu}_{T_1}(a)$ and keeps playing this action for the remaining $T_2 = T - T_1$ steps.

Upper-bounding the regret

The total regret of π can be decomposed into regrets accumulated in the two phases:

$$\begin{aligned} \mathit{Regret}_\pi(T) &= T \cdot \mu^* - \sum_{t=1}^{T} \mathbb{E}^\pi[r_t] \\ &= \underbrace{T_1 \cdot \mu^* - \sum_{t=1}^{T_1} \mathbb{E}^\pi[r_t]}_{R_1,\ \text{exploration regret}} + \underbrace{T_2 \cdot \mu^* - \sum_{t=T_1+1}^{T} \mathbb{E}^\pi[r_t]}_{R_2,\ \text{exploitation regret}} \end{aligned}$$

Clearly $R_1 \leq 1 \cdot T_1 \in \mathcal{O}(\log(T))$.

R_2 depends on whether π correctly classifies the optimal arm at timestep $T_1 + 1$.

- If yes, then $R_2 = 0$.
- If no, then $R_2 \leq T_2 \leq T \in \mathcal{O}(T)$.

Bounding the probability of misclassification

We have

$$R_2 \leq T \cdot \mathbb{P}^\pi[Mis],$$

where Mis is the event that a^* does not have the maximal empirical mean after T_1 steps.

By union bound

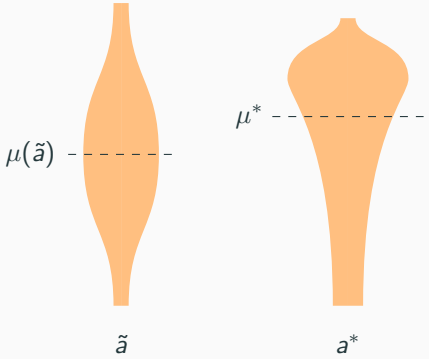
$$\mathbb{P}^\pi[Mis] \leq \sum_{b \neq a^*} \mathbb{P}^\pi[\hat{\mu}_{T_1}(b) \geq \hat{\mu}_{T_1}(a^*)].$$

Clearly, the second-best action, call it $\tilde{a}$, has the highest likelihood of achieving higher empirical mean than a^*. I.e.,

$$\mathbb{P}^\pi[Mis] \leq (|A| - 1) \cdot \mathbb{P}^\pi[\hat{\mu}_{T_1}(\tilde{a}) \geq \hat{\mu}_{T_1}(a^*)].$$

It thus suffices to bound the probability of $\tilde{a}$ overtaking a^* in empirical mean.

Probability of empirical deviation



For $\tilde{a}$ to overtake a^*, at least one of the arms must have empirical mean after T_1 steps at least $\frac{\Delta_{\min}}{2}$-away from their true mean.

Hence, it suffices to bound the probability that an empirical mean deviates too much from the true mean.

Hoeffding's inequality for large deviations

Theorem 60: Hoeffding's inequality

Let D be a distribution taking values in $[0,1]$. Let μ be the mean of D and let $\hat{\mu}_n = \sum_{i=1}^{n} x_i$, where each x_i is an independent sample from D. Then, for any $n \in \mathbb{N}^+$ and any $\delta > 0$:

$$\mathbb{P}\left[|\hat{\mu}_n - \mu| \geq \delta\right] \leq 2e^{-2n\delta^2}$$

Recall $T_1 \approx \frac{\log(T)\cdot|A|\cdot 4}{\Delta_{\min}^2}$. Then, e.g. for a^*:

$$\mathbb{P}^\pi\left[|\mu_{T_1}(a^*) - \mu^*| \geq \frac{\Delta_{\min}}{2}\right] \leq 2e^{-2\frac{T_1}{|A|}\frac{\Delta_{\min}^2}{4}} \approx 2e^{-2\log(T)} = \frac{2}{T^2}.$$

Hence, $\mathbb{P}^\pi[Mis] \leq C \cdot \frac{1}{T^2}$ for some constant C.

It follows that the exploitation regret R_2 is $\leq T \cdot \mathbb{P}^\pi[Mis] = C' \cdot \frac{1}{T}$ for some constant C'.

Final bound on regret of π

The total regret of π is:

$$Regret_\pi(T) = \underbrace{R_1(T)}_{\in \mathcal{O}(\log(T))} + \underbrace{R_2(T)}_{\in \mathcal{O}(1)} \in \mathcal{O}(\log(T)).$$

The logarithmic regret is the best possible:

Theorem 61: Lai and Robbins ("Asymptotically efficient adaptive allocation rules", 1985)

Any policy has regret in $\Omega(\log(T))$.

The disadvantage of π is that it needs to know both T and $\Delta_{\min}$ in advance.

Knowledge of T can be discarded by using ε-greedy policies with adaptive ε: if $\varepsilon_t = \min\{1, \frac{|A|}{\Delta_{\min}^2 \cdot t}\}$, then the regret is logarithmic (see David Silver's slides).

But there is actually a policy yielding logarithmic regret without the advance knowledge of either T or $\Delta_{\min}$.

Optimism in the face of uncertainty

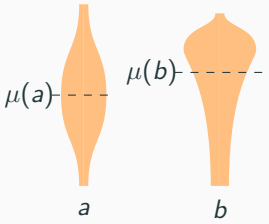
The idea is that under-explored arms should be explored more: they could be better than we currently think (and if not, exploring them should disprove that).

We will seek optimistic estimates of $\mu(a)$ in the form of upper confidence bounds: we seek an empirical quantity $U_t(a)$ such that with high probability

$$\mu(a) \leq \underbrace{\hat{\mu}_t(a) + U_t(a)}_{UCB_t(a)}.$$

The policy will always pick action maximizing UCB_t.

Moreover, $U_t(a)$ should be as tight as possible given available information. In particular, it should hold that if $N_t(a) \leq N_t(b)$, then $U_t(a) \leq U_t(b)$.



