Contents

1 Introduction

Script of the Reinforcment Learning Course by David Silver as uploaded on YouTube from 2015

written by Alex Liebenau

created in November 2022

2 Lecture two: Markov Decision Process (MDP)

2.1 Markov Process

The current state characterises the process -¿ we are told the state –¿ environment is fully observable

Almost all RL problems can be formalised as MDPs
–¿ Optimal control primarily deals with continious MDPs
–¿ Any partially observable problems can be converted into MDPs
–¿ Bandits are MDPs with one state

"The future is independent of the past given the present"

$$\mathbb{P}\left[S_{t+1} \mid S_t\right] \;=\; \mathbb{P}\left[S_{t+1} \mid S_1, \ldots, S_t\right] \tag{1}$$

What happens next only depends on what happend on the state before - you can throw away anything else.

For a Markov state s and successor state s', the *state transition probability* is defined as

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

State transition matrix ρ defines transition probabilites from all states s to all successor states s'.

$$\mathcal{P} \;=\; from \begin{pmatrix} & to & \\ \mathcal{P}_{11} & \cdots & \mathcal{P}_{1n} \\ \vdots & & \\ \mathcal{P}_{n1} & \cdots & \mathcal{P}_{nn} \end{pmatrix} \tag{3}$$

Each row of the matrix sums to 1!
A Markov process is a memoryless random process, i.e. a sequence of random states, S_1, S_2, ... with the Markov propoerty. It is defined as a tuple $\langle \mathcal{S}, \mathcal{P} \rangle$.

- S is a (finite) set of states
- $\mathcal{P}$ is a state transition probability matrix

2.2 Markov Reward process (MRP)

The Markov Reward Process is defined as a tuple $\langle \mathcal{S}, \mathcal{P}, \mathcal{R}, \gamma \rangle$.

- $\mathcal{S}$ is a (finite) set of states
- $\mathcal{P}$ is a state transition probability matrix
- $\mathcal{R}$ is a reward function, $\mathcal{R}_s \;=\; \mathbb{E}[R_{t+1} \mid S_t = s]$

- γ is a discount factor, $\gamma \in [0, 1]$

The *return* $\mathcal{G}_t$ is the total discounted reward from time-step t.

$$\mathcal{G}_t \;=\; R_{t+1} \;+\; \gamma R_{t+2} \;+\; \ldots \;=\; \sum_{k=0}^{\infty} \gamma^k \, R_{t+k+1} \tag{4}$$

- The discount γ is the present value of future rewards - the closer γ is to zero, the less are later rewards accounted (e.g. more 'short-sighted').
- The value of receiving reward $\mathcal{R}$ after $k + 1$ time-steps is $\gamma^k R$.

Why discount?

- Unless you really trust your model and believe that everything turns out as planned, you need to discount in deviations - Uncertainty about the future may not be fully represented
- Mathematically convenient to discount rewards, avoids infinite returns
- If the reward is financial, immediate rewards may earn more interest than delayed rewards
- Animal/human behaviour shows preference for immediate reward
- It is sometimes possible to use *undiscounted* Markov reward process (i.e. $\gamma = 1$), e.g. if all sequences terminate.

The value function

The value function *v(s)* gives the long-term value of state s. It defines the expected return in a MRP starting from state s:

$$v(s) \;=\; \mathbb{E}\left[G_t \mid S_t \;=\; s\right] \tag{5}$$

The value function can be decomposed into two parts:

- immediate reward $R + 1$
- discounted value of successor state $\gamma v(S_{t+1})$

This resolves into the **Bellman equation for MRPs**:

$$v(s) \;=\; \mathbb{E}\left[G_t \mid S_t \;=\; s\right] \;=\; \mathbb{E}\left[R_{t+1} \;+\; \gamma v(S_{t+1}) \mid S_t \;=\; s\right] \tag{6}$$

By averaging all possible outcomes we get

$$v(s) \;=\; \mathcal{R}_s \;+\; \gamma \sum_{s' \in \mathcal{S}} \mathcal{P}_{ss'} \, v(s') \tag{7}$$

The Bellman equation can be concisely using matrices, where v is a column vector with one entry per state

$$v \;=\; \mathcal{R} \;+\; \gamma\,\mathcal{P}\,v \quad \rightarrow \quad \begin{pmatrix} v(1) \\ \vdots \\ v(n) \end{pmatrix} \;=\; \begin{pmatrix} \mathcal{R}_1 \\ \vdots \\ \mathcal{R}_n \end{pmatrix} \;+\; \gamma \begin{pmatrix} \mathcal{P}_{11} & \dots & \mathcal{P}_{1n} \\ \vdots & & \\ \mathcal{P}_{n1} & \dots & \mathcal{P}_{nn} \end{pmatrix} \begin{pmatrix} v(1) \\ \vdots \\ v(n) \end{pmatrix} \tag{8}$$

The Bellman equation is linear. It can be solved directly by $v = (I - \gamma\mathcal{P})^{-1}\mathcal{R}$.

