Decision Making Under Uncertainty Ch4-Ch5 Stanford AA228 course notes – Philippe Weingertner

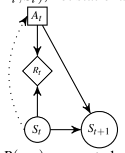
DMU Ch.4 Sequential Problems

Optimal decisions making requires reasoning about future sequences of actions and observations

Assumption in Ch.4: model is known i.e. we know $T(s' \mid s,a), R(s,a)$ and environement is fully observable

Formulation

MDP $< \mathscr{S}, \mathscr{A}, T, R >$ stationary representation
Markov assumption: current state only depends on your previous state and the action you took to get there
Stationary: T, R do not change with time
Decision Network but with $P(S_{t+1} \mid S_t, A_t)$ and $P(R_t \mid A_t, S_t)$, not stationary, in the general case



$R(s,a)$: expected reward received when executing action a from state s. Here we assume it is a deterministic function (but not required).
Utility function decomposed into rewards $R_{0:t}$

Utility and Reward
Finite horizon:
- $\sum_{t=0}^{n-1} r_t$

Infinite horizon:
- discounted sum $\sum_{t=0}^{\infty} \gamma^t r_t$ with $\gamma \in [0,1($
- or average reward $\underset{n\to\infty}{lim} \frac{1}{n} \sum_{t=0}^{n-1} r_t$

Discount factor γ : gives a higher value to plans that reach a reward sooner and bounds the utility of a state, ensuring it does not reach infinity.
A small γ discounts future rewards more heavily. A solution with a small γ will be greedy, meaning that it will prefer immediate rewards more than long-term rewards. The opposite is true for a large discount factor.

Dynamic Programming

DP is a method for solving problems by breaking them into subproblems. DP is more efficient than brute force methods because it leverages the solutions of subproblems. Examples: Viterbi, shortest path ...

Simplify a complicated problem by breaking it down into simpler sub-problems in a recursive manner.

Policy: determines action given past history of states and actions $a = \pi_t(h_t) = \pi_t(s_{0:t}, a_{0:t-1})$

But with MDP we just care about current state as s_t d-separates past from future

$\Longrightarrow \pi_t(s_t)$ or $\pi(s_t)$ if the policy is stationary

With finite horizon you may want to change your policy (e.g. if you know you will die tomorrow)

T^π is the transition function for only the policy π so we have $T(s' \mid s, \pi(s))$ and the next states s' is now only a function of the current state s

$U^\pi(s) = Q^\pi(s, \pi(s))$: **expected utility** of executing π from state s aka *Value function*

$U(s) = \underset{a}{max}\, Q(s,a)$

$U^* = \underset{\pi}{max}\, U^\pi$ is the optimal value function i.e. the value of being in state s assuming you follow an optimal policy

An optimal policy π^* is a policy that maximizes expected utility: $\pi^*(s) = \underset{\pi}{argmax}\, U^\pi(s)$ for all states s

The optimal policy isn't necessarily unique, but the optimal value for each state is unique.

Recursive Equation $\Rightarrow$ Dynamic Programming
$U_k^\pi(s) = R(s, \pi(s)) + \gamma \sum_{s'} T(s' \mid s, \pi(s)) U_{k-1}^\pi(s')$
With a fixed policy π
The iteration number, subscript k, can be interpreted as the number of steps ***remaining*** to the end
$\gamma \to 1 \Rightarrow$ less myopic

Bellman Equations
$U_k^*(s) = \underset{a}{max}[R(s,a) + \gamma \sum_{s'} T(s' \mid s,a) U_{k-1}^*(s')]$
$\pi^*(s) = \underset{a}{argmax}[R(s,a) + \gamma \sum_{s'} T(s' \mid s,a) U^*(s')]$
$U^*(s) = \underset{a}{max}[R(s,a) + \gamma \sum_{s'} T(s' \mid s,a) U^*(s')]$

PI and VI are algorithms used to find optimal policies. Both are forms of DP. PI directly updates the policy. VI updates the expected utility of each state using the Bellman equation and retrieves the optimal policy from those utilities.

