Deep Reinforcement Learning

M. Soleymani
Sharif University of Technology
Fall 2017

Slides are based on Fei Fei Li and colleagues lectures, cs231n, Stanford 2017
and some from Surguy Levin lectures, cs294-112, Berkeley 2016.

Supervised Learning

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

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

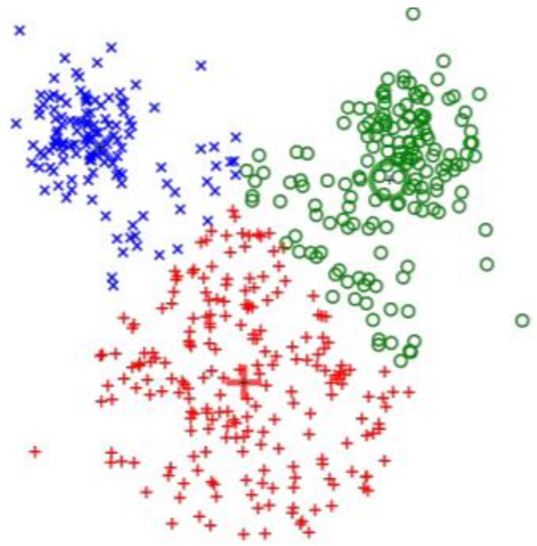
- Examples: Classification, regression, object detection, semantic segmentation, image captioning, etc.

Unsupervised Learning

- Data: x
  - Just data, no labels!

- Goal: Learn some underlying hidden structure of the data

- Examples: Clustering, dimensionality reduction, feature learning, density estimation, etc.



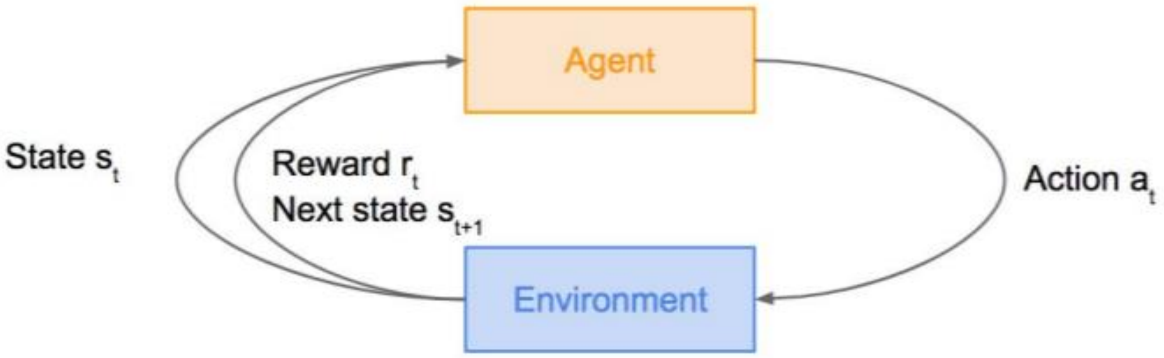
K-means clustering

Reinforcement Learning

- **Goal**: Learn how to take actions in order to maximize reward
  - Concerned with taking sequences of actions

- Described in terms of agent interacting with a previously unknown environment, trying to maximize cumulative reward

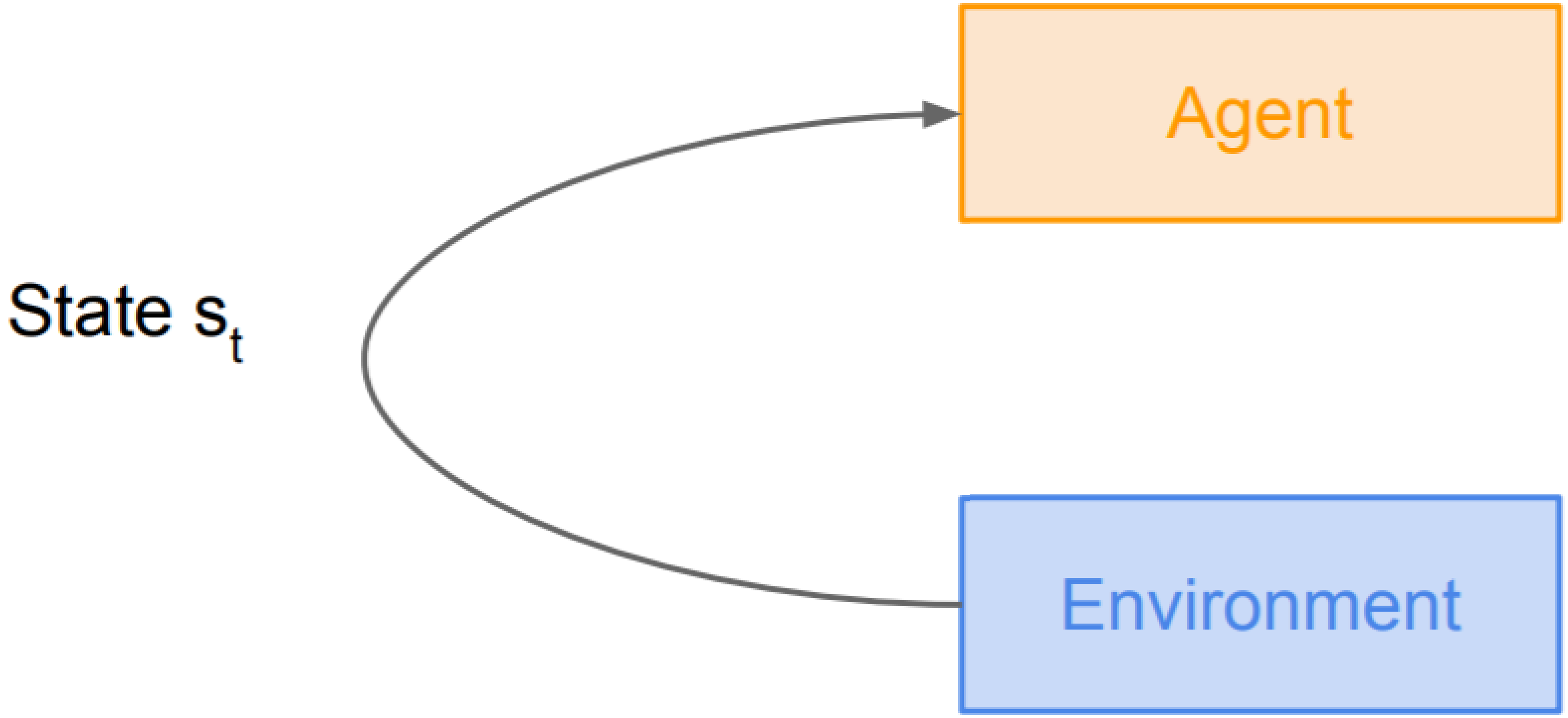
an **agent** interacting with an **environment**, which provides numeric **reward** signals



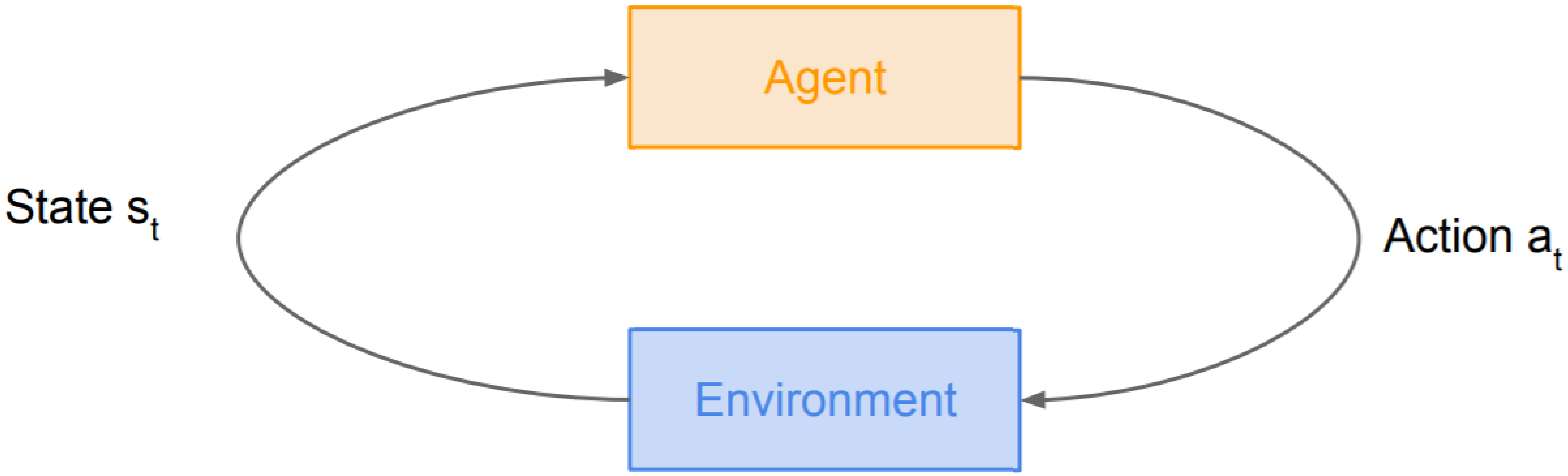
Overview

- What is Reinforcement Learning?
- Markov Decision Processes
- Q-Learning
- Policy Gradients

