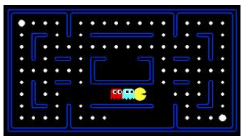
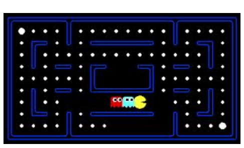
Games II



Announcement

- Midterm is next week (Wednesday, 5/8, 6pm-8pm)
- Topics: all material up to and including today's lecture
- Logistics: look for detailed post on Ed

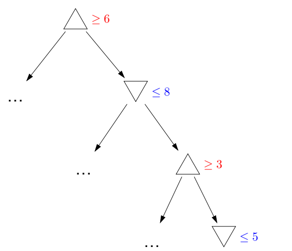
Games: alpha-beta pruning recap



Alpha-beta pruning

Key idea: optimal path

The optimal path is path that minimax policies take.
Values of all nodes on path are the same.



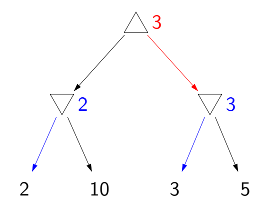
- a_s: lower bound on value of max node s
- b_s: upper bound on value of min node s
- Prune a node if its interval doesn't have non-trivial overlap with every ancestor (store $\alpha_s = \max_{s' \preceq s} a_{s'}$ and $\beta_s = \min_{s' \preceq s} b_{s'}$)

- In general, let's think about the minimax values in the game tree. The value of a node is equal to the utility of at least one of its leaf nodes (because all the values are just propagated from the leaves with min and max applied to them). Call the first path (ordering by children left-to-right) that leads to the first such leaf node the **optimal path**. An important observation is that the values of all nodes on the optimal path are the same (equal to the minimax value of the root).
- Since we are interested in computing the value of the root node, if we can certify that a node is not on the optimal path, then we can prune it and its subtree.
- To do this, during the depth-first exhaustive search of the game tree, we think about maintaining a lower bound ($\geq a_s$) for all the max nodes s and an upper bound ($\leq b_s$) for all the min nodes s.
- If the interval of the current node does not non-trivially overlap the interval of every one of its ancestors, then we can prune the current node. In the example, we've determined the root's node must be ≥ 6. Once we get to the node on at ply 4 and determine that node is ≤ 5, we can prune the rest of its children since it is impossible that this node will be on the optimal path (≤ 5 and ≥ 6 are incompatible). Remember that all the nodes on the optimal path have the same value.
- Implementation note: for each max node s, rather than keeping a_s, we keep α_s, which is the maximum value of $a_{s'}$ over s and all its max node ancestors. Similarly, for each min node s, rather than keeping b_s, we keep β_s, which is the minimum value of $b_{s'}$ over s and all its min node ancestors. That way, at any given node, we can check interval overlap in constant time regardless of how deep we are in the tree.

Move ordering

Pruning depends on order of actions.

Can't prune the 5 node:



[live solution: alpha-beta pruning]

- We have so far shown that alpha-beta pruning correctly computes the minimax value at the root, and seems to save some work by pruning subtrees. But how much of a savings do we get?
- The answer is that it depends on the order in which we explore the children. This simple example shows that with one ordering, we can prune the final leaf, but in the second, we can't.

Move ordering

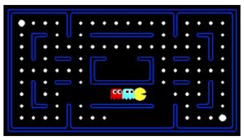
Which ordering to choose?

- Worst ordering: $O(b^{2 \cdot d})$ time
- Best ordering: $O((\sqrt{b - \frac{3}{4}} + \frac{1}{2})^{2 \cdot d}) \simeq O(b^{2 \cdot 0.5d})$ time
- Random ordering: $O(b^{2 \cdot 0.75d})$ time when $b = 2$
- Random ordering: $O((\frac{b-1+\sqrt{b^2+14b+1}}{4})^{2 \cdot d})$ for general b

In practice, can use evaluation function Eval(s):

- Max nodes: order successors by decreasing Eval(s)
- Min nodes: order successors by increasing Eval(s)

Games: TD-learning



Evaluation function

Old: hand-crafted

Example: chess

Eval(s) = material + mobility + king-safety + center-control
material $= 10^{100}(K - K') + 9(Q - Q') + 5(R - R') +$
$3(B - B' + N - N') + 1(P - P')$
mobility $= 0.1$(num-legal-moves − num-legal-moves$'$)
...

New: learn from data

$$\text{Eval}(s) = V(s; \mathbf{w})$$

- Having a good evaluation function is one of the most important components of game playing. So far we've shown how one can manually specify the evaluation function by hand. However, this can be quite tedious, and moreover, how does one figure out to weigh the different factors? In this lecture, we will consider a method for learning this evaluation function automatically from data.
- The three ingredients in any machine learning approach are to determine the (i) model family (in this case, what is $V(s; \mathbf{w})$?), (ii) where the data comes from, and (iii) the actual learning algorithm. We will go through each of these in turn.

Model for evaluation functions

Linear:

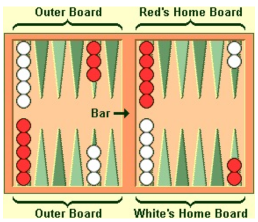
$$V(s; \mathbf{w}) = \mathbf{w} \cdot \phi(s)$$

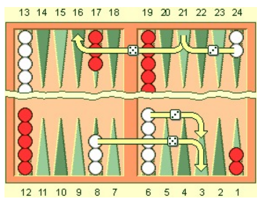
Neural network:

$$V(s; \mathbf{w}, \mathbf{v}_{1:k}) = \sum_{j=1}^{k} w_j \sigma(\mathbf{v}_j \cdot \phi(s))$$

- When we looked at Q-learning, we considered linear evaluation functions (remember, linear in the weights $\mathbf{w}$). This is the simplest case, but it might not be suitable in some cases.
- But the evaluation function can really be any parametrized function. For example, the original TD-Gammon program used a neural network, which allows us to represent more expressive functions that capture the non-linear interactions between different features.
- Any model that you could use for regression in supervised learning you could also use here.

