Lecture 17: More on Markov Decision Processes. Reinforcement learning

- Learning a model: maximum likelihood
- Learning a value function directly
  - Monte Carlo
  - Temporal-difference (TD) learning

Recall: MDPs, Policies, Value functions

- An MDP consists of states S, actions A, rewards $r_a(s)$ and transition probabilities $T_a(s, s')$
- A policy π describes how actions are picked at each state:

$$\pi(s, a) = P(a_t = a | s_t = s)$$

- The value function of a policy, V^π, is defined as:

$$V^\pi(s) = E_\pi[r_{t+1} + \gamma r_{t+2} + \gamma^2 r_{t+3} + \dots]$$

- We can find V^π by solving a linear system of equations
- Policy iteration gives a greedy local search procedure based on the value of policies

Optimal Policies and Optimal Value Functions

- Our goal is to find a policy that has maximum expected utility, i.e. maximum value
- Does policy iteration fulfill this goal?
- The *optimal value function* V^* is defined as the best value that can be achieved at any state:

$$V^*(s) = \max_\pi V^\pi(s)$$

- *In a finite MDP, there exists a unique optimal value function* (shown by Bellman, 1957)
- Any policy that achieves the optimal value function is called *optimal policy*
- There has to be at least one deterministic optimal policy
- Both value iteration and policy iteration can be used to obtain an optimal value function.

Main idea

- Turn recursive Bellman equations into update rules
- Eg value iteration
  1. Start with an arbitrary initial approximation V_0
  2. On each iteration, update the value function estimate:

$$V_{k+1}(s) \leftarrow \max_a \left(r_a(s) + \gamma \sum_{s'} T_a(s, s') V_k(s') \right), \forall s$$

  3. Stop when the maximum value change between iterations is below a threshold
- The algorithm converges (in the limit) to the true V^*
- Similar update for policy evaluation.

A More Efficient Algorithm

- Instead of updating all states on every iteration, focus on *important states*
- Here, we can define important as *visited often*

  E.g., board positions that occur on every game, rather than just once in 100 games
- *Asynchronous dynamic programming*:
  - Generate trajectories through the MDP
  - Update states whenever they appear on such a trajectory
- This focuses the updates on states that are actually possible.

How Is Learning Tied with Dynamic Programming?

- Observe transitions in the environment, learn an *approximate model* $\hat{r}_a(s), \hat{T}_a(s, s')$
  - Use maximum likelihood to compute probabilities
  - Use supervised learning for the rewards
- Pretend the approximate model is correct and use it for any dynamic programming method
- This approach is called *model-based reinforcement learning*
- Many believers, especially in the robotics community

Simplest Case

- We have a coin X that can land in two positions (head or tail)
- Let $P(X = H) = \theta$ be the unknown probability of the coin landing head
- In this case, X is a *Bernoulli (binomial) random variable*
- Given a sequence of independent tosses $x_1, x_2, \dots x_m$ we want to estimate θ.

More Generally: Statistical Parameter Fitting

- Given instances $x_1, \ldots x_m$ that are *independently identically distributes (i.i.d.)*:
  - The set of possible values for each variable in each instance is known
  - Each instance is obtained independently of the other instances
  - Each instance is sampled from the same distribution
- Find a set of parameters θ such that the data can be summarized by a probability $P(x_j|\theta)$
- θ depends on the family of probability distributions we consider (e.g. binomial, multinomial, Gaussian etc.)

Coin Toss Example

- Suppose you see the sequence:

$$H, T, H, H, H, T, H, H, H, T$$

- Which of these values of $P(X = H) = \theta$ do you think is best?
  - 0.2
  - 0.5
  - 0.7
  - 0.9

How Good Is a Parameter Set?

