CS 5522: Artificial Intelligence II
Reinforcement Learning III (Advanced Topics)

Instructor: Wei Xu
Ohio State University

[These slides were adapted from CS188 Intro to AI at UC Berkeley; and David Silver, Pieter Abbeel, John Schulman, Vien Ngo, Marc Toussaint, ]
Reinforcement Learning

- We still assume an MDP:
  - A set of states $s \in S$
  - A set of actions (per state) $A$
  - A model $T(s,a,s')$
  - A reward function $R(s,a,s')$
- Still looking for a policy $\pi(s)$

- New twist: don’t know $T$ or $R$, so must try out actions

- Big idea: Compute all averages over $T$ using sample outcomes
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:

$$V^\pi(s) \leftarrow V^\pi(s) + \alpha(sample - V^\pi(s))$$
A Unified View*: Temporal-Difference

\[ V(S_t) \leftarrow V(S_t) + \alpha (R_{t+1} + \gamma V(S_{t+1}) - V(S_t)) \]
A Unified View*: Monte-Carlo (Direct Estimation)

\[ V(S_t) \leftarrow V(S_t) + \alpha (G_t - V(S_t)) \]
A Unified View*: Bellman Updates (Dynamic Programming)

\[ V(S_t) \leftarrow \mathbb{E}_\pi [R_{t+1} + \gamma V(S_{t+1})] \]
Large-Scale Reinforcement Learning

- Reinforcement learning can be used to solve large problems
  - Backgammon: $10^{20}$ states
  - Go: $10^{170}$ states
  - Helicopter: continuous state space

- Solution:
  - Estimate value function with function approximation
  - Generalize from seen states to unseen states
  - Update (a small number of) parameters using temporal-difference learning (or Monte-Carlo* methods).
Approximate Q-Learning
Generalizing Across States

- Basic Q-Learning keeps a table of all q-values.
- In realistic situations, we cannot possibly learn about every single state!
  - Too many states to visit them all in training
  - Too many states to hold the q-tables in memory
- Instead, we want to generalize:
  - Learn about some small number of training states from experience
  - Generalize that experience to new, similar situations
  - This is a fundamental idea in machine learning, and we’ll see it over and over again

[Image of robot reading a book and Pacman game]
Example: Pacman

Let’s say we discover through experience that this state is bad:

In naïve q-learning, we know nothing about this state:

Or even this one!

[Demo: Q-learning - pacman - tiny - watch all (L11D5)]
[Demo: Q-learning - pacman - tiny - silent train (L11D6)]
[Demo: Q-learning - pacman - tricky - watch all (L11D7)]
Solution: describe a state using a vector of features (properties)

- Features are functions from states to real numbers (often 0/1) that capture important properties of the state
- Example features:
  - Distance to closest ghost
  - Distance to closest dot
  - Number of ghosts
  - $1 / (\text{dist to dot})^2$
  - Is Pacman in a tunnel? (0/1)
  - ...... etc.
  - Is it the exact state on this slide?
- Can also describe a q-state $(s, a)$ with features (e.g. action moves closer to food)
Linear Value Functions

- 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 \( = [r + \gamma \max_{a'} Q(s', a')] - Q(s, a) \)
  
  \[ Q(s, a) \leftarrow Q(s, a) + \alpha \text{[difference]} \]
  
  \[ w_i \leftarrow w_i + \alpha \text{[difference]} f_i(s, a) \]  

- **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
Minimizing Error

Imagine we had only one point \( x \), with features \( f(x) \), target value \( y \), and weights \( w \); minimizing least square errors by Stochastic Gradient Decent:

\[
\text{error}(w) = \frac{1}{2} \left( y - \sum_k w_k f_k(x) \right)^2
\]

\[
\frac{\partial \text{error}(w)}{\partial w_m} = - \left( y - \sum_k w_k f_k(x) \right) f_m(x)
\]

\[
w_m \leftarrow w_m + \alpha \left( y - \sum_k w_k f_k(x) \right) f_m(x)
\]

Approximate q update explained:

\[
w_m \leftarrow w_m + \alpha \left[ r + \gamma \max_a Q(s', a') - Q(s, a) \right] f_m(s, a)
\]

“target” “prediction”
Various Function Approximator *

- We can consider differentiable function approximators, e.g.
  - Linear combinations of features
  - Neural network
  - Decision tree
  - Nearest neighbor
  - ...

