

# Reinforcement Learning

CSE 4309 – Machine Learning

Vassilis Athitsos

Computer Science and Engineering Department

University of Texas at Arlington

# Markov Decision Processes (MDPs)

- In Markov Decision Processes, we made two assumptions:
- Assumption 1: we have a Markovian transition model,  $p(s' | s, a)$ .
  - This model gives us the probability of the next state being  $s'$ , if the agent is at state  $s$  and takes action  $a$ .
- Assumption 2: **Discounted Additive Rewards**.
  - The utility  $U_h$  of a state sequence is:  $U_h(s_0, s_1, \dots, s_T) = \sum_{t=0}^T \gamma^t R(s_t)$
- In an MDP, both the transition model  $p(s' | s, a)$  and the reward function  $R(s)$  are **known to the agent in advance**.
- What we do not know, and we need to compute using specific algorithms, is the optimal policy and the utility of each state.

3				+1
2				-1
1	START			
	1	2	3	4

# Reinforcement Learning

- Compared to MDPs, **Reinforcement Learning** (RL) is a more difficult version of sequential decision problems.
- As in MDPs, we want to compute:
  - The optimal policy (or, at least, a good policy).
  - The utility of each state
- However, in contrast to MDPs, in RL we **do not know**:
  - The transition model  $p(s' | s, a)$ .
  - The reward function  $R(s)$ .
- Figuring out what policy to follow is more complicated, with an unknown transition model and an unknown reward function.

3				+1
2				-1
1	START			
	1	2	3	4

# Reinforcement Learning

- The only way the agent can learn anything in RL is by taking actions and observing results.
- The observed outcomes of an action are:
  - The resulting state, which can be used as data for estimating  $p(s' | s, a)$ .
  - The reward  $R(s')$  at the resulting state  $s'$ , which can simply be memorized.
- As the agent moves around, and observes state transitions and rewards, the agent can learn about the environment, by:
  - Memorizing the reward  $R(s)$  at each state that the agent visits.
  - Learning a model of state transitions  $p(s' | s, a)$ .
  - Estimating the utility  $U(s)$  of each state.
  - Estimating a good (or optimal) policy to follow.

3	-0.04	-0.04	-0.04	+1
2	-0.04		-0.04	-1
1	-0.04	-0.04	-0.04	-0.04
	1	2	3	4

# Reinforcement Learning

- For example:
  - The agent is at state (1,3).
  - The agent takes action "go up".
  - The result of the action is to actually move left, to state (1,2).
  - The agent gets a reward of -0.04 for visiting state (1,2).
- Based on that experience, the agent learns the following:
  - $R(1,2) = -0.04$ . This can be memorized, it will never change.
  - The result of action "go up" from state (1,3), in a single attempt, was state (1,2).

3	-0.04	-0.04	-0.04	+1
2	-0.04		-0.04	-1
1	-0.04	-0.04	-0.04	-0.04
	1	2	3	4

# Reinforcement Learning

- How can the agent estimate probability  $p((1,2) \mid (1,3), \text{"go up"})$  from this experience?
  - Frequentist approach: estimated  $p((1,2) \mid (1,3), \text{"go up"}) = 1$ .
    - Too aggressive!
    - It can be updated to more accurate estimates in the future, whenever the agent revisits state (1,3) again and tries action "go up" again.
  - Bayesian approach: start with some priors on the probability that  $p((1,2) \mid (1,3), \text{"go up"}) = \theta$ , and update the estimate after each observation.
    - Similar to the sunrise example we saw at the beginning of the course.

3	-0.04	-0.04	-0.04	+1
2	-0.04		-0.04	-1
1	-0.04	-0.04	-0.04	-0.04
	1	2	3	4

# Reinforcement Learning

- Some times, rewards are only observed when the agent gets to specific states.
- Without loss of generality, we can assume that the rest of the states have zero rewards.
- Example: chess.
  - The agent plays games of chess, against itself or against others.
  - During each game, the agent makes different moves, without feedback as to whether each move was good or bad.
  - At some point, the game is over, and the reward is +1 for winning, 0 for a tie, -1 for losing.

3				+1
2				-1
1				
	1	2	3	4

# Reinforcement Learning vs. Supervised Learning

- Learning to play chess can be approached as a reinforcement learning problem, or as a supervised learning problem.
- Reinforcement learning approach:
  - The agent plays games of chess, against itself or against others.
  - During each game, the agent makes different moves, without feedback as to whether each move was good or bad.
  - At some point, the game is over, and the reward is +1 for winning, 0 for a tie, -1 for losing.
- Supervised learning approach:
  - The agent plays games of chess, against itself or against others.
  - During each game, the agent makes different moves.
  - For every move, an expert provides an evaluation of that move.
- Pros and cons of each approach?













# Reinforcement Learning vs. Supervised Learning

- In the chess example, the big advantage of reinforcement learning is that no effort is required from human experts to evaluate moves.
  - Lots of training data can be generated by having the agent play against itself or other artificial agents.
  - No human time is spent.
- Supervised learning requires significant human effort.
  - If that effort can be spared, supervised learning has more information, and thus should learn a better strategy.
  - However, in many sequential decision problems, the state space is so large, that it is infeasible for humans to evaluate a sufficiently large number of states.

# Passive and Active RL











- In typical RL problems, the agent proceeds step-by-step, where every step involves:
  - Deciding what action to take, based on its current policy.
  - Taking the action, observing the outcome, and modifying accordingly its current policy.
- This problem is called **active reinforcement learning**, and we will look at some methods for solving it.
- However, first, we will study an easier RL problem, called **passive reinforcement learning**.
- In this easier version:
  - The policy is fixed.
  - The transition model and reward function are still unknown.
  - The goal is simply to compute the utility value of each state.

# Passive Reinforcement Learning

3				+1
2				-1
1				
	1	2	3	4

- In the figure we see an example of a specific environment and a policy.
  - Suppose that  $R(s) = -0.04$  if  $s$  is non-terminal.
  - Suppose that  $\gamma = 1$ .
  - Suppose that the transition model  $p(s' | s, a)$  is the same as before:
    - 80% chance of moving in the direction the agent wanted.
    - 20% chance of moving perpendicular to the intended direction.
    - If the agent hits a wall, the agent remains in the current state.

# Passive Reinforcement Learning

3				+1
2				-1
1				
	1	2	3	4

- This problem has similarities with the policy evaluation problem in MDPs.
- The main differences:
  - Here, the reward function  $R(s)$  is not known in advance.
  - Here, the transition model  $p(s' | s, a)$  is not known in advance.
- The **adaptive dynamic programming** (ADP) algorithm address these issues, by doing the following at each step:
  - Update  $R(s)$  and  $p(s' | s, a)$  based on current observations (reward of current state, result of last action).
  - Using the updated  $R(s)$  and  $p(s' | s, a)$ , run the **PolicyEvaluation** function to estimate the utility of each state.

# Adaptive Dynamic Programming

- The ADP algorithm takes many arguments.
- There are **input** arguments, whose value is fixed:
  - $(s, a)$ : the previous state  $s$  and the previous action  $a$  that led from  $s$  to  $s'$ .
  - $(s', r')$ : the current state  $s'$ , and the reward  $r'$  received at state  $s'$ .
  - $\pi$ : the policy mapping each state  $s$  to an action  $\pi(s)$ .
  - $\gamma$ : the discount factor.
  - $K$ : parameter, that is passed to **PolicyEvaluation**, and specifies how many rounds of updates to perform in that function.
- There are also **input/output** arguments, whose values can change.
  - $\mathbb{S}$ : the set of states, which **is not known in advance**.
  - $R, U$ : the reward and utility tables, storing rewards and utilities of states.
  - $p$ : a 3D table of transitions, storing values  $p(s' \mid s, a)$ .
  - $N_{sa}$ : a table, where  $N_{sa}[s, a]$  counts all times  $a$  was performed from state  $s$ .
  - $N_{sat}$ : a table, where  $N_{sat}[s, a, t]$  counts all times  $a$  led from  $s$  to  $t$ .

