RL

notes for final

Intro

- Utility / index functions are transitive and ordinal (exact values don't matter)
- Concept of diminishing marginal benefit of money (Bernoulli's proposal: take logs)

Bandit problems, Markov chains and MDPs

- n-armed bandit problem
  - after each play a_t, you get reward r_t, where $E\{r_t | a_t\} = Q^*(a_t)$ ← action-value
  - to be greedy: $a_t^* = \arg\max_a Q_t(a)$ where $a_t = a_t^*$ is exploitation and $a_t \neq a_t^*$ is exploration
  - to keep a running estimation of the action-value: $Q_{k+1} = Q_k + \frac{1}{k+1}[r_{k+1} - Q_k]$
  - for non-stationary case, use learning rate α instead of $\frac{1}{k+1}$: $Q_{k+1} = Q_k + \alpha[r_{k+1} - Q_k]$
  - can help to make $Q_0(a)$ (the initialized action value) optimistically high to encourage exploration; this bias disappears over time
  - softmax: $$\frac{\exp(Q_t(a)/\tau)}{\sum_{b=1}^{n} \exp(Q_t(b)/\tau)}$$ where τ is "temperature"
    $\tau \to \infty$: actions equiprobable
    $\tau \to 0$: greedy
  - regret = [reward sum of optimal strategy] − [sum of actual collected rewards]
    $$= T\mu^* - \sum_{t=1}^{T} \underbrace{E[r_{i_t}(t)]}_{\hat{r}_t} \quad \text{where } \mu^* = \max_k \mu_k,\ T = \#\text{ rounds}$$

    If average regret per round goes to zero w/ probability 1, strategy has "no-regret" property, meaning it's guaranteed to converge to the optimal strategy. ε-greedy is sub-optimal (has regret)
  - interval estimation: greedily choose arm with highest upper bound on confidence interval. This is called upper confidence bound (UCB) strategy. There are some fancy ways of determining UCB.
- Markov Decision Process (MDP)
  - R_t is the "return" and is defined as some specific function of the reward sequence
    - simplest case is the sum of the rewards: $R_t = r_{t+1} + r_{t+2} + \cdots + r_T$ (episodic tasks)
    - if infinite (continuing tasks): $R_t = r_{t+1} + \gamma r_{t+2} + \gamma^2 r_{t+3} + \cdots = \sum_{k=0}^{\infty} \gamma^k r_{t+k+1}$
    - unified notation: $R_t = \sum_{k=0}^{T} \gamma^k r_{t+k+1}$ where $T = \infty$ or $\gamma = 1$ (but not both)

    - Markov property: when the state signal retains all relevant information
      - theory built w/ this assumption, although with care it can be applied to non-Markov cases
  - finite MDP: state and action spaces are finite
  - $P_{ss'}^{a} = Pr\{s_{t+1} = s' | s_t = s, a_t = a\}$ (transition probabilities)

    $R_{ss'}^{a} = E\{r_{t+1} | s_t = s, a_t = a, s_{t+1} = s'\}$

    converges to average of rewards for state s following policy π
  - state-value: $V^{\pi}(s) = E_{\pi}\{R_t | s_t = s\}$
    action-value: $Q^{\pi}(s,a) = E_{\pi}\{R_t | s_t = s, a_t = a\}$ (... for policy π)
  - Bellman equation: $V^{\pi}(s) = \sum_a \pi(s,a) \sum_{s'} P_{ss'}^{a}[R_{ss'}^{a} + \gamma V^{\pi}(s')]$

- Bellman optimality equations:

$$V^*(s) = \max_\pi V^\pi(s)$$

$$Q^*(s,a) = \max_\pi Q^\pi(s,a)$$

$$V^*(s) = \max_{a \in A(s)} Q^{\pi^*}(s,a)$$

$$= \max_a E_{\pi^*}\{R_t \mid s_t = s, a_t = a\}$$

$$\dots$$

replaces sum over actions and $\pi(s,a)$

now $*$ instead of π

$$= \max_a \sum_{s'} P^a_{ss'} \left[R^a_{ss'} + \gamma V^*(s')\right]$$

$$Q^*(s,a) = E\{r_{t+1} + \gamma \max_{a'} Q^*(s_{t+1}, a') \mid s_t = s, a_t = a\}$$

$$= \sum_{s'} P^a_{ss'} \left[R^a_{ss'} + \gamma \underbrace{\max_{a'} Q^*(s', a')}_{\to V^*(s')}\right]$$

- any policy that is greedy with respect to the optimal value function V^* is an optimal policy; V^* already takes into account the reward consequences of all possible future behavior
- Having Q^* makes choosing optimal actions even easier. At the cost of representing a function of state-action pairs, instead of just states, the optimal action-value function allows ~~for~~ optimal actions to be selected w/o having to know anything about successor states and their values – that is, w/o having to know about the system's dynamics

