Session 5

Reinforcement Learning

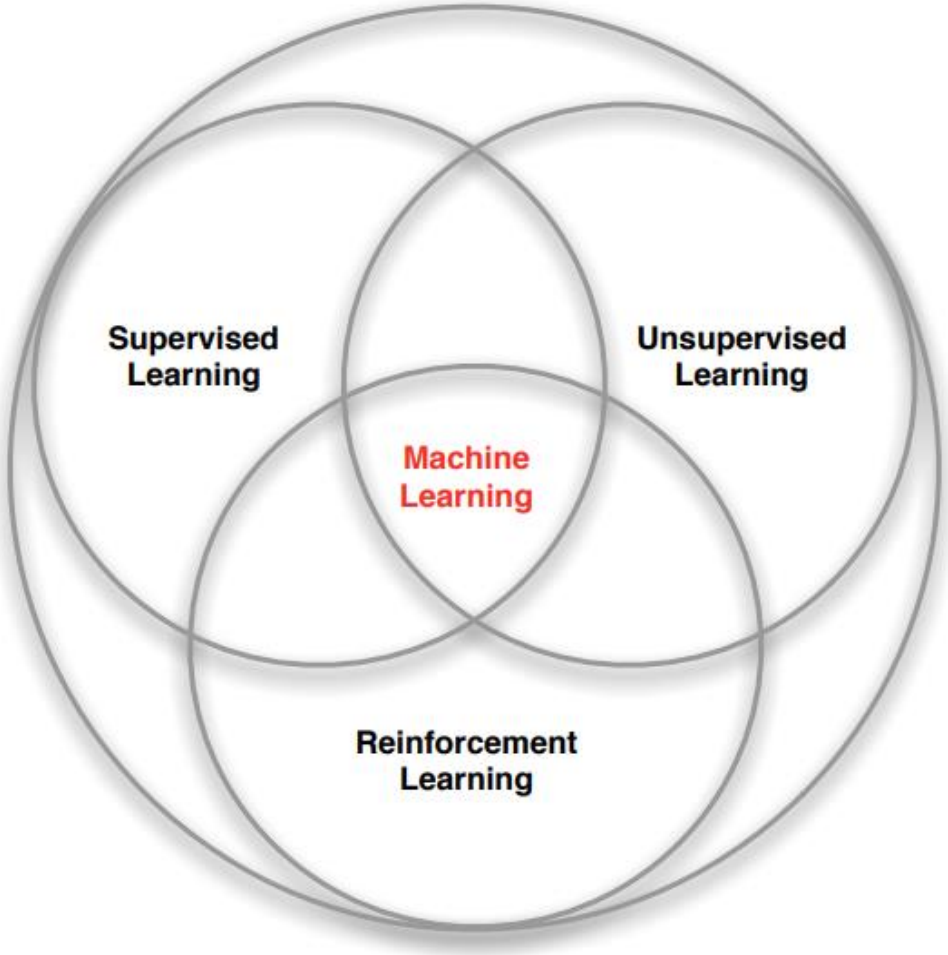
Acknowledgements

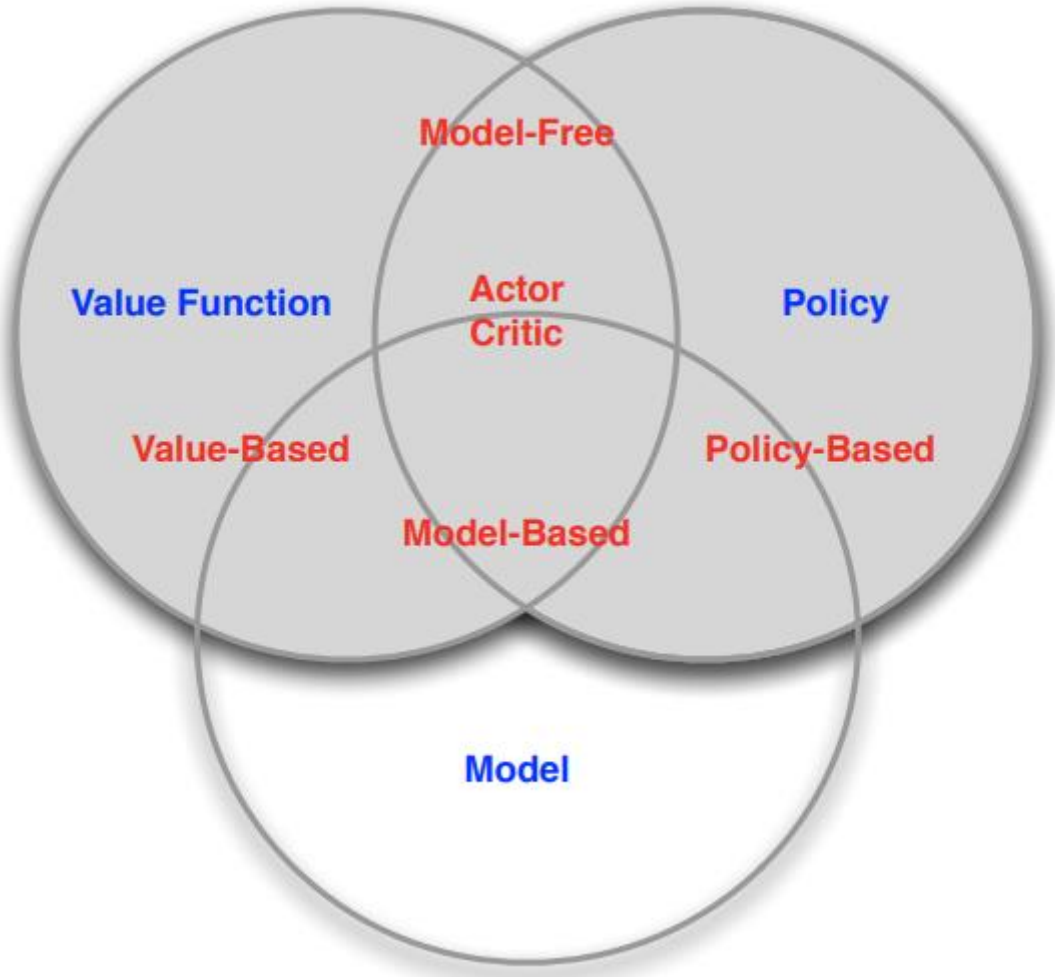
- Stanford University cs234 class:

http://cs231n.stanford.edu/slides/2017/cs231n_2017_lecture14.pdf

- OpenAI spinningup
  https://spinningup.openai.com/en/latest/spinningup/rl_intro.html
- David Sliver

https://www.davidsilver.uk/wp-content/uploads/2020/03/intro_RL.pdf

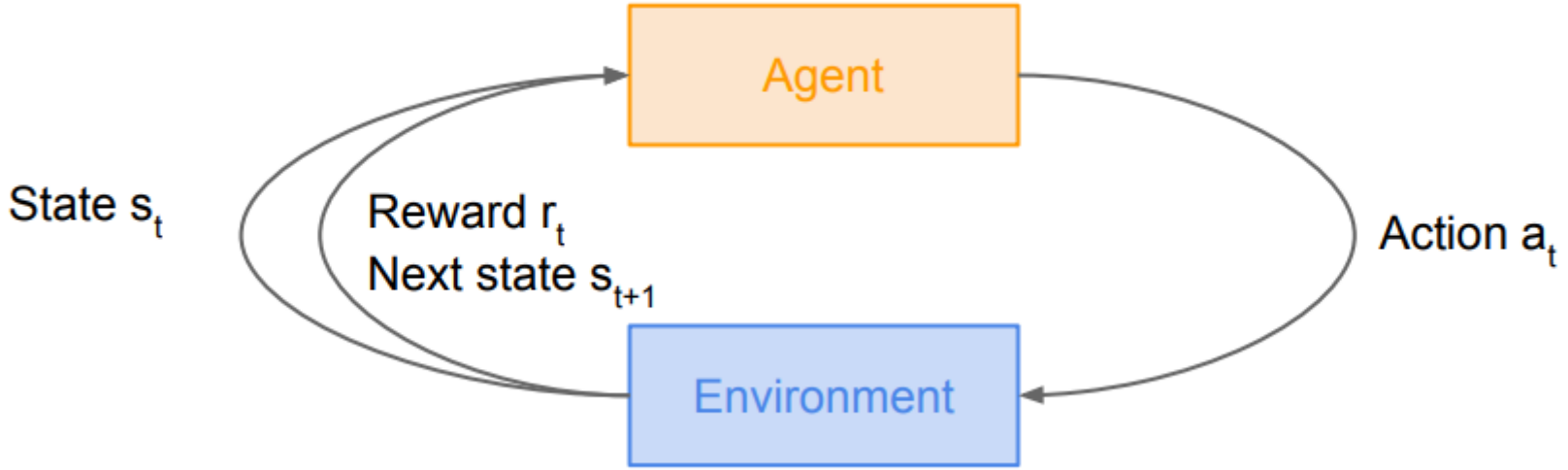




Figures from [David Sliver]

Introduction

- What makes reinforcement learning different from other machine learning paradigms?
  - There is no supervisor, only a reward signal
  - Feedback is delayed, not instantaneous
  - Time really matters (sequential, non i.i.d data)
  - Agent's actions affect the subsequent data it receives



Reinforcement learning(RL)

- The main characters of RL are the **agent** and the **environment**. The environment is the world that the agent lives in and interacts with. At every step of interaction, the agent sees a (possibly partial) observation of the state of the world, and then decides on an action to take. The environment changes when the agent acts on it, but may also change on its own.
- The agent also perceives a **reward** signal from the environment, a number that tells it how good or bad the current world state is. The goal of the agent is to maximize its cumulative reward, called **return**.
- Reinforcement learning are ways that agents can learn behaviors to achieve its goal.

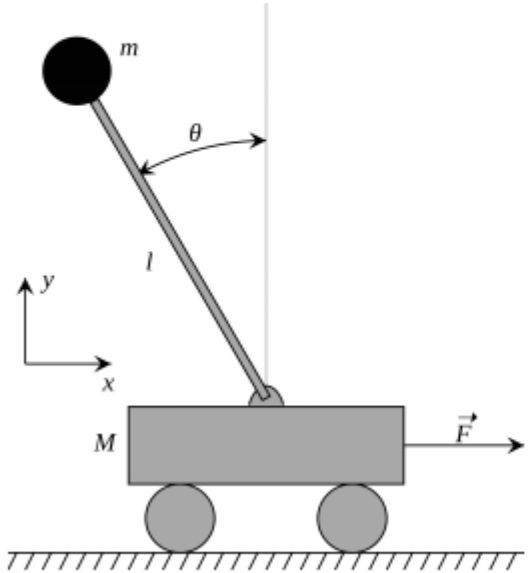
The fundamental problems

- Two fundamental problems in sequential decision making
  - Reinforcement Learning:
    - The environment is initially unknown
    - The agent interacts with the environment
    - The agent improves its policy
  - Planning:
    - A model of the environment is known
    - The agent performs computations with its model (without any external interaction)
    - The agent improves its policy
    - a.k.a. deliberation, reasoning, introspection, pondering, thought, search

Example of RL application

- Fly stunt maneuvres in a helicopter
  - +ve reward for following desired trajectory
  - −ve reward for crashing
- Defeat the world champion at Backgammon
  - +/−ve reward for winning/losing a game
- Manage an investment portfolio
  - +ve reward for each $ in bank
- Control a power station
  - +ve reward for producing power
  - −ve reward for exceeding safety thresholds
- Make a humanoid robot walk
  - +ve reward for forward motion
  - −ve reward for falling over
- Play many different Atari games better than humans
  - +/−ve reward for increasing/decreasing score