15
Deep Reinforcement Learning in Atari *

state $s_t$

reward $r_t$

action $a_t$
Deep Q-Network in Atari *

- End-to-end learning of values $Q(s, a)$ from pixels $s$
- Input state $s$ is stack of raw pixels from last 4 frames
- Output is $Q(s, a)$ for 18 joystick/button positions
- Reward is change in score for that step

Network architecture and hyperparameters fixed across all games  

[Mnih et al. 2013; 2016]
DQN Results in Atari

[Diagram showing performance of DQN in various Atari games, with percentages indicating the level of performance compared to human level for each game.]
Demo: DQN for Atari Breakout*

https://www.youtube.com/watch?v=TmPfTpjtdgg
Demo: DQN for Space Invaders*

https://www.youtube.com/watch?v=W2CAghUiofY
Summary: Approaches To Reinforcement Learning

(experience data)

- $s, a, r, s', a', r', s'', a'', r'', s'''' \ldots$

**Model-based RL**
- learn MDP model
  - $\hat{T}(s, a, s')$
  - $\hat{R}(s, a, s')$
- dynamic programming
  - $V^*(s)$ Value Iteration
- policy
  - $\pi^*(s)$
- Model-based RL

**Value-based RL**
- learn value functions
  - $V^*(s)$ or $Q^*(s, a)$
- policy
  - $\pi^*(s)$
  - e.g. $\epsilon$-greedy
- Value-based RL

**Model-free RL**
- optimize policy
  - $\pi^*(s)$
  - Q-learning!
- Policy Search
- Policy-based RL
- Model-free RL
Policy Search
Policy Search

- Simplest policy search: start with an ok solution (e.g. Q-learning) then fine-tune by hill climbing on feature weights
  - Start with an initial linear value function or Q-function
  - Nudge each feature weight up and down and see if your policy is better than before (hill climbing)

- learn policies that maximize rewards, not the values that predict them
Why Policy Optimization?

- Often $\pi$ can be simpler than $Q$ or $V$:
  - Q: need to efficiently solve $\arg\max_a Q_w(s, a)$
  - Challenge for continuous / high-dimensional action spaces (e.g. robotic grasp)

- Often the feature-based policies that work well (win games, maximize utilities) aren’t the ones that approximate $V / Q$ best
  - E.g. your value functions from project 2 were probably horrible estimates of future rewards, but they still produced good decisions
  - Q-learning’s priority: get Q-values close (modeling)
  - Action selection priority: get ordering of Q-values right (prediction)
  - We’ll see this distinction between modeling and prediction again later in the course

- Can learn stochastic policies
  - We can also parametrize the policy $\pi_\theta(s, a) = P(a|s, \theta)$
Introduction

Rock-Paper-Scissors Example

Two-player game of rock-paper-scissors

- Scissors beats paper
- Rock beats scissors
- Paper beats rock

Consider policies for *iterated* rock-paper-scissors

- A deterministic policy is easily exploited
- A uniform random policy is optimal
Policy Objective Functions

- Goal: given policy $\pi_\theta(s, a)$ with parameters $\theta$, find best $\theta$
- But how do we measure the quality of a policy $\pi_\theta$?
- In episodic environments we can use the start value

$$J_1(\theta) = V^{\pi_\theta}(s_1) = \mathbb{E}_{\pi_\theta}[v_1]$$

- In continuing environments we can use the average value

$$J_{avV}(\theta) = \sum_s d^{\pi_\theta}(s) V^{\pi_\theta}(s)$$

- Or the average reward per time-step

$$J_{avR}(\theta) = \sum_s d^{\pi_\theta}(s) \sum_a \pi_\theta(s, a) R_s^a$$

- where $d^{\pi_\theta}(s)$ is stationary distribution of Markov chain for $\pi_\theta$
Policy Optimization *

- Policy-based reinforcement learning is an **optimisation** problem

- Find $\theta$ that maximises $J(\theta)$

- Some approaches do not use gradient
  - Hill climbing
  - Simplex / amoeba / Nelder Mead
  - Genetic algorithms

