Deep Reinforcement Learning

Learn to make good sequence of decision

Some slides were adated/taken from various sources, including Andrew Ng's Coursera Lectures, CS231n: Convolutional Neural Networks for Visual Recognition lectures, Stanford University CS Waterloo Canada lectures, Aykut Erdem, et.al. tutorial on Deep Learning in Computer Vision, Ismini Lourentzou's lecture slide on "Introduction to Deep Learning", Ramprasaath's lecture slides, and many more. We thankfully acknowledge them. Students are requested to use this material for their study only and NOT to distribute it.

Outlines

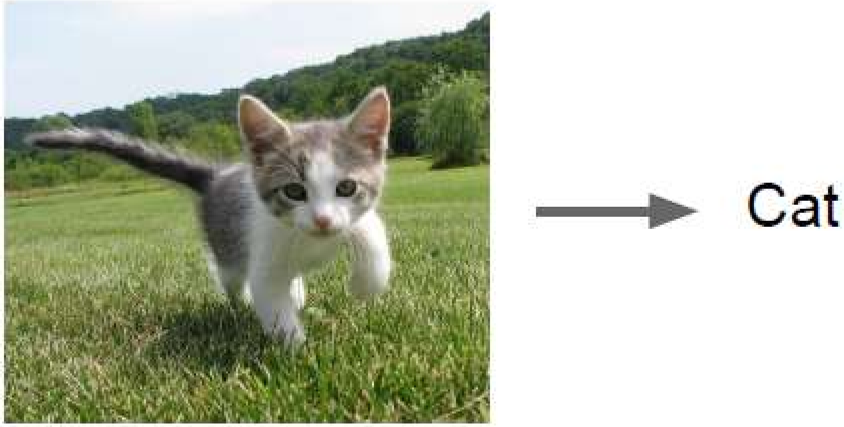
- What is Reinforcement Learning?
- Markov Decision Processes
- Q-Learning
- Policy Gradients

