Reinforcement Learning

Bart Lammers

May 2022

Contents

1 Markov Decision Process

- S: state space s_0, s_1, s_2
- P_0: starting state distribution
- A: Action space a_0, a_1, a_2
- P: transition function $P(s_{t+1}|a_t, s_t)$
- R: reward function $R(s_t, a_t, s_{t+1}) = r_t$

2 RL concepts

Policy

- Sample the action to take from the policy function given the state
- $\mu(s_t) \rightarrow a_t$: deterministic
- $a_t \sim \pi(s_t)$: stochastic

Episode / roll-out / trajectory

- The history (states and actions) of a single run
- $\tau = [s_0, a_0, s_1, a_1, s_2, a_0, ...]$

Return

- Cumulative reward in a trajectory

3 RL course David Silver

Link to teaching material

4 Lecture 1 - introduction

4.1 RL problem

Agent and environment
At each step t the agent:

- Executes action A_t
- Receives observation O_t
- Receives scalar reward R_t

At each step t the environment:

- Receives action A_t
- Emits observation O_t
- Emits scalar reward R_t

This interaction between agent and environment yields a stream of data which is the experience of the agent and the data that the reinforcement learning problem is concerned with.

History - $H_t = A_1, O_1, R_1, ..., A_t, O_t, R_t$

- A: action
- O: observation
- R: reward

History determines what happens next but isn't very useful. More useful is **state**: a summary of the history that contains the information that determines what happens next. The state is a function of the history $S_t = f(H_t)$

Three different versions of state:

- Environment state S_t^e: information private to the environment that captures it's state, whatever it uses to pick the next observation / reward
- Agent state S_t^a: agents internal representation used to pick the next action. Can be any function of the history - we can constrcut this state based on what we need!
- Information state / Markov state / sufficient statistic: contains all useful information from the history. A state is Markov if and only if: $P[S_{t+1}|S_t] = P[S_{t+1}|S_1, ..., S_t]$ (future is independent of the past given the present)

Observability

- Full observability: $O_t = S_t^a = S_t^e$ - this is a Markov Decision Process (MDP)
- Partial observability $S_t^a \neq S_t^e$ - this is a partially observable MDP - agent must construct it's own state. Examples:
  - Complete history (naive)
  - Beliefs on environment state (Bayesian approach) - probability distribution over the states
  - Recurrent neural network: linear combination of new observation O_t and the previous state $S_{t_1}^a$ followed by a non-linearity

4.2 RL agent components

An agent may contain the following components:

- Policy: agent behavior - mapping from state to action
  - Deterministic: $a = \pi(s)$
  - Stochastic: $\pi(a|s) = P[A = a|S = s]$
- Value function: prediction of expected future reward - mapping from state and action to reward
  - Depends on your policy. Value function is always FOR a policy
  - $v_\pi(s) = E_\pi[R_t + \gamma R_{t+1} + \gamma^2 R_{t+2} + ...|S_t = s]$
  - Basis for good decisions because it allows to compare different possible actions based on the expected future rewards that they are going to yield
- Model: predicts what the environment will do next
  - This is optional, there are a lot of "model-free" models
  - We break this up into Transitions and Rewards (immediate rewards)
  - Transitions: $\mathcal{P}^a_{ss'} = P[S' = s'|S = s, A = a]$
  - Rewards: $\mathcal{R}^a_s = P[R|S = s, A = a]$

Categorizing RL agents

Based on policy / value function

- Value based
  - No policy - policy is implicit, it just reads out the value function and picks the action that yields the highest value
  - Has a value function
- Policy based
  - Has a policy - directly determines actions based on state, not through expected values
  - No value function
- Actor critic
  - Combines both together
  - Has a policy and a value function

Based on model / model-free

4.3 Key problems within RL

Sequential decision making - two fundamental problems

- Reinforcement learning - environment is unknown initially and the agent learns about it by interacting - trial and error
- Planning - we know the environment upfront
- These problems are intimately linked. It is not uncommon to first do reinforcement learning and then, once the environment is known, do planning

Exploration vs exploitation

- Exploration learns more about the environment
- Maximize rewards using the information you have already collected

Prediction and control

- Prediction: evaluate the future (given a policy, how well will I do)
- Control: optimize the future (find the best policy)
- Typically we need to solve the prediction problem to then solve the control problem

5 Lecture 2 - Markov Decision Processes

Lecture outline

- Markov processes / Markov chains: vanilla
- Markov reward processes: adding reward in
- Markov decision processes: adding actions in

