Section overview

Motivation

1. Motivation
   - Autonomous Systems
   - Supervised learning
   - Unsupervised learning
   - Reinforcement Learning
2. Reinforcement Learning 101
3. Lets go Markov
4. Markov Decision Process
5. Dynamic Programming
6. Model-Free Learning
7. Model-Free Control
8. Closing thoughts

Autonomous Systems

Definition (Autonomous Systems)

Autonomous systems operate independently from other systems by having their own decision making process.

Example

- Robots & some drones
- Game AIs & some social network bots
- Everything with neurons (humans and animals).

3 flavours of Machine Learning

Machine Learning operates on 3 fundamental areas

- Supervised learning
- Unsupervised learning
- Reinforcement learning

Definition (Supervised learning)

Learn an unknown mapping $f(x) = y$ from training data given in the form of input x and output y pairs.

The training typically involves iteratively minimising the error between predicted and actual output.
The aim is that the learned mapping f performs well, generalises, to other input-output pairs that the system was not trained on.
Output y can be categorical or numerical, the task is then respectively classification or regression/function approximation.

Example

- x =blood test, y = cancer diagnosis
- x =weather today, y =temperature tomorrow
- x =photos of people, y = identity of person

Definition (Unsupervised learning)

Discover an underlying/"hidden"/"latent" structure within data x.

The aim is to find an underlying structure that explains the data x in a more efficient way.
We can think of this as reducing the amount of bits required to store the "important" features of the data set, akin to lossy data compression.
This is possible in broadly 2 ways:

- by reducing the dimensions in the data, dimensionality reduction
- by discretising the data, clustering.

Example

- x =height and weight of people
- x =photos of people
- x =all webpages

Reinforcement Learning (RL)

What makes reinforcement learning different from other machine learning paradigms?

- There is no supervisor signal (i.e. pairs of input and output data), only a reward signal.
- Feedback is delayed, not instantaneous.
- Time really matters (data is sequential, non i.i.d data).
- Agent's actions affect the subsequent data it receives.
- The agent learns a policy, which enables the agent to maximise long-term rewards.

$\Rightarrow$ Therefore, natural choice for learning in autonomous systems.

Section overview

Reinforcement Learning 101

1. Motivation
2. Reinforcement Learning 101
   - Introduction to RL
   - Bayesian Decision Making
   - Why probabilistic reasoning?
   - Talking about Bayes
3. Lets go Markov
4. Markov Decision Process
5. Dynamic Programming
6. Model-Free Learning
7. Model-Free Control
8. Closing thoughts

Uses of Reinforcement Learning

Reinforcement learning solves problems of control, i.e. choosing the "best"/optimal action at the right time.






Learning to control our body: Long history of high jump

Olympic control policies with

Gold medal reward: High-jump

Ethel Catherwood
(Canada), 1928
gold medal

Cornelius Johnson
(USA), 1936,
gold medal

Dick Fosbury
(USA), 1968,
gold medal

History of success in reinforcement learning:

- Backgammon (Tesauro, 1994)
- Inventory Management (Van Roy, Bertsekas, Lee & Tsitsiklis, 1996)
- Dynamic Channel Allocation (e.g. Singh & Bertsekas, 1997)
- Elevator Scheduling (Crites & Barto, 1998)
- RoboCup Soccer (e.g. Stone & Veloso, 1999)
- Helicopter drone control (e.g. Ng, 2003, Abbeel & Ng, 2006)
- Drug dosage for Diabetes (e.g. Bohte et Faisal, 2013)
- Anaesthesia-Depth control in surgery (e.g. Lowery & Faisal, 2013)
- Playing Atari video games - from pixels to joystick command (DeepMind, 2015)
- Grand-master level Go playing (DeepMind, 2016)

Examples: Learning to play Go against human grand master

nature International weekly journal of science

Home | News & Comment | Research | Careers & Jobs | Current Issue | Archive | Audio & Video | For Au

Archive > Volume 529 > Issue 7587 > Articles > Article

NATURE | ARTICLE

日本語要約

Mastering the game of Go with deep neural networks and tree search

David Silver, Aja Huang, Chris J. Maddison, Arthur Guez, Laurent Sifre, George van den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, Sander Dieleman, Dominik Grewe, John Nham, Nal Kalchbrenner, Ilya Sutskever, Timothy Lillicrap, Madeleine Leach, Koray Kavukcuoglu, Thore Graepel & Demis Hassabis

Affiliations | Contributions | Corresponding authors

Nature **529**, 484–489 (28 January 2016) | doi:10.1038/nature16961
Received 11 November 2015 | Accepted 05 January 2016 | Published online 27 January 2016

PDF | Citation | Reprints | Rights & permissions | Article metrics

Abstract

Abstract · Introduction · Supervised learning of policy networks · Reinforcement learning of policy networks · Reinforcement learning of value networks · Searching with policy and value networks · Evaluating the playing strength of AlphaGo · Discussion · Methods · References · Acknowledgements · Author information · Extended data figures and tables · Supplementary information · Comments

The game of Go has long been viewed as the most challenging of classic games for artificial intelligence owing to its enormous search space and the difficulty of evaluating board positions and moves. Here we introduce a new approach to computer Go that uses 'value networks' to evaluate board positions and 'policy networks' to select moves. These deep neural networks are trained by a novel combination of supervised learning from human expert games, and reinforcement learning from games of self-play. Without any lookahead search, the neural

AI & Machine Learning @ Imperial I

The Cross-Faculty Network in Artificial Intelligence (or simply @ImperialAI) is the central hub for AI at Imperial College. We have over 800 regulars at Faculty, PostDoc, PhD student levels and is open to all students at Imperial. We form London's largest academic AI community.

- Follow us on Twitter @ImperialAI
- Signup to our email announcements https://mailman.ic.ac.uk/mailman/listinfo/ai-talks
- Visit our homepage https://www.imperial.ac.uk/ai
- AI Talks Mailing List
  - If you are interested in receiving emails regarding industry events, talks and seminars at or around Imperial College, please sign up here: https://mailman.ic.ac.uk/mailman/listinfo/ai-talks

AI & Machine Learning @ Imperial II

- Target audience: anybody interested in AI & Machine Learning

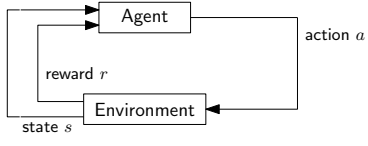
Machine Learning Tutorials

- Target audience: final-year students, PhD students, post-docs and academics with an interest in machine learning.

Machine Learning Reading Group

- The machine learning reading group is a weekly event where we discuss machine learning papers. We have not decided yet on a date for this term, but this will happen shortly.
- We have a mailing list for the reading group; feel free to sign up to be up to date regarding the next meetings and the papers that we read:
  https://mailman.ic.ac.uk/mailman/listinfo/ml-reading-group
- Target audience: PhD students, post-docs, academics

Reinforcement Learning Framework



- Agent interacts with environment to gain knowledge
- Explores and receives rewards
- Actions change the state of the environment
- Choose actions to maximize long-term reward

Ground work needed for Reinforcement Learning

We understand: Control is sequential decision making. Optimal control is sequential decision making so as to minimise a cost or maximise a reward ("$reward = -cost$").
Reinforcement Learning involves learning an optimal control of an "a priori" unknown system, unknown environment with unknown rewards – only experience will teach us.

Probability theory refresher

Example (London weather)

$A \in Rain, Sun$ and $B \in Windy, Calm$ $p(A\ B)$ is shorthand for probability that events A and B occur simultaneously.

$$P = \begin{pmatrix} 0.1 & 0.4 & Rain \\ 0.25 & 0.25 & Sun \\ Windy & Calm & \end{pmatrix} \quad (1)$$

All possible combination of event probabilities must sum to 1: $\sum_A \sum_B P(A\ B) = 1$.

1. What is the probability of rain $P('Rain')$? (marginalisation)
2. What is the probability of raining if it is a windy day $P('Rain'|'Windy')$?(conditioning)

Basic probability theory

Definition (Bayes Theorem)

$p(AB) = p(A|B) \times p(B) = p(B|A) \times p(A)$

Example (Clinical screening)

$p(AB)$ is the joint probability of blood test outcomes and a person's disease state.
$P(A) = P(\text{Person has disease}) = 1\%$ of population
$P(B) = P(\text{Person has positive blood test}) = 10\%$ of population
$P(B|A) = P(\text{blood test is positive given that person is ill}) = 70\%$
$P(A|B) = P(\text{person is ill given that blood test was positive}) = ?$

Probabilistic Reasoning as Extension of Basic Logic: Boole vs Bayes

- Bayesian inference is a simple mathematical theory which characterises plausible reasoning in the presence of uncertainties. Think about learning as the reduction of uncertainty about what we want to know.
- However, classic Boolean logic excludes certain forms of "plausible reasoning" (real-world):
  We observe that A is false. We find B becomes less plausible... although no conclusion can be drawn from classical logic. We observe that B is true. It seems A becomes more plausible.
- We use this form of reasoning daily: Our friend is late. Was she H1 abducted by aliens, H2 abducted by kidnappers or H3 delayed by traffic. How do we conclude H3 is the most plausible answer?

  Hint: "Probability theory is in fact only common sense reduced to calculus" Pierre Simon Laplace (1749-1827).

Probabilistic Reasoning: Plausibilities

"For plausible reasoning It is necessary to extend the discrete true and false values of truth to continuous plausibilities."[1]
E.T. Jaynes (1922-1998) identified 3 mathematical criteria which must apply to all plausibilities:

1. The degrees of plausibility are represented by real numbers.
2. These numbers must be based on the rules of common sense.
   a. Consistency or non-contradiction: when the same result can be reached through different means, the same plausibility value must be found in all cases.
   b. Honesty: All available data must be taken into account.
   c. Reproducibility: Equal levels of knowledge must have the same degree of plausibility.

[1] E. T. Jaynes in "Probability theory: The logic of science" (2003) Cambridge University Press.

Probabilistic Reasoning: Probabilities

- The Cox-Jaynes's theorem proves these plausibilities to be sufficient to define the universal mathematical rules which apply to plausibility p, up to an arbitrary monotonic function. Crucially, these rules **are** the rules of probability.
- If consistent rules for processing plausibility are the result of the evolution of the human brain (or any another autonomous agent), then they must approximate the standard rules of probability. Otherwise one can be systematically exploited to be at a loss ("Dutch Book").
- Important: these probabilities are no longer interpreted as the relative frequency of events (Frequentist interpretation), but rather as measurements of the degree of subjective knowledge or belief (Bayesian interpretation).

Bayesian talk

In the context of Bayesian or statistical machine learning we often want to learn unknown parameters of a model, therefore we want to determine the probability of the parameters given the data we observed, i.e. the posterior probability:

$$p(Parameters|Data) = \frac{p(Data|Parameters)p(Parameters)}{p(Data)}$$

we call $p(Data|Parameters)$ the likelihood or evidence and $p(Parameters)$ our prior belief over the possible parameter values.

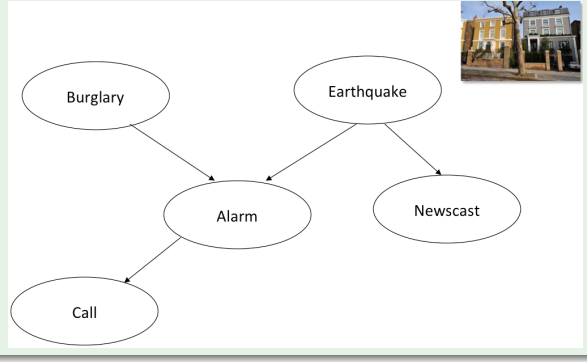
Graphical Model 1

Example (Mr Holmes's conundrum)

- Mr. Holmes is on holiday.
- He receives a call from his neighbor that the alarm of his house went off.
- He thinks that somebody broke into his house.
- Afterwards he hears an announcement on radio that a small earthquake just happened
- Since the alarm has been going off during an earthquake,
- He concludes it is more likely that earthquake causes the alarm (explaining away the burglary).

Graphical Model 2

Example (Mr Holmes's conundrum)



Structure of dependencies between random variables visualised as directed edges for conditional probabilities ($P[target|source]$) from source to target nodes and nodes with pure source nodes reflecting prior probabilities ($P[source]$). The Probabilistic Graphical model is a graphical representation of factorised joint probability distribution.

Benefits of Being Bayesian

- We can make inferences based on uncertain information: "Is the barking dog dangerous?"
- Probabilities used to represent degrees of belief
  Strength of a belief is given a value between 0 and 1
  e.g. our belief in proposition A ("dog is going to bite") being true is $P(A) = 0.95$
- Principled decision making: Is
  $P(A) \times$ Cost of A $>$ $P(\neg A) \times$ Cost of $\neg A$?
  $\Rightarrow$ We can select optimal actions based on probabilistic inference.

Bayesian Decision Theory

- Make optimal decisions a^* by maximizing an expected utility

$$a^* \in \arg\max_a \mathbb{E}\left[r(a)\right] = \arg\max_a \sum_{j=1}^{M} r(s_j, a) p(s_j)$$

a : decision/action

s : information about environment/state indexed from 1 to M.

Examples

- Reinforcement Learning (computer science, neuroscience & psychology)
- Optimal control theory (engineering, robotics)
- Bayesian sequential decision theory (statistics, cryptography)

Section overview

Lets go Markov

1. Motivation
2. Reinforcement Learning 101
3. Lets go Markov
   - Markov Process
   - Markov Reward Process
   - From state to action: Policy
4. Markov Decision Process
5. Dynamic Programming
6. Model-Free Learning
7. Model-Free Control
8. Closing thoughts

Markov Process

Definition

A Markov Process is a tuple $(\mathcal{S}, \mathcal{P})$,
$\mathcal{S}$ is a set of states
$\mathcal{P}_{ss'}$ is a state transition probability matrix,

$$\mathcal{P}_{ss'} = P\left[S_{t+1} = s' | S_t = s\right] \tag{2}$$

A Markov process generates a chain of Markov states governed by probabilistic transitions.

Markov Property

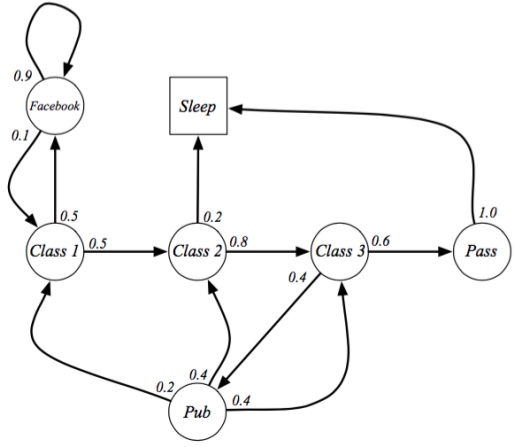
Definition

A state s_t is Markov if and only if $P[s_{t+1}|s_t] = P[s_{t+1}|s_1, \ldots s_t]$

The future is independent of the past given the present.