Supervised Learning

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

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

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



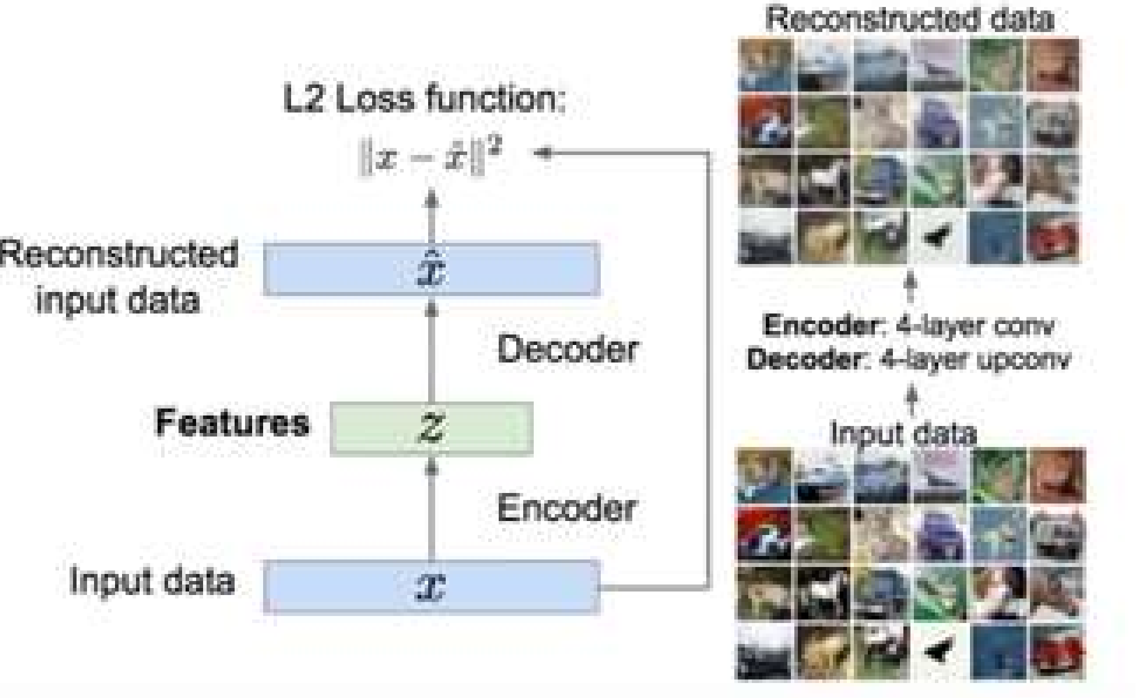
Classification

Un-Supervised 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



Reinforcement Learning

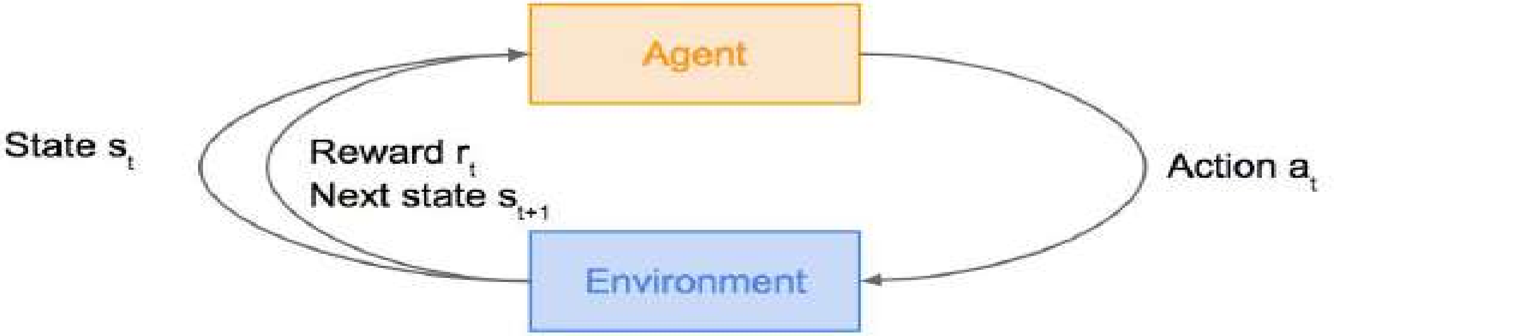
Repeated interaction with world

Learn to make good sequence of decisions

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



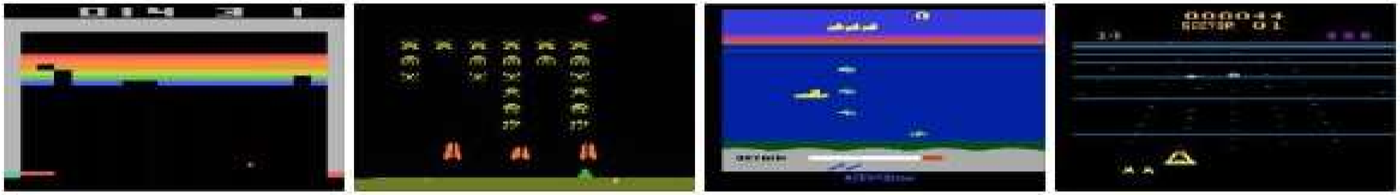
Agent

State s_t

Reward r_t
Next state s_{t+1}

Action a_t

Environment

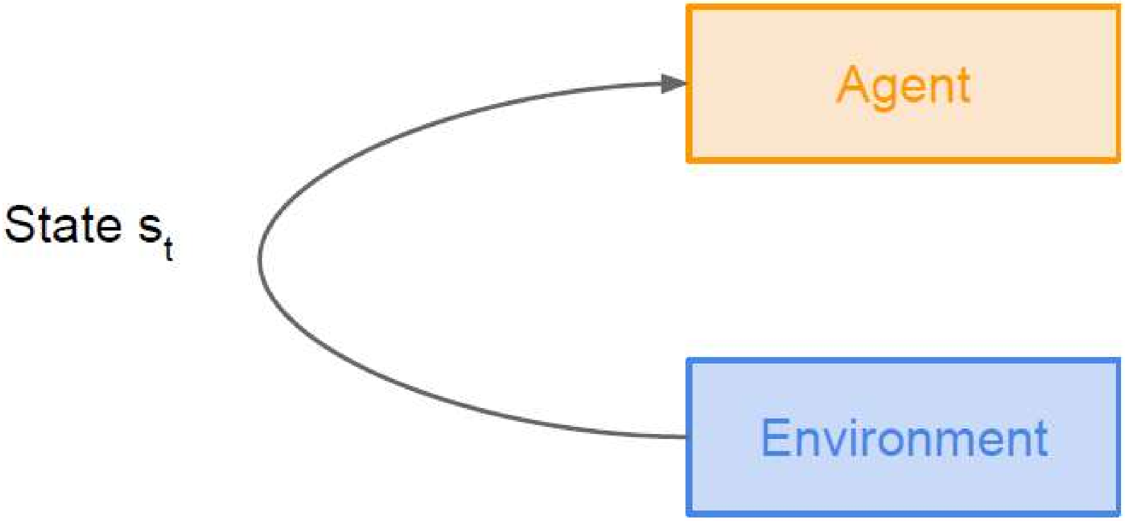


Reinforcement Learning

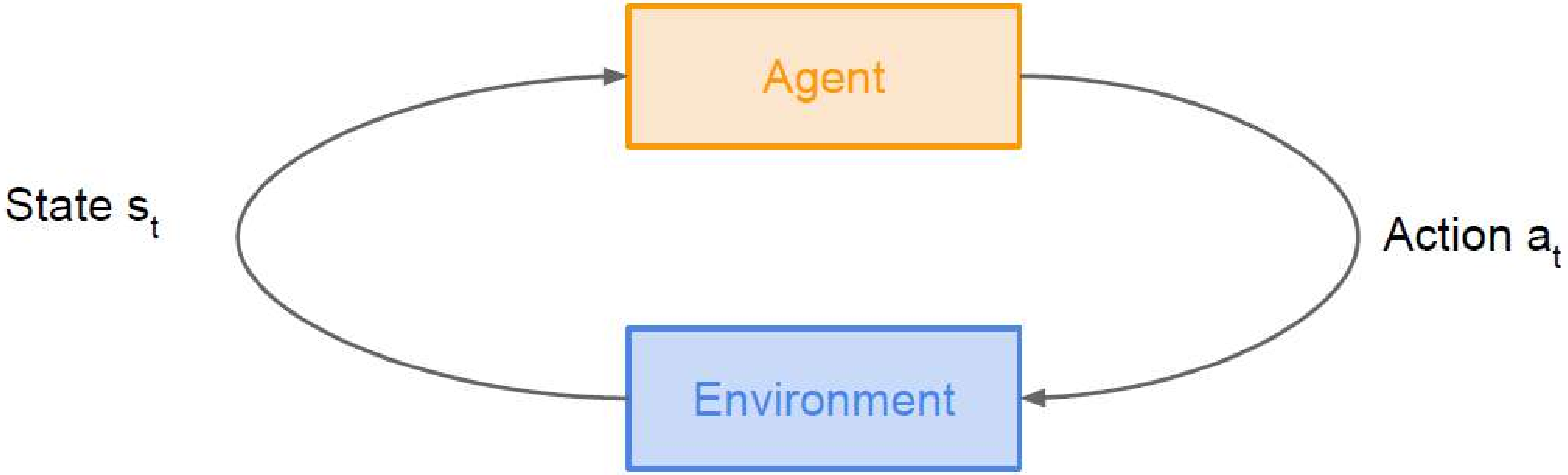
Agent

Environment

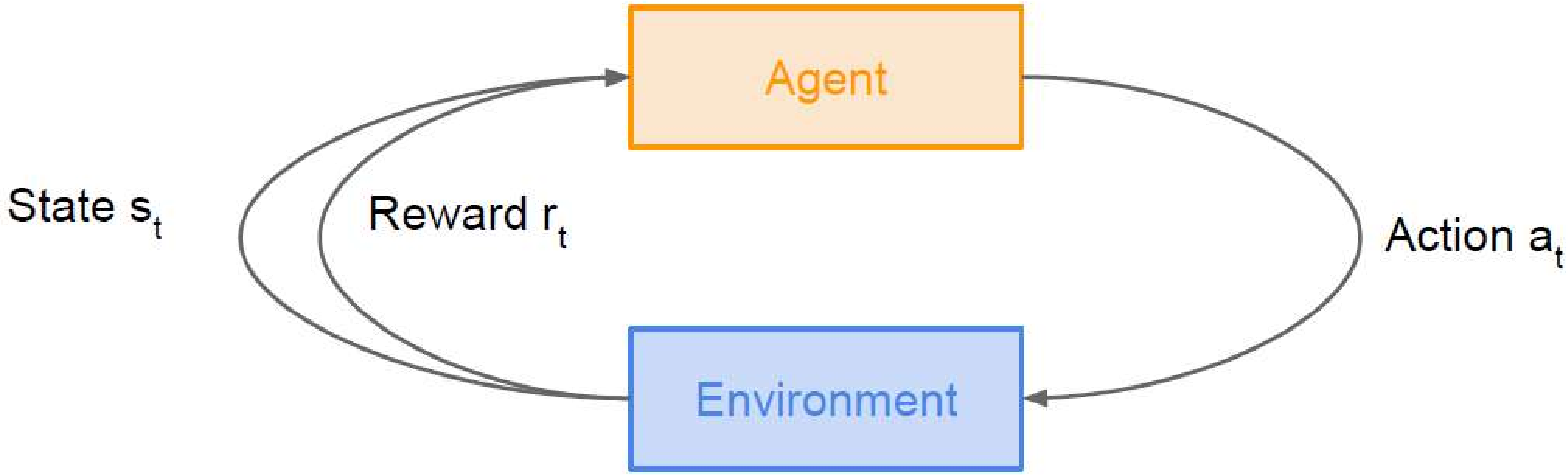
Reinforcement Learning



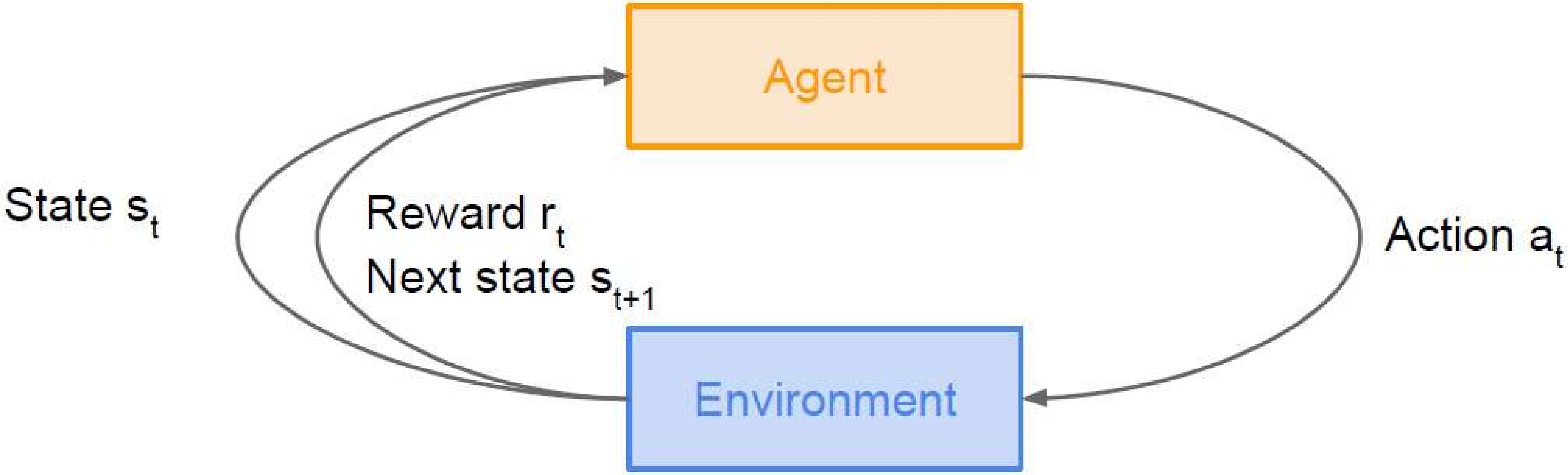
Reinforcement Learning



Reinforcement Learning

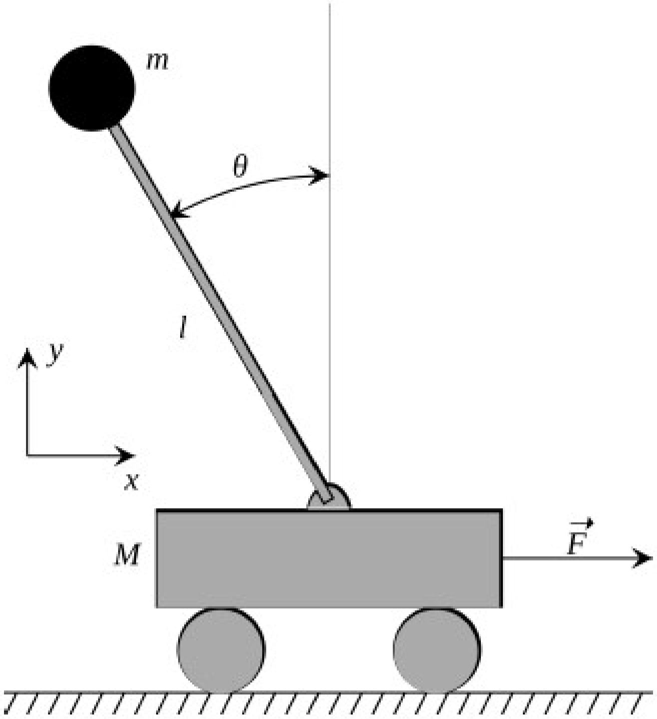


Reinforcement Learning



Cart-Pole Problem

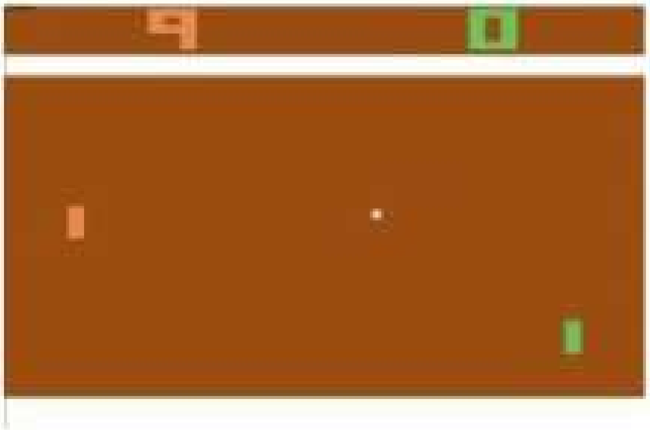
Objective : Balance a pole on top of a movable cart
State **:** angle, angular speed, position, horizontal velocity
Action **:** horizontal force applied on the cart
Reward **:** 1 at each time step if the pole is upright

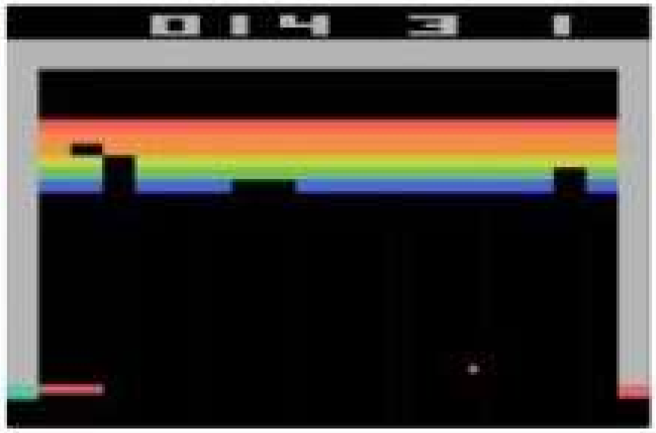


Robot Locomotion

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

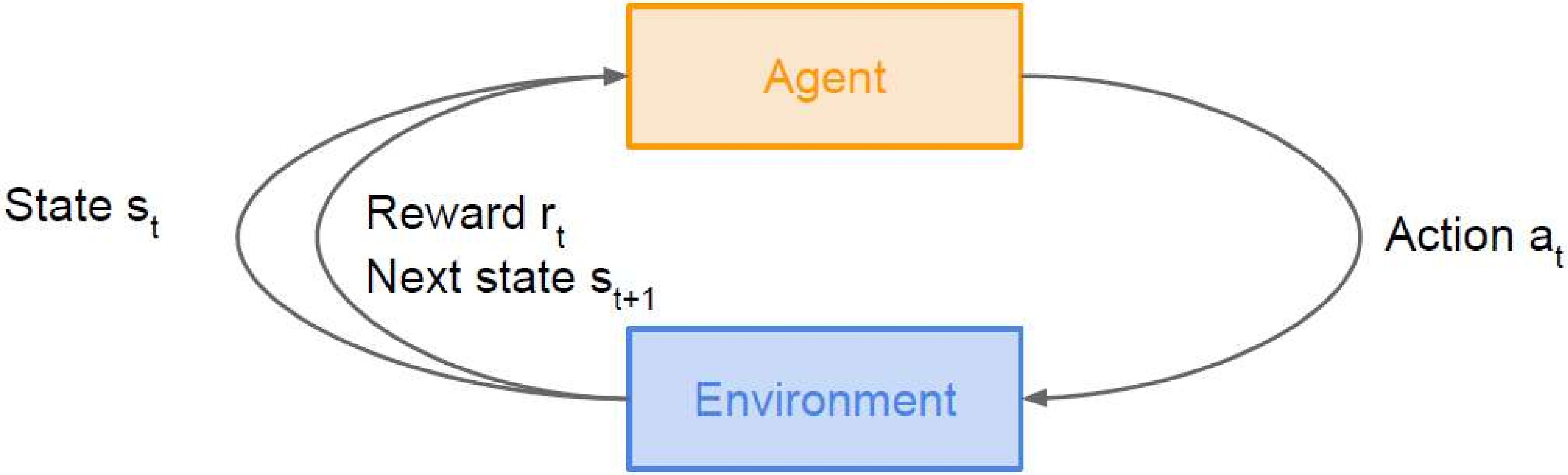
Atari Games





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

RL: Mathematical Formulation



Markov Decision Process

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

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

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

Markov Decision Process

S is a set of a finite state that describes the environment.

A is a set of a finite actions that describes the action that can be taken by the agent

P is a probability matrix that tells the probability of moving from one state to the other.

