Lecture 5:
Reinforcement Learning: Control

ECS7002P - Artificial Intelligence in Games

Diego Perez Liebana - diego.perez@qmul.ac.uk
Office: CS.301

Game AI Group

https://gaigresearch.github.io/

Queen Mary University of London

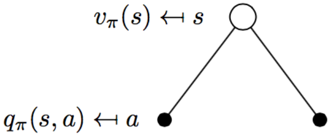
Outline

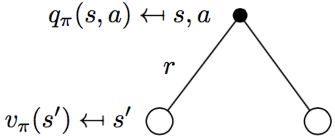
Model Free Control in RL

Exploration versus Exploitation

Advanced Materials: TD(λ) and SARSA

Key reminders

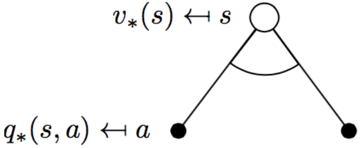


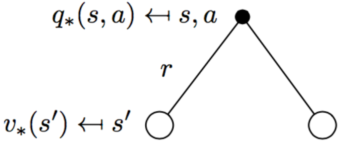


$$v_\pi(s) = \sum_{a \in \mathcal{A}} \pi(a|s) q_\pi(s,a)$$

$$q_\pi(s,a) = \mathcal{R}_s^a + \gamma \sum_{s' \in \mathcal{S}} \mathcal{P}_{ss'}^a v_\pi(s')$$

Key reminders

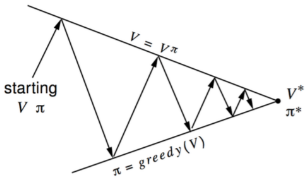


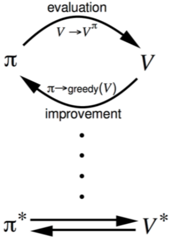


$$v_*(s) = \max_a q_*(s,a)$$

$$q_*(s,a) = \mathcal{R}_s^a + \gamma \sum_{s' \in \mathcal{S}} \mathcal{P}_{ss'}^a v_*(s')$$

Key Reminder - Policy Iteration





How to obtain π^*? We iterate through the following two steps:

1. **Policy evaluation:** Estimate v_π
   Dynamic Programming in Model-based
   (e.g. Iterative policy evaluation)
2. **Policy improvement:** Generate $\pi' \geqslant \pi$
   e.g. Greedy policy improvement

Q? Does Policy Iteration get stuck in Local Maxima? **No.**
Bellman expectation equation has v_π as a `fixed` point (contraction mapping theorem). `https://runzhe-yang.science/2017-10-04-contraction/`

Q? What else can we use to do *Policy Evaluation*?

Reinforcement Learning: Control

Model Free Control in RL

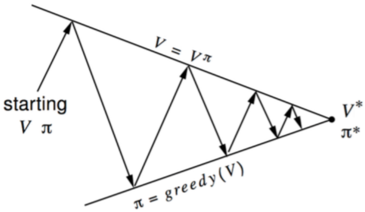
On vs Off-policy

On-policy: learn about policy π using π to sample

- On-policy Monte Carlo Control
- On-policy Temporal-Difference Learning (SARSA)

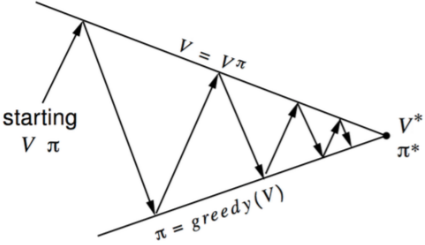
Off-policy: learn about policy π using π' to sample

- Off-policy learning (Q-Learning)



Model-Free Policy Iteration

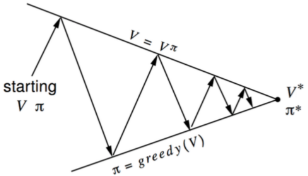
1. **Policy evaluation:** Estimate v_π
   Dynamic Programming (e.g. Iterative policy evaluation)
2. **Policy improvement:** Generate $\pi' \geqslant \pi$
   e.g. Greedy policy improvement



We are going to play with what do we use for Policy Evaluation and Improvement for the agent's behaviour. In **model-free**, can we do?

- Policy Evaluation: Monte-Carlo policy evaluation ($V = v_\pi$)
- Policy Improvement: Greedy policy improvement?

Model-Free Policy Iteration



A first problem: Monte-Carlo Policy Evaluation can't find the true value $V = v_\pi$ in model-free: we do not have full knowledge of $P^a_{ss'}$! Also, **improving** a policy (acting greedily with respect to V) requires the knowledge of the model ($P_{ss'}$):

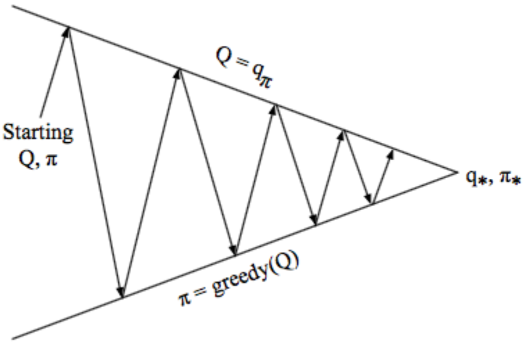
$$\pi'(s) = \arg\max_{a \in A} R^a_s + P^a_{ss'} v(s')$$

The alternative is to use action-value function $Q(s, a)$

$$\pi'(s) = \arg\max_{a \in A} q_\pi(s, a)$$

Therefore, Monte Carlo can aim to approximate $Q_\pi(s, a)$. By caching a $Q_\pi(s, a)$ values (i.e. averaging returns), we can do control in a model-free setting, by picking the action that maximizes these q-values as policy.