- The Computational complexity is $\mathrm{O}(n^3)$ for n states.
- Direct solution only possible for small MRPs
- Iterative methods for large MRPs, e.g. Dynamic programming, Monte-Carlo evaluation, Temporal-Difference learning

2.3 Markov Decision Process (MDP)

A MDP is a MRP with decisions. It is an *environment* in which all states are Markov.
A MDP is a tuple $\langle \mathcal{S}, \mathcal{A}, \mathcal{P}, \mathcal{R}, \gamma \rangle$.

- $\mathcal{S}$ is a (finite) set of states
- $\mathcal{A}$ is a (finite) set of actions
- $\mathcal{P}$ is a state transition probability matrix
  $\mathcal{P}^a_{ss'} \;=\; \mathbb{P}\left[S_{t+1} \;=\; s' \,|\, S_t \;=\; s,\; A_t \;=\; a\right]$
- $\mathcal{R}$ is a reward function, $\mathcal{R}^a_s \;=\; \mathbb{E}[R_{t+1} \,|\, S_t = s,\; A_t \;=\; a]$
- γ is a discount factor, $\gamma \in [0, 1]$

A policy π is a distribution over actions given states. It fully defines the behaviour of an agent.

$$\pi(a|s) \;=\; \mathbb{P}\left[\; S_t = s,\; A_t \;=\; a \,\right] \tag{9}$$

In an MDP, the policies depend on the current state (not the history). Policies are stationary: $A_t = \pi(\dot{|}S_t), \forall t > 0$

Given an MDP $\mathcal{M} = \langle \mathcal{S}, \mathcal{A}, \mathcal{P}, \mathcal{R}, \gamma \rangle$ and a policy π:

- The state sequence $S_1, S_2, \dots$ is a Markov process $\langle \mathcal{S}, \mathcal{P}^\pi \rangle$
- The state and reward sequence $S_1, , R_2, S_2, \dots$ is a Markov reward process $\langle \mathcal{S},\, \mathcal{P},\, \mathcal{R},\, \gamma \rangle$ where

$$\mathcal{P}^\pi_{s,s'} \;=\; \sum_{a \in \mathcal{A}} \pi(a|s)\, \mathcal{P}^a_{s,s'} \qquad \mathcal{R}^\pi_s \;=\; \sum_{a \in \mathcal{A}} \pi(a|s)\, \mathcal{R}^a_s \tag{10}$$

The *state-value function* $v_\pi(s)$ of an MDP is the expected return starting from state s, and then following policy π:

$$v_\pi(s) \;=\; \mathbb{E}_\pi[G_t \mid S_t \;=\; s] \tag{11}$$

The *action-value function* $q_\pi(s, a)$ of an MDP is the expected return starting from state s, taking action a, and then following policy π:

$$q_\pi(s, a) \;=\; \mathbb{E}_\pi[G_t \mid S_t \;=\; s,\; A_t \;=\; a] \tag{12}$$

The state-value and action-value functions can again be decomposed into Bellman equations consisting of immediate reward plus discounted value of successor state:

$$v_\pi \;=\; \mathbb{E}_\pi \left[R_{t+1} \;+\; \gamma v_\pi(S_{t+1}) \;\mid S_t \;=\; s\right] \tag{13}$$

$$q_\pi \;=\; \mathbb{E}_\pi \left[R_{t+1} \;+\; \gamma v_\pi(S_{t+1},\; A_{t+1}) \;\mid S_t \;=\; s,\; A_t \;=\; a\right] \tag{14}$$

Basically, the state-value averages over the different actions that can be taken:

$$v_\pi(s) \;=\; \sum_{a \in \mathcal{A}} \pi(a|s)\, q_\pi(s,\; a) \tag{15}$$

The other way around, by using the probabilities of the transition dynamics we can average through the values of the successing states we can evaluate a certain action:

$$q_\pi(s,\; a) \;=\; \mathcal{R}_s^a \;+\; \gamma \sum_{s' \in \mathcal{S}} \mathcal{P}_{ss'}^a\, v_\pi(s') \tag{16}$$

- v is is telling how good it is to be in a particular state
- q is telling how good it is to take a particular action

Sticking these together in order to solve MDPs end up in a two-step lookahead:

- Consider all action we might take next (v)
- Consider all the things the environment might do to us (q)
- Evaluate the successor state after what the environment did after that point