Policy Evaluation
Computing the expected utility obtained from executing a policy. It can be computed:
DP with $U_0^\pi(s) = 0, U_1^\pi(s) = R(s, \pi(s))$ and then
$U_t^\pi(s) = R(s, \pi(s)) + \gamma \sum_{s'} T(s' \mid s, \pi(s)) U_{t-1}^\pi(s')$
For ∞ horizon, computed well with enough iterations:
$U^\pi(s) = R(s, \pi(s)) + \gamma \sum_{s'} T(s' \mid s, \pi(s)) U^\pi(s')$
Or by solving a system of $|S|$ linear equations
matrices: $U^\pi = R^\pi + \gamma T^\pi U^\pi$
dims: $n \times 1 = n \times 1 + (n \times n)(n \times 1)$
$U^\pi = (1 - \gamma T^\pi)^{-1} R^\pi$ in $O(n^3)$

Policy Iteration: every steps (policy evaluation then policy improvement) leads to improvement. As nb policies is finite $\Longrightarrow$ algo terminates with an optimal solution

```
function PolicyIteration($\pi_0$)
    $k \leftarrow 0$
    repeat
        Compute $U^{\pi_k}$
        $\pi_{k+1}(s) = argmax_a[R(s,a) + \gamma \sum_{s'} T(s'|s,a) U^{\pi_k}(s')]$ for all states $s$
        $k \leftarrow k+1$
    until $\pi_k = \pi_{k-1}$
    return $\pi_k$
```

Variants of PI: e.g. approx U^{π_k} using only a few iters of iterative policy eval instead of computing U exactly.

Value Iteration: simple and easy to implement.

Terminates when *residual* $= \|U_k - U_{k-1}\| < \delta = \varepsilon \frac{1-\gamma}{\gamma}$ which gurantees U is within ε of U^* and policy loss is less than 2ε

```
function ValueIteration
    $k \leftarrow 0$
    $U_0(s) \leftarrow 0$ for all states $s$
    repeat
        $U_{k+1}(s) \leftarrow max_a[R(s,a) + \gamma \sum_{s'} T(s'|s,a) U_k(s')]$ for all states $s$
        $k \leftarrow k+1$
    until convergence
    $\pi_k(s) \leftarrow \arg\max_a \left[R(s,a) + \gamma \sum_{s'} T(s'|s,a) U_k(s')\right]$
    return $U_k, \pi_k$
```

In asynchronous VI, only a subset of the states may be updated per iteration. The state ordering is important because different orderings can take a different number of iterations to converge to the optimal value function.

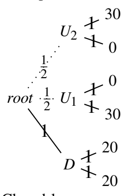
Gauss-Seidel Value Iteration
Is an asynchronous VI where only a subset of the states is updated per iteration. We sweep through an ordering of states and **update in place one state at a time**.
No copy(U) required, 50% mem required, and usually faster CV

You must understand how U(s) is updated after 1st, 2nd, 3rd ... sweep around a reward state (diffusion effect) and how λ impacts the final $U(s)$ (the higher, the more the rewards propagate far away. The smaller the λ, the shortest path to a low reward is taken. even if a better reward exist farther away. With higher λ we would take a longer path to a higher reward)

Planning: closed and open loop
Planning: process of using a model to choose an action in a sequential problem
Closed-loop:
- Accounts for future state information and uncertainty $\Longrightarrow$ develop a reactive plan
- VI is more difficult to scale

Open-loop:
- Develop a static plan. Execute first steps and replan. Many PP algos here. MPC as well.
- MPC can scale easily to large and continuous spaces

Example:



Closed-loop:
- plan UP and then either UP or DOWN for $U = 30$

Open-loop:
- plan DOWN, DOWN for $U(D,D) = 20$
- bcz $U(UP, UP) = U(UP, DOWN) = 15$

Closed-loop control attempts to compute an optimal action for every state you can be in. Open-loop control computes an optimal trajectory, and gives you the action to take along the trajectory, but doesn't tell you what to do if you deviate from the trajectory. Open-loop planning methods do not account for future state information. In control systems engineering this future state information is called feedback. Closed-loop methods plan by anticipating feedback from the environment and considering the ability to take actions in future states.

Structured Representations

Factored MDP: dynamic Decision Network. Actions, rewards and states factored into multiple nodes. Then use Decision Trees (or Decision Diagrams) to represent conditional probabilities.

