CSC413/2516 Lecture 11: Q-Learning & the Game of Go

Jimmy Ba and Bo Wang
Overview

- Reinforcement learning for deep learners
  - Previously, we have seen supervised learning and unsupervised learning with neural networks.
- Today: Q-learning
  - Learn an action-value function that predicts future returns
- Case study: AlphaGo uses both a policy network and a value network
Overview

- Agent interacts with an environment, which we treat as a black box.
- Your RL code accesses it only through an API since it’s external to the agent.
  - I.e., you’re not “allowed” to inspect the transition probabilities, reward distributions, etc.

![Diagram of Agent and Environment Interaction]

State: s
Reward: r

Agent

Actions: a

Environment
Recap: Markov Decision Processes

- The environment is represented as a Markov decision process (MDP) $\mathcal{M}$.
- Markov assumption: all relevant information is encapsulated in the current state.
- Components of an MDP:
  - initial state distribution $p(s_0)$
  - transition distribution $p(s_{t+1} \mid s_t, a_t)$
  - reward function $r(s_t, a_t)$
- policy $\pi_\theta(a_t \mid s_t)$ parameterized by $\theta$
- Assume a fully observable environment, i.e. $s_t$ can be observed directly
Finite and Infinite Horizon

- **Last time:** finite horizon MDPs
  - Fixed number of steps $T$ per episode
  - Maximize expected return $R = E_{p(\tau)}[r(\tau)]$
- **Now:** more convenient to assume infinite horizon
  - We can’t sum infinitely many rewards, so we need to discount them: $100$ a year from now is worth less than $100$ today
  - Discounted return
    $$ G_t = r_t + \gamma r_{t+1} + \gamma^2 r_{t+2} + \cdots $$
  - Want to choose an action to maximize expected discounted return
  - The parameter $\gamma < 1$ is called the discount factor
    - small $\gamma = \text{myopic}$
    - large $\gamma = \text{farsighted}$
Value Function

- **Value function** $V^\pi(s)$ of a state $s$ under policy $\pi$: the expected discounted return if we start in $s$ and follow $\pi$

\[
V^\pi(s) = \mathbb{E}[G_t \mid s_t = s]
\]

\[
= \mathbb{E} \left[ \sum_{i=0}^{\infty} \gamma^i r_{t+i} \mid s_t = s \right]
\]

- Computing the value function is generally impractical, but we can try to approximate (learn) it.
- The benefit is credit assignment: see directly how an action affects future returns rather than wait for rollouts.
• Rewards: -1 per time step
• Undiscounted ($\gamma = 1$)
• Actions: N, E, S, W
• State: current location
Value Function

![Value Function Diagram]

- Two grids are shown, each with arrows indicating possible moves.
- The grids have values associated with each cell, ranging from -23 to 16.
- The goal is to navigate from the start to the goal with the highest value.

---

**Value Function**

Jimmy Ba and Bo Wang  
CSC413/2516 Lecture 11: Q-Learning & the Game of Go  
8 / 38
The value function has a recursive formula

\[ V^\pi(s) = \mathbb{E}_{a_t, a_{t+i}, s_{t+i}} \left[ \sum_{i=0}^{\infty} \gamma^i r_{t+i} | s_t = s \right] \]

\[ = \mathbb{E}_{a_t} [r_t | s_t = s] \cdot \gamma \mathbb{E}_{s_{t+1}} [V^\pi(s_{t+1}) | s_t = s] \]

\[ = \sum_{a, r} P^\pi(a | s_t) p(r | a, s_t) \cdot r + \gamma \sum_{a, s'} P^\pi(a | s_t) p(s' | a, s_t) \cdot V^\pi(s') \]
Action-Value Function

Can we use a value function to choose actions?

\[
\arg \max_a r(s_t, a) + \gamma \mathbb{E}_{p(s_{t+1} | s_t, a_t)}[V^\pi(s_{t+1})]
\]
Action-Value Function

- Can we use a value function to choose actions?

\[
\arg \max_a r(s_t, a) + \gamma \mathbb{E}_{p(s_{t+1} \mid s_t, a_t)}[V^\pi(s_{t+1})]
\]

- Problem: this requires taking the expectation with respect to the environment’s dynamics, which we don’t have direct access to!

- Instead learn an action-value function, or Q-function: expected returns if you take action \(a\) and then follow your policy

\[
Q^\pi(s, a) = \mathbb{E}[G_t \mid s_t = s, a_t = a]
\]

- Relationship:

\[
V^\pi(s) = \sum_a \pi(a \mid s) Q^\pi(s, a)
\]

- Optimal action:

\[
\arg \max_a Q^\pi(s, a)
\]
The **Bellman Equation** is a recursive formula for the action-value function:

\[ Q^\pi(s, a) = r(s, a) + \gamma \mathbb{E}_{p(s' | s, a)} \pi(a' | s') [Q^\pi(s', a')] \]

- There are various Bellman equations, and most RL algorithms are based on repeatedly applying one of them.
The optimal policy $\pi^*$ is the one that maximizes the expected discounted return, and the optimal action-value function $Q^*$ is the action-value function for $\pi^*$.

The Optimal Bellman Equation gives a recursive formula for $Q^*$:

$$Q^*(s, a) = r(s, a) + \gamma \mathbb{E}_{p(s' | s, a)} \left[ \max_{a'} Q^*(s_{t+1}, a') \mid s_t = s, a_t = a \right]$$

This system of equations characterizes the optimal action-value function. So maybe we can approximate $Q^*$ by trying to solve the optimal Bellman equation!
Q-Learning

- Let $Q$ be an action-value function which hopefully approximates $Q^*$.
- The **Bellman error** is the update to our expected return when we observe the next state $s'$.

$$
\begin{align*}
    r(s_t, a_t) + \gamma \max_a Q(s_{t+1}, a) - Q(s_t, a_t)
\end{align*}
$$

inside $\mathbb{E}$ in RHS of Bellman eqn

- The Bellman equation says the Bellman error is 0 at convergence.
- **Q-learning** is an algorithm that repeatedly adjusts $Q$ to minimize the Bellman error.
- Each time we sample consecutive states and actions $(s_t, a_t, s_{t+1})$:

$$
Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha \left[ r(s_t, a_t) + \gamma \max_a Q(s_{t+1}, a) - Q(s_t, a_t) \right]
$$

Bellman error

Jimmy Ba and Bo Wang
CSC413/2516 Lecture 11: Q-Learning & the Game of Go
13 / 38
Exploration-Exploitation Tradeoff

- Notice: Q-learning only learns about the states and actions it visits.

- **Exploration-exploitation tradeoff**: the agent should sometimes pick suboptimal actions in order to visit new states and actions.

- **Simple solution**: $\epsilon$-greedy policy
  - With probability $1 - \epsilon$, choose the optimal action according to $Q$
  - With probability $\epsilon$, choose a random action

- Believe it or not, $\epsilon$-greedy is still used today!
Initialize $Q(s, a), \forall s \in S, a \in A(s)$, arbitrarily, and $Q(terminal-state, \cdot) = 0$

Repeat (for each episode):
  Initialize $S$
  Repeat (for each step of episode):
    Choose $A$ from $S$ using policy derived from $Q$ (e.g., $\varepsilon$-greedy)
    Take action $A$, observe $R, S'$
    \[ Q(S, A) \leftarrow Q(S, A) + \alpha \left[ R + \gamma \max_a Q(S', a) - Q(S, A) \right] \]
    $S \leftarrow S'$;
  until $S$ is terminal
Function Approximation

- So far, we’ve been assuming a tabular representation of $Q$: one entry for every state/action pair.
- This is impractical to store for all but the simplest problems, and doesn’t share structure between related states.
- Solution: approximate $Q$ using a parameterized function, e.g.
  - linear function approximation: $Q(s, a) = w^T \psi(s, a)$
  - compute $Q$ with a neural net
- Update $Q$ using backprop:

$$
t \leftarrow r(s_t, a_t) + \gamma \max_a Q(s_{t+1}, a)
$$

$$
\theta \leftarrow \theta + \alpha(t - Q(s, a)) \frac{\partial Q}{\partial \theta}
$$
Approximating $Q$ with a neural net is a decades-old idea, but DeepMind got it to work really well on Atari games in 2013 ("deep Q-learning")

They used a very small network by today’s standards.

Main technical innovation: store experience into a replay buffer, and perform Q-learning using stored experience.

- Gains sample efficiency by separating environment interaction from optimization — don’t need new experience for every SGD update!
Atari

• Mnih et al., *Nature* 2015. Human-level control through deep reinforcement learning
• Network was given raw pixels as observations
• Same architecture shared between all games
• Assume fully observable environment, even though that’s not the case
• After about a day of training on a particular game, often beat “human-level” performance (number of points within 5 minutes of play)
  • Did very well on reactive games, poorly on ones that require planning (e.g. Montezuma’s Revenge)
• https://www.youtube.com/watch?v=V1eYniJ0Rnk
• https://www.youtube.com/watch?v=4MlZncshy1Q
Wireheading

- If rats have a lever that causes an electrode to stimulate certain “reward centers” in their brain, they’ll keep pressing the lever at the expense of sleep, food, etc.
- RL algorithms show this “wireheading” behavior if the reward function isn’t designed carefully
- https://blog.openai.com/faulty-reward-functions/
Policy Gradient vs. Q-Learning

- Policy gradient and Q-learning use two very different choices of representation: policies and value functions.
- Advantage of both methods: don’t need to model the environment.
- Pros/cons of policy gradient:
  - Pro: unbiased estimate of gradient of expected return.
  - Pro: can handle a large space of actions (since you only need to sample one).
  - Con: high variance updates (implies poor sample efficiency).
  - Con: doesn’t do credit assignment.
- Pros/cons of Q-learning:
  - Pro: lower variance updates, more sample efficient.
  - Pro: does credit assignment.
  - Con: biased updates since Q function is approximate (drinks its own Kool-Aid).
  - Con: hard to handle many actions (since you need to take the max).
AlphaGo

- Most of the problem domains we’ve discussed so far were natural application areas for deep learning (e.g. vision, language)
  - We know they can be done on a neural architecture (i.e. the human brain)
  - The predictions are inherently ambiguous, so we need to find statistical structure
- Board games are a classic AI domain which relied heavily on sophisticated search techniques with a little bit of machine learning
  - Full observations, deterministic environment — why would we need uncertainty?
- The second part of the lecture is about AlphaGo, DeepMind’s Go playing system which took the world by storm in 2016 by defeating the human Go champion Lee Sedol
- Combines ideas from our last two lectures (policy gradient and value function learning)
AlphaGo

Some milestones in computer game playing:

- 1949 — Claude Shannon proposes the idea of game tree search, explaining how games could be solved algorithmically in principle
- 1951 — Alan Turing writes a chess program that he executes by hand
- 1956 — Arthur Samuel writes a program that plays checkers better than he does
- 1968 — An algorithm defeats human novices at Go

...silence...

- 1992 — TD-Gammon plays backgammon competitively with the best human players
- 1996 — Chinook wins the US National Checkers Championship
- 1997 — DeepBlue defeats world chess champion Garry Kasparov

After chess, Go was humanity’s last stand
Go

- Played on a $19 \times 19$ board
- Two players, black and white, each place one stone per turn
- Capture opponent’s stones by surrounding them
What makes Go so challenging:

- Hundreds of legal moves from any position, many of which are plausible
- Games can last hundreds of moves
- Unlike Chess, endgames are too complicated to solve exactly (endgames had been a major strength of computer players for games like Chess)
- Heavily dependent on pattern recognition
Game Trees

- Each node corresponds to a legal state of the game.
- The children of a node correspond to possible actions taken by a player.
- Leaf nodes are ones where we can compute the value since a win/draw condition was met

https://www.cs.cmu.edu/~adamchik/15-121/lectures/Game%20Trees/Game%20Trees.html
Game Trees

- To label the internal nodes, take the max over the children if it’s Player 1’s turn, min over the children if it’s Player 2’s turn
Game Trees

- As Claude Shannon pointed out in 1949, for games with finite numbers of states, you can solve them in principle by drawing out the whole game tree.
- Ways to deal with the exponential blowup
  - Search to some fixed depth, and then estimate the value using an evaluation function
  - Prioritize exploring the most promising actions for each player (according to the evaluation function)
- Having a good evaluation function is key to good performance
  - Traditionally, this was the main application of machine learning to game playing
  - For programs like Deep Blue, the evaluation function would be a learned linear function of carefully hand-designed features
Now for DeepMind’s computer Go player, AlphaGo...
Supervised Learning to Predict Expert Moves

- Can a computer play Go without any search?
Can a computer play Go without any search?

**Input:** a $19 \times 19$ ternary (black/white/empty) image — about half the size of MNIST!

**Prediction:** a distribution over all (legal) next moves

**Training data:** KGS Go Server, consisting of 160,000 games and 29 million board/next-move pairs

**Architecture:** fairly generic conv net

When playing for real, choose the highest-probability move rather than sampling from the distribution

This network, which just predicted expert moves, could beat a fairly strong program called GnuGo 97% of the time.

  * This was amazing — basically all strong game players had been based on some sort of search over the game tree
Self-Play and REINFORCE

- The problem from training with expert data: there are only 160,000 games in the database. What if we overfit?
- There is effectively infinite data from self-play
  - Have the network repeatedly play against itself as its opponent
  - For stability, it should also play against older versions of itself
- Start with the policy which samples from the predictive distribution over expert moves
  - The network which computes the policy is called the policy network
- REINFORCE algorithm: update the policy to maximize the expected reward $r$ at the end of the game (in this case, $r = +1$ for win, $-1$ for loss)
- If $\theta$ denotes the parameters of the policy network, $a_t$ is the action at time $t$, and $s_t$ is the state of the board, and $z$ the rollout of the rest of the game using the current policy

\[
R = \mathbb{E}_{a_t \sim p_{\theta}(a_t \mid s_t)} [\mathbb{E}[r(z) \mid s_t, a_t]]
\]
Self-Play and REINFORCE

- Gradient of the expected reward:

\[
\frac{\partial R}{\partial \theta} = \frac{\partial R}{\partial \theta} \mathbb{E}_{a_t \sim p_\theta(a_t | s_t)}[\mathbb{E}[r(z) | s_t, a_t]]
\]

\[
= \frac{\partial}{\partial \theta} \sum_{a_t} \sum_z p_\theta(a_t | s_t) p(z | s_t, a_t) R(z)
\]

\[
= \sum_{a_t} \sum_z p(z) R(z) \frac{\partial}{\partial \theta} p_\theta(a_t | s_t)
\]

\[
= \sum_{a_t} \sum_z p(z | s_t, a_t) R(z) p_\theta(a_t | s_t) \frac{\partial}{\partial \theta} \log p_\theta(a_t | s_t)
\]

\[
= \mathbb{E}_{p_{\theta}(a_t | s_t)} \left[ \mathbb{E}_{p(z | s_t, a_t)} \left[ R(z) \frac{\partial}{\partial \theta} \log p_\theta(a_t | s_t) \right] \right]
\]

- English translation: sample the action from the policy, then sample the rollout for the rest of the game.

  - If you win, update the parameters to make the action more likely. If you lose, update them to make it less likely.
Monte Carlo Tree Search

- In 2006, computer Go was revolutionized by a technique called Monte Carlo Tree Search.

Estimate the value of a position by simulating lots of rollouts, i.e. games played randomly using a quick-and-dirty policy.

Keep track of number of wins and losses for each node in the tree.

Key question: how to select which parts of the tree to evaluate?

Silver et al., 2016
Monte Carlo Tree Search

- The selection step determines which part of the game tree to spend computational resources on simulating.
- This is an instance of the exploration-exploitation
  - Want to focus on good actions for the current player
  - But want to explore parts of the tree we’re still uncertain about
- **Uniform Confidence Bound (UCB)** is a common heuristic; choose the node which has the largest frequentist upper confidence bound on its value:

\[
\mu_i + \sqrt{\frac{2 \log N}{N_i}}
\]

- \(\mu_i\) = fraction of wins for action \(i\), \(N_i\) = number of times we’ve tried action \(i\), \(N\) = total times we’ve visited this node
We just saw the policy network. But AlphaGo also has another network called a value network.

This network tries to predict, for a given position, which player has the advantage.

This is just a vanilla conv net trained with least-squares regression.

Data comes from the board positions and outcomes encountered during self-play.
Policy and Value Networks

- AlphaGo combined the policy and value networks with Monte Carlo Tree Search
- Policy network used to simulate rollouts
- Value network used to evaluate leaf positions
AlphaGo Timeline

- **Summer 2014** — start of the project (internship project for UofT grad student Chris Maddison)
- **October 2015** — AlphaGo defeats European champion
  - First time a computer Go player defeated a human professional without handicap — previously believed to be a decade away
- **January 2016** — publication of Nature article “Mastering the game of Go with deep neural networks and tree search”
- **March 2016** — AlphaGo defeats gradmaster Lee Sedol
- **October 2017** — AlphaGo Zero far surpasses the original AlphaGo without training on any human data
- **December 2017** — it beats the best chess programs too, for good measure
Most of the Go world expected AlphaGo to lose 5-0 (even after it had beaten the European champion)

It won the match 4-1

Some of its moves seemed bizarre to human experts, but turned out to be really good

Its one loss occurred when Lee Sedol played a move unlike anything in the training data
Further reading:


- Talk by the DeepMind CEO: [https://www.youtube.com/watch?v=aiwQsa_7ZIQ&list=PLqYmG7hTraZCGIymT8wVVIXLWkKPNBoFN&index=8](https://www.youtube.com/watch?v=aiwQsa_7ZIQ&list=PLqYmG7hTraZCGIymT8wVVIXLWkKPNBoFN&index=8)