By double-averaging over the policy as well as the transistion probability we get the answer to how good it is to be in a particular state (as the Bellman Equation):

$$v_\pi(s) \;=\; \sum_{a \in \mathcal{A}} \pi(a|s) \left(\mathcal{R}_s^a \;+\; \gamma \sum_{s' \in \mathcal{S}} \mathcal{P}_{ss'}^a\, v_\pi(s') \right) \tag{17}$$

Starting from a particular action, we can also do the same two-step lookahead and see where the wind blows us, and from there consider which action might be taken next. By averaging the same way above we get same recursive relationship:

$$q_\pi(s,\, a) \;=\; \mathcal{R}_s^a \;+\; \gamma \sum_{s' \in \mathcal{S}} \mathcal{P}_{ss'}^a \sum_{a' \in \mathcal{A}} \pi(a'|s')\, q_\pi(s',\, a') \tag{18}$$

This shows how the value- and action function of the next step relates to itself, we therefore get a recursive relationship. **After all, the value function of the current time step is equal to the immediate reward plus the value function of where you end up.**

To flatten the MDP into a MRP one could average out the matrices as in (7) following policy π and then solve accordingly as in (8). Thereby we can determine the value function.

.

_____ **For more info on undiscounted MDPs:**

Neurodynamic Programming / Dynamic Programming; Dimitri Bertsekas Dynamic Programming and Optimal Control (2 Vol Set) _____

2.4 The Optimal Value Function

The optimal *state-value* and *action-value* functions are the maximum functions over all policies

$$v_*(s) \;=\; \max_\pi \; v_\pi(s) \qquad q_*(s,a) \;=\; \max_\pi \; q_\pi(s,a) \tag{19}$$

- The optimal value function specifies the best performance in an MDP
- An MDP is 'solved' when we know the optimal value function

If you know q_*, you're basically done. It tells you under all different ways to behave which one gives the most reward.

There is always one optimal policy. This optimal policy will then achieve the optimal *state-value* and *action-value* function v_{π_*} and q_{π_*}. So we can define a partial ordering over policies:

$$\pi \;\geq\; \pi' \quad if \quad v_\pi(s) \geq v_{\pi'}(s), \forall s \tag{20}$$

An optimal policy can be found by maximising over $q_{\pi_*}(s,a)$:

$$\pi_*(a|s) \;=\; \begin{cases} 1 & \text{if } a \;=\; argmax_{a \in \mathcal{A}} \; q_{\pi_*}(s,a) \\ 0 & \text{otherweise} \end{cases} \tag{21}$$

The Bellman Optimality Equation

The optimal value functions are recursively related by the Bellman optimality equations:

Instead of taking the average of all actions, you pick the maximum. That determines the optimal value function of a state:

$$v_* = \max_a \; q_*(s, a) \tag{22}$$

Now looking at the action-value function, we can inductively assume that each of the states we might end up in has a optimal state-value v_*. Therefore, similar as for (16), to determine the action-value we add the immediate reward to the average of the optimal state-values where we could end up in:

$$q_*(s,\, a) \;=\; \mathcal{R}_s^a \;+\; \gamma \sum_{s' \in \mathcal{S}} \mathcal{P}_{ss'}^a \; v_*(s') \tag{23}$$

As we did before in (17), we can put these two together to create the recursive relationship with the two-step lookahead. For the optimal solution, we pick the maximum instead of the average. This is the Bellman optimality equation for v_*:

$$v_*(s) \;=\; \max_{a \in \mathcal{A}} \left(\mathcal{R}_s^a \;+\; \gamma \sum_{s' \in \mathcal{S}} \mathcal{P}_{ss'}^a \; v_*(s') \right) \tag{24}$$

By reordering the same idea, we can again define the Bellman optimality equation for q_*.

$$q_*(s,\, a) \;=\; \mathcal{R}_s^a \;+\; \gamma \sum_{s' \in \mathcal{S}} \mathcal{P}_{ss'}^a \; \sum_{a' \in \mathcal{A}} \max_{a' \in \mathcal{A}} \; q_*(s', a') \tag{25}$$

Solving the Bellman Optimality Equation

- The Bellman Optimality Equation is non-linear
  → Solving by inverting matrices like in (7) will not work
- No closed form solution (in general)
- Many iterative solution methods
  → Value iteration, policy interation, **Q-Learning**, Sarsa

3 Lection Three: Planning by Dynamic Programming

3.1 Introduction

Dynamic: sequential or temporal component to problem
Programming: optimising a 'problem', e.g. a policy

→ Solving complex problems by braking them down into subproblems, then combine their solutions

Properties of problems for dynamic programming:

- Optimal substructure:
  → Principle of optimality applies
  → Optimal solution can be decomposed into subproblems
- Overlapping substructure
  Subproblems reoccur many times → Solutions can be cached and reused

MDPs statisfy both properties. The Bellman Equation gives the recursive decompostion. The value function stores and reuses the solutions. Dynamic Programming thereby assumes full knowledge of the MDP, it is used for *planning* in an MDP.

