Deep Reinforcement Learning

M. Soleymani
Sharif University of Technology
Spring 2020

Most slides are based on Bhiksha Raj, 11-785, CMU 2019,
some slides from Fei Fei Li and colleagues lectures, cs231n, Stanford 2018,
and some from Surguy Levin lectures, cs294-112, Berkeley 2016.

Q-Learning

- Currently most-popular RL algorithm
- Topics not covered:
  - Value function approximation
  - Continuous state spaces
  - Deep-Q learning

Scaling up the problem..

- We've assumed a discrete set of states
- And a discrete set of actions

- Value functions can be stored as a table
  - One entry per state
- Action value functions can be stored as a table
  - One entry per state-action combination
- Policy can be stored as a table
  - One probability entry per state-action combination

- None of this is feasible if
  - The state space grows too large (e.g. chess)
  - Or the states are continuous valued

Problem

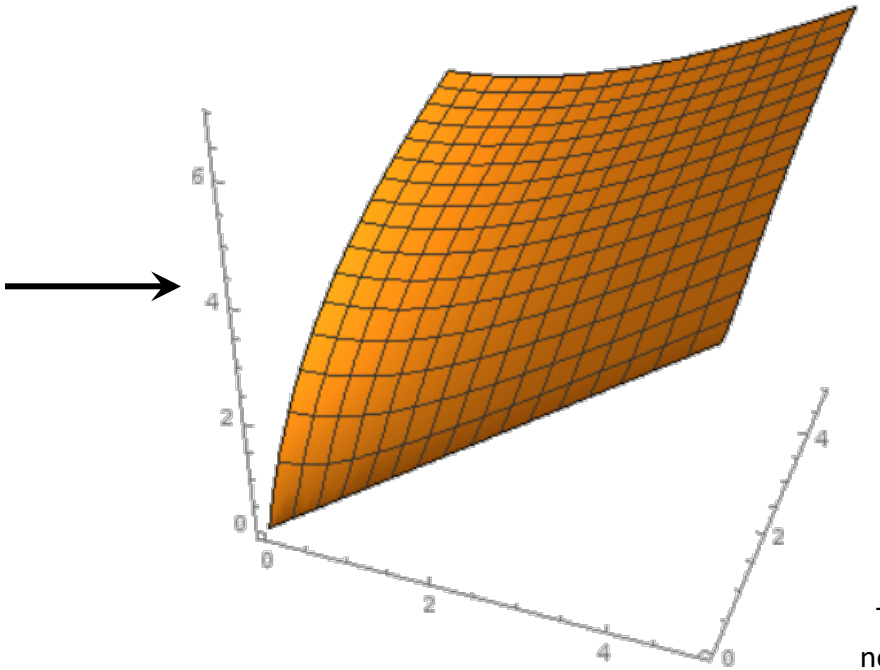
- Not scalable.
  - Must compute Q(s,a) for every state-action pair.
    - it computationally infeasible to compute for entire state space!

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

Continuous State Space

- Tabular methods won't work if our state space is infinite or huge
- E.g. position on a [0, 5] x [0, 5] square, instead of a 5x5 grid.

4.4	4.5	4.8	5.3	5.9
3.9	4.0	4.4	4.9	5.6
3.2	3.4	3.8	4.0	5.1
2.2	2.4	3.0	3.7	4.6
0	1.0	2.0	3.0	4.0



The graphs show the negative value function

Parameterized Functions

- Instead of using a table of Q-values, we use a parametrized function:

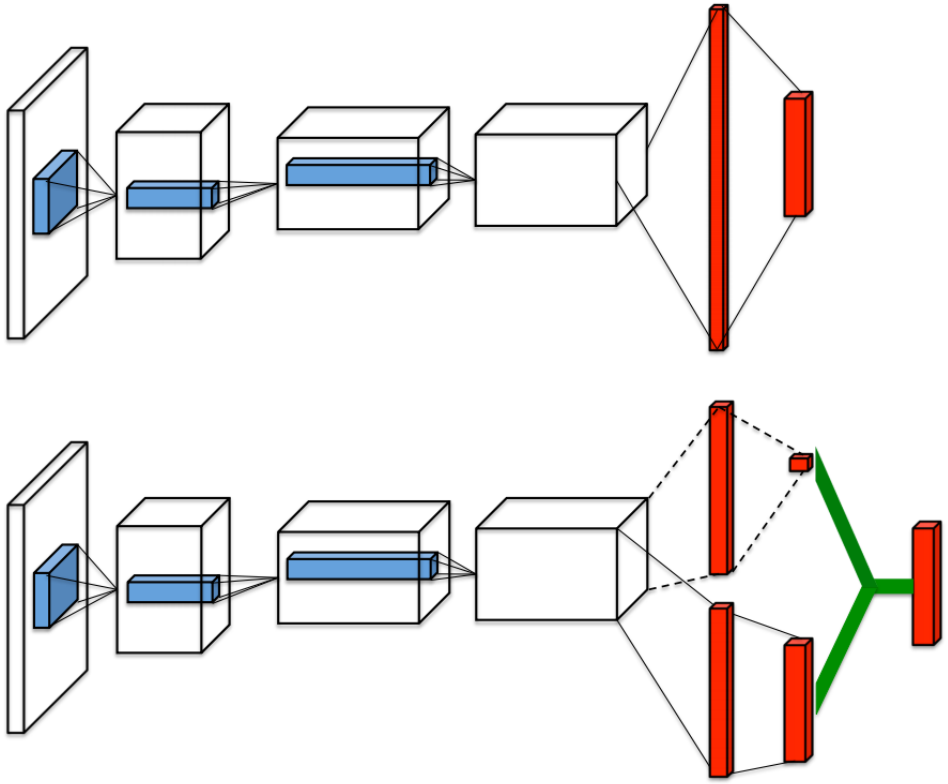
$$Q(s, a|\theta)$$

- If the function approximator is a deep network => Deep RL

- Instead of writing values to the table, we fit the parameters to minimize the prediction error of the “Q function”

$$\theta_{k+1} \leftarrow \theta_k - \eta \nabla_\theta \left(Loss(Q(s, a|\theta_k), Q_{\mathrm{s,a}}^{\mathrm{target}}) \right)$$

Parameterized Functions



Case Study: Playing Atari Games (seen before)

Objective: Complete the game with the highest score

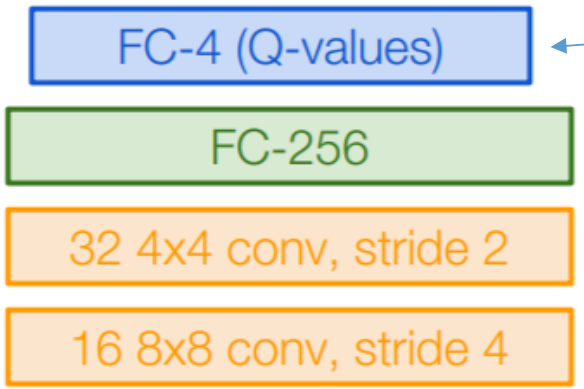
State: Raw pixel inputs of the game state
Action: Game controls e.g. Left, Right, Up, Down
Reward: Score increase/decrease at each time step

[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!



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

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

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

Solving for the optimal policy: Q-learning

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

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

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

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

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

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

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

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

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]

Target Q

$$\theta_{i+1} \leftarrow \theta_i - \eta \nabla_\theta \left(Loss(Q(s, a|\theta_i), Q_{\text{s,a}}^{\text{target}}) \right)$$

→ What is $Q_{\text{s,a}}^{\text{target}}$?

As in TD, use bootstrapping for the target :

$$Q_{\text{s,a}}^{\text{target}} = r + \gamma \underset{a' \in \mathcal{A}}{\text{argmax}}\, Q(s', a'|\theta_i)$$

And $Loss$ can be L2 distance

DQN (v0)

- Initialize θ_1
- For each episode e
  - Initialize s_1
  - For $t = 1 \ldots Termination$
    - Choose action a_t using ε –greedy policy obtained from θ_t
    - Observe r_t, s_{t+1}
    - $Q_{target} = r_t + \gamma max_a Q(s_{t+1}, a|\theta_t)$
    - $\theta_{t+1} = \theta_t - \eta \nabla_\theta \left\| Q_{target} - Q(s_t, a_t|\theta_t) \right\|_2^2$

Deep Q Network

- Note : $\nabla_\theta \left\| Q_{target} - Q(s_t, a_t|\theta_t) \right\|_2^2$ does **not** consider Q_{target} as depending of θ_t (although it does). Therefore this is **semi-gradient descent**.

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
    - e.g. if maximizing action is to move left, training samples will be dominated by samples from left-hand side => 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

Parameterized Functions

- Fundamental issue: limited capacity
  - A table of Q values will never forget any values that you write into it
  - But, modifying the parameters of a Q-function will affect its *overall* behavior
    - Fitting the parameters to match one (s, a) pair can change the function's output at (s', a').
    - If we don't visit (s', a') for a long time, the function's output can diverge considerably from the values previously stored there.

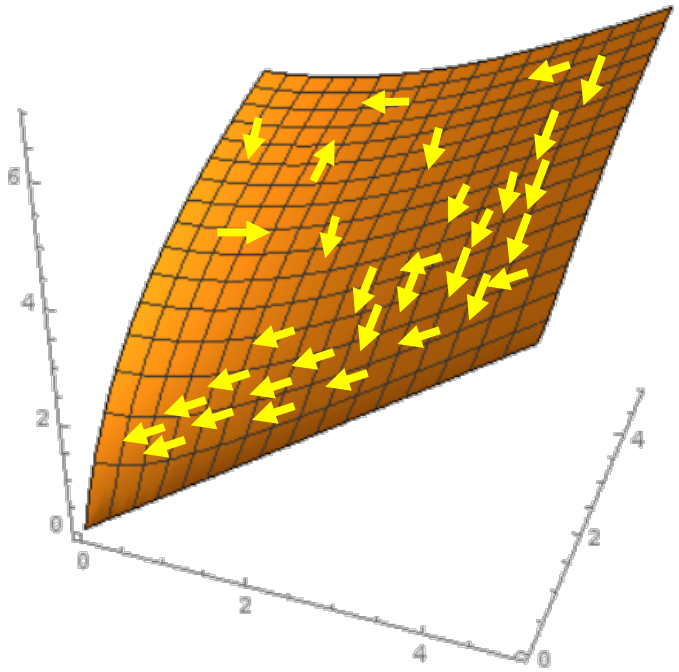
Tables have full capacity

- Q-learning works well with Q-tables
  - The sample data is going to be heavily biased toward optimal actions $\big(s, \pi^*(s)\big)$, or close approximations thereof.
  - But still, ϵ-greedy policy will ensure that we will visit all state-action pairs arbitrarily many times if we explore long enough.
  - The action-value for uncommon inputs will still converge, just more slowly.