Example: Backgammon

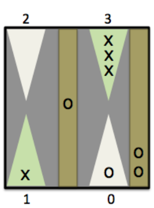




- As an example, let's consider the classic game of backgammon. Backgammon is a two-player game of strategy and chance in which the objective is to be the first to remove all your pieces from the board.
- The simplified version is that on your turn, you roll two dice, and choose two of your pieces to move forward that many positions.
- You cannot land on a position containing more than one opponent piece. If you land on exactly one opponent piece, then that piece goes on the bar and has start over from the beginning. (See the Wikipedia article for the full rules.).

Features for Backgammon

state s



Features $\phi(s)$:

$[(\#\text{ o in column 0}) = 1]$: 1

$[(\#\text{ o on bar})]$: 1

$[(\text{fraction o removed})]$: $\frac{1}{2}$

$[(\#\text{ x in column 1}) = 1]$: 1

$[(\#\text{ x in column 3}) = 3]$: 1

$[(\text{is it o's turn})]$: 1

- As an example, we can define the following features for Backgammon, which are inspired by the ones used by TD-Gammon.
- Note that the features are pretty generic; there is no explicit modeling of strategies such as trying to avoid having singleton pieces (because it could get clobbered) or preferences for how the pieces are distributed across the board.
- On the other hand, the features are mostly **indicator** features, which is a common trick to allow for more expressive functions using the machinery of linear regression. For example, instead of having one feature whose value is the number of pieces in a particular column, we can have multiple features for indicating whether the number of pieces is over some threshold.

Generating data

Generate using policies based on current $V(s; \mathbf{w})$:

$$\pi_{\text{agent}}(s; \mathbf{w}) = \arg \max_{a \in \text{Actions}(s)} V(\text{Succ}(s, a); \mathbf{w})$$

$$\pi_{\text{opp}}(s; \mathbf{w}) = \arg \min_{a \in \text{Actions}(s)} V(\text{Succ}(s, a); \mathbf{w})$$

Note: don't need to randomize (ϵ-greedy) because game is already stochastic (backgammon has dice) and there's function approximation

Generate episode:

$$s_0; a_1, r_1, s_1; a_2, r_2, s_2; a_3, r_3, s_3; \dots; a_n, r_n, s_n$$

- The second ingredient of doing learning is generating the data. As in reinforcement learning, we will generate a sequence of states, actions, and rewards by simulation — that is, by playing the game.
- In order to play the game, we need two exploration policies: one for the agent, one for the opponent. The policy of the dice is fixed to be uniform over $\{1, \dots, 6\}$ as expected.
- A natural policy to use is one that uses our current estimate of the value $V(s; \mathbf{w})$. Specifically, the agent's policy will consider all possible actions from a state, use the value function to evaluate how good each of the successor states are, and then choose the action leading to the highest value. Generically, we would include $\text{Reward}(s, a, \text{Succ}(s, a))$, but in games, all the reward is at the end, so $r_t = 0$ for $t < n$ and $r_n = \text{Utility}(s_n)$. Symmetrically, the opponent's policy will choose the action that leads to the lowest possible value.
- Given this choice of π_{agent} and π_{opp}, we generate the actions $a_t = \pi_{\text{Player}(s_{t-1})}(s_{t-1})$, successors $s_t = \text{Succ}(s_{t-1}, a_t)$, and rewards $r_t = \text{Reward}(s_{t-1}, a_t, s_t)$.
- In reinforcement learning, we saw that using an exploration policy based on just the current value function is a bad idea, because we can get stuck exploiting local optima and not exploring. In the specific case of Backgammon, using deterministic exploration policies for the agent and opponent turns out to be fine, because the randomness from the dice naturally provides exploration.

Learning algorithm

Episode:

$s_0; a_1, r_1, s_1; a_2, r_2, s_2; a_3, r_3, s_3; \ldots, a_n, r_n, s_n$

A small piece of experience:

$$(s, a, r, s')$$

Prediction:

$$V(s; \mathbf{w})$$

Target:

$$r + \gamma V(s'; \mathbf{w})$$

- With a model family $V(s; \mathbf{w})$ and data $s_0, a_1, r_1, s_1, \ldots$ in hand, let's turn to the learning algorithm.
- A general principle in learning is to figure out the **prediction** and the **target**. The prediction is just the value of the current function at the current state s, and the target uses the data by looking at the immediate reward r plus the value of the function applied to to the successor state s' (discounted by γ). This is analogous to the SARSA update for Q-values, where our target actually depends on a one-step lookahead prediction.

General framework

Objective function:

$$\frac{1}{2}(\text{prediction}(\mathbf{w}) - \text{target})^2$$

Gradient:

$$(\text{prediction}(\mathbf{w}) - \text{target}) \nabla_{\mathbf{w}} \text{prediction}(\mathbf{w})$$

Update:

$$\mathbf{w} \leftarrow \mathbf{w} - \eta \underbrace{(\text{prediction}(\mathbf{w}) - \text{target}) \nabla_{\mathbf{w}} \text{prediction}(\mathbf{w})}_{\text{gradient}}$$

- Having identified a prediction and target, the next step is to figure out how to update the weights. The general strategy is to set up an objective function that encourages the prediction and target to be close (by penalizing their squared distance).
- Then we just take the gradient with respect to the weights $\mathbf{w}$.
- Note that even though technically the target also depends on the weights $\mathbf{w}$, we treat this as constant for this derivation. The resulting learning algorithm by no means finds the global minimum of this objective function. We are simply using the objective function to motivate the update rule.

