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.

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

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):

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

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

- Vector notation:

$$\mathbf{v} = \mathcal{R} + \gamma \mathcal{P} \mathbf{v} \tag{14}$$

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

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

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

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

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

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

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

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")

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

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

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

Optimal Policy and optimal value and Q function

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

Definition (Optimal Value function)

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

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)

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

In analogy to before we also have

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

$\mathbb{M}^{\pi^*}$ 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

- *The Bellman (Optimality) equations have a unique solution.*
- *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 — Dynamic Programming

Dynamic Programming – Origins 2

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

- Optimal substructure, meaning that the solution to a given optimisation problem can be obtained by the combination of optimal solutions to its sub-problems.
- 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 until value function converge, then be greedy to update the policy, then go back to step 2 until the policy converge.

Policy Iteration Algorithms

1. Initialization
   $V(s) \in \mathbb{R}$ 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 $\leftarrow$ *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* $\leftarrow$ *false*
   If *policy-stable*, then stop; else go to 2

Value Iteration Algorithms

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 — 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 — 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^*.

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)

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

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 $Returns(S_t)$
      $V(S_t) \leftarrow$ average($Returns(S_t)$)

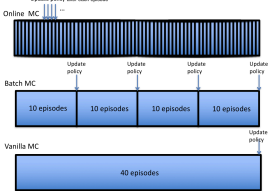
First Visit MC vs Evert Visit MC

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

Batch VS Online Monte-Carlo

Reinforcement Learning — Model-Free Learning — Monte Carlo Learning

Batch & Online Monte-Carlo



No Need to store samples traces

Reinforcement Learning — Model-Free Learning — Monte Carlo Learning

Incremental Monte-Carlo Updates

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

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

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

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

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

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

Temporal Difference Learning

Reinforcement Learning — TD Learning

Temporal-Difference (TD) Learning

- TD methods learn directly from episodes of experience (but also works for non-episodic tasks)
- TD is model-free: no knowledge of MDP transitions or rewards needed
- TD learns from incomplete episodes, by bootstrapping
- 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 — TD Learning

Temporal Difference Learning update rule

Recall that we use the following Monte-Carlo update

$$V(s_t) \leftarrow V(s_t) + \alpha(R_t - V(s_t)) \tag{8}$$

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.

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

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

```
1: procedure TD-Estimation(π)
2:     Init
3:         V̂(s) ← arbitrary value, for all s ∈ S.
4:     EndInit
5:     repeat(For each episode)
6:         Initialise s
7:         repeat(For each step of episode)
8:             a action chosen from π at s
9:             Take action a; observe r, and next state, s'
10:            δ ← r + γV̂(s') − V̂(s)
11:            V̂(s) ← V̂(s) + αδ
12:            s ← s'
13:        until s is absorbing state
14:    until Done
15: end procedure
```

Advantages & Disadvantages of MC & TD

- 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
- TD can learn without the final outcome
- 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.

- on-policy methods, which attempt to evaluate or improve the policy that is used to make decisions
- 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 $\epsilon \in [0, 1]$.

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

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 $Returns(S_t, A_t)$
      $Q(S_t, A_t) \leftarrow$ 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)

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

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

- In designing Monte Carlo control methods we have followed the overall schema of generalised policy iteration (GPI). Rather than use a model to compute the value of each state, they simply average many returns that start in the state. Because a state's value is the expected return, this average can become a good approximation to the value.
- 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.
- 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.
- 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.

- 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

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 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

Initialize $Q(s, a), \forall s \in \mathcal{S}, a \in \mathcal{A}(s)$, arbitrarily, and $Q(terminal\text{-}state, \cdot) = 0$
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)
    $Q(S, A) \leftarrow Q(S, A) + \alpha[R + \gamma Q(S', A') - Q(S, A)]$
    $S \leftarrow S'$; $A \leftarrow A'$;
  until S is terminal

Theorem

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

- *GLIE sequence of policies $\pi^k(a, s)$*
- *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 behaviour policy because it is the policy controlling the agent and generating behaviour.
- The overall problem is called off-policy learning because it is learning about a policy given only experience off (not following) that policy.
- Another mnemonic way to think of is to imagine the cockpit of the agent, and to think if the target policy is switched "on" to run the agent or if it is switched "off" and the agent is left to his own behaviour (policy).

