Learning From Data
Lecture 10: Reinforcement Learning

Yang Li **yangli@sz.tsinghua.edu.cn**

TBSI

November 29, 2024

Today's Lecture

Reinforcement Learning

- What's reinforcement learning?
- Mathematical formulation: Markov Decision Process (MDP)
- Model Learning for MDP, Fitted Value Iteration
- Deep reinforcement learning (Deep Q-networks)

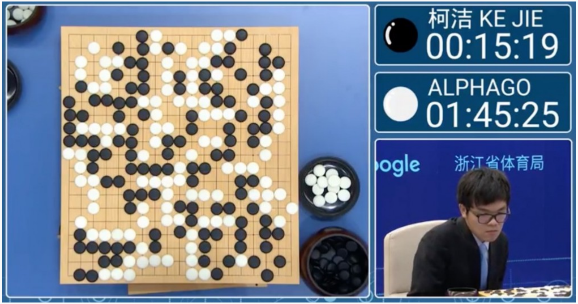
Reinforcement Learning: Autonomous Car, Helicopter

Stanley, Winner of DARPA Grand Challenge (2005)
Inverted autonomous helicopter flight (2004)

Other applications include robotic control, computational economics, health care...

Deep Reinforcement Learning: AlphaGo

AlphaGo beat World Go Champion Kejie (2017)



Nature paper on by AlphaGo team

Deep Reinforcement Learning: OpenAI

OpenAI beats Dota2 world champion (2017)

Elon Musk @elonmusk — Follow

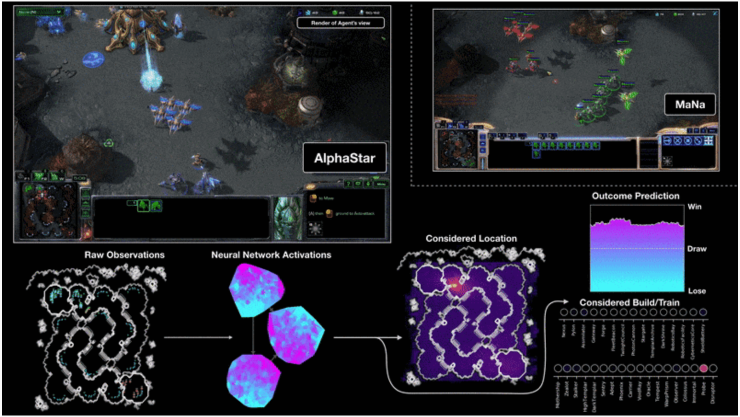
OpenAI first ever to defeat world's best players in competitive eSports. Vastly more complex than traditional board games like chess & Go.

3:15 AM - Aug 12, 2017

647 · 6,818 · 23,006

Multi-Agent Reinforcement Learning: AlphaStar

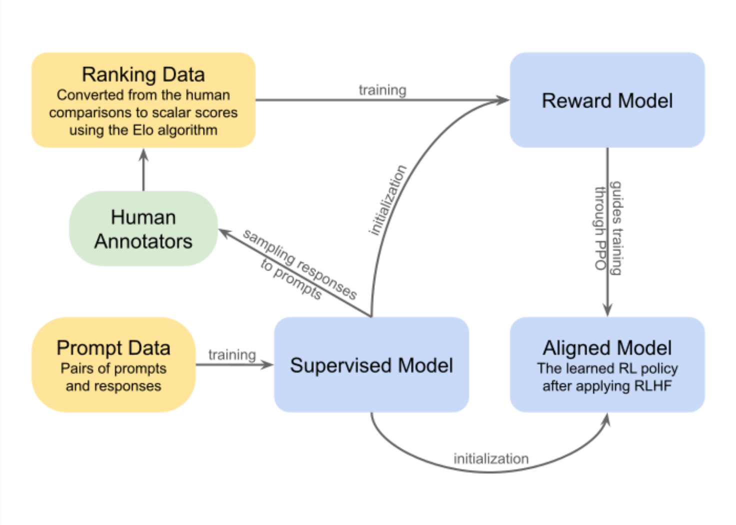
AlphaStar reached Grandmaster level in StarCraft II (2019)



https://www.nature.com/articles/s41586-019-1724-z

Reinforcement Learning with Human Feedback

Align an intelligent agent to human preferences. (2019)



What is reinforcement learning?

Sequential decision making

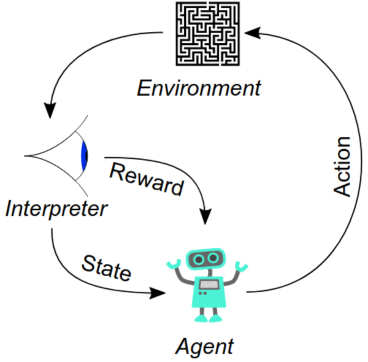
To decide, from **experience**, the **sequence of actions** to perform in an **uncertain environment** in order to achieve some **goals**.

- e.g. play games, robotic control, autonomous driving, smart grid
- Do not have full knowledge of the environment a priori
- Difficult to label a sample as "the right answer" for a learning goal

What is reinforcement learning?

A learning framework to solve sequential decision making problem, inspired by behavior psychology (Sutton, 1984)

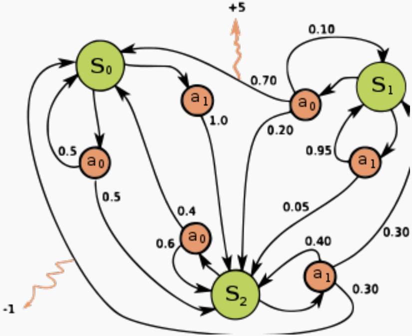
- An agent interacts with an environment which provides a "reward function" to indicate how "well" the learning agent is doing
- The agents take actions to maximize the cumulative "reward"



Markov Decision Process

A Markov decision process $(S, A, \{P_{sa}\}, \gamma, R)$

- S: a set of **states** (environment)
- A: a set of **actions**
- $P_{sa} := P(s_{t+1}|s_t, a_t)$: **state transition** probabilities. *Markov property:* $P(s_{t+1}|s_t, a_t) = P(s_{t+1}|s_t, a_t, \dots, s_0, a_0)$.
- $R : S \times A \to \mathbb{R}$ is a **reward function**
- $\gamma \in [0, 1)$: discount factor



$S = \{S_0, S_1, S_2\}$

$A = \{a_0, a_1\}$

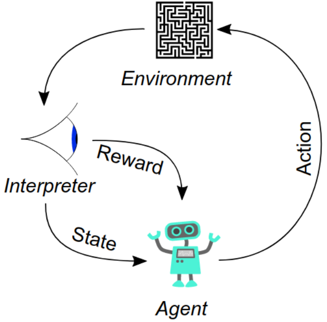
$R(s_1, a_0) = 5, R(s_2, a_1) = -1$

	S_0	S_1	S_2
S_0, a_0	0.5	0	0.5
S_0, a_1	0	0	1
S_1, a_0	0.7	0.1	0.2
S_1, a_1	0	0.95	0.05
S_2, a_0	0.4	0.6	0
S_2, a_1	0.3	0.3	0.4

Markov Decision Process: Overview

At time step $t = 0$ with initial state $s_0 \in S$
for $t = 0$ until done:

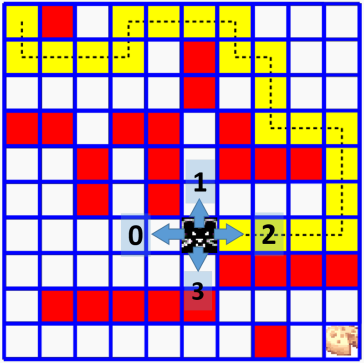
- Agent selects action at $a_t \in A$
- Environment yields reward $r_t = R(s_t, a_t)$
- Environment samples next state $s_{t+1} \sim P_{sa}$
- Agent receives reward r_t and next state s_{t+1}



A **policy** $\pi : S \rightarrow A$ specifies what action to take in each state

Goal: find optimal policy π^* that maximizes cumulative discounted reward

MDP Example: Maze Solver



https://www.samyzaf.com/ML/rl/qmaze.html

Goal: get to the bottom-right corner of the nxn maze

- S: position of the agent (mouse)
- A: {Left, Right, Up, Down}
- $P_{sa}(s') = \begin{cases} 1 & s' \text{ is next move} \\ 0 & \text{otherwise} \end{cases}$
- $R(a,s) =$
  $\begin{cases} -0.05 & \text{move to free cell} \\ -1 & \text{move to wall/block} \\ 1 & \text{move to goal} \end{cases}$
- $\gamma \in [0,1)$: discount factor

MDP Example: Maze Solver

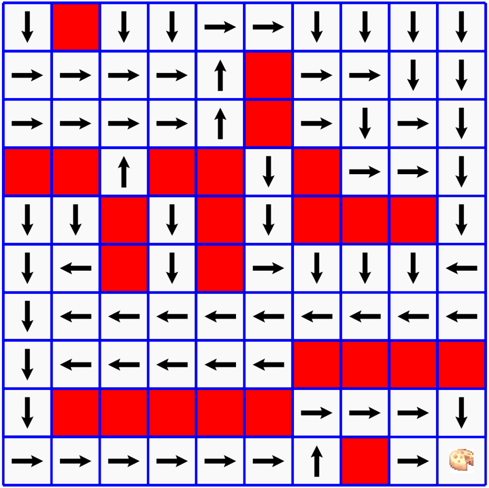
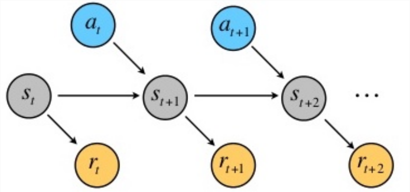


Figure: An optimal policy function $\pi(s)$ learned by the solver.

https://www.samyzaf.com/ML/rl/qmaze.html

Markov Decision Process

Consider a sequence of states $s_0, s_1, \ldots$ with actions $a_0, a_1, \ldots,$



Total payoff of a sequence:

$$R(s_0, a_0) + \gamma R(s_1, a_1) + \gamma^2 R(s_2, a_2) + \ldots$$

For simplicity, let's assume rewards only depends on state s, i.e.

$$R(s_0) + \gamma R(s_1) + \gamma^2 R(s_2) + \ldots$$

Future reward at step t is discounted by γ^t

Policy & value functions

Goal of reinforcement learning: choose actions that maximize the **expected total payoff**

$$\mathbb{E}_{\mu_0, P_{sa}, \pi}[R(s_0) + \gamma R(s_1) + \gamma^2 R(s_2) + \dots]$$

A **policy** is any function $\pi\colon S \to A$.
A **value function** of policy π is the expected payoff if we start from s, take actions according to π:

$$V^\pi(s) \triangleq \mathbb{E}[R(s_0) + \gamma R(s_1) + \gamma^2 R(s_2) + \dots | s_0 = s, \pi]$$

Given π, value function satisfies the **Bellman equation**: *Why?*

$$V^\pi(s) = R(s) + \gamma \sum_{s' \in S} P_{s\pi(s)}(s') V^\pi(s')$$

Bellman Equation

Value function of π at s:

$$V^\pi(s) = \mathbb{E}[R(s_0, \pi(s_0) + \gamma R(s_1, \pi(s_1)) + \gamma^2 R(s_2, \pi(s_2)) + \dots | s_0 = s, \pi]$$

Assume action is known:

$$\begin{aligned} V^\pi(s) &= \mathbb{E}[R(s_0) + \gamma R(s_1) + \gamma^2 R(s_2) + \dots | s_0 = s, \pi] \\ &= \mathbb{E}[R(s)] + \gamma \mathbb{E}[R(s_1) + \gamma R(s_2) + \dots | s_0 = s, \pi] \\ &= R(s) + \gamma \mathbb{E}_{s' \sim P_{s,\pi(s)}}[V^\pi(s')] \\ &= R(s) + \gamma \sum_{s' \in S} P_{s,\pi(s)}(s') V^\pi(s') \ \textit{discrete state} \\ &\text{or } R(s) + \gamma \int_{s'} P_{s,\pi(s)}(s') V^\pi(s') ds' \ \textit{continuous state} \end{aligned}$$

Policy & value functions

For a finite state space, given R, P_{sa}, π, we can find $V^{\pi}(s)$ using Bellman's equation:

$$V^{\pi}(s) = R(s) + \gamma \sum_{s' \in S} P_{s\pi(s)}(s') V^{\pi}(s')$$

$V^{\pi}(s)$ can be solved as $|S|$ linear equations with $|S|$ unknowns.

Optimal value and policy

We define the **optimal value function**

$$\begin{aligned} V^*(s) = \max_\pi V^\pi(s) &= R(s) + \max_\pi \gamma \sum_{s' \in S} P_{s,\pi(s)}(s') V^\pi(s') \\ &= R(s) + \max_{a \in A} \gamma \sum_{s' \in S} P_{sa}(s') V^*(s') \end{aligned}$$

Let $\pi^* : S \to A$ be the policy that attains the 'max':

$$\pi^*(s) = \operatorname*{argmax}_{a \in A} \sum_{s' \in S} P_{sa}(s') V^*(s')$$

Then for every state s and every policy π, we can show

$$V^*(s) = V^{\pi^*}(s) \geq V^\pi(s)$$

π^* is the optimal policy for any initial state s

Optimal value and policy

Proposition 1

For every state s,

$$V^*(s) = V^{\pi^*}(s)$$

Proof.

□

Solving finite-state MDP: value iteration

Assume the MDP has finite state and action space.

```
1. For each state s, initialize V(s) := 0
2. Repeat until convergence {
        Update V(s) := R(s) + max_{a∈A} γ ∑_{s'∈S} P_sa(s') V(s')
          for every state s
   }
```

Two ways to update $V(s)$:

- Synchronous update:

  Set $V_0(s) := V(s)$ for all states $s \in S$
  For each $s \in S$:
  $$V(s) := R(s) + \max_{a \in A} \gamma \sum_{s' \in S} P_{sa}(s') V_0(s')$$

- Asynchronous update:

  For each $s \in S$:
  $$V(s) := R(s) + \max_{a \in A} \gamma \sum_{s' \in S} P_{sa}(s') V(s')$$

Solving finite-state MDP: policy iteration

```
1. Initialize π randomly
2. Repeat until convergence {
    a. Let   V := V^π
    b. For each state s,
         π(s) := argmax_{a∈A} Σ_{s'} P_{sa}(s') V(s')
   }
```

Step (a) can be done by solving Bellman's equation.

Discussion

Both value iteration and policy iteration will converge to V^* and π^*

Value iteration vs. policy iteration

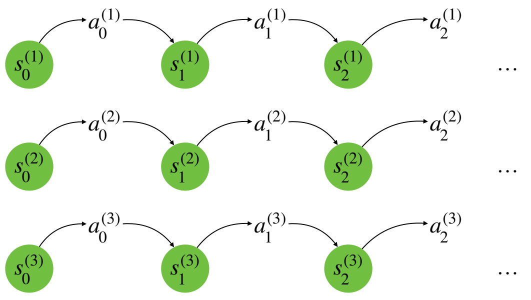
- Policy iteration is more efficient and converge faster for small MDP
- Value iteration is more practical for MDP's with large state spaces

Learning a model for finite-state MDP

Suppose the reward function $R(s)$ and the transition probability P_{sa} is not known. How to estimate them from data?

Experience from MDP

Given policy π, execute π repeatedly in the environment:



⋮

Estimate model from experience

Estimate P_{sa}

Maximum likelihood estimate of state transition probability:

$$P_{sa}(s') = P(s'|s,a) = \frac{\#\{s \xrightarrow{a} s'\}}{\#\{s \xrightarrow{a} \cdot\}}$$

If $\#\{s \xrightarrow{a} \cdot\} = 0$, set $P_{sa}(s') = \frac{1}{|S|}$.

Estimate $R(s)$

Let $R(s)^{(t)}$ be the immediate reward of state s in the t-th trail,

$$R(s) = \mathbb{E}[R(s)^{(t)}] = \frac{1}{m}\sum_{t=1}^{m} R(s)^{(t)}$$

Algorithm: MDP Model Learning

1. Initialize π randomly, $V(s) := 0$ for all s
2. Repeat until convergence {
   a. Execute π for m trails
   b. Update P_{sa} and R using the accumulated experience
   c. $V :=$ValueIteration(P_{sa}, R, V)
   b. Update π greedily with respect to V:
      $\pi(s) := \text{argmax}_{a \in A} \sum_{s'} P_{sa}(s')V(s')$
   }

ValueIteration(P_{sa}, R, V_0)

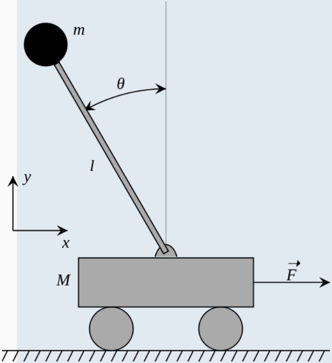
1. Initialize $V = V_0$
2. Repeat until convergence {
      Update $V(s) := R(s) + \max_{a \in A} \gamma \sum_{s' \in S} P_{sa}(s')V(s')$
      for every state s
   }

Continuous state MDPs

An MDP may have an infinite number of states:

- A car's state : $(x, y, \theta, \dot{x}, \dot{y}, \dot{\theta})$
- A helicopter's state : $(x, y, z, \phi, \theta, \psi, \dot{x}, \dot{y}, \dot{z}, \dot{\phi}, \dot{\theta}, \dot{\psi})$

1D Inverted Pendulum



Control goal: balance the pole on the cart

- State representation: $(x, \theta, \dot{x}, \dot{\theta})$
- Action: force F on the car
- Reward: $+1$ each time the pole is upright

Due to the Curse of Dimensionality, discretization rarely works well in continuous state with more than 1-2 dimensions

Value function approximation

How to approximate V directly without resorting to discretization?

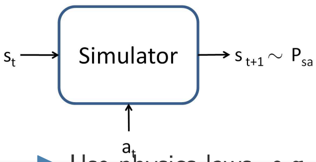
Main ideas:

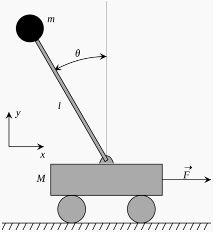
- Obtain a *model* or *simulator* for the MDP, to produce **experience tuples**: $\langle s, a, s', r \rangle$
- Sample $s^{(1)}, \ldots, s^{(m)}$ from the state space S, estimate their optimal expected total payoff using the model, i.e. $y^{(1)} \approx V(s^{(1)}), y^{(2)} \approx V(s^{(2)}), \ldots$
- Approximate V as a function of state s using supervised learning from $(s^{(1)}, y^{(1)}), (s^{(2)}, y^{(2)}), \ldots$ e.g.

$$V(s) = \theta^T \phi(s)$$

Obtaining a simulator

A **simulator** is a black box that generates the next state s_{t+1} given current state s_t and action a_t.





- Use physics laws. e.g. equation of motion for the inversed pendulum problem:

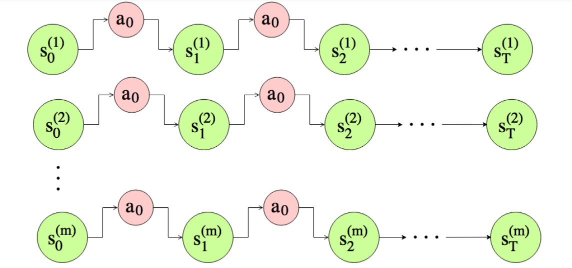
$$(m+M)\ddot{x} + mL(\dot{\theta}^2 \sin\theta - \ddot{\theta}\cos(\theta)) = F$$

$$g\sin\theta + \ddot{x}\cos\theta = L\ddot{\theta}$$

- Use out-of-the-shelf simulation software
- Game simulator

Obtaining a model from data

Execute m trails in which we repeatedly take actions in an MDP, each trial for T timesteps.



Learn a prediction model $s_{t+1} = h_\theta\left(\begin{bmatrix} s_t \\ a_t \end{bmatrix}\right)$ by picking

$$\theta^* = \underset{\theta}{\operatorname{argmin}} \sum_{i=1}^{m} \sum_{t=0}^{T-1} \left\| s_{t+1}^{(i)} - h_\theta\left(\begin{bmatrix} s_t^{(i)} \\ a_t^{(i)} \end{bmatrix}\right) \right\|^2$$

Obtaining a model from data

Popular prediction models

- Linear function: $h_\theta = As_t + Ba_t$
- Linear function with feature mapping: $h_\theta = A\phi_s(s_t) + B\phi_a(a_t)$
- Neural network

Build a simulator using the model:

- Deterministic model: $s_{t+1} = h_\theta\left(\begin{bmatrix} s_t \\ a_t \end{bmatrix}\right)$
- Stochastic model: $s_{t+1} = h_\theta\left(\begin{bmatrix} s_t \\ a_t \end{bmatrix}\right) + \epsilon_t \ , \ \ \epsilon_t \sim \mathcal{N}(0, \Sigma)$

Value function approximation

How to approximate V directly without resorting to discretization?

Main ideas:

- Obtain a model or simulator for the MDP
- Sample $s^{(1)}, \dots, s^{(m)}$ from the state space S, estimate their optimal expected total payoff using the model, i.e. $y^{(1)} \approx V(s^{(1)}), y^{(2)} \approx V(s^{(2)}), \dots$
- Approximate V as a function of state s using supervised learning from $(s^{(1)}, y^{(1)}), (s^{(2)}, y^{(2)}), \dots$ e.g.

$$V(s) = \theta^T \phi(s)$$

Value function for continuous states

Update for finite-state value function:

$$V(s) := R(s) + \gamma \max_{a \in A} \sum_{s' \in S} P_{sa}(s') V(s')$$

Update we want for continuous-state value function:

$$\begin{aligned} V(s) &:= R(s) + \gamma \max_{a \in A} \int_{s'} P_{sa}(s') V(s') ds' \\ &= R(s) + \gamma \max_{a \in A} \mathbb{E}_{s' \sim P_{sa}} [V(s')] \end{aligned}$$

For each sample state s, we compute $y^{(i)}$ to approximate $R(s) + \gamma \max_{a \in A} \mathbb{E}_{s' \sim P_{s^{(i)}a}} [V(s')]$ using finite samples drawn from P_{sa}

Value function approximation

How to approximate V directly without resorting to discretization?

Main ideas:

- Obtain a model or simulator for the MDP
- Sample $s^{(1)}, \dots, s^{(m)}$ from the state space S, estimate their optimal expected total payoff using the model, i.e. $y^{(1)} \approx V(s^{(1)}), y^{(2)} \approx V(s^{(2)}), \dots$
- Approximate V as a function of state s using supervised learning from $(s^{(1)}, y^{(1)}), (s^{(2)}, y^{(2)}), \dots$ e.g.

$$V(s) = \theta^T \phi(s)$$

Fitted value iteration

Algorithm: Fitted value iteration (Stochastic Model)

1. Sample $s^{(1)}, \dots, s^{(m)} \in S$
2. Initialize $\theta := 0$
2. Repeat {
   a. For each sample $s^{(i)}$
      For each action a:
         Sample $s'_1, \dots, s'_k \sim P_{s^{(i)},a}$ using a model
         Compute $Q(a) = \frac{1}{k}\sum_{j=1}^{k} R(s^{(i)}) + \gamma V(s'_j)$
            ↑ estimates $R(s^{(i)}) + \gamma \mathbb{E}_{s' \sim P_{s'a}}[V(s')]$
            where $V(s) := \theta^T \phi(s)$

      $y^{(i)} = \max_a Q(a)$
      ↑ estimates $R(s^{(i)}) + \gamma \max_a \mathbb{E}_{s' \sim P_{s'a}}[V(s')]$
   b. Update θ using supervised learning:
      $\theta := \text{argmin}_\theta \frac{1}{2}\sum_{i=1}^{m}(\theta^T \phi(s^{(i)}) - y^{(i)})^2$
   }

If the model is deterministic, set $k = 1$

Computing the optimal policy

After obtaining the value function approximation V, the corresponding policy is

$$\pi(s) = \underset{a}{\text{argmax}}\, \mathbb{E}_{s' \sim P_{sa}}[V(s')])$$

Estimate the optimal policy from experience:

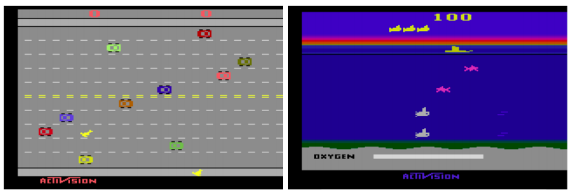
```
For each action a :
   1. Sample s'_1, ..., s'_k ~ P_{s,a} using a model
   2. Compute Q(a) = (1/k) sum_{j=1}^k R(s) + γV(s'_j)
π(s) = argmax_a Q(a)
```

Instead of linear regression, other learning algorithms can be used to estimate $V(s)$.

Two Outstanding Success Stories

Atari AI [Minh et al. 2015]

- Plays a variety of Atari 2600 video games at superhuman level
- Trained directly from image pixels, based on a single reward signal



AlphaGo [Silver et al. 2016]

- A hybrid deep RL system
- Trained using supervised and reinforcement learning, in combination with a traditional tree-search algorithm.

Deep Reinforcement Learning

Main difference from classic RL:

- Use deep network to represent value function
- Optimize value function end-to-end
- Use stochastic gradient descent

Q-Value Function

Given policy π which produce sample sequence $(s_0, a_0, r_0), (s_1, a_1, r_1), \ldots$

- Value function of π :

$$V^\pi(s) = \mathbb{E}\left[\sum_{t \geq 0} \gamma^t r_t \middle| s_0 = s, \pi\right]$$

- The **Q-value function** $Q^\pi(s, a)$ is the expected payoff if we take a at state s and follow π

$$Q^\pi(s, a) = \mathbb{E}\left[\sum_{t \geq 0} \gamma^t r_t \middle| s_0 = s, a_0 = a, \pi\right]$$

- The optimal Q-value function is:

$$Q^*(s, a) = \max_\pi Q^\pi(s, a) = \max_\pi \mathbb{E}\left[\sum_{t \geq 0} \gamma^t r_t \middle| s_0 = s, a_0 = a, \pi\right]$$

Q-Learning

Bellman's equation for Q-Value function:

$$Q^*(s,a) = \mathbb{E}_{s' \sim \mathcal{E}}[r(s) + \gamma \max_{a'} Q^*(s', a')|s, a]$$

Value iteration is not practical when the search space is large.

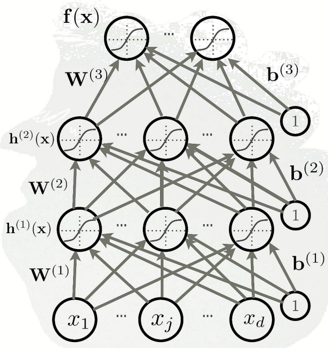
e.g. In an Atari game, each frame is an 128-color 210×160 image, then $|S| = 128^{210 \times 160}$

- Uses a function approximation:

$$Q(s, a; \theta) \approx Q^*(s, a)$$

- In deep Q-learning, $Q(s, a; \theta)$ is a neural network

Neural Network Review



Training goal: $\min_\theta \sum_{i=1}^m L(f(x^{(i)};\theta), y^{(i)})$

Forward propagation

Initialize $h^{(0)}(x) = x$

For each layer $l = 1 \dots d$:

- $a^{(l)}(x) = W^{(l)} h^{(l-1)}(x) + b^{(l)}$
- $h^{(l)}(x) = g(a^{(l)}(x))$

Evaluate loss function $L(h^{(d)}(x), y)$

Backward propagation

Compute gradient $\frac{dL}{dh^{(d)}}$

For each layer $l = d \dots 1$:

- Update gradient for parameters in layer l

Q-Networks

Training goal: find $Q(s, a; \theta)$ that fits Bellman's equation:
$Q^*(s, a) = \mathbb{E}_{s' \sim \mathcal{E}}[r(s) + \gamma \max_{a'} Q^*(s', a')|s, a]$

Forward Pass

Loss function:

$$L_i(\theta_i) = \mathbb{E}_{s,a}[(y_i - Q(s, a; \theta_i)^2]$$

where $y_i = \mathbb{E}_{s' \sim \mathcal{E}}[r + \gamma \max_{a'} Q(s', a'; \theta_{i-1})|s, a]$

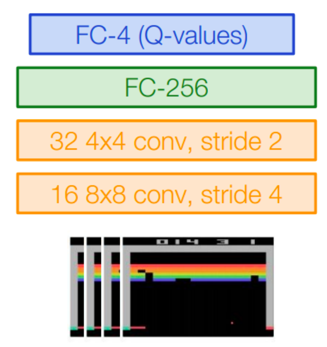
Backward Pass

Update parameter θ by computing gradient

$$\nabla_{\theta_i} L_i(\theta_i) = \mathbb{E}_{s,a,s' \sim \mathcal{E}} \left[\left(r + \gamma \max_{a'} Q(s', a'; \theta_{i-1}) - Q(s, a; \theta_i) \right) \nabla_\theta Q(s, a; \theta_i) \right]$$

Deep Q-Network Architecture

- Input: 4 consecutive frames
- Preprocessing: convert to grayscale, down-sampling, cropping. Final dimension $84 \times 84 \times 4$
- Output: Q-value functions for 4 actions $Q(s, a_1)$, $Q(s, a_2)$, $Q(s, a_3)$, $Q(s, a_4)$



Experience Replay

Challenge of standard deep Q-learning: correlated input

- invalidate the i.i.d. assumption on training samples
- current policy may restrict action samples we experience in the environment

Experience replay

- Store past transitions (s_t, a_t, r_t, s_{t+1}) within a sliding window in the **replay memory** D.
- Train Q-Network using random mini-batch sampled from D to reduce sample correlation
- Also reduces total running time by reusing samples

The Algorithm

Algorithm 1 Deep Q-learning with Experience Replay

Initialize replay memory $\mathcal{D}$ to capacity N
Initialize action-value function Q with random weights
for episode $= 1, M$ **do**
  Initialise sequence $s_1 = \{x_1\}$ and preprocessed sequenced $\phi_1 = \phi(s_1)$
  for $t = 1, T$ **do**
    With probability ϵ select a random action a_t
    otherwise select $a_t = \max_a Q^*(\phi(s_t), a; \theta)$
    Execute action a_t in emulator and observe reward r_t and image x_{t+1}
    Set $s_{t+1} = s_t, a_t, x_{t+1}$ and preprocess $\phi_{t+1} = \phi(s_{t+1})$
    Store transition $(\phi_t, a_t, r_t, \phi_{t+1})$ in $\mathcal{D}$
    Sample random minibatch of transitions $(\phi_j, a_j, r_j, \phi_{j+1})$ from $\mathcal{D}$
    Set $y_j = \begin{cases} r_j & \text{for terminal } \phi_{j+1} \\ r_j + \gamma \max_{a'} Q(\phi_{j+1}, a'; \theta) & \text{for non-terminal } \phi_{j+1} \end{cases}$
    Perform a gradient descent step on $(y_j - Q(\phi_j, a_j; \theta))^2$ according to equation 3
  end for
end for

Parameter ϵ controls the exploration vs. optimization trade-off

Reinforcement Learning Demo

See Demo.
https://cs.stanford.edu/people/karpathy/convnetjs/demo/rldemo.html

Model-based vs Model-free Reinforcement Learning

- **Model-based**: rely on the use of an explicit transition model P_{sa}, which is learned or known a priori (e.g. Value/Policy Iteration)
- **Model-free**: learns the policy or value function directly without modeling P_{sa} (e.g. Q-learning, Policy Gradient).

Figure: A Taxonomy of RL algorithms (OpenAI)