Cart-Pole Problem



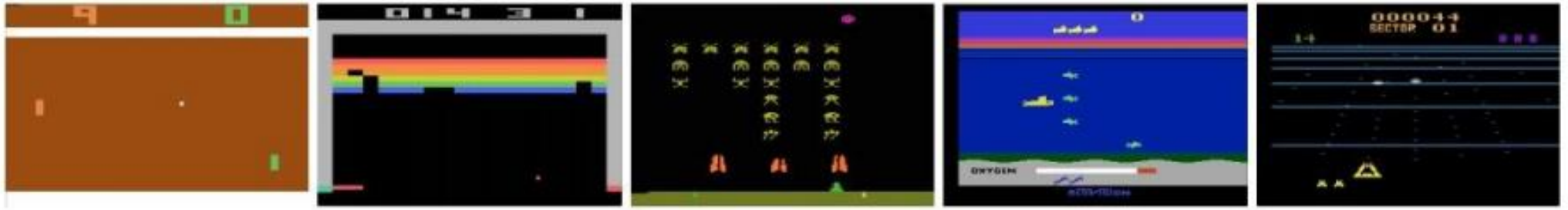
Objective: Balance a pole on top of a movable cart

State: angle, angular speed, position, horizontal velocity
Action: horizontal force applied on the cart
Reward: 1 at each time step if the pole is upright

Real life application with cart pole

Simulated game

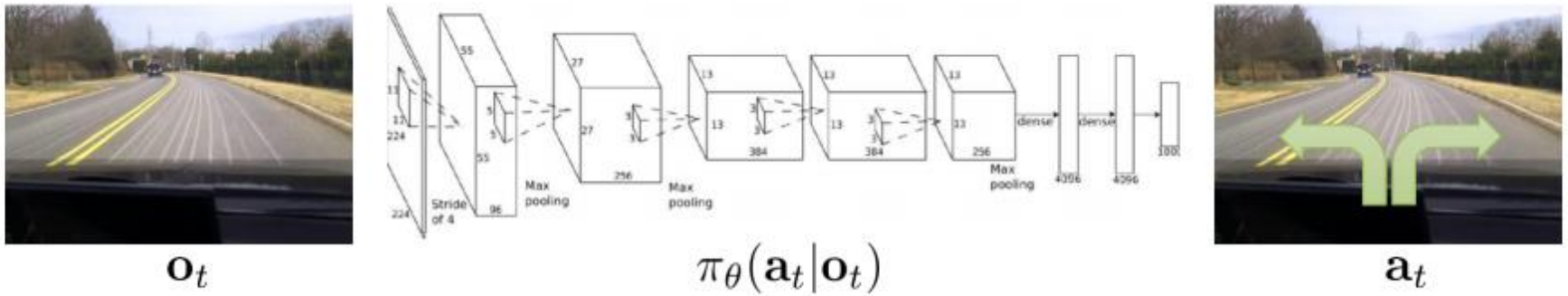
Atari Games

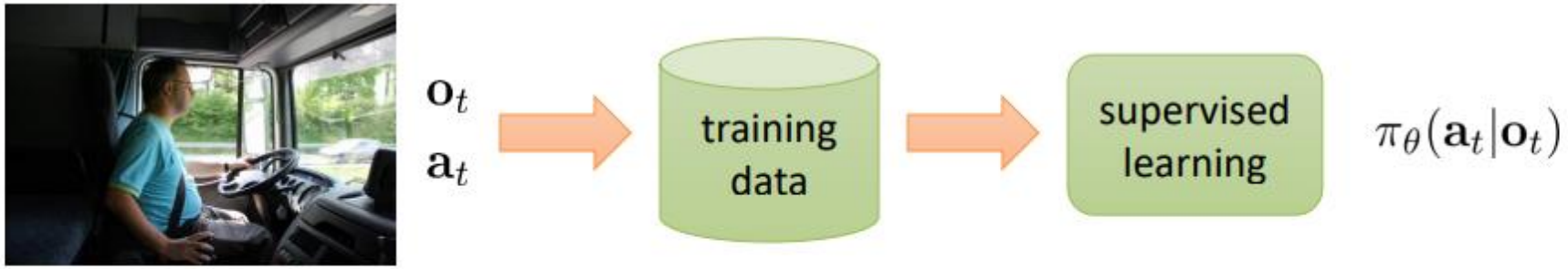


Objective: Complete the game with the highest score

State: Raw pixel inputs of the game state
Action: Game controls e.g. Left, Right, Up, Down
Reward: Score increase/decrease at each time step