Limited Capacity of $Q(s, a|\theta)$

- The Q-function will fit more closely to more common inputs, even at the expense of lower accuracy for less common inputs.

- Just exploring the whole state-action space isn't enough. We also need to visit those states often enough so the function computes accurate Q-values before they are "forgotten".



Experience Replay

- The raw data obtained from Q-learning is:
  - Highly correlated: current data can look very different from data from several episodes ago if the policy changed significantly.
  - Very unevenly distributed: only ϵ probability of choosing suboptimal actions.

- Instead, create a *replay buffer* holding past experiences, so we can train the Q-function using this data.

Experience Replay

- We have control over how the experiences are added, sampled and deleted.
  - Can make the samples look independent
  - Can emphasize old experiences more
  - Can change frequency depending on accuracy

- What is the best way to sample? (A trade off!)
  - On the one hand, our function has limited capacity, so we should let it optimize more strongly for the common case
  - On the other hand, our function needs explore uncommon examples just enough to compute accurate action-values, so it can avoid missing out on better policies

DQN (with Experience Replay)

- Initialize θ_0
- Initialize buffer with some random episodes
- For each episode e
  - Initialize s_1, a_1
  - For $t = 1 \ldots Termination$
    - Choose action a_t using ε –greedy policy obtained from θ_t
    - Observe r_t, s_{t+1}
    - Add s_t, a_t, r_t, s_{t+1} to the buffer
    - Sample from the buffer a batch of tuples s, a, r, s_{new}
    - $Q_{target} = r + \gamma max_a Q(s_{\text{new}}, a|\theta_t)$
    - $\theta_{t+1} = \theta_t - \eta \nabla_\theta \left\| Q_{target} - Q(s, a|\theta_t) \right\|_2^2$

Moving target

- We already have moving targets in Q-learning itself

- The problem is much worse with Q-functions though. Optimizing the function at one state-action pair affects *all other state-action pairs*.
  - The target value is fluctuating at all inputs in the function's domain, and all updates will shift the target value across the entire domain.

Frozen target function

- Solution : Create two copies of the Q-function.
  - The "target copy" is frozen and used to compute the target Q-values.
  - The "learner copy" will be trained on the targets.

$$Q_{\text{learner}}(s_t, a_t) \leftarrow_{\text{fit}} r_t + \gamma \max_{\text{a}} \left(Q_{\text{target}}(s_{t+1}, a) \right)$$

- Just need to periodically update the target copy to match the learner copy.

Fixed target DQN

- Initialize $\theta_0, \theta^* = \theta_0$
- Initialize buffer with some random episodes
- For each episode e
  - Initialize S_1, A_1
    - For $t = 1 \ldots Termination$
    - If $t\%k = 0$ then update $\theta^* = \theta_t$
    - Choose action a_t using ε –greedy policy obtained from θ_t
    - Observe r_t, s_{t+1}
    - Add s_t, a_t, r_t, s_{t+1} to the buffer
    - Sample from the buffer a batch of tuples s, a, r, s_{new}
    - $Q_{target} = r + \gamma \max_a Q(s_{\text{new}}, a|\theta^*)$
    - $\theta_{t+1} = \theta_t - \eta \nabla_\theta \left\| Q_{target} - Q(s, a|\theta_t) \right\|_2^2$

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

Initialize replay memory $\mathcal{D}$ to capacity N
Initialize action-value function Q with random weights

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**
  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

Play M episodes (full games)

[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**
    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

For each time-step of game

[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 $\mathcal{D}$
    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

Store transition in replay memory

[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]

Performance

	Breakout	R. Raid	Enduro	Sequest	S. Invaders
DQN	**316.8**	**7446.6**	**1006.3**	**2894.4**	**1088.9**
Naive DQN	3.2	1453.0	29.1	275.8	302.0
Linear	3.0	2346.9	62.0	656.9	301.3

Replay	○	○	×	×
Target	○	×	○	×
Breakout	**316.8**	240.7	10.2	3.2
River Raid	**7446.6**	4102.8	2867.7	1453.0
Seaquest	**2894.4**	822.6	1003.0	275.8
Space Invaders	**1088.9**	826.3	373.2	302.0

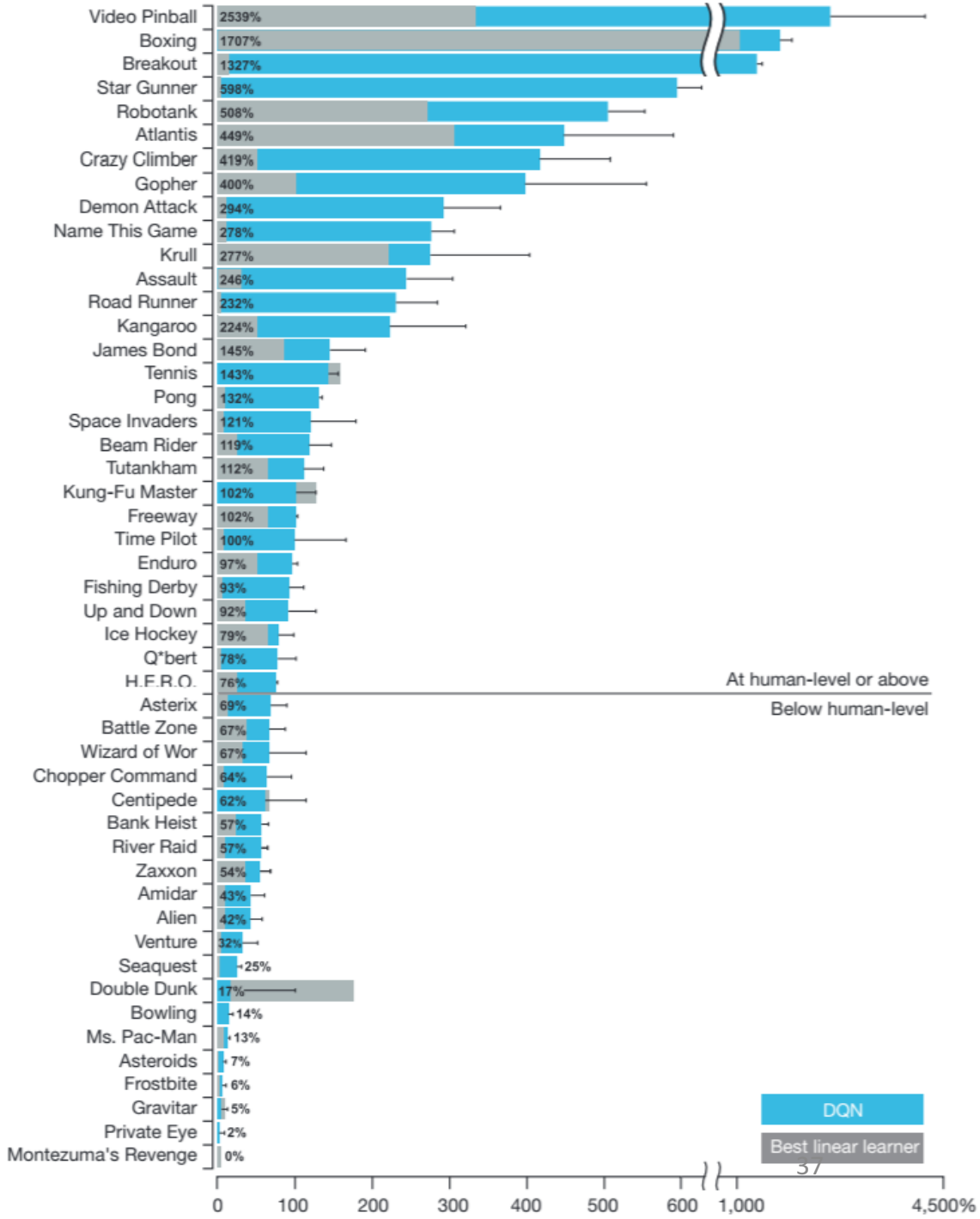
https://www.youtube.com/watch?v=V1eYniJ0Rnk

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?

