Reinforcement Learning

CS4881 Artificial Intelligence,
Jay Urbain

Credits: *Machine Learning, Tom Mitchell*

AIMA, Russell and Norvig

Kardi Teknomo, Q-Learning Using Matlab

Reinforcement Learning

- 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.

Reinforcement Learning

- How can an agent can learn from ***success* and *failure***, or ***reward*** and ***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.
  - For example, 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.

Reinforcement Learning

Consider learning to choose actions:

- Robot learning to dock on battery charger
- Learning to choose actions to optimize factory output
- Learning to play Tic-tac-toe, Chess, or Wumpus World!

Note several problem characteristics:

- Delayed reward *(no instant gratification!)*
- Opportunity for active exploration
- Possibility that the state is only partially observable
- Possible need to learn multiple tasks with same sensors/effectors

Example Backgammon, Tesauro 1995

Learn to play Backgammon

Immediate reward

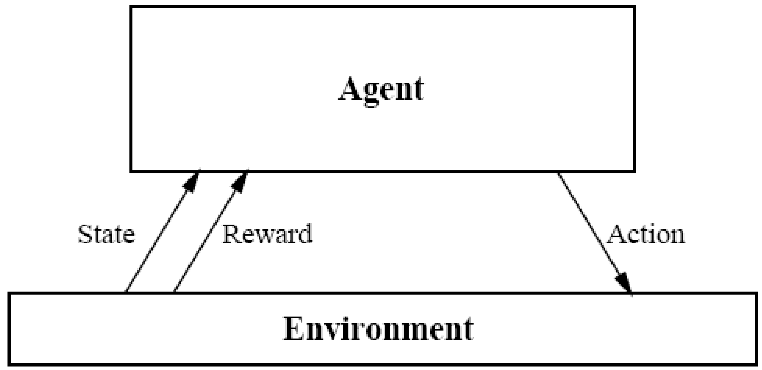
- +100 win
- -100 lose
- 0 for all other states

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.

Reinforcement Learning Problem



$$s_0 \xrightarrow[r_0]{a_0} s_1 \xrightarrow[r_1]{a_1} s_2 \xrightarrow[r_2]{a_2} \ldots$$

- Goal: Learn to choose actions that maximize reward

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

Markov Decision Process Defined

- A discrete time stochastic control process.
- At each time step, the process is in some state *s*, and the decision maker may choose any action ***a*** that is available in state ***s*****.**
- The process responds at the next time step by randomly moving into a new state *s'*, and giving the decision maker a corresponding reward $R_a(s,s')$.

Markov Decision Process

Assume

- Finite set of states $\boldsymbol{S}$
- Set of actions $\boldsymbol{A}$
- At each discrete time **t**, agent observes state $\boldsymbol{s_t} \in \boldsymbol{S}$ and chooses action $\boldsymbol{a_t} \in \boldsymbol{A}$
- *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* ***π : 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 the discount factor (from economics) for future rewards.

Problem:

- Target function *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:

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

- 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****

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

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

Problem:

- This works if the agent knows the set of states and resulting set of actions: $\delta : S \times A \rightarrow S, \qquad r : S \times A \rightarrow \Re$
- 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!

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

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

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

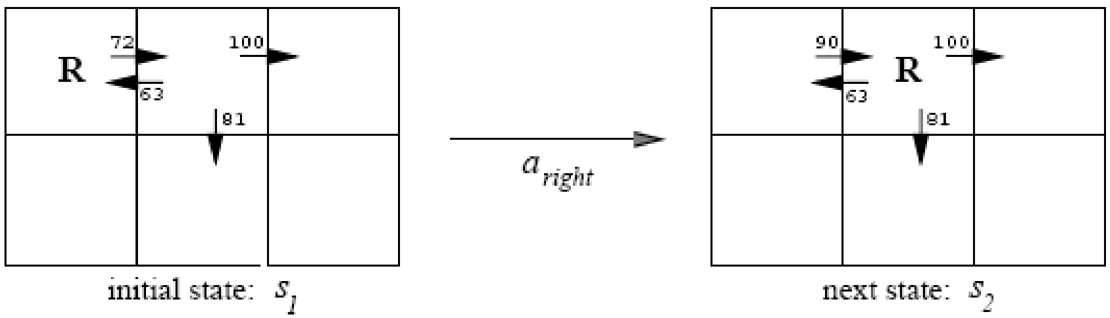
Do forever:

- 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



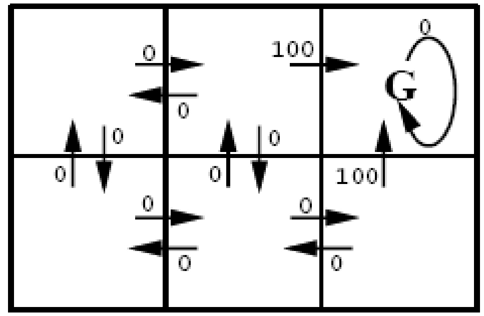
$$
\begin{aligned}
\hat{Q}(s_1, a_{right}) &\leftarrow r + \gamma \max_{a'} \hat{Q}(s_2, a') \\
&\leftarrow 0 + 0.9 \ \max\{63, 81, 100\} \\
&\leftarrow 90
\end{aligned}
$$

notice if rewards non-negative, then

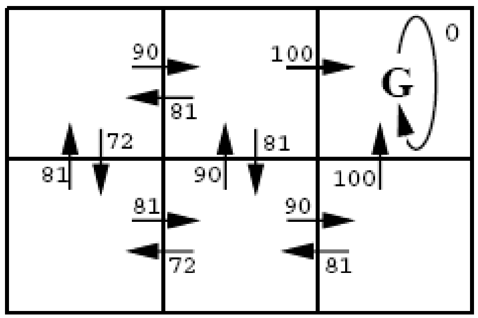
$$(\forall s, a, n) \quad \hat{Q}_{n+1}(s,a) \geq \hat{Q}_n(s,a)$$

and

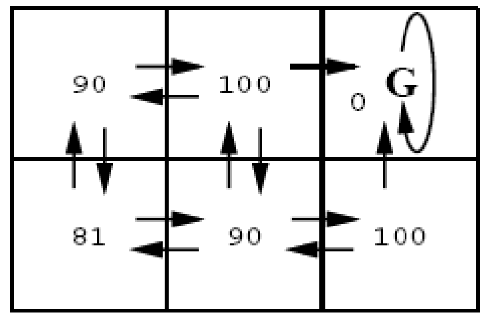
$$(\forall s, a, n) \quad 0 \leq \hat{Q}_n(s,a) \leq Q(s,a)$$



