Lecture 15: Bandit problems. Markov Processes

- Bandit problems
- Action values (and now to compute them)
- Exploration-exploitation trade-off
- Simple exploration strategies
  - ϵ-greedy
  - Softmax (Boltzmann) exploration
  - Optimism in the face of uncertainty
- Markov chains
- Markov decision processes

Recall: Lotteries and utilities

- Last time we defined a lottery as a set of *lottery* as a set of outcomes and a probability distribution over them
- If an agent has a "consistent" set of preferences over outcomes, each outcome can be associated with a *utility (reward, payoff)* (a real number)
- The utility of a lottery $L = (C, P)$ is the expected value of the utility of its outcomes:

$$U(L) = \sum_{c_i \in C} P(c_i) U(c_i)$$

- From now on, we will always talk about utilities instead of consequences
- The goal of a rational agent will be to *maximize its expected utility, over the long term*

Bandit problems

- Invented in early 1950s by Robbins to model decision making under uncertainty when the environment is unknown
- Named after the original name of slot machines

- A *k-armed bandit* is a collection of k *actions (arms)*, each having a lottery associated with it
- Unlike in the settings discussed before, the lotteries are *unknown ahead of time*
- The best action must be determined by interacting with the environment

Application: Internet advertising

- You are a large Internet company who sells advertising on the web site of their search engine
- You get paid by a company placing an ad with you if the ad gets clicked
- On any web page, you can choose one of n possible ads to display
- Each ad can be viewed as an action, with an unknown probability of success
- If the action succeeds, there is a reward, otherwise no reward
- What is the best strategy to advertise?
- Note that this setup requires *no knowledge* of the user, the ad content, the web page content... which is great if you need to make a decision very fast

Application: Network server selection

- Suppose you can choose to send a job from a user to be processed to one of several servers
- The servers have different processing speed (due to geographic location, load, ...)
- Each server can be viewed as an arm
- Over time, you want to learn what is the best arm to play
- Used in routing, DNS server selection, cloud computing...

Playing a bandit

- You can choose repeatedly among the k actions; each choice is called a *play*
- After each play a_t the machine gives a reward r_t drawn from the distribution associated with a_t
- The *value of action a* is its expected utility:

$$Q^*(a) = E\{r|a\}$$

  (Note that r is drawn from a probability distribution that depends on a)
- The objective is to play in a way that maximizes the reward obtained in the long run (e.g. over 1000 plays)

Estimating action values

- Suppose that action a has been chosen n times, and the rewards received were $r_1, r_2 \dots r_n$.
- Then we can estimate the value of the action as the *sample average* of the rewards obtained:

$$Q_n(a) = \frac{1}{n}(r_1 + r_2 + \dots r_n)$$

- As we take the action more, this estimate becomes more accurate (law of large numbers) and in the limit:

$$\lim_{n\to\infty} Q_n(a) = Q^*(a)$$

One can even express how fast Q_n approaches Q^*

Estimating action values incrementally

- Do we need to remember all rewards so far to estimate $Q_n(a)$?
No! Just keep a running average:

$$\begin{aligned} Q_{n+1}(a) &= \frac{1}{n+1}(r_1 + r_2 + \dots + r_{n+1}) \\ &= \frac{1}{n+1}r_{n+1} + \frac{1}{n+1}\frac{n}{n}(r_1 + r_2 + \dots + r_n) \\ &= \frac{1}{n+1}r_{n+1} + \frac{n}{n+1}Q_n(a) \\ &= Q_n(a) + \frac{1}{n+1}(r_{n+1} - Q_n(a)) \end{aligned}$$

- The first term is the old value estimate; the second term is the *error* between the new sample and the old value estimate, weighted by a *step size*
- We will see this pattern of update a lot in learning algorithms

What if the problem is non-stationary?

- Using the sample average works if the problem we want to learn is *stationary* ($Q^*(a)$ does not change)
- In some applications (e.g. advertising) $Q^*(a)$ may change over time, so we want the most recent rewards to be emphasized
- Instead of $1/n$ use a *constant step size* $\alpha \in (0, 1)$ in the value updates:

$$Q_{n+1}(a) = Q_n(a) + \alpha(r_{n+1} - Q_n(a))$$

- This leads to a *recency-weighted average*:

$$Q_n(a) = (1 - \alpha)^n Q_0(a) + \sum_{i=1}^{n} \alpha(1 - \alpha)^{n-i} r_i$$

Because $\alpha \in (0, 1)$, the most recent reward, r_n, has the highest weight, and the weights decrease exponentially for older rewards.

