How to plan when you know how your actions affect your environment?

MDP lectures

How to learn from data about how your environment works?

Machine Learning lectures

How to trade off collecting data vs. accomplishing goals?

Today (also VPI lecture)

Putting it all together:

RL lectures next week
Exploration vs. Exploitation Dilemma

- Online decision-making involves a fundamental choice:
  - **Exploitation**  Make the best decision given current information
  - **Exploration**  Gather more information

- The best long-term strategy may involve short-term sacrifices

- Gather enough information to make the best overall decisions
Examples

- Restaurant Selection
  - Exploitation: Go to your favourite restaurant
  - Exploration: Try a new restaurant

- Online Banner Advertisements
  - Exploitation: Show the most successful advert
  - Exploration: Show a different advert

- Oil Drilling
  - Exploitation: Drill at the best known location
  - Exploration: Drill at a new location

- Game Playing
  - Exploitation: Play the move you believe is best
  - Exploration: Play an experimental move
Principles

- **Naive Exploration**
  - Add noise to greedy policy (e.g. $\epsilon$-greedy)

- **Optimistic Initialisation**
  - Assume the best until proven otherwise

- **Optimism in the Face of Uncertainty**
  - Prefer actions with uncertain values

- **Probability Matching**
  - Select actions according to probability they are best
The Multi-Armed Bandit

- A multi-armed bandit is a tuple $\langle A, R \rangle$
- $A$ is a known set of $m$ actions (or “arms”)
- $R^a(r) = \mathbb{P}[r|a]$ is an unknown probability distribution over rewards
- At each step $t$ the agent selects an action $a_t \in A$
- The environment generates a reward $r_t \sim R^{a_t}$
- The goal is to maximise cumulative reward $\sum_{\tau=1}^{t} r_{\tau}$
Regret

- The action-value is the mean reward for action $a$,
  \[ Q(a) = \mathbb{E}[r|a] \]
- The optimal value $V^*$ is
  \[ V^* = Q(a^*) = \max_{a \in A} Q(a) \]
- The regret is the opportunity loss for one step
  \[ l_t = \mathbb{E}[V^* - Q(a_t)] \]
- The total regret is the total opportunity loss
  \[ L_t = \mathbb{E}\left[ \sum_{\tau=1}^{t} V^* - Q(a_{\tau}) \right] \]
- Maximise cumulative reward $\equiv$ minimise total regret
Counting Regret

- The *count* $N_t(a)$ is expected number of selections for action $a$
- The *gap* $\Delta_a$ is the difference in value between action $a$ and optimal action $a^*$, $\Delta_a = V^* - Q(a)$
- Regret is a function of gaps and the counts

$$L_t = \mathbb{E} \left[ \sum_{\tau=1}^{t} V^* - Q(a_\tau) \right]$$

$$= \sum_{a \in A} \mathbb{E} [N_t(a)] (V^* - Q(a))$$

$$= \sum_{a \in A} \mathbb{E} [N_t(a)] \Delta_a$$

- A good algorithm ensures small counts for large gaps
- Problem: gaps are not known!
Remember regret is \textit{expectation} of suboptimality

- If an algorithm \textit{forever} explores it will have linear total regret
- If an algorithm \textit{never} explores it will have linear total regret
- Is it possible to achieve sublinear total regret?
Greedy Algorithm

action_counts = [0 for a in range(n_actions)]
total_reward_per_action = [0 for a in range(n_actions)]

for t in range(inf):
    Qhat_a = [total_reward_per_action[a] / action_counts[a] for a in range(n_actions)]
    a_t = argmax(Qhat_a)
    r_t = take_action_and_get_random_reward(a_t)
    action_counts[a_t] += 1
    total_reward_per_action[a_t] += r_t
Greedy Algorithm