- Greater efficiency often possible using gradient
  - Gradient descent
  - Conjugate gradient
  - Quasi-newton

- We focus on gradient descent, many extensions possible
- And on methods that exploit sequential structure
Policy Gradient *

- Let $J(\theta)$ be any policy objective function
- Policy gradient algorithms search for a \textit{local} maximum in $J(\theta)$ by ascending the gradient of the policy, w.r.t. parameters $\theta$

$$\Delta \theta = \alpha \nabla_{\theta} J(\theta)$$

- Where $\nabla_{\theta} J(\theta)$ is the \textit{policy gradient}

$$\nabla_{\theta} J(\theta) = \left( \begin{array}{c} \frac{\partial J(\theta)}{\partial \theta_1} \\ \vdots \\ \frac{\partial J(\theta)}{\partial \theta_n} \end{array} \right)$$

- and $\alpha$ is a step-size parameter
To evaluate policy gradient of $\pi_\theta(s, a)$

For each dimension $k \in [1, n]$

- Estimate $k$th partial derivative of objective function w.r.t. $\theta$
- By perturbing $\theta$ by small amount $\epsilon$ in $k$th dimension

$$\frac{\partial J(\theta)}{\partial \theta_k} \approx \frac{J(\theta + \epsilon u_k) - J(\theta)}{\epsilon}$$

where $u_k$ is unit vector with 1 in $k$th component, 0 elsewhere

- Uses $n$ evaluations to compute policy gradient in $n$ dimensions
- Simple, noisy, inefficient - but sometimes effective
- Works for arbitrary policies, even if policy is not differentiable
Example: AIBO Walk by Finite Difference Policy Gradient

- **Goal:** learn a fast AIBO walk (useful for Robocup)
- **AIBO walk policy** is controlled by 12 numbers (elliptical loci)
- Adapt these parameters by finite difference policy gradient
- Evaluate performance of policy by field traversal time

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Initial Value</th>
<th>Best Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Front height</td>
<td>0.05</td>
<td>0.35</td>
</tr>
<tr>
<td>Front loci (x offset)</td>
<td>0.35</td>
<td>4.081</td>
</tr>
<tr>
<td>Front loci (y offset)</td>
<td>-2.8</td>
<td>5.285</td>
</tr>
<tr>
<td>Locus skew multiplier</td>
<td>0.035</td>
<td>0.049</td>
</tr>
<tr>
<td>Front time through locus</td>
<td>5.6</td>
<td>10.843</td>
</tr>
<tr>
<td>Rear height</td>
<td>0.05</td>
<td>0.35</td>
</tr>
<tr>
<td>Rear loci (x offset)</td>
<td>0.35</td>
<td>4.081</td>
</tr>
<tr>
<td>Rear loci (y offset)</td>
<td>-2.8</td>
<td>5.285</td>
</tr>
<tr>
<td>Rear time through locus</td>
<td>5.6</td>
<td>10.843</td>
</tr>
</tbody>
</table>

Figure 1. Maximum forward velocities of the best current gaits. For the elliptical height, the half-elliptical locus used by our team is shown in Figure 2. The turning rate of the Aibo is used to determine the skew of each foot.

Figure 5. The training environment for our experiments. Each Aibo is trained on a large green field resembling a RoboCup field. The Aibos are controlled by computer algorithms that are updated during the training process.

Figure 6. The velocity of the best policy learned after 23 iterations. All values are given in millimeters per second (mm/s).

- There is video of the training process at: [http://www.cs.cmu.edu/~AmericanOpen03/](http://www.cs.cmu.edu/~AmericanOpen03/)
- For information on the UT Austin Villa team, visit: [http://www.cs.utexas.edu/~AustinVilla/legged/learned-walk/](http://www.cs.utexas.edu/~AustinVilla/legged/learned-walk/)
- There is also video of the learning process:
  - [http://www.cs.cmu.edu/~AmericanOpen03/](http://www.cs.cmu.edu/~AmericanOpen03/)

