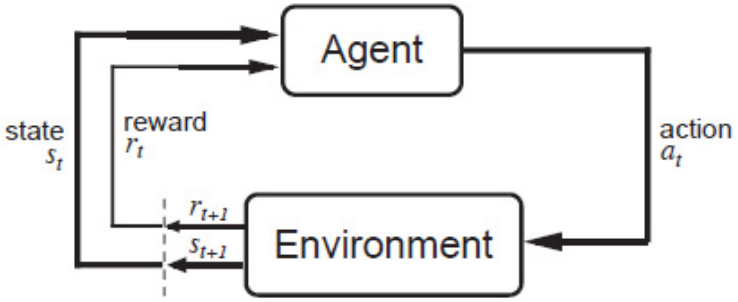
INF 552, Machine Learning for Data Science

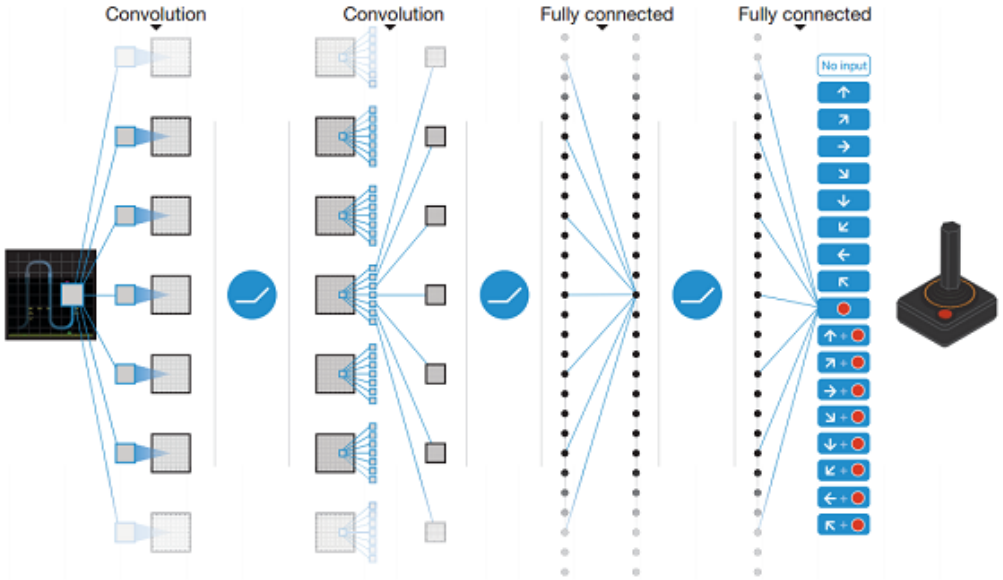
University of Southern California

M. R. Rajati, PhD

Lesson 13
Reinforcement Learning



AlphaGo



Overview

• Supervised Learning: Immediate feedback (labels provided for every input).

• Unsupervised Learning: No feedback (no labels provided).

• Reinforcement Learning: Delayed scalar feedback (a number called reward).

Overview

• RL deals with agents that must sense & act upon their environment.