- We consider algorithms that estimate $\hat{Q}_t(a) \approx Q(a)$
- Estimate the value of each action by Monte-Carlo evaluation

$$\hat{Q}_t(a) = \frac{1}{N_t(a)} \sum_{t=1}^{T} r_t 1(a_t = a)$$

- The greedy algorithm selects action with highest value

$$a^*_t = \arg\max_{a \in A} \hat{Q}_t(a)$$

- Greedy can lock onto a suboptimal action forever
- $\Rightarrow$ Greedy has linear total regret
The *action-value* is the mean reward for action $a$,

$$Q(a) = \mathbb{E}[r|a]$$

The *optimal value* $V^*$ is

$$V^* = Q(a^*) = \max_{a \in A} Q(a)$$

The *regret* is the opportunity loss for one step

$$l_t = \mathbb{E}[V^* - Q(a_t)]$$

The *total regret* is the total opportunity loss

$$L_t = \mathbb{E}\left[ \sum_{\tau=1}^{t} V^* - Q(a_\tau) \right]$$

The *gap* $\Delta_a$ is the difference in value between action $a$ and optimal action $a^*$,

$$\Delta_a = V^* - Q(a)$$
The \( \epsilon \)-greedy algorithm continues to explore forever

- With probability \( 1 - \epsilon \) select \( a = \arg\max_{a \in A} \hat{Q}(a) \)
- With probability \( \epsilon \) select a random action

Constant \( \epsilon \) causes a positive lower bound on regret:

\[
I_t \geq \frac{\epsilon}{A} \sum_{a \in A} \Delta_a
\]

\( \Rightarrow \) \( \epsilon \)-greedy has linear total regret
action_counts = [0 for a in range(n_actions)]
total_reward_per_action = [0 for a in range(n_actions)]

for t in range(inf):
    Qhat_a = [total_reward_per_action[a] / action_counts[a] for a in range(n_actions)]
    a_t = argmax(Qhat_a)
    if random_uniform_between_0_and_1() < epsilon:
        a_t = random_action(n_actions)
    r_t = take_action_and_get_random_reward(a_t)
    action_counts[a_t] += 1
    total_reward_per_action[a_t] += r_t
Optimistic Initialisation

- Simple and practical idea: initialise $Q(a)$ to high value
- Update action value by incremental Monte-Carlo evaluation
- Starting with $N(a) > 0$

$$
\hat{Q}_t(a_t) = \hat{Q}_{t-1} + \frac{1}{N_t(a_t)}(r_t - \hat{Q}_{t-1})
$$

- Encourages systematic exploration early on
- But can still lock onto suboptimal action
- $\Rightarrow$ greedy + optimistic initialisation has linear total regret
- $\Rightarrow \epsilon$-greedy + optimistic initialisation has linear total regret
action_counts = [5 for a in range(n_actions)]
total_reward_per_action = [10 for a in range(n_actions)]

for t in range(inf):
    Qhat_a = [total_reward_per_action[a] / action_counts[a] for a in range(n_actions)]
    a_t = argmax(Qhat_a)
    r_t = take_action_and_get_random_reward(a_t)
    action_counts[a_t] += 1
    total_reward_per_action[a_t] += r_t

    if random_uniform_between_0_and_1() < epsilon:
        a_t = random_action(n_actions)

    r_t = take_action_and_get_random_reward(a_t)
    action_counts[a_t] += 1
    total_reward_per_action[a_t] += r_t
Decaying $\epsilon_t$-Greedy Algorithm

- Pick a decay schedule for $\epsilon_1, \epsilon_2, \ldots$.
- Consider the following schedule

$$c > 0$$

$$d = \min_{a: \Delta_a > 0} \Delta_i$$

$$\epsilon_t = \min \left\{ 1, \frac{c|\mathcal{A}|}{d^2 t} \right\}$$

- Decaying $\epsilon_t$-greedy has logarithmic asymptotic total regret!
- Unfortunately, schedule requires advance knowledge of gaps
- Goal: find an algorithm with sublinear regret for any multi-armed bandit (without knowledge of $R$)
action_counts = [0 for a in range(n_actions)]
total_reward_per_action = [0 for a in range(n_actions)]

for t in range(inf):
    Qhat_a = [total_reward_per_action[a] / action_counts[a] for a in range(n_actions)]
    a_t = argmax(Qhat_a)
    epsilon_t = min{1, starting_epsilon / (t+1)}
    if random_uniform_between_0_and_1() < epsilon_t:
        a_t = random_action(n_actions)
    r_t = take_action_and_get_random_reward(a_t)
    action_counts[a_t] += 1
    total_reward_per_action[a_t] += r_t
Lower Bound

- The performance of any algorithm is determined by similarity between optimal arm and other arms.
- Hard problems have similar-looking arms with different means.
- This is described formally by the gap $\Delta_a$ and the similarity in distributions $KL(\mathcal{R}^a \parallel \mathcal{R}^{a*})$.