Q-learning

Initialize $Q(s, a), \forall s \in \mathcal{S}, a \in \mathcal{A}(s)$, arbitrarily, and $Q(terminal\text{-}state, \cdot) = 0$
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'
    $Q(S, A) \leftarrow Q(S, A) + \alpha[R + \gamma \max_a Q(S', a) - Q(S, A)]$
    $S \leftarrow S'$;
  until S is terminal

- We have no **explicit** policies 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:

- 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.

- 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.
- 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.
- 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.
- 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 $\mathbf{w}$ 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 $\hat{V}(s, w)$ and true value function $V^\pi(s)$.

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

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:

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

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:

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

- For example, using linear Monte-Carlo policy evaluation

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

- 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:

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

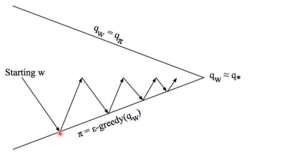
- For example, using linear TD

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

- 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.

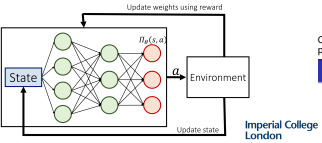
From evaluation to control: FA in GPI



- 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



Imperial College London

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:

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

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.

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

Taking the gradient of E wrt w we obtain:

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

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

   - The network forgets transitions that were in the not so recent past (overwriting past memory, in neuroscience known as "extinction")
   - Moreover, because networks are monolithic, changes in one part of the network can have side-effects on other parts of the network [Why is this not a problem for table-based RL?]

Solution: Experience Replay[2] (alternative use Hierarchical RL).

- Experiences (traces) are stored and replayed in mini-batches to train the neural networks on more than just the last episode.
- Instead of running Q-learning on each state/action pairs as they occur during simulation or actual experience, the experience replay buffer system stores the traces sampled.

- Mini-batches are only feasible when running multiple passes with the same data is somewhat stable with respect to the samples. I.e. the transitions should show low variance/entropy for the next immediate outcomes (reward, next state) given the same state, action pair.
- Replaying past data is also a more efficient use of previous experience, by learning (training the neural network) with it multiple times. This is key when gaining real-world experience is costly (e.g.simulation time) as the Q-learning updates are incremental and do not converge quickly.

The learning phase is separated from gaining experience, and based on taking random samples from the mini-batch table.
DQN interleaves the two processes - acting and learning. Improving the policy will lead to a different behaviour (and different state action pairs), resulting too large replay buffers getting stale.
In summary

1. Reduction of correlation between experiences in updating DNN – mini-batches effectively make the samples more independent and identically distributed.
2. Increased learning speed with mini-batches due to reuse/multi-use of experience
3. Reusing/Replaying past transitions to avoid catastrophic forgetting of associated rewards

Experience Replay Algorithms
Initialize replay memory $\mathcal{D}$ to capacity N
Initialize action-value function Q with random weights
for episode $= 1, M$ **do**
  Initialise state s_t
  for $t = 1, T$ **do**
    With probability ϵ select a random action a_t
    otherwise select $a_t = \max_a Q^*(s_t, a; \theta)$
    Execute action a_t and observe reward r_t and state s_{t+1}
    Store transition (s_t, a_t, r_t, s_{t+1}) in $\mathcal{D}$
    Set $s_{t+1} = s_t$
    Sample random minibatch of transitions (s_t, a_t, r_t, s_{t+1}) from $\mathcal{D}$
    Set $y_j = \begin{cases} r_j & \text{for terminal } s_{t+1} \\ r_j + \gamma \max_{a'} Q(s_{t+1}, a'; \theta) & \text{for non-terminal } s_{t+1} \end{cases}$
    Perform a gradient descent step on $(y_j - Q(s_t, a_j; \theta))^2$
  end for
end for

DQN: Target Network I

Whenever we run an update step on a Q-network's state we also update "near by" states (monlithic architecture + generalisation ability of CNNs).

$$Q(s, a) \leftarrow Q(s_t, a) + \alpha[r_{t+1} + \gamma \max_{a'} Q(s_{t+1}, a') - Q(s_t, a)] \quad (4)$$

Issue: Unstable training resulting from bootstrapping a continuous state space representation

- We may have an unstable learning process, because changes to $Q(s, a)$ change $Q(s', a)$ (with $s - s' \approx \delta$) which changes $Q(s, a)$ on the next CNN update and then in turn $Q(s', a)$ forth. This can lead to a resonance effect
- The effect may result in a run-away bias from bootstrapping and subsequently dominating the system numerically, causing the estimated Q values to diverge.