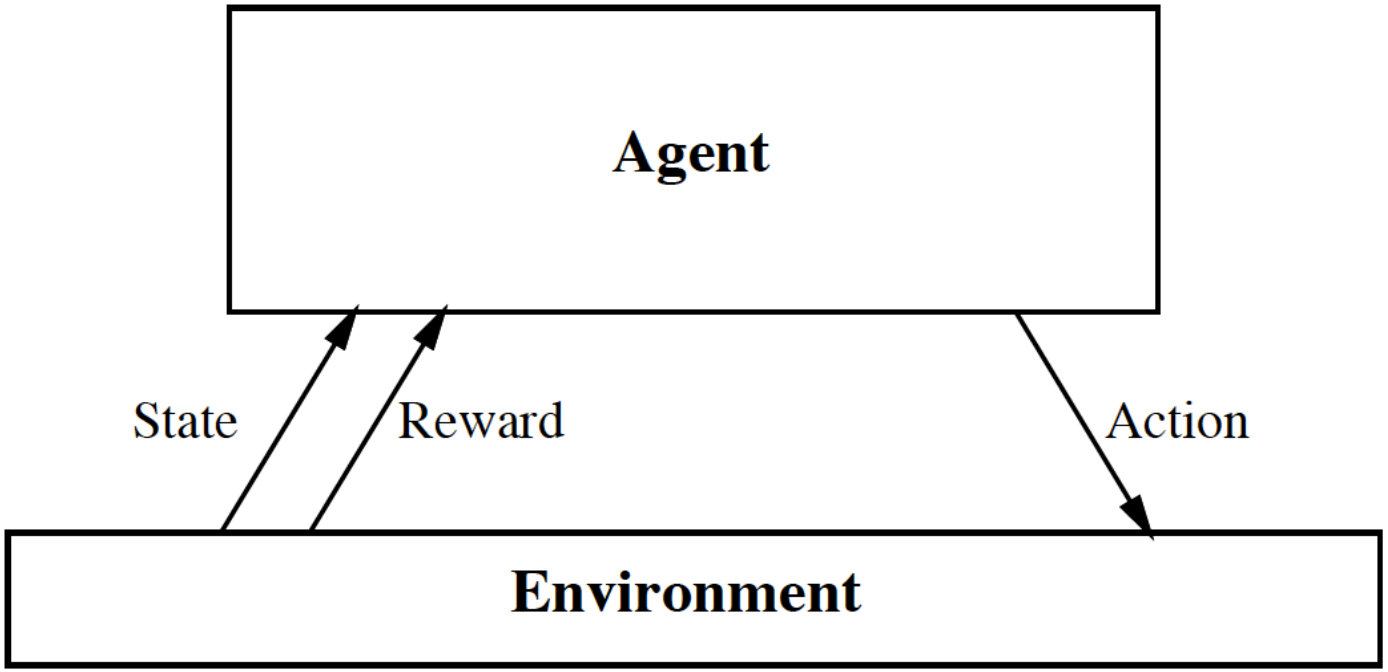
•This combines classical agent-based AI and machine learning techniques.

It is a very comprehensive problem setting.

Overview

- Examples:
  - A robot cleaning my room and recharging its battery
  - Robot-soccer
  - How to invest in shares
  - Modeling the economy through rational agents
  - Learning how to fly a helicopter
  - Scheduling planes to their destinations
  - and so on

The Big Picture



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

Your action influences the state of the world which determines its reward

Complications

- The outcome of your actions may be uncertain

- You may not be able to perfectly sense the state of the world

- The reward may be stochastic.

- Reward is delayed (i.e. finding food in a maze)

Complications

- You may have no clue (model) of how rewards are being paid off.

- The world may change while you try to learn it

- How much time do you need to explore uncharted territory before you exploit what you have learned?

The Task

• To learn an optimal *policy* that maps states of the world to actions of the agent.

I.e., if this patch of room is dirty, I clean it. If my battery is empty, I recharge it.

$$\pi : S \rightarrow A$$

• What is it that the agent tries to optimize?
Answer: the total future discounted reward:

The Task

- What is it that the agent tries to optimize?
  Answer: the total future discounted reward:

$$V^{\pi}(s_t) = r_t + \gamma r_{t+1} + \gamma^2 r_{t+2} + \ldots$$

$$= \sum_{i=0}^{\infty} \gamma^i r_{t+i} \qquad 0 \le \gamma < 1$$

Note: immediate reward is worth more than future reward.

What would happen to mouse in a maze with gamma = 0 ?

Value Function

- Let's say we have access to the optimal value function that computes the total future discounted reward $V^*(s)$

- What would be the optimal policy $\pi^*(s)$?

