Reinforcement Learning

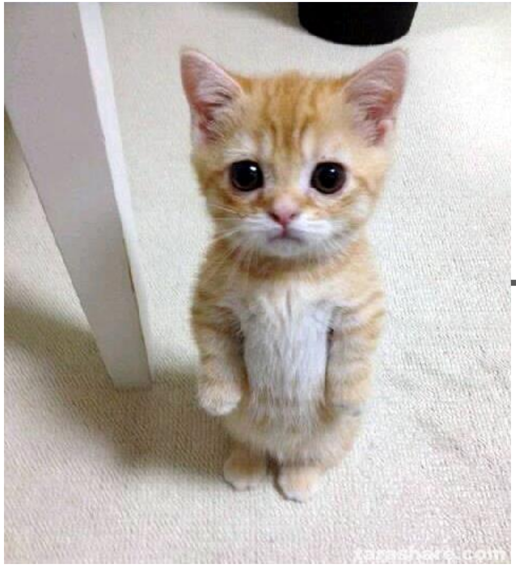
Lecture 14

So far… Supervised Learning

Data: `(x, y)`: `x` is data, `y` is label

Goal: Learn a function to map `x -> y`

Examples: Classification, regression, object detection, image captioning, etc.



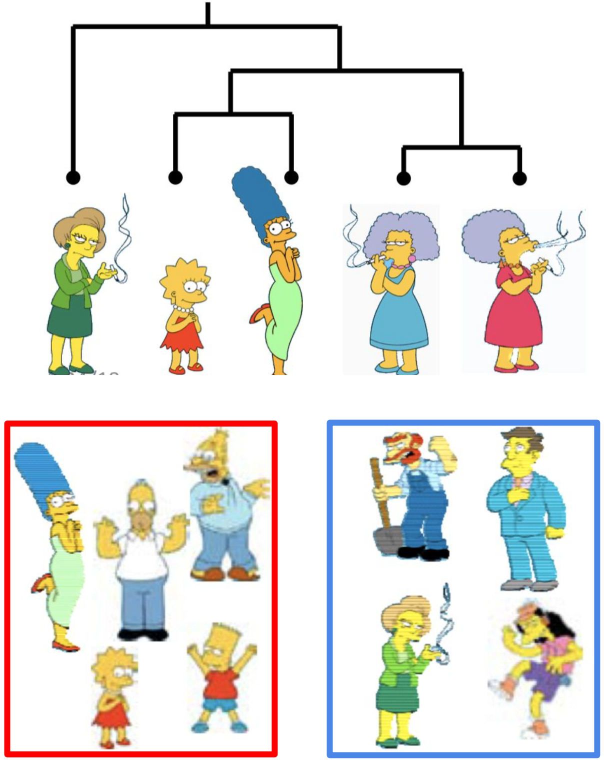
A Cat

So far... Unsupervised Learning

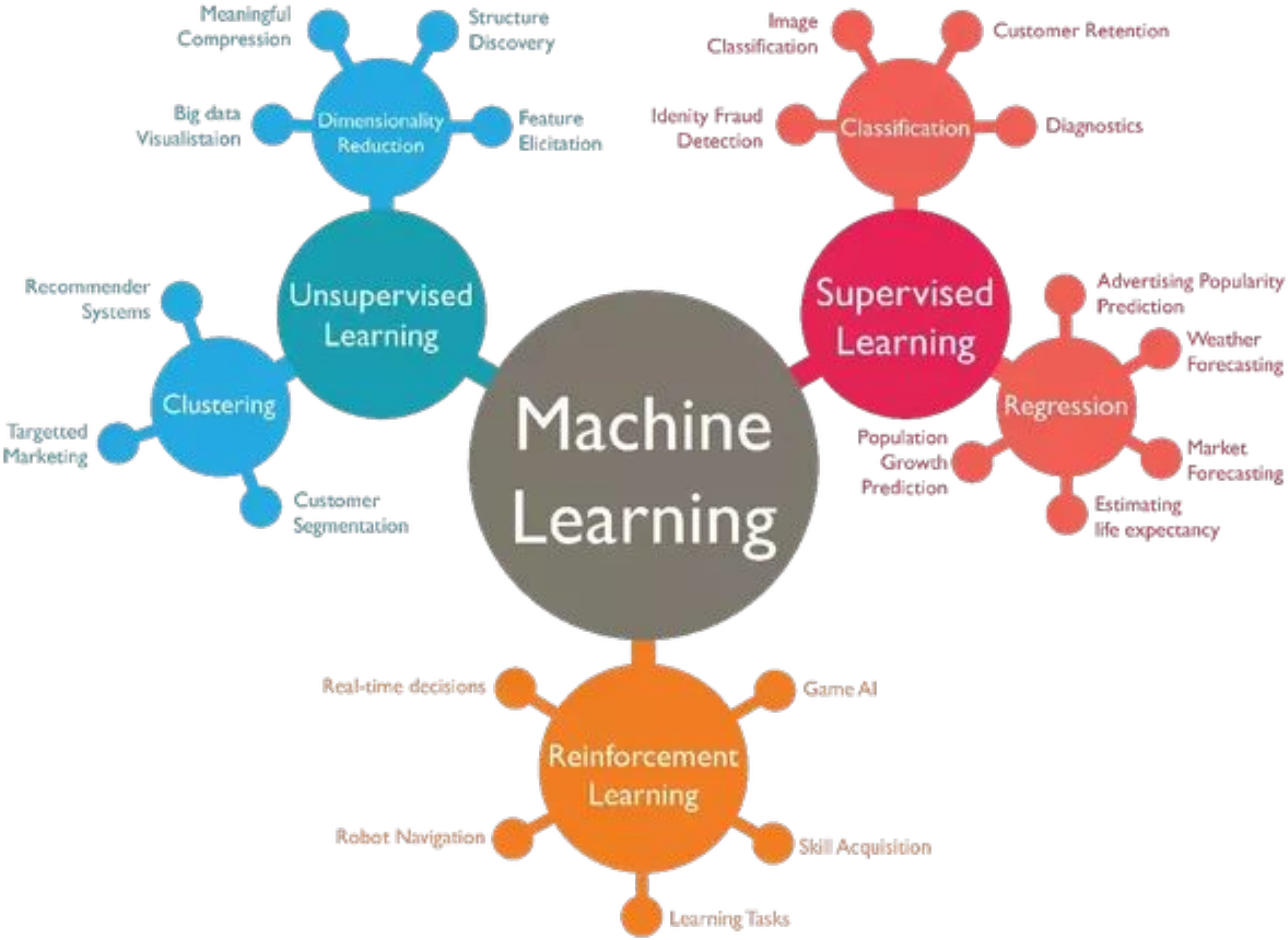
Data: **x** Just data, no labels!

Goal: Learn some underlying hidden structure of the data

Examples: Clustering, dimensionality reduction, feature learning, etc.



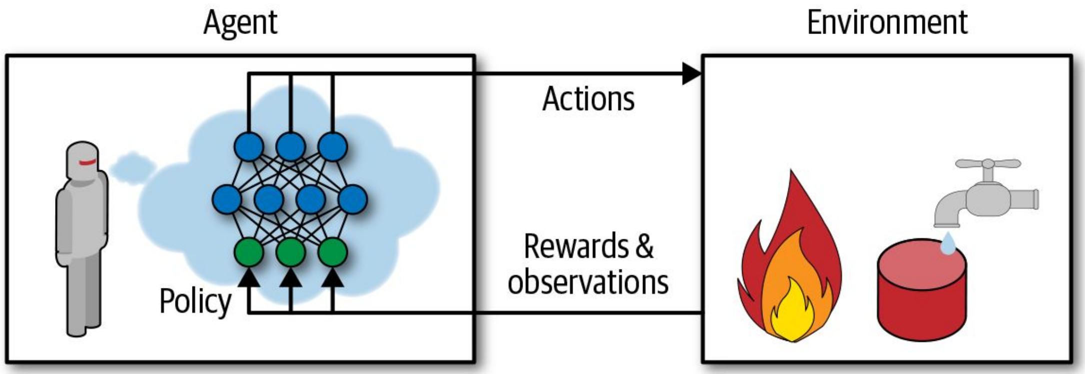
Family **Employees**



Machine Learning
Unsupervised Learning
Dimensionality Reduction
Meaningful Compression
Structure Discovery
Big data Visualistaion
Feature Elicitation
Clustering
Recommender Systems
Targetted Marketing
Customer Segmentation
Supervised Learning
Classification
Image Classification
Customer Retention
Idenity Fraud Detection
Diagnostics
Regression
Advertising Popularity Prediction
Weather Forecasting
Population Growth Prediction
Market Forecasting
Estimating life expectancy
Reinforcement Learning
Real-time decisions
Game AI
Robot Navigation
Skill Acquisition
Learning Tasks

Today: Reinforcement Learning (RL)

Problems involving an **agent** interacting with an **environment**, which provides numeric reward sig



At each step, the agent:

- Executes an **action**
- Observe a new **state**
- Receive some **reward**

Goal: Learn how to take actions from a policy in order to maximize reward

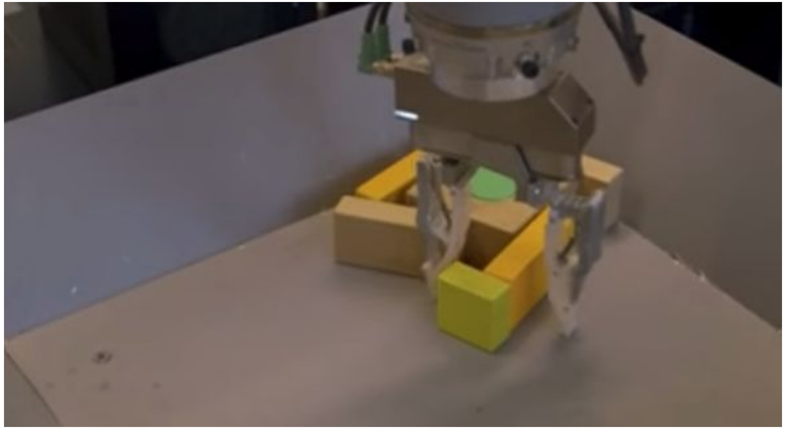
Example: Grasping Objects Problem

Goal: Pick an Object with different shape

State: Raw pixels from camera

Actions: Move arm, grasp

Reward: positive when pickup is successful



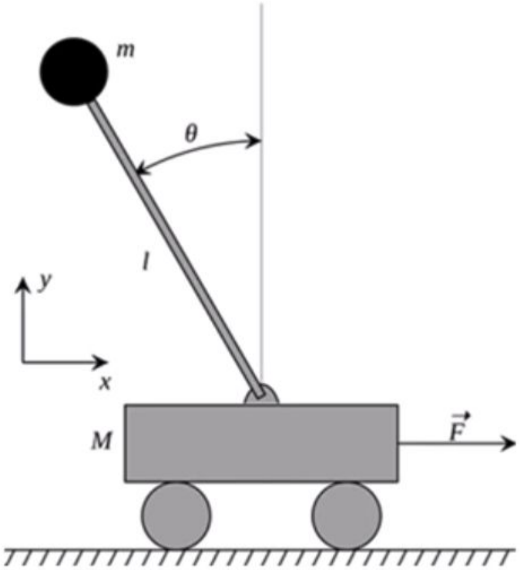
Example: Cart-Pole Balancing Problem

Goal: Balance the pole of top of a moving cart

State: Pole angle, angular speed, cart position, horizontal velocity

Actions: Horizontal force to the cart

Reward: 1 at each time step if the pole is upright



Example: DeepTraffic

Goal: train an RL agent that can successfully navigate through traffic

State: as a grid, where each cell will be the speed of the vehicle inside it

Actions: Accelerate, Break, Left, Right, No action

Reward: positive when high speed is maintained.