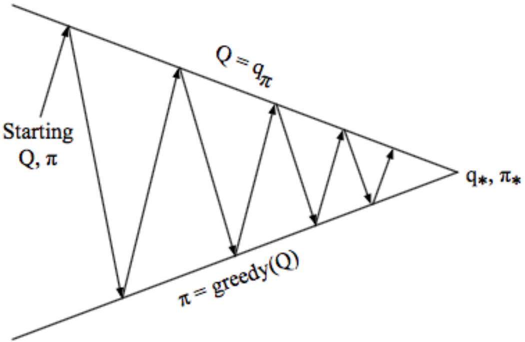
On-policy Monte Carlo Control



- **Policy evaluation:** Monte Carlo policy evaluation, $Q = q_\pi$
- **Policy improvement:** Greedy policy improvement

Q? Is greedy policy the best policy to use?

On-policy Monte Carlo Control



- **Policy evaluation:** Monte Carlo policy evaluation, $Q = q_\pi$
- **Policy improvement:** Greedy policy improvement

Q? Is greedy policy the best policy to use?
No. By acting greedily, we do not explore the search space sufficiently enough.

Q? Why wasn't this a problem before with Dynamic Programming?

ϵ-Greedy Exploration

ϵ-Greedy Exploration:

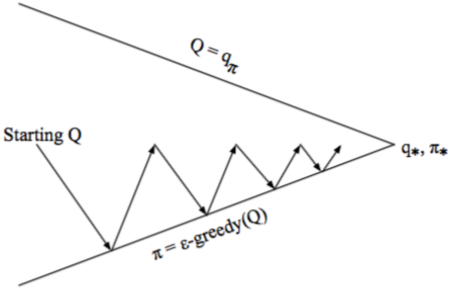
- Simplest idea for ensuring continual exploration
- Ensures all actions are tried with > 0 probability
- With $1 - \epsilon$ probability, choose the greedy action.
- With ϵ probability, choose another action at random.

$$\pi(a \mid s) = \begin{cases} (1-\epsilon) & a^* = \arg\max_{a \in A} q(s,a) \\ (\epsilon) & \text{otherwise.} \end{cases}$$

Monte Carlo Control

- **Policy evaluation:** Monte Carlo policy evaluation, $Q = q_\pi$
- **Policy improvement:** ϵ-Greedy policy improvement

- **Every Episode:** Perform Policy Improvement after every single episode: Collect all the steps during an episode, updating the q-values for the pairs (s,a) visited **only**, and improve the policy straight away.



Greedy in the Limit with Infinite Exploration (GLIE)

At some point, we want to stop exploring and pick the action that maximizes Q(s,a) all the time! We need to find π^*.

A way to do this is to decrease the value of ϵ after each episode, until it reaches 0 (for example, $\epsilon \leftarrow \frac{1}{k}$).

---

procedure GLIE_MC
  Initialize q(s,a) arbitrarily, q(terminal state)$= 0$ ▷ i.e. $Q(s,a) = 0 \ \forall s \in S, a \in A$
  for all $k \in (1 : N)$ **do** ▷ During N iterations of GLIE MC
    Generate an episode using ϵ-greedy policy π (EP_π)
    for all $s, a \in EP_\pi$ **do**
      $N(s_t, a_t) \leftarrow N(s_t) + 1$ ▷ Increment visit counter
      $Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \frac{1}{N(s_t, a_t)}(G_t - Q(s_t, a_t))$ ▷ Update the Q-value of this pair (s_t, a_t)
    end for
    $\epsilon \leftarrow \frac{1}{k}$
  end for
end procedure

GLIE Monte Carlo Control converges to the optimal action-value function:

$$Q(s,a) \rightarrow q^*(s,a)$$

Note: the algorithms improves $Q(s,a)$, which is used by the policy: policy improvement after every episode.

Off-policy Learning

Off-policy learning uses two different policies:

- Target policy ($\pi(a \mid s)$): policy that is being evaluated and improved.
- Behaviour policy ($\mu(a \mid s)$) used to sample the MDP, generating the sequence of $\{S_1, A_1, R_1, \ldots, S_T\}$.

Why?

Off-policy Learning

Off-policy learning uses two different policies:

- Target policy ($\pi(a \mid s)$): policy that is being evaluated and improved.
- Behaviour policy ($\mu(a \mid s)$) used to sample the MDP, generating the sequence of $\{S_1, A_1, R_1, \ldots, S_T\}$.

Why?