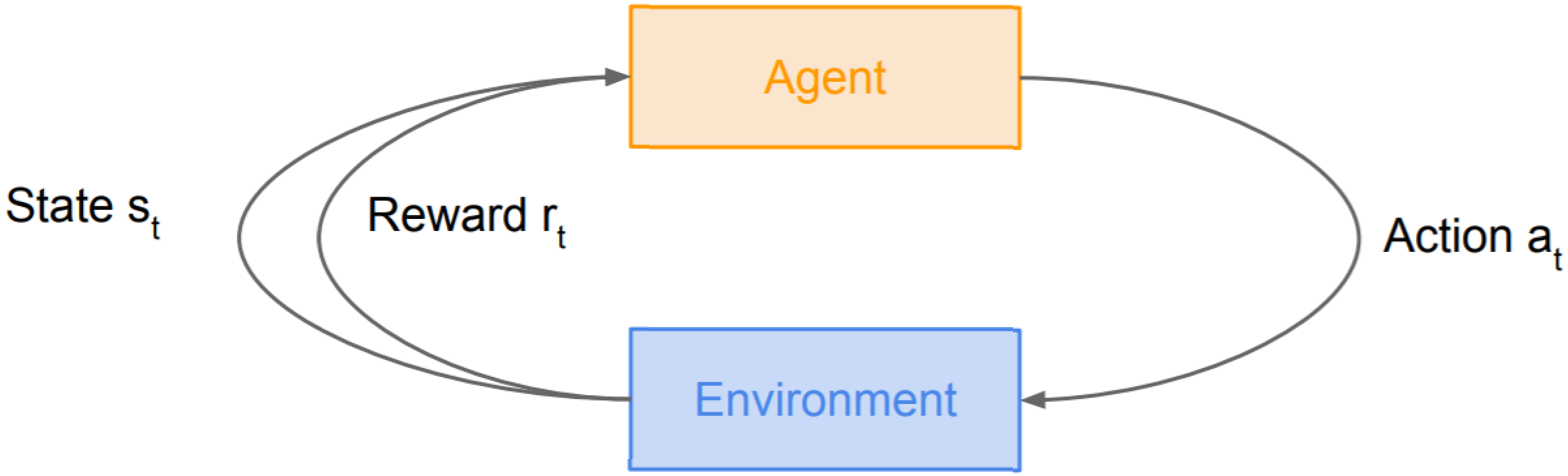
Reinforcement Learning



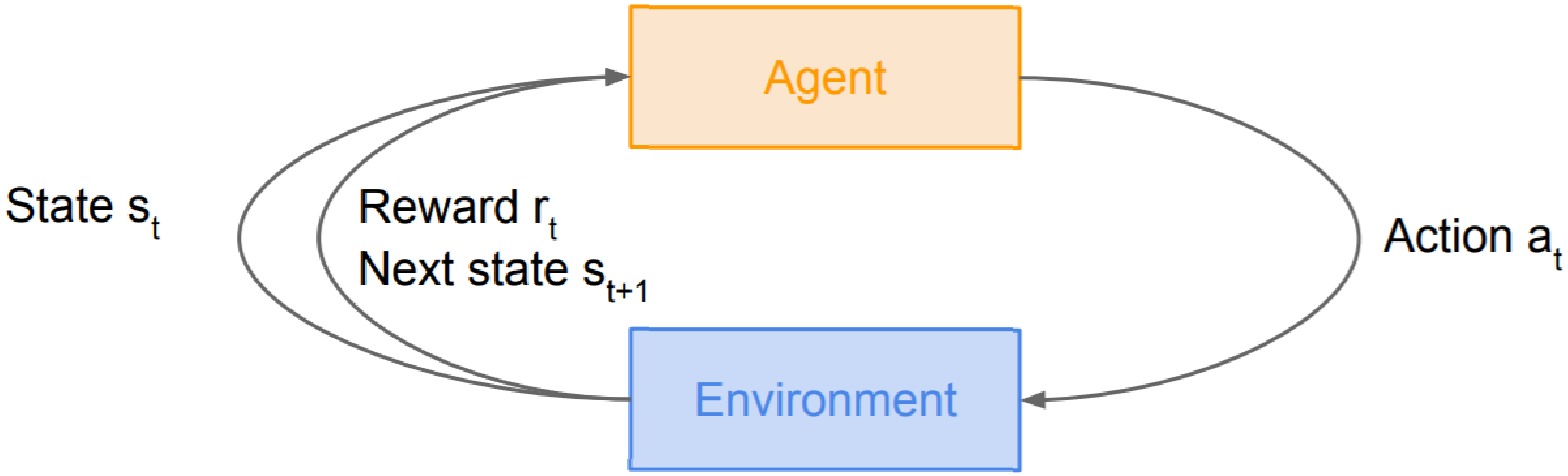
Reinforcement Learning



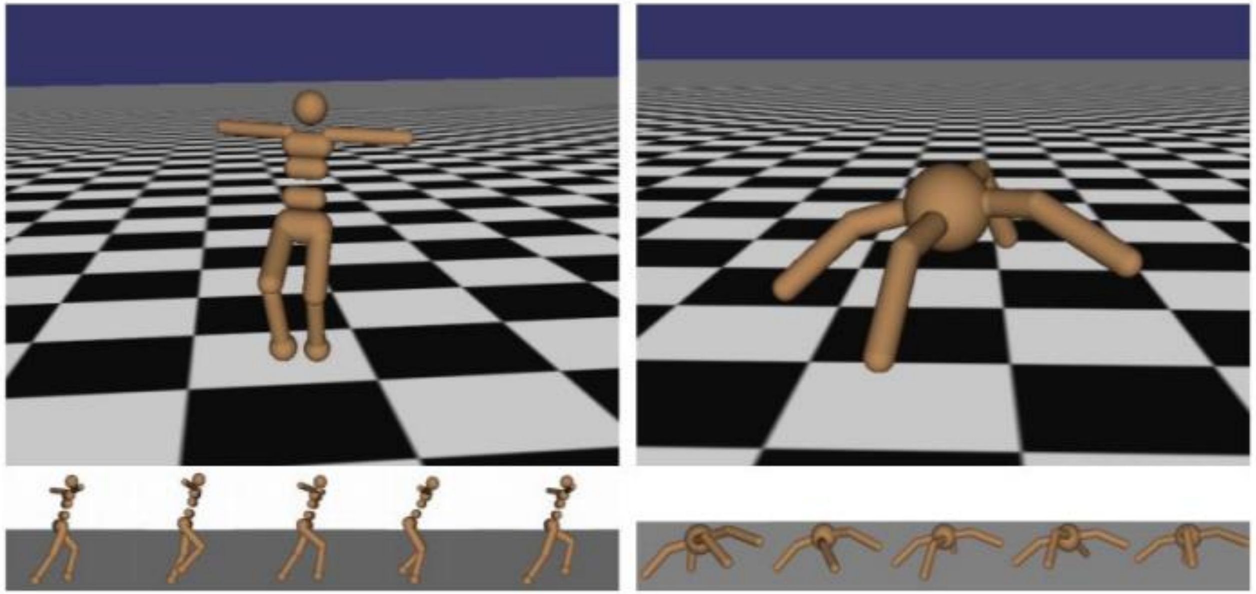
Reinforcement Learning



Reinforcement Learning



Robot Locomotion



Objective: Make the robot move forward

State: Angle and position of the joints
Action: Torques applied on joints
Reward: 1 at each time step upright + forward movement

Motor Control and Robotics

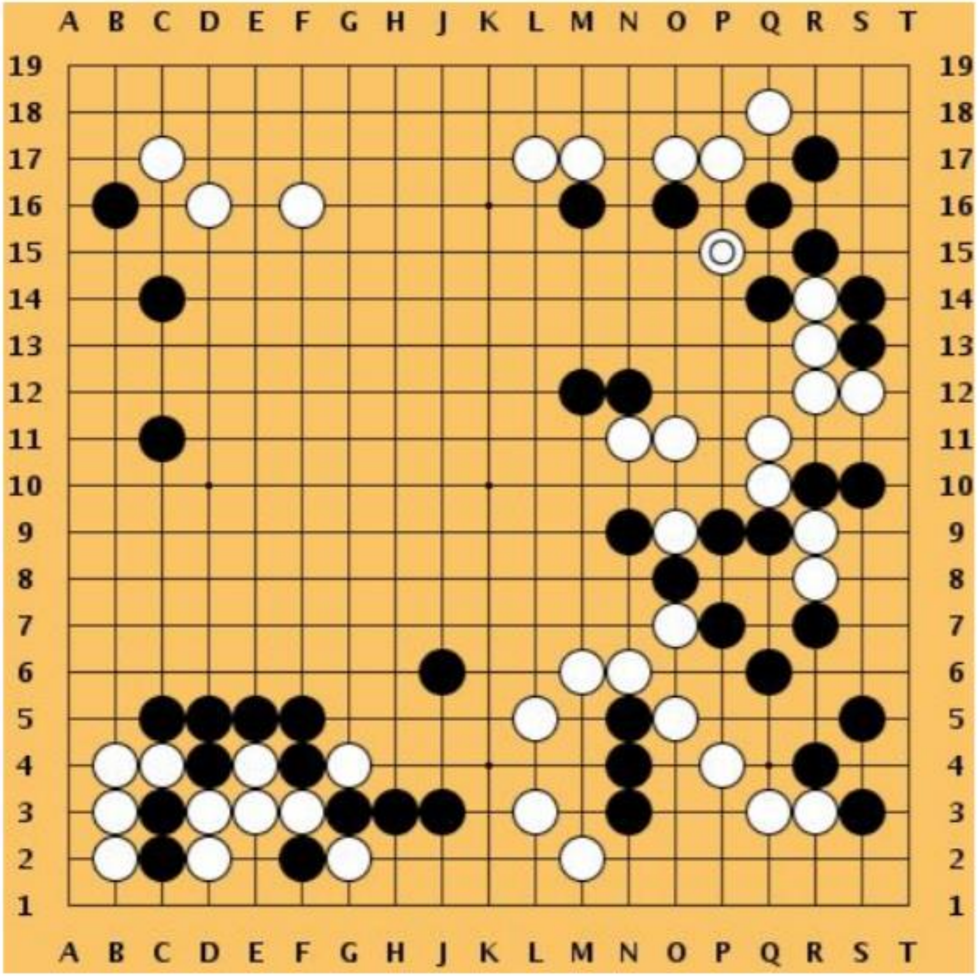
- Robotics:
  - Observations: camera images, joint angles
  - Actions: joint torques
  - Rewards: stay balanced, navigate to target locations, serve and protect humans

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

Go



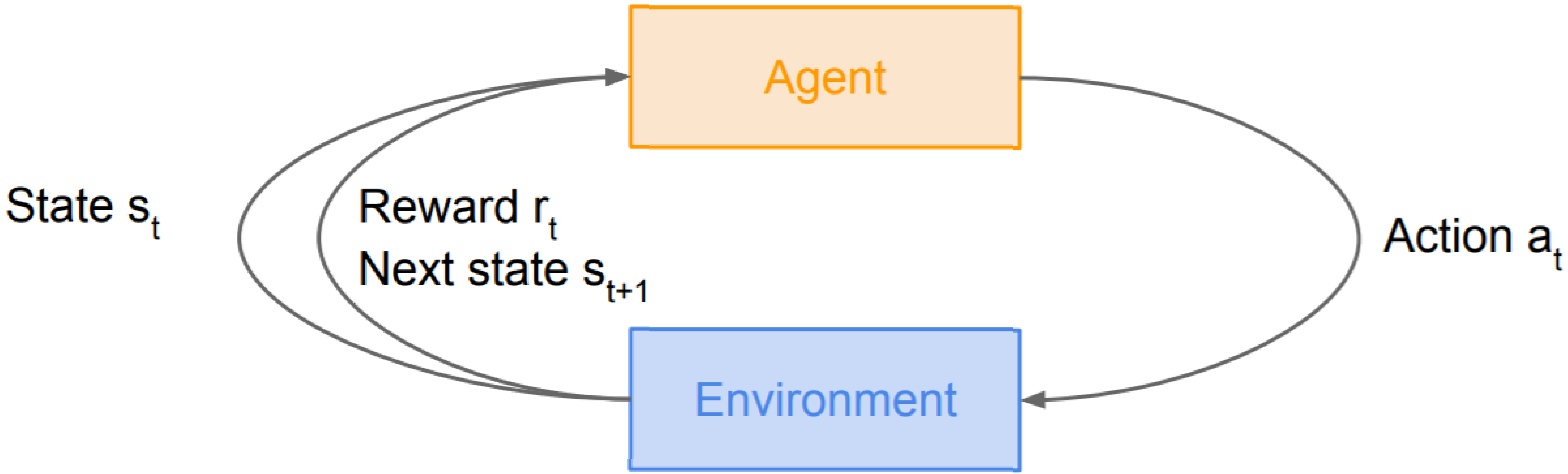
Objective: Win the game!

State: Position of all pieces
Action: Where to put the next piece down
Reward: 1 if win at the end of the game, 0 otherwise

How Does RL Relate to Other Machine Learning Problems?

- Differences between RL and supervised learning:
  - You don't have full access to the function you're trying to optimize
    - must query it through interaction.
  - Interacting with a stateful world: input x_t depend on your previous actions

How can we mathematically formalize the RL problem?



