Machine Learning

Reinforcement Learning

(slides from Bryan Pardo, Ian Horswill, thanks in part to Bill Smart at Washington University in St. Louis)
Learning Types

• Supervised learning:
  – (Input, output) pairs of the function to be learned can be perceived or are given.

  Back-propagation in Neural Nets

• Unsupervised Learning:
  – No information about desired outcomes given

  K-means clustering

• Reinforcement learning:
  – Reward or punishment for actions

  Q-Learning
Reinforcement Learning

• Task
  – Learn how to behave to achieve a goal
  – Learn through experience from trial and error

• Examples
  – Game playing: The agent knows when it wins, but doesn’t know the appropriate action in each state along the way

  – Control: a robot can measure whether it put a dish away without breaking it, but which action(s) cause success or failure?
1. Observe state, $s_t$
2. Decide on an action, $a_t$
3. Perform action
4. Observe new state, $s_{t+1}$
5. Observe reward, $r_{t+1}$
6. Learn from experience
7. Repeat

• Goal: Find a control policy that will maximize the observed rewards over the lifetime of the agent
An Example: Gridworld

• Canonical RL domain
  States are grid cells
  4 actions: N, S, E, W
  Reward for entering top right cell
  -0.01 for every other move
Mathematics of RL

• Before we talk about RL, we need to cover some background material
  – Simple decision theory
  – Markov Decision Processes
  – Value functions
  – Dynamic programming
Making Single Decisions

- Single decision to be made
  - Multiple discrete actions
  - Each action has a reward associated with it

- Goal is to maximize reward
  - Not hard: just pick the action with the largest reward

- State 0 has a value of 2
  - Sum of rewards from taking the best action from the state
Markov Decision Processes

• We can generalize the previous example to multiple sequential decisions
  – Each decision affects subsequent decisions

• This is formally modeled by a Markov Decision Process (MDP)
• Formally, a MDP is
  – A set of states, $S = \{s_1, s_2, \ldots, s_n\}$
  – A set of actions, $A = \{a_1, a_2, \ldots, a_m\}$
  – A reward function, $R: S \times A \times S \rightarrow \mathbb{R}$
  – A transition function, $P_{ij}^a = P(s_{t+1} = j | s_t = i, a_t = a)$
    • Sometimes $T: S \times A \rightarrow S$

• We want to learn a policy, $\pi: S \rightarrow A$
  – Maximize sum of rewards we see over our lifetime
Policies

- A policy $\pi(s)$ returns what action to take in state $s$.
- There are 3 policies for this MDP
  - Policy 1: $0 \rightarrow 1 \rightarrow 3 \rightarrow 5$
  - Policy 2: $0 \rightarrow 1 \rightarrow 4 \rightarrow 5$
  - Policy 3: $0 \rightarrow 2 \rightarrow 4 \rightarrow 5$
Comparing Policies

• Which policy is best?
• Order them by how much reward they see

Policy 1: \(0 \rightarrow 1 \rightarrow 3 \rightarrow 5 = 1 + 1 + 1 = 3\)
Policy 2: \(0 \rightarrow 1 \rightarrow 4 \rightarrow 5 = 1 + 1 + 10 = 12\)
Policy 3: \(0 \rightarrow 2 \rightarrow 4 \rightarrow 5 = 2 - 1000 + 10 = -988\)
Value Functions

• We can associate a value with each state
  – For a fixed policy
  – How good is it to run policy $\pi$ from that state $s$
  – This is the state value function, $V$

$V^1(s_0) = 3$
$V^2(s_0) = 12$
$V^3(s_0) = -988$

$V^1(s_1) = 2$
$V^2(s_1) = 11$
$V^1(s_3) = 1$

$V^3(s_2) = -990$
$V^2(s_4) = 10$
$V^3(s_4) = 10$
Problems with Our Function

- Consider this MDP
  - Number of steps is now unlimited because of loops
  - Value of states 1 and 2 is infinite for some policies

\[ V^1(s_0) = 1 + V^1(s_0) \]

- This is bad
  - All policies with a non-zero reward cycle have infinite value
Adding up the rewards

• We had said:
  – The reward for a policy (called the return) is just the sum of the rewards you get at every time step:
    \[ R = r_1 + r_2 + r_3 + \cdots \]
    – And then look for a policy that maximizes the expected value of this sum

• But we don’t do that
  – Infinities
Discount factor

• The pure sum is **infinite** and in any case **model errors** (e.g. the agent dying) usually mean our estimates of rewards get less accurate the farther we look in the future.