Structured DP:

- perform updates on the leaves of the DT instead of all the states
- aggregate states and leverage additive decomposition of R and U functions.
- resulting policy may be represented as a DT: interior nodes test state var and leaf nodes correspond to actions

Linear Representations

We can find exact optimal policy for continuous state and action spaces (no discretization required) if:

Two assumptions for exact continuous solution
1. Dynamics are linear Gaussian: $T(z \mid s,a) = \mathcal{N}(z \mid T_s s + T_a a, \Sigma)$ with z, s, a vectors
2. Reward is quadratic: $R(s,a) = s^\top R_s\, s + a^\top R_a\, a$ where R_s negative semi-definite and R_a negative definite (<0)

$U_n(s) = s^\top V_n\, s + q_n = scalar$ and $\pi_n(s)$ is a product of matrices operations (transpose, multiplication, inverse)
Subscript n is the number of remaining steps (book: assumed finite horizon undiscounted reward problem)
PBs when you have multi-modalities

Approximate Dynamic Programming

Finding approximately optimal policies for problems with large or continuous spaces. Shares ideas with RL (i.e. without known model)

Local Approximation
Grid discretization to n points and multilinear interpol
E.g. $U(s)$ interpolation based on 4 nearest neighbors with a weight that is distance dependent. But the way we compute $\lambda_i = U(s_i)$ is a bit counter-intuitive ! Cf algo LocalApproxValueIteration
$U(s) = \sum_{i=1}^n \lambda_i \beta_i(s)$ with λ_i value at s_i weighted by $\beta_i(s)$ s.t. $\sum_i \beta_i(s) = 1$ and more weight to closest states
$U(s) = \vec{\lambda}^\top \vec{\beta}(s)$
Nearest Neighbor: $\beta_i(s) = \begin{cases} 1 & \text{if } s_i \text{ closest to } s_i \\ 0 & \text{otherwise} \end{cases}$
Linear interpolation with $N(s) = \{s_1, s_2\}$ from $s_{1:n}$: $U(s) = \lambda_1 \left(1 - \frac{s - s_1}{s_2 - s_1}\right) + \lambda_2 \left(1 - \frac{s_2 - s}{s_2 - s_1}\right)$
When $n \nearrow$ we have a grid with 2^n points per hyper-cube $\Longrightarrow$ do linear interpolation on a simplex of $n+1$ points (in $2D$: within triangles instead of squares)

$\lambda, u, \beta(s')$ are vectors in the below algo

You have to fix β and it will CV to a unique λ solution

```
function LocalApproxValueIteration
    $\lambda \leftarrow 0$
    loop
        for $i \leftarrow 1$ to $n$ do
            $u_i \leftarrow max_a[R(s_i,a) + \gamma \int_{s'} T(s'|s_i,a) \lambda^T \beta(s')]$
        $\lambda \leftarrow u$
    return $\lambda$
```

Global Approximation
We are fitting a surface over the whole state space.
E.g. polynomial approximation of $U(s) = \sum_{i=1}^m \lambda_i s^i$
Or use a Neural Network ...
Use m features as inputs to the approx value fct
$U(s) = \sum_{i=1}^m \lambda_i \beta_i(s)$
Set of basis functions or features: $\beta_i : \mathscr{S} \to \mathbb{R}$ e.g. $\beta_2(s) = s^3$
$\lambda_{1:m}$ are a set of parameters for linear approximation

```
function LinearRegValueIteration
    $\lambda \leftarrow 0$
    loop
        for $i \leftarrow 1$ to $n$ do
            $u_i \leftarrow max_a[R(s_i,a) + \gamma \sum_{s'} T(s'|s_i,a) \lambda^T \beta(s')]$
        $\lambda_{1:m} \leftarrow Regress(\beta, s_{1:n}, u_{1:n})$
    return $\lambda$
```

Linear Regression in Global Approximation
The Regress function finds the λ that leads to the best approximation of the target values $u_{1:n}$ at points $s_{1:n}$ using the basis fct β. A common regression objective is MSE: $\sum_{i=1}^n (\lambda^T \beta(s_i) - u_i)^2$. Linear Least-Squares regression can compute the λ that min sum-squared error through simple matrix ops.

Online Methods

Do not compute the policy for the entire state space offline. Restrict computation to states reachable from current state. May be a MUCH smaller space !!!

Downside: intensive computation and simulation at each timestep so it might not be appropriate when computational resources are at a premium.