One-sided Hoeffding's inequality

Theorem 62: Hoeffding's inequality (one-sided)

Let D be a distribution taking values in $[0,1]$. Let μ be the mean of D and let $\hat{\mu}_n = \sum_{i=1}^{n} x_i$, where each x_i is an independent sample from D. Then, for any $n \in \mathbb{N}^+$ and any $\delta > 0$:

$$\mathbb{P}\left[\mu - \hat{\mu}_n \geq \delta\right] \leq e^{-2n\delta^2}.$$

Let $1 - p$ be some required confidence level. We want to find "tight" $U_t(a)$ such that $\mu^* \geq \hat{\mu}_t(a) + U_t(a)$ with probability at most p, i.e.

$$\mathbb{P}\left[\mu^* - \hat{\mu}_t(a) \geq U_t(a)\right] \leq p.$$

By (one-sided) Hoeffding:

$$\mathbb{P}\left[\mu^* - \hat{\mu}_t(a) \geq U_t(a)\right] \leq e^{-2N_t(a)U_t^2(a)}.$$

Deriving UCB formula

From the previous we have:

$$\mathbb{P}\big[\mu^* - \hat{\mu}_t(a) \geq U_t(a)\big] \leq e^{-2N_t(a)U_t^2(a)}.$$

Performing substitution $p = e^{-2N_t(a)U_t^2(a)}$ we derive the expression for $U_t(a)$:

$$U_t(a) = \sqrt{\frac{log(1/p)}{2N_t(a)}}$$

For this $U_t(a)$, it holds $\mathbb{P}[\mu^* - \mu_t(a) \geq U_t(a)] \leq p$, i.e.

$$\mathbb{P}\big[\mu^* \leq \hat{\mu}_t(a) + U_t(a)\big] \geq 1 - p.$$

We want to increase the confidence over time, i.e. p should be a function of t with $p \to 0$ as $t \to \infty$. The standard approach is to put $p = \frac{1}{t^c}$ for a suitable constant c. Then

$$U_t(a) = \sqrt{\frac{log(1/p)}{2N_t(a)}} = \sqrt{\frac{c\log(t)}{2N_t(a)}} = \sqrt{\frac{c}{2}}\sqrt{\frac{\log t}{N_t(a)}} = C \cdot \sqrt{\frac{\log t}{N_t(a)}}$$

Summary of UCB policy

In each step t, the UCB policy selects the arm with the highest upper confidence bound, i.e. arm a such that

$$a = \arg\max_{a \in A} UCB_t(a) = \arg\max_{a \in A} \left(\hat{\mu}_t(a) + C \cdot \sqrt{\frac{\log t}{N_t(a)}} \right),$$

where C is a hyperparameter known as exploration constant. It can be shown that for $[0, 1]$-valued rewards, choosing $C = \sqrt{2}$ suffices to achieve logarithmic regret

Theorem 63

When the reward distributions are over the interval $[0, 1]$, then for $C = \sqrt{2}$ it holds

$$Regret_{UCB}(T) \in \mathcal{O}(\log T).$$

Proof in optional reading: Auer, Cesa-Bianchi, Fischer: Finite-time Analysis of the Multiarmed Bandit Problem. In *Machine Learning* (47), 2002.

Concluding remarks on MABs

- UCB-like algorithms with logarithmic regret were developed also for more general cases (e.g. distributions with unbounded support, where typically some shape of the distribution or a known bound on its variance is known).
- The MAB model introduced here can be generalized in many ways (Bayesian bandits, contextual bandits, adversarial bandits) - rich area of research and applications.

Optional reading: Slivkins: Introduction to Multi-Armed Bandits (arXiv).

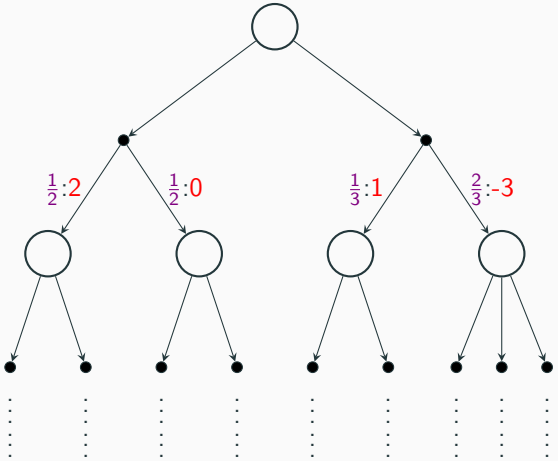
- UCB-like approaches can be used also in RL, e.g. instead of ε-greedy policies use

$$\pi(s) = \operatorname*{arg\,max}_{a \in \mathcal{A}(s)} \Big(Q(s,a) + C \cdot \sqrt{\frac{\log N(s)}{N(s,a)}} \Big).$$

- Most prominently, this idea is applied in the context of Monte-Carlo tree search algorithms.

Monte Carlo Tree Search

Tree? What tree?



- Rooted tree with alternating state nodes and action nodes.
- Action nodes equipped with rewards and probability distributions over children. We denote $p(a)$ the probability distribution over children of a.
- Policy π in the tree assigns to each state node a distribution over its children.
- For simplicity, we consider discount factor $\gamma = 1$.

Trees vs. MDPs

The trees we will consider can be considered as special type of MDPs. Due to lack of cycles, all policies can be considered memoryless in the tree.

However, the tree MDPs capture full generality of MDPs due to MDP unfolding.

Definition 64: Unfolding of MDP

For an MDP $\mathcal{M} = (\mathcal{S}, \mathcal{A}, p, r)$, the unfolding of $\mathcal{M}$ is a (possibly infinite) tree-shaped MDP $\mathit{Unfold}(\mathcal{M})$ such that:

- the states nodes of $\mathit{Unfold}(\mathcal{M})$ are the histories of $\mathcal{M}$ (=trajectory prefixes);
- if a is an action node that is a child of state node corresponding to history h, then in $\mathit{Unfold}(\mathcal{M})$ the probability of child h, a, s of a is equal to $p(s|\mathit{last}(h), a)$;
- the reward function lifted similarly: $r_{\mathit{Unfold}(\mathcal{M})}(h, a, (h, a, s)) = r_{\mathcal{M}}(\mathit{last}(h), a, s)$

Advantage of tree representation

The tree unfolding of an MDP implicitly carries, in each node, the information about the whole history up to the reaching of that node. This can be advantageous when dealing with tasks/environments where the knowledge of history is important, e.g.:

- partially observable environments (cf. Atari)
- tasks with more complex objectives than maximizing the expected discounted return
- adversarial environments (games) - requires distinguishing between player 1 and player 2 nodes (we will see later)

In what follows, we will consider trees encoding episodic MDP tasks: the tree is potentially infinite, but the probability of all infinite branches is zero. (For the sake of concreteness, imagine chess where our opponent is playing some fixed known, possibly randomized, strategy.)

MCTS requirements

The tree in which MCTS operates can be either given explicitly (if finite and small enough) or represented by some black-box model which allows for the following:

- given a state node *node*, return a list of all action-node children of *node*;
- given an action node *anode*, return a sample from $p(anode)$, i.e. sample a child of *anode* according to the probability distribution specified by the tree;
- given an action node *anode* and its child *node*, return the reward labeling the edge between the two nodes.

The black-box can have a form of a finite explicitly represented MDP, or (in turn) of the sampling-model of such an MDP (as in classical RL).

Monte Carlo tree search – high-level properties

Monte Carlo tree search (MCTS) is an online algorithm for solving (=maximizing the expected return in) tree-shaped MDPs.

Online = it does not compute a complete policy for the whole MDP; instead given the current node = (current state and the history of states and actions visited so far,) the algorithm computes the (approximately) best action to play in the current step.

