Learning in games II

Jonathan Shapiro

School of Computer Science
University of Manchester

Announcements

1. Last chance to sign up for a project group is Friday (Oct 27, 2023) 6pm.
2. **Remember:** you do not need to sign up for a project group. I will assign you to a group arbitrarily.
3. Project groups must have 3 – 4 members
4. No teaching next week — Reading Week.

Good Resources

- A good book on this “Reinforcement Learning: An Introduction”, R. Sutton and A. Barto, MIT Press, 1998 (available on-line at `http://incompleteideas.net/sutton/book/bookdraft2017nov5.pdf`
- Another freely available book (via University Library) is “Learning to Play - Reinforcement Learning” by Aske Plaat, 2020. Available as an e-book here.
- Rich Sutton’s website `http://incompleteideas.net/sutton/publications.html` is full of relevant stuff.

Last time

Immediate reward reinforcement learning

1. At time t the agent is in state s_t;
2. It chooses and takes an action a_t from a set of actions $a_t \in A_{s_t}$;
3. It then received a reward r_t.

Goal:

1. To learn the expected reward for every action taken from every state;
2. to use that to find the action from every state which maximizes the expected reward.
3. By both *exploring* and *exploiting*.

Two methods — Method 1: Tabular approach

Tabular approach: Maintain a table $Q(s, a)$.

- The current estimate of the expected reward for taking action a from state s.
- *This requires that every state-action pair be visited many times.*

The Learning Equation

Uses a constant learning rate α between 0 and 1.

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha \left[r_t - Q(s_t, a_t) \right]$$

Tabular learning algorithm

```
1. Select $\epsilon$ value
2. Initialize $Q(s, a)$ for all $s$ and $a$; $s_1 \leftarrow$ start state; $t \leftarrow 1$.
3. Repeat
    3.1 Observe state $s_t$.
    3.2 Using epsilon greedy, choose either the best action
        according to the current $Q(s_t, a)$ or a random action.
    3.3 Observe reward $r_t$
    3.4 Update $Q(s_t, a_t)$ for the action $a_t$ taken from state $s_t$.

            $Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha \left[ r_t - Q(s_t, a_t) \right]$

    3.5 $t \leftarrow t + 1$
4. until there is no time left
```

The update equation could also use a decreasing learning rate $\alpha = \frac{1}{t}$.

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \frac{1}{t}\left[r_t - Q(s_t, a_t)\right]$$

After training — in use

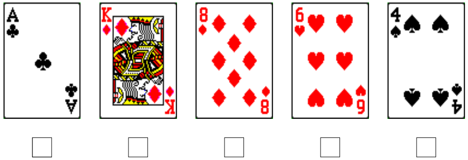
First train, then use[1].

- Observe the state.
- Determine all possible actions.
- Look in the table for the action which produces the highest reward.

A “greedy” policy.

[1] Reinforcement learning is very amenable to life-long learning

Example — video poker



Hand	Reward
Royal Flush	249
Straight Flush	49
Four of a kind	24
Full House	8
Flush	5
Straight	3
Three of a Kind	2
Two Pair	1
Jacks or Better	0
Nothing	-1
Theoretical Return	-0.0195

Too many states!

Number of states: $\binom{52}{5} = 2,598,960$

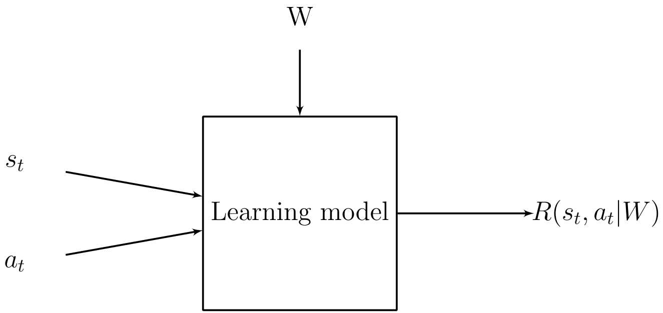
Two methods — method 2: function approximation

Function approximation: A supervised *regression*[2] model $R(s, a|\mathbf{W})$

- Uses a of learning parameters (e.g. “weights”) $\mathbf{W} = (w_1, \ldots, w_d)$.
- *This can perhaps generalize to unseen state-action pairs.*

[2] not classification

Function approximation



- Enter the current state, and each action (to find the best action).
- Take the chosen action; observe the reward.
- Train the model to produce the reward as output for the given inputs.