Markov Decision Process

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

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 $r_t \sim R(.\,|s_t, a_t)$
  - Environment samples next state $s_{t+1} \sim P(.\,|s_t, a_t)$
  - 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:

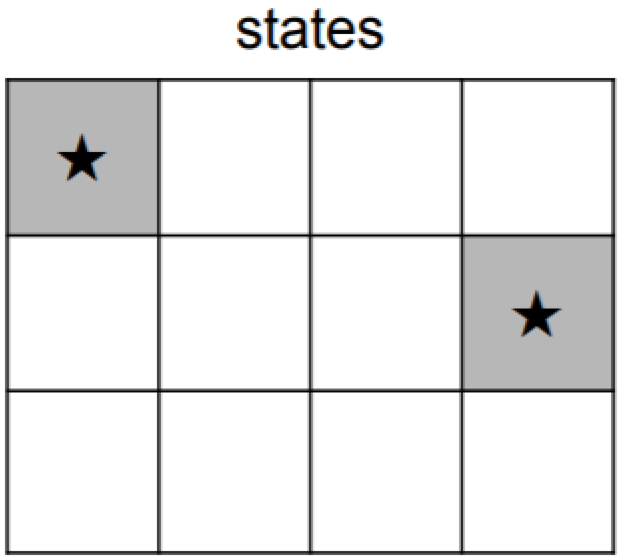
$$r_0 + \gamma r_1 + \gamma^2 r_2 + \cdots = \sum_{k=0}^{\infty} \gamma^k r_k$$

A simple MDP: Grid World

actions = {

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

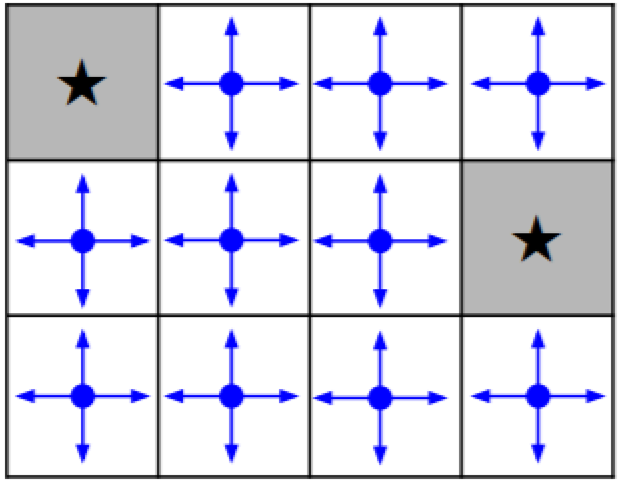
}



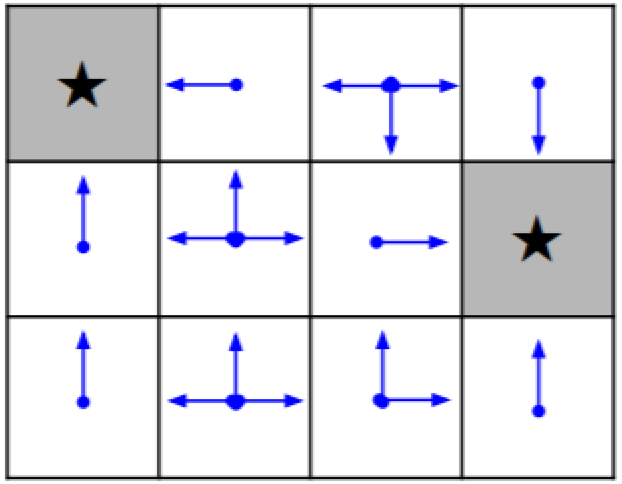
Set a negative "reward" for each transition (e.g. $r = -1$)

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

A simple MDP: Grid World



Random Policy



Optimal Policy

The optimal policy π^*

- We want to find **optimal policy** $\boldsymbol{\pi}^*$ that maximizes the sum of rewards.
- How do we handle the randomness (initial state, transition probability…)?
  - Maximize the expected sum of rewards!

$$\pi^* = \arg\max_{\pi} \mathbb{E}\left[\sum_{t\geq 0} \gamma^t r_t | \pi\right]$$

$$s_0 \sim p(s_0)$$

$$a_t \sim \pi(\cdot|s_t)$$

$$s_{t+1} \sim p(\cdot|s_t, a_t)$$

Definitions: Value function and Q-value function

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:

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

Value function for policy π

$$V^{\pi}(s) = E\{\sum_{k=0}^{\infty} \gamma^k r_t \,|s_0 = s, \pi\}$$

$$Q^{\pi}(s,a) = E\{\sum_{k=0}^{\infty} \gamma^k r_t \,|s_0 = s, a_0 = a \,, \pi\,\}$$

- $V^{\pi}(s)$: How good for the agent to be in the state s when its policy is π
  - It is simply the expected sum of discounted rewards upon starting in state s and taking actions according to π

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

Bellman Equations

$$Q^{\pi}(s,a) = E[r + \gamma Q^{\pi}(s',a')|s, a, \pi]$$

Bellman optimality equation

$$V^*(s) = \max_{a \in \mathcal{A}(s)} E[r + \gamma V^*(s')|s, a]$$

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

Bellman equation

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 $r + \gamma Q^*(s', a')$

Optimal policy

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

- It can also be computed as:

$$\pi^*(s) = \underset{a \in \mathcal{A}(s)}{\operatorname{argmax}}\, Q^*(s, a)$$

Solving for the optimal policy: Value iteration

Value iteration algorithm: Use Bellman equation as an iterative update

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

Q_i will converge to Q^* as i -> infinity

Solving for the optimal policy: Q-learning algorithm

Initialize $\hat{Q}(s, a)$ arbitrarily

Repeat (for each episode):

Initialize s

Repeat (for each step of episode):

e.g., greedy, ε-greedy

Choose a from s using a policy derived from $\hat{Q}$

Take action a, receive reward r, observe new state s'

$$\hat{Q}(s, a) \leftarrow \hat{Q}(s, a) + \alpha \left[r + \gamma \max_{a'} \hat{Q}(s', a') - \hat{Q}(s, a) \right]$$

$s \leftarrow s'$

until s is terminal

Problem

- Not scalable.
  - Must compute Q(s,a) for every state-action pair.
    - it computationally infeasible to compute for entire state space!

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

Solving for the optimal policy: Q-learning

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

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

function parameters (weights)

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

Iteratively try to make the Q-value close to the target value (yi) it should have (according to Bellman Equations).

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

Backward Pass

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

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

[Mnih et al., Playing Atari with Deep Reinforcement Learning, NIPS Workshop 2013; Nature 2015]

Case Study: Playing Atari Games (seen before)

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

[Mnih et al., Playing Atari with Deep Reinforcement Learning, NIPS Workshop 2013; Nature 2015]

Q-network Architecture

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

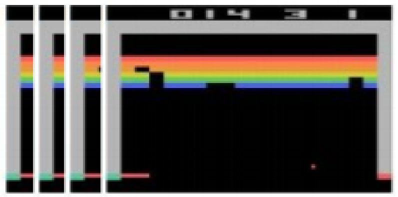
A single feedforward pass to compute Q-values for all actions from the current state => efficient!

FC-4 (Q-values)

FC-256

32 4x4 conv, stride 2

16 8x8 conv, stride 4



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

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

Number of actions between 4-18 depending on Atari game

[Mnih et al., Playing Atari with Deep Reinforcement Learning, NIPS Workshop 2013; Nature 2015]

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
    - 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})
  - Train Q-network on random minibatches of transitions from the replay memory

✓ Each transition can also contribute to multiple weight updates => greater data efficiency

✓ Smoothing out learning and avoiding oscillations or divergence in the parameters

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

[Mnih et al., Playing Atari with Deep Reinforcement Learning, NIPS Workshop 2013; Nature 2015]

Putting it together: Deep Q-Learning with Experience Replay

Algorithm 1 Deep Q-learning with Experience Replay

$\boxed{\begin{array}{l}\text{Initialize replay memory } \mathcal{D} \text{ to capacity } N \\ \text{Initialize action-value function } Q \text{ with random weights}\end{array}}$ → Initialize replay memory, Q-network

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

[Mnih et al., Playing Atari with Deep Reinforcement Learning, 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** — Play M episodes (full games)
  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

[Mnih et al., Playing Atari with Deep Reinforcement Learning, 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

Initialize state (starting game screen pixels) at the beginning of each episode

[Mnih et al., Playing Atari with Deep Reinforcement Learning, 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** — For each time-step of game
        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

[Mnih et al., Playing Atari with Deep Reinforcement Learning, 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

With small probability, select a random action (explore), otherwise select greedy action from current policy

[Mnih et al., Playing Atari with Deep Reinforcement Learning, 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

Take the selected action observe
the reward and next state

[Mnih et al., Playing Atari with Deep Reinforcement Learning, 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 $\mathcal{D}$ → Store transition in replay memory
    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

[Mnih et al., Playing Atari with Deep Reinforcement Learning, NIPS Workshop 2013; Nature 2015]

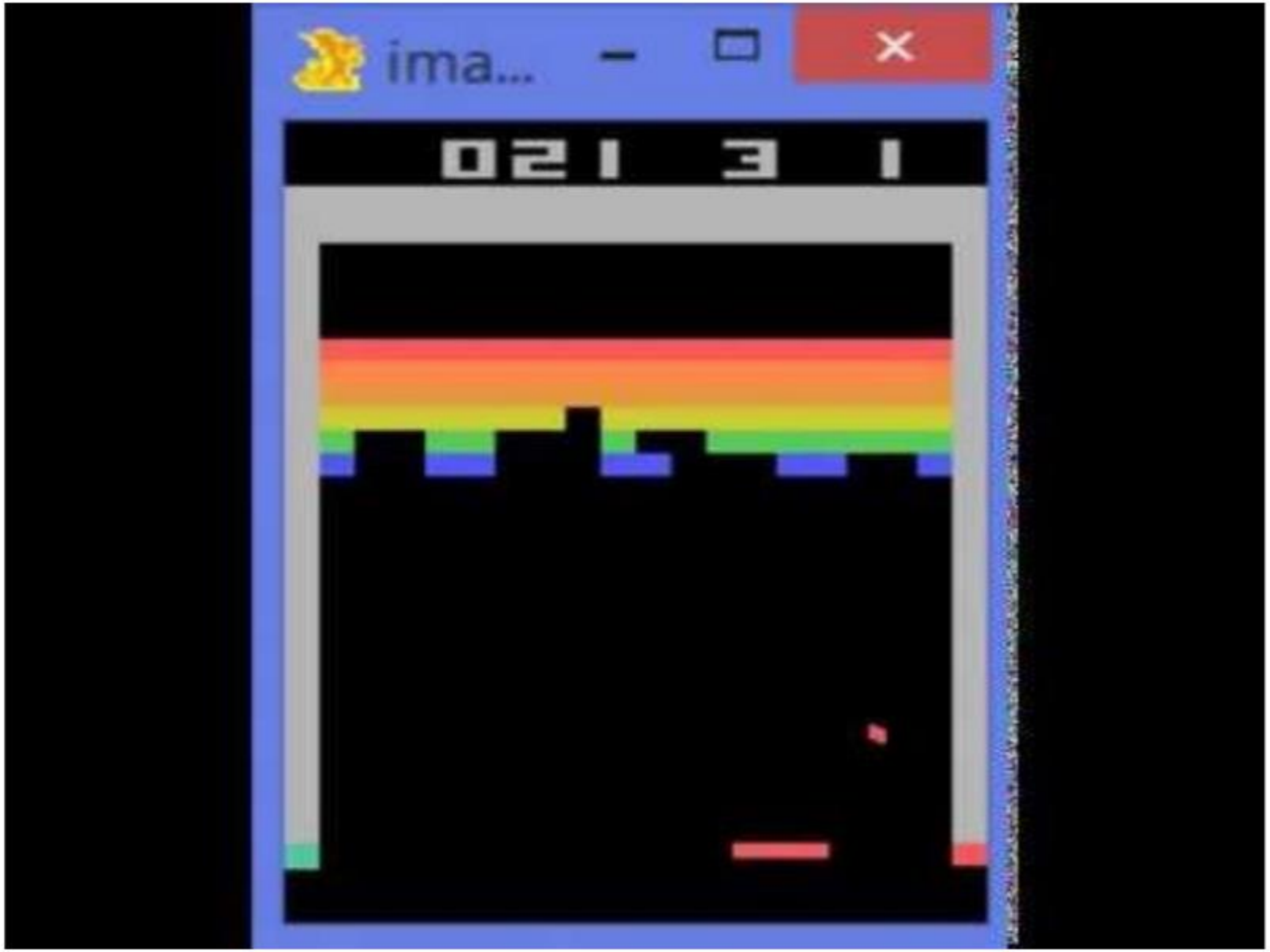
Putting it together: Deep Q-Learning with Experience Replay

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

Sample a random minibatch of transitions and perform a gradient descent step

[Mnih et al., Playing Atari with Deep Reinforcement Learning, NIPS Workshop 2013; Nature 2015]