For prediction

- **Input:** An MDP $\langle \mathcal{S}, \mathcal{A}, \mathcal{P}, \mathcal{R}, \gamma \rangle$ and policy π or an MRP $\langle \mathcal{S}, \mathcal{P}^\pi, \mathcal{R}^\pi, \gamma \rangle$
- **Output:** value function v_π

or

For Optimal Control

- **Input:** An MDP $\langle \mathcal{S}, \mathcal{A}, \mathcal{P}, \mathcal{R}, \gamma \rangle$
- **Output:** Optimal value function v_* and optimal policy π_*

3.2 Synchronous Dynamic programming algorithms

Policy Evaluation and Policy Iteration

Solution: iterative application of Bellman Equation $v_1 \to v_2 \to v_3 \to \dots \; v_n$
Using *synchronous* backups:

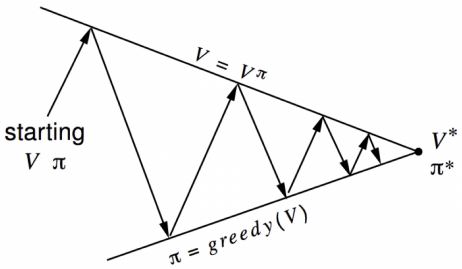
- At each iteration k + 1
- For all states $s \in \mathcal{S}$

- Update $v_{k+1}(s)$ from $v_k(s')$ where s' is a successor state of s

Given a policy π, how do zou find out if it's optimal?

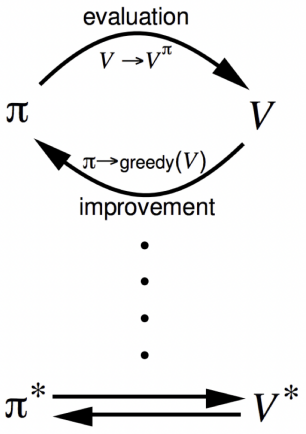
- *Evaluate* the policy $\pi \rightarrow$ Compute the value function v_π
- *Improve* the policy by acting greedily with respect to v_π

Acting greedily means picking the action that returns the most reward. This will in every case improve the value function. In general, it always needs a lot of improvement and evaluation. This process will *always* converge to π_*, no matter in which state you begin.



Policy evaluation Estimate v_π
Iterative policy evaluation

Policy improvement Generate $\pi' \geq \pi$
Greedy policy improvement



What does policy improvement mean?

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

- We can *improve* the policy by acting greedily: $\pi'(s) = argmax_{a \in \mathcal{A}} \; q_{\pi_*}(s, a)$
- This will improve the value from any state s over one step:
  $q_\pi(s, \pi'(s)) \; = \; max_{a \in \mathcal{A}} \; q_{\pi_*}(s, a) \; \geq \; q_{\pi_*}(s, a) \; + \; v_\pi$
- Keep iterating over the coming time-steps
- If improvement stops:
  $q_\pi(s, \pi'(s)) \; = \; max_{a \in \mathcal{A}} \; q_\pi(s, a) \; = \; q_\pi(s, a) \; = \; v_\pi$
  $\rightarrow$ Bellman optimality equation is satisfied: $v_\pi(s) \; = \; max_{a \in \mathcal{A}} \; q_\pi(s, a)$
- Therefore: $v_\pi \; = \; v_*$ for all $s \in \mathcal{S}$ $\rightarrow$ π is an optimal policy

Value iteration

Any optimal policy can be subdivided into two components:

- An optimal first action A_*

- Followed by an optimal policy from successor state S'

The policy is optimal, if from each state we might end up the policy is optimal from that state onwards. If we know the solution to subproblems $v'_*(s)$, the solution $v_*(s)$ can be found iteratively by one-step lookahead:

$$v_*(s) \leftarrow \max_{a \in \mathcal{A}} \left(\mathcal{R}_s^a + \gamma \sum_{s' \in \mathcal{S}} \mathcal{P}_{ss'}^a \, v_*(s') \right) \tag{26}$$

The idea of value iteration is to apply these updates iteratively, thereby moving through the whole state-space. The intuition is that you begin with your final reward and work your way backwards.

Unlike policy iteration, there is no explicit policy. Intermediate value functions may not correspond to any policy.

Problem	Bellman Equation	Algorithm
Prediction	Bellman Expectation Equation	Iterative Policy Evaluation
Control	Bellman Expectation Equation + Greedy Policy Improvement	Policy Iteration
Control	Bellman Optimality Equation	Value Iteration

These algorithms are based on the state-value function $v_\pi(s)$ or $v_*(s)$. They have a complexity of $\mathcal{O}(mn^2)$ per iteration, for m actions and n states. The algorithms could also apply to $q_\pi(s)$ or $q_*(s)$, where the complexity would be $\mathcal{O}(m^2n^2)$.