5.1 Markov processes

- Markov property: current state captures all required information, rest of history can be thrown away
- State transition matrix $\mathcal{P}_{ss'} = P[S_{t+1} = s' | S_t = s]$
- Markov process $(\mathcal{S}, \mathcal{P})$: memoryless random process, ie. a sequence of random states. It is defined by:
  - $\mathcal{S}$ - a finite set of states
  - $\mathcal{P}$ - a transition probability matrix

5.2 Markov Reward Processes

- Markov reward process $(\mathcal{S}, \mathcal{P}, \mathcal{R}, \gamma)$: Markov chain with value judgements (how much reward have I accumulated across this path). In RL we care about maximising the cumulative rewards over time!
  - $\mathcal{R}$ - reward function $\mathcal{R}_s = E[R_{t+1}|S_t = s]$, R_t is the immediate reward at single moment. This function tells us, given that we are in state s, how much reward will we receive in the next timestamp
  - γ - discount factor
- Example of the student Markov chain (three classes, can get distracted by Facebook or the pub), with the Markov reward process we now add values to each state, which are assigned when you enter the state (pass course: +10, take class: -2, go to pub: +3)
- **Return** $G_t = R_{t+1} + \gamma R_{t+2} + ... = \sum_{k=0}^{\infty} \gamma^k R_{t+k+1}$ is the total discounted reward from time step t onwards - can be thought of as the goal, it's what we try to optimize. Using the discount factor γ we make it finite
- Why discount?
  - Because there is more uncertainty into the future, make the model focus on things in its span of control
  - Because it is mathematically convenient to have a finite metric
- **Value function** $v(s) = E[G_t|S_t = s]$ - gives the long-term value of state s
  - In other words: expected return starting from state s
  - Central quantity we are interested in within RL
  - Can estimate by sampling episodes from a MDP. Then for a certain state s calculating the return of each episode. The estimate for the value function is the average of the returns
- Bellman equation - the value function $v(S_t)$ can be decomposed into two parts:
  - Immediate reward R_{t+1}
  - Discounted value of next state $\gamma v(S_{t+1})$
  - Matrix notation: $v = \mathcal{R} + \gamma \mathcal{P} v$, where v is a column vector with each element corresponding to a state, $\mathcal{R}$ is a column vector of immediate rewards and $\mathcal{P}$ is the transition probability matrix
  - $v(s) = \mathcal{R}_s + \gamma \sum_{s' \in \mathcal{S}} \mathcal{P}_{ss'} v(s')$ - can be seen as a "one step look ahead" / one step tree. This is a convenient thing to memorize as we'll also use it during computation: we sum the immediate reward and the weighted average (weights are the probabilities) of the value function outputs in the next states

- Evaluating the rewards can be solved directly for small MDPs since it's a linear equation: $v = (I - \gamma\mathcal{P})^{-1}\mathcal{R}$ with computational complexity $O(n^3)$ due to inverting a matrix with dimensions equal to the transition probability matrix. If we want to maximize rewards (Markov Decision Processes) we cannot solve it anymore analytically

5.3 Markov Decision Processes

- Markov Decision Processes $(\mathcal{S}, \mathcal{A}, \mathcal{P}, \mathcal{R}, \gamma)$ is a Markov reward process with decisions (so adding one more level of complexity: actions $\mathcal{A}$)
  - $\mathcal{A}$ is a finite set of actions
  - The state transition matrix now depends on which action we take: $\mathcal{P}^a_{ss'} = P[S_{t+1} = s' | S_t = s, A_t = a]$
  - The reward function may now depends on the action we take: $\mathcal{R}^a_s = E[R_{t+1} | S_t = s, A_t = a]$
  - Now, in the student MDP example, rather than having all random transitions (according to the probabilities), the agent has some control. It can choose actions in certain states (eg. "study", "Facebook"), but some states still have only random transitions ("if you go to the pub, anything might happen")
- **Policy** $\pi(a|s) = P[A_t = a | S_t = s]$ is a distribution over actions given state. It is a mapping of state to action. It fully defines the behavior of the agent. Due to the Markov property, policies are stationary (time-independent)
- Side note: for any Markov Decision Process, if we fix the policy, we can always "flatten" it into a Markov Process or a Markov Reward Process
- **Value function** - now we have two value functions:
  - The *state-value function*

$$v_\pi(s) = E_\pi[G_t | S_t = s] \tag{1}$$

    now depends on the policy π - it's the expected return, starting from state s, and then following the policy
  - The *action-value function*

$$q_\pi(s, a) = E_\pi[G_t | S_t = s, A_t = a] \tag{2}$$

    is the key quantity we are going to use to determine the best actions! It's the expected return, starting from state s, taking action a, and then following the policy
- Bellman equations for these value functions, again the same decomposition using recursion:

- State-value function:

$$v_\pi(s) = E_\pi[R_{t+1} + \gamma v_\pi(S_{t+1})|S_t = s] \tag{3}$$

- Action-value function:

$$q_\pi(s,a) = E_\pi[R_{t+1} + \gamma q_\pi(S_{t+1}, A_{t+1})|S_t = s, A_t = a] \tag{4}$$

- Averaging over states and actions allows us to estimate these quantities (**lecture minute 50**: useful intuitive explanation on these decompositions using look-ahead trees):
  - State-value function:

$$v_\pi(s) = \sum_{a\in\mathcal{A}} \pi(a|s) q_\pi(s,a) \tag{5}$$

    in words: "average over the actions that we might take" - if we are in state s now, then we take the weighted average of the expected return over all the actions (given by the action-value function $q_\pi(s,a)$), where the weights are the probabilities of taking each action (given by the policy $\pi(a|s)$)
  - Action-value function:

$$q_\pi(s,a) = \mathcal{R}_s^a + \gamma \sum_{s'\in\mathcal{S}} \mathcal{P}_{ss'}^a v_\pi(s') \tag{6}$$

    in words: "average over the states we might end up in" - if we are in state s now and take action a, then sum the immediate reward for taking the action ($\mathcal{R}_s^a$), to the weighted average (weights given by the transition matrix $\mathcal{P}_{ss'}^a$) of values of the states (given by $v_\pi(s')$) we might end up in after taking action a
  - State-value function - stitching together (recursive relation that allows us to understand v in terms of itself, "this is how we end up solving MDPs"):

$$v_\pi(s) = \sum_{a\in\mathcal{A}} \pi(a|s) \left(\mathcal{R}_s^a + \gamma \sum_{s'\in\mathcal{S}} \mathcal{P}_{ss'}^a v_\pi(s') \right) \tag{7}$$

    in words: averaging over both all the actions we might take (the policy gives us the probabilities of the actions) and what the environment might do to us (transition probability matrix gives us the probabilities for each next state given the action)
  - Action-value function - stitching together (recursive relation that allows us to understand q in terms of itself, "this is how we end up solving MDPs"):

$$q_\pi(s,a) = \mathcal{R}_s^a + \gamma \sum_{s'\in\mathcal{S}} \mathcal{P}_{ss'}^a \sum_{a'\in\mathcal{A}} \pi(a'|s') q_\pi(s',a') \tag{8}$$

- Solving an MDP: what is the optimal path (evolution) that will lead to the maximal reward in expectation?
  - Optimal state-value function: $v_*(s) = \max_\pi v_\pi(s)$ — does not yet tell you how to behave — in the example student-MDP, this function tells you what value each of the states (nodes) have
  - Optimal action-value function: $q_*(s,a) = \max_\pi q_\pi(s,a)$ — this one tells you, if I am in state s and I take next action a, what is the expected maximum reward I can get from then on wards. If you have this one you can iterator over the actions and pick the one with the highest maximum expected reward — in the example student-MDP, this function tells you the value of each action (edges)
- **Optimal policy theorem** - "sanity check", what you hoped was true is true - $\pi \geq \pi'$ if $v_\pi(s) \geq v_{\pi'}(s), \forall s$ - for any MDP:
  - There exists an optimal policy, that is better than or equal to all other policies, $\pi_* \geq \pi, \forall \pi$
  - All optimal policies achieve the optimal value function, $v_{\pi_*}(s) = v_*(s)$
  - All optimal policies achieve the optimal action-value function, $q_{\pi_*}(s,a) = q_*(s,a)$
- Finding an optimal policy: at each step pick the action that maximizes $q_*(s,a)$ — there is always a deterministic optimal policy for any MDP

$$\pi_*(a|s) = \begin{cases} 1 & \text{if } a = \underset{a \in \mathcal{A}}{\text{argmax}}\ q_*(s,a) \\ 0 & \text{otherwise} \end{cases} \tag{9}$$

- **Bellman Optimality Equation** - this is the equation generally know as the Bellman equation - "this one tells you how to really solve your MDP" - shows how to relate the optimal value functions to itself by recursion - again intution is the one-step look-ahead charts
  - For the state-value function - instead of before taking the average, we now take the action that maximizes the q value:

$$v_*(s) = \max_a q_*(s,a) \tag{10}$$

  - For the action-value function - incorporates chance, "where the wind may take us", where will the environment take us next - so here we're averaging over transition probabilities again:

$$q_*(s,a) = \mathcal{R}_s^a + \gamma \sum_{s' \in \mathcal{S}} \mathcal{P}_{ss'}^a v_*(s') \tag{11}$$