Function approximation using gradient descent

1. Initialize weights (e.g small, random values) and learning rate α (small and positive).
2. Repeat
   - Observe state s_t.
   - Using ϵ-greedy, choose either a random action, or what is the best action according to $R(s_t, a|W)$, $a_t = \text{argmax}_{a'} R(s_t, a'|\theta)$.
   - Observe reward r_t.
   - Update parameters to minimize the mean squared difference:
     $$[r_t - R(s, a|\mathbf{W})]^2.$$
   - $t \leftarrow t + 1$
3. Until out of time

Skip GD

Gradient descent learning of the weights

- One method to do an optimization like this is *gradient descent*.
- It is an iterative, local improvement algorithm, like stochastic hill-climbing, but using information about the most improving direction (minus the gradient in this case).

Aside on gradient descent

Suppose you have a function, g, of many variables
$\mathbf{x} = (x_1, x_2, \ldots, x_d)$.
The gradient of g, denoted Grad(g) or $\nabla g(\mathbf{x})$ is,

- A vector;
- With magnitude proportional to the steepness at that point
- With direction that points most uphill.

It is computed as a vector of partial derivatives, i.e. the *i*th component of the gradient is

$$\nabla_i = \frac{\partial}{\partial x_i} g(x_1, x_2, \ldots, x_k).$$

Suppose you want to find the values of **x** which make $g(\mathbf{x})$ as small as possible. From properties of the gradient, we know

$$x_i \leftarrow x_i - \alpha \nabla_i g(\mathbf{x})$$

is guaranteed to make g smaller, if the stepsize (or learning rate) α is small enough.

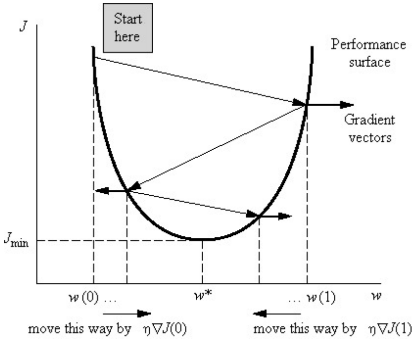


Figure: The gradient of a one-dimensional quadratic curve

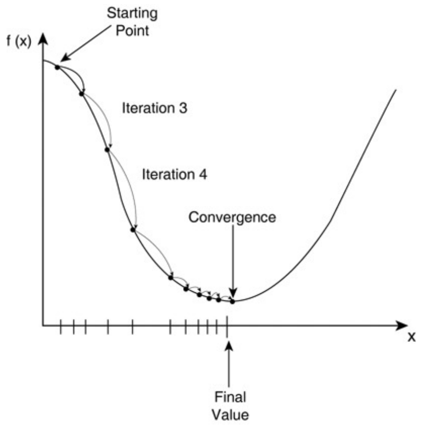


Figure: Gradient descent one a one-dimensional quadratic curve

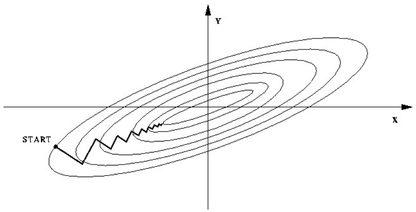


Figure: Gradient on a two-dimensional quadratic curve

Immediate reward RL not applicable to game learning

- In extensive form games, a sequence of actions is required to receive a reward, but values are required for all decision nodes.
- How do you associate individual moves with final performance?

