Midterm 2 Review
Probability Recap

- **Conditional probability**
  \[ P(x|y) = \frac{P(x, y)}{P(y)} \]

- **Product rule**
  \[ P(x, y) = P(x|y)P(y) \]

- **Chain rule**
  \[ P(X_1, X_2, \ldots, X_n) = P(X_1)P(X_2|X_1)P(X_3|X_1, X_2)\ldots \]
  \[ = \prod_{i=1}^{n} P(X_i|X_1, \ldots, X_{i-1}) \]

- **X, Y independent if and only if:**
  \[ \forall x, y : P(x, y) = P(x)P(y) \]

- **X and Y are conditionally independent given Z if and only if:**
  \[ X \independent Y | Z \]
  \[ \forall x, y, z : P(x, y|z) = P(x|z)P(y|z) \]
Markov Decision Processes

- An MDP is defined by:
  - A set of states \( s \in S \)
  - A set of actions \( a \in A \)
  - A transition function \( T(s, a, s') \)
    - Probability that \( a \) from \( s \) leads to \( s' \), i.e., \( P(s'| s, a) \)
    - Also called the model or the dynamics
  - A reward function \( R(s, a, s') \)
    - Sometimes just \( R(s) \) or \( R(s') \)
  - A start state
  - Maybe a terminal state

- MDPs are non-deterministic search problems
  - One way to solve them is with expectimax search
  - We’ll have a new tool soon
What is Markov about MDPs?

- “Markov” generally means that given the present state, the future and the past are independent.

- For Markov decision processes, “Markov” means action outcomes depend only on the current state.

\[
P(S_{t+1} = s' | S_t = s_t, A_t = a_t, S_{t-1} = s_{t-1}, A_{t-1}, \ldots, S_0 = s_0)
= P(S_{t+1} = s' | S_t = s_t, A_t = a_t)
\]

- This is just like search, where the successor function could only depend on the current state (not the history).
Important Quantities

- The value (utility) of a state $s$: 
  \[ V^*(s) = \text{expected utility starting in } s \text{ and acting optimally} \]

- The value (utility) of a q-state $(s,a)$: 
  \[ Q^*(s,a) = \text{expected utility starting out having taken action } a \text{ from state } s \text{ and (thereafter) acting optimally} \]

- The optimal policy: 
  \[ \pi^*(s) = \text{optimal action from state } s \]
Bellman Equations

- Fundamental operation: compute the (expectimax) value of a state
  - Expected utility under optimal action
  - Average sum of (discounted) rewards
  - This is just what expectimax computed!

- Recursive definition of value:

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

  \[ Q^*(s, a) = \sum_{s'} T(s, a, s') \left[ R(s, a, s') + \gamma V^*(s') \right] \]

  \[ V^*(s) = \max_a \sum_{s'} T(s, a, s') \left[ R(s, a, s') + \gamma V^*(s') \right] \]
Start with $V_0(s) = 0$: no time steps left means an expected reward sum of zero.

Given vector of $V_k(s)$ values, do one ply of expectimax from each state:

$$V_{k+1}(s) \leftarrow \max_a \sum_{s'} T(s, a, s') \left[ R(s, a, s') + \gamma V_k(s') \right]$$

Repeat until convergence.

Complexity of each iteration: $O(S^2A)$

Theorem: will converge to unique optimal values
  - Basic idea: approximations get refined towards optimal values
  - Policy may converge long before values do
Policy Iteration

- Alternative approach for optimal values:
  - Step 1: Policy evaluation: calculate utilities for some fixed policy (not optimal utilities!) until convergence
  - Step 2: Policy improvement: update policy using one-step look-ahead with resulting converged (but not optimal!) utilities as future values
  - Repeat steps until policy converges

- This is policy iteration
  - It’s still optimal!
  - Can converge (much) faster under some conditions
Policy Iteration

- **Evaluation:** For fixed current policy $\pi$, find values with policy evaluation:
  - Iterate until values converge:
    \[
    V_{k+1}^{\pi_i}(s) \leftarrow \sum_{s'} T(s, \pi_i(s), s') \left[ R(s, \pi_i(s), s') + \gamma V_k^{\pi_i}(s') \right]
    \]

- **Improvement:** For fixed values, get a better policy using policy extraction
  - One-step look-ahead:
    \[
    \pi_{i+1}(s) = \arg \max_a \sum_{s'} T(s, a, s') \left[ R(s, a, s') + \gamma V^{\pi_i}(s') \right]
    \]
Temporal Difference Learning

- **Big idea: learn from every experience!**
  - Update $V(s)$ each time we experience a transition $(s, a, s', r)$
  - Likely outcomes $s'$ will contribute updates more often

- **Temporal difference learning of values**
  - Policy still fixed, still doing evaluation!
  - Move values toward value of whatever successor occurs: running average