- In calculating the TD error calculation, the target function is changing too frequently when using convolutional neural networks.

Solution: Slow things down (as resonance dampener)
Using a separate target network Q', updated every fewer time (e.g. every 1000 steps) with a copy of the latest learned weight parameters, controls this stability (relaxation time).

1. Initialise two Q networks: a main Q-network (Q), and a target network (Q')
2. When calculating the TD-error, use Q', not Q
3. Infrequently set $Q' = Q$
4. This gives the highly fluctuating Q time to settle (we can measure this so called "relaxation time" empirically and set it accordingly) before updating Q'

Clipping Rewards: Varation in rewards maeks the training unstable ⇒ Clip positive reward to 1, and negative reward to -1

DQN: Clipping rewards I

Clipping rewards:

- Each ATARI game has different score scales. For example, in Pong, players can get 1 point when winning the play. Otherwise, players get -1 point. However, in Space Invaders, players get 10-30 points when defeating invaders. This difference would make training unstable.
- Clipping Rewards: clips the rewards, with all positive rewards are set +1 and all negative rewards are set -1.

Skipping Frame to reducing computational cost and accelerating training times.

DQN: Skipping frames I

Skipping frames:

- Computer can simulate games (e.g. the ATARI simulator does this at 60 Hz) much faster than a human player would be able to react to the game, and this implies that the video game design and the required actions can be slower.
- Frame skipping uses only every 4 video game frames (i.e. 15 Hz) ... (and frame stacking uses the past 4 frames as inputs), thus reducing computational cost and accelerating training times. It also makes the game run at a speed comparable to human reaction time.

Dobule Q network
Normal network will produce a maximisation bias

Max at the Casino II

- So when update the value for $Q(Start, Left)$ we perform a *max* over actions in successor state Casino (e.g. from traces yielding $r(Casino, A, Hotel) = -1, r(Casino, B, Hotel) = +1, r(Casino, A, Hotel) = -1$) and immediately note down the $+1$ reward from choosing B.
- In the limit of large number of trials all actions will yield an average reward of -0.02, but that is only asymptotically reached.
- This results in a maximisation bias, namely that when taking a *max* over all actions with (very) finite data we may always overestimate values. This illustrates a general relationship (that we do not prove here), but we can see empirically.

Use the main netowrk instead of the target network to do action selection (compared with normal target network)

Double Q Learning (van Hasselt et al, 2017, AAAI) I

Situation: In regular Deep Q-learning we use the target network for two tasks:

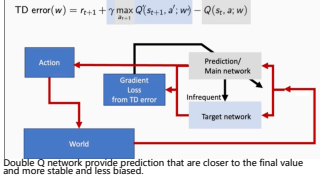
1. Identify action with highest Q value,
2. Determine Q value of this action

Complication: The maximum Q value may be overestimated (variance-bias problem), because an unusually high value from the main network Q does not mean that there is an unusually high value from the target network Q'.
The problem with (Deep) Q-Learning is that the same samples (i.e. the Q network) are being used to decide which action is the best (highest expected reward), and the same samples are also being used to estimate that specific action-value.
Solution: In Double Q-Learning (van Hasselt et al, 2017, AAAI) we separate the two estimates: We use a target network Q' for 1. but

use the regular Q for 2. (or the converse). Thus, we make the selection of the action with highest Q value, and selection of the Q value used in Bellman updates become independent from each other (different networks) so we become less susceptible to variance in each estimate. This reduces the frequency by which the maximum Q value may be overestimated, as it is less likely that both the networks are overestimating the same action.

Target Network

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



Double Q network provide prediction that are closer to the final value and more stable and less biased.