- Answer: we choose the action that maximizes:

$$\pi^*(s) = \arg\max_a \left[r(s,a) + \gamma V^*(\delta(s,a)) \right]$$

Value Function

- We assume that we know what the reward will be if we perform action "*a*" in state "*s*":

$$r(s,a)$$

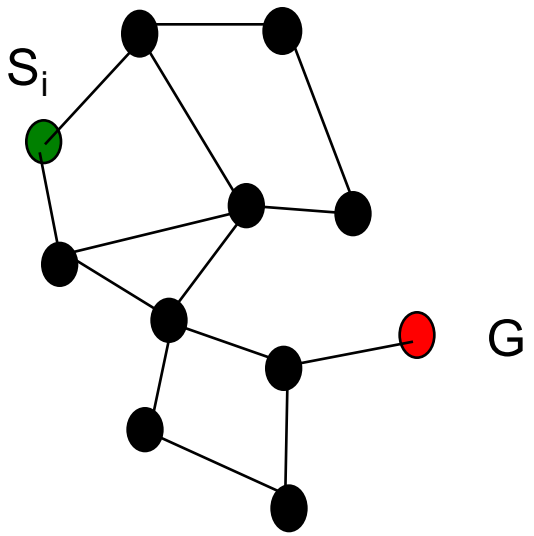
- We also assume we know what the next state of the world will be if we perform action "*a*" in state "*s*":

$$s_{t+1} = \delta(s_t, a)$$

Example I

• Consider some complicated graph, and we would like to find the shortest path from a node S_i to a goal node *G*.

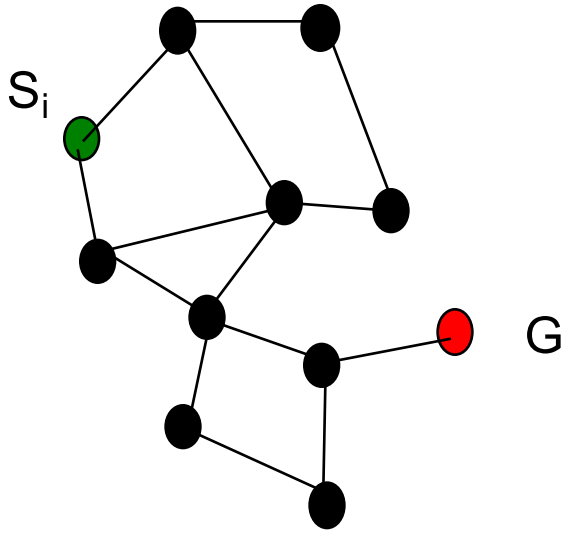
• Traversing an edge will cost you "length edge" dollars.



Example I

• The value function encodes the total remaining distance to the goal node from any node s, i.e.
V(*s*) = "*1 / distance*" to goal from *s*.

• If you know *V*(*s*), the problem is trivial. You simply choose the node that has highest *V*(*s*).

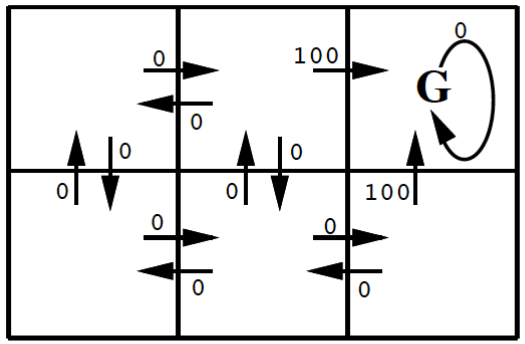


Example II

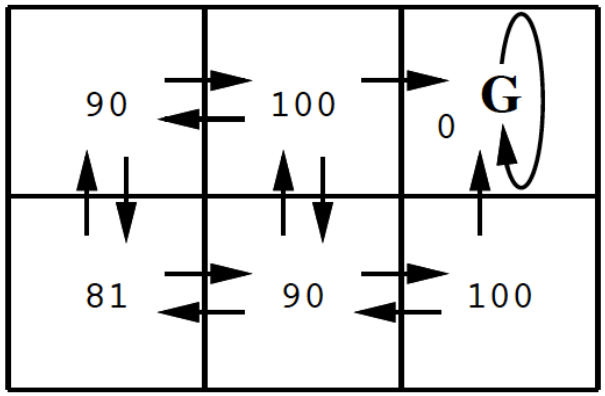
- A simple deterministic world.
- Each grid square represents a distinct state, each arrow a distinct action.
- The immediate reward function, r(s, a) gives reward 100 for actions entering the goal state G, and zero otherwise. Values of $V^*(s)$ follow from $r(s, a)$, and the discount factor $\gamma = 0.9$.
- An optimal policy is also shown.