Imitation learning



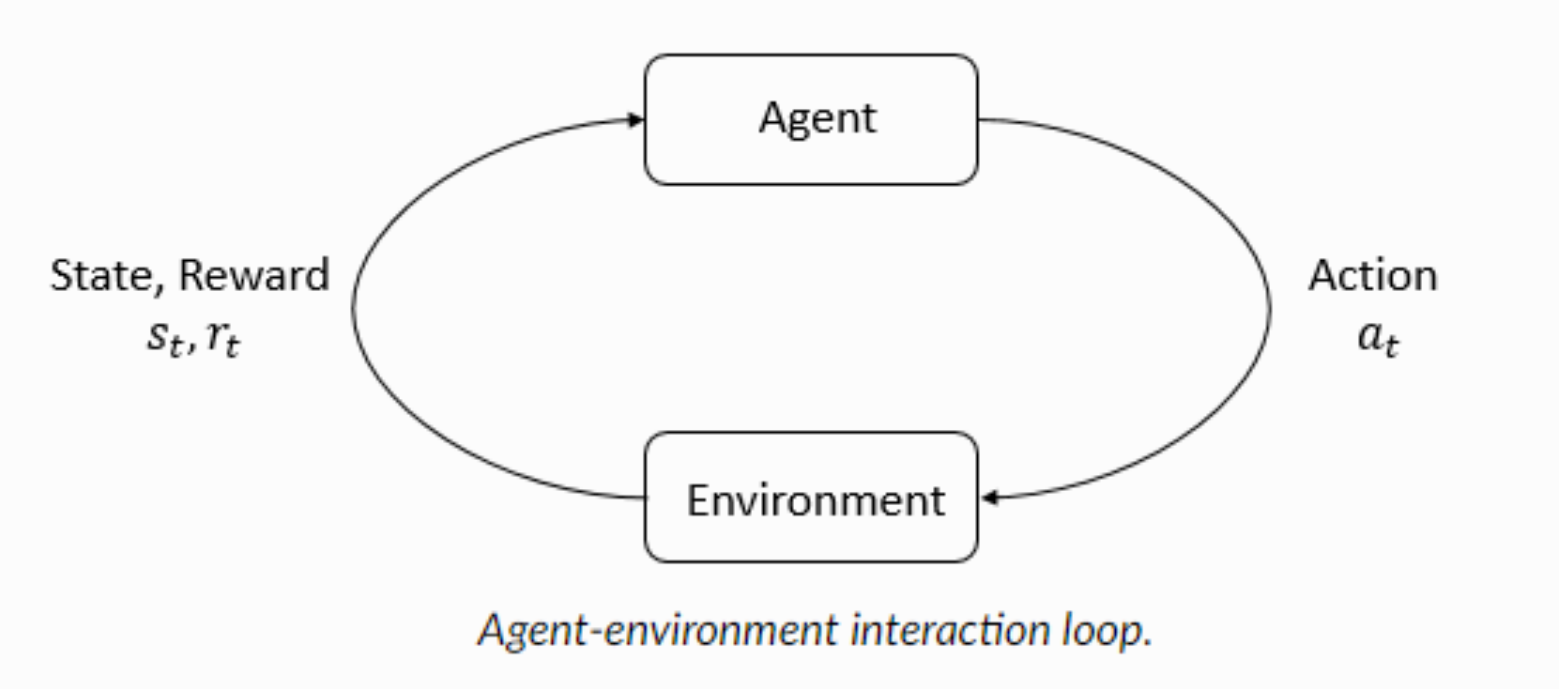


Exploration and Exploitation

- Reinforcement learning is like trial-and-error learning
- The agent should discover a good policy from its experiences of the environment without losing too much reward along the way
  - **Exploration** finds more information about the environment
  - **Exploitation** exploits known information to maximize reward
- It is usually important to explore as well as exploit

- Examples
  - Restaurant Selection
    - Exploitation: go to your favorite restaurant
    - Exploration: Try a new restaurant Online
  - Banner Advertisements:
    - Exploitation: Show the most successful advert
    - Exploration: Show a different advert
  - 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

Introduction



Agent-environment interaction loop.

How can we mathematically formalize the RL problem?

Terminology (1)

- states (s) and observations (o):
  - A state s is a complete description of the state of the world. There is no information about the world which is hidden from the state.
  - An observation o is a partial description of a state, which may omit information
- Action spaces (A): The set of all valid actions in a given environment. There are discrete action spaces (game). Other environments, like where the agent controls a robot in a physical world, have continuous action spaces. In continuous spaces, actions are real-valued vectors.
- Policies (π):
  - A **policy** is a rule used by an agent to decide what actions to take. It can be deterministic or stochastic. It is essential the agent's brain
  - In deep RL, we deal with **parameterized policies**: policies whose outputs are computable functions that depend on a set of parameters (e.g. the weights and biases of a neural network) which we can adjust to change the behavior via some optimization algorithm

Terminology (2)

- Trajectories τ: It is a sequence of states and actions

$$\tau = (s_0, a_0, s_1, a_1, \dots)$$

- Reward and return: The reward function R is critically important in reinforcement learning. It depends on the current state of the world, the action just taken, and the next state of the world

$$r_t = R(s_t, a_t, s_{t+1})$$

- **Advantage Functions:** it describes how much better this action is than others on average. That is to say, we want to know the relative **advantage** of that action. We make this concept precise with the **advantage function.**

$$A^{\pi}(s, a) = Q^{\pi}(s, a) - V^{\pi}(s)$$

Markov Decision Process

- Mathematical formulation of the RL problem
- **Markov property**: Current state completely characterises the state of the world

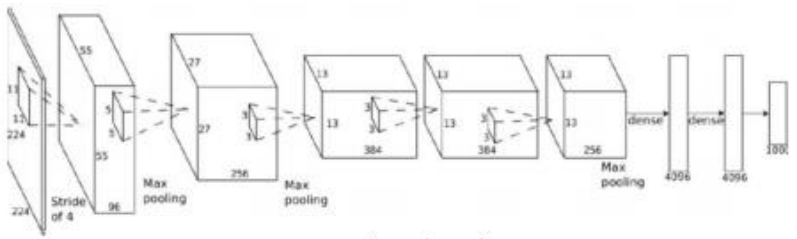
Defined by: $(\mathcal{S}, \mathcal{A}, \mathcal{R}, \mathbb{P}, \gamma)$

$\mathcal{S}$: set of possible states
$\mathcal{A}$: set of possible actions
$\mathcal{R}$: distribution of reward given (state, action) pair
$\mathbb{P}$: transition probability i.e. distribution over next state given (state, action) pair
γ : discount factor

Markov Decision Process

- At time step t=0, environment samples initial state $s_0 \sim p(s_0)$
- Then, for t=0 until done:
  - Agent selects action a_t
  - Environment samples reward $r_t \sim R(. \mid s_t, a_t)$
  - Environment samples next state $s_{t+1} \sim P(. \mid s_t, a_t)$
  - Agent receives reward r_t and next state s_{t+1}