- Learn from other agents, even from humans.
- Re-use old policies used in the past $(\pi_1, \pi_2, \ldots, \pi_{t-1})$.
- Learn about an *optimal* policy while following an *exploration* policy.
- Learn about *multiple* policies while following *one* policy.

Q-Learning

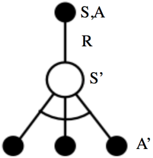
Q-Learning is probably the most famous Off-policy Learning in RL:

- Target policy ($\pi(a \mid s)$) is the greedy policy:

$$\pi(s_{t+1}) = \arg\max_{a' \in A} Q(s_{t+1}, a')$$

- Behaviour policy ($\mu(a \mid s)$) is the ϵ-greedy policy.
- Both policies improve on each iteration of the algorithm.
- We are learning action-values ($Q(s, a)$):

$$Q(s, a) \leftarrow Q(s, a) + \alpha(R + \gamma \max_{a'} Q(s', a') - Q(s, a))$$



We are trying to learn how to act optimally (target policy) while exploring (using the behaviour policy).

Q-Learning

procedure QLearning
  Initialize q(s,a) arbitrarily, q(terminal state)$= 0$ ▷ i.e. $Q(s,a) = 0 \ \forall s \in S, a \in A$
  for all $k \in (1:N)$ **do** ▷ During N Episodes of QLearning
    for all $s \in EP_\pi$ **do** ▷ For all states in the episode
      $s' \leftarrow$ Choose an action a using the ϵ-Greedy policy, $\pi(s)$ (derived from $Q(S,A)$).
      Determine the *target* to learn from with the max q-value: $R + \gamma \max_{a'} Q(s', a')$
      Update $Q(s,a)$: $Q(s,a) \leftarrow Q(s,a) + \alpha(R + \gamma \max_{a'} Q(s', a') - Q(s,a))$
      $s \leftarrow s'$
    end for
     Until s is terminal
  end for
end procedure

Q-Learning converges to the optimal action-value function:

$$Q(s,a) \to q^*(s,a)$$

Reinforcement Learning: Control

Exploration versus Exploitation

Exploration vs. Exploitation Dilemma

As we have seen in this and previous lectures, selecting actions involves a fundamental choice:

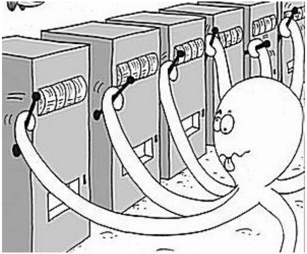
- *Exploitation*: Make the best decision based on current information.
- *Exploration*: Gather more information about the environment. This is: not choosing the best action found so far.

The objective is to gather enough information to make the best overall decision. The best long-term strategy may involve short-term sub-optimal selections.

There are different ways to explore:

- Random exploration: ϵ-greedy, Softmax, ...
- Optimism in the face of uncertainty: estimate the uncertainty of a value, and prefer to explore those with higher uncertainty.

The Multi-Armed Bandit Problem



- A multi-armed bandit is a tuple $< A, R >$.
- A is a known set of actions (arms).
- Set of unknown distributions $\{R_1, R_2, \ldots, R_k\}$ of rewards, one per action.
- Played iteratively, during H action selections.
- Mean values of these reward distributions: $\{\mu_1, \mu_2, \ldots, \mu_k\}$
- The goal is to maximize the sum of rewards (minimizing the loss).
- Each action is pulling one lever. How do you choose?

Regret

Regret is the opportunity loss (total, or for one step). How much did I lose because I did not choose the optimal action?

- Given the action value $Q(a)$ and the optimal value V^*

$$Q(a) = \mathbb{E}[r \mid a] \qquad\qquad V^* = Q(a*) = \max_{a \in A} Q(a)$$

  - The **regret** is the opportunity loss for one step:

$$l_t = \mathbb{E}[V^* - Q(a)]$$

  - The **total regret** is the total opportunity loss:

$$L_t = \mathbb{E}[\sum_{t=1}^{T}(V^* - Q(a))]$$

- The objective is to minimize the total regret, which maximizes the cumulative reward.

Regret: gaps and counts

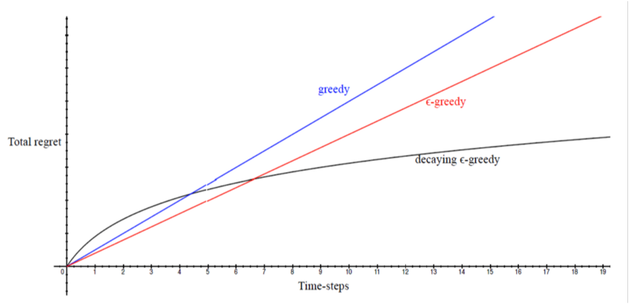
How do we count regret?

- The *count* $N_t(a)$ is the expected number of selections of action a.
- The *gap* Δ_a is $V^* - Q(a)$: difference in value between picking a and a^*.
- Total regret can be expressed as a function of *gaps* and *counts*:

$$
\begin{aligned}
L_t &= \mathbb{E}[\sum_{t=1}^{T}(V^* - Q(a))] \\
&= \sum_{a \in A} \mathbb{E}[N_t(a)](V^* - Q(a)) \\
&= \sum_{a \in A} \mathbb{E}[N_t(a)]\Delta_a
\end{aligned}
$$

- Therefore, a good algorithm produces small *counts* for large *gaps*, and viceversa, in order to minimize the total regret (L_t).
- **Q?** What's the problem?