- The present state s_t captures all information the history of the agents events.
- Once the state is known, the any data of the history is no longer needed.

Example: University life Markov Process



Round states (transient states), Square states (terminal states).
[2]

[2]Example by David Silver (UCL)

State Transition Probabilities

For a Markov state s and successor state s', the state transition probability is defined by $\mathcal{P}_{ss'} = P\left[s_{t+1} = s' | s_t = s\right]$. The state transition matrix defines transition probabilities from all states s to all successor states s'.
Because all transition probabilities have to be accounted for to give a total probability of 1, we have $\sum_{s'} \mathcal{P}_{ss'} = 1$ (we need to end up somewhere after leaving s, including returning to s). We choose the matrix row notation so that the rows have to sum to one.

Definition (Stationarity)

If the $P\left[s_{t+1} | s_t\right]$ do not depend on t, but only on the origin and destination states, we say the Markov chain is stationary or homogenous.

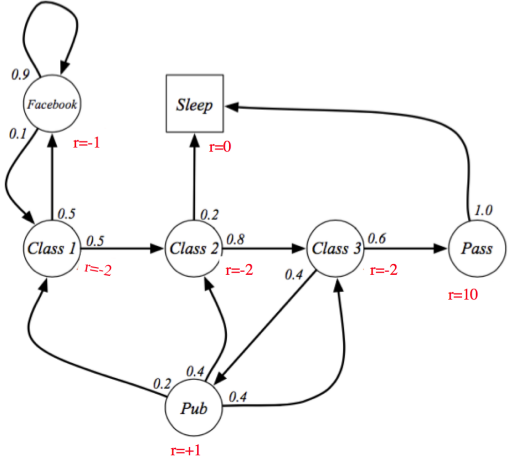
Markov Reward Process

A Markov Reward Process (MRP) is a Markov chain which emits rewards.

Definition (Markov Reward Process)

A Markov Reward Process is a tuple $(\mathcal{S}, \mathcal{P}, \mathcal{R}, \gamma)$
$\mathcal{S}$ is a set of states
$\mathcal{P}_{ss'}$ is a state transition probability matrix
$\mathcal{R}_s = \mathbb{E}[r_{t+1}|S_t = s]$ is an expected immediate reward that we collect upon departing state s, this reward collection occurs at time step $t+1$
$\gamma \in [0, 1]$ is a discount factor.

Example: University life Markov Reward Process



What do samples from this process look like?

Return

Definition (Return)

The return R_t is the total discounted reward from time-step t.

$$R_t = r_{t+1} + \gamma r_{t+2} + \cdots = \sum_{k=0}^{\infty} \gamma^k r_{t+k+1} \tag{3}$$

The factor $\gamma \in [0, 1]$ is how we discount the present value of future rewards.
The value of receiving reward r after $k+1$ time-steps is $\gamma^k r$.
The discount values immediate reward higher than delayed reward:

- γ close to 0 leads to "myopic" (short-sighted) evaluation.
- γ close to 1 leads to "far-sighted" evaluation.

University life MRP returns

Sample returns for Student MRP:
Starting from $S_1 = C1$ with $\gamma = \frac{1}{2}$

$$R_1 = r_2 + \gamma r_3 + \cdots + \gamma^{T-2} r_T$$

T is for the time it takes to reach the terminal state.

Example (Sample Runs)

C1 C2 C3 Pass Sleep	$R_1 = -2 + \frac{1}{2} \times -2 + \frac{1}{2}^2 \times -2 + \frac{1}{2}^3 \times 10$
C1 FB FB C1 C2 Sleep	$R_1 =$
...	$-2 + \frac{1}{2} \times -1 + \frac{1}{2}^2 \times -1 + \frac{1}{2}^3 \times -2 + \frac{1}{2}^4 \times -2$
C1 FB FB FB ...	$R_1 = -2 + \frac{1}{2} \times -1 + \frac{1}{2}^2 \times -1 + \frac{1}{2}^3 \times -1 + \ldots$

What is the value of being in C1, C2, C3?

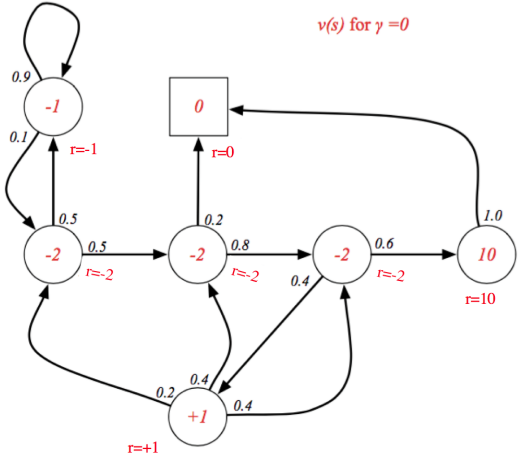
Why discounting is a good idea?

Most Markov reward processes are discounted with a $\gamma < 1$. Why?

- Mathematically convenient to discount rewards
- Avoids infinite returns in cyclic or infinite processes
- Uncertainty about the future may not be fully represented
- If the reward is financial, immediate rewards may earn more interest than delayed rewards
- Human and animal decision making shows preference for immediate reward
- It is sometimes useful to adopt undiscounted processes (i.e. $\gamma = 1$), e.g. if all sequences terminate and also when sequences are equally long (why?).

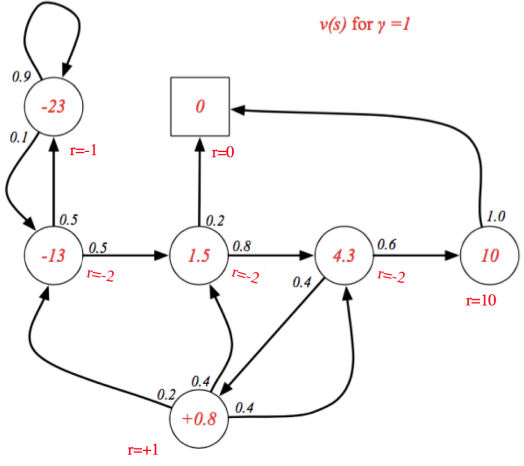
Myopic MRP value function: $\gamma = 0$

What is the value of being in a state?



Immediate reward and total return match, but what if $\gamma > 0$?

Far sighted MRP value function $\gamma = 1$ I



Far sighted MRP value function $\gamma = 1$ II

- Check for self-consistency $v(\text{C3}) =?$
  Using Bellman Equation:
  $v(\text{C3}) = -2 + 1 \times 0.6 \times 10 + 1 \times 0.4 \times 0.8 = 4.32 \approx 4.3$
  There are cycles in the graph, why is the value of some states not infinite?

  This is self-consistent (numbers in figure are rounded to one digit for space reasons)

- Check for self-consistency $v(\text{C1}) =?$
  Using Bellman Equation:
  $v(\text{C1}) = -2 + 1 \times 0.5 \times -23 + 1 \times 0.5 \times 1.5 = -12.75 \approx -13$
  This is self-consistent (numbers in figure are rounded to one digit for space reasons)

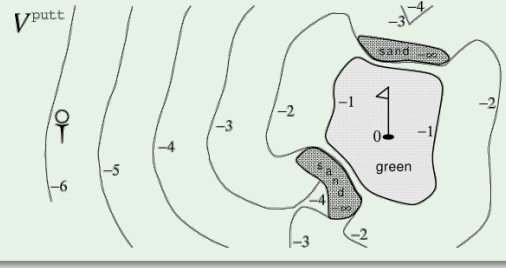
State Value Function

Definition (State value function)

The state value function $v(s)$ of an MRP is the expected return R starting from state s at time t.

$$v(s) = \mathbb{E}\left[R_t | S_t = s\right] \tag{4}$$

Example (Golf)



Bellman Equation for MRPs

$$
\begin{aligned}
v(s) &= \mathbb{E}\left[R_t | S_t = s\right] && (5)\\
&= \mathbb{E}\left[r_{t+1} + \gamma r_{t+2} + \gamma^2 r_{t+3} + \ldots | S_t = s\right] && (6)\\
&= \mathbb{E}\left[r_{t+1} + \gamma(r_{t+2} + \gamma r_{t+3} + \ldots) | S_t = s\right] && (7)\\
&= \mathbb{E}\left[r_{t+1} + \gamma R_{t+1} | S_t = s\right] && (8)\\
&= \mathbb{E}\left[r_{t+1} + \gamma v(S_{t+1}) | S_t = s\right] && (9)
\end{aligned}
$$

Value equation decomposes into 2 terms:

- Immediate reward r_{t+1}
- Discounted return of successor state $\gamma v(S_{t+1})$

Note for the mathematically orthodox: Between the two last derivation steps we sweeped under the carpet that we are using the Tower rule/Law of Total expectation to replace the expected return for state with that of the successor state, which is beyond

Forms of the Bellman Equation for MRPs

- Expectation notation:

$$v(s) = \mathbb{E}\left[R_t | S_t = s\right]$$

- Sum notation (expectation written out):

$$v(s) = \mathcal{R}_s + \gamma \sum_{s' \in \mathcal{S}} \mathcal{P}_{ss'} v(s') \tag{10}$$

  We have n of these equations, one for each state.

- Vector notation:

$$\mathbf{v} = \mathcal{R} + \gamma \mathcal{P} \mathbf{v} \tag{11}$$

  The vector $\mathbf{v}$ is n-dimensional.

Direct solution

The Bellman equation is a linear, self-consistent equation:

$$\mathbf{v} = \mathcal{R} + \gamma \mathcal{P}\mathbf{v}$$

we can solve for it directly:

$$\begin{aligned} \mathbf{v} &= \mathcal{R} + \gamma \mathcal{P}\mathbf{v} && (12) \\ (\mathbb{1} - \gamma \mathcal{P})\mathbf{v} &= \mathcal{R} && (13) \\ \mathbf{v} &= (\mathbb{1} - \gamma \mathcal{P})^{-1}\mathcal{R} && (14) \end{aligned}$$

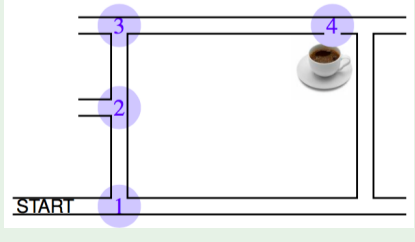
Matrix inversion is computational expensive at $\mathcal{O}(n^3)$ for n states (e.g. Backgammon has 10^{20} states), so direct solution only feasible for small MRPs. Fortunately there are many iterative methods for solving large MRPs:

- Dynamic programming
- Monte-Carlo evaluation
- Temporal-Difference learning

These are at the core of Reinforcement learning, we will learn all 3 algorithms. By the way you have met the solution of self-consistent equations before, whenever you solved for a set of n linear equations in n unknowns you exploited that the equations and the unknowns had to be self-consistent (i.e. related to each other by the common structure of the problem).

From state to action: policy

Example (Coffee Process)

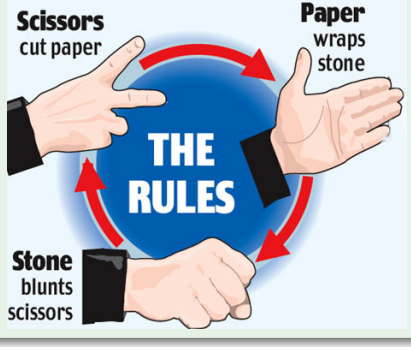


States are $1, 2, 3$ and 4. What to do where?

$$\begin{aligned} \pi(1) &= \text{turn left and walk} \\ \pi(2) &= \text{go straight and walk} \\ \pi(3) &= \text{turn right and walk} \\ \pi(4) &= \text{turn right and walk} \end{aligned}$$

Rock, Paper & Scissor Process

Example (Rock, Paper & Scissor Process)



Following a rigid policy can be disadvantageous and exposes the agent to being systematically exploited (see "Dutch Book" argument).

Definition (Policy)

A policy $\pi_t(a, s) = P[A_t = a | S_t = s]$ is the conditional probability distribution to execute an action $a \in \mathcal{A}$ given that one is in state $s \in \mathcal{S}$ at time t.

The general form of the policy is called a probabilistic or stochastic policy, so π is a probability. If for a given state s only a single a is possible, then the policy is deterministic: $\pi(a, s) = 1$ and $\pi(a', s) = 0, \forall a \neq a'$. A shorthand is to write $\pi_t(s) = a$, implying that the function π returns an action for a given state.
Now we "only" need to work out how to choose an action ...

Lottery decision making

Example

Optimal decision maximises our expected return

Actions	Reward
a_1: play	s_1: Win the lottery
a_2: save	s_2: Loose the lottery

$$a^* = \arg\max_{a_i} \sum_{j=1}^{2} \mathcal{R}_{s_j}^{a_i} P\left[s_j | a_i\right] \qquad (15)$$

Were a_i are the actions available in state s_i, i.e. $a_i \in \mathcal{A}(s_i)$.

$$\begin{array}{ll} P\left[s_1|a_1\right] = 10^{-7} & \mathcal{R}_1^1 = 500,000 \text{ USD} \\ P\left[s_2|a_1\right] = 1 - 10^{-7} & \mathcal{R}_1^2 = -1 \text{ USD} \\ P\left[s_1|a_2\right] = 0 & \mathcal{R}_2^1 = 0 \text{ USD} \\ P\left[s_2|a_2\right] = 1 & \mathcal{R}_2^2 = 0 \text{ USD} \end{array}$$

What is the optimal action for this decision problem?

Section overview

Markov Decision Process

1. Motivation
2. Reinforcement Learning 101
3. Lets go Markov
4. Markov Decision Process
   - Value function
   - Policy Evaluation
   - Optimal value function
   - Bellmann Optimality
5. Dynamic Programming
6. Model-Free Learning
7. Model-Free Control
8. Closing thoughts

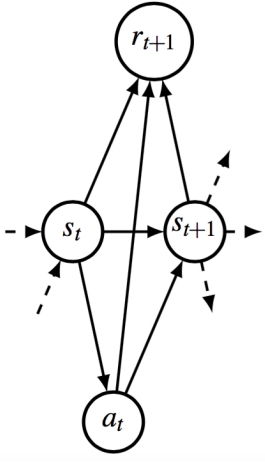
Markov Decision Process (MDP)

Definition (Markov Decision Process)

- $\mathcal{S}$: State space
- $\mathcal{A}$: Action space
- $\mathcal{P}^{a}_{ss'}$: Transition probability $p(s_{t+1}|s_t, a_t)$
- $\gamma \in [0, 1]$: Discount factor
- $\mathcal{R}^{a}_{ss'} = r(s, a, s')$ immediate/instantaneous reward function. In temporal notation[a] $r_{t+1} = r(s_{t+1}, s_t, a_t)$
- π: Policy
  - Stochastic: $\boldsymbol{a} \sim p_\pi(\boldsymbol{a}|\boldsymbol{s})$
    alternative notations:
    $p_\pi(\boldsymbol{a}|\boldsymbol{s}) = \pi(\boldsymbol{a}|\boldsymbol{s}) \equiv \pi(a, s)$ (probability)
  - Deterministic: $\boldsymbol{a} = \pi(\boldsymbol{s})$ (indicator function)

[a] This means the reward is collected upon the transition from s_t to s_{t+1}, which we determine to occur at time point $t + 1$.

