349 Machine Learning

Fall 2024

Deep Reinforcement Learning

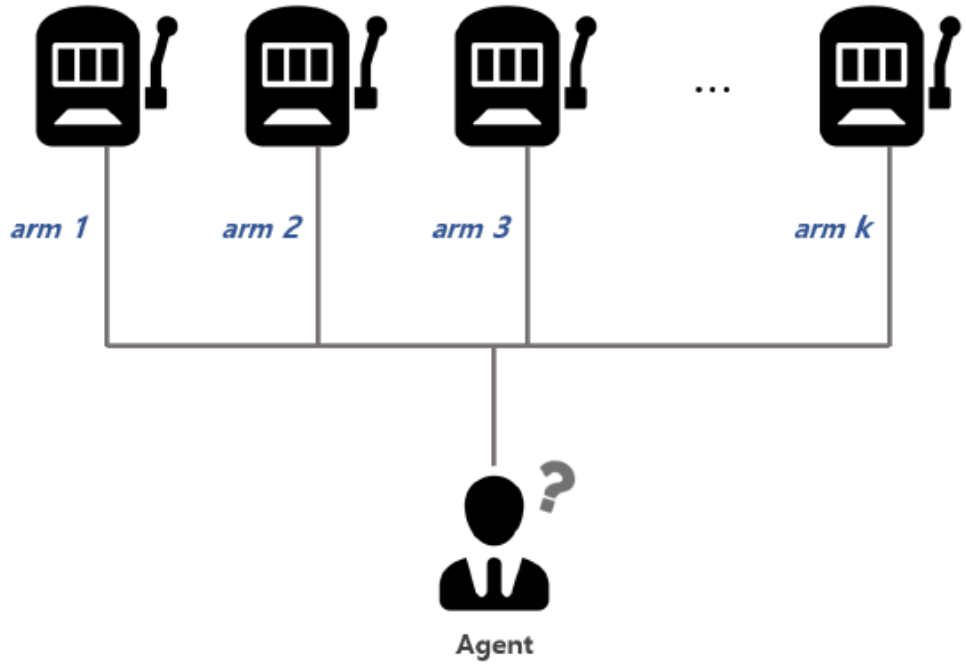
Learning Types

- Supervised learning:
  - (Input, output) pairs of the function to be learned are given (e.g. image labeling)
- Unsupervised Learning:
  - No human labels provided (e.g. language modeling, image reconstruction)
- Reinforcement learning:
  - Reward or punishment for actions (winning or losing a game)

Reinforcement Learning

- Task
  - Learn how to behave to achieve a goal
  - Learn through experience from trial and error
- Examples
  - Game playing: The agent knows when it wins, but doesn't know the appropriate action in each state along the way
  - Control: a traffic system can measure the delay of cars, but not know how to decrease it.

Multi Armed Bandit Problem



Which slot machine do I play?

image from https://velog.io/@taejinjeong/Reinforcement-Learning-Multi-Armed-Bandit-Problem

Exploration vs. Exploitation

- We want to pick good actions most of the time, but also do some exploration
- Exploring means we can learn better policies
- But we want to balance known good actions with exploratory ones
- This is the **exploration / exploitation** problem

Rewards

- Rewards measure how well the policy is doing
  - Often correspond to events in the world
    - Current load on a machine
    - Reaching the coffee machine
    - Program crashing
  - Everything else gets a 0 reward

These are sparse rewards

These are dense rewards

- Things work better if the rewards are incremental
  - For example, gain/loss of stock portfolio or slot machine
  - These reward functions are often hard to design

Picking Actions

ε-greedy

- Pick best (greedy) action with probability 1 - ε
- Otherwise, pick a random action

- Softmax function
  - Pick an action randomly based on its Q-value

$$p(A = a \mid S = s) = \frac{\exp(Q(s, a))}{\sum_{a'} \exp(Q(s, a'))}$$

Multi-Armed Bandits and Epsilon-Greedy

A simple bandit algorithm

Initialize, for $a = 1$ to k:

$Q(a) \leftarrow 0$

$N(a) \leftarrow 0$

Loop forever:

$$A \leftarrow \begin{cases} \text{argmax}_a\, Q(a) & \text{with probability } 1-\varepsilon \quad \text{(breaking ties randomly)} \\ \text{a random action} & \text{with probability } \varepsilon \end{cases}$$

