

# Reinforcement Learning

*from <http://cs231n.stanford.edu/slides/2019/>*

# Administrative

## Grades:

- Midterm grades released last night, see Piazza for more information and statistics
- A2 and milestone grades scheduled for later this week

# Administrative

## Projects:

- All teams must register their project, see Piazza for registration form
- Tiny ImageNet evaluation server is online

# Administrative

## Survey:

- Please fill out the course survey!
- Link on Piazza or <https://goo.gl/forms/eQpVW7IPjqapsDkB2>

# So far... Supervised Learning

**Data:**  $(x, y)$

$x$  is data,  $y$  is label

**Goal:** Learn a *function* to map  $x \rightarrow y$

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



→ Cat

Classification

[This image is CC0 public domain](#)

# So far... Unsupervised Learning

**Data:**  $x$

Just data, no labels!

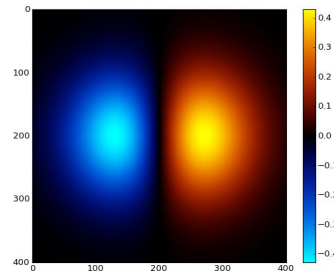
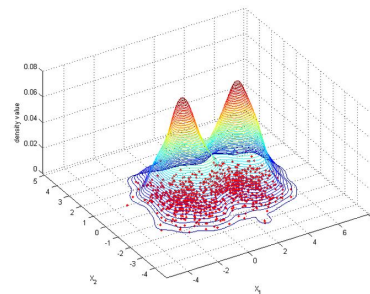
**Goal:** Learn some underlying hidden *structure* of the data

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



Figure copyright Ian Goodfellow, 2016. Reproduced with permission.

1-d density estimation



2-d density estimation

2-d density images [left](#) and [right](#)  
are CC0 public domain

# Today: Reinforcement Learning

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

**Goal:** Learn how to take actions in order to maximize reward



Atari games figure copyright Volodymyr Mnih et al., 2013. Reproduced with permission.

# Overview

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



# Reinforcement Learning

Agent

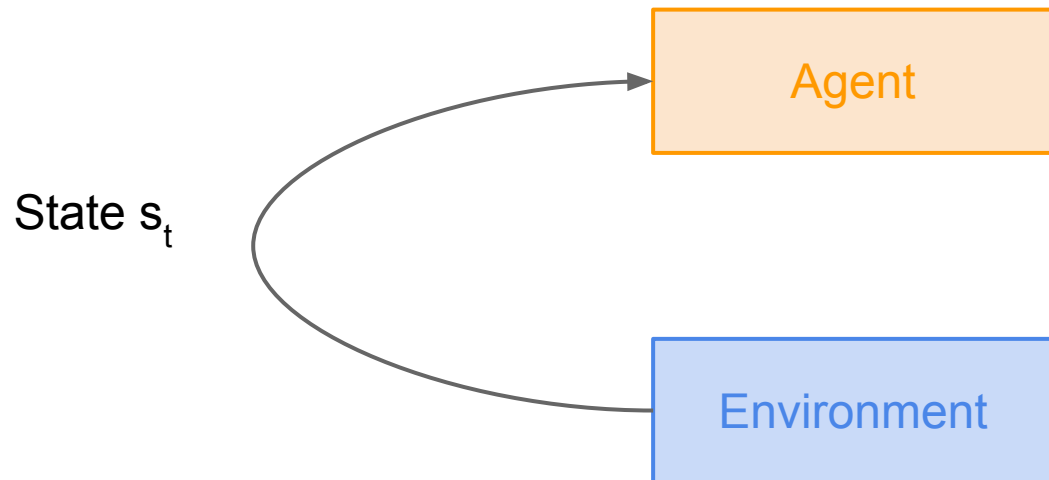


```
graph TD; Agent[Agent] --> Environment[Environment]; Environment --> Agent;
```

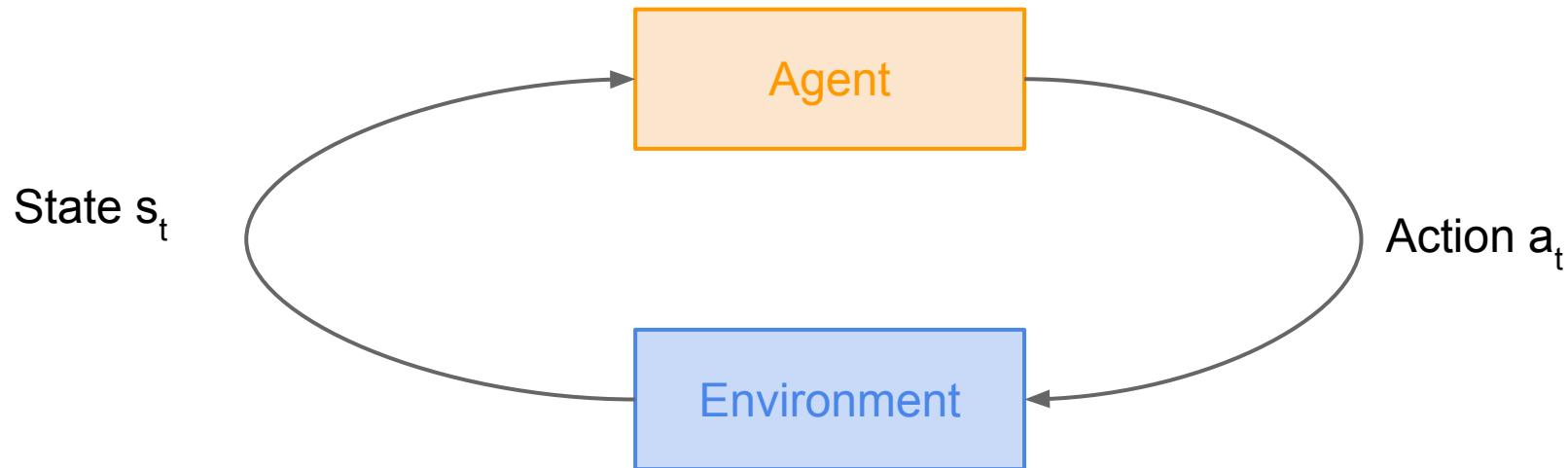
The diagram illustrates the Reinforcement Learning loop. It consists of two main components: the Agent and the Environment. The Agent is represented by an orange box at the top, and the Environment is represented by a blue box at the bottom. Arrows indicate a bidirectional flow of information: the Agent sends actions to the Environment, and the Environment sends observations and rewards back to the Agent.