Direct Policy Estimation

- It's also possible to make a deep neural network that directly produces a distribution over actions given a state
  - Also known as a policy network, or the policy gradient method
  - Useful when **the action space is also large or continuous**

Policy Network

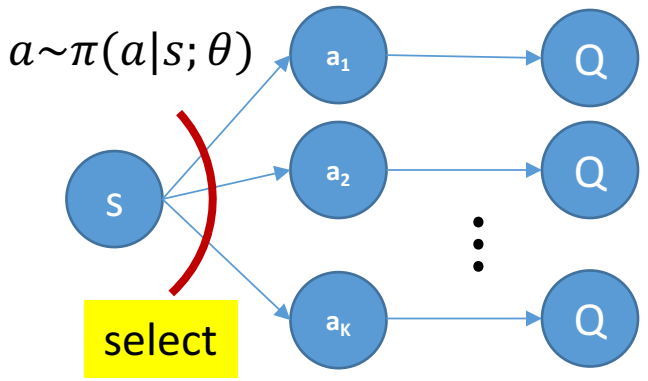
- Train a neural network to prescribe actions at each state:

$$\pi(a|s;\theta)$$

  - Input is s, output is probability distribution over a
  - Could be deterministic
- **Problem** : how to train such a network?
- No golden truth
  - Unlike *value* functions, where there is a *target* value for the value at each state
    - Against which we can compute a loss

Maximizing return

- ***Learn policy to maximize expected return!***



- **Problem:** For discrete action space, the return is not differentiable with respect to policy function parameters
  - Selection is not a differentiable operation

How to choose policy

- In any run starting at a state S we get
  - $(s_1 = s,)\ a_1\ r_1\ s_2\ a_2\ r_2\ s_3\ a_3\ r_3\ \ldots$
- The trajectory T associated with the run is
  - $\tau = s\ a_1\ r_1\ s_2\ a_2\ r_2\ s_3\ a_3\ r_3\ \ldots$
- The total return over the run (at t=1) is
  - $G = r_1 + \gamma r_2 + \gamma^2 r_3 \ldots.$
- The choice of θ in $\pi(a|s;\theta)$ will modify the trajectory and thereby the return

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:

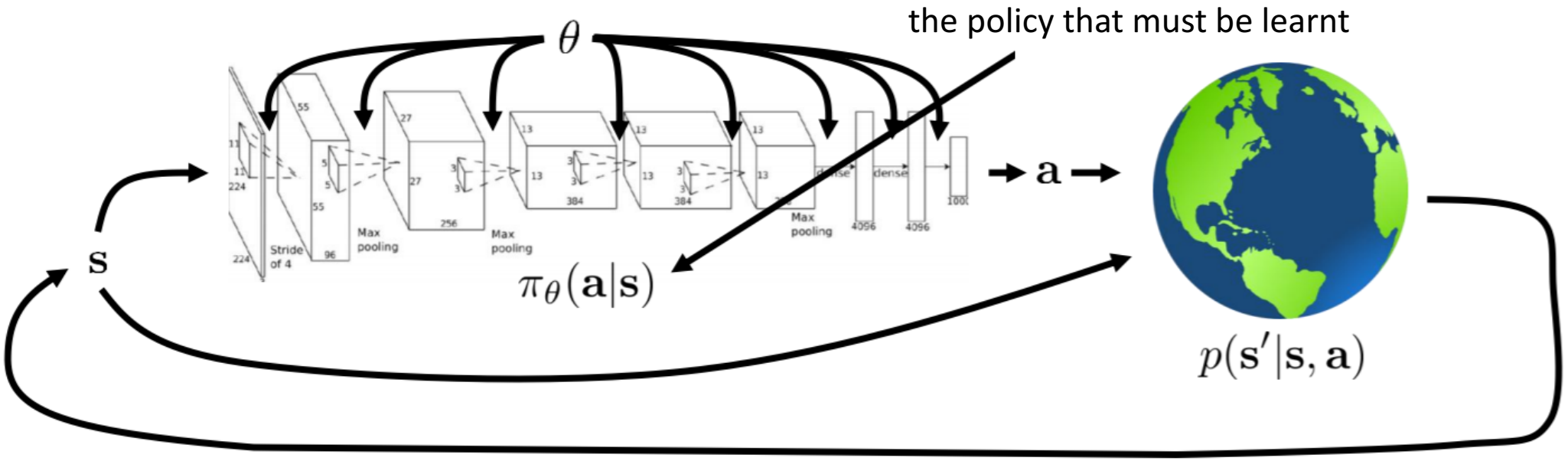
$$J(\theta) = \mathbb{E}\left[\sum_{t \geq 0} \gamma^t r_t | \pi_\theta\right]$$

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

How can we do this?

Gradient ascent on policy parameters!

The goal of RL



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

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

The objective

- The probability of a trajectory τ is a function of $\pi(a|s;\theta)$ and hence of θ
  - $\tau \sim P(\tau;\theta)$
- The probability of a return G is a function of the trajectory τ
  - $G(\tau)$
- Objective: to maximize expected return

$$\underset{\theta}{\operatorname{argmax}} J(\theta) = \underset{\theta}{\operatorname{argmax}} \sum_{\tau} P(\tau;\theta)\, G(\tau)$$

REINFORCE algorithm

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

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

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

Solution

- Recast differentiation as an *expectation* operation
  - Can now be approximated by sampling
  - Policy gradient method

- Compute expected returns using an action-value function approximator
  - Actor-critic methods

Gradient of the objective

$$J(\theta) = E_{\tau \sim P(\tau;\theta)}[G(\tau)] = \sum_{\tau} P(\tau;\theta)\, G(\tau)$$

$$\nabla_\theta J(\theta) = \sum_{\tau} \nabla_\theta P(\tau;\theta)\, G(\tau)$$

- A simple trick:

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

$$\nabla_\theta J(\theta) = \sum_{\tau} P(\tau;\theta)\nabla_\theta \log P(\tau;\theta)\, G(\tau)$$

$$\nabla_\theta J(\theta) = E_{\tau \sim P(\tau;\theta)} \nabla_\theta \log P(\tau;\theta)\, G(\tau)$$

We can estimate it with Monte Carlo sampling

The trajectory

- The trajectory: $\tau = s\ a_1\ r_1\ s_2\ a_2\ r_2\ s_3\ a_3\ r_3\ \ldots$
- The probability of τ, under the policy function $\pi(a|s;\theta)$ is

$$P(\tau;\theta) = \prod_{t\geq 1} P(s_t|s_{t-1})\pi(a_t|s_t;\theta)$$
$$= P(s_1)\pi(a_1|s_1;\theta)P(s_2|s_1,a_1)\pi(a_2|s_2;\theta)\ldots$$

- Taking logs

$$\log P(\tau;\theta) = \sum_{t\geq 1} \log P(s_{t+1}|s_t,a_t) + \sum_{t\geq 1} \log \pi(a_t|s_t;\theta)$$

- Giving us the deriviative

$$\nabla_\theta \log P(\tau;\theta) = \sum_t \nabla_\theta \log \pi(a_t|s_t;\theta)$$

Does not depend on transition probabilities

Gradient of the objective

$$\nabla_\theta J(\theta) = E_{\tau \sim P(\tau;\theta)} \left[\left(\sum_t \nabla_\theta \log \pi(a_t | s_t; \theta) \right) G(\tau) \right]$$

- This is a simple expectation that can be approximated by sampling!

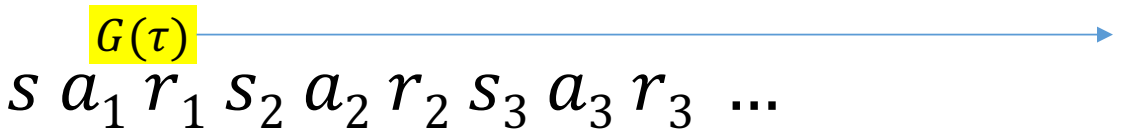
Policy Gradients

- Record an episode (or episodes)

$$s\ a_1\ r_1\ s_2\ a_2\ r_2\ s_3\ a_3\ r_3\ \ldots$$

Policy Gradients

- Episode



- Compute returns at each time

Policy Gradients

- Episode

$G(\tau)$ →

$$s\ a_1\ r_1\ s_2\ a_2\ r_2\ s_3\ a_3\ r_3\ \ldots$$

$\log \pi(a_1|s_1;\theta)$ $\log \pi(a_2|s_2;\theta)$ $\log \pi(a_3|s_3;\theta)$

- Compute returns at each time
- Compute log policy at each time

Policy Gradients

- Episode

$G(\tau)$ →

$$s\ a_1\ r_1\ s_2\ a_2\ r_2\ s_3\ a_3\ r_3\ \ldots$$

$\log \pi(a_1|s_1;\theta)$ $\log \pi(a_2|s_2;\theta)$ $\log \pi(a_3|s_3;\theta)$

$$\nabla_\theta J(\theta) \approx \sum_t \nabla_\theta \log \pi(a_t|s_t;\theta) G(\tau)$$

- Compute returns at each time
- Compute log policy at each time
- Compute gradient

Policy Gradients

- Episode

$G(\tau)$

$$s\ a_1\ r_1\ s_2\ a_2\ r_2\ s_3\ a_3\ r_3\ \ldots$$

$\log \pi(a_1|s_1;\theta)$ $\log \pi(a_2|s_2;\theta)$ $\log \pi(a_3|s_3;\theta)$

$$\nabla_\theta J(\theta) \approx \sum_t \nabla_\theta \log \pi(a_t|s_t;\theta) G(\tau)$$

$$\theta = \theta + \eta \nabla_\theta J(\theta)$$

- Compute returns at each time
- Compute log policy at each time
- Compute gradient
- Update network parameters
  - Ideally $\nabla_\theta J(\theta)$ is averaged over many episodes

Its like Maximum Likelihood

- The gradient actually looks like the derivative of a log likelihood function

$$\nabla_\theta J(\theta) = E_{\tau \sim P(\tau;\theta)}[\nabla_\theta \log P(\tau;\theta)\, G(\tau)]$$

- Can be written as

$$\nabla_\theta J(\theta) = E_{\tau \sim P(\tau;\theta)}\left[\nabla_\theta \log P(\tau;\theta)^{G(\tau)}\right]$$

- Maximization increases the probability of trajectories with greater return
  - If you see a trajectory you increase its probability

Its like Maximum Likelihood

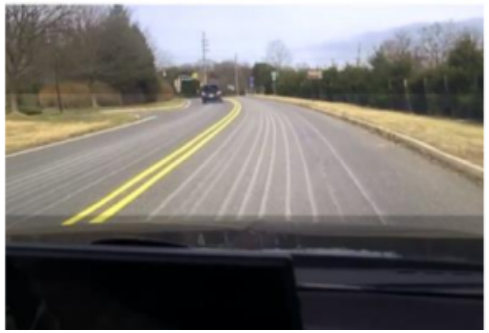
- The gradient actually looks like the derivative of a log likelihood function

$$\nabla_\theta J(\theta) \approx \sum_t \nabla_\theta \log \pi(a_t|s_t;\theta)^{G(\tau)}$$