Sample of $V(s)$:

$$sample = R(s, \pi(s), s') + \gamma V^\pi(s')$$

Update to $V(s)$:

$$V^\pi(s) \leftarrow (1 - \alpha)V^\pi(s) + (\alpha)sample$$

Same update:

$$V^\pi(s) \leftarrow V^\pi(s) + \alpha(sample - V^\pi(s))$$
Q-Learning

- **Q-Learning** sample-based Q-value iteration

\[ Q_{k+1}(s, a) \leftarrow \sum_{s'} T(s, a, s') \left[ R(s, a, s') + \gamma \max_{a'} Q_k(s', a') \right] \]

- **Learn Q(s,a) values as you go**
  - Receive a sample \((s,a,s',r)\)
  - Consider your old estimate: \(Q(s, a)\)
  - Consider your new sample estimate:

\[ \text{sample} = R(s, a, s') + \gamma \max_{a'} Q(s', a') \]

- Incorporate the new estimate into a running average:

\[ Q(s, a) \leftarrow (1 - \alpha)Q(s, a) + (\alpha)[\text{sample}] \]
Using a feature representation, we can write a q function (or value function) for any state using a few weights:

\[ V(s) = w_1 f_1(s) + w_2 f_2(s) + \ldots + w_n f_n(s) \]

\[ Q(s, a) = w_1 f_1(s, a) + w_2 f_2(s, a) + \ldots + w_n f_n(s, a) \]

- Advantage: our experience is summed up in a few powerful numbers

- Disadvantage: states may share features but actually be very different in value!
Approximate Q-Learning

\[ Q(s, a) = w_1 f_1(s, a) + w_2 f_2(s, a) + \ldots + w_n f_n(s, a) \]

- **Q-learning with linear Q-functions:**
  
  transition = (s, a, r, s')
  
  difference = \[ r + \gamma \max_{a'} Q(s', a') \] - Q(s, a)
  
  \[ Q(s, a) \leftarrow Q(s, a) + \alpha \text{ [difference]} \]

- **Intuitive interpretation:**
  - Adjust weights of active features
  - E.g., if something unexpectedly bad happens, blame the features that were on: disprefer all states with that state’s features

- **Formal justification:** online least squares
DQN

- Approximate Q-Learning at scale.
- Uses Neural Network for Q-value function approximation.
Two approaches to model-free RL

- **Learn Q-values**
  - Trains Q-values to be consistent. Not directly optimizing for performance.
  - Use an objective based on the Bellman Equation

\[
Q_{k+1}(s, a) \leftarrow \sum_{s'} T(s, a, s') \left[ R(s, a, s') + \gamma \max_{a'} Q_k(s', a') \right]
\]

- **Learn Policy Directly**
  - Have a parameterized policy \( \pi_\theta \)
  - Update the parameters \( \theta \) to optimize performance of policy.
Find a policy that maximizes expected utility (discounted cumulative rewards)

\[ \pi^* = \arg\max_{\pi} E_{\pi} \left[ \sum_{t=0}^{\infty} \gamma^t R(s, \pi(s), s') \right] \]
Notation

- **Trajectory (rollout, episode)** $\tau = (s_0, a_0, s_1, a_1, \ldots)$
  - $s_0 \sim \rho_0(\cdot)$ (initial state distribution)
  - $s_{t+1} \sim P(\cdot | s_t, a_t)$ (transition probabilities)
- **Rewards** $r_t = R(s_t, a_t, s_{t+1})$
- **Finite-horizon undiscounted return of a trajectory**
  \[ R(\tau) = \sum_{t=0}^{T} r_t \]
- **Actions are sampled from a stochastic parameterized policy** $\pi_\theta$
  \[ a_t \sim \pi_\theta(\cdot | s_t) \]
Notation

- Probability of a trajectory (rollout, episode) $\tau = (s_0, a_0, s_1, a_1, \ldots)$
  \[ P(\tau|\pi) = \rho_0(s_0) \prod_{t=0}^{T-1} P(s_{t+1}|s_t, a_t)\pi_\theta(a_t|s_t) \]

- Expected Return of a policy $J(\pi)$
  \[ J(\pi) = \sum_\tau P(\tau|\pi) R(\tau) = E_{\tau \sim \pi}[R(\tau)] \]

- Goal of RL: Solve the following optimization problem
  \[ \pi^* = \arg\max_{\pi} J(\pi) \]
We can now perform gradient ascent to improve our policy!

\[ \theta_{k+1} \leftarrow \theta_k + \alpha \nabla_\theta J(\pi_\theta) \bigg|_{\theta_k} \]

\[ \nabla_\theta J(\pi_\theta) = E_{\tau \sim \pi_\theta} \left[ \sum_{t=0}^{T} \nabla_\theta \log \pi_\theta(a_t|s_t) \ R(\tau) \right] \]

Estimate with a sample mean over a set D of policy rollouts given current parameters:

\[ \approx \frac{1}{|D|} \sum_{\tau \in D} \sum_{t=0}^{T} (\nabla_\theta \log \pi_\theta(a_t|s_t) \ R(\tau)) \]
There will be one short answer question about AlphaGo.

Review high-level ideas from slides. No need to understand how everything works nor how to implement.