**Theorem** (Lai and Robbins)

*Asymptotic total regret is at least logarithmic in number of steps.*

For large enough $t$: 

$$L_t \geq \log t \sum_{a | \Delta_a > 0} \frac{\Delta_a}{KL(\mathcal{R}^a \parallel \mathcal{R}^{a*})}$$

- expected reward of best action -
- expected reward of a
- a measure of how different
- the reward distributions are
Regret

- The **action-value** is the mean reward for action $a$,
  \[ Q(a) = \mathbb{E}[r|a] \]

- The **optimal value** $V^*$ is
  \[ V^* = Q(a^*) = \max_{a \in A} Q(a) \]

- The **regret** is the opportunity loss for one step
  \[ l_t = \mathbb{E}[V^* - Q(a_t)] \]

- The **total regret** is the total opportunity loss
  \[ L_t = \mathbb{E} \left[ \sum_{\tau=1}^{t} V^* - Q(a_\tau) \right] \]

- The **gap** $\Delta_a$ is the difference in value between action $a$ and optimal action $a^*$, $\Delta_a = V^* - Q(a)$
Optimism in the Face of Uncertainty

- Which action should we pick?
- The more uncertain we are about an action-value
- The more important it is to explore that action
- It could turn out to be the best action
Optimism in the Face of Uncertainty (2)

- After picking blue action
- We are less uncertain about the value
- And more likely to pick another action
- Until we home in on best action
Upper Confidence Bounds

- Estimate an upper confidence $\hat{U}_t(a)$ for each action value
- Such that $Q(a) \leq \hat{Q}_t(a) + \hat{U}_t(a)$ with high probability
- This depends on the number of times $N(a)$ has been selected
  - Small $N_t(a) \Rightarrow$ large $\hat{U}_t(a)$ (estimated value is uncertain)
  - Large $N_t(a) \Rightarrow$ small $\hat{U}_t(a)$ (estimated value is accurate)
- Select action maximising Upper Confidence Bound (UCB)

$$a_t = \arg\max_{a \in A} \hat{Q}_t(a) + \hat{U}_t(a)$$
Hoeffding’s Inequality

Theorem (Hoeffding’s Inequality)

Let $X_1, \ldots, X_t$ be i.i.d. random variables in $[0,1]$, and let 
$\overline{X}_t = \frac{1}{t} \sum_{\tau=1}^{t} X_\tau$ be the sample mean. Then

$$\mathbb{P} \left[ \mathbb{E} [X] > \overline{X}_t + u \right] \leq e^{-2tu^2}$$

- We will apply Hoeffding’s Inequality to rewards of the bandit
- conditioned on selecting action $a$

$$\mathbb{P} \left[ Q(a) > \hat{Q}_t(a) + U_t(a) \right] \leq e^{-2N_t(a)U_t(a)^2}$$
Calculating Upper Confidence Bounds

- Pick a probability $p$ that true value exceeds UCB
- Now solve for $U_t(a)$

$$e^{-2N_t(a)U_t(a)^2} = p$$

$$U_t(a) = \sqrt{\frac{- \log p}{2N_t(a)}}$$

- Reduce $p$ as we observe more rewards, e.g. $p = t^{-4}$
- Ensures we select optimal action as $t \to \infty$

$$U_t(a) = \sqrt{\frac{2 \log t}{N_t(a)}}$$
This leads to the UCB1 algorithm

\[ a_t = \arg\max_{a \in A} Q(a) + \sqrt{\frac{2 \log t}{N_t(a)}} \]

