CSCI567 Machine Learning (Fall 2025)

Haipeng Luo

University of Southern California

Nov 21, 2025

Exam 2 Logistics

Date: Friday, Dec 5th

Time: 2:00-4:00pm (plus another 20 mins for uploading)

Location: THH 201 (Initial A-R) and SGM 101 (Initial S-Z)

Individual effort, close-book (no cheat sheet), no calculators or any other electronics, *but need your phone to upload your solutions to Gradescope from 4:00-4:20pm*

Exam 2 Coverage

Coverage: mostly Lec 7-11 (just see the sample)

Exam 2 Coverage

Coverage: mostly Lec 7-11 (just see the sample)

Seven problems in total

- one problem of 15 multiple-choice multiple-answer questions
  - *please note the new instructions!!*
- six other homework-like problems, each has a couple sub-problems
  - clustering, EM, HMM, RNN/transformer, bandits, RL

Outline

1. Review of last lecture
2. Basics of Reinforcement learning
3. Deep Q-Networks and Atari Games
4. Policy Gradient, Actor-Critic, and AlphaGo

Outline

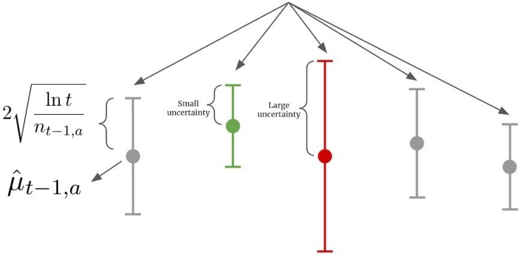
UCB for multi-armed bandits

Adaptive exploration-exploitation trade-off via optimism

Upper Confidence Bound (UCB) algorithm

For $t = 1, \ldots, T$, pick $a_t = \mathrm{argmax}_a \ \mathrm{UCB}_{t,a}$ where

$$\mathrm{UCB}_{t,a} \triangleq \hat{\mu}_{t-1,a} + 2\sqrt{\frac{\ln t}{n_{t-1,a}}}$$



Self-play for dueling bandits (preference feedback)

Exp3 for dueling bandits (selecting b_t)

Input: a learning rate parameter $\eta > 0$

For $t = 1, \ldots, T$,

- compute arm distribution $\boldsymbol{q}_t = \text{softmax}\left(-\eta \sum_{\tau=1}^{t-1} \boldsymbol{\ell}_\tau\right)$
- sample b_t from $\boldsymbol{q}_t$
- observe loss feedback $\mathbb{I}[a_t \succ b_t]$ (a_t selected by opponent)
- construct estimator $\boldsymbol{\ell}_t \in \mathbb{R}_+^K$ where for each b: $\boldsymbol{\ell}_{t,b} = \frac{\mathbb{I}[b_t = b]\mathbb{I}[a_t \succ b]}{q_{t,b}}$

Losses versus rewards

Exp3 for dueling bandits (CORRECT way to select a_t)

For $t = 1, \ldots, T$,

- sample a_t from arm distribution $\boldsymbol{p}_t = \text{softmax}\left(-\eta \sum_{\tau=1}^{t-1} \boldsymbol{\ell}_\tau\right)$
- observe reward feedback $\mathbb{I}[a_t \succ b_t]$ (b_t selected by opponent)
- construct estimator $\boldsymbol{\ell}_t \in \mathbb{R}_+^K$ where for each a: $\boldsymbol{\ell}_{t,a} = \frac{\mathbb{I}[a_t = a]\mathbb{I}[a \prec b_t]}{p_{t,a}}$

- from $\text{softmax}\left(\eta \sum_{\tau=1}^{t-1} \boldsymbol{r}_\tau\right)$ to $\text{softmax}\left(-\eta \sum_{\tau=1}^{t-1} \boldsymbol{\ell}_\tau\right)$
- from $\boldsymbol{r}_{t,a} = \frac{\mathbb{I}[a_t = a]\mathbb{I}[a \succ b_t]}{p_{t,a}}$ to $\boldsymbol{\ell}_{t,a} = \frac{\mathbb{I}[a_t = a]\mathbb{I}[a \prec b_t]}{p_{t,a}}$

How to find Nash Equilibra of a zero-sum game?

Even for games *as large as poker*, **can approximately find one via self-play and regret minimization!**

Self-play for zero-sum games

Input: multi-armed bandit algorithms $\mathcal{A}$ and $\mathcal{B}$

For $t = 1, \ldots, T$,

- get arm distributions $\boldsymbol{p}_t$ and $\boldsymbol{q}_t$ from $\mathcal{A}$ and $\mathcal{B}$ respectively
- sample a_t from $\boldsymbol{p}_t$ and b_t from $\boldsymbol{q}_t$
- observe M_{a_t,b_t} (plus noise), feed it as reward to $\mathcal{A}$ and as loss to $\mathcal{B}$

Low regret $\Rightarrow$ convergence to NE

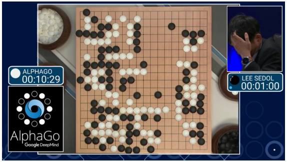
Outline

Recent Successes of Deep Reinforcement Learning (RL)

Atari (2013)

Recent Successes of Deep Reinforcement Learning (RL)

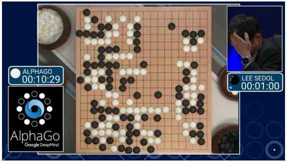
Atari (2013)



Go (2015)

Recent Successes of Deep Reinforcement Learning (RL)

Atari (2013)

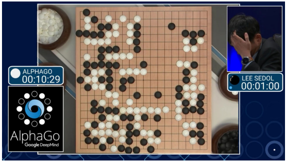


Go (2015)

Dota 2 (2017)

Recent Successes of Deep Reinforcement Learning (RL)

Atari (2013)



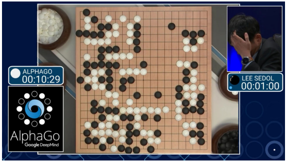
Go (2015)

Dota 2 (2017)

StarCraft (2019)

Recent Successes of Deep Reinforcement Learning (RL)

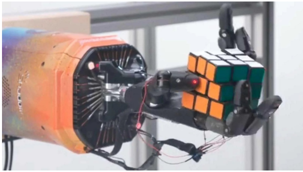
Atari (2013)



Go (2015)

Dota 2 (2017)

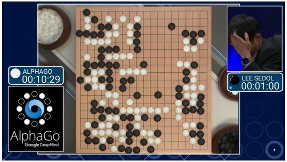
StarCraft (2019)



Rubik's Cube (2019)

Recent Successes of Deep Reinforcement Learning (RL)

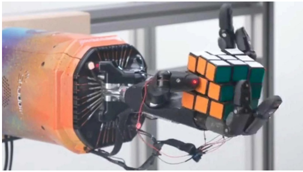
Atari (2013)



Go (2015)

Dota 2 (2017)

StarCraft (2019)

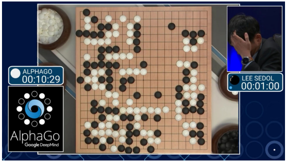


Rubik's Cube (2019)

ChatGPT (2022)

Recent Successes of Deep Reinforcement Learning (RL)

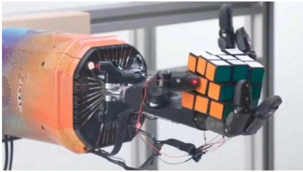
Atari (2013)



Go (2015)

Dota 2 (2017)

StarCraft (2019)



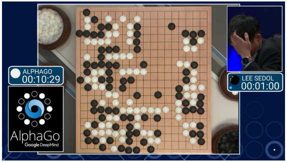
Rubik's Cube (2019)

ChatGPT (2022)

Deep RL = RL + deep neural net models,

Recent Successes of Deep Reinforcement Learning (RL)

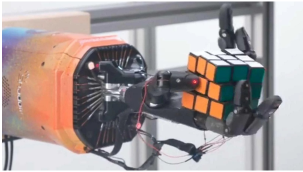
Atari (2013)



Go (2015)

Dota 2 (2017)

StarCraft (2019)



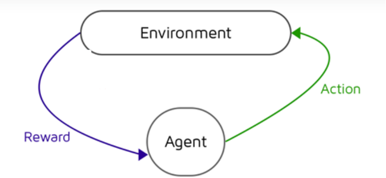
Rubik's Cube (2019)

ChatGPT (2022)

Deep RL = RL + deep neural net models, *so what really is RL?*

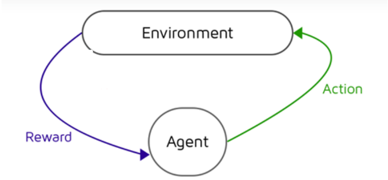
Motivation

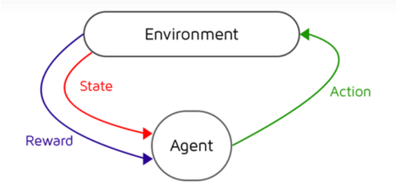
Multi-armed bandit is among the simplest decision making problems with limited feedback.



Motivation

Multi-armed bandit is among the simplest decision making problems with limited feedback.

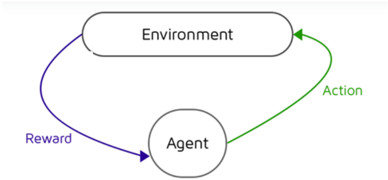


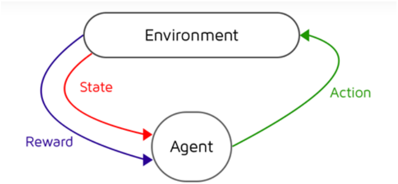


It's often too simple to capture many real-life problems. One thing it fails to capture is the "state" of the learning agent, which has impacts on the reward of each action.

Motivation

Multi-armed bandit is among the simplest decision making problems with limited feedback.





It's often too simple to capture many real-life problems. One thing it fails to capture is the "state" of the learning agent, which has impacts on the reward of each action.

- e.g. for Atari games, after making one move, the agent moves to a different state, with possible different rewards for each action

Reinforcement learning

Reinforcement learning (RL) is one way to deal with this issue.

Reinforcement learning

Reinforcement learning (RL) is one way to deal with this issue.

The foundation of RL is **Markov Decision Process (MDP)**,
a combination of Markov model (Lec 8) and multi-armed bandit (Lec 10)

Markov Decision Processes (MDPs)

An MDP is parameterized by five elements

Markov Decision Processes (MDPs)

An MDP is parameterized by five elements

- $\mathcal{S}$: a set of possible states

Markov Decision Processes (MDPs)

An MDP is parameterized by five elements

- $\mathcal{S}$: a set of possible states
- $\mathcal{A}$: a set of possible actions

Markov Decision Processes (MDPs)

An MDP is parameterized by five elements

- $\mathcal{S}$: a set of possible states
- $\mathcal{A}$: a set of possible actions
- P: transition probability, $P(s'|s,a)$ is the probability of transiting from state s to state s' after taking action a (Markov property)

Markov Decision Processes (MDPs)

An MDP is parameterized by five elements

- $\mathcal{S}$: a set of possible states
- $\mathcal{A}$: a set of possible actions
- P: transition probability, $P(s'|s,a)$ is the probability of transiting from state s to state s' after taking action a (Markov property)
- r: reward function, $r(s,a)$ is (expected) reward of action a at state s

Markov Decision Processes (MDPs)

An MDP is parameterized by five elements

- $\mathcal{S}$: a set of possible states
- $\mathcal{A}$: a set of possible actions
- P: transition probability, $P(s'|s,a)$ is the probability of transiting from state s to state s' after taking action a (Markov property)
- r: reward function, $r(s,a)$ is (expected) reward of action a at state s
- $\gamma \in (0,1]$: discount factor, informally, 1 dollar tomorrow is only worth γ when viewed from today (inflation)

Markov Decision Processes (MDPs)

An MDP is parameterized by five elements

- $\mathcal{S}$: a set of possible states
- $\mathcal{A}$: a set of possible actions
- P: transition probability, $P(s'|s,a)$ is the probability of transiting from state s to state s' after taking action a (Markov property)
- r: reward function, $r(s,a)$ is (expected) reward of action a at state s
- $\gamma \in (0,1]$: discount factor, informally, 1 dollar tomorrow is only worth γ when viewed from today (inflation)

Different from simple Markov chains, the state transition is influenced by the taken action.