Environment

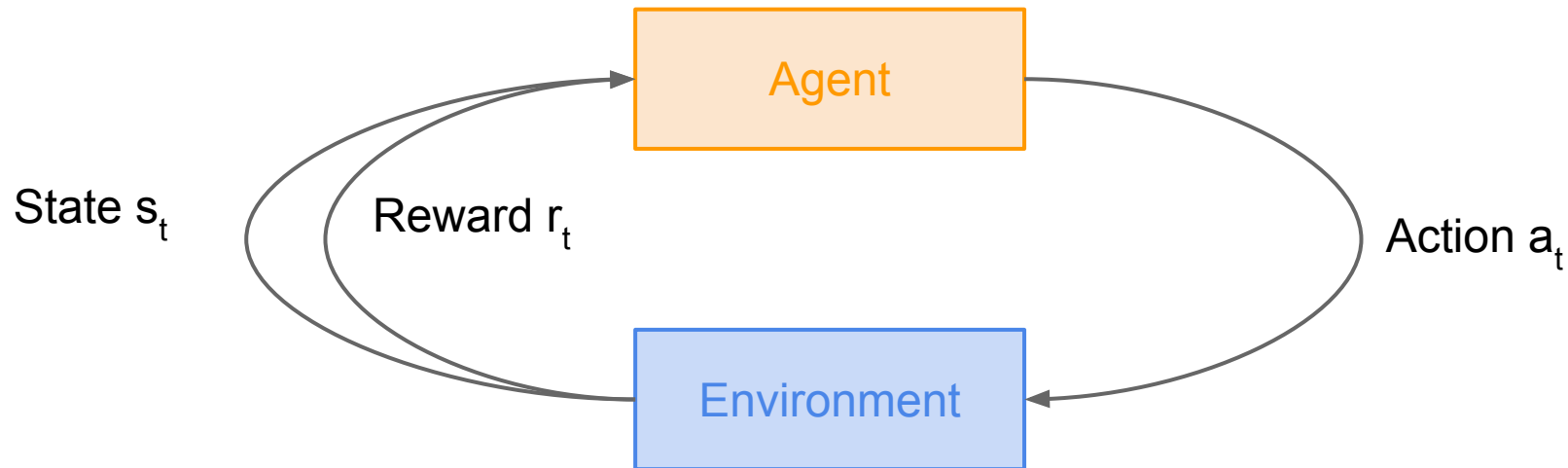
# Reinforcement Learning



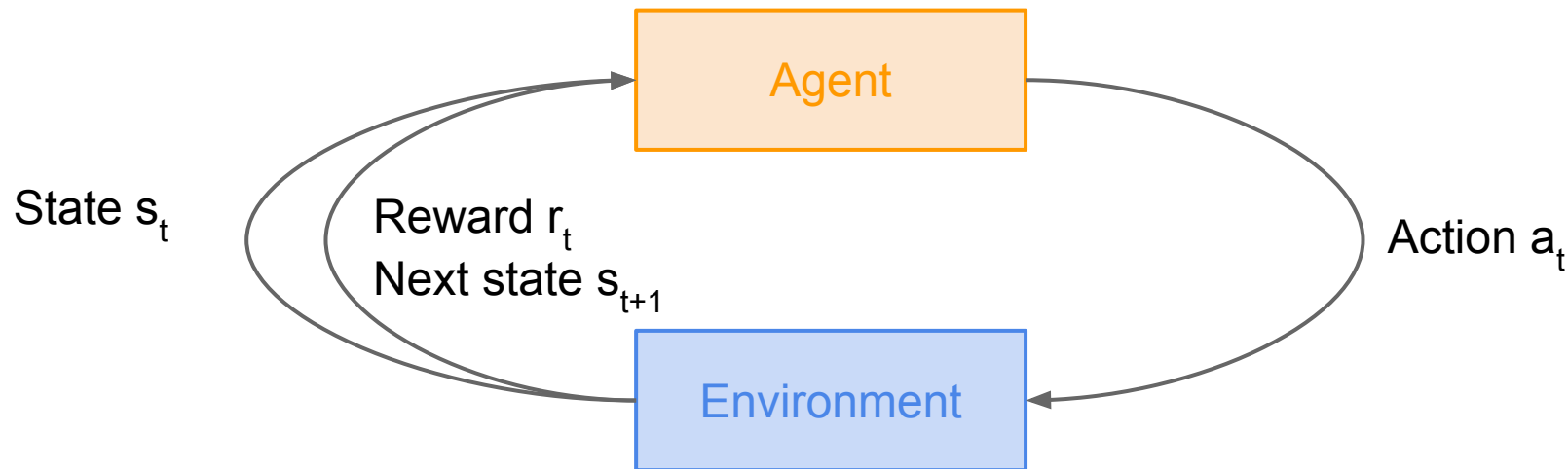
# Reinforcement Learning



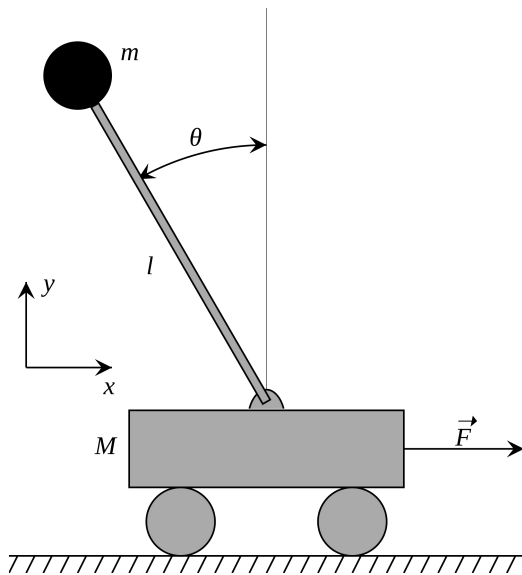
# Reinforcement Learning



# Reinforcement Learning



# Cart-Pole Problem



**Objective:** Balance a pole on top of a movable cart

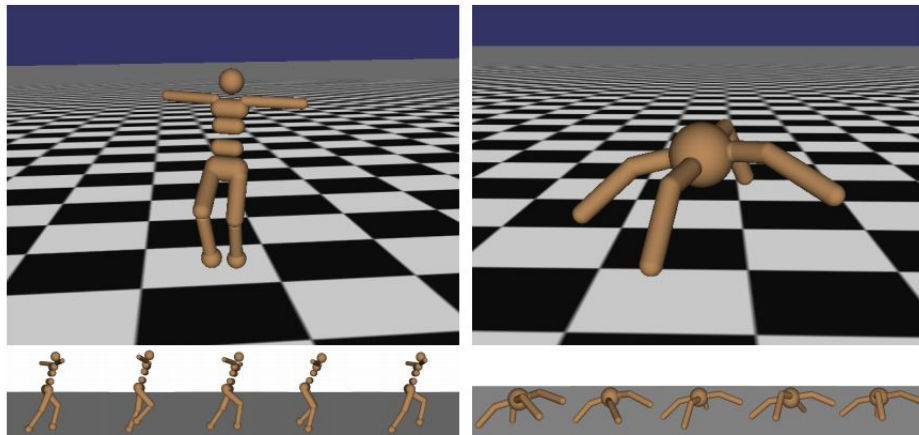
**State:** angle, angular speed, position, horizontal velocity

**Action:** horizontal force applied on the cart

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

This image is CC0 public domain

# 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

Figures copyright John Schulman et al., 2016. Reproduced with permission.