- Putting those together to get to recursive equation for v_* – two step look-ahead, first over actions (taking the max) then over states we might end up in (averaging over transition probabilities)

$$v_*(s) = \max_a \left(\mathcal{R}_s^a + \gamma \sum_{s' \in \mathcal{S}} \mathcal{P}_{ss'}^a v_*(s') \right) \tag{12}$$

- Similarly to get to a recursive equation for q_* – two step look-ahead, first over transition probabilities in the environment, then over the best actions ($\max_{a'} q_*(s', a')$) we might take from there

$$q_*(s, a) = \mathcal{R}_s^a + \gamma \sum_{s' \in \mathcal{S}} \mathcal{P}_{ss'}^a \max_{a'} q_*(s', a') \tag{13}$$

- How do we solve these Bellman equations now? These functions are now non-linear due to taking the max, so there is no close-form solution. Now we need to resolve to dynamic programming solution that solve it iteratively:
  - Value iteration
  - Policy iteration
  - Q-learning
  - Sarsa
- Extensions to MDPs not discussed in this class:
  - Infinite and continuous MDPs (eg. continuous actions, continuous timestamps)
  - Partially observable MDPs
  - Undiscounted, average reward MDPs

6 Lecture 3 - Dynamic Programming

Lecture outline

- Introduction
- Policy evaluation
- Policy iteration
- Value iteration
- Extensions to dynamic programming
- Contraction mapping

6.1 Introduction

- Policy evaluation, policy iteration and value iteration are three major paradigms. In the next lectures we will "pull" ideas from these areas and develop them into functional tools we can use in RL
- **Dynamic programming**
  - **Dynamic**: sequential / temporal / step-by-step component to the problem
  - **Programming**: from the "optimizing a program" perspective (like linear programming)
  - It's a method of solving complex problems by breaking them into sub-programs and solving those, then putting them back together
  - Dynamic programming can apply to problems with two properties:
    - Optimal substructure: optimal solution can be decomposed into sub-problems
    - Overlapping sub-problems: sub-problems recur many times (**so that we can cache and re-use solutions**)
  - MDPs satisfy both through the Bellman equations. The recursive nature of the value function enables storing and re-using solutions
- Today we are going to use dynamic programming to solve the MDP, and assume full knowledge of the MDP ("someone tells us the transition and reward structure"). This makes it a **planning** problem — this is not yet "full reinforcement learning"
- Can be used for:
  - Prediction / policy evaluation: input MDP $\langle \mathcal{S}, \mathcal{A}, \mathcal{P}, \mathcal{R}, \gamma \rangle$ and policy π - output: value function v_π
  - Control / optimization: input MDP $\langle \mathcal{S}, \mathcal{A}, \mathcal{P}, \mathcal{R}, \gamma \rangle$ - output: optimal value function v_* and optimal policy π_*

6.2 Policy evaluation

- Problem: "prediction" - evaluate a given policy π (in other words: what is the value function v_π?)
- Solution: iterative application of **Bellman expectation equation** in a backup computation
- Synchronous back-up: at each iteration update all states using Bellman equation (for each state doing a one-step look ahead), and then continue with next iteration. Is proven to converge to v_π. Update: $\mathbf{v}^{k+1} = \mathcal{R}^\pi + \gamma \mathcal{P}^\pi \mathbf{v}^k$ (matrix notation)

- Example with small Small Gridworld, finding out the number of steps to a terminal state by setting the reward from each state to the next to -1, **given** a policy of taking a step up, down, left or right all with 0.25 probability:
  - Starting point for the iterative process is to set a naive value (in this case zero) as the value for each state - it does not matter where you start
  - By iteratively computing the next values for states from the previous value and the values from the bordering states (doing one-step look-aheads) the values converge to the true value function given the policy π
  - Moving towards policy iteration: if we now instead set a greedy policy on these estimate of the value function (always go to the state with the highest value), we see that we quickly arrive at the optimal policy (easy to see from the example), even though we have approximated the value function for a different policy (the random one), and we have only taken three iteration steps. Intuition is that we can use the value function which has been approximated for a different policy to build a new policy that is better than the previous one, by acting greedily

6.3 Policy iteration

- Problem: "control" - determine the optimal policy given the rewards and the system dynamics (transition probabilities)
- Solution: iterative application of **Bellman expectation equation** followed by a **greedy policy improvement**
- Given a policy π, iteratively:
  1. **Evaluate** the policy $v_\pi(s) = E[R_{t+1} + \gamma R_{t+2} + ...|S_t = s]$
  2. **Improve** the policy by acting greedily wrt. v_π (in each state taking the action that maximizes the value function)
- This process always iterates to the optimal policy π_*.
- This concept of repeated (1) evaluation of the value function given a policy and then (2) improvement to by acting greedily on the new estimated value function is used again and again in RL. Guaranteed to converge to the optimal value function v_* and the optimal policy π_*
- You can do multiple iterations of step 1 and then a single step 2 in each iteration. If you do a single step 1 (evaluate) followed by a single step 2 (improve), it relates closely to value iteration.