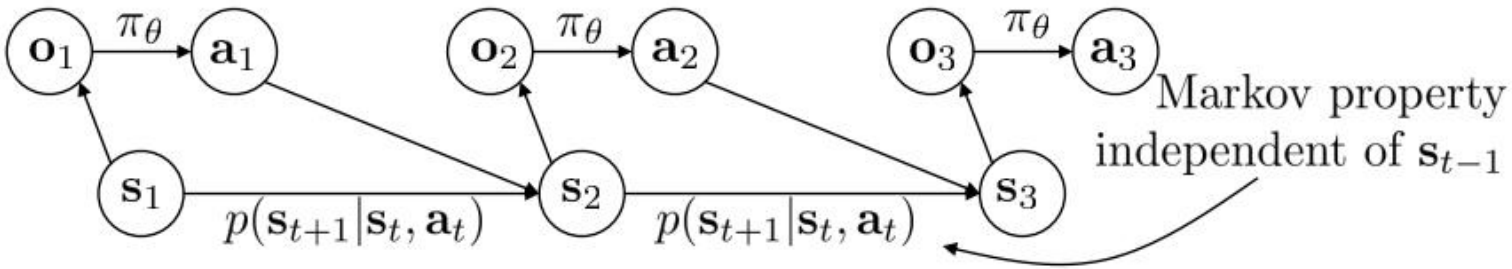
- A policy **π** is a function from S to A that specifies what action to take in each state
- **Objective**: find policy **π*** that maximizes cumulative discounted reward: $\sum_{t \geq 0} \gamma^t r_t$



$\mathbf{o}_t$ $\pi_\theta(\mathbf{a}_t|\mathbf{o}_t)$ $\mathbf{a}_t$

$\mathbf{s}_t$ – state
$\mathbf{o}_t$ – observation
$\mathbf{a}_t$ – action

$\pi_\theta(\mathbf{a}_t|\mathbf{o}_t)$ – policy
$\pi_\theta(\mathbf{a}_t|\mathbf{s}_t)$ – policy (fully observed)

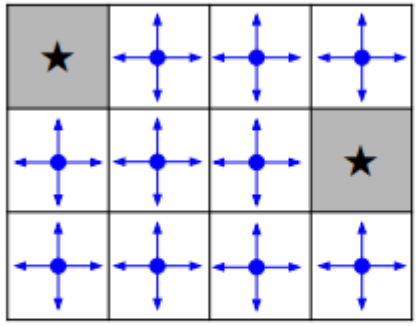


Example

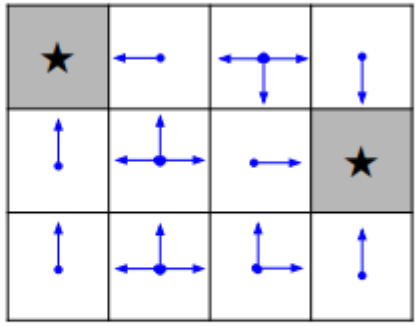
actions = {

1. right
2. left
3. up
4. down

}



Random Policy



Optimal Policy

Objective: reach one of the terminal states (greyed out) in the least number of actions

The optimal policy

We want to find optimal policy π^* that maximizes the sum of rewards.

How do we handle the randomness (initial state, transition probability…)?
Maximize the **expected sum of rewards!**

Formally: $\pi^* = \arg\max_{\pi} \mathbb{E}\left[\sum_{t \geq 0} \gamma^t r_t | \pi\right]$ with $s_0 \sim p(s_0), a_t \sim \pi(\cdot|s_t), s_{t+1} \sim p(\cdot|s_t, a_t)$

Value function and Q function

Following a policy produces sample trajectories (or paths) $s_0, a_0, r_0, s_1, a_1, r_1, \ldots$

How good is a state?

The **value function** at state s, is the expected cumulative reward from following the policy from state s:

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

How good is a state-action pair?

The **Q-value function** at state s and action a, is the expected cumulative reward from taking action a in state s and then following the policy:

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

Bellman Equations

- All value functions obey special self-consistency equations called Bellman equations. The basic idea behind the Bellman equations is this:

The value of your starting point is the reward you expect to get from being there, plus the value of wherever you land next.

The optimal Q-value function Q* is the maximum expected cumulative reward achievable from a given (state, action) pair:

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

Q* satisfies the following **Bellman equation**:

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

Intuition: if the optimal state-action values for the next time-step Q*(s',a') are known, then the optimal strategy is to take the action that maximizes the expected value of $r + \gamma Q^*(s',a')$

The optimal policy π^* corresponds to taking the best action in any state as specified by Q*

Solving for the optimal policy

- First approach is by value iteration

$$Q_{i+1}(s,a) = E\left[r + \gamma \max_{a'} Q_i(s',a') \middle| s,a\right]$$

- Q_{i+1} will converge to Q^* as i close to infinity

What is the problem here?

Not scalable. Must compute Q(s,a) for every state-action pair. If state is e.g. current game state pixels, computationally infeasible to compute for entire state space!

Solution: use a function approximator to estimate Q(s,a). E.g. a neural network!

Q-learning: Use a function approximator to estimate the action-value function by parameter θ

$$Q_{i+1}(s,a;\theta) \approx Q^*(s,a)$$

If the function approximator is a deep neural network => deep q-learning!

Solving for the optimal policy: Q-learning

Remember: want to find a Q-function that satisfies the Bellman Equation:

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

Forward Pass

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

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

Iteratively try to make the Q-value close to the target value (y_i) it should have, if Q-function corresponds to optimal Q* (and optimal policy π^*)

Backward Pass

Gradient update (with respect to Q-function parameters θ):

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

Types of RL algorithms

$$\theta^\star = \arg\max_\theta E_{\tau\sim p_\theta(\tau)}\left[\sum_t r(\mathbf{s}_t, \mathbf{a}_t)\right]$$

- Policy gradients: directly differentiate the above objective
- Value-based: estimate value function or Q-function of the optimal policy (no explicit policy)
- Actor-critic: estimate value function or Q-function of the current policy, use it to improve policy
- Model-based RL: estimate the transition model, and then…
  - Use it for planning (no explicit policy)
  - Use it to improve a policy
  - Something else