3.3 Asynchronous dynamic programming

It is not necessary to update every state at each sweep of the algorithm. In *asynchronous* backups, the relationship of updating every state in each iteration is broken. This saves a lot of unnecessary work and as long as all states keep being selected at some time, the algorithm will still converge to the optimal solution.

In-place dynamic programming

Synchronous value iteration stores two copies of the value function:

$$v_{new}(s) \leftarrow \max_{a \in \mathcal{A}} \left(\mathcal{R}_s^a + \gamma \sum_{s' \in \mathcal{S}} \mathcal{P}_{ss'}^a \, v_{old}(s') \right) \tag{27}$$

$$v_{old} \leftarrow v_{new} \tag{28}$$

In-place value iteration only stores one copy of value function by using the latest version of the value function available. That means if a state has been visited

before, the algorithm uses the already calculated value instead of the old one. This tends to be more efficient, but the order of the states being swept is not relevant for the efficiency of the algorithm.

Prioritized sweeping

By using the magnitude of the Bellman error to guide state selection, the states with the largest difference after an update have a higher priority. These will have the most effect in downstream computing. It is important though to update the Bellman error after each backup, which required knowledge of reverse-dynamics of a system for the predecessor states. The Bellman error is described as following:

$$\left| \max_{a \in \mathcal{A}} \left(\mathcal{R}_s^a \; + \; \gamma \sum_{s' \in \mathcal{S}} \mathcal{P}_{ss'}^a \, v_{old}(s') \right) \; - \; v(s) \right| \tag{29}$$

Real-time Dynamic Programming

The idea of *Real-time programming* is to only update the states which are relevant to the agent. The experience of the agent is used to guide the the selection of states. After each time-step S_t, A_t, R_{t+1}, a backup of state S_t is done.

$$v(S_t) \; \leftarrow \; \max_{a \in \mathcal{A}} \left(\mathcal{R}_{S_t}^a \; + \; \gamma \sum_{s' \in \mathcal{S}} \mathcal{P}_{S_t s'}^a \, v(s') \right) \tag{30}$$

Full-Width and Sample backups

Dynamic programming uses *full-width backups*, meaning that every state is backed up. For medium-sized problems consisting of a few millions of states, this is fine. But for large problems, dynamic programming suffer from the *curse of dimensionality*, following from the exponential growth of the number of states with the number of state variables.

This is where sample backups come into play. Instead of backing up the whole state-space (which can be already too expensive for one iteration), only certain samples are backed up. The advantages are:

- Model-free: No advance knowledge of MDP required
- Breaks the curse of dimensionality through sampling
- Cost of backup is constatnt, independant of $|n| = \mathcal{S}$

4 Lecture Four: Model-Free Prediction

4.1 Monte-Carlo Learning

Monte-Carlo Learning methods learn directly from episodes of experience. It is therefore important that the MDP is applied to *episodic* MDP, where the (complete) episodes do terminate. The value function is assumed to be the simplest possible, the mean return of the samples.

The goal is to learn the value function v_π of a random policy π by looking at some streams $S_1, A_1, R_2, \ldots, S_k \sim \pi$ of episode of experience and then evaluate the *return* as the total discounted reward $\mathcal{G}_t$ form (4). Instead of using the *expected* return of the value function, Monte-Carlo Learning uses the *empirical mean* from the sampled episodes.

First-Visit Monte-Carlo policy evaluation

- To evaluate the state s, the first time-step it is visited in an episode, an incremental counter is initialized: $N(s) \leftarrow N(s) + 1$
- The total return is added up: $S(s) \leftarrow S(s) + G_t$
- The value of the state is estimated by mean return: $V(s) = S(s)/N(s)$

By the law of large numbers, the value of the state $V(s)$ converges to the actual value v_π of the state is visited many times so that the estimated mean consists of a large number of visits: $V(s) \to v_\pi$ as $N(s) \to \infty$

The **Every-Visit Monte-Carlo policy evaluation** works in a very similar fashion, just that the counter and return is incremented - as the name suggests - every time the episode visits a state.

Incremental Monte-Carlo

The mean $\mu_1, \mu_2, \ldots$ of a sequence $x_1, x_2, \ldots$ can be computed incrementally:

$$\begin{aligned} \mu_k &= \frac{1}{k} \sum_{j=1}^{k} x_j \\ &= \frac{1}{k} \left(x_k + \sum_{j=1}^{k-1} x_j \right) \\ &= \frac{1}{k} \left(x_k + (k-1)\, \mu_{k-1} \right) \\ &= \mu_{k-1} + \frac{1}{k} \left(x_k - \mu_{k-1} \right) \end{aligned} \tag{31}$$

The difference $x_k - \mu_{k-1}$ basically defines the error term, where μ_{k-1} represents the expected value and x_k the actual value. In the process of (31), the value of the mean will then get an update slightly into the direction of the error term.

