Total

Markov Definition

Definition (Bayes Theorem)

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

Markov Property

Definition

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

The future is independent of the past given the present.

- The present state s_t captures all information in the history of the agent's events.
- Once the state is known, then any data of the history is no longer needed.

Definition (Stationarity)

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

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

Definition (Markov Reward Process)

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

Why Discounting is a good idea

Why discounting is a good idea?

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

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

Definition (State value function)

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

$$v(s) = \mathbb{E}[R_t|S_t = s] \qquad (7)$$

Bellman Equation

Forms of the Bellman Equation for MRPs

- Expectation notation:

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

- Sum notation (expectation written out):

$$v(s) = \mathcal{R}_s + \gamma \sum_{s' \in S} \mathcal{P}_{ss'} v(s') \qquad (13)$$

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

- Vector notation:

$$\mathbf{v} = \mathcal{R} + \gamma \mathcal{P}\mathbf{v} \qquad (14)$$

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

Direct solution of Bellman Equation

Direct solution

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

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

we can solve for it directly:

$$
\begin{aligned}
\mathbf{v} &= \mathcal{R} + \gamma \mathcal{P}\mathbf{v} && (16)\\
\mathbf{v} - \gamma \mathcal{P}\mathbf{v} &= \mathcal{R} && (17)\\
(1 - \gamma \mathcal{P})\mathbf{v} &= \mathcal{R} && (18)\\
\mathbf{v} &= (1 - \gamma \mathcal{P})^{-1}\mathcal{R} && (19)
\end{aligned}
$$

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

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

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

Definition (Policy)

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

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

Solve bellman equation directly

Value function self-consistency

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

$$V^\pi(s) = \sum_{a \in \mathcal{A}} \pi(s,a) \sum_{s' \in \mathcal{S}} \mathcal{P}^a_{ss'}\left(\mathcal{R}^a_{ss'} + \gamma V^\pi(s')\right)$$

This is a version of Bellmann's equation. A fundamental property of value functions is that they satisfy a set of recursive consistency equations.
Crucially V^π has unique solution.

Iterative policy evaluation

Reinforcement Learning
└Markov Decision Process —Policy Evaluation

Iterative Policy Evaluation Algorithm

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

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

State-Action Value function as Cost-To-Go

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

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

$$Q^\pi(s,a) = \mathbb{E}[R_t|S_t = s, A_t = a] = \mathbb{E}\left[\sum_{k=0}^{\infty} \gamma^k r_{t+k+1}|S_t = s, A_t = a\right] \qquad (20)$$

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

$$V^\pi(s) = \sum_{a \in \mathcal{A}} \pi(s,a) Q^\pi(s,a) \qquad (21)$$

Optimal Policy and optimal value and Q function

Reinforcement Learning
└Markov Decision Process —Optimal value function

Optimal Value and Cost-to-Go function for MDPs

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

Definition (Optimal State Value function)

$$V^*(s) = \max_\pi V^\pi(s), \forall s \in \mathcal{S} \qquad (22)$$

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

Definition (Optimal State-Action Value function)

$$Q^*(s,a) = \max_\pi Q^\pi(s,a), \forall s \in \mathcal{S}, a \in \mathcal{A} \qquad (23)$$

In analogy to before we also have

$$Q^*(s,a) = \mathbb{E}[r_{t+1} + \gamma V^*(s_{t+1})|S_t = s, A_t = a] \qquad (24)$$

Bellman optimality equation for V^*

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

Definition (Bellmann Optimality Equation for V)

$$
\begin{aligned}
V^*(s) &= \max_a \sum_{s'} P\left[s'|s,a\right]\left(r(s,a,s') + \gamma V^*(s')\right) && (25)\\
&= \max_a \sum_{s'} \mathcal{P}^a_{ss'}\left(\mathcal{R}^a_{ss'} + \gamma V^*(s')\right) && (26)
\end{aligned}
$$

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

Three Assumption for BOE

On solving the Bellman Optimality Equations

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

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

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

Convergence Rule

Theorem (Bellman Optimality Equation convergence theorem)

For an MDP with a finite state and action space

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

Dynamic Programming

Two assumption of DP

1. MDP to be finite
2. A perfect model for environment, means we know the trasition and reward function

Reinforcement Learning 8/65
└Dynamic Programming

Dynamic Programming – Origins 2

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

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

Example

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

For every states ⇒

Policy Improvement Theorem

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

In short, update value function using policy untill value function converge, then be greedy to update the policy, then go back to step 2 untill the policy converge.