- It depends on how likely it is to generate the observed data
- Let D be the data set (all the instances)
- The *likelihood* of parameter set θ given data set D is defined as:

$$L(\theta|D) = P(D|\theta)$$

- If the instances are i.i.d., we have:

$$L(\theta|D) = P(D|\theta) = P(x_1, x_2, \ldots x_m|\theta) = \prod_{j=1}^{m} P(x_j|\theta)$$

Example: Coin Tossing

- Suppose you see the following data:

$$D = H, T, H, T, T$$

What is the likelihood for a parameter θ?

$$L(\theta|D) = \theta(1-\theta)\theta(1-\theta)(1-\theta) = \theta^{N_H}(1-\theta)^{N_T}$$

Sufficient Statistics

- To compute the likelihood in the coin tossing example, we only need to know $N(H)$ and $N(T)$ (number of heads and tails)
- We say that $N(H)$ and $N(T)$ are *sufficient statistics* for this probabilistic model (binomial distribution)
- In general, a sufficient statistic of the data is a function of the data that summarizes enough information to compute the likelihood
- Formally, $s(D)$ is a sufficient statistic if, for any two data sets D and D',

$$s(D) = s(D') \Rightarrow L(\theta|D) = L(\theta|D')$$

Maximum Likelihood Estimation (MLE)

- *Choose parameters that maximize the likelihood function*
- We want to maximize:

$$L(\theta|D) = \prod_{j=1}^{m} P(x_j|\theta)$$

  This is a product, and products are hard to maximize!
- Standard trick is to maximize $\log L(\theta|D)$ instead

$$\log L(\theta|D) = \sum_{j=1}^{m} \log P(x_j|\theta)$$

- To maximize, we take the derivatives of this function with respect to θ and set them to 0

MLE Applied to the Binomial Data

- The likelihood is:

$$L(\theta|D) = \theta^{N(H)}(1-\theta)^{N(T)}$$

- The log likelihood is:

$$\log L(\theta|D) = N(H)\log\theta + N(T)\log(1-\theta)$$

- Take the derivative of the log likelihood and set it to 0:

$$\frac{\partial}{\partial\theta}\log L(\theta|D) = \frac{N(H)}{\theta} + \frac{N(T)}{1-\theta}(-1) = 0$$

- Solving this gives

$$\theta = \frac{N(H)}{N(H)+N(T)}$$

Observations

- Depending on our choice of probability distribution, when we take the gradient of the likelihood we may not be able to find θ analytically
- An alternative is to do **gradient descent** instead:

1. Start with some guess $\hat{\theta}$
2. Update $\hat{\theta}$:
$$\hat{\theta} \leftarrow \hat{\theta} + \alpha \frac{\partial}{\partial \theta} \log L(\theta|D)$$
where $\alpha \in (0,1)$ is a **learning rate**
3. Go back to 2 (for some number of iterations, or until θ stops changing significantly

- Sometimes we can also determine a **confidence interval** around the value of θ

MLE for multinomial distribution

- Suppose that instead of tossing a coin, we roll a K-faced die
- The set of parameters in this case is $p(k) = \theta_k, k = 1, \ldots K$
- We have the additional constraint that $\sum_{k=1}^{K} \theta_k = 1$
- What is the log likelihood in this case?

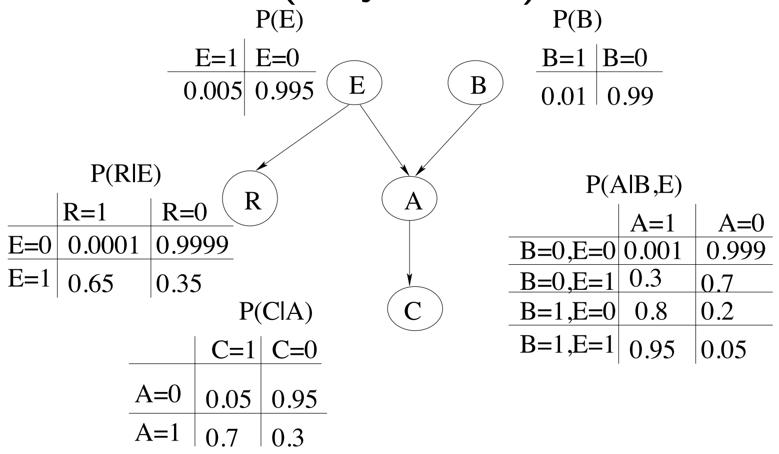
$$\log L(\theta|D) = \sum_k N_k \log \theta_k$$

  where N_k is the number of times value k appears in the data
- We want to maximize the likelihood, but now this is a *constrained* optimization problem
- (Without the details of the proof) the best parameters are given by the "empirical frequencies":

$$\hat{\theta}_k = \frac{N_k}{\sum_k N_k}$$

MLE for Bayes Nets

- Recall: For more complicated distributions, involving multiple variables, we can use a graph structure (Bayes net)



