Reinforcement Learning

CS4881 Artificial Intelligence, SE3250 Game Design, CS498 Machine Learning

Jay Urbain, PhD

Credits: *Machine Learning, Tom Mitchell*

AIMA, Russell and Norvig

Kardi Teknomo, Q-Learning Using Matlab

Reinforcement Learning (RL)

- **RL** is concerned with how an ***agent*** ought to take ***actions*** in an ***environment*** so as to ***maximize*** some notion of ***cumulative reward.***
- ***Learning by interacting with your environment*** is a foundational idea underlying nearly all theories of learning and intelligence.

Figure 2: Balance Board Control

Babies Take the Wheel of Driving Robots
Giving disabled babies the power of movement could improve their development.
http://www.technologyreview.com/blog/mimssbits/25667/

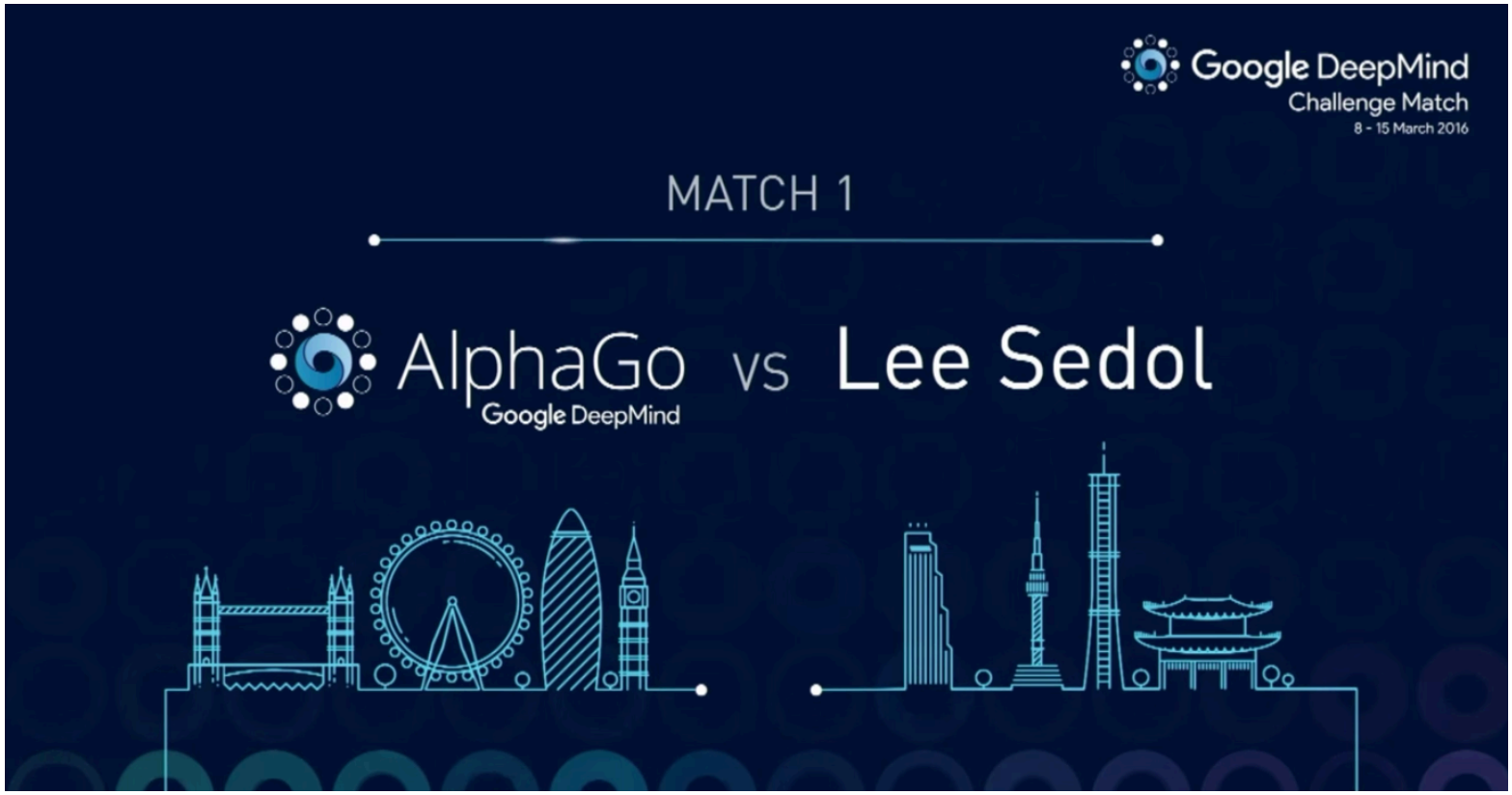
RL – Basic Idea

- We learn by interacting with our environment:
  - produces a wealth of information about cause and effect, about the consequences of actions, and about what to do in order to achieve goals.
  - Throughout our lives, such interactions are a major source of knowledge about our environment and ourselves.

RL Examples - Chess

- A master chess player makes a move.
- The choice is informed both by planning:
  - anticipating possible replies and counter replies
  - and by immediate, intuitive judgments of the ***desirability*** of particular positions and moves.

Deep Blue versus Garry Kasparov

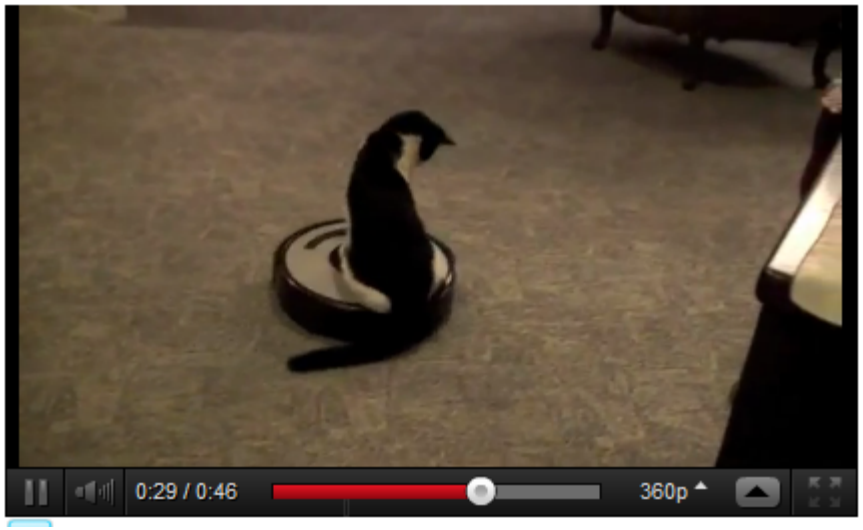

Google DeepMind
Challenge Match
8 - 15 March 2016
MATCH 1
AlphaGo
Google DeepMind
vs
Lee Sedol