Our team (UT Austin Villa) first approached the gait problem in 2002. The CMU German team used a simplified version of the parameterization discussed here. We use the same parameter assignments as defining open-loop policy that can be tuned to walk at 245 mm/s in the RoboCup 2003 competition.
We now compute the policy gradient *analytically*

Assume policy $\pi_\theta$ is differentiable whenever it is non-zero

and we know the gradient $\nabla_\theta \pi_\theta(s, a)$

**Likelihood ratios** exploit the following identity

$$
\nabla_\theta \pi_\theta(s, a) = \pi_\theta(s, a) \frac{\nabla_\theta \pi_\theta(s, a)}{\pi_\theta(s, a)}
$$

$$
= \pi_\theta(s, a) \nabla_\theta \log \pi_\theta(s, a)
$$

The *score function* is $\nabla_\theta \log \pi_\theta(s, a)$
Softmax Policy *

- Weight actions using linear combination of features $\phi(s, a)^\top \theta$
- Probability of action is proportional to exponentiated weight

$$\pi_\theta(s, a) \propto e^{\phi(s, a)^\top \theta}$$

- The score function is

$$\nabla_\theta \log \pi_\theta(s, a) = \phi(s, a) - \mathbb{E}_{\pi_\theta} [\phi(s, \cdot)]$$
One-step MDPs *

- Consider a simple class of one-step MDPs
  - Starting in state \( s \sim d(s) \)
  - Terminating after one time-step with reward \( r = \mathcal{R}_{s,a} \)
- Use likelihood ratios to compute the policy gradient

\[
J(\theta) = \mathbb{E}_{\pi_\theta}[r] \\
= \sum_{s \in S} d(s) \sum_{a \in A} \pi_\theta(s, a) \mathcal{R}_{s,a} \\
\nabla_\theta J(\theta) = \sum_{s \in S} d(s) \sum_{a \in A} \pi_\theta(s, a) \nabla_\theta \log \pi_\theta(s, a) \mathcal{R}_{s,a} \\
= \mathbb{E}_{\pi_\theta} [\nabla_\theta \log \pi_\theta(s, a) r]
\]
The policy gradient theorem generalises the likelihood ratio approach to multi-step MDPs.

- Replaces instantaneous reward $r$ with long-term value $Q^\pi(s, a)$.
- Policy gradient theorem applies to start state objective, average reward and average value objective.

**Theorem**

For any differentiable policy $\pi_\theta(s, a)$, for any of the policy objective functions $J = J_1, J_{avR}$, or $\frac{1}{1-\gamma} J_{avV}$, the policy gradient is

$$\nabla_\theta J(\theta) = \mathbb{E}_{\pi_\theta} [\nabla_\theta \log \pi_\theta(s, a) \, Q^{\pi_\theta}(s, a)]$$
Policy Gradient Intuition *

- Increase probability of paths with positive $R$
- Decrease probability of paths with negative $R$
Example: Policy Optimization Success Stories

Kohl and Stone, 2004
Ng et al, 2004
Tedrake et al, 2005
Kober and Peters, 2009

Mnih et al, 2015 (A3C)
Silver et al, 2014 (DPG)
Lillicrap et al, 2015 (DDPG)

Schulman et al, 2016 (TRPO + GAE)
Levine*, Finn*, et al, 2016 (GPS)

Silver*, Huang*, et al, 2016 (AlphaGo**)
Demo: Autonomous Helicopter

[Ng et al. 2004; Abbeel et al. 2010]

https://www.youtube.com/watch?v=VCdxqn0fcnE
Reinforcement Learning Landscape *

- **Policy Optimization**
  - DFO / Evolution
  - Policy Gradients
  - Actor-Critic Methods

- **Dynamic Programming**
  - Policy Iteration
  - Value Iteration
  - Q-Learning

- **DQNs**
  - Mnih et al., Nature 2015
  - Double DQN: Van Hasselt et al, AAAI 2015
  - Dueling Architecture: Wang et al, ICML 2016
  - Prioritized Replay: Schaul et al, ICLR 2016
  - David Silver ICML 2016 tutorial
We’re done with Part I: Search and Planning!

We’ve seen how AI methods can solve problems in:
- Search
- Games
- Markov Decision Problems
- Reinforcement Learning

Next up: Part II: Uncertainty and Learning!