Terminology: one step of MCTS = performing a computation to determine the best action given the current situation.

MCTS can be employed over multiple steps, until a whole trajectory is generated. Thus, MCTS can be seen as an algorithmic representation of a policy.

MCTS iteratively builds a structure called search tree, which is a finite sub-tree of the original (possibly infinite) MDP tree. The search tree is a global data structure shared across the steps.

MCTS master loop

Algorithm 13: MCTS master loop

Input: Tree-shaped MDP $\mathcal{M}$ with root s_0

$node \leftarrow s_0$

$\mathcal{T} \leftarrow$ tree with single node (root) s_0 and all its child action nodes

while *node is not terminal* **do**

  `BuildTree(`$\mathcal{T}$`)` `// Modifies` $\mathcal{T}$

  $a \leftarrow$ `ActualSelect(`$\mathcal{T}$`)`

  $node' \sim p(a)$

  $node \leftarrow node'$

  $\mathcal{T} \leftarrow$ sub-tree of $\mathcal{T}$ rooted in *node*

  `//` `If` *node* `not in` $\mathcal{T}$`, this is a single-node tree with` *node* `as root.`

Building the search tree

Building the search tree also involves sampling trajectories from the environment. This is standard in RL: the algorithms we have seen so far were sampling from the environment to compute policies, and the policies themselves can be seen as prescriptions for interacting with (=sampling from) the environment.

In contrast, planning/MCTS literature often differentiates between

- the actual online actions played by the algorithm when interacting with the "true" environment, i.e. outcomes of `ActualSelect` function (e.g. moving a chess piece on physical a board); and
- simulated actions performed when building the tree (e.g. imagining outcomes of various chess moves when thinking about the next move)

The reason for the distinction is that in practice, we might want the simulated actions to be performed in a virtual model of the environment (for sake of sample efficiency), while the actual actions in the "true" environment.

We will stick to the nomenclature simulated/actual action to distinguish between actions played during/at the end of individual steps.

Search tree statistics

The `BuildTree` function maintains several statistics of each node *node* of the tree (both state and action node):

- $N(node)$ – the visit count of *node* how many times has the node been visited during the run of the algorithm
- $R(node)$ – the total return achieved from that node during the run of the algorithm (counting only rewards collected on the paths from *node* to a leaf)
- $V(node) = \frac{R(node)}{V(node)}$ – the estimated value of the node (i.e., the estimate of best expected return achievable from the node; for action nodes this includes the immediate reward of that action).

Like the whole search tree $\mathcal{T}$, these statistics are global to the whole MCTS algorithm, and are typically not reset between `BuildTree` calls.

The `ActualSelect` function also uses these statistics.

Invariant: all state nodes in $\mathcal{T}$ have all there child action nodes also in $\mathcal{T}$

Building the search three: four phases

Input: Tree-shaped MDP $\mathcal{M}$ with root s_0
$node \leftarrow s_0$
$\mathcal{T} \leftarrow$ tree with single state node (root) s_0 and all its (action node) children
while *node is not terminal* **do**
    `BuildTree(`$\mathcal{T}$`)`
    $a \leftarrow$ `ActualSelect(`$\mathcal{T}$`)`
    $node' \sim p(a)$
    $node \leftarrow node'$
    $\mathcal{T} \leftarrow$ sub-tree of $\mathcal{T}$ rooted in *node*
    `// If` *node* `not in` $\mathcal{T}$`, this is a single-node tree with` *node* `as root.`

Tree building proceeds by repeatedly performing these 4 phases:

1. Search.
2. Expansion.
3. Rollout.
4. Backup.

The phases are repeated until a predetermined timeout.

Tree building

Algorithm 14: Tree building

Procedure `BuildTree`($\mathcal{T}$)**:**

  repeat

    ($node$, $anode$) $\leftarrow$ `Search`($\mathcal{T}$) `// Traverse top-down to the ''best'' node not in` $\mathcal{T}$.

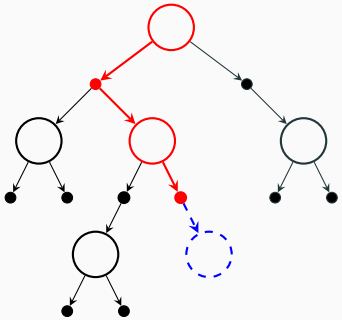
    `Expand`($\mathcal{T}$, $node$, $anode$) `// Add the discovered node to` $\mathcal{T}$.

    (R, a) $\leftarrow$ `Rollout`($node$) `// MC estimate of` $node$`'s value via default policy.`

    `Backup`($\mathcal{T}$, R, a) `// Update statistics on branch from root to` $node$.

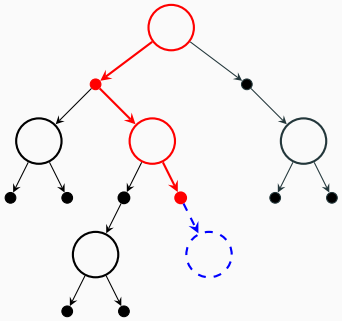
  until *timeout*

Search phase



- Traverse $\mathcal{T}$ from the root to the most promising action leaf. Then sample a state node child of this leaf (not yet in tree).
- Traverse = in each state node, select an action and in each action node, sample a successor state node from the transition function of the environment.
- Hence on each level, we also need to select "most promising" action in the current node.
- Most promising = with best value estimate, but we also take exploration/exploitation dilemma into account.

Search phase 2



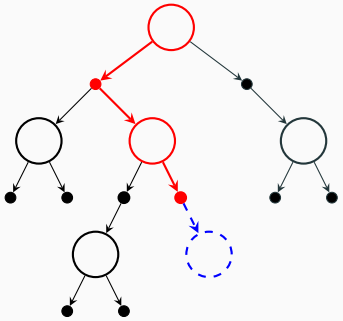
- We treat the decision in each concrete node as a separate multi-armed bandit problem. That is, when traversing a state node s, we select an action node a such that

$$a = \operatorname*{arg\,max}_{a \text{ child of } s} \left(V(a) + C \cdot \sqrt{\frac{\log N(s)}{N(a)}} \right),$$

where C is an exploration constant.

- I.e., when sampling, while in the tree we follow a UCB behavior policy. This approach is called upper confidence bound on trees (UCT), first proposed in optional reading: Kocsis, Szepesvári: *Bandit-Based Monte Carlo Planning*. In proceedings of ECML 2006.