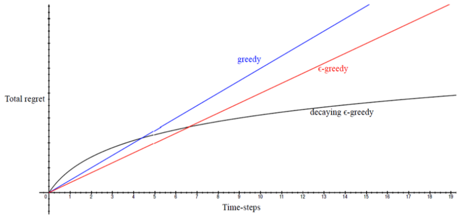
Linear and Sublinear Regret



- Greedy: the algorithm never explores, the total regret is linear.
- Greedy with optimistic initialization
  - Initialize all $Q(a)$ to the maximum possible reward, then act greedily.
  - Still greedy, total regret is linear.
- (Constant)-ϵ greedy. It never stops exploring, hence the total regret is linear.
- (Decaying)-ϵ greedy: reduces slowly the value of ϵ at each step, it achieves sublinear regret.

Logarithmic regret

Can we do better?



- (Decaying)-ϵ greedy can achieve logarithmic regret ... **if we know the gaps in advance**, with the following decaying schedule:

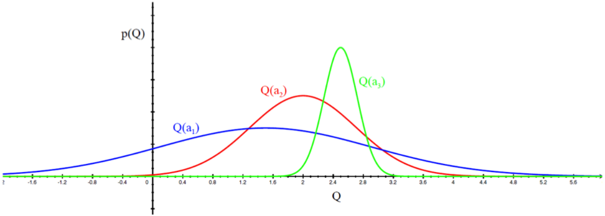
$$c > 0 \qquad d = \min_{a|\Delta_a > 0} \Delta_i$$

$$\epsilon_t = min\{1, \frac{c \mid A \mid}{d^2 t}\}$$

- Logarithmic regret is actually the best we can do!
- **Goal**: an algorithm with logarithmic regret without knowing the gaps (Δ).

Bounds

Optimism in the face of uncertainty.



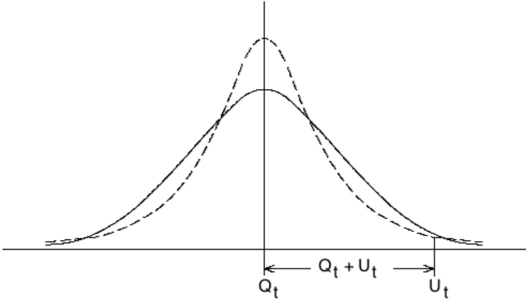
- Which action should we pick? The more uncertain we are about an state-action value, the more important is to explore that action.
- After picking an action, we are less uncertain about it, and more likely to pick another action.
- We keep until we build confidence on the action-value of each action.
- We know how to calculate the action-value, but how do we build the confidence?

Building the Bounds

How do we build this confidence?

- Estimate an upper confidence $U_t(a)$ for each action value that depends on the number of times a has been selected ($N(a)$).
  - Small $N(a)$: large $U_t(a)$ that implies uncertainty.
  - Large $N(a)$: small $U_t(a)$ that implies more accuracy.
- The action must be selected maximizing the Upper Confidence Bound (UCB):

$$a_t = \arg\max_{a \in A} \{Q_t(a) + U_t(a)\}$$



Upper Confidence Bounds

Theorem (Hoeffding's Inequality)

Let $X_1, X_2, \ldots, X_t$ be identically and independently distributed random variables in $[0, 1]$, and let $\bar{X}_t = \frac{1}{\tau}\sum_{\tau=1}^{t} X_\tau$ be the sample mean. Then:

$$\mathbb{P}[\mathbb{E}[X] > \bar{X}_t + u] \leq e^{-2tu^2}$$

This means: "What is the probability that the difference between the empirical and the actual mean is greater than u?". Or, in other words, "What is the probability of making a mistake greater than u when estimating the mean?". This theorem says that this probability is no more than e^{-2tu^2}, for **any distribution**, if the random variables are bounded in $[0, 1]$. In our case:

$$\mathbb{P}[Q(a) > Q_t(a) + U_t(a)] \leq e^{-2N_t(a)U_t(a)^2}$$

Deriving from this, we obtain that, when $t \to \infty$: $U_t(a) = \sqrt{\frac{2log(t)}{N_t(a)}}$

UCB1

This leads to the UCB1 algorithm:

$$a_t = \arg\max_{a \in A} Q(a) + \sqrt{\frac{2 log(t)}{N_t(a)}} = \arg\max_{a \in A(s)} Q(s,a) + C\sqrt{\frac{ln\ N(s)}{N(s,a)}}$$

- $Q(s,a)$: Action-state value of action a from state s.
- $N(s)$: Times the state s has been visited.
- $N(s,a)$: Times the action a has been selected from state s.
- C: balances between *exploitation* and *exploration*:
  - Value of C is application dependent.
  - Example: single player games with rewards in $[0,1]$: $C = \sqrt{2}$.