Example: College Life

Goal: Survival? Happiness?

State: Sight, hearing, taste, smell, touch, feel

Action: Think, move, speak

Reward: Grades? Money? Love?

Environment and Actions

Fully Observable (Chess) vs. **Partially Observable** (Poker)

Single Agent (Atari) vs. **Multi Agent** (DeepTraffic)

Deterministic (Cart Pole) vs. **Stochastic** (DeepTraffic)

Static (Chess) vs. **Dynamic** (DeepTraffic)

Discrete (Chess) vs. **Continuous** (Cart Pole)

RL in Humans

Humans appear to learn to walk through "very few examples" of trial and error.

"**How** we learn how to walk" is an open question…some possible answers:

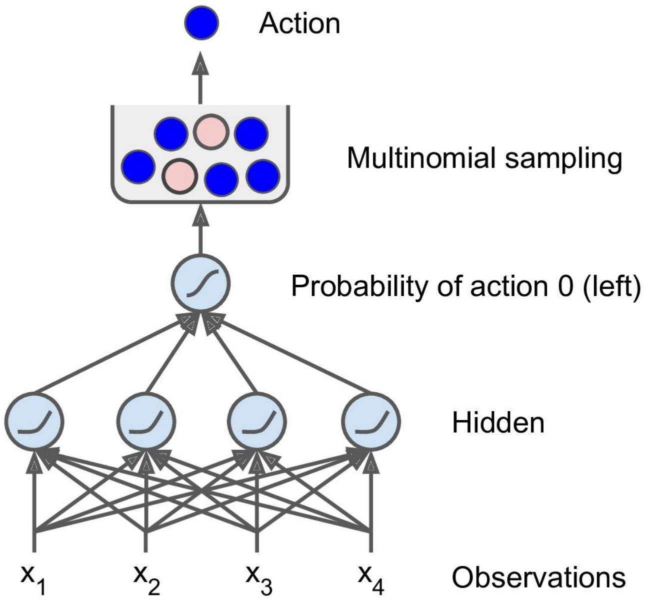
- **Hardware**: 230M years of bipedal movement data
- **Imitation Learning**: Observation of other human walking
- **Algorithms**: probably better than backprop and SGD

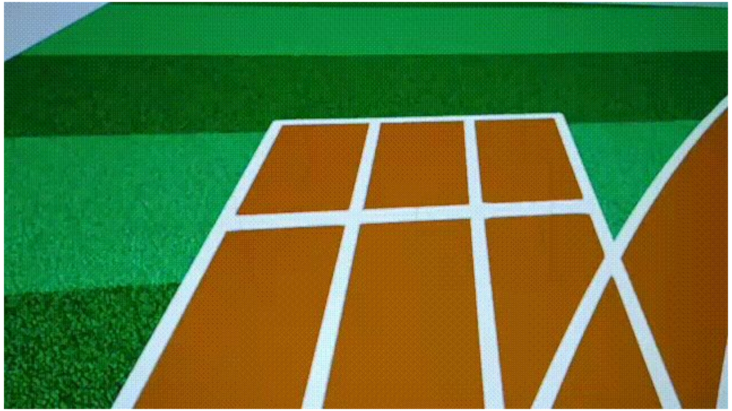
Deep RL

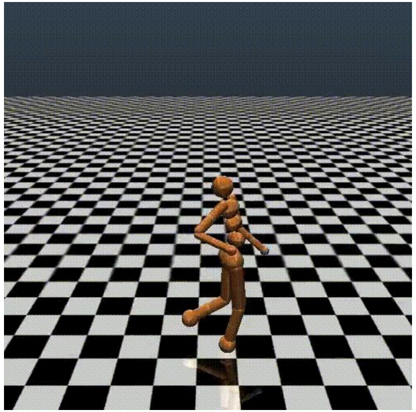
Deep? Deep RL = RL + Neural Networks

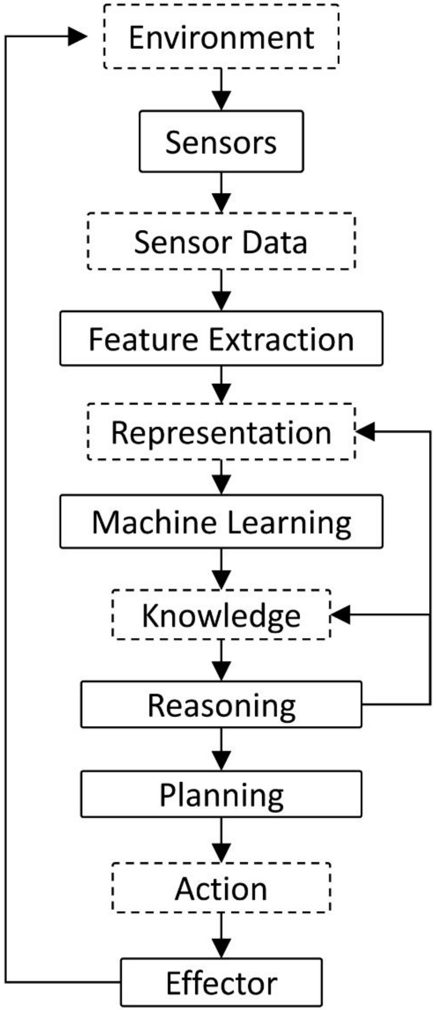
How? Trial and Error in a world that provides occasional rewards

⇒ a framework for learning to solve sequential decision-making problems.





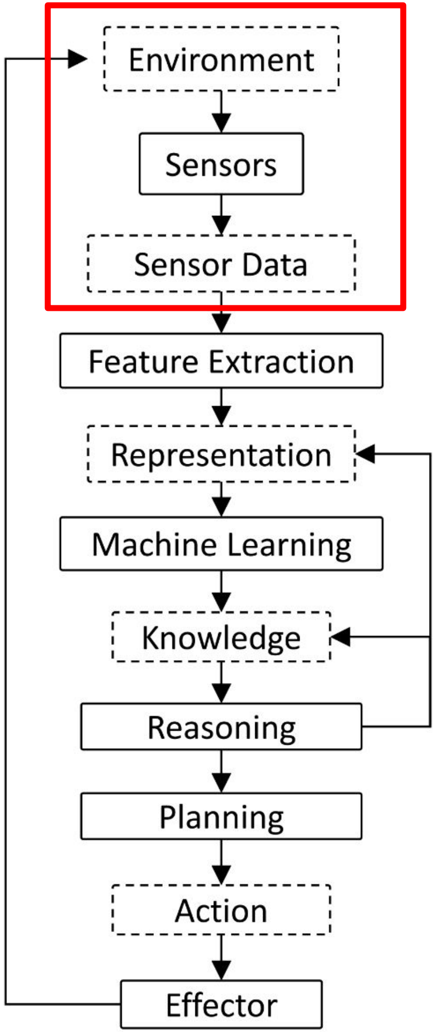




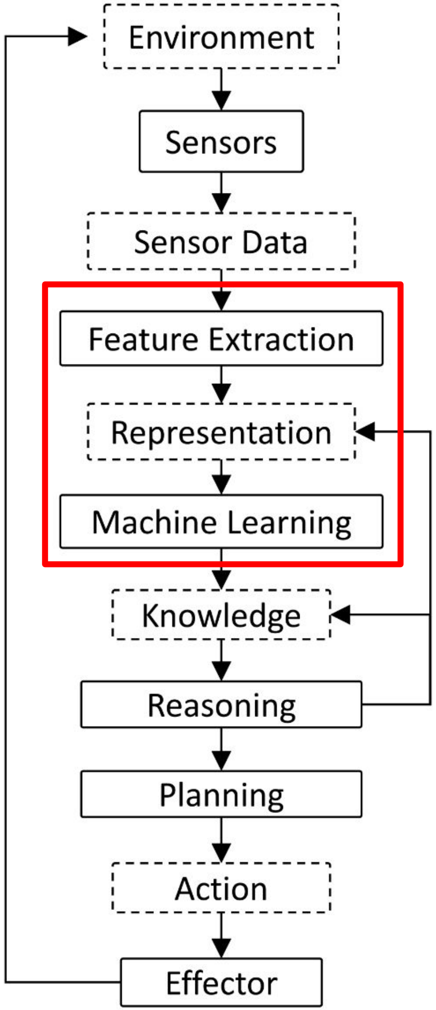
The Continuous Learning Cycle

What can be learned?

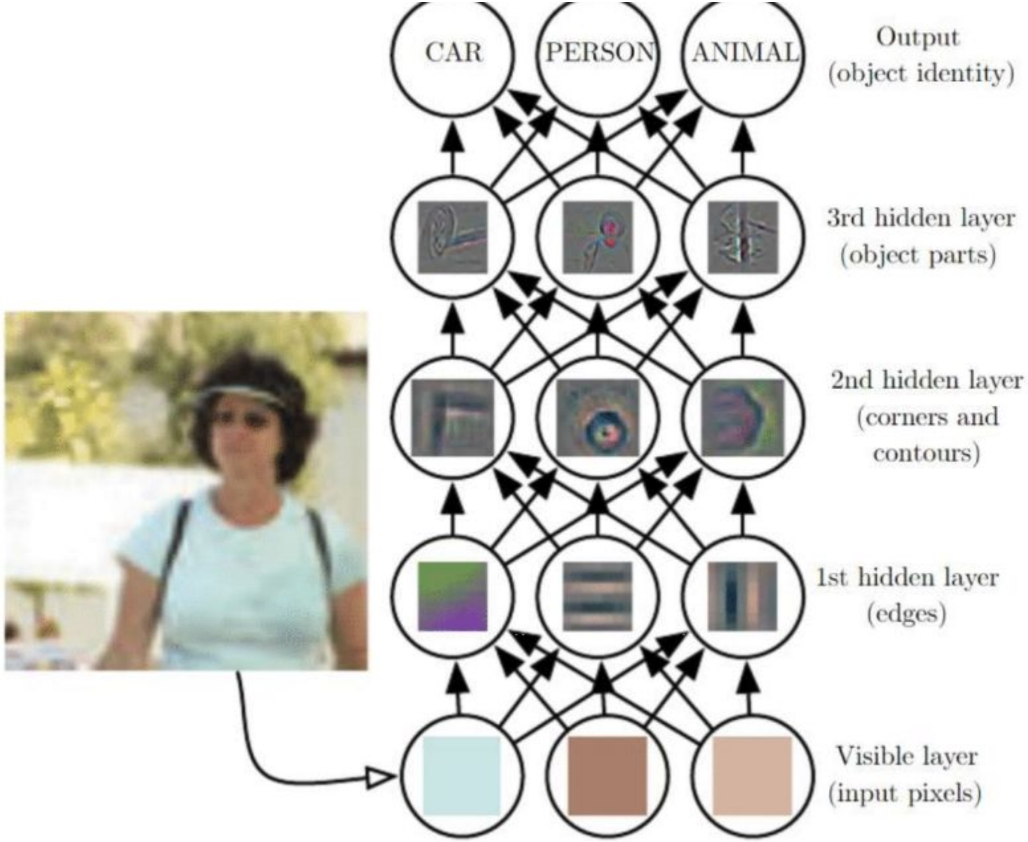
Sensors


Environment
Sensors
Sensor Data
Feature Extraction
Representation
Machine Learning
Knowledge
Reasoning
Planning
Action
Effector