Temporal difference (TD) learning

Algorithm: TD learning

On each (s, a, r, s'):

$$\mathbf{w} \leftarrow \mathbf{w} - \eta[\underbrace{V(s; \mathbf{w})}_{\text{prediction}} - \underbrace{(r + \gamma V(s'; \mathbf{w}))}_{\text{target}}] \nabla_{\mathbf{w}} V(s; \mathbf{w})$$

For linear functions:

$$V(s; \mathbf{w}) = \mathbf{w} \cdot \phi(s)$$

$$\nabla_{\mathbf{w}} V(s; \mathbf{w}) = \phi(s)$$

- Plugging in the prediction and the target in our setting yields the TD learning algorithm. For linear functions, recall that the gradient is just the feature vector.

Example of TD learning

Step size $\eta = 0.5$, discount $\gamma = 1$, reward is end utility

Example: TD learning

S1	**r:0**	**S4**	**r:0**	**S8**	**r:1**	**S9**
$\phi: \binom{0}{1}$		$\phi: \binom{1}{0}$		$\phi: \binom{1}{2}$		$\phi: \binom{1}{0}$
$\mathbf{w}: \binom{0}{0}$	p:0 t:0 p-t:0	$\mathbf{w}: \binom{0}{0}$	p:0 t:0 p-t:0	$\mathbf{w}: \binom{0}{0}$	p:0 t:1 p-t:-1	$\mathbf{w}: \binom{0.5}{1}$
S1	**r:0**	**S2**	**r:0**	**S6**	**r:0**	**S10**
$\phi: \binom{0}{1}$		$\phi: \binom{1}{0}$		$\phi: \binom{0}{0}$		$\phi: \binom{1}{0}$
$\mathbf{w}: \binom{0.5}{1}$	p:1 t:0.5 p-t:0.5	$\mathbf{w}: \binom{0.5}{0.75}$	p:0.5 t:0 p-t:0.5	$\mathbf{w}: \binom{0.25}{0.75}$	p:0 t:0.25 p-t:-0.25	$\mathbf{w}: \binom{0.25}{0.75}$

- Here's an example of TD learning in action. We have two episodes: [S1, 0, S4, 0, S8, 1, S9] and [S1, 0, S2, 0, S6, 0, S10].
- In games, all the reward comes at the end and the discount is 1. We have omitted the action because TD learning doesn't depend on the action.
- Under each state, we have written its feature vector, and the weight vector before updating on that state. Note that no updates are made until the first non-zero reward. Our prediction is 0, and the target is $1 + 0$, so we subtract $-0.5[1, 2]$ from the weights to get $[0.5, 1]$.
- In the second row, we have our second episode, and now notice that even though all the rewards are zero, we are still making updates to the weight vectors since the prediction and targets computed based on adjacent states are different.

Comparison

Algorithm: TD learning

On each (s, a, r, s'):

$$\mathbf{w} \leftarrow \mathbf{w} - \eta[\underbrace{\hat{V}_\pi(s;\mathbf{w})}_{\text{prediction}} - \underbrace{(r + \gamma \hat{V}_\pi(s';\mathbf{w}))}_{\text{target}}]\nabla_{\mathbf{w}} \hat{V}_\pi(s;\mathbf{w})$$

Algorithm: Q-learning

On each (s, a, r, s'):

$$\mathbf{w} \leftarrow \mathbf{w} - \eta[\underbrace{\hat{Q}_{\text{opt}}(s, a;\mathbf{w})}_{\text{prediction}} - \underbrace{(r + \gamma \max_{a' \in \text{Actions}(s)} \hat{Q}_{\text{opt}}(s', a';\mathbf{w}))}_{\text{target}}]\nabla_{\mathbf{w}} \hat{Q}_{\text{opt}}(s, a;\mathbf{w})$$

Comparison

Q-learning:

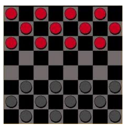
- Operate on $\hat{Q}_{\text{opt}}(s, a;\mathbf{w})$
- Off-policy: value is based on estimate of optimal policy
- To use, don't need to know MDP transitions $T(s, a, s')$

TD learning:

- Operate on $\hat{V}_\pi(s;\mathbf{w})$
- On-policy: value is based on exploration policy (usually based on $\hat{V}_\pi$)
- To use, need to know rules of the game $\text{Succ}(s, a)$

- TD learning is very similar to Q-learning. Both algorithms learn from the same data and are based on gradient-based weight updates.
- The main difference is that Q-learning learns the Q-value, which measures how good an action is to take in a state, whereas TD learning learns the value function, which measures how good it is to be in a state.
- Q-learning is an off-policy algorithm, which means that it tries to compute Q_{opt}, associated with the optimal policy (not Q_π), whereas TD learning is on-policy, which means that it tries to compute V_π, the value associated with a fixed policy π. Note that the action a does not show up in the TD updates because a is given by the fixed policy π. Of course, we usually are trying to optimize the policy, so we would set π to be the current guess of optimal policy $\pi(s) = \arg\max_{a \in \text{Actions}(s)} V(\text{Succ}(s, a); \mathbf{w})$.
- When we don't know the transition probabilities and in particular the successors, the value function isn't enough, because we don't know what effect our actions will have. However, in the game playing setting, we do know the transitions (the rules of the game), so using the value function is sufficient.

Learning to play checkers

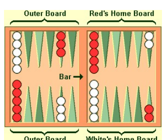


Arthur Samuel's checkers program [1959]:

- Learned by playing itself repeatedly (self-play)
- Smart features, linear evaluation function, use intermediate rewards
- Used alpha-beta pruning + search heuristics
- Reach human amateur level of play
- IBM 701: 9K of memory!