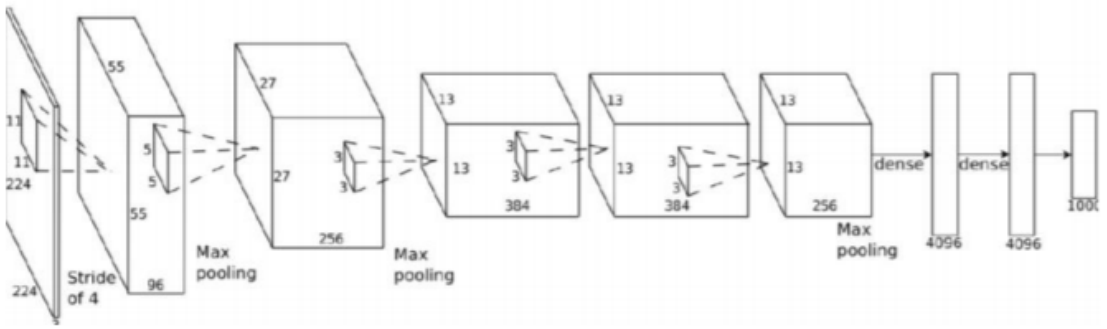
- Maximization increases the probability of all seen actions
  - At the cost of the probability of unseen actions
  - Usual ML estimator

Evaluating the policy gradient

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



s_t



$\pi_\theta(a_t|s_t)$

a_t

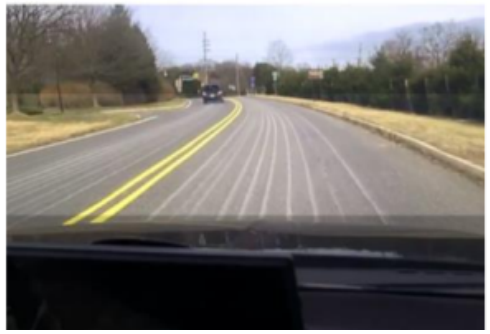
s_t

- Policy gradient:

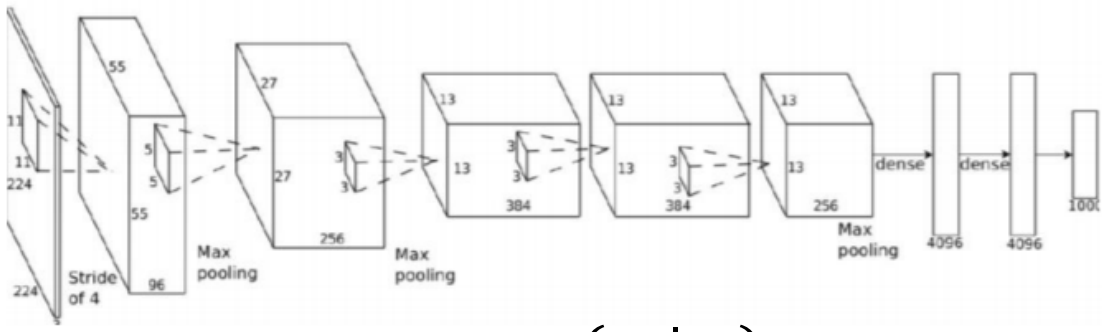
$$\nabla_\theta J(\theta) \approx \frac{1}{N}\sum_{n=1}^{N} G\left(\tau^{(n)}\right) \sum_{t\geq 0} \nabla_\theta \log \pi_\theta \left(a_t^{(n)} | s_t^{(n)}\right)$$

- Maximum Likelihood:

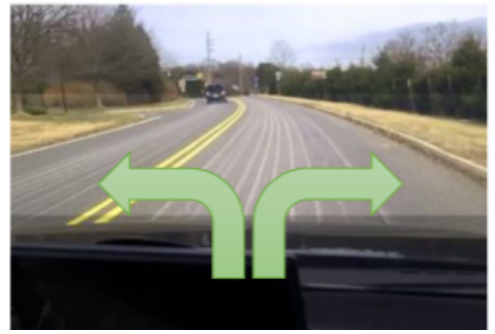
$$\nabla_\theta J(\theta) \approx \frac{1}{N}\sum_{n=1}^{N} \sum_{t\geq 0} \nabla_\theta \log \pi_\theta \left(a_t^{(n)} | s_t^{(n)}\right)$$



s_t



$\pi_\theta(a_t|s_t)$



a_t

$\mathbf{s}_t$

$\mathbf{a}_t$

$\pi_\theta(a_t|s_t)$

What did we just do?

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

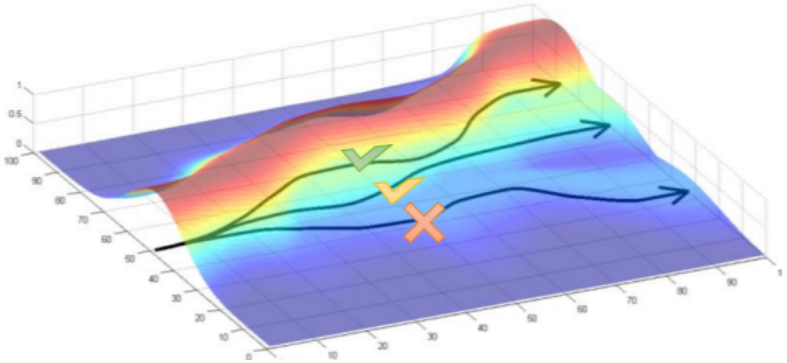
$$\sum_{t\geq 0} \nabla_\theta \log \pi_\theta\left(a_t^{(n)} | s_t^{(n)}\right)$$

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

good stuff is made more likely

bad stuff is made less likely

simply formalizes the notion of “trial and error”!



Intuition

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

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.

A simple extension

$$\nabla_\theta J(\theta) = E_{\tau \sim P(\tau;\theta)} \left[\left(\sum_t \nabla_\theta \log \pi(a_t | s_t; \theta) \right) G(\tau) \right]$$

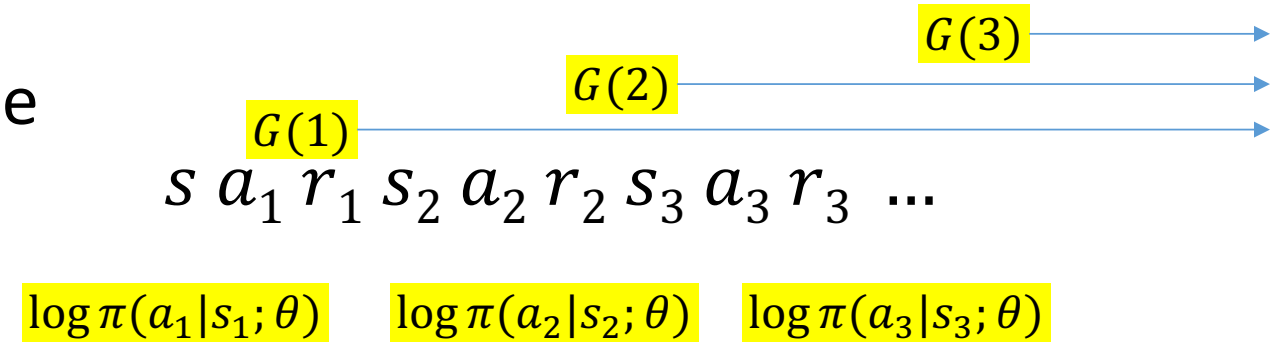
- Better to compute the above instead as follows

$$\nabla_\theta J(\theta) = E_{\tau \sim P(\tau;\theta)} \left[\sum_t \nabla_\theta \log \pi(a_t | s_t; \theta)\, G(t) \right]$$

- This too can be estimated by sampling

Policy Gradients: A simple extension

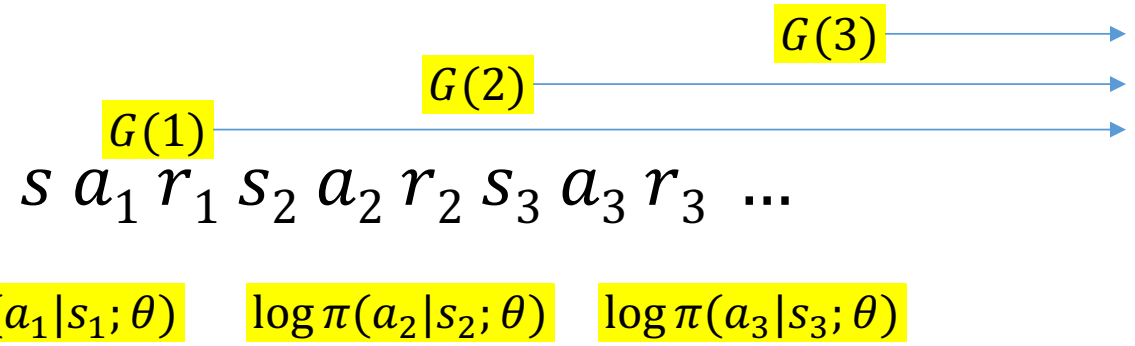
- Episode



- Compute returns at each time
- Compute log policy at each time

Policy Gradients: A simple extension

- Episode

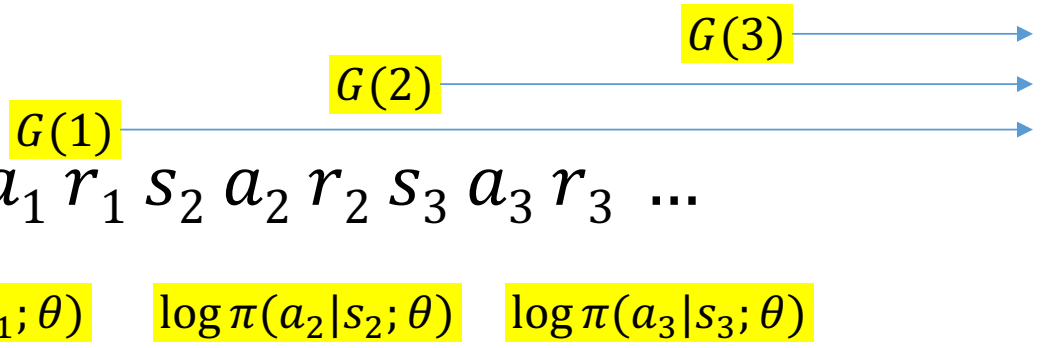


$$\nabla_\theta J(\theta) \approx \frac{1}{\tau}\sum_t \nabla_\theta \log \pi(a_t|s_t;\theta) G(t)$$