Called *learning with delayed rewards.*

Tic-Tac-Toe example

(on visualizer)

Reinforcement learning for games

(and other delayed-reward problems.)

- $\times$ Play to the end of the game, then back up results to the moves made.
- ✓ Learn to predict expected *future rewards*
  - using one-step look ahead.

Temporal-difference (TD) learning

The value of a state/move

The value of a state $V(s)$: current reward plus predicted future reward from that state using a given policy from now on.

The value of an action from a state $Q(s, a)$: current plus predicted future reward making that move from that state using a given policy from now on.

“Rewards” $=$ expected rewards in a stochastic environment.

The (discounted) future reward

Future reward: at time t, the future reward is,

$$r_t + r_{t+1} + r_{t+2} + \cdots = \sum_{t'=t}^{\text{end}} r_{t'}.$$

Discounted future reward: the longer you wait, the less valued is the reward (by factor $0 \leq \gamma \leq 1$),

$$r_t + \gamma r_{t+1} + \gamma^2 r_{t+2} + \cdots = \sum_{t'=t}^{\text{end}} r_{t'} \gamma^{t'-t}$$

Rewards become less valuable the longer you have to wait for them.

Why discount rewards

- Encourage shorter paths.
- The longer the path; more likely errors, changes, etc.
- The longer you wait to achieve the reward, the less it is worth

Learning the future reward

Value of a non-terminal node will be the expected future reward, discounted by how long it takes to get it.

$$\begin{aligned}
V(s_t) &= \text{ The value of node } s \text{ encountered at time } t\ ; && (1)\\
&= \sum_{t'=t}^{\text{game end}} r_{t'}\gamma^{t'-t};\ \text{averaged over many games} \\
&= r_t + \gamma \sum_{t'=t+1}^{\text{game end}} r_{t'}\gamma^{t'-(t+1)}; \\
&= r_t + \gamma V(s_{t+1}); \\
&\boxed{V(s_t) = r_t + \gamma V(s_{t+1})} && (2)
\end{aligned}$$

- However, *we don't know the future reward*.
- We estimate it by the value of the best next state using our current estimate of $V(s_t)$.
- Note, early in learning this will be rubbish; later in learning it could be a reasonable estimate.

The future rewards depend on the policy we are using

We will assume a “greedy” policy

- No exploration; pure exploitation.
- During learning we use an exploring strategy(e.g. epsilon greedy).
- After learning is over and in use, we use a purely exploiting strategy.

Called an *off-policy* approach.

Tabular Q-learning

The *Q*-table predicts:

- The immediate reward,
- Plus the expected (discounted) reward assuming the best actions are taken from this point onward.

The *Q*-function should be:

$$\begin{aligned} Q(s_t, a_t) = &\text{ the immediate reward} \\ &+ \text{ plus the estimate of the future rewards} \\ &= r_t + \gamma \max_{a'} Q(s_{t+1}, a'). \end{aligned}$$

The Q-learning algorithm (general case)

Repeat

- Observe current state s_t
- Choose action using a policy derived from Q as before.

  Exploiting: $a_t = \text{argmax}_{a'} Q(s_t, a')$
  Exploring: $a_t =$ random action.
- Observe new state s_{t+1} and any reward r_t.
- Update Q according to

$$\begin{aligned} Q(s_t, a_t) \leftarrow\ & Q(s_t, a_t) \\ & + \alpha \left[r_t + \gamma \max_{a'} Q(s_{t+1}, a') - Q(s_t, a_t) \right] \end{aligned}$$

The Q-learning equation

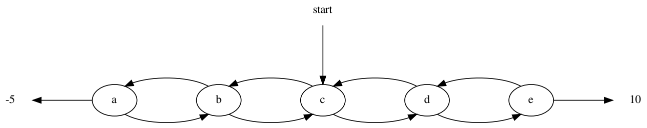
Update Q according to

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha \left[\underbrace{r_t}_{\text{current reward}} + \overbrace{\gamma \max_{a'} Q(s_{t+1}, a')}^{\text{predicted discounted future reward}} - Q(s_t, a_t) \right] \tag{3}$$

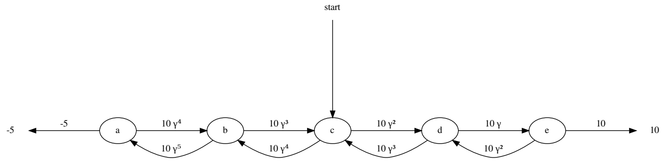
What does Q-learning learn?