RL Examples - Controller

- Helicopter flight controls given a basic model.
  - **Autonomous Helicopter Aerobatics through Apprenticeship Learning**, Pieter Abbeel, Adam Coates, and Andrew Y. Ng. *IJRR*, 2010. [IJRR] Dataset
  - http://heli.stanford.edu/

RL Examples – Roomba

- A mobile robot decides whether it should enter a new room in search of more trash to collect or start trying to find its way back to its battery recharging station.



Example Backgammon, Tesauro 1995

Learn to play Backgammon

Immediate reward

- +100 win
- -100 lose
- 0 for all other states

Trained by playing 1.5M games against itself

Now ~equal to best human player

Backgammon is a board game for two players in which pieces are moved according to the roll of dice. The winner is the first to remove all of his/her own pieces from the board.

RL Interaction

- All examples involve ***interaction*** between an ***active decision-making agent*** and its ***environment***, within which the agent seeks to achieve a ***goal*** despite *uncertainty* about its environment.
  - The agent's actions are permitted to ***affect the future state*** of the environment.
  - Correct choice requires taking into account indirect, ***delayed consequences of actions***, and thus may require foresight or planning.
  - Effects of ***actions cannot be fully predicted (stochastic environment, i.e., actions do not necessarily result in desired state).***
  - The agent can ***use its experience to improve its performance over time***.

Elements of RL

- Reinforcement learning system:
  - *policy*
  - *reward function*
  - *value function*
  - optionally, a *model* of the environment.

RL - Policy

Policy

- What the agent learns
- The learning agent's way of behaving at a given time.
- A mapping from perceived states of the environment to actions to be taken when in those states.
- I.e., for a given state, what's the best action to take.

RL – Reward and Value Functions

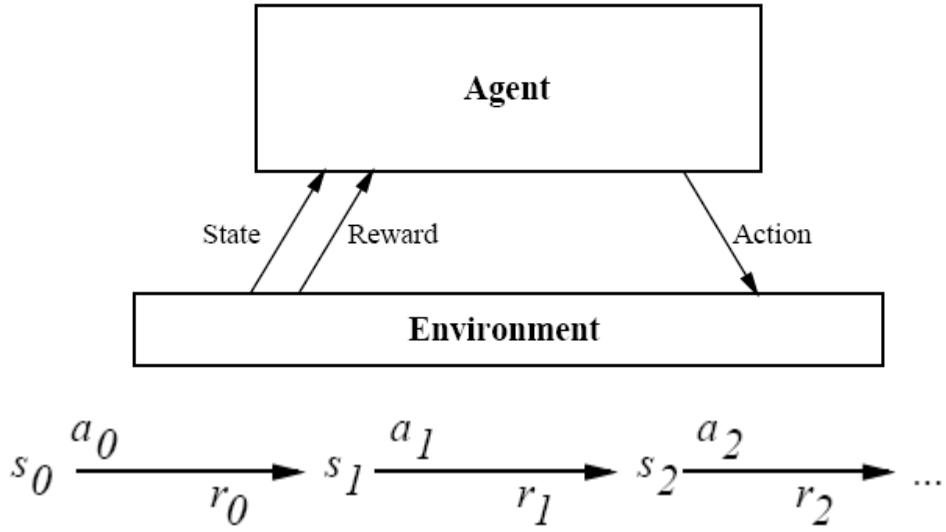
- ***Reward*** function indicates what is good in an immediate sense, whereas ***value*** *function* (utility) specifies what is good in the long run.

- ***Value*** of a state is the total amount of reward an agent can expect to accumulate over the future, starting from that state (*discounted value*).

RL – *Model* of the environment

- ***Models*** provide a ***framework*** of the environment to be explored.
- ***Models*** are used for ***planning***, by which we mean any way of deciding on a course of action by considering possible future situations before they are actually experienced.

Markov Decision Process – Reinforcement Learning Problem



- Goal: Learn to choose actions that maximize discounted cumulative reward:

$$r_0 + \gamma r_1 + \gamma^2 r_2 + \ldots \text{ , where } 0 \leq \gamma < 1$$

Markov Decision Process

Assume

- Finite set of states $\boldsymbol{S}$
- Set of actions $\boldsymbol{A}$
- At each discrete time **t**, agent observes state $\boldsymbol{s_t \in S}$ and chooses action $\boldsymbol{a_t \in A}$
- *Then receives reward* $\boldsymbol{r_t}$ *and state changes to* $\boldsymbol{s_{t+1}}$

Markov assumption: $s_{t+1} = \delta(s_t, a_t)$

- $\boldsymbol{r_t}$ *and* $\boldsymbol{s_{t+1}}$ depend only on *current* state $\boldsymbol{s_t}$ & action $\boldsymbol{a_t}$.
- The value of all future rewards are subsumed by $\boldsymbol{r_t}$
- Functions $\boldsymbol{\delta}$ and $\boldsymbol{r}$ can be nondeterministic functions, that are not known by the agent.

Agent's Learning Task

Execute actions in the environment, observe results and learn

action policy ***p: S -> A*** that maximizes the discounted value of future rewards:

$$r_0 + \gamma r_1 + \gamma^2 r_2 + \ldots \text{ , where } 0 \leq \gamma < 1$$

Where $0 \leq \gamma < 1$ is a discount factor (from economics) for future rewards.