# Adaptive Dynamic Programming

- **Input** arguments:
  - $(s, a)$ : previous state and action.
  - $(s', r')$ : current state and reward.
  - $\pi$ : the policy.
  - $\gamma$ : the discount factor.
  - $K$ : used in **PolicyEvaluation**.
- **Input/output** arguments:
  - $\mathbb{S}$ : the set of states.
  - $R, U$ : reward and utility tables.
  - $p$ : table of values  $p(s' | s, a)$ .
  - $N_{sa}$ : 2D table, counts times action  $a$  was taken in state  $s$ .
  - $N_{sat}$ : 3D table, counts number of times  $a$  led from  $s$  to  $t$ .
- The task of the ADP algorithm is to process a single step:
  - Process the result of the last action  $a$ , which led from state  $s$  to state  $s'$ .
    - Update values  $p(t | s, a)$ .
  - Record the observed reward at state  $s'$ .
  - Call **PolicyEvaluation** to update the utility table.
- The agent runs the ADP algorithm **after each move**.

# ADP Pseudocode

**function** **ADP**( $s, a, s', r', \pi, \gamma, K, \mathbb{S}, R, U, p, N_{sa}, N_{sat}$ )

**if** this is the first time we visit  $s'$ :

$$R[s'] = r'$$

$$U[s'] = r'$$

$\mathbb{S} = \mathbb{S} \cup \{s'\}$  // add  $s'$  to set of known states.

**if**  $s$  is not **null**: //  $s$  is null if no states have been visited before

**if**  $N_{sa}[s, a]$  exists:  $N_{sa}[s, a] += 1$  **else**:  $N_{sa}[s, a] = 1$

**if**  $N_{sat}[s, a, s']$  exists:  $N_{sat}[s, a, s'] += 1$  **else**:  $N_{sat}[s, a, s'] = 1$

**for each** state  $t$  such that  $N_{sat}[s, a, t] \neq 0$ :

$$p(t \mid s, a) = \frac{N_{sat}[s, a, t]}{N_{sa}[s, a]}$$

$U = \mathbf{PolicyEvaluation}(\mathbb{S}, p, R, \gamma, \pi, K, U)$

# ADP Pseudocode

Step 1: If we never saw  $s'$  before, add  $s'$  to set of states, and to tables  $R$  and  $U$ .

**function** **ADP**( $s, a, s', r', \pi, \gamma, K, \mathbb{S}, R, U, p, N_{sa}, N_{sat}$ )

**if** this is the first time we visit  $s'$ :

$$R[s'] = r'$$

$$U[s'] = r'$$

$\mathbb{S} = \mathbb{S} \cup \{s'\}$  // add  $s'$  to set of known states.

**if**  $s$  is not **null**: //  $s$  is null if no states have been visited before

**if**  $N_{sa}[s, a]$  exists:  $N_{sa}[s, a] += 1$  **else**:  $N_{sa}[s, a] = 1$

**if**  $N_{sat}[s, a, s']$  exists:  $N_{sat}[s, a, s'] += 1$  **else**:  $N_{sat}[s, a, s'] = 1$

**for each** state  $t$  such that  $N_{sat}[s, a, t] \neq 0$ :

$$p(t \mid s, a) = \frac{N_{sat}[s, a, t]}{N_{sa}[s, a]}$$

$U = \text{PolicyEvaluation}(\mathbb{S}, p, R, \gamma, \pi, K, U)$



# ADP Pseudocode

Step 2: update transition model  $p(t \mid s, a)$ .

**function** **ADP**( $s, a, s', r', \pi, \gamma, K, \mathbb{S}, R, U, p, N_{sa}, N_{sat}$ )

**if** this is the first time we visit  $s'$ :

$$R[s'] = r'$$

$$U[s'] = r'$$

$\mathbb{S} = \mathbb{S} \cup \{s'\}$  // add  $s'$  to set of known states.

**if**  $s$  is not **null**: //  $s$  is null if no states have been visited before

**if**  $N_{sa}[s, a]$  exists:  $N_{sa}[s, a] += 1$  **else**:  $N_{sa}[s, a] = 1$

**if**  $N_{sat}[s, a, s']$  exists:  $N_{sat}[s, a, s'] += 1$  **else**:  $N_{sat}[s, a, s'] = 1$

**for each** state  $t$  such that  $N_{sat}[s, a, t] \neq 0$ :

$$p(t \mid s, a) = \frac{N_{sat}[s, a, t]}{N_{sa}[s, a]}$$

$U = \text{PolicyEvaluation}(\mathbb{S}, p, R, \gamma, \pi, K, U)$

# ADP Pseudocode

Step 3: Call **PolicyEvaluation** to update utility table U.

**function ADP**( $s, a, s', r', \pi, \gamma, K, \mathbb{S}, R, U, p, N_{sa}, N_{sat}$ )

**if** this is the first time we visit  $s'$ :

$$R[s'] = r'$$

$$U[s'] = r'$$

$\mathbb{S} = \mathbb{S} \cup \{s'\}$  // add  $s'$  to set of known states.

**if**  $s$  is not **null**: //  $s$  is null if no states have been visited before

**if**  $N_{sa}[s, a]$  exists:  $N_{sa}[s, a] += 1$  **else**:  $N_{sa}[s, a] = 1$

**if**  $N_{sat}[s, a, s']$  exists:  $N_{sat}[s, a, s'] += 1$  **else**:  $N_{sat}[s, a, s'] = 1$

**for each** state  $t$  such that  $N_{sat}[s, a, t] \neq 0$ :

$$p(t \mid s, a) = \frac{N_{sat}[s, a, t]}{N_{sa}[s, a]}$$

**U = PolicyEvaluation**( $\mathbb{S}, p, R, \gamma, \pi, K, U$ )

# ADP Implementation Notes

- The algorithm uses several tables:
  - $R, U, p, N_{sa}, N_{sat}$ .
- In MDPs, we could implement these tables as arrays, because we knew the dimensions. For example:
  - The size of  $R$  and  $U$  was the number of states  $N$ .
  - The size of  $p$  was  $N \times N \times \text{number\_of\_actions}$ .
- In the ADP algorithm, **we do not know the number of states in advance**.
  - Thus, tables  $R, U, p, N_{sa}, N_{sat}$  should be stored using data structures that allow new elements to be added, like:
    - Dictionaries in Python.
    - HashMaps in Java.

# ADP Implementation Notes

- For the state transition model  $p$ :
  - In the pseudocode, we denote entries of  $p$  as  $p(t \mid s, a)$ .
  - In an actual implementation,  $p$  can just be a three-dimensional hash map (or any other appropriate structure) mapping values  $t, s, a$  to the current estimate of  $p(t \mid s, a)$ .

# ADP Implementation Notes

**if**  $N_{sa}[s, a]$  exists:  $N_{sa}[s, a] += 1$  **else**:  $N_{sa}[s, a] = 1$

**if**  $N_{sat}[s, a, t]$  exists:  $N_{sat}[s, a, t] += 1$  **else**:  $N_{sat}[s, a, t] = 1$

- When updating entries  $N_{sa}[s, a]$  and  $N_{sat}[s, a, t]$ , we need to check if those entries already exist.
  - If this is the first time we take action  $a$  from state  $s$ , then  $N_{sa}[s, a]$  has not been initialized yet.
  - If this is the first time that state  $t$  was the outcome of action  $a$  from state  $s$ , then  $N_{sat}[s, a, t]$  has not been initialized yet.
  - If any of those entries has not been initialized, we initialize it to a value of 1.
  - If any of those entries has been initialized, we increment it.

# End-to-End Estimation of Utilities

- ADP processes a single move of the agent at a time.
- How would we model an agent's behavior over time, if the agent uses ADP after each move?
- **Input:** values  $\pi, \gamma, K$  must be specified in advance.
- **Initialization:** initialize all variables appropriately.
- **Main loop** (running for ever):
  - Start from a legal initial state.
  - Repeat:
    - Use sensors to sense the current state and current reward.
    - Call **ADP** to update variables appropriately.
    - If the current state is non-terminal, pick and execute the next action.
  - Until the current state is a terminal state.