6.4 Value iteration

- Problem: "control" - determine the optimal policy given the rewards and the system dynamics (transition probabilities)
- Solution: iterative application of the **Bellman optimality equation**
- Note: still, we are doing "planning" and not solving the full RL problem since we see the rewards and transition probabilities as given
- Value iteration works "directly in value space" by iteratively applying the Bellman optimality equation
  - It converges to the value function for the **optimal policy**
  - It does not build the policy as an intermediate step. In value iteration you do a single evaluate step, followed by a single greedy improve step over the new value function (argmax in policy iteration gives you the policy, which step to take), but it does so in a single step (by taking a max, giving the next optimal value)
  - There is no explicit policy, and in intermediate steps it may not correspond to any policy, they are just intermediate constructs

6.5 Extensions to dynamic programming

- Asynchronous dynamic programming allows to decrease computation cost by selecting which states to update in some way. This still converges to the correct solution as long as all states are still being selected
- Some flavours:
  - In-place value iteration: don't wait until you have updated the value function for all states, instead just always use the latest values from bordering states, also if they have been updated just before in the current evolution. With some ordering tricks this can be much more efficient
  - Prioritised sweeping: come up with a "priority" metrics that says how important it is to update a state in the MDP? The magnitude of the change in the Bellman equation in the last state updates.
  - Real-time dynamic programming: select the states that the agent actually visits when you apply the agent in the environment ("real world")
- Dynamic programming, due to it's "full width back-ups" suffers from Bellman's curse of dimensionality: the number of states grows exponentially with the number of states. To make this more efficient we can instead sample the back-ups from the dynamics from the environment. This also opens the door for model-free RL because instead of knowing the dynamics, we just sample from them

7 Lecture 4 - Model-Free Prediction

Lecture outline

- Introduction
- Monte-carlo learning
- Temporal-difference learning
- TD(λ)

Very useful slides - not everything is captured in this summary

7.1 Introduction

- Explanation **model free**: so far, we looked at MDPs for which we knew the rewards and dynamics (transition probabilities) given. Now we consider the case where we don't exactly know how the system works "no one tells us about the environment and the agent still needs to learn how to behave optimally".
- Monte-carlo methods: complete a full trajectory and estimate the reward of the trajectory summing the sample returns in the steps
- Temporal-difference methods: look one step ahead and then estimate the total reward of the trajectory, can be much more efficient
- TD(λ) - unification of the above two methods where we can take any number of steps and then estimate
- Overview of last / this / next lecture:
  - Last lecture: planning by dynamic programming - solving **known** MDPs (known reward function and known dynamics)
  - This lecture: model-free prediction - estimating the value function for **unknown** MDPs, given a policy
  - Next lecture: model-free control - optimise the value function for **unknown** MDPs, and therefore find the optimal policy

7.2 Monte-Carlo reinforcement learning

- Goal: learn v_π (expected future return from every state under policy π) from episodes of experience under policy π
- MC is efficient and widely used in practice and is a simple idea that allows for full RL

- Looks at complete episodes, episode must always terminate ("only works for episodic MDPs"). Looks at sample returns to estimate the value function (empirical mean return from each state onwards is the estimate for the value function)
- In these model-free methods, we depend on sampling rather than full sweeps over all states (like in dynamic programming). This is nice because it means that the computation does not depend on the size of the state space any more
- In this lecture we are evaluating a policy, which means we only care about the states that are visited by the policy. In the next lecture, we will discuss optimization. Which means we consider if the policy needs to visit new states and check if it increases our value function. This is the problem of exploration which is a central problem in RL and will be a topic when we discuss optimization only
- Only issue is how do we estimate v for all states if we only have trajectories and can't pick which states we'll visit? Two different solutions:
  - First-visit MC policy evaluation: per state, keep track of 1) the number of visits to that state, and 2) the total return collected across episodes in that state. Then value is estimated by the mean return (sum of returns divided by the number of visits). Note: the counter and total value are only incremented the **first time** the state is visited per episode
  - Every-visit MC policy evaluation: similar to first-visit, but now we increment the total return accumulated in the state and the counter in every state visit in the episode (not just in the first visit). Otherwise identical to first-visit. Both are valid estimators
- These methods can also be used online by computing the mean incrementally after each episode (when we have the return of the episode)

$$\mu_k = \mu_{k-1} + \frac{1}{k}(x_k - \mu_{k-1}) \tag{14}$$

  we'll see many algorithms that use an update statement similar to this
- For Monte-Carlo methods we can use the following incremental updates (after each episode, when we have the return G_t, this way we don't need to maintain old returns and just update the statistics)