How to choose actions?

- Suppose you played a 2-armed bandit 3 times (2 for the left arm, 1 for the right arm). With the left arm, you won once and lost once. With the right arm you lost. What should you do next?
- Suppose you played a 2-armed bandit 30 times (20 for the left arm, 10 for the right arm). With the left arm, you won 10 times and lost 10 times. With the right arm you won 7 times and lost 3 times. What should you do next?
- Suppose you played a 2-armed bandit 3000 times (2000 for the left arm, 1000 for the right arm). With the left arm, you won 1000 times and lost 1000 times. With the right arm you won 700 times and lost 300 times. What should you do next?

Exploration-exploitation trade-off

- On one hand, you need to *explore* actions, to figure out which one is best (which means some amount of random choice)
- On the other hand, you want to *exploit* the knowledge you already have, which means picking the *greedy action*:

$$a_t^* = \arg\max_a Q_t(a)$$

- You cannot explore all the time (because you may lose big-time)
- You cannot exploit all the time, because you may end up with a sub-optimal action
- If the environment is stationary, you may want to reduce exploration over time
- If the environment is not stationary, you can never stop exploring

Exploration-exploitation trade-off

- Simple randomized strategies:
  - ϵ-greedy
  - Softmax (Boltzman) exploration
- Deterministic strategy: optimism in the face of uncertainty
  - Optimistic initialization
  - Confidence intervals
  - UCB1
  - ...
- A lot of other work!
  - Gittins indices
  - Action elimination
  - ...

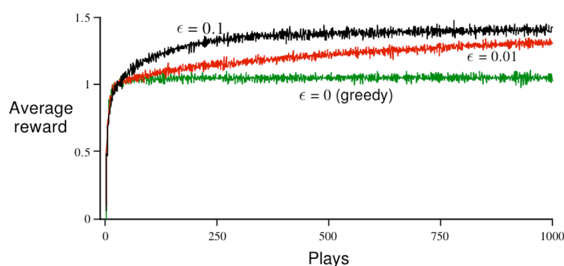
ϵ-greedy action selection

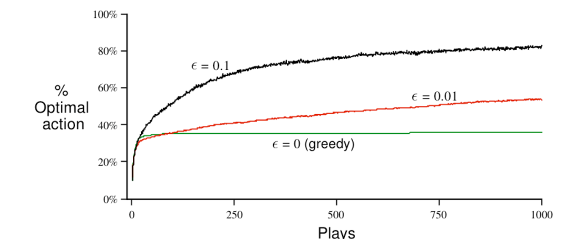
- Pick $\epsilon \in (0, 1)$ (a constant) - usually small (e.g. 0.1)
- On every play, with probability ϵ you pull a random arm
- With probability $1 - \epsilon$, pull the best arm according to current estimates (greedy action)
- You can make ϵ depend on time (e..g $1/t$, $1/\sqrt{t}$, ...)
- Advantage: Very simple! Easy to understand, easy to implement
- Disadvantage: leads to *discontinuities* (a small change in action values may lead to a big change in policy)

Illustration: 10-armed bandit problem

- The mean reward for each arm is chosen form a normal distribution with mean 0 and standard deviation 1
- Rewards are generated from a normal distribution around the true mean, with st. dev. 1
- We average 2000 different independent runs: start from scratch, do 1000 pulls
- How does ϵ influence the algorithm performance?

