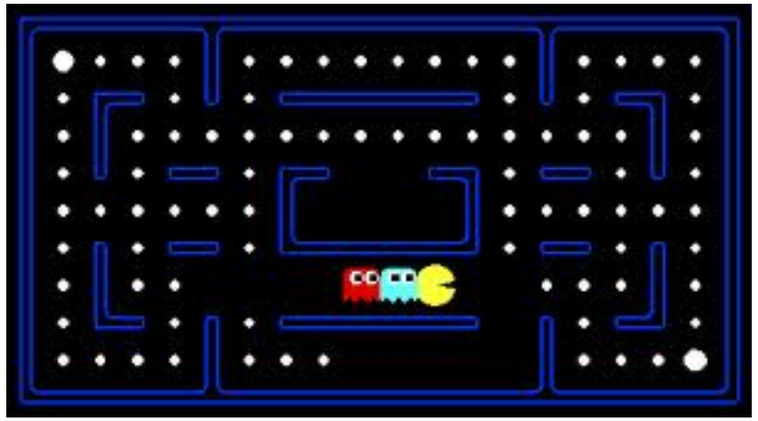
Games: TD-learning



Evaluation function

Old: hand-crafted

Example: chess

$$\text{Eval}(s) = \text{material} + \text{mobility} + \text{king-safety} + \text{center-control}$$

$$\text{material} = 10^{100}(K - K') + 9(Q - Q') + 5(R - R') + 3(B - B' + N - N') + 1(P - P')$$

$$\text{mobility} = 0.1(\text{num-legal-moves} - \text{num-legal-moves}')$$

...

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:

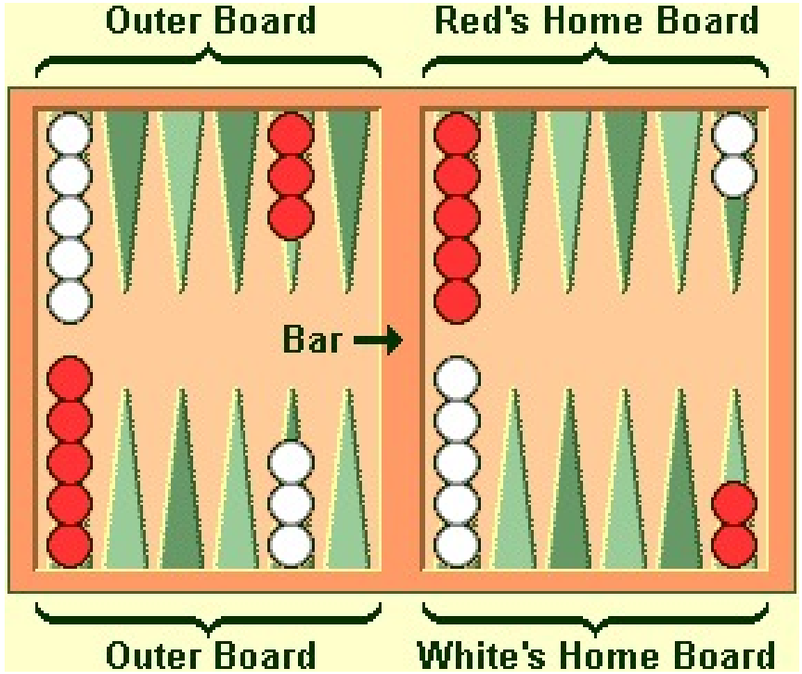
$$V(s; \mathbf{w}) = \mathbf{w} \cdot \phi(s)$$

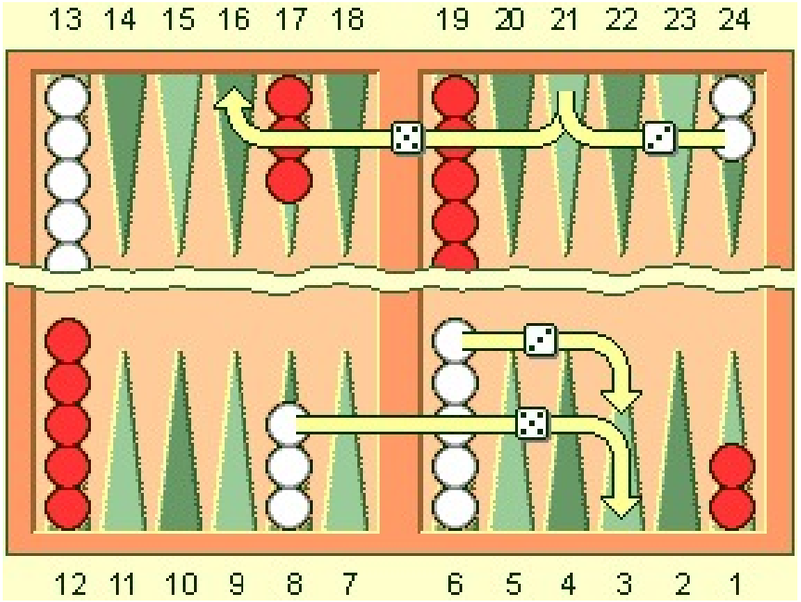
Neural network:

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

- 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

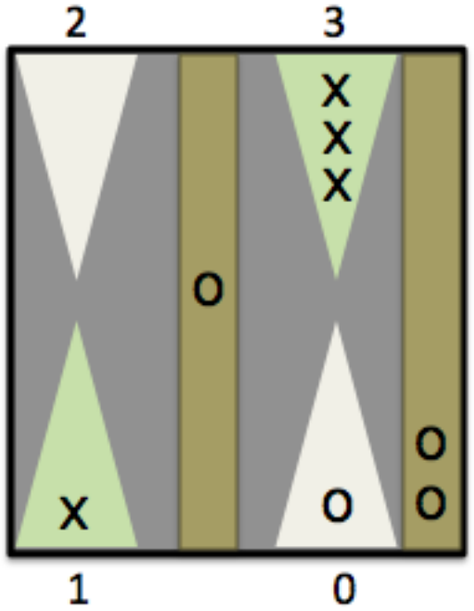




- As an example, let's consider the classic game of backgammon. Backgammon is a two-player game of strategy and chance in which the objective is to be the first to remove all your pieces from the board.
- The simplified version is that on your turn, you roll two dice, and choose two of your pieces to 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:

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

- 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 $\{1, \ldots, 6\}$ 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**
$\phi: \begin{pmatrix}0\\1\end{pmatrix}$		$\phi: \begin{pmatrix}1\\0\end{pmatrix}$		$\phi: \begin{pmatrix}1\\2\end{pmatrix}$		$\phi: \begin{pmatrix}1\\0\end{pmatrix}$
$\mathbf{w}: \begin{pmatrix}0\\0\end{pmatrix}$	p:0 t:0 p-t:0	$\mathbf{w}: \begin{pmatrix}0\\0\end{pmatrix}$	p:0 t:0 p-t:0	$\mathbf{w}: \begin{pmatrix}0\\0\end{pmatrix}$	p:0 t:1 p-t:-1	$\mathbf{w}: \begin{pmatrix}0.5\\1\end{pmatrix}$

S1	**r:0**	**S2**	**r:0**	**S6**	**r:0**	**S10**
$\phi: \begin{pmatrix}0\\1\end{pmatrix}$		$\phi: \begin{pmatrix}1\\0\end{pmatrix}$		$\phi: \begin{pmatrix}0\\0\end{pmatrix}$		$\phi: \begin{pmatrix}1\\0\end{pmatrix}$
$\mathbf{w}: \begin{pmatrix}0.5\\1\end{pmatrix}$	p:1 t:0.5 p-t:0.5	$\mathbf{w}: \begin{pmatrix}0.5\\0.75\end{pmatrix}$	p:0.5 t:0 p-t:0.5	$\mathbf{w}: \begin{pmatrix}0.25\\0.75\end{pmatrix}$	p:0 t:0.25 p-t:-0.25	$\mathbf{w}: \begin{pmatrix}0.25\\0.75\end{pmatrix}$

- 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:

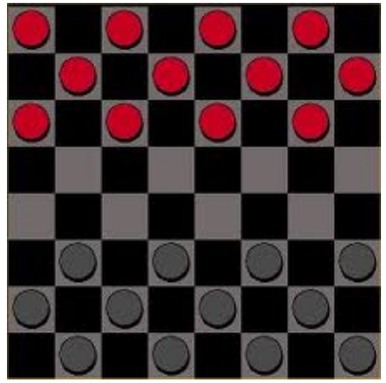
- Operate on $\hat{Q}_{\text{opt}}(s, a; \mathbf{w})$
- Off-policy: value is based on estimate of optimal policy
- To use, don't need to know MDP transitions $T(s, a, s')$

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

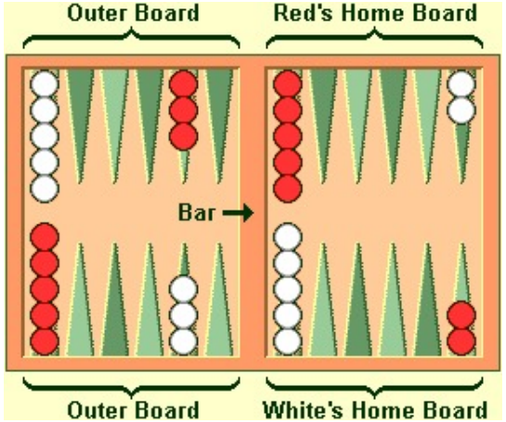


Arthur Samuel's checkers program [1959]:

- 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

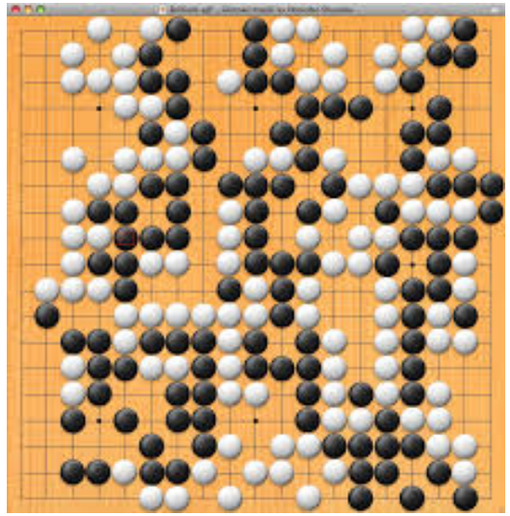


Gerald Tesauro's TD-Gammon [1992]:

- 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