$Q(s, a)$ converges to

- the expected future reward one gets taking action a from state s,
- discounted by the time to get it,
- assuming the best action is chosen from this point forward.



What are the values of the Q-table after infinite training?
E.g. $Q(c, \text{right})$?



It is γ raised to the number of steps to achieve the reward, *assuming optimal steps taken going forward*

Tabular Q-learning is proven to converge to the correct answer if every state is visited, and every action is used an infinite number of times. I.e. via learning,

Applications to games

Q-learning in a game setting works the same way, except:

1. When a player makes a move, the updated state is returned only after the opponent makes a move.
2. When either player gets a reward, the opponent gets minus that reward.
    - which is associated with the last move each made.

- We will have two learning agents playing games against each other many many times.
- Only after the learning phases is over will can the agents be used to play against real players.

Q-learning with function approximation

- As before, there will be too many nodes in a typical game tree.
- As before, we define a parametrized approximation function,

$$R(s, a|\mathbf{W}), \text{ where } \mathbf{W} \text{ is a vector of parameters.}$$

- Then use gradient descent to minimize

$$\left[r_t + \gamma \max_{a'} R(s_{t+1}, a'|\mathbf{W}) - R(s_t, a_t|\mathbf{W})\right]^2,$$

Skip TD

Problems

1. Q-learning can still be too slow. A heuristic speed up is often used.
2. Called TD(λ).

TD(λ) learning

- TD(0)-learning with a look-up table is provable correct, but very slow.
- Initially only the state which led immediately to a reward has its value updated. Only through many sequences does learning work its way backwards toward the initial states.
- Why not update the value of every state in the sequence which led to the reward?

TD(λ) learning

Idea: When reward received, assign credit (or blame) to the action

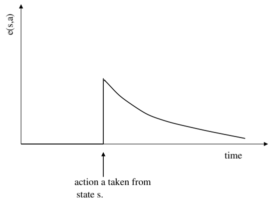
1. just previous with a weight of 1,
2. the one before that with a weight of $\lambda \times \gamma$,
3. the one before that with a weight of $(\lambda \times \gamma)^2$, etc..

Here λ is a parameter between 0 and 1.

Eligibility Traces

An efficient way of accounting for states in the learning sequence.
Let $e(n)$ denote the "eligibility" of node n. This is related to how recently in the sequence node n was visited.

$$e(n) \leftarrow \begin{cases} e(n)+1; & \text{if node } n \text{ is visited} \\ \lambda\gamma e(n); & \text{otherwise} \end{cases}$$



Skip TD Skip TD

TD(λ) Rule — lookup table version

```
Initialize $V(n)$ arbitrarily.
Repeat (for each game)
  ▶ Initialize $n$ = root and $e(n) \leftarrow 0$ for all nodes $n$. Repeat
      ▶ If current node is a terminal node
          ▶ Observe payoff $U(n)$
          ▶ $V(n) \leftarrow V(n) + \alpha\,[U(n) - V(n)]$
          ▶ $n \leftarrow$ root
      ▶ Choose child node $n'$ from current node $n$ using policy
        derived from $V$ (e.g. $\epsilon$-greedy)
      ▶ Observe payoff $U(n)$
      ▶ $\delta \leftarrow U(n') + \gamma V(n') - V(n)$
      ▶ $e(n) \leftarrow e(n) + 1$;
      ▶ For all nodes $n''$
          ▶ $V(n'') \leftarrow V(n'') + \alpha\delta e(n'')$
          ▶ $e(n'') \leftarrow \gamma\lambda e(n'')$
      ▶ $n \leftarrow n'$ or if $n'$ is terminal, go back to root.
  ▶ Until end of sequence
```