└Dynamic Programming —Policy Iteration

Policy Iteration Algorithm

1. Initialization
   $V(s) \in \Re$ and $\pi(s) \in \mathcal{A}(s)$ arbitrarily for all $s \in \mathcal{S}$
2. Policy Evaluation
   Repeat
   $\Delta \leftarrow 0$
   For each $s \in \mathcal{S}$:
   $v \leftarrow V(s)$
   $V(s) \leftarrow \sum_{s'} \mathcal{P}^{\pi(s)}_{ss'}\left[\mathcal{R}^{\pi(s)}_{ss'} + \gamma V(s')\right]$
   $\Delta \leftarrow \max(\Delta, |v - V(s)|)$
   until $\Delta < \theta$ (a small positive number)
3. Policy Improvement
   policy-stable ← *true*
   For each $s \in \mathcal{S}$:
   $b \leftarrow \pi(s)$
   $\pi(s) \leftarrow \arg\max_a \sum_{s'} \mathcal{P}^a_{ss'}\left[\mathcal{R}^a_{ss'} + \gamma V(s')\right]$
   If $b \neq \pi(s)$, then *policy-stable* ← *false*
   If *policy-stable*, then stop; else go to 2

Reinforcement Learning
└Dynamic Programming —Value Iteration

Value Iteration Algorithm

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

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

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

Unlike policy iteration, there is no explicit policy.

If value of all states are updated same time or individually

Reinforcement Learning 27/65
└Dynamic Programming —Backup strategies

Synchronous vs Asynchronous backups

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

Reinforcement Learning 28/65
└Dynamic Programming —Backup strategies

Let us take a step back from DP

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

Why value iteration is guranteed to converge

2. Contraction Mapping Property:

- Value iteration repeatedly applies the Bellman update:
  $$V_{k+1}(s) = \max_a \sum_{s'} P(s'|s,a)\left[R(s,a,s') + \gamma V_k(s')\right].$$
- This update is a **contraction mapping** with respect to the max-norm $\|\cdot\|_\infty$:
  $$\|V_{k+1} - V^*\|_\infty \leq \gamma \|V_k - V^*\|_\infty.$$
- The contraction mapping property arises because $\gamma < 1$, which ensures that each iteration brings V_k closer to V^*.

Summary:

Value iteration is guaranteed to converge because:

1. The Bellman update is a contraction mapping.
2. The Banach fixed-point theorem ensures convergence to the unique optimal value function V^*.
3. The discount factor $\gamma < 1$ ensures bounded rewards and stability.
4. A finite state and action space limits the computational complexity, ensuring a solution in finite time.

Monte Carlo

Model-Free Reinforcement Learning: Monte Carlo (MC)

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

Monte-Carlo Policy Evaluation

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

Input: a policy π to be evaluated
Initialize:
$V(s) \in \mathbb{R}$, arbitrarily, for all $s \in \mathcal{S}$
$Returns(s) \leftarrow$ an empty list, for all $s \in \mathcal{S}$

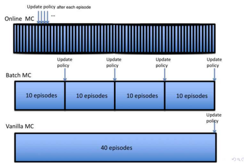
Loop forever (for each episode):
Generate an episode following π: $S_0, A_0, R_1, S_1, A_1, R_2, \ldots, S_{T-1}, A_{T-1}, R_T$
$G \leftarrow 0$
Loop for each step of episode, $t = T-1, T-2, \ldots, 0$:
$G \leftarrow \gamma G + R_{t+1}$
Unless S_t appears in $S_0, S_1, \ldots, S_{t-1}$:
Append G to $Returns(S_t)$
$V(S_t) \leftarrow \text{average}(Returns(S_t))$

First Visit MC vs Evert Visit MC

Batch VS Online Monte-Carlo

Reinforcement Learning 43/65
└Model-Free Learning —Monte Carlo Learning

Batch & Online Monte-Carlo



No Need to store samples traces

Reinforcement Learning 46/65
└Model-Free Learning —Monte Carlo Learning

Incremental Monte-Carlo Updates

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

1. Update $V(s)$ incrementally after step $s_t, a_t, r_{t+1}, s_{t+1}$
2. For each state s_t with return R_t (up to this point) and $N(s)$ the visit counter to this state:

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

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

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

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

Temporal Difference Learning

Reinforcement Learning 53/65
└TD Learning

Temporal-Difference (TD) Learning

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

MC update the value using the actual return R, where dp use the estimated return to update the value