Dealing with Bias II

- The two straight horizontal orange (for DQN) and blue (for Double DQN) lines in the top row are computed by running the corresponding agents after learning concluded (end of training), and averaging the actual discounted return obtained from each visited state. These straight lines would match the learning curves at the right side of the plots if there is no bias during learning and training lasted long enough.
- We thus see that learning with DDQN is much not only more realistic (closer to the final values) and more stable and less biased (far less systematic overshooting).

Policy Gradient
Why using policy based method:

For some environment that has large action space (especially continuous action space), iterate through all actions might take a long time, thus have a policy that direct output action would be more efficient.

Policy gradients interim summary I

- Using gradient ascent, we can move toward the direction suggested by the gradient to find the best policy that produces the highest return.
- A value function may be used to learn the policy weights but this is not required for action selection
- Policy gradient methods are methods for learning the policy weights using the gradient of some performance measure with respect to the policy weights
- Search directly for the optimal policy π^* by optimising policy parameters θ
- Policy gradient methods seek to maximise performance and so the policy weights are updated using gradient ascent

Objective of Policy Gradient
Recall that the optimal policy is the policy that achieves maximum expected (future) return

- Find

$$\theta^* = \arg\max_\theta \mathbb{E}\left[\sum_t r(s_t, a_t)\right]_{\tau \sim p_\theta(\tau)}$$

(infinite horizon case)

-

$$\theta^* = \arg\max_\theta \sum_{t=0}^{T} \mathbb{E}\left[r(s_t, a_t)\right]_{(s_t, a_t) \sim p_\theta(\tau)}$$

(finite horizon case)

We can use any parametric supervised machine learning model to learn policies $\pi(a|s;\theta)$ where θ represents the learned parameters Finit difference are simple, inefficient(especially when have tons of parameters), and sometime efficient (it can even work for non-differentiable ones)

Approach 1: Finite Difference Policy Gradient III

- The partial derivative of the k-th component of θ is approximated by finite differences

$$\frac{\partial J}{\partial \theta_k} \approx \frac{J(\boldsymbol{\theta} + \boldsymbol{u}_k\epsilon) - J(\boldsymbol{\theta})}{\epsilon}$$

- Update rule $\theta_k = \theta_k + \alpha(\frac{(J(\boldsymbol{\theta} + \boldsymbol{u}_k\epsilon) - J(\boldsymbol{\theta})}{\epsilon})$

This is a simple, noisy and inefficient procedure that is sometimes effective, as it works for arbitrary policies (even non-differentiable ones).
Could we find a less empirical/numerical approach to optimise the policy gradient?

Directly calculating the policy gradient

$\theta^* = \arg\max_\theta J(\theta)$

$J(\theta) = E_{\tau \sim \pi_\theta(\tau)}[r(\tau)]$ — log of both sides $\log \pi_\theta(\tau) = \log p(s_1) + \sum_{t=1}^{T} \log \pi_\theta(\mathbf{a}_t|\mathbf{s}_t) + \log p(\mathbf{s}_{t+1}|\mathbf{s}_t, \mathbf{a}_t)$ — $\log \prod_{t=1}^{T} \pi_\theta(\mathbf{a}_t|\mathbf{s}_t) p(\mathbf{s}_{t+1}|\mathbf{s}_t, \mathbf{a}_t) = p(\mathbf{s}_1) \prod_{t=1}^{T} \pi_\theta(\mathbf{a}_t|\mathbf{s}_t) p(\mathbf{s}_{t+1}|\mathbf{s}_t, \mathbf{a}_t)$ — $\pi_\theta(\tau)$

$\nabla_\theta J(\theta) = E_{\tau \sim \pi_\theta(\tau)}[\nabla_\theta \log \pi_\theta(\tau) r(\tau)]$

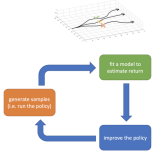
$\nabla_\theta \left[\log p(\mathbf{s}_1) + \sum_{t=1}^{T} \log \pi_\theta(\mathbf{a}_t|\mathbf{s}_t) + \log p(\mathbf{s}_{t+1}|\mathbf{s}_t, \mathbf{a}_t)\right]$

$\nabla_\theta J(\theta) = E_{\tau \sim \pi_\theta(\tau)}\left[\left(\sum_{t=1}^{T} \nabla_\theta \log \pi_\theta(\mathbf{a}_t|\mathbf{s}_t)\right)\left(\sum_{t=1}^{T} r(\mathbf{s}_t, \mathbf{a}_t)\right)\right]$

We arrived at this final equation, a version of the policy gradient theorem, by expanding the state sequence from a fixed starting point $\mathbf{s}_1$. More generally we can obtain results for the stationary distribution of starting states (not discussed here).
REINFORCE algorithms

REINFORCE-ing good trajectories over bad ones

A direct implementation of the Policy Gradient Theorem result is the classic 3-step Policy Gradient algorithm REINFORCE.



