CSE 6740 Lecture 21

How Do I Learn Actions From Data? (Reinforcement Learning)

Alexander Gray
agray@cc.gatech.edu

Georgia Institute of Technology

Today

1. Reinforcement Learning
2. RL Methods: Known Environment
3. RL Methods: Unknown Environment

Reinforcement Learning

Learning optimal *actions* from data.

Reinforcement Learning

Reinforcement learning (RL) is about learning optimal actions. The framework of RL is that you are an agent in an environment having discrete states s, and in each state you can take some actions a, and when taking certain actions in certain states you get a scalar reward r. For example:

Environment: You're in state 65. You have four possible actions.

You: I'll take action 2.

Environment: You received a reward of 7 units. You are now in state 15. You have two possible actions.

You: I'll take action 1.

Environment: You received a reward of -4 units...

Optimal Policy

In general our model is that the rewards and next-state transitions are non-deterministic, *i.e.* occur with some probability.

Your job is to find a *policy* π, saying which action should be taken in each state, that maximizes some long-run measure of reward. One example is the *finite-horizon reward*:

$$\mathbb{E}\left(\sum_{t=0}^{h} r_t\right) \tag{1}$$

which says you should optimize your expected reward for the next h steps, and need not worry about what will happen after that.

Optimal Policy

Since h is arbitrary, an infinite horizon reward is generally preferred, like the *average reward*:

$$\lim_{h \to \infty} \mathbb{E}\left(\frac{1}{h}\sum_{t=0}^{h} r_t\right). \tag{2}$$

One possible problem with this is that we might prefer a policy which gains a large amount of reward earlier. The geometrically *discounted reward* does that:

$$\mathbb{E}\left(\sum_{t=0}^{\infty} \gamma^t r_t\right), \tag{3}$$

where $0 \leq \gamma < 1$ is a mathematical device for bounding the infinite sum.

Optimality and Convergence

Which objective function should we use? The finite-horizon model is appropriate when the agent's lifetime is known, or there is a hard deadline. Average-reward and discounted-reward lead to different behaviors, and which one is better is a matter of debate. More is known about the discounted reward case because it is mathematically more convenient. For example, for this model, it can be shown that there exists an optimal deterministic (always choose the same action in a state), stationary (doesn't change over time) policy.

We can analyze whether a procedure converges to the optimal policy, and we can determine its rate of convergence to the optimal policy.

Exploration vs. Exploitation

In RL there is a tension between the advantage of exploring new parts of the environment and the advantage of exploiting parts of the environment we already know are good.

For example consider the simplest possible RL problem, the K-*armed bandit* problem. You are in a room with K slot machines, and are permitted h pulls. Each has an unknown probability p_k of paying off 1, otherwise pays 0. This is a single-state model where there are K actions corresponding to the arms we can pull.

It turns out there are good rigorous solutions to this problem (such as *Gittens indices*), but that they don't generalize to multiple states, which lead to delayed reward.

Delayed Reward

Suppose there are multiple states, as in our starting example. The agent's actions determine not only the immediate reward gained, but also the next state of the environment. So in order to decide on the optimal action to take in the current state, if it is maximizing a long-run measure of reward, it must account for the value of future states.

Just having multiple states creates the necessity of considering delayed reward.

Markov Decision Processes

A common model for the multiple-state case is the *Markov decision process* (MDP), which consists of discrete states $\{s\}$, discrete actions $\{a\}$, a real-valued reward function $R : \{s\} \times \{a\} \rightarrow \mathbb{R}$, and a state transition function $T : \{s\} \times \{a\} \rightarrow \beta(s)$, where $\beta(s)$ is a discrete probability distribution over all possible next states.

We write $T(s, a, s')$ for the probability of makig a transition from state s to state s' using action a. This is Markov because the probability of going to a state depends only on the last state.

RL Methods: Known Environment

First let's consider how we would find the optimal policy if we had a correct model of the environment (the rewards and state transitions).

Value Function

We'll assume the infinite-horizon discounted reward model.

We'll speak of the optimal *value* of a state - the expected discounted sum of reward that the agent will gain if it starts in that state and executes the optimal policy:

$$V^*(s) = \max_{\pi} \mathbb{E} \left(\sum_{t=0}^{\infty} \gamma^t r_t \right). \tag{4}$$

Value Function

This optimal value function is unique and can be defined as the solution to the simultaneous equations

$$V^*(s) = \max_a \left(R(s,a) + \gamma \sum_{s'} T(s,a,s') V^*(s') \right), \qquad (5)$$

which assert that the value of a state s is the expected instantaneous reward plus the expected discounted value of the next state, using the best available action. Given the optimal value function, we can specify the optimal policy as

$$\pi^*(s) = \arg\max_a \left(R(s,a) + \gamma \sum_{s'} T(s,a,s') V^*(s') \right). \qquad (6)$$