• So we weight future returns less by a factor $\gamma$ (the **discount rate**):
  \[ R = r_1 + \gamma r_2 + \gamma^2 r_3 + \cdots \]

• And then our goal is to find a policy that **maximizes expected time-discounted reward**.
What to do??

• A (now randomized) **policy**  
  \[ \pi: S \times \mathcal{A} \rightarrow [0,1] \]  
gives the probability of \( \pi \) running a given action in a given state  
  
  – A **deterministic policy** \( \pi: S \rightarrow \mathcal{A} \) is a policy that in state \( s \) runs action \( \pi(s) \) always  

• We want to **pick a policy** that will **maximize expected reward**  

• First: how do we even compute the expected reward for a **given** policy?
Value functions

- One you decide on a given policy, $\pi$, you can compute the expected return for the policy
- We express that in terms of the **state value function** for the policy
  - $V^\pi(s)$ is the **expected return** when starting from state $s$ and running the policy $\pi$
    \[
    V^\pi(s) = \sum_a \pi(s, a) \sum_{s'} \mathcal{P}^a_{ss'} [\mathcal{R}^a_{ss'} + \gamma V^\pi(s')] \]
  - This **averages** over
    - All possible actions $\pi$ could take from state $s$
    - All possible successor states $s'$ those actions could land us in
    - All possible rewards we could get from it
- And sums for each of them
  - The reward $\mathcal{R}^a_{ss'}$ you get with
  - The expected return $V^\pi(s')$ for running the policy from the resulting state $s'$, subject to the discount rate $\gamma$
Computing $V^\pi$ (aka policy evaluation)

- The naïve thing to do is just to evaluate the definition directly:
  \[ V^\pi(s) = \sum_a \pi(s, a) \sum_{s'} P_{ss'}^a \left[ R_{ss'}^a + \gamma V^\pi(s') \right] \]

- That is, we interpret it as a function definition:

```plaintext
V(\pi, s) {
    sum = 0;
    foreach a {
        foreach s' {
            r = R[a, s, s'] + \gamma * V(\pi, s');
            sum += \pi(s, a) * P[a, s, s'] * r;
        }
    }
    return sum;
}
```

- Of course, this code won’t work
- Why?
Computing $V^\pi$ (aka policy evaluation)

- It’s an infinite recursion

```plaintext
V(\pi, s) {
    sum = 0;
    foreach a {
        foreach s' {
            r = R[a,s,s'] + \gamma * V(\pi, s');
            sum += \pi(s, a) * P[a,s,s'] * r;
        }
    }
    return sum;
}
```
Computing $V^\pi$ (aka policy evaluation)

• But we can fix it by only recursing a certain number of times

• This raises the question of whether this will give us the right answer
  – We’ll get to that later

```c
V(\pi, s, k) {  
    if (k == 0) return 0;  
    sum = 0;  
    foreach a {  
        foreach s’ {  
            r = R[a,s,s’]+ \gamma*V(\pi, s’, k-1);  
            sum += \pi(s, a)*P[a,s,s’]*r;  
        }  
    }  
    return sum;  
}
```
Computing $V^\pi$ (aka policy evaluation)

- But even this still has a massive problem
- Can you see what it is?

```python
V(\pi, s, k) {
    if (k == 0) return 0;
    sum = 0;
    foreach a {
        foreach s’ {
            r = R[a,s,s’] + \gamma * V(\pi, s’, k-1);
            sum += \pi(s, a)*P[a,s,s’]*r;
        }
    }
    return sum;
}
```
Computing $V^\pi$ (aka policy evaluation)