Illustration: 10-armed bandit problem





- If $\epsilon = 0$, convergence to a sub-optimal strategy
- If ϵ is too low, convergence is slow
- If ϵ is too high (not pictured here) rewards received during learning may be too low, and have high variance

Softmax action selection

- Key idea: make the action probabilities a *function of the current action values*
- Like in simulated annealing, we use the Boltzman distribution:
- At time t we choose action a with probability proportional to:

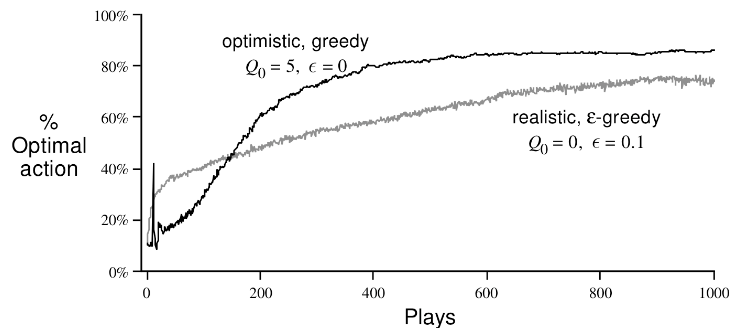
$$e^{Q_t(a)/\tau}$$

- Normalize probabilities so they sum to 1 over the actions
- τ is a temperature parameter, with effect similar to the case of simulated annealing

Optimism in the face of uncertainty

- If you do not know anything about an action, assume it's great!
- Very powerful idea - recall A^* and admissible heuristics
- Simple implementation: just initialize all action values higher than they could possibly be
- Choose actions according to a *deterministic strategy*: always pick the action with the best current estimate
- Whatever you do, you will be "disappointed", which leads to trying out all actions
- This is a *deterministic strategy*: always pick the action with the best current estimate

Illustration: optimistic initialization



- Leads to more rapid exploration that ϵ-greedy, which is bad in the short run but good in the long run
- Once the optimal strategy is found, it stays there (since there is no randomness)

More sophisticated idea: Confidence intervals

- Suppose you have a random variable X with mean $E[X] = \mu$
- The *standard deviation* of X measures how much X is spread around its mean:

$$\sigma = \sqrt{E[(X-\mu)^2]}$$

  This can be estimated from samples
- Idea: add one standard deviation to the mean, choose actions greedily wrt this bound

Upper Confidence Bounds (UCB)

- Cf. UCT algorithm for search, but here the confidence bounds are remembered over time rather than just generated through simulations whenever needed.
- Very similar formula: pick greedily wrt

$$Q(a) + \sqrt{\frac{2 \log n}{n(a)}}$$

  where n is the total number of actions executed so far and $n(a)$ is the number of times action a was picked
- Several tweaks of the "bonus" term have been proposed, and a lot of theoretical analysis for this type of method has been done

Which algorithm is best?

- All algorithms converge in the limit to correct action values (given appropriate parameter changes if need be), assuming the environment is stationary
- UCB has provably the fastest convergence when considering *regret*:

$$\sum_{t=1}^{\infty} Q(a^*) - Q(a_t)$$

  This sum has $O(\log(t))$ for UCB, and a matching lower bound exists
- However, when considering a *finite training period* after which the greedy policy is used forever, the simple strategies often perform much better.

Contextual bandits

- The usual bandit problem has no notion of "state", we just observe some interactions and payoffs.
- In general, more information may be available, e.g. placing ads on a Gmail web page, you can observe the current words displayed
- *Contextual bandits* have some state information, summarized in a vector of measurements $\mathbf{s}$

  E.g. What words out of a large dictionary appear on the web page
- The value of an action a will now be dependent on the state, e.g.

$$Q(\mathbf{s}, a) = \mathbf{w}_a^T \mathbf{s}$$

  where $\mathbf{w}_a$ are vectors of parameters (one for every action
- We will talk in a bit about learning the parameters $\mathbf{w}$ in this context
- Exploration methods remain very similar

Sequential Decision Making

- Decision graphs provide a useful tool for decision making
- If more than one decision has to be taken, reasoning about all of them in general is very expensive
- In bandit problems, the assumption is of *repeated* interaction with an unknown environment over time
- But once an action is taken, the environment is still the same (does not change as a result of the action)
- *Markov Decision Processes (MDPs)* provide a framework for modelling *sequential decision making*, where the environment has different states which change over time as a result of the agent's actions.