NB: in Ch 4, we know our model $T(s' \mid s,a), R(s,a)$

Forward Search
Simple online action-selection: looks ahead from s_0 to some depth d . Full enumeration. As long as your goal is within d-steps you get it
Complexity: $O((|S| \times |A|)^d)$ exponential time

```
function SelectAction($s, d$)
    if $d = 0$ then
        return $(NIL, 0)$                    ▷ BB: (NIL, $\underline{U}$(s))
    $(a^*, v^*) \leftarrow (NIL, -\infty)$
    for $a \in A(s)$ do ▷ BB: in descending order of $\overline{U}$
        $v \leftarrow R(s,a)$        ▷ BB: prune if $\overline{U}(s,a) < v^*$
        for $s' \in S(s,a)$ do
            $(a', v') \leftarrow$ SelectAction$(s, d-1)$
            $v \leftarrow v + \gamma T(s' \mid s,a) v'$
        if $v > v^*$ then
            $(a^*, v^*) \leftarrow (a, v)$
    return $(a^*, v^*)$
```

Branch and Bound Search
Prune: do not recurse down path of low values
Lower bound for $\underline{U}(s)$ and Upper bound for $\overline{U}(s,a)$ provided by expert knowledge
- If best possible thing that can happen with (s,a) i.e. $\overline{U}(s,a) < v^* \Longrightarrow$ we can end here
- NB: for $a \in A(s)$ must be in descending order of upper bound to prune correctly
- At end of exploration depth: return $\underline{U}(s)$ as value estimate

Worst case complexity as Forward Search
If $\underline{U}$ and $\overline{U}$ are true you are not compromising optimality

Sparse Sampling
Use a Generative Model $(s', r) \sim G(s,a)$
No need for explicit T, R here
Algo similar to forward search except it iterates over n samples averaged together so that in the loop $v \leftarrow v + (r + \gamma v')/n$ instead of $v \leftarrow v + \gamma T(s' \mid s,a) v'$
Avoids worst case complexity but still $O((n \times |A|)^d)$
May compromise optimality

```
function SparseSamplingSelectAction($s, d$)
    if $d = 0$ then
        return $(NIL, 0)$
    $(a^*, v^*) \leftarrow (NIL, -\infty)$
    for $a \in A(s)$ do
        $v \leftarrow 0$
        for $i \leftarrow 1$ to $n$ do
            $(s', r) \sim G(s,a)$
            $(a', v') \leftarrow$ SelectAction$(s', d-1)$
            $v \leftarrow v + (r + \gamma v')/n$
        if $v > v^*$ then
            $(a^*, v^*) \leftarrow (a, v)$
    return $(a^*, v^*)$
```

MCTS vs SS: complexity does not grow exp with d and in the limit CV $\to U^*$

```
function MctsSelectAction($s, d$)
    loop
        Simulate$(s, d, \pi_0)$
    return arg max$_a$ $Q(s,a)$
function Simulate$(s, d, \pi_0)$
    if $d = 0$ then
        return 0
    if $s \notin T$ then                          ▷ Expansion phase
        for $a \in A(s)$ do
            $(N(s,a), Q(s,a)) \leftarrow (N_0(s,a), Q_0(s,a))$
        $T = T \cup \{s\}$
        return Rollout$(s, d, \pi_0)$            ▷ Rollout phase
    $a \leftarrow$ arg max$_a$ $Q(s,a) + c\sqrt{\frac{logN(s)}{N(s,a)}}$     ▷ Search phase
    $(s', r) \sim G(s,a)$
    $q \leftarrow r + \lambda$ Simulate$(s, d-1, \pi_0)$
    $N(s,a) \leftarrow N(s,a) + 1$
    $Q(s,a) \leftarrow Q(s,a) + \frac{q - Q(s,a)}{N(s,a)}$
    return $q$
```

```
function MctsRollout$(s, d, \pi_0)$
    if $d = 0$ then
        return 0
    $a \sim \pi_0(s)$
    $(s', r) \sim G(s,a)$
    return $r + \lambda$ Rollout$(s', d-1, \pi_0)$
```

MCTS simus are run for e.g. a fixed # iters (# MCTS nodes = # MCTS iters !). We then execute the action that maximizes $Q(s,a)$. Then we can rerun MCTS to select the next action. It is common to carry over the values of $N(s,a)$ and $Q(s,a)$ computed in the previous step.

Ch4 Synthesis: DP can be used for problems with small state and action spaces and has the advantage of being guaranteed to converge to the optimal policy. ADP is often used for problems with large or continuous state and action spaces. In these problems, DP may be intractable so an approximation to the optimal policy is often good enough. Online methods are used for problems with very large or continous state and action spaces where finding a good approximation to the optimal policy over the entire state space is intractable. In an online method, the optimal policy is approximated for only the current state. This approximation greatly reduces the computational complexity but also requires computation every time a new state is reached.