Example II

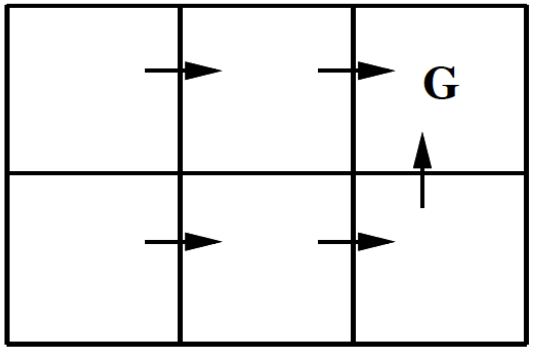
Find your way to the goal.



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



$V^*(s)$ values



One optimal policy

$$V^{\pi}(s_t) = r_t + \gamma r_{t+1} + \gamma^2 r_{t+2} + \ldots$$

$$= \sum_{i=0}^{\infty} \gamma^i r_{t+i} \qquad 0 \le \gamma < 1$$

Q-Function

• One approach to RL is then to try to estimate $V^*(s)$.

Bellman Equation:

$$V^*(s) \leftarrow \max_a \left[r(s,a) + \gamma V^*(\delta(s,a)) \right]$$

• However, this approach requires you to know $r(s,a)$ and $\delta(s,a)$.

• This is unrealistic in many real problems. What is the reward if a robot is exploring mars and decides to take a right turn?

Q-Function

- Fortunately we can circumvent this problem by exploring and experiencing how the world reacts to our actions. We need to *learn* r & δ.

Q-Function

- We want a function that directly learns good state-action pairs, i.e. what action should I take in this state. We call this $Q(s,a)$.

Q-Function

- Let us define the evaluation function $Q(s, a)$ so that its value is the maximum discounted cumulative reward that can be achieved starting from state s and applying action a as the first action.
- In other words, the value of Q is the reward received immediately upon executing action a from state s, plus the value (discounted by γ) of following the optimal policy thereafter.

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

Q-Function

• Why is this rewrite important? Because it shows that if the agent learns the Q function instead of the V^* function, it will be able to select optimal actions even when it has no knowledge of the functions r and δ.

Q-Function

• It need only consider each available action a in its current state s and choose the action that maximizes $Q(s, a)$.

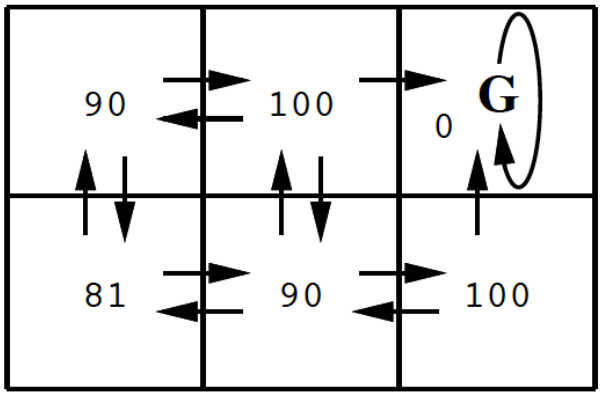
$$\pi^*(s) = \arg\max_a Q(s,a)$$

$$V^*(s) = \max_a Q(s,a)$$

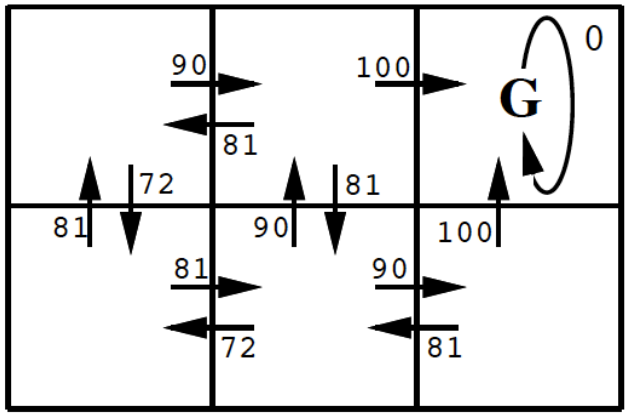
Example

- To illustrate, the figure in the next slide shows the Q values for every state and action in the simple grid world.
- The *Q* value for each state-action transition equals the *r* value for this transition plus the V^* value for the resulting state discounted by γ.
- The optimal policy shown in the figure corresponds to selecting actions with maximal *Q* values.