This can be applied to the Monte-Carlo Learning. The value $V(s)$ of a state s will be updated incrementally after an episode $S_1, A_1, R_2, \ldots, S_t$:

- For each state S_t with return G_t:
  $N(s) \leftarrow N(s) + 1$
  $V(S_t) \leftarrow V(S_t) + \frac{1}{N(s)}(G_t - V(S_t))$
- In non-stationary problems, it can be useful to track a running mean, e.g. forget old episodes
  $V(S_t) \leftarrow V(S_t) + \alpha(G_t - V(S_t))$

4.2 Temporal-Difference Learning

Like with Monte-Carlo Learning, Temporal-Difference Learning learns from episodes of experience and needs no knowledge of the MDP transitions or rewards, they are both *model-free*. In contrast to Monte Carlo Learning however, Temporal-Difference Learning also learns from incomplete episodes, they don't have to terminate. By taking partial episodes and estimating the remaining return, it can learn from incomplete episodes. It iteratively updates the guesses of the return while going thorugh an episode. This is called *bootstrapping*.

The simplest Temporal-Difference Learning algorithm: TD(0)

- The value of V_s is updated towards the *estimated* return: $R_{t+1} + \gamma V(S_{t+1})$

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

- $R_{t+1} + \gamma V(S_{t+1})$ is called the TD target (the estimated return)
- $\delta_t = (R_{t+1} + \gamma V(S_{t+1}) - V(S_t))$ is called the TD error

As seen in caption 1, the predicted total travel time gets updated after every state in TD Learning, while in MC learning all the values of the states will be updated after the episode.

Advantages and Disadvantages of Temporal-Difference vs. Monte-Carlo Learning

- TD can learn before knowing the final outcome
  - TD can learn online after every step
  - MC must wait until the end of an episode before return is known
- TD can learn without the final outcome
  - TD can learn from incomplete sequences
  - MC can only learn from complete sequences
  - TD works in continuing environments

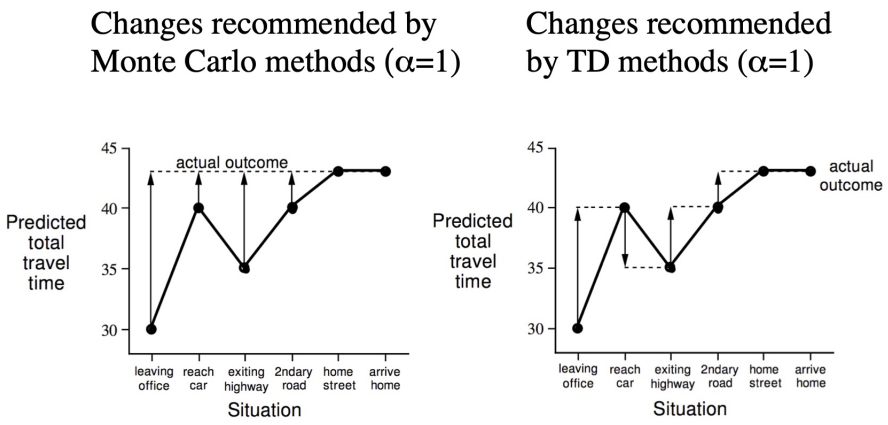


Figure 1: The difference between Monte-Carlo and Temporal-Difference Learning.

  - MC only works for episodic (terminating) environments

- Certainty equivalence:
  - MC converges to solution with minimal mean-squared error
    → Best fit to the observed returns
  - TD converges to solution of max likelihood Markov model
    → Solution to the MDP that best fits the data

- TD exploits Markov property by using it, therefore is more efficient in a Markov environment. MC ignores the Markov property, therefore does not exploit it.

Bias / Variance Trade-Off

- The return G_t and the true TD target $R_{t+1}+\gamma v_\pi(S_{t+1})$ are both *unbiased* estimates of $v_\pi(S_t)$.

- The actual TD target $R_{t+1}+\gamma V(S_{t+1})$ is a *biased* estimate of $v_\pi(S_t)$. It is much lower variance than the return:
  - Return depends on *many* random actions, transitions and rewards.
  - TD target depends on *one* random action, transition and reward.

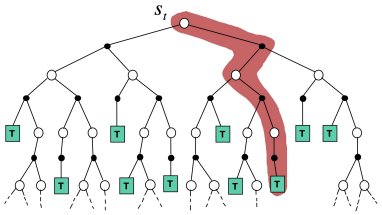
- MC has high variance, zero bias
  - Good convergence properties, even with function approximation
  - Not very sensitive to initial value

  - Very simple to understand and use.