Markov Decision Processes (MDPs)

An MDP is parameterized by five elements

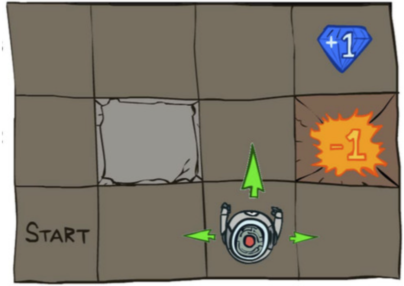
- $\mathcal{S}$: a set of possible states
- $\mathcal{A}$: a set of possible actions
- P: transition probability, $P(s'|s,a)$ is the probability of transiting from state s to state s' after taking action a (Markov property)
- r: reward function, $r(s,a)$ is (expected) reward of action a at state s
- $\gamma \in (0,1]$: discount factor, informally, 1 dollar tomorrow is only worth γ when viewed from today (inflation)

Different from simple Markov chains, the state transition is influenced by the taken action.

Different from Multi-armed bandit, the reward depends on the state.

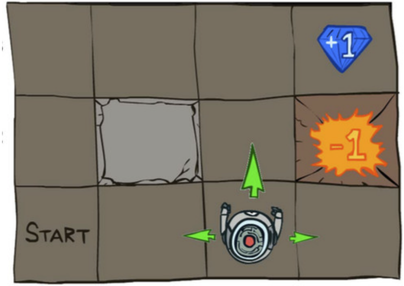
Example

Canonical example: a grid world



Example

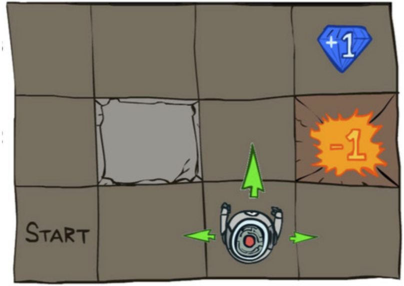
Canonical example: a grid world



- each grid is a state

Example

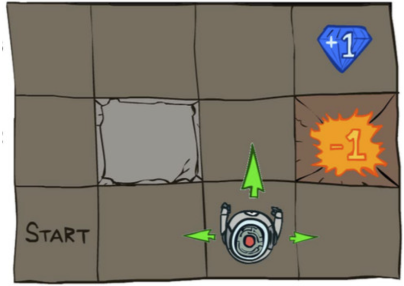
Canonical example: a grid world

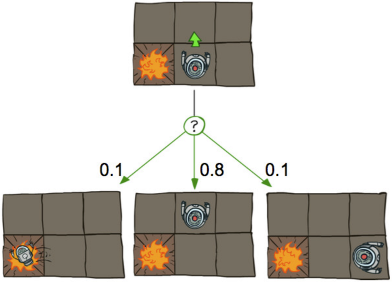


- each grid is a state
- 4 actions: up, down, left, right

Example

Canonical example: a grid world



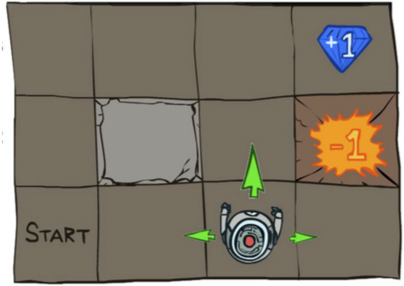


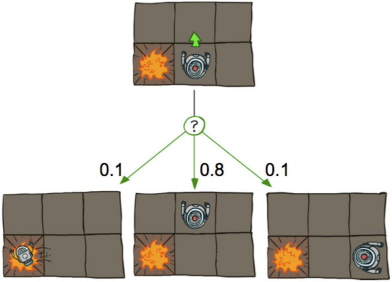
transition model P

- each grid is a state
- 4 actions: up, down, left, right

Example

Canonical example: a grid world





transition model P

- each grid is a state
- 4 actions: up, down, left, right
- reward is 1 for diamond, -1 for fire, and 0 everywhere else

Policy

A **policy** π specifies the probability of taking action a at state s as $\pi(a|s)$.

Policy

A **policy** π specifies the probability of taking action a at state s as $\pi(a|s)$.

If we start from state $s_1 \in \mathcal{S}$ and act according to a policy π, the discounted rewards for time $1, 2, \ldots$ are respectively

Policy

A **policy** π specifies the probability of taking action a at state s as $\pi(a|s)$.

If we start from state $s_1 \in \mathcal{S}$ and act according to a policy π, the discounted rewards for time $1, 2, \ldots$ are respectively

$$r(s_1, a_1), \;\; \gamma r(s_2, a_2), \;\; \gamma^2 r(s_3, a_3), \;\; \cdots$$

where $a_t \sim \pi(\cdot|s_t)$ and $s_{t+1} \sim P(\cdot|s_t, a_t)$

Policy

A **policy** π specifies the probability of taking action a at state s as $\pi(a|s)$.

If we start from state $s_1 \in \mathcal{S}$ and act according to a policy π, the discounted rewards for time $1, 2, \ldots$ are respectively

$$r(s_1, a_1), \;\; \gamma r(s_2, a_2), \;\; \gamma^2 r(s_3, a_3), \;\; \cdots$$

where $a_t \sim \pi(\cdot|s_t)$ and $s_{t+1} \sim P(\cdot|s_t, a_t)$

If we follow the policy forever, the total (discounted) reward is

$$\mathbb{E}\left[\sum_{t=1}^{\infty} \gamma^{t-1} r(s_t, a_t)\right]$$

Optimal Policy and Bellman Equation

First goal: knowing all parameters, *how to find the optimal policy*

$$\underset{\pi}{\operatorname{argmax}} \ \mathbb{E}\left[\sum_{t=1}^{\infty} \gamma^{t-1} r(s_t, a_t)\right] \quad ?$$

Optimal Policy and Bellman Equation

First goal: knowing all parameters, *how to find the optimal policy*

$$\underset{\pi}{\operatorname{argmax}} \ \mathbb{E}\left[\sum_{t=1}^{\infty} \gamma^{t-1} r(s_t, a_t)\right] \quad ?$$

We first answer a related question: *what is the maximum reward one can achieve starting from an arbitrary state s?*

Optimal Policy and Bellman Equation

First goal: knowing all parameters, *how to find the optimal policy*

$$\operatorname*{argmax}_{\pi} \mathbb{E}\left[\sum_{t=1}^{\infty} \gamma^{t-1} r(s_t, a_t)\right] \quad ?$$

We first answer a related question: *what is the maximum reward one can achieve starting from an arbitrary state s?*

$$V(s) = \max_{\pi} \mathbb{E}\left[\sum_{t=1}^{\infty} \gamma^{t-1} r(s_t, a_t) \,\middle|\, s_1 = s\right]$$

Optimal Policy and Bellman Equation

First goal: knowing all parameters, *how to find the optimal policy*

$$\operatorname*{argmax}_{\pi} \mathbb{E}\left[\sum_{t=1}^{\infty} \gamma^{t-1} r(s_t, a_t)\right] \quad ?$$

We first answer a related question: *what is the maximum reward one can achieve starting from an arbitrary state s?*

$$\begin{aligned} V(s) &= \max_{\pi} \mathbb{E}\left[\sum_{t=1}^{\infty} \gamma^{t-1} r(s_t, a_t) \,\middle|\, s_1 = s\right] \\ &= \max_{a \in \mathcal{A}} \left(r(s,a) + \gamma \sum_{s' \in \mathcal{S}} P(s'|s,a) V(s') \right) \end{aligned}$$

Optimal Policy and Bellman Equation

First goal: knowing all parameters, *how to find the optimal policy*

$$\operatorname*{argmax}_{\pi} \ \mathbb{E}\left[\sum_{t=1}^{\infty} \gamma^{t-1} r(s_t, a_t)\right] \quad ?$$

We first answer a related question: *what is the maximum reward one can achieve starting from an arbitrary state s?*

$$\begin{aligned} V(s) &= \max_{\pi} \mathbb{E}\left[\sum_{t=1}^{\infty} \gamma^{t-1} r(s_t, a_t) \;\middle|\; s_1 = s\right] \\ &= \max_{a \in \mathcal{A}} \left(r(s, a) + \gamma \sum_{s' \in \mathcal{S}} P(s'|s, a) V(s') \right) \end{aligned}$$

V is called the **optimal value function**.

Optimal Policy and Bellman Equation

First goal: knowing all parameters, *how to find the optimal policy*

$$\operatorname*{argmax}_{\pi} \; \mathbb{E}\left[\sum_{t=1}^{\infty} \gamma^{t-1} r(s_t, a_t)\right] \quad ?$$

We first answer a related question: *what is the maximum reward one can achieve starting from an arbitrary state s?*

$$\begin{aligned} V(s) &= \max_{\pi} \mathbb{E}\left[\sum_{t=1}^{\infty} \gamma^{t-1} r(s_t, a_t) \;\middle|\; s_1 = s\right] \\ &= \max_{a \in \mathcal{A}} \left(r(s,a) + \gamma \sum_{s' \in \mathcal{S}} P(s'|s,a) V(s') \right) \end{aligned}$$

V is called the **optimal value function**. It satisfies the above **Bellman equation**: $|\mathcal{S}|$ nonlinear equations with $|\mathcal{S}|$ unknowns, *how to solve it?*

Value Iteration

Value Iteration

Initialize $V_1(s) = 0$ for all $s \in \mathcal{S}$

Value Iteration

Value Iteration

Initialize $V_1(s) = 0$ for all $s \in \mathcal{S}$

For $k = 1, 2, \ldots$ (until convergence), perform **Bellman update**:

$$V_{k+1}(s) \leftarrow \max_{a \in \mathcal{A}} \left(r(s,a) + \gamma \sum_{s' \in \mathcal{S}} P(s'|s,a) V_k(s') \right), \ \ \forall s \in \mathcal{S}$$

Value Iteration

Value Iteration

Initialize $V_1(s) = 0$ for all $s \in \mathcal{S}$

For $k = 1, 2, \dots$ (until convergence), perform **Bellman update**:

$$V_{k+1}(s) \leftarrow \max_{a \in \mathcal{A}} \left(r(s,a) + \gamma \sum_{s' \in \mathcal{S}} P(s'|s,a) V_k(s') \right), \ \ \forall s \in \mathcal{S}$$

Value iteration converges *exponentially fast*! (HW4)

Value Iteration

Value Iteration

Initialize $V_1(s) = 0$ for all $s \in \mathcal{S}$

For $k = 1, 2, \ldots$ (until convergence), perform **Bellman update**:

$$V_{k+1}(s) \leftarrow \max_{a \in \mathcal{A}} \left(r(s,a) + \gamma \sum_{s' \in \mathcal{S}} P(s'|s,a) V_k(s') \right), \quad \forall s \in \mathcal{S}$$

Value iteration converges *exponentially fast*! (HW4)

Knowing V, the optimal policy π^* is simply

$$\pi^*(s) = \operatorname*{argmax}_{a \in \mathcal{A}} \left(r(s,a) + \gamma \sum_{s' \in \mathcal{S}} P(s'|s,a) V(s') \right)$$

Learning MDPs

Now suppose we do not know the parameters of the MDP

- transition probability P
- reward function r

How do we find the optimal policy?