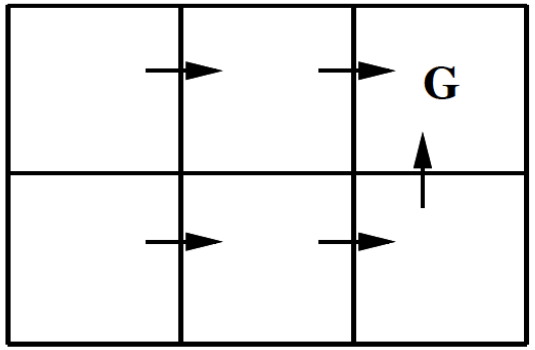
Example II



$V^*(s)$ values



$Q(s,a)$ values



One optimal policy

Check that

$$\pi^*(s) = \arg\max_a Q(s,a)$$

$$V^*(s) = \max_a Q(s,a)$$

Q-Learning

- Learning the Q function corresponds to learning the optimal policy. How can Q be learned?
- The key problem is finding a reliable way to estimate training values for Q, given only a sequence of immediate rewards r spread out over time.

Q-Learning

- This can be accomplished through iterative approximation.
- To see how, notice the close relationship between Q and V^*,

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

Q-Learning

which allows rewriting the Q function as:

$$Q(s,a) \equiv r(s,a) + \gamma V^*(\delta(s,a))$$
$$= r(s,a) + \gamma \max_{a'} Q(\delta(s,a),a')$$

This still depends on $r(s,a)$ and $\delta(s,a)$; however,

Q-Learning

$$Q(s,a) \equiv r(s,a) + \gamma V^*(\delta(s,a))$$
$$= r(s,a) + \gamma \max_{a'} Q(\delta(s,a),a')$$

this recursive definition of Q provides the basis for algorithms that iteratively approximate Q.

$\hat{Q}$ refers to the learner's estimate, or hypothesis, of the actual Q function.

Q-Learning

$$Q(s,a) \equiv r(s,a) + \gamma V^*(\delta(s,a))$$
$$= r(s,a) + \gamma \max_{a'} Q(\delta(s,a),a')$$

- Imagine the robot is exploring its environment, trying new actions as it goes.
- At every step it receives some reward "*r*", and it observes the environment change into a new state *s'* for action *a*.
- How can we use these observations, (*s*,*a*,*s'*,*r*) to learn a model?

Q-Learning

- The learner represents its hypothesis $\hat{Q}$ by a large table with a separate entry for each state-action pair.
- The table entry for the pair (s, a) stores the value for $\hat{Q}(s,a)$, learner's current hypothesis about the actual but unknown value $Q(s,a)$.
- The table can be initially filled with random values (though it is easier to understand the algorithm if one assumes initial values of zero).

Q-Learning

- The agent repeatedly observes its current state s, chooses some action a, executes this action, then observes the resulting reward $r = r(s, a)$ and the new state $s' = \delta(s, a)$.

Q-Learning

• It then updates the table entry for $\hat{Q}(s,a)$ following each such transition, according to the rule:

$$\hat{Q}(s,a) \leftarrow r + \gamma \max_{a'} \hat{Q}(s',a') \qquad s' = s_{t+1}$$

Q-Learning

$$\hat{Q}(s,a) \leftarrow r + \gamma \max_{a'} \hat{Q}(s',a') \qquad s' = s_{t+1}$$

- This equation continually estimates Q at state s consistent with an estimate of Q at state s', one step in the future: temporal difference (TD) learning.

- Note that s' is closer to goal, and hence more "reliable", but still an estimate itself.