# Atari Games



**Objective:** Complete the game with the highest score

**State:** Raw pixel inputs of the game state

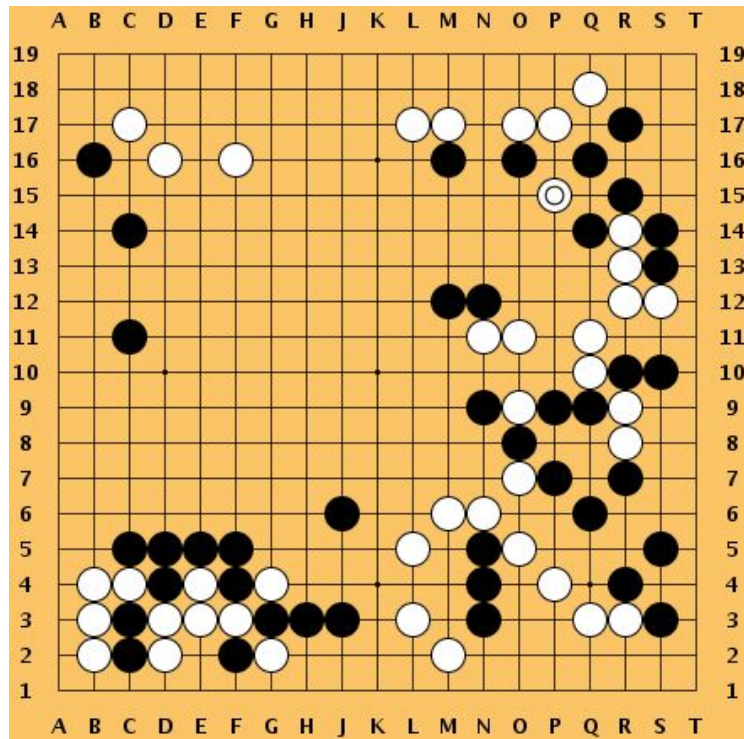
**Action:** Game controls e.g. Left, Right, Up, Down

**Reward:** Score increase/decrease at each time step

Figures copyright Volodymyr Mnih et al., 2013. Reproduced with permission.



# Go



**Objective:** Win the game!

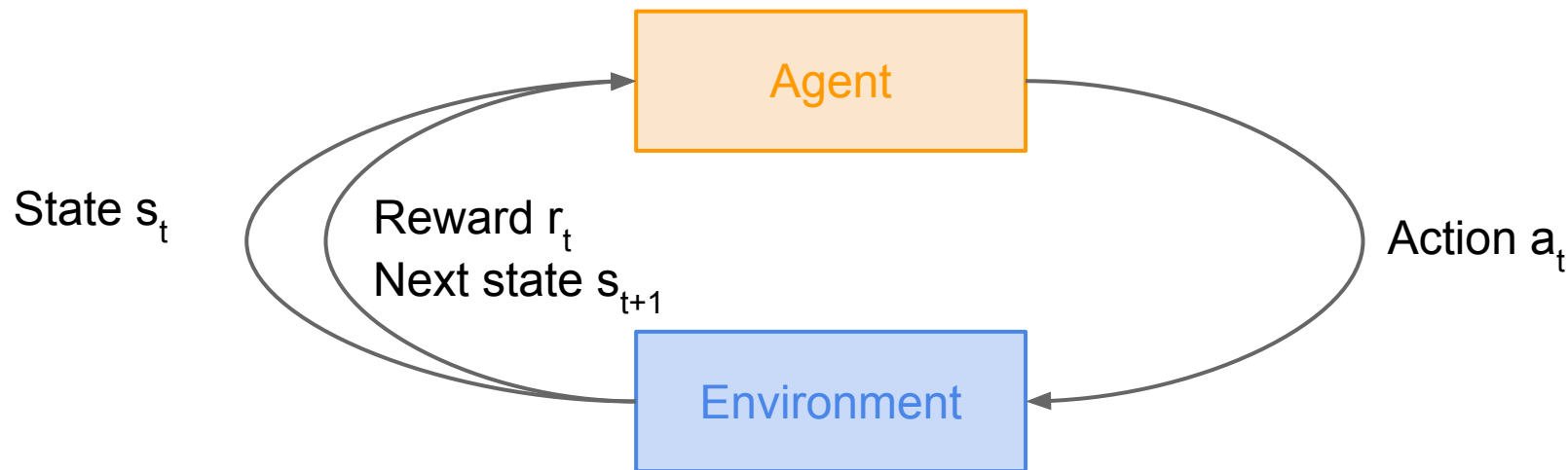
**State:** Position of all pieces

**Action:** Where to put the next piece down

**Reward:** 1 if win at the end of the game, 0 otherwise

[This image is CC0 public domain](#)

# 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

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

$\mathcal{S}$  : set of possible states

$\mathcal{A}$  : set of possible actions

$\mathcal{R}$  : distribution of reward given (state, action) pair

$\mathbb{P}$  : transition probability i.e. distribution over next state given (state, action) pair

$\gamma$  : discount factor

# Markov Decision Process

- At time step  $t=0$ , environment samples initial state  $s_0 \sim p(s_0)$
- Then, for  $t=0$  until done:
  - Agent selects action  $a_t$
  - Environment samples reward  $r_t \sim R(\cdot | s_t, a_t)$
  - Environment samples next state  $s_{t+1} \sim P(\cdot | s_t, a_t)$
  - Agent receives reward  $r_t$  and next state  $s_{t+1}$
- A policy  $\pi$  is a function from  $S$  to  $A$  that specifies what action to take in each state
- **Objective:** find policy  $\pi^*$  that maximizes cumulative discounted reward:  $\sum_{t \geq 0} \gamma^t r_t$


# A simple MDP: Grid World

actions = {

1. right 

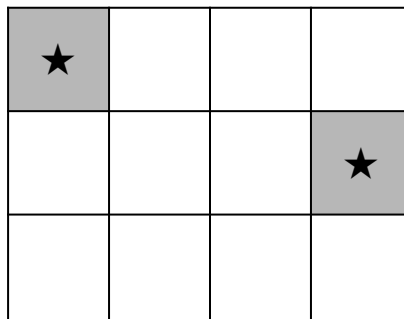
2. left 

3. up 

4. down 

}

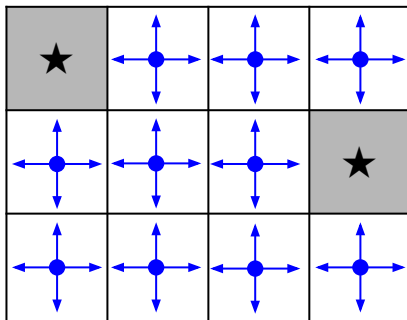
states



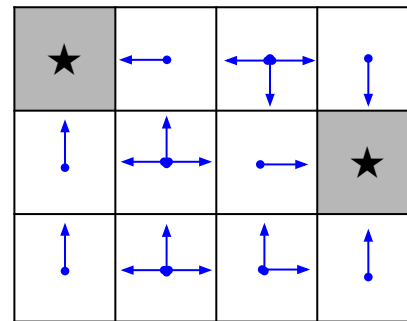
Set a negative “reward”  
for each transition  
(e.g.  $r = -1$ )

**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 $\pi^*$

We want to find optimal policy  $\pi^*$  that maximizes the sum of rewards.