DMU Ch.5 Model Uncertainty

Reinforcement learning is typically used in problems in which there is model uncertainty, that is, an unknown transition and reward model.

Challenges:

1. Balance exploitation with exploration
2. Rewards may be rx long after important decisions have been made
3. How to generalize from limited experience

Exploration and Exploitation

Exploration versus exploitation describes the decision between exploiting information you know and exploring to find new information. If you exploit the knowledge you have, but never explore any new options, you might be missing out on even better rewards. If you spend all your time exploring, and never exploit what you know, you might not get any reward.

Multi-armed bandits
Slot machine with n arms. Arm i pays off 1 with proba θ_i and 0 with proba $1-\theta_i$, limited to h pulls
MDP: 1 state, n acts, unknown $R(s,a)$, h finite horizon
Prior for $\theta_i : Beta(1,1)$
Experience: ω_i wins and ℓ_i losses for arm i
$\Longrightarrow$Posterior for $\theta_i : B(\omega_i+1,\ell_i+1)$
Better than MaxLLH estimate bad with few samples
Proba of winning: $\rho_i = P(win_i \mid \omega_i, \ell_i)$
$\rho_i = \int_0^1 \theta \times Beta(\theta \mid \omega_i+1, \ell_i+1)d\theta = \frac{\omega_i+1}{\omega_i+\ell_i+2}$
ρ_i is our estimate of θ_i(expectation of a random variable)
Choose arm i with bigger ρ_i

Ad Hoc Exploration Strategies

1. ε-greedy:
   - With proba ε: random arm
   - Otherwise: $\underset{i}{argmax}\,\rho_i$
2. Explore for k steps and then exploit
3. Directed exploration: $P(arm_i) \propto e^{\lambda\rho_i}$ and softmax
   - $\lambda \to 0$ complete exploration
   - $\lambda \to \infty$ greedy selection
4. Upper Confidence Bound: prio to larger uncertainty
   - e.g. α^{th} percentile of a pdf (upper bound of a confidence interval)
   - $\alpha \nearrow$ exploration $\nearrow$

Optimal Exploration Strategies
$s = (\omega_{1:n}, \ell_{1:n})$ is our MDP state or belief state
Expected payoff after pulling arm i : $Q^*(s,i)$
$U^*(s) = \underset{i}{max}\, Q^*(s,i)$ and π^*(s)=$\underset{i}{argmax}\, Q^*$(s,i)
Recursive Eq. and use DP
Use Bellman Equation recursivity
$Q^*(\omega_1,\ell_1,\ldots,\omega_n,\ell_n,i) =$
$\frac{\omega_i+1}{\omega_i+\ell_i+2}\left(1+U^*(\ldots,\omega_i+1,\ell_i,\ldots)\right)$
$+\left(1-\frac{\omega_i+1}{\omega_i+\ell_i+2}\right)\left(0+U^*(\ldots,\omega_i,\ell_i+1,\ldots)\right)$
With no pulls left: $U^*(\omega_1,\ell_1,\ldots,\omega_n,\ell_n) = 0$
$\Longrightarrow$resolve backwards and recursively
From $\sum_i(\omega_i+\ell_i) = h$ to $\sum_i(\omega_i+\ell_i) = h-1$
With 1 pull left i.e. when $\sum_i(\omega_i+\ell_i) = h-1$
$Q^*(\omega_1,\ell_1,\ldots,\omega_n,\ell_n,i) = \frac{\omega_i+1}{\omega_i+\ell_i+2}$
$U^*(s) = \underset{i}{max}\,\frac{\omega_i+1}{\omega_i+\ell_i+2}$
And so on
For ∞ horizon: cf *Gittins allocation index* method
Issue: computation and memory via the number of belief states $(2n)^h$ *grows exponentially in h*

Maximum Likelihood Model-Based Methods

Solving problems with multiple states is more challenging than bandit problems. Bcz we need to plan to visit states to determine their value

One RL approach: estimates T and R models directly from experience

Iteratively:

- Update counts: $N(s,a,s'), N(s,a) = \sum_{s'} N(s,a,s'), \rho(s,a) = \sum r, R(s,a) = \frac{\rho(s,a)}{N(s,a)}$
- Estimate T,R based on counts
- Update Q based on estimated T,R