Example of value iteration

→	→	→	+1
↓		→	-1
→	→	↑	↑

Example of one policy π

0.86	0.90	0.93	+1
0.82		0.69	-1
0.78	0.75	0.71	0.49

Example of one value function V^{π}

$$V^{\pi}(s) = E[R(s_0) + \gamma V^{\pi}(s') \,|\, \pi, s: s_0]$$

Value iteration: use the Bellman equations to solve V*

Initialize V(s)=0

From every s update: $V(s) \coloneqq R(s) + \max_a \gamma \sum_{s'} P_{sa}(s')V(s')$

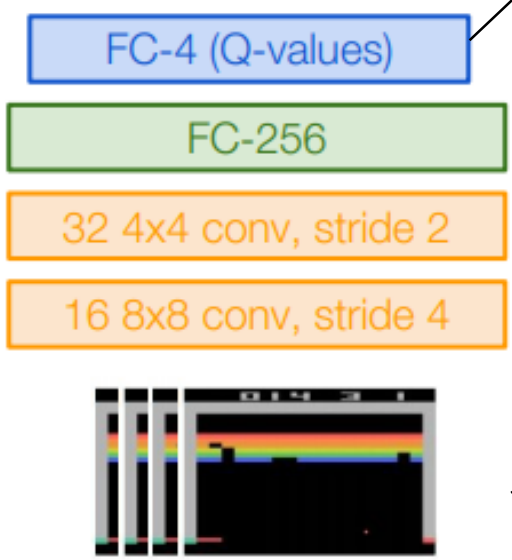
$$w: \sum_s P_{sa}(s')V^*(s') = 0.8x0.75 + 0.1x0.69 + 0.1 * 0.71 = 0.78$$

$$N: \sum_s P_{sa}(s')V^*(s') = 0.8x0.69 + 0.1x0.75 + 0.1 * 0.49 = 0.66$$

[Mnih et al. NIPS Workshop 2013; Nature 2015]

Q-network Architecture

$Q(s, a; \theta)$:
neural network
with weights θ



Last FC layer has 4 output (if 4 actions), corresponding to Q(st , a1), Q(st , a2), Q(st , a3), Q(st ,a4)

Familiar conv layers, FC layer

Input: state s_t

Current state s_t: 84x84x4 stack of last 4 frames
(after RGB->grayscale conversion, downsampling, and cropping)

Training the Q-network: Experience Replay

- Learning from batches of consecutive samples is problematic:
  - Samples are correlated => inefficient learning
  - Current Q-network parameters determines next training samples (e.g. if maximizing action is to move left, training samples will be dominated by samples from left-hand size) ➔ can lead to bad feedback loops
- Address these problems using experience replay
  - Continually update a replay memory table of transitions (s_t, a_t, r_t, s_{t+1}) as game (experience) episodes are played
  - Train Q-network on random minibatches of transitions from the replay memory, instead of consecutive samples

Each transition can also contribute to multiple weight updates
➔greater data efficiency

[Mnih et al. NIPS Workshop 2013; Nature 2015]

Putting it together: Deep Q-Learning with Experience Replay

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

Video by Károly Zsolnai-Fehér

Policy Gradients

- What is a problem with Q-learning?
  - 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
- But the policy can be much simpler: just close your hand
- Can we learn a policy directly, e.g. finding the best policy from a collection of policies?

Policy Gradients

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

For each policy, define its value:

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

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

How can we do this?

Gradient ascent on policy parameters!

Reinforcement learning

Intuition

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

Interpretation:

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

Might seem simplistic to say that if a trajectory is good then all its actions were good. But in expectation, it averages out!

However, this also suffers from high variance because credit assignment is really hard. Can we help the estimator?

Variance reduction

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

First idea: Push up probabilities of an action seen, only by the cumulative future reward from that state

$$\nabla_\theta J(\theta) \approx \sum_{t \geq 0} \left(\sum_{t' \geq t} r_{t'} \right) \nabla_\theta \log \pi_\theta(a_t|s_t)$$

Second idea: Use discount factor γ to ignore delayed effects

$$\nabla_\theta J(\theta) \approx \sum_{t \geq 0} \left(\sum_{t' \geq t} \gamma^{t'-t} r_{t'} \right) \nabla_\theta \log \pi_\theta(a_t|s_t)$$

Variance reduction: Baseline

Problem: The raw value of a trajectory isn't necessarily meaningful. For example, if rewards are all positive, you keep pushing up probabilities of actions.

What is important then? Whether a reward is better or worse than what you expect to get

Idea: Introduce a baseline function dependent on the state.
Concretely, estimator is now:

$$\nabla_\theta J(\theta) \approx \sum_{t \geq 0} \left(\sum_{t' \geq t} \gamma^{t'-t} r_{t'} - b(s_t) \right) \nabla_\theta \log \pi_\theta(a_t|s_t)$$

A simple baseline: constant moving average of rewards experienced so far from all trajectories