- The idea of using machine learning for game playing goes as far back as Arthur Samuel's checkers program. Many of the ideas (using features, alpha-beta pruning) were employed, resulting in a program that reached a human amateur level of play. Not bad for 1959!

Learning to play Backgammon

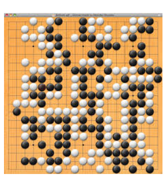


Gerald Tesauro's TD-Gammon [1992]:

- Learned weights by playing itself repeatedly (1 million times)
- Dumb features, neural network, no intermediate rewards
- Reached human expert level of play, provided new insights into opening

- Tesauro refined some of the ideas from Samuel with his famous TD-Gammon program provided the next advance, using a variant of TD learning called TD(λ). It had dumber features, but a more expressive evaluation function (neural network), and was able to reach an expert level of play.

Learning to play Go



AlphaGo Zero [2017]:

- Learned by self play (4.9 million games)
- Dumb features (stone positions), neural network, no intermediate rewards, Monte Carlo Tree Search
- Beat AlphaGo, which beat Le Sedol in 2016
- Provided new insights into the game

- Very recently, self-play reinforcement learning has been applied to the game of Go. AlphaGo Zero uses a single neural nework to predict winning probabily and actions to be taken, using raw board positions as inputs. Starting from random weights, the network is trained to gradually improve its predictions and match the results of an approximate (Monte Carlo) tree search algorithm.

Summary so far

- Parametrize evaluation functions using features
- TD learning: learn an evaluation function

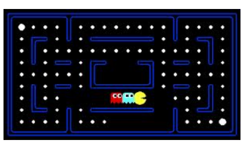
$$(\text{prediction}(\mathbf{w}) - \text{target})^2$$

Up next:

Turn-based → Simultaneous

Zero-sum → Non-zero-sum

Games: simultaneous games



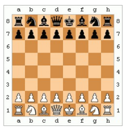
think and share

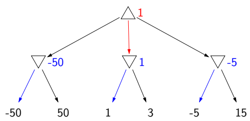
Question

For a simultaneous two-player zero-sum game (like rock-paper-scissors), can you still be optimal if you reveal your strategy?

- yes
- no

Turn-based games:





Simultaneous games:

?

- Game trees were our primary tool to model turn-based games. However, in simultaneous games, there is no ordering on the player's moves, so we need to develop new tools to model these games. Later, we will see that game trees will still be valuable in understanding simultaneous games.

Two-finger Morra

Example: two-finger Morra

Players A and B each show 1 or 2 fingers.
If both show 1, B gives A 2 dollars.
If both show 2, B gives A 4 dollars.
Otherwise, A gives B 3 dollars.

[play with a partner]

think and share

Question

What was the outcome?

player A chose 1, player B chose 1
player A chose 1, player B chose 2
player A chose 2, player B chose 1
player A chose 2, player B chose 2

Payoff matrix

Definition: single-move simultaneous game

Players = {A, B}

Actions: possible actions

$V(a,b)$: **A's utility** if A chooses action a, B chooses b

(let V be **payoff matrix**)

Example: two-finger Morra payoff matrix

A \ B	1 finger	2 fingers
1 finger	2	-3
2 fingers	-3	4

- In this lecture, we will consider only single move games. There are two players, A and B who both select from one of the available actions. The value or utility of the game is captured by a payoff matrix V whose dimensionality is $|\text{Actions}| \times |\text{Actions}|$. We will be analyzing everything from A's perspective, so entry $V(a,b)$ is the utility that A gets if he/she chooses action a and player B chooses b.

Strategies (policies)

Definition: pure strategy

A pure strategy is a single action:

$a \in$ Actions

Definition: mixed strategy

A mixed strategy is a probability distribution

$0 \le \pi(a) \le 1$ for $a \in$ Actions

Example: two-finger Morra strategies

Always 1: $\pi = [1, 0]$

Always 2: $\pi = [0, 1]$

Uniformly random: $\pi = [\frac{1}{2}, \frac{1}{2}]$

- Each player has a **strategy** (or a policy). A pure strategy (deterministic policy) is just a single action. Note that there's no notion of state since we are only considering single-move games.
- More generally, we will consider **mixed strategies** (randomized policy), which is a probability distribution over actions. We will represent a mixed strategy π by the vector of probabilities.

Game evaluation

Definition: game evaluation

The **value** of the game if player A follows π_A and player B follows π_B is

$$V(\pi_\text{A}, \pi_\text{B}) = \sum_{a,b} \pi_\text{A}(a)\pi_B(b)V(a,b)$$

Example: two-finger Morra

Player A always chooses 1: $\pi_\text{A} = [1, 0]$

Player B picks randomly: $\pi_\text{B} = [\frac{1}{2}, \frac{1}{2}]$

Value: $\boxed{-\frac{1}{2}}$

[whiteboard: matrix]

- Given a game (payoff matrix) and the strategies for the two players, we can define the value of the game.
- For pure strategies, the value of the game by definition is just reading out the appropriate entry from the payoff matrix.
- For mixed strategies, the value of the game (that is, the expected utility for player A) is gotten by summing over the possible actions that the players choose: $V(\pi_\text{A}, \pi_\text{B}) = \sum_{a \in \text{Actions}} \sum_{b \in \text{Actions}} \pi_\text{A}(a)\pi_\text{B}(b)V(a,b)$. We can also write this expression concisely using matrix-vector multiplications: $\pi_A^\top V \pi_B$.

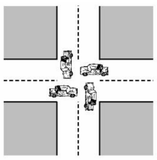
How to optimize?

Game value:

$$V(\pi_A, \pi_B)$$

Challenge: player A wants to maximize, player B wants to minimize...