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

$$V(S_t) \leftarrow V(S_t) + \frac{1}{N(S_t)}(G_t - V(S_t))$$

- **In the real world**, things can drift around (it's a non-stationary problem), there you always need to forget old episodes by tracking a running mean. You do this by having a constant step size α (this is an exponential moving average):

$$V(S_t) \leftarrow V(S_t) + \alpha(G_t - V(S_t)) \tag{15}$$

7.3 Temporal-Difference learning

- Goal: still policy evaluation (learn v_π online from experience), next class we'll see how we can use TD for control
- Simplest temporal-difference learning algorithm: TD(0)
  - Update $V(S_t)$ towards the estimated return $R_{t+1} + \gamma V(S_{t+1})$:

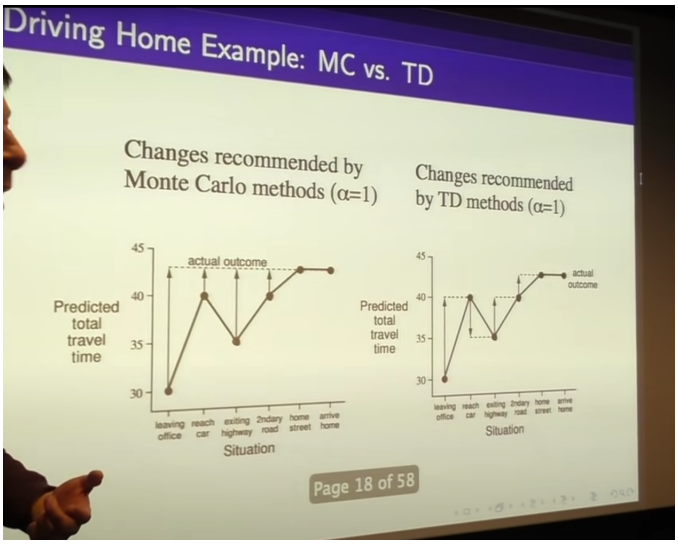
$$V(S_t) \leftarrow V(S_t) + \alpha(R_{t+1} + \gamma V(S_{t+1}) - V(S_t)) \tag{16}$$

  - Difference with every-visit MC is that we substitute the empirical sampled reward G_t with the *TD target* $R_{t+1} + \gamma V(S_{t+1})$
  - The estimated return / TD target again consists of two parts (like in the Bellman equations):
    - R_{t+1} - immediate reward
    - $\gamma V(S_{t+1})$ - discounted value of the next step
- Intuition into why TD works, where does it get it's real-world signal from? Every step you get real rewards from the environment, and the dynamics of the environment affect the state you and up in after you take a certain action. These rewards and dynamics are sampled from the environment and therefore "ground" you in the real-world, and every update step makes the estimated value function approximate the true value function closer and closer
- **Bootstrapping**:
  - Allows learning from incomplete episodes - you update a value function for a state only based on the output of the value function for the surrounding states
  - TD "updates a guess towards a guess" - intuition: start of by guessing the full reward at the begin of the episode (evaluate value function), take a couple of steps, then guess again (evaluate value function), then update the original guess towards you substitute guess
  - MC does not bootstrap (instead it completed the full episode and uses the sampled return to update the value function for the current state)
  - Dynamic programming bootstraps (using the updates defined by the Bellman equations)

  - TD bootstraps

- **Sampling**
  - MC samples
  - Dynamic programming does not sample (calculates the expected values given the rewards and transition probabilities)
  - TD samples