Lidar
Camera
(Visible, Infrared)
Radar
GPS
Stereo Camera
Microphone
Networking
(Wired, Wireless)
IMU



Representations



Knowledge / Reasoning

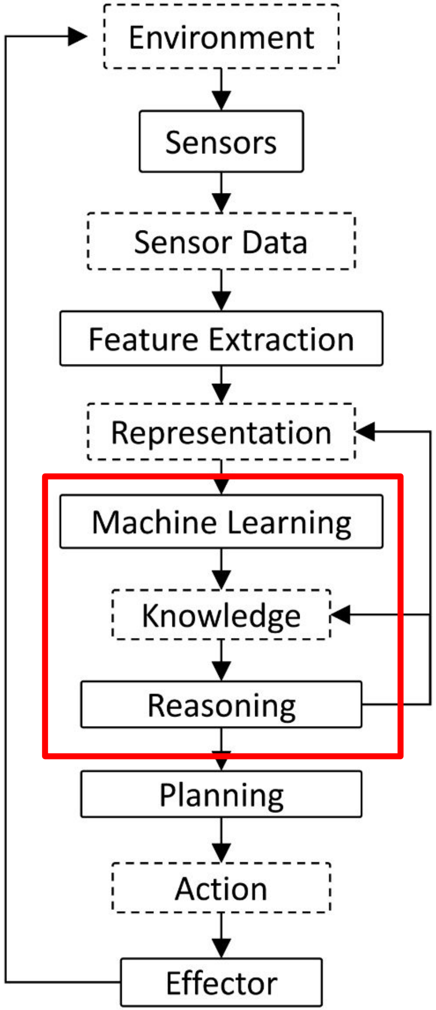
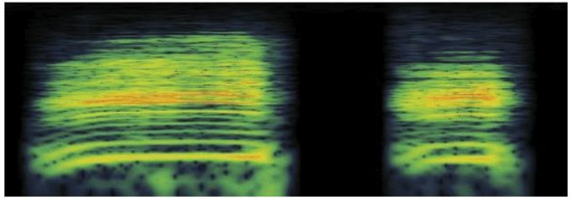
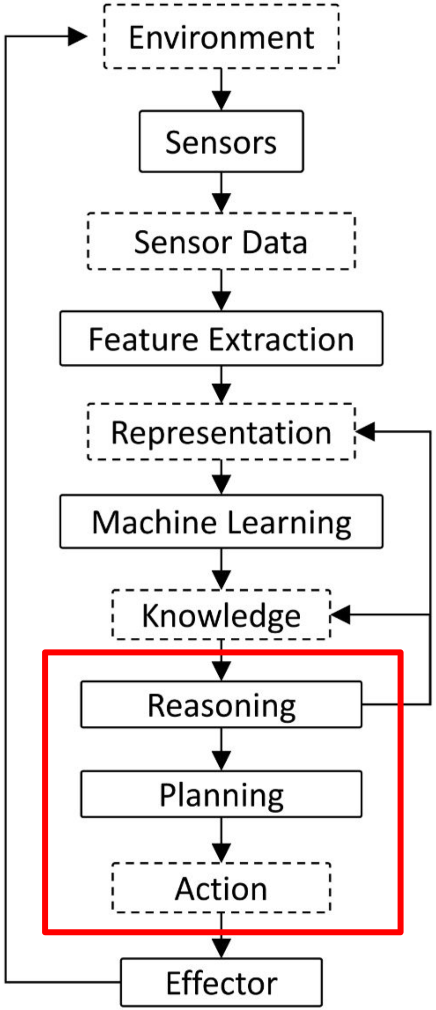


Image Recognition:
If it looks like a duck

Audio Recognition:
Quacks like a duck

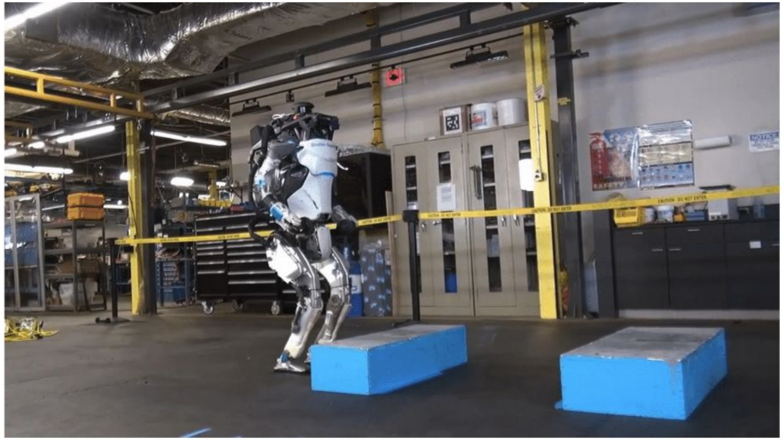


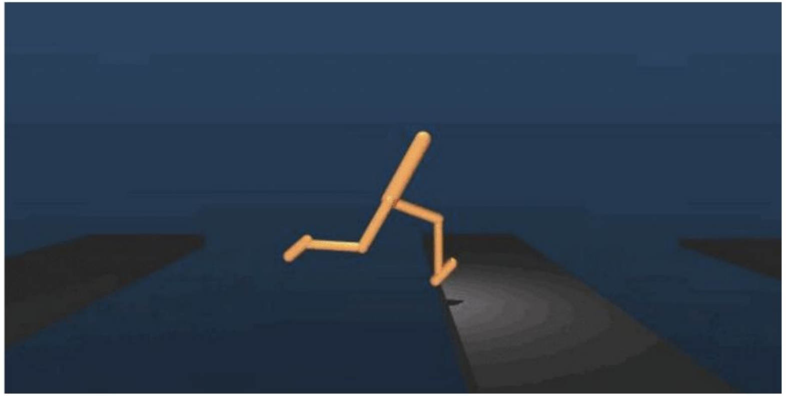
Activity Recognition:
Swims like a duck



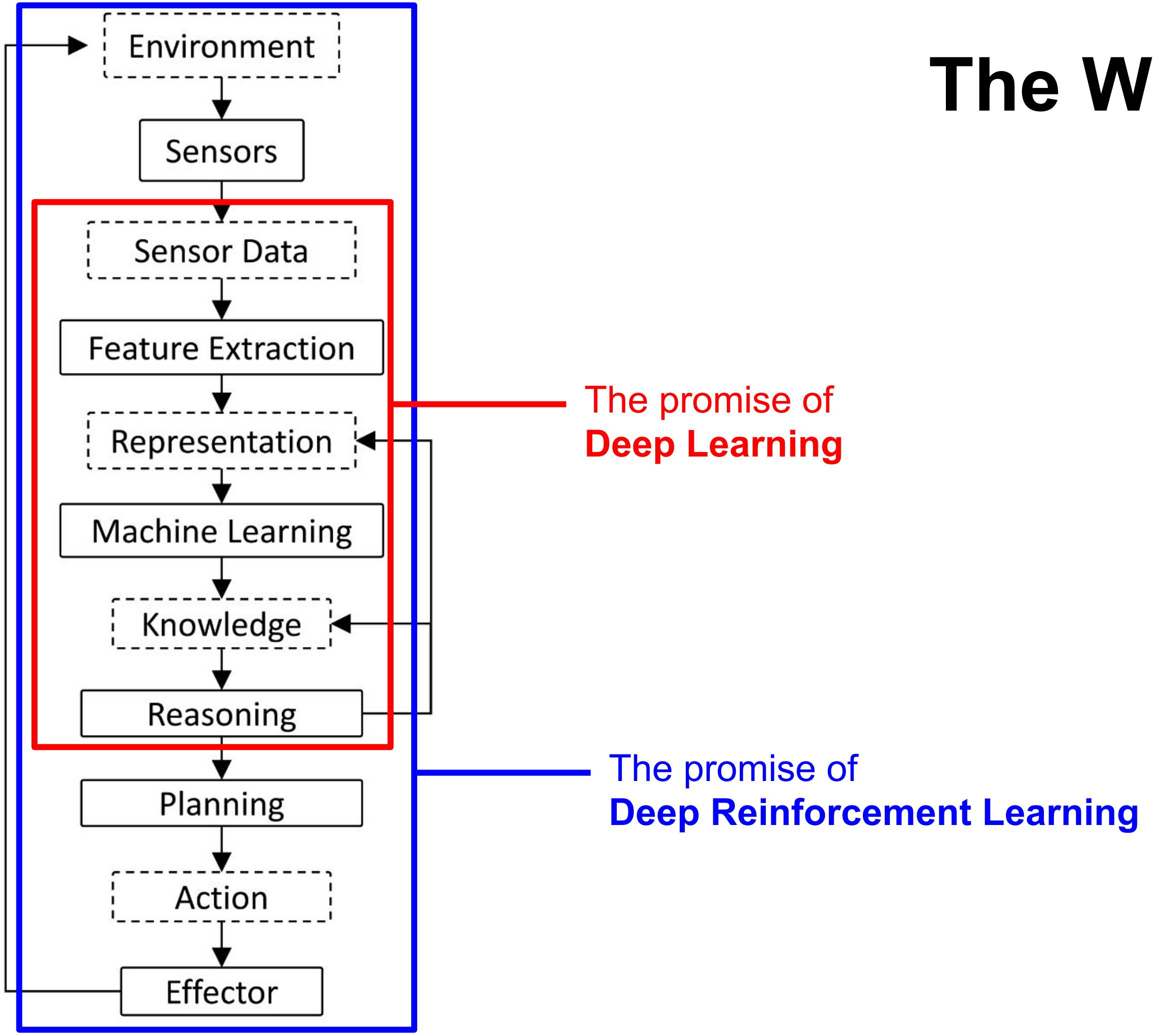
Environment
Sensors
Sensor Data
Feature Extraction
Representation
Machine Learning
Knowledge
Reasoning
Planning
Action
Effector