Graphical model of an MDP



Circles denote variables, edges denote conditional dependences between variables.

Value function for MDPs

How good is it to be in a given state?

Definition (Value function)

$$V^{\pi}(s) = \mathbb{E}_{\pi}\left[R_t | S_t = s\right] = E\left[\sum_{k=0}^{\infty} \gamma^t r_{t+k+1} | S_t = s\right] \tag{16}$$

where R_t is a discounted total return and the r_{t+k+1} are immediate rewards.

We will see how the value function is a self-consistent linear equation, in analogy to what we derived for the MRP.

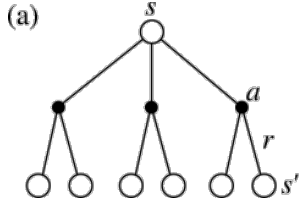
Value function self-consistency

$$
\begin{aligned}
V^{\pi}(s) &= \mathbb{E}_{\pi}\left[R_t | S_t{=}s\right] \\
&= \mathbb{E}_{\pi}\left[\sum_{k=0}^{\infty} \gamma^k r_{t+k+1} | S_t{=}s\right] \\
&= \mathbb{E}_{\pi}\left[r_{t+1} + \gamma \sum_{k=0}^{\infty} \gamma^k r_{t+k+2} | S_t{=}s\right] \\
&= \mathbb{E}\left[r_{t+1} | S_t{=}s\right] + \gamma \mathbb{E}\left[\sum_{k=0}^{\infty} \gamma^k r_{t+k+2} | S_t{=}s\right]
\end{aligned}
$$

To understand the next step better, let us unpack the meaning of the two expectation terms $\mathbb{E}\left[\ldots\right]$.

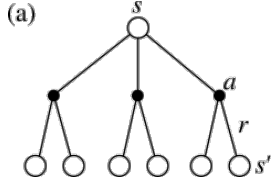
Backup diagrams

Figure (a) shows a backup diagram: We use the diagram to trace the paths from state s to successor states s' given that we chose an action a. We use these to transfer state value information **back** to a state s from its successor states $'s$. This is the update or backup operation that forms the heart of reinforcement learning methods,



Backups and value function unpacking

We can use the intuition from this diagram, to "**backup**" the current state value from successor states. To be able to calculate $v(s)$ we only need to average over all possible traces and their rewards.



- the chosen action a (given by policy $P[a|s] = \pi(a, s)$)
- the probability of transition to s' (give by transition probability $P[s'|s, a] = \mathcal{P}^a_{ss'}$)
- the instantaneous reward r (given by reward function $r(s, a, s') = \mathcal{R}^a_{ss'}$)
- the value of state s' (hint this is actually $v(s')$, a recursive definition)

We can now derive the probabilities and values of the paths in the diagram.

Backups and value function unpacking

$$\begin{aligned}
\mathbb{E}\left[r_{t+1}|S_t = s\right] &= \sum_{a\in\mathcal{A}} P\left[a|s\right]\left(\sum_{s'\in\mathcal{S}} P\left[s'|s,a\right] r(s,a,s')\right) \\
\mathbb{E}\left[\sum_{k=0}^{\infty}\gamma^k r_{t+k+2}|S_t{=}s\right] &= \sum_{a\in\mathcal{A}} P\left[a|s\right]\left(\sum_{s'\in\mathcal{S}} P\left[s'|s,a\right] \sum_{k=0}^{\infty}\gamma^k r_{t+k+2}\right)
\end{aligned}$$

We also have (from the definition of the value function):

$$\begin{aligned}
V^{\pi}(s') &= \mathbb{E}\left[R_{t+1}|S_t = s'\right] \\
&= \mathbb{E}\left[\sum_{k=0}^{\infty}\gamma^k r_{t+k+2}|S_{t+1} = s'\right]
\end{aligned}$$

With this information we can now to proceed unpack the value function.

Value function self-consistency

$$
\begin{aligned}
V^{\pi}(s) &= \mathbb{E}_{\pi}\left[R_t | S_t = s\right] \\
&= \mathbb{E}_{\pi}\left[\sum_{k=0}^{\infty} \gamma^k r_{t+k+1} | S_t = s\right] \\
&= \mathbb{E}_{\pi}\left[r_{t+1} + \gamma \sum_{k=0}^{\infty} \gamma^k r_{t+k+2} | S_t = s\right] \\
&= \sum_{a \in \mathcal{A}} \pi(a, s) \left(\sum_{s' \in \mathcal{S}} \mathcal{P}^a_{ss'} \left(\mathcal{R}^a_{ss'} + \gamma \mathbb{E}_{\pi}\left[\sum_{k=0}^{\infty} \gamma^k r_{t+k+2} | S_{t+1} = s'\right]\right)\right) \\
&= \sum_{a \in \mathcal{A}} \pi(s, a) \sum_{s' \in \mathcal{S}} \mathcal{P}^a_{ss'} \left(\mathcal{R}^a_{ss'} + \gamma V^{\pi}(s')\right)
\end{aligned}
$$

A fundamental property of value functions is that they satisfy a set of recursive consistency equations. Crucially V^{π} has unique solution.

Policy Evaluation

1. Let us consider how to compute the value function for an arbitrary policy. This is called policy evaluation in the DP literature. We also refer to it as the prediction problem.
2. We could simply systematically apply Bellman's equation over and over again to obtain better estimates for $V_1(s), V_2(s), \ldots, V_k(s)$ iteratively, hoping that the solution converges (it is guaranteed to do so, more on this later). This algorithm is called iterative policy evaluation. Each iteration step k provides successively better approximations to the solution.
3. A typical stopping condition for iterative policy evaluation is to measure by how little the largest change in value function was between two iteration steps and set a cut-off threshold.

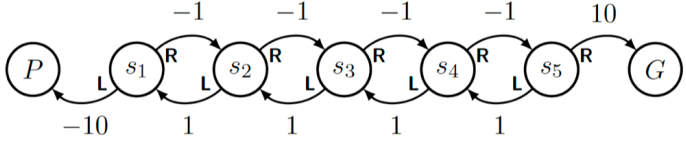
Iterative Policy Evaluation Algorithm

Input π, the policy to be evaluated
Initialize $V(s) = 0$, for all $s \in \mathcal{S}^+$
Repeat
  $\Delta \leftarrow 0$
  For each $s \in \mathcal{S}$:
    $v \leftarrow V(s)$
    $V(s) \leftarrow \sum_a \pi(s,a) \sum_{s'} \mathcal{P}^a_{ss'} \left[\mathcal{R}^a_{ss'} + \gamma V(s')\right]$
    $\Delta \leftarrow \max(\Delta, |v - V(s)|)$
until $\Delta < \theta$ (a small positive number)
Output $V \approx V^\pi$

In Iterative Policy evaluation we sweep through all successor states, we call this kind of operation a full backup.

To produce each successive approximation V_{k+1} from V_k iterative policy evaluation applies the same operation to each state s: it replaces the old value of s in place with a new value obtained from the old values of the successor states of s, and the expected immediate rewards, along **all** the one-step transitions possible under the policy being evaluated. We could also run a code version storing old and new arrays for V. This turns out to converge slower, why?

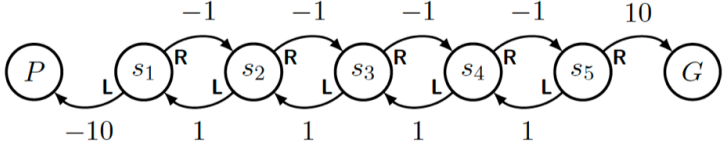
Stair Climbing MDP



Example (Stair Climbing)

Actions are *Left* and *Right* and states are $P, s_1, \ldots, s_5, G$. States P and G are absorbing and we start in s_3. $\gamma = 0.9$. Let us assume an unbiased policy, where every action is equally probable $\pi(s, L) = \frac{1}{2} = \pi(s, R)$. We want now to evaluate the value function $V^{\pi}(s)$ for each state in s.

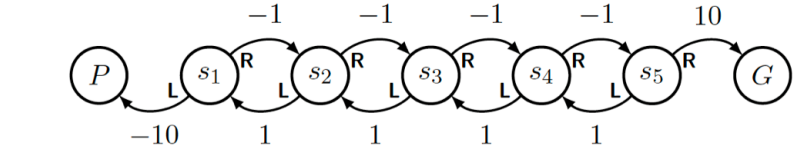
Policy evaluation: Stair climbing 1



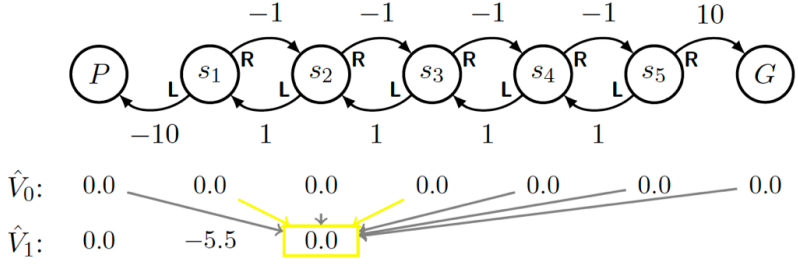
$\hat{V}_0$: 0.0 0.0 0.0 0.0 0.0 0.0 0.0

$\hat{V}_1$:

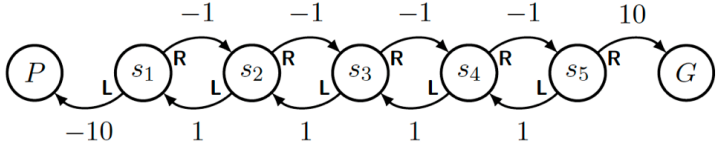
Policy evaluation: Stair climbing 2



Policy evaluation: Stair climbing 3

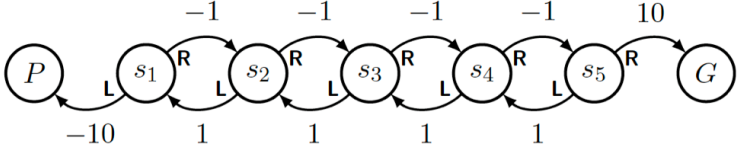


Policy evaluation: Stair climbing 4



$\hat{V}_0$:	0.0	0.0	0.0	0.0	0.0	0.0	0.0
$\hat{V}_1$:	0.0	−5.5	0.0	0.0	0.0	5.5	0.0

Policy evaluation: Stair climbing 5



$\hat{V}_0$:	0.0	0.0	0.0	0.0	0.0	0.0	0.0
$\hat{V}_1$:	0.0	−5.5	0.0	0.0	0.0	5.5	0.0
$\hat{V}_2$:	0.0	−5.5	−2.48	0.0	2.48	5.5	0.0
$\hat{V}_3$:	0.0	−6.61	−2.48	0.0	2.48	6.61	0.0
$\hat{V}_4$:	0.0	−6.61	−2.98	0.0	2.98	6.61	0.0
$\hat{V}_\infty$:	0.0	−6.90	−3.10	0.0	3.10	6.90	0.0

State-Action Value function as Cost-To-Go

How good is it to be in a given state and take a given action when you follow a policy π:

Definition (State-Action Value function "Cost to Go")

$$Q^{\pi}(s,a) = \mathbb{E}\left[R_t | S_t = s, A_t = a\right] = \mathbb{E}\left[\sum_{k=0}^{\infty} \gamma^k r_{t+k+1} | S_t = s, A_t = a\right] \tag{17}$$

The relation between (state) value function and the state-action value function is straightforward:

$$V^{\pi}(s) = \sum_{a \in \mathcal{A}} \pi(s,a) Q^{\pi}(s,a) \tag{18}$$

Optimal Value and Cost-to-Go function for MDPs

Value functions define a partial ordering over policies. A policy is defined to be better than or equal to a policy if its expected return is greater than or equal to that of for all states. In other words, $\pi \geq \pi'$ if and only if $V^{\pi}(s) \geq V^{\pi'}(s)$ for all $s \in \mathcal{S}$.

Definition (Optimal Value function)

$$V^*(s) = \max_{\pi} V^{\pi}(s) \tag{19}$$

Therefore, the policy π^* that maximises the value function is the optimal policy. There is always at least one optimal policy. There may be more than one optimal policy.