R is a set of rewards that depend on the state and the action taken. Rewards are not necessarily positive, they should be seen as outcome of an action done by the agent when it is at a certain state. So negative reward indicates bad result, whereas positive reward indicates good result.

γ is a discount factor, that tells how important future rewards are to the current state. Discount factor is a value between 0 and 1. A reward R that occurs N steps in the future from the current state, is multiplied by γ^N to describe its importance to the current state. For example consider $\gamma = 0.9$ and a reward $R = 10$ that is 3 steps ahead of our current state. The importance of this reward to us from where we stand is equal to $(0.9^3)*10 = 7.29$.

Value Functions

Now with the MDP in place, we have a description of the environment but still we don’t know how the agent should act in this environment.

The rule we impose on the agent is that it must act in a way to maximize the collected rewards.

But to be able to do that, the agent should have something to estimate the position or state it is currently in.

Consider again the labyrinth, if the agent is few steps from the exit, his position has much more value than a position at the center of the labyrinth.

We call this is to estimate the Value of the position or the Value of the state.

Once we know the value of each state, we can figure out what is the best way to act (simply by following the state that with the highest value).

State Value Function

Still we need to figure out a way to quantify the value of each state. This is done by a function called Sate-Value Function:

$$v_\pi(s) = \sum_a \pi(a|s) \sum_{s',r} p(s',r|s,a)\Big[r + \gamma v_\pi(s')\Big], \quad \text{for all } s \in \mathcal{S}$$

What is important to learn about this equation is that the value of each state is the average of discounted future rewards.

If we look closely at the equation we see that ***v(s)*** is expressed in terms of immediate rewards ***r*** and the value of neighboring states ***v(s')***.