Learning MDPs

Now suppose we do not know the parameters of the MDP

- transition probability P
- reward function r

How do we find the optimal policy?

- model-based approaches
- model-free approaches

Model-Based Approaches

Key idea: learn the model P and r explicitly from samples

Model-Based Approaches

Key idea: learn the model P and r explicitly from samples

Suppose we have a sequence of interactions: $s_1, a_1, r_1, \ldots, s_T, a_T, r_T$,

Model-Based Approaches

Key idea: learn the model P and r explicitly from samples

Suppose we have a sequence of interactions: $s_1, a_1, r_1, \ldots, s_T, a_T, r_T$, then the MLE for P and r are simply

$$P(s'|s,a) \propto \#\text{transitions from } s \text{ to } s' \text{ after taking action } a$$
$$r(s,a) = \text{average observed reward at state } s \text{ after taking action } a$$

Model-Based Approaches

Key idea: learn the model P and r explicitly from samples

Suppose we have a sequence of interactions: $s_1, a_1, r_1, \ldots, s_T, a_T, r_T$, then the MLE for P and r are simply

$$P(s'|s,a) \propto \#\text{transitions from } s \text{ to } s' \text{ after taking action } a$$
$$r(s,a) = \text{average observed reward at state } s \text{ after taking action } a$$

Having estimates of the parameters we can then apply value iteration to find the optimal policy.

Model-Based Approaches

How do we collect data $s_1, a_1, r_1, s_2, a_2, r_2, \ldots, s_T, a_T, r_T$?

Model-Based Approaches

How do we collect data $s_1, a_1, r_1, s_2, a_2, r_2, \ldots, s_T, a_T, r_T$?

Let's adopt the ϵ-Greedy idea again to ensure exploration.

Model-Based Approaches

How do we collect data $s_1, a_1, r_1, s_2, a_2, r_2, \ldots, s_T, a_T, r_T$?

Let's adopt the ϵ-Greedy idea again to ensure exploration.

A sketch for model-based approaches

Initialize V

Model-Based Approaches

How do we collect data $s_1, a_1, r_1, s_2, a_2, r_2, \ldots, s_T, a_T, r_T$?

Let's adopt the ϵ-Greedy idea again to ensure exploration.

A sketch for model-based approaches

Initialize V

For $t = 1, 2, \ldots,$

- **with probability ϵ, explore**: pick an action uniformly at random

Model-Based Approaches

How do we collect data $s_1, a_1, r_1, s_2, a_2, r_2, \ldots, s_T, a_T, r_T$?

Let's adopt the ϵ-Greedy idea again to ensure exploration.

A sketch for model-based approaches

Initialize V

For $t = 1, 2, \ldots,$

- **with probability** ϵ**, explore**: pick an action uniformly at random
- **with probability** $1 - \epsilon$**, exploit**: pick the optimal action based on V

Model-Based Approaches

How do we collect data $s_1, a_1, r_1, s_2, a_2, r_2, \ldots, s_T, a_T, r_T$?

Let's adopt the ϵ-Greedy idea again to ensure exploration.

A sketch for model-based approaches

Initialize V

For $t = 1, 2, \ldots,$

- **with probability** ϵ**, explore**: pick an action uniformly at random
- **with probability** $1 - \epsilon$**, exploit**: pick the optimal action based on V
- update the model parameters P, r

Model-Based Approaches

How do we collect data $s_1, a_1, r_1, s_2, a_2, r_2, \ldots, s_T, a_T, r_T$?

Let's adopt the ϵ-Greedy idea again to ensure exploration.

A sketch for model-based approaches

Initialize V

For $t = 1, 2, \ldots,$

- **with probability** ϵ**, explore**: pick an action uniformly at random
- **with probability** $1 - \epsilon$**, exploit**: pick the optimal action based on V
- update the model parameters P, r
- update the value function V (via value iteration)

Model-Free Approaches

Key idea: do not learn the model explicitly.

Model-Free Approaches

Key idea: do not learn the model explicitly. *What do we learn then?*

Model-Free Approaches

Key idea: do not learn the model explicitly. *What do we learn then?*

Define the $Q : \mathcal{S} \times \mathcal{A} \to \mathbb{R}$ function as

$$Q(s, a) = r(s, a) + \gamma \sum_{s' \in \mathcal{S}} P(s'|s, a) \max_{a' \in \mathcal{A}} Q(s', a')$$

In words, $Q(s, a)$ is the expected reward one can achieve starting from state s with action a, then acting optimally.

Model-Free Approaches

Key idea: do not learn the model explicitly. *What do we learn then?*

Define the $Q : \mathcal{S} \times \mathcal{A} \to \mathbb{R}$ function as

$$Q(s,a) = r(s,a) + \gamma \sum_{s' \in \mathcal{S}} P(s'|s,a) \max_{a' \in \mathcal{A}} Q(s',a')$$

In words, $Q(s,a)$ is the expected reward one can achieve starting from state s with action a, then acting optimally.

Clearly, $V(s) = \max_a Q(s,a)$.

Model-Free Approaches

Key idea: do not learn the model explicitly. *What do we learn then?*

Define the $Q : \mathcal{S} \times \mathcal{A} \to \mathbb{R}$ function as

$$Q(s, a) = r(s, a) + \gamma \sum_{s' \in \mathcal{S}} P(s'|s, a) \max_{a' \in \mathcal{A}} Q(s', a')$$

In words, $Q(s, a)$ is the expected reward one can achieve starting from state s with action a, then acting optimally.

Clearly, $V(s) = \max_a Q(s, a)$.

Knowing $Q(s, a)$, the optimal policy at state s is simply $\operatorname{argmax}_a Q(s, a)$.

Model-Free Approaches

Key idea: do not learn the model explicitly. *What do we learn then?*

Define the $Q : \mathcal{S} \times \mathcal{A} \to \mathbb{R}$ function as

$$Q(s,a) = r(s,a) + \gamma \sum_{s' \in \mathcal{S}} P(s'|s,a) \max_{a' \in \mathcal{A}} Q(s',a')$$

In words, $Q(s,a)$ is the expected reward one can achieve starting from state s with action a, then acting optimally.

Clearly, $V(s) = \max_a Q(s,a)$.

Knowing $Q(s,a)$, the optimal policy at state s is simply $\operatorname{argmax}_a Q(s,a)$.

Model-free approaches learn the Q function directly from samples.

Temporal Difference (TD error)

How to learn the Q function?

$$Q(s, a) = r(s, a) + \gamma \sum_{s' \in \mathcal{S}} P(s'|s, a) \max_{a' \in \mathcal{A}} Q(s', a')$$

Temporal Difference (TD error)

How to learn the Q function?

$$Q(s,a) = r(s,a) + \gamma \sum_{s' \in \mathcal{S}} P(s'|s,a) \max_{a' \in \mathcal{A}} Q(s',a')$$

Given experience (s_t, a_t, r_t, s_{t+1}), with the current guess on Q,
$y_t = r_t + \gamma \max_{a'} Q(s_{t+1}, a')$ is like a sample of the RHS of the equation.

Temporal Difference (TD error)

How to learn the Q function?

$$Q(s,a) = r(s,a) + \gamma \sum_{s' \in \mathcal{S}} P(s'|s,a) \max_{a' \in \mathcal{A}} Q(s',a')$$

Given experience (s_t, a_t, r_t, s_{t+1}), with the current guess on Q, $y_t = r_t + \gamma \max_{a'} Q(s_{t+1}, a')$ is like a sample of the RHS of the equation.

So it's natural to do the following update (with learning rate α):

$$Q(s_t, a_t) \leftarrow (1-\alpha) Q(s_t, a_t) + \alpha y_t$$

Temporal Difference (TD error)

How to learn the Q function?

$$Q(s,a) = r(s,a) + \gamma \sum_{s' \in \mathcal{S}} P(s'|s,a) \max_{a' \in \mathcal{A}} Q(s',a')$$

Given experience (s_t, a_t, r_t, s_{t+1}), with the current guess on Q, $y_t = r_t + \gamma \max_{a'} Q(s_{t+1}, a')$ is like a sample of the RHS of the equation.

So it's natural to do the following update (with learning rate α):

$$\begin{aligned} Q(s_t, a_t) &\leftarrow (1-\alpha)Q(s_t, a_t) + \alpha y_t \\ &= Q(s_t, a_t) + \alpha \underbrace{(y_t - Q(s_t, a_t))}_{\text{temporal difference}} \end{aligned}$$

Temporal Difference (TD error)

How to learn the Q function?

$$Q(s,a) = r(s,a) + \gamma \sum_{s' \in \mathcal{S}} P(s'|s,a) \max_{a' \in \mathcal{A}} Q(s',a')$$

Given experience (s_t, a_t, r_t, s_{t+1}), with the current guess on Q, $y_t = r_t + \gamma \max_{a'} Q(s_{t+1}, a')$ is like a sample of the RHS of the equation.

So it's natural to do the following update (with learning rate α):

$$\begin{aligned} Q(s_t,a_t) &\leftarrow (1-\alpha)Q(s_t,a_t) + \alpha y_t \\ &= Q(s_t,a_t) + \alpha \underbrace{(y_t - Q(s_t,a_t))}_{\text{temporal difference}} \\ &= Q(s_t,a_t) - \alpha \frac{\partial\left(\frac{1}{2}\left(Q(s_t,a_t) - y_t\right)^2\right)}{\partial Q(s_t,a_t)} \end{aligned}$$

which is gradient descent w.r.t. squared loss $\frac{1}{2}\left(Q(s_t,a_t) - y_t\right)^2$.

Q-learning

The simplest model-free algorithm:

Q-learning

Initialize Q

Q-learning

The simplest model-free algorithm:

Q-learning

Initialize Q

For $t = 1, 2, \ldots,$

- **with probability** ϵ**, explore**: a_t is chosen uniformly at random

Q-learning

The simplest model-free algorithm:

Q-learning

Initialize Q

For $t = 1, 2, \ldots,$

- **with probability** ϵ**, explore**: a_t is chosen uniformly at random
- **with probability** $1 - \epsilon$**, exploit**: $a_t = \operatorname{argmax}_a Q(s_t, a)$

Q-learning

The simplest model-free algorithm:

Q-learning

Initialize Q

For $t = 1, 2, \ldots,$

- **with probability** ϵ**, explore**: a_t is chosen uniformly at random
- **with probability** $1 - \epsilon$**, exploit**: $a_t = \operatorname{argmax}_a Q(s_t, a)$
- execute action a_t, receive reward r_t, arrive at state s_{t+1}

Q-learning

The simplest model-free algorithm:

Q-learning

Initialize Q

For $t = 1, 2, \ldots,$

- **with probability** ϵ**, explore**: a_t is chosen uniformly at random
- **with probability** $1 - \epsilon$**, exploit**: $a_t = \mathrm{argmax}_a\, Q(s_t, a)$
- execute action a_t, receive reward r_t, arrive at state s_{t+1}
- update the Q function

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

for some learning rate α.

Comparisons

	Model-based	Model-free
What it learns	model parameters $P, r, \ldots$	Q function

Comparisons

	Model-based	Model-free
What it learns	model parameters $P, r, \ldots$	Q function
Space	$O(\|\mathcal{S}\|^2\|\mathcal{A}\|)$	$O(\|\mathcal{S}\|\|\mathcal{A}\|)$

Comparisons

	Model-based	Model-free
What it learns	model parameters $P, r, \ldots$	Q function
Space	$O(\lvert\mathcal{S}\rvert^2\lvert\mathcal{A}\rvert)$	$O(\lvert\mathcal{S}\rvert\lvert\mathcal{A}\rvert)$
Sample efficiency	usually better	usually worse

Outline

Function approximation

Algorithms discussed so far (called **tabular algorithms**) run in time/space poly($|\mathcal{S}||\mathcal{A}|$), which is impractical.