A simpler case: prediction through time

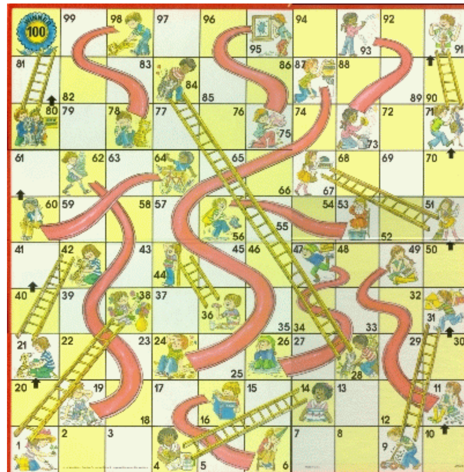
- You want to ask your parents for more money, and you are imagining the phone conversation
- How should you start?
  - "I need more money"
  - "I got an A on the AI midterm!"
  - "I'm just calling to see how your doing..."
- You want to *predict* how likely they are to give you money, based on the different ways in which the dialogue could unfold.
- Any dialogue / communication can be viewed as a *sequence of interactions* through time

Application examples

- Robotics
  - Where is the robot?
  - If it goes forward, will it bump into a wall?
- Medical applications
  - Fetal heart rate monitoring: is the baby sick or healthy based on the current readings?
  - Monitoring epileptic seizures: based on the neural activity, is a seizure likely to occur?
- Dialogue systems
- Speech recognition
- Web-based applications: will the customer purchase a product based on their buying history?

Markov Chains

- Have you played Candyland? Chutes and ladders?



Example: Simplified Chutes & Ladders

12	11	10	9	8 ↑	7
1	2	3	4	5	6

- Start at state 1
- Roll a die, then move a number of positions given by its value.
- If you land on square 5, you are teleported to 8.
- Whomever gets to 12 first wins!
- Note that there is no skill involved...

Markov Chain Example

12	11	10	9	8 ↑	7
1	2	3	4	5	6

- There is a discrete clock pacing the interaction of the agent with the environment, $t = 0, 1, 2, \dots$
- The agent can be in one of a set of *states*, $S = \{1, 2, \dots 12\}$
- The initial state (at time $t = 0$) is $s_0 = 1$.
- If the agent is in state s_t at time t, the state at time $t + 1$, s_{t+1} is determined *only based on the dice roll at time t*

Example (Continued)

- The probability of the next state, s_{t+1}, *does not depend* on how the agent got to the current state, s_t.
- This is called the *Markov property*
- E.g., suppose that at time t, the agent is in state 3. Then regardless how it got to state 4 (by rolling a 2 or two 1s):

$$\begin{aligned}
P(s_{t+1} = 4 | s_t = 3) &= 1/6 \\
P(s_{t+1} = 8 | s_t = 3) &= 1/3 \text{ (you roll 2 and get teleported or you roll 5)} \\
P(s_{t+1} = 6 | s_t = 3) &= 1/6 \\
P(s_{t+1} = 7 | s_t = 3) &= 1/6 \\
P(s_{t+1} = 9 | s_t = 3) &= 1/6
\end{aligned}$$

- So the game is completely described by the *probability distribution of the next state given the current state*.

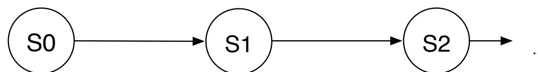
Markov Chain Definition

- Set of *states* S
- *Transition probabilities*: $T : S \times S \to [0, 1]$

$$T(s, s') = P(s_{t+1} = s' | s_t = s)$$

- *Initial state distribution*: $P_0 : S \to [0, 1]$

$$P_0(s) = P(s_0 = s)$$



Things that Can Be Computed

- What is the expected number of time steps (dice rolls) to the finish line?
- What is the expected number of time steps until we reach a state for the first time?
- What is the probability of being in a given state s at time t?
- After t time steps, what is the probability that we have ever been in a given state s?

Example: Decision Making

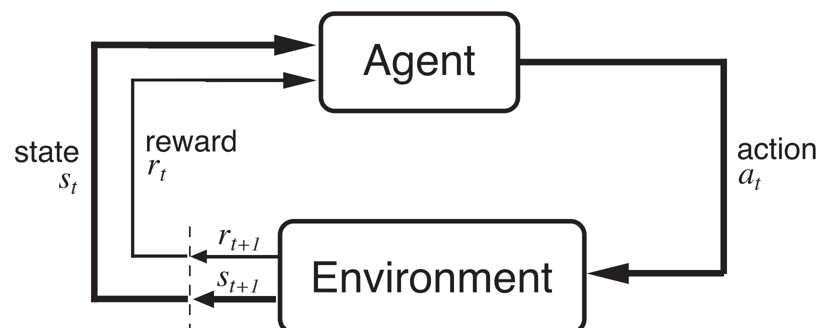
- Suppose that we played the game with two dice
- You roll both dice and then have a choice
  - Take either of the two rolls
  - Take the sum of the two rolls
  - Take the difference of the two rolls
- When you are finished playing the game, your mom will give you a snack, so you want to finish as quickly as possible