- Markov chains
  - example transition matrix P:

end states — rows sum to 1

$$P^{(1)} = P = \text{start states}\begin{bmatrix} 0.08 & 0.184 & 0.368 & 0.368 \\ 0.632 & 0.368 & 0 & 0 \\ 0.264 & 0.368 & 0.368 & 0 \\ 0.08 & 0.184 & 0.368 & 0.368 \end{bmatrix}$$

$$P^{(8)} = \begin{pmatrix} 0.286 & 0.285 & 0.264 & 0.166 \\ 0.286 & 0.285 & 0.264 & 0.166 \\ 0.286 & 0.285 & 0.264 & 0.166 \\ 0.286 & 0.285 & 0.264 & 0.166 \end{pmatrix}$$

$$P^T \vec{\pi} = \vec{\pi}$$

$$(P^T - I)\vec{\pi} = 0 \quad \text{s.t. } \sum \pi_i = 1$$

- Markov decision model
  - machine maintenance example:

$$P = \begin{array}{c|c} & \begin{array}{cccc} 0 & 1 & 2 & 3 \end{array} \\ \hline \begin{array}{c} 0 \\ 1 \\ 2 \\ 3 \end{array} & \begin{bmatrix} 0 & 7/8 & 1/16 & 1/16 \\ 0 & 3/4 & 1/8 & 1/8 \\ 0 & 0 & 1/2 & 1/2 \\ 1 & 0 & 0 & 0 \end{bmatrix} \end{array}$$

start states

Solve by:

$$0 \cdot \pi_0 + 0 \cdot \pi_1 + 0 \cdot \pi_2 + 1 \cdot \pi_3 = \pi_0$$

$$\tfrac{7}{8} \cdot \pi_0 + \tfrac{3}{4} \cdot \pi_1 + 0\,\pi_2 + 0\,\pi_3 = \pi_1$$

$$\dots = \pi_2$$

$$\dots = \pi_3$$

as system of linear equations

Steady state probs end up being:

$$\pi_0 = \frac{2}{13}, \quad \pi_1 = \frac{7}{13}, \quad \pi_2 = \frac{2}{13}, \quad \pi_3 = \frac{2}{13}$$

long-run cost is: $c_0 \pi_0 + c_1 \pi_1 + c_2 \pi_2 + c_3 \pi_3$

RL
notes for final

Dynamic programming, policy + value iteration, MC methods }

- relation between methods (+ TD):
  - dynamic programming: developed mathematically, but requires complete model of environment
  - MC: no model needed, but can't do incremental
  - TD: no model needed and incremental, but complex to analyze

- DP algorithms obtained by turning Bellman equations into assignments
  - policy evaluation: computing state-value function V^π ("prediction problem")
    - can solve as system of linear equations, but iteration is preferred
    - update rule:

$$V_{k+1}(s) = E_\pi \{ r_{t+1} + \gamma V_k(s_{t+1}) \mid s_t = s \}$$

next iteration

$$= \sum_a \pi(s,a) \sum_{s'} P^a_{ss'} \left[R^a_{ss'} + \gamma V_k(s') \right]$$

current iteration

    - converges to V^π as $k \to \infty$
    - "full" backup: based on all possible next states instead of sample next state
  - policy improvement: making π greedy with respect to value function
    - policy improvement theorem says if it's better to select a once in state s and then follow π, it's better still to to select a every time s is encountered
      so: $Q^\pi(s, \pi'(s)) \geq V^\pi(s)$ (for all $s \in S$) then policy $\pi' \geq \pi$. Also: $V^{\pi'}(s) \geq V^\pi(s)$

  - policy iteration: $\pi_0 \xrightarrow{E} V^{\pi_0} \xrightarrow{I} \pi_1 \xrightarrow{E} V^{\pi_1} \xrightarrow{I} \pi_2 \xrightarrow{E} \cdots \xrightarrow{I} \pi^* \xrightarrow{E} V^*$

    E = policy eval.
    I = policy improvement

    - converges in surprisingly few iterations

  - value iteration: policy iteration, but where policy evaluation is stopped after just one sweep
    - combined in simple backup operation:

$$V_{k+1}(s) = \max_a \sum_{s'} P^a_{ss'} \left[R^a_{ss'} + \gamma V_k(s') \right]$$

  - generalized policy iteration (GPI): iterating policy evaluation and policy improvement processes, independent of the details of the two processes

- Monte Carlo methods
  - can learn from on-line experience; full model not needed (only needs sample transitions)
    - assumes experience divided into episodes
  - most important ideas from DP carry over
  - both first-visit and every-visit MC converge to $V^\pi(s)$ as the number of visits approaches ∞
  - generating samples in some situations (e.g., blackjack example) is much easier than calculating all of the expected reward and transition probabilities
  - without a model (ie, missing $R^a_{ss'}$ and $P^a_{ss'}$), must estimate Q^* instead of V^*
    - policy improvement: $\pi(s) = \arg\max_a Q(s,a)$ we have these
  - Monte Carlo ES (exploring starts): algorithm for policy eval. where eval. and improvement are alternated on an episode-by-episode basis
  - can be used w/ simulation, and can focus on small subset of states