Problem:

- Target function to learn *is* the action policy, but we have *no* training examples of form ***<s,a>.***
- Training examples are of the form of taking action ***a*** from a given state ***s*** and observing rewards:***<<s,a>,r>.***

Value Function

- For each possible policy, π*: (S->A),* the agent might adopt, we can define an evaluation function V^{π} over states:

$$V^{\pi}(s) \equiv r_t + \gamma r_{t+1} + \gamma^2 r_{t+2} + \ldots \equiv \sum_{i=0}^{\infty} \gamma^i r_{t+i}$$

- Where the rewards r_t, r_{t+1},… are generated by following policy π starting at state s.
- The task is to learn (search for) the optimal policy π^*

$$\pi^* \equiv \underset{\pi}{\operatorname{argmax}} V^{\pi}(s), (\forall s)$$

What to Learn

- Might try to *have agent learning the optimal evaluation function V*.*
- Could then do look-ahead search to choose best action from any state *s*:

$$\pi^*(s) = \operatorname{argmax}_a [r(s,a) + \gamma V^*(\delta(s,a))]$$

Problem:

- This works if the agent knows the set of states and resulting set of actions:

$$\delta : S \times A \to S, \qquad r : S \times A \to \Re$$

- But when it doesn't, it can't select actions this way.

Q Function

Define new function very similar to V^*

$$Q(s,a) \equiv r(s,a) + \gamma V^*(\delta(s,a))$$

If agent learns Q, it can choose optimal action even without knowing δ!

$$\pi^*(s) = \operatorname{argmax}_a [r(s,a) + \gamma V^*(\delta(s,a))]$$

$$\pi^*(s) = \operatorname{argmax}_a Q(s,a)$$

Q is the evaluation function the agent will learn

Training Rule to Learn Q

Note Q and V^* closely related:

$$V^*(s) = \max_{a'} Q(s, a')$$

Which allows us to write Q recursively as

$$\begin{aligned} Q(s_t, a_t) &= r(s_t, a_t) + \gamma V^*(\delta(s_t, a_t))) \\ &= r(s_t, a_t) + \gamma \max_{a'} Q(s_{t+1}, a') \end{aligned}$$

Nice! Let $\hat{Q}$ denote learner's current approximation to Q. Consider training rule

$$\hat{Q}(s, a) \leftarrow r + \gamma \max_{a'} \hat{Q}(s', a')$$

where s' is the state resulting from applying action a in state s

Q Learning from Deterministic Worlds

For each s, a initialize table entry $\hat{Q}(s, a) \leftarrow 0$

Observe current state s

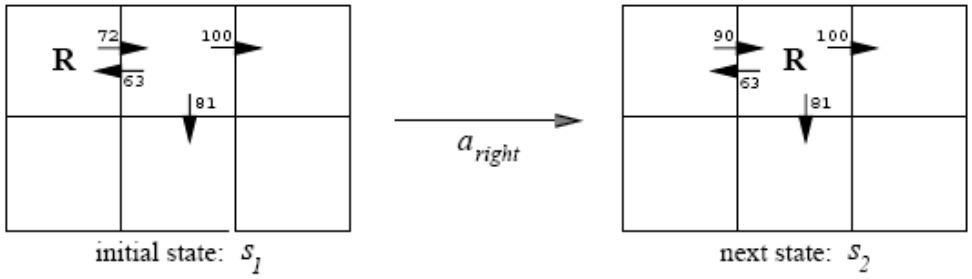
Do forever:

- Select an action a and execute it
- Receive immediate reward r
- Observe the new state s'
- Update the table entry for $\hat{Q}(s, a)$ as follows:

$$\hat{Q}(s, a) \leftarrow r + \gamma \max_{a'} \hat{Q}(s', a')$$

- $s \leftarrow s'$

Updating Q approximation



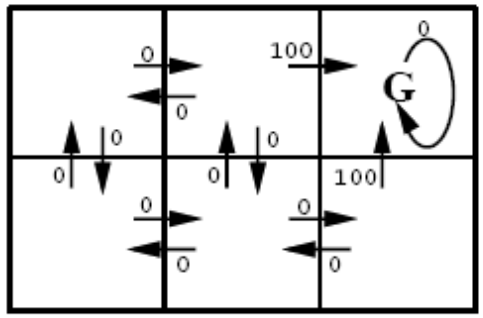
$$\begin{aligned}\hat{Q}(s_1, a_{right}) &\leftarrow r + \gamma \max_{a'} \hat{Q}(s_2, a') \\ &\leftarrow 0 + 0.9 \max\{63, 81, 100\} \\ &\leftarrow 90\end{aligned}$$

notice if rewards non-negative, then

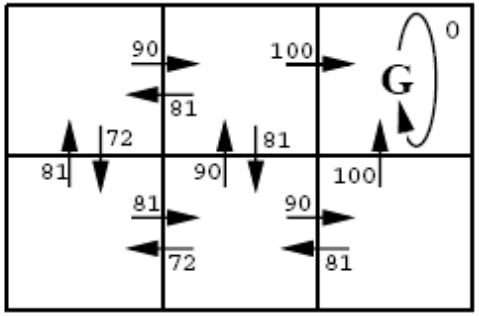
$$(\forall s, a, n) \quad \hat{Q}_{n+1}(s, a) \geq \hat{Q}_n(s, a)$$

and

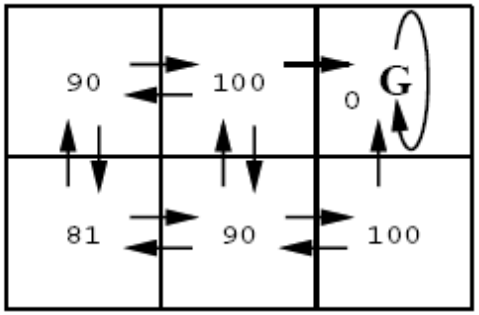
$$(\forall s, a, n) \quad 0 \leq \hat{Q}_n(s, a) \leq Q(s, a)$$