It’s massively **inefficient**

- $V(\pi, s', k-1)$ gets recomputed once for each value of $a$
  - (each iteration of the outer loop)
- Worse, the recursive calls that those calls make get repeated too
- So if there are $n$ different actions, then $V(\pi, s', k-i)$ gets computed $n^i$ times
- How do we fix this?

```java
V(\pi, s, k) {
  if (k == 0) return 0;
  sum = 0;
  foreach a {
    foreach s' {
      r = R[a,s,s'] + \gamma * V(\pi, s', k-1);
      sum += \pi(s, a) * P[a,s,s'] * r;
    }
  }
  return sum;
}
```
Dynamic programming

- Compute each value of $V(\pi, s', k)$ **once only**
- Stash it in a **table**
- Use the value in the table for subsequent calls

- This is known as **top-down dynamic programming** or **memoization**
  - C.f. 214, 336, and some versions of 111

```c
V(\pi, s, k) {
  if (k == 0) return 0;
  if (table[s,k] filled in)
    return table[s,k]
  sum = 0;
  foreach a {
    foreach s’ {
      r = R[a,s,s’] + \gamma * V(\pi, s’, k-1);
      sum += \pi(s, a)*P[a,s,s’]*r;
    }
  }
  table[s,k] = sum;
  return sum;
}
```
Dynamic programming

- However, since we know we’ll end up computing all the entries in table[s,k] anyway, why bother with the annoying recursion?
  - Just compute all the entries for table[s,0]
  - Then compute all the entries for table[s,1]
  - Then compute all the entries for table[s,2]
  - Etc.

\[
V(\pi, s, k) \{
  \text{if } (k == 0) \text{ return } 0;
  \text{if } (\text{table}[s,k] \text{ filled in})
  \quad \text{return } \text{table}[s,k]
  \text{sum} = 0;
  \text{foreach } a \{
    \text{foreach } s' \{
      r = R[a,s,s'] + \gamma * V(\pi, s', k-1);
      \text{sum += } \pi(s, a) * P[a,s,s'] * r;
    \}
  \}
  \text{table}[s,k] = \text{sum};
  \text{return sum;}
\}
\]
Dynamic programming

- Here’s the code

- Just call this, and then the estimated values of V are all in \( \text{table}[s, k] \)

- This is known as **bottom-up dynamic programming**

```cpp
FillTable() {
    foreach s
        table[s,0]=0
    for i=1 to k
        foreach s {
            sum = 0;
            foreach a {
                foreach s' {
                    r = R[a,s,s']+ \gamma*V(\pi, s', k-1);
                    sum += \pi(s, a)*P[a,s,s']*r;
                }
                table[s,k] = sum;
            }
        }
}
```
Dynamic programming

• Dynamic programming was originally invented by Bellman for solving MDPs

• It was called dynamic programming because
  – Programming in those days meant optimization
  – He solved an optimization involving time
  – He thought the word dynamic made it sound more impressive (no, really!)

```java
public static void FillTable()
{
    foreach s
        table[s,0]=0
    for i=1 to k
        foreach s {
            sum = 0;
            foreach a {
                foreach s’ {
                    r = R[a,s,s’]+ γ*V(π, s’, k-1);  
                    sum += π(s, a)*P[a,s,s’]*r;
                }
                table[s,k] = sum;
            }
        }
}
```
Getting back to the equations…

- We’re trying to compute
  \[
  V^\pi(s) = \sum_a \pi(s, a) \sum_{s'} P_{ss'}^a [R_{ss'}^a + \gamma V^\pi(s')] \]

- And we basically said we could compute it by computing
  \[
  V_k^\pi(s) = \sum_a \pi(s, a) \sum_{s'} P_{ss'}^a [R_{ss'}^a + \gamma V_{k-1}^\pi(s')] \]

- For large values of \( k \)
- This was Bellman’s original formulation
Finding the optimal policy

• The **optimal state value function** would be the one that does whatever the best policies do in any given state:
  \[ V^*(s) = \max_\pi V^\pi(s) \]

• If we knew what \( V^* \) was, we could compute an **optimal action-state value function** for it:
  \[ Q^*(s, a) = Q^{\pi^*}(s, a) = \sum_{s', a'} P^a_{ss'} [R^a_{ss'} + \gamma V^*(s')] \]