How do we handle the randomness (initial state, transition probability...)?

# The optimal policy $\pi^*$

We want to find optimal policy  $\pi^*$  that maximizes the sum of rewards.

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

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



# Definitions: Value function and Q-value function

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

# Definitions: Value function and Q-value function

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

How good is a state?

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

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

# Definitions: Value function and Q-value function

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

## How good is a state?

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

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

## 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:

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

# Bellman equation

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

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

# Bellman equation

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

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

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

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

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

# Bellman equation

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

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

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

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

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

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

# Solving for the optimal policy

**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  $i \rightarrow \text{infinity}$

# Solving for the optimal policy

**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  $i \rightarrow \text{infinity}$

What's the problem with this?



# Solving for the optimal policy

**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  $i \rightarrow \infty$

What's the problem with this?

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

# Solving for the optimal policy

**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  $i \rightarrow \infty$

What's the problem with this?

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

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

# Solving for the optimal policy: Q-learning

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

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

# Solving for the optimal policy: Q-learning

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

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

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

# 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:

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

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

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

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

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

## Backward Pass

Gradient update (with respect to Q-function parameters  $\theta$ ):

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



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

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

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

## Backward Pass

Gradient update (with respect to Q-function parameters  $\theta$ ):

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

# Case Study: Playing Atari Games



**Objective:** Complete the game with the highest score

**State:** Raw pixel inputs of the game state

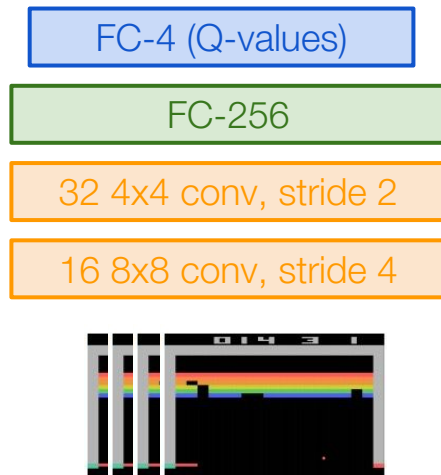
**Action:** Game controls e.g. Left, Right, Up, Down

**Reward:** Score increase/decrease at each time step

Figures copyright Volodymyr Mnih et al., 2013. Reproduced with permission.

# Q-network Architecture

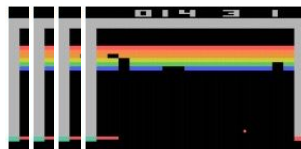
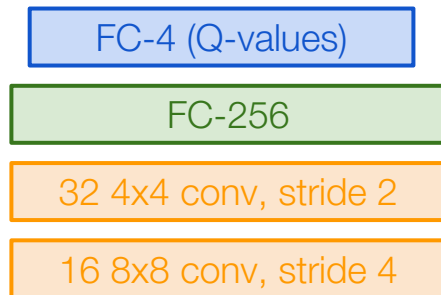
$Q(s, a; \theta)$  :  
neural network  
with weights  $\theta$



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

# Q-network Architecture

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

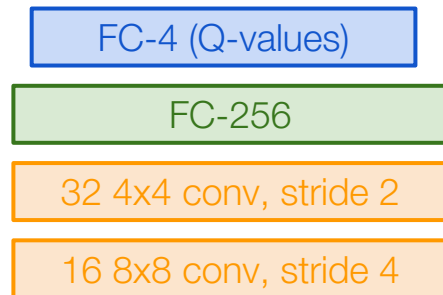


← Input: state  $s_t$

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

# Q-network Architecture

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



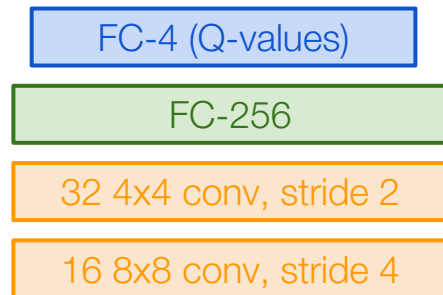
← Familiar conv layers,  
FC layer



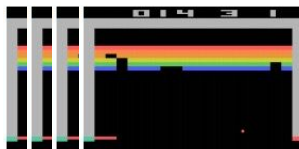
**Current state  $s_t$ : 84x84x4 stack of last 4 frames**  
(after RGB->grayscale conversion, downsampling, and cropping)

# Q-network Architecture

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



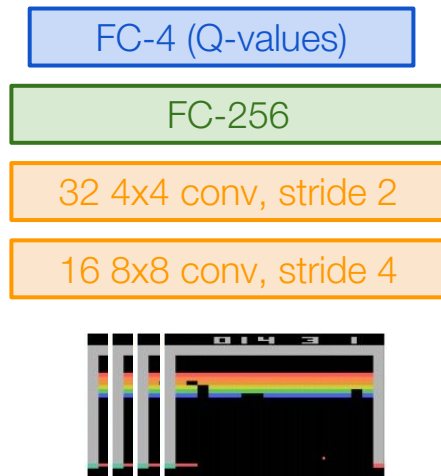
← Last FC layer has 4-d output (if 4 actions), corresponding to  $Q(s_t, a_1)$ ,  $Q(s_t, a_2)$ ,  $Q(s_t, a_3)$ ,  $Q(s_t, a_4)$



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

# Q-network Architecture

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



← Last FC layer has 4-d output (if 4 actions), corresponding to  $Q(s_t, a_1)$ ,  $Q(s_t, a_2)$ ,  $Q(s_t, a_3)$ ,  $Q(s_t, a_4)$

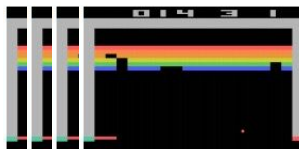
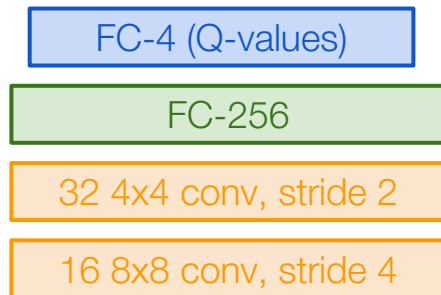
Number of actions between 4-18 depending on Atari game

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

# Q-network Architecture

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

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



**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),  
corresponding to  $Q(s_t, a_1)$ ,  $Q(s_t, a_2)$ ,  $Q(s_t, a_3)$ ,  
 $Q(s_t, a_4)$

Number of actions between 4-18  
depending on Atari game



# Training the Q-network: Loss function (from before)

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

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

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

## Backward Pass

Gradient update (with respect to Q-function parameters  $\theta$ ):

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

# Training the Q-network: Experience Replay

Learning from batches of consecutive samples is problematic:

- Samples are correlated => inefficient learning
- Current Q-network parameters determines next training samples (e.g. if maximizing action is to move left, training samples will be dominated by samples from left-hand size) => can lead to bad feedback loops

# Training the Q-network: Experience Replay

Learning from batches of consecutive samples is problematic:

- Samples are correlated => inefficient learning
- Current Q-network parameters determines next training samples (e.g. if maximizing action is to move left, training samples will be dominated by samples from left-hand size) => can lead to bad feedback loops

Address these problems using **experience replay**

- Continually update a **replay memory** table of transitions ( $s_t, a_t, r_t, s_{t+1}$ ) as game (experience) episodes are played
- Train Q-network on random minibatches of transitions from the replay memory, instead of consecutive samples

# Training the Q-network: Experience Replay

Learning from batches of consecutive samples is problematic:

- Samples are correlated => inefficient learning
- Current Q-network parameters determines next training samples (e.g. if maximizing action is to move left, training samples will be dominated by samples from left-hand size) => can lead to bad feedback loops

Address these problems using **experience replay**

- Continually update a **replay memory** table of transitions ( $s_t, a_t, r_t, s_{t+1}$ ) as game (experience) episodes are played
- Train Q-network on random minibatches of transitions from the replay memory, instead of consecutive samples

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

# 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  $\epsilon$  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
  
```

---

# 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**

    Initialize sequence  $s_1 = \{x_1\}$  and preprocessed sequenced  $\phi_1 = \phi(s_1)$

**for**  $t = 1, T$  **do**

        With probability  $\epsilon$  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 replay memory, Q-network

# 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)

    Initialize sequence  $s_1 = \{x_1\}$  and preprocessed sequence  $\phi_1 = \phi(s_1)$

**for**  $t = 1, T$  **do**

        With probability  $\epsilon$  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**

---



# 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**

    Initialize sequence  $s_1 = \{x_1\}$  and preprocessed sequenced  $\phi_1 = \phi(s_1)$

**for**  $t = 1, T$  **do**

        With probability  $\epsilon$  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



# 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**

    Initialize sequence  $s_1 = \{x_1\}$  and preprocessed sequenced  $\phi_1 = \phi(s_1)$

**for**  $t = 1, T$  **do**

        With probability  $\epsilon$  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 timestep  $t$   
of the game

# 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**

    Initialize sequence  $s_1 = \{x_1\}$  and preprocessed sequenced  $\phi_1 = \phi(s_1)$

**for**  $t = 1, T$  **do**

        With probability  $\epsilon$  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

# 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**

    Initialize sequence  $s_1 = \{x_1\}$  and preprocessed sequenced  $\phi_1 = \phi(s_1)$

**for**  $t = 1, T$  **do**

        With probability  $\epsilon$  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 action ( $a_t$ ),  
and observe the  
reward  $r_t$  and next  
state  $s_{t+1}$

# 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**

    Initialize sequence  $s_1 = \{x_1\}$  and preprocessed sequenced  $\phi_1 = \phi(s_1)$