Reinforcement Learning 54/65
└TD Learning

Temporal Difference Learning update rule

Recall that we use the following Monte-Carlo update

$$V(s_t) \leftarrow V(s_t) + \alpha\left(R_t - V(s_t)\right) \qquad (8)$$

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

$$V(s_t) \leftarrow V(s_t) + \alpha\left(r_{t+1} + \gamma V(s_{t+1}) - V(s_t)\right) \qquad (9)$$

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

TDerminology

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

TD value function estimation Algorithm

```
procedure TD-ESTIMATION(π)
    Init
        V̂(s) ← arbitrary value, for all s ∈ S.
    EndInit
    repeat(For each episode)
        Initialise s
        repeat(For each step of episode)
            a action chosen from π at s
            Take action a; observe r, and next state, s'
            δ ← r + γV̂(s') − V̂(s)
            V̂(s) ← V̂(s) + αδ
            s ← s'
        until s is absorbing state
    until Done
end procedure
```

Advantages & Disadvantages of MC & TD

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

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

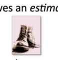
What is Markov Property

Key Characteristics of the Markov Property:

1. **Dependence on Current State**:
   - Only the current state s_t and action a_t influence the future, regardless of how the system arrived at s_t.
2. **Simplifies Modeling**:
   - By assuming the Markov property, the transition dynamics $P(s_{t+1}|s_t, a_t)$ can be described without considering the full history of the process.

Summary DP vs TD vs MC

Bootstrapping: update involves an *estimate*
- MC does not bootstrap
- DP bootstraps
- TD bootstraps



Sampling: update does not involve an *expected value*
- MC samples
- DP does not sample
- TD samples

MC Control

Why use model free control

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

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

On-policy vs off-policy

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

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

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

Means we have a policy that gives a non-zero probablity to all possible actions

Definition

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

Epsilon Greedy

ϵ-greedy policies

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

Definition

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

$$\pi(s,a) = \begin{cases} 1 - \epsilon + \frac{\epsilon}{|\mathcal{A}(s)|}, & \text{if } a^* = \arg\max_a Q(s,a)\\ \frac{\epsilon}{|\mathcal{A}(s)|}, & \text{if } a \neq a^* \end{cases} \qquad (5)$$

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

On-policy first-visit MC control (for ε-soft policies), estimates $\pi \approx \pi_*$

Algorithm parameter: small $\varepsilon > 0$
Initialize:
$\pi \leftarrow$ an arbitrary ε-soft policy
$Q(s,a) \in \mathbb{R}$ (arbitrarily), for all $s \in \mathcal{S}$, $a \in \mathcal{A}(s)$
$Returns(s,a) \leftarrow$ empty list, for all $s \in \mathcal{S}$, $a \in \mathcal{A}(s)$
Repeat forever (for each episode):
Generate an episode following π: $S_0, A_0, R_1, \ldots, S_{T-1}, A_{T-1}, R_T$
$G \leftarrow 0$
Loop for each step of episode, $t = T-1, T-2, \ldots, 0$:
$G \leftarrow \gamma G + R_{t+1}$
Unless the pair S_t, A_t appears in $S_0, A_0, S_1, A_1, \ldots, S_{t-1}, A_{t-1}$:
Append G to $Returns(S_t, A_t)$
$Q(S_t, A_t) \leftarrow \text{average}(Returns(S_t, A_t))$
$A^* \leftarrow \arg\max_a Q(S_t, a)$ (with ties broken arbitrarily)
For all $a \in \mathcal{A}(S_t)$:
$\pi(a|S_t) \leftarrow \begin{cases} 1 - \varepsilon + \varepsilon/|\mathcal{A}(S_t)| & \text{if } a = A^*\\ \varepsilon/|\mathcal{A}(S_t)| & \text{if } a \neq A^* \end{cases}$

Convergence of exploratory MC algorithms

Definition

Greedy in the Limit with Infinite Exploration (GLIE)

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

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

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

Example

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

Summary MC control I

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

1. One approach is to ignore this problem by assuming that episodes begin with state-action pairs randomly selected to cover all possibilities. Such exploring starts can sometimes be arranged in applications with simulated episodes, but are unlikely in learning from real experience.
2. In on-policy methods, the agent commits to always exploring and tries to find the best policy that still explores. In off-policy methods, the agent also explores, but learns a deterministic optimal policy that may be unrelated to the policy followed.

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

TD Control

Difference between online and offline learning

Differences Between Offline and Online Learning

Aspect	Offline Learning	Online Learning
Data	Pre-collected dataset (fixed and static).	Dynamically collected through interaction with the environment.
Interaction	No interaction during training (training is offline).	Continuous interaction and learning during training.
Updates	Policy/value function is updated using the fixed dataset once or iteratively (batch learning).	Policy is updated incrementally after every interaction.
Adaptability	Not adaptive to new environments unless retrained.	Adaptive to changing environments in real-time.
Exploration	Limited to what the dataset covers (no active exploration).	Actively explores to discover new states and actions.