The General Problem: Control Learning

- Robot learning to dock on battery charger
- Choosing actions to optimize factory output
- Playing Backgammon, Go, Poker, ...
- Choosing medical tests and treatments for a patient with a chronic illness
- Conversation
- Portofolio management
- Flying a helicopter
- Queue / router control

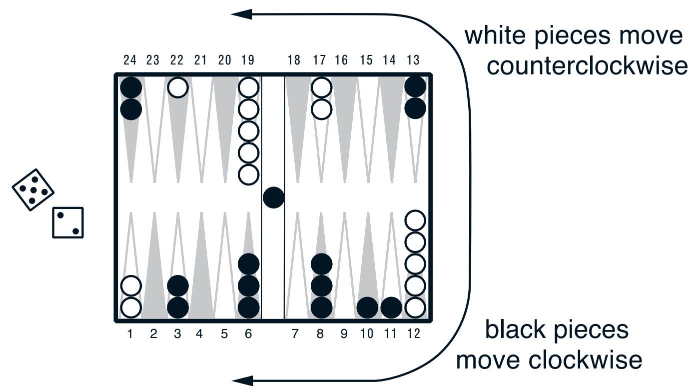
All of these are sequential decision making problems

Reinforcement Learning Problem



- At each discrete time t, the agent (learning system) observes state $s_t \in S$ and chooses action $a_t \in A$
- Then it receives an immediate *reward* r_{t+1} and the state changes to s_{t+1}

Example: Backgammon (Tesauro, 1992-1995)

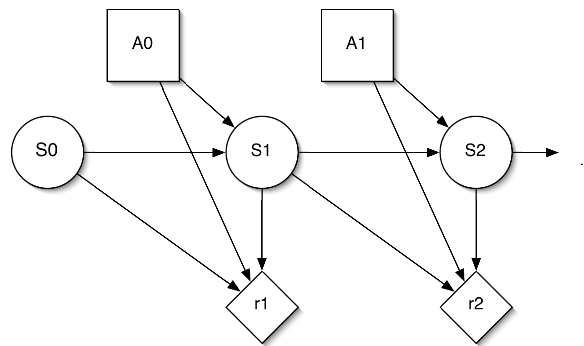


- The states are board positions in which the agent can move
- The actions are the possible moves
- Reward is 0 until the end of the game, when it is ± 1 depending on whether the agent wins or loses

Markov Decision Processes (MDPs)

- Finite set of *states* S (we will lift this later)
- Finite set of *actions* A
- $\gamma =$ *discount factor* for future rewards (between 0 and 1, usually close to 1). Two possible interpretations:
  - At each time step there is a $1 - \gamma$ chance that the agent dies, and does not receive rewards afterwards
  - Inflation rate: if you receive the same amount of money in a year, it will be worth less
- *Markov assumption:* s_{t+1} and r_{t+1} depend only on s_t and a_t but not on anything that happened before time t

MDPs as Decision Graphs



- The graph may be *infinite*
- But it has a very regular structure!
- At each time slice *the structure and parameters are shared*
- We will exploit this property to get efficient inference

Models for MDPs

- Because of the Markov property, an MDP can be completely described by:
  - *Reward function* $r : S \times A \to \mathbb{R}$
    $r_a(s)$ = the immediate reward if the agent is in state s and takes action a
    This is the *short-term utility* of the action
  - *Transition model* (dynamics): $T : S \times A \times S \to [0, 1]$
    $T_a(s, s')$ = probability of going from s to s' under action a

$$T_a(s, s') = P(s_{t+1} = s' | s_t = s, a_t = a)$$

- These form the *model* of the environment

Planning in MDPs

- The goal of an agent in an MDP is to be rational, i.e., *maximize its expected utility* (respect MEU principle)
- In this case, maximizing the immediate utility (given by the immediate reward) is not sufficient.
  - E.g., the agent might pick an action that gives instant gratification, even if it later makes it "die"
- Hence, the goal is to maximize *long-term utility*, also called *return*
- The return is defined as an additive function of all rewards received by the agent.