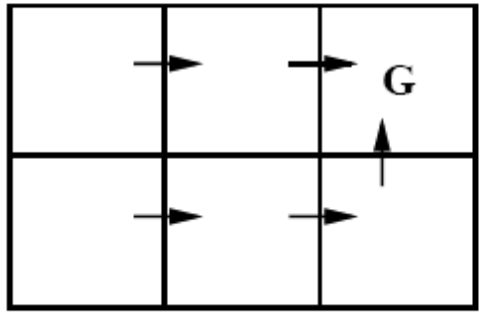
$r(s, a)$ (immediate reward) values



$Q(s, a)$ values



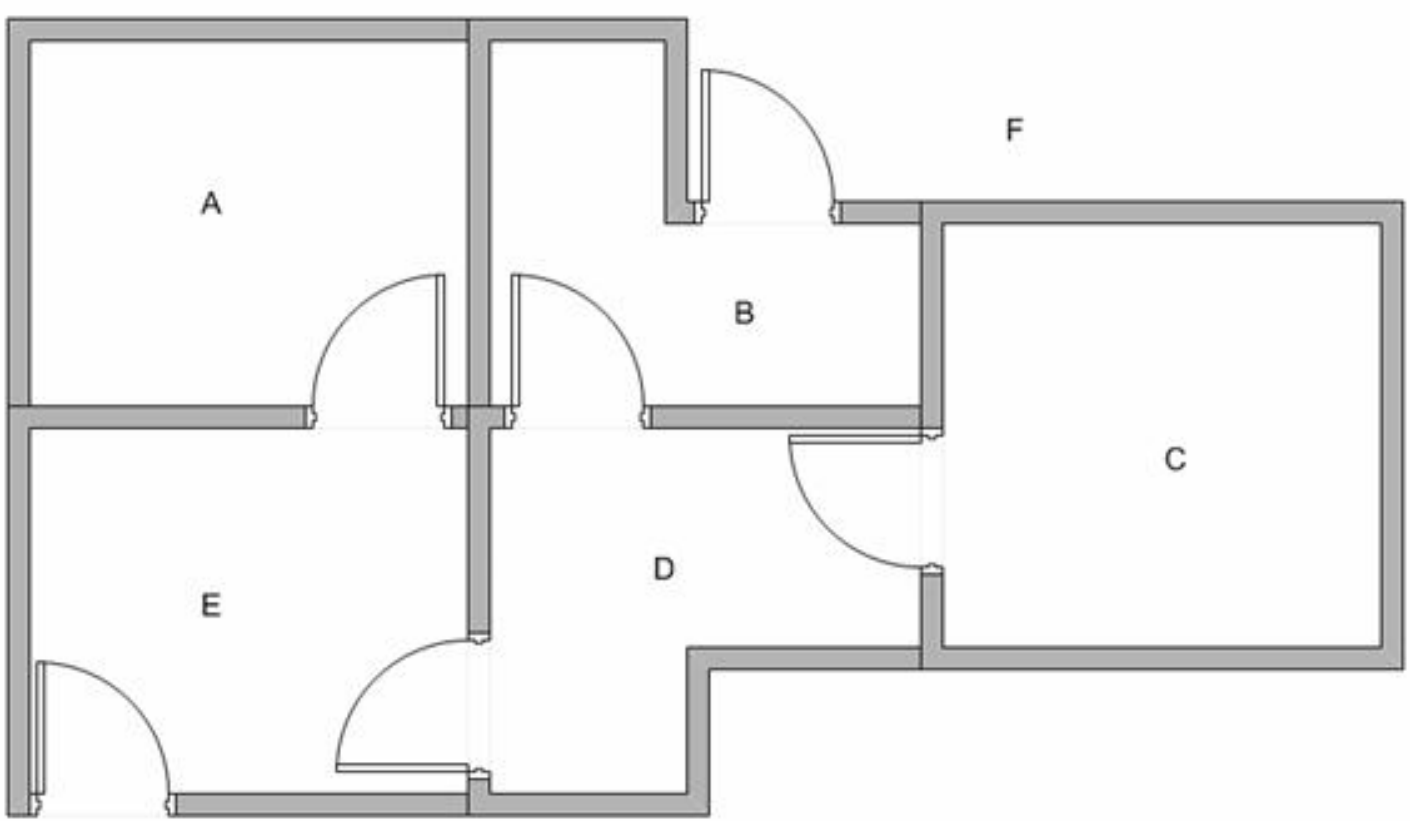
$V^*(s)$ values



One optimal policy

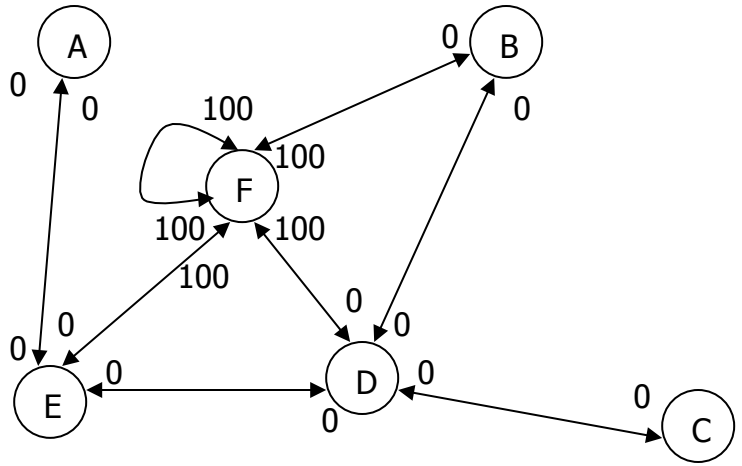
Q-Learning – Modeling the environment

- 5 rooms in a building connected by doors. Goal state is F (leave building).