- Temporal-difference (TD) learning has several advantages over Monte-Carlo (MC)
  - Lower variance
  - Online
  - Incomplete sequences

SARSA - On-Policy learning TD control

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

Theorem

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

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

Off-Policy methods

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

Q-Learning algorithm

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

- We have no **explicit** policy written here. We only have a representation of the Q function.
- The target policy is implicit in the greedy term $\max_a Q(S', a)$
- The behaviour policy is the ϵ-greedy version of the target policy.
- Both policies are updated on each step, because we update the Q function after each step.

Target policy and behaviour policy

For on policy method, same policy is used to generate episode and to optimise. However, for off-policy method, the target policy are the one use to optimise and behaviour policy are the one used to generate episode.

Summary MC learning and control I

The Monte Carlo methods presented in this chapter learn value functions and optimal policies from experience in the form of sample episodes. This gives them at least three kinds of advantages over DP methods:

1. In designing Monte Carlo control methods we have followed the overall schema of generalized policy iteration (GPI). Rather than use a model to compute the value of each state, they simply average many returns that start in the state. Because a state's value is the expected return, this average can become a good approximation to the value.
2. Maintaining sufficient exploration is an issue in Monte Carlo control methods. It is not enough just to select the actions currently estimated to be best, because then no returns will be obtained for alternative actions, and it may never be learned that they are actually better.
3. One approach is to ignore this problem by assuming that episodes begin with state-action pairs randomly selected to cover all possibilities. Such exploring starts can sometimes be arranged in applications with simulated episodes, but are unlikely in learning from real experience.
4. In on-policy methods, the agent commits to always exploring and tries to find the best policy that still explores. In off-policy methods, the agent also explores, but learns a deterministic optimal policy that may be unrelated to the policy followed.
5. Off-policy Monte Carlo prediction refers to learning the value function of a target policy from data generated by a different behaviour policy. Such learning methods are all based on some form of importance sampling, that is, on weighting returns by the ratio of the probabilities of taking the observed actions under the two policies.

Function Approximation

Approximating functions

Generalise from seen states to unseen states Update parameter w using MC or TD learning

- Estimate value function with function approximation

$$
\begin{aligned}
V^\pi(s) &\approx \hat{V}(s, \mathbf{w})\\
Q^\pi(s,a) &\approx \hat{Q}(s,a,\mathbf{w})
\end{aligned}
$$

- Generalise from seen states to unseen states (fundamental ability of any good machine learning system)
- Update function approximation parameter $\mathbf{w}$ using MC or TD learning

Stochastic Gradient Descent

Goal: find parameter vector $\mathbf{w}$ minimising mean-squared error between approximate value function $\hat{V}(s,\mathbf{w})$ and true value function $V^\pi(s)$.

$$J(\mathbf{w}) = \mathbb{E}\left[\left(V^\pi(s) - \hat{V}(s,\mathbf{w})\right)^2\right] \qquad (1)$$

Gradient decent finds a local minimum by sliding down the gradient

$$
\begin{aligned}
\Delta\mathbf{w} &= -\frac{1}{2}\alpha \nabla_{\mathbf{w}} J(\mathbf{w}) && (2)\\
&= \alpha \mathbb{E}\left[\left(V^\pi(s) - \hat{V}(s,\mathbf{w})\right)\nabla_{\mathbf{w}}\hat{V}(s,\mathbf{w})\right] && (3)
\end{aligned}
$$

The learning factor $-\frac{1}{2}\alpha$, where the learning rate α controls the step size. The term $\frac{1}{2}$ is chosen so that is cancels out with the derivative of the squared error. The term is negative as we want to perform gradient descent (the gradient points upwards otherwise).
Stochastic gradient descent samples the gradient and the average update is equal to the full gradient update:

$$\Delta w = \alpha\left(V^\pi(s) - \hat{V}(s,\mathbf{w})\right)\nabla_{\mathbf{w}}\hat{V}(s,\mathbf{w}) \qquad (4)$$

Linear Value Function Approximation II

- Update rule is particularly simple

$$
\begin{aligned}
\nabla_{\mathbf{w}}\hat{V}(s,\mathbf{w}) &= \nabla_{\mathbf{w}}(\mathbf{x}(s)^\top \mathbf{w}) = x(s)\\
\Delta\mathbf{w} &= \alpha\left(V^\pi(s) - \hat{V}(s,\mathbf{w})\right)\mathbf{x}(s)
\end{aligned}
$$