- *temporal difference (TD) learning*
  - comparison of updates:
    - DP: $V(s_t) \leftarrow E\{r_{t+1} + \gamma V(s_t)\}$
    - MC: $V(s_t) \leftarrow V(s_t) + \alpha\,[\,R_t - V(s_t)\,]$ — target: actual return from s_t to end of episode
    - TD: $V(s_t) \leftarrow V(s_t) + \alpha\,[\,\underbrace{r_{t+1} + \gamma V(s_{t+1}) - V(s_t)}_{\text{TD error } \delta_t}\,]$ (sample backup, like MC)

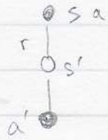
    target: estimate of the return

    both MC and TD have the same form
  - advantages:
    - no model needed
    - naturally implemented on-line
    - converge faster than constant-α MC methods
  - they bootstrap: learn a guess from a guess (like DP but not like MC)
  - they sample (like MC but not like DP)
  - TD vs. MC:

A, 0, B, 0	B, 1	MC thinks $V(A)=0$ b/c the only complete episode starting w/ A is A, 0, B, 0
B, 1	B, 1	
B, 1	B, 1	TD thinks $V(A)>0$ b/c A leads to B and $V(B)>0$
B, 1	B, 0	

  - Sarsa: instead of considering transitions from state to state and learning values of states, consider state-action pair to state-action pair and learn values of state-action pairs:

    $$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha\,[\,r_{t+1} + \gamma\, Q(s_{t+1}, a_{t+1}) - Q(s_t, a_t)\,]$$

    - this is on-policy TD control
  - Q-learning: off-policy TD control; learned Q directly approximates Q^* independent of policy

    $$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha\,[\,r_{t+1} + \gamma \max_a Q(s_{t+1}, a) - Q(s_t, a_t)\,]$$

  - actor-critic methods: TD method that separates the policy and value function. Actor selects actions and maintains policy by (often greedily and deterministically) maximizing reward r; critic estimates value fun and finds TD error δ_t (often not deterministic)
    - δ drives learning of both actor and critic

      $$\delta = r_{t+1} + \gamma V(s_{t+1}) - V(s_t)$$ (same term appears in TD update)

    - this updates actor's preference $p(s,a)$ for choosing action a in state s

      $$p(s_t, a_t) \leftarrow p(s_t, a_t) + \beta\,\delta_t$$

      (β: learning rate) where $\pi_t(s,a) = \Pr\{a_t = a, s_t = s\} = \dfrac{\exp(p(s,a))}{\sum_b \exp(p(s,b))}$ for example

update value function along a path that may not be the one you sampled or experienced

On-policy and off-policy learning | estimate true value of policy while using it for control (can be done w/ max operator)

- To explore w/ on policy, use ε-greedy; w/ off-policy, use behavior policy that's good at exploring and infer optimal policy from that
- ε-soft seems to mean $\pi(s,a) > 0,\ \forall s, \forall a$
- learning one policy while following another (off-policy)
  - behavior: $p'_i(s_t) = \prod_{k=t}^{T_i(s)-1} \pi'(s_k, a_k)\, P^{a_k}_{s_k s_{k+1}}$
  - estimation: $p_i(s_t) = \prod_{k=t}^{T_i(s)-1} \pi(s_k, a_k)\, P^{a_k}_{s_k s_{k+1}}$

    $p_i(s_t)$ is probability of getting the same sequence of states and actions from π (estimation); $p'_i(s_t)$ is same but for behavior

    w/ off-policy MC: $Q(s,a) = \dfrac{\sum \frac{p_i}{p'_i} R'}{\sum \frac{p_i}{p'_i}}$
  - calculate value as $V^\pi(s) = \dfrac{\sum_{i=1}^{n_s} \frac{p_i(s)}{p'_i(s)} R_i(s)}{\sum_{i=1}^{n_s} \frac{p_i(s)}{p'_i(s)}}$

    $T_i(s)$ = time of termination of the i^{th} episode involving state s

Generalization and function approximation

- generalization helpful for:
  - large state/action spaces
  - continuous valued states and actions
  - too many new states during learning
  - memory, time, data...
- important that supervised learning model (neural network, decision trees, etc.) can occur online and can handle nonstationary target functions
- $V_t(s)$ is smooth differentiable function of $\vec{\theta}_t$ where $\vec{\theta}_t = (\theta_t(1), \theta_t(2), \ldots, \theta_t(n))^T$ (weights)
- generally no $\vec{\theta}$ that gets all states, or even all examples, exactly correct
- goal is to minimize MSE:

$$MSE(\vec{\theta}_t) = \sum_{s \in S} P(s)\left[V^\pi(s) - V_t(s)\right]^2$$ where P is a distribution weighting the errors of different states