Actions



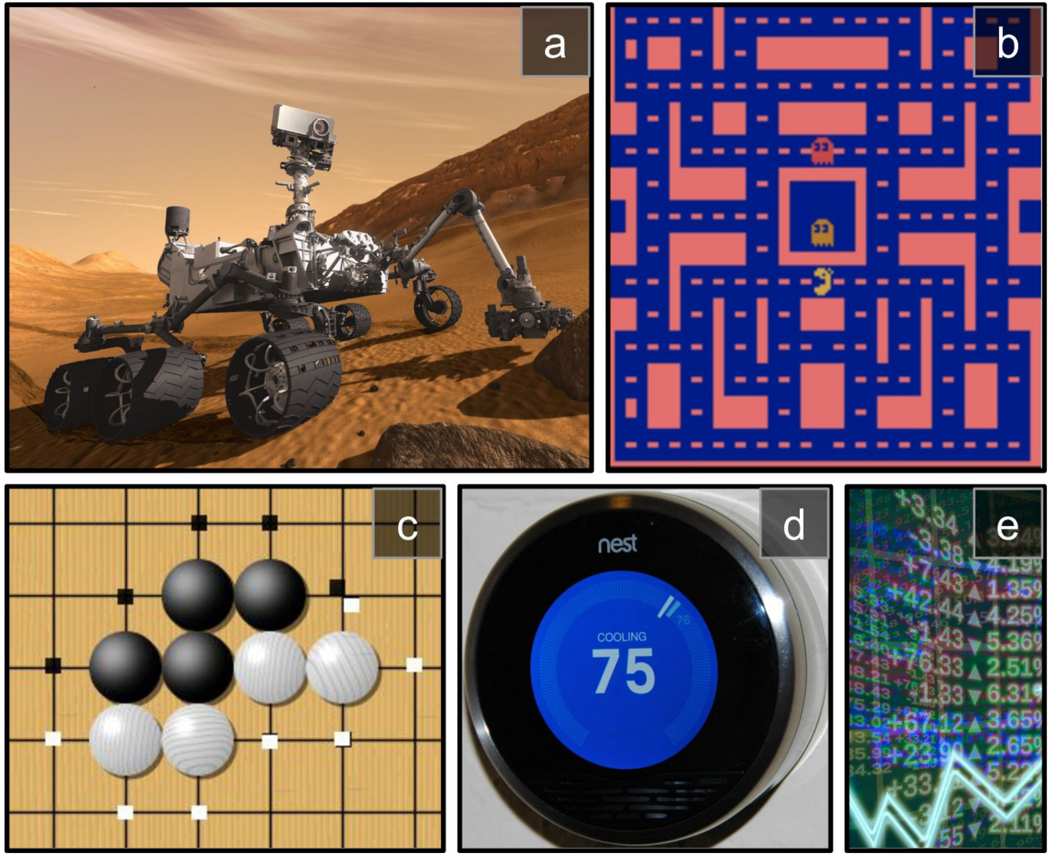


The Whole Cycle



RL Applications

a. Robotics
b. Ms. Pac-man
c. Go player
d. Thermostat
e. Automatic Trader



Major Components of an RL Agent

An RL agent may be directly or indirectly trying to learn an:

- **Policy**: agent's behavior function
- **Value Function**: how good is each state and/or action
- **Model**: agent's representation of the environment

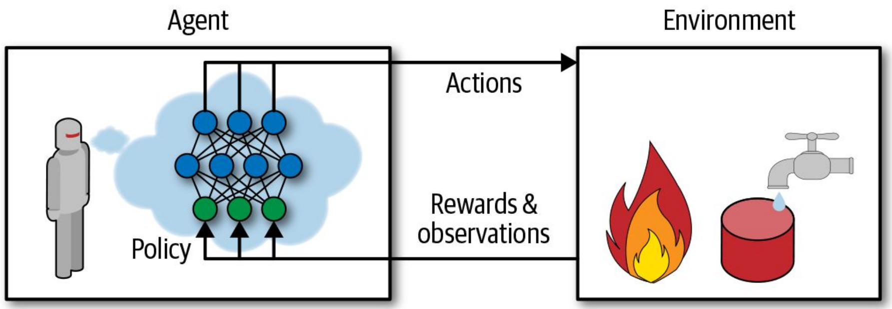
$$s_0, a_0, r_1, s_1, a_1, r_2, \ldots, s_{n-1}, a_{n-1}, r_n, s_n$$

state

action

reward

terminal state



Markov Decision Process

Markov Decision Process is the mathematical formulation for RL problem

Markov property: current state completely characterizes the state of the world.

$$(\mathcal{S}, \mathcal{A}, \mathcal{R}, \mathbb{P}, \gamma)$$

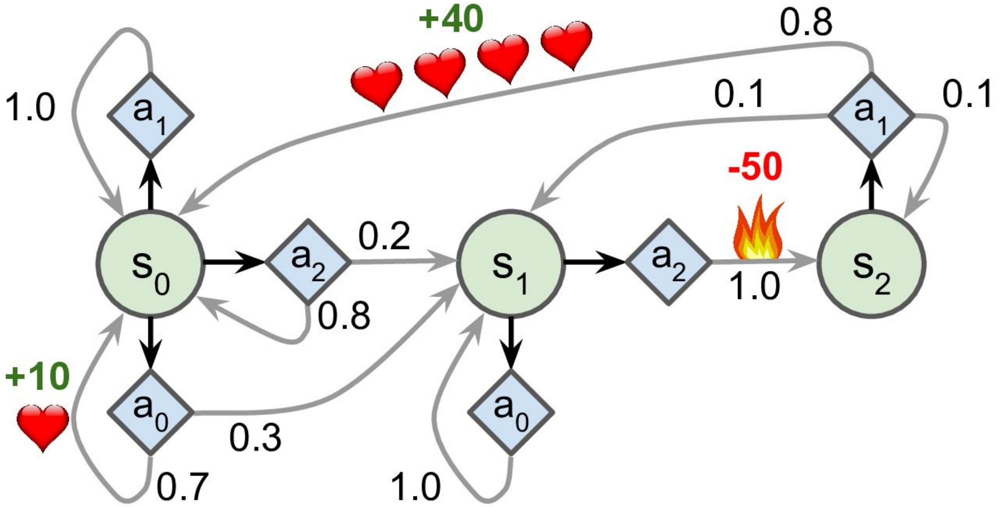
Set of possible states

Set of possible actions

Reward distribution given (state, action)

Transition probability distribution of next state

Discount factor



Markov Decision Process

At time step $t = 0$, environment samples initial state $s_0 \sim \mathbb{P}(s_0)$

Then for $t = 0$ until done:

- Agent selects action a_t
- Environment samples reward $r_t \sim \mathcal{R}(.\,|s_t, a_t)$
- Environment samples next state $s_{t+1} \sim \mathbb{P}(.\,|s_t, a_t)$
- Agent receives the reward r_t and next state s_{t+1}

A policy π is a function from S to A that specifies what action to take in each state

Objective: find policy π^* that maximizes cumulative discounted reward

Maximize Reward

Future Reward: $R_t = r_t + r_{t+1} + r_{t+2} + \ldots + r_n$

Discounted Future Reward: $R_t = r_t + \gamma r_{t+1} + \gamma^2 r_{t+2} + \ldots + \gamma^{n-t} r_n$

A good strategy for an agent would be to always choose an action that maximizes the discounted future reward

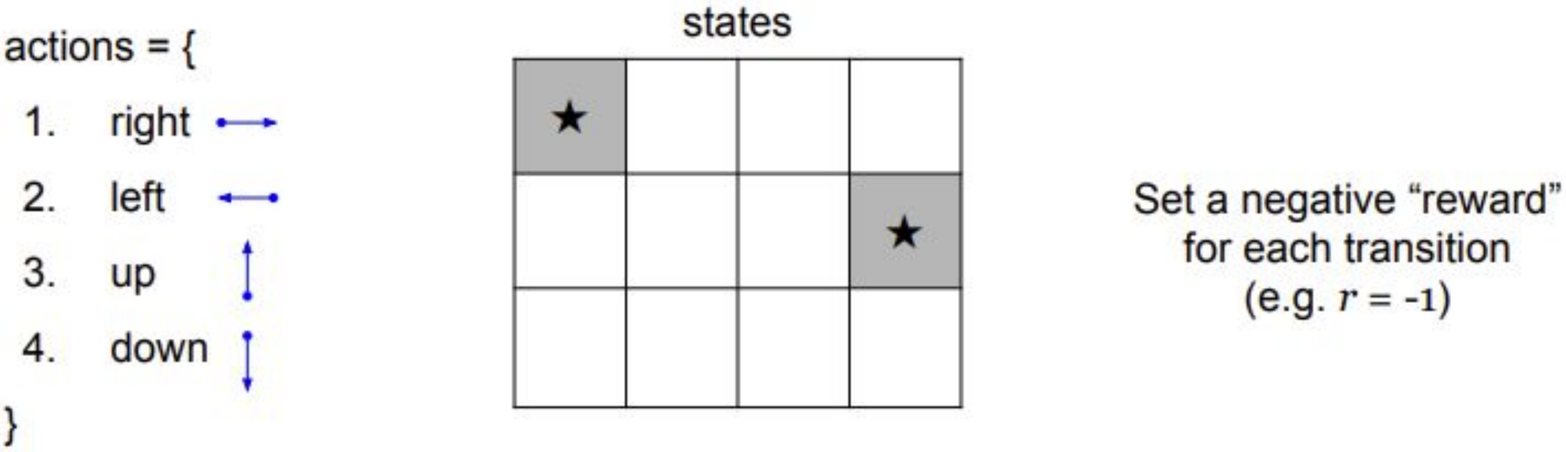
Why?

Uncertainty due the environment, partial observability

Real life example: Either Live it up today, or save $ for tomorrow?

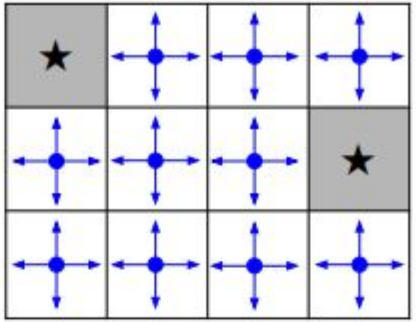
Moving in a grid world

Objective: reach one of the terminal states (stars) using the least number of actions.

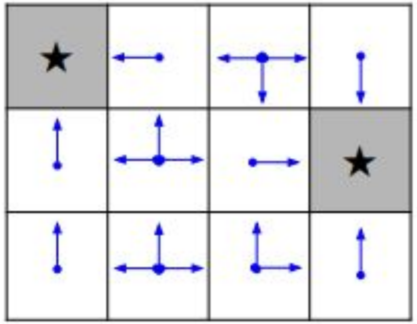


Policy to move in a grid world

Objective: reach one of the terminal states (stars) using the least number of actions.



Random Policy



Optimal Policy

3 Types of Reinforcement Learning

- **Model-based**: Learn the model then use and update it often.
- **Value-based**: Learn the state or state-action value, act by choosing best action, and explore if necessary
- **Policy-based:** Learn the stochastic policy function that maps state to action, act by sampling that policy.