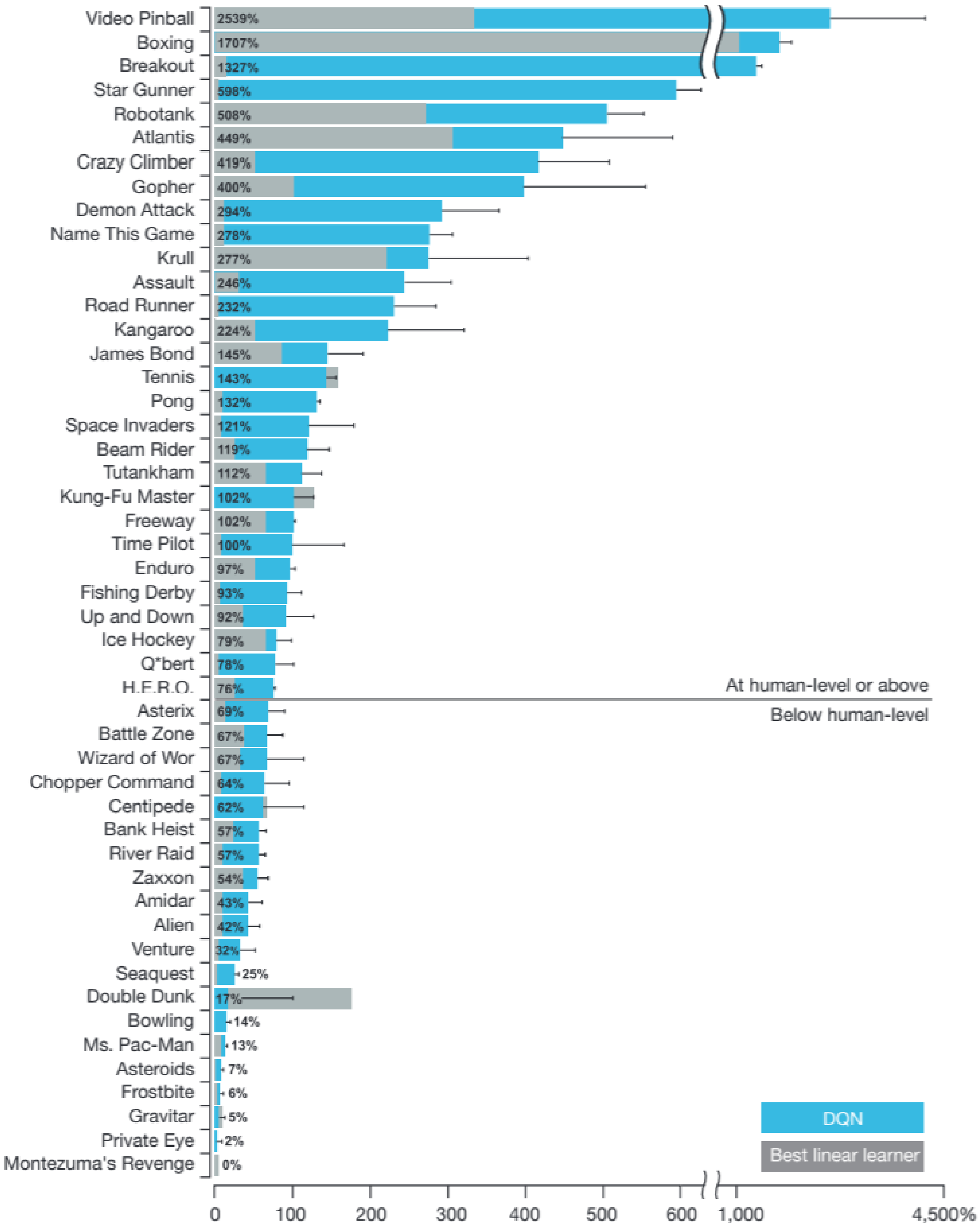
https://www.youtube.com/watch?v=V1eYniJ0Rnk

Video by Károly Zsolnai-Fehér. Reproduced with permission.

Results on 49 Games

- The architecture and hyperparameter values were the same for all 49 games.

- DQN achieved performance comparable to or better than an experienced human on 29 out of 49 games.



[V. Mnih et al., Human-level control through deep reinforcement learning, Nature 2015]

Policy Gradients

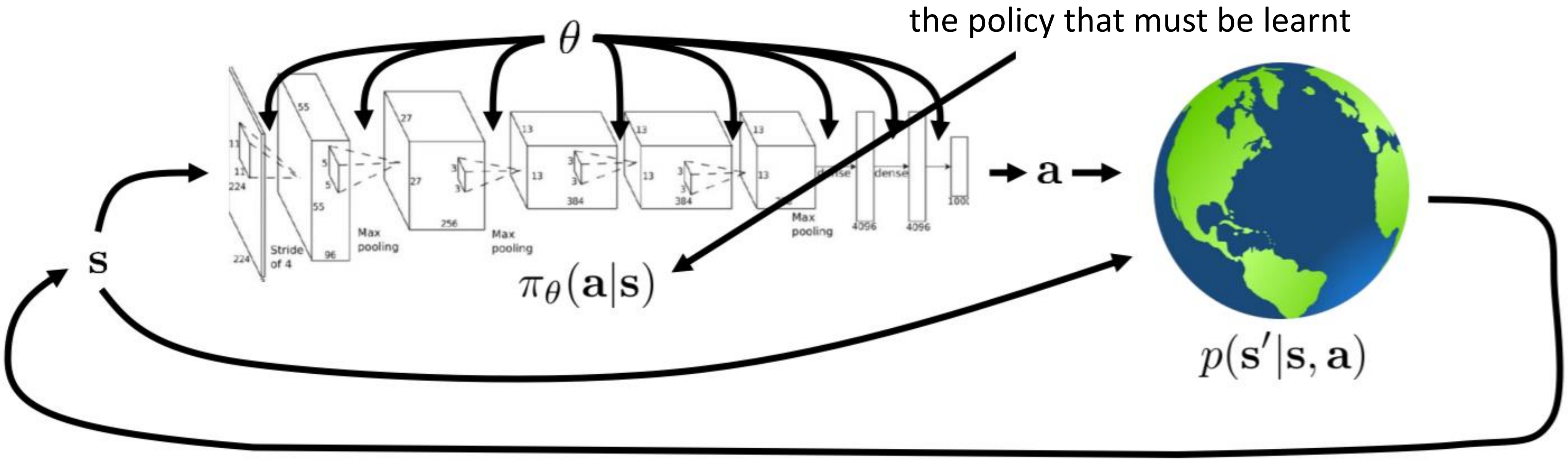
- What is a problem with Q-learning?
  - The Q-function can be very complicated!

- Hard to learn exact value of every (state, action) pair

- But the policy can be much simple

- Can we learn a policy directly, e.g. finding the best policy from a collection of policies?

The goal of RL



$$\underbrace{p_\theta(\mathbf{s}_1,\mathbf{a}_1,\ldots,\mathbf{s}_T,\mathbf{a}_T)}_{p_\theta(\tau)} = p(\mathbf{s}_1)\prod_{t=1}^{T}\pi_\theta(\mathbf{a}_t|\mathbf{s}_t)p(\mathbf{s}_{t+1}|\mathbf{s}_t,\mathbf{a}_t)$$

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

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!

REINFORCE algorithm

Mathematically, we can write:

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

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

REINFORCE algorithm

Expected reward: $$J(\theta) = \mathbb{E}_{\tau \sim p(\tau;\theta)} [r(\tau)]$$
$$= \int_\tau r(\tau) p(\tau;\theta) \mathrm{d}\tau$$

Now let's differentiate this: $$\nabla_\theta J(\theta) = \int_\tau r(\tau) \nabla_\theta p(\tau;\theta) \mathrm{d}\tau$$

Intractable! Gradient of an expectation is problematic when p depends on θ

REINFORCE algorithm

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

$$= \int_\tau r(\tau)p(\tau;\theta)\mathrm{d}\tau$$

Now let's differentiate this: $\nabla_\theta J(\theta) = \int_\tau r(\tau)\nabla_\theta p(\tau;\theta)\mathrm{d}\tau$

However, we can use a nice trick: $\nabla_\theta p(\tau;\theta) = p(\tau;\theta)\frac{\nabla_\theta p(\tau;\theta)}{p(\tau;\theta)} = p(\tau;\theta)\nabla_\theta \log p(\tau;\theta)$

$$\nabla_\theta J(\theta) = \int_\tau r(\tau) \nabla_\theta p(\tau; \theta) \mathrm{d}\tau$$

$$\nabla_\theta p(\tau; \theta) = p(\tau; \theta) \frac{\nabla_\theta p(\tau; \theta)}{p(\tau; \theta)} = p(\tau; \theta) \nabla_\theta \log p(\tau; \theta)$$

$$\begin{aligned} \Rightarrow \nabla_\theta J(\theta) &= \int_\tau \left(r(\tau) \nabla_\theta \log p(\tau; \theta) \right) p(\tau; \theta) \mathrm{d}\tau \\ &= \mathbb{E}_{\tau \sim p(\tau; \theta)} \left[r(\tau) \nabla_\theta \log p(\tau; \theta) \right] \end{aligned}$$

Can estimate with Monte Carlo sampling

REINFORCE algorithm

Can we compute those quantities without knowing the transition probabilities?

We have: $p(\tau;\theta) = \prod_{t\geq 0} p(s_{t+1}|s_t, a_t)\pi_\theta(a_t|s_t)$

Thus: $\log p(\tau;\theta) = \sum_{t\geq 0} \log p(s_{t+1}|s_t, a_t) + \log \pi_\theta(a_t|s_t)$

And when differentiating: $\nabla_\theta \log p(\tau;\theta) = \sum_{t\geq 0} \nabla_\theta \log \pi_\theta(a_t|s_t)$

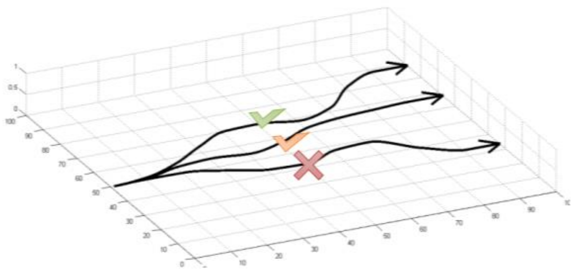
Doesn't depend on transition probabilities!

REINFORCE algorithm

$$\begin{aligned}\nabla_\theta J(\theta) &= \int_\tau \left(r(\tau)\nabla_\theta \log p(\tau;\theta)\right) p(\tau;\theta)\mathrm{d}\tau \\ &= \mathbb{E}_{\tau\sim p(\tau;\theta)}\left[r(\tau)\nabla_\theta \log p(\tau;\theta)\right]\end{aligned}$$

$$\nabla_\theta \log p(\tau;\theta) = \sum_{t\geq 0} \nabla_\theta \log \pi_\theta(a_t|s_t)$$

$$\nabla_\theta J(\theta) \approx \frac{1}{N}\sum_{n=1}^{N} r(\tau^{(n)})\, \nabla_\theta \log p\left(\tau^{(n)};\theta\right)$$



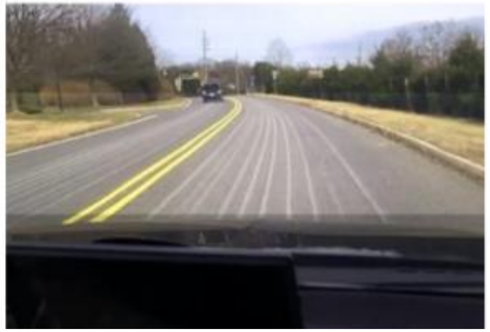
$$\nabla_\theta J(\theta) \approx \frac{1}{N}\sum_{n=1}^{N}\sum_{t\geq 0} r\left(s_t^{(n)}, a_t^{(n)}\right)\sum_{t\geq 0}\nabla_\theta \log \pi_\theta\left(a_t^{(n)}|s_t^{(n)}\right)$$

REINFORCE Algorithm

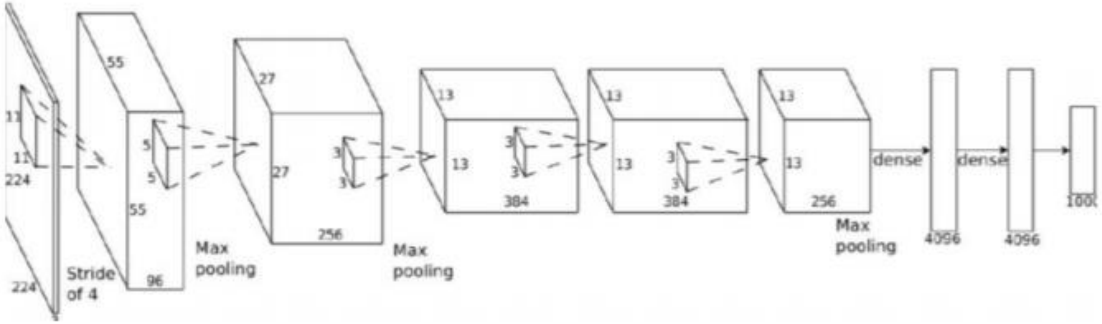
- Repeat
  - Sample $\{\tau^{(n)}\}$ from $\pi_\theta(a|s)$ (run the policy)
  - $\nabla_\theta J(\theta) \approx \frac{1}{N}\sum_{n=1}^{N}\sum_{t\geq 0} r\left(s_t^{(n)}, a_t^{(n)}\right)\sum_{t\geq 0}\nabla_\theta \log \pi_\theta\left(a_t^{(n)}|s_t^{(n)}\right)$
  - $\theta \leftarrow \theta + \alpha\nabla_\theta J(\theta)$