- if minimize error on observed examples, don't have to worry about P
- gradient descent:

$$\vec{\theta}_{t+1} = \vec{\theta}_t - \frac{1}{2}\alpha \nabla_{\vec{\theta}_t} \overbrace{\left(\left[V^\pi(s_t) - V_t(s_t)\right]^2\right)}^{MSE(\vec{\theta}_t)}$$

$\nabla_{\vec{\theta}_t} f(\vec{\theta}_t)$ is vector of partial derivs: $\left(\frac{\partial f(\vec{\theta}_t)}{\partial \theta_t(1)}, \frac{\partial f(\vec{\theta}_t)}{\partial \theta_t(2)}, \ldots, \frac{\partial f(\vec{\theta}_t)}{\partial \theta_t(n)}\right)^T$

$$= \vec{\theta}_t + \alpha\left[\underbrace{V^\pi(s_t)}_{v_t \text{ more generally, since } V^\pi(s_t) \text{ isn't known}} - V_t(s_t)\right]\nabla_{\vec{\theta}_t} V_t(s_t)$$

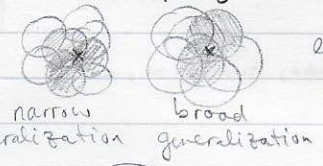
$(E\{v_t\} = V^\pi(s_t))$

- linear methods: like $w^T x$

gradient to be used in gradient descent: $\nabla_{\vec{\theta}_t} V_t(s) = \vec{\phi}_s$

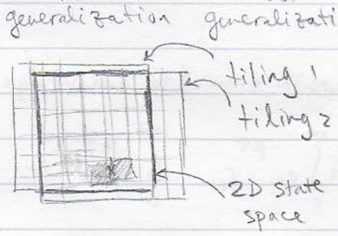
$$V_t(s) = \vec{\theta}_t^{\,T} \vec{\phi}_s = \sum_{i=1}^{n} \theta_t(i)\phi_s(i)$$

  - since only one optimum, guaranteed to converge
  - in practice, can be very efficient, but depends a lot on how states are represented in terms of features
- number of weights usually much less than number of states, so changing any component of weight vector will affect many states ⇒ more powerful but needs to be managed w/ care
- Why minimize MSE? We want better policy, but unclear how else to get at that other than value prediction
- coding:
  - coarse coding:



each circle is a binary variable

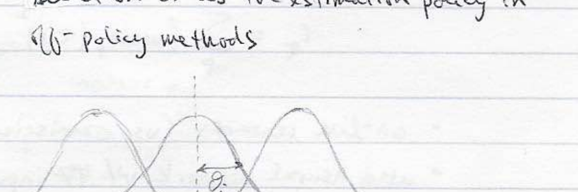
  - tile coding:



- policy improvement:
  - action selection: $a_t^* = \arg\max_a Q_t(s_t, a)$
  - can use this as part of ϵ-greedy action selection or as the estimation policy in off-policy methods

  - radial basis functions (RBFs):

e.g., Gaussians $$(\phi_i)_s = \exp\left(-\frac{\|s - c_i\|^2}{2\sigma_i^2}\right)$$



- beating "curse of dimensionality":
  - Kanerva coding: select set of binary prototypes, use Hamming distance as distance measure
  - "lazy learning" schemes: remember all data; to get new value, find nearest neighbor and interpolate

Abstraction and hierarchy

- Semi-MDP:
  - defined in terms of $P(s', \tau \mid s, a)$, the transition probability, where τ is waiting time
  - now represent reward as $R(s,a)$ or just r, which is the reward expected to accumulate during waiting time, τ, in a particular state and action (i.e., it's generalited for time)
  - Bellman equation can be written as:

$$V^*(s) = \max_{a \in A_s} \left[r + \sum_{s', \tau} \gamma^{\tau} P(s', \tau \mid s, a) V^*(s') \right]$$

now summing over time, so put γ inside summation

probability now function of time

compare to:

$$V^*(s) = \max_{a \in A_s} \left[r_s^a + \gamma \sum_{s'} P_{ss'}^a V^*(s') \right]$$

  - Bellman equation for state-action value function:

$$Q^*(s,a) = r + \sum_{s', \tau} \gamma^{\tau} P(s', \tau \mid s, a) \max_{a' \in A_s} Q^*(s', a')$$

  - Q-learning algo goes from...