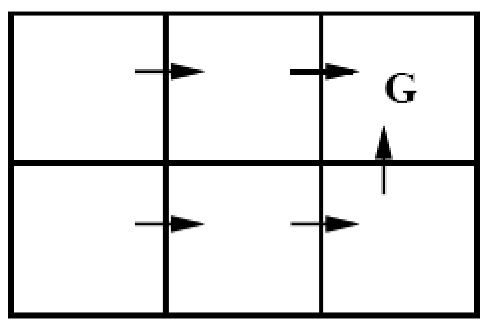
$r(s, a)$ (immediate reward) values



$Q(s, a)$ values



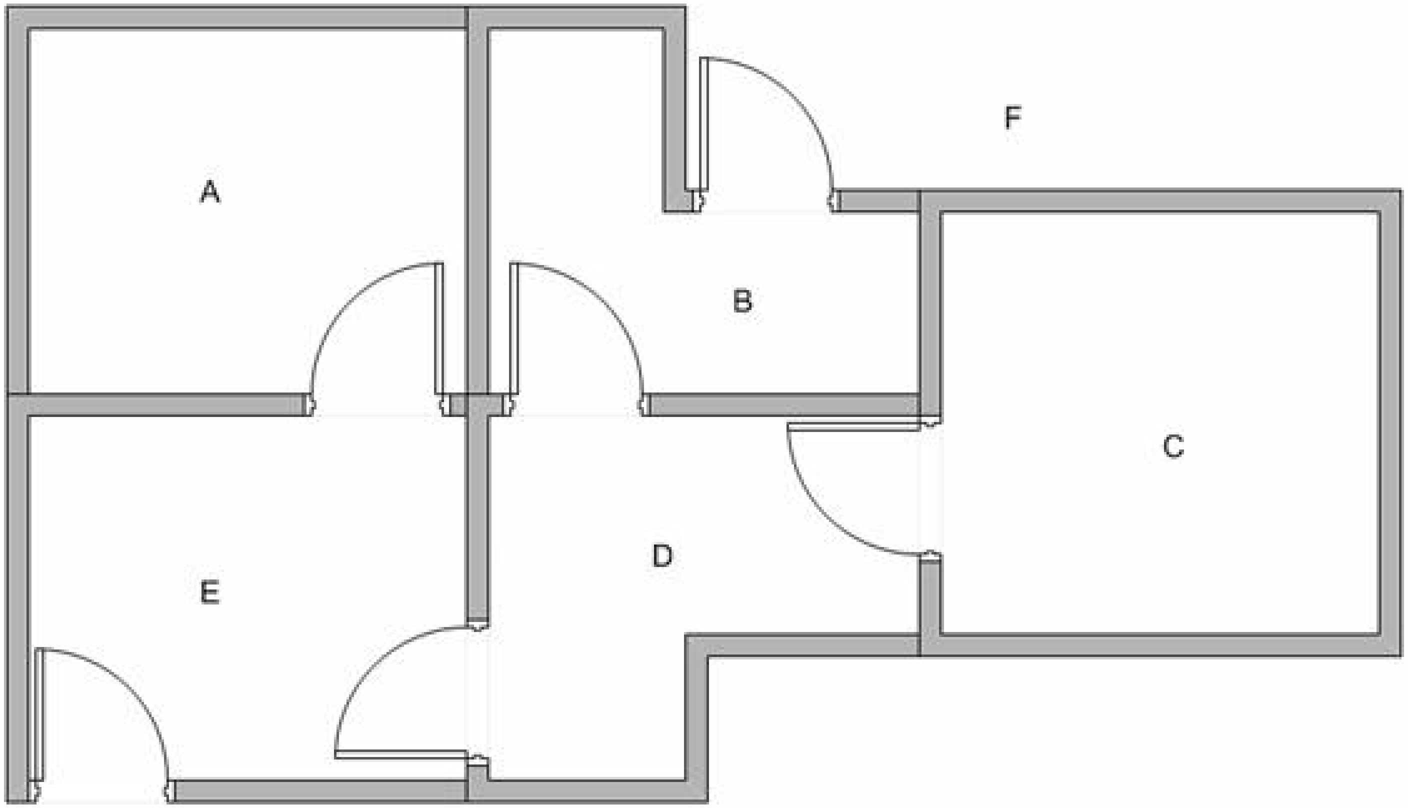
$V^*(s)$ values



One optimal policy

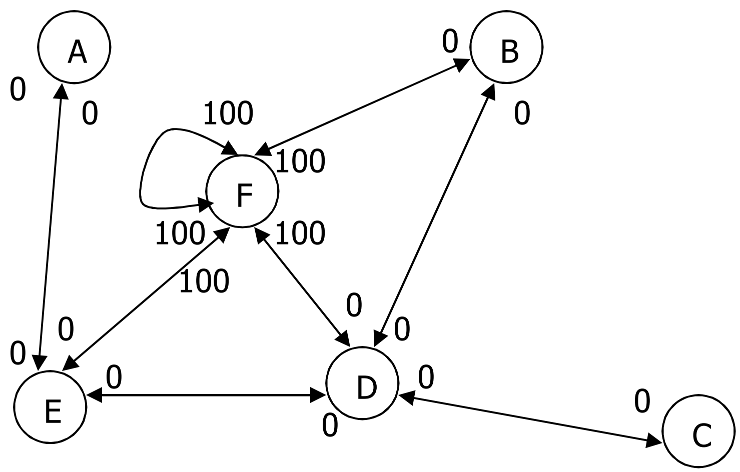
Modeling the environment

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



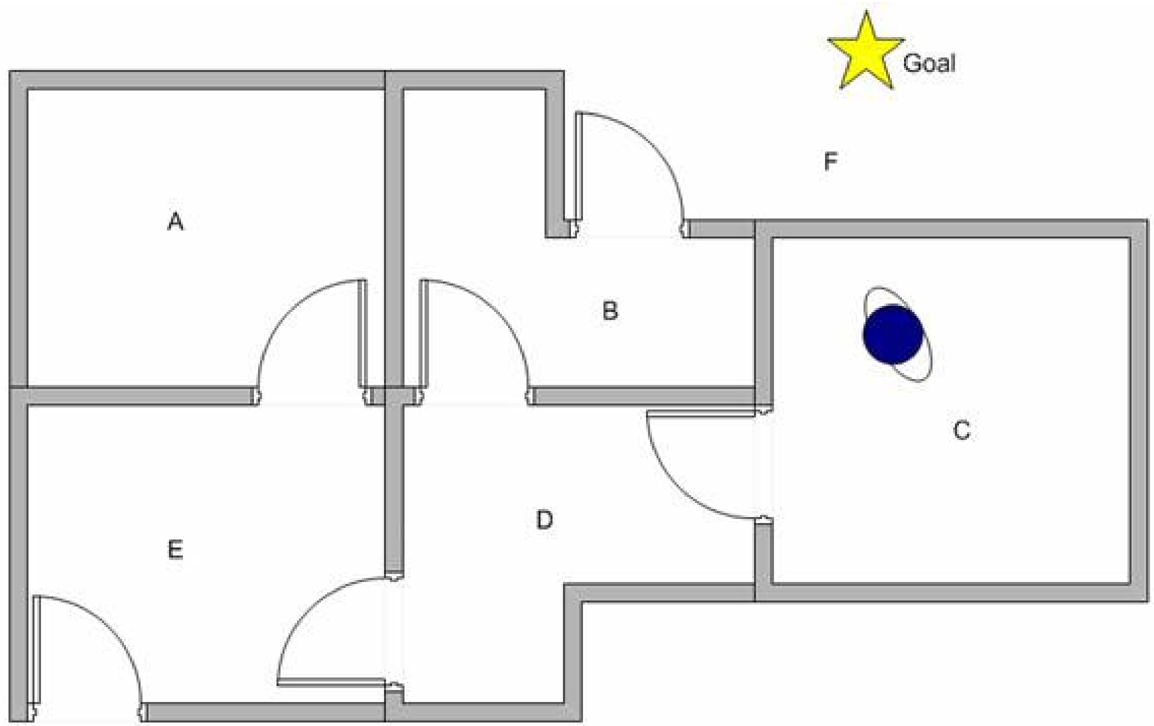
Set goal state

- 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	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 - 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.

$$\mathbf{Q(B,F) = R(B,F) + 0.8 * MAX[Q(F,B), Q(F,E), Q(F,F)]}$$
$$\mathbf{= 100 + 0.8 * MAX[0,0,0] = 100}$$

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 not reach 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:

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

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

- 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

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

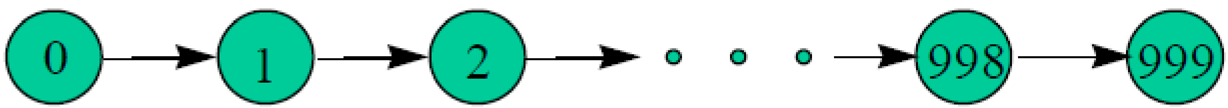
Can still prove convergence of $\hat{Q}$ to Q [Watkins and Dayan, 1992]

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.

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:

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

Ongoing research

- Replace Q approximation table with 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