P(E)

E=1	E=0
0.005	0.995

P(B)

B=1	B=0
0.01	0.99

P(R|E)

	R=1	R=0
E=0	0.0001	0.9999
E=1	0.65	0.35

P(A|B,E)

	A=1	A=0
B=0,E=0	0.001	0.999
B=0,E=1	0.3	0.7
B=1,E=0	0.8	0.2
B=1,E=1	0.95	0.05

P(C|A)

	C=1	C=0
A=0	0.05	0.95
A=1	0.7	0.3

- Each node has a conditional probability distribution of the variable at the node given its parents (eg multinomial)
- The joint probability distribution is obtained as a product of the probability distributions at the nodes.

MLE for Bayes Nets

- Instances are of the form $\langle r_j, e_j, b_j, a_j, c_j \rangle, j = 1, \ldots m$

$$\begin{aligned}
L(\theta|D) &= \prod_{j=1}^{m} p(r_j, e_j, b_j, c_j, a_j|\theta) \text{ (from i.i.d)} \\
&= \prod_{j=1}^{m} p(e_j)p(r_j|e_j)p(b_j)p(a_j|e_j, b_j)p(c_j|e_j) \text{ (factorization)} \\
&= (\prod_{j=1}^{m} p(e_j))(\prod_{j=1}^{m} p(r_j|e_j))(\prod_{j=1}^{m} p(b_j))(\prod_{j=1}^{m} p(a_j|e_j, b_j))(\prod_{j=1}^{m} p(c_j|e_j)) \\
&= \prod_{i=1}^{n} L(\theta_i|D)
\end{aligned}$$

where θ_i are the parameters associated with node i.

Consistency of MLE

- For any estimator, we would like the parameters to converge to the "best possible" values as the number of examples grows

  We need to define "best possible" for probability distributions
- Let p and q be two probability distributions over X. The **Kullback-Leibler divergence** between p and q is defined as:

$$KL(p, q) = \sum_x p(x) \log \frac{p(x)}{q(x)}$$

A very brief detour into information theory

- Suppose I want to send some data over a noisy channel
- I have 4 possible values that I could send (e.g. A,C,G,T) and I want to encode them into bits such as to have short messages.
- Suppose that all values are equally likely. What is the best encoding?

A very brief detour into information theory (2)

- Now suppose I know A occurs with probability 0.5, C and G with probability 0.25 and T with probability 0.125. What is the best encoding?
- What is the expected length of the message I have to send?

Optimal encoding

- Suppose that I am receiving messages from an alphabet of m letters, and letter j has probability p_j
- The optimal encoding (by Shannon's theorem) will give $-\log_2 p_j$ bits to letter j
- So the expected message length if I used the optimal encoding will be equal to the **entropy** of p:

$$-\sum_j p_j \log_2 p_j$$

Interpretation of KL divergence

- Suppose now that letters would be coming from p but I don't know this. Instead, I believe letters are coming from q, and I use q to make the optimal encoding.
- The expected length of my messages will be $-\sum_j p_j \log_2 q_j$
- The amount of bits I waste with this encoding is:

$$-\sum_j p_j \log_2 q_j + \sum_j p_j \log_2 p_j = \sum_j p_j \log_2 \frac{p_j}{q_j} = KL(p,q)$$

Properties of MLE

- MLE is a **consistent estimator**, in the sense that (under a set of standard assumptions), w.p.1, we have:

  $$\lim_{|D|\to\infty} \theta = \theta^*,$$

  where θ^* is the "best" set of parameters: $\theta^* = \arg\min_\theta KL(p^*(X), p(X|\theta))$ (p^* is the true distribution)
- With a small amount of data, the variance may be high (what happens if we observe just one coin toss?)