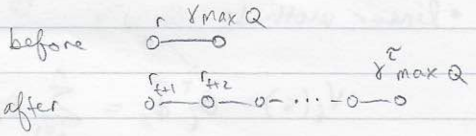
$$Q_{k+1}(s,a) = (1 - \alpha_k) Q_k(s,a) + \alpha_k \left[\underbrace{r + \gamma \max_{a' \in A_s} Q_k(s', a')} \right]$$

to...

$$\cdots \left[\underbrace{r_{t+1} + \gamma r_{t+2} + \cdots + \gamma^{\tau - 1} r_{t+\tau}}_{\text{reward expanded}} + \gamma^{\tau} \max \cdots \right]$$

τ added to discount

- basically, this just assumes a whole bunch of steps with potential rewards have occured... with the bootstrapped value at the end



  - continuous time case where decisions made in discrete jumps

$$R_t = \sum_{k=0}^{\infty} \gamma^k r_{t+k+1} \quad \text{or} \quad R_t = \int_0^{\infty} e^{-\beta \tau} r_{t+\tau} \, d\tau$$

reward at discrete time step in discrete case

reward "rate" in continuous case

  - Semi-Markov Q-learning:

integrate — just a difference; no integration

$$Q(s,a) \leftarrow (1-\alpha) Q(s,a) + \alpha \left[\int_{t_1}^{t_2} e^{-\beta(\tau - t_1)} r_{\tau} \, d\tau + e^{-\beta(t_2 - t_1)} \max_{a'} Q(s', a') \right]$$

  - case study: elevator
    - has parameters like floor time, stop time, turn time, ...
    - model arrivals as Poisson
    - conservatively around 10^{22} states,
    - performance criterion: average squared wait time (to encourage fast and fair service):

$$r_{\tau} = \sum_{p} (\text{wait}_p(\tau))^2, \text{ then define return } R = \int_0^{\infty} e^{-\beta \tau} r_{\tau} \, d\tau$$

p → person

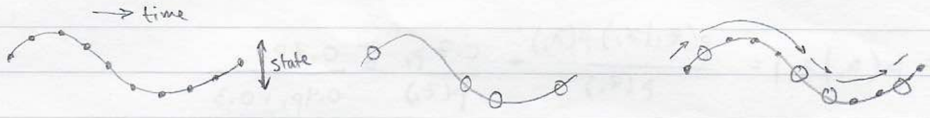
- on-line rewards (estimated given button presses) almost as good as omniscient rewards (from simulation)
- used neural network w/ 47 inputs, 20 sigmoid hidden units, 1 or 2 output units
- performed better than other models

RL

notes for final

Abstraction and hierarchy (cont'd)

- Options
  - a behavior defined in terms of: $o = \{ I_o, \pi_o, \beta_o \}$
    - I_o = set of states in which o can be initiated
    - $\pi_o(s)$ = policy mapping state to action when o is executing
    - $\beta_o(s)$ = probability that o terminates in state
  - options define a semi-MDP



MDP	SMDP	options over MDP
- discrete time - homogeneous discount (γ)	- continuous time - discrete events - interval-dependent discount (γ^τ)	- discrete time - overlaid discrete events - interval-dependent discount

  - all Bellman equations and DP results extend for value functions over options and models of options
  - now we can define policy over options as well: $\mu: S \times O \rightarrow [0,1]$ (?)
    - redefine value functions: $V^\mu(s)$, $Q^\mu(s,o)$, $V_o^*(s)$, $Q_o^*(s,o)$
  - optimal value function:

$$V_O^*(s) = \max_{\mu \in \Pi(O)} V^\mu(s)$$

$$= \max_{o \in O_s} E\left[r + \gamma^k V_O^*(s') \mid \mathcal{E}(o,s) \right]$$

where reward r is expanded SMDP reward $r_{t+1} + \gamma r_{t+2} + \dots + \gamma^{\tau-1} r_{t+\tau}$

  - tend to cause faster convergence than when using primitive actions only
  - learning options
    - hard in contexts such as classical planning
    - subgoals created based on commonalities across multiple paths to a solution ⇒ cast finding these commonalities as multiple-instance learning problem
    - identify target concept on basis of positive and negative "bags" of instances
    - this is searching for bottlenecks in observation space

Partial observability and the POMDP model

- since state isn't observable, integrate over posterior distribution over states:

$$\underset{\substack{\uparrow \\ \text{planning horizon}}}{V_T(b)} = \max_{\substack{u \\ \text{action}}} \Big[r(b,u) + \gamma \int \underbrace{V_{T-1}(b')\, p(b' \mid u,b)\, db'}_{\text{posterior}} \Big] \longrightarrow$$

$b = p(x_1)$, for example, when there are 2 states x_1 and x_2

$r(b,\overset{\text{action}}{u})$ can be, for example,

$p(x_2)$ $p(x_1)$

$= 1 - p(x_1)$

- $r(b,u)$ is obtained by integrating over all states:

$$r(b,u) = E_x[r(x,u)]$$

prob of state x

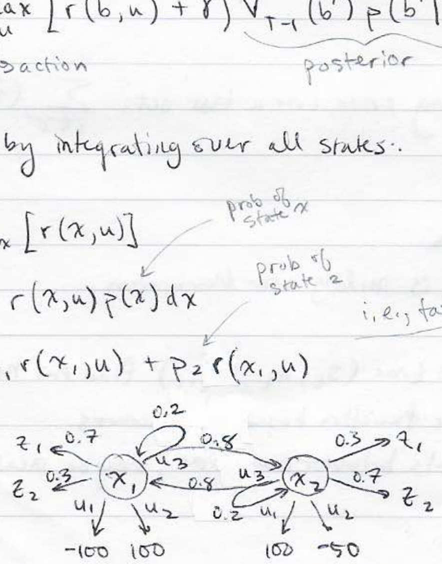
$$= \int r(x,u) p(x)\, dx$$

prob of state 2

i.e., taking integral of this

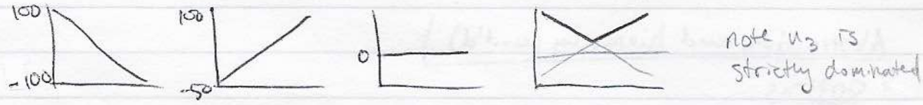
$$= p_1 r(x_1,u) + p_2 r(x_1,u)$$

- illustrative example:



$r(b;u_1)$ $\quad r(b,u_2)$ $\quad r(b,u_3)$ $\quad V_1(b) = \max_u r(b,u)$

payoffs from illustrative example:



note u_3 is strictly dominated

$$\text{so } V_1(b) = \max \begin{cases} -100\, p_1 & +100(1-p_1) \\ 100\, p_1 & -50(1-p_1) \end{cases}$$

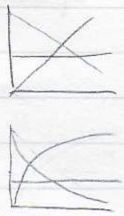
(prune away $r(b,u_3)$)

$$\pi = \begin{cases} u_1 & \text{if } p_1 \le \frac{3}{7} \\ u_2 & \text{if } p_1 > \frac{3}{7} \end{cases}$$

update beliefs with Bayes' rule: (posterior)

$$p_1' = p(x_1|z_1) = \frac{p(z_1|x_1)\,p(x_1)}{p(z_1)} = \frac{0.7\,p_1}{p(z_1)}$$

$$p_2' = \frac{0.3(1-p_1)}{p(z_1)}$$



where $p(z_1) = \sum_i p(z_1|x_i)\,p(x_i) = 0.7\,p_1 + 0.3(1-p_1) = 0.4\,p_1 + 0.3$

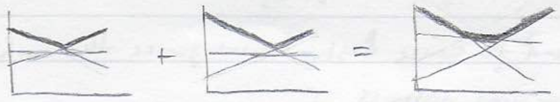
say we observe z_1; now:

replace p_i from before with p_i'

$$V_1(b|z_1) = \max \begin{cases} -100\, p_1' & +100\, p_2' \\ 100\, p_1' & -50\, p_2' \end{cases}$$

and take weighted average to get $\bar{V}_1(b) = E_z[V_1(b|z)]$:

$$\bar{V}_1(b) = \sum_{i=1}^{2} p(z_i)\, V_1(b|z_i)$$



also have to (somehow) take state transitions into account...

- Summary / conclusions / etc.
  - need pruning or else number of linear components of V blows up very quickly (squares for each measurement)
  - resulting value functions are piecewise linear and convex
  - POMDPs have only been applied (s. far) to very small state spaces with a small number of possible observations and actions

Inverse reinforcement learning

- simple approach: behavioral cloning
  - find policy $\hat{\pi}$ that minimizes the following error over a test set: $\sum_{t \in T'} (\hat{\pi}(s_t) - a_t)^2$
  - framed as a supervised learning problem
  - problem:
    - agent inflexible; can't change goals
    - poor performance when environment is mildly non-Markovian
- inverse reinforcement learning (IRL):
  - given observations of agent's behavior over time (s_t, a_t, s_{t+1}) find out how reward is distributed
  - assumes agent derives actions from a value function based on rewards
- for IRL, you're given measurements of an agent's behavior over time, maybe measurements of sensory inputs, and maybe a model of the environment

RL

notes for final

Inverse reinforcement learning (cont'd) |

- motivations:
  - computational models for animal learning
  - agent design (e.g., self-driving cars)
  - multi-agent systems: learning opponents' reward functions
- given state space S, action space A, transition model $P(s'|s,a)$, data $s_0, a_0, s_1, a_1, \ldots$, but not $R^a_{ss'}$
- need to find reward function $\hat{R}$ such that $V^{\pi^*} \geq V^{\pi}$ for all π:

$$E\left\{\sum_{t=0}^{\infty} \gamma^t \hat{R}(s_t) \mid \pi^*\right\} \geq E\left\{\sum_{t=0}^{\infty} \gamma^t \hat{R}(s_t) \mid \pi\right\}, \ \forall \pi$$