State Value Function

$$v_\pi(s) = \sum_a \pi(a|s) \sum_{s',r} p(s',r|s,a)\Big[r + \gamma v_\pi(s')\Big], \quad \text{for all } s \in \mathcal{S}$$

1. When at a state ***s***, consider all possible actions, then,

2. for each action ***a*** get the probability ***p(s', r | s, a)*** of getting immediate reward ***r***, and moving to neighboring state ***s'***, knowing that we are at state ***s*** and we performed action ***a***.

3. Add the reward ***r*** to the discounted value of neighbor state ***s'*** given by ***γ*v(s')***. Multiply the result with ***p(s', r | s, a)***, and we get ***p(s', r | s, a)*[r+γ*v(s')]***.

4. Repeat steps 2 and 3 for all states **s'** neighboring ***s,*** as well as all the possible rewards ***r***, and compute the sum of the results. Now we have ***Sum(p(s', r | s, a)*[r+γ*v(s')])*** over all ***s'*** and ***r.***

5. This will give the average of discounted future rewards when taking only one action ***a***. However there are multiple actions to be taken so we have to average over all actions. To do so we multiply by probability ***π(a|s)*** that action ***a*** be performed at state ***s***, and we sum over all possible actions in that state.

Action Value Function

•It suffices to think that when we are at a state we have a probability to take some action that might lead us to different states with different rewards.

•So the value of our state is the average of all discounted rewards that we might get when performing all of the possible actions at the current state.

•Now we have a function that gives us the value of each state. It tells us how good is to be at each one of them.

•Of course we still have to make the computation in order to get a number that represents the value of that state.

•***v(s)*** tells how good to be in state ***s***, but it does not tell how good to perform action ***a*** while in state ***s***. This is the purpose of **the Action-Value function**:

$$q_\pi(s,a) = \sum_{s',r} p(s',r|s,a)\Big[r + \gamma v_\pi(s')\Big]$$

Action Value Function

$$q_\pi(s,a) \;=\; \sum_{s',r} p(s',r|s,a)\Big[r+\gamma v_\pi(s')\Big]$$

The explication is simple, we take ***v(s)*** and instead of performing all actions, we decide to perform only one action.

We no more ask ourselves what is the probability of using action ***a*** over all possible actions, but we say we will use action ***a***.

So the sum of ***π(a|s)*** does not apply anymore.

With this logic we reduce ***v(s)*** from averaging over all possible actions to simply using one selected action, we denote this as ***q(s,a)***.

Action Value Function

Notice that ***q(s,a)*** checks the value of the action at state ***s*** while the values ***v(s')*** of neighboring states are kept unchanged. So in this sense it checks how good this action is in the current state while keeping everything else the same.

It is also easy to notice that ***v(s)*** is the weighted average of ***q(s,a)*** over all possible actions at state ***s***

$$v_\pi(s) = \sum_a \pi(a|s)\, q_\pi(s,a)$$

Policy

The act of selecting an action at each state is called “policy” and is denoted as π.

- some policies are better than others due to the selection of actions over others in one or more states.
- It is important to note that a policy π is better than π' if all ***v(s)*** under π are greater or equal to all ***v(s)*** under π'.
- It follows that in order to **maximize the collected rewards** we have to find the best possible policy, called optimal policy and denoted π^*.

Optimal Value Functions and Policy

An optimal value state function ***v*(s)*** is a function that gives the maximum value at each state among all policies:

$v_*(s) = \max_\pi v_\pi(s), \quad \text{for all } s \in \mathcal{S}$ where

$$v_\pi(s) = \sum_a \pi(a|s) \sum_{s',r} p(s',r|s,a)\Big[r + \gamma v_\pi(s')\Big], \quad \text{for all } s \in \mathcal{S}$$

Similarly an optimal action state function ***q*(s)*** is the function that gives the maximum ***q*** value at each state among all policies:

$$q_*(s,a) = \max_\pi q_\pi(s,a), \quad \text{for all } s \in \mathcal{S} \text{ and } a \in \mathcal{A}(s)$$

it follows that

$v_*(s) = \max_{a \in \mathcal{A}(s)} q_{\pi_*}(s,a)$ where $q_\pi(s,a) = \sum_{s',r} p(s',r|s,a)\Big[r + \gamma v_\pi(s')\Big]$

$$v_*(s) = \max_a \sum_{s',r} p(s',r|s,a)\big[r + \gamma v_*(s')\big]$$

Optimal Policy

Notice that ***v(s)*** is the average of values produced by all actions, while ***v*(s)*** is the maximum value that one action can produce among all other actions.

$$q_*(s,a) = \sum_{s',r} p(s',r|s,a)\left[r + \gamma \max_{a'} q_*(s',a')\right]$$

Since ***q(s,a)*** is the value produced by a specific action ***a*** at state ***s***, ***q*(s,a)*** is the value produced by a specific action ***a*** at state ***s***, while selecting maximum ***q*** values in the other states.

Earlier we said that a policy π is better than another policy π’ if it produces higher or equal ***v(s)*** value at each state:

$$\pi \geq \pi' \text{ if } v_\pi(s) \geq v_{\pi'}(s), \forall s$$

Optimal Policy

It follows that the optimal policy π* produces, at each state, higher or equal ***v(s)*** values than any other policy π .

$$\pi_* \geq \pi \ \text{ if } v_{\pi_*}(s) \geq v_{\pi}\ (s), \forall s\ \forall \pi$$

This means that any ***v(s)*** and ***q(s,a)*** following the optimal policy π* are equal to ***v*(s)*** and ***q*(s,a)*** respectively.

All optimal policies achieve the optimal value function,

$v_{\pi_*}(s) = v_*(s)$

All optimal policies achieve the optimal action-value function,

$q_{\pi_*}(s, a) = q_*(s, a)$

How to find the optimal policy π* ?

So the question that remains is, how to find the optimal policy π* ?
The optimal policy should always return the action that produces ***q*(s,a)***

$$\pi_*(a|s) = \begin{cases} 1 & \text{if } a = \underset{a \in \mathcal{A}}{\operatorname{argmax}}\, q_*(s,a) \\ 0 & otherwise \end{cases}$$

- There is always a deterministic optimal policy for any MDP

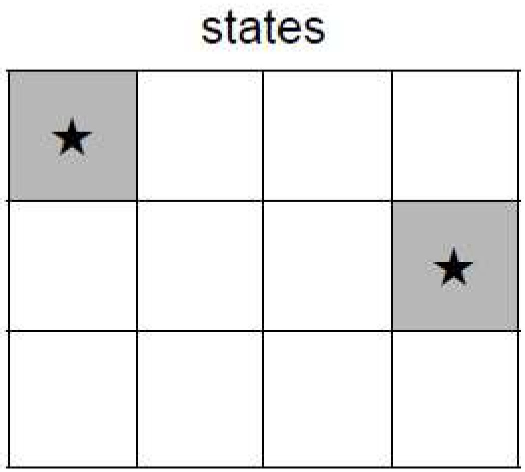
- If we know ***q*(s,a)*** it will be very easy to find π*.

MDP Example: Grid World

actions = {

1. right →
2. left ←
3. up ↑
4. down ↓

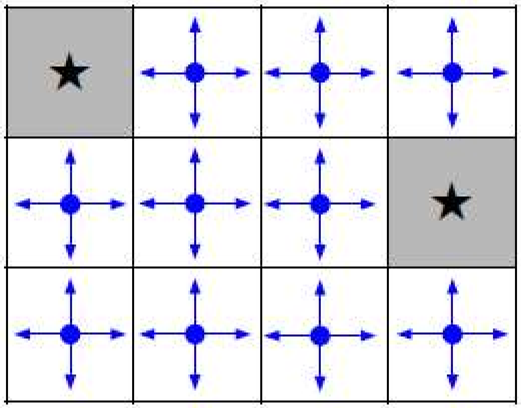
}



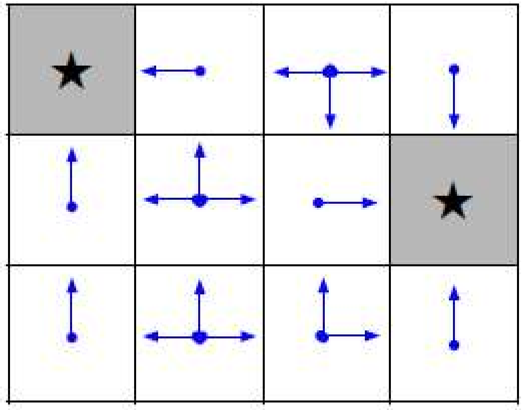
Set a negative "reward" for each transition (e.g. $r = -1$)

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

MDP Example: Grid World



Random Policy



Optimal Policy

Optimal Policy

Note that the optimal policy is not unique, and this can be easily verified.

