Recap: Minimax Trees

Minimax Search

\[\text{function } \text{Max-Value}(\text{state}) \text{ returns a utility value}\]
\[\text{if } \text{Terminal-Test}(\text{state}) \text{ then return } \text{Utility}(\text{state})\]
\[s = -\infty\]
\[\text{for } a \in \text{Successors}(\text{state}) \text{ do } s = \max(s, \text{Min-Value}(a))\]
\[\text{return } s\]

\[\text{function } \text{Min-Value}(\text{state}) \text{ returns a utility value}\]
\[\text{if } \text{Terminal-Test}(\text{state}) \text{ then return } \text{Utility}(\text{state})\]
\[s = \infty\]
\[\text{for } a \in \text{Successors}(\text{state}) \text{ do } s = \min(s, \text{Max-Value}(a))\]
\[\text{return } s\]

α-β Pruning Example

[Code in book]

α-β Pruning

- **General configuration**
  - \(\alpha\) is the best value (to MAX) found so far off the current path
  - If \(V\) is worse than \(\alpha\), MAX will avoid it, so prune \(V\)'s branch
  - Define \(\beta\) similarly for MIN
### α-β Pruning Properties

- Pruning has **no effect** on final result
- Good move ordering improves effectiveness of pruning
- With “perfect ordering”:
  - Time complexity drops to $O(b^{m/2})$
  - Doubles solvable depth
  - Full search, e.g. chess, is still hopeless!
- A simple example of metareasoning, here reasoning about which computations are relevant

### Resource Limits

- Cannot search to leaves
- Limited search
  - Instead, search a limited portion of the tree
  - Replace terminal utilities with an eval function for non-terminal positions
  - Guarantee of optimal play is gone
- Example:
  - Suppose we have 100 seconds, can explore 10K nodes / sec
  - So can check 1M nodes per move
  - α-β reaches about depth 8 – decent chess program

### Evaluation Functions

- Function which scores non-terminals
- Ideal function: returns the utility of the position
- In practice: typically weighted linear sum of features:
  
  $$Eval(s) = w_1 f_1(s) + w_2 f_2(s) + \ldots + w_n f_n(s)$$

  - e.g. $f_1(s) = \text{(num white queens} - \text{num black queens)}, \text{etc.}$

### Function Approximation

- Problem: inefficient to learn each state’s utility (or eval function) one by one
- Solution: what we learn about one state (or position) should generalize to similar states
  - Very much like supervised learning
  - If states are treated entirely independently, we can only learn on very small state spaces
  
### Linear Value Functions

- Another option: values are linear functions of features of states (or action-state pairs)
  
  $$V^\phi(s) = \sum f_i(s)$$

- Good if you can describe states well using a few features (e.g. for game playing board evaluations)
- Now we only have to learn a few weights rather than a value for each state

### Recap: Model-Free Learning

- Recall MDP value updates for a given estimate of $U$
  - If you know the model $T$, use Bellman update
  
  $$U(s) \leftarrow R(s) + \gamma \max_{a'} \sum_{s'} T(s' | s) U(s')$$

- Temporal difference learning (TD)
  - Make (epsilon greedy) action choice (or follow a provided policy)
  
  $$\pi(s) = \arg \max_a \sum_{s'} U(s' | s, a)$$

  - Update using results of the action
  
  $$U(s) \leftarrow (1 - \alpha) U(s) + \alpha \left[ R(s) + \gamma U(s') \right]$$
Example: Tabular Value Updates

- **Example: Blackjack**
  - +1 for win, -1 for loss or bust, 0 for tie
  - Our hand shows 14, current policy says “hit”
  - We hit, get an 8, bust (end up in s' = “lose”)

- **Update**
  - Old U(s) = 0.5
  - Observed R(s) = 0
  - Old U(s') = -1
  - New U(s) = U(s) + α [γ (R(s) + U(s')) - U(s)]
  - If α = 0.1, γ = 1.0
  - New U(s) = 0.5 + 0.1 [0 + (-1) - 0.5] = 0.35

TD Updates: Linear Values

- **Assume a linear value function:**
  \[ U_q(s) = \sum_{f_k} \theta_k f_k(s) \]
- **Can almost do a TD update:**
  \[ U(s) \leftarrow U(s) + \alpha \left( R(s) + \gamma U(s') - U(s) \right) \]
  - Problem: we can’t “increment” U(s) explicitly
  - Solution: update the weights of the features at that state
  \[ \theta_k \leftarrow \theta_k + \alpha \left( R(s) + \gamma U(s') - U(s) \right) f_k(s) \]

Learning Eval Parameters with TD

- Ideally, want eval(s) to be the utility of s
- Idea: use techniques from reinforcement learning
  - Samuel’s 1959 checkers system
  - Tesauro’s 1992 backgammon system (TD-Gammon)
- Basic approach: temporal difference updates
  - Begin in state s
  - Choose action using limited minimax search
  - See what opponent does
  - End up in state s'
  - Do a value update of U(s) using U(s')
  - Not guaranteed to converge against an adversary, but can work in practice

Q-Learning

- With TD updates on values
  - You don’t need the model to update the utility estimates
  - You still do need it to figure out what action to take!
- Q-Learning with TD updates
  - No model needed to learn or to choose actions
  \[ Q_{t+1}(a, s) \leftarrow (1 - \alpha)Q_t(a, s) + \alpha \left( R(s) + \gamma \max_{a'} Q_t(a', s') \right) \]
  \[ \pi(s) = \arg \max_a Q(a, s) \]

TD Updates for Linear Qs

- Can use TD learning with linear Qs
  - (Actually it’s just like the perceptron!)
- Old Q-learning update:
  \[ Q(a, s) \leftarrow Q(a, s) + \alpha \left( R(s) + \gamma \max_{a'} Q(a', s') - Q(a, s) \right) \]
  - Simply update weights of features in Qθ(a,s)
  \[ \theta_k \leftarrow \theta_k + \alpha \left( R(s) + \gamma \max_{a'} Q_k(a', s') - Q_k(a, s) \right) f_k(a, s) \]

Coming Up

- Real-world applications
  - Large scale machine / reinforcement learning
  - NLP: language understanding and translation
  - Vision: object and face recognition