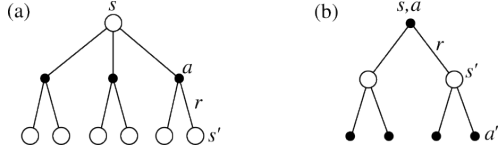
Definition (Optimal State-Action Value function)

$$Q^*(s,a) = \max_{\pi} Q^{\pi}(s,a) \tag{20}$$

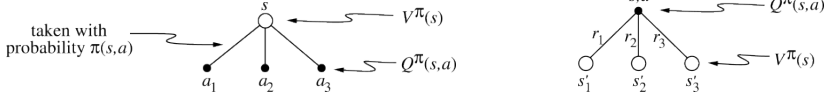
In analogy to before we also have

$$Q^*(s,a) = \mathbb{E}\left[r_{t+1} + \gamma V^*(s_{t+1}) | S_t = s, A_t = a\right] \tag{21}$$

Bellman Equation Backups



Value function $V^{\pi}(s)$ (Fig a) and State-Action Value function $Q^{\pi}(s, a)$ (Fig b) backup diagrams. Relationship between these and the V and Q functions in action:



Bellman optimality equation (derivation)

The value function V^{π^*} for a policy π^* must satisfy the self-consistency condition given by the Bellman equation. Moreover, if we want to have the optimal value for a state, we should only choose the action(s) that yield the highest value, $V(s) = \sum_{a \in \mathcal{A}} \pi(s,a) Q^{\pi}(s,a)$ should become $\max_{a \in \mathcal{A}} Q^{\pi}(s,a)$:

$$
\begin{aligned}
V^*(s) &= \max_{a \in \mathcal{A}} \sum_{a \in \mathcal{A}} \pi(s,a) Q^{\pi}(s,a) \\
&= \max_{a \in \mathcal{A}} Q^{\pi^*}(s,a) \\
&= \max_{a} \mathbb{E}\left[R_t | S_t = s, A_t = a\right] \\
&= \max_{a} \mathbb{E}\left[\sum_{k=0}^{\infty} \gamma^k r_{t+k+1} | S_t = s, A_t = a\right] \\
&= \max_{a} \mathbb{E}\left[r_{t+1} + \gamma \sum_{k=0}^{\infty} \gamma^k r_{t+k+2} | S_t = s, A_t = a\right] \\
&= \max_{a} \mathbb{E}\left[r_{t+1} + \gamma V^*(s_{t+1}) | S_t = s, A_t = a\right] \\
&= \max_{a} \sum_{s'} P\left[s' | s, a\right] \left(r(s,a,s') + \gamma V^*(s')\right) \\
&= \max_{a} \sum_{s'} \mathcal{P}^a_{ss'} \left(\mathcal{R}^a_{ss'} + \gamma V^*(s')\right)
\end{aligned}
$$

Bellman optimality equation for V^*

V^{π^*} is the optimal value function and so conveniently the self-consistency condition can be rewritten in a form without reference to any specific policy π^*: $V^{\pi^*} = V^*$. This yields the Bellman Optimality Equation for an optimal policy:

Definition (Bellmann Optimality Equation for V)

$$V^*(s) = \max_a \sum_{s'} P\left[s'|s,a\right]\left(r(s,a,s') + \gamma V^*(s')\right) \quad (22)$$

$$= \max_a \sum_{s'} \mathcal{P}^a_{ss'}\left(\mathcal{R}^a_{ss'} + \gamma V^*(s')\right) \quad (23)$$

Intuitively, the Bellman Optimality Equation expresses the fact that the value of a state under an optimal policy must equal the expected return for the best action from that state.

Bellman optimality equation for Q^*

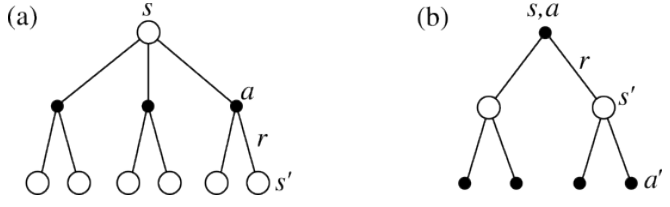
Definition (Bellman Optimality Equation for Q^*)

$$Q^*(s,a) = \mathbb{E}\left[r_{t+1} + \gamma \max_{a'} Q^*(s_{t+1}, a') | S_t{=}s, A_t{=}a\right] \quad (24)$$

$$= \sum_{s'} \mathcal{P}^a_{ss'} \left(\mathcal{R}^a_{ss'} + \gamma \max_{a'} Q^*(s', a')\right) \quad (25)$$

Note how the Bellmann Optimality Equation does not need $\pi^*(a,s)$ explicitly to compute $V^*(s)$ – this is very useful (Why?).
For finite MDPs, the Bellman optimality equation has a unique solution independent of the policy. Why?
The Bellman optimality equation is a set of N non-linear equations, where $N = |\mathcal{S}|$, with N unknowns. Because for known $\mathcal{P}^a_{ss'}, \mathcal{R}^a_{ss'}$ one can solve this system of equations for using any method for solving systems of nonlinear equations.

Bellman Optimality Equation Backups



Value function $V(s)$ (a) and State-Action Value function $Q(s, a)$ (b) backup diagrams.

On solving the Bellman Optimality Equations

Explicitly solving the Bellman optimality equation provides one route to finding an optimal policy, and thus to solving the reinforcement learning problem. This solution approach can often be challenging at best, if not impossible, because it is like an exhaustive search, looking ahead at all possible traces, computing their probabilities of occurrence and their desirabilities in terms of expected rewards. Moreover, this solution relies on at least 3 assumptions that are rarely true in practice:

- we accurately know the dynamics of the environment
- we have computational resources to find the solution
- the Markov property.

Thus, in reinforcement learning often we have to (and want to) settle for approximate solutions.

Bellman Optimality Equation convergence

Theorem (Bellman Optimality Equation convergence theorem)

For an MDP with a finite state and action space

1. *The Bellman (Optimality) equations have a unique solution.*
2. *The values produced by value iteration converge to the solution of the Bellman equtions.*

A complete proof of this theorem is beyond the scope of this course. It rests on a very useful lemma from analysis called the Banach Fixed Point Theorem, a.k.a the Contraction Mapping Theorem. But we can gain some intuition of the proof.

Convergence proof I

Intuition for the proof

I. the Bellman Optimality Equation (BOE) $V^{\pi^*}(s) = \max_{a \in \mathcal{A}} \sum_{s'} \mathcal{P}^a_{ss'} \left(\mathcal{R}^a_{ss'} + \gamma V^{\pi^*}(s') \right)$ is a so called fixed point equation.

II. Suppose we start with some arbitrary value function $V(\dots)$. We can apply the right hand side of the BOE to $V(\dots)$ to get a new value function $V'(\dots)$. Thus, the right hand side of the BOE defines an operator K on the space of value functions (taking a function as argument and yielding a function).

III. The Bellman optimality equation states that the optimal value function is a fixed point: $V^{\pi^*} = K(V^{\pi^*})$.

IV. A simple process to find a fixpoint of K is as follows:

Convergence proof II

1. Begin with any value function V_0, e.g. $V_o(s) = 0 \forall s$.
2. Apply K to it to get V_1: $V_1 \leftarrow K(V_0)$
3. Apply K again to get V_2: $V_2 \leftarrow K(V_1)$
4. ...

This is in fact what value iteration does.

Convergence proof III

V. This process of iterating K converges to a limit and the limit is unique. Why? K has a special property: it is a contraction. This means that given any two value functions V_1 and V_2, applying to them K brings them closer together:

$$\|K(V_1) - K(V_2)\|_\infty \leq \gamma \|V_1 - V_2\|_\infty \tag{26}$$

where $V_1 \neq V_2$ and norm $\|\ldots\|$ is the sup norm/max norm

$$\|V\|_\infty = \max_{s \in \mathcal{S}} |V(s)| \tag{27}$$

VI. The constant $\gamma \in [0, 1[$ is the MDP's discount factor. With $\gamma < 1$, the norm between two distinct points decreases strictly. Intuitive reasoning: if we have a process that keeps bringing points closer to each other, and we iterate the process, then we will converge to a fixed point.

Convergence proof IV

- The Contraction Mapping Theorem says that under appropriate conditions, which are satisfied by the BOE and MDPs, this is in fact the case: the process of iterating K has a limit, which is the unique fixed point of K: V^{π^*}.

Section overview

Dynamic Programming

1. Motivation
2. Reinforcement Learning 101
3. Lets go Markov
4. Markov Decision Process
5. Dynamic Programming
   - Policy Improvement
   - Policy Iteration
   - Value Iteration
   - Backup strategies
6. Model-Free Learning
7. Model-Free Control
8. Closing thoughts

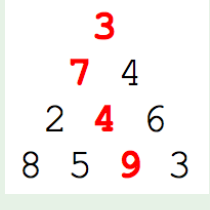
Dynamic Programming

The term dynamic programming (DP) refers to a collection of algorithms that can be used to compute optimal policies given a perfect model of the environment as a Markov decision process (MDP). We assume for convenience all MDPs to be finite. DP provides an essential foundation for understanding the RL methods out there. In fact, all of these machine learning methods can be viewed as ways to obtain the same effect as DP, only with less computation and without assuming a perfect model of the environment.

Maximum path sum

Example (Maximum path sum 1)

By starting at the top of the triangle below and moving to adjacent numbers on the row below, find the path with the maximum sum.



The maximum total from top to bottom is $3 + 7 + 4 + 9 = 23$.

Maximum path sum 2

Example

```
                75
               95 64
              17 47 82
             18 35 87 10
            20 04 82 47 65
           19 01 23 75 03 34
          88 02 77 73 07 63 67
         99 65 04 28 06 16 70 92
        41 41 26 56 83 40 80 70 33
       41 48 72 33 47 32 37 16 94 29
      53 71 44 65 25 43 91 52 97 51 14
     70 11 33 28 77 73 17 78 39 68 17 57
    91 71 52 38 17 14 91 43 58 50 27 29 48
   63 66 04 68 89 53 67 30 73 16 69 87 40 31
04 62 98 27 23 09 70 98 73 93 38 53 60 04 23
```

How many possible routes are there? 16384
What is the maximum path sum? 1074

Maximum Path Sum solutions

Brute Force

The brute force approach to this problem would involve tracing every path and computing the maximum path cost. The time complexity of this algorithm is $\mathcal{O}(2^{n-1})$ where n is the number of rows in the triangle.

For 100 rows we need to evaluate 2^{99} paths.

Dynamic Programming

An efficient maximum path solution algorithm is to take a dynamic programming approach. We can split up the triangle into small sub-triangles, and working from the bottom-up, we can then compute the maximum path in a single pass. For each cell in the triangle we find the max of the two nodes below it and add that to the node value. This algorithm has time complexity $\mathcal{O}(n)$, where n is the number of nodes, a vast improvement over brute force.

Dynamic Programming – Origins 1

Bellman developed Dynamic programming (DP) to solve Markov Decision Processes. DP has grown beyond its original purpose into both a mathematical optimisation and a computer programming method. In these contexts it refers to simplifying a complicated problem by breaking it down into simpler sub-problems in a recursive manner.
DP exploits the fact that decisions that span several points in time do often break apart recursively. Bellman called this the "Principle of Optimality".
A problem that can be solved optimally by breaking it into sub-problems and then recursively finding the optimal solutions to the sub-problems is said to have optimal substructure. If sub-problems can be nested recursively inside larger problems, so that dynamic programming methods are applicable, then there is a relation between the value of the larger problem and the values of the sub-problems. In the optimisation literature this relationship is called the Bellman equation.

Dynamic Programming – Origins 2

To be be able to apply Dynamic Programming requires problems to have

1. Optimal substructure, meaning that the solution to a given optimisation problem can be obtained by the combination of optimal solutions to its sub-problems.
2. Overlapping sub-problems, meaning that the space of sub-problems must be small, i.e., any recursive algorithm solving the problem should solve the same sub-problems over and over, rather than generating new sub-problems.

Example

The maximum path sum problem or Dijkstra's algorithm for the shortest path problem are dynamic programming solutions. In contrast, if a problem can be solved by combining optimal solutions to **non-overlapping** sub-problems, the strategy is called "divide and conquer" instead (e.g. quick sort).

Policy Improvement

Our reason for computing the value function for a policy is to help find better policies.

Example

Suppose we have determined the value function for an arbitrary deterministic policy π. We know how good it is to follow the current policy π from s – the value is $V^{\pi}(s)$. Would it be better or worse to change to the new policy π', e.g. $\pi(s) = a$ and $\pi'(s) = a'$? One way is to select for state s alone a different action and continue using the old policy otherwise. The value of this must be, by definition $Q^{\pi}(s, a')$.
The key test is whether $Q^{\pi}(s, a') > V^{\pi}(s)$ or not. If it is greater, that is, if it is better to select a' in state s and thereafter follow $\pi(s)$ and that the new policy would in fact be a better overall.

Policy Improvement Theorem

Policy Improvement Theorem

Let π and π' be any two deterministic policies such that $\forall s \in \mathcal{S}$: $Q^{\pi}(s, \pi'(s)) \geq V^{\pi}(s)$.
Then π' must be as good or better than π: $V^{\pi'}(s) \geq V^{\pi}(s), \forall s \in \mathcal{S}$.

Note, that if the first inequality is strict in any state, then the latter inequality must be strict in at least one state.

Policy Improvement Theorem: Proof I

Consider a deterministic policy $\pi(s) = a$.

- We can always be as good or improve by acting greedily

$$\pi'(s) = \arg\max_{a \in \mathcal{A}} Q^{\pi}(s, a) \tag{28}$$

- This must improve the value from any state s for one step,

$$Q^{\pi}(s, \pi(s')) = \max_{a \in \mathcal{A}} Q^{\pi}(s, a) \geq Q^{\pi}(s, \pi(s)) = V^{\pi}(s) \tag{29}$$

Policy Improvement Theorem: Proof II

- Therefore, it follows it improves the value function, $V^{\pi'}(s) \geq V^{\pi}(s)$

$$\begin{aligned} V^{\pi}(s) &\leq Q^{\pi}(s, \pi'(s)) \\ &= \mathbb{E}\left[R_{t+1} + \gamma V^{\pi}(S_{t+1}) | S_t = s\right]_{\pi'} \\ &\leq \mathbb{E}\left[R_{t+1} + \gamma Q^{\pi}(S_{t+1}), \pi'(S_{t+1}) | S_t = s\right] \\ &\leq \mathbb{E}\left[R_{t+1} + \gamma R_{t+2} + \gamma^2 Q^{\pi}(S_{t+2}), \pi'(S_{t+2}) | S_t = s\right] \\ &\leq \mathbb{E}\left[R_{t+1} + \gamma R_{t+2} + \ldots | S_t = s\right] \\ &= V^{\pi'}(s) \end{aligned}$$

- Halting condition: If there is no more improvement, i.e.

$$Q^{\pi}(s, \pi(s')) = \max_{a \in \mathcal{A}} Q^{\pi}(s, a) \geq Q^{\pi}(s, \pi(s)) = V^{\pi}(s) \qquad (30)$$

Policy Improvement Theorem: Proof III

- Then the Bellman Optimality Equation (BOE)must have been satisfied

$$V^{\pi}(s) = \max_{a \in \mathcal{A}} Q^{\pi}(s, a) \tag{31}$$

- Hence

$$V^{\pi}(s) = V^{\pi^*}(s) = V^*(s) \ \forall s \in \mathcal{S} \text{ and } \pi = \pi^* \qquad \square \tag{32}$$

Policy Iteration

Definition

Once a policy, π , has been improved using V^{π} to yield a better policy π', we can compute $V^{\pi'}$ and improve it again to π'', to yield an even better $V^{\pi''}$. We can thus obtain a sequence of monotonically improving policies and value functions. This way of finding an optimal policy is called policy iteration.

Note, that each policy evaluation is an iterative computation in itself, and is started with the value function for the previous policy. This typically results in a great increase in the speed of convergence of policy evaluation and policy iteration often converges in surprisingly few iterations. The policy improvement theorem assures us that these policies are better than the original policy.