Value Iteration

So one way to find an optimal policy is to find the optimal value function. It can be found by a simple iterative algorithm, *value iteration*, which can be shown to converge to the correct V^* values:

initialize $V(s)$ arbitrarily
loop until policy is good enough
    loop over all s
        loop over all a
            $Q(s,a) = R(s,a) + \gamma \sum_{s'} T(s,a,s') V(s')$
        $V(s) = \max_a Q(s,a)$

We can determine when to stop using this fact: If $\max_s |V_i(s) - V_{i-1}(s)| < \epsilon$, $\max_s |V_i(s) - V^*(s)| < 2\epsilon\gamma\ (1-\gamma)$.

Value Iteration

The cost of each iteration scales with S^2A, where S is the number of states and A the number of actions. The number of iterations needed to reach the optimal value function is polynomial in S, the magnitude of the largest reward, and $1/(1-\gamma)$.

These iterative updates are called *backups*. The assignments to V need not be done in this order, as long as the value of each state gets updated infinitely often in an infinite run, for convergence to hold.

Value Iteration

Shown are *full backups*, since they make use of information from all possible successor states. However it can be shown that to obtain convergence it is sufficient to sample from the successor states, using *sample backups* of the form

$$Q(s,a) = Q(s,a) + \alpha\left(r + \gamma \max_{a'} Q(s',a') - Q(s,a)\right) \quad (7)$$

as long as each pairing of a and s is updated infinitely often in an infinite run, s' is sampled from the distribution $T(s,a,s')$, r is sampled with mean $R(s,a)$ and bounded variance, and the "learning rate" α is decreased slowly.

Policy Iteration

We can also manipulate the policy directly, without first finding the optimal value function, using *policy iteration*:

choose an arbitrary policy π'
loop
  $\pi = \pi'$
    compute the value function of policy π:
      solve the linear equations
      $V_\pi(s) = R(s, \pi(s)) + \gamma \sum_{s'} T(s, a, s') V_\pi(s')$
    improve the policy at each state:
      $\pi'(s) = \arg\max_a \left(R(s, a) + \gamma \sum_{s'} T(s, a, s') V_\pi(s')\right)$
until $\pi = \pi'$

Policy Iteration

The value function of a policy is the expected discounted reward that will be gained, at each state, by executing that policy. It can be determined by solving a set of linear equations.

Once we know the value of each state under the current policy, we consider whether the value could be improved by changing the first action taken. If it can, we change the policy to take the new action whenever it is in that situation.

This step is guaranteed to strictly improve the performance of that policy. When no improvements are possible, then the policy is guaranteed to be optimal.

Policy Iteration

Since there are A^S distinct policies, this algorithm terminates in at most an exponential number of iterations. However, it is known that it terminates in some (unknown) time polynomial in the state space.

Value iteration or policy iteration? Each iteration of policy iteration costs $O(AS^2 + S^3)$, but there are fewer iterations. Arguments have been put forward for both.

RL Methods: Unknown Environment

Now suppose we don't have prior knowledge of the environment (the rewards and state transitions).

Adaptive Heuristic Critic Method

There are two approaches we can take. We can learn a controller without learning a model of the environment, or we can learn a model and derive a controller from it. We'll start with model-free, or direct methods.

The idea of the *adaptive heuristic critic* (AHC) method is to modify policy iteration for the case where we don't know the model. We'll start with some policy, then learn the value function for that policy. Then we'll fix the value function and learn a new policy that maximizes the new value function, and so on.

Temporal Difference Learning

We'll replace the value function computation in policy iteration, which consisted of solving a set of linear equations, with an algorithm called *TD(0)*, a form of *temporal difference learning* which consists of the update rule

$$V(s) = V(s) + \alpha \left(r + \gamma V(s') - V(s) \right). \tag{8}$$

Whenever a state s is visited, its estimated value is updated to be closer to $r + \gamma V(s')$; r is the instantaneous reward received and $V(s')$ is the estimated value of the actually occurring next state. This is analogous to the sample-backup rule from value iteration.

Provided α is slowly decreasing, for a fixed policy, TD(0) converges to the optimal value function.

Temporal Difference Learning

TD(0) can be generalized to TD(λ). TD(0) looks just one step ahead when adjusting value estimates. The TD(λ) udpate rule is similar:

$$V(u) = V(u) + \alpha \left(r + \gamma V(s') - V(s) \right) e(u) \tag{9}$$

but it is applied to *every state*, rather than just to the immediately previous state s. The *eligibility trace* of a state

$$e(s) = \sum_{z=1}^{t} (\lambda\gamma)^{t-z} \delta_{s,s_z}, \quad \text{where } \delta_{s,s_z} = \begin{cases} 1 & \text{if } s = s_z \\ 0 & \text{otherwise} \end{cases} \tag{10}$$

represents the degree to which the state has been visited in the past. This is more expensive than TD(0) but often converges much more quickly.