• And back-solve the **optimal policy** from that:
  \[ \pi^*(s, a) = \begin{cases} 1, & a = \max_{a'} Q^*(s, a') \\ 0, & \text{otherwise} \end{cases} \]
Bellman’s optimality criteria

- Bellman showed that the **optimal value function** is one that does what the **optimal policy** does for any given state:

\[
V^*(s) = \max_a Q^\pi^*(s, a)
\]

\[
= \max_a \sum_{s'} P_{ss'}^a \left[ R_{ss'}^a + \gamma V^*(s') \right]
\]
Value iteration

- So we can approximate $V^*$ using the same dynamic programming trick used for policy evaluation:

$$V^*(s) = \lim_{k \to \infty} V_k(s)$$

$$V_k(s) = \max_a \sum_{s'} P_{ss'}^a [R_{ss'}^a + \gamma V_{k-1}(s')]$$
Value iteration

Initialize $V$ arbitrarily, e.g., $V(s) = 0$, for all $s \in S^+$

Repeat
\[ \Delta \leftarrow 0 \]
For each $s \in S$:
\[ v \leftarrow V(s) \]
\[ V(s) \leftarrow \max_a \sum_{s'} P_{ss'}^a [R_{ss'}^a + \gamma V(s')] \]
\[ \Delta \leftarrow \max(\Delta, |v - V(s)|) \]
until $\Delta < \theta$ (a small positive number)

Output a deterministic policy, $\pi$, such that
\[ \pi(s) = \arg \max_a \sum_{s'} P_{ss'}^a [R_{ss'}^a + \gamma V(s')] \]
Conclusion

• Value iteration gives us a **greedy** policy provided we have a **perfect model** of the world
  – In the form of $P^a_{ss'}$ and $R^a_{ss'}$

• Next time we’ll look at **learning policies from experience** without assuming a prior model
Recall

- Optimal “value function”:

\[
V^*(s) = \lim_{k \to \infty} V_k(s)
\]

\[
V_k(s) = \max_a \sum_{s'} P^a_{ss'} [R^a_{ss'} + \gamma V_{k-1}(s')]
\]
Learning from Experience

• We need
  – Model of the world \( P_{ss'} \)
  – Reward model \( R_{ss'} \)

• How do we get them?
  – One option, we write them down
    • Design reward function, physical model, etc.
  – What about uncertain environments? => LEARN
Gee, it’s easy

• Collect experience by moving through the world
  – $s_0, a_0, r_1, s_1, a_1, r_2, s_2, a_2, r_3, s_3, a_3, r_4, s_4, a_4, r_5, s_5, \ldots$

• Use these to estimate world, reward models

• Solve for the optimal value function

• Compute the optimal policy from it
Example

From Russell and Norvig
What’s wrong with that?

- Intractable for all but the simplest problems
- Spends a ton of time in low-value states
Let’s start with tractability

• Do we really have to learn $P^a_{ss'}$?

• Related question – you may be able to play Pac-Man. Does that mean you’ve computed the stochastic model underlying Pac-Man?
  – No.

• Idea: learn what to do next, *without* world model
TD(0)-Learning Algorithm

• Input – a **fixed** policy $\pi$ to evaluate
• Initialize $V^\pi(s)$ to 0
• For each ‘episode’ (episode = series of actions)
  – Repeat until out of actions:
    1. Observe state $s$
    2. Perform action according to the policy $\pi(s)$
    3. $V(s) \leftarrow (1-\alpha)V(s) + \alpha[r + \gamma V(s')]$
    4. $s \leftarrow s'$

$r$ = reward
$\alpha$ = learning rate
$\gamma$ = discount factor

Note: this formulation is from Sutton & Barto’s “Reinforcement Learning”
• TD(0)’s $V(s)$ estimate will converge to $V^\pi(s)$
  – After an infinite number of experiences
  – If we decay the learning rate s.t.:
    \[ \sum_{t=0}^{\infty} \alpha_t = \infty \quad \sum_{t=0}^{\infty} \alpha_t^2 < \infty \]
  – …so
    \[ \alpha_t = \frac{c}{c + t} \text{ will work} \]

• => We can get $V^\pi(s)$ more tractably… but $V^*(s)$?
  – And we’re still spending lots of time in low-val states
Exploration vs. Exploitation