Search phase: pseudocode



Algorithm 15: Search phase

Function `Search`($\mathcal{T}$)**:**

$node \leftarrow$ root of $\mathcal{T}$

repeat

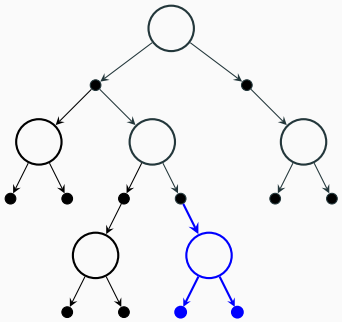
$$anode \leftarrow \underset{a \text{ child of } node}{\arg\max} \; V(a) + C \cdot \sqrt{\frac{\log N(node)}{N(a)}}$$

$node \sim p(anode)$

until *anode is a leaf*

return *node*, *anode*

Expansion phase



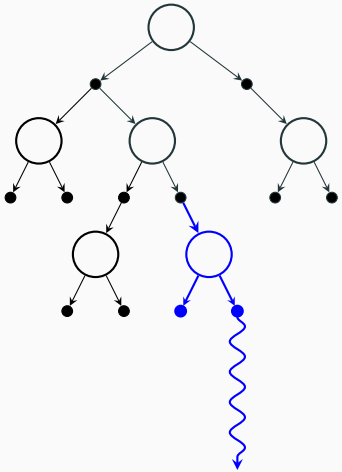
Algorithm 16: Expansion phase

Procedure `Expand`$(\mathcal{T}, node, anode)$**:**

- add *node* to $\mathcal{T}$ as a child of *anode*
- $N(node) \leftarrow 0$
- $R(node) \leftarrow 0$
- $V(node) \leftarrow 0$
- `// Add all action-node children of` *node*
- **foreach** *child a of node* **do**
  - add *a* to $\mathcal{T}$ as a child of *node*
  - $N(a) \leftarrow 0$
  - $R(a) \leftarrow 0$
  - $V(a) \leftarrow 0$

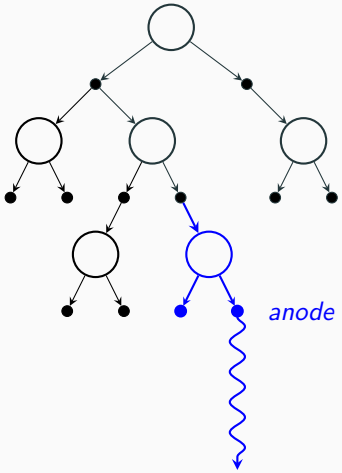
(Same initialization used in root.)

Rollout phase



- Perform a Monte Carlo sample of the rest of the trajectory from the newly added node. Record the return obtained.
- Since we are now outside of the tree, UCT cannot be used (no statistics). Instead, we follow some fixed behavior policy called default policy.
- Typical choice of default policy: in each node, select uniformly from all actions.
- Domain knowledge can be used to craft more intricate default policies.

Rollout phase: pseudocode

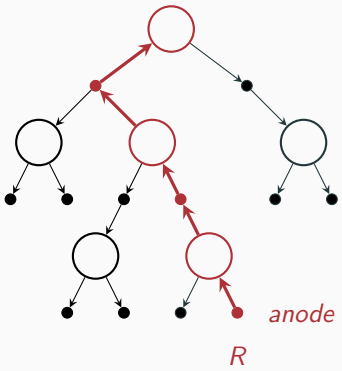


Algorithm 17: Rollout with uniform default policy

Function `Rollout(`*node*`)`**:**

- $R \leftarrow 0$
- **while** *node not terminal* **do**
  - sample a uniformly from all children of *node*
  - **if** *this is the first iteration of the loop* **then**
    - $anode \leftarrow a$
  - $node' \sim p(a)$
  - $R \leftarrow R + r(node, a, node')$
  - $node \leftarrow node'$
- **return** $R, anode$

Backup phase



For all nodes currently in $\mathcal{T}$ traversed in the previous phases, update their statistics using the rollout outcome:

Algorithm 18: Backup

Procedure `Backup`$(\mathcal{T}, \mathit{anode}, R)$**:**

- $n \leftarrow \mathit{anode}$
- **while** *true* **do**
  - $N(n) \leftarrow N(n) + 1$
  - $R(n) \leftarrow R(n) + R$
  - $V(n) \leftarrow R(n)/N(n)$
  - **if** *n is the root* **then break**
  - **if** *n is a state node* **then**
    - $R \leftarrow R +$ the reward of the edge connecting n to its parent
  - $n \leftarrow \mathit{parent}(n)$

MCTS Tree building summary

Procedure `BuildTree`($\mathcal{T}$):

  repeat

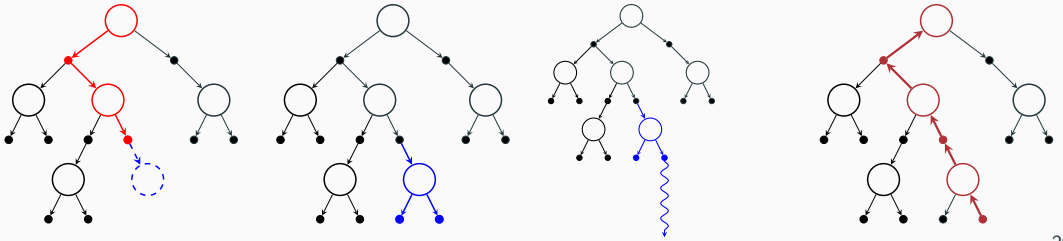
    ($node$, $anode$) $\leftarrow$ `Search`($\mathcal{T}$)

    `Expand`($\mathcal{T}$, $node$, $anode$)

    (R, a) $\leftarrow$ `Rollout`($node$)

    `Backup`($\mathcal{T}$, R, a)

  until *timeout*



MCTS: Actual action selection

Input: Tree-shaped MDP $\mathcal{M}$ with root s_0
$node \leftarrow s_0$
$\mathcal{T} \leftarrow$ tree with single state node (root) s_0 and all its (action node) children
while *node is not terminal* **do**
  `BuildTree(`$\mathcal{T}$`)`
  $a \leftarrow$ `ActualSelect(`$\mathcal{T}$`)`
  $node' \sim p(a)$
  $node \leftarrow node'$
  $\mathcal{T} \leftarrow$ sub-tree of $\mathcal{T}$ rooted in *node*
  `// If` *node* `not in` $\mathcal{T}$`, this is a single-node tree with` *node* `as root.`

- Usually just greedy according to value estimates in the root:

$$a = \operatorname*{arg\,max}_{a \text{ child of } root(\mathcal{T})} V(a)$$

.

- Alternative: according to visit count:

$$a = \operatorname*{arg\,max}_{a \text{ child of } root(\mathcal{T})} N(a), \text{ or}$$

$$a \sim softmax(N_a)_{a \text{ child of } root(\mathcal{T})}$$

(both used in AlphaZero).