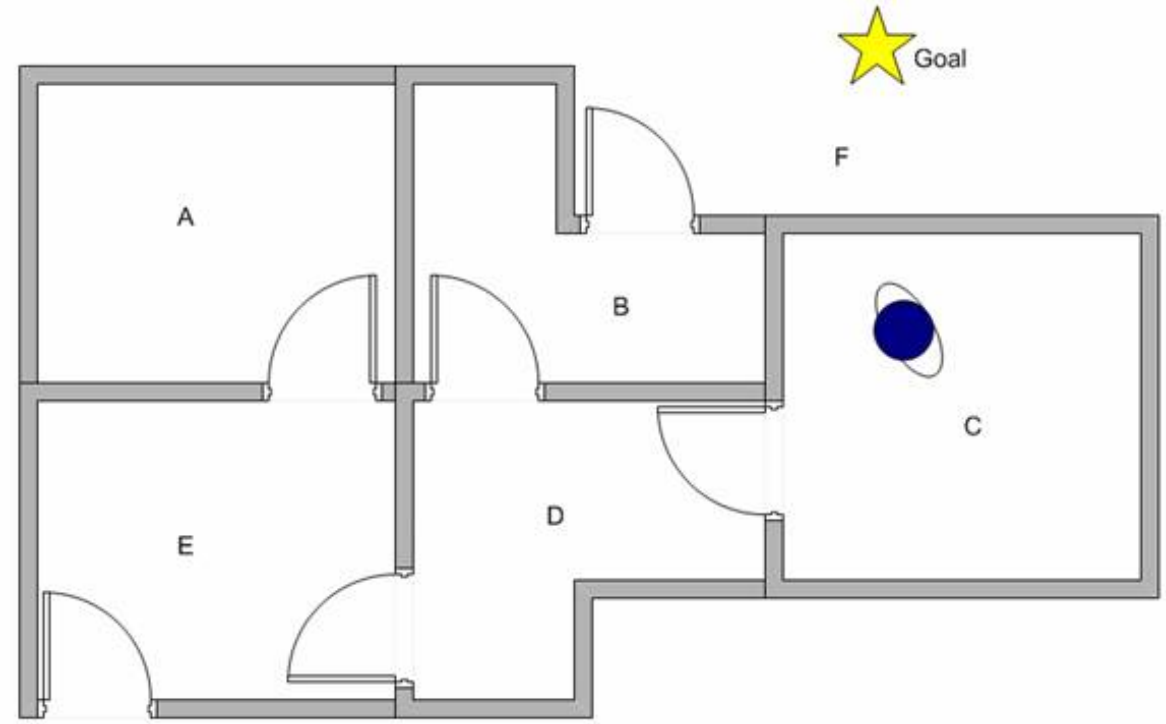
Set goal state

- Represent the rooms by a graph, each room as a vertex and each door as an edge.
- Set the target room to reinforce “good” behavior.
- If we put an agent in any room, we want the agent to go outside the building, i.e., the goal room is the node F.



Learn through experience

- Robot can *learn through experience*.
- It does not know which sequence of doors the agent must pass to go outside the building.



Q Learning

- We can put the state diagram and the instant reward values into the following reward table, or matrix **R** .

- ***Q=R(state,action) + gama*MAX(next state, all actions)***

Reward	Action					
State	A	B	C	D	E	F
A	-	-	-	-	0	-
B	-	-	-	0	-	100
C	-	-	-	0	-	-
D	-	0	0	-	0	-
E	0	-	-	0	-	100
F	-	0	-	-	0	100

Q Learning

- Set the value of learning parameter *gamma=0.8* and initial state as room **B**.
- From the second row (state B) of matrix **R.** There are two possible actions for the current state B, go to state D, or go to state F. By random selection, we select to go to **F** as our action.

Reward	Action					
State	A	B	C	D	E	F
A	-	-	-	-	0	-
B	**-**	**-**	**-**	**0**	-	**100**
C	-	-	-	0	-	-
D	-	0	0	-	0	-
E	0	-	-	0	-	100
F	-	0	-	-	0	100

Q Learning

- Now that we are in state **F**. Look at the sixth row of reward matrix **R,** i.e. state F. It has 3 possible actions to go to state B, E or F.

Q(B,F) =R(B,F) + 0.8*MAX[Q(F,B), Q(F,E), Q(F,F)]

=100 + 0.8*MAX[0,0,0]=100

Q Learning

- We update the goal state **F** with 100 (which is the current value) and stop since this is a goal state.

Q (1)	Action					
State	A	B	C	D	E	F
A	-	-	-	-	0	-
B	-	-	-	0	-	100
C	-	-	-	0	-	-
D	-	0	0	-	0	-
E	0	-	-	0	-	100
F	-	0	-	-	0	100

- This completes an *episode*.

Q Learning - another episode

- For the next episode, start with a random initial state. This time our initial state is **D**.
- Look at the fourth row of matrix **R;** it has 3 possible actions, that is to go to state B, C and E. By random selection, we select to go to state **B** as our action.

Reward

State	Action A	B	C	D	E	F
A	-	-	-	-	0	-
B	-	-	-	0	-	100
C	-	-	-	0	-	-
D	**-**	**0**	**0**	**-**	**0**	**-**
E	0	-	-	0	-	100
F	-	0	-	-	0	100

Q Learning - another episode

- Now we are in state B. Look at the second row of reward matrix **R** (i.e. state B). It has 2 possible actions to go to state D or state F. Then, we compute the **Q** value.

Q(D,B) =R(D,B) + 0.8*MAX[Q(B,D), Q(B,F)]

=0 + 0.8*MAX[0,100]=80

Q (2)	Action					
State	A	B	C	D	E	F
A	-	-	-	-	0	-
B	-	-	-	0	-	100
C	-	-	-	0	-	-
D	-	**80**	0	-	0	-
E	0	-	-	0	-	100
F	-	0	-	-	0	100

Q Learning - another episode

- The next state is **B**, which now becomes the current state.
- Repeat the inner loop in the *Q learning algorithm* because state **B** is not the goal state.
- There are two possible actions from the current state **B**, that is to go to state **D**, or go to state **F**. By lucky draw, our action selected is state **F**.
- Now we think of state **F** that has 3 possible actions to go to state B, E or F. We compute the *Q* value using the maximum value of these possible actions.