Suppose an agent in the Labyrinth has arrived to a state with 3 possible moves (or actions), go left, go right, or go forward. Suppose going forward leads the agent to a dead end, while going left or right will lead the agent to the exit using the same number of steps.

It is clear that there are two optimal policies (not only one) at this state, one that instructs the agent to go left and the other instructs it to go right.

Markov Decision Process

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

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

The optimal policy π^*

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

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

Maximize the **expected sum of rewards!**

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

As we see earlier

$$\pi_*(a|s) = \begin{cases} 1 & \text{if } a = \underset{a \in \mathcal{A}}{\text{argmax}}\ q_*(s,a) \\ 0 & otherwise \end{cases}$$

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, …

How good is a state?

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

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

As we see earlier

$$v_\pi(s) = \sum_a \pi(a|s) \sum_{s',r} p(s',r|s,a)\Big[r + \gamma v_\pi(s')\Big], \quad \text{for all } s \in \mathcal{S}$$

How good is a state-action pair?

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

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

As we see earlier

$$q_\pi(s,a) \;=\; \sum_{s',r} p(s',r|s,a)\Big[r + \gamma v_\pi(s')\Big]$$

Bellman equation

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

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

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

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

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

As we see earlier

$$q_*(s,a) \;=\; \sum_{s',r} p(s',r|s,a)\Big[r + \gamma \max_{a'} q_*(s',a')\Big]$$

The optimal policy **π*** 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 -> infinity

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

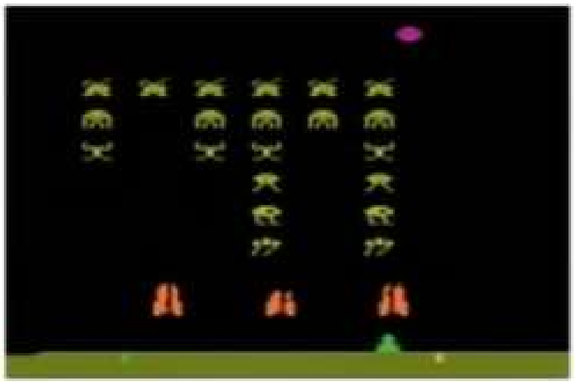
Q-learning: Use a function approximator to estimate the action-value function

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

function parameters (weights)

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

Example: Playing Atari Games



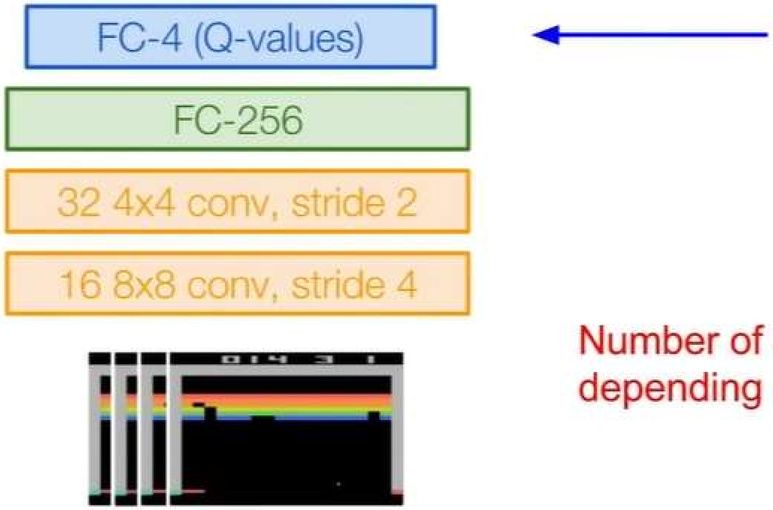
Objective: Complete the game with the highest score

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

Q Network Architecture: Atari games

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

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



FC-4 (Q-values)

FC-256

32 4x4 conv, stride 2

16 8x8 conv, stride 4

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)

Solving for the optimal policy: Q-learning

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

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

Forward Pass

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

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

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

Backward Pass

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

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

Training the Q-network: Experience Replay

Learning from batches of consecutive samples is problematic:

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

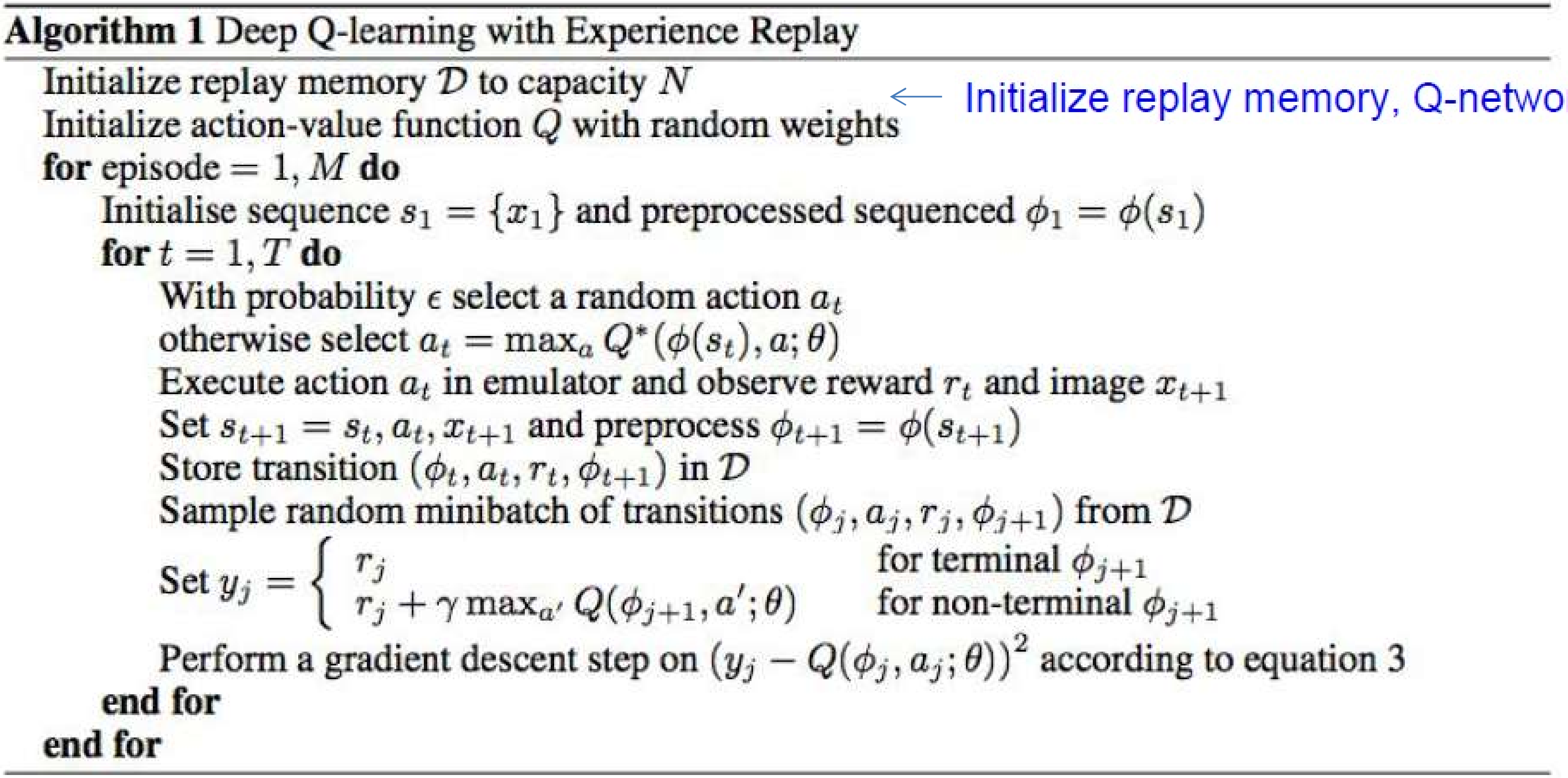
Address these problems using **experience replay**