# Agent Model According to ADP

```
function AgentModelADP( $\pi, \gamma, K$ )
```

```
    // Initialization
```

```
    Initialize  $\mathbb{S}$  to an empty set
```

```
    Initialize  $U, p, N_{sa}, N_{sat}$  to empty tables
```

```
    while (true):    // Main loop: Execute mission after mission
```

```
        Initialize variables  $s, a$  to null    // no previous state
```

```
         $s'$  = initial state (chosen randomly, if multiple initial states exist)
```

```
        while (true):    // Execute one mission, from start to end
```

```
             $(s', r') = \text{SenseStateAndReward}()$ 
```

```
            ADP( $s, a, s', r', \pi, \gamma, K, \mathbb{S}, R, U, p, N_{sa}, N_{sat}$ )
```

```
            if  $s'$  is terminal: break    // Done with this mission, move on to next
```

```
             $a = \pi(s')$ 
```

```
            ExecuteAction( $a$ )
```

```
             $s = s'$ 
```

# Agent Model According to ADP

```
function AgentModelADP( $\pi, \gamma, K$ )
```

```
    // Initialization
```

```
    Initialize  $\mathbb{S}$  to an empty set
```

```
    Initialize  $U, p, N_{sa}, N_{sat}$  to empty tables
```

```
    while (true):    // Main loop: Execute mission after mission
```

```
        Initialize variables  $s, a$  to null    // no previous state
```

```
         $s'$  = initial state (chosen randomly, if multiple initial states exist)
```

```
        while (true):    // Execute one mission, from start to end
```

```
             $(s', r') = \text{SenseStateAndReward}()$ 
```

```
             $\text{ADP}(s, a, s', r', \pi, \gamma, K, \mathbb{S}, R, U, p, N_{sa}, N_{sat})$ 
```

```
            if  $s'$  is terminal: break    // Done with this mission, move on to next
```

```
             $a = \pi(s')$ 
```

```
            ExecuteAction( $a$ )
```

```
             $s = s'$ 
```

First, we initialize all variables appropriately.



# Agent Model According to ADP

**function AgentModelADP( $\pi, \gamma, K$ )**

**// Initialization**

Initialize  $\mathbb{S}$  to an empty set

Initialize  $U, p, N_{sa}, N_{sat}$  to empty tables

**while (true):**    **// Main loop: Execute mission after mission**

Initialize variables  $s, a$  to **null**    **// no previous state**

$s'$  = initial state (chosen randomly, if multiple initial states exist)

**while (true):**    **// Execute one mission, from start to end**

$(s', r') = \text{SenseStateAndReward}()$

**ADP( $s, a, s', r', \pi, \gamma, K, \mathbb{S}, R, U, p, N_{sa}, N_{sat}$ )**

**if  $s'$  is terminal: break**    **// Done with this mission, move on to next**

$a = \pi(s')$

**ExecuteAction( $a$ )**

$s = s'$

Main loop: Execute mission  
after mission, forever.

# Agent Model According to ADP

**function AgentModelADP( $\pi, \gamma, K$ )**

**// Initialization**

Initialize  $\mathbb{S}$  to an empty set

Initialize  $U, p, N_{sa}, N_{sat}$  to empty tables

**while (true):**     **// Main loop: Execute mission after mission**

Initialize variables  $s, a$  to **null**     **// no previous state**

$s'$  = initial state (chosen randomly, if multiple initial states exist)

**while (true):**     **// Execute one mission, from start to end**

$(s', r') = \text{SenseStateAndReward}()$

**ADP**( $s, a, s', r', \pi, \gamma, K, \mathbb{S}, R, U, p, N_{sa}, N_{sat}$ )

**if**  $s'$  is terminal: **break**     **// Done with this mission, move on to next**

$a = \pi(s')$

**ExecuteAction**( $a$ )

$s = s'$

To start the mission, move to a legal initial state.

# Agent Model According to ADP

```
function AgentModelADP( $\pi, \gamma, K$ )
```

```
  // Initialization
```

```
  Initialize  $\mathbb{S}$  to an empty set
```

```
  Initialize  $U, p, N_{sa}, N_{sat}$  to empty tables
```

```
  while (true):    // Main loop: Execute mission after mission
```

```
    Initialize variables  $s, a$  to null    // no previous state
```

```
     $s'$  = initial state (chosen randomly, if multiple initial states exist)
```

```
    while (true):  // Execute one mission, from start to end
```

```
       $(s', r') = \text{SenseStateAndReward}()$ 
```

```
       $\text{ADP}(s, a, s', r', \pi, \gamma, K, \mathbb{S}, R, U, p, N_{sa}, N_{sat})$ 
```

```
      if  $s'$  is terminal: break // Done with this mission, move on to next
```

```
       $a = \pi(s')$ 
```

```
       $\text{ExecuteAction}(a)$ 
```

```
       $s = s'$ 
```

The inner loop processes a single mission, from beginning to end.

# Agent Model According to ADP

```
function AgentModelADP( $\pi, \gamma, K$ )
```

```
  // Initialization
```

```
  Initialize  $\mathbb{S}$  to an empty set
```

```
  Initialize  $U, p, N_{sa}, N_{sat}$  to empty tables
```

```
  while (true):    // Main loop: Execute mission after mission
```

```
    Initialize variables  $s, a$  to null    // no previous state
```

```
     $s'$  = initial state (chosen randomly, if multiple initial states exist)
```

```
    while (true):  // Execute one mission, from start to end
```

```
       $(s', r') = \text{SenseStateAndReward}()$ 
```

```
       $\text{ADP}(s, a, s', r', \pi, \gamma, K, \mathbb{S}, R, U, p, N_{sa}, N_{sat})$ 
```

```
      if  $s'$  is terminal: break // Done with this mission, move on to next
```

```
       $a = \pi(s')$ 
```

```
      ExecuteAction( $a$ )
```

```
       $s = s'$ 
```

Use the sensors to sense current state and reward.

# Agent Model According to ADP

**function AgentModelADP( $\pi, \gamma, K$ )**

**// Initialization**

Initialize  $\mathbb{S}$  to an empty set

Initialize  $U, p, N_{sa}, N_{sat}$  to empty tables

**while (true):**     **// Main loop: Execute mission after mission**

Initialize variables  $s, a$  to **null**     **// no previous state**

$s'$  = initial state (chosen randomly, if multiple initial states exist)

**while (true):**     **// Execute one mission, from start to end**

$(s', r') = \text{SenseStateAndReward}()$

**ADP( $s, a, s', r', \pi, \gamma, K, \mathbb{S}, R, U, p, N_{sa}, N_{sat}$ )**

**if  $s'$  is terminal: break**     **// Done with this mission, move on to next**

$a = \pi(s')$

**ExecuteAction( $a$ )**

$s = s'$

Call **ADP** to update the model.

# Agent Model According to ADP

```
function AgentModelADP( $\pi, \gamma, K$ )
```

```
  // Initialization
```

```
  Initialize  $\mathbb{S}$  to an empty set
```

```
  Initialize  $U, p, N_{sa}, N_{sat}$  to empty tables
```

```
  while (true):    // Main loop: Execute mission after mission
```

```
    Initialize variables  $s, a$  to null    // no previous state
```

```
     $s'$  = initial state (chosen randomly, if multiple initial states exist)
```

```
    while (true):  // Execute one mission, from start to end
```

```
       $(s', r') = \text{SenseStateAndReward}()$ 
```

```
       $\text{ADP}(s, a, s', r', \pi, \gamma, K, \mathbb{S}, R, U, p, N_{sa}, N_{sat})$ 
```

```
      if  $s'$  is terminal: break // Done with this mission, move on to next
```

```
       $a = \pi(s')$ 
```

```
      ExecuteAction( $a$ )
```

```
       $s = s'$ 
```

If we reached a terminal state, we are done with this mission.

# Agent Model According to ADP

**function AgentModelADP( $\pi, \gamma, K$ )**

**// Initialization**

Initialize  $\mathbb{S}$  to an empty set

Initialize  $U, p, N_{sa}, N_{sat}$  to empty tables

**while (true):**     **// Main loop: Execute mission after mission**

Initialize variables  $s, a$  to **null**     **// no previous state**

$s'$  = initial state (chosen randomly, if multiple initial states exist)

**while (true):**     **// Execute one mission, from start to end**

$(s', r') = \text{SenseStateAndReward}()$

**ADP**( $s, a, s', r', \pi, \gamma, K, \mathbb{S}, R, U, p, N_{sa}, N_{sat}$ )

**if**  $s'$  is terminal: **break**     **// Done with this mission, move on to next**

$a = \pi(s')$

**ExecuteAction**( $a$ )

$s = s'$

Pick and execute  
the next action.