$R \leftarrow bandit(A)$

$N(A) \leftarrow N(A) + 1$

$Q(A) \leftarrow Q(A) + \frac{1}{N(A)}\left[R - Q(A)\right]$

Multi-Armed Bandits and Epsilon-Greedy

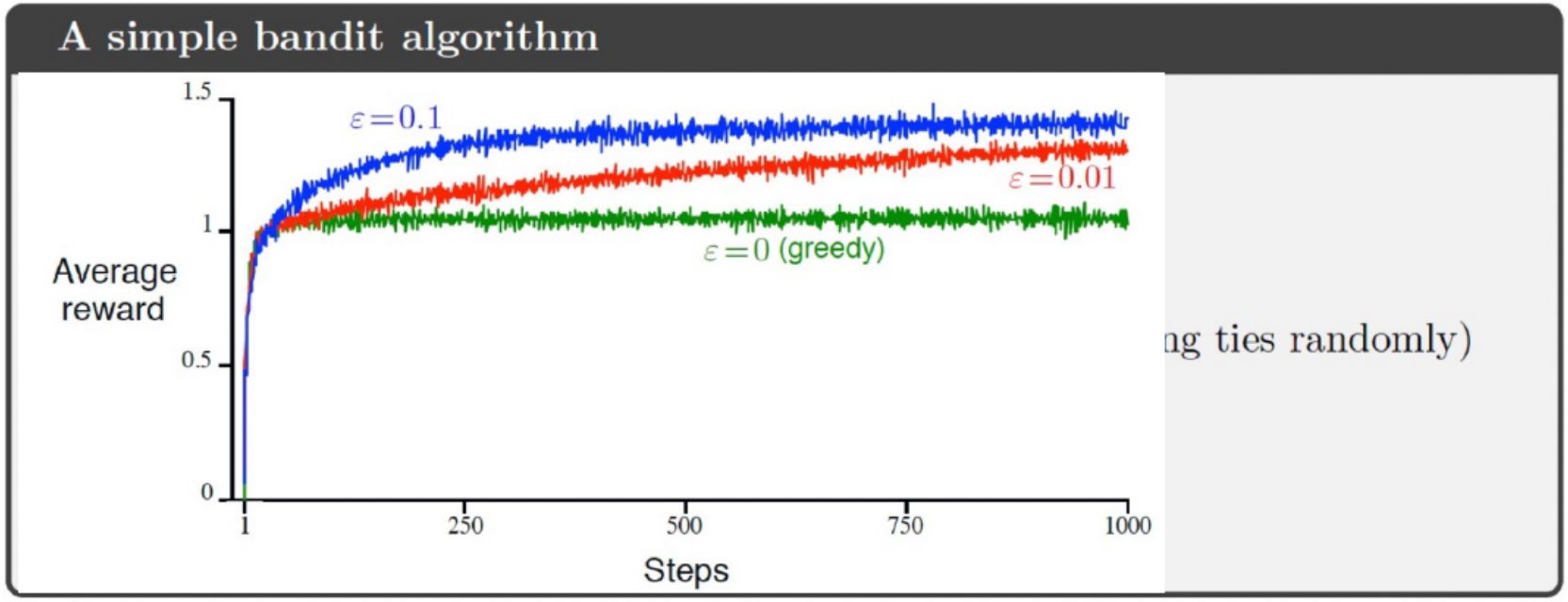
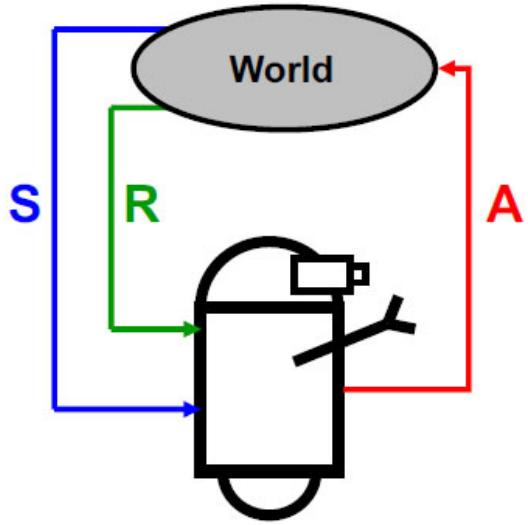


Image from Sutton & Barto book

General Reinforcement Learning Framework

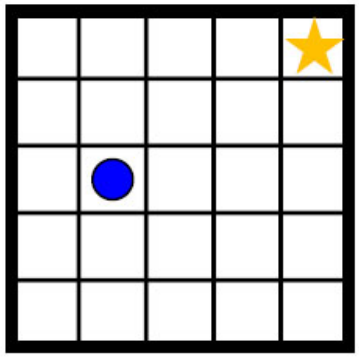
1. Observe state, s_t
2. Decide on an action, a_t
3. Perform action
4. Observe new state, s_{t+1}
5. Observe reward, r_{t+1}
6. Learn from experience
7. Repeat