Bellman's Principle of Optimality

Let us collect what we learned together:

Theorem (Principle of Optimality)

A policy $\pi(a|s)$ achieves the optimal value from state s, $V^{\pi}(s) = V^{*}(s)$, if and only if

1. *For any state s' reachable from s, i.e. $\exists a : p(s', s, a) > 0$*
2. *π achieves the optimal value from state s', $V^{\pi}(s') = V^{*}(s')$.*

Any optimal policy can be subdivided into two components:

1. An optimal action a^{*}.
2. Followed by an optimal policy from successor state s'

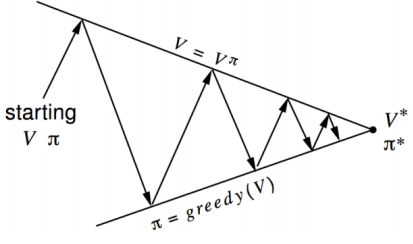
Policy Iteration Algorithm

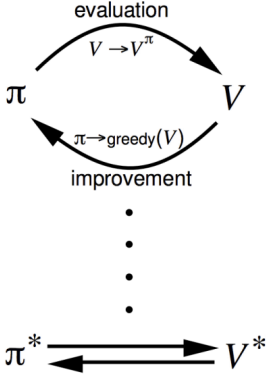
1. Initialization
   $V(s) \in \Re$ and $\pi(s) \in \mathcal{A}(s)$ arbitrarily for all $s \in \mathcal{S}$

2. Policy Evaluation
   Repeat
   $\Delta \leftarrow 0$
   For each $s \in \mathcal{S}$:
   $v \leftarrow V(s)$
   $V(s) \leftarrow \sum_{s'} \mathcal{P}_{ss'}^{\pi(s)} \left[\mathcal{R}_{ss'}^{\pi(s)} + \gamma V(s') \right]$
   $\Delta \leftarrow \max(\Delta, |v - V(s)|)$
   until $\Delta < \theta$ (a small positive number)

3. Policy Improvement
   policy-stable $\leftarrow$ *true*
   For each $s \in \mathcal{S}$:
   $b \leftarrow \pi(s)$
   $\pi(s) \leftarrow \arg\max_a \sum_{s'} \mathcal{P}_{ss'}^{a} \left[\mathcal{R}_{ss'}^{a} + \gamma V(s') \right]$
   If $b \neq \pi(s)$, then *policy-stable* $\leftarrow$ *false*
   If *policy-stable*, then stop; else go to 2

Generalised Policy Iteration Algorithm





Policy evaluation Estimate v_π
Any policy evaluation algorithm

Policy improvement Generate $\pi' \geq \pi$
Any policy improvement algorithm

More than one way to iterate a policy

One drawback to policy iteration is that each iteration involves a full policy evaluation (which can be protracted iterative computation requiring multiple sweeps through the state set). If policy evaluation is done iteratively, then convergence exactly to occurs only in the limit. Do we need to wait for policy evaluation need to converge to V^π?
We can introduce a stopping condition:

- ϵ-convergence of value function (e.g. $\forall s : V_{100}(s) - V_{101}(s) \leq \epsilon$)
- Stop after k iterations of iterative policy evaluation.

Prediction: Smaller k means more policy improvements and fewer policy iterations are executed till convergence.
Is there a trade-off between the two that minimises total numerical effort?

The $k = 1$ case

- The policy evaluation step of policy iteration can be truncated in several ways without losing the convergence guarantees of policy iteration. One important special case is when policy evaluation is stopped after just one sweep (one backup of each state) of the value function. This algorithm is called value iteration.
- Value iteration is obtained simply by turning the Bellman Optimality Equation into an update rule.

Value Iteration

The Dynamic Programming you knew (shortest path algorithms or similar) are in fact just deterministic policy MDPs with deterministic actions, lets recast them in the language of MDPs:

- If we know the solution to subproblems $V^*(s')$
- Then solution $V^*(s)$ can be found by one-step look-ahead

$$V^*(s) \leftarrow \max_{a \in \mathcal{A}} \sum_{s'} \mathcal{P}^a_{ss'} [\mathcal{R}^a_{ss'} + \gamma V^*(s')] \tag{33}$$

- The idea of value iteration is to apply these updates iteratively
- Intuition: start with final rewards and work backwards (consider our maximal path sum example or ...)

Value Iteration Algorithm

Initialize V arbitrarily, e.g.,$V(s) = 0$, for all $s \in \mathcal{S}^+$

Repeat
  $\Delta \leftarrow 0$
  For each $s \in \mathcal{S}$:
    $v \leftarrow V(s)$
    $V(s) \leftarrow \max_a \sum_{s'} \mathcal{P}^a_{ss'} \left[\mathcal{R}^a_{ss'} + \gamma V(s')\right]$
    $\Delta \leftarrow \max(\Delta, |v - V(s)|)$
until $\Delta < \theta$ (a small positive number)

Output a deterministic policy, π, such that
  $\pi(s) = \arg\max_a \sum_{s'} \mathcal{P}^a_{ss'} \left[\mathcal{R}^a_{ss'} + \gamma V(s')\right]$

Unlike policy iteration, there is no explicit policy.

Deterministic Value Iteration

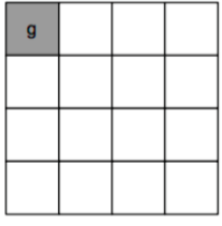
The Dynamic Programming you knew (shortest path algorithms or similar) are in fact just deterministic policy MDPs with deterministic actions, lets recast them in the language of MDPs:

- If we know the solution to subproblems $V^*(s')$
- Then solution $V^*(s)$ can be found by one-step look-ahead

$$V^*(s) \leftarrow \max_{a \in \mathcal{A}} \sum_{s'} \mathcal{P}^a_{ss'} [\mathcal{R}^a_{ss'} + \gamma V^*(s')] \tag{34}$$

- The idea of value iteration is to apply these updates iteratively
- Intuition: start with final rewards and work backwards (consider our maximal path sum example or ...)

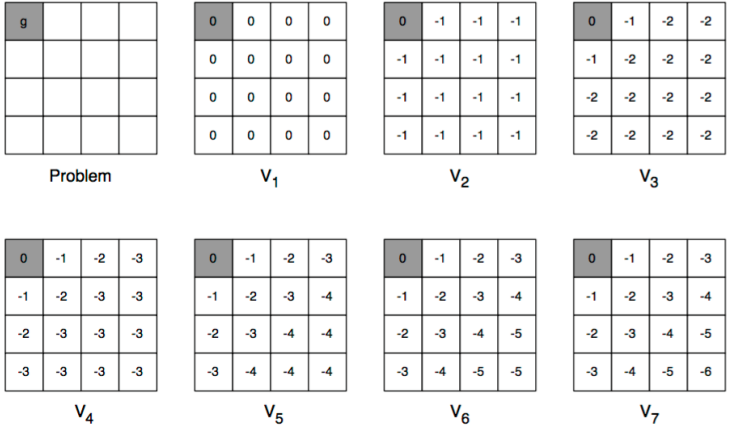
Shortest path problem in a grid world



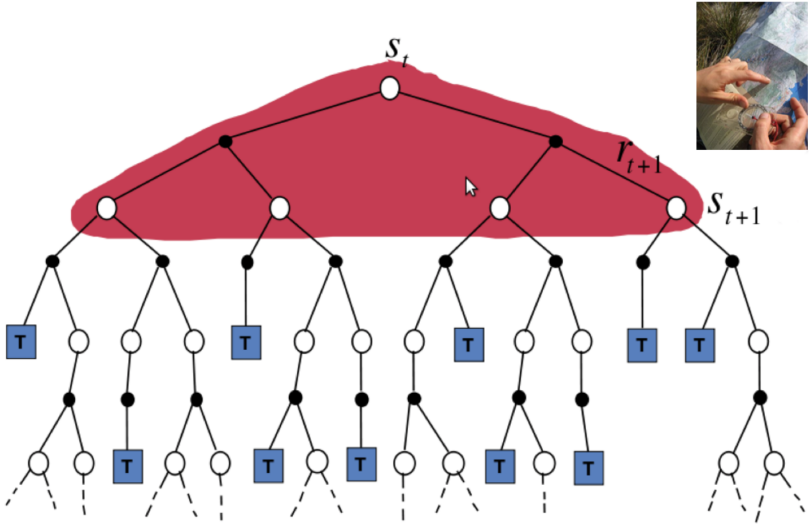
Problem

- States equals locations in grid
- Walls impede transitions.
- 4 possible actions $\mathcal{A} = W, N, E, S$.
  No standing still.
- Each step one cost.
- Terminal/goal state (0 reward)
  top-left corner.

Deterministic value iteration in shortest path



Dynamic Programming performs full backups



Synchronous vs Asynchronous backups

- DP methods can use synchronous backups i.e. all states are backed up in parallel (this requires two copies of the value function)
- Asynchronous DP backups states target only states individually – in any order (one copy of value function)
  - For each selected state we apply the appropriate backup in-place
  - This can significantly reduce computation and
  - we are still guaranteed to converge if over time all states continue to be selected

Let us look at 2 simple ideas for smarter asynchronous DP

1. Prioritised sweeping

- Use the magnitude of Bellman error to guide state selection, choose to update in-place state s, for which

$$\| v(s) - \max_{a \in \mathcal{A}} \left(\mathcal{R}_s^a + \gamma \sum_{s' \in \mathcal{S}} P\left[s'|s,a\right] v(s') \right) \| \tag{35}$$

- Backup the state with the largest remaining Bellman error
- Update Bellman error of affected states after each backup
- Requires knowledge of reverse dynamics $P\left[s|s',a\right]$
- Efficient implementation by maintaining a priority queue

2. Real-time dynamic programming

Idea: update only states that are relevant to agent (Is this a problem for convergence?). Use agent experience to guide the selection of states

- After each time-step S_t, A_t, R_{t+1}
- Backup the state S_t that was just visited, i.e.

$$V(S_t) \leftarrow max_{a \in \mathcal{A}} \left(\mathcal{R}^a_{S_t} + \gamma \sum_{s' \in \mathcal{S}} P\left[s' | S_t, a\right] V(s') \right) \tag{36}$$

Let us take a step back from DP

- DP uses full-width backups
- For each synchronous or asynchronous backup
  - We need complete knowledge of the MDP transitions and reward function:
    aside from optimising the policy, our agent is not doing a lot of machine learning
  - Every successor state and action is considered
- DP is effective for medium-sized problems (millions of states)
- For large problems DP suffers the Curse of Dimensionality:
  Number of states $N = |\mathcal{S}\|$ grows exponentially with number of (continuous) state variables
- Even one backup can be too expensive (e.g. consider control of an n-joint robot arm).

Bootstrapping

Dynamic Programming updates value estimates based on other value estimates – efficient use of data, thanks to the optimal sub-problem structure.

Bootstrapping: Update value estimates based on other value estimates. This is akin to the "magical" ability to pull oneself up by ones boot straps.

Note, in the case of DP here we look at self-consistency, so in the end each states values is grounded in all the states immediate

Let the learning begin

We will consider next sample backups:

- Using the rewards and transitions, which the agent samples by experience: S, A, R, S?
- instead of given reward function $\mathcal{R}$ and transition dynamics $\mathcal{P}$
- Advantages:
  1. Model-free: no advance knowledge of MDP required
  2. Breaks the curse of dimensionality through sampling (no full-backups, instead sample backups)
  3. Cost of backups remains constant and independent of $N = \|\mathcal{S}\|$

Section overview

Model-Free Learning

1. Motivation
2. Reinforcement Learning 101
3. Lets go Markov
4. Markov Decision Process
5. Dynamic Programming
6. Model-Free Learning
   - Monte Carlo Learning
   - TD Learning
7. Model-Free Control
8. Closing thoughts

Model-Free Learning

- Past: Planning by dynamic programming
  Solve a known MDP (no learning), but we learned to optimise our planning using DP.
- Now: Model-free prediction ("Policy evaluation")
  Estimate the value function of an unknown MDP
- Next: Model-free control ("Policy improvement")
  Optimise the value function of an unknown MDP

Model-Free Reinforcement Learning: Monte Carlo (MC)

1. MC methods learn directly from episodes of experience
2. MC is model-free: no knowledge of MDP transitions or rewards needed
3. MC learns from complete episodes (of sample traces): no bootstrapping
4. MC uses the simplest possible idea: value of state = mean return
   $\Rightarrow$ BUT this can only be applied to episodic MDPs that have terminal states.

Monte Carlo (MC) Methods

We want to learn the value function for a given policy π.

- Recall that the value of a state is the expected return–expected cumulative future discounted reward–starting from that state.
- An obvious way to estimate it from experience, then, is simply to average the returns observed after visits to that state.
- As more returns are observed, the average should converge to the expected value.

This simple idea underlies all Monte Carlo methods.

MC Policy Evaluation

- Goal: learn V^π from traces τ of episodes of length T that we experience under policy π
  $\tau \equiv s_1, a_1, r_2, \ldots, s_k$
- We already defined the return as the total discounted reward:
  $R_t = r_{t+1} + \gamma r_{t+2} + \cdots + \gamma^{T-1} r_T$
- We already defined the value function as the expected return:
  $V^\pi(s) = \mathbb{E}\left[R_t | S_t = s\right]$
- Monte-Carlo policy evaluation uses empirical mean returns instead of expected return.

Monte-Carlo Policy Evaluation

1: **procedure** MONTECARLOESTIMATION(π)
2: **Init**
3: $\hat{V}(s) \leftarrow$ arbitrary value, for all $s \in S$.
4: $Returns(s) \leftarrow$ an empty list, for all $s \in S$.
5: **EndInit**
6: **repeat**
7: Get trace, τ, using π.
8: **for all** s appearing in τ **do**
9: $R \leftarrow$ return from first appearance of s in τ.
10: Append R to $Returns(s)$
11: $\hat{V}(s) \leftarrow$ average($Returns(s)$)
12: **until** forever

Above we have the First visit MC algorithm, as we append the return of the episode from the first occurrence of a state s. Another version is Every visit MC, where we append the return of the episode (from that point) on every occurrence of state s in the episode.

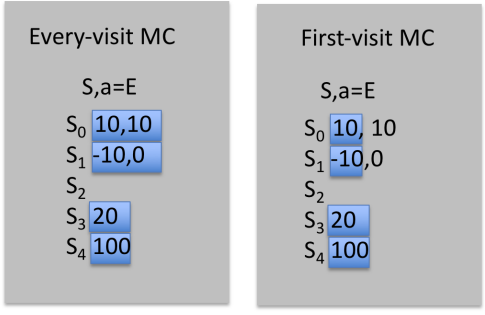
Averaging the returns: First visit vs Every visit

For a single entry, the $Returns(s)$ list of returns is averaged over either the first or all entries of an episode.