**Theorem**

The UCB algorithm achieves logarithmic asymptotic total regret

For large enough \( t \):

\[ L_t \leq 8 \log t \sum_{a \mid \Delta_a > 0} \Delta_a \]
action_counts = [0 for a in range(n_actions)]
total_reward_per_action = [0 for a in range(n_actions)]

for t in range(inf):
    Qhat_a = [total_reward_per_action[a] / action_counts[a] for a in range(n_actions)]
    Qoptimist_a = [Qhat_a[a] + sqrt(2 log t / action_counts[a]) for a in range(n_actions)]
    a_t = argmax(Qoptimist_a)
    r_t = take_action_and_get_random_reward(a_t)
    action_counts[a_t] += 1
    total_reward_per_action[a_t] += r_t
Why optimism?

• If you’re optimistic when you don’t know...
  • you check it out and learn the truth
• If you’re pessimistic when you don’t know...
  • you never check it out and never learn the truth
• Definition of *admissible* for a heuristic?
• If there’s one environment with lots of dangers and another environment with no major dangers...
  • Which environment would you rather be optimistic in?
  • Which environment would you rather be pessimistic in?
So far we have made no assumptions about the reward distribution $\mathcal{R}$
- Except bounds on rewards
- **Bayesian bandits** exploit prior knowledge of rewards, $p[\mathcal{R}]$
- They compute posterior distribution of rewards $p[\mathcal{R} | h_t]$
  - where $h_t = a_1, r_1, ..., a_{t-1}, r_{t-1}$ is the history
- Use posterior to guide exploration
  - Upper confidence bounds (Bayesian UCB)
  - Probability matching (Thompson sampling)
- Better performance if prior knowledge is accurate
• Maintain Gaussian probability distributions over the expected reward for each action
• Choose the action that maximizes mean + c * standard deviation
Probability matching selects action $a$ according to probability that $a$ is the optimal action

$$\pi(a \mid h_t) = \mathbb{P} \left[ Q(a) > Q(a'), \forall a' \neq a \mid h_t \right]$$

- Probability matching is optimistic in the face of uncertainty
  - Uncertain actions have higher probability of being max
- Can be difficult to compute analytically from posterior
Thompson Sampling

- Thompson sampling implements probability matching
  \[
  \pi(a \mid h_t) = \mathbb{P} \left[ Q(a) > Q(a'), \forall a' \neq a \mid h_t \right] \\
  = \mathbb{E}_{\mathcal{R} \mid h_t} \left[ \mathbf{1}(a = \arg\max_{a \in A} Q(a)) \right]
  \]

- Use Bayes law to compute posterior distribution \( p[\mathcal{R} \mid h_t] \)
- Sample a reward distribution \( \mathcal{R} \) from posterior
- Compute action-value function \( Q(a) = \mathbb{E}[\mathcal{R}_a] \)
- Select action maximising value on sample, \( a_t = \arg\max_{a \in A} Q(a) \)
- Thompson sampling achieves Lai and Robbins lower bound!
Value of Information

- Exploration is useful because it gains information
- Can we quantify the value of information?
  - How much reward a decision-maker would be prepared to pay in order to have that information, prior to making a decision
  - Long-term reward after getting information - immediate reward
- Information gain is higher in uncertain situations
- Therefore it makes sense to explore uncertain situations more
- If we know value of information, we can trade-off exploration and exploitation optimally
A contextual bandit is a tuple $\langle A, S, R \rangle$

- $A$ is a known set of actions (or “arms”)
- $S = \mathbb{P}[s]$ is an unknown distribution over states (or “contexts”)
- $R_s^a(r) = \mathbb{P}[r|s, a]$ is an unknown probability distribution over rewards

At each step $t$

- Environment generates state $s_t \sim S$
- Agent selects action $a_t \in A$
- Environment generates reward $r_t \sim R_{s_t}^{a_t}$

Goal is to maximise cumulative reward
$$\sum_{T=1}^{t} r_T$$
## Linear Regression