Q-Learning Summary

Q learning algorithm

For each s, a initialize the table entry $\hat{Q}(s, a)$ to zero.
Observe the 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:

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

- $s \leftarrow s'$

Q-Learning

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

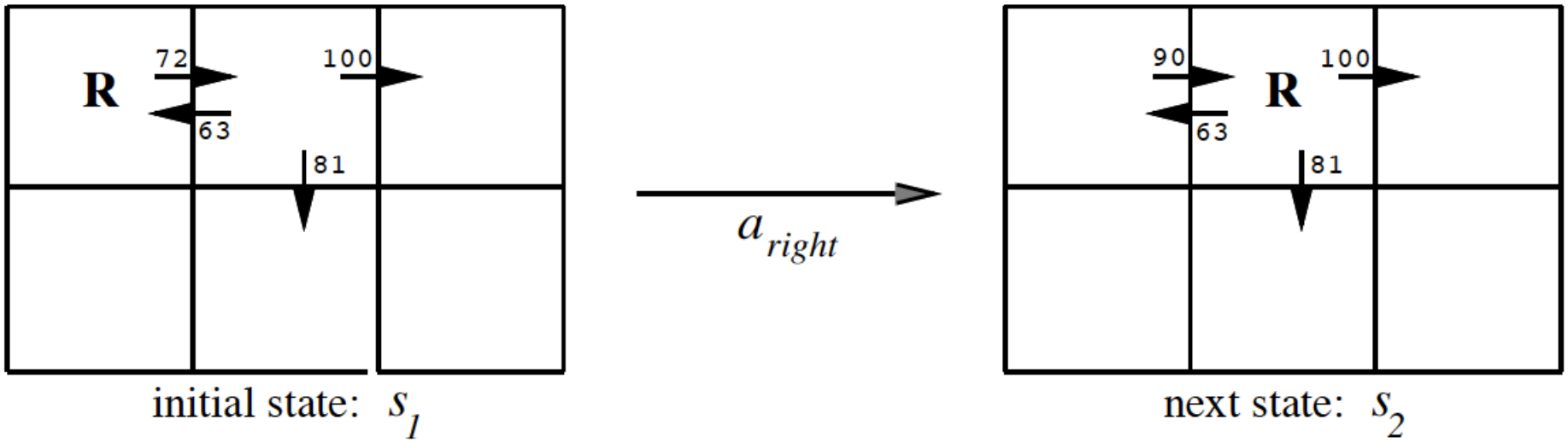
$s' = s_{t+1}$

• We do an update after each state-action pair. I.e., we are learning online!

• We are learning useful things about explored state-action pairs. These are typically most useful because they are likely to be encountered again.

• Under suitable conditions, these updates can actually be proved to converge to the real answer.

Example: Q-Learning



$$\hat{Q}(s_1, a_{right}) \leftarrow r + \gamma \max_{a'} \hat{Q}(s_2, a')$$

$$\leftarrow 0 + 0.9 \max\{66, 81, 100\}$$

$$\leftarrow 90$$

Q-learning propagates Q-estimates 1-step backwards

Exploration / Exploitation

- The algorithm does not specify how actions are chosen by the agent.
- One strategy would be in state s to select the action a that maximizes $\hat{Q}(s,a)$, thereby *exploiting* its current approximation $\hat{Q}$.

Exploration / Exploitation

- Using this strategy the agent runs the risk that it will overcommit to actions that are found during early training to have high *Q* values, while failing to explore other actions that have even higher values.
- It is common in Q learning to use a probabilistic approach to selecting actions.

Exploration / Exploitation

- Actions with higher Q values are assigned higher probabilities, but every action is assigned a nonzero probability. One way to assign such probabilities is

$$P(a_i \mid s) = \frac{e^{\hat{Q}(s,a_i)/T}}{\sum_j e^{\hat{Q}(s,a_j)/T}}$$

- where $P(a_i \mid s)$ is the probability of selecting action a_i, given that the agent is in state s, and where $T > 0$ is a constant that determines how strongly the selection favors actions with high Q values.

Exploration / Exploitation

- Hence it is good to try new things so now and then, e.g. If *T* large lots of exploring, if *T* small, exploit current policy. One can decrease *T* over time to first explore, and then converge and exploit.
- For example *T*= *c/k*+*d* where *k* is iteration of the algorithm

$$P(a_i \mid s) = \frac{e^{\hat{Q}(s,a_i)/T}}{\sum_j e^{\hat{Q}(s,a_j)/T}}$$

- Decreasing *T* over time is sometimes called *simulated annealing*, which is inspired by annealing process in metals. *T* is sometimes called the *Temperature*.