**for**  $t = 1, T$  **do**

        With probability  $\epsilon$  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



# 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**

    Initialize sequence  $s_1 = \{x_1\}$  and preprocessed sequenced  $\phi_1 = \phi(s_1)$

**for**  $t = 1, T$  **do**

        With probability  $\epsilon$  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  $y_j = \begin{cases} r_j & \text{for terminal } \phi_{j+1} \\ r_j + \gamma \max_{a'} Q(\phi_{j+1}, a'; \theta) & \text{for non-terminal } \phi_{j+1} \end{cases}$

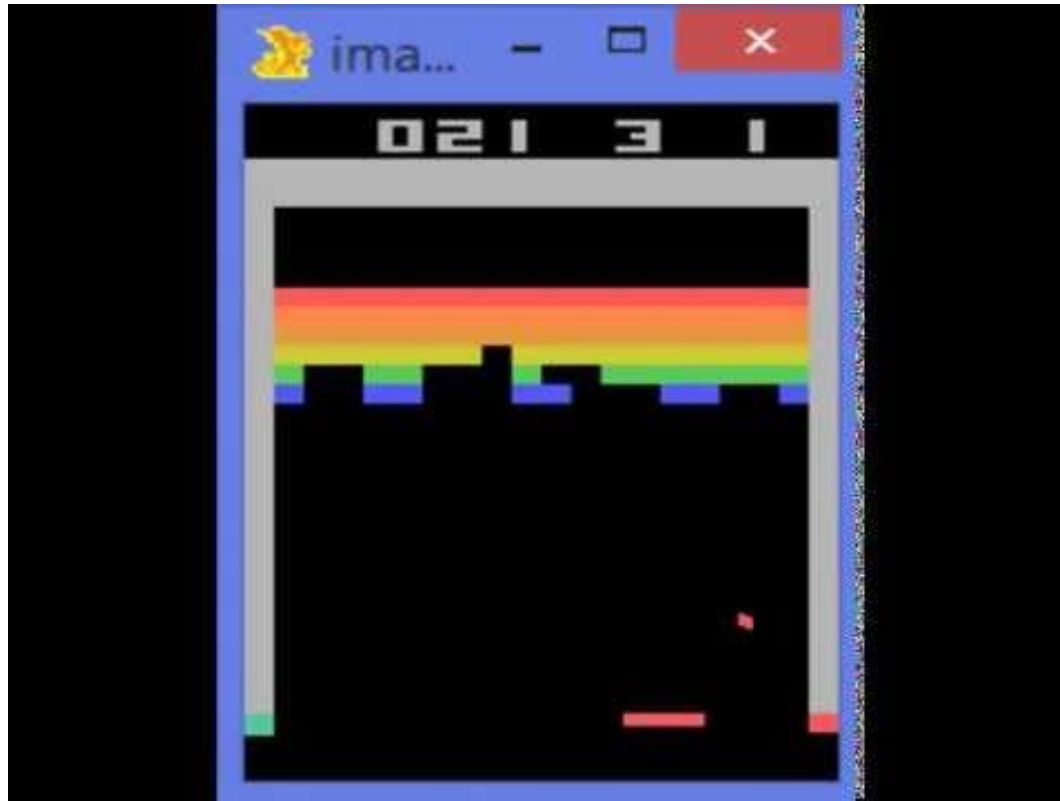
        Perform a gradient descent step on  $(y_j - Q(\phi_j, a_j; \theta))^2$  according to equation 3

**end for**

**end for**

---

← Experience Replay:  
Sample a random  
minibatch of transitions  
from replay memory  
and perform a gradient  
descent step



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

Video by Károly Zsolnai-Fehér. Reproduced with permission.

# Policy Gradients

What is a problem with Q-learning?

The Q-function can be very complicated!

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

# Policy Gradients

What is a problem with Q-learning?

The Q-function can be very complicated!

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

But the policy can be much simpler: just close your hand

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



# Policy Gradients

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

For each policy, define its value:

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

# 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?

# 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:

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

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

# REINFORCE algorithm

Expected reward:

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

# REINFORCE algorithm

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

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

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

# REINFORCE algorithm

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

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

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

Intractable! Gradient of an expectation is problematic when  $p$  depends on  $\theta$

# REINFORCE algorithm

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

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

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

Intractable! Gradient of an expectation is problematic when  $p$  depends on  $\theta$

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



# REINFORCE algorithm

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

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

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

Intractable! Gradient of an expectation is problematic when  $p$  depends on  $\theta$

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

If we inject this back:

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

Can estimate with  
Monte Carlo sampling

# REINFORCE algorithm

Can we compute those quantities without knowing the transition probabilities?

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

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

# 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

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