### How many of each kind of movie has the user watched?

<table>
<thead>
<tr>
<th>Oscar winner</th>
<th>Will Farrell vibes</th>
<th>Action</th>
<th>Slow pace</th>
<th>Documentary</th>
<th>Num Oscar noms</th>
<th>Year made</th>
<th>Will Farrell vibes</th>
<th>Slow pace</th>
<th>Action</th>
<th>Reward</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>2</td>
<td>0</td>
<td>0</td>
<td>2</td>
<td>2020</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>3</td>
<td>0</td>
<td>2017</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>5</td>
<td>1999</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>2008</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>2012</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>2</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>2024</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>-1</td>
</tr>
</tbody>
</table>

### Properties of recommended movie

**Reward:** 0 for not clicking; 1 for watching; -1 for watching part and quitting.
Linear Regression

- Action-value function is expected reward for state $s$ and action $a$

$$Q(s, a) = \mathbb{E}[r|s, a]$$

- Estimate value function with a linear function approximator

$$Q_\theta(s, a) = \phi(s, a)^\top \theta \approx Q(s, a)$$

- Estimate parameters by least squares regression

$$A_t = \sum_{\tau=1}^{t} \phi(s_\tau, a_\tau) \phi(s_\tau, a_\tau)^\top$$

$$b_t = \sum_{\tau=1}^{t} \phi(s_\tau, a_\tau) r_\tau$$

$$\theta_t = A_t^{-1} b_t$$
Least squares regression estimates the mean action-value $Q_\theta(s, a)$.

But it can also estimate the variance of the action-value $\sigma^2_\theta(s, a)$. 

*I.e. the uncertainty due to parameter estimation error.*

Add on a bonus for uncertainty, $U_\theta(s, a) = c\sigma$

*i.e. define UCB to be $c$ standard deviations above the mean.*
Define confidence ellipsoid $\mathcal{E}_t$ around parameters $\theta_t$

Such that $\mathcal{E}_t$ includes true parameters $\theta^*$ with high probability

Use this ellipsoid to estimate the uncertainty of action values

Pick parameters within ellipsoid that maximise action value

$$\arg\max_{\theta \in \mathcal{E}} Q_\theta(s, a)$$
The same principles for exploration/exploitation apply to MDPs
• We’ve seen how to compute the optimal policy in a known MDP
• We’ve just discussed a bandit setting with one state (no transitions)
  • But the reward function is unknown
• What if we are in an MDP, but we don’t know the transition function
  or the reward function?
Greedy (And Slow) RL Algorithm

• Finite state space, finite action space
• For simplicity, say reward only depends on new state

transition_counts = array of zeros of size $S \times A \times S$
state_counts = array of zeros of size $S$
rewards_by_state = array of zeros of size $S$
s = get_initial_state()
while True:
    transition_matrix = estimate_T(transition_counts)
    reward_function = estimate_R(state_counts, rewards_by_state)
    policy = solve_mdp(transition_matrix, reward_function)  # expensive!
    a = policy[s]
    s', r = get_next_state_and_reward(s, a)
    transition_counts[s][a][s'] += 1
    state_counts[s'] += 1
    rewards_by_state[s'] += r
Optimistic Initialisation: Model-Based RL

- Construct an optimistic model of the MDP
- Initialise transitions to go to heaven
  - (i.e. transition to terminal state with $r_{max}$ reward)
- Solve optimistic MDP by favourite planning algorithm
  - policy iteration
  - value iteration
  - tree search
  - ...
- Encourages systematic exploration of states and actions
- e.g. RMax algorithm (Brafman and Tennenholtz)
transition_counts = array of zeros of size (S+1) x A x (S+1)
state_counts = array of zeros of size S+1
rewards_by_state = array of zeros of size S+1
rewards_by_state[-1] = rmax # maximum possible reward ("heaven state")
state_counts[-1] = 1
for s in range(S+1):
    for a in range(A):
        transition_counts[s][a][-1] += 1  # pretend we've seen transition to heaven
s = get_initial_state()
while True:
    transition_matrix = estimate_T(transition_counts)  # real Rmax waits to update
    reward_function = estimate_R(state_counts, rewards_by_state)
policy = solve_mdp(transition_matrix, reward_function)
a = policy[s]
s', r = get_next_state_and_reward(s, a)
transition_counts[s][a][s'] += 1
state_counts[s'] += 1
rewards_by_state[s'] += r