Goal: Find a control policy that will maximize the observed rewards over the lifetime of the agent

A Canonical Example: Gridworld

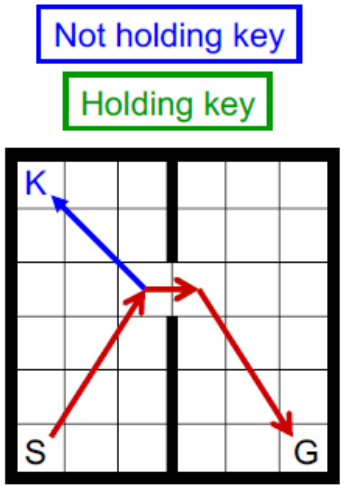
- States are grid cells
- 4 actions: N, S, E, W
- Suppose we want the agent to get to the top-right corner, as fast as possible.
- What should our rewards be?



The Markov Property

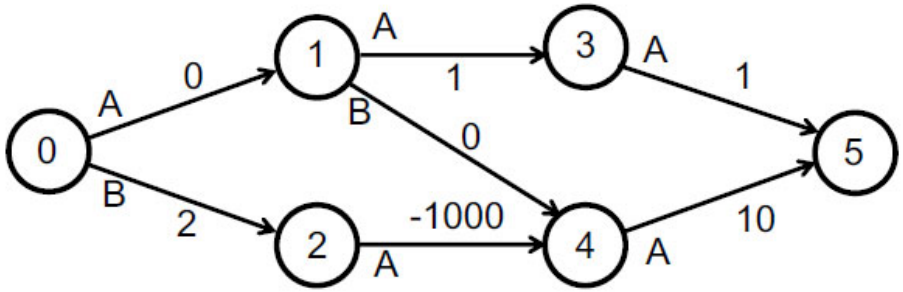
- RL needs a set of states that are Markov
  - Everything you need to know to act is included in the state
  - Not allowed to consult the past

- Rule-of-thumb
  - If you can calculate the reward function from the state without any additional information, you're OK



Markov Decision Processes

- Imagine drawing a graph of all possible states of the environment
- An edge connects states s_1 to s_2 if a single action connects them
  - Each decision affects subsequent decisions!
- Each action also provides a (possibly zero-value) reward
- This is formally modeled by a Markov Decision Process (MDP)



Markov Decision Processes

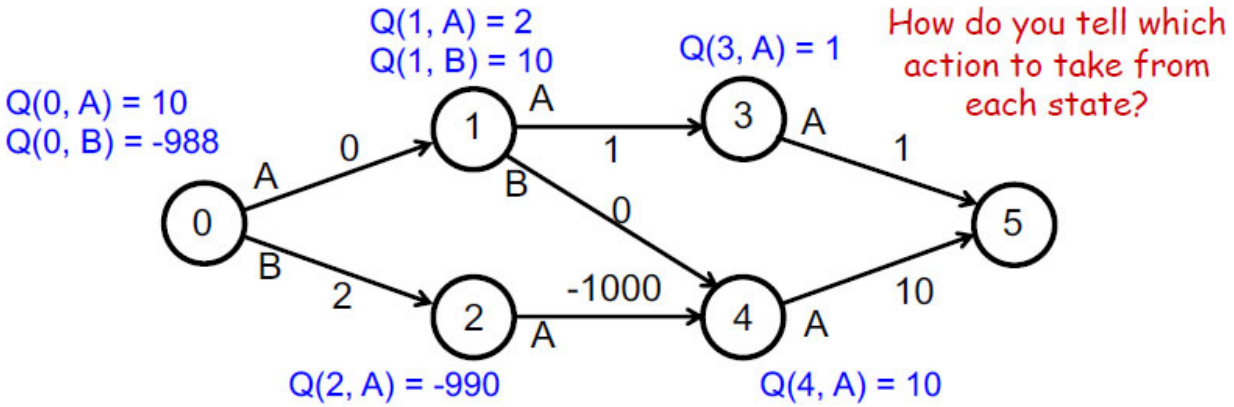
- Formally, a MDP is
  - A set of states, $S = \{s_1, s_2, \ldots, s_n\}$
  - A set of actions, $A = \{a_1, a_2, \ldots, a_m\}$
  - A reward function, $R: S \times A \times S \rightarrow \Re$
  - A transition function,
    - Sometimes $T: S \times A \rightarrow S$

$$P_{ij}^{a} = P(s_{t+1} = j \mid s_t = i, a_t = a)$$

- We want to learn a policy, $\pi: S \rightarrow A$
  - Maximize sum of rewards we see over our lifetime