# Agent Model According to ADP

**function AgentModelADP( $\pi, \gamma, K$ )**

**// Initialization**

Initialize  $\mathbb{S}$  to an empty set

Initialize  $U, p, N_{sa}, N_{sat}$  to empty tables

**while (true):**     **// Main loop: Execute mission after mission**

Initialize variables  $s, a$  to **null**     **// no previous state**

$s'$  = initial state (chosen randomly, if multiple initial states exist)

**while (true):**     **// Execute one mission, from start to end**

$(s', r') = \text{SenseStateAndReward}()$

**ADP**( $s, a, s', r', \pi, \gamma, K, \mathbb{S}, R, U, p, N_{sa}, N_{sat}$ )

**if**  $s'$  is terminal: **break**     **// Done with this mission, move on to next**

$a = \pi(s')$

**ExecuteAction**( $a$ )

$s = s'$

Store in  $s$  the value of the previous state.



# Agent Model According to ADP

- The **AgentModelADP** algorithm models an agent that behaves according to a **fixed** policy  $\pi$ , and that updates its utility estimates using **ADP**.
- The agent interacts with the environment using two functions:
  - **SenseStateAndReward()**, for getting the current state and reward from sensors.
  - **ExecuteAction( $a$ )**, for executing a specific action.
- These functions can be simulated, if we want to simply test the algorithm in software (for example, for a homework assignment).

# Temporal-Difference Learning

- **Temporal-Difference Learning (TDL)** is an alternative to ADP for solving the passive reinforcement learning problem.
- Key difference: complicated vs. simple update at each step.
- ADP does a more complicated update, where the **PolicyEvaluation** function is called to update the utilities of all known states.
- TDL does a very simple update, it only changes the utility value of the previous state.

# Temporal-Difference Learning

- The TDL algorithm takes mostly the same arguments as ADP.
- There are **input** arguments, whose value is fixed:
  - $(s, r, a)$ : the previous state  $s$ , the reward  $r$  obtained at state  $s$ , the previous action  $a$  that led from  $s$  to  $s'$ .
  - $(s', r')$ : the current state  $s'$ , and the reward  $r'$  received at state  $s'$ .
  - $\pi$ : the policy mapping each state  $s$  to an action  $\pi(s)$ .
  - $\gamma$ : the discount factor.
  - $\eta$ : a **function**, specifying a **learning rate that decreases over time**.
- There are also **input/output** arguments, whose values can change.
  - $R, U$ : the reward and utility tables, storing rewards and utilities of states.
  - $N_s$ : a table, where  $N_s[s]$  counts all times state  $s$  has been visited.

# TDL Pseudocode

```
function TDL( $s, r, a, s', r', \pi, \gamma, \eta, R, U, N_s$ )
```

```
  if this is the first time we visit  $s'$ :
```

```
     $R[s'] = r'$ 
```

```
     $U[s'] = r'$ 
```

```
  if  $s$  is not null: //  $s$  is null if no states have been visited before
```

```
    if  $N_s[s]$  exists:  $N_s[s] += 1$  else:  $N_s[s] = 1$ 
```

```
     $c = \eta(N_s[s])$  // we call function  $\eta$  with input  $N_s[s]$ 
```

```
     $U[s] = (1 - c)U[s] + c(R[s] + \gamma U[s'])$ 
```

- Step 1:
  - If we never saw  $s'$  before, add values for  $s'$  to tables  $R$  and  $U$ .

# TDL Pseudocode

```
function TDL( $s, r, a, s', r', \pi, \gamma, \eta, R, U, N_s$ )  
  if this is the first time we visit  $s'$ :  
     $R[s'] = r'$   
     $U[s'] = r'$   
  if  $s$  is not null: //  $s$  is null if no states have been visited before  
    if  $N_s[s]$  exists:  $N_s[s] += 1$  else:  $N_s[s] = 1$   
     $c = \eta(N_s[s])$  // we call function  $\eta$  with input  $N_s[s]$   
     $U[s] = (1 - c)U[s] + c(R[s] + \gamma U[s'])$ 
```

- Step 2:
  - Increment  $N_s[s]$  to indicate that we have visited state  $s$  one more time.

# TDL Pseudocode

```
function TDL( $s, r, a, s', r', \pi, \gamma, \eta, R, U, N_s$ )  
  if this is the first time we visit  $s'$ :  
     $R[s'] = r'$   
     $U[s'] = r'$   
  if  $s$  is not null: //  $s$  is null if no states have been visited before  
    if  $N_s[s]$  exists:  $N_s[s] += 1$  else:  $N_s[s] = 1$   
     $c = \eta(N_s[s])$  // we call function  $\eta$  with input  $N_s[s]$   
     $U[s] = (1 - c)U[s] + c(R[s] + \gamma U[s'])$ 
```

- Step 3:
  - Compute the learning rate to be used at the current step.
  - **Do not make the mistake to read  $\eta(N_s[s])$  as multiplication.**
  - $\eta(N_s[s])$  is a function call. We call function  $\eta$ , with argument  $N_s[s]$ .
  - Function  $\eta$  must be chosen appropriately to guarantee that **TDL** updates converge eventually to the right utility values.

# TDL Pseudocode

```
function TDL( $s, r, a, s', r', \pi, \gamma, \eta, R, U, N_s$ )  
  if this is the first time we visit  $s'$ :  
     $R[s'] = r'$   
     $U[s'] = r'$   
  if  $s$  is not null: //  $s$  is null if no states have been visited before  
    if  $N_s[s]$  exists:  $N_s[s] += 1$  else:  $N_s[s] = 1$   
     $c = \eta(N_s[s])$  // we call function  $\eta$  with input  $N_s[s]$   
     $U[s] = (1 - c)U[s] + c(R[s] + \gamma U[s'])$ 
```

- Step 3:
  - We will not prove it, but it is sufficient if function  $\eta$  decays as fast as  $\Theta\left(\frac{1}{n}\right)$ .
  - For the  $3 \times 4$  grid example, the Russell & Norvig textbook defines  $\eta$  as:

$$\eta(n) = \frac{60}{59 + n}$$

# TDL Pseudocode

```
function TDL( $s, r, a, s', r', \pi, \gamma, \eta, R, U, N_s$ )  
  if this is the first time we visit  $s'$ :  
     $R[s'] = r'$   
     $U[s'] = r'$   
  if  $s$  is not null: //  $s$  is null if no states have been visited before  
    if  $N_s[s]$  exists:  $N_s[s] += 1$  else:  $N_s[s] = 1$   
     $c = \eta(N_s[s])$  // we call function  $\eta$  with input  $N_s[s]$   
     $U[s] = (1 - c)U[s] + c(R[s] + \gamma U[s'])$ 
```

- Step 4: Adjust  $U[s]$ , by taking a **weighted average** of:
  - The previous estimate for  $U[s]$ , with weight  $(1 - c) = (1 - \eta(N_s[s]))$ .
  - Estimate  $R[s] + \gamma U[s']$ , which would be true if **the successor of  $s$  were always  $s'$** , with weight  $c = \eta(N_s[s])$ .
- Even though this update is very simple, values  $U[s]$  converge (over multiple calls to **TDL**) to the correct value.



# A Closer Look at the TDL Update

$$U[s] = (1 - c)U[s] + c(R[s] + \gamma U[s'])$$

- The TDL update equation is shown above.
  - $s$  is the previous state.
  - $s'$  is the current state, produced by performing some action  $a$ .
- Sometimes, unlikely transitions occur.
  - It may be that  $s'$  was a low-probability outcome for action  $a$  and state  $s$ .
- In that case,  $U[s]$  may become less accurate after the update.
  - If this is the first time that the agent visited state  $s$ , then the  $U[s]$  value may end up being far from the correct value.
- However: as long as the learning rate decreases appropriately over time,  $U[s]$  will converge to the right value.
  - Unlikely transitions happen rarely, so, over time, they affect the  $U[s]$  value only a small fraction of the times.