- TD method, compared to Monte Carlo:
  - Also learns directly from episodes of experiences
  - Also is model-free (MDP transitions and rewards unknown)
  - Contrary to Monte Carlo, it learns from *incomplete* episodes, by bootstrapping (guessing the remaining reward instead of fully completing the episode) therefore it can also learn without the final outcome (incomplete sequence) whereas MC can't
  - TD works in continuing (non-terminating) environments whereas MC only works for episodic (terminating) environments
  - TD learns online after every step, MC must wait until the end of the episode when the return is known
  - Intuitive example: let's say you are in a car and almost crash into an other car but in the last second the crash is averted. In MC, you would still arrive at your target and get a positive reward, and you would not be able to learn from this near-death experience. In TD, your estimate of the total reward you were going to get would reflect the near-crash situation, since it would be very negative just before the crash. This would allow you to learn (maybe I should have slowed down a bit earlier) without actually crashing the car
  - Good example of difference in the updates in the "driving home example" (minute 41):
  - Bias / variance trade-off:
    * Return used in MC is $G_t = R_{t+1} + \gamma R_{t+2} + ... + \gamma^{T-1} R_T$ (sample of the actual returns) which is an unbiased estimator of $v_\pi(S_t)$
    * For TD:
      · If we would use the true TD target $R_{t+1} + \gamma v_\pi(S_{t+1})$ it would be an unbiased estimator of $v_\pi(S_t)$ - but we don't have the true value function $v_\pi(S_{t+1})$
      · Instead we use $R_{t+1} + \gamma V(S_{t+1})$ which is a biased estimator of $v_\pi(S_t)$ but with much lower variance - the variance is lower because $V(S_{t+1})$ is just a fixed function, which you evaluate and returns a single number, whereas the sampled returns contain a lot of variation because there are a lot of steps



where the environment can have a lot of random influence (due to actions, transitions, rewards) - "we're only looking at the variance included in the first step (due to the single R_{t+1} term) and not over the whole trajectory"

- MC has high variance, zero bias - directly copied from slide:
  - Good convergence properties
  - (even with function approximation) (Note: in later lectures we'll see that determining the value function is impractical because there are so many states, so instead we'll use function approximator for it)
  - Not very sensitive to initial value
  - Very simple to understand and use
- TD has low variance, some bias - directly copied from slide:
  - Usually more efficient than MC
  - TD(0) converges to $v_\pi(s)$
  - (but not always with function approximation)
  - More sensitive to initial value
- TD exploits Markov property (the current state is all you need) by implicitly building the Markov property, which makes it more efficient in Markov environments
- MC does not exploit Markov property, it ignores it and uses all the sampled returns, which makes it more effective in non-Markov environments (eg. partially observed, messy state signal)

- Usually, in-practice, you a Markov to a certain degree and then you can adjust you algorithm to be more or less Markov (resemble more TD or MC)

7.4 TD lambda

- TD(λ) generalizes the idea of TD(0) to more steps into the future "n-step prediction"
- n-step temporal difference learning:

$$V(S_t) \leftarrow V(S_t) + \alpha \left(G_t^{(n)} - V(S_t) \right) \tag{17}$$

  where $G_t^{(n)} = R_{t+1} + \gamma R_{t+2} + ... + \gamma^{n-1} R_{t+n} + \gamma^n V(S_{t+n})$
- Which n should we pick? slide shows a study on the effect of the choice of n and α (step-size) on the root mean squared error. Clearly shows that the settings affect the performance quite heavily. Not ideal: we want algorithms that are robust to hyper parameter settings. TD(λ) is an algorithm that efficiently considers all n at once
- Intuition for the idea of TD(λ) is that you can average n-step returns over different n, leading to a more robust update step. For instance updating V(S_t) towards $\frac{1}{3}G^{(2)} + \frac{1}{3}G^{(4)} + \frac{1}{3}G^{(6)}$ instead towards a since G
- TD(λ) takes a geometrically weighted average over all n, using weight $(1-\lambda)\lambda^{n-1}$, with last weight λ^{T-t-1} given to the actual final return. The sum of all weights is 1
- With $\lambda = 1$, TD(λ) reduces to MC
- We could use another weighting, but geometric weighting is very efficient since it is memory-less and therefore can be computed very efficiently
- Now there are two flavours of this:
  - **Forward looking**
    - This gives the theory
    - Looks forward into the future to compute G_t^λ and uses that to update the value function for the state we are in now
    - Compute an geometrically weighted estimate of all future returns

$$G_t^\lambda = (1-\lambda) \sum_{n=1}^{\infty} \lambda^{n-1} G_t^{(n)} \tag{18}$$

      and use that in the update function for $V(S_t)$ shown above
    - However, this has the same downside as MC in the sense that you need to fully complete an episode before you can compute G_t^λ

- **Backward looking**
  - This gives the practical algorithm
  - Achieves the same result as forward looking TD(λ) without having to wait until the end of the episode (we want to have online, every step, from incomplete sequences)
  - For every state we start tracking an **eligibility trace**, sort of a decaying memory of the visits to a state, combines recency and frequency

$$E_0(s) = 0$$

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

  - The update term is again, like in the definition of TD(0), defined in terms of the immediate reward and value function of the next state $V(S_{t+1})$

$$\delta_t = R_{t+1} + \gamma V(S_{t+1}) - V(S_t) \tag{19}$$

  - But now, in backward looking TD(λ), we update the value function for a certain state $V(s)$ in proportion to the eligibility trace $E_t(s)$

$$V(s) \leftarrow V(s) + \alpha\delta_t E_t(s) \tag{20}$$

- Theorem: the sum of offline updates is identical for forward-view and backward-view TD(λ)