Evaluating the policy gradient

$$\nabla_\theta J(\theta) \approx \frac{1}{N}\sum_{n=1}^{N} \underbrace{\sum_{t\geq 0} r\left(s_t^{(n)}, a_t^{(n)}\right)}_{r\left(\tau^{(n)}\right)} \sum_{t\geq 0} \nabla_\theta \log \pi_\theta\left(a_t^{(n)} | s_t^{(n)}\right)$$

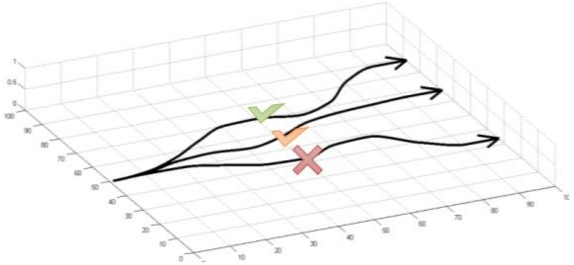


s_t



$\pi_\theta(a_t|s_t)$

a_t

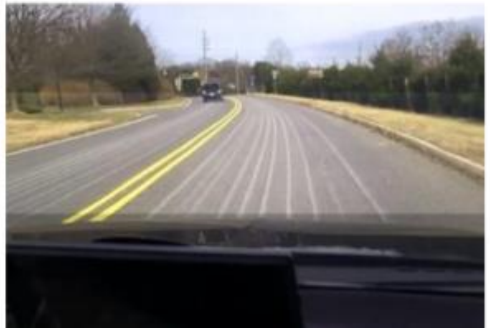


- Policy gradient:

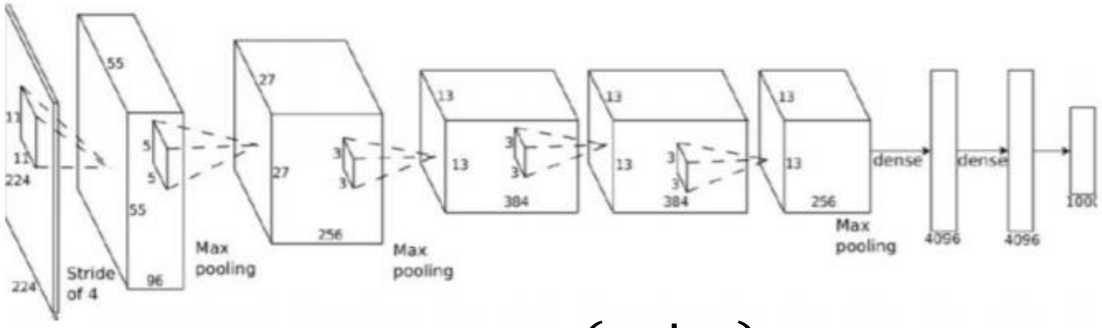
$$\nabla_\theta J(\theta) \approx \frac{1}{N} \sum_{n=1}^{N} \underbrace{\sum_{t\geq 0} r\left(s_t^{(n)}, a_t^{(n)}\right)}_{r\left(\tau^{(n)}\right)} \sum_{t\geq 0} \nabla_\theta \log \pi_\theta\left(a_t^{(n)} | s_t^{(n)}\right)$$

- Maximum Likelihood:

$$\nabla_\theta J(\theta) \approx \frac{1}{N} \sum_{n=1}^{N} \sum_{t\geq 0} \nabla_\theta \log \pi_\theta\left(a_t^{(n)} | s_t^{(n)}\right)$$



s_t

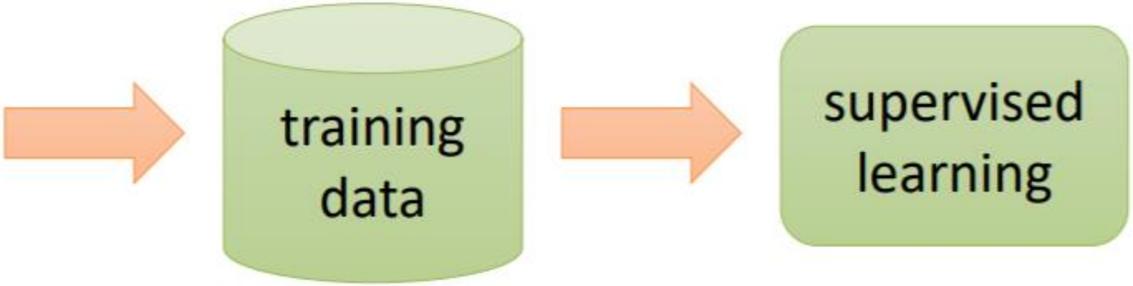


$\pi_\theta(a_t|s_t)$

a_t

$\mathbf{s}_t$

$\mathbf{a}_t$



$\pi_\theta(a_t|s_t)$

Intuition

$$\nabla_\theta J(\theta) \approx \frac{1}{N}\sum_{n=1}^{N} r\left(\tau^{(n)}\right)\sum_{t\geq 0}\nabla_\theta \log \pi_\theta\left(a_t^{(n)}|s_t^{(n)}\right)$$

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.

What did we just do?

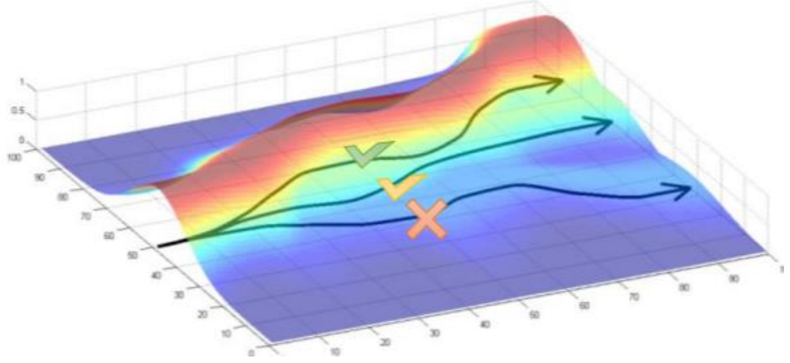
$$\nabla_\theta J(\theta) \approx \frac{1}{N}\sum_{n=1}^{N} r\left(\tau^{(n)}\right)\underbrace{\nabla_\theta \log p_\theta\left(\tau^{(n)}\right)}_{\sum_{t\geq 0}\nabla_\theta \log \pi_\theta\left(a_t^{(n)}|s_t^{(n)}\right)}$$

$$\nabla_\theta J_{ML}(\theta) \approx \frac{1}{N}\sum_{n=1}^{N} \nabla_\theta \log p_\theta\left(\tau^{(n)}\right)$$

good stuff is made more likely

bad stuff is made less likely

simply formalizes the notion of "trial and error"!



Reducing variance

$$\nabla_\theta J(\theta) \approx \frac{1}{N} \sum_{n=1}^{N} \sum_{t \geq 0} r\left(s_t^{(n)}, a_t^{(n)}\right) \sum_{t \geq 0} \nabla_\theta \log \pi_\theta \left(a_t^{(n)} | s_t^{(n)}\right)$$

- Causality:

policy at time t' cannot affect reward at time t when $t < t'$

$$\nabla_\theta J(\theta) \approx \frac{1}{N} \sum_{n=1}^{N} \sum_{t \geq 0} \left(\sum_{t' \geq t} r\left(s_{t'}^{(n)}, a_{t'}^{(n)}\right) \right) \nabla_\theta \log \pi_\theta \left(a_t^{(n)} | s_t^{(n)}\right)$$

Variance reduction

Gradient estimator: $$\nabla_\theta J(\theta) \approx \frac{1}{N}\sum_{n=1}^{N}\sum_{t\geq 0} r\left(s_t^{(n)}, a_t^{(n)}\right)\sum_{t\geq 0}\nabla_\theta \log \pi_\theta\left(a_t^{(n)}|s_t^{(n)}\right)$$

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

$$\nabla_\theta J(\theta) \approx \frac{1}{N}\sum_{n=1}^{N}\sum_{t\geq 0}\sum_{t'\geq t} r\left(s_{t'}^{(n)}, a_{t'}^{(n)}\right)\nabla_\theta \log \pi_\theta\left(a_t^{(n)}|s_t^{(n)}\right)$$

Second idea: Use discount factor γ to ignore delayed effects

$$\nabla_\theta J(\theta) \approx \frac{1}{N}\sum_{n=1}^{N}\sum_{t\geq 0}\sum_{t'\geq t} \gamma^{t'-t} r\left(s_{t'}^{(n)}, a_{t'}^{(n)}\right)\nabla_\theta \log \pi_\theta\left(a_t^{(n)}|s_t^{(n)}\right)$$

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.

$$\nabla_\theta J(\theta) \approx \frac{1}{N} \sum_{n=1}^{N} \sum_{t \geq 0} \left(\sum_{t' \geq t} \gamma^{t'-t} r\left(s_{t'}^{(n)}, a_{t'}^{(n)}\right) - b\left(s_t^{(n)}\right) \right) \nabla_\theta \log \pi_\theta \left(a_t^{(n)} | s_t^{(n)}\right)$$

Simple baseline: $b = \frac{1}{N} \sum_{n=1}^{N} r\left(\tau^{(n)}\right)$ average reward is not the best baseline, but it's pretty good!

How to choose the baseline?

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

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

- Variance reduction techniques seen so far are typically used in “Vanilla REINFORCE”

Policy gradient in practice

- Remember that the gradient has high variance
  - This isn't the same as supervised learning!
  - Gradients will be really noisy!
- Consider using much larger batches
- Tweaking learning rates is very hard
  - Adaptive step size rules like ADAM can be OK-ish
  - policy gradient-specific learning rate adjustment methods

REINFORCE Algorithm

- Repeat
  - Sample $\{\tau^{(n)}\}$ from $\pi_\theta(a_t|s_t)$ (run the policy)
  - $\nabla_\theta J(\theta) \approx \frac{1}{N}\sum_{n=1}^{N}\sum_{t\geq 0}\gamma^{t'-t}r\left(s_t^{(n)}, a_t^{(n)}\right)\sum_{t\geq 0}\nabla_\theta \log \pi_\theta\left(a_t^{(n)}|s_t^{(n)}\right)$
  - $\theta \leftarrow \theta + \alpha\nabla_\theta J(\theta)$

$$\nabla_\theta J(\theta) \approx \frac{1}{N}\sum_{n=1}^{N}\sum_{t\geq 0}\underbrace{\left(\sum_{t'\geq t}\gamma^{t'-t}r\left(s_{t'}^{(n)}, a_{t'}^{(n)}\right)\right)}_{\text{Reward to go } Q_t^{(n)}}\nabla_\theta \log \pi_\theta\left(a_t^{(n)}|s_t^{(n)}\right)$$

How to choose the baseline?