# Agent Model According to TDL

```
function AgentModelTDL( $\pi, \gamma, \eta$ )
```

```
// Initialization
```

```
Initialize  $R, U, N_s$  to empty tables
```

```
while (true):    // Main loop
```

```
    Initialize variables  $s, r, a$  to null
```

```
     $s'$  = initial state (chosen randomly, if multiple initial states exist)
```

```
    while (true):    // Execute one mission, from start to end
```

```
         $(s', r') = \text{SenseStateAndReward}()$ 
```

```
        TDL( $s, r, a, s', r', \pi, \gamma, \eta, R, U, N_s$ )
```

```
        if  $s'$  is terminal: break // Done with this mission
```

```
         $a = \pi(s')$ 
```

```
        ExecuteAction( $a$ )
```

```
         $(s, r) = (s', r')$ 
```

End-to-end model of an agent that uses TDL.

Similar in logic to **AgentModelADP**.

# ADP vs. TDL

- ADP spends more time and effort in its updates.
  - It calls **PolicyEvaluation** to update utilities as much as possible using the new information.
- TDL does a rather minimal update.
  - It just updates the utility of the previous state, taking a weighted average of:
    - the previous estimate for the utility of the previous state
    - the estimated utility that was obtained as a result of the last action.
- The pros and cons are rather obvious:
  - ADP takes more time to process a single step, and estimates converge after fewer steps.
  - TDL is faster to execute for a single step, but estimates need more steps to converge, compared to ADP.

# Active Reinforcement Learning

- Active Reinforcement Learning is the problem of actually figuring out what to do.
  - The policy is not given.
  - Rewards are not known in advance.
  - The transition model is not known in advance.
- MDPs and Passive Reinforcement Learning solved easier problems.
  - In MDPs, rewards and transitions are known.
  - In passive reinforcement learning, the policy is given and the agent just wants to estimate the utility of each state.

# A Greedy Approach

- Main loop: at each step
  - Use sensors to sense the current state  $s'$  and current reward  $r'$ .
  - Call **ADP** to update the utility estimates  $U[s]$ .
  - Choose the next action using the same equation that we used in MDPs:

$$a = \operatorname{argmax}_{a \in A(s')} \left\{ \sum_t [p(t | s', a) U[t]] \right\}$$

- The key idea is that, at each step, the agent picks the best action it can find, according to its current model.
  - Since the current model may not be correct, the action is not necessarily the truly best action.
  - However, the model keeps getting updated at each step.
- Does this approach eventually converge to the optimal policy?

# A Greedy Approach

- Main loop: at each step
  - Use sensors to sense the current state  $s'$  and current reward  $r'$ .
  - Call **ADP** to update the utility estimates  $U[s]$ .
  - Choose the next action using the same equation that we used in MDPs:

$$a = \operatorname{argmax}_{a \in A(s')} \left\{ \sum_t [p(t | s', a) U[t]] \right\}$$

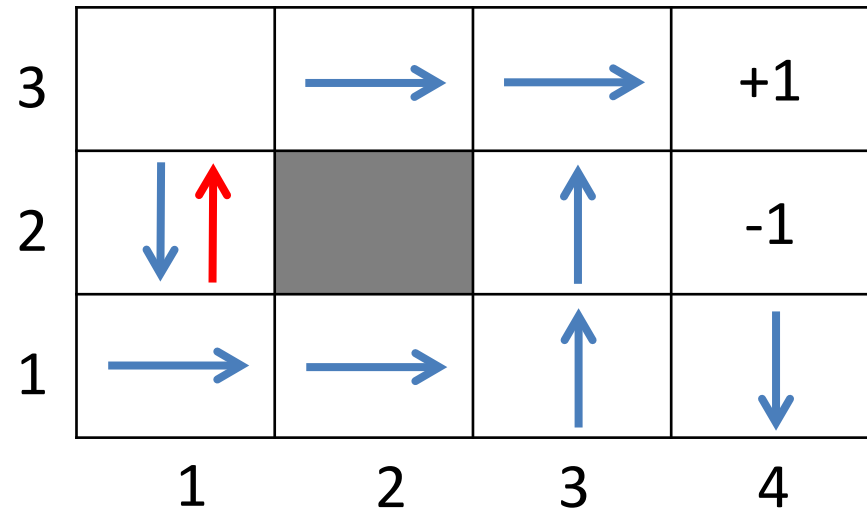
- The key idea is that, at each step, the agent picks the best action it can find, according to its current model.
  - Since the current model may not be correct, the action is not necessarily the truly best action.
  - However, the model keeps getting updated at each step.
- Does this approach eventually converge to the optimal policy?
  - Unfortunately, no.

# Problem with the Greedy Approach

3		→	→	+1
2	↓ ↑		↑	-1
1	→	→	↑	↓
	1	2	3	4

- This example illustrates the problem with the previous approach.
- In state (2,1), according to the current (inaccurate) model, the optimal action is "go down".
  - State (3,1) has never been visited, so its estimated utility  $U[(3,1)]$  is 0.
- The true optimal action is "go up" (shown with the red arrow).
- However, the agent will never visit state (3,1), because none of the actions in the current policy can possibly lead to state (3,1).
- As a result, estimated utility  $U[(3,1)]$  will never be updated.
- Thus, the agent will never figure out that "go up" is the best choice for state (2,1).

# Exploration and Exploitation



- The greedy approach, where the agent always chooses what seems to be the best action, that approach is called **exploitation**.
  - In that case, the agent may never figure out what the best action is.
- The only way to solve this problem is to allow some **exploration**.
  - Every now and then, the agent should take actions that, according to its current model, are not the best actions to take.
  - This way the agent can, eventually, identify better choices that it was not aware of at first.



# How Much Exploration?

- Allowing zero exploration makes it possible that the agent will never figure out an optimal (or a good) policy.
- Doing too much exploration means that the agent, while learning a lot, keeps making bad choices and accumulating poor rewards.
- One approach is to choose, at each step  $t$ , a random action with probability  $\frac{1}{t}$ , so that this probability decreases over time.
- This way, in our example, action "go up" will eventually be chosen from state (2,1), state (3,1) will eventually be visited, and the policy will eventually converge to the optimal policy.

3		→	→	+1
2	↓ ↑		↑	-1
1	→	→	↑	↓
	1	2	3	4

# How Much Exploration?

- Allowing zero exploration makes it possible that the agent will never figure out an optimal (or a good) policy.
- Doing too much exploration means that the agent, while learning a lot, keeps making bad choices and accumulating poor rewards.
- One approach is to choose, at each step  $t$ , a random action with probability  $\frac{1}{t}$ , so that this probability decreases over time.
- If we follow this approach, it can be proven (we skip the proof) that the agent's policy eventually converges to the optimal policy.
- However, convergence is relatively slow.

3		→	→	+1
2	↓ ↑		↑	-1
1	→	→	↑	↓
	1	2	3	4

# Encouraging Exploration

- Another approach is to artificially bump up the utility values of states that have not been visited much.
- We denote these "bumped-up" utilities as  $U^+[s]$ :

$$U^+(s) = R(s) + \gamma \max_{a \in A(s)} f \left( \sum_{s'} [p(s' | s, a) U^+(s')], N_{sa}[s, a] \right)$$

- In the above equation, note the use of function  $f(u, n)$ :
  - $u$  is the estimated utility value of taking action  $a$  at state  $s$ .
  - $n$  is the number of times the agent has taken action  $a$  at state  $s$ .
- The job of  $f$  is to "bump up" utility values of (state, action) pairs that have not been tried many times.

# Encouraging Exploration

$$U^+(s) = R(s) + \gamma \max_{a \in A(s)} f \left( \sum_{s'} [p(s' | s, a) U^+(s')], N_{sa}[s, a] \right)$$

- Function  $f$  can be defined in different ways. For example:

$$f(u, n) = \begin{cases} R^+ & \text{if } n < N_e \\ u & \text{otherwise} \end{cases}$$

- In the above definition:
  - $R^+$  is the maximum possible reward in the environment.
  - $N_e$  is a hard-coded parameter, provided as input to the algorithm.
- With this definition:
  - If the agent has not tried action  $a$  from state  $s$  many times, then  $N_{sa}[s, a] < N_e$ , and function  $f$  returns  $R^+$  which is the max possible reward.
  - Otherwise, function  $f$  returns the more realistic utility estimate  $u$ .