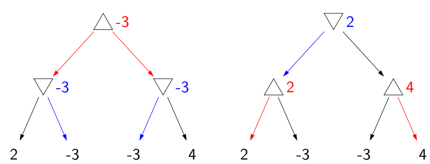
simultaneously



- Having established the values of fixed policies, let's try to optimize the policies themselves. Here, we run into a predicament: player A wants to maximize V but player B wants to minimize V **simultaneously**.
- Unlike turn-based games, we can't just consider one at a time. But let's consider the turn-based variant anyway to see where it leads us.

Pure strategies: who goes first?

Player A goes first: Player B goes first:



Proposition: going second is no worse

$$\max_a \min_b V(a,b) \le \min_b \max_a V(a,b)$$

- Let us first consider pure strategies, where each player just chooses one action. The game can be modeled by using the standard minimax game trees that we're used to.
- The main point is that if player A goes first, he gets -3, but if he goes second, he gets 2. In general, it's at least as good to go second, and often it is strictly better. This is intuitive, because seeing what the first player does gives more information.

Mixed strategies

Example: two-finger Morra

Player A reveals: $\pi_{\mathsf{A}} = [\frac{1}{2}, \frac{1}{2}]$

Value $V(\pi_{\mathsf{A}}, \pi_B) = \pi_B(1)(-\frac{1}{2}) + \pi_B(2)(+\frac{1}{2})$

Optimal strategy for player B is $\pi_B = [1, 0]$ **(pure!)**

Proposition: second player can play pure strategy

For any fixed mixed strategy π_{A}:

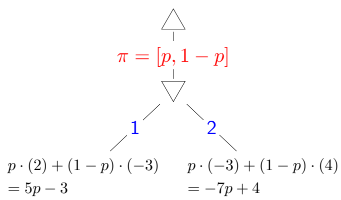
$$\min_{\pi_{\mathsf{B}}} V(\pi_{\mathsf{A}}, \pi_B)$$

can be attained by a pure strategy.

- Now let us consider mixed strategies. First, let's be clear on what playing a mixed strategy means. If player A chooses a mixed strategy, he reveals to player B the full probability distribution over actions, but importantly not a particular action (because that would be the same as choosing a pure strategy).
- As a warmup, suppose that player A reveals $\pi_{\mathsf{A}} = [\frac{1}{2}, \frac{1}{2}]$. If we plug this strategy into the definition for the value of the game, we will find that the value is a convex combination between $\frac{1}{2}(2) + \frac{1}{2}(-3) = -\frac{1}{2}$ and $\frac{1}{2}(-3) + \frac{1}{2}(4) = \frac{1}{2}$. The value of π_B that minimizes this value is $[1, 0]$. The important part is that this is a **pure strategy**.
- It turns out that no matter what the payoff matrix V is, as soon as π_{A} is fixed, then the optimal choice for π_{B} is a pure strategy. Ths is useful because it will allow us to analyze games with mixed strategies more easily.

Mixed strategies

Player A first reveals his/her mixed strategy



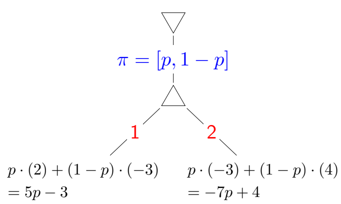
Minimax value of game:

$$\max_{0 \le p \le 1} \min\{5p - 3, -7p + 4\} = \boxed{-\frac{1}{12}} \text{ (with } p = \tfrac{7}{12}\text{)}$$

- Now let us try to draw the minimax game tree where the player A first chooses a mixed strategy, and then player B chooses a pure strategy.
- There are an uncountably infinite number of mixed strategies for player A, but we can summarize all of these actions by writing a single action template $\pi = [p, 1-p]$.
- Given player A's action, we can compute the value if player B either chooses 1 or 2. For example, if player B chooses 1, then the value of the game is $5p - 3$ (with probability p, player A chooses 1 and the value is 2; with probability $1-p$ the value is -3). If player B chooses action 2, then the value of the game is $-7p + 4$.
- The value of the min node is $F(p) = \min\{5p - 3, -7p + 4\}$. The value of the max node (and thus the minimax value of the game) is $\max_{0 \le 1 \le p} F(p)$.
- What is the best strategy for player A then? We just have to find the p that maximizes $F(p)$, which is the minimum over two linear functions of p. If we plot this function, we will see that the maximum of $F(p)$ is attained when $5p - 3 = -7p + 4$, which is when $p = \frac{7}{12}$. Plugging that value of p back in yields $F(p) = -\frac{1}{12}$, the minimax value of the game if player A goes first and is allowed to choose a mixed strategy.
- Note that if player A decides on $p = \frac{7}{12}$, it doesn't matter whether player B chooses 1 or 2; the payoff will be the same: $-\frac{1}{12}$. This also means that whatever mixed strategy (over 1 and 2) player B plays, the payoff would also be $-\frac{1}{12}$.

Mixed strategies

Player B first reveals his/her mixed strategy



Minimax value of game:

$$\min_{p \in [0,1]} \max\{5p - 3, -7p + 4\} = \boxed{-\frac{1}{12}} \text{ (with } p = \tfrac{7}{12}\text{)}$$

- Now let us consider the case where player B chooses a mixed strategy $\pi = [p, 1-p]$ first. If we perform the analogous calculations, we'll find that we get that the minimax value of the game is exactly the same ($-\frac{1}{12}$)!
- Recall that for pure strategies, there was a gap between going first and going second, but here, we see that for mixed strategies, there is no such gap, at least in this example.
- Here, we have been computed minimax values in the conceptually same manner as we were doing it for turn-based games. The only difference is that our actions are mixed strategies (represented by a probability distribution) rather than discrete choices. We therefore introduce a variable (e.g., p) to represent the actual distribution, and any game value that we compute below that variable is a function of p rather than a specific number.