MCTS with Function Approximators: AlphaZero

Towards AlphaZero

- AlphaGo Lee (2016): uses lots of domain knowledge, won 4-1 over 9-dan Go champion Lee Sedol
- AlphaGo Zero (2017): zero domain knowledge
- AlphaZero (2017-2018): general game-playing (and MDP solving) algorithm

AlphaZero literature

AlphaGo Zero:

Silver et al.: Mastering the game of Go without human knowledge. In *Nature*, vol. 550 (2017).

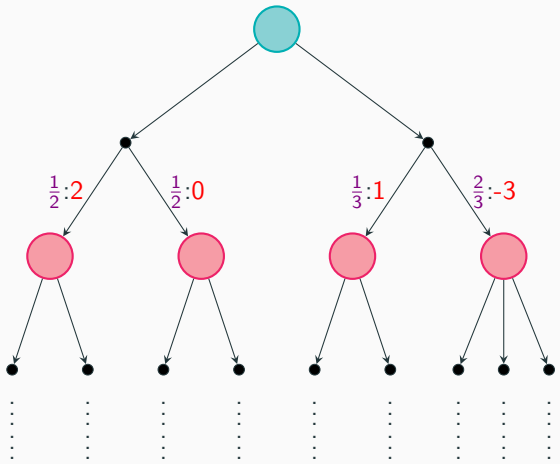
- Focuses on Go, but has a rich methodology section explaining design details.

AlphaZero:

Silver et al.: A general reinforcement learning algorithm that masters chess, shogi, and Go through self-play. In *Science*, vol. 362 (2018).

- More general, but many designed choices already explained in the AlphaGo paper not covered: need to look into code for details.

MDP vs. game trees



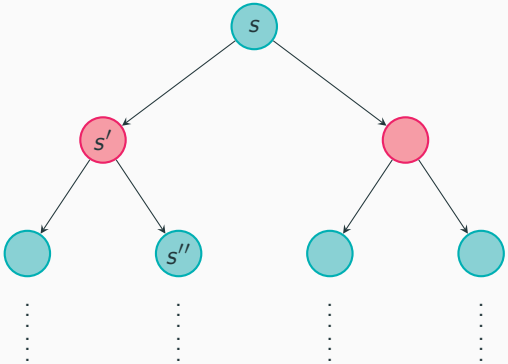
- State nodes alternate between maximizer and minimizer nodes. Maximizer/minimizer player wants to maximize/minimize the expected return achieved in its subtree.

For chess-type games, targeted by AlphaZero, simpler models suffice:

- Rewards only in terminal states (-1,0,+1).
- Deterministic action, each action node has only one successor.

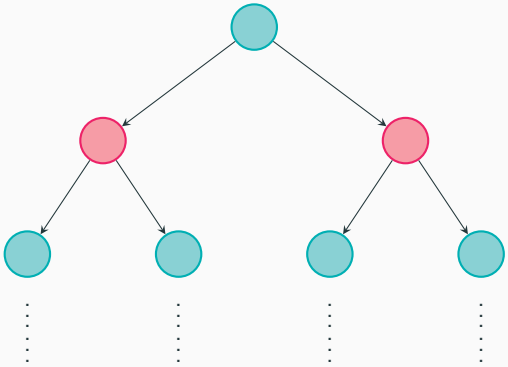
In figures, we will sometimes omit action nodes, but pseudocodes will refer to the most general formulation.

AlphaZero actual play



- We are in some state node (game position) s.
- We call AlphaZero to suggest an action to play.
- We play the action in the game and observe the next state s'.
- We wait for the oponent to play an action in his state and observe the next state s''.
- Repeat.

AlphaZero agent



Algorithm 19: AlphaZero single game

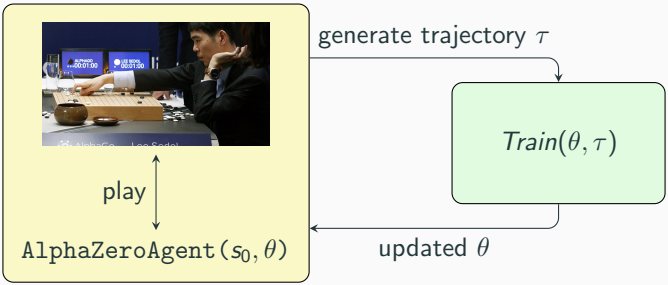
Input: Game tree with root s_0, param's θ

Function `AlphaZeroAgent`(s_0, θ)**:**

- $node \leftarrow s_0$
- $\mathcal{T} \leftarrow$ tree with single node (root) s_0 and all its child action nodes
- **while** *node is not terminal* **do**
  - `BuildTree`$(\mathcal{T}, \theta)$ `// Modifies` $\mathcal{T}$
  - $a \leftarrow$ `ActualSelect`$(\mathcal{T})$
  - $node' \sim p(a)$
  - oponent responds by playing a'
  - $node'' \sim p(a)$
  - $node \leftarrow node''$
  - $\mathcal{T} \leftarrow$ sub-tree of $\mathcal{T}$ rooted in *node*

AlphaZero = MCTS + function approximation

- The `BuildTree()` function of AlphaZero replaces rollouts with an evaluation of newly discovered nodes via a function approximator.
- The function approximator is trained on data generate by repeated plays in the actual environment (i.e., repeated calls of the `AlphaZeroAgent()` procedure.)



Slightly inaccurate AlphaZero training diagram (see later).

AlphaZero tree & approximator

Search tree of AlphaZero differs from plain MCTS in that each action node *anode* carries an additional attribute: prior probability $Pr(anode)$.

Given a state node *node*, AlphaZero's approximator predicts these quantities:

- $\pi_\theta(node) \in \mathbb{R}^{|\mathcal{A}|}$: the prior probability vector, assigning a probability to each action in *node*;
- $v_\theta(node)$: the value estimate of *node*

The training will push θ so that

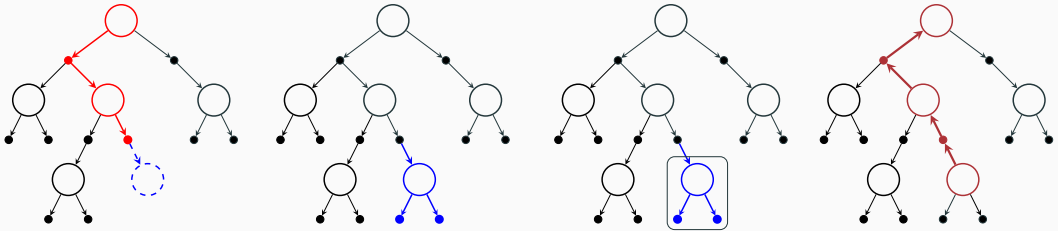
- π_θ better approximates the policy played by AlphaZero in the actual game;
- v_θ better approximates the expected return of AlphaZero in the actual game.