Q-Functions

- Define value without specifying the policy
  - Specify the value of taking action A from state S and then performing optimally, thereafter



Reinforcement Learning

- What happens if we don't have the whole MDP?
  - We know the states and actions
  - We don't have the system model (transition function) or reward function
- We're only allowed to sample from the MDP
  - Can observe experiences (s, a, r, s')
  - Need to perform actions to generate new experiences
- This is the general RL problem
  - Sometimes called Approximate Dynamic Programming (ADP)

Q-Learning

- Q-learning iteratively approximates the state-action value function, Q
  - We won't estimate the MDP directly
  - Learns the value function and policy simultaneously

- Keep an estimate of Q(s, a) in a table
  - Update these estimates as we gather more experience
  - Estimates do not depend on exploration policy
  - Q-learning is an off-policy method

Q-Learning Algorithm

Q-learning (off-policy TD control) for estimating $\pi \approx \pi_*$

Algorithm parameters: step size $\alpha \in (0, 1]$, small $\varepsilon > 0$
Initialize $Q(s, a)$, for all $s \in \mathcal{S}^+, a \in \mathcal{A}(s)$, arbitrarily except that $Q(terminal, \cdot) = 0$

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

$0 \leq \alpha \leq 1$ is the learning rate & we should decay α, just like in TD

The Update Formula

$Q(s, a) \leftarrow Q(s, a) + \alpha\ (R + \gamma\ \max_{a'} Q(s', a') - Q(s, a))$

This can be written another way…

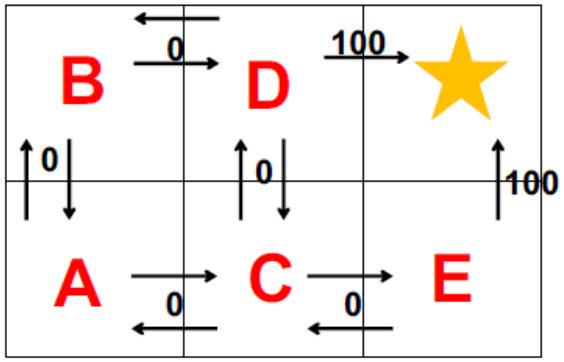
$Q(s, a) \leftarrow (1- \alpha)\ Q(s, a) + \alpha\ (R + \gamma\ \max_{a'} Q(s', a'))$

Looked at this way, it is more obvious that α controls whether we value past experience more or new experience more.

Q-Learning

- Q-learning, learns the expected utility of taking a particular action **a** in state **s**

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

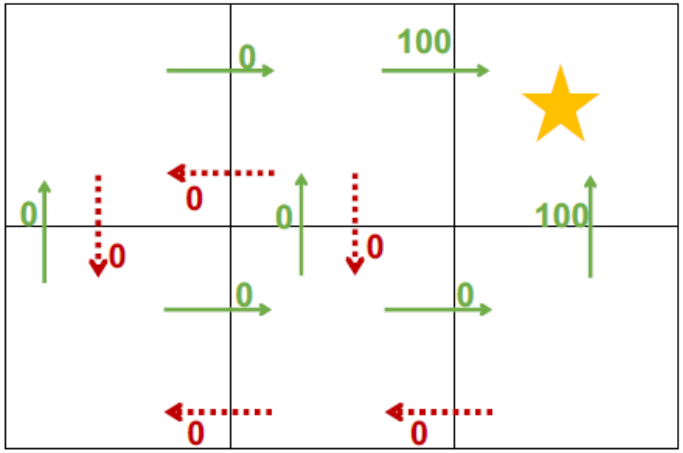


r (state, action)
immediate reward values

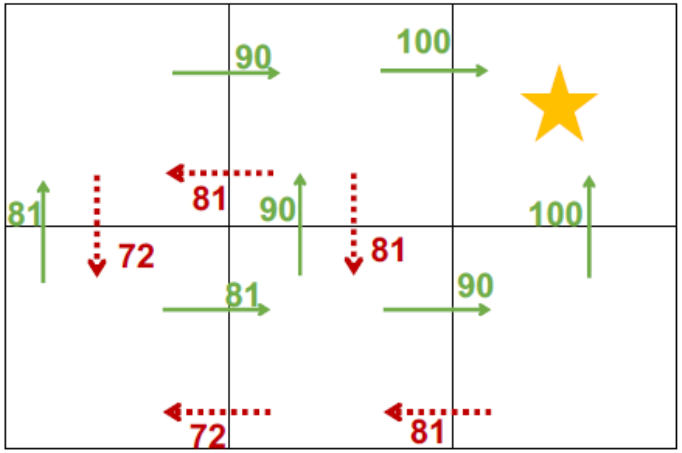
State	North	South	East	West
A		■		■
B	■			■
C		■		
D	■			
E		■	■	