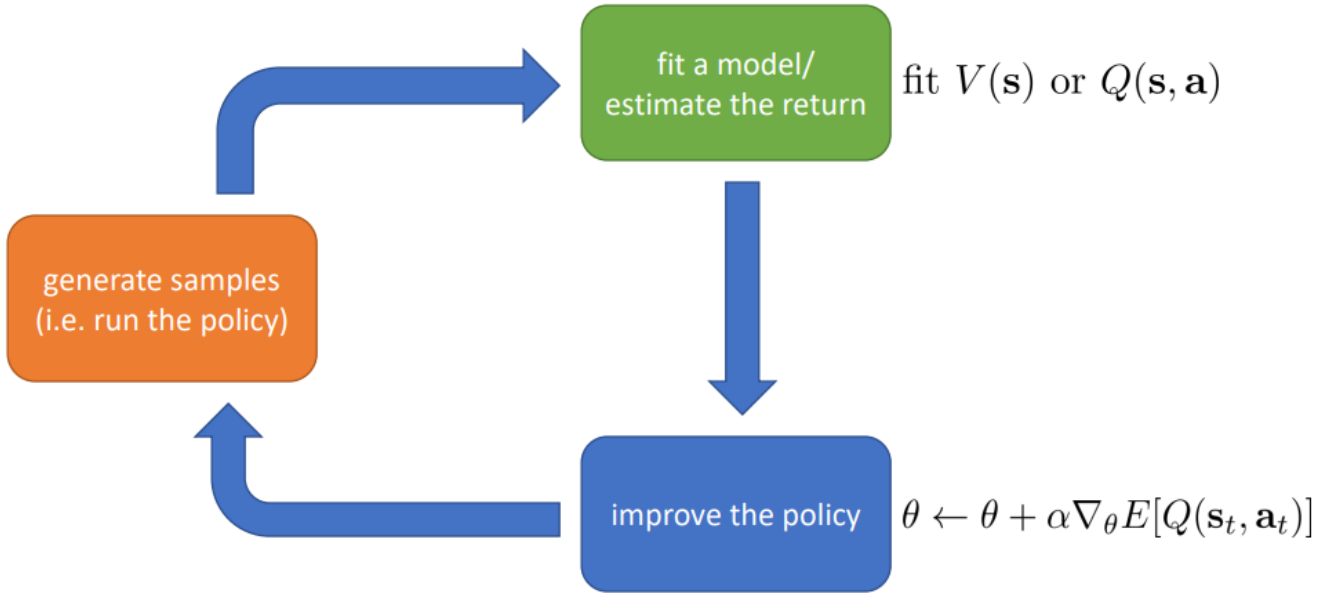
How to choose baseline?

- A better baseline: Want to push up the probability of an action from a state, if this action was better than the expected value of what we should get from that state.

➔Q-function and value function!

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

- Problem: we don't know Q and V. Can we learn them?



Actor-Critic Algorithm

- Yes, using Q-learning! We can combine Policy Gradients and Q-learning by training both an actor (the policy) and a critic (the Q-function).
  - The actor decides which action to take, and the critic tells the actor how good its action was and how it should adjust
  - Also 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

Initialize policy parameters θ, critic parameters ϕ
For iteration=1, 2 … **do**
  Sample m trajectories under the current policy
  $\Delta\theta \leftarrow 0$
  For i=1, …, m **do**
    For t=1, … , T **do**

$$A_t = \sum_{t' \geq t} \gamma^{t'-t} r_t^i - V_\phi(s_t^i)$$

$$\Delta\theta \leftarrow \Delta\theta + A_t \nabla_\theta \log(a_t^i | s_t^i)$$

  $\Delta\phi \leftarrow \sum_i \sum_t \nabla_\phi ||A_t^i||^2$
  $\theta \leftarrow \alpha\Delta\theta$
  $\phi \leftarrow \beta\Delta\phi$
End for

REINFORCE in action: Recurrent Attention Model (RAM)

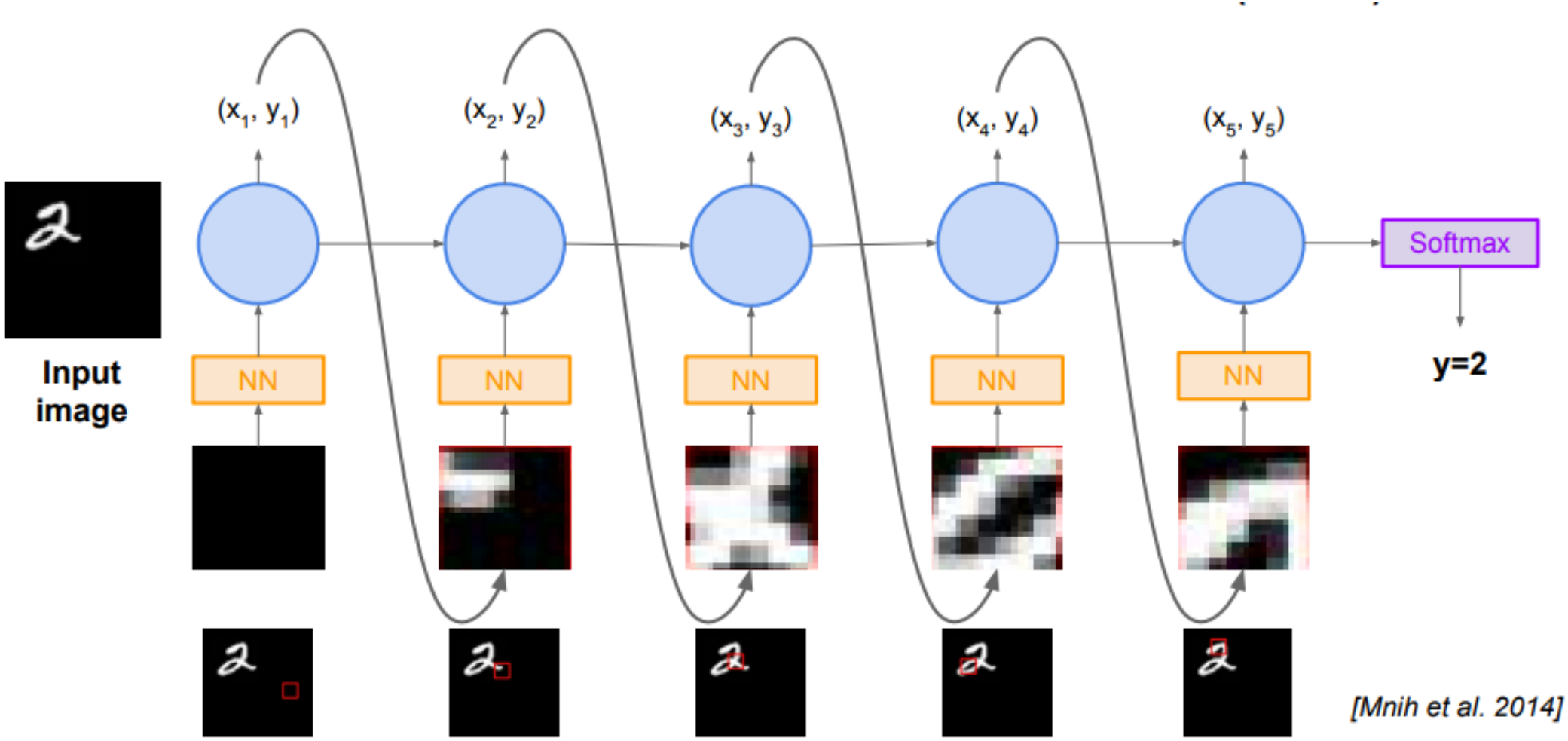
- Objective: Image Classification
- Take a sequence of "glimpses" selectively focusing on regions of the image, to predict class
  - Inspiration from human perception and eye movements
  - Saves computational resources
  - scalability
- Able to ignore clutter / irrelevant parts of image
  - State: Glimpses seen so far
  - Action: (x,y) coordinates (center of glimpse) of where to look next in image
  - Reward: 1 at the final timestep if image correctly classified, 0 otherwise
- Glimpsing is a non-differentiable operation => learn policy for how to take glimpse actions using REINFORCE Given state of glimpses seen so far, use RNN to model the state and output next action