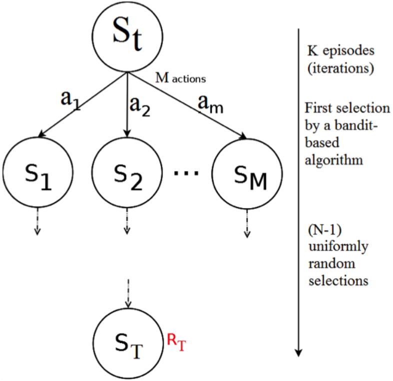
- UCB1 achieves **logarithmic** total regret.
- We don't need to know the gaps.

- There are many UCB variants, UCB1 is just one of them.
- Other theorems derive other UCB policies.

Flat UCB

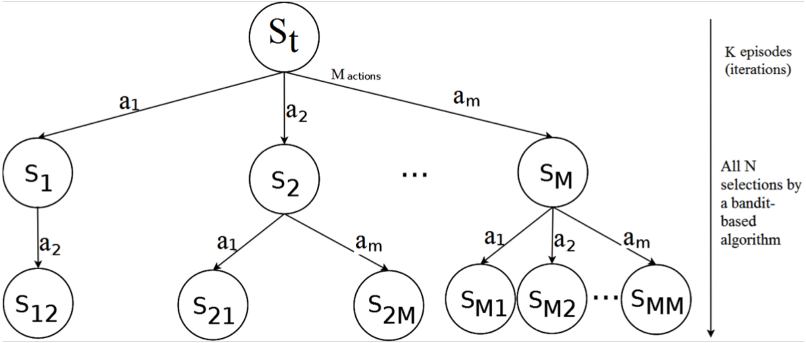
We can use UCB1 (or any other UCB policy) for searching the action-state space. For example, the *Flat UCB* algorithm:

- Iteratively, apply K episodes. For each one of them:
- Select the first action from S_t with UCB.
- Pick actions uniformly at random until reaching a terminal state (**roll-out**).
- This estimates state-action values $Q(s, a)$ from the state S_t.
- Note that the UCB policy improves at each episode.



Building a tree

By applying a UCB policy, we can add a node (that represents a state) at each iteration:



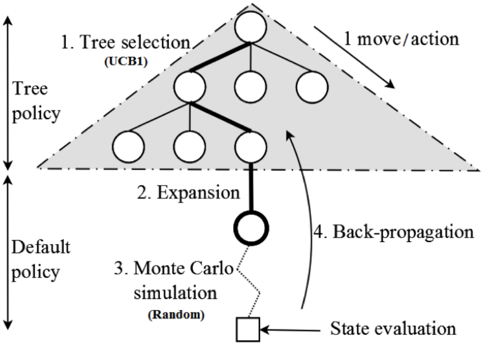
- The tree grows asymmetrically, towards the most promising parts of the search space.
- However, this is limited by how far can we look ahead into the future.
- If we add a node for each state visited during the random roll-outs, the tree would be too big!

Monte Carlo Tree Search (MCTS)

Monte Carlo Tree Search: adding Monte Carlo simulations after a new node is added to the tree.

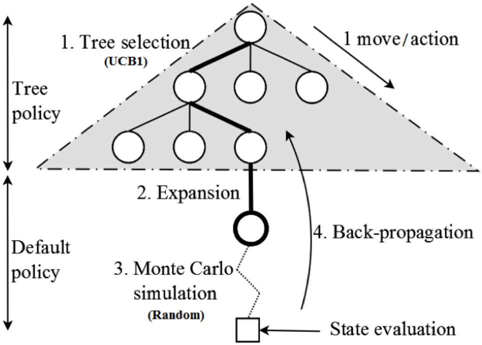
- 2 different policies are used on each episode:
  - **Tree policy** improves on each iteration. It is used while the simulation is in the tree. Some naming conventions:
    - UCT Algorithm: MCTS with any UCB tree selection policy.
    - Plain UCT Algorithm: MCTS with UCB1 as tree selection policy.
  - **Default policy** is fixed through all iterations. It is used while the simulation is outside the tree. Picks actions uniformly at random.
- On each iteration:
  - $Q(s, a)$ on each node in the tree is updated.
  - $N(s)$ and $N(s, a)$ on each node of the tree are updated.
  - Tree policy is based on $Q(s, a)$, thus it improves on each iteration.
- MCTS converges to the optimal search tree.

Monte Carlo Tree Search (MCTS)



1. **Tree Selection:** Following the tree policy (i.e. UCB1), navigate the tree until reaching a node with at least one child state not in the tree (this is, not all actions have been picked from that state in the tree).
2. **Expansion:** Add a new node in the tree, as a child of the node reached in the tree selection step.