Q(B,F) =R(B,F) + 0.8*MAX[Q(F,B), Q(F,E), Q(F,F)]

=100 + 0.8*MAX[0,0,0]=100

Q Learning - another episode

- This result does not change the Q matrix.
- Because **F** is the goal state, we finish this episode.
- Our agent's *brain* now contains an updated matrix **Q**:

Q (2)	Action						
State	A	B	C	D	E	F	max
A	-	-	-	-	0	-	0
B	-	-	-	0	-	100	100
C	-	-	-	0	-	-	0
D	-	80	0	-	0	-	80
E	0	-	-	0	-	100	100
F	-	0	-	-	0	100	100

Q Learning - more episodes

- As our agent learns more episodes, it will evolve towards convergence for values of the Q (reward) matrix:

Q (1)	Action						
State	A	B	C	D	E	F	max
A	-	-	-	-	80	-	80
B	-	-	-	0	-	180	180
C	-	-	-	0	-	-	0
D	-	80	0	-	80	-	80
E	0	-	-	0	-	180	180
F	-	80	-	-	80	180	180

Q Learning - many episodes

- If our agent learns *more* and *more* experience through many episodes, it will finally reach convergence values of Q matrix as:

Q (n)	Action					
State	A	B	C	D	E	F
A	-	-	-	-	144	-
B	-	-	-	64	-	244
C	-	-	-	64	-	-
D	-	144	0	-	144	-
E	64	-	-	64	-	244
F	-	144	-	-	144	244

Q Learning - many episodes

- This **Q** matrix, then can be normalized into a percentage by dividing all valid entries with the highest number.
- We can now read optimal policy from max state.

Q (n) Action

State	A	B	C	D	E	F	max	Max State
A	-	-	-	-	59%	-	59%	E
B	-	-	-	26 %	-	100%	100 %	F
C	-	-	-	26 %	-	-	26%	D
D	-	59%	0%	-	59%	-	59%	B
E	26%	-	-	26 %	-	100%	100 %	F
F	-	59%	-	-	59%	100%	100 %	F

Q Learning Algorithm

- Set parameter , and environment reward matrix **R**
- Initialize matrix **Q** as zero matrix
- For each episode:
  - Select random initial state
  - Do while we have not reached the goal state
    - Select one among all possible actions for the current state
    - Using this possible action, *consider* to go to the next state
    - Get maximum Q value of this next state based on all possible actions
    - Compute $\mathbf{Q}(state, action) = \mathbf{R}(state, action) + \gamma \cdot Max\left[\mathbf{Q}(next\ state, all\ actions)\right]$
    - Set the next state as the current state
- End Do
- End For

Algorithm to utilize the Q matrix

Input: **Q** matrix, initial state

1. Set current state = initial state
2. From current state, find action that produce maximum Q value
3. Set current state = next state
4. Go to 2 until current state = goal state

Q_{hat} convergence

- Q_{hat} (Q approximation) converges to Q in a deterministic world where each <s,a> is visited infinitely often.

- Proof: Define a full interval to be an interval during which each <s,a> is visited. During each full interval the largest error in Q_{hat} table is reduced by a factor of γ

Non-deterministic case

- What if reward and next state are non-deterministic?
- Redefine V and Q by taking expected values:

$$V^{\pi}(s) \equiv E[r_t + \gamma r_{t+1} + \gamma^2 r_{t+2} + \ldots] \equiv E[\sum_{i=0}^{\infty} \gamma^i r_{t+i}]$$

$$Q(s,a) \equiv E[r(s,a) + \gamma V^*(\delta(s,a))]$$

- Note: Bayesian RL is an active research field

Non-deterministic case

Q learning generalizes to nondeterministic worlds

Alter training rule to

$$\hat{Q}_n(s,a) \leftarrow (1-\alpha_n)\hat{Q}_{n-1}(s,a)+\alpha_n[r+\max_{a'}\hat{Q}_{n-1}(s',a')]$$

where

$$\alpha_n = \frac{1}{1+visits_n(s,a)}$$

Can still prove convergence of $\hat{Q}$ to Q [Watkins and Dayan, 1992]

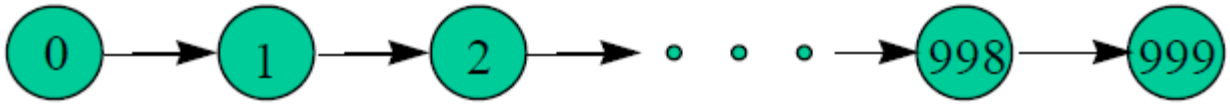
Temporal difference learning

- The number of training iterations necessary to sufficiently represent the optimal *Q-function* scales poorly with respect to the size of the time interval between states.
- The greater the number of actions per unit time, the greater the number of training iterations required to adequately represent the optimal Q-function.
- TD uses the concept of discounting the cumulative reinforcement versus the reinforcement from a single state transition.

$$Q\left(s_{t,}a_{t}\right)=Q\left(s_{t,}a_{t}\right)+\alpha\left[r_{t+1}+\gamma Q\left(s_{t+1,}a_{t+1}\right)-Q\left(s_{t,}a_{t}\right)\right]$$

Temporal difference learning

- Given the MDP with 1000 states.
- State 0 is initial state & state 999 is an absorbing state.
- Each state transition returns a cost (reinforcement) of 1, and the value for 999 is defined to be 1.



- Q-learning, value interaction can solve this problem.
- However TD(lambda) can solve it faster.
- TD(lambda) updates the value of the current state based on a weighted combination of the values of all future states versus only the value of the immediate successor state. Basic Bellman eqn:

$$V(\mathbf{x}_t, \mathbf{w}_t) = r(\mathbf{x}_t) + \gamma V(\mathbf{x}_{t+1}, \mathbf{w}_t)$$

Temporal difference learning –

Update rule uses difference in utilities between successive states

Q learning: reduce discrepancy between successive Q estimates

One step time difference:

$$Q^{(1)}(s_t, a_t) \equiv r_t + \gamma \max_a \hat{Q}(s_{t+1}, a)$$

Why not two steps?

$$Q^{(2)}(s_t, a_t) \equiv r_t + \gamma r_{t+1} + \gamma^2 \max_a \hat{Q}(s_{t+2}, a)$$

Or n?

$$Q^{(n)}(s_t, a_t) \equiv r_t + \gamma r_{t+1} + \cdots + \gamma^{(n-1)} r_{t+n-1} + \gamma^n \max_a \hat{Q}(s_{t+n}, a)$$

Blend all of these:

$$Q^{\lambda}(s_t, a_t) \equiv (1-\lambda) \left[Q^{(1)}(s_t, a_t) + \lambda Q^{(2)}(s_t, a_t) + \lambda^2 Q^{(3)}(s_t, a_t) \right.$$