Issue: compute expensive (so we may prefer to update only e.g. every 5 steps of experiments)

```
1:  function MaximumLlhModelBasedRL
2:      $t \leftarrow 0$
3:      $s_0 \leftarrow$ initial state
4:      Initialize $N, \rho$ and $Q$
5:      loop
6:          Choose action $a_t$ based on some explor strat
7:          Observe new state $s_{t+1}$ and reward $r_t$
8:          $N(s_t,a_t,s_{t+1}) \leftarrow N(s_t,a_t,s_{t+1})+1$
9:          $\rho(s_t,a_t) \leftarrow \rho(s_t,a_t)+r_t$
10:         Update $Q$ based on revised estimate of $T,R$
11:         $t \leftarrow t+1$
```

Online: in the sense that we are getting new data as we interact with the world. Once T,R are estimated we could apply sparse sampling, Branch & Bound or MCTS ...

Offline: as we are computing the policy for the **full** state space

DP is an offline method

Randomized Updates: Dyna algo
Instead of using DP to update Q in line 10 above just do
For the current state:
$Q(s,a) \leftarrow R(s,a) + \gamma\sum_{s'} T(s' \mid s,a)\underset{a'}{max}Q(s',a')$
With R,T estimates
Then perform some number of additional updates of Q for random states and actions depending on how much time available between decisions
Then use Q to choose an action (softmax or whatever other exploration strategy)

Prioritized Sweeping
Use a priority queue to help identify which states require updating Q the most
Efficient updates vs Model-Free and Eligibility traces !
The process of updating the highest prio state in the queue coninues for some fixed # iterations or until the queue becomes empty

```
function PrioritizedSweeping(s)
    Increase prio of s to $\infty$
    while prio queue is not empty do
        $s \leftarrow$ highest prio state
        Update($s$)
function Update(s)
    $u \leftarrow U(s)$
    $U(s) \leftarrow max_a [R(s,a) + \gamma\sum_{s'} T(s' \mid s,a)U(s')]$
    for $(s',a') \in pred(s)$ do
        $p \leftarrow T(s \mid s',a') \times |U(s)-u|$
        Increase priority of $s'$ to $p$
```

Bayesian Model-Based Methods

Optimally balance Exploration with Exploitation without relying on heuristics

$T(s' \mid s,a)$ model parameters: θ with $|S|^2 \times |\mathscr{A}|$ components representing every possible transition probability

The component of θ that governs the transition probability $T(s' \mid s,a)$ is denoted $\theta_{(s,a,s')}$

$\theta_{(s,a)}$ is a vector of dim $|\mathscr{S}| \times 1$

Discrete state space here =>Dirichlet distributions

Beliefs over Model Parameters
Prior: $b_0(\theta) = \prod_s\prod_a Dir(\theta_{(s,a)} \mid \overrightarrow{\alpha_{(s,a)}})$
Posterior: $b_t(\theta) = \prod_s\prod_a Dir(\theta_{(s,a)} \mid \overrightarrow{\alpha_{(s,a)}} + \overrightarrow{m_{(s,a)}})$
With $\overrightarrow{m_{(s,a)}}$: a vector of transition counts

BAMDP: Bayes-Adaptive MDP
Augmented State: (s,b)
$\mathscr{S}$ discrete, $\mathscr{B}$ **high dim continuous**
T model represented via θ : what we are uncertain about
$T(s',b' \mid s,b,a)$: you start in state s with belief b and execute action a
We transit to a new belief and to a new state
$T(s',b' \mid s,b,a) = \delta_{\tau(s,b,a,s')}(b')P(s' \mid s,b,a)$

- b' is computed fully deterministically. Dir Bayes update ! Just update the pseudo-counts ! So δ is 1 or 0
- With belief $\theta_{(s,a)} \sim Dir(\alpha_1,\alpha_2,\alpha_3) \Longrightarrow P(s_2 \mid s,a) = \frac{\alpha_2}{\alpha_1+\alpha_2+\alpha_3}$

$P(s' \mid s,b,a) = \int_\theta b(\theta)P(s' \mid s,\theta,a)d\theta = \int_\theta b(\theta)\theta_{(s,a,s')}d\theta = \frac{\alpha_{(s,a,s')}}{\sum_{s''}\theta_{(s,a,s'')}}$
It is an expectation over a Dirichlet distribution
Mean of a Dirichlet distribution: $\mathbb{E}[X_i] = \frac{\alpha_i}{\sum_k \alpha_k}$
With lots of experiences: we are converging to MaxLlh estimate.
With few experiences: we are smoothing things.
Exemple of evolutions:

- Prior: unniform $Dir(1,\ldots,1)$
- Then sthg like a gaussian
- Then at ∞ a spike , the MaxLLH

Dirichlet Distribution
$P(\theta) = \frac{\Gamma(\alpha_0)}{\prod_i\Gamma(\alpha_i)}\prod_i \theta_i^{\alpha_i-1}$
$\Gamma(x+1) = x\Gamma(x)$ and $\theta = (\theta_1,\ldots,\theta_n)$
$\mathbb{E}[\theta_i] = \int \theta_i P(\theta)d\theta = \frac{\alpha_i}{\sum_k \alpha_k}$

BAMDP solution: solving the optimal function over the belief space
$U^*(s,b) = \underset{a}{max}[R(s,a) + \gamma \sum_{s'} P(s' \mid s,b,a)U^*(s',\tau(s,b,a,s'))$
We can not use VI or PI bcz b is continuous
$\Longrightarrow$ approx methods or online methods

BAMDP solution: Thompson sampling
Idea:

- Sample a transition model from our Dir distribution
- Solve it with DP
- Take the greedy action
- Gather new data and update the pseudo-counts

And so on ...
Tendency to over explore but it is REALLY GOOD: no need to rely on ad-hoc exploration strategies
Resolving MDP at every step can be expensive

Model-Free Methods

We do not try to estimate T,R but we try to learn Q or U directly.

For really large pbs, you do not want to store huge T matrix (wildfire: $2^{100x100}$states) $\Rightarrow$pretend you do not know it and just run a bunch of simus

Downside: you need a WHOLE BUNCH OF SAMPLES

Incremental Estimation
X a random variable, try to estimate the mean: $\mu = \mathbb{E}[X]$
With samples $x_1,\ldots,x_n$
$\hat{x}_n = \frac{1}{n}\sum_{i=1}^n x_i$
$\Longleftrightarrow \hat{x}_n = \hat{x}_{n-1} + \frac{1}{n}(x_n - \hat{x}_{n-1})$
$\hat{x} \leftarrow \hat{x} + \alpha(n)(x-\hat{x})$
Key equation in Temporal Difference Learning
$\hat{x} \leftarrow \hat{x} + \alpha(x-\hat{x})$
With x new meas and $\hat{x}$ current estimate
Temporal difference error: $x-\hat{x}$ is the difference between a sample and our previous estimate
Constant LR $\Rightarrow$ decays the influence of past samples exponentially
And as we collect experience, more recent examples based on better Q , are better
A larger LR means new samples have a greater effect on the current estimate

$Q(s,a)$ represents the utility of taking action a in state s. It assumes after taking this action, you act optimally afterwards. Q represents the utility of a specific state-action combination. On the other hand, $U(s)$ is the utility of being in a specific state regardless of action. Each U is the best Q-value for a given state, considering all possible actions.

$$Q(s,a) = R(s,a) + \sum_{s'} T(s' \mid s,a)\,\underset{a'}{max}\,Q(s',a')$$

Q-learning
$Q(s,a) = R(s,a) + \gamma\sum_{s'} T(s' \mid s,a)U(s')$
With $U(s') = \underset{a'}{max}\,Q(s',a')$
How to update Q directly after we observe r and s' ?
$Q(s,a) \leftarrow Q(s,a) + \alpha\left(\mathbf{r} + \gamma\underset{\mathbf{a'}}{\mathbf{max}}\,\mathbf{Q(s',a')} - Q(s,a)\right)$
This update + good exploration strategy $\Rightarrow Q(s,a) \to Q^*(s,a)$

We can init Q to values other than 0 to encode any prior knowledge we may have

```
function QLearning
    $t \leftarrow 0$
    $s_0 \leftarrow initialstate$
    Initialize $Q$
    loop
        Choose $a_t$ based on $Q$ and some Exploration
        Observe new state $s_{t+1}$ and reward $r_t$
        $Q(s_t,a_t) \leftarrow Q(s_t,a_t) + \alpha(r_t + \gamma max_a Q(s_{t+1},a) - Q(s_t,a_t))$
        $t \leftarrow t+1$
```

Q-learning and Sarsa both work on the principle of incremental estimation but Sarsa has the advantage over Q-learning of not needing to iterate over all possible actions

SARSA
Uses $(s_t,a_t,r_t,s_{t+1},a_{t+1})$
$Q(s_t,a_t) \leftarrow Q(s_t,a_t) + \alpha(\mathbf{r_t} + \gamma\mathbf{Q(s_{t+1},a_{t+1})} - Q(s_t,a_t))$
Performs in general better than Q-learning (empirically)

Q-learning and SARSA are SUPER SLOW to CV

You need to re-experience a TON of TIMES in order to backpropagate your (spare or delayed) rewards.