- Before we had to learn a value function through function approximation and then derive a corresponding policy
- Often learning a value function can be intractable (more unstable / was, at the time, intractable for large state spaces).
- REINFORCE provided a way to directly optimize policies to get around this problem.

REINFORCE algorithm:

1. sample $\{\tau^i\}$ from $\pi_\theta(\mathbf{a}_t|\mathbf{s}_t)$ (run the policy)
2. $\nabla_\theta J(\theta) \approx \sum_i \left(\sum_t \nabla_\theta \log \pi_\theta(\mathbf{a}_t^i|\mathbf{s}_t^i)\right)\left(\sum_t r(\mathbf{s}_t^i, \mathbf{a}_t^i)\right)$
3. $\theta \leftarrow \theta + \alpha \nabla_\theta J(\theta)$

- The method suffer from high variance in the sampled trajectories, thus stabilising model parameters is difficult.

What was the impact of REINFORCE? I

- Before: we had to learn a value function through function approximation and then derive a corresponding policy
- But, often learning a value function can be intractable (more unstable, computationally demanding for large state spaces, observability).
- REINFORCE: provided a way to directly optimize policies to get around this problem. It was the direct result of rendering the PG theorem useful after its derivation

REINFORCE was the crucial first step at popularising Policy Gradients (PGs).

REINFORCE algorithms suffer greatly from variance, a single erratic trajectory can cause a suboptimal shift into wired area of optimisation.

REINFORCE

- In the limit of large amounts of data, the model will converge to the optimal parameters
- But, the REINFORCE method suffers from high variance in the sampled trajectories, thus stabilising model parameters θ can be difficult.
- Any erratic trajectory can cause a sub-optimal shift in the policy distribution if by chance generates high rewards or vice versa.
- Many subsequent algorithms were proposed to reduce the variance while keeping the bias unchanged by being smarter correlating rewards with trajectories. These may involve subtracting a baseline from the reward term (so as to keep values smaller) and by using a so called advantage term (Schulman et al, 2016).

Gaussian Policy
In robotics and many other real-world control systems it makes sense to have continuous policies. A simple way is to introduce Gaussian policies, i.e. $\pi(a|s)$ as a Gaussian distributions

$$\nabla_\theta J(\theta) \approx \frac{1}{N}\sum_{i=1}^{N}\left(\sum_{t=1}^{T}\nabla_\theta \log \pi_\theta(\mathbf{a}_{i,t}|\mathbf{s}_{i,t})\right)\left(\sum_{t=1}^{T} r(\mathbf{s}_{i,t}, \mathbf{a}_{i,t})\right)$$

example: $\pi_\theta(\mathbf{a}_t|\mathbf{s}_t) = \mathcal{N}(f_{\text{neural network}}(\mathbf{s}_t); \Sigma)$

$$\log \pi_\theta(\mathbf{a}_t|\mathbf{s}_t) = -\frac{1}{2}\|f(\mathbf{s}_t) - \mathbf{a}_t\|_\Sigma^2 + \text{const}$$

$$\nabla_\theta \log \pi_\theta(\mathbf{a}_t|\mathbf{s}_t) = -\frac{1}{2}\Sigma^{-1}(f(\mathbf{s}_t) - \mathbf{a}_t)\frac{df}{d\theta}$$

We can interpret the term $f(\mathbf{s}_t, \theta) - \mathbf{a}_t$ as a curious form of supervised learning - where we fit the network output computed on the states $f(s)$ to the actions a.
Policy gradients is trial-and-error (like MC)
It is curious because pure maximum-likelihood-type supervised learning (fitting states and actions to each other) is actually reward-weighted fitting in Policy-Gradient-based reinforcement learning.