- TD has low variance, some bias
  - Usually more efficient than MC
  - TD(0) converges to $v_\pi(s)$ - but not always with function approximation
  - More sensitive to initial value

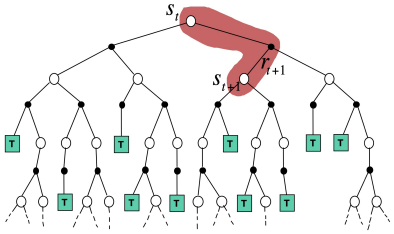
Monte-Carlo Backup:

$$V(S_t) \leftarrow V(S_t) + \alpha\,(G_t - V(S_t))$$



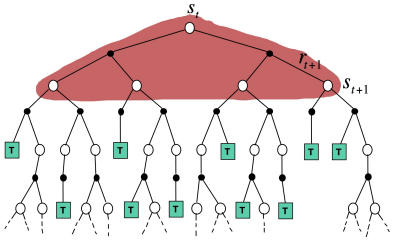
Temporal-Difference Backup:

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



Dynamic Programming Backup:

$$V(S_t) \leftarrow \mathbb{E}_\pi\,[R_{t+1} + \gamma V(S_{t+1})]$$



Note: The *'Exhaustive search'* would consider every branch as in Dynamic Programming, but then follow its leafs until the end like Monte-Carlo instead of taking an estimate.

- **Bootstrapping:** update involves an estimate
  - MC does not bootstrap
  - TD and DP do
- **Samling:** update samples an expectation
  - MC and TD sample
  - DP does not sample

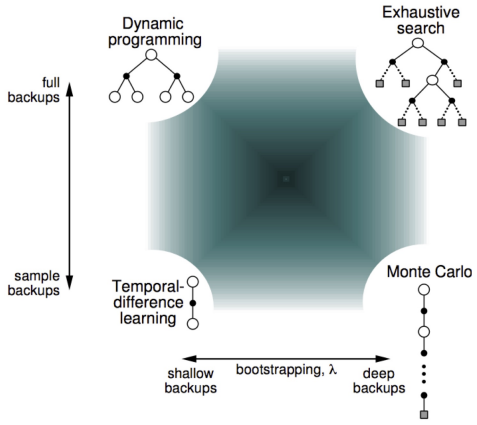
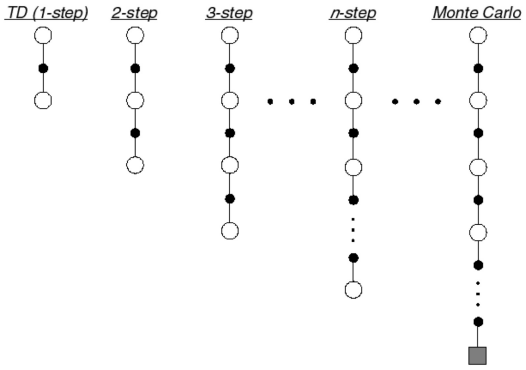


Figure 2: Unified view of Reinforcement Learning

4.3 TD(λ)

n-step Prediction

- Let TD target look *n* steps into the future



n-step Return

$$
\begin{aligned}
n = 1 \quad (TD) \qquad & G_t^{(1)} &&= R_{t+1} \, \gamma V(S_{t+1}) \\
n = 2 \qquad & G_t^{(2)} &&= R_{t+1} \, \gamma R_{t+2} \, + \, \gamma^2 V(S_{t+2}) \\
\vdots \qquad & \vdots \qquad && \\
n = \infty \quad (MC) \qquad & G_t^{(\infty)} &&= R_{t+1} \, + \, \gamma R_{t+2} \, + \, \gamma^{T-1} R_T
\end{aligned}
$$