$\Rightarrow$ in practice, use sample averages in place of expectation

- write value in terms of reward (using Bellman's equation):

$$V^{\pi}(s) = r + \gamma \sum_{s'} P^a_{ss'} V^{\pi}(s')$$ matrix form

$$V^{\pi} = [V^{\pi}(s_1), V^{\pi}(s_2), \ldots, V^{\pi}(s_N)]^T$$

$$R = [r_{s_1}, r_{s_2}, \ldots, r_{s_N}]^T$$ reward in state s_i

$$V^{\pi} = R + \gamma P^a V^{\pi}$$ rearrange terms

$$R = (I - \gamma P^a) V^{\pi}$$ and again

$$V^{\pi} = (I - \gamma P^a)^{-1} R$$

$$P^a = \begin{bmatrix} P^a_{11} & P^a_{12} & \cdots & P^a_{1N} \\ P^a_{21} & P^a_{22} & \cdots & P^a_{2N} \\ \vdots & \vdots & \ddots & \vdots \\ P^a_{N1} & P^a_{N2} & \cdots & P^a_{NN} \end{bmatrix}$$

- using Bellman's state-action function, we can find that

$$P^a V^{\pi} \succeq P^b V^{\pi}, \ \forall b \in A \setminus \{a\}$$

and substituting in definition of V^{π}...

$$(P^a - P^b)(I - \gamma P^a)^{-1} R \succeq 0$$ element-wise inequality

- to make any single step deviation from π as costly as possible, choose function R to maximize:

$$\sum_{s \in S} \left(Q^{\pi}(s,a) - \max_{b \in A \setminus \{a\}} Q^{\pi}(s,b)\right)$$ maximizes diff b/t optimal action and next-best action

regularization to favor simpler reward functions

$$\max \sum_{i=1}^{N} \min_{a \in \{a_2, \ldots, a_k\}} \left\{(P^{a_1}(i) - P^{a_2}(i))(I - \gamma P^{a_1})^{-1} R\right\} - \lambda \|R\|_1$$

$$\text{s.t.} \quad (P^{a_1} - P^{a_2})(I - \gamma P^{a_1})^{-1} R \succeq 0 \quad \forall a \in A \setminus \{a_1\}$$

$$|R_i| \leq R_{max}, \ i = 1, \ldots, N$$

this is the linear programming (LP) formulation

- can use function approximation, defining R as:

$$R(s) = \alpha_1 \phi_1(s) + \alpha_2 \phi_2(s) + \cdots + \alpha_d \phi_d(s)$$

linearity of expectation gives:

$$V^{\pi} = \alpha_1 V_1^{\pi} + \alpha_2 V_2^{\pi} + \cdots + \alpha_d V_d^{\pi}$$

and can then substitute this into the LP formulation

Exploration and Controlled sensing |

- Bayesian models
  - to update mean θ_n and precision β_n (i.e., after n obs) after observing measurement W (a random variable) (has $\theta_W^2, \frac{1}{\beta_W}$):

$$\theta_{n+1} = \frac{\beta_n \theta_n + \beta_W W_{n+1}}{\beta_n + \beta_W} = \frac{\overbrace{\text{precision} \cdot \text{mean}}^{\text{old}} + \overbrace{\text{precision} \cdot \text{mean}}^{\text{new}}}{\text{sum of precisions}}$$

$$\beta_{n+1} = \beta_n + \beta_W$$

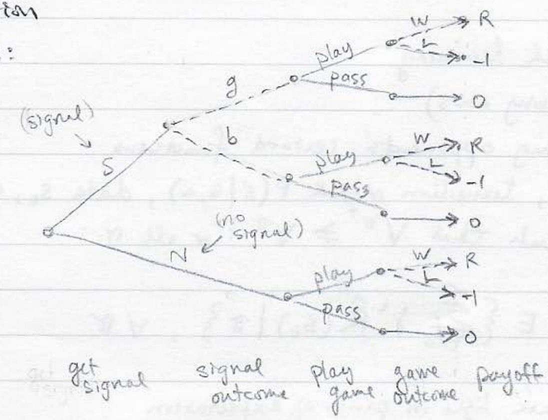
$$\theta_{n+1} = (\beta_{n+1})^{-1}(\beta_n \theta_n + \beta_W W_{n+1})$$

  - updating variance: $E[Var(\mu|W)] = Var(\mu) - Var(E[\mu|W])$ $\Rightarrow$ smaller than original

random variable name, not a mean

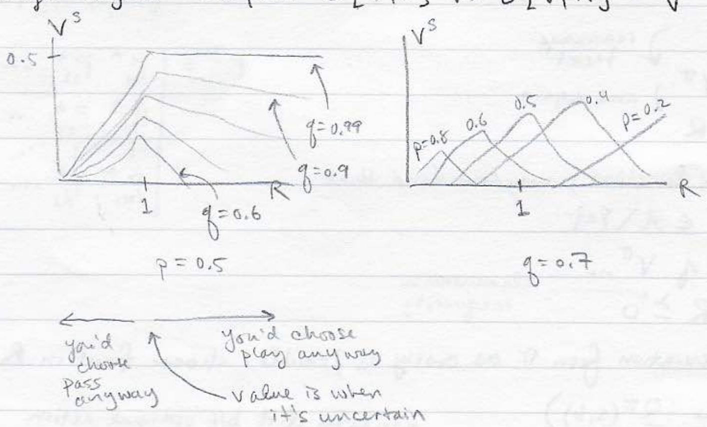
Exploration and controlled sensing (cont'd)