Value-based Method: Q-learning

Solving the Optimal Policy: Q-Learning

State-action value function: $Q_\pi(s,a)$ expected return when starting in s, performing a, and following π

Q-Learning: Use any policy to estimate Q that maximizes future reward:

- Q directly approximates Q* (Bellman optimality equation)
- Independent of the policy being followed
- Only requirement: keep updating each (s,a) pair

$$\mathbf{Q}_{t+1}(s_t, a_t) = \mathbf{Q}_t(s_t, a_t) + \alpha\left(\mathbf{R}_{t+1} + \gamma \max_a \mathbf{Q}_t(s_{t+1}, a) - \mathbf{Q}_t(s_t, a_t)\right)$$

new state **old state** **learning rate** **reward** **discount factor**

Q-Learning: Value Iteration

$$\mathbf{Q}_{t+1}(s_t, a_t) = \mathbf{Q}_t(s_t, a_t) + \alpha\left(\mathbf{R}_{t+1} + \gamma \max_a \mathbf{Q}_t(s_{t+1}, a) - \mathbf{Q}_t(s_t, a_t)\right)$$

new state

old state

learning rate

reward

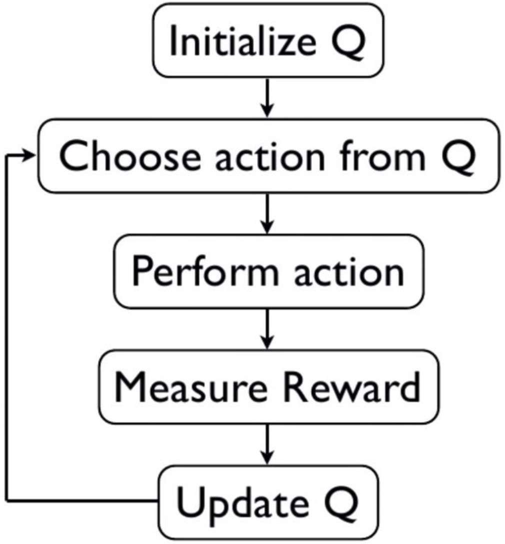
discount factor

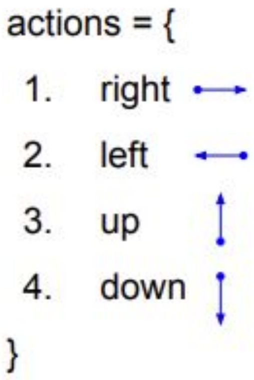
	A1	A2	A3	A4
S1	+1	+2	-1	0
S2	+2	0	+1	-2
S3	-1	+1	0	-2
S4	-2	0	+1	+1

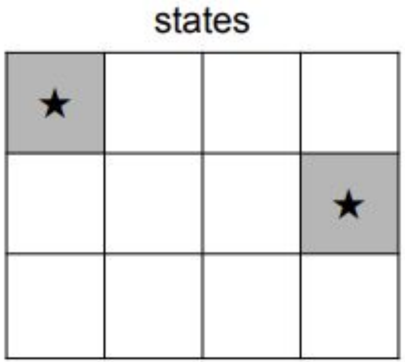
```
initialize Q[num_states,num_actions] arbitrarily
observe initial state s
repeat
      select and carry out an action a
      observe reward r and new state s'
      Q[s,a] = Q[s,a] + α(r + γ max_a' Q[s',a'] - Q[s,a])
      s = s'
until terminated
```

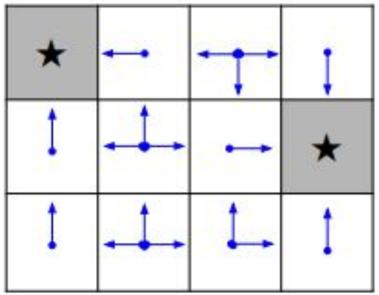
Example: Moving in a grid world

Objective: reach one of the terminal states (stars) using the least number of actions.









Optimal Policy

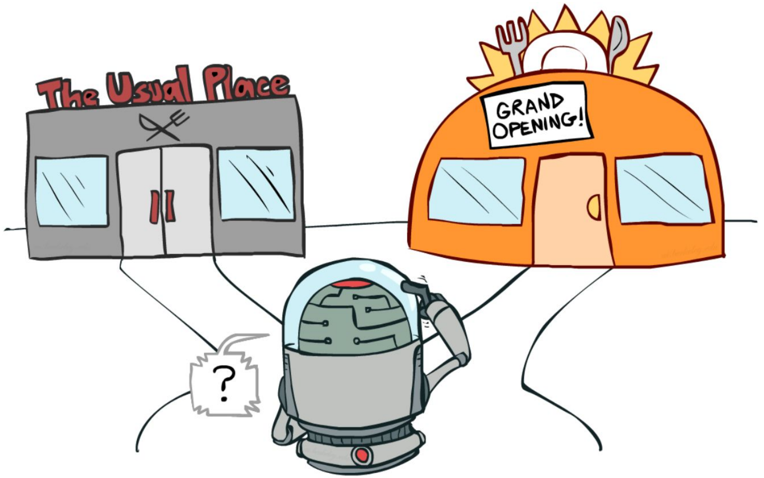
Exploration vs. Exploitation

Deterministic/greedy policy won't explore all actions

- Don't know anything about the environment at the beginning
- Need to try all actions to find the optimal one

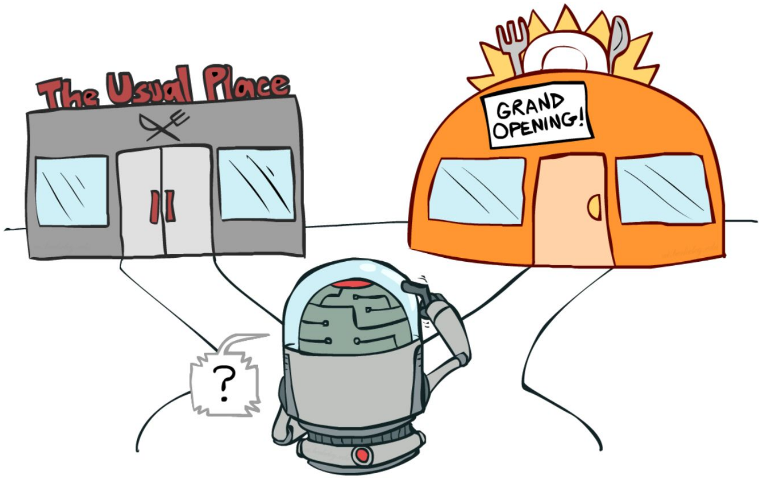
ε-greedy policy

- With probability 1-ε perform the greedy action, otherwise random action
- Slowly move toward greedy policy: ε → 0



Exploration vs. Exploitation Examples

- **Restaurant Selection**
  - Exploitation: Go to your favourite restaurant
    Exploration: Try a new restaurant Online
- **Banner Ads**
  - Exploitation: Show the most successful ads
    Exploration: Show a different ads
- **Oil Drilling**
  - Exploitation: Drill at the best known location
    Exploration: Drill at a new location
- **Game Playing**
  - Exploitation: Play the move you believe is best
    Exploration: Play an experimental move



Q-Learning: Representation Matters

Unfortunately, value iteration is **impractical**

- Limited states/actions
- Cannot generalize to unobserved states

Think about the **Breakout** Arcade game

State: screen pixels

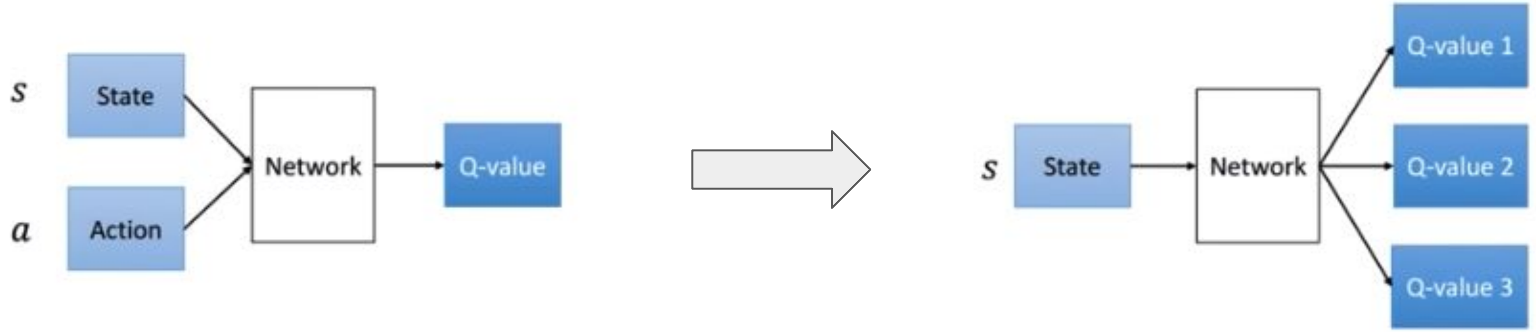
- Image size: 84 x 84 (resized)
- Consecutive 4 images
- Grayscale with 256 gray levels
- → **$256^{84x84x4}$** rows in the Q-table! ($256^{28,224}$ =$10^{69,970}$ >> 10^{82} atoms in the universe)

Deep RL = RL + Neural Networks

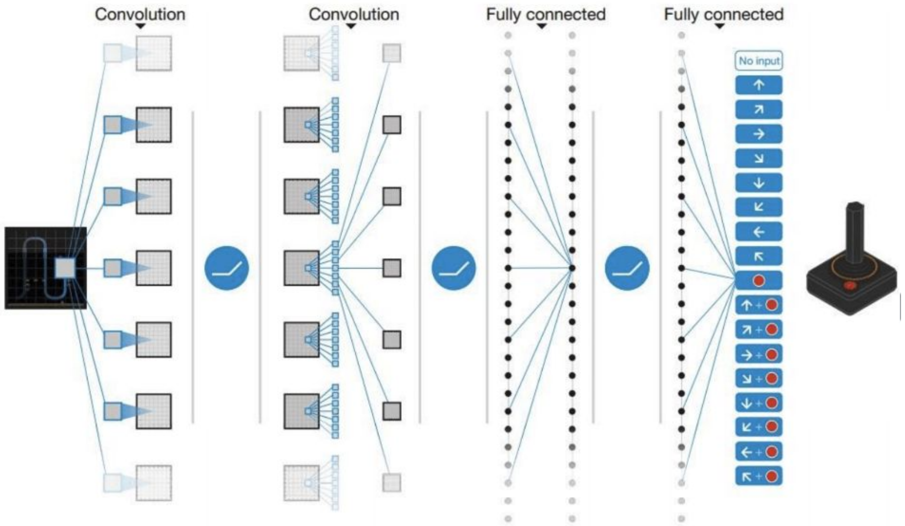
Use a deep neural network to approximate
Q-function → Deep Q-Network (DQN):

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

network
parameters



Deep Q-Network (DQN) Architecture



Layer	Input	Filter size	Stride	Num filters	Activation	Output
conv1	84x84x4	8x8	4	32	ReLU	20x20x32
conv2	20x20x32	4x4	2	64	ReLU	9x9x64
conv3	9x9x64	3x3	1	64	ReLU	7x7x64
fc4	7x7x64			512	ReLU	512
fc5	512			18	Linear	18