Improvements

• One can trade-off memory and computation by cashing (s,s',r) for observed transitions. After a while, as $Q(s',a')$ has changed, you can "replay" the update:

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

• One can actively search for state-action pairs for which $Q(s,a)$ is expected to change a lot (prioritized sweeping).

Temporal Difference Learning

- Q learning algorithm learns by iteratively reducing the discrepancy between Q value estimates for adjacent state
- Q learning is a special case of *temporal difference* algorithms that learn by reducing discrepancies between estimates made by the agent at different times.

Temporal Difference Learning

- The raining rule we studied reduces the difference between the estimated Q values of a state and its immediate successor
- However, we could design an algorithm that reduces discrepancies between this state and more distant descendants or ancestors.

Temporal Difference Learning

- Recall that our Q learning training rule calculates a training value for $\hat{Q}(s_t, a_t)$ in terms of the values for $\hat{Q}(s_{t+1}, a_{t+1})$ where s_{t+1} is the result of applying action a_t to the state s_t.
- Let $Q^{(1)}(s_t, a_t)$ denote the training value calculated by this one-step lookahead:

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

Temporal Difference Learning

- One alternative way to compute a training value for $Q(s_t, a_t)$ is to base it on the observed rewards for 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, in general, for *n* steps

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

Temporal Difference Learning

- A general method for blending these alternative training estimates, called TD(λ).
- The idea is to use a constant $0 \leq \lambda \leq 1$ to combine the estimates obtained from various lookahead distances in the following fashion

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

Temporal Difference Learning

- An equivalent recursive definition for Q^λ is

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

Temporal Difference Learning

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

$$+\lambda\ Q^{\lambda}(s_{t+1}, a_{t+1})]$$

- If $\lambda = 0$ we have our original training estimate $Q^{(1)}$, which considers only one-step discrepancies in the Q estimates.
- As λ is increased, the algorithm places increasing emphasis on discrepancies based on more distant lookaheads.

Temporal Difference Learning

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

$$+\lambda\ Q^{\lambda}(s_{t+1}, a_{t+1})]$$

- At the extreme value $\lambda = 1$, only the observed r_{t+i} values are considered, with no contribution from the current Q estimate.
- The motivation for the TD(λ) method is that in some settings training will be more efficient if more distant lookaheads are considered.

Extensions

- To deal with stochastic environments, we need to maximize *expected* future discounted reward:

$$Q(s,a) = E[r(s,a)] + \gamma \sum_{s'} P(s' \mid s,a) \max_{a'} Q(s',a')$$

- Often the state space is too large to deal with all states and adopt a table-lookup approach. In this case we need to learn a function:

$$Q(s,a) \approx f_\theta(s,a)$$

Extensions

$$Q(s,a) \approx f_\theta(s,a)$$

• Neural network with back-propagation have been quite successful.

• For instance, TD-Gammon is a back-gammon program that plays at expert level.
•state-space very large, trained by playing against itself, uses NN to approximate value function, uses *TD*(*λ*) for learning.

More on Function Approximation

- For instance: linear function:

$$Q(s,a) \approx f_\theta(s,a) = \sum_{k=1}^{K} \theta_k^a \Phi_k(s)$$

The features Φ are fixed measurements of the state (e.g. # stones on the board).

We only learn the parameters theta.

- Update rule: (start in state s, take action a, observe reward *r* and end up in state *s'*)

$$\theta_k^a \leftarrow \theta_k^a + \alpha \left(r + \gamma \max_{a'} \hat{Q}(s',a') - \hat{Q}(s,a) \right) \Phi_k(s)$$

change in Q

Conclusion

• Reinforcement learning addresses a very broad and relevant question:

How can we learn to survive in our environment?

• We have looked at Q-learning, which simply learns from experience.

No model of the world is needed.

Conclusion

- We made simplifying assumptions: e.g. state of the world only depends on last state and action. This is the *Markov* assumption. The model is called a *Markov Decision Process (MDP).*

Conclusion

- We assumed deterministic dynamics, reward function, but the world really is stochastic.

- There are many extensions to speed up learning.

- There have been many successful real world applications.