07-280 Notes
Pre-reading: Deep Reinforcement Learning

Carnegie Mellon University
School of Computer Science

Contents

1 From Tables to Neural Networks **2**

2 Deep Q-Networks (DQN) **2**

3 AlphaGo and AlphaZero **3**

3.1 Searching Ahead: Monte Carlo Tree Search 3

3.2 AlphaGo 3

3.3 AlphaGo Zero and AlphaZero 4

4 RL for Language Models **4**

4.1 RLHF and InstructGPT: Rewards from Human Preferences 4

4.2 RL for Reasoning: DeepSeek-R1 [Optional] 5

5 Summary **6**

6 References **6**

1 From Tables to Neural Networks

In the RL prereading, Prince's house had three rooms, so we could store a value for every state in a table. Real problems are much bigger: Go has around 10^{170} board positions, and a language model chooses each word from a vocabulary of tens of thousands of tokens. No table could ever hold all of that.

Deep reinforcement learning keeps the same RL ideas but replaces the tables with **neural networks**:

- a **Q-network** $Q_\theta(s, a)$ instead of a Q-table, or
- a **policy network** $\pi_\theta(a \mid s)$ instead of a table of action probabilities.

A network can *generalize*: positions it has never seen still get sensible values if they look like positions it has seen. This prereading walks through four milestones, from Atari games to the models behind today's chatbots.

2 Deep Q-Networks (DQN)

Recall the Q-learning update from the RL prereading:

$$Q(s, a) \leftarrow Q(s, a) + \alpha \left[r + \gamma \max_{a'} Q(s', a') - Q(s, a) \right]$$

We saw that this is a gradient descent step on the squared error between the prediction $Q(s, a)$ and the target $r + \gamma \max_{a'} Q(s', a')$. A **Deep Q-Network** (DQN) does exactly that, but the prediction comes from a neural network with parameters θ:

$$\ell(\theta) = \frac{1}{2} \Big(\underbrace{r + \gamma \max_{a'} Q_{\bar{\theta}}(s', a')}_{\text{target } y} - Q_\theta(s, a) \Big)^2$$

This is just the regression loss from earlier in the course, with the target y built from a reward and the network's own estimate of the future.

Example: Prince's Q-network currently predicts $Q_\theta(L, \texttt{Move}) = 0.3$. He moves from the Living Room to the Kitchen with reward 0, and the network estimates $\max_{a'} Q_{\bar{\theta}}(K, a') = 0.9$. With $\gamma = 0.8$:

$$y = 0 + 0.8(0.9) = 0.72, \qquad \ell = \tfrac{1}{2}(0.72 - 0.3)^2 \approx 0.088$$

The prediction is too low, so gradient descent adjusts θ to raise $Q_\theta(L, \texttt{Move})$ toward 0.72.

Training a network on its own predictions is unstable, so DQN adds two tricks:

1. **Experience replay**: store past samples (s, a, r, s') in a big **replay buffer** and train on random minibatches from it. Consecutive samples from one game are very similar; shuffling them makes the data look more like the independent samples supervised learning expects, and each sample gets reused many times.
2. **Target network**: compute the target y with a separate, slowly updated copy of the network, $Q_{\bar{\theta}}$. Otherwise every update would also move the target we're chasing, like trying to hit a target that moves every time you aim.

With these two tricks, a single DQN architecture learned to play dozens of Atari games directly from screen pixels, reaching human-level scores on many of them (Mnih et al., 2015).

Note: DQN's $\max_{a'}$ only works when there are a handful of actions to compare, like the buttons on an Atari controller. For huge action spaces, like every possible sentence, we'll use policy networks instead.

3 AlphaGo and AlphaZero

In 2016, DeepMind's **AlphaGo** beat Go world champion Lee Sedol 4 games to 1, a milestone many expected to be decades away. It combined a policy network, a value network, and tree search.

3.1 Searching Ahead: Monte Carlo Tree Search

Monte Carlo tree search (MCTS) thinks ahead by simulation. Starting from the current position, it repeatedly:

1. **Selects** a path down the tree of moves it has already explored, favoring moves that have looked good so far but also trying moves it has visited less (exploration vs. exploitation again!).
2. **Expands** the tree by adding one new position at the end of that path.
3. **Evaluates** (or *simulates*) the new position, originally by playing random moves to the end of the game.
4. **Backs up** (or *backpropagates*) the result, updating the visit count and average outcome of every move along the path. Each average is a Monte Carlo estimate, like Monte Carlo policy evaluation in the RL prereading.

After many simulations, MCTS plays the move from the current position that it visited most.

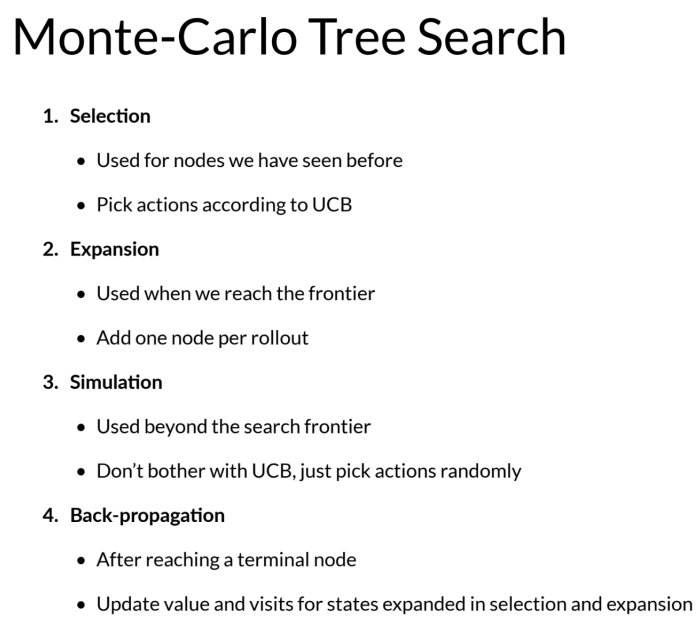


Figure 1: The four steps of MCTS (from the CMU 10-703/403 slides).

3.2 AlphaGo

Go has too many moves to search blindly, so AlphaGo used neural networks to focus the search:

1. **Imitate experts.** Train a policy network $\pi_\theta(a \mid s)$ to predict the moves human experts made, using millions of positions from expert games. This is ordinary supervised classification.
2. **Improve with RL.** Have the policy play against older versions of itself, with reward $+1$ for a win and -1 for a loss at the end of the game, and update it with

   $$\Delta\theta \propto z \, \nabla_\theta \log \pi_\theta(a_t \mid s_t)$$

   where z is the game outcome. That's REINFORCE from the RL prereading!
3. **Learn to judge positions.** Train a value network $V_\phi(s)$ to predict who will win from a position, using self-play games. This is regression.
4. **Search with both.** During a game, MCTS uses the policy network to decide which moves are worth exploring (narrowing the search) and the value network to evaluate positions without playing to the end (shortening the search).

3.3 AlphaGo Zero and AlphaZero

AlphaGo Zero (2017) dropped the human games and learned only from **self-play**, starting from random moves, and still beat the original AlphaGo. The key idea: MCTS guided by the network picks better moves than the network alone, so the network can learn by imitating its own search.

One network with two outputs, a policy $\pi_\theta(a \mid s)$ and a value $V_\theta(s)$, is trained in a loop:

1. Play a game against itself, running MCTS (about 1600 simulations) before every move.
2. At each position, record the search's visit counts as an improved policy, and at the end, record who won, $z = \pm 1$.
3. Train the policy output to match the visit-count distribution (a classification loss) and the value output to predict z (a regression loss).

Example: in some position, the network's policy over three legal moves is $(0.4, 0.4, 0.2)$. After 10 MCTS simulations, the moves were visited $(6, 3, 1)$ times, giving the improved policy $(0.6, 0.3, 0.1)$. The search found that move 1 is better than the network thought, so training pushes the network's output from $(0.4, 0.4, 0.2)$ toward $(0.6, 0.3, 0.1)$. Next game, the network starts out stronger, so the search built on top of it is stronger too.

AlphaZero applied the same algorithm, unchanged, to chess and shogi as well as Go, and beat the strongest existing programs in all three.

4 RL for Language Models

A large language model (LLM) writes a response one **token** (word piece) at a time, sampling each from a probability distribution. That's a policy π_θ: the state is the prompt plus the text so far, and the action is the next token. If we can score a finished response with a reward, we can improve the model with policy gradients. The big question is where the reward comes from.

4.1 RLHF and InstructGPT: Rewards from Human Preferences

A model trained only to predict internet text isn't automatically helpful. **Reinforcement learning from human feedback** (RLHF), used for **InstructGPT** (Ouyang et al., 2022), fixes this in three steps:

1. **Supervised fine-tuning (SFT).** Train the model to imitate good responses written by people.
2. **Train a reward model.** People rank several model responses to the same prompt, and a separate **reward model** learns to score preferred responses higher. (It's easier for people to compare two answers than to give each a number.)
3. **Optimize with RL.** Fine-tune the model to maximize the reward model's score using PPO, an improved policy gradient method. A penalty keeps it close to the SFT model, so it can't find strange responses that fool the reward model (**reward hacking**).

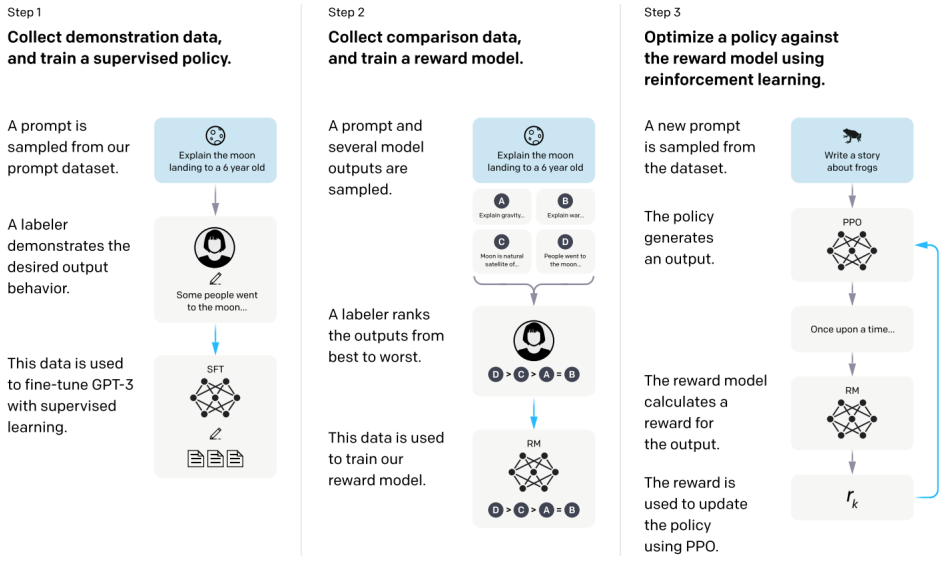


Figure 2: The three steps of RLHF (from the InstructGPT paper).

The result: people preferred responses from a 1.3-billion-parameter InstructGPT model over those from GPT-3, which has 175 billion parameters, a model more than 100 times larger.

4.2 RL for Reasoning: DeepSeek-R1 [Optional]

For a math problem with a known answer, we don't need people at all: the reward is 1 if the final answer is correct and 0 otherwise. This **verifiable reward** can't be fooled the way a learned reward model can.

DeepSeek-R1 (2025) was trained with **group relative policy optimization** (GRPO), a policy gradient method that uses the baseline idea from the RL prereading. For each question, the model samples a group of answers, and each answer's advantage is its reward compared with the group's average:

$$A_i = \frac{r_i - \text{mean}(r_1, \ldots, r_G)}{\text{std}(r_1, \ldots, r_G)}$$

Example: four answers with rewards $(1, 0, 0, 1)$ have mean 0.5 and standard deviation 0.5, so the advantages are $(+1, -1, -1, +1)$: correct answers become more likely, wrong ones less likely.

Trained this way with no human-written solutions, **DeepSeek-R1-Zero** (a nod to AlphaGo Zero) improved from about 16% to 71% on competition math problems (AIME 2024). Nobody told it *how* to reason, yet it learned on its own to write longer solutions, double-check its work, and backtrack from mistakes, because that earned more reward.

5 Summary

Method	Learns	Reward from	Builds on
DQN	$Q_\theta(s, a)$	game score	Q-learning
AlphaGo Zero	π_θ and V_θ	win or loss	self-play + search
RLHF (InstructGPT)	π_θ	human rankings (via reward model)	policy gradients
GRPO (DeepSeek-R1)	π_θ	answer checker	policy gradients + baseline

All of these reuse the ideas from the RL prereading. DQN is Q-learning with a neural network. AlphaGo's RL step and RLHF's PPO step are policy gradients, and GRPO's group average is the baseline trick. The biggest change from tables to today's models is the scale of the network, and the source of the reward.

6 References

- Kocsis & Szepesvári (2006). Bandit based Monte-Carlo planning. *ECML*.
- Mnih et al. (2015). Human-level control through deep reinforcement learning. *Nature*.
- Silver et al. (2016). Mastering the game of Go with deep neural networks and tree search. *Nature*.
- Silver et al. (2017). Mastering the game of Go without human knowledge. *Nature*.
- Schulman et al. (2017). Proximal policy optimization algorithms. *arXiv*.
- Silver et al. (2018). A general reinforcement learning algorithm that masters chess, shogi, and Go through self-play. *Science*.
- Ouyang et al. (2022). Training language models to follow instructions with human feedback. *NeurIPS*.
- Shao et al. (2024). DeepSeekMath: Pushing the limits of mathematical reasoning in open language models. *arXiv*.
- DeepSeek-AI (2025). DeepSeek-R1: Incentivizing reasoning capability in LLMs via reinforcement learning. *arXiv*.

Parts of these notes are adapted from CMU 10-703/10-403 (Deep Reinforcement Learning) and 10-301/601 (Introduction to Machine Learning) lecture materials.