Returns

- The *return* R_t for a trajectory, starting from time step t, can be defined depending on the type of task
- *Episodic tasks* (e.g. games, trips through a maze etc)

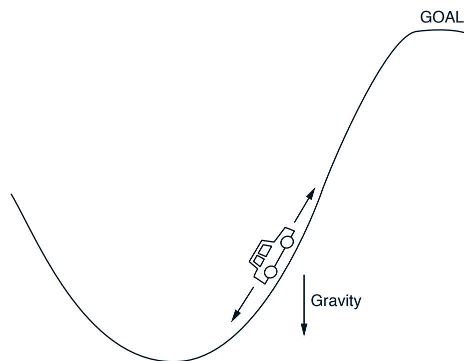
$$R_t = r_{t+1} + r_{t+2} + \cdots + r_T$$

  where T is the time when a terminal state is reached
- *Continuing tasks* (tasks which may go on forever):

$$R_t = r_{t+1} + \gamma r_{t+2} + \gamma^2 r_{t+3} + \cdots = \sum_{k=1}^{\infty} \gamma^{t+k-1} r_{t+k}$$

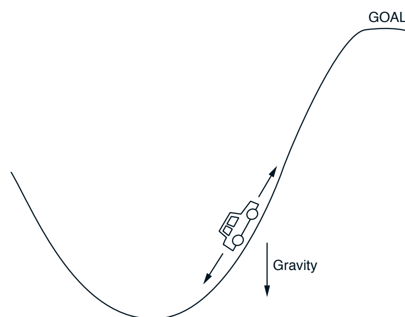
  Discount factor $\gamma < 1$ ensures that the return is finite, assuming that rewards are bounded.

Example: Mountain-Car



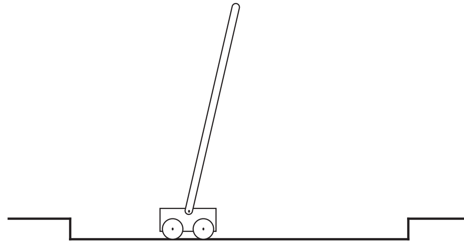
- States: position and velocity
- Actions: accelerate forward, accelerate backward, coast
- We want the car to get to the top of the hill as quickly as possible
- How do we define the rewards? What is the return?

Example: Mountain-Car



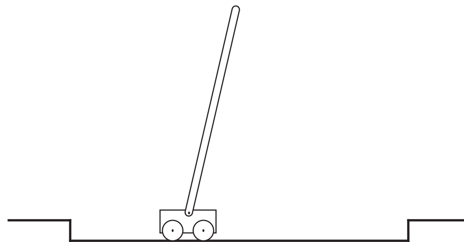
- States: position and velocity
- Actions: accelerate forward, accelerate backward, coast
- Two reward formulations:
  1. reward $= -1$ for every time step, until car reaches the top
  2. reward $= 1$ at the top, 0 otherwise $\gamma < 1$
- In both cases, the return is maximized by minimizing the number of steps to the top of the hill

Example: Pole Balancing



- We can push the cart along the track
- The goal is to avoid failure: pole falling beyond a given angle, or cart hitting the end of the track
- What are the states, actions, rewards and return?

Example: Pole Balancing



- States are described by 4 variables: angle and angular velocity of the pole relative to the cart, position and speed of cart along the track
- We can think of 3 possible actions: push left, push right, do nothing
- Episodic task formulation: reward = +1 for each step before failure
  $\Rightarrow$ return = number of steps before failure
- Continuing task formulation: reward = -1 upon failure, 0 otherwise, $\gamma < 1$
  $\Rightarrow$ return = $-\gamma^k$ if there are k steps before failure

Formulating Problems as MDPs

- The *rewards are quite "objective"* (unlike, e.g., heuristics), they are intended to capture the goal for the problem
- Often there are several ways to formulate a sequential decision problem as an MDP
- It is important that the state is defined in such a way that the Markov property holds
- Sometimes we may start with a more informative or lenient reward structure in the beginning, then change it to reflect the real task
- In psychology/animal learning, this is called *shaping*