General theorem

Theorem: minimax theorem [von Neumann, 1928]

For every simultaneous two-player zero-sum game with a finite number of actions:

$$\max_{\pi_A} \min_{\pi_B} V(\pi_A, \pi_B) = \min_{\pi_B} \max_{\pi_A} V(\pi_A, \pi_B),$$

where π_A, π_B range over **mixed strategies**.

Upshot: revealing your optimal mixed strategy doesn't hurt you!

Proof: linear programming duality

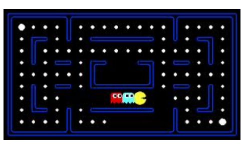
Algorithm: compute policies using linear programming

- It turns out that having no gap is not a coincidence, and is actually one of the most celebrated mathematical results: the von Neumann minimax theorem. The theorem states that for any simultaneous two-player zero-sum game with a finite set of actions (like the ones we've been considering), we can just swap the min and the max: it doesn't matter which player reveals his/her strategy first, as long as their strategy is optimal. This is significant because we were stressing out about how to analyze the game when two players play simultaneously, but now we find that both orderings of the players yield the same answer. It is important to remember that this statement is true only for mixed strategies, not for pure strategies.
- This theorem can be proved using linear programming duality, and policies can be computed also using linear programming. The sketch of the idea is as follows: recall that the optimal strategy for the second player is always deterministic, which means that the $\max_{\pi_A} \min_{\pi_B} \cdots$ turns into $\max_{\pi_A} \min_b \cdots$. The min is now over n actions, and can be rewritten as n linear constraints, yielding a linear program.
- As an aside, recall that we also had a minimax result for turn-based games, where the max and the min were over agent and opponent policies, which map states to actions. In that case, optimal policies were always deterministic because at each state, there is only one player choosing.

Summary

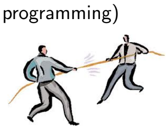
- Challenge: deal with simultaneous min/max moves
- Pure strategies: going second is better
- Mixed strategies: doesn't matter (von Neumann's minimax theorem)

Games: non-zero-sum games



Utility functions

Competitive games: minimax (linear programming)



Collaborative games: pure maximization (plain search)

Real life: ?

- So far, we have focused on competitive games, where the utility of one player is the exact opposite of the utility of the other. The minimax principle is the appropriate tool for modeling these scenarios.
- On the other extreme, we have collaborative games, where the two players have the same utility function. This case is less interesting, because we are just doing pure maximization (e.g., finding the largest element in the payoff matrix or performing search).
- In many practical real life scenarios, games are somewhere in between pure competition and pure collaboration. This is where things get interesting...

Prisoner's dilemma

Example: Prisoner's dilemma

Prosecutor asks A and B individually if each will testify against the other.

If both testify, then both are sentenced to 5 years in jail.

If both refuse, then both are sentenced to 1 year in jail.

If only one testifies, then he/she gets out for free; the other gets a 10-year sentence.

[play with a partner]

think and share

Question

What was the outcome?

- player A testified, player B testified
- player A refused, player B testified
- player A testified, player B refused
- player A refused, player B refused

Prisoner's dilemma

Example: payoff matrix

B \ A	testify	refuse
testify	$A = -5, B = -5$	$A = -10, B = 0$
refuse	$A = 0, B = -10$	$A = -1, B = -1$

Definition: payoff matrix

Let $V_p(\pi_{\mathsf{A}}, \pi_{\mathsf{B}})$ be the utility for player p.

- In the prisoner's dilemma, the players get both penalized only a little bit if they both refuse to testify, but if one of them defects, then the other will get penalized a huge amount. So in practice, what tends to happen is that both will testify and both get sentenced to 5 years, which is clearly worse than if they both had cooperated.

Nash equilibrium

Can't apply von Neumann's minimax theorem (not zero-sum), but get something weaker:

Definition: Nash equilibrium

A **Nash equilibrium** is $(\pi_{\mathsf{A}}^*, \pi_{\mathsf{B}}^*)$ such that no player has an incentive to change his/her strategy:

$V_{\mathsf{A}}(\pi_{\mathsf{A}}^*, \pi_B^*) \geq V_{\mathsf{A}}(\pi_{\mathsf{A}}, \pi_{\mathsf{B}}^*)$ for all π_A

$V_{\mathsf{B}}(\pi_{\mathsf{A}}^*, \pi_B^*) \geq V_{\mathsf{B}}(\pi_{\mathsf{A}}^*, \pi_{\mathsf{B}})$ for all π_B

Theorem: Nash's existence theorem [1950]

In any finite-player game with finite number of actions, there exists **at least one** Nash equilibrium.

- Since we no longer have a zero-sum game, we cannot apply the minimax theorem, but we can still get a weaker result.
- A Nash equilibrium is kind of a state point, where no player has an incentive to change his/her policy unilaterally. Another major result in game theory is Nash's existence theorem, which states that any game with a finite number of players (importantly, not necessarily zero-sum) has at least one Nash equilibrium (a stable point). It turns out that finding one is hard, but we can be sure that one exists.

Examples of Nash equilibria

Example: Two-finger Morra

Nash equilibrium: A and B both play $\pi = [\frac{7}{12}, \frac{5}{12}]$.

Example: Collaborative two-finger Morra

Two Nash equilibria:

- A and B both play 1 (value is 2).
- A and B both play 2 (value is 4).

Example: Prisoner's dilemma

Nash equilibrium: A and B both testify.