Monte Carlo Tree Search (MCTS)

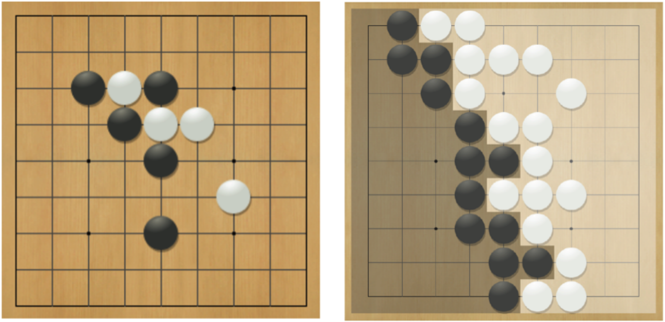


3. **Monte Carlo simulation:** Following the default policy (picking actions uniformly at random), advance the state until a terminal state (or a pre-defined maximum number of steps). The state at the end of the simulation is evaluated (obtain the reward R).
4. **Backpropagation:** Update the values of Q(s,a), N(s) and N(s,a) of the nodes visited in the tree during steps 1 and 2.

Use Case: MCTS and the Game of Go

- 2500 years old 2-player game.
- Considered the hardest classic board game, and a challenge task for AI.
- Traditional game-tree search (minimax, alpha-beta search) has failed in Go: they can't reach human-play level.

Use Case: MCTS and the Game of Go



- Played on different boards: 19×19, 13×13, 9×9
- Black and White stones placed down alternatively
- Surrounded stones are captured and removed
- The player with more territory wins the game
- The rules are simple ... the strategy is not.

Use Case: MCTS and the Game of Go

Why is Go so difficult?

- The game is long (average of 200 moves; Chess: 40-50).
- Large branching factor (average of 250 legal plays/move; Chess: 35-40).
- But, primarily, lack of a good state evaluation function. It is not easy to evaluate how good or bad an intermediate state is:
  - A piece placed early in the game may have a strong influence later in the game, even if it will eventually be captured.
  - It can be impossible to determine if a group will be captured without considering the rest of the board.
  - Most positions are dynamic (there are always unsafe stones in the board).

How to approach it?

- Domain knowledge: find patterns in the board that represent strong plays.
- Use MCTS enhancements: AMAF, RAVE.
- Use delayed rewards.

- From 2016: MCTS + Deep Neural Networks (Alpha Go).

Use Case: MCTS and the Game of Go

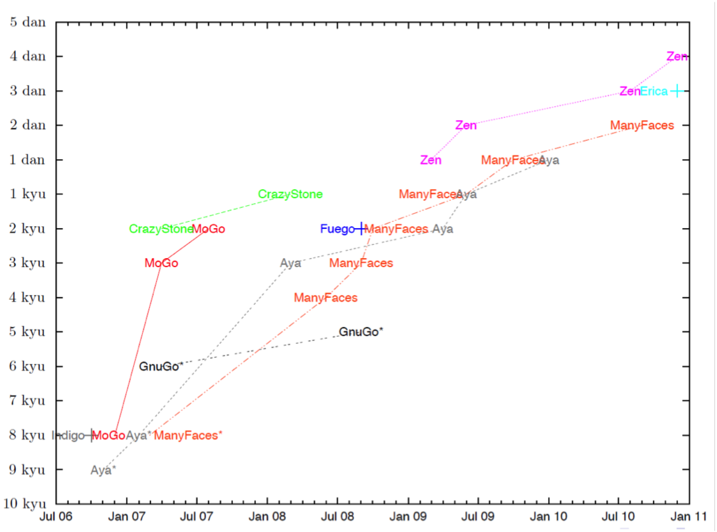
MCTS was the first algorithm to achieve human-play level in the version with the small board.

1990	Abramson demonstrates that Monte Carlo simulations can be used to evaluate value of state [1].
1993	Brügmann [31] applies Monte Carlo methods to the field of computer Go.
1998	Ginsberg's GIB program competes with expert Bridge players.
1998	MAVEN defeats the world scrabble champion [199].
2002	Auer et al. [13] propose UCB1 for multi-armed bandit, laying the theoretical foundation for UCT.
2006	Coulom [70] describes Monte Carlo evaluations for tree-based search, coining the term Monte Carlo tree search.
2006	Kocsis and Szepesvari [119] associate UCB with tree-based search to give the UCT algorithm.
2006	Gelly et al. [96] apply UCT to computer Go with remarkable success, with their program MoGo.
2006	Chaslot et al. describe MCTS as a broader framework for game AI [52] and general domains [54].
2007	CadiaPlayer becomes world champion General Game Player [83].
2008	MoGo achieves *dan* (master) level at 9×9 Go [128].
2009	Fuego beats top human professional at 9×9 Go [81].
2009	MoHex becomes world champion Hex player [7].

Timeline of events leading to the widespread popularity of MCTS.

Use Case: MCTS and the Game of Go

Popular MCTS Go Players:



Use Case: MCTS and the Game of Go

Go, MCTS and Deep Learning:

Acknowledgements

Additional materials: most of the materials for this course are based on the following resources.

- Prof. David Silver's course on Reinforcement Learning:

  http://www0.cs.ucl.ac.uk/staff/d.silver/web/Teaching.html

- *Reinforcement Learning: An Introduction*, by Andrew Barto and Richard S. Sutton (2017 Edition):

  http://incompleteideas.net/book/bookdraft2017nov5.pdf

Misc

All labs from now on: Monday 4-6pm, ITL ground floor.

- That includes the MCQ tests!!

MSc Projects.