Trace example:

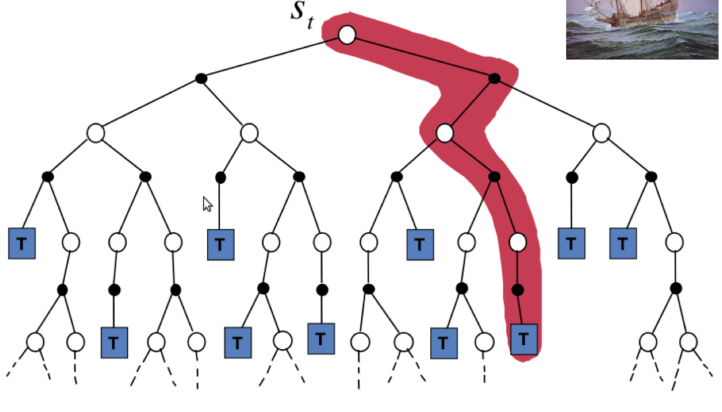
(S_0, a=E,r=10) (S_1,a=E, r=-10) (S_0 ,a=E, r=10)

(S_3 , a=E, r=20) (S_1 , a=E, r=0) (S_4 , a=E, r=100)



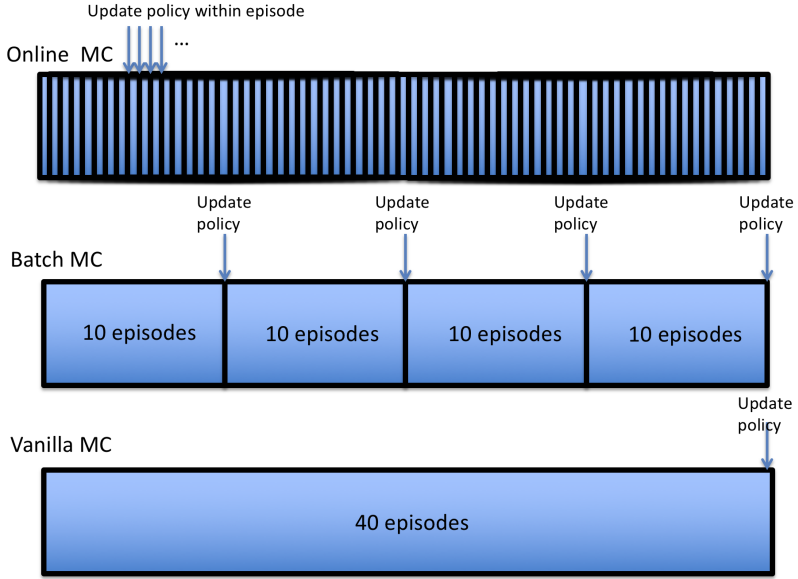
Monte Carlo performs sample trace evaluations

Another way is to sample values for V



But, remember "you cannot backup death"
(what does this imply?)

Batch & Online Monte-Carlo



Batch vs online averaging

Normally we batch process data $x_1, x_2, \ldots$ to calculate its mean μ

$$\mu = \frac{1}{k}\sum_{j=1}^{k} x_j \tag{37}$$

In an RL world we would like to be able to do this online while we experience new data and compute the current means $\mu_1, \mu_2, \ldots$ efficiently.

$$\begin{aligned}
\mu_k &= \frac{1}{k}\sum_{j=1}^{k} x_j \\
&= \frac{1}{k}(x_k + \sum_{j=1}^{k-1} x_j) \\
&= \frac{1}{k}(x_k + (k-1)\mu_{k-1}) \\
&= \mu_{k-1} + \frac{1}{k}(x_k - \mu_{k-1})
\end{aligned}$$

Incremental estimation updates

Consider the structure of the update on the average

$$\mu_k = \mu_{k-1} + \frac{1}{k}(x_k - \mu_{k-1})$$

It has the form of an incremental estimation computation

$$\Delta = \mu_k - \mu_{k-1} = \frac{1}{k}(x_k - \mu_{k-1})$$

- a small weighting factor ≤ 1
- an old estimated value μ that is updated with
- new data x ("the difference pulls the estimate in the direction of the data")

We will encounter this structure difference structure throughout Model-Free Learning.

Incremental Monte-Carlo Updates

We can now update value functions without having to store sample traces:

1. Update $V(s)$ incrementally after episode s_1, a_1, r_2, s_t
2. For each state s_t with return R_t (up to this point) and $N(s)$ the visit counter to this state:

$$\begin{aligned} N(s_t) &\leftarrow N(s_t) + 1 \\ V(s_t) &\leftarrow V(s_t) + \frac{1}{N(s_t)}(R_t - V(s_t)) \end{aligned}$$

Moreover, if the world is non-stationary, it can be useful to track a running mean, i.e. by gradually forgetting old episodes.

$$V(s_t) \leftarrow V(s_t) + \alpha(R_t - V(s_t))$$

The parameter α controls the rate of forgetting old episodes.
Why should we consider non-stationary conditions?

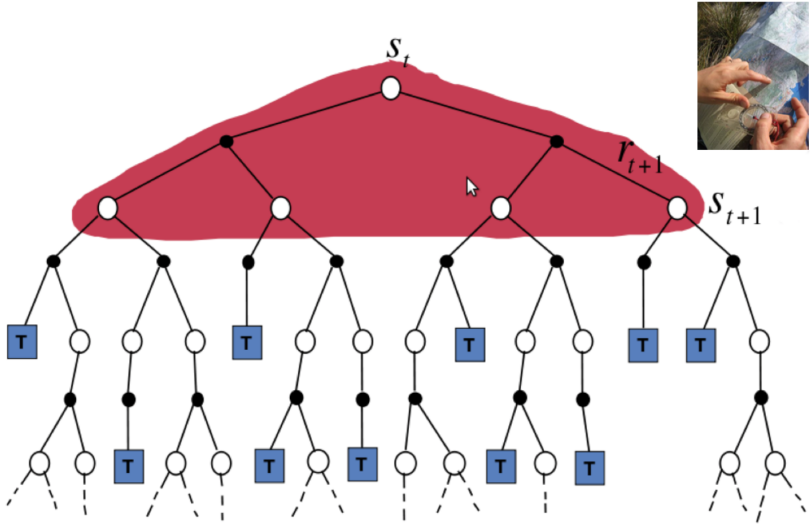
Combine and conquer

DP and MC techniques both have desirable properties, that conflict with each other:

- Bootstrapping: Dynamic Programming updates value estimates based on other value estimates – efficient use of data, thanks to the optimal sub-problem structure. Note, in the case of RL here we look at episodic tasks and so we know that at the end of the episode we will be grounded by the actual return outcome.
- Sampling: Monte-Carlo methods sample rewards from the environment – no need to know the MDP.

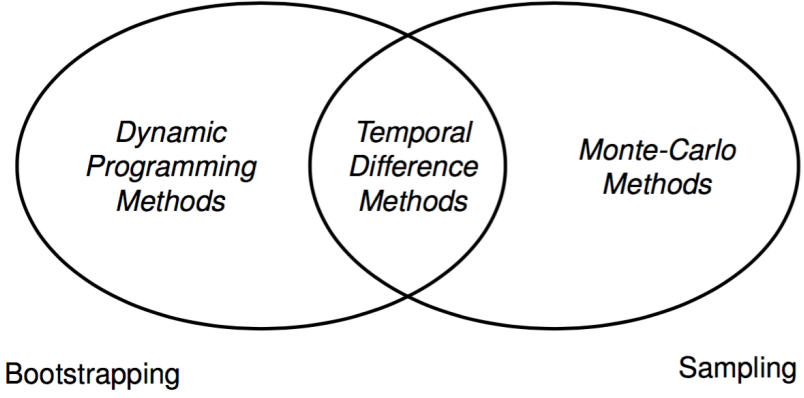
Can we have the cake and eat it?

Dynamic Programming performs full backups



Temporal Difference Learning

Temporal difference (TD) learning sits between Dynamic Programming and Monte-Carlo methods.



Temporal-Difference (TD) Learning

1. TD methods learn directly from episodes of experience (but also works for non-episodic tasks)
2. TD is model-free: no knowledge of MDP transitions or rewards needed
3. TD learns from incomplete episodes, by bootstrapping
4. TD updates a guess towards a guess

Temporal Difference Learning update rule

Recall that we use the following Monte-Carlo update

$$V(s_t) \leftarrow V(s_t) + \alpha(R_t - V(s_t)) \tag{38}$$

We update the value of $V(s_t)$ towards the actual return R_t. Note, how we use only measurements to form our estimates.
Temporal Difference methods perform a similar update after every time-step, i.e.

$$V(s_t) \leftarrow V(s_t) + \alpha\left(r_{t+1} + \gamma V(s_{t+1}) - V(s_t)\right) \tag{39}$$

We update the value of $V(s_t)$ towards the estimated return $r_{t+1} + \gamma V(s_{t+1})$ Note, how we are combining a measurements r_{t+1} with an estimate $V(s_{t+1})$ to produce a better estimate $V(s_t)$.

TDerminology

- $r_{t+1} + \gamma V(s_{t+1}) - V(s_t)$ is the Temporal Difference Error.
- $r_{t+1} + \gamma V(s_{t+1})$ is the Temporal Difference Target

TD value function estimation Algorithm

procedure TD-ESTIMATION(π)
  Init
    $\hat{V}(s) \leftarrow$ arbitrary value, for all $s \in S$.
  EndInit
  repeat(For each episode)
    Initialise s
    repeat(For each step of episode)
      a action chosen from π at s
      Take action a; observe r, and next state, s'
      $\delta \leftarrow r + \gamma \hat{V}(s') - \hat{V}(s)$
      $\hat{V}(s) \leftarrow \hat{V}(s) + \alpha\delta$
      $s \leftarrow s'$
    until s is absorbing state
  until Done
end procedure

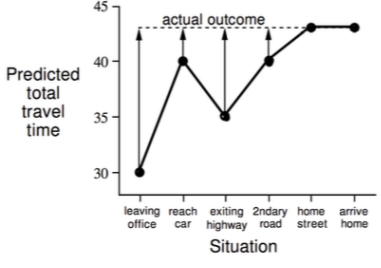
ETA prediction example

Consider a Markov Reward Process for a GPS navigation system that produces an Estimated Time of Arrival (ETA) (MRP because here we are interested in planning not control), the value function is the time to arrival for each state, i.e. each minute that passes incurs reward -1.

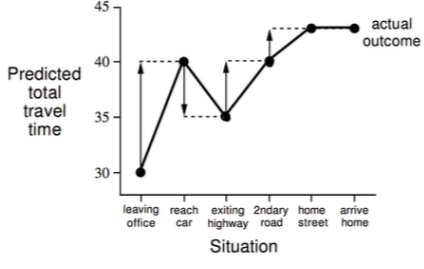
State	Elapsed Time (minutes)	Predicted Time to Go	Predicted Total Time
leaving office	0	30	30
reach car, raining	5	35	40
exit highway	20	15	35
behind truck	30	10	40
home street	40	3	43
arrive home	43	0	43

ETA prediction example: MC vs TD

Changes recommended by Monte Carlo methods (α=1)



Changes recommended by TD methods (α=1)

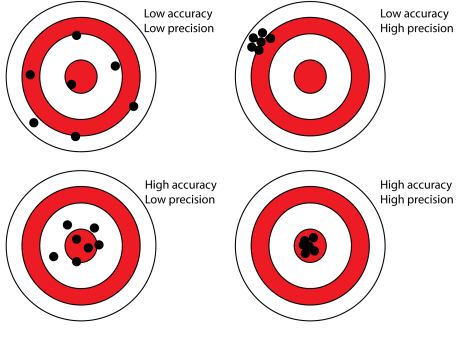


Advantages & Disadvantages of MC & TD

1. TD can learn before knowing the final outcome ("you can back-up near-death")
   - TD can learn online after every step
   - MC must wait until end of episode before return is known
2. TD can learn without the final outcome
3. TD can learn from incomplete sequences
   - MC can only learn from complete sequences
   - TD works in continuing (non-terminating) environments MC only works for episodic (terminating) environments

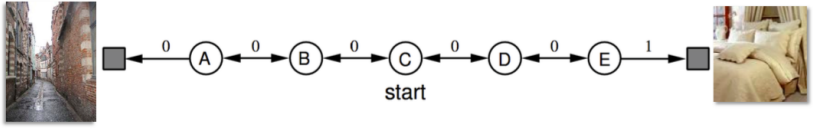
The Bias-Variance Tradeoff

1. Sample return $R_t = r_{t+1} + \gamma r_{t+2} + \cdots + \gamma^{T?1} r_T$ is unbiased estimate of $V^\pi(s_t)$
2. "True" TD target $r_{t+1} + \gamma V^\pi(s_{t+1})$ is unbiased estimate of $V^\pi(s_t)$
3. Estimated TD target $r_{t+1} + \gamma V(s_{t+1})$ is biased estimate of $V^\pi(s_t)$
4. TD target is much lower variance than the return:
   - Sample return depends on many random actions, transitions, rewards.
   - TD target depends on one random action, transition, reward

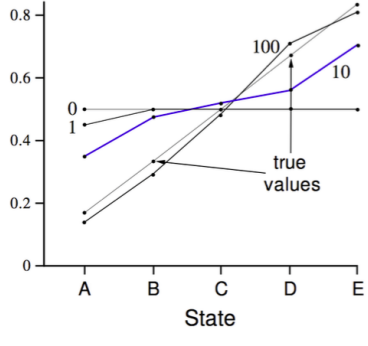


- MC has high variance, zero bias
  - Good convergence properties
  - even with function approximation (to be discussed later)
  - not very sensitive to initial value
  - very simple to understand and use
- TD exploits Markov property
  $\Rightarrow$ Usually more efficient in Markov environments
- TD has low variance, some bias
  - Usually more efficient than MC
  - TD – to be precise TD(0) – converges to $V^{\pi}(s)$
  - convergence not guaranteed with function approximation
  - more sensitive to initial value
- MC does not exploit Markov property
  $\Rightarrow$ Usually more effective in non-Markov environments