- Here are three examples of Nash equilibria. The minimax strategies for zero-sum are also equilibria (and they are global optima).
- For purely collaborative games, the equilibria are simply the entries of the payoff matrix for which no other entry in the row or column are larger. There are often multiple local optima here.
- In the Prisoner's dilemma, the Nash equilibrium is when both players testify. This is of course not the highest possible reward, but it is stable in the sense that neither player would want to change his/her strategy. If both players had refused, then one of the players could testify to improve his/her payoff (from -1 to 0).

Summary so far

Simultaneous zero-sum games:

- von Neumann's minimax theorem
- Multiple minimax strategies, single game value

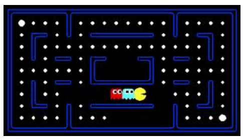
Simultaneous non-zero-sum games:

- Nash's existence theorem
- Multiple Nash equilibria, multiple game values

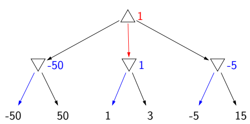
Huge literature in game theory / economics

- For simultaneous zero-sum games, all minimax strategies have the same game value (and thus it makes sense to talk about the value of a game). For non-zero-sum games, different Nash equilibria could have different game values (for example, consider the collaborative version of two-finger Morra).

Games: recap



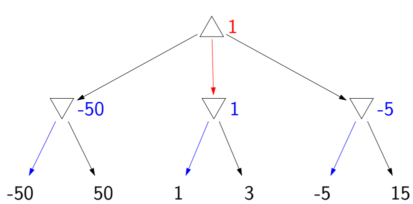
Summary



- Game trees: model opponents, randomness
- Minimax: find optimal policy against an adversary
- Evaluation functions: domain-specific, approximate
- Alpha-beta pruning: domain-general, exact

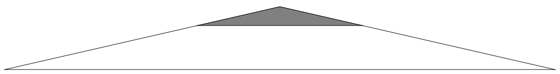
Review: minimax

agent (max) versus opponent (min)



- Recall that the central object of study is the game tree. Game play starts at the root (starting state) and descends to a leaf (end state), where at each node s (state), the player whose turn it is (Player(s)) chooses an action $a \in$ Actions(s), which leads to one of the children Succ(s, a).
- The **minimax principle** provides one way for the agent (your computer program) to compute a pair of minimax policies for both the agent and the opponent $(\pi^*_{\text{agent}}, \pi^*_{\text{opp}})$.
- For each node s, we have the minimax value of the game $V_{\text{minmax}}(s)$, representing the expected utility if both the agent and the opponent play optimally. Each node where it's the agent's turn is a max node (right-side up triangle), and its value is the maximum over the children's values. Each node where it's the opponent's turn is a min node (upside-down triangle), and its value is the minimum over the children's values.
- Important properties of the minimax policies: The agent can only decrease the game value (do worse) by changing his/her strategy, and the opponent can only increase the game value (do worse) by changing his/her strategy.

Review: depth-limited search



$$V_{\text{minmax}}(s, d) = \begin{cases} \text{Utility}(s) & \text{IsEnd}(s) \\ \text{Eval}(s) & d = 0 \\ \max_{a \in \text{Actions}(s)} V_{\text{minmax}}(\text{Succ}(s, a), d) & \text{Player}(s) = \text{agent} \\ \min_{a \in \text{Actions}(s)} V_{\text{minmax}}(\text{Succ}(s, a), d - 1) & \text{Player}(s) = \text{opp} \end{cases}$$

Use: at state s, choose action resulting in $V_{\text{minmax}}(s, d_{\text{max}})$

- In order to approximately compute the minimax value, we used a **depth-limited search**, where we compute $V_{\text{minmax}}(s, d_{\text{max}})$, the approximate value of s if we are only allowed to search to at most depth d_{max}.
- Each time we hit $d = 0$, we invoke an evaluation function $\text{Eval}(s)$, which provides a fast reflex way to assess the value of the game at state s.

Summary

- Main challenge: not just one objective
- Minimax principle: guard against adversary in turn-based games
- Simultaneous non-zero-sum games: mixed strategies, Nash equilibria
- Strategy: **search game tree + learned evaluation function**

- Games are an extraordinary rich topic of study, and we have only seen the tip of the iceberg. Beyond simultaneous non-zero-sum games, which are already complex, there are also games involving partial information (e.g., poker).
- But even if we just focus on two-player zero-sum games, things are quite interesting. To build a good game-playing agent involves integrating the two main thrusts of AI: search and learning, which are really symbiotic. We can't possibly search an exponentially large number of possible futures, which means we fall back to an evaluation function. But in order to learn an evaluation function, we need to search over enough possible futures to build an accurate model of the likely outcome of the game.

Chess

1997: IBM's Deep Blue defeated world champion Gary Kasparov

Fast computers:

- Alpha-beta search over 30 billion positions, depth 14
- Singular extensions up to depth 20

Domain knowledge:

- Evaluation function: 8000 features
- 4000 "opening book" moves, all endgames with 5 pieces
- 700,000 grandmaster games
- Null move heuristic: opponent gets to move twice

Checkers

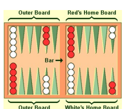
1990: Jonathan Schaeffer's **Chinook** defeated human champion; ran on standard PC

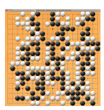
Closure:

- 2007: Checkers solved in the minimax sense (outcome is draw), but doesn't mean you can't win
- Alpha-beta search + 39 trillion endgame positions

Backgammon and Go

Alpha-beta search isn't enough...





Challenge: large branching factor

- Backgammon: randomness from dice (can't prune!)
- Go: large board size (361 positions)

Solution: learning

- For games such as checkers and chess with a manageable branching factor, one can rely heavily on minimax search along with alpha-beta pruning and a lot of computation power. A good amount of domain knowledge can be employed as to attain or surpass human-level performance.
- However, games such as Backgammon and Go require more due to the large branching factor. Backgammon does not intrinsically have a larger branching factor, but much of this branching is due to the randomness from the dice, which cannot be pruned (it doesn't make sense to talk about the most promising dice move).
- As a result, programs for these games have relied a lot on TD learning to produce good evaluation functions without searching the entire space.