Formulating Games as MDPs

- Suppose you played a game against a fixed opponent (possibly stochastic), which acts only based on the current board
- We can formulate this problem as an MDP by *making the opponent part of the environment*
- The states are all possible board positions for your player
- The actions are the legal moves in each state where it is your player's turn
- If we do not care about the length of the game, then $\gamma = 1$
- Rewards can be $+1$ for winning, -1 for losing, 0 for a tie (and 0 throughout the game)
- But it would be hard to define the transition probabilities!
- Later we will talk about how to learn such information from data/experimentation

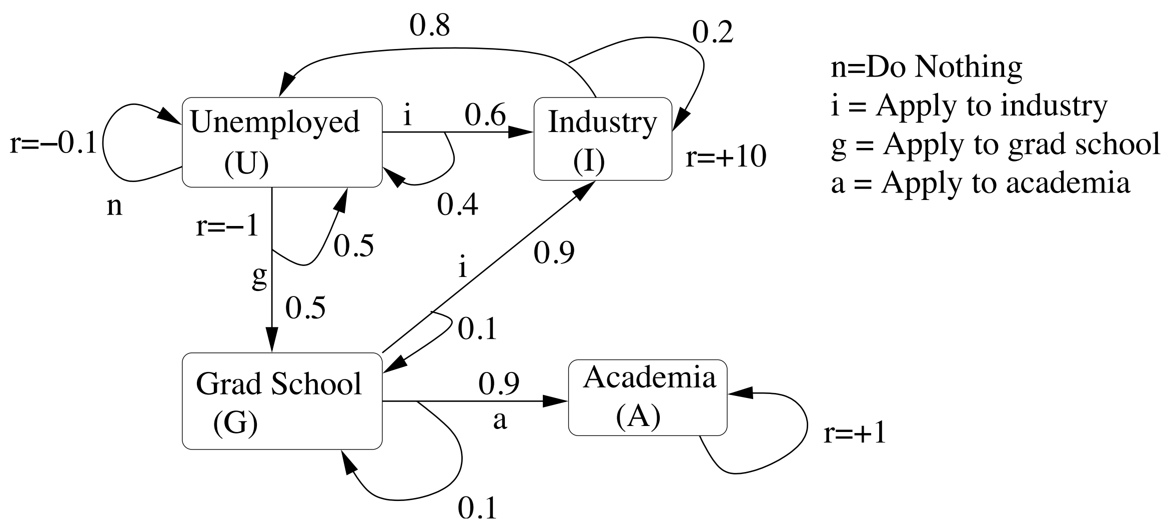
Policies

- The goal of the agent is to find a way of behaving, called a *policy* (plan or strategy) that maximizes the expected value of the return, $E[R_t], \forall t$
- A *policy* is a way of choosing actions based on the state:
  - *Stochastic policy*: in a given state, the agent can "roll a die" and choose different actions

$$\pi : S \times A \to [0,1], \;\; \pi(s,a) = P(a_t = a | s_t = s)$$

  - *Deterministic policy*: in each state the agent chooses a unique action $\pi : S \to A, \;\; \pi(s) = a$

Example: Career Options



What is the best policy?

Value Functions

- Because we want to find a policy which maximizes the expected return, it is a good idea to *estimate the expected return*
- Then we can *search* through the space of policies for a good policy
- *Value functions* represent the expected return, for every state, given a certain policy
- Computing value functions is an intermediate step towards computing good policies

State Value Function

- The *state value function of a policy* π is a function $V^\pi : S \to \mathbb{R}$
- The *value of state* s *under policy* π is the expected return if the agent starts from state s and picks actions according to policy π:

$$V^\pi(s) = E_\pi[R_t | s_t = s]$$

- For a finite state space, we can represent this as an array, with one entry for every state
- We will talk later about methods used for very large or continuous state spaces

Computing the value of policy π

- First, re-write the return a bit:

$$\begin{aligned} R_t &= r_{t+1} + \gamma r_{t+2} + \gamma^2 r_{t+3} + \cdots \\ &= r_{t+1} + \gamma \left(r_{t+2} + \gamma r_{t+3} + \cdots \right) \\ &= r_{t+1} + \gamma R_{t+1} \end{aligned}$$

- Based on this observation, V^π becomes:

$$V^\pi(s) = E_\pi[R_t | s_t = s] = E_\pi[r_{t+1} + \gamma R_{t+1} | s_t = s]$$

- Now we need to recall some properties of expectations...