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!

Therefore when sampling a trajectory  $\tau$ , we can estimate  $J(\theta)$  with

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

# Intuition

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

**Interpretation:**

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

# Intuition

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

## Interpretation:

- If  $r(\tau)$  is high, push up the probabilities of the actions seen
- If  $r(\tau)$  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!**

# Intuition

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

## Interpretation:

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

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

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



# Variance reduction

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

# Variance reduction

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

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

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

# Variance reduction

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

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

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

**Second idea:** Use discount factor  $\gamma$  to ignore delayed effects

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

# Variance reduction: Baseline

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

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

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

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

# 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

# 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”

# How to choose the baseline?

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

Q: What does this remind you of?

# How to choose the baseline?

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

Q: What does this remind you of?

A: Q-function and value function!



# How to choose the baseline?

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

Q: What does this remind you of?

A: Q-function and value function!

Intuitively, we are happy with an action  $a_t$  in a state  $s_t$  if  $Q^\pi(s_t, a_t) - V^\pi(s_t)$  is large. On the contrary, we are unhappy with an action if it's small.

# How to choose the baseline?

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

Q: What does this remind you of?

A: Q-function and value function!

Intuitively, we are happy with an action  $a_t$  in a state  $s_t$  if  $Q^\pi(s_t, a_t) - V^\pi(s_t)$  is large. On the contrary, we are unhappy with an action if it's small.

Using this, we get the estimator: 
$$\nabla_\theta J(\theta) \approx \sum_{t \geq 0} (Q^{\pi_\theta}(s_t, a_t) - V^{\pi_\theta}(s_t)) \nabla_\theta \log \pi_\theta(a_t | s_t)$$

# Actor-Critic Algorithm

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

**Yes**, using Q-learning! We can combine Policy Gradients and Q-learning by training both an **actor** (the policy) and a **critic** (the Q-function).

- The actor decides which action to take, and the critic tells the actor how good its action was and how it should adjust
- Also alleviates the task of the critic as it only has to learn the values of (state, action) pairs generated by the policy