• We want to pick good actions most of the time, but also do some exploration
• Exploring means we can learn better policies
• But, we want to balance known good actions with exploratory ones
• This is called the exploration/exploitation problem
Let’s Explore! And exploit

- **On-policy algorithms**
  - Final policy is influenced by the exploration policy
  - Generally, the exploration policy needs to be “close” to the final policy
  - Can get stuck in local maxima

- **Off-policy algorithms**
  - Final policy is independent of exploration policy
  - Can use arbitrary exploration policies
  - Will not get stuck in local maxima
We’ll learn Q

• Rather than $V^*(s)$, we’ll learn:
  – $Q(s, a) = \text{the expected utility of taking a particular action } a \text{ in state } s$
Picking Actions

\( \varepsilon \)-greedy
- Pick best (greedy) action with probability \( \varepsilon \)
- Otherwise, pick a random action

- Boltzmann (Soft-Max)
  - Pick an action based on its Q-value

\[
P(a | s) = \frac{\text{e}^{\left(\frac{Q(s, a)}{\tau}\right)}}{\sum_{a'} \text{e}^{\left(\frac{Q(s, a')}{\tau}\right)}}
\]

...where \( \tau \) is the “temperature”
Two methods

• SARSA (on-policy)
• Q-learning (off-policy)
SARSA

- SARSA iteratively approximates the state-action value function, Q
  - SARSA learns the policy and the value function simultaneously

- Keep an estimate of Q(s, a) in a table
  - Update these estimates based on experiences
  - Estimates depend on the exploration policy
  - SARSA is an on-policy method
  - Policy is derived from current value estimates
SARSA Algorithm

1. Initialize $Q(s, a)$ to small random values, $\forall s, a$
2. Observe state, $s$
3. $a \leftarrow \pi(s)$
   (policy derived from $Q$, e.g. $\epsilon$-greedy)
4. Observe next state, $s'$, and reward, $r$
5. $Q(s, a) \leftarrow (1-\alpha)Q(s, a) + \alpha(r + \gamma Q(s', \pi(s')))$
6. Go to 2

- $0 \leq \alpha \leq 1$ is the learning rate
  - We should decay this, just like TD
Q-Learning

- Q-learning iteratively approximates the state-action value function, Q
  - Like SARSA, we won’t estimate a world model
  - Learns the value function and policy simultaneously

- Keep an estimate of Q(s, a) in a table
  - Update these estimates as we gather more experience
  - Estimates do not depend on exploration policy
  - Q-learning is an off-policy method

[Watkins & Dayan, 92]
Q-Learning Algorithm

1. Initialize $Q(s, a)$ to small random values, $\forall s, a$ (what if you make them 0? What if they are big?)
2. Observe state, $s$
3. Pick action $a$ using policy derived from $Q$
4. Observe next state, $s'$, and reward, $r$
5. $Q(s, a) \leftarrow (1 - \alpha)Q(s, a) + \alpha(r + \gamma \max_{a'}Q(s', a'))$
6. $s \leftarrow s'$
7. Go to 2

$0 \leq \alpha \leq 1$ is the learning rate & we should decay $\alpha$, just like in TD
This formulation is from Sutton & Barto’s “Reinforcement Learning”
Q-learning vs. SARSA

• SARSA:
  – $Q(s, a) \leftarrow (1-\alpha)Q(s, a) + \alpha(r + \gamma Q(s', \pi(s')))$

• Q-learning:
  – $Q(s, a) \leftarrow (1 - \alpha)Q(s, a) + \alpha(r + \gamma \max_{a'}Q(s', a'))$

• In both algorithms, actions chosen according to the Q being learned (exploit while exploring)…
  – So why is Q-learning “off-policy”?
Reinforcement Learning for Robotics?

• Challenges
  – Actions have physical consequences
  – State-action space is continuous/high-dim
    • Sparse! And, how to get $\max_{a'}()$?
  – Bottom line: RL not feasible in robots w/out modifications

• Good news
  – Good framework to start with
  – Parallel to human/animal learning
    • (vs. input/output pairs in supervised learning)
  – Modifications have been developed to port RL to robots