# Q-Learning

- Q-Learning is a method for active reinforcement learning.
- It is based on temporal-difference learning.
- Q-Learning learns a function  $Q(s, a)$ , which is defined as the utility obtained by performing action  $a$  from state  $s$ :

$$Q(s, a) = R(s) + \gamma \sum_{s'} [p(s' | s, a) U(s')]$$

# Q-Learning

- Q-Learning is a method for active reinforcement learning.
- It is based on temporal-difference learning.
- Q-Learning learns a function  $Q(s, a)$ , which is defined as the utility obtained by performing action  $a$  from state  $s$ :

$$Q(s, a) = R(s) + \gamma \sum_{s'} [p(s' | s, a) U(s')]$$

Reward at state  $s$

Expected utility of  
outcomes of action  $a$

# Q-Learning

- Q-Learning is a method for active reinforcement learning.
- It is based on temporal-difference learning.
- Q-Learning learns a function  $Q(s, a)$ , which is defined as the utility obtained by performing action  $a$  from state  $s$ :

$$Q(s, a) = R(s) + \gamma \sum_{s'} [p(s' | s, a) U(s')]$$

- Q-values are obviously related to state utilities:

$$U(s) = \max_a Q(s, a)$$

# Q-Learning

- Definition of the Q function:

$$Q(s, a) = R(s) + \gamma \sum_{s'} [p(s' | s, a) U(s')]$$

- Relation between Q values and utilities of states:

$$U(s) = \max_a Q(s, a)$$

- By substituting  $\max_{a'} Q(s', a')$  for  $U(s')$ , we get

$$Q(s, a) = R(s) + \gamma \sum_{s'} [p(s' | s, a) \max_{a'} Q(s', a')] ]$$



# Q-Learning Update Step

- The Q-Learning method updates Q values after each step.
- There are **input** arguments, whose value is fixed:
  - $(s, r, a)$ : the previous state  $s$ , the reward  $r$  obtained at state  $s$ , the previous action  $a$  that led from  $s$  to  $s'$ .
  - $(s', r')$ : the current state  $s'$ , and the reward  $r'$  received at state  $s'$ .
  - $\gamma$ : the discount factor.
  - $\eta$ : a **function**, specifying a **learning rate that decreases over time**.
- There are also **input/output** arguments, whose values can change.
  - $Q$ : the table of Q-values, storing utilities of (state, action) pairs.
  - $N_{sa}$ : a table, where  $N_{sa}[s, a]$  counts all times state  $s$  has been visited.

# Q-Learning Update: Pseudocode

```
function Q_Learning_Update( $s, r, a, s', r', \gamma, \eta, Q, N_{sa}$ )  
    if  $s'$  is a terminal state:  
         $Q[s', \text{None}] = r'$   
    if  $s$  is not null: //  $s$  is null if no states have been visited before  
        if  $N_{sa}[s, a]$  exists:  $N_{sa}[s, a] += 1$   
        else:  $N_{sa}[s, a] = 1$   
         $c = \eta(N_{sa}[s, a])$  // we call function  $\eta$  with input  $N_{sa}[s, a]$   
         $Q[s, a] = (1 - c)Q[s, a] + c(r + \gamma \max_{a'} Q[s', a'])$ 
```

- The update step, similar to TD-Learning, computes the weighted average of:
  - The previous estimate for  $Q[s, a]$ .
  - $r + \gamma \max_{a'} Q[s', a']$ , which is the expected utility given that the successor of  $s$  is  $s'$ .

# Agent Model for Q-Learning

```
function AgentModel_Q_Learning( $\gamma, \eta$ )
```

```
    // Initialization
```

```
    Initialize  $Q, N_{sa}$  to empty tables,
```

```
    while (true)    // Main loop: Execute mission after mission
```

```
        Initialize variables  $s, r, a$  to null    // no previous state
```

```
         $s'$  = initial state (chosen randomly, if multiple initial states exist)
```

```
        while (true):    // Execute one mission, from start to end
```

```
            ( $s', r'$ ) = SenseStateAndReward()
```

```
            Q_Learning_Update( $s, r, a, s', r', \gamma, \eta, Q, N_{sa}$ )
```

```
            if  $s'$  is terminal: break    // Done with this mission
```

```
             $a = \underset{a' \in A(s')}{\operatorname{argmax}} f(Q[s', a'], N_{sa}[s', a'])$ 
```

```
            ExecuteAction( $a$ )
```

```
            ( $s, r$ ) = ( $s', r'$ )
```

The line in red makes this version of RL **active** as opposed to **passive**.

# Choosing Actions with Q-Learning

- In the **AgentModelADP** and **AgentModelTDL** pseudocode, the next action was chosen using this line:

$$a = \pi(s')$$

- In those cases, the next action is provided by a **fixed** policy  $\pi$ .

- In the **AgentModel\_Q\_Learning** pseudocode, the action is chosen using this line:

$$a = \operatorname{argmax}_{a' \in A(s')} f(Q[s', a'], N_{sa}[s', a'])$$

- The next action is chosen taking into account the updated  $Q[s', a']$  values.
- Thus, the policy that the agent follows is not fixed. As values  $Q[s', a']$  change, the winning action for each state can also change.

# Choosing Actions with Q-Learning

- In the **AgentModel\_Q\_Learning** pseudocode, the action is chosen using this line:

$$a = \operatorname{argmax}_{a' \in A(s')} f(Q[s', a'], N_{sa}[s', a'])$$

- For the next action, we don't just choose the action that maximizes the estimated utility  $Q[s', a']$ .
  - Doing that would allow for zero exploration.
- Instead, we choose the action that maximizes  $f(Q[s', a'], N_{sa}[s', a'])$ .
- As we saw before, the role of function  $f$  is to encourage exploration, by giving higher values to  $(s', a')$  pairs that have not been explored a sufficient number of times.

# Generalization in RL

- The Q-Learning method is a solution to the active reinforcement learning.
- The complexity of that solution is at least linear to the number of states.
  - We cannot expect utility estimates to be accurate for most states, unless the agent visits most states.
- In some cases, linear time complexity is prohibitive.
  - The number of states for backgammon is estimated at  $10^{20}$ .
  - The number of states for chess is estimated at  $10^{40}$ .
- We need a way to estimate utilities of states that have never been visited.

# Function Approximation

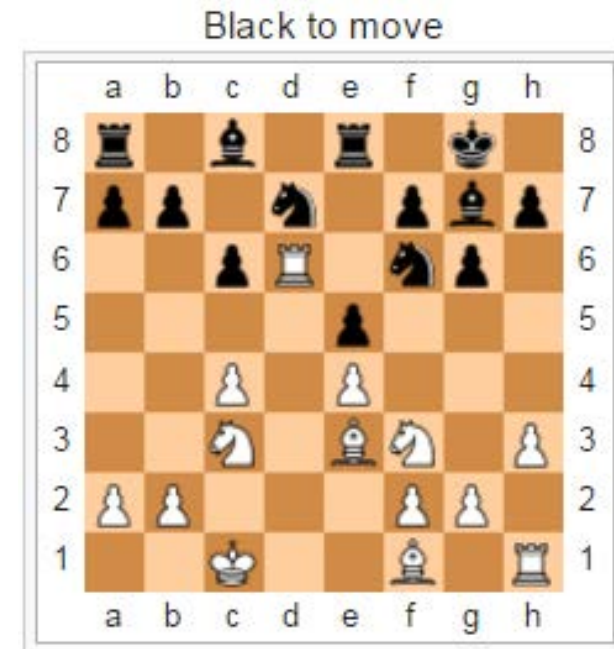
- We can approximate a utility function, or a Q-function, using a parametric function.
- For example: we can approximate the Q-function using a function  $\hat{Q}_\theta$  that is a linear combination of a set of features:

$$\hat{Q}_\theta(s, a) = \theta_1 f_1(s, a) + \theta_2 f_2(s, a) + \cdots + \theta_n f_n(s, a)$$

- The parameter vector  $\theta = (\theta_1, \dots, \theta_n)$  is a vector of real numbers, that get optimized during learning.
- The  $f_i$  functions are basis functions, extracting features from a state-action pair.