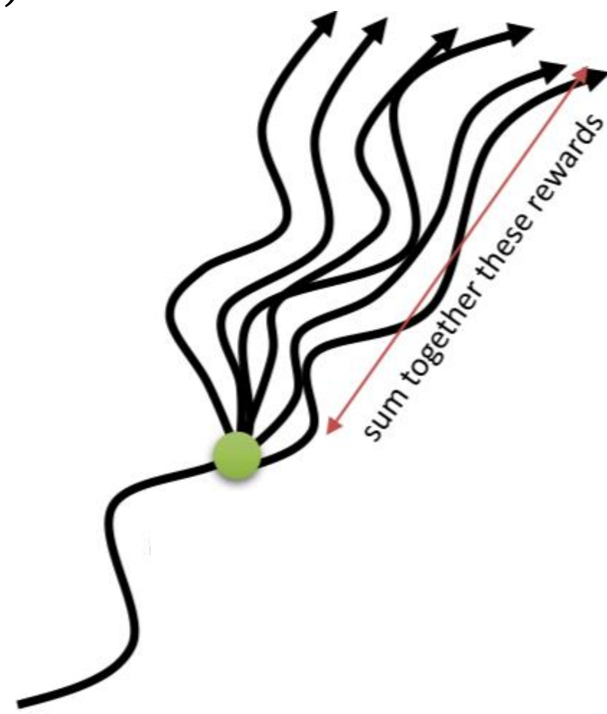
- A better baseline (to push up the probability of an action from a state):
  - if this action was better than the expected value of what we should get from that state.

$$Q^{\pi}(s_t, a_t) - V^{\pi}(s_t)$$

- We are happy with an action a_t in a state s_t if it is large
- we are unhappy with an action if it's small

Improving the policy gradient

$$\nabla_\theta J(\theta) \approx \frac{1}{N}\sum_{n=1}^{N}\sum_{t\geq 0}\underbrace{\left(\sum_{t'\geq t}\gamma^{t'-t} r\left(s_{t'}^{(n)}, a_{t'}^{(n)}\right)\right)}_{\text{Reward to go } \hat{Q}_t^{(n)}} \nabla_\theta \log \pi_\theta\left(a_t^{(n)} | s_t^{(n)}\right)$$

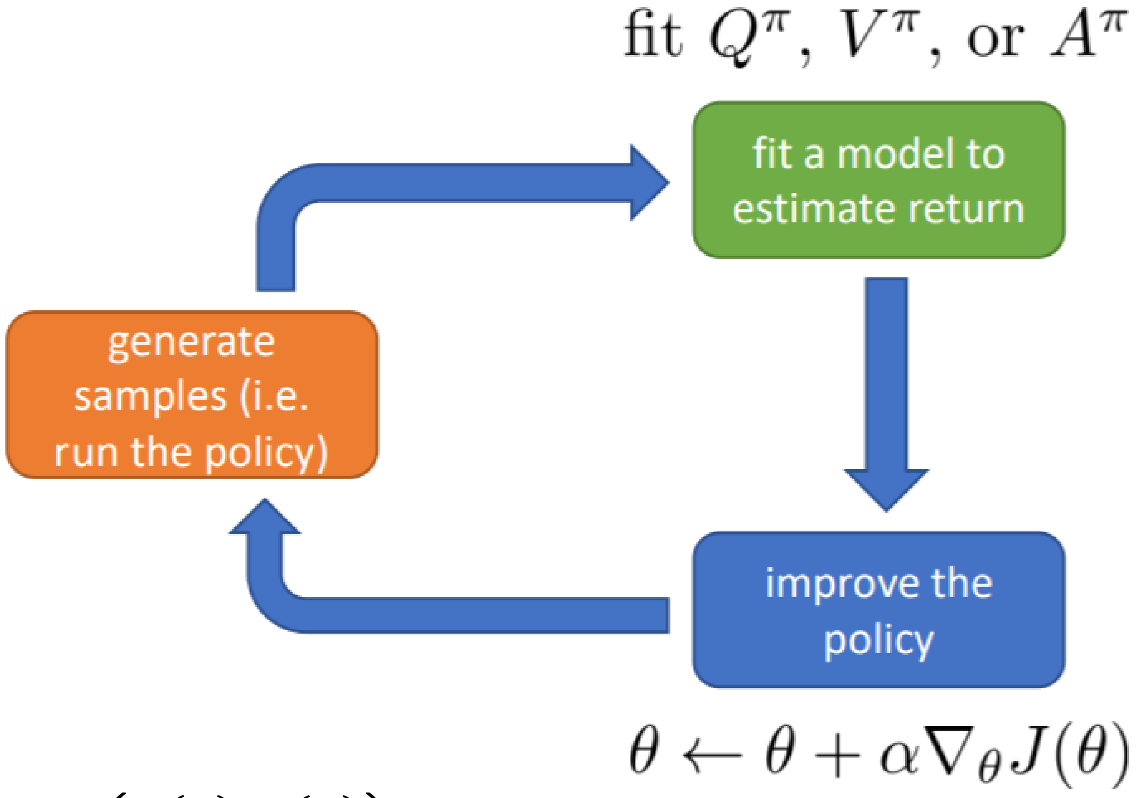


- $\hat{Q}_t^{(n)}$: estimate of expected reward if we take action $a_t^{(n)}$ in state $s_t^{(n)}$
- $Q(s_t, a_t) = \sum_{t'\geq t} E_{p_\theta}\left[\gamma^{t'-t} r(s_{t'}, a_{t'}) | s_t, a_t\right]$
  - True expected reward to go
- $V(s_t) = E_{a_t \sim \pi_\theta(a_t|s_t)} Q(s_t, a_t)$
- $\nabla_\theta J(\theta) \approx \frac{1}{N}\sum_{n=1}^{N}\sum_{t\geq 0}\left(Q\left(s_t^{(n)}, a_t^{(n)}\right) - V\left(s_t^{(n)}\right)\right)\nabla_\theta \log \pi_\theta\left(a_t^{(n)} | s_t^{(n)}\right)$

State & state-action value functions

- $Q^\pi(s_t, a_t) = \sum_{t' \geq t} E_{p_\theta}\left[\gamma^{t'-t} r(s_{t'}, a_{t'}) | s_t, a_t\right]$
  - True expected reward to go
- $V^\pi(s_t) = E_{a_t \sim \pi_\theta(a_t|s_t)} Q(s_t, a_t)$
  - Total reward from s_t
- $A^\pi(s_t, a_t) = Q^\pi(s_t, a_t) - V^\pi(s_t)$
  - How much better a_t is

Remark: we can define by the advantage function how much an action was better than expected



- $\nabla_\theta J(\theta) \approx \frac{1}{N} \sum_{n=1}^{N} \sum_{t \geq 0} A^\pi\left(s_t^{(n)}, a_t^{(n)}\right) \nabla_\theta \log \pi_\theta\left(a_t^{(n)} | s_t^{(n)}\right)$

Improving the policy gradient: summary

$$\nabla_\theta J(\theta) \approx \frac{1}{N}\sum_{n=1}^{N}\left(\sum_{t\geq 0}\gamma^{t'-t} r\left(s_t^{(n)}, a_t^{(n)}\right) - b\right)\sum_{t\geq 0}\nabla_\theta \log \pi_\theta\left(a_t^{(n)} | s_t^{(n)}\right)$$

Instead of using this unbiased, but high variance single-sample estimate, use A^π that is an estimation of expectation

$$\nabla_\theta J(\theta) \approx \frac{1}{N}\sum_{n=1}^{N}\sum_{t\geq 0} A^\pi\left(s_t^{(n)}, a_t^{(n)}\right)\nabla_\theta \log \pi_\theta\left(a_t^{(n)} | s_t^{(n)}\right)$$

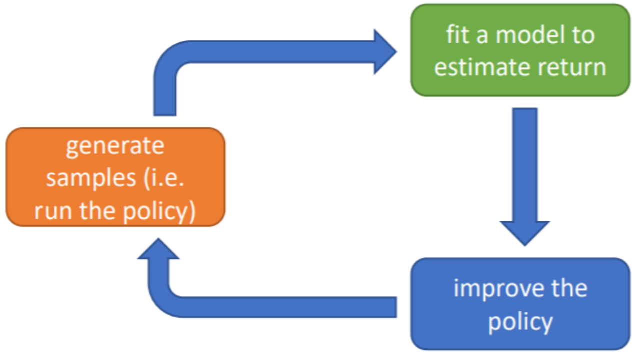
Actor-Critic Algorithm

- **Problem**: we don't know value function
- Can we learn them?
  - Yes, like 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
  - 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

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

Value function fitting

fit Q^π, V^π, or A^π



$\theta \leftarrow \theta + \alpha \nabla_\theta J(\theta)$

fit *what* to *what*?

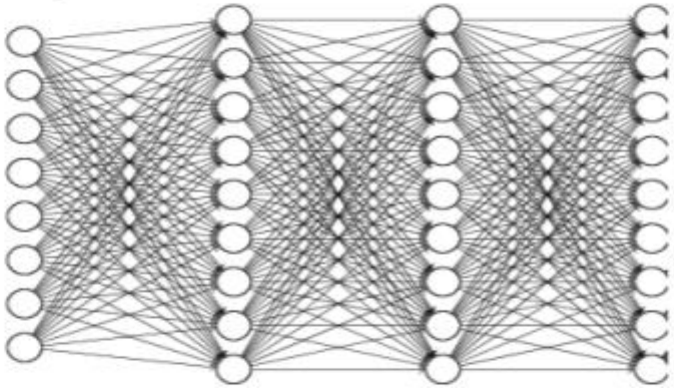
Q^π, V^π, A^π?

$$Q^\pi(\mathbf{s}_t, \mathbf{a}_t) = r(\mathbf{s}_t, \mathbf{a}_t) + \underbrace{\gamma E_{\mathbf{s}_{t+1} \sim p(\mathbf{s}_{t+1}|\mathbf{s}_t, \mathbf{a}_t)}[V^\pi(\mathbf{s}_{t+1})]}_{V^\pi(\mathbf{s}_{t+1})}$$

$$A^\pi(\mathbf{s}_t, \mathbf{a}_t) \approx r(\mathbf{s}_t, \mathbf{a}_t) + \gamma V^\pi(\mathbf{s}_{t+1}) - V^\pi(\mathbf{s}_t)$$

let's just fit $V^\pi(\mathbf{s})$!

$\mathbf{s}$



$\hat{V}^\pi(\mathbf{s})$

parameters ϕ

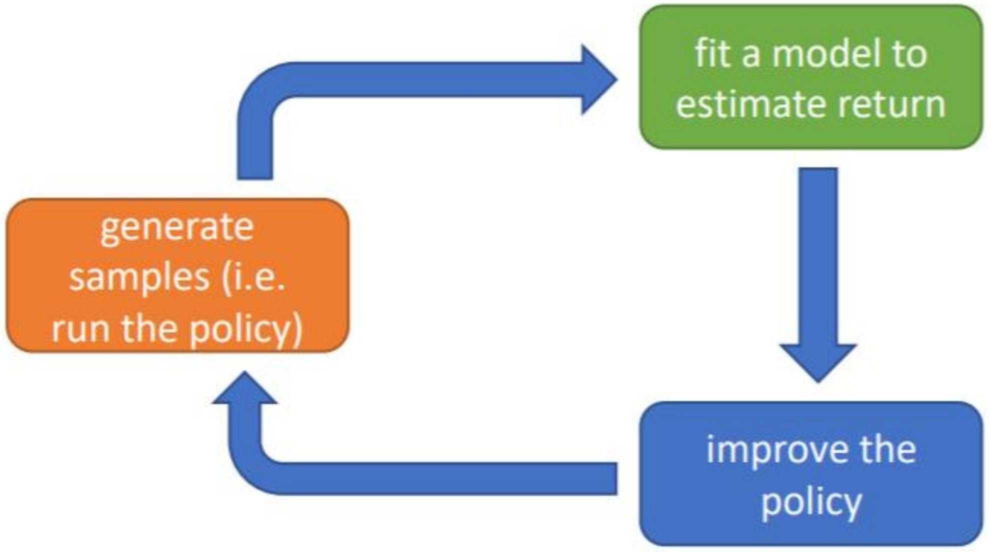
An actor-critic algorithm

batch actor-critic algorithm:

repeat

1. sample $\{\mathbf{s}_i, \mathbf{a}_i\}$ from $\pi_\theta(\mathbf{a}|\mathbf{s})$ (run it on the robot)
2. fit $\hat{V}^\pi_\phi(\mathbf{s})$ to sampled reward sums
3. evaluate $\hat{A}^\pi(\mathbf{s}_i, \mathbf{a}_i) = r(\mathbf{s}_i, \mathbf{a}_i) + \hat{V}^\pi_\phi(\mathbf{s}'_i) - \hat{V}^\pi_\phi(\mathbf{s}_i)$
4. $\nabla_\theta J(\theta) \approx \sum_i \nabla_\theta \log \pi_\theta(\mathbf{a}_i|\mathbf{s}_i)\hat{A}^\pi(\mathbf{s}_i, \mathbf{a}_i)$
5. $\theta \leftarrow \theta + \alpha \nabla_\theta J(\theta)$

fit $\hat{V}^\pi$

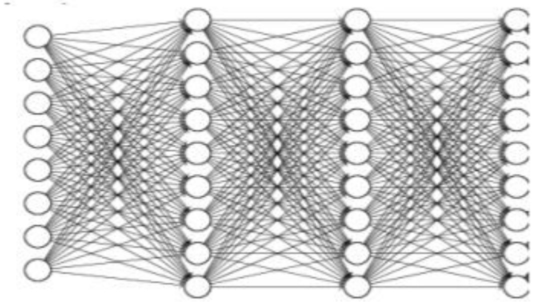


$\theta \leftarrow \theta + \alpha \nabla_\theta J(\theta)$

$$y_t \approx r(s_t, a_t) + \hat{V}^\pi_\phi(s_{t+1})$$

supervised regression: $\mathcal{L}(\phi) = \sum_t \left(\hat{V}^\pi_\phi(s_t) - y_t\right)^2$