Temporal difference learning –

the reinforcement is the difference between the ideal prediction and the current prediction

$$Q^{\lambda}(s_t, a_t) \equiv (1-\lambda)\left[Q^{(1)}(s_t, a_t) + \lambda Q^{(2)}(s_t, a_t) + \lambda^2 Q^{(3)}(s_t, a_t)\right.$$

Equivalent expression:

$$Q^{\lambda}(s_t, a_t) = r_t + \gamma[\ (1-\lambda)\max_a \hat{Q}(s_t, a_t) + \lambda\ Q^{\lambda}(s_{t+1}, a_{t+1})]$$

$$Q(s_t, a_t) = Q(s_t, a_t) + \alpha\left[r_{t+1} + \gamma Q(s_{t+1}, a_{t+1}) - Q(s_t, a_t)\right]$$

Ongoing research

- Replace Q approximation table with Deep Neural Net, Bayes Net or other generalized classifier
- Handle case where state only partially observable
- Design optimal exploration strategies
- Extend to continuous action, state
- Learn and use delta function: *S x A->S*
- Relationship to dynamic programming
- Applications to dynamic markets (Stock, auctions, etc.)

Applications of Reinforcement Learning

- Intelligent Trading Agents for Massively Multi-player Game Economies, John Reeder, Gita Sukthankar, M. Georgiopoulos, G. Anagnostopoulos
- Learning to be a Bot: Reinforcement Learning in Shooter Games, Michelle McPartland, Marcus Gallagher
- Agent Learning using Action-Dependent Learning Rates in Computer Role-Playing Games, Maria Cutumisu, Duane Szafron, Michael Bowling, Richard S. Sutton
- Combining Model-Based Meta-Reasoning and Reinforcement Learning for Adapting Game-Playing Agents, Patrick Ulam, Joshua Jones, Ashok Goel

Demos

- **Demo - Control of an octopus arm using GPTD**
- http://videolectures.net/icml07_engel_demo/

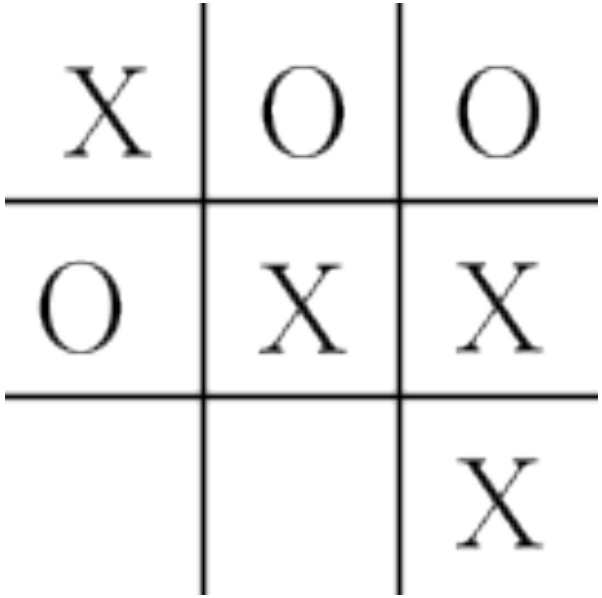
- Reinforcement Learning Repository at UMass, Amherst
- http://www-all.cs.umass.edu/rlr/domains.html

- Robot arm
- http://iridia.ulb.ac.be/~fvandenb/qlearning/qlearning.html

- Cat-Mouse applet
- http://www.cse.unsw.edu.au/~cs9417ml/RL1/applet.html

RL Example – Tic Tac Toe

How might we construct a player that will find the imperfections in its opponent's play and learn to maximize its chances of winning?



Tic Tac Toe – *Minimax*

- ***Minimax*** algorithm assumes a particular way of playing by the opponent – an infallible opponent.
- ***Minimax*** player would never reach a game state from which it could lose – even though this may be a valuable strategy.
- ***Minimax*** will not scale to complex environments with large state space.

X	O	O
O	X	X
		X

Tic Tac Toe – Dynamic Programming

- ***Dynamic programming***, can *compute* an optimal solution for any opponent, but requires as input a complete specification of that opponent – not realistic in many game playing settings.

Tic Tac Toe – Learning a Model

- Assume that information is not available ***a priori*** for this problem, as it is not for the vast majority of problems of practical interest.
- Such information can be estimated from experience playing many games against the opponent!
- ***Learn a model*** of the opponent's behavior up to some level of confidence, and then apply ***dynamic programming*** to compute an optimal solution given the approximate opponent model.

Reinforcement Learning

- How can an agent ***learn*** from ***success/failure***, or ***reward/punishment*** to achieve a goal?
- An agent can learn to play chess by ***supervised learning*** – given examples of game situations along with the best moves for those situations.
- What if there is no friendly teacher providing examples for us to learn from, what can a poor agent do?
- How can an agent learn from it's own experiences?

Reinforcement Learning – Basic Idea

- By trying random moves and observing the outcomes of its actions, an agent can *eventually* build a predictive model of its environment.
- Given enough experience playing a game, an agent will know what the board will look like after it makes a given move and even how the opponent is likely to reply in a give situation.