- Compute returns at each time
- Compute log policy at each time
- Compute gradient

Policy Gradients: A simple extension

- Episode



$$\nabla_\theta J(\theta) \approx \frac{1}{\tau}\sum_t \nabla_\theta \log \pi(a_t|s_t;\theta) G(t)$$

$$\theta = \theta + \eta \nabla_\theta J(\theta)$$

- Compute returns at each time
- Compute log policy at each time
- Compute gradient
- Update network parameters
  - Ideally $\nabla_\theta J(\theta)$ is averaged over many episodes

Reducing variance

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

$G\left(\tau^{(n)}\right)$

- Causality:

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

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

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

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

Merely seeing a trajectory isn't good

- We want to emphasize trajectories with high return and *reduce* the probability of low-return trajectories

- If an action results in more returns than the current average return for the state, we must improve its probability
  - If it results in less, we must decrease it

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.

$$\nabla_\theta J(\theta) \approx \frac{1}{N}\sum_{n=1}^{N}\sum_{t\geq 0}\left(\overbrace{\sum_{k=0}^{T-t}\gamma^k r\left(s_{t+k}^{(n)}, a_{t+k}^{(n)}\right)}^{G\left(s_t^{(n)}\right)} - V\left(s_t^{(n)}\right)\right)\nabla_\theta \log \pi_\theta\left(a_t^{(n)}|s_t^{(n)}\right)$$

Simple baseline: $V\left(s_t^{(n)}\right) = \bar{G}(s_t)$

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

Its like Maximum Likelihood

- Subtract the *expected* return for the current state

$$\nabla_\theta J(\theta) \approx \sum_t \nabla_\theta \log \pi(a_t|s_t;\theta)^{G(t)-V(S_t)}$$

- $A(t) = G(t) - V(S_t)$ is the *advantage function*
  - How much advantage the current action has over the average
- Train $\pi(a_t|s_t;\theta)$ to maximize advantage

Reinforce

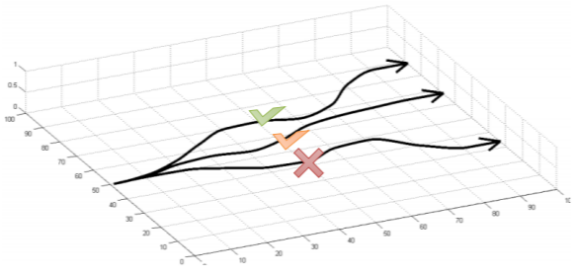
- Initialize θ
- For each episode e
  - Initialize s_1
  - For $t = 1 \ldots Termination$
    - Choose action a_t using policy obtained from θ
    - Observe r_t, s_{t+1}
    - Compute the returns $G(S_t)$, then the advantages A_t
    - Compute $J(\theta) = \frac{1}{T}\sum_t \log\big(\pi_\theta(a_t|s_t)\big) A_t$
    - $\theta \leftarrow \theta + \eta \nabla_\theta J(\theta)$

REINFORCE algorithm: Summary

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

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

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



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

Instability

- In Reinforce, the estimator for the expected return has high variance: rewards on one episode act as estimates for state action value functions.

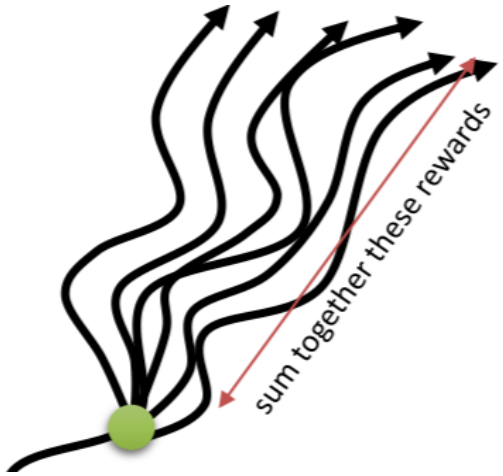
$$G(S_t) = \sum_{t' \geq t} \gamma^{t'-t} r_{t'}$$

- It also requires entire runs of episodes
  - Not online
- It can be made more stable through function approximation of the value function

Actor-Critic

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

Instead of this we can use the temporal difference to estimate this average and also the baseline



Actor-Critic

- In actor-critic methods, two networks are used :
- The **actor** is the policy network : $\pi(s, a|\theta) = \pi_\theta(a|s)$ and is used to predict the next action
- The **critic** is a state value network : $g(s|\phi) = V_\phi(s)$ and is used to guide the optimization direction of the actor

- To estimate the expected return based on an episode, we use a one-step lookahead:

$$G(S_t) = r_t + \gamma V_\phi(s_{t+1})$$

- Or by M-step lookahead :

$$G(S_t) = \sum_{0 \le k \le M-1} \gamma^k r_{t+k} + \gamma^M V_\phi(s_{t+N})$$

Advantage Actor Critic (A2C)

Rethink the advantages

The critic can also be used as the "baseline" when computing the advantages:

$$A_t = G(S_t) - V_\phi(S_t)$$

The trajectory's probability is improved if it is better than the trajectories previously followed.

The critic is trained on how well it predicted the return.

Another view

- $E[G_t] = Q(s_t, a_t)$
- 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.

$$Q(s_t, a_t) - V(s_t)$$

- 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

Another view

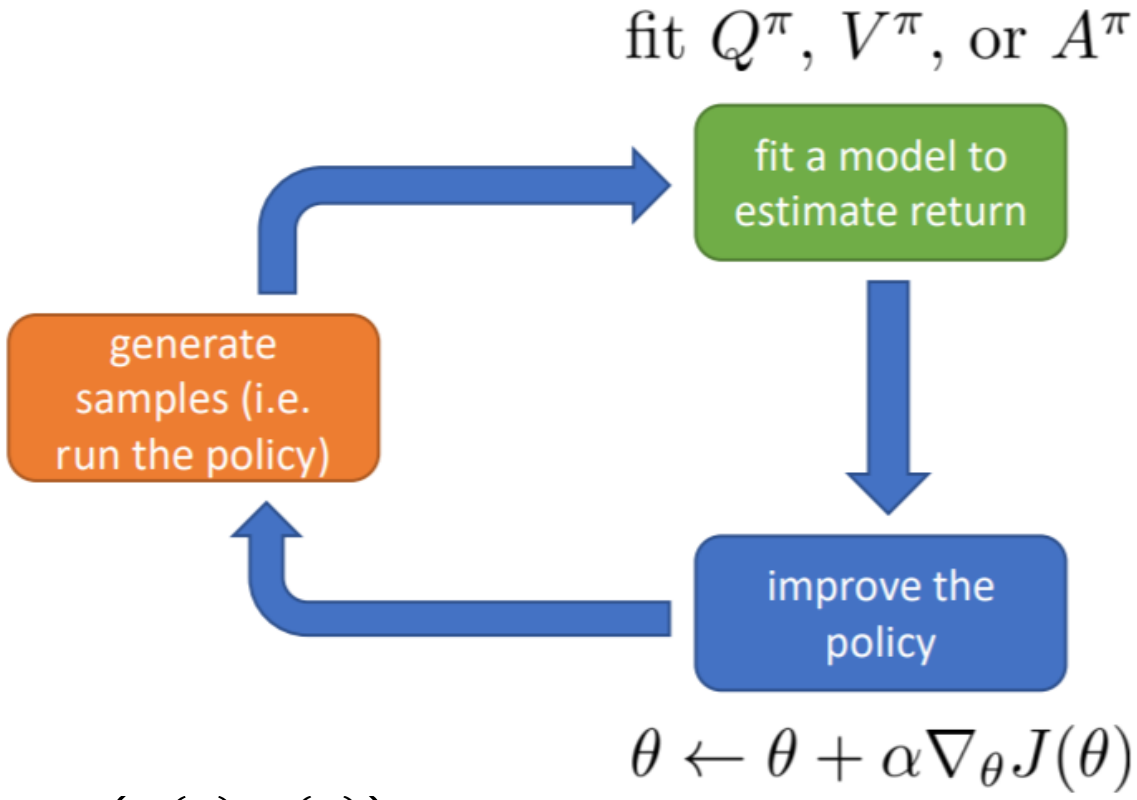
$$\nabla_\theta J(\theta) \approx \frac{1}{N}\sum_{n=1}^{N}\sum_{t\geq 0}\left(\underbrace{\sum_{k=0}\gamma^k r\left(s_{t+k}^{(n)}, a_{t+k}^{(n)}\right)}_{\text{Reward to go } \hat{Q}_t^{(n)}}\right)\nabla_\theta \log \pi_\theta\left(a_t^{(n)}|s_t^{(n)}\right)$$

- $\hat{Q}_t^{(n)}$: estimate of expected reward if we take action $a_t^{(n)}$ in state $s_t^{(n)}$
- $Q(s_t, a_t) = \sum_{k=0}^{T-t} E_{p_\theta}[\gamma^k r(s_{t+k}, a_{t+k})|s_t, a_t]$
  - True expected reward to go
- $V(s_t) = E_{a_t \sim \pi_\theta(a_t|s_t)} Q(s_t, a_t)$
- $\nabla_\theta J(\theta) \approx \frac{1}{N}\sum_{n=1}^{N}\sum_{t\geq 0}\left(Q\left(s_t^{(n)}, a_t^{(n)}\right) - V\left(s_t^{(n)}\right)\right)\nabla_\theta \log \pi_\theta\left(a_t^{(n)}|s_t^{(n)}\right)$

Another view

- $Q^{\pi}(s_t, a_t) = \sum_{k=0}^{T-t} E_{p_\theta}[\gamma^k r(s_{t+k}, a_{t+k}) | s_t, a_t]$
  - 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



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

Value function fitting

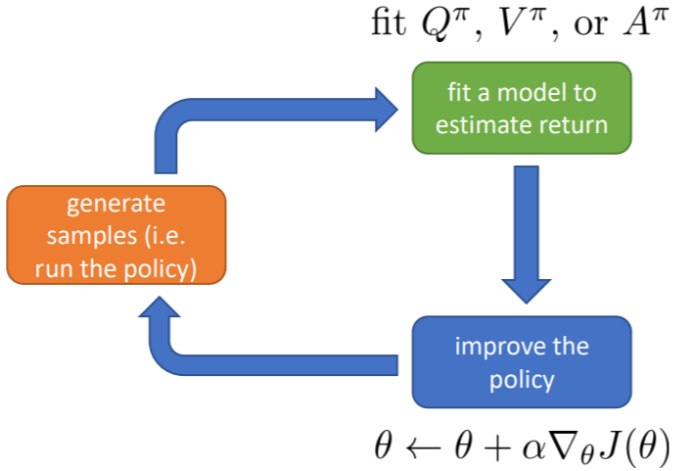
fit *what* to *what*?