$\mathbf{s}$

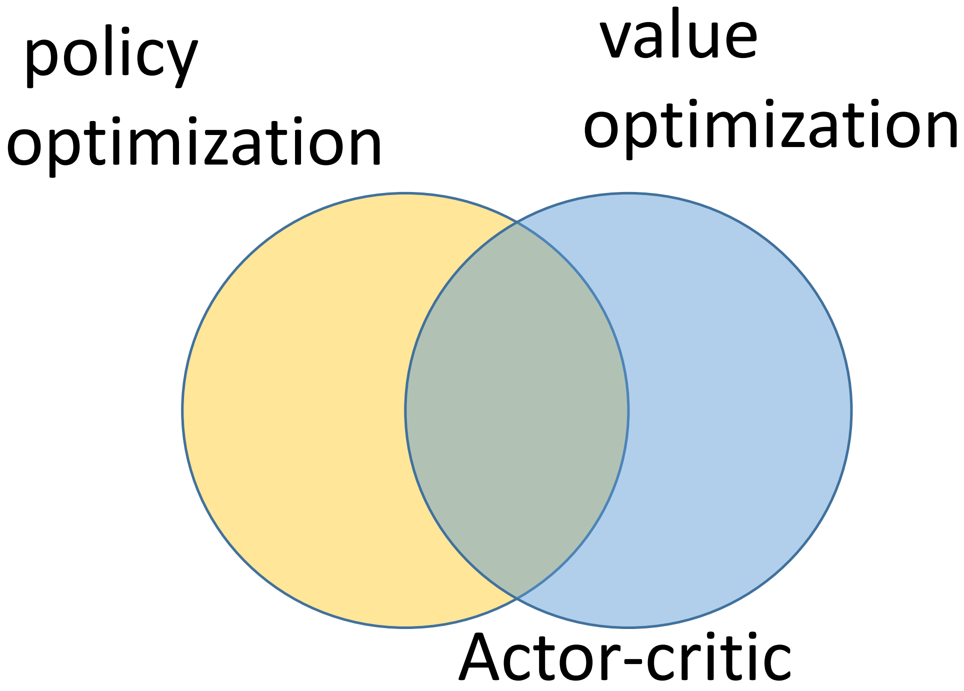


$\hat{V}^\pi(\mathbf{s})$

parameters ϕ

$$V^\pi(\mathbf{s}_t) = \sum_{t'=t}^{T} E_{\pi_\theta}\left[r(\mathbf{s}_{t'}, \mathbf{a}_{t'})|\mathbf{s}_t\right]$$

Actor-critic methods



Advantages of Policy-based RL

- Advantages
  - Better convergence properties
  - Effective in high dimensional or continuous action spaces
  - Can learn stochastic policies

- Disadvantages
  - Typically converges to a local rather than global optimum
  - Evaluating a policy is typically inefficient and high variance

RL in Other ML Problems

- Hard Attention
  - Observation: current image window
  - Action: where to look
  - Reward: classification

V. Mnih et al., “Recurrent models of visual attention”, NIPS 2014.

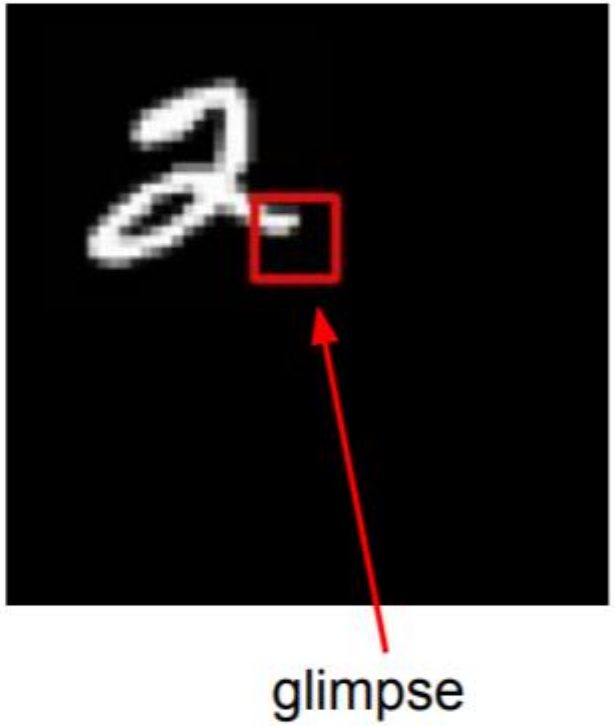
- Sequential/structured prediction, e.g., machine translation
  - Observations: words in source language
  - Actions: emit word in target language
  - Rewards: sentence-level metric, e.g. BLEU score

M. Ranzato et al., "Sequence level training with recurrent neural networks“, 2015.

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



[Mnih et al. 2014]

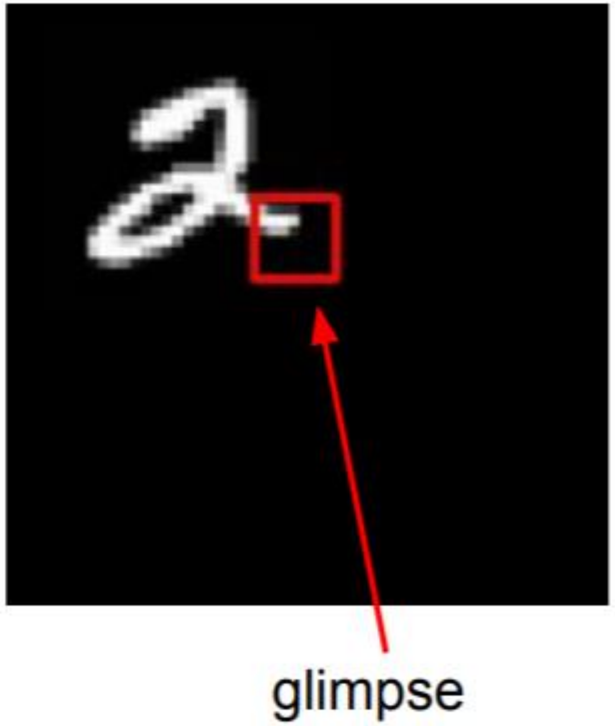
REINFORCE in action: Recurrent Attention Model (RAM)

- **Objective**: Image Classification

- **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

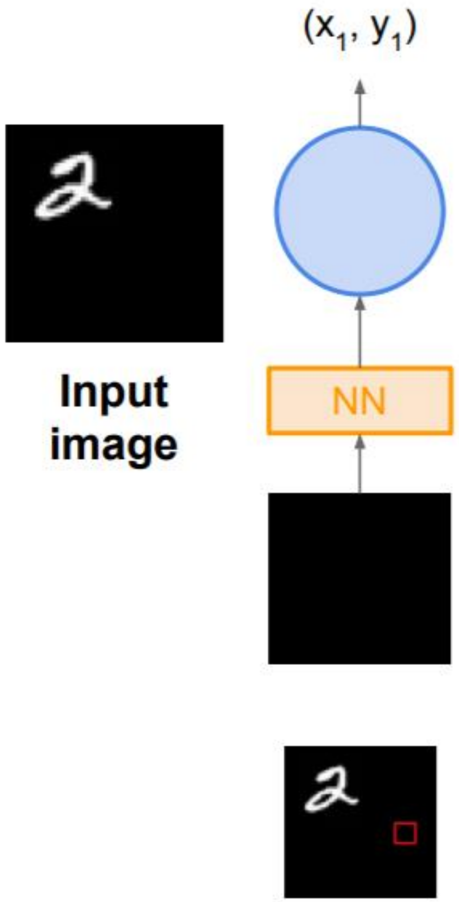
  => learn policy for how to take glimpse actions using REINFORCE



[Mnih et al. 2014]

REINFORCE in action: Recurrent Attention Model (RAM)

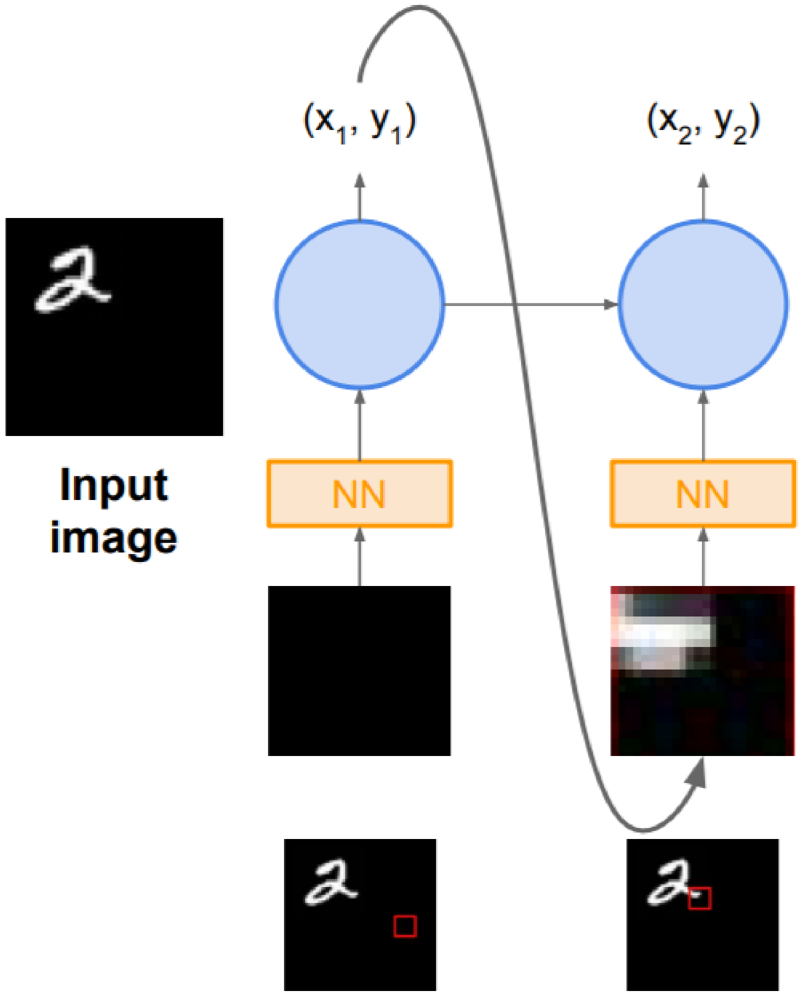
- Given state of glimpses seen so far, use RNN to model the state and output next action



[Mnih et al. 2014]

REINFORCE in action: Recurrent Attention Model (RAM)