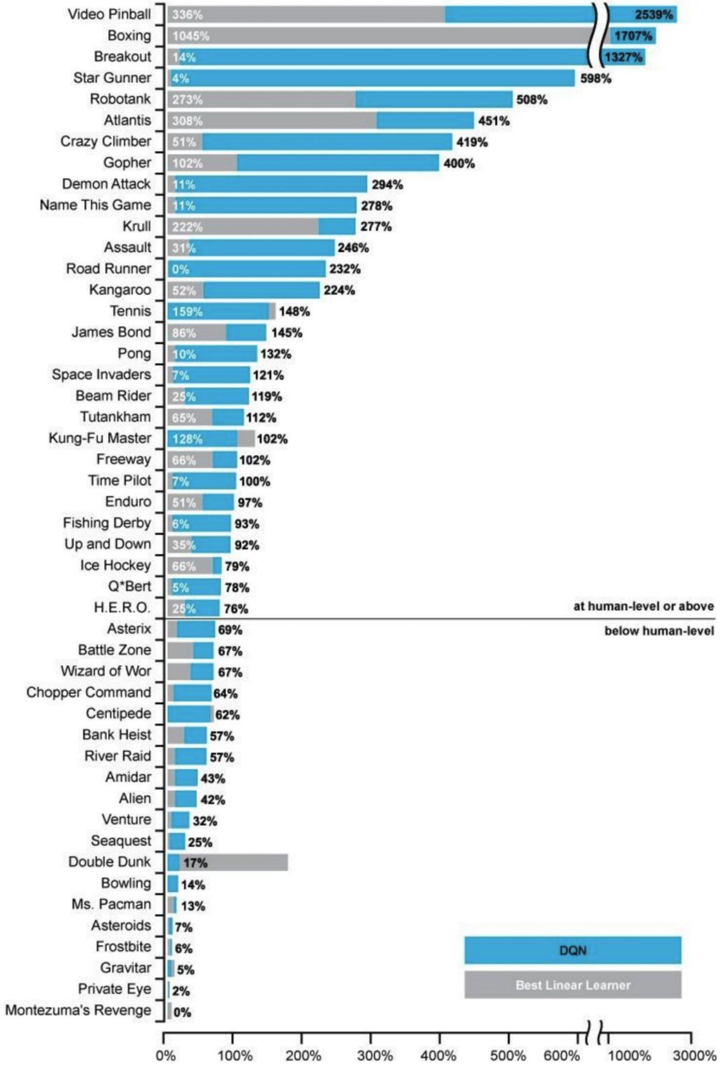
Mnih et al. “Playing atari with deep reinforcement learning.” 2013.

Demo video

Experience Replay

- Current Q-network parameters determines next training examples → can lead to **bad feedback loop**
- Stores experience (actions, state transitions, and rewards) and create **mini-batches** from them for the training process
- Continually update a **replay memory table** of transitions as game experience are played
- Update Q-network on random mini-batch of transitions from the replay memory, instead of consecutive samples

DQN on Atari



Video Pinball
336%
2539%
Boxing
1045%
1707%
Breakout
14%
1327%
Star Gunner
4%
598%
Robotank
273%
508%
Atlantis
308%
451%
Crazy Climber
51%
419%
Gopher
102%
400%
Demon Attack
11%
294%
Name This Game
11%
278%
Krull
222%
277%
Assault
31%
246%
Road Runner
0%
232%
Kangaroo
52%
224%
Tennis
159%
148%
James Bond
86%
145%
Pong
10%
132%
Space Invaders
7%
121%
Beam Rider
25%
119%
Tutankham
65%
112%
Kung-Fu Master
128%
102%
Freeway
66%
102%
Time Pilot
7%
100%
Enduro
51%
97%
Fishing Derby
6%
93%
Up and Down
35%
92%
Ice Hockey
66%
79%
Q*Bert
5%
78%
H.E.R.O.
25%
76%
at human-level or above
below human-level
Asterix
69%
Battle Zone
67%
Wizard of Wor
67%
Chopper Command
64%
Centipede
62%
Bank Heist
57%
River Raid
57%
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%
DQN
Best Linear Learner
0%
100%
200%
300%
400%
500%
600%
1000%
3000%

Policy-based Method: Policy Gradient

Policy Gradients

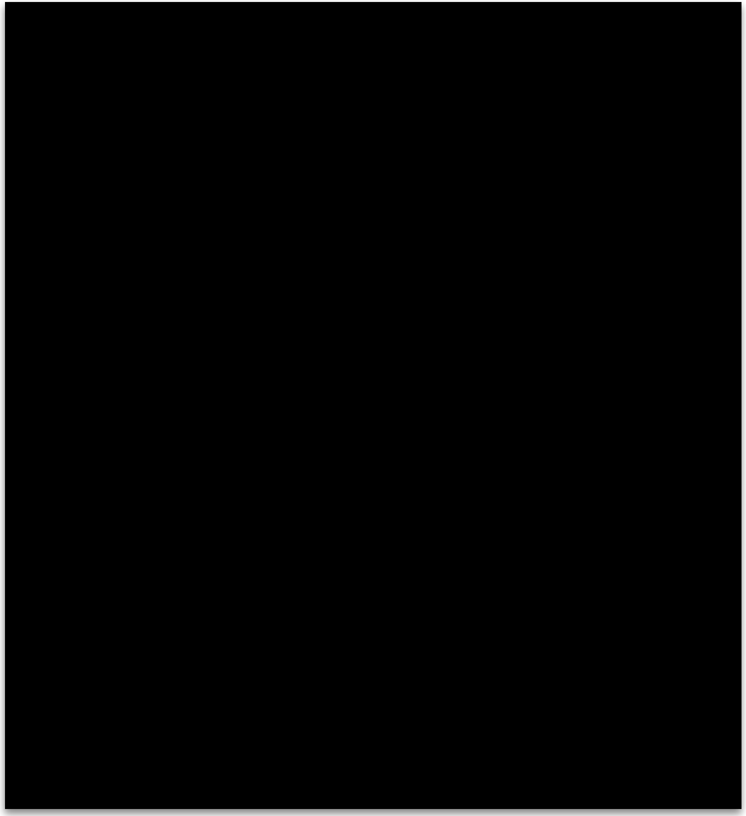
A problem with Q-learning is that the Q-function can be very complicated!

> Example: a robot grasping an object has a **very high dimensional state** ⇒ hard to learn exact value of every (state, action) pair

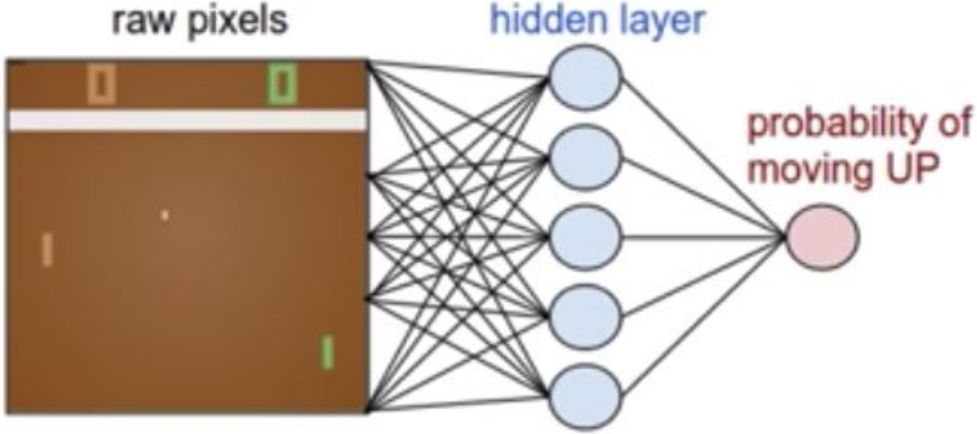
The policy could be much simpler: just close your hand/claw

Can we optimize a policy **directly** by finding the best one from a collection of policies?

Policy Gradients -- on Pong

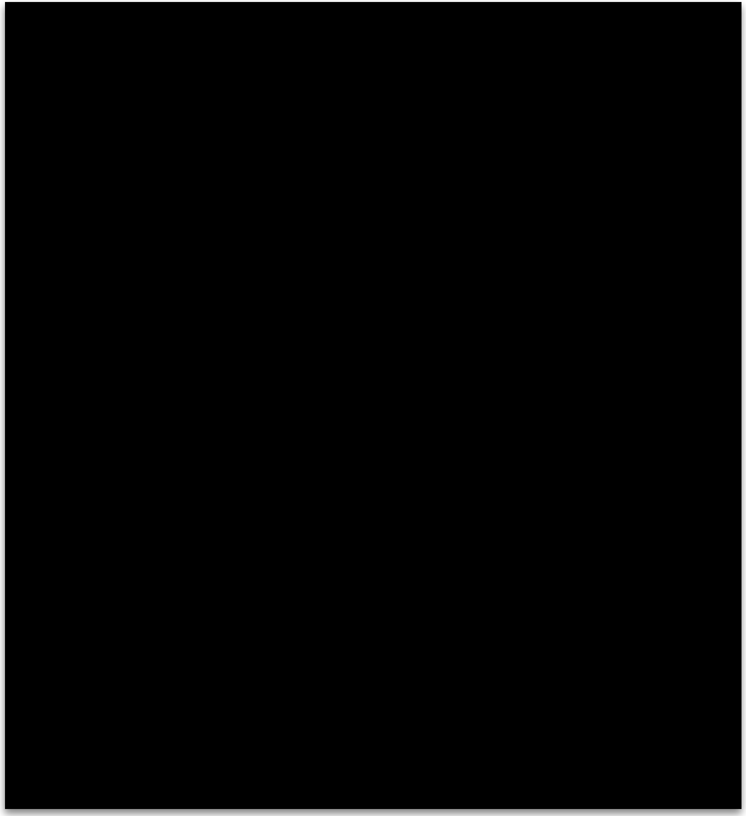


- 80 x 80 image (difference image)
- 2 actions: up or down
- 200,000 Pong games
- This is a step towards **general purpose AI** !

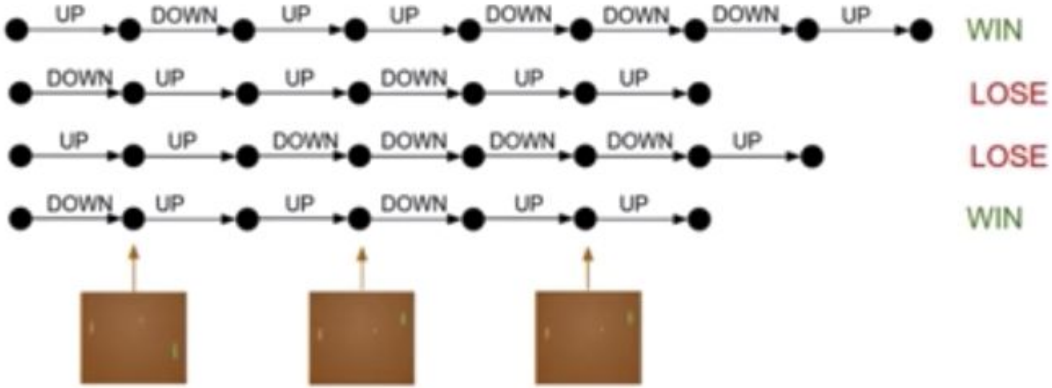


Karpathy et al. “Deep Reinforcement Learning: Pong from Pixels.” 2016.

Policy Gradients -- on Pong



Action trajectories/**series**:



Karpathy et al. “Deep Reinforcement Learning: Pong from Pixels.” 2016.