Function approximation

Algorithms discussed so far (called **tabular algorithms**) run in time/space poly($|\mathcal{S}||\mathcal{A}|$), which is impractical. (Go has about 2×10^{170} states!)

Function approximation

Algorithms discussed so far (called **tabular algorithms**) run in time/space poly($|\mathcal{S}||\mathcal{A}|$), which is impractical. (Go has about 2×10^{170} states!)

To overcome this issue, we approximate Q by a function parametrized by θ:

$$Q_{\theta}(s, a) \approx Q(s, a), \; \forall \, (s, a)$$

Function approximation

Algorithms discussed so far (called **tabular algorithms**) run in time/space poly($|\mathcal{S}||\mathcal{A}|$), which is impractical. (Go has about 2×10^{170} states!)

To overcome this issue, we approximate Q by a function parametrized by θ:

$$Q_{\theta}(s, a) \approx Q(s, a), \ \forall \ (s, a)$$

- (simplest) linear function approximation: $Q_\theta(s, a) = \langle \theta, \phi(s, a) \rangle$ for some "feature" $\phi(s, a)$

Function approximation

Algorithms discussed so far (called **tabular algorithms**) run in time/space poly($|\mathcal{S}||\mathcal{A}|$), which is impractical. (Go has about 2×10^{170} states!)

To overcome this issue, we approximate Q by a function parametrized by θ:

$$Q_{\theta}(s, a) \approx Q(s, a), \ \forall \ (s, a)$$

- (simplest) linear function approximation: $Q_\theta(s, a) = \langle \theta, \phi(s, a) \rangle$ for some "feature" $\phi(s, a)$
- **deep Q-network** (DQN): Q_θ is a neural net with weight θ

Q-learning with function approximation

How to learn θ?

Q-learning with function approximation

How to learn θ?

Recall in the tabular case, with $y_t = r_t + \gamma \max_{a'} Q(s_{t+1}, a')$:

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha \underbrace{(y_t - Q(s_t, a_t))}_{\text{temporal difference}}$$

Q-learning with function approximation

How to learn θ?

Recall in the tabular case, with $y_t = r_t + \gamma \max_{a'} Q(s_{t+1}, a')$:

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha \underbrace{\left(y_t - Q(s_t, a_t)\right)}_{\text{temporal difference}}$$

$$= Q(s_t, a_t) - \alpha \frac{\partial \left(\frac{1}{2} \left(Q(s_t, a_t) - y_t \right)^2 \right)}{\partial Q(s_t, a_t)}$$

Q-learning with function approximation

How to learn θ?

Recall in the tabular case, with $y_t = r_t + \gamma \max_{a'} Q(s_{t+1}, a')$:

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha \underbrace{(y_t - Q(s_t, a_t))}_{\text{temporal difference}}$$

$$= Q(s_t, a_t) - \alpha \frac{\partial \left(\frac{1}{2}\left(Q(s_t, a_t) - y_t\right)^2\right)}{\partial Q(s_t, a_t)}$$

A natural generalization: perform gradient descent on θ with squared loss $\frac{1}{2}\left(Q_\theta(s_t, a_t) - y_t\right)^2$:

Q-learning with function approximation

How to learn θ?

Recall in the tabular case, with $y_t = r_t + \gamma \max_{a'} Q(s_{t+1}, a')$:

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha \underbrace{\left(y_t - Q(s_t, a_t)\right)}_{\text{temporal difference}}$$

$$= Q(s_t, a_t) - \alpha \frac{\partial \left(\frac{1}{2}\left(Q(s_t, a_t) - y_t\right)^2\right)}{\partial Q(s_t, a_t)}$$

A natural generalization: <u>perform gradient descent on θ with squared loss</u> $\frac{1}{2}\left(Q_\theta(s_t, a_t) - y_t\right)^2$:

$$\theta \leftarrow \theta - \alpha \nabla_\theta \left(\frac{1}{2}\left(Q_\theta(s_t, a_t) - y_t\right)^2\right)$$

Q-learning with function approximation

How to learn θ?

Recall in the tabular case, with $y_t = r_t + \gamma \max_{a'} Q(s_{t+1}, a')$:

$$\begin{aligned} Q(s_t, a_t) &\leftarrow Q(s_t, a_t) + \alpha \underbrace{\left(y_t - Q(s_t, a_t)\right)}_{\text{temporal difference}} \\ &= Q(s_t, a_t) - \alpha \frac{\partial \left(\frac{1}{2}\left(Q(s_t, a_t) - y_t\right)^2\right)}{\partial Q(s_t, a_t)} \end{aligned}$$

A natural generalization: perform gradient descent on θ with squared loss $\frac{1}{2}\left(Q_\theta(s_t, a_t) - y_t\right)^2$:

$$\begin{aligned} \theta &\leftarrow \theta - \alpha \nabla_\theta \left(\frac{1}{2}\left(Q_\theta(s_t, a_t) - y_t\right)^2\right) \\ &= \theta - \alpha \left(Q_\theta(s_t, a_t) - y_t\right) \nabla_\theta Q_\theta(s_t, a_t) \end{aligned}$$

Q-learning with function approximation

Q-learning

Initialize θ randomly

Q-learning with function approximation

Q-learning

Initialize θ randomly

For $t = 1, 2, \ldots,$

- **with probability ϵ, explore**: a_t is chosen uniformly at random

Q-learning with function approximation

Q-learning

Initialize θ randomly

For $t = 1, 2, \ldots,$

- **with probability** ϵ**, explore**: a_t is chosen uniformly at random
- **with probability** $1 - \epsilon$**, exploit**: $a_t = \operatorname{argmax}_a Q_\theta(s_t, a)$

Q-learning with function approximation

Q-learning

Initialize θ randomly

For $t = 1, 2, \ldots,$

- **with probability** ϵ**, explore**: a_t is chosen uniformly at random
- **with probability** $1 - \epsilon$**, exploit**: $a_t = \mathrm{argmax}_a \, Q_\theta(s_t, a)$
- execute action a_t, receive reward r_t, arrive at state s_{t+1}

Q-learning with function approximation

Q-learning

Initialize θ randomly

For $t = 1, 2, \ldots,$

- **with probability** ϵ**, explore**: a_t is chosen uniformly at random
- **with probability** $1 - \epsilon$**, exploit**: $a_t = \operatorname{argmax}_a Q_\theta(s_t, a)$
- execute action a_t, receive reward r_t, arrive at state s_{t+1}
- update the parameter of the Q function

$$\theta \leftarrow \theta - \alpha \left(Q_\theta(s_t, a_t) - y_t \right) \nabla_\theta Q_\theta(s_t, a_t)$$

for some learning rate α.

Case study: superhuman AI for Atari games [Deepmind, 2013]

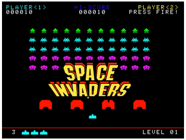
Case study: superhuman AI for Atari games [Deepmind, 2013]

Model each Atari game as an MDP $(\mathcal{S}, \mathcal{A}, P, r, \gamma)$:

Case study: superhuman AI for Atari games [Deepmind, 2013]

Model each Atari game as an MDP $(\mathcal{S}, \mathcal{A}, P, r, \gamma)$:

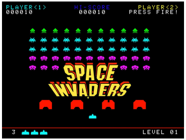
- states: **raw images** (84×84 after preprocessing)



Case study: superhuman AI for Atari games [Deepmind, 2013]

Model each Atari game as an MDP $(\mathcal{S}, \mathcal{A}, P, r, \gamma)$:

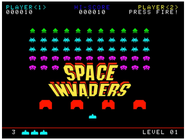
- states: **raw images** (84×84 after preprocessing)
  - no feature engineering, end-to-end (from pixel to action) reinforcement learning, just like humans



Case study: superhuman AI for Atari games [Deepmind, 2013]

Model each Atari game as an MDP $(\mathcal{S}, \mathcal{A}, P, r, \gamma)$:

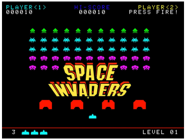
- states: **raw images** (84×84 after preprocessing)
  - no feature engineering, end-to-end (from pixel to action) reinforcement learning, just like humans
  - stack 4 most recent frames as one state (to make things Markovian)



Case study: superhuman AI for Atari games [Deepmind, 2013]

Model each Atari game as an MDP $(\mathcal{S}, \mathcal{A}, P, r, \gamma)$:

- states: **raw images** $(84 \times 84$ after preprocessing)
  - no feature engineering, end-to-end (from pixel to action) reinforcement learning, just like humans
  - stack 4 most recent frames as one state (to make things Markovian)

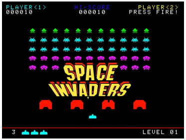


- 18 possible actions:

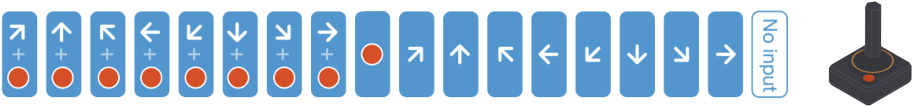
Case study: superhuman AI for Atari games [Deepmind, 2013]

Model each Atari game as an MDP $(\mathcal{S}, \mathcal{A}, P, r, \gamma)$:

- states: **raw images** (84×84 after preprocessing)
  - no feature engineering, end-to-end (from pixel to action) reinforcement learning, just like humans
  - stack 4 most recent frames as one state (to make things Markovian)



- 18 possible actions:

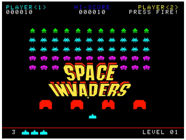


- transition: determined by each game

Case study: superhuman AI for Atari games [Deepmind, 2013]

Model each Atari game as an MDP $(\mathcal{S}, \mathcal{A}, P, r, \gamma)$:

- states: **raw images** (84×84 after preprocessing)
  - no feature engineering, end-to-end (from pixel to action) reinforcement learning, just like humans
  - stack 4 most recent frames as one state (to make things Markovian)



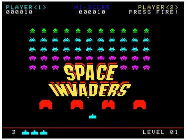
- 18 possible actions:

- transition: determined by each game
- reward: change in score

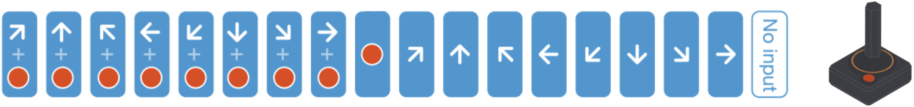
Case study: superhuman AI for Atari games [Deepmind, 2013]

Model each Atari game as an MDP $(\mathcal{S}, \mathcal{A}, P, r, \gamma)$:

- states: **raw images** (84×84 after preprocessing)
  - no feature engineering, end-to-end (from pixel to action) reinforcement learning, just like humans
  - stack 4 most recent frames as one state (to make things Markovian)

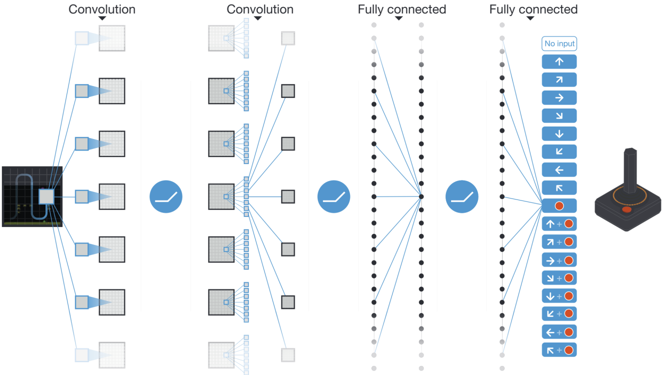


- 18 possible actions:



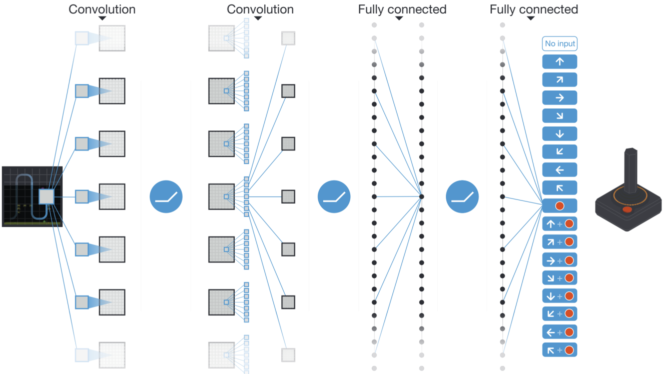
- transition: determined by each game
- reward: change in score
- $\gamma = 0.99$ (but note that the game will end at some point)

Deep Q-Network



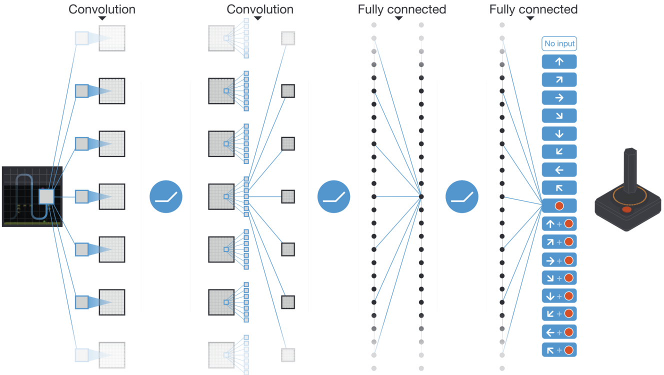
Deep Q-Network

- input: $84 \times 84 \times 4$ images



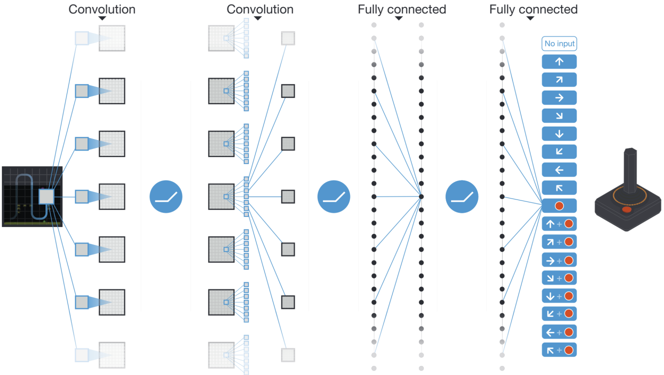
Deep Q-Network

- input: $84 \times 84 \times 4$ images
- 3 convolutional layers + 2 fully-connected layers, 3M parameters



Deep Q-Network

- input: $84 \times 84 \times 4$ images
- 3 convolutional layers + 2 fully-connected layers, 3M parameters
- each of the 18 outputs specifies the Q-value of the corresponding action given a certain state input



Training

For each game, run Q-learning for $T = 50$M (around 38 days of game experience),

Training

For each game, run Q-learning for $T = 50$M (around 38 days of game experience), with **two more tricks**:

Training

For each game, run Q-learning for $T = 50$M (around 38 days of game experience), with **two more tricks**:

- use a target network $\bar{\theta}$ to stabilize training

Training

For each game, run Q-learning for $T = 50$M (around 38 days of game experience), with **two more tricks**:

- use a target network $\bar{\theta}$ to stabilize training

$$y_t = r_t + \gamma \max_{a'} Q_\theta(s_{t+1}, a')$$

Training

For each game, run Q-learning for $T = 50$M (around 38 days of game experience), with **two more tricks**:

- use a target network $\bar{\theta}$ to stabilize training

$$y_t = r_t + \gamma \max_{a'} Q_\theta(s_{t+1}, a') \implies y_t = r_t + \gamma \max_{a'} Q_{\bar{\theta}}(s_{t+1}, a')$$

Training

For each game, run Q-learning for $T = 50$M (around 38 days of game experience), with **two more tricks**:

- use a target network $\bar{\theta}$ to stabilize training

$$y_t = r_t + \gamma \max_{a'} Q_\theta(s_{t+1}, a') \implies y_t = r_t + \gamma \max_{a'} Q_{\bar{\theta}}(s_{t+1}, a')$$

  - $\bar{\theta}$ is a snapshot of θ, updated every 10K rounds

Training

For each game, run Q-learning for $T = 50$M (around 38 days of game experience), with **two more tricks**:

- use a target network $\bar{\theta}$ to stabilize training

$$y_t = r_t + \gamma \max_{a'} Q_\theta(s_{t+1}, a') \implies y_t = r_t + \gamma \max_{a'} Q_{\bar{\theta}}(s_{t+1}, a')$$

  - $\bar{\theta}$ is a snapshot of θ, updated every 10K rounds

- use experience replay to reduce correlation / increase data efficiency

Training

For each game, run Q-learning for $T = 50$M (around 38 days of game experience), with **two more tricks**:

- use a target network $\bar{\theta}$ to stabilize training

$$y_t = r_t + \gamma \max_{a'} Q_\theta(s_{t+1}, a') \implies y_t = r_t + \gamma \max_{a'} Q_{\bar{\theta}}(s_{t+1}, a')$$

  - $\bar{\theta}$ is a snapshot of θ, updated every 10K rounds

- use experience replay to reduce correlation / increase data efficiency
  - instead of using one sample in each update,

$$(Q_\theta(s_t, a_t) - y_t)^2$$

Training

For each game, run Q-learning for $T = 50\text{M}$ (around 38 days of game experience), with **two more tricks**:

- use a target network $\bar{\theta}$ to stabilize training

$$y_t = r_t + \gamma \max_{a'} Q_\theta(s_{t+1}, a') \implies y_t = r_t + \gamma \max_{a'} Q_{\bar{\theta}}(s_{t+1}, a')$$

  - $\bar{\theta}$ is a snapshot of θ, updated every 10K rounds

- use experience replay to reduce correlation / increase data efficiency
  - instead of using one sample in each update, use a minibatch of 32 samples randomly selected from the most recent 1M frames

$$(Q_\theta(s_t, a_t) - y_t)^2 \implies \sum_{k \in \text{minibatch}} (Q_\theta(s_k, a_k) - y_k)^2$$

More on experience replay

Use a minibatch of samples from previous experience

- target: from $(Q_\theta(s_t, a_t) - y_t)^2$ to $\sum_{k \in \text{minibatch}} (Q_\theta(s_k, a_k) - y_k)^2$

More on experience replay

Use a minibatch of samples from previous experience

- target: from $(Q_\theta(s_t, a_t) - y_t)^2$ to $\sum_{k \in \text{minibatch}} (Q_\theta(s_k, a_k) - y_k)^2$
- update: from

$$\theta \leftarrow \theta - \alpha \left(Q_\theta(s_t, a_t) - y_t\right) \nabla_\theta Q_\theta(s_t, a_t)$$

to

$$\theta \leftarrow \theta - \alpha \sum_{k \in \text{minibatch}} \left(Q_\theta(s_k, a_k) - y_k\right) \nabla_\theta Q_\theta(s_k, a_k)$$

More on experience replay

Use a minibatch of samples from previous experience

- target: from $(Q_\theta(s_t, a_t) - y_t)^2$ to $\sum_{k \in \text{minibatch}} (Q_\theta(s_k, a_k) - y_k)^2$
- update: from

$$\theta \leftarrow \theta - \alpha \left(Q_\theta(s_t, a_t) - y_t\right) \nabla_\theta Q_\theta(s_t, a_t)$$

to

$$\theta \leftarrow \theta - \alpha \sum_{k \in \text{minibatch}} \left(Q_\theta(s_k, a_k) - y_k\right) \nabla_\theta Q_\theta(s_k, a_k)$$

- in the tabular case, it means from (see programming project)

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) - \alpha(Q(s_t, a_t) - y_t)$$

to

$$Q(s_k, a_k) \leftarrow Q(s_k, a_k) - \alpha(Q(s_k, a_k) - y_k), \quad \forall k \in \text{minibatch}$$

Results

- tested on 49 Atari Games, 5 mins each game for 30 times

Results

- tested on 49 Atari Games, 5 mins each game for 30 times
- same model architecture, same algorithm, same hyperparameters

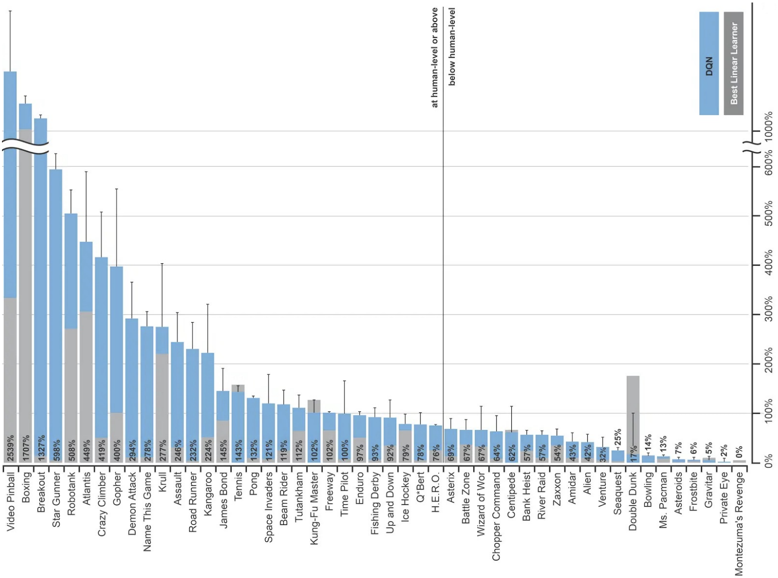
Results

- tested on 49 Atari Games, 5 mins each game for 30 times
- same model architecture, same algorithm, same hyperparameters
- compared against best linear learner and a professional human tester

Results

- tested on 49 Atari Games, 5 mins each game for 30 times
- same model architecture, same algorithm, same hyperparameters
- compared against best linear learner and a professional human tester
- report $\frac{\text{DQN score} - \text{random play score}}{\text{human score} - \text{random play score}} \times 100\%$

Results



DQN
Best Linear Learner
at human-level or above
below human-level
0%
100%
200%
300%
400%
500%
600%
1000%
Video Pinball 2539%
Boxing 1707%
Breakout 1327%
Star Gunner 598%
Robotank 508%
Atlantis 449%
Crazy Climber 419%
Gopher 400%
Demon Attack 294%
Name This Game 278%
Krull 277%
Assault 246%
Road Runner 232%
Kangaroo 224%
James Bond 145%
Tennis 143%
Pong 132%
Space Invaders 121%
Beam Rider 119%
Tutankham 112%
Kung-Fu Master 102%
Freeway 102%
Time Pilot 100%
Enduro 97%
Fishing Derby 93%
Up and Down 92%
Ice Hockey 79%
Q*Bert 78%
H.E.R.O. 76%
Asterix 69%
Battle Zone 67%
Wizard of Wor 67%
Chopper Command 64%
Centipede 62%
Bank Heist 57%
River Raid 57%
Zaxxon 54%
Amidar 43%
Alien 42%
Venture 32%
Seaquest 25%
Double Dunk 17%
Bowling 14%
Ms. Pacman 13%
Asteroids 7%
Frostbite 6%
Gravitar 5%
Private Eye 2%
Montezuma's Revenge 0%

Outline

Learning policies directly

Another popular class of RL algorithms learns the policy directly:

Learning policies directly

Another popular class of RL algorithms learns the policy directly:

$$\max_{\pi} \text{ "expected reward of policy } \pi\text{"}$$

Learning policies directly

Another popular class of RL algorithms learns the policy directly:

$$\max_{\pi} \text{ "expected reward of policy } \pi\text{"}$$

To handle large scale problems, consider a parameterized policy class $\Pi = \{\pi_\rho : \rho \in \Omega\}$ (e.g., a set of neural nets)

Learning policies directly

Another popular class of RL algorithms learns the policy directly:

$$\max_{\pi} \text{ ``expected reward of policy } \pi\text{''}$$

To handle large scale problems, consider a parameterized policy class $\Pi = \{\pi_\rho : \rho \in \Omega\}$ (e.g., a set of neural nets) and solve

$$\max_{\rho \in \Omega} \text{ ``expected reward of policy } \pi_\rho\text{''}$$

Learning policies directly

Another popular class of RL algorithms learns the policy directly:

$$\max_{\pi} \text{ ``expected reward of policy } \pi\text{''}$$

To handle large scale problems, consider a parameterized policy class $\Pi = \{\pi_\rho : \rho \in \Omega\}$ (e.g., a set of neural nets) and solve

$$\max_{\rho \in \Omega} \text{ ``expected reward of policy } \pi_\rho\text{''}$$

via stochastic gradient descent

Policy gradient theorem

For simplicity, suppose $\gamma = 1$ and a trajectory ends after H steps.

Policy gradient theorem

For simplicity, suppose $\gamma = 1$ and a trajectory ends after H steps.

Expected reward of π_ρ can be written as

$$R(\pi_\rho) = \sum_\tau P_\rho(\tau) R(\tau)$$

Policy gradient theorem

For simplicity, suppose $\gamma = 1$ and a trajectory ends after H steps.

Expected reward of π_ρ can be written as

$$R(\pi_\rho) = \sum_{\tau} P_\rho(\tau) R(\tau)$$

- $\tau = (s_1, a_1, \ldots, s_H, a_H)$ ranges over all possible H-step trajectories

Policy gradient theorem

For simplicity, suppose $\gamma = 1$ and a trajectory ends after H steps.

Expected reward of π_ρ can be written as

$$R(\pi_\rho) = \sum_\tau P_\rho(\tau) R(\tau)$$

- $\tau = (s_1, a_1, \ldots, s_H, a_H)$ ranges over all possible H-step trajectories
- $P_\rho(\tau)$ is the probability of encountering trajectory τ under policy π_ρ

Policy gradient theorem

For simplicity, suppose $\gamma = 1$ and a trajectory ends after H steps.

Expected reward of π_ρ can be written as

$$R(\pi_\rho) = \sum_\tau P_\rho(\tau) R(\tau)$$

- $\tau = (s_1, a_1, \dots, s_H, a_H)$ ranges over all possible H-step trajectories
- $P_\rho(\tau)$ is the probability of encountering trajectory τ under policy π_ρ
- $R(\tau) = \sum_{h=1}^{H} r(s_h, a_h)$ is the cumulative reward for trajectory τ

Policy gradient theorem

For simplicity, suppose $\gamma = 1$ and a trajectory ends after H steps.

Expected reward of π_ρ can be written as

$$R(\pi_\rho) = \sum_{\tau} P_\rho(\tau) R(\tau)$$

- $\tau = (s_1, a_1, \ldots, s_H, a_H)$ ranges over all possible H-step trajectories
- $P_\rho(\tau)$ is the probability of encountering trajectory τ under policy π_ρ
- $R(\tau) = \sum_{h=1}^{H} r(s_h, a_h)$ is the cumulative reward for trajectory τ

So we have

$$\nabla_\rho R(\pi_\rho) = \sum_{\tau} \nabla_\rho P_\rho(\tau) R(\tau)$$

Policy gradient theorem

For simplicity, suppose $\gamma = 1$ and a trajectory ends after H steps.

Expected reward of π_ρ can be written as

$$R(\pi_\rho) = \sum_\tau P_\rho(\tau) R(\tau)$$

- $\tau = (s_1, a_1, \ldots, s_H, a_H)$ ranges over all possible H-step trajectories
- $P_\rho(\tau)$ is the probability of encountering trajectory τ under policy π_ρ
- $R(\tau) = \sum_{h=1}^{H} r(s_h, a_h)$ is the cumulative reward for trajectory τ

So we have

$$\nabla_\rho R(\pi_\rho) = \sum_\tau \nabla_\rho P_\rho(\tau) R(\tau)$$

How do we efficiently compute/approximate it?

Policy gradient theorem (cont.)

$$\nabla_\rho R(\pi_\rho) = \sum_\tau \nabla_\rho P_\rho(\tau) R(\tau)$$

Policy gradient theorem (cont.)

$$\nabla_\rho R(\pi_\rho) = \sum_\tau \nabla_\rho P_\rho(\tau) R(\tau) = \sum_\tau P_\rho(\tau) \frac{\nabla_\rho P_\rho(\tau)}{P_\rho(\tau)} R(\tau)$$

Policy gradient theorem (cont.)

$$\nabla_\rho R(\pi_\rho) = \sum_\tau \nabla_\rho P_\rho(\tau) R(\tau) = \sum_\tau P_\rho(\tau) \frac{\nabla_\rho P_\rho(\tau)}{P_\rho(\tau)} R(\tau)$$

$$= \sum_\tau P_\rho(\tau) \nabla_\rho \log P_\rho(\tau) R(\tau) \qquad \text{(log derivative trick)}$$

Policy gradient theorem (cont.)

$$
\begin{aligned}
\nabla_\rho R(\pi_\rho) &= \sum_\tau \nabla_\rho P_\rho(\tau) R(\tau) = \sum_\tau P_\rho(\tau) \frac{\nabla_\rho P_\rho(\tau)}{P_\rho(\tau)} R(\tau) \\
&= \sum_\tau P_\rho(\tau) \nabla_\rho \log P_\rho(\tau) R(\tau) && \text{(log derivative trick)} \\
&= \mathbb{E}_\tau \left[\nabla_\rho \log P_\rho(\tau) R(\tau) \right] && \text{(written as an } \textbf{expectation}\text{)}
\end{aligned}
$$

Policy gradient theorem (cont.)

$$
\begin{aligned}
\nabla_\rho R(\pi_\rho) &= \sum_\tau \nabla_\rho P_\rho(\tau) R(\tau) = \sum_\tau P_\rho(\tau) \frac{\nabla_\rho P_\rho(\tau)}{P_\rho(\tau)} R(\tau) \\
&= \sum_\tau P_\rho(\tau) \nabla_\rho \log P_\rho(\tau) R(\tau) && \text{(log derivative trick)} \\
&= \mathbb{E}_\tau \left[\nabla_\rho \log P_\rho(\tau) R(\tau) \right] && \text{(written as an } \textbf{expectation}\text{)} \\
&= \mathbb{E}_\tau \left[\nabla_\rho \log \left(\Pi_{h=1}^H \pi_\rho(a_h | s_h) P(s_{h+1} | s_h, a_h) \right) R(\tau) \right]
\end{aligned}
$$

Policy gradient theorem (cont.)

$$\nabla_\rho R(\pi_\rho) = \sum_\tau \nabla_\rho P_\rho(\tau) R(\tau) = \sum_\tau P_\rho(\tau) \frac{\nabla_\rho P_\rho(\tau)}{P_\rho(\tau)} R(\tau)$$

$$= \sum_\tau P_\rho(\tau) \nabla_\rho \log P_\rho(\tau) R(\tau)$$ (log derivative trick)

$$= \mathbb{E}_\tau \left[\nabla_\rho \log P_\rho(\tau) R(\tau)\right]$$ (written as an **expectation**)

$$= \mathbb{E}_\tau \left[\nabla_\rho \log \left(\Pi_{h=1}^H \pi_\rho(a_h|s_h) P(s_{h+1}|s_h, a_h)\right) R(\tau)\right]$$

$$= \mathbb{E}_\tau \left[\left(\sum_{h=1}^H \nabla_\rho \log \pi_\rho(a_h|s_h)\right) R(\tau)\right]$$ (*transition doesn't matter!*)

Policy gradient theorem (cont.)

$$\nabla_\rho R(\pi_\rho) = \sum_\tau \nabla_\rho P_\rho(\tau) R(\tau) = \sum_\tau P_\rho(\tau) \frac{\nabla_\rho P_\rho(\tau)}{P_\rho(\tau)} R(\tau)$$

$$= \sum_\tau P_\rho(\tau) \nabla_\rho \log P_\rho(\tau) R(\tau) \qquad \text{(log derivative trick)}$$

$$= \mathbb{E}_\tau \left[\nabla_\rho \log P_\rho(\tau) R(\tau) \right] \qquad \text{(written as an } \textbf{expectation}\text{)}$$

$$= \mathbb{E}_\tau \left[\nabla_\rho \log \left(\Pi_{h=1}^{H} \pi_\rho(a_h | s_h) P(s_{h+1} | s_h, a_h) \right) R(\tau) \right]$$

$$= \mathbb{E}_\tau \left[\left(\sum_{h=1}^{H} \nabla_\rho \log \pi_\rho(a_h | s_h) \right) R(\tau) \right] \qquad \text{(\textit{transition doesn't matter!})}$$

which can be approximated by sampling n trajectories using π_ρ and taking the empirical average:

$$\frac{1}{n} \sum_{i=1}^{n} \left(\sum_{h=1}^{H} \nabla_\rho \log \pi_\rho(a_h^{(i)} | s_h^{(i)}) \right) R(\tau^{(i)})$$

Reducing variance of gradient estimators via baselines

The key to make policy gradient work is to **reduce variance** of gradient estimators.

Reducing variance of gradient estimators via baselines

The key to make policy gradient work is to **reduce variance** of gradient estimators. Subtracting a "baseline" is a standard way to achieve so:

Reducing variance of gradient estimators via baselines

The key to make policy gradient work is to **reduce variance** of gradient estimators. Subtracting a "baseline" is a standard way to achieve so:

$$\nabla_\rho R(\pi_\rho) = \mathbb{E}_\tau \left[\sum_{h=1}^{H} \nabla_\rho \log \pi_\rho(a_h|s_h) R(\tau) \right]$$

Reducing variance of gradient estimators via baselines

The key to make policy gradient work is to **reduce variance** of gradient estimators. Subtracting a "baseline" is a standard way to achieve so:

$$\begin{aligned}\nabla_\rho R(\pi_\rho) &= \mathbb{E}_\tau\left[\sum_{h=1}^{H} \nabla_\rho \log \pi_\rho(a_h|s_h) R(\tau)\right] \\ &= \mathbb{E}_\tau\left[\sum_{h=1}^{H} \nabla_\rho \log \pi_\rho(a_h|s_h)\left(R(\tau) - b(s_{1:h}, a_{1:h-1})\right)\right]\end{aligned}$$

Reducing variance of gradient estimators via baselines

The key to make policy gradient work is to **reduce variance** of gradient estimators. Subtracting a "baseline" is a standard way to achieve so:

$$\nabla_\rho R(\pi_\rho) = \mathbb{E}_\tau \left[\sum_{h=1}^{H} \nabla_\rho \log \pi_\rho(a_h|s_h) R(\tau) \right]$$

$$= \mathbb{E}_\tau \left[\sum_{h=1}^{H} \nabla_\rho \log \pi_\rho(a_h|s_h) \left(R(\tau) - b(s_{1:h}, a_{1:h-1}) \right) \right]$$

This holds *for any b that only depends on $s_{1:h}, a_{1:h-1}$*, because

$$\mathbb{E}_{a_h} \left[\nabla_\rho \log \pi_\rho(a_h|s_h) b \right] = b \sum_{a_h \in \mathcal{A}} \pi_\rho(a_h|s_h) \frac{\nabla_\rho \pi_\rho(a_h|s_h)}{\pi_\rho(a_h|s_h)}$$

Reducing variance of gradient estimators via baselines

The key to make policy gradient work is to **reduce variance** of gradient estimators. Subtracting a "baseline" is a standard way to achieve so:

$$\nabla_\rho R(\pi_\rho) = \mathbb{E}_\tau \left[\sum_{h=1}^{H} \nabla_\rho \log \pi_\rho(a_h|s_h) R(\tau) \right]$$

$$= \mathbb{E}_\tau \left[\sum_{h=1}^{H} \nabla_\rho \log \pi_\rho(a_h|s_h) \left(R(\tau) - b(s_{1:h}, a_{1:h-1}) \right) \right]$$

This holds *for any b that only depends on* $s_{1:h}, a_{1:h-1}$, because

$$\mathbb{E}_{a_h} \left[\nabla_\rho \log \pi_\rho(a_h|s_h) b \right] = b \sum_{a_h \in \mathcal{A}} \pi_\rho(a_h|s_h) \frac{\nabla_\rho \pi_\rho(a_h|s_h)}{\pi_\rho(a_h|s_h)}$$

$$= b \nabla_\rho \sum_{a_h \in \mathcal{A}} \pi_\rho(a_h|s_h)$$

Reducing variance of gradient estimators via baselines

The key to make policy gradient work is to **reduce variance** of gradient estimators. Subtracting a "baseline" is a standard way to achieve so:

$$\begin{aligned}\nabla_\rho R(\pi_\rho) &= \mathbb{E}_\tau \left[\sum_{h=1}^{H} \nabla_\rho \log \pi_\rho(a_h|s_h) R(\tau) \right] \\ &= \mathbb{E}_\tau \left[\sum_{h=1}^{H} \nabla_\rho \log \pi_\rho(a_h|s_h) \left(R(\tau) - b(s_{1:h}, a_{1:h-1}) \right) \right]\end{aligned}$$

This holds *for any b that only depends on* $s_{1:h}, a_{1:h-1}$, because

$$\mathbb{E}_{a_h} \left[\nabla_\rho \log \pi_\rho(a_h|s_h) b \right] = b \sum_{a_h \in \mathcal{A}} \pi_\rho(a_h|s_h) \frac{\nabla_\rho \pi_\rho(a_h|s_h)}{\pi_\rho(a_h|s_h)}$$

$$= b \nabla_\rho \sum_{a_h \in \mathcal{A}} \pi_\rho(a_h|s_h) = b \nabla_\rho 1$$

Reducing variance of gradient estimators via baselines

The key to make policy gradient work is to **reduce variance** of gradient estimators. Subtracting a "baseline" is a standard way to achieve so:

$$\begin{aligned}\nabla_\rho R(\pi_\rho) &= \mathbb{E}_\tau\left[\sum_{h=1}^{H} \nabla_\rho \log \pi_\rho(a_h|s_h) R(\tau)\right] \\ &= \mathbb{E}_\tau\left[\sum_{h=1}^{H} \nabla_\rho \log \pi_\rho(a_h|s_h) \left(R(\tau) - b(s_{1:h}, a_{1:h-1})\right)\right]\end{aligned}$$

This holds *for any b that only depends on $s_{1:h}, a_{1:h-1}$*, because

$$\mathbb{E}_{a_h}\left[\nabla_\rho \log \pi_\rho(a_h|s_h) b\right] = b \sum_{a_h \in \mathcal{A}} \pi_\rho(a_h|s_h) \frac{\nabla_\rho \pi_\rho(a_h|s_h)}{\pi_\rho(a_h|s_h)}$$

$$= b \nabla_\rho \sum_{a_h \in \mathcal{A}} \pi_\rho(a_h|s_h) = b \nabla_\rho 1 = 0$$

Which baselines?

$$\nabla_\rho R(\pi_\rho) = \mathbb{E}_\tau \left[\sum_{h=1}^{H} \nabla_\rho \log \pi_\rho(a_h|s_h) \left(R(\tau) - b(s_{1:h}, a_{1:h-1}) \right) \right]$$

Want $b(s_{1:h}, a_{1:h-1})$ to be close to $R(\tau)$,

Which baselines?

$$\nabla_\rho R(\pi_\rho) = \mathbb{E}_\tau \left[\sum_{h=1}^{H} \nabla_\rho \log \pi_\rho(a_h|s_h) \left(R(\tau) - b(s_{1:h}, a_{1:h-1})\right) \right]$$

Want $b(s_{1:h}, a_{1:h-1})$ to be close to $R(\tau)$, leading to an **idealized** choice:

"observed reward before h" + "expected reward starting from h"

Which baselines?

$$\nabla_\rho R(\pi_\rho) = \mathbb{E}_\tau \left[\sum_{h=1}^{H} \nabla_\rho \log \pi_\rho(a_h|s_h) \left(R(\tau) - b(s_{1:h}, a_{1:h-1})\right) \right]$$

Want $b(s_{1:h}, a_{1:h-1})$ to be close to $R(\tau)$, leading to an **idealized** choice:

"observed reward before h" + "expected reward starting from h"

$$= \left(\sum_{h'=1}^{h-1} r(s_{h'}, a_{h'}) \right) + \underbrace{\mathbb{E}\left[\sum_{h'=h}^{H} r(s_{h'}, a_{h'}) \mid s_{h'} = s_h \right]}_{V_{\pi_\rho}(s_h)}$$

Which baselines?

$$\nabla_\rho R(\pi_\rho) = \mathbb{E}_\tau \left[\sum_{h=1}^{H} \nabla_\rho \log \pi_\rho(a_h|s_h) \left(R(\tau) - b(s_{1:h}, a_{1:h-1}) \right) \right]$$

Want $b(s_{1:h}, a_{1:h-1})$ to be close to $R(\tau)$, leading to an **idealized** choice:

"observed reward before h" + "expected reward starting from h"

$$= \left(\sum_{h'=1}^{h-1} r(s_{h'}, a_{h'}) \right) + \underbrace{\mathbb{E}\left[\sum_{h'=h}^{H} r(s_{h'}, a_{h'}) \mid s_{h'} = s_h \right]}_{V_{\pi_\rho}(s_h)}$$

V_{π_ρ}, called a **critic**,

Which baselines?

$$\nabla_\rho R(\pi_\rho) = \mathbb{E}_\tau \left[\sum_{h=1}^{H} \nabla_\rho \log \pi_\rho(a_h|s_h) \left(R(\tau) - b(s_{1:h}, a_{1:h-1})\right) \right]$$

Want $b(s_{1:h}, a_{1:h-1})$ to be close to $R(\tau)$, leading to an **idealized** choice:

"observed reward before h" + "expected reward starting from h"

$$= \left(\sum_{h'=1}^{h-1} r(s_{h'}, a_{h'}) \right) + \underbrace{\mathbb{E}\left[\sum_{h'=h}^{H} r(s_{h'}, a_{h'}) \mid s_{h'} = s_h \right]}_{V_{\pi_\rho}(s_h)}$$

V_{π_ρ}, called a **critic**, is usually **approximated** by another network θ:

"observed reward before h" + "estimated reward starting from h"

Which baselines?

$$\nabla_\rho R(\pi_\rho) = \mathbb{E}_\tau \left[\sum_{h=1}^{H} \nabla_\rho \log \pi_\rho(a_h|s_h) \left(R(\tau) - b(s_{1:h}, a_{1:h-1}) \right) \right]$$

Want $b(s_{1:h}, a_{1:h-1})$ to be close to $R(\tau)$, leading to an **idealized** choice:

"observed reward before h" + "expected reward starting from h"

$$= \left(\sum_{h'=1}^{h-1} r(s_{h'}, a_{h'}) \right) + \underbrace{\mathbb{E} \left[\sum_{h'=h}^{H} r(s_{h'}, a_{h'}) \mid s_{h'} = s_h \right]}_{V_{\pi_\rho}(s_h)}$$

V_{π_ρ}, called a **critic**, is usually **approximated** by another network θ:

"observed reward before h" + "estimated reward starting from h"

$$= \left(\sum_{h'=1}^{h-1} r(s_{h'}, a_{h'}) \right) + V_\theta(s_h)$$

Actor-Critic methods

Repeat:

- Critic evaluates the current policy π_ρ by fitting V_θ from samples using square loss:

$$\min_\theta \sum_{j=1}^{m} \sum_{h=1}^{H} \left(V_\theta\left(s_h^{(j)}\right) - \sum_{h'=h}^{H} r\left(s_{h'}^{(j)}, a_{h'}^{(j)}\right) \right)^2$$

Actor-Critic methods

Repeat:

- Critic evaluates the current policy π_ρ by fitting V_θ from samples using square loss:

$$\min_\theta \sum_{j=1}^{m} \sum_{h=1}^{H} \left(V_\theta \left(s_h^{(j)} \right) - \sum_{h'=h}^{H} r \left(s_{h'}^{(j)}, a_{h'}^{(j)} \right) \right)^2$$

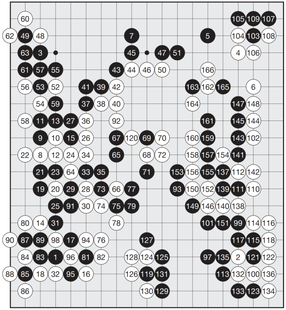
- Actor improves the current policy π_ρ via stochastic gradient descent:

$$\rho \leftarrow \rho - \frac{\alpha}{n} \sum_{i=1}^{n} \sum_{h=1}^{H} \nabla_\rho \log \pi_\rho(a_h^{(i)} | s_h^{(i)}) \underbrace{\left(\sum_{h'=h}^{H} r \left(s_{h'}^{(i)}, a_{h'}^{(i)} \right) - V_\theta(s_h^{(i)}) \right)}_{= R(\tau^{(i)}) - b(s_{1:h}^{(i)}, a_{1:h-1}^{(i)})}$$

Case study: AlphaGo

[Deepmind, 2015]

Model Go as an MDP $(\mathcal{S}, \mathcal{A}, P, r, \gamma)$:

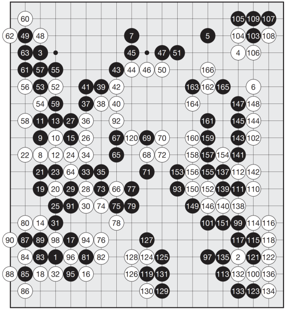


Case study: AlphaGo

[Deepmind, 2015]

Model Go as an MDP $(\mathcal{S}, \mathcal{A}, P, r, \gamma)$:

- states: each 19×19 position of the game is pre-processed into an $19 \times 19 \times 48$ image stack consisting of feature planes

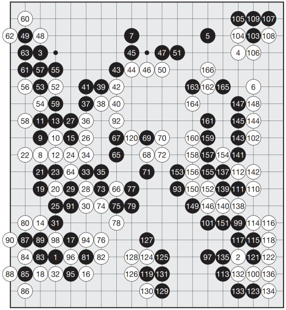


Case study: AlphaGo

[Deepmind, 2015]

Model Go as an MDP $(\mathcal{S}, \mathcal{A}, P, r, \gamma)$:

- states: each 19×19 position of the game is pre-processed into an $19 \times 19 \times 48$ image stack consisting of feature planes
- actions: all legal next moves

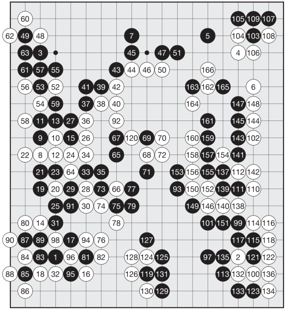


Case study: AlphaGo

[Deepmind, 2015]

Model Go as an MDP $(\mathcal{S}, \mathcal{A}, P, r, \gamma)$:

- states: each 19×19 position of the game is pre-processed into an $19 \times 19 \times 48$ image stack consisting of feature planes
- actions: all legal next moves
- transition: determined by the opponent

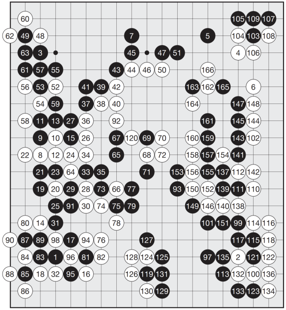


Case study: AlphaGo

[Deepmind, 2015]

Model Go as an MDP $(\mathcal{S}, \mathcal{A}, P, r, \gamma)$:

- states: each 19×19 position of the game is pre-processed into an $19 \times 19 \times 48$ image stack consisting of feature planes
- actions: all legal next moves
- transition: determined by the opponent
- reward: only the ending state has reward (1 if win, -1 if lose)

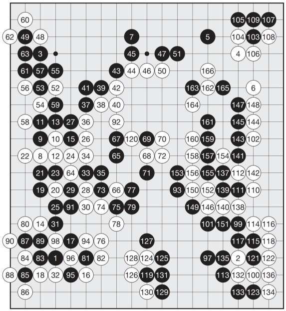


Case study: AlphaGo

[Deepmind, 2015]

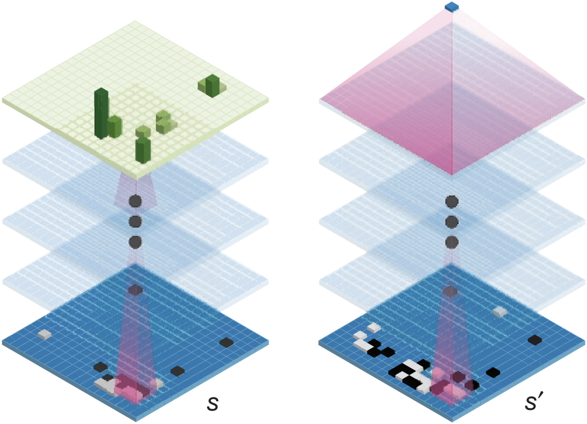
Model Go as an MDP $(\mathcal{S}, \mathcal{A}, P, r, \gamma)$:

- states: each 19×19 position of the game is pre-processed into an $19 \times 19 \times 48$ image stack consisting of feature planes
- actions: all legal next moves
- transition: determined by the opponent
- reward: only the ending state has reward (1 if win, -1 if lose)
- $\gamma = 1$



Policy/value networks

Both π_ρ and V_θ are large convolutional neural nets:



Training

Step 1: first train a policy π_σ using pure **supervised learning** from 30M expert moves (a multiclass classification task)

Training

Step 1: first train a policy π_σ using pure **supervised learning** from 30M expert moves (a multiclass classification task)

Step 2: use **actor-critic** to train policy network π_ρ and value network V_θ

Training

Step 1: first train a policy π_σ using pure **supervised learning** from 30M expert moves (a multiclass classification task)

Step 2: use **actor-critic** to train policy network π_ρ and value network V_θ

- initialize ρ as σ

Training

Step 1: first train a policy π_σ using pure **supervised learning** from 30M expert moves (a multiclass classification task)

Step 2: use **actor-critic** to train policy network π_ρ and value network V_θ

- initialize ρ as σ
- self-play: every 500 iterations, add current ρ to an opponent pool; in each iteration, randomly sampled one from this pool as the opponent

Training

Step 1: first train a policy π_σ using pure **supervised learning** from 30M expert moves (a multiclass classification task)

Step 2: use **actor-critic** to train policy network π_ρ and value network V_θ

- initialize ρ as σ
- self-play: every 500 iterations, add current ρ to an opponent pool; in each iteration, randomly sampled one from this pool as the opponent
- trained for 10K iterations, each with 128 games

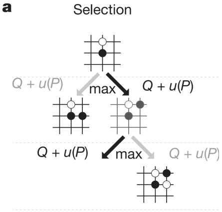
Testing (actual play)

"**Monte-Carlo Tree Search**" with the help of policy/value networks:

Testing (actual play)

"**Monte-Carlo Tree Search**" with the help of policy/value networks:

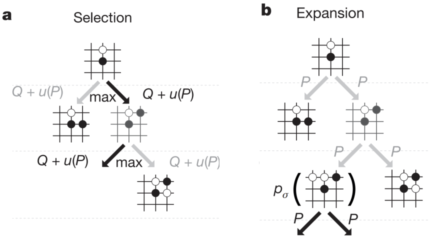
- **select** a move with highest estimated quality Q + UCB (inversely proportional to #visits, just like bandits)



Testing (actual play)

"**Monte-Carlo Tree Search**" with the help of policy/value networks:

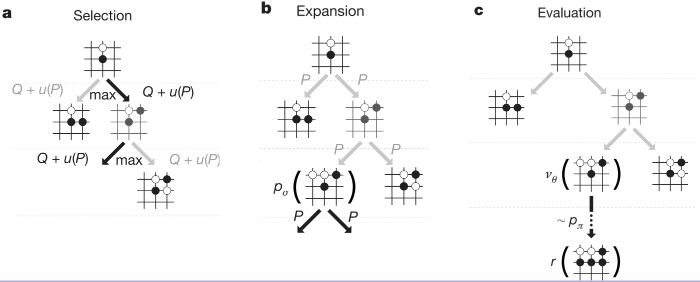
- **select** a move with highest estimated quality Q + UCB (inversely proportional to #visits, just like bandits)
- **expand** a leaf (when visited more than a certain times) using π_σ



Testing (actual play)

"**Monte-Carlo Tree Search**" with the help of policy/value networks:

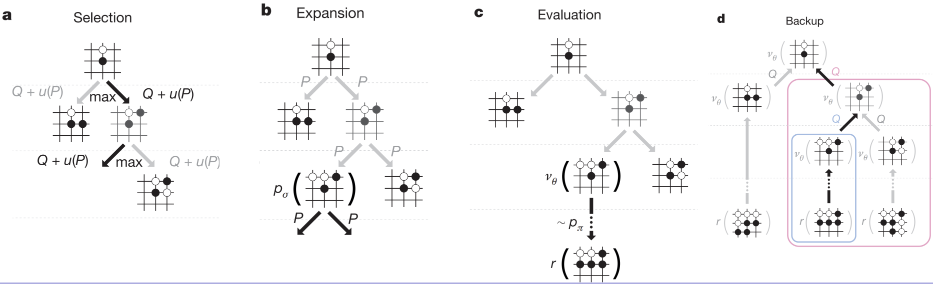
- **select** a move with highest estimated quality Q + UCB (inversely proportional to #visits, just like bandits)
- **expand** a leaf (when visited more than a certain times) using π_σ
- **evaluate** the leaf by averaging V_θ's prediction and a random rollout



Testing (actual play)

"**Monte-Carlo Tree Search**" with the help of policy/value networks:

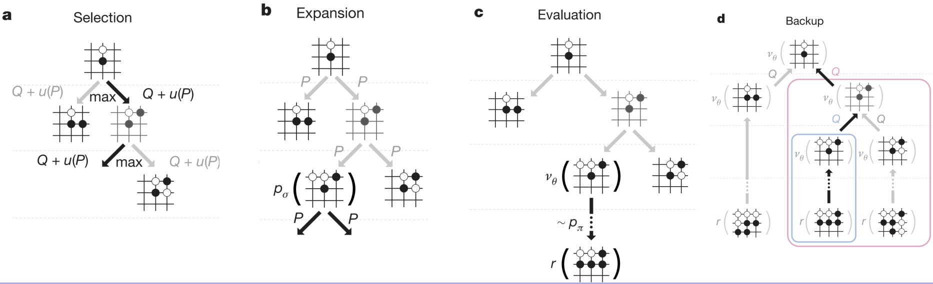
- **select** a move with highest estimated quality Q + UCB (inversely proportional to #visits, just like bandits)
- **expand** a leaf (when visited more than a certain times) using π_σ
- **evaluate** the leaf by averaging V_θ's prediction and a random rollout
- **update** the quality Q value along the traversed edges



Testing (actual play)

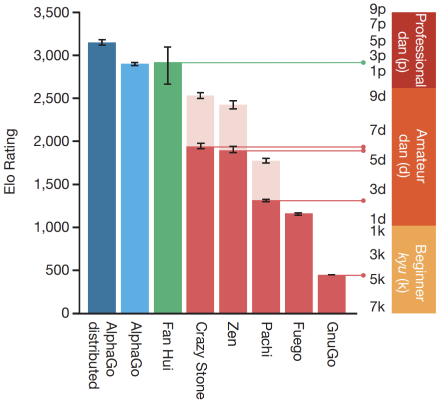
"**Monte-Carlo Tree Search**" with the help of policy/value networks:

- **select** a move with highest estimated quality Q + UCB (inversely proportional to #visits, just like bandits)
- **expand** a leaf (when visited more than a certain times) using π_σ
- **evaluate** the leaf by averaging V_θ's prediction and a random rollout
- **update** the quality Q value along the traversed edges
- when the search halts, select the most visited move at the root



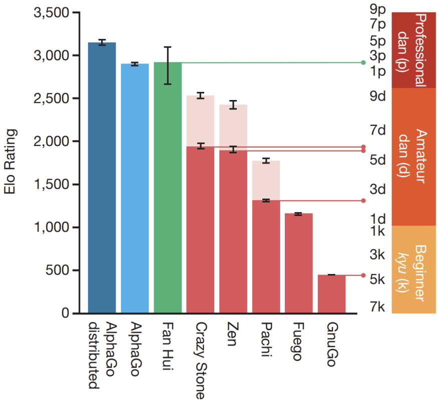
Results

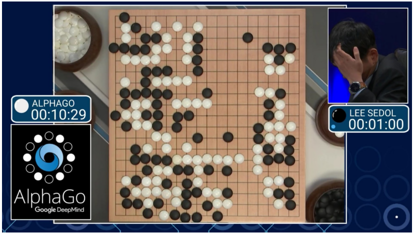
- 99.8% win rate against other Go programs



Results

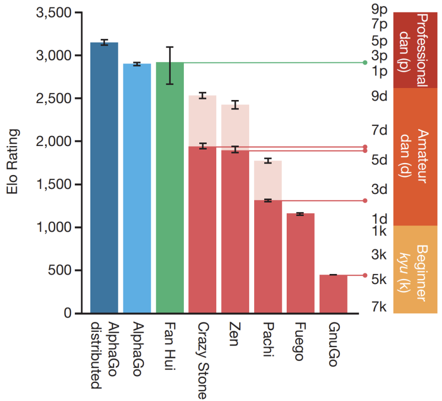
- 99.8% win rate against other Go programs
- $5\text{-}0$ Fan Hui (2013/2014/2015 European Go champion)

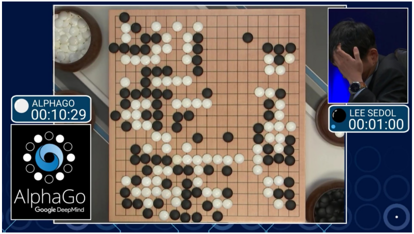




Results

- 99.8% win rate against other Go programs
- 5-0 Fan Hui (2013/2014/2015 European Go champion)
- first superhuman AI for Go, previously believed to be a decade away





Summary

A brief introduction to (deep) RL:

- **foundation**: MDP, value iteration, model-based/free learning

Summary

A brief introduction to (deep) RL:

- **foundation**: MDP, value iteration, model-based/free learning
- **large-scale and practical** deep RL methods:

Summary

A brief introduction to (deep) RL:

- **foundation**: MDP, value iteration, model-based/free learning
- **large-scale and practical** deep RL methods:
  - Q-learning with function approximation, DQN, and their success in Atari games

Summary

A brief introduction to (deep) RL:

- **foundation**: MDP, value iteration, model-based/free learning
- **large-scale and practical** deep RL methods:
  - Q-learning with function approximation, DQN, and their success in Atari games
  - policy gradient, actor-critic methods, and their success in Go