Q-Learning

The work of the two components of AHC can be done in a unified manner using the *Q-learning* method. Let $Q^*(s, a)$ be the expected discounted reward of taking action a in state s, then continuing by choosing actions optimally. Note that $V^*(s)$ is the value of s assuming the best action is taken initially, so $V^*(s) = \max_a Q^*(s, a)$. We have

$$Q^*(s, a) = R(s, a) + \gamma \sum_{s'} T(s, a, s') \max_{a'} Q^*(s', a'). \qquad (11)$$

Since $V^*(s) = \max_a Q^*(s, a)$, we have $\pi^*(s) = \arg\max_a Q^*(s, a)$ as an optimal policy.

Q-Learning

The Q-learning update rule is

$$Q(s,a) = Q(s,a) + \alpha \left(r + \gamma \max_{a'} Q(s',a') - Q(s,a) \right) \qquad (12)$$

If each action is taken in each state an infinite number of times on an infinite run and α is decayed appropriately, this will converge to Q^* with probability 1.

This is the overall most popular method for the unknown-model case.

Certainty Equivalence

Simple idea: learn the T and R functions by exploring the environment and keeping statistics about the results of each action; next, compute an optimal policy using methods we've already seen. This method is called *certainty equivalence*.

However, this is unsatisfactory for several reasons: It makes an arbitrary division between the learning phase and the acting phase (though we can also compute a new optimal policy at every step), no exploration strategy is specified, and the environment may change.

Dyna

The Dyna method occupies a middle ground. At each step:

1. Update the model, incrementing statistics for the transitions and rewards, T and R.
2. Update the policy at state s using the latest T and R:

$$Q(s,a) = R(s,a) + \gamma \sum_{s'} T(s,a,s') \max_{a'} Q(s',a'). \tag{13}$$

3. Perform K additional updates: choose K state-action pairs at random and update them using the same rule.
4. Choose a next action based on the Q values, perhaps modified by an exploration strategy.

Many fewer iterations than Q-learning, faster overall.

Exploration

Some possible exploration strategies:

- Greedy: take the action with highest expected reward.
- Almost greedy: greedy with some chance of doing other things.
- Interval-based: also include confidence band, accounting for how many trials we've observed for a given state-action pair.

Prioritized Sweeping

Now instead of K random state-action pairs, update the K most interesting (= surprising) states from a priority queue:

1. Remember its current value: $V_{\text{old}} = V(s)$.
2. Update the state's value:

$$V(s) = \max_a \left(R(s,a) + \gamma \sum_{s'} T(s,a,s') V(s') \right) . \qquad (14)$$

3. Set the state's priority back to 0.
4. Compute the change $\Delta = |V_{\text{old}} - V(s)|$.
5. Each state s'' leading to s has its priority promoted to $\Delta T(s'', a, s)$, unless its priority was already bigger.

Many fewer iterations than Dyna, faster overall.

Function Approximation

Instead of a giant table of state-action pairs, use a function.

Function Approximation

Usually we can't actually store something of the order of all state-action pairs, or even all states. So we need a compact form, which we can obtain by function approximation.

Using a giant table also makes inefficient use of experience. In a large, smooth state space we'd expect similar states to have similar values and similar optimal actions.

There are various mappings we can hope to represent compactly, if $s \in \mathcal{S}, a \in \mathcal{A}$: $\mathcal{S} \to \mathcal{A}$ (policies), $\mathcal{S} \to \mathbb{R}$ (value functions), $\mathcal{S} \times \mathcal{A} \to \mathbb{R}$ (Q functions and rewards), $\mathcal{S} \times \mathcal{A} \times \mathcal{S} \to [0, 1]$ (transition probabilities), $\mathcal{S} \times \mathcal{A} \to \mathcal{S}$ (deterministic transitions).

Function Approximation

There are several approaches to generalization for the case of immediate reward, as in bandit problems, where there is no dependence from one state to another.

For delayed reward, a popular and representative approach is to approximate the value function and do value iteration. This is a regression problem, where we'd like to be efficient in an *online*, or *incremental* setting. It also turns out we need to be very accurate. Kernel regression is often used.

Function Approximation

It turns out that function approximation within RL is tricky. For example, value iteration no longer is guaranteed to converge if the value function is approximated with some error. Simple examples have been given showing value function errors growing arbitrarily large when generalization is used with value iteration.

For some kinds of function approximators (based on gradients), convergence to locally optimal solutions can be shown.

