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

\[
\text{Sample of } V(s): \quad \text{sample} = R(s, \pi(s), s') + \gamma V^\pi(s')
\]

\[
\text{Update to } V(s): \quad V^\pi(s) \leftarrow (1 - \alpha)V^\pi(s) + (\alpha)\text{sample}
\]

\[
\text{Same update:} \quad V^\pi(s) \leftarrow V^\pi(s) + \alpha(\text{sample} - V^\pi(s))
\]
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)

$$V(S_t) \leftarrow \mathbb{E}_\pi [R_{t+1} + \gamma V(S_{t+1})]$$
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
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)
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

- **Exact Q's**
  
  - \(Q(s, a)\)

- **Approximate Q's**
  
  - \(w_i \leftarrow w_i + \alpha \ [\text{difference}] f_i(s, a)\)
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
  - ...
Deep Reinforcement Learning in Atari *
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]
<table>
<thead>
<tr>
<th>Game</th>
<th>DQN Results</th>
<th>Baseline (at human-level or above)</th>
<th>Baseline (below human-level)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Video Pinball</td>
<td>396%</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Boxing</td>
<td>1064%</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Breakout</td>
<td>114%</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Star Gunner</td>
<td>4%</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Robotank</td>
<td>273%</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Atlantis</td>
<td>308%</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Crazy Climber</td>
<td>419%</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Gopher</td>
<td>102%</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Demon Attack</td>
<td>294%</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Name This Game</td>
<td>11%</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Krull</td>
<td>31%</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Assault</td>
<td>31%</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Road Runner</td>
<td>51%</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Kangaroo</td>
<td>52%</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Tennis</td>
<td>148%</td>
<td></td>
<td></td>
</tr>
<tr>
<td>James Bond</td>
<td>148%</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Pong</td>
<td>58%</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Space Invaders</td>
<td>7%</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Beam Rider</td>
<td>20%</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Tutankham</td>
<td>112%</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Kung-Fu Master</td>
<td>128%</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Freeway</td>
<td>102%</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Time Pilot</td>
<td>18%</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Enduro</td>
<td>97%</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Fishing Derby</td>
<td>93%</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Up and Down</td>
<td>92%</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Ice Hockey</td>
<td>79%</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Q*Bert</td>
<td>78%</td>
<td></td>
<td></td>
</tr>
<tr>
<td>H.E.R.O.</td>
<td>76%</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Asterix</td>
<td>69%</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Battle Zone</td>
<td>67%</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Wizard of Wor</td>
<td>67%</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Chopper Command</td>
<td>64%</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Centipede</td>
<td>62%</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Bank Heist</td>
<td>57%</td>
<td></td>
<td></td>
</tr>
<tr>
<td>River Raid</td>
<td>57%</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Amidar</td>
<td>43%</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Alien</td>
<td>42%</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Venture</td>
<td>32%</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Seastorm</td>
<td>25%</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Double Dink</td>
<td>17%</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Bowling</td>
<td>14%</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Ms. Pacman</td>
<td>13%</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Asteroids</td>
<td>7%</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Frostbite</td>
<td>6%</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Gravitar</td>
<td>5%</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Private Eye</td>
<td>2%</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Montezuma’s Revenge</td>
<td>0%</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
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
    \(\pi^*(s)\)

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

- **Model-free RL**
  - optimize policy
    \(\pi^*(s)\)

- **Policy-based RL**
  - Policy Search
  - Q-learning!
Policy Search
Policy Search

- Simplest policy search:
  - 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
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 $\pi_\theta(s, a) = P(a | s, \theta)$
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-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 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 policy gradient

$$\nabla_{\theta} J(\theta) = \begin{pmatrix} \frac{\partial J(\theta)}{\partial \theta_1} \\ \vdots \\ \frac{\partial J(\theta)}{\partial \theta_n} \end{pmatrix}$$

- 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
We now compute the policy gradient \textit{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 $\phi(s, a)^T \theta$
- Probability of action is proportional to exponentiated weight

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

- The score function is

$$\nabla_\theta \log \pi_\theta(s, a) = \phi(s, a) - E_{\pi_\theta} [\phi(s, \cdot)]$$
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 \(r = R_{s,a}\)
- Use likelihood ratios to compute the policy gradient

\[
J(\theta) = \mathbb{E}_{\pi_\theta} [r] \\
= \sum_{s \in S} d(s) \sum_{a \in A} \pi_\theta(s, a) R_{s,a}
\]

\[
\nabla_\theta J(\theta) = \sum_{s \in S} d(s) \sum_{a \in A} \pi_\theta(s, a) \nabla_\theta \log \pi_\theta(s, a) R_{s,a}
\]

\[
= \mathbb{E}_{\pi_\theta} [\nabla_\theta \log \pi_\theta(s, a) r]
\]
Policy Gradient Theorem

- 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

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

https://www.youtube.com/watch?v=VCdxqn0fcnE

[Ng et al. 2004; Abbeel et al. 2010]
Reinforcement Learning Landscape *

DQN: 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
Conclusion

- 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!