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

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

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

Deep Q-Learning with Experience Replay



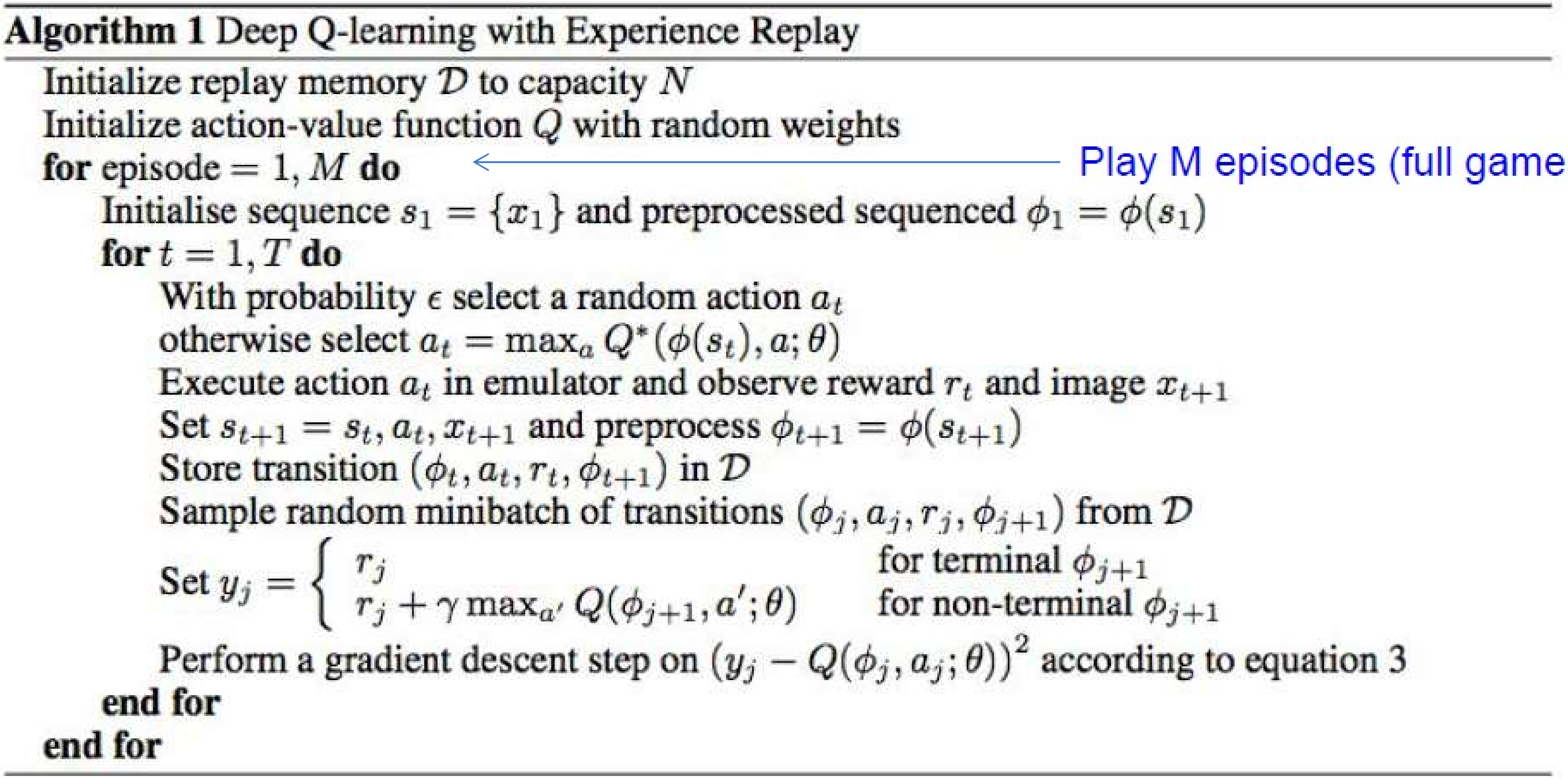
Algorithm 1 Deep Q-learning with Experience Replay

```
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
```

← Initialize replay memory, Q-network

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

Deep Q-Learning with Experience Replay



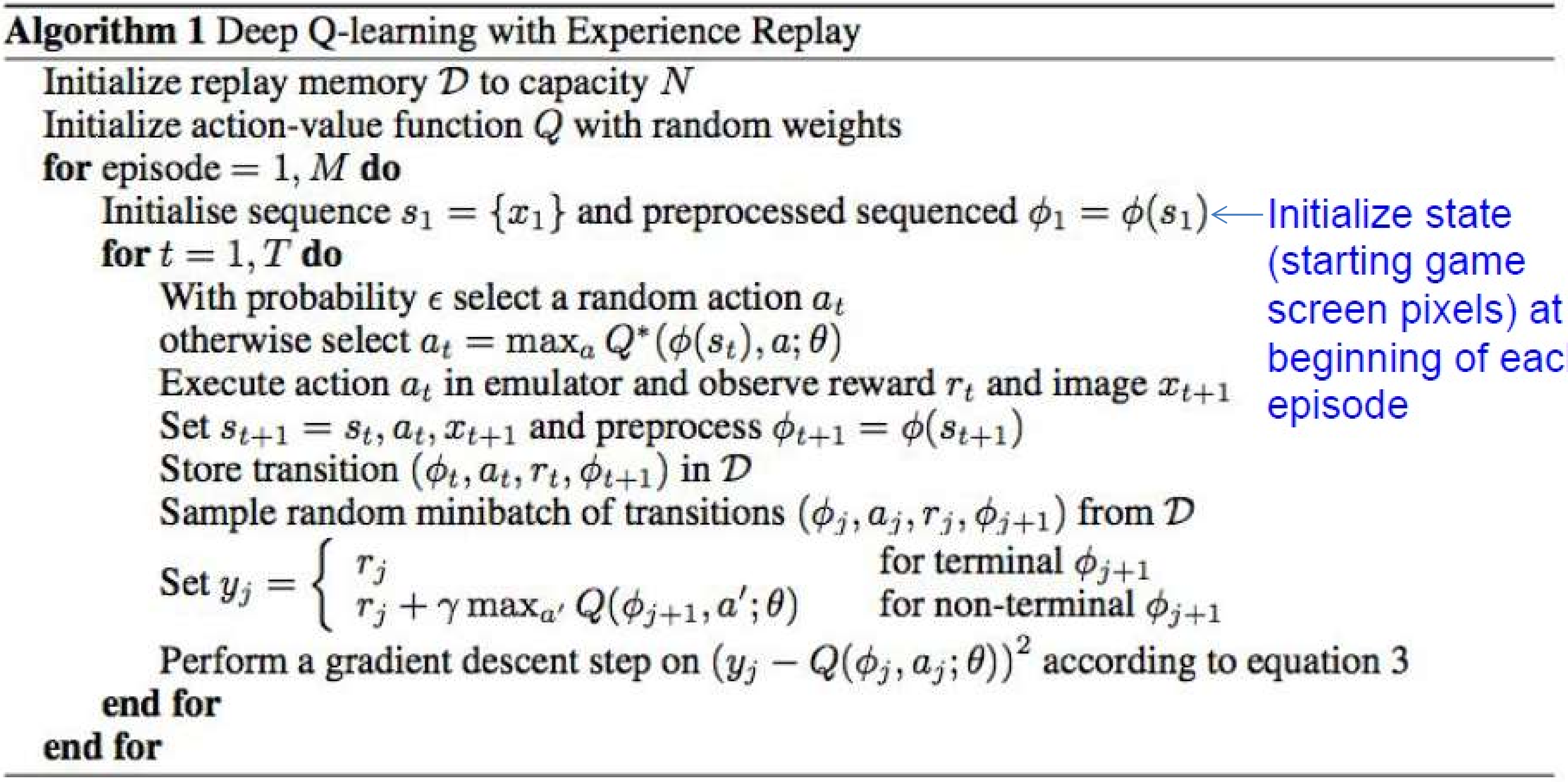
Algorithm 1 Deep Q-learning with Experience Replay

```
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
```

