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).*

higher order Markov Decision Process
many previous actions

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.