- Can also incorporate Q-learning tricks e.g. experience replay
- **Remark:** we can define by the **advantage function** how much an action was better than expected

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

# Actor-Critic Algorithm

Initialize policy parameters  $\theta$ , critic parameters  $\phi$

**For** iteration=1, 2 ... **do**

    Sample  $m$  trajectories under the current policy

$\Delta\theta \leftarrow 0$

**For**  $i=1, \dots, m$  **do**

**For**  $t=1, \dots, T$  **do**

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

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

$$\Delta\phi \leftarrow \sum_i \sum_t \nabla_{\phi} \|A_t^i\|^2$$

$$\theta \leftarrow \theta + \alpha \Delta\theta$$

$$\phi \leftarrow \phi + \beta \Delta\phi$$

**End for**

# REINFORCE in action: Recurrent Attention Model (RAM)

**Objective:** Image Classification

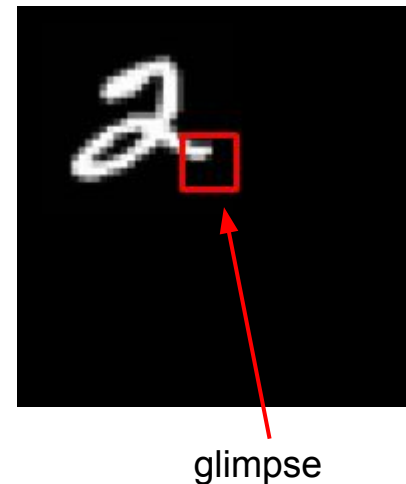
Take a sequence of “glimpses” selectively focusing on regions of the image, to predict class

- Inspiration from human perception and eye movements
- Saves computational resources => scalability
- Able to ignore clutter / irrelevant parts of image

**State:** Glimpses seen so far

**Action:** (x,y) coordinates (center of glimpse) of where to look next in image

**Reward:** 1 at the final timestep if image correctly classified, 0 otherwise



*[Mnih et al. 2014]*

# REINFORCE in action: Recurrent Attention Model (RAM)

**Objective:** Image Classification

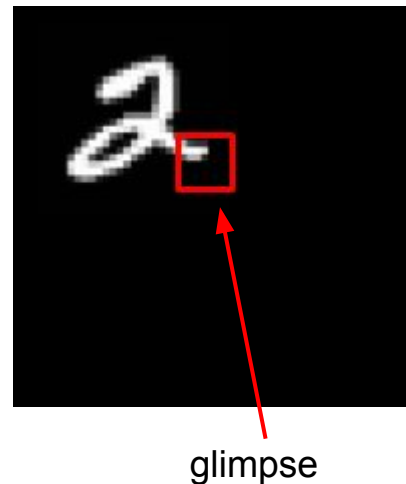
Take a sequence of “glimpses” selectively focusing on regions of the image, to predict class

- Inspiration from human perception and eye movements
- Saves computational resources => scalability
- Able to ignore clutter / irrelevant parts of image

**State:** Glimpses seen so far

**Action:** (x,y) coordinates (center of glimpse) of where to look next in image

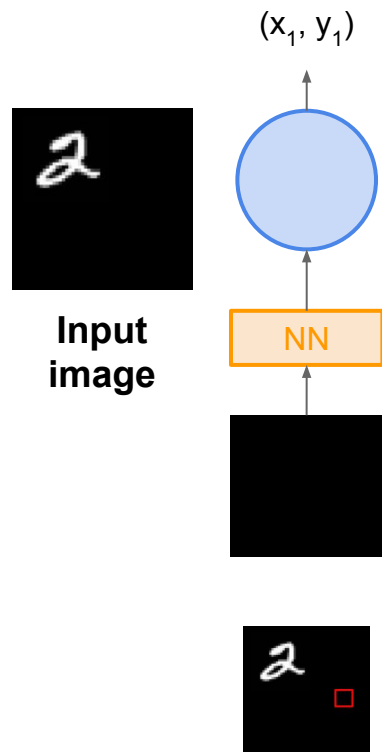
**Reward:** 1 at the final timestep if image correctly classified, 0 otherwise



Glimpsing is a non-differentiable operation => learn policy for how to take glimpse actions using REINFORCE  
Given state of glimpses seen so far, use RNN to model the state and output next action

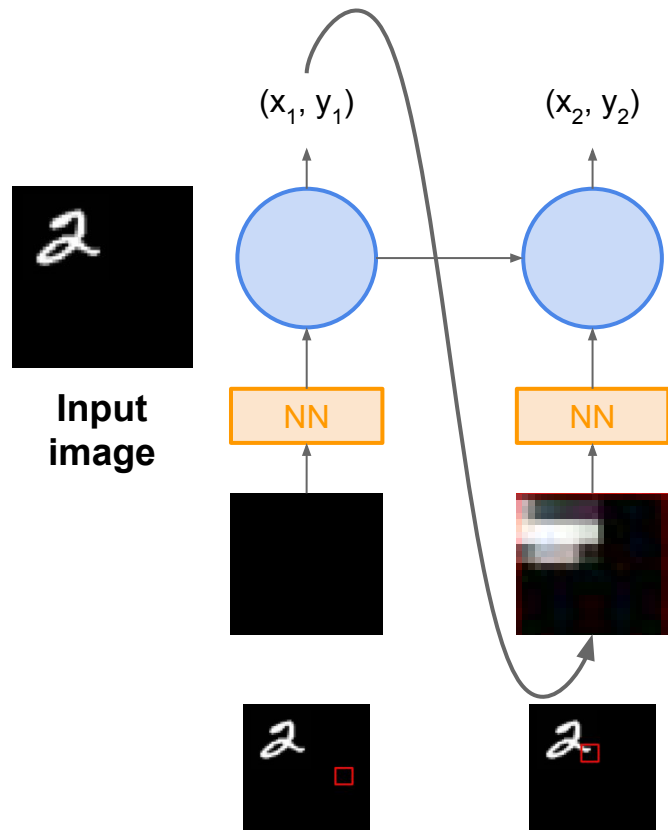
*[Mnih et al. 2014]*

# REINFORCE in action: Recurrent Attention Model (RAM)



*[Mnih et al. 2014]*

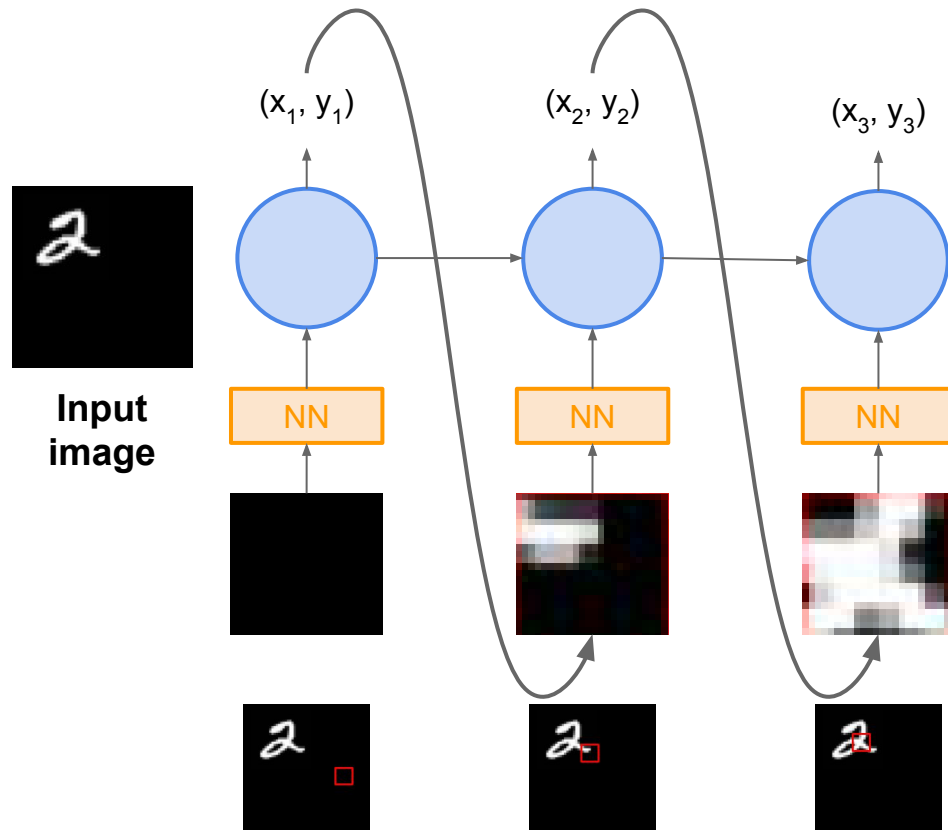
# REINFORCE in action: Recurrent Attention Model (RAM)



[Mnih et al. 2014]

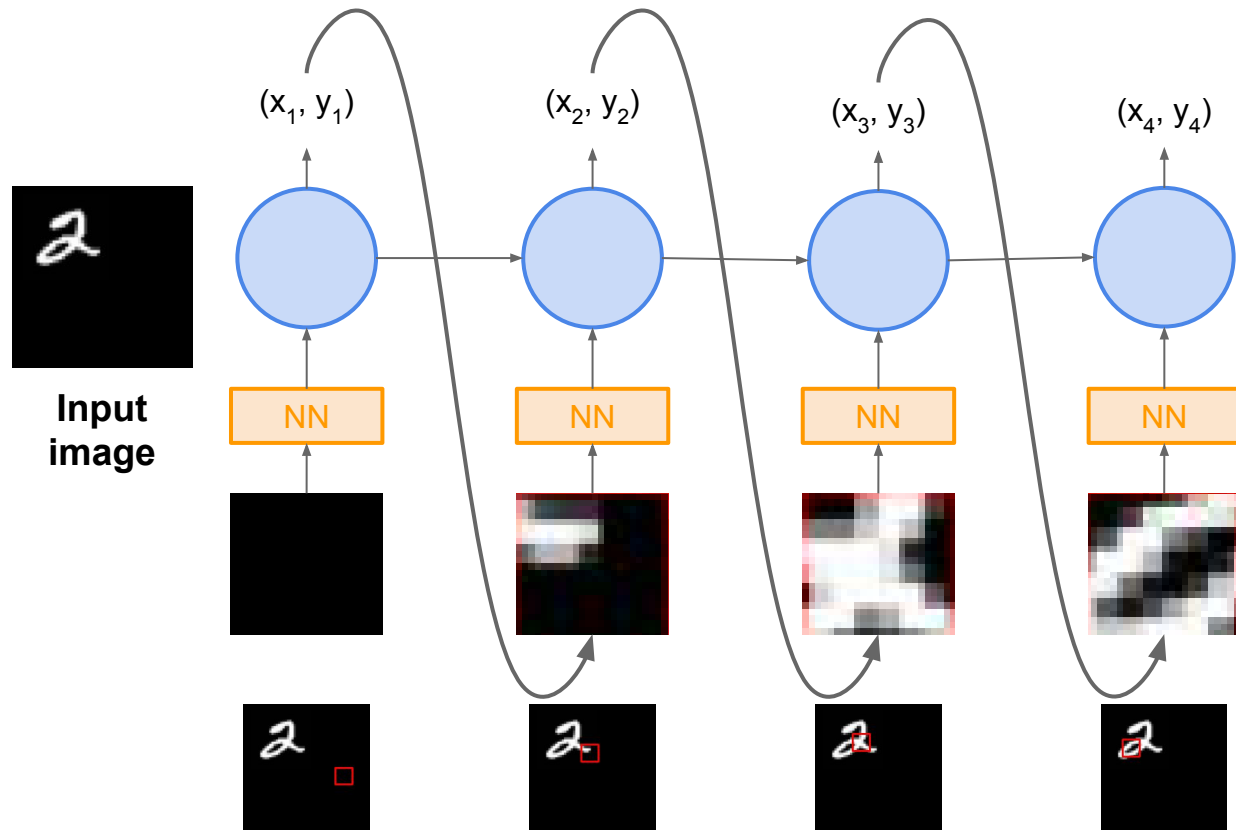


# REINFORCE in action: Recurrent Attention Model (RAM)



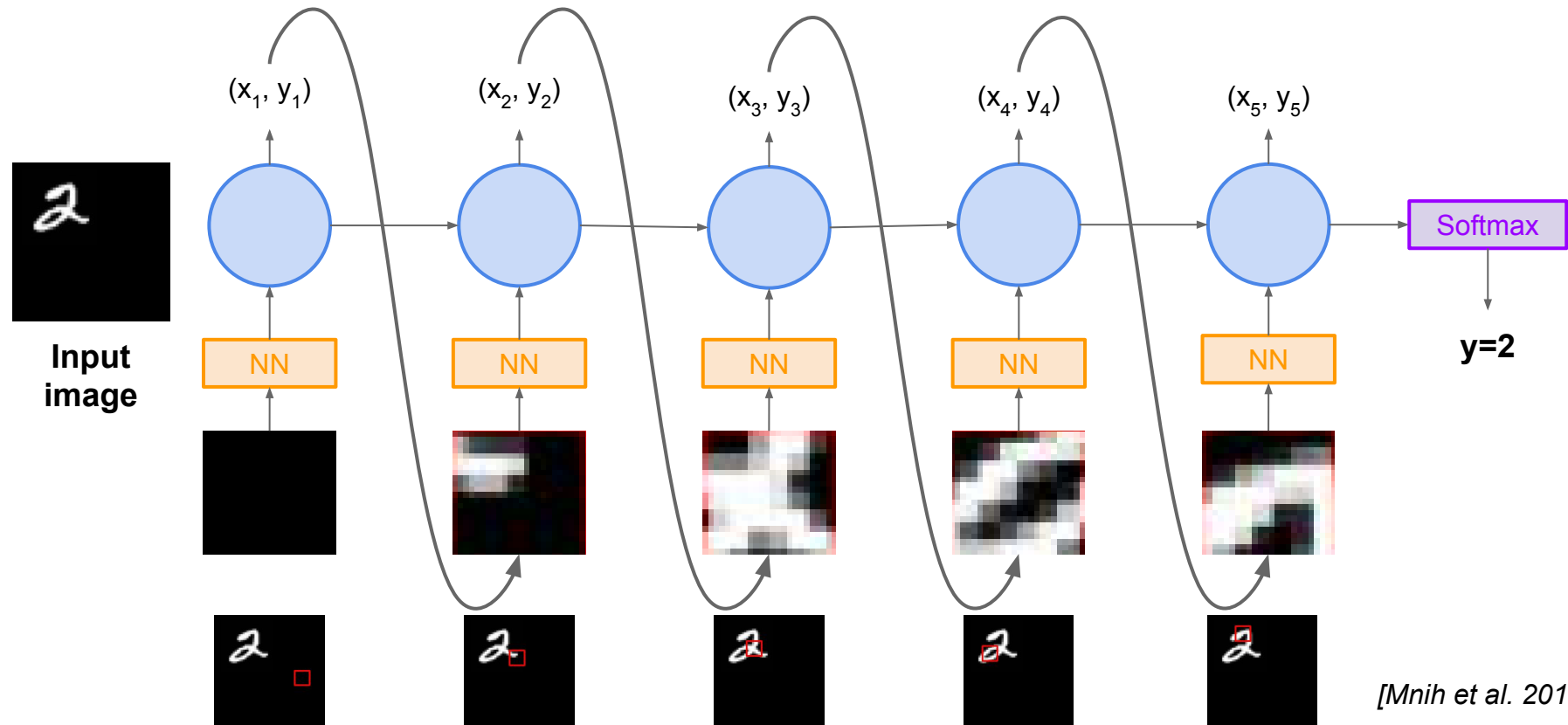
[Mnih et al. 2014]

# REINFORCE in action: Recurrent Attention Model (RAM)

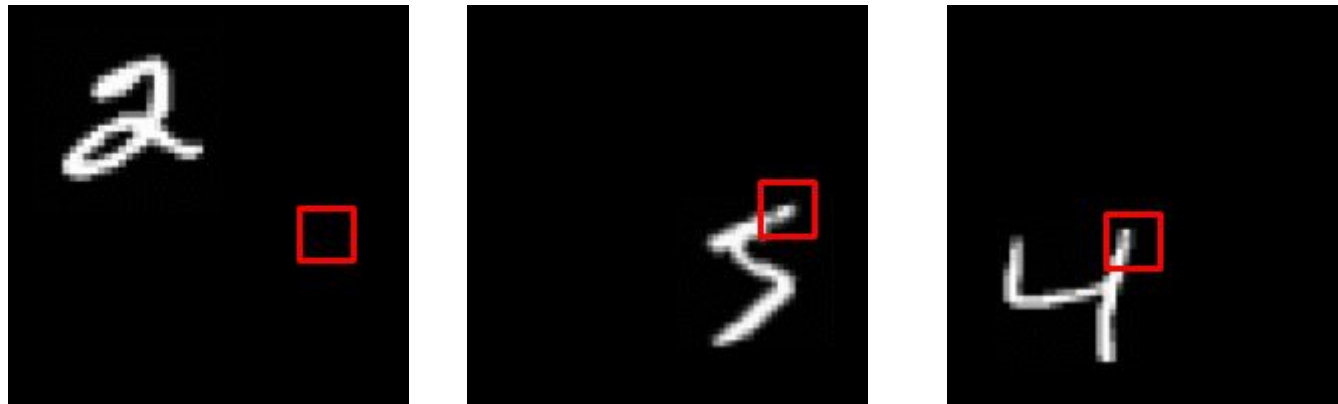


[Mnih et al. 2014]

# REINFORCE in action: Recurrent Attention Model (RAM)



# 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!

Figures copyright Daniel Levy, 2017. Reproduced with permission.

*[Mnih et al. 2014]*

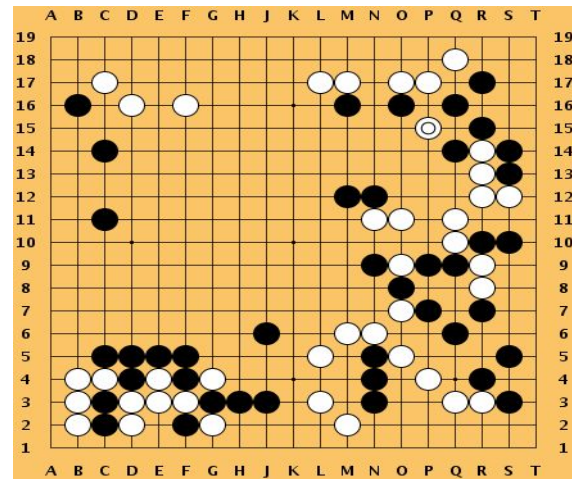
# More policy gradients: AlphaGo

## Overview:

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

## 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 combine policy and value networks in a Monte Carlo Tree Search algorithm to select actions by lookahead search



[Silver et al.,  
Nature 2016]

This image is CC0 public domain

# 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