Policy Gradients (PG)

Formally, let's define a class of policies: $\Pi = \{\pi_\theta, \theta \in \mathbb{R}^m\}$

For each policy, define its value:

$$J(\theta) = \mathbb{E}\Big[\sum_{t\geq 0} \gamma^t r_t | \pi_\theta\Big]$$

We want to find the optimal policy: $\theta^* = \arg\max_\theta J(\theta)$

How to do this? → **Gradient Descent** (Ascent) $\nabla_\theta J(\theta)$ on policy parameter θ!

Policy Gradients (PG)

Mathematically, we can rewrite J() in terms of action trajectory:

$$J(\theta) = \mathbb{E}_{\tau \sim p(\tau;\theta)}\left[r(\tau)\right]$$

Where **r(τ)** is the reward of a trajectory: $\tau = (s_0, a_0, r_0, s_1, a_1, r_2, \ldots)$

Gradient Estimator (skipping the derivation...):

$$\nabla_\theta J(\theta) \approx \sum_{t \geq 0} r(\tau) \nabla_\theta \log \pi_\theta(a_t | s_t)$$

Interpretation:

- If **r(τ)** is high, push up the probabilities of the seen actions
- If **r(τ)** is low, push down the probabilities of the seen actions

PG comparing to DQN

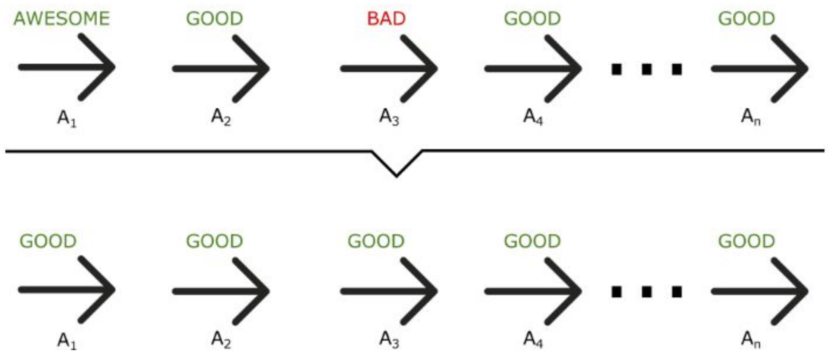
Pros:

+ **Messy World:** If Q function is too complex to learn, DQN may fail while PG will still learn a good policy
+ **Speed**: Faster convergence
+ **Stochastic**: PG is capable of learning stochastic policies while DQN cannot
+ **Continuous actions:** It's easier to model PG on continuous space

Cons:

- **Data**: Sample Inefficient (need more data)
- **Stability**: Less stable during training process
- **Credit assignment:** Poor assignment to (state, action) pairs for delayed rewards

The problem with Policy Gradients



We have to wait until the end of a trajectory to calculate the reward. If the reward were high, all actions that we took were good, even if some were **really bad**

As a consequence, we need to have a **LOT** of samples to have an optimal policy. This means slow learning and long time to converge

What if we can do update at each step?!

Hybrid Method: Actor-Critic

Introducing Actor-Critic Algorithm

Using two neural networks:

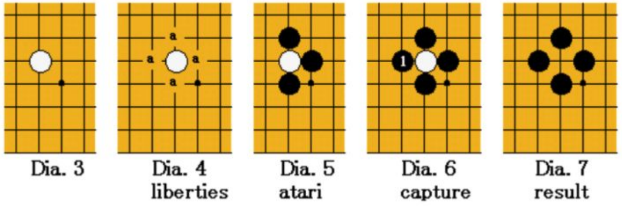
1. **An Actor** that measures how good the action taken (**value-based**)
2. **A Critic** that controls how our actor behaves (**policy-based**)

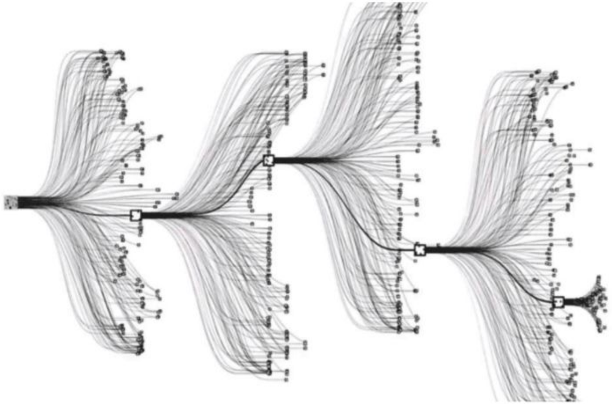
Actor Critic Algorithm

- The actor decides which action to take, and the critic tells the actor how good its action was and how it should adjust
- Alleviates the task of the critic as it only has to learn the values of (state, action) pairs generated by the policy
- Can also incorporate Q-learning tricks (e.g. experience replay)
- **Remark**: we can define by the **advantage function** how much an action was better than expected $A^\pi(s,a) = Q^\pi(s,a) - V^\pi(s)$
- Using this, our gradient estimator becomes:

$$\nabla_\theta J(\theta) \approx \sum_{t\geq 0} (Q^{\pi_\theta}(s_t, a_t) - V^{\pi_\theta}(s_t)) \nabla_\theta \log \pi_\theta(a_t|s_t)$$

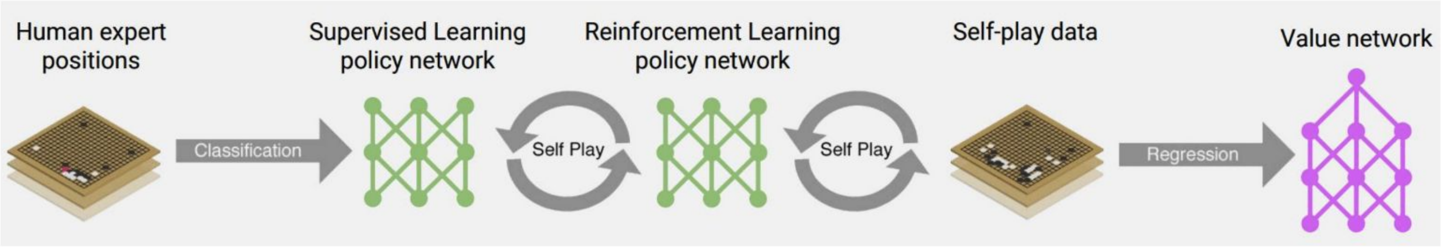
Application: Game of Go

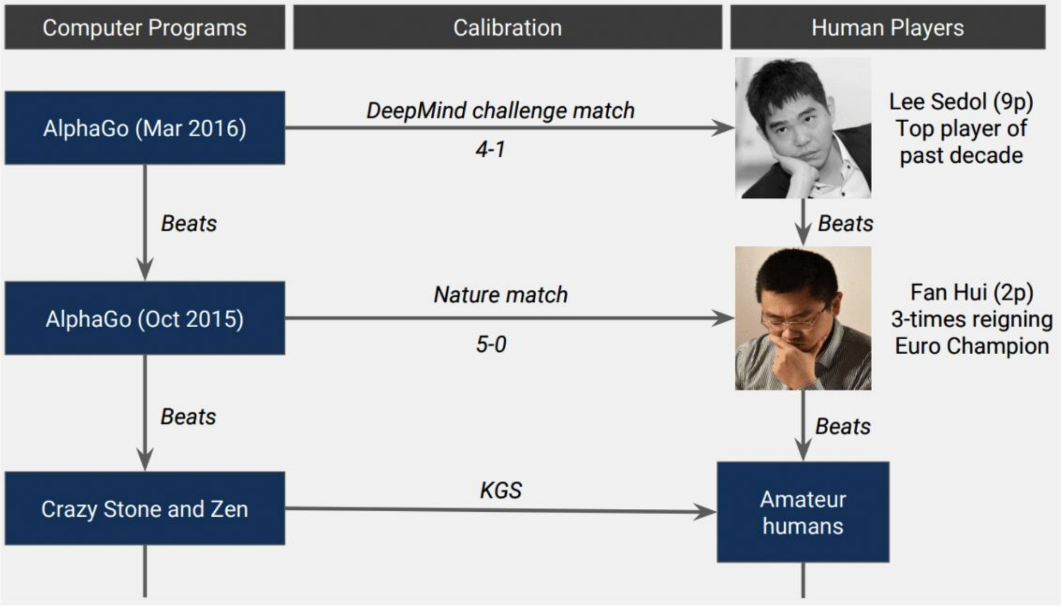




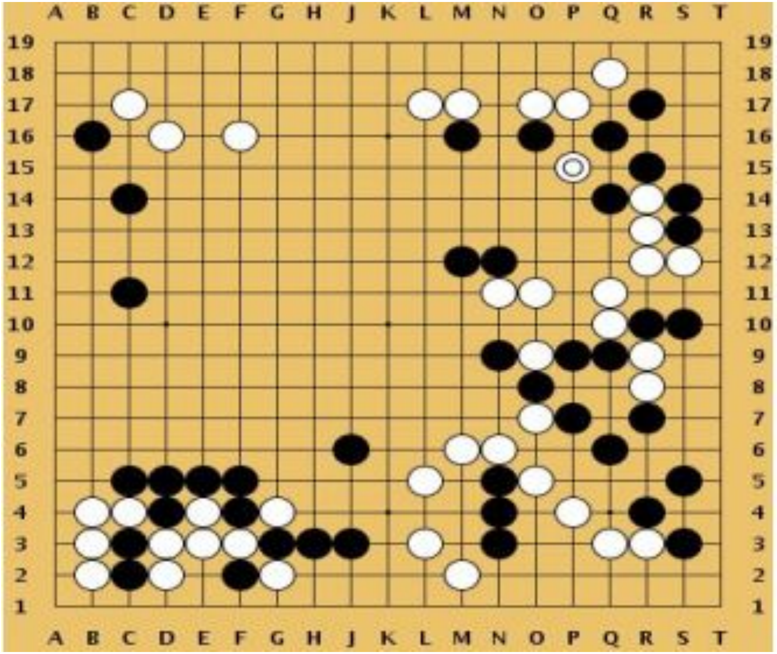
Game size	Board size N	3^N	Percent legal	legal game positions (A094777)[11]
1×1	1	3	33%	1
2×2	4	81	70%	57
3×3	9	19,683	64%	12,675
4×4	16	43,046,721	56%	24,318,165
5×5	25	8.47×10^{11}	49%	4.1×10^{11}
9×9	81	4.4×10^{38}	23.4%	1.039×10^{38}
13×13	169	4.3×10^{80}	8.66%	3.72497923×10^{79}
19×19	361	1.74×10^{172}	1.196%	$2.08168199382\times10^{170}$

Alpha Go


Human expert positions
Supervised Learning policy network
Reinforcement Learning policy network
Self-play data
Value network
Classification
Self Play
Self Play
Regression


Computer Programs
Calibration
Human Players
AlphaGo (Mar 2016)
DeepMind challenge match
4-1
Lee Sedol (9p)
Top player of
past decade
Beats
Beats
AlphaGo (Oct 2015)
Nature match
5-0
Fan Hui (2p)
3-times reigning
Euro Champion
Beats
Beats
Crazy Stone and Zen
KGS
Amateur
humans

AlphaGo and variants



AlphaGo [Nature 2016]:

- Required many engineering tricks
- Bootstrapped from human play
- Beat 18-time world champion Lee Sedol

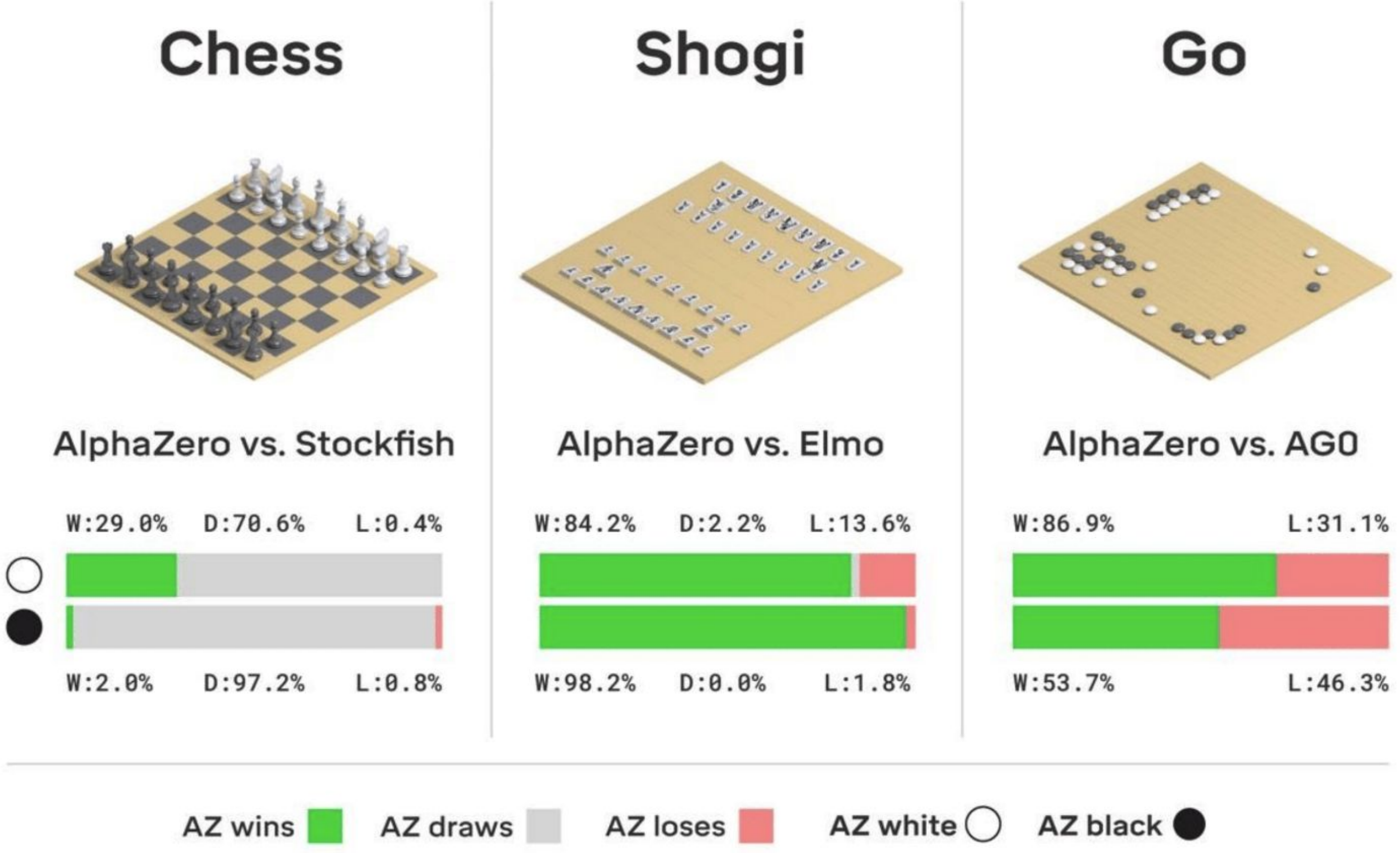
AlphaGo Zero [Nature 2017]:

- Simplified and elegant version of AlphaGo
- No longer bootstrapped from human play
- Beat (at the time) #1 world ranked Ke Jie

Alpha Zero: [Science 2018]

- Generalized to beat world champion programs on chess and shogi as well

Alpha Zero in action



Summary: Learning Objectives

- ✓ Overview of Reinforcement Learning (RL)
- ✓ Value-based Q-Learning method and Deep Q-Network
- ✓ A Policy-based method called Policy Gradients
- ✓ The Actor-critic algorithm and an application in AlphaGo

Next Steps in RL

Background

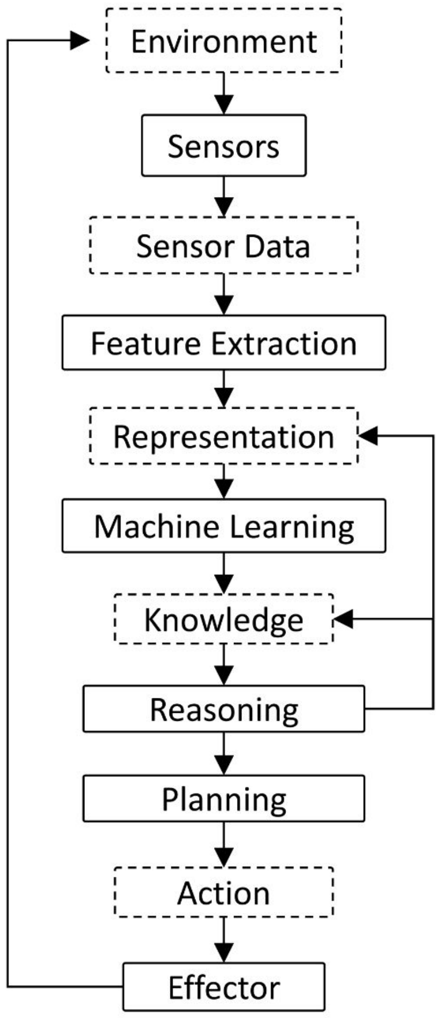
- Fundamentals in probability, statistics, multivariate calculus.
- Deep learning basics
- Deep RL basics
- TensorFlow (or PyTorch)

Learn by doing

- Implement core deep RL algorithms (discussed today)
- Look for tricks in papers that were key to get it to work
- Iterate fast in simple environments

Research

- Improve on an existing approach
- Focus on an unsolved task / benchmark
- Create a new task / problem that hasn't been addressed



Acknowledgement

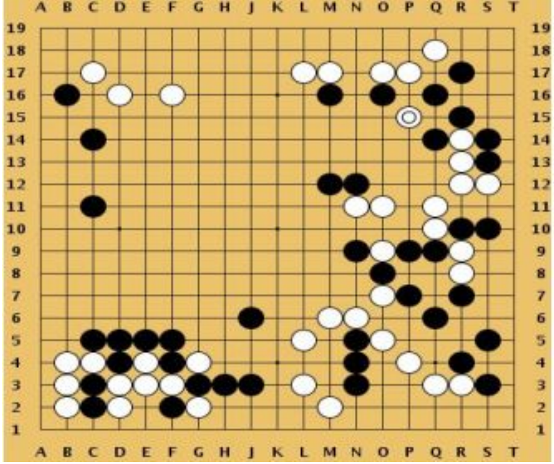
Slides contain materials and figures reproduced from Lex Fridman at MIT and Serena Yeung at Stanford for educational purposes only.

Massachusetts
Institute of
Technology

Stanford
University

Bonus Slides

PG Application: AlphaGo



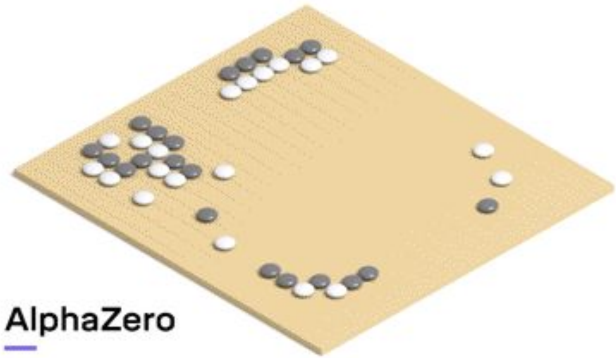
Mix of supervised learning and reinforcement learning

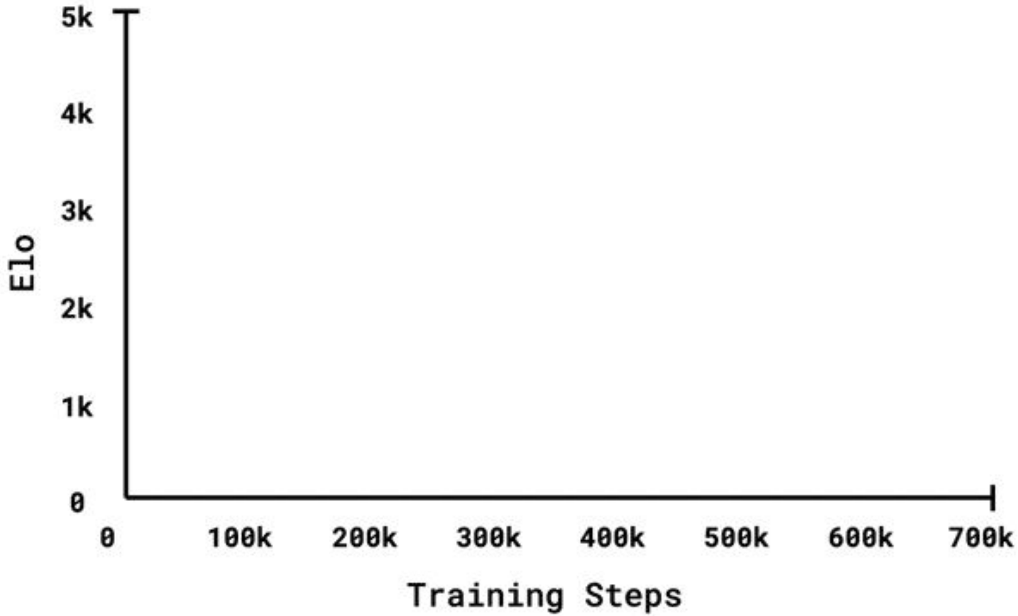
How to beat the Go world champion:

- Featurize the board (stone color, move legality, bias,…)
- Initialize policy network with supervised training from professional go games, then continue training using policy gradient (**play against itself** from random previous iterations, +1 / -1 reward for winning / losing)
- Also learn value network (the critic)
- Finally, combine policy and value networks in a Monte Carlo Tree Search algorithm to select actions by look-ahead search

Alpha Zero in action

Elo rating is a method to calculate relative skill level of player in a zero-sum based game such as chess. Each training step represents 4,096 board positions





DQN and double DQN

Loss function:

$$L = \mathbb{E}[(\underbrace{r + \gamma \max_{a'} \mathbf{Q}(s', a')}_{\text{Target (y)}} - \underbrace{\mathbf{Q}(s, a; \theta)}_{\text{prediction}})^2]$$

DQN: same network for both Q

Double DQN: separate network for each Q to help reduce bias introduced by the inaccuracies of Q network at the beginning of training