- information acquisition
  - simple game:



with no signal: $E[V|N] = \max\{0, pR - (1-p)\}$ (assume you pass if expected value is negative)

- assume signal correctly predicts outcome w/ prob q: $P[S=g|W] = P[S=b|L] = q$
- decision to get signal: compare $E[V|S]$ vs. $E[V|N] = V^S(R, p, q)$



- Bayesian models (again)
  - with multiple measurements: just add n in front of W terms

$$\beta_n = \beta_0 + n\beta_W$$

$$\theta_n = \frac{\beta_0\theta_0 + n\beta_W \bar{W}_n}{\beta_0 + n\beta_W} \quad \text{where } \bar{W}_n = \frac{1}{n}\sum_{k=1}^{n} W_k$$

- Want to derive greedy heuristic for information gain
  - use entropy of expected information: $H_p(x) = -\int p(x)\log p(x)\,dx$ or $-\sum_x p(x)\log p(x)$
  - want to <u>minimize</u> expected entropy of belief after executing an action:

$$H_b(x'|z,u) = -\int \underbrace{B(b,z,u)(x')}_{\text{"}p(x)\text{"}} \log B\underbrace{(b,z,u)(x')}_{\text{"}x\text{"}}\,dx'$$

i.e., make them more different

  - information gain is: $I_b(u) = H_p(x) - H_b(x'|u)$
  - so choose u to max I_b (subject to cost): $\pi(b) = \underset{u}{\operatorname{argmax}}\ \alpha\underbrace{\left(H_p(x) - E_z[H_b(x'|z,u)]\right)}_{\text{expected information gain}} + \underbrace{\int r(x,u)b(x)dx}_{\text{expected cost}}$

r negative

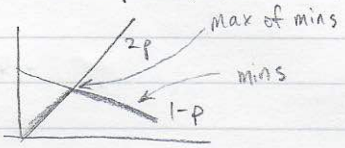
RL

notes for final

<u>multi-agent reinforcement learning!</u>

- already know prisoner's dilemma, Bach/Stravinsky, matching pennies, concepts of constant sum games, dominated strategies, mixed strategies
- minimax seems to be an (optimal?) strategy in zero-sum games (or at least your opponent is taking actions that minimize your payoff)
  - e.g., opponent decides to choose R with prob p and L with prob 1−p

		opponent	
		L	R
player	L	(1,−1)	(0,0)
	R	(0,0)	(2,−2)



if hider hides in R every time, chooser gets $\frac{1}{3}(2) = \frac{2}{3}$

for L, it's $\frac{2}{3}(1) = \frac{2}{3}$ } same

note player chooses worse potential payoff option more often

hider loses $\frac{2}{3}$ on average, so that's the cost of playing for him (must be paid $\geq \frac{2}{3}$ to play)

- Nash equilibrium: all players are playing best responses to each other
- algorithms for stochastic games (SG) solution:
  - "a first algorithm for SG solution [Shapley]"

    1. initialize V arbitrarily
    2. Repeat

       (a) For each state, $s \in S$, compute the matrix

       given V, get π $\quad G_s(V) = \left[g_{a \in A} : R(s,a) + \gamma \sum_{s' \in S} T(s,a,s')V(s') \right]$

       opponent actions

       $G_s(V) = \left[\quad \right]$

       player actions

       (b) For each state, $s \in S$, update V,

       given π, get V $\quad V(s) \leftarrow \text{Value}\left[G_s(V)\right]$

    performs linear programming to solve the game; returns expected value of playing equilibrium strategy

    - this is nearly identical to value iteration for MDPs, with the "max" operator replaced by the value operator
  - policy iteration algorithm for SGs:

    1. Initialize V arbitrarily
    2. Repeat

       $\rho_i \leftarrow \text{Solve}_i\left[G_s(V)\right]$

       $V(s) \leftarrow E\{\sum \gamma^t r_t \mid s_0 = s, \rho_i\}$

    ρ_i = stochastic policy

    Solve_i returns player i's equilibrium strategy
  - Q-learning for SGs:

    1. Initialize $Q(s \in S, a \in A)$ arbitrarily, and set α to be the learning rate
    2. Repeat

       (a) From state s select action a_i that solves the matrix game $\left[Q(s,a)_{a \in A}\right]$ with some exploration

       (b) Observing joint-action a, reward r, and state s′

       $$Q(s,a) \leftarrow (1-\alpha)\, Q(s,a) + \alpha\left(r + \gamma V(s')\right)$$

       where

       $$V(s) = \text{Value}\left(\left[Q(s,a)_{a \in A}\right]\right)$$