Q-Learning

Q-learning learns the expected utility of taking a particular action **a** in state **s**



r(state, action)
immediate reward values



Q(state, action) values

Q-Learning

- Iteratively approximates the state-action value function, Q
  - We won't estimate the MDP directly
  - Learns the value function and policy simultaneously

- Keep an estimate of Q(s, a) in a table
  - Update these estimates as we gather more experience
  - Estimates do not depend on exploration policy
  - Q-learning is an off-policy method

On-Policy vs. Off-Policy

- On-policy algorithms
  - Final policy is influenced by the exploration policy
  - Generally, the exploration policy needs to be "close" to the final policy
  - Can get stuck in local maxima

- Off-policy algorithms
  - Final policy is independent of exploration policy
  - Can use arbitrary exploration policies
  - Will not get stuck in local maxima

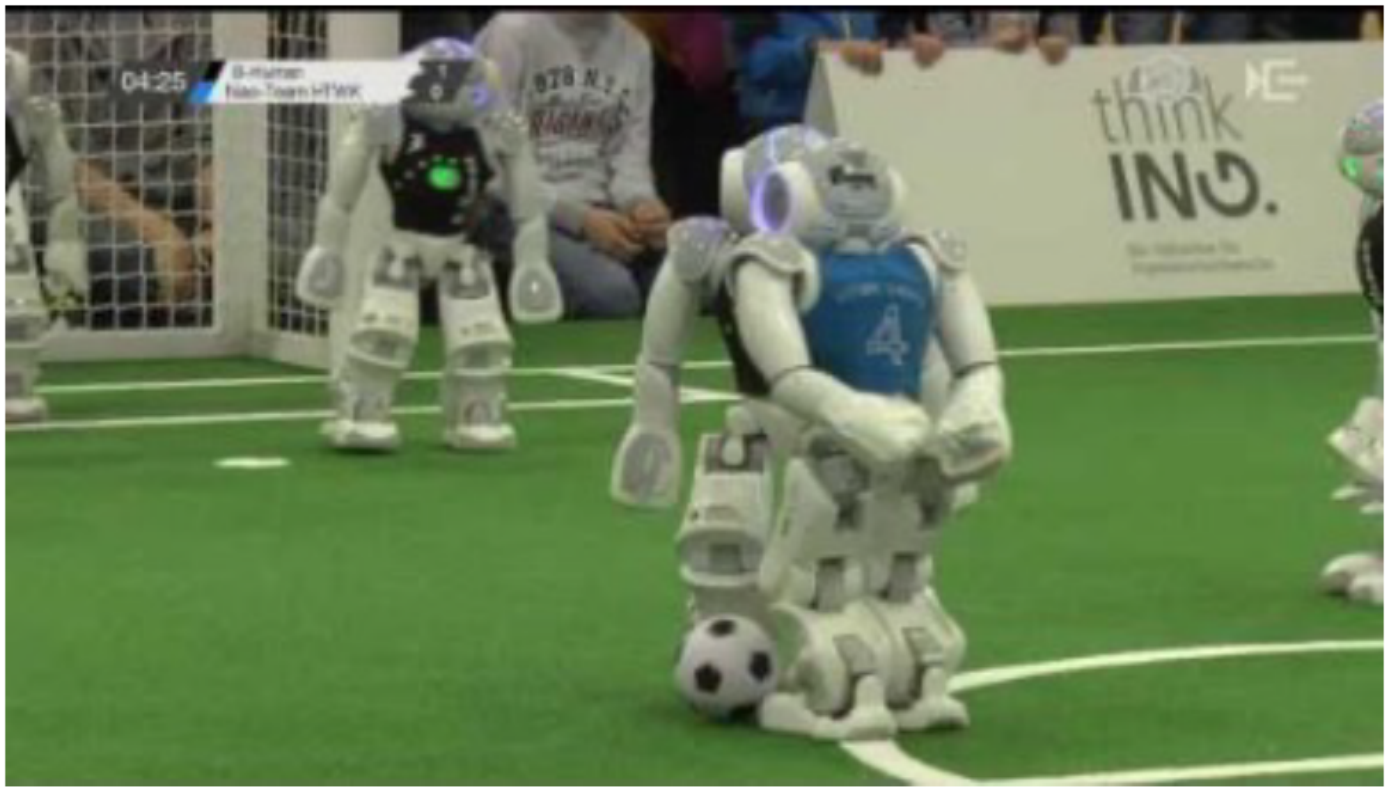
Given enough experience

Convergence Guarantees

- The convergence guarantees for RL are "in the limit"
  - The word "infinite" crops up several times