Sarsa(λ) is a modified version of Sarsa that assigns credit to states and actions that were encountered on the way to a reward state. This assignment of reward to intermediate states and actions is referred to as using eligibility traces. Eligibility traces can speed up learning for problems that have sparse rewards such games that result in a win or loss at the end.

Eligibility Traces
Idea: backpropagate and keep track of what you have experienced so far and automatically decay it by a factor of λ

$$\begin{tikzcd}
\gamma^4\lambda^4\Delta & \gamma^3\lambda^3\Delta & \gamma^2\lambda^2\Delta & \gamma\lambda\Delta & \Delta=\alpha\delta \\
A \arrow[r] & B \arrow[r] & C \arrow[r] & D \arrow[r] & E
\end{tikzcd}$$

With Q-learning and SARSA, states A,B,C are not updated. But we want to assign credit to past states and actions leading to a good reward: nevertheless closer states shall get bigger credit.
We keep track of an exponentially decaying visit count $N(s,a)$ for all state-action pairs.
Although the impact is especially pronounced in env with sparse reward, the algo can speed learning in general: $Q(\lambda), SARSA(\lambda)$

```
function SarsaLambdaLearning($\lambda$)
    Initialize Q and N
    $t \leftarrow 0$
    $s_0, a_0 \leftarrow$ initial state and action
    loop
        Observe reward $r_t$ and new state $s_{t+1}$
        Choose action $a_{t+1}$ based on Exploration
        $N(s_t,a_t) \leftarrow N(s_t,a_t)+1$
        $\delta \leftarrow r_t + \gamma Q(s_{t+1},a_{t+1}) - Q(s_t,a_t)$
        for $s \in S$ do
            for $a \in A$ do
                $Q(s,a) \leftarrow Q(s,a) + \alpha\delta N(s,a)$
                $N(s,a) \leftarrow \gamma\lambda N(s,a)$
        $t \leftarrow t+1$
```

Model-based vs Model-free RL

Model-based approaches can usually find a better policy than model-free approaches at the cost of being more computationally expensive.

Generalization

Generalization is needed when the problem is large enough that we cannot experience all state-action pairs. We therefore need to generalize from limited experience to states that we have not visited.

- Interpolation (multilinear or simplex-based) can be used as a local approximation method.
- Perceptrons or neural networks can be used as global approximators for the Q-values.

Same formula for local and global approximation but different interpretations

We want to approximate from $Q(s,a)$ or $U(s)$ but not from $\pi(s)$

Local Approximation
Idea, modify e.g. Q-learning with linear approximation: $\underset{a}{max}\,Q(s_{t+1},a) \approx \underset{a}{max}\,\theta^T\beta(s_{t+1},a)$ or $U(s) \approx \theta^T\beta(s)$

- Reduced number of estimates of $Q(s,a)$ stored in $\theta = [\theta_{(s,a)}]$ vector
- Weighted s.t. $\sum_{s'}\beta(s',a) = 1$, typically distance related

Global Approximation
Typical example: use a Neural Network that can represent non-linear functions
CV when using this form of function approximation is not guaranteed but it can be good in practice
Safety issue with NN trained through sampling: if you do not train against lot of failure cases, the learned policy may not be very robust

- Augment RL to capture safety bounds
- Train it as usual but then **proof** the result can never brings you into an undesirable state

Q-learning with global linear approximation
$Q(s,a) = \theta^T\beta(s,a)$ e.g. $= \theta_1 s + \theta_2 a + \theta_3 sa + \theta_4 s^2$
Require some prior knowledge to generate appropriate basis functions. For $U(s)$ you may think Taylor series !
$\theta \leftarrow \theta + \alpha(r_t + \gamma\,\underset{a}{max}\,\theta^T\beta(s_{t+1},a) - \theta^T\beta(s_t,a_t))\beta(s_t,a_t)$
vector $\leftarrow$ *vector* $+ \alpha$ *salar* $\times$ *vector*

Remember: #policies IS NOT policy matrix size ! For wildfire: #policies= $4^{2^{100^2}} \times 100^2$

Direct Policy Search: when policy space is limited