Random walk example: TD vs MC



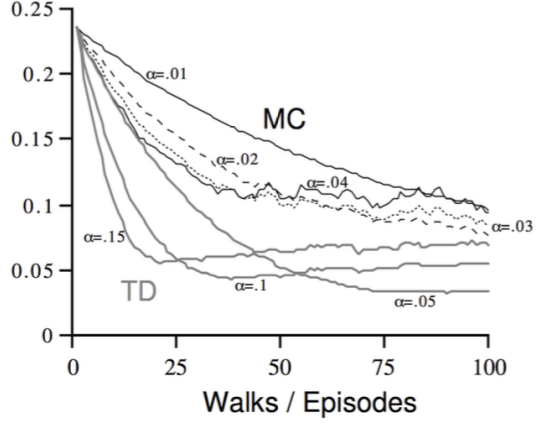
Estimated value



Values learned by TD(0) after various numbers of episodes

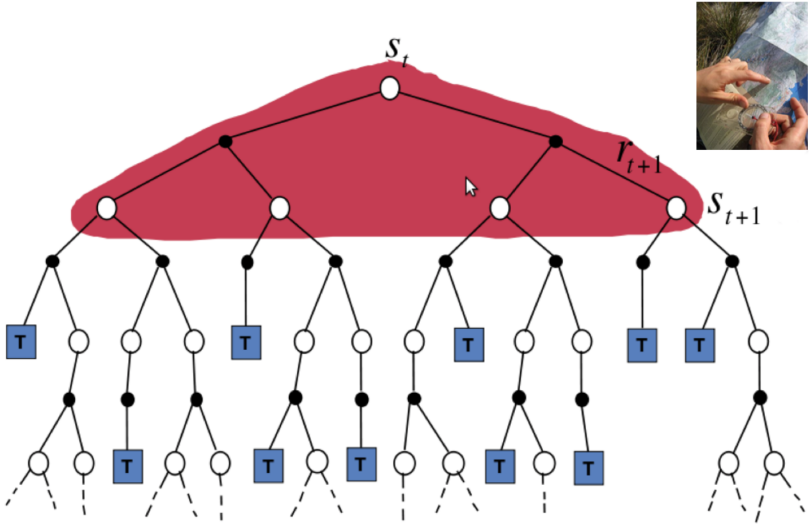
Random walk example: TD vs MC

RMS error, averaged over states



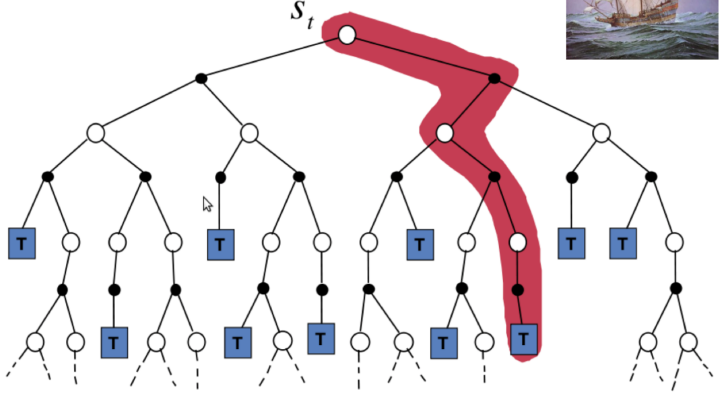
Data averaged over
100 sequences of episodes

Dynamic Programming performs full backups



Monte Carlo performs sample trace evaluations

Another way is to sample values for V



TD performs bootstrapping from 1-step samples

Summary DP vs TD vs MC

Bootstrapping: update involves an *estimate*

– MC does not bootstrap
– DP bootstraps
– TD bootstraps

Sampling: update does not involve an *expected value*

– MC samples
– DP does not sample
– TD samples

Section overview

Model-Free Control

1. Motivation
2. Reinforcement Learning 101
3. Lets go Markov
4. Markov Decision Process
5. Dynamic Programming
6. Model-Free Learning
7. Model-Free Control
   - MC Policy Improvement
   - On-Policy Methods
   - TD control
   - Off-Policy methods
   - Q-Learning
8. Closing thoughts

Model-Free RL

- We completed the basics of Model-Free Learning: Estimate the value function of an unknown MDP
- We need to understand how to do Model-Free Control: Optimise the value function of an unknown MDP

We can use Model-Free Control in two important scenarios:

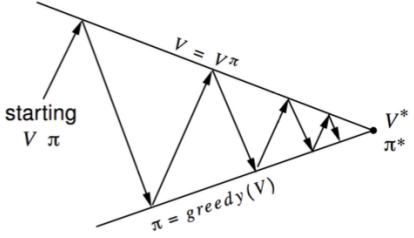
1. MDP model is known, but is too big to use (Curse of Dimensionality), except by sampling
2. MDP model is unknown, but experience can be sampled.

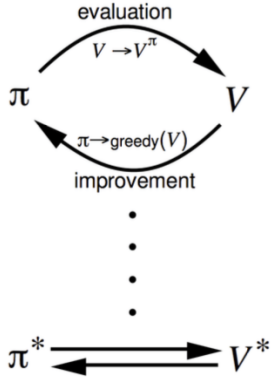
Model-Free Control

How do we find a policy in a model-free world, where value function or action-value functions are learned by MC or TD evaluation? We follow the idea of generalised policy iteration (GPI) for MDPs. Then and here GPI maintains both an approximate policy and an approximate value function, yet converges to an optimal policy.

- The value function is repeatedly altered to more closely approximate the value function for the current policy.
- and the policy is repeatedly improved with respect to the current value function.

Generalised Policy Iteration on State Value function





Policy evaluation Estimate v_π
e.g. Iterative policy evaluation

Policy improvement Generate $\pi' \geq \pi$
e.g. Greedy policy improvement

MC Policy Improvement

- Policy improvement is done by making the policy greedy with respect to the current value function. Here we limit ourselves to an action-value function, and therefore no model (we are model-free after all) is needed to construct the greedy policy.
- For any action-value function $Q(s, a)$, the corresponding greedy policy is the one that, for each $s \in \mathcal{S}$ deterministically chooses an action with maximal action-value:

$$\pi(s) = \arg\max_a Q(s, a)$$

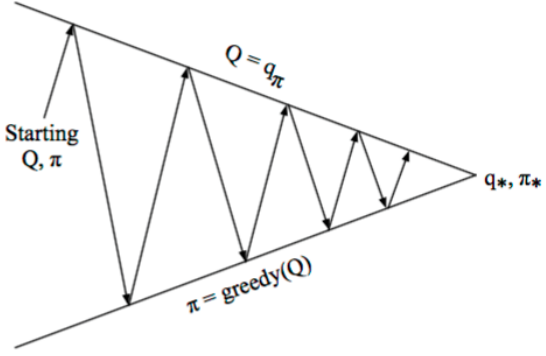
- Policy improvement then can be done by constructing each π^{k+1} as the greedy policy with respect to Q^{π_k}.

Working with V or Q implies model-knowledge

- Greedy policy improvement over $V(s)$ requires model-knowledge of MDP: $\pi'(s) = \arg\max_{a \in \mathcal{A}} \mathcal{R}_{sa}^{s'} + \mathcal{P}_{ss'}^{a} V(s')$
- Greedy policy improvement over $Q(s, a)$ is model-free: $\pi'(s) = \arg\max_{a \in \mathcal{A}} Q(s, a)$

  In the previous lectures we considered transitions from state to state and learned the values of states. Now we consider transitions from state-action pair to state-action pair, and learn the value of state-action pairs. Formally these two cases are identical: they are both Markov chains with a reward process.

Generalised Policy Improvement on State-Action Value function



Policy evaluation Monte-Carlo policy evaluation, $Q = q_\pi$

Policy improvement Greedy policy improvement?

Exploring starts

How do we know that we have collected enough data to estimate the value of a state? For the moment, let us assume that we do observe an infinite number of episodes. Also, we assume we start in many different initial states. i.e. by performing exploring starts (start in random states).
Thus, many episodes are experienced, with the approximate action-value (Q) function approaching the true function asymptotically. Under these assumptions, Monte Carlo methods will compute each V^{π} or Q^{π} exactly, for arbitrary π.

MC Policy Improvement - Proof sketch I

The policy improvement theorem we already encountered in DP applies also here, but we need to take care that we deal with mean return being now replaced by an average empirical return. Therefore, for π_k and π_{k+1} because, for all $s \in \mathcal{S}$:

$$
\begin{aligned}
Q^{\pi_k}(s, \pi_{k+1}(s)) &= Q^{\pi_k}(s, \arg\max_a Q^{\pi_k}(s, a)) && (40)\\
&= \max_a Q^{\pi_k}(s, a) && (41)\\
&\geq Q^{\pi_k}(s, \pi_k(s)) && (42)\\
&= V^{\pi_k}(s) && (43)
\end{aligned}
$$

Thus, each π^{k+1} is better or just as good as π^k, in the letter case they are optimal policies. This in turn assures us that the overall process converges to the optimal policy and optimal value function.

MC Policy Improvement - Proof sketch II

In this way Monte Carlo methods can be used to find optimal policies given only sample episodes and no other knowledge of the environment's dynamics.

Proof Assumptions revisited I

We made two assumptions in the proof:

1. policy evaluation can be done with an infinite number of episodes.
2. episodes have exploring starts to ensure we experience the full world.

For practical algorithms we will need to remove both assumptions. The first assumption is to relax the infinite number of episode assumption.

a. as we have already seen for DP the iteration converges within specific bounds, and so MC will also converge within finite limits up to a bound.

Proof Assumptions revisited II

- we do value iteration, i.e. in which only one iteration of iterative policy evaluation is performed between each step of policy improvement.

  For Monte Carlo policy evaluation it is natural to alternate between evaluation and improvement on an episode-by-episode basis. After each episode, the observed returns are used for policy evaluation, and then the policy is improved at all the states visited in the episode.

Exploring Starts vs Starting to Explore

How can we avoid the unlikely assumption of exploring starts? The only general way to ensure that all actions are selected infinitely often is for the agent to continue to select them.

1. on-policy methods, which attempt to evaluate or improve the policy that is used to make decisions
2. off-policy methods, that evaluate or improve a policy different from that used to generate the data.

} Policy used to explore & policy you improve are the same

With off-policy / soft policy methods, all actions are executable

On-Policy vs Off-Policy Learning

- On-policy learning is "Learn on the job"
- Learn about policy π from experience sampled from π

- Off-policy learning is "Look over someone's shoulder"
- Learn about policy π from experience sampled from π'

On-Policy Methods: Soft control

Definition

Soft policies have in general $\pi(a, s) > 0 \; \forall s \in \mathcal{S}, \forall a \in \mathcal{A}$. I.e. we have a finite probability to explore all actions.

↳ Actions must be executable

ϵ-greedy policies

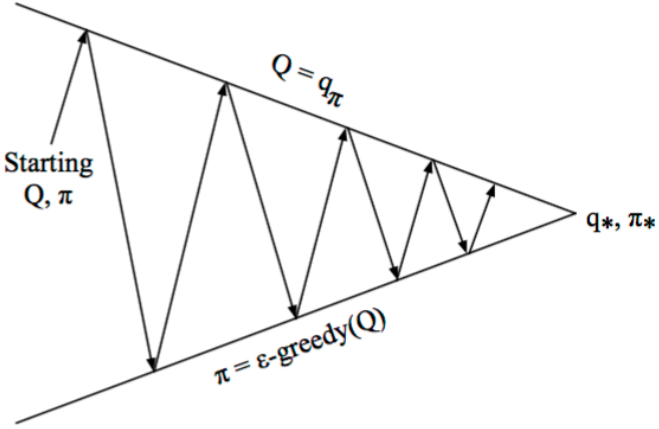
ϵ-greedy policies are a form of soft policy, where the greedy action $a*$ (as selected from being greedy on the value or action-value function and choosing the arg max action) has a high probability of being selected, while all other actions available in the state, have an equal share of an ϵ probability (that allows us to explore the non-greedy/ optimal action).

Definition

ϵ-greedy policy with $\epsilon \in [0, 1]$.

$$\pi(s, a) = \begin{cases} 1 - \epsilon + \frac{\epsilon}{|A(s)|}, & \text{if } a^* = \underset{a}{\text{argmax}} Q(s, a) \\ \frac{\epsilon}{|A(s)|}, & \text{if } a \neq a^* \end{cases} \tag{44}$$

Soft policy policy improvement



Policy evaluation Monte-Carlo policy evaluation, $Q = q_\pi$

Policy improvement ϵ-greedy policy improvement

On-policy ϵ-greedy first-visit MC control algorithm

Initialize, for all $s \in \mathcal{S}$, $a \in \mathcal{A}(s)$:
$Q(s,a) \leftarrow$ arbitrary
$Returns(s,a) \leftarrow$ empty list
$\pi(a|s) \leftarrow$ an arbitrary ε-soft policy

Repeat forever:
(a) Generate an episode using π
(b) For each pair s, a appearing in the episode:
$G \leftarrow$ return following the first occurrence of s, a
Append G to $Returns(s,a)$
$Q(s,a) \leftarrow$ average$(Returns(s,a))$
(c) For each s in the episode:
$a^* \leftarrow \arg\max_a Q(s,a)$
For all $a \in \mathcal{A}(s)$:

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

Convergence of exploratory MC algorithms

Definition

Greedy in the Limit with Infinite Exploration (GLIE)

1. All state-action pairs are explored infinitely many times, $\lim_{k\to\infty} N_k(s,a) = \infty$
2. The policy converges on a greedy policy, $\lim_{k\to\infty} \pi_k(a,s) = (a == \arg\max_{a'\in\mathcal{A}} Q_k(s,a'))$

where the infix comparison operator $==$ evaluates to 1 if true and 0 else.

GLIE basically tells us when a schedule for adapting the exploration parameter is sufficient to ensure convergence.

Example

The ϵ-greedy operation is GLIE if ϵ reduces to zero with $\epsilon_k = \frac{1}{k}$.

MC Batch Learning to Control

MC methods can batched, when the transitions are collected and then the estimates are computed once based on a large number of transitions.

procedure MONTECARLOBATCHOPTIMISATION(n)
  Init
    $\hat{Q}(s,a) \leftarrow$ arbitrary value, for all $s \in S, a \in A$.
    $\pi \leftarrow \varepsilon$-*greedy*$(\hat{Q})$
  EndInit
  repeat (For each batch)
    Returns$(s,a) \leftarrow$ an empty list, for all $s \in S$.
    for $i = 1$ **to** n **do**
      Get trace, τ, using π.
      for all (s,a) appearing in τ **do**
        $R \leftarrow$ return from first appearance of (s,a) in τ.
        Append R to *Returns*(s,a)
    for all $s \in S, a \in A$ **do**
      $\hat{Q}(s,a) \leftarrow$ average(*Returns*(s,a))
    $\pi \leftarrow \varepsilon$-*greedy*$(\hat{Q})$
  until Done
  return *greedy*$(\hat{Q})$

MC Iterative Learning to Control

MC methods can be iterative, when after each transition the memory is changed and the transition is thrown away

procedure MONTECARLOITERATIVEOPTIMISATION(n)
  Init
    $\hat{Q}(s,a) \leftarrow$ arbitrary value, for all $s \in S, a \in A$.
    $\pi \leftarrow \varepsilon$-*greedy*$(\hat{Q})$
  EndInit
  for $i = 1$ **to** n **do**
    Get trace, τ, using π.
    for all (s,a) appearing in τ **do**
      $R \leftarrow$ return from first appearance of (s,a) in τ.
      $\hat{Q}(s,a) \leftarrow \hat{Q}(s_t,a_t) + \alpha \left[R - \hat{Q}(s_t,a_t)\right]$
    $\pi \leftarrow \varepsilon$-*greedy*$(\hat{Q})$
  return *greedy*$(\hat{Q})$

Summary MC control I

1. In designing Monte Carlo control methods we have followed the overall schema of generalised policy iteration (GPI). Rather than use a model to compute the value of each state, they simply average many returns that start in the state. Because a state's value is the expected return, this average can become a good approximation to the value.
2. Maintaining sufficient exploration is an issue in Monte Carlo control methods. It is not enough just to select the actions currently estimated to be best, because then no returns will be obtained for alternative actions, and it may never be learned that they are actually better.