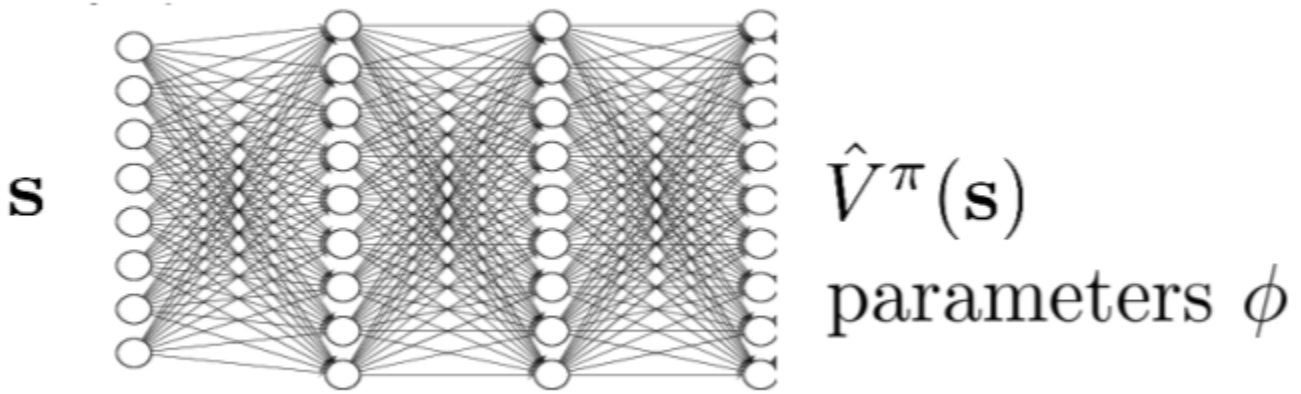
Q^π, V^π, A^π?

$$Q^\pi(s_t, a_t) = E[r_{t+1} + \gamma V^\pi(s_{t+1})]$$
$$\approx r(s_t, a_t) + \gamma V^\pi(s_{t+1})$$

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

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



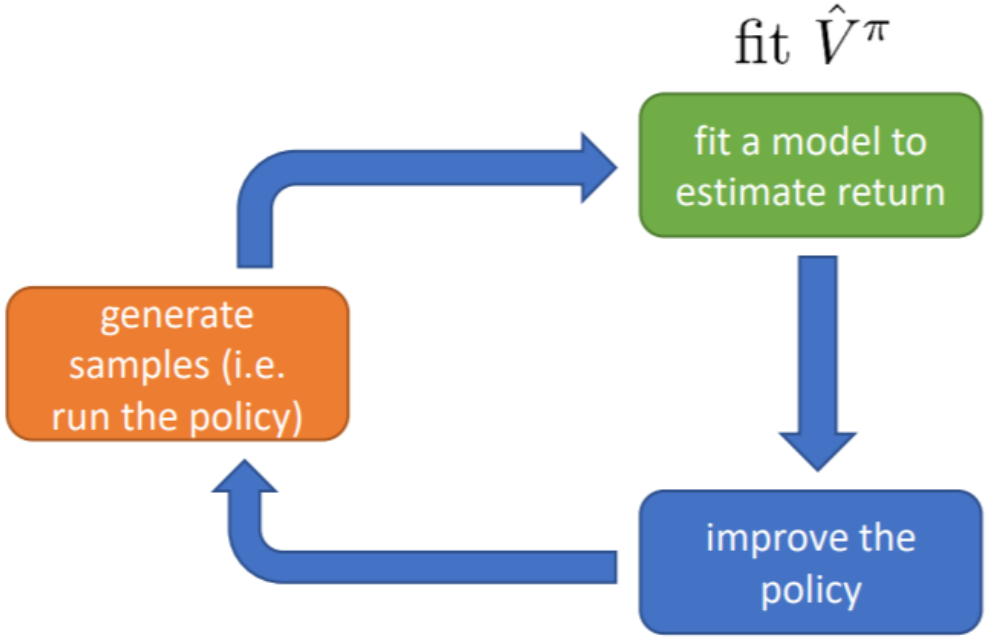


An actor-critic algorithm

batch actor-critic algorithm:

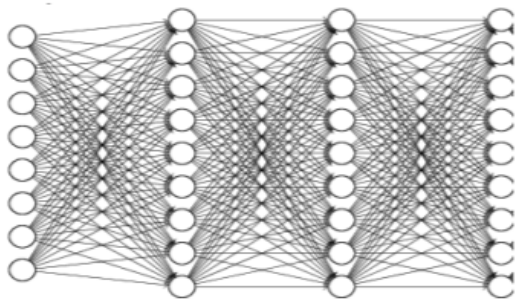
repeat

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 $\hat{A}^\pi(\mathbf{s}_i, \mathbf{a}_i) = r(\mathbf{s}_i, \mathbf{a}_i) + \gamma \hat{V}^\pi_\phi(\mathbf{s}'_i) - \hat{V}^\pi_\phi(\mathbf{s}_i)$
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. $\theta \leftarrow \theta + \alpha \nabla_\theta J(\theta)$



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

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



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

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

A2C

- Initialize θ_a, θ_c
- For each episode e
  - Initialize s_1
  - For $t = 1 \ldots Termination$
    - Choose action a_t using policy obtained from θ
    - Observe r_t, s_{t+1}
  - Compute the returns $G(s_t) = \sum_{0 \le k \le M-1} \gamma^k r_{t+k} + \gamma^M V_\theta(s_{t+M})$ if t + $M < T$, else $\sum_{0 \le k \le T-t-1} \gamma^k r_{t+k}$
  - Compute the advantages $A_t = G(S_t) - V_\theta(S_t)$
  - Compute $L_a(\theta) = \frac{1}{T}\sum_t \log\left(\pi_\theta(a_t|s_t)\right) A_t$, $L_c(\phi) = \frac{1}{T}\sum_t \left(G(s_t) - V_\phi(s_t)\right)^2$
  - $\theta \leftarrow \theta + \eta_a \nabla_\theta L_a(\theta)$, $\phi \leftarrow \phi - \eta_c \nabla_\phi L_c(\phi)$,

M-step look ahead

Extensions

- A2C can be applied in a multi-thread environment on several episodes simultaneously, with a final mini-batch update
- **Asynchronous Advantage Actor-Critic** (A3C) (Deepmind, 2016): Each thread performs its updates without waiting for the others to end → **each thread keeps its own version of the parameters**. They upload their gradients asynchronously to a master server that performs batch updates
- Experience Replay can be adapted to A2C → ACER algorithm (Deepmind 2017)

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

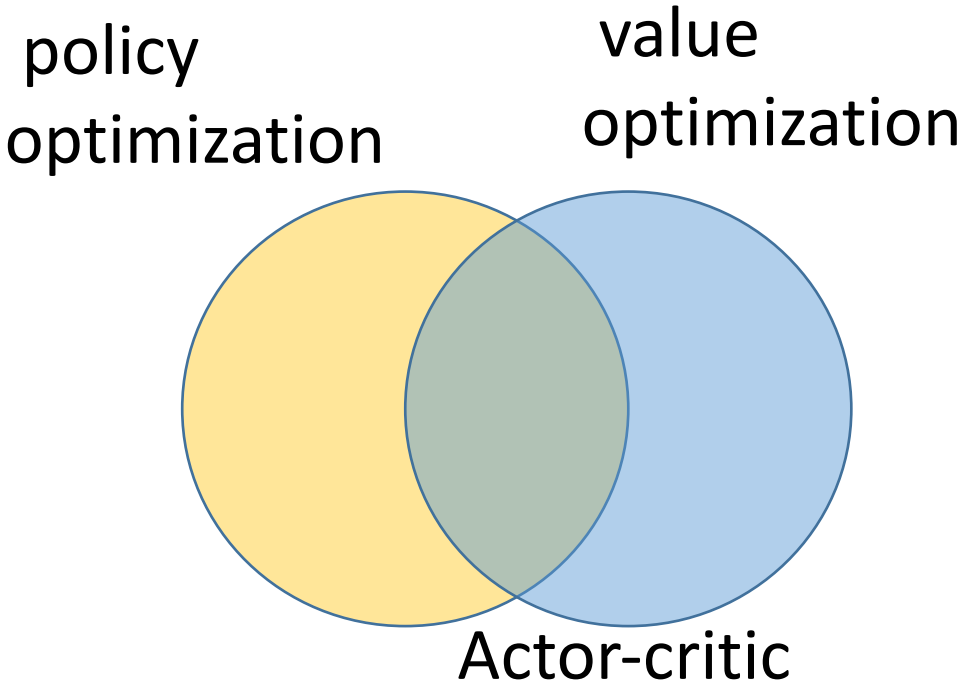
Continuous action space

- Action probabilities $\pi_\theta(a_t|s_t)$: We have seen the discrete action space case (n labels + softmax) → Very large or **continuous space** ?