Application: TD-Gammon

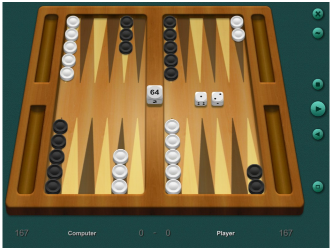
Tesauro, 1994

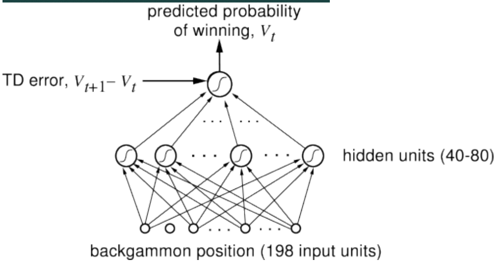
TD(λ) learning has been used to produce a very good backgammon playing program. **It produces the board evaluation function, $V(s)$.**

Predictor Network: A multi-layer perceptron is used to predict the outcome of the game from the current board position (originally coded as raw board position; later hand-crafted features used).

Controller: A program generates all legal moves from the current position. Predictor network scores them; that with the highest predicted outcome (J) is the moved used.

TD-gammon (cont)





predicted probability of winning, V_t
TD error, $V_{t+1} - V_t$
hidden units (40-80)
backgammon position (198 input units)

Training: Data consists of sequences of moves from the standard start position to a result. Program plays itself over and over.

1. Network starts from random state (no knowledge of game)
2. Initially, games incredibly long (100's to 1000's of moves versus 50 – 60 moves for typical human game).
3. Later, basic strategies emerge: hitting opponents, building points, playing safe, ...

Results: TD-Gammon 1.0 Contained 40 hidden units, trained for 200,000 games. Plays at strong intermediate level; good enough to win regional tournaments.

Results: TD-Gammon 2.1 Close to the world's best player.

Has changed how humans play certain board positions (discovered new strategies which appear to be better).

TD-Gammon (cont)

"TD-Gammon has definitely come into its own. There is no question in my mind that its positional judgment is far better than mine. Only on small technical areas can I claim a definite advantage over it . . . . I find a comparison of TD-Gammon and the high-level chess computers fascinating. The chess computers are tremendous in tactical positions where variations can be calculated out. Their weakness is in vague positional games, where it is not obvious what is going on . . . . TD-Gammon is just the opposite. Its strength is in the vague positional battles where judgment, not calculation , is the key. There, it has a definite edge over humans . . . . In particular, its judgment on bold vs. safe play decisions, which is what backgammon really is all about, is nothing short of phenomenal . . . . Instead of a dumb machine which can calculate things much faster than humans such as the chess playing computers, you have built a smart machine which learns from experience pretty much the same way that humans do"

Kit Woolsey, perennial world top 10 backgammon player (ranked #3 when this was written), quoted in Tesauro, 1995.

More on TD-gammon

TD-gammon is no more, but apparently the current, strong computer backgammon players are based on TD(λ): Snowie, Jellyfish, Bgblitz, GNUbg.
Why did it work? (Theories)

1. Playing regime — TD-gammon learned via self-play. Pollack and Blair (1997) argue that this was key, by showing a simpler model learned effectively in self-play. The randomness in the game (the dice) means that exploration happens automatically.
2. Board representation — use of hand-crafted features improved performance considerably.

Other Applications

- Applied in many control applications.
- Dominant paradigm in animal learning. Explains many puzzles of classical conditioning, how ants find shortest paths and how rats navigate.

Conclusions

- For delayed-reward problems, use expected future reward or expected cumulative discounted reward as the value.
- This can be learned using Q-learning by learning at each decision. The current estimate of the value function is used to predict the future rewards.
- TD-lambda can be used to learn a board evaluation function. It produces a world-class backgammon player.