Summary MC control II

3. One approach is to ignore this problem by assuming that episodes begin with state-action pairs randomly selected to cover all possibilities. Such exploring starts can sometimes be arranged in applications with simulated episodes, but are unlikely in learning from real experience.

4. In on-policy methods, the agent commits to always exploring and tries to find the best policy that still explores. In off-policy methods, the agent also explores, but learns a deterministic optimal policy that may be unrelated to the policy followed.

Summary MC control III

5. Off-policy Monte Carlo prediction refers to learning the value function of a target policy from data generated by a different behaviour policy. Such learning methods are all based on some form of importance sampling, that is, on weighting returns by the ratio of the probabilities of taking the observed actions under the two policies.

MC vs TD control

Temporal-difference (TD) learning has several advantages over Monte-Carlo (MC)

- Lower variance
- Online
- Incomplete sequences

Natural idea: use TD instead of MC in our control loop

- Apply TD to $Q(S, A)$
- Use ϵ-greedy policy improvement
- Update every time-step

Sarsa: On-Policy TD Control I

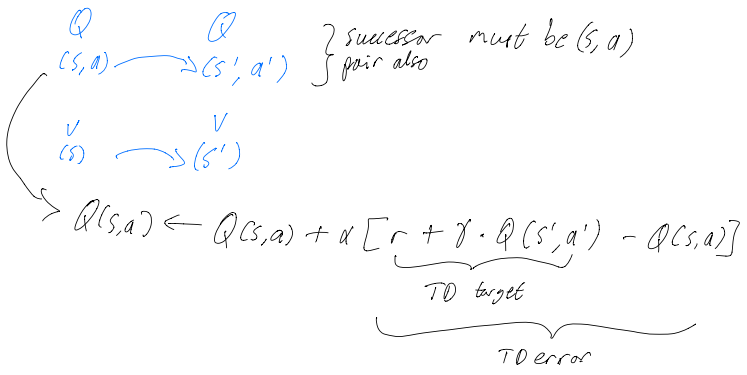
- To derive on-policy TD control, we follow the established pattern of generalised policy iteration (GPI) we encountered for DP and MC, only this time using TD methods for the evaluation or prediction part.
- The first step is to learn an action-value function rather than a state-value function. In particular, for an on-policy method we must estimate $Q^\pi(s, a)$ for the current behaviour policy π and for all states s and actions a. This can be done using essentially the same TD method described above for learning V^π.

$$Q(S, A) \leftarrow Q(S, A) + \alpha\big(r + \gamma Q(S', A') - Q(S, A)\big) \qquad (45)$$

learning rate

SARSA

- The theorems (we touched upon) assuring us the convergence of state values under TD evaluation also apply to state-action pairs.


Q
(s, a)
Q
(s', a')
successor pair also
must be (s, a)
V
(s)
V
(s')
Q(s,a) ← Q(s,a) + α [r + γ · Q(s', a') − Q(s,a)]
TD target
TD error

Sarsa: On-Policy TD Control II

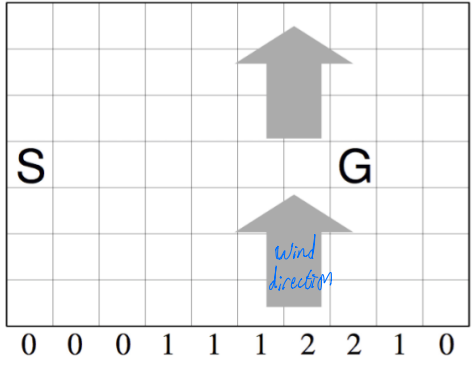
- SARSA reflects that the main function for updating the Q-value depends on the current state of the agent s_1, the action the agent chooses a_1, the reward r_2 the agent gets for choosing this action, the state s_2 that the agent will now be in after taking that action, and finally the next action a_2 the agent will choose in its new state.

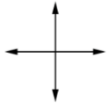
SARSA - On-Policy learning TD control

Initialize $Q(s,a), \forall s \in \mathcal{S}, a \in \mathcal{A}(s)$, arbitrarily, and $Q(\textit{terminal-state}, \cdot) = 0$
Repeat (for each episode):
  Initialize S
  Choose A from S using policy derived from Q (e.g., ε-greedy)
  Repeat (for each step of episode):
    Take action A, observe R, S'
    Choose A' from S' using policy derived from Q (e.g., ε-greedy)
    $Q(S,A) \leftarrow Q(S,A) + \alpha\big[R + \gamma Q(S',A') - Q(S,A)\big]$ Update Q with updated estimates
    $S \leftarrow S'$; $A \leftarrow A'$;
  until S is terminal

You may have noticed, that taking every letter in the quintuple $(s_t, a_t, r_t, s_{t+1}, a_{t+1})$ yields the word SARSA.

SARSA on Windy Grid World I

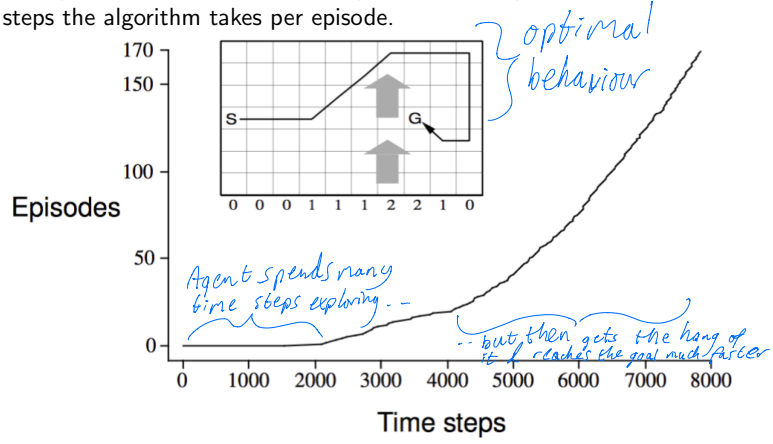




Start from S, reward is -1 per time-step, until reaching terminal state G. Wind is blowing from below with strength 0,1,2 (units of displacement).

SARSA on Windy Grid World I

This graph is a new way of looking at the learning curve, it shows how many steps the algorithm takes per episode.



At the start there is a lot of exploration (long episodes) but very quickly the episode become shorter.

Convergence of SARSA

Theorem

SARSA converges to the optimal action-value function $Q(s,a) \rightarrow Q^{\infty}(s,a)$, *under the following conditions*

1. *GLIE sequence of policies* $\pi^k(a,s)$
2. *Robbins-Monroe sequence of step-sizes* α_t
   1. $\sum_{t=1}^{\infty} \alpha_t = \infty$
   2. $\sum_{t=1}^{\infty} \alpha_t^2 < \infty$

How you choose α so that you converge

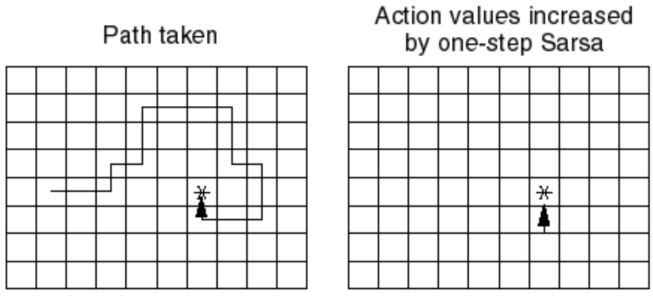
So, we have to control both the exploration ϵ and the learning rate α over time.

Example

What series for α satisfy these conditions?

1. reciprocals of constant positive constants? No, diverges.
2. reciprocals of powers of 2 of the steps ? Yes, converges to 2.

Traces, SARSA and sparse rewards I



We observe: almost all updates did not benefit the Q function, as the reward was sparse. Each time we run the trace we would slowly propagate backward the value through many episodes.
How can we deal with sparse rewards?

- One approach, called SARSA-Lambda uses so called Eligibility traces to propagate sparse rewards through a trace.

correlation of a certain part of the trace with receiving reward

Traces, SARSA and sparse rewards II

- Other approaches, such as Hindsight Experience Replay (HER) convert unsuccessful episodes into artificially rewarded ones (we basically failed to do X, but we have achieved Y)

Off-Policy methods

- We estimated value Q^π given a supply of episodes generated using a policy π.
- Suppose now that all we have are episodes generated from a different policy π'. That is, suppose we wish to estimate Q^π or V^π but all we have are episodes following another policy π'.
- We call π the target policy because learning its value function is the target of the learning process, and we call π' the behaviour policy because it is the policy controlling the agent and generating behaviour.
- The overall problem is called off-policy learning because it is learning about a policy given only experience off (not following) that policy.
- Another mnemonic way to think of is to imagine the cockpit of the agent, and to think if the target policy is switched "on" to run the agent or if it is switched "off" and the agent is left to his own behaviour (policy).

2 seperate policies

What behaviour policies work for off-policy learning?

In order to use episodes from π' to estimate values for π, we must require that every action taken under π is also taken, at least occasionally, under π'. That is, we require that $\pi(a, s) > 0 \Rightarrow \pi'(a, s) > 0$. This is called the assumption of coverage. I.e. coverage requires that π' must be stochastic where it is not identical to π.

One of the most important breakthroughs in reinforcement learning was the development of off-policy TD control algorithm known as Q-learning (Watkins, 1989).

Have 2 policies, one is an ε-greedy version of the other

↳ one is greedy, the other is ε-greedy

Towards Q Learning

- We now consider off-policy learning of action-values $Q(s, a)$
- The next action is chosen using behaviour policy $a_{t+1} \sim \pi'(\cdot|s_t)$ ← action we should've done
- But we consider alternative successor action $a' \sim \pi(\cdot|s_t)$ ←
- Then, we update $Q(s_t, a_t)$ in direction of the value of our alternative (better) action

$$max_a Q(s_{t+1}, a) \tag{46}$$

Q-Learning: Off-Policy TD Control

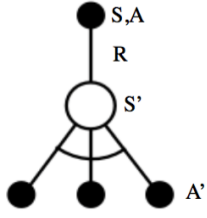
- We now we allow both behaviour π' and target policies π to improve.
- The target policy π is greedy w.r.t. $Q(s, a)$

$$\pi(S_{t+1}) = \arg\max Q(s_{t+1}, a') \tag{47}$$

- The behaviour policy π' is e.g. ϵ-greedy w.r.t. $Q(s, a)$
- The Q-learning target then simplifies too

$$\begin{aligned} & r_{t+1} + \gamma Q(s_{t+1}, a') \\ = \; & r_{t+1} + \gamma Q(s_{t+1}, \arg\max Q(s_{t+1}, a')) \\ = \; & r_{t+1} + \max_{a'} \gamma Q(s_{t+1}, a') \end{aligned}$$

Q-Learning: Off-Policy TD Control



$$Q(S,A) \leftarrow Q(S,A) + \alpha \left(R + \gamma \max_{a'} Q(S',a') - Q(S,A) \right)$$

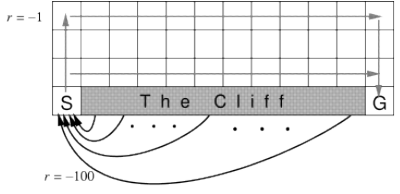
Note: We have a switch in notation in the figures R is immediate return (r)

Q-Learning algorithm

Initialize $Q(s,a), \forall s \in \mathcal{S}, a \in \mathcal{A}(s)$, arbitrarily, and $Q(\textit{terminal-state}, \cdot) = 0$
Repeat (for each episode):
  Initialize S
  Repeat (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

} Q table of state-action pairs

Cliff walking example - SARSA vs Q-Learning



- If you hit the cliff, you die with -100
- Transition costs of -1 otherwise

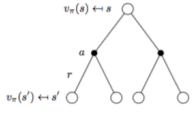
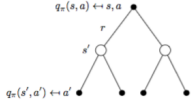
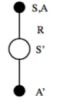
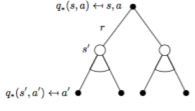
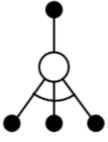
lower because it executes the risky policy but saves the safe path

Reduce α → reduce frequency of ripples

Example: Asynchronous Agents

- Instead of experience replay, train multiple agents in parallel
- Update weights asynchronously resulting in improved exploration
- Experience replay = off-policy training

Summarising

	Full Backup (DP)	*Sample Backup (TD)*
Bellman Expectation Equation for $v_\pi(s)$	 Iterative Policy Evaluation	 TD Learning
Bellman Expectation Equation for $q_\pi(s,a)$	 Q-Policy Iteration	 Sarsa
Bellman Optimality Equation for $q_*(s,a)$	 Q-Value Iteration	 Q-Learning

↳(Not covered)

Summarising

Full Backup (DP)	*Sample Backup (TD)*
Iterative Policy Evaluation	TD Learning
$V(s) \leftarrow \mathbb{E}\left[R + \gamma V(S') \mid s\right]$	$V(S) \overset{\alpha}{\leftarrow} R + \gamma V(S')$
Q-Policy Iteration	Sarsa
$Q(s,a) \leftarrow \mathbb{E}\left[R + \gamma Q(S',A') \mid s,a\right]$	$Q(S,A) \overset{\alpha}{\leftarrow} R + \gamma Q(S',A')$
Q-Value Iteration	Q-Learning
$Q(s,a) \leftarrow \mathbb{E}\left[R + \gamma \max_{a' \in \mathcal{A}} Q(S',a') \mid s,a\right]$	$Q(S,A) \overset{\alpha}{\leftarrow} R + \gamma \max_{a' \in \mathcal{A}} Q(S',a')$

where $x \leftarrow^{\alpha} y$ is defined as $x \rightarrow x + \alpha(y - x)$.