- You can use a network that **predict the parameters of a distribution** and sample an action from it. Ex : $a_t \sim \mathrm{N}(\mu, \sigma)$ with $\mu, \sigma = \pi(s_t|\theta)$ (similar to the encoder of a VAE) → Reinforce/A2C can be used (with the reparametrization trick).

- Most general case : $f(s_t|\theta) = a_t$. What algorithm can I use ?

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

Example: RL in Other ML Problems

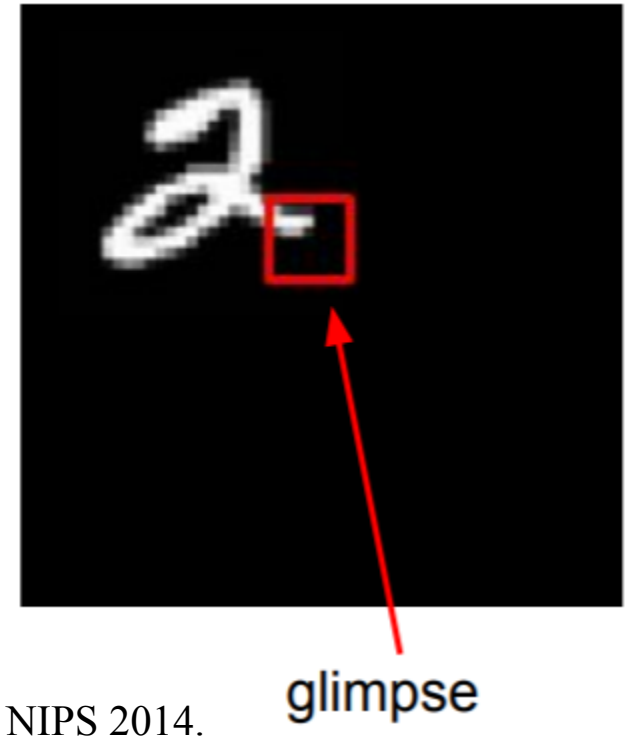
- Hard Attention
  - Observation: current image window
  - Action: where to look
  - Reward: classification

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

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



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

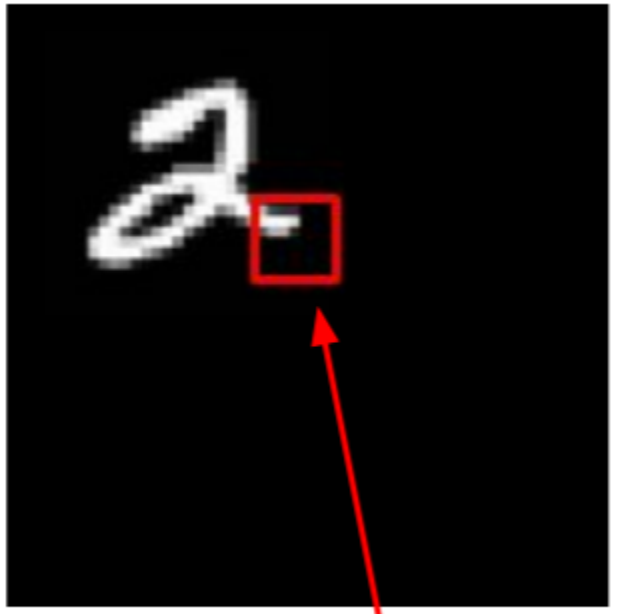
REINFORCE in action: Recurrent Attention Model (RAM)

- **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

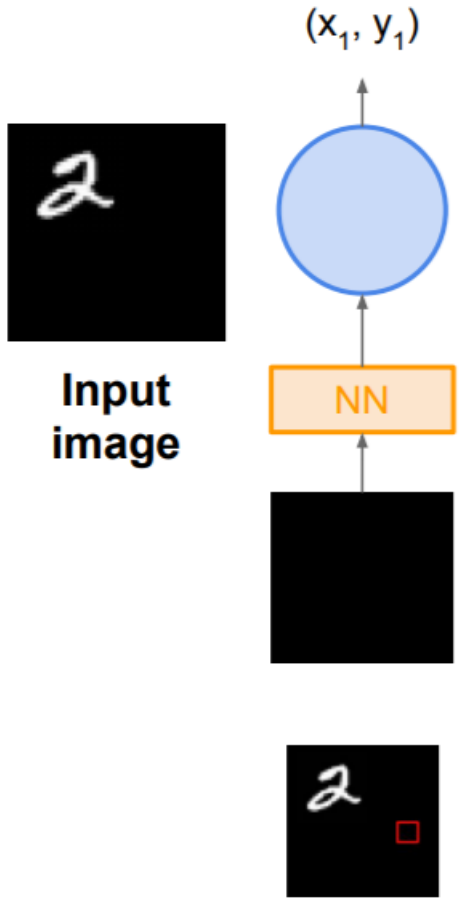
  => learn policy for how to take glimpse actions using REINFORCE



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

REINFORCE in action: Recurrent Attention Model (RAM)

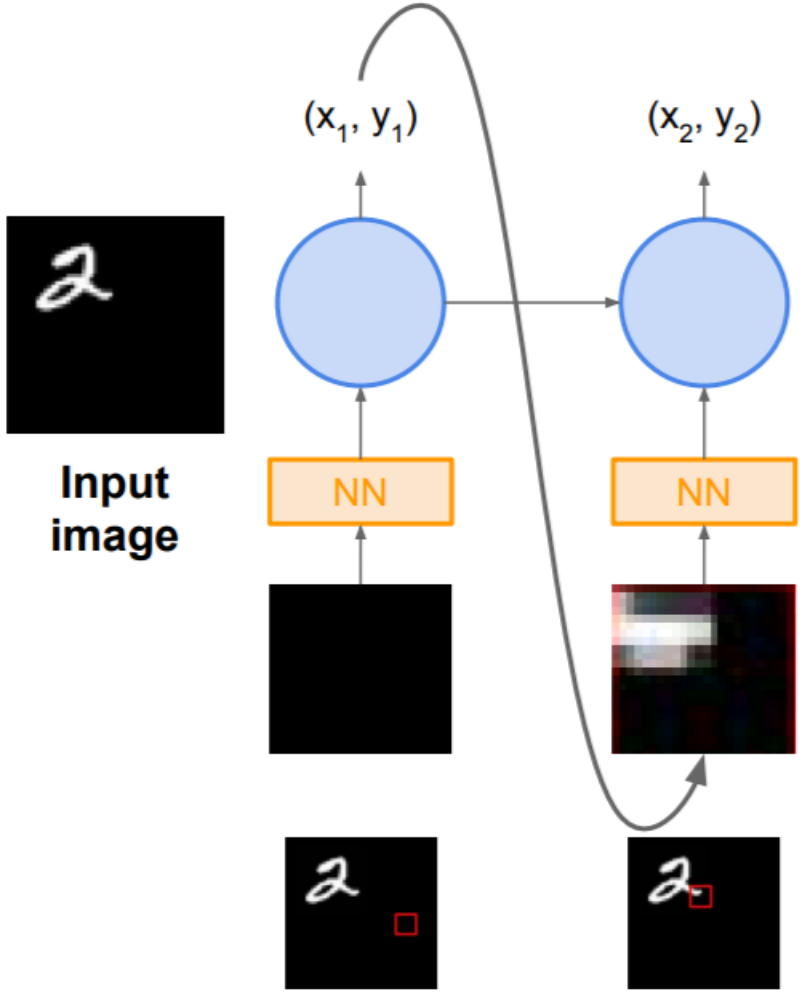
- Given state of glimpses seen so far, use RNN to model the state and output next action



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

REINFORCE in action: Recurrent Attention Model (RAM)

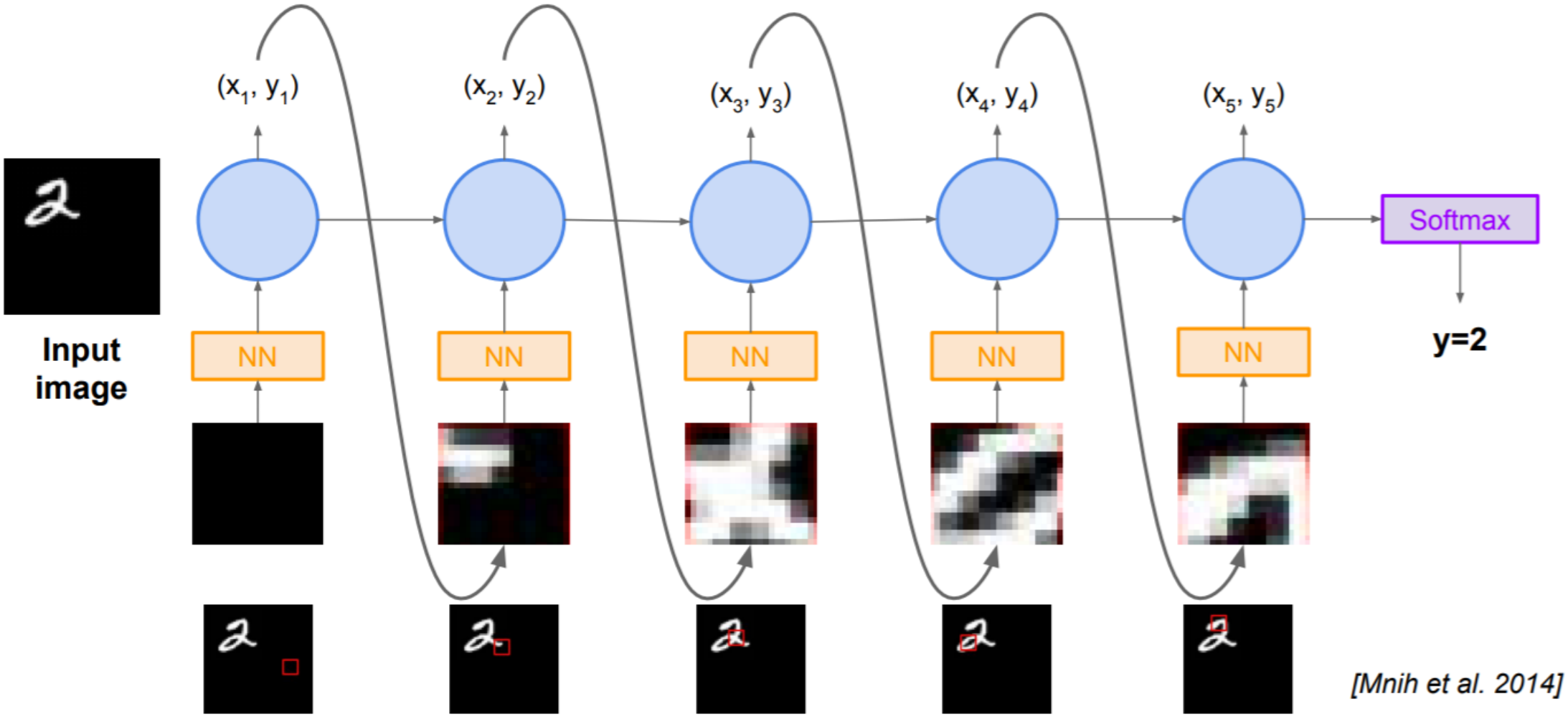
- Given state of glimpses seen so far, use RNN to model the state and output next action