[Mnih et al. 2014]

Why so many RL algorithms?

- Different tradeoffs
  - Sample efficiency
  - Stability & ease of use
- Different assumptions
  - Stochastic or deterministic?
  - Continuous or discrete?
  - Episodic or infinite horizon?
- Different things are easy or hard in different settings
  - Easier to represent the policy?
  - Easier to represent the model?

REINFORCE in action: Recurrent Attention Model (RAM)

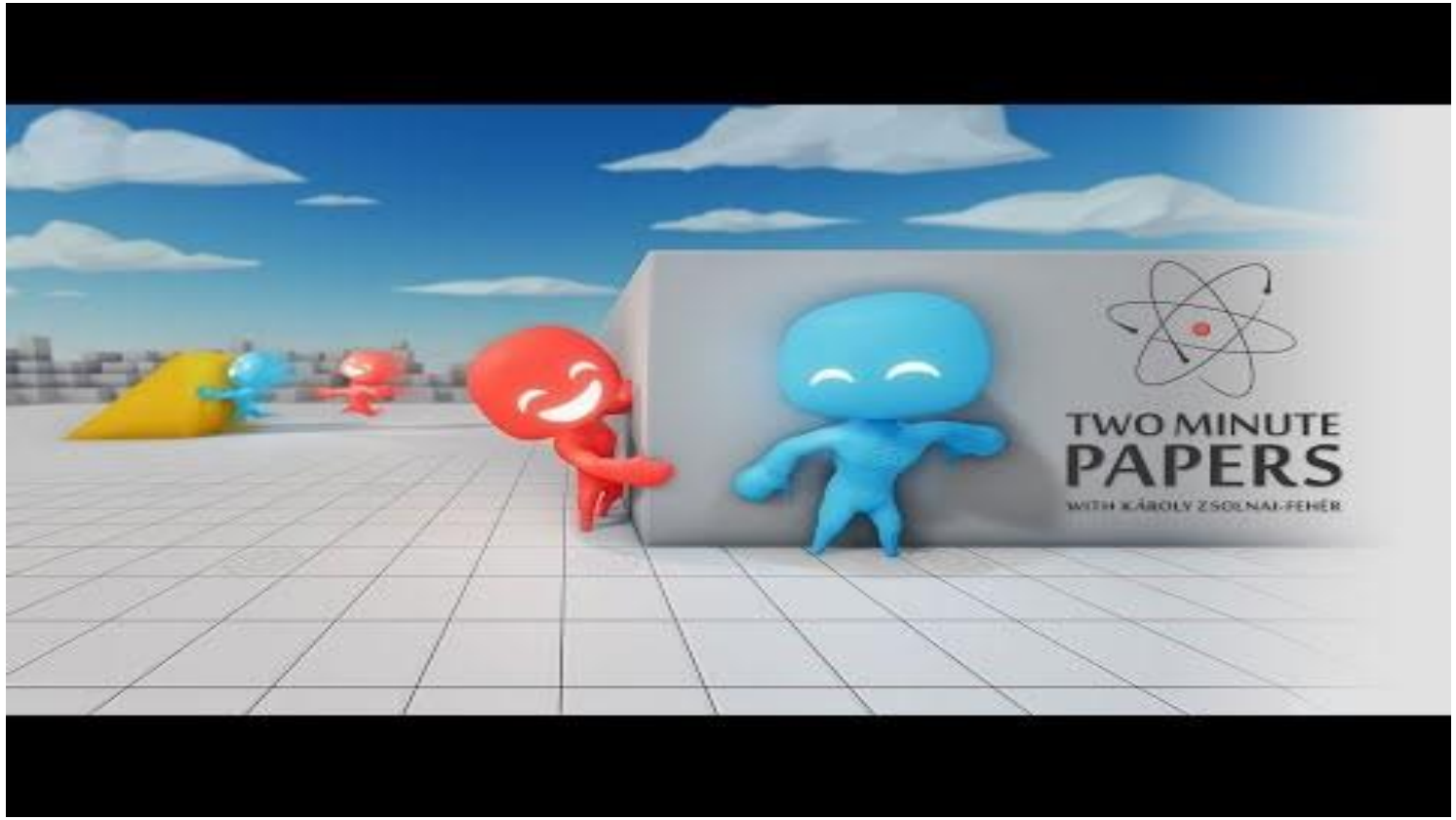


Summary

- **Policy gradients**: very general but suffer from high variance so requires a lot of samples. Challenge: sample-efficiency
- **Q-learning**: does not always work but when it works, usually more sample-efficient. Challenge: exploration

- Guarantees:
  - Policy Gradients: Converges to a local minima of $J(\xi)$, often good enough!
  - Q-learning: Zero guarantees since you are approximating Bellman equation with a complicated function approximator

See how smart they are!



Official website: https://openai.com/blog/emergent-tool-use/

1:13

- $Q^*(s,a) = max_\pi E[\sum_{t\geq 0} \gamma^t r\_t \,|s_0 = s, a_0 = a, \pi]$