Play M episodes (full games)

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

Deep Q-Learning with Experience Replay



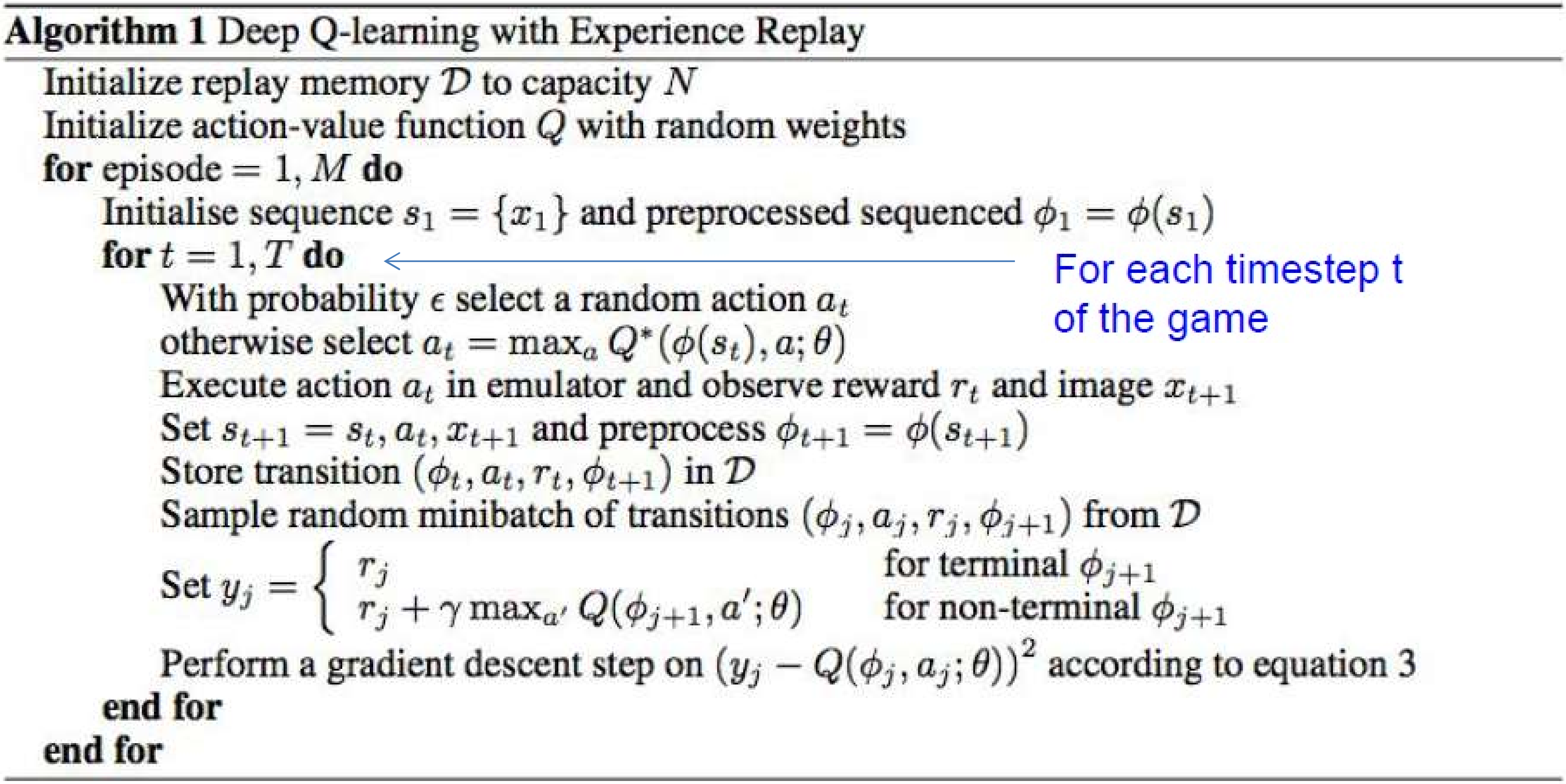
Algorithm 1 Deep Q-learning with Experience Replay

```
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
```

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

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

Deep Q-Learning with Experience Replay



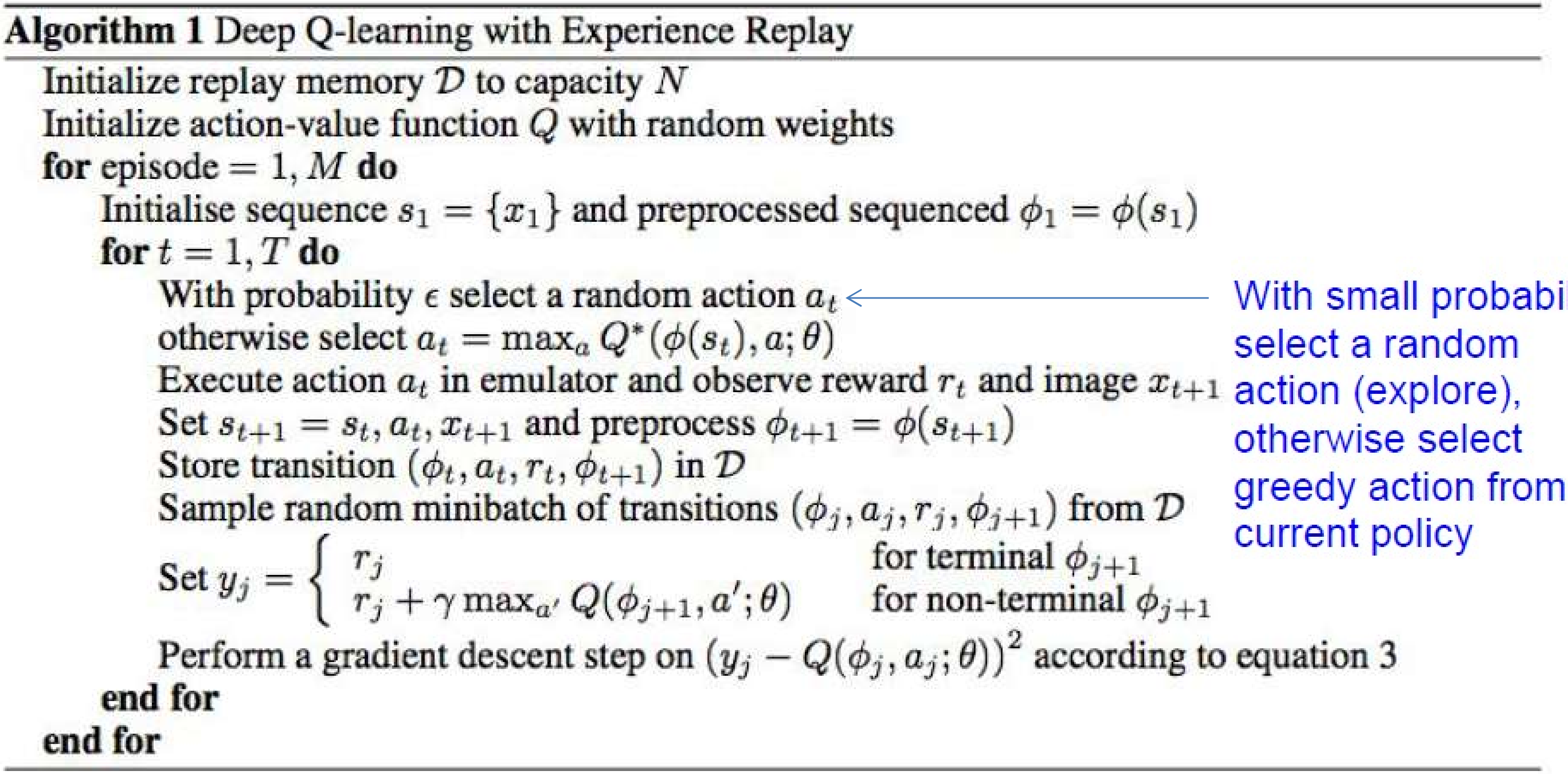
Algorithm 1 Deep Q-learning with Experience Replay

```
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**
```