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

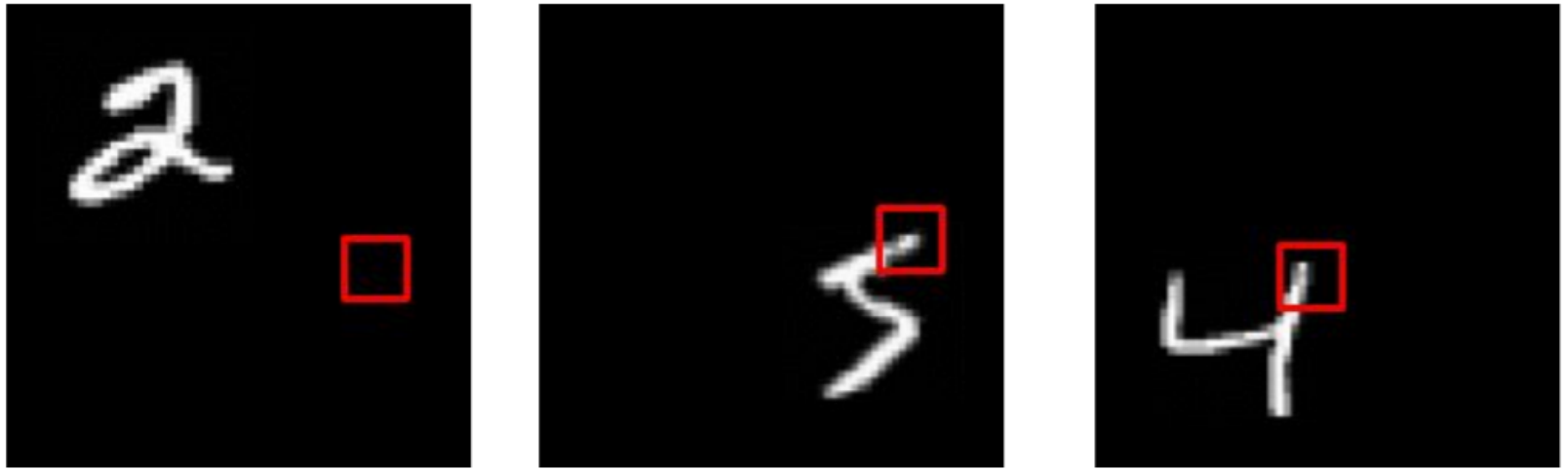
REINFORCE in action: Recurrent Attention Model (RAM)

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



[Mnih et al. 2014]

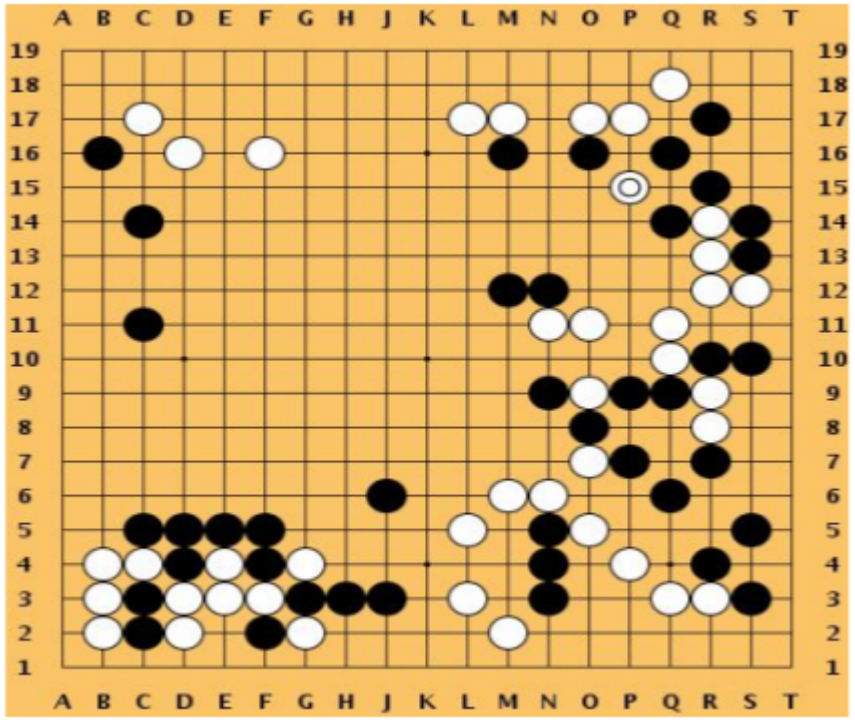
REINFORCE in action: Recurrent Attention Model (RAM)

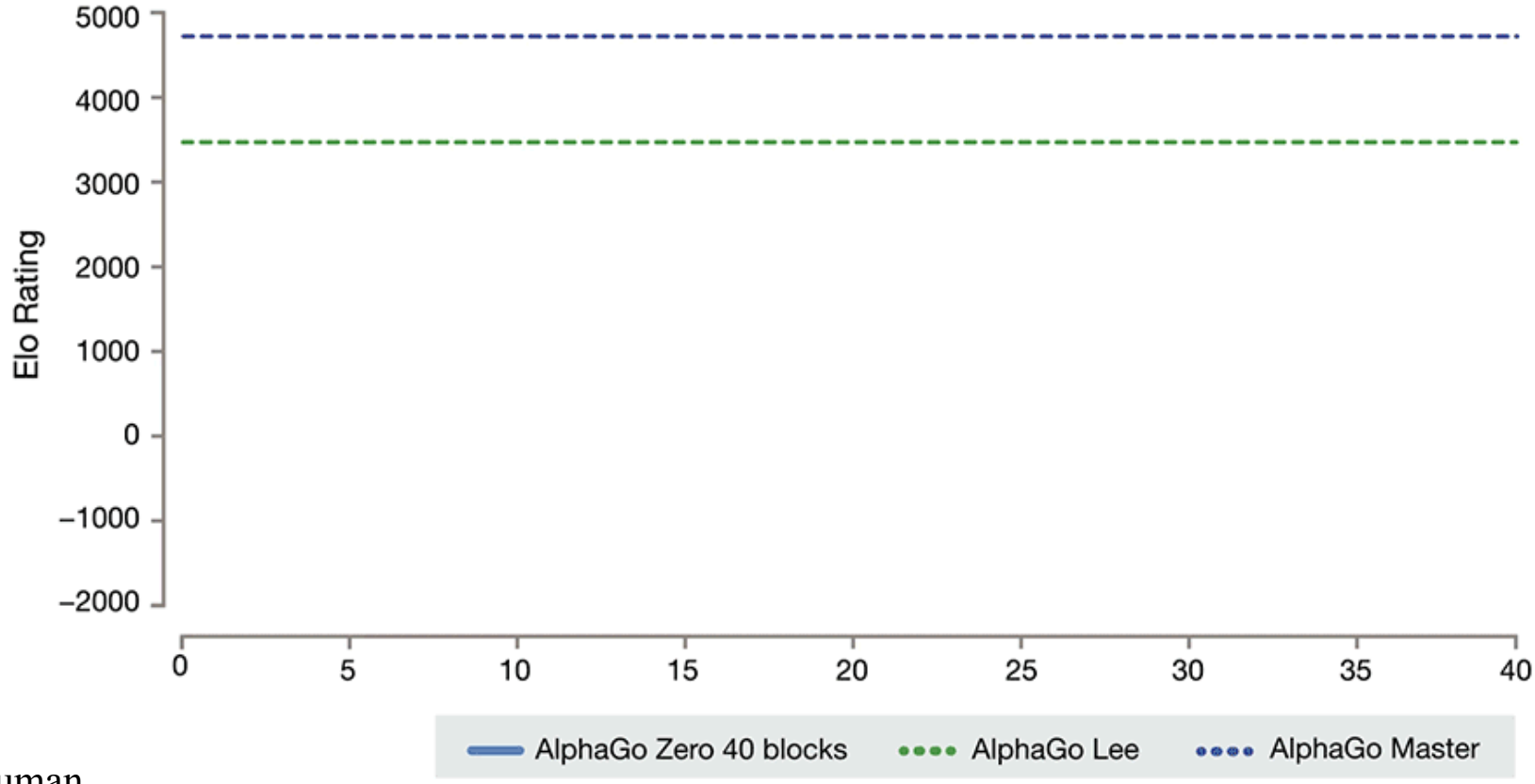


Has also been used in many other tasks including fine-grained image recognition, image captioning, and visual question-answering!

[Mnih et al. 2014]

More policy gradients: AlphaGo Zero





Silver et al., Mastering the game of Go without human knowledge, Nature, 2017.

https://deepmind.com/blog/article/alphago-zero-starting-scratch

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

Summary

- Parameterized Functions
- Deep Q Networks (DQNs)
  - Experience-replay
  - Target functions
- Policy gradients
  - Reinforce
  - Actor-Critic