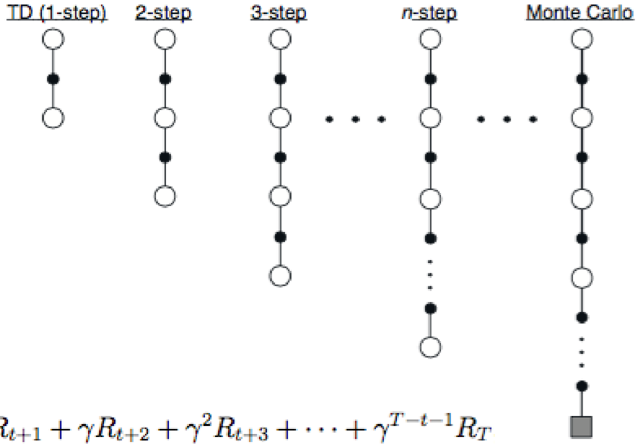
- All projects have been designed aiming to response research questions that can be publishable in AI/Games conferences.
- Forward Models for Statistical Forward Planning methods:
  - Learning FMs, Abstract FMs, Incorrect FMs.
- Implementing and testing AI methods in a table-top board games:
  - SFP for wargames, automatic AI scripting, competition and cooperation in table-top board games.
- Designing and implementing a game description language:
  - Table-top board games, tile-based games, VGDL 3.0.

Reinforcement Learning: Control

Advanced Materials: TD(λ) and SARSA

n-step Prediction

(the rest of this slide deck will not feature in the progress/MCQ test)



$$G_t = R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \cdots + \gamma^{T-t-1} R_T$$

n-step Return

The n-step returns look like this:

$n = 1$ (TD)	$G_t^{(1)} = R_{t+1} + \gamma V(S_{t+1})$
$n = 2$	$G_t^{(2)} = R_{t+1} + \gamma R_{t+2} + \gamma^2 V(S_{t+2})$
$n = 3$	$G_t^{(3)} = R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \gamma^3 V(S_{t+3})$
...	...
$n = \infty$	$G_t^{(\infty)} = R_{t+1} + \gamma R_{t+2} + \cdots + \gamma^{T-1} R_T$

Therefore, we can define:

Definition (n-step Return)

$$G_t^{(n)} = R_{t+1} + \gamma R_{t+2} + \cdots + \gamma^{n-1} R_{t+n} + \gamma^n V(S_{t+n})$$

And:

Definition (n-step temporal-difference learning)

$$V(S_t) \leftarrow V(S_t) + \alpha(G_t^{(n)} - V(S_t))$$

Averaging n-step returns

Backups can be done not just toward any n-step return, but also toward any average of n-step returns. For instance, an average of a 2-step and 4-step return would be:

$$\frac{1}{2}G^{(2)} + \frac{1}{2}G^{(4)} = \frac{1}{2}G_t^{t+2}(V_t(S_{t+2})) + \frac{1}{2}G_t^{t+4}(V_t(S_{t+4}))$$

Q? Can we combine information from all time-steps?

TD(λ): averages n-step backups, weighting each one of them proportionally to λ^{n-1} ($\lambda \in [0,1]$) and normalized by $(1-\lambda)$ so the sum of weights $= 1$.

Definition (λ-Return)

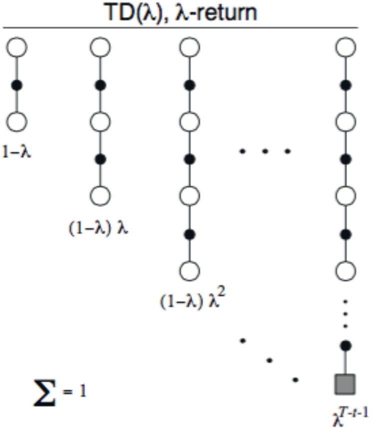
$$(1-\lambda)\sum_{n=1}^{\infty}\lambda^{n-1}G_t^n$$

λ-return

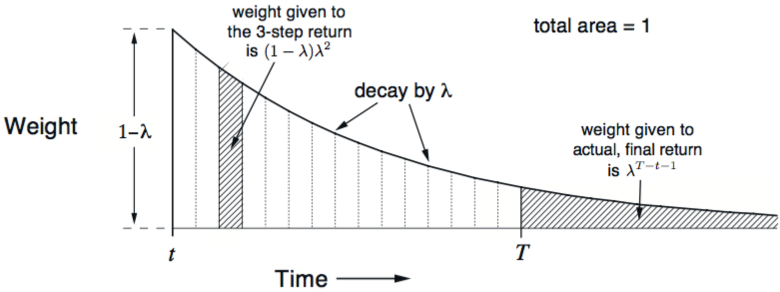
$$(1-\lambda)\sum_{n=1}^{\infty}\lambda^{n-1}G_t^n$$

$\lambda = 0$: the overall backup reduces to the **first** component $\rightarrow$ TD(0)

$\lambda = 1$: the overall backup reduces to the **last** component $\rightarrow$ MC