- Don't let this put you off
  - Value convergence is different than policy convergence
  - We're more interested in policy convergence
  - If one action is significantly better than the others, policy convergence will happen relatively quickly

RoboCup: Combining Many Tasks

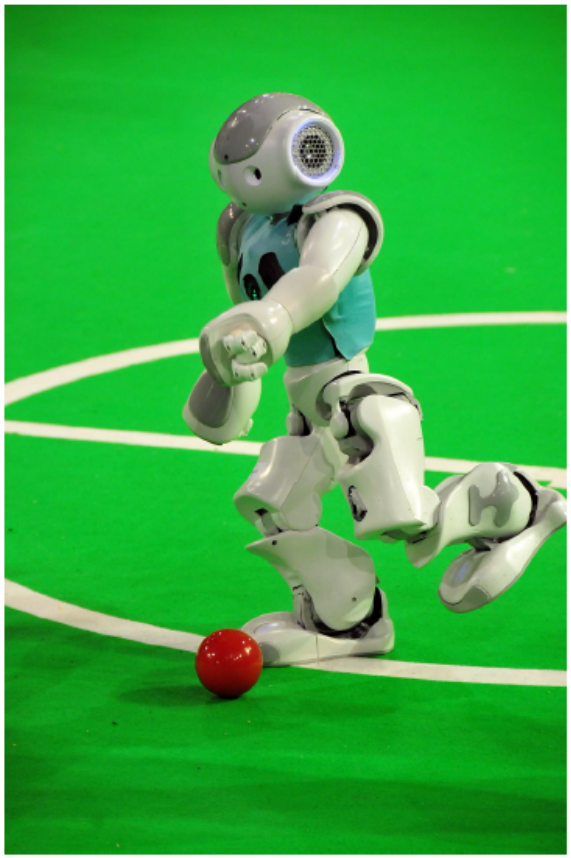


https://youtu.be/xkoXeF9oVH4

RoboCup: Combining Many Tasks

Learning to Walk

- RoboCup legged league
  - Walking quickly is a *big* advantage
- Robots have a parameterized gait controller
  - Multiple REAL VALUED parameters
  - Controls step length, height, etc.
- Robots walk across soccer pitch and are timed
  - Reward is a function of the time taken

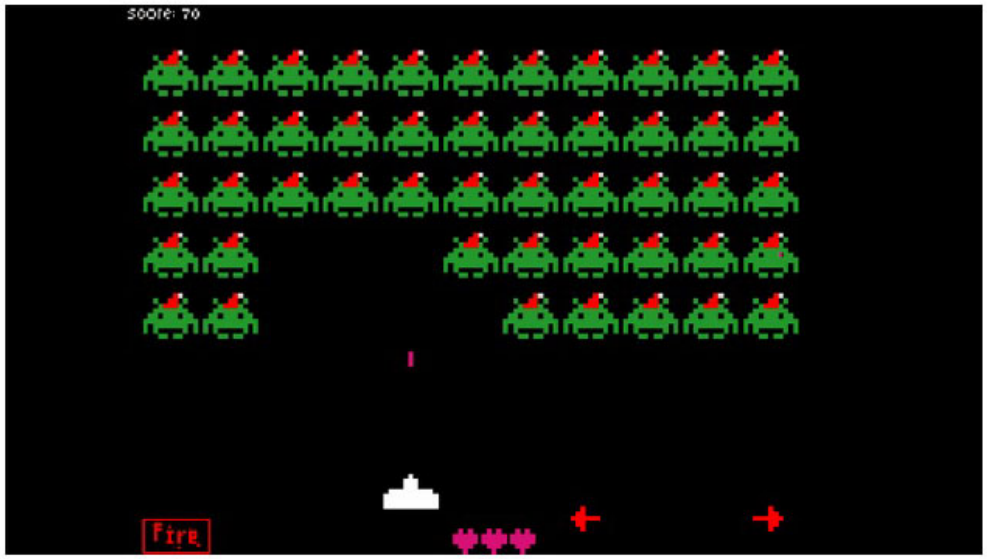


State Space & Combinatorics

- The idea is to learn a probability distribution over the set of actions possible at each state
- We've assumed that there is a table of states and actions
- How big could such a table get?