8 Lecture 5 - Model-free Control

Lecture outline

- Introduction
- On-policy Monte-Carl control
- On-policy Temporal-Difference control
- Off-policy learning

This lecture: using the tools from last lecture (model-free prediction), work towards model-free control. Next lectures: scaling up to larger problems

8.1 Introduction

- Most problems have an MDP underlying, but the MDP is unknown or the MDP is known but it is so complicated / big that it can't be used. Here, model-free control provides a solution by sampling
- On-policy: "learning on the job", learning about policy π from experience sampled from policy π

- Off-policy: "looking over someones shoulder", learning about policy π from experience sampled from policy μ
- Refresher generalised policy iteration - iteratively converge on the true value function by:
  1. Policy evaluation: estimate v_π (eg. Monte-Carlo policy evaluation)
  2. Policy improvement: generate $\pi' \geq \pi$ (eg. by taking a greedy step)

  we'll use this going forward and vary what we "slot in" for 1. and 2.
- The issue we have to solve now is on what basis to do the policy improvement:
  - So far we have been estimating the value function for a given policy, but to go from that to the next best step we need the dynamics of the system: $\pi' = \underset{a \in \mathcal{A}}{\text{argmax}}\ \mathcal{R}_s^a + \mathcal{P}_{ss'}^a V(s')$
  - So instead we will have to use the q-function, which is model-free: $\pi' = \underset{a \in \mathcal{A}}{\text{argmax}}\ Q(s,a)$
- So instead of doing 1. MC policy evaluation on v_π, we will do it on q_π, and then 2. take a greedy step using q_π
- But, if we only take greedy steps, we don't explore and won't evaluate states we haven't seen before, hence the estimated value function and state-value function (q) won't be accurate since you might be stuck in a **local optimum**
- ϵ-greedy exploration: simplest idea (and hard to beat) for ensuring continual exploration
  - With probability $1 - \epsilon$ take the greedy action
  - With probability ϵ choose one of the m action randomly (greedy action is one of the m options and can therefore also be picked randomly)
  - Mathematically:

$$\pi(a|s) = \begin{cases} \epsilon/m + 1 - \epsilon & \text{if } a^* = \underset{a \in \mathcal{A}}{\text{argmax}}\ Q(s,a) \\ \epsilon/m & \text{otherwise} \end{cases} \tag{21}$$

  - Theorem that guarantees that the ϵ-greedy policy yields an improvement at each step over the policy that you had - "guarantees that the ϵ-greedy policy is at least as good as what you started with"
- A policy is **GLIE** (Greedy in the Limit with Infinite Exploration) when it has two characteristics:

  - All state-action combinations are explored infinitely many times (ensures we don't miss anything): $\lim_{k\to\infty} N_k(s,a) = \infty$
  - The policy converges on a greedy policy (ensures that it will resemble the Bellman optimality equation, which contains a max): $\lim_{k\to\infty} \pi_k(a|s) = \mathbf{1}(a = \operatorname{argmax}_{a'\in\mathcal{A}} Q_k(s,a')$

- ϵ-greedy is GLIE when ϵ follows a hyperbolic schedule: $\epsilon_k = \frac{1}{k}$
- GLIE Monte-Carlo control
  - Sample kth episode using π: $S_1, A_1, R_1, ..., S_T \sim \pi$
  - For each state S_t and action A_t in the episode,

$$N(S_t, A_t) \leftarrow N(S_t, A_t) + 1$$

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \frac{1}{N(S_t, A_t)}(G_t - Q(S_t, A_t))$$

  - Improve policy based on new action-value function

$$\epsilon \leftarrow 1/k$$

$$\pi \leftarrow \epsilon - \text{greedy}(Q)$$

  - Guaranteed to converge to the optimal action-value function: $Q(s,a) \rightarrow q_*(s,a)$
  - In practice:
    - No need to store π (it's implicit), instead you just store and evaluate $Q(s,a)$
    - It's most efficient to update $Q(s,a)$ in every episode (not wait until you've collected a batch of episodes
    - Starting values for $Q(s,a)$ don't matter much for this algorithm, since we have the term $\frac{1}{N(S_t,A_t)}$, which evaluates to 1 the first time when $N(S_t, A_t) = 1$ and therefore replaces the initial value
    - If you want to visualize the value function you can calculate the value function from $Q(s,a)$ by summing over the actions you can take from each state

- 38:00