λ-return

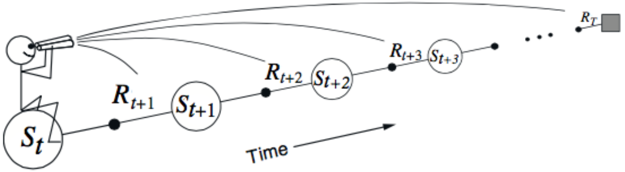


λ-**Return**: $G_t^\lambda = (1-\lambda)\sum_{n=1}^{\infty} \lambda^{n-1} G_t^{(n)}$ (combines all n-step returns $G_t^{(n)}$)

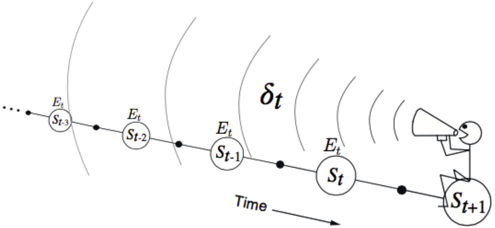
Forward View TD(λ**)**: $V(S_t) \leftarrow V(S_t) + \alpha(G_t^\lambda - V(S_t))$

Forward vs. Backward view

Forward view TD(λ): This is the TD view we have seen so far. It suffers from the same problems as MC (it's computed from complete episodes).

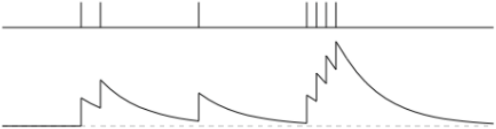


Backward view TD(λ) provides a way to do this with incomplete sequences. We need **eligibility traces**.



Backward view TD(λ)

Eligibility traces combine credit to most frequent and most recent states simultaneously:



Definition (Eligibility traces $E(s)$)

$$E_0(s) = 0$$

$$E_t(s) = \gamma\lambda E_{t-1}(s) + \mathbf{1}(S_t = s)$$

We keep an eligibility trace for every state s.

- Every time a state is visited, the eligibility trace increases.
- Eligibility traces decrease continuously. If a state receives no visits, it will decay up to a minimum.
- λ determines how rapidly the trace decays.

TD(λ)

TD(λ) updates value $v(s)$ for every state s, in proportion to TD-error δ_t and the eligibility trace $E_t(s)$.

$$\delta_t = R_{t+1} + \gamma v(S_{t+1}) - v(s_t);$$

$$E_t(s) = \gamma \lambda E_{t-1}(s) + \mathbf{1}(S_t = s)$$

$$v(s) \leftarrow v(s) + \alpha \delta_t E_t(s)$$

If $\lambda = 0$, $E_t(s) \leftarrow \mathbf{1}(S_t = s)$. This is an automatic, abrupt decay. Thus, we only care about the current state, so $v(s) \leftarrow v(s) + \alpha\delta_t$, which is equivalent to TD(0).

If $\lambda = 1$, there is no decay, we care about all states visited. This is (roughly) equivalent to every-visit MC.

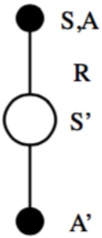
If $0 < \lambda < 1$, less credit is given to δ_t errors from the past. The closer λ to 0, the more abrupt the decay of $E_t(s)$ becomes, and past δ_t errors have less effect on the update of $v(s)$.

SARSA

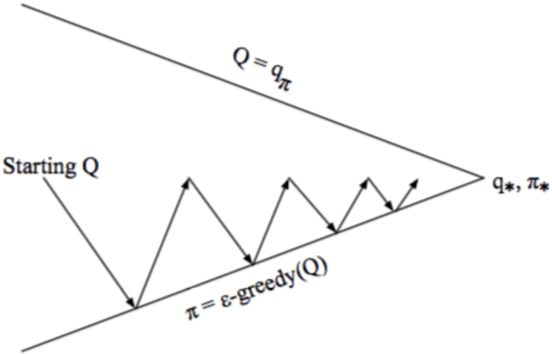
Same concept as in GLIE-MC, but using TD instead of MC.

This removes the need from simulating until the end of the episode.

$$Q(S,A) \leftarrow Q(S,A) + \alpha(R + \gamma Q(S',A') - Q(S,A))$$



SARSA



- **Policy evaluation:** Sarsa, $Q = q_\pi$
- **Policy improvement:** ϵ-Greedy policy improvement

- **Every Episode:** Perform Policy Improvement after every single episode.

SARSA

procedure SARSA
  Initialize q(s,a) arbitrarily, q(terminal state)$= 0$ ▷ i.e. $Q(s,a) = 0 \; \forall s \in S, a \in A$
  for all $k \in (1:N)$ **do** ▷ During N iterations of SARSA
    for all $s \in S$ **do** ▷ For all states
      Choose a from s using ϵ-Greedy policy, $\pi(s)$.
      Take action a, observe R and s'.
      Choose a' from s' using ϵ-Greedy policy, $\pi(s)$.
      $Q(S,A) \leftarrow Q(S,A) + \alpha(R + \gamma Q(S',A') - Q(S,A))$
    end for
  end for
end procedure

SARSA converges to the optimal action-value function:

$$Q(s,a) \rightarrow q^*(s,a)$$

n-step SARSA, SARSA(λ), $\cdots \rightarrow$ same degree of control between MC Control and SARSA by tuning λ.