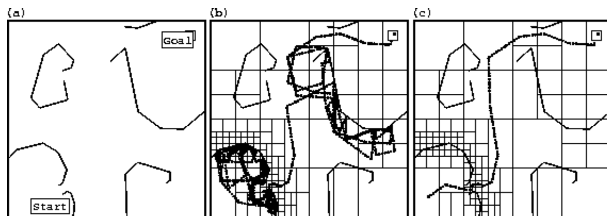
Adaptive Resolution Methods

It would be nice if we could partition the environment into regions of states that can be considered the same for the purposes of learning and generating actions.

One example is *variable-resolution dynamic programming*, in which the state space is partitioned into regions hierarchically. The coarse regions at the higher levels of the tree are refined only in parts of the state space which are predicted to be important. Importance is based on running virtual trajectories through state space. While very efficient, it requires a guess regarding valid trajectories through state space.

Adaptive Resolution Methods

For continuous state spaces with deterministic transitions, the *PartiGame* algorithm is very efficient, for example for a maze problem.

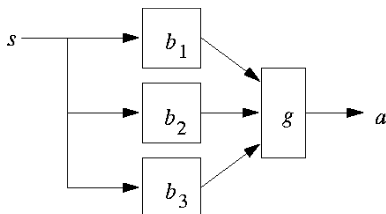


In each cell, the possible actions consist of going to neighboring cells. Refinement is done when it is detected that the cells are too coarse to allow movement between obstacles. State and action representation and exploration are all made more efficient.

Hierarchy

Another related idea is that of representing the RL problem as a hierarchy of learning problems. The solution to this is generally an approximation to that of the original problem, but more efficient.

A common idea is to think of a collection of *behaviors* which map states into low-level actions and a *gating function* which decides, based on the current state, which behavior's actions should be allowed to execute.



Hierarchy

In some architectures, the behaviors are fixed in advance and the gating function is learned through reinforcement. In some, the behaviors are learned through reinforcement, and the gating function fixed. Learning at both levels simultaneously is also done.

The generalization of this two-level structure results in a hierarchy.

These approaches are generally quite effective but fairly domain-specific.

Variant Problems

POMDP's, active learning.

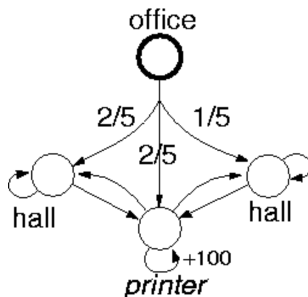
Partially Observable MDP

In many real-world environments, it will not be possible for the agent to have perfect and complete perception of the state of the environment. Unfortunately, complete observability is necessary for learning methods based on MDP's.

Now consider an MDP in which the agent makes observations of the state of the environment, but these observations may be noisy and provide incomplete information. This is called a *partially observable Markov decision process* (POMDP).

Partially Observable MDP

Here's an example, where we are trying to go from an office to the printer. It shows the difficulty of just taking our apparent observations to be the true state – two different things can appear the same.

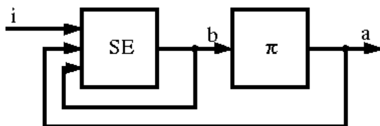


Although the underlying dynamics are Markovian, optimal decision-making requires tracking or modeling the entire history, making this a non-Markovian problem.

Partially Observable MDP

We can do this by maintaining a probability distribution over all states. One approach is to model the state and policy using using a finite-history window. This is equivalent to a higher-order Markov model.

Another approach is to model the environment as an HMM, including the hidden state, then derive a policy from that model.



The policy derivation can be formulated as an MDP, but this doesn't account for future uncertainty.

Partially Observable MDP

In the case where we have an accurate simulator of the underlying dynamics, we can find good policies, as in the *PEGASUS* method. In this case we can evaluate policies well by drawing sample trajectories and using a well-known Monte Carlo variance reduction idea, which we'll discuss later.

An example for which this was demonstrated is helicopter control, in which a simulator was available.

Active Learning

Instead of taking the data as given, the problem of *active learning* asks “If I could have another data point of my choice, which would it be?”. This problem is often motivated by the cost of obtaining data/labels or of learning.

Active learning can sometimes be thought of as a special case of RL, having one state. At its heart is an exploration-exploitation tradeoff.

A *myopic* active learner does not consider the future effect of choosing a certain data point, and only greedily chooses the next best point.

Active Learning

Virtually all existing approaches actually choose from a finite set of existing points. A harder problem is to propose a novel point to measure.

Depending on the type of estimation problem, many heuristics exist. For classification, the point having the highest uncertainty under the current model is often chosen.

For regression or density estimation, the point at which the variance of the current model is highest is often chosen. Under maximum likelihood this can often be interpreted as equivalent to maximizing an information gain.

Main Things You Should Know

- What reinforcement learning is
- What a Markov decision process is
- What value functions and policies are
- Known-environment vs. unknown-environment
- What Q-learning is
- What active learning is