State Space & Combinatorics

Complex state spaces



Mnih, V., Kavukcuoglu, K., Silver, D., Graves, A., Antonoglou, I., Wierstra, D., & Riedmiller, M. (2013). Playing atari with deep reinforcement learning. *arXiv preprint arXiv:1312.5602*.

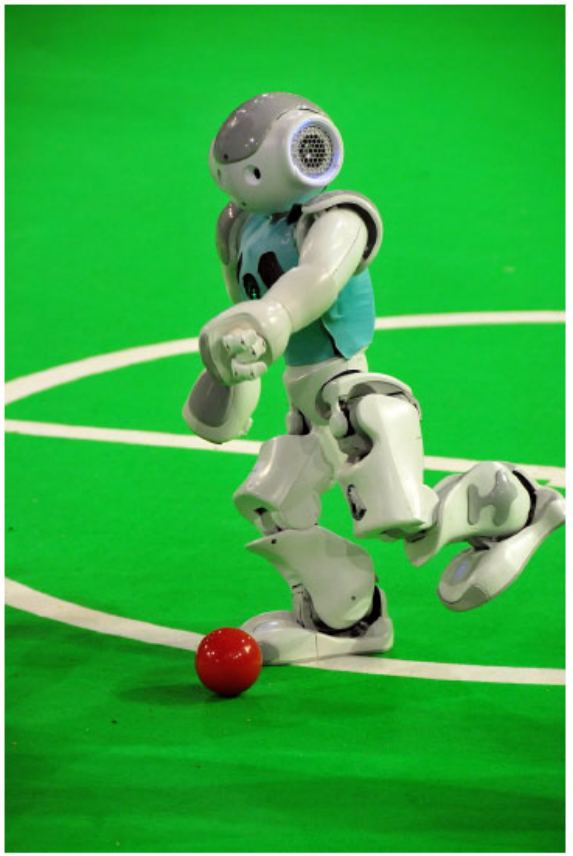


Image courtesy of: Ralf Roletschek
https://www.wikidata.org/wiki/Q15080600

Replace Table With…

- …a parameterized function that can output the policy
- Historically, this could be any function
- These days it means…you guessed it…a deep net
- ...and this also means we need differentiable policy gradient

Vanilla Policy Gradients

- Define a policy $\pi_\theta(a \mid s)$
  - Uses parameters θ to map from state s to action a
  - This is a distribution over the action space, from which we sample one action
- Take actions according to this policy:
  - Start in state s_0, sample action a_0 from $\pi_\theta(a \mid s_0)$, get reward r_0, arrive in s_1
  - From state s_1, sample action a_1 from $\pi_\theta(a \mid s_1)$, etc.
- How do we compute a gradient with respect to θ?

Vanilla Policy Gradients

- How do we compute a gradient with respect to θ?
  - At each timestep, agent was in state s_t and policy gave us $\pi_\theta(a \mid s_t)$
  - We picked an action a_t according to that distribution
  - We also received a (possibly zero) reward r_t
- Our VPG "pseudo-loss" looks like:
  - $R(T) = \sum_{t=0 \ldots T} r_t$
  - $L = \sum_t \log p(a_t \mid s_t; \theta)\, R(T)$
- Can think of r_t as (noisy) labels, but $\pi_\theta(a \mid s_t)$ is differentiable

Vanilla Policy Gradients

- How do we compute a gradient with respect to θ?
  - At each timestep, agent was in state s_t and policy gave us $\pi_\theta(a \mid s_t)$
  - We picked an action a_t according to that distribution
  - We also received a (possibly zero) reward r_t
- Our VPG "pseudo-loss" looks like:
  - $R(T) = \sum_{t=0 \ldots T} r_t$
  - $L = \sum_t \log p(a_t \mid s_t; \theta)\, R(T)$

$$\hat{g} = \frac{1}{|\mathcal{D}|} \sum_{\tau \in \mathcal{D}} \sum_{t=0}^{T} \nabla_\theta \log \pi_\theta(a_t|s_t) R(\tau)$$

- Can think of r_t as (noisy) labels, but $\pi_\theta(a \mid s_t)$ is differentiable