# Basis Functions for Chess

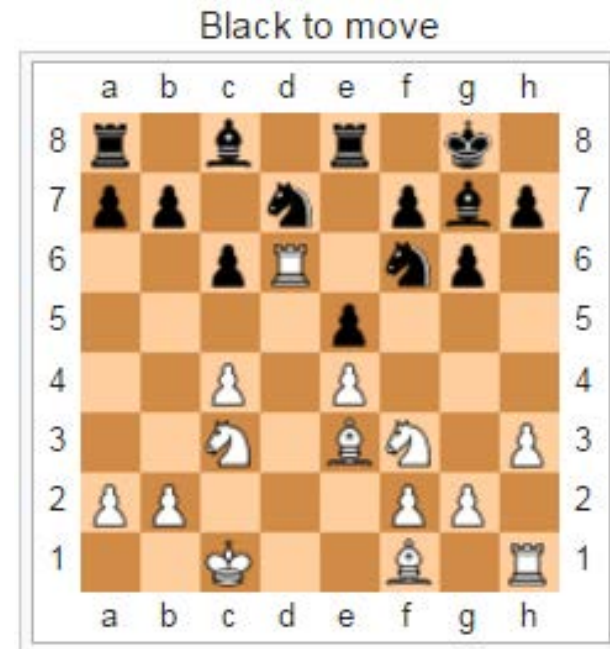
- As an example, we can look at one way to define basis functions for chess.
- A state  $s$  is defined by:
  - The position of pieces on the board.
  - A number indicating whose turn it is.
    - e.g., 1 for white,  $-1$  for black.
- An action  $a$  is a move that can be played at state  $s$ .
  - We denote by  $G(s, a)$  the resulting state if we play move  $a$  at state  $s$ .





# Basis Functions for Chess

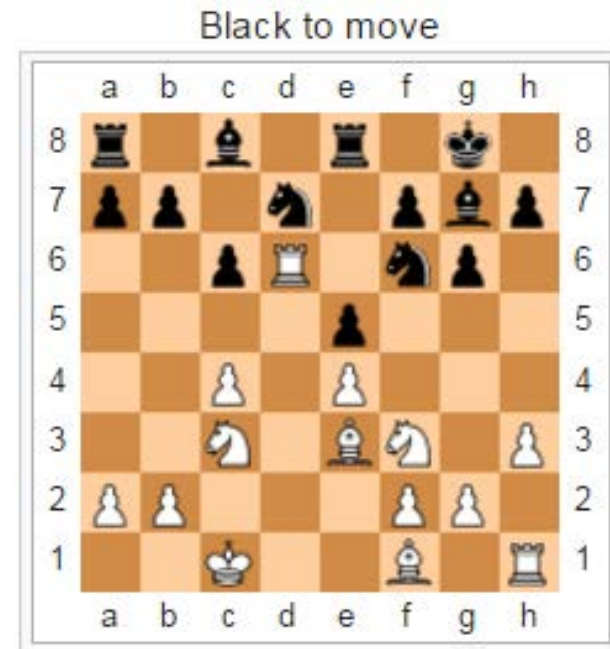
- We can define some basis functions  $f_i(s, a)$  as follows:
  - $f_1(s, a)$  = number of white queens in  $s$ .
  - $f_2(s, a)$  = number of black queens in  $s$ .
  - $f_3(s, a)$  = number of white bishops in  $s$ .
  - $f_4(s, a)$  = number of black bishops in  $s$ .
  - $f_5(s, a)$  = number of white knights in  $s$ .
  - $f_6(s, a)$  = number of black knights in  $s$ .
  - $f_7(s, a)$  = number of white rooks in  $s$ .
  - $f_8(s, a)$  = number of black rooks in  $s$ .
  - $f_9(s, a)$  = number of white pawns in  $s$ .
  - $f_{10}(s, a)$  = number of black pawns in  $s$ .



Basis functions  $f_1$  to  $f_{10}$  extract features from the state  $s$ , but tell us nothing about the action  $a$ .

# Basis Functions for Chess

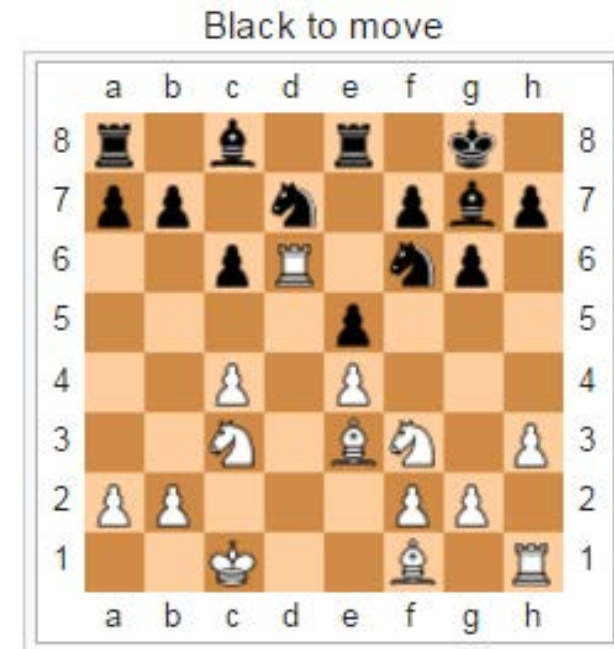
- We can define some basis functions  $f_i(s, a)$  as follows:
  - $f_{11}(s, a) = \#$  of white queens in  $G(s, a)$ .
  - $f_{12}(s, a) = \#$  of black queens in  $G(s, a)$ .
  - $f_{13}(s, a) = \#$  of white bishops in  $G(s, a)$ .
  - $f_{14}(s, a) = \#$  of black bishops in  $G(s, a)$ .
  - $f_{15}(s, a) = \#$  of white knights in  $G(s, a)$ .
  - $f_{16}(s, a) = \#$  of black knights in  $G(s, a)$ .
  - $f_{17}(s, a) = \#$  of white rooks in  $G(s, a)$ .
  - $f_{18}(s, a) = \#$  of black rooks in  $G(s, a)$ .
  - $f_{19}(s, a) = \#$  of white pawns in  $G(s, a)$ .
  - $f_{20}(s, a) = \#$  of black pawns in  $G(s, a)$ .



Basis functions  $f_{11}$  to  $f_{20}$  extract features from state  $G(s, a)$ , which is the result of performing action  $a$  from state  $s$ .

# Basis Functions for Chess

- The previous basis functions are far from exhaustive.
- We can define more basic functions, to capture other important aspects of a state, such as:
  - Which pieces are threatened.
  - Which pieces are protected by other pieces.
  - The number of legal moves available to each player.
  - ...



# Learning a Parametric Q-Function

$$\hat{Q}_{\theta}(s, a) = \theta_1 f_1(s, a) + \theta_2 f_2(s, a) + \cdots + \theta_n f_n(s, a)$$

- Once we have decided on the basis functions to use, we can use Q-Learning to learn the parameters  $\theta_i$ .
- In the original version of Q-Learning, the update rule was:

$$Q[s, a] = (1 - c)Q[s, a] + c(r + \gamma \max_{a'} Q(s', a'))$$

- Now that we are using parametric function  $\hat{Q}_{\theta}$ , we can optimize parameters  $\theta_i$  using gradient descent:
  - Define an error function  $E(s, a, \theta_1, \dots, \theta_n)$
  - Compute derivatives  $\frac{\partial E}{\partial \theta_i}$ .
  - Move each  $\theta_i$  away from the direction of the gradient:  $\theta_i = \theta_i - c \frac{\partial E}{\partial \theta_i}$ .

# Learning a Parametric Q-Function

$$\hat{Q}_{\theta}(s, a) = \theta_1 f_1(s, a) + \theta_2 f_2(s, a) + \cdots + \theta_n f_n(s, a)$$

- We define the error  $E(s, a, \theta)$  as:

$$E(s, a, \boldsymbol{\theta}) = \frac{1}{2} \left( \underbrace{\hat{Q}_{\theta}(s, a)}_{\text{prediction}} - \underbrace{\left( R(s) + \gamma \max_{a'} \hat{Q}_{\theta}(s', a') \right)}_{\text{observation}} \right)^2$$

- This is a standard definition of squared error, where:
- $\hat{Q}_{\theta}(s, a)$  is the **prediction** of the utility for pair  $(s, a)$ , according to parameter vector  $\boldsymbol{\theta}$ .
- $R(s) + \gamma \max_{a'} \hat{Q}_{\theta}(s', a')$  is the **observed utility** for pair  $(s, a)$ , according to the last step, where action  $a$  led from state  $s$  to state  $s'$ .