$\nabla_\theta J(\theta) \approx \frac{1}{N}\sum_{i=1}^{N}\left(\sum_{t=1}^{T}\nabla_\theta \log \pi_\theta(\mathbf{a}_{i,t}|\mathbf{s}_{i,t})\right)\left(\sum_{t=1}^{T} r(\mathbf{s}_{i,t}, \mathbf{a}_{i,t})\right)$

$\nabla_\theta J(\theta) \approx \frac{1}{N}\sum_{i=1}^{N}\sum_{t=1}^{T}\nabla_\theta \log \pi_\theta(\tau_i) r(\tau_i)$ — maximum likelihood: $\nabla_\theta J_{\text{ML}}(\theta) \approx \frac{1}{N}\sum_{i=1}^{N}\nabla_\theta \log \pi_\theta(\tau_i)$

$\sum_{t=1}^{T}\nabla_\theta \log_\theta \pi_\theta(\mathbf{a}_{i,t}|\mathbf{s}_{i,t})$

These averages can be reinterpreted as weighted average over policies

- We increase the weight of trajectories that are good in reward
- We decrease the weight of trajectories that are bad in rewards

The action in future cannot affect the actions in the past.

Ways to reduce variance: Causality I

- Apply Causality to reduce the amount of total data (and thus variability):
- Strictly speaking $\nabla_\theta J(\theta) = \sum_{t=1}^{N}(\sum_{t=1}^{T}\nabla_\theta \log \pi_\theta(\mathbf{a}_t'|\mathbf{s}_t'))(\sum_{t=1}^{T} r(\mathbf{s}_t', \mathbf{a}_t'))$
- but, the policy at time $t' > t$ cannot affect the reward in the past (time t), so we can restrict the terms entering the sums tp $\nabla_\theta J(\theta) = \sum_{t=1}^{N}(\sum_{t=1}^{T}\nabla_\theta \log \pi_\theta(\mathbf{a}_t'|\mathbf{s}_t'))(\sum_{t'=t}^{T} r(\mathbf{s}_t', \mathbf{a}_t'))$

Actor Critics
Policy based method has low bias but larger variance

Actors & Critics

- Policy-Based methods (actors)
  - Find the optimal policy directly without the Q/V-function.
  - Policy-based are considered to have a faster convergence and are better for continuous and stochastic environments.
  - Examples: Policy Gradients algorithms such as REINFORCE
- Value Based methods (critics)
  - Find/approximate optimal value function (mapping action to value).
  - These are considered more sample efficient and steady.
  - Examples: Q Learning, Deep Q Networks, Double Deep Q Networks, etc.

Best of both worlds with Deep Actor-Critic Methods I

Why combine deep RL methods as Actor-Critics?

- Policy Gradient agents might take a lot of actions over the course of an episode, it is hard to assign correct credit to many actions, which means that these updates have a high variance
- Therefore, it may take a lot of updates for your policy to converge.
- Moreover, PG method only works in episodic regimes (cf. Monte Carlo tabular RL). If your agent is not acting in an episodic environment, it will never get an update (cf. "You cannot backup from death")

TD Error

- Generate the trace
- Update the policy value
- calcualte the TD error
- update the Q value error

A simple Q-driven policy-gradient actor-critic.

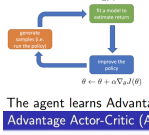
Algorithm 1 Q Actor Critic
Initialize parameters s, θ, w and learning rates α_θ, α_w; sample $a \sim \pi_\theta(a|s)$.
for $t = 1 \ldots T$: **do**
  Sample reward $r_t \sim R(s, a)$ and next state $s' \sim P(s'|s, a)$
  Then sample the next action $a' \sim \pi_\theta(a'|s')$
  Update the policy parameters: $\theta \leftarrow \theta + \alpha_\theta Q_w(s, a)\nabla_\theta \log \pi_\theta(a|s)$; Compute the correction (TD error) for action-value at time t:
  $\delta_t = r_t + \gamma Q_w(s', a') - Q_w(s, a)$
  and use it to update the parameters of Q function:
  $w \leftarrow w + \alpha_w \delta_t \nabla_w Q_w(s, a)$
  Move to $a \leftarrow a'$ and $s \leftarrow s'$
end for

Advantage Actor Critic (A2C)