For each timestep t of the game

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

Deep Q-Learning with Experience Replay



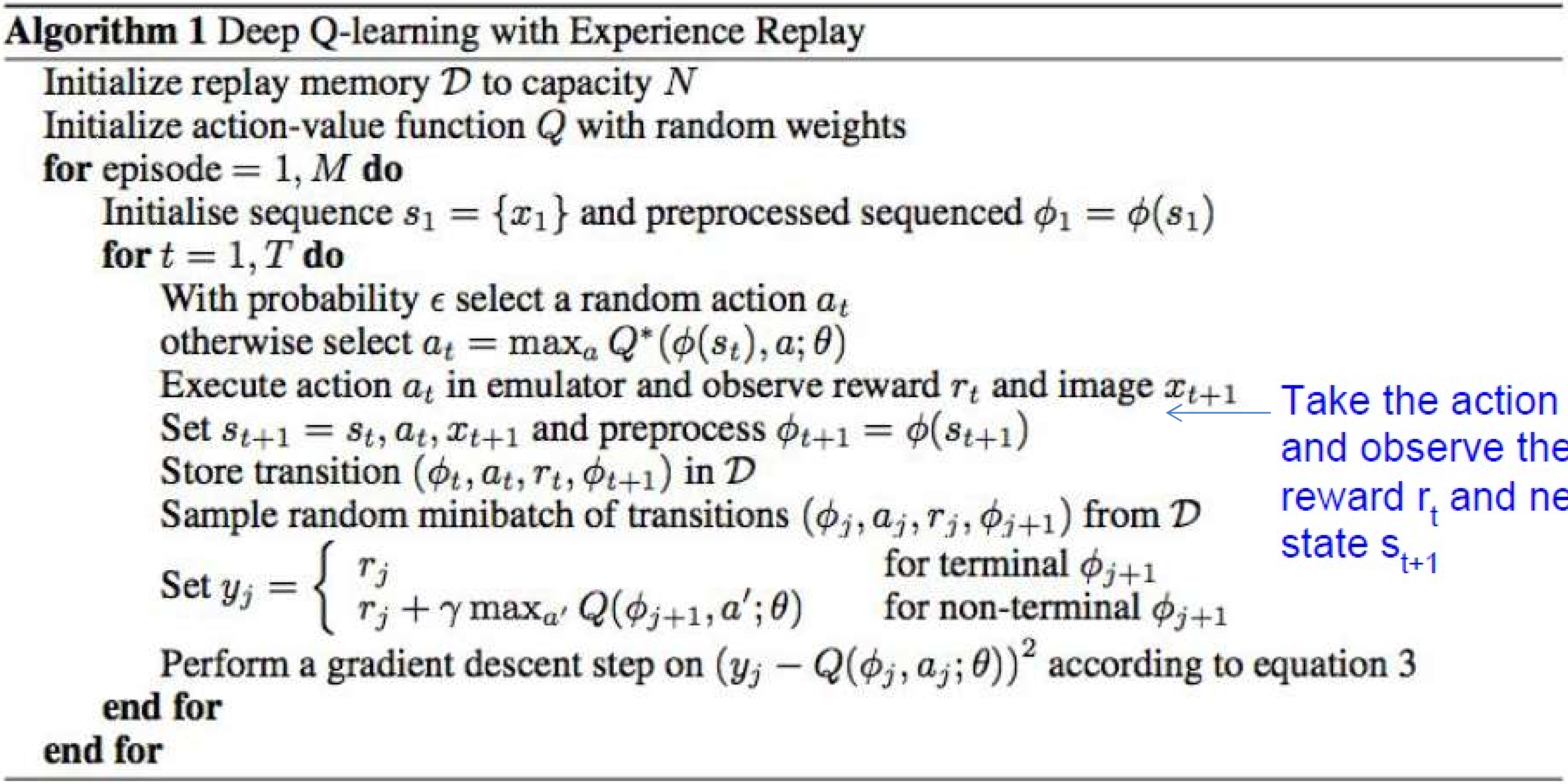
Algorithm 1 Deep Q-learning with Experience Replay

```
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
```

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

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

Deep Q-Learning with Experience Replay



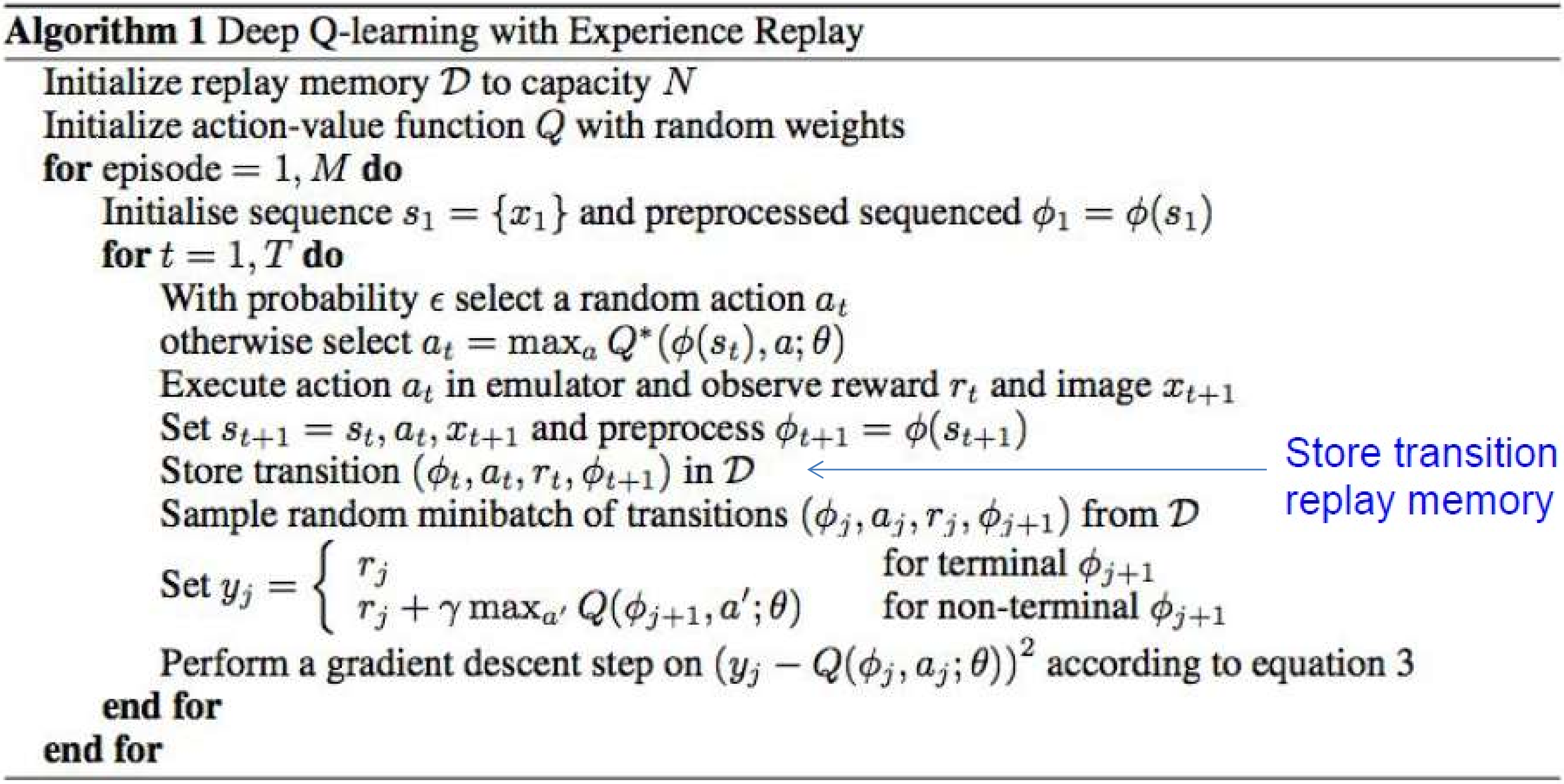
Algorithm 1 Deep Q-learning with Experience Replay

```
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
```

Take the action (a_t), and observe the reward r_t and next state s_{t+1}

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

Deep Q-Learning with Experience Replay