Detour: Properties of expectations

- Expectation is *additive*: $E[X + Y] = E[X] + E[Y]$
  Proof: Suppose X and Y are discrete, taking values in $\mathcal{X}$ and $\mathcal{Y}$

$$\begin{aligned} E[X + Y] &= \sum_{x_i \in \mathcal{X}, y_i \in \mathcal{Y}} (x_i + y_i) p(x_i, y_i) \\ &= \sum_{x_i \in \mathcal{X}} x_i \sum_{y_i \in \mathcal{Y}} p(x_i, y_i) + \sum_{y_i \in \mathcal{Y}} y_i \sum_{x_i \in \mathcal{X}} p(x_i, y_i) \\ &= \sum_{x_i \in \mathcal{X}} x_i p(x_i) + \sum_{y_i \in \mathcal{Y}} y_i p(y_i) = E[X] + E[Y] \end{aligned}$$

- $E[cX] = cE[X]$ is $c \in \mathbb{R}$ is a constant
  Proof: $E[cX] = \sum_{x_i} c x_i p(x_i) = c \sum_{x_i} x_i p(x_i) = cE[X]$

Detour: Properties of expectations (2)

- The expectation of the product of random variables *is not* equal to the product of expectations, unless the variables are independent

$$E[XY] = \sum_{x_i \in \mathcal{X}, y_i \in \mathcal{Y}} x_i y_i p(x_i, y_i) = \sum_{x_i \in \mathcal{X}, y_i \in \mathcal{Y}} x_i y_i p(x_i | y_i) p(y_i)$$

- If X and Y are independent, then $p(x_i|y_i) = p(x_i)$, we can re-arrange the sums and products and get $E[X]E[Y]$ on the right-hand side
- But is X and Y are not independent, the right-hand side does not decompose!

Going back to value functions...

- We can re-write the value function as:

$$\begin{aligned} V^\pi(s) &= E_\pi[R_t | s_t = s] = E_\pi[r_{t+1} + \gamma R_{t+1} | s_t = s] \\ &= E_\pi[r_{t+1}] + \gamma E[R_{t+1} | s_t = s] \text{ (by linearity of expectation)} \\ &= \sum_{a \in A} \pi(s, a) r_a(s) + \gamma E[R_{t+1} | s_t = s] \text{ (by using definitions)} \end{aligned}$$

- The second term looks a lot like a value function, if we were to condition on s_{t+1} instead of s_t
- So we re-write as:

$$E[R_{t+1} | s_t = s] = \sum_{a \in A} \pi(s, a) \sum_{s' \in S} T_a(s, s') E[R_{t+1} | s_{t+1} = s']$$

- The last term is just $V^\pi(s')$

Bellman equations for policy evaluation

- By putting all the previous pieces together, we get:

$$V^\pi(s) = \sum_{a \in A} \pi(s,a) \left(r_a(s) + \gamma \sum_{s' \in S} T_a(s,s') V^\pi(s') \right)$$

- This is a *system of linear equations* (one for every state) whose unique solution is V^π.
- The uniqueness is ensured under mild technical conditions on the transitions p
- So if we want to find V^π, we could try to solve this system!

Iterative Policy Evaluation

- Main idea: turn Bellman equations into update rules.
  1. Start with some initial guess V_0
  2. During every iteration k, update the value function for all states:

$$V_{k+1}(s) \leftarrow \sum_{a \in A} \pi(s,a) \left(r_a(s) + \gamma \sum_{s' \in S} T_a(s,s') V_k(s') \right), \forall s$$

  3. Stop when the maximum change between two iterations is smaller than a desired threshold (the values stop changing)
- This is a *bootstrapping* algorithm: the value of one state is updated based on the current estimates of the values of successor states
- This is a dynamic programming algorithm
- If you have a linear system that is very big, using this approach avoids a big matrix inversion