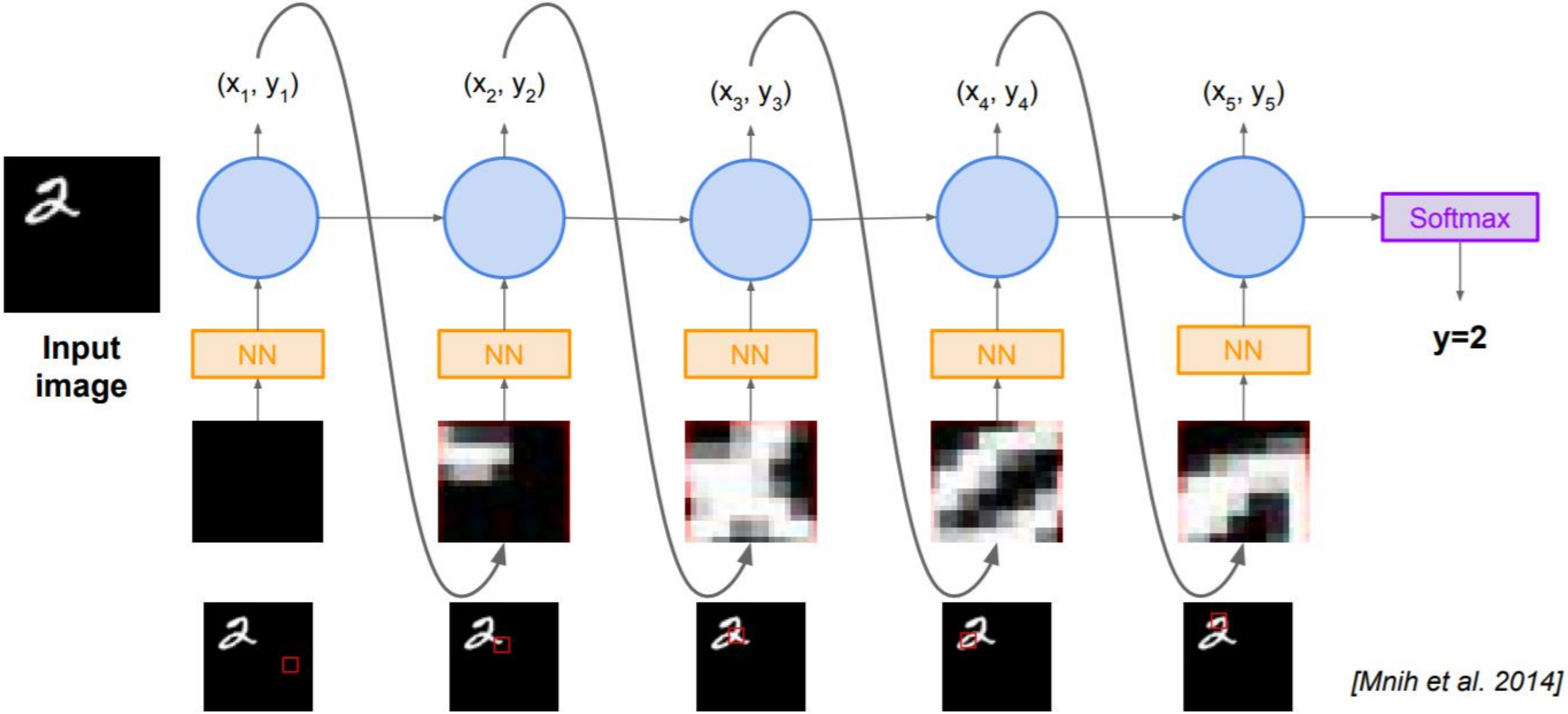
- Given state of glimpses seen so far, use RNN to model the state and output next action



[Mnih et al. 2014]

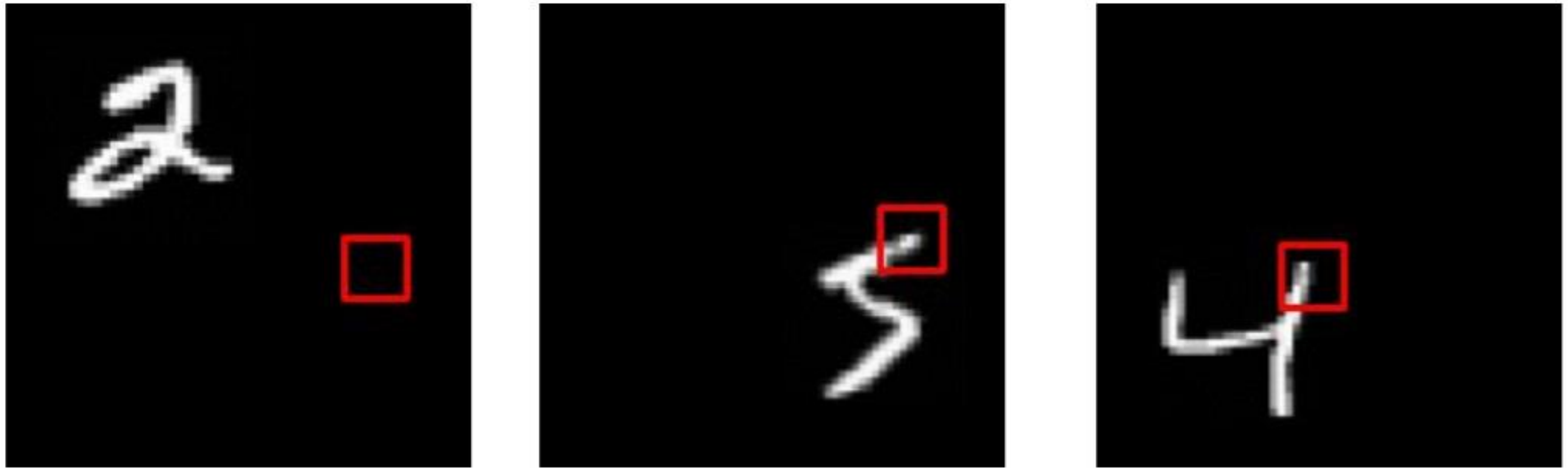
REINFORCE in action: Recurrent Attention Model (RAM)

- Given state of glimpses seen so far, use RNN to model the state and output next action



[Mnih et al. 2014]

REINFORCE in action: Recurrent Attention Model (RAM)



Has also been used in many other tasks including fine-grained image recognition, image captioning, and visual question-answering!

[Mnih et al. 2014]

Sequence generation: disadvantages of previous methods

- Model was trained on a different distribution of inputs from the ones encounters during test (generated by itself)
- Errors made along the way will quickly accumulate (**exposure bias**)
- The **loss function** used to train these models is at the **word level**
- Training these models to directly optimize **metrics like BLEU** (by which they are typically evaluated) is hard
  - because these are not differentiable
  - BLEU: comparing the sequence of actions from the current policy against the optimal action sequence

M. Ranzato et al., "Sequence level training with recurrent neural networks", 2015.

Sequence level evaluation

- A greedy left-to-right process which does not necessarily produce the most likely sequence according to the model

$$\prod_{t=1}^{T} \max_{w_{t+1}} p_\theta(w_{t+1}|w_t^g, \mathbf{h}_{t+1}) \leq \max_{w_1,\ldots,w_T} \prod_{t=1}^{T} p_\theta(w_{t+1}|w_t^g, \mathbf{h}_{t+1})$$

- One of the existing methods to reduce this effect is **Beam Search**
  - It pursues not only one but k next word candidates at each point.

M. Ranzato et al., "Sequence level training with recurrent neural networks", 2015.

Sequence level training

- Starts from the greedy policy and then slowly deviate from it to let the model explore and make use of its own predictions.
  - greedy policy is obtaind by maximum likelihood on training data

M. Ranzato et al., “Sequence level training with recurrent neural networks”, 2015.

Sequence level training: Loss function

$$L_\theta = -\sum_{w_1^g,\ldots,w_T^g} p_\theta(w_1^g,\ldots,w_T^g) r(w_1^g,\ldots,w_T^g) = -\mathbb{E}_{[w_1^g,\ldots w_T^g]\sim p_\theta} r(w_1^g,\ldots,w_T^g),$$

$$\frac{\partial L_\theta}{\partial \theta} = \sum_t \frac{\partial L_\theta}{\partial \mathbf{o}_t} \frac{\partial \mathbf{o}_t}{\partial \theta}$$

$$\frac{\partial L_\theta}{\partial \mathbf{o}_t} = \left(r(w_1^g,\ldots,w_T^g) - \bar{r}_{t+1}\right)\left(p_\theta(w_{t+1}|w_t^g, \mathbf{h}_{t+1}, \mathbf{c}_t) - \mathbf{1}(w_{t+1}^g)\right)$$

encourages a word choice w_{t+1}^g if $r > \bar{r}_{t+1}$, or discourages it if $r < \bar{r}_{t+1}$

- baseline $\bar{r}_t$ is estimated by a linear regressor which takes as input the hidden states h_t of the RNN

M. Ranzato et al., "Sequence level training with recurrent neural networks", 2015.

Sequence level training

Data: a set of sequences with their corresponding context.
Result: RNN optimized for generation.
Initialize RNN at random and set N^{XENT}, $N^{\text{XE+R}}$ and Δ;
for $s = T, 1, -\Delta$ **do**
  if $s == T$ **then**
    train RNN for N^{XENT} epochs using XENT only;
  else
    train RNN for $N^{\text{XE+R}}$ epochs. Use XENT loss in the first s steps, and REINFORCE (sampling from the model) in the remaining $T - s$ steps;
  end
end

XENT $$\frac{\partial L_\theta^{\text{XENT}}}{\partial \mathbf{o}_t} = p_\theta(w_{t+1}|w_t, \mathbf{h}_{t+1}, \mathbf{c}_t) - \mathbf{1}(w_{t+1})$$

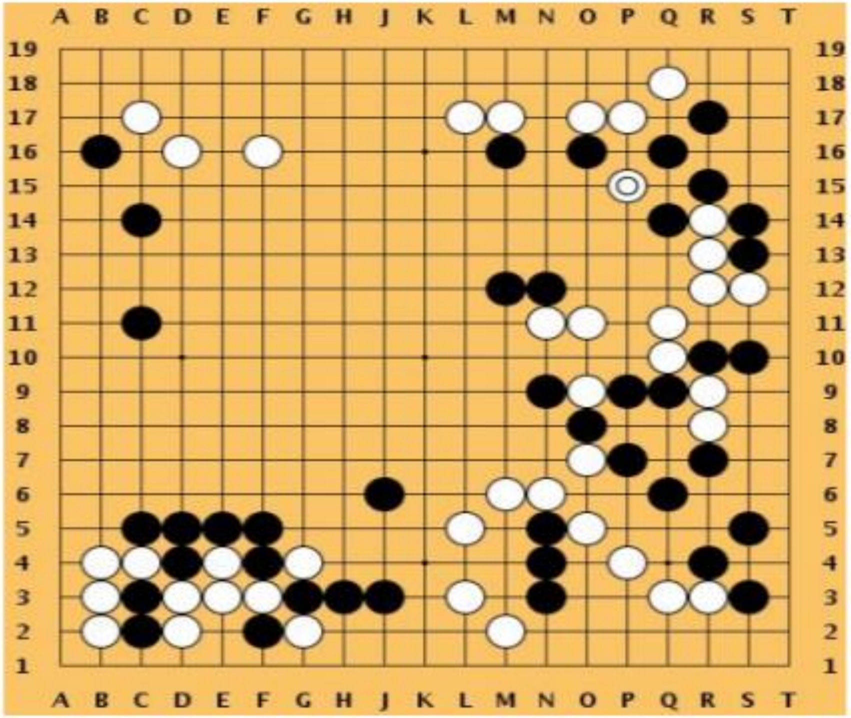
XE+R $$\frac{\partial L_\theta}{\partial \mathbf{o}_t} = \left(r(w_1^g, \dots, w_T^g) - \bar{r}_{t+1}\right)\left(p_\theta(w_{t+1}|w_t^g, \mathbf{h}_{t+1}, \mathbf{c}_t) - \mathbf{1}(w_{t+1}^g)\right)$$

M. Ranzato et al., “Sequence level training with recurrent neural networks”, 2015.

More policy gradients: AlphaGo

Overview:

- Mix of supervised learning and reinforcement learning
- Mix of old methods (Monte Carlo Tree Search) and recent ones (deep RL)



[Silver et al., Nature 2016]

More policy gradients: AlphaGo

- 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 (critic)
  - Finally, combine policy and value networks in a Monte Carlo Tree Search algorithm to select actions by lookahead search

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 $J(\theta)$, often good enough!
  - Q-learning: Zero guarantees since you are approximating Bellman equation with a complicated function approximator