AlphaGo

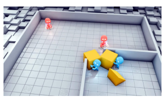
- Supervised learning: on human games
- Reinforcement learning: on self-play games
- Evaluation function: convolutional neural network (value network)
- Policy: convolutional neural network (policy network)
- Monte Carlo Tree Search: search / lookahead

- The most recent visible advance in game playing was March 2016, when Google DeepMind's AlphaGo program defeated Le Sedol, one of the best professional Go players 4-1.
- AlphaGo took the best ideas from game playing and machine learning. DeepMind executed these ideas well with lots of computational resources, but these ideas should already be familiar to you.
- The learning algorithm consisted of two phases: a supervised learning phase, where a policy was trained on games played by humans (30 million positions) from the KGS Go server; and a reinforcement learning phase, where the algorithm played itself in attempt to improve, similar to what we say with Backgammon.
- The model consists of two pieces: a value network, which is used to evaluate board positions (the evaluation function); and a policy network, which predicts which move to make from any given board position (the policy). Both are based on convolutional neural networks.
- Finally, the policy network is not used directly to select a move, but rather to guide the search over possible moves in an algorithm similar to Monte Carlo Tree Search.

Coordination games

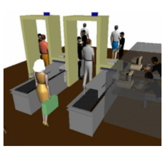
Hanabi: players need to signal to each other and coordinate in a decentralized fashion to collaboratively win.

Hide-and-Seek: OpenAI has developed agents with emergent behaviors to play hide and seek.



Other games

Security games: allocate limited resources to protect a valuable target. Used by TSA security, Coast Guard, protect wildlife against poachers, etc.

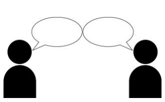


- The techniques that we've developed for game playing go far beyond recreational uses. Whenever there are multiple parties involved with conflicting interests, game theory can be employed to model the situation.
- For example, in a security game a defender needs to protect a valuable target from a malicious attacker. Game theory can be used to model these scenarios and devise optimal (randomized) strategies. Some of these techniques are used by TSA security at airports, to schedule patrol routes by the Coast Guard, and even to protect wildlife from poachers.

Other games

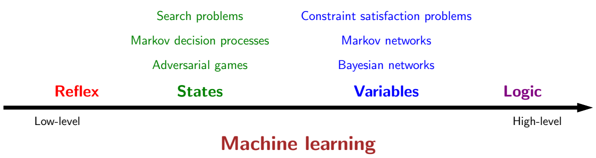
Resource allocation: users share a resource (e.g., network bandwidth); selfish interests leads to volunteer's dilemma

Language: people have speaking and listening strategies, mostly collaborative, applied to dialog systems



- For example, in resource allocation, we might have n people wanting to access some Internet resource. If all of them access the resource, then all of them suffer because of congestion. Suppose that if $n-1$ connect, then those people can access the resource and are happy, but the one person left out suffers. Who should volunteer to step out (this is the volunteer's dilemma)?
- Another interesting application is modeling communication. There are two players, the speaker and the listener, and the speaker's actions are to choose what words to use to convey a message. Usually, it's a collaborative game where utility is high when communication is successful and efficient. These game-theoretic techniques have been applied to building dialog systems.

Course plan

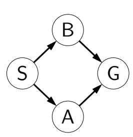


State-based models

[Modeling]		
Framework	search problems	MDPs/games
Objective	minimum cost paths	maximum value policies
[Inference]		
Tree-based	backtracking	minimax/expectimax
Graph-based	DP, UCS, A*	value/policy iteration
[Learning]		
Methods	structured perceptron	Q-learning, TD learning

- **Modeling**: In the context of state-based models, we seek to find minimum cost paths (for search problems) or maximum value policies (for MDPs and games).
- **Inference**: To compute these solutions, we can either work on the search/game tree or on the state graph. In the former case, we end up with recursive procedures which take exponential time but require very little memory (generally linear in the size of the solution). In the latter case, where we are fortunate to have few enough states to fit into memory, we can work directly on the graph, which can often yield an exponential savings in time.
- Given that we can find the optimal solution with respect to a fixed model, the final question is where this model actually comes from. **Learning** provides the answer: from data. You should think of machine learning as not just a way to do binary classification, but more as a way of life, which can be used to support a variety of different models.
- In the rest of the course, modeling, inference, and learning will continue to be the three pillars of all techniques we will develop.

State-based models: takeaway 1

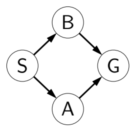


Key idea: specify locally, optimize globally

Modeling: specifies local interactions
Inference: find globally optimal solutions

- One high-level takeaway is the motto: specify locally, optimize globally. When we're building a search problem, we only need to specify how the states are connected through actions and what the local action costs are; we need not specify the long-term consequences of taking an action. It is the job of the inference to take all of this local information into account and produce globally optimal solutions (minimum cost paths).
- This separation is quite powerful in light of modeling and inference: having to worry only about local interactions makes modeling easier, but we still get the benefits of a globally optimal solution via inference which are constructed independent of the domain-specific details.
- We will see this local specification + global optimization pattern again in the context of variable-based models.

State-based models: takeaway 2



Key idea: state

A **state** is a summary of all the past actions sufficient to choose future actions **optimally**.

Mindset: move through states (nodes) via actions (edges)

- The second high-level takeaway which is core to state-based models is the notion of **state**. The state, which summarizes previous actions, is one of the key tools that allows us to manage the exponential search problems frequently encountered in AI.