# Learning a Parametric Q-Function

$$\hat{Q}_{\theta}(s, a) = \theta_1 f_1(s, a) + \theta_2 f_2(s, a) + \cdots + \theta_n f_n(s, a)$$

- We define the error  $E(s, a, \theta)$  as:

$$E(s, a, \theta) = \frac{1}{2} \left( \underbrace{\hat{Q}_{\theta}(s, a)}_{\text{prediction}} - \underbrace{\left( R(s) + \gamma \max_{a'} \hat{Q}_{\theta}(s', a') \right)}_{\text{observation}} \right)^2$$

- In this error definition, if we were doing supervised learning, instead of the observation we would use the ground truth.
- In reinforcement learning, we do not have ground truth, so we treat each observation as ground truth.
- This may lead to a few inaccurate updates, but in the long term the estimate will converge to the best approximation.

# Learning a Parametric Q-Function

$$\hat{Q}_{\theta}(s, a) = \theta_1 f_1(s, a) + \theta_2 f_2(s, a) + \dots + \theta_n f_n(s, a)$$
$$E(s, a, \boldsymbol{\theta}) = \frac{1}{2} \left( \underbrace{\hat{Q}_{\theta}(s, a)}_{\text{prediction}} - \underbrace{\left( R(s) + \gamma \max_{a'} \hat{Q}_{\theta}(s', a') \right)}_{\text{observation}} \right)^2$$

- The next step is to compute  $\frac{\partial E}{\partial \theta_i}$ . We use the chain rule.
- Define:
  - $g(x) = \frac{1}{2} x^2$
  - $h(s, a, \boldsymbol{\theta}) = \hat{Q}_{\theta}(s, a) - \left( R(s) + \gamma \max_{a'} \hat{Q}_{\theta}(s', a') \right)$
- Then,  $E(s, a, \boldsymbol{\theta}) = g(h(s, a, \boldsymbol{\theta}))$ , and  $\frac{\partial E}{\partial \theta_i} = \frac{\partial g}{\partial h} \frac{\partial h}{\partial \theta_i}$

# Learning a Parametric Q-Function

$$\hat{Q}_{\theta}(s, a) = \theta_1 f_1(s, a) + \theta_2 f_2(s, a) + \dots + \theta_n f_n(s, a)$$
$$E(s, a, \boldsymbol{\theta}) = \frac{1}{2} \left( \underbrace{\hat{Q}_{\theta}(s, a)}_{\text{prediction}} - \underbrace{\left( R(s) + \gamma \max_{a'} \hat{Q}_{\theta}(s', a') \right)}_{\text{observation}} \right)^2$$

- $g(h) = \frac{1}{2} h^2$
- $h(s, a, \boldsymbol{\theta}) = \hat{Q}_{\theta}(s, a) - \left( R(s) + \gamma \max_{a'} \hat{Q}_{\theta}(s', a') \right)$
- $\frac{\partial g}{\partial h} = \frac{1}{2} 2h = h$
- So,  $\frac{\partial E}{\partial \theta_i} = \frac{\partial g}{\partial h} \frac{\partial h}{\partial \theta_i} = h \frac{\partial h}{\partial \theta_i}$



# Learning a Parametric Q-Function

$$\hat{Q}_\theta(s, a) = \theta_1 f_1(s, a) + \theta_2 f_2(s, a) + \dots + \theta_n f_n(s, a)$$
$$E(s, a, \theta) = \frac{1}{2} \left( \underbrace{\hat{Q}_\theta(s, a)}_{\text{prediction}} - \underbrace{\left( R(s) + \gamma \max_{a'} \hat{Q}_\theta(s', a') \right)}_{\text{observation}} \right)^2$$

- $h(s, a, \theta) = \hat{Q}_\theta(s, a) - \left( R(s) + \gamma \max_{a'} \hat{Q}_\theta(s', a') \right)$
- To compute  $\frac{\partial h}{\partial \theta_i}$ , we treat the observation as independent of  $\theta_i$ .
  - It is not really a constant, since  $\theta_i$  affects  $\hat{Q}_\theta(s', a')$ .
  - However, treating the observation as independent of  $\theta_i$  simplifies calculations, and does not hurt accuracy of the final Q-learning result.

# Learning a Parametric Q-Function

$$\hat{Q}_{\theta}(s, a) = \theta_1 f_1(s, a) + \theta_2 f_2(s, a) + \cdots + \theta_n f_n(s, a)$$
$$E(s, a, \boldsymbol{\theta}) = \frac{1}{2} \left( \underbrace{\hat{Q}_{\theta}(s, a)}_{\text{prediction}} - \underbrace{\left( R(s) + \gamma \max_{a'} \hat{Q}_{\theta}(s', a') \right)}_{\text{observation}} \right)^2$$

- $h(s, a, \boldsymbol{\theta}) = \hat{Q}_{\theta}(s, a) - \left( R(s) + \gamma \max_{a'} \hat{Q}_{\theta}(s', a') \right)$
- To compute  $\frac{\partial h}{\partial \theta_i}$ , we treat the observation as independent of  $\theta_i$ .
- Thus,  $\frac{\partial h}{\partial \theta_i} = \frac{\partial \hat{Q}_{\theta}(s, a)}{\partial \theta_i} = f_i(s, a)$ .

# Learning a Parametric Q-Function

$$\hat{Q}_{\theta}(s, a) = \theta_1 f_1(s, a) + \theta_2 f_2(s, a) + \dots + \theta_n f_n(s, a)$$
$$E(s, a, \boldsymbol{\theta}) = \frac{1}{2} \left( \underbrace{\hat{Q}_{\theta}(s, a)}_{\text{prediction}} - \underbrace{\left( R(s) + \gamma \max_{a'} \hat{Q}_{\theta}(s', a') \right)}_{\text{observation}} \right)^2$$

- Combining our previous results we get:

$$\frac{\partial E}{\partial \theta_i} = \left( \hat{Q}_{\theta}(s, a) - \left( R(s) + \gamma \max_{a'} \hat{Q}_{\theta}(s', a') \right) \right) f_i(s, a)$$

# Learning a Parametric Q-Function

$$\frac{\partial E}{\partial \theta_i} = \left( \hat{Q}_\theta(s, a) - \left( R(s) + \gamma \max_{a'} \hat{Q}_\theta(s', a') \right) \right) f_i(s, a)$$

- So, for Q-Learning, the update rule becomes:

$$\theta_i = \theta_i - c \frac{\partial E}{\partial \theta_i}, \text{ or equivalently}$$

$$\theta_i = \theta_i - c \left( \hat{Q}_\theta(s, a) - \left( R(s) + \gamma \max_{a'} \hat{Q}_\theta(s', a') \right) \right) f_i(s, a)$$

- As usual when using gradient descent, we are moving parameter vector  $\boldsymbol{\theta}$  a small step away from the direction of the error gradient.
  - $c$  is the size of that step.
  - $c$  itself is calculated as  $c = \eta(N_{sa}[s, a])$ , and decreases as  $N_{sa}[s, a]$  increases.

# Reinforcement Learning - Recap

- The goal in reinforcement learning is to learn what action to take at each state, so as to maximize the expected utility of the agent.
  - The states themselves are unknown until they are observed.
  - Rewards of states are unknown until they are observed.
  - The state transition model, i.e., the probability  $p(s' \mid s, a)$  that taking action  $a$  at state  $s$  leads to state  $s'$ , is unknown.

# Reinforcement Learning - Recap

- To solve the reinforcement learning problem, we solved a sequence of easier problems:
  - The MDP problem: rewards and transition probabilities are known in advance.
  - Passive reinforcement learning: rewards and transition probabilities are unknown, but the policy is fixed.
    - The agent updates utility estimates after each step, and the estimates converge to the correct values eventually.
  - Q-Learning: learns optimal policy. Rewards and transition probabilities are unknown.
    - The agent updates utility estimates after each step, and picks the next action balancing those estimates and the need to explore.
  - Parametric Q-Learning: allows describing utilities in large state spaces with relatively few parameters.