1. sample $\{\mathbf{s}_i, \mathbf{a}_i\}$ from $\pi_\theta(\mathbf{a}|\mathbf{s})$ (run it on the robot)
2. fit $\hat{V}_\phi^\pi(\mathbf{s})$ to sampled reward sums
3. evaluate $\hat{A}^\pi(\mathbf{s}_i, \mathbf{a}_i) = r(\mathbf{s}_i, \mathbf{a}_i) + \hat{V}_\phi^\pi(\mathbf{s}_i) - \hat{V}_\phi^\pi(\mathbf{s}_i)$
4. $\nabla_\theta J(\theta) \approx \sum_i \nabla_\theta \log \pi_\theta(\mathbf{a}_i|\mathbf{s}_i)\hat{A}^\pi(\mathbf{s}_i, \mathbf{a}_i)$
5. $\theta \leftarrow \theta + \alpha \nabla_\theta J(\theta)$

$y_{i,t} \approx r(\mathbf{s}_{i,t}, \mathbf{a}_{i,t}) + \hat{V}_\phi^\pi(\mathbf{s}_{i,t+1})$

$\mathcal{L}(\phi) = \frac{1}{2}\sum_i \left\|\hat{V}_\phi^\pi(\mathbf{s}_i) - y_i\right\|^2$



The agent learns Advantage values instead of Q values.

Advantage Actor-Critic (A2C) IV

- That way the evaluation of an action is based not only on how good the action is, but also how much better it can be.
- This reduces high variance of policy networks and stabilises the model during training.
- Side note, Advantages also allow us to reduce variance without introducting bias in policy gradient methods (we weight a trajectory rollout of the policy with the advantage, not its return)

A3C is the parallel and asynchronous version of A2C, the there is a global network which takes the gradient of each agent, and the agent will sometimes update their states according to global network.

Asynchronous Advantage Actor-Critic (A3C) II

- The agents (or workers) are trained in parallel and update periodically a global network, which holds shared parameters.

- The updates are not happening simultaneously and thatâs where the asynchronous comes from.

DDPG (Lillicrap et al.,2015) I

Deep Deterministic Policy Gradients (DDPG) is a deterministic policy, model-free, off-policy, actor-critic algorithm which

- learns the value model (Q-function) and a policy π.
- learns by off-policy methods and uses the Bellman equation to learn the Q-function (boostrapping part 1)
- uses the Q-function to learn the policy π (bootstrapping part 2)

We will next look first at the algorithm (as re-printed from Lillicrap et al, 2015) and then what ideas it contains. Note, Lillicrap 2015's notation differs from ours: θ_Q is the parameters of the value function approximation (we used w) and θ_μ is the parameters of the policy function approximation (we used θ). R is the replay buffer (we used it for the total Return).

DDPG (Lillicrap et al.,2015) I

To get DDPG to work well a few special elements were introduced:

- Exploration in continuous action spaces
  - Use Gaussian distributed noise that perturbs a mean action $\mu'(s) = \mu_\theta(s) + \mathcal{N}$ (i.e. Gaussian Policies)
  - This is almost equivalent to discrete action ϵ-greediness (only that we explore more often and much closer to the intended action μ).
- To control learning variability DDPG uses replay buffers/minibatches & target networks (just as in DQN)
- Error-based learning on the mean-squared Bellman error (MSBE)

  - MSBE (here denoted by L, tells us how closely our approximation of Q comes to satisfying the Bellman equation (compare this to function approximation Q-learning/DQN). $L = \frac{1}{N}\sum_i (y_i - Q(s_i, a_i|\theta_Q))^2$
- Policy gradient based learning on the actor network
  - Here we are using a deterministic action version of the PG theorem, which proof (not examinable here) was published in Silver et al (2014).
  - DQN updates the target network in jumps with many episodes remaining frozen. This does not work well in continuous control cases, where small changes in policy need to remain linked to small changes in control.
  - DDPG therefore uses "smooth" updates on the parameter vector θ (that is shared by both actor and critic) which updates slowly but steadily over steps t: $\theta^{t+1} = \beta\theta^t + (1-\beta)\theta^{t'}, 0 < \beta \ll 1$

  - The parameter β controls the degree of blending (or forgetting) between the current parameter estimate θ and its update θ' at t to determine the parameter vector at the next $t+1$.