RL– *Step #1*

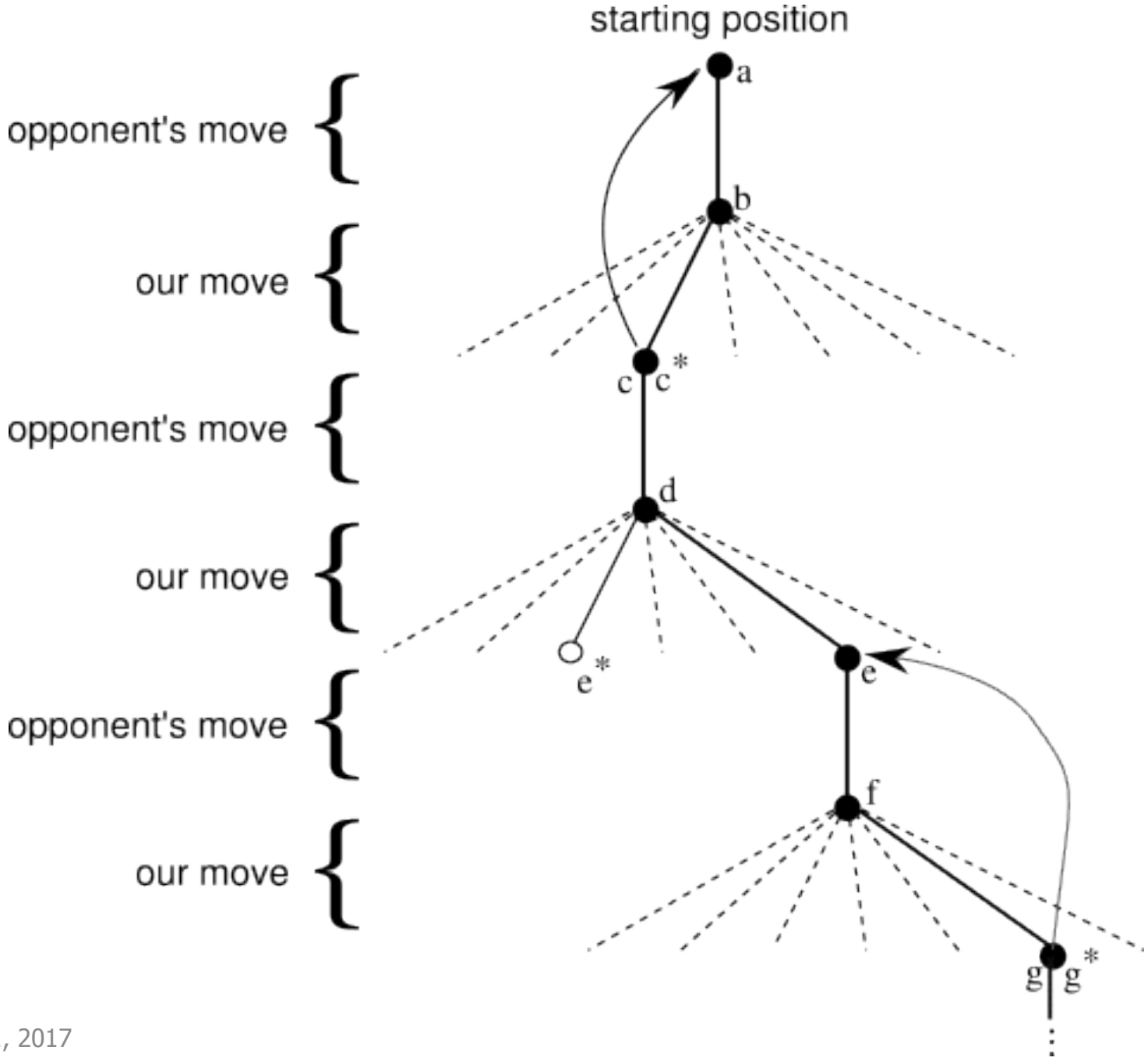
Set up a table of numbers, one for each possible state of the game.

- Each number will be the latest estimate of the *probability* of our winning from that state.
- We treat this estimate as the state's *value*, and the whole table is the learned value function.
- If state A has higher value than state B, or is considered "better" than state B, i.e., current estimate of the probability of winning from A is higher than it is from B.
- Assuming we always play X's, then for all states with three Xs in a row/ diagonal the probability of winning is 1.
- All states with three O's in a row/diagonal, we loose.
- Set the initial values of all the other states to 0.5, representing a guess that we have a 50% chance of winning.

RL– *Step #2*

Play *many* games against the opponent.

- Select our moves by examining the states that would result from each of our possible moves (one for each blank space on the board) and look up their current values in the table.
- Most of the time we move ***greedily***, selecting the move that leads to the state with greatest value, that is, with the highest estimated probability of winning.
- Occasionally, we select randomly from among the other moves instead: ***exploratory*** moves because they cause us to experience states that we might otherwise never see.


starting position
opponent's move
our move
opponent's move
our move
opponent's move
our move
a
b
c
c*
d
e*
e
f
g
g*

RL Tic-tac-toe – *Step #3*

While learning/playing, change the values of the states in which we find ourselves during the game.

- Make them more accurate estimates of the probabilities of winning.
- Do this, by "backing up" (discounting) the value of the state after each greedy move to the state before the move.
- I.e., the current value of the earlier state is adjusted to be closer to the value of the later state.
- This can be done by moving the earlier state's value a fraction of the way toward the value of the later state.

RL Tic-tac-toe – *Temporal Difference Learning*

- α is the earning rate for convergence
- $V(s') - V(s)$ is the temporal difference

$$V(s) \leftarrow V(s) + \alpha \Big[V(s') - V(s) \Big],$$

- Without the learning rate, the value function may not converge. Illustrates difference between evolutionary learning and policy based learning.

- Demo Tic Tac Toe project!

RL Tic-tac-toe – *Key features*

1. Emphasis on learning while ***interacting with an environment***.
2. There is a ***clear goal***, and correct behavior requires planning or foresight that takes into account delayed effects of one's choices.
   - The simple reinforcement learning player would learn to set up multi-move traps for a shortsighted opponent.
   - Striking feature of the reinforcement learning solution that it can achieve the effects of planning and

RL Tic-tac-toe - *Summary*

- TTT: relatively small, finite state set, whereas reinforcement learning can be used when the state set is very large, or even infinite.
- Gerry Tesauro (1992, 1995) combined the algorithm described above with an artificial neural network to learn to play backgammon, which has approximately 10^{20} states.
  - With this many states it is impossible to experience more than a small fraction of the states.
  - The neural network provides the ability to ***generalize*** from its experience.
  - New states it selects, are based on information saved from similar states faced in the past, as determined by its network.
- How well a reinforcement learning system can work in problems with such large state sets is intimately tied to how appropriately it can generalize from past experience.

RL Summary

- Machine learning approach for understanding and automating ***goal-directed learning*** and decision-making.
- Distinguished from other approaches by its emphasis on learning by the individual from direct ***interaction with its environment***.
- RL uses a ***formal framework defining the interaction between a learning agent and its environment*** (table) in terms of states, actions, and rewards.
- Use of ***value functions*** distinguishes reinforcement learning methods from evolutionary methods that search directly in policy space guided by scalar evaluations of entire policies.