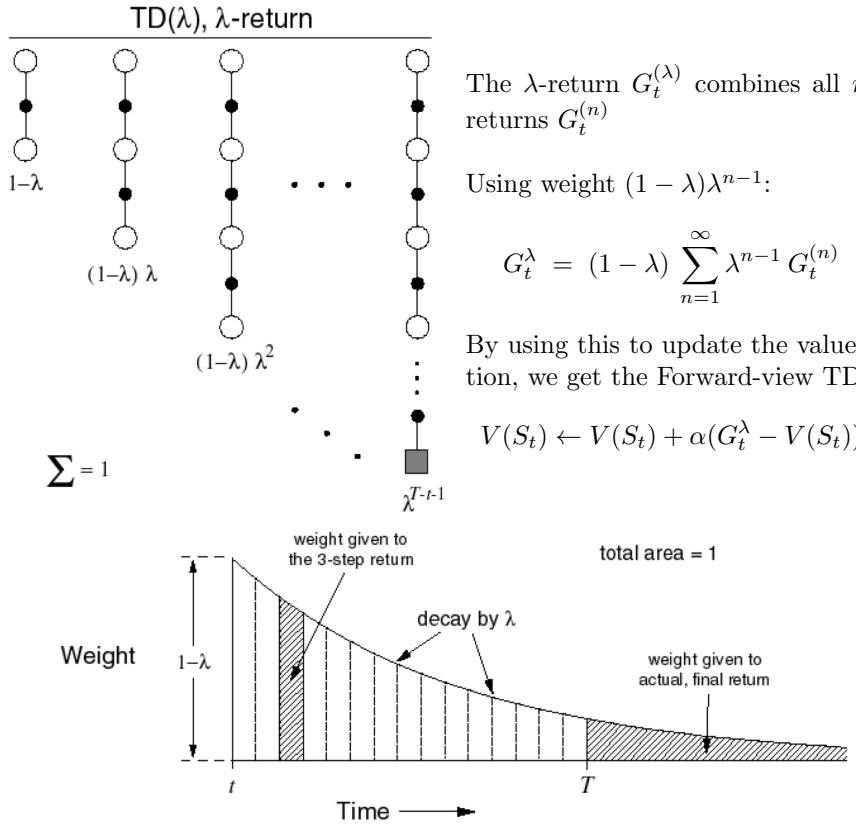
Define the n-step return for the estimator:

$$G_t^{(n)} \;=\; R_{t+1} \;+\; \gamma R_{t+2} \;+\ldots+\; \gamma^{n-1} R_{t+n} \;+\; \gamma^n V(S_{t+n}) \tag{32}$$

n-step Temporal-Difference Learning:

$$V(S_t) \leftarrow V(S_t) + \alpha(G_t^{(n)} - V(S_t) \tag{33}$$

Forward-view of TD(λ)



The λ-return $G_t^{(\lambda)}$ combines all n-step returns $G_t^{(n)}$

Using weight $(1-\lambda)\lambda^{n-1}$:

$$G_t^{\lambda} \;=\; (1-\lambda) \sum_{n=1}^{\infty} \lambda^{n-1}\, G_t^{(n)} \tag{34}$$

By using this to update the value function, we get the Forward-view TD(λ)

$$V(S_t) \leftarrow V(S_t) + \alpha(G_t^{\lambda} - V(S_t)) \tag{35}$$

The Forward-view looks into the future to compute $G_t^{(\lambda)}$. Like Monte-Carlo Learning, it can only be computed from complete episodes.

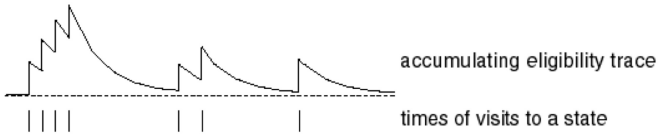
Backward-view of TD(λ)

The Forward-view provides theory, the Backward-view provides mechanism

It updates online, every step, from incomplete sequences.

- **Frequency heuristic:** Assign credit to most frequent states
- **Recency heuristic:** Assign credit to most recent states

→ *Eligibility traces* combine both heuristics!



$$E_0(s) = 0$$
$$E_t(s) = \gamma\lambda E_{t-1}(s) + \mathbf{1}(S_t = s)$$

Backward-view TD(λ)-Algorithm:

- Keep an eligibility trace for every state s
- Update value $V(s)$ for every state s
- In proportion to TD-error δ_t and eligibility trace $E_t(s)$

$$\delta_t = R_{t+1} + \gamma V(S_{t+1}) - V(S_t) \tag{36}$$
$$V(s) \leftarrow V(s) + \alpha\delta_t E_t(s) \tag{37}$$

Relationship between Forward and Backward TD(λ)

The sum of the offline[1] updates is the same identical for forward-view and backward-view TD(λ)

$$\sum_{t=1}^{T} \alpha\delta_t E_t(s) = \sum_{t=1}^{T} \alpha(G_t^\lambda - V(S_t))\,\mathbf{1}(S_t = s) \tag{38}$$

Offline updates	$\lambda = 0$	$\lambda \in (0,1)$	$\lambda = 1$
Backward view	TD(0)	TD(λ)	TD(1)
	‖	‖	‖
Forward view	TD(0)	Forward TD(λ)	MC
Online updates	$\lambda = 0$	$\lambda \in (0,1)$	$\lambda = 1$
Backward view	TD(0)	TD(λ)	TD(1)
	‖	≠	≠
Forward view	TD(0)	Forward TD(λ)	MC
	‖	‖	‖
Exact Online	TD(0)	Exact Online TD(λ)	Exact Online TD(1)

[1] Even for online updating it is now possible to achieve equivalence for forward and backward view TD(λ) - see