Definition

Update = learning step size × prediction error × feature value

Monte-Carlo with Value Function Approximation

- Return R_t is an unbiased, noisy sample of true value $V^\pi(s_t)$
- We apply supervised learning to "training data" of state return trace:

$$(s_1, r_1), (s_2, r_2), \ldots, (s_T, r_T) \qquad (7)$$

- For example, using linear Monte-Carlo policy evaluation

$$
\begin{aligned}
\Delta\mathbf{w} &= \alpha\left(R_t - \hat{V}(s_t,\mathbf{w})\right)\nabla_{\mathbf{w}}\hat{V}(s_t,\mathbf{w}) && (8)\\
&= \alpha\left(R_t - \hat{V}(s_t,\mathbf{w})\right)\mathbf{x}(s_t) && (9)
\end{aligned}
$$

- Monte-Carlo evaluation converges to a local optimum, even when using non-linear value function approximation (provable)

TD Learning with Value Function Approximation I

- The TD-target $R_{t+1} + \gamma\hat{V}(s_{t+1},\mathbf{w})$ is a biased (single) sample of the true value $V^\pi(s_t)$
- We still perform supervised learning on "digested" training data:

$$(s_1, r_2 + \gamma\hat{V}(s_2,\mathbf{w})), (s_2, r_3 + \gamma\hat{V}(s_3,\mathbf{w})), \ldots, (s_T, r_T) \qquad (10)$$

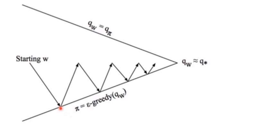
- For example, using linear TD

$$
\begin{aligned}
\Delta\mathbf{w} &= \alpha\left(r + \gamma\hat{V}(s',\mathbf{w}) - \hat{V}(s,\mathbf{w})\right)\nabla_{\mathbf{w}}\hat{V}(s,\mathbf{w}) && (11)\\
&= \alpha\left(r + \gamma\hat{V}(s',\mathbf{w}) - \hat{V}(s,\mathbf{w})\right)\mathbf{x}(s) && (12)
\end{aligned}
$$

- Linear TD converges "close" to the global optimum (provable). This does not extend to non-linear TD (see Sutton & Barto, 2018).

The estimates make the estimates closer to the real V/Q value, but not reach the real V/Q value, in the end, it will get close enough to the true V/Q value.

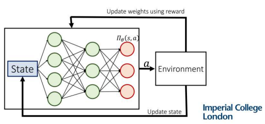
From evaluation to control: FA in GPI



- Policy evaluation: Approximate policy evaluation $\hat{Q}(s,a,\mathbf{e}) \approx Q^\pi(s,a)$
- Policy improvement: ϵ-greedy policy improvement

DQN

Deep Reinforcement Learning



DQN: Bringing Deep Learning into RL I

In principle it sounds easy: use convolutional neural networks to link screen shots to values.
Given our TD Q-learning update:

$$Q(s_t,a) \leftarrow Q(s_t,a) + \alpha\left[r_{t+1} + \gamma \max_{a_{t+1}} Q(s_{t+1},a') - Q(s_t,a)\right] \qquad (1)$$

we want to learn $Q(s_{t+1}, a'; w)$ as a parametrised function (a neural network) with parameters w.
We can define the TD error as our learning target that we want to reduce to zero.

$$\text{TD error}(w) = r_{t+1} + \gamma \max_{a_{t+1}} Q(s_{t+1}, a'; w) - Q(s_t, a; w) \qquad (2)$$

Taking the gradient of E wrt w we obtain:

$$\Delta w = \alpha\left[r + \gamma \max_{a_{t+1}} Q(s_{t+1}, a_{t+1}; w) - Q(s_t, a; w)\right]\nabla_w Q(s_t, a; w) \qquad (3)$$

However, the ATARI playing problem required more engineering to solve properly

1. Experience Replay
2. Target Network
3. Clipping of Rewards
4. Skipping of Frames

DQN: Experience Replay I

The CNN is easily overfitting the latest experienced episodes and the network becomes worse at dealing with different experiences of the game world:

1. Complication 1: Inefficient use of interactive experience
   - Training deep neural networks requires many updates, each has its own transition
   - But now each state transition is used only once, then discarded
   - so we would need to constantly revisit the same state transitions to train.

This is inefficient and slow.

2. Complication 2: Experience distribution
   - Interactive learning produces training samples that are highly correlated (agents recent actions reflect recent policy)
   - The probability distribution of the input data changes