Model-based reinforcement learning

- Very simple outline:
  - Learn a model of the reward (eg by averaging; more on this next time)
  - Learn a model of the probability distribution (eg by using MLE)
  - Do dynamic programming updates using the learned model as if it were true, to obtain a value function and a policy
- Works very well if you have to optimize many reward functions on the same environment (same transitions/dynamics)
- But you have to fit a probability distribution, which is quadratic in the number of states (so could be very big)
- Obtaining the value of a fixed policy is then cubic in the number of states, and then we have to tun multiple iterations...
- Can we get an algorithm *linear* in the number of states?

Monte Carlo Methods

- Suppose we have an episodic task: the agent interacts with the environment in trials or episodes, which terminate at some point
- The agent behaves according to some policy π for a while, generating several trajectories.
- How can we compute V^π?
- Compute $V^\pi(s)$ by *averaging the observed returns* after s on the trajectories in which s was visited.
- Like in bandits, we can do this incrementally: after received return R_t, we update

$$V(s_t) \leftarrow V(s_t) + \alpha(R_t - V(s_t))$$

  where $\alpha \in (0, 1)$ is a learning rate parameter

Temporal-Difference (TD) Prediction

- Monte Carlo uses as a target estimate for the value function the actual return, R_t:
$$V(s_t) \leftarrow V(s_t) + \alpha\left[R_t - V(s_t)\right]$$
- The simplest TD method, TD(0), uses instead an *estimate* of the return:
$$V(s_t) \leftarrow V(s_t) + \alpha\left[r_{t+1} + \gamma V(s_{t+1}) - V(s_t)\right]$$

  If $V(s_{t+1})$ were correct, this would be like a dynamic programming target!

TD Is Hybrid between Dynamic Programming and Monte Carlo!

- Like DP, it *bootstraps* (computes the value of a state based on estimates of the successors)
- Like MC, it estimates expected values by *sampling*

TD Learning Algorithm

```
1. Initialize the value function, $V(s) = 0, \forall s$
2. Repeat as many times as wanted:
  (a) Pick a start state $s$ for the current trial
  (b) Repeat for every time step $t$:
      i. Choose action $a$ based on policy $\pi$ and the current state $s$
      ii. Take action $a$, observed reward $r$ and new state $s'$
      iii. Compute the TD error: $\delta \leftarrow r + \gamma V(s') - V(s)$
      iv. Update the value function:
              $V(s) \leftarrow V(s) + \alpha_s \delta$
      v. $s \leftarrow s'$
      vi. If $s'$ is not a terminal state, go to 2b
```

Example

Suppose you start will all 0 guesses and observe the following episodes:

- B,1
- B,1
- B,1
- B,1
- B,0
- A,0; B (reward not seen yet)

What would you predict for $V(B)$? What would you predict for $V(A)$?

Example: TD vs Monte Carlo

- For B, it is clear that $V(B) = 4/5$.
- If you use Monte Carlo, at this point you can only predict your initial guess for A (which is 0)
- If you use TD, at this point you would predict $0 + 4/5$! And you would adjust the value of A towards this target.

Example (continued)

Suppose you start will all 0 guesses and observe the following episodes:

- B,1
- B,1
- B,1
- B,1
- B,0
- A,0; B 0

What would you predict for $V(B)$? What would you predict for $V(A)$?

Example: Value Prediction

- The estimate for B would be $4/6$
- The estimate for A, if we use Monte Carlo is 0; this minimizes the sum-squared error on the training data
- If you were to learn a model out of this data and do dynamic programming, you would estimate the A goes to B, so the value of A would be $0 + 4/6$
- TD is an *incremental* algorithm: it would adjust the value of A towards $4/5$, which is the current estimate for B (before the continuation from B is seen)
- This is closer to dynamic programming than Monte Carlo
- TD estimates take into account *time sequence*

Advantages

- No model of the environment is required! TD only needs experience with the environment.
- On-line, incremental learning:
  - Can learn before knowing the final outcome
  - Less memory and peak computation are required
- Both TD and MC converge (under mild assumptions), but TD usually learns faster.