Just like the Policy Gradient theorem can be thought of a policy-centered equivalent of the Bellman Theorem, DDPG can considered the continuous action version of the discrete action DQN. Why?

Desired qualities for Reinforcement Learning

- Stable learning & little sensitivity to hyperparameters: we want to avoid parameter tuning which is (computationally) expensive and more an art than a science. Ideally, RL algorithms work out of the box.
- Sample Efficiency: Good sample complexity is the prerequisite for efficient (and in some cases effective) skill acquisition.
- Most algorithms we covered so far are relatively poor at sample efficiency (although we have seen how introducing features imporaved their stability of learning). This is ok for simulation settings or "toy"-problems such as video games.

- Many real-world applications, e.g. robots require us to put this second aspect, good exploration at the centre of our attention, as interactive learning is costly and does not scale well.

An operational definition of the sample complexity of exploration for a reinforcement learning algorithm is the number of time steps in which the algorithm does not select near-optimal actions.

SAC - Objective Function I

In the Soft Actor-Critic we use the entropy concept to boost its sample complexity. The primary goal of SAC is to maximize a modified expected cumulative reward, which includes an entropy term to encourage exploration:

$$J(\pi) = \mathbb{E}_{(\mathbf{s}_t, \mathbf{a}_t) \sim \rho_\pi}\left[\sum_t \gamma^t \left(r(\mathbf{s}_t, \mathbf{a}_t) + \lambda\mathcal{H}(\pi(\cdot|\mathbf{s}_t))\right)\right] \quad (7)$$

Here,

- $J(\pi)$ is the cost-to-go objective function,
- $r(\mathbf{s}_t, \mathbf{a}_t)$ is the immediate reward function,
- $\rho_\pi = p_\pi(s, a)$ is the state-action distribution under policy π,
- γ is the discount factor,
- λ is a temperature parameter controlling the importance of the entropy term $\mathcal{H}$,
- $\pi(\cdot|\mathbf{s}_t)$ is the policy distribution given state $\mathbf{s}_t$.

SAC - Critic Update I

SAC employs two Q-value functions, Q_{w_1} and Q_{w_2} (cf. target networks in DDQN), to mitigate maximisation bias in policy improvement. In this entropy-regularized reinforcement learning, the agent gets an bonus reward at each time step proportional to the entropy of the policy at that timestep.

The critic (Q-functions) are updated to minimize the Bellman residual:

$$J_Q(\theta_i) = \mathbb{E}_{(\mathbf{s}_t, \mathbf{a}_t) \sim \mathcal{D}}\left[\frac{1}{2}\left(Q_{w_i}(\mathbf{s}_t, \mathbf{a}_t) - \left(r(\mathbf{s}_t, \mathbf{a}_t) + \gamma\mathbb{E}_{\mathbf{s}_{t+1} \sim \rho}\left[\min_{i=1,2} Q_{w_i}(\mathbf{s}_{t+1}, \mathbf{a}') - \lambda\log\pi(\mathbf{a}'|\mathbf{s}_{t+1})\right]\right)\right)^2\right]$$

where $\mathcal{D}$ is the replay buffer and ρ_{dyn} represents the environment dynamics. Note $\mathbf{a}'$ comes from the current policy vπ, not the replay buffer

SAC - Critic Update II

$$J_Q(\theta_i) = \mathbb{E}_{(\mathbf{s}_t, \mathbf{a}_t) \sim \mathcal{D}}\left[\frac{1}{2}\left(Q_{w_i}(\mathbf{s}_t, \mathbf{a}_t) - \left(r(\mathbf{s}_t, \mathbf{a}_t) + \gamma\mathbb{E}_{\mathbf{s}_{t+1} \sim \rho}\left[\min_{i=1,2} Q_{w_i}(\mathbf{s}_{t+1}, \mathbf{a}') - \lambda\log\pi(\mathbf{a}'|\mathbf{s}_{t+1})\right]\right)\right)^2\right]$$

The Eq. can be interpreted as follows: Across all sampled states from our experience replay buffer, decrease the squared difference between the prediction of our Q-value network and the expected prediction of the Q function plus the entropy of the policy function π measured here by the negative log of the policy function (note, not to be confused with the log π as a Policy Gradient term).