However, AlphaZero does not play according to π_θ: it uses the MCTS mechanism to improve upon π_θ. Hence, AlphaZero is often presented as a MCTS-based policy improvement scheme.

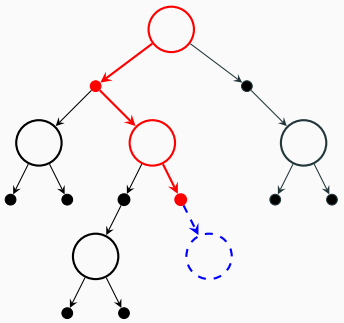
AlphaZero: Tree building

Procedure `BuildTree`$(\mathcal{T}, \theta)$**:**

- `AddNoise`$(\mathcal{T})$
- **repeat**
  - $(node, anode) \leftarrow$ `Search`$(\mathcal{T})$ `// Search phase`
  - $R \leftarrow$ `ExpandPredict`$(\mathcal{T}, node, anode, \theta)$ `// Expansion + prediction`
  - `Backup`$(\mathcal{T}, R, node)$
- **until** *timeout*



AlphaZero: Search phase (theory)



Algorithm 20: Search phase

Function `Search`($\mathcal{T}$)**:**

  repeat

    if *node is maximizer's* **then** $sgn \leftarrow 1$

    else $sgn \leftarrow -1$

    $node \leftarrow$ root of $\mathcal{T}$

    $$anode \leftarrow \underset{a \text{ child of } node}{\arg\max} \; V(a) \cdot sgn + C \cdot Pr(a) \cdot \sqrt{\frac{\log N(node)}{N(a)}}$$

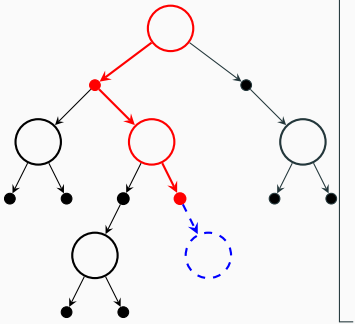
    $node \sim p(anode)$

  until *anode is a leaf*

  return *node*, *anode*

Note the self-play implemented by *sgn* and the modulation of exploration via prior probabilities.

AlphaZero: Search phase (DeepMind implementation)



Algorithm 21: Search phase

Function `Search`($\mathcal{T}$):

  repeat

    if *node is maximizer's* **then** $sgn \leftarrow 1$

    else $sgn \leftarrow -1$

    $node \leftarrow$ root of $\mathcal{T}$

    $$anode \leftarrow \underset{a \text{ child of } node}{\arg\max} \; V(a) \cdot sgn + C(node) \cdot Pr(a) \cdot \frac{\sqrt{N(node)}}{N(a)+1}$$

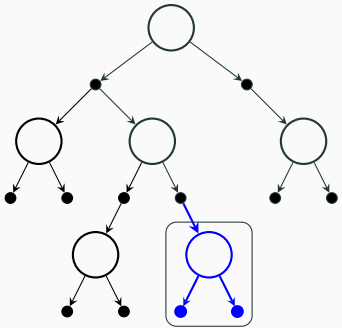
    $node \sim p(anode)$

  until *anode is a leaf*

  return *node*, *anode*

where $C(node) = \log\left(\frac{(+N(node)+c_{base}}{c_{base}}\right) + c_{init}$ with $c_{init} = 1.25$ and $c_{base} = 19652$.

AlphaZero: Expansion & Prediction



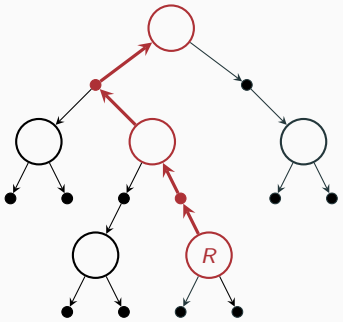
Algorithm 22: Expansion phase

Procedure `ExpandPredict`$(\mathcal{T}, node, anode, \theta)$**:**

- add $node$ to $\mathcal{T}$ as a child of $anode$
- $N(node) \leftarrow 0$
- $R(node) \leftarrow 0$
- $V(node) \leftarrow 0$
- `// Add all action-node children of` $node$
- **foreach** *child a of $node$* **do**
  - add a to $\mathcal{T}$ as a child of $node$
  - $N(a) \leftarrow 0$
  - $R(a) \leftarrow 0$
  - $V(a) \leftarrow 0$
  - $Pr(a) \leftarrow \pi_\theta(a)$
- **return** $v_\theta(node)$

(Same initialization used in root.)

AlphaZero: Backup



For all nodes currently in $\mathcal{T}$ traversed in the previous phases, update their statistics using the predicted outcome:

Algorithm 23: Backup

Procedure `Backup`$(\mathcal{T}, R, node)$**:**

- $n \leftarrow node$
- **while** *true* **do**
  - $N(n) \leftarrow N(n) + 1$
  - $R(n) \leftarrow R(n) + R$
  - $V(n) \leftarrow R(n)/N(n)$
  - **if** *n is the root* **then break**
  - **if** *n is a state node* **then**
    - $R \leftarrow R +$ the reward of the edge connecting n to its parent
  - $n \leftarrow parent(n)$

Dirichlet noise for root exploration

Procedure `BuildTree`$(\mathcal{T}, \theta)$**:**

- `AddNoise`$(\mathcal{T})$
- **repeat**
  - $(node, anode) \leftarrow$ `Search`$(\mathcal{T})$ `// Search phase`
  - $R \leftarrow$ `ExpandPredict`$(\mathcal{T}, node, anode, \theta)$ `// Expansion + prediction`
  - `Backup`$(\mathcal{T}, R, node)$
- **until** *timeout*

Before simulations start, we add Dirichlet noise to prior probabilities in the root node, encouraging exploration. (Dirichlet distribution = a "distribution over discrete distributions").

I.e., if the root has k child action nodes, we sample a vector $\nu \sim Dirichlet(\vec{\alpha})$ and for each action child a of root we perform:

$$Pr(a) \leftarrow (1 - \varepsilon) \cdot Pr(a) + \varepsilon \cdot \nu(a),$$

where $\vec{\alpha} \in \mathbb{R}^k_{>0}$ and $\varepsilon \in (0, 1)$ are hyperparameters.

AlphaZero: Actual action selection

Algorithm 24: AlphaZero single game

Input: Game tree with root s_0, param's θ

Function `AlphaZeroAgent`(s_0, θ)**:**

- $node \leftarrow s_0$
- $\mathcal{T} \leftarrow$ tree with single node (root) s_0 and all its child action nodes
- **while** *node is not terminal* **do**
  - `BuildTree`$(\mathcal{T}, \theta)$ `// Modifies` $\mathcal{T}$
  - $a \leftarrow$ `ActualSelect`$(\mathcal{T})$
  - $node' \sim p(a)$
  - oponent responds by playing a'
  - $node'' \sim p(a)$
  - $node \leftarrow node''$
  - $\mathcal{T} \leftarrow$ sub-tree of $\mathcal{T}$ rooted in *node*

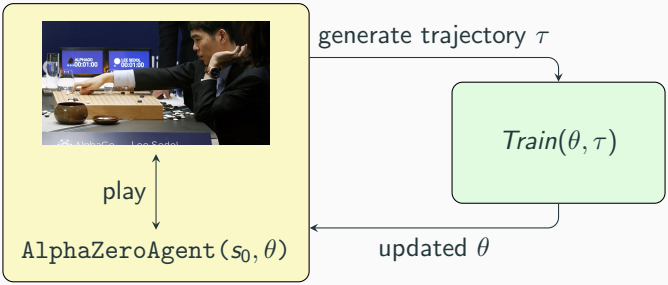
Action in actual play determined by visit count:

$$a = \underset{a \text{ child of } root(\mathcal{T})}{\arg\max} N(a), \text{ or}$$

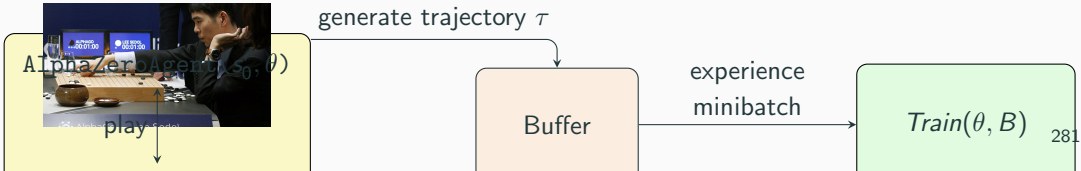
$$a \sim softmax(N_a)_{a \text{ child of } root(\mathcal{T})}.$$

The first (greedy) approach typically used in deployment (e.g. competitive play) or later in training, while softmax typically used early in training (with temperature annealed over time to decrease exploration).

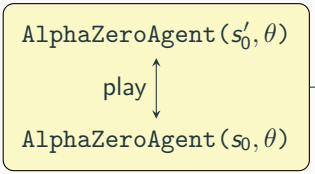
AlphaZero training



Slightly inaccurate AlphaZero training diagram (see later).



AlphaZero training details



Produces play $s_0, a_0, r_1, s_1, a_1, \ldots, r_T, s_T$
For each s_t we also store its statistics and statistics of its child action nodes.

- For each timestep t:
  - Compute total return $G_t = \sum_{i=t+1}^{T} r_i$
  - Compute the target policy π_t s.t. $\pi_t(a) = \frac{N(a)}{\sum_{b \text{ child of s}} N(b)}$
  - if s_t is maximizer state, add (s_t, π_t, G_t) to buffer
  - if s_t is minimizer state, add $(s_t, \pi_t, -G_t)$ to buffer
- Given sampled experience $e = (s, \pi, g)$, θ is updated so as to minimize the loss

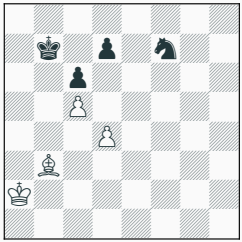
$$\mathcal{L}(\theta, e) = (v_\theta(s) - g)^2 + \sum_a \pi_t(a) \cdot \log \pi_\theta(a) + c \cdot ||\theta||^2.$$

We typically perform minibatch updates akin to DQNs (average updates over a number of sampled experiences).

State representation: chess example

The function approximator consists of feature extraction layers followed by value and policy heads. The approximator is fed certain information carried by each state node.

For chess, each node contains k last positions along its history. Each position is represented as a set of 8×8 feature planes. Most of them represent the position of pieces, with one plane for each type of pieces of a concrete player.



$$\begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix} \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix} \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

Additional planes encode castlings, repetitions, en-passant availability etc.

Action representation: chess example

The policy head outputs $4,672$ action logits from which we derive the distribution over moves by applying softmax.

Each move determined by:

- position of the moved piece ($8 \cdot 8 = 64$ possibilities)
- 73 possibilities of what to do with the piece:
  - $7 \cdot 8 = 56$ "standard moves": choice of 8 directions (N, NW, W, ..., NE) and advancing by 1–7 fields in the chosen direction
  - 8 "knight jumps": one of 8 possible L-shaped jumps
  - 9 possibilities for pawn underpromotion (knight, bishop, rook; each either via forward move or diagonal capture)

Total $64 \cdot 73 = 4,672$ moves. Illegal moves are masked out and the remaining ones renormalized.

AlphaZero approximator architecture for chess

Body + value and policy heads. Body:

- Initial convolutional layer with ReLU nonlinearity and batch normalization.
- Followed by 19 residual blocks, each block with 2 convolutional layers (again with ReLU and batch norm.) and a skip connection around them.
- All the above conv. layers apply 256 filters with 3×3 kernel size and stride 1.

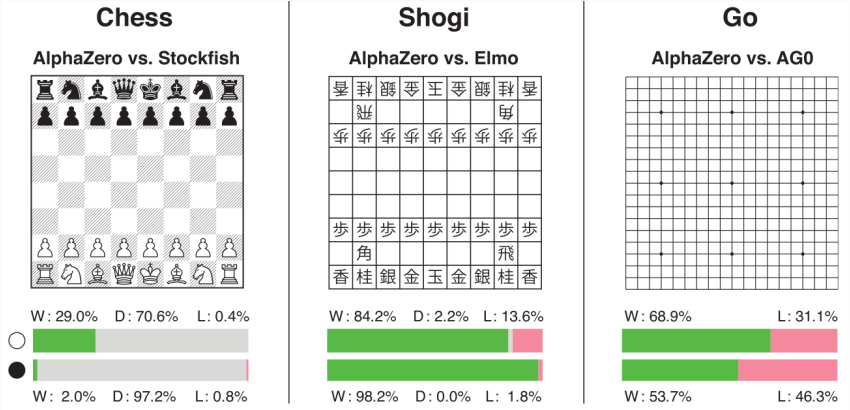
Policy head:

- Single-filter unit-kernel conv. layer with stride 1 (+ ReLU and batch norm.)
- Then ReLU with 256 neurons.
- Then fully connected layer.
- Then final tanh layer of size 1.

Policy head:

- One more conv. layer as above (inlc. ReLU and batch norm).
- Then a final conv. layer of 73 unit-kernel(?) filters.

AlphaZero experiments (from Silver et al. Science paper)



AZ trained for $700,000$ steps, $\sim$ 8 hours for chess on 5000 TPUs.

Note: this was 2018 version of Stockfish without NN evaluation. Modern version of Stockfish use NN evaluation and typically outperform open-source AlphaZero-based chess agents (e.g. LeelaChessZero).

Limitations of AlphaZero

- MCTS simulations need a (software) simulator of the environment (at least, we need to be able to perform sampling from an arbitrary given state). Such simulator might not always be available. (Actual environment too complex, dynamics unknown (Atari), ...).
- Moreover, AlphaZero is most suited for "discrete enough" domains (such as chess). Working with frame-like states of Atari is impractical. (E.g. how to check whether a given frame is already included in the search tree?)

The MuZero algorithm solves both issues by learning a deep model of the environment, including a suitable encoding of states.

Schrittwieser et al: Mastering Atari, Go, chess and shogi by planning with a learned model. In *Nature*, vol 588 (2020).

MuZero: Main idea

Original MuZero works only for deterministic environments.

The high-level structure is similar to AlphaZero. However, the nodes in MCTS simulations are formed by elements of a fixed latent space $\mathcal{LS}$ (Typically some low-dimensional vector space.)

From the actual plays, the algorithm trains the following networks:

- The representation function $g_\theta \colon \mathcal{S} \to \mathcal{LS}$
- The dynamic dunction $h_\theta \colon \mathcal{LS} \times \mathcal{A} \to \mathcal{LS} \times \mathbb{R}$
- The value and policy approximators $v_\theta \colon \mathcal{LS} \to \mathbb{R}$ and $\pi_\theta \colon \mathcal{LS} \to \mathcal{D}(\mathcal{A})$ (same role as in AlphaZero).

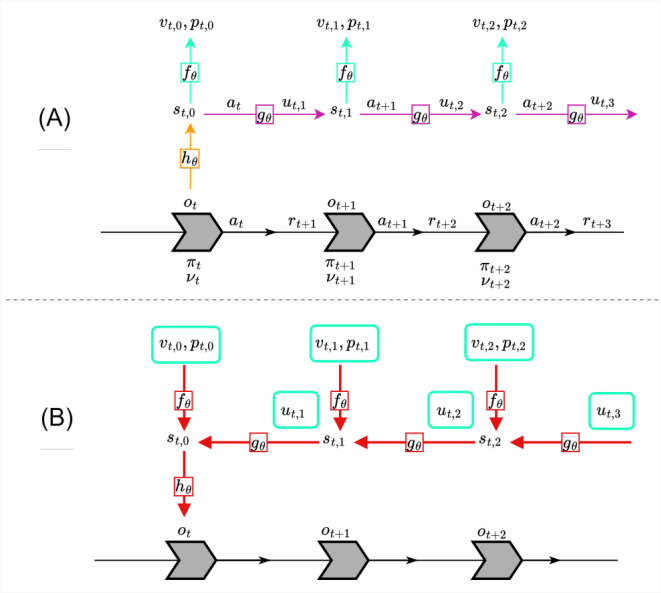
MuZero: High-level picture

Apart from training (see next slide for intuition), the "master loop" of a MuZero agent looks similar to AlphaZero. During actual play:

- In a current state, perform MCTS simulations to determine the best action.
- Play the action.
- Observe reward and new state in the actual environment.
- Repeat.

The major difference is in the "MCTS simulation" part, which is performed in the latent space.

MuZero training intuition (slide by Richard Schwarz)



MCTS in latent space

Given the current actual state s:

- Embed s into latent space to get latent state $\tilde{s} = g_\theta(s)$.
- Using the dynamics function h_θ, build the MCTS search tree from $\tilde{s}$ using the usual UCT approach. The nodes of the tree are elements of the latent space! (The approximators v_θ, π_θ are also used during the build, as in normal AlphaZero.)
- Use statistics of the root to select the best action to play in actual game.

MuZero experiments

Table 1 | Comparison of MuZero against previous agents in Atari

Agent	Median (%)	Mean (%)	Environment frames	Training time	Training steps
Ape-X[20]	434.1	1,695.6	22.8 billion	5 days	8.64 million
R2D2[19]	1,920.6	4,024.9	37.5 billion	5 days	2.16 million
MuZero	**2,041.1**	**4,999.2**	20.0 billion	12 hours	1 million
IMPALA[18]	191.8	957.6	200 million	–	–
Rainbow[36]	231.1	–	200 million	10 days	–
UNREAL[a 42]	250[a]	880[a]	250 million	–	–
LASER[37]	431	–	200 million	–	–
MuZero Reanalyze	**731.1**	**2,168.9**	200 million	12 hours	1 million

source: Schrittwieser et al: Mastering Atari, Go, chess and shogi by planning with a learned model

Further developments of MuZero and MCTS

- extension to stochastic environments:

  Antonoglou, Schrittwieser, Ozair, Hubert, Silver: *Planning in Stochastic Environments with a Learned Model.* In proceedings of ICLR 2022.

- Tricks to increase sample efficiency (e.g. GumbelZero).
- ...

MCTS has formed a basis for practical algorithms in various domains, e.g.:

- AlphaChip
- AlphaGeometry

Glimpse of the future

Google DeepMind

December 2nd, 2024

Mastering Board Games by External and Internal Planning with Language Models

John Schultz[*,1], **Jakub Adamek**[*,1], **Matej Jusup**[†,3], **Marc Lanctot**[1], **Michael Kaisers**[1], **Sarah Perrin**[1], **Daniel Hennes**[1], **Jeremy Shar**[1], **Cannada Lewis**[2], **Anian Ruoss**[1], **Tom Zahavy**[1], **Petar Veličković**[1], **Laurel Prince**[1], **Satinder Singh**[1], **Eric Malmi**[**,1] **and Nenad Tomašev**[**,1]

[*]Equal contributions, [**]Equal senior authorship, [†]Research conducted during an internship at Google, [1]Google DeepMind, [2]Google, [3]ETH Zürich

While large language models perform well on a range of complex tasks (e.g., text generation, question answering, summarization), robust multi-step planning and reasoning remains a considerable challenge for them. In this paper we show that search-based planning can significantly improve LLMs' playing strength across several board games (Chess, Fischer Random / Chess960, Connect Four, and Hex). We introduce, compare and contrast two major approaches: In *external search*, the model guides Monte Carlo Tree Search (MCTS) rollouts and evaluations without calls to an external engine, and in *internal search*, the model directly generates in-context a linearized tree of potential futures and a resulting final choice. Both build on a language model pre-trained on relevant domain knowledge, capturing the transition and value functions across these games. We find that our pre-training method minimizes hallucinations, as our model is highly accurate regarding state prediction and legal moves. Additionally, both internal and external search indeed improve win-rates against state-of-the-art bots, even reaching Grandmaster-level performance in chess while operating on a similar move count search budget per decision as human Grandmasters. The way we combine search with domain knowledge is not specific to board games, suggesting direct extensions into more general language model inference and training techniques.

Keywords: Search, planning, language models, games, chess.