Vanilla Policy Gradients

$$
\begin{aligned}
\nabla_\theta J(\pi_\theta) &= \nabla_\theta \underset{\tau\sim\pi_\theta}{\mathrm{E}}[R(\tau)] \\
&= \nabla_\theta \int_\tau P(\tau|\theta)R(\tau) && \text{Expand expectation} \\
&= \int_\tau \nabla_\theta P(\tau|\theta)R(\tau) && \text{Bring gradient under integral} \\
&= \int_\tau P(\tau|\theta)\nabla_\theta \log P(\tau|\theta)R(\tau) && \text{Log-derivative trick} \\
&= \underset{\tau\sim\pi_\theta}{\mathrm{E}}[\nabla_\theta \log P(\tau|\theta)R(\tau)] && \text{Return to expectation form} \\
\therefore \nabla_\theta J(\pi_\theta) &= \underset{\tau\sim\pi_\theta}{\mathrm{E}}\left[\sum_{t=0}^{T} \nabla_\theta \log \pi_\theta(a_t|s_t)R(\tau)\right] && \text{Expression for grad-log-prob}
\end{aligned}
$$

https://spinningup.openai.com/en/latest/spinningup/rl_intro3.html#deriving-the-simplest-policy-gradient

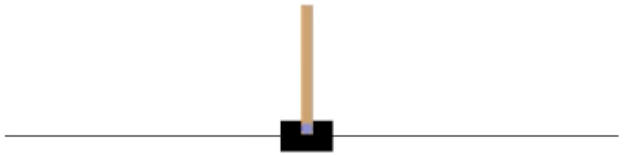
Policy Gradient Methods

- This results in hill-climbing in policy space
  - Has all the problems of hill-climbing (local maxima != global maximum)
  - But we can also use tricks like random restarts, momentum, etc.

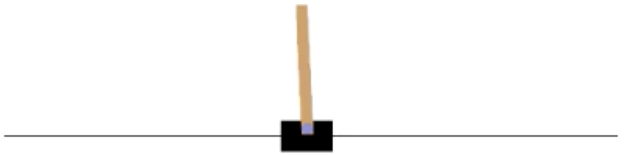
- This is a good approach if you have a parameterized policy
  - Typically faster than value-based methods (e.g., Q-Learning)
  - "Safe" exploration, if you have a good policy
  - Learns locally-best parameters *for that policy*

Policy Gradient Notebook

10 Epochs



100 Epochs



Another nice guide at: https://spinningup.openai.com/en/latest/algorithms/vpg.html

Back to Q-Learning

Q-learning (off-policy TD control) for estimating $\pi \approx \pi_*$

Algorithm parameters: step size $\alpha \in (0, 1]$, small $\varepsilon > 0$
Initialize $Q(s, a)$, for all $s \in \mathcal{S}^+, a \in \mathcal{A}(s)$, arbitrarily except that $Q(terminal, \cdot) = 0$

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

$0 \leq \alpha \leq 1$ is the learning rate & we should decay α, just like in TD

Note: this formulation is from Sutton & Barto's "Reinforcement Learning"

Back to Q-Learning

$Q(s, a) \leftarrow (1- \alpha)\ Q(s, a) + \alpha\ (R + \gamma\ \max_{a'} Q(s', a'))$

This converges in the limit to "true" $Q^*(s, a)$

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

Going to A Parameterized Model

Take the standard Bellman equation for estimating the Q function:

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

Take a loss function for a parameterized function with target value y_i and parameters θ_i

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

Take the gradient:

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

Deep Q-Learning

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

This Was a Breakthrough

	B. Rider	Breakout	Enduro	Pong	Q*bert	Seaquest	S. Invaders
Random	354	1.2	0	−20.4	157	110	179
Sarsa [3]	996	5.2	129	−19	614	665	271
Contingency [4]	1743	6	159	−17	960	723	268
DQN	**4092**	**168**	**470**	**20**	**1952**	**1705**	**581**
Human	7456	31	368	−3	18900	28010	3690
HNeat Best [8]	3616	52	106	19	1800	920	**1720**
HNeat Pixel [8]	1332	4	91	−16	1325	800	1145
DQN Best	**5184**	**225**	**661**	**21**	**4500**	**1740**	1075

Table 1: The upper table compares average total reward for various learning methods by running an ϵ-greedy policy with $\epsilon = 0.05$ for a fixed number of steps. The lower table reports results of the single best performing episode for HNeat and DQN. HNeat produces deterministic policies that always get the same score while DQN used an ϵ-greedy policy with $\epsilon = 0.05$.

Mnih, V., Kavukcuoglu, K., Silver, D., Graves, A., Antonoglou, I., Wierstra, D., & Riedmiller, M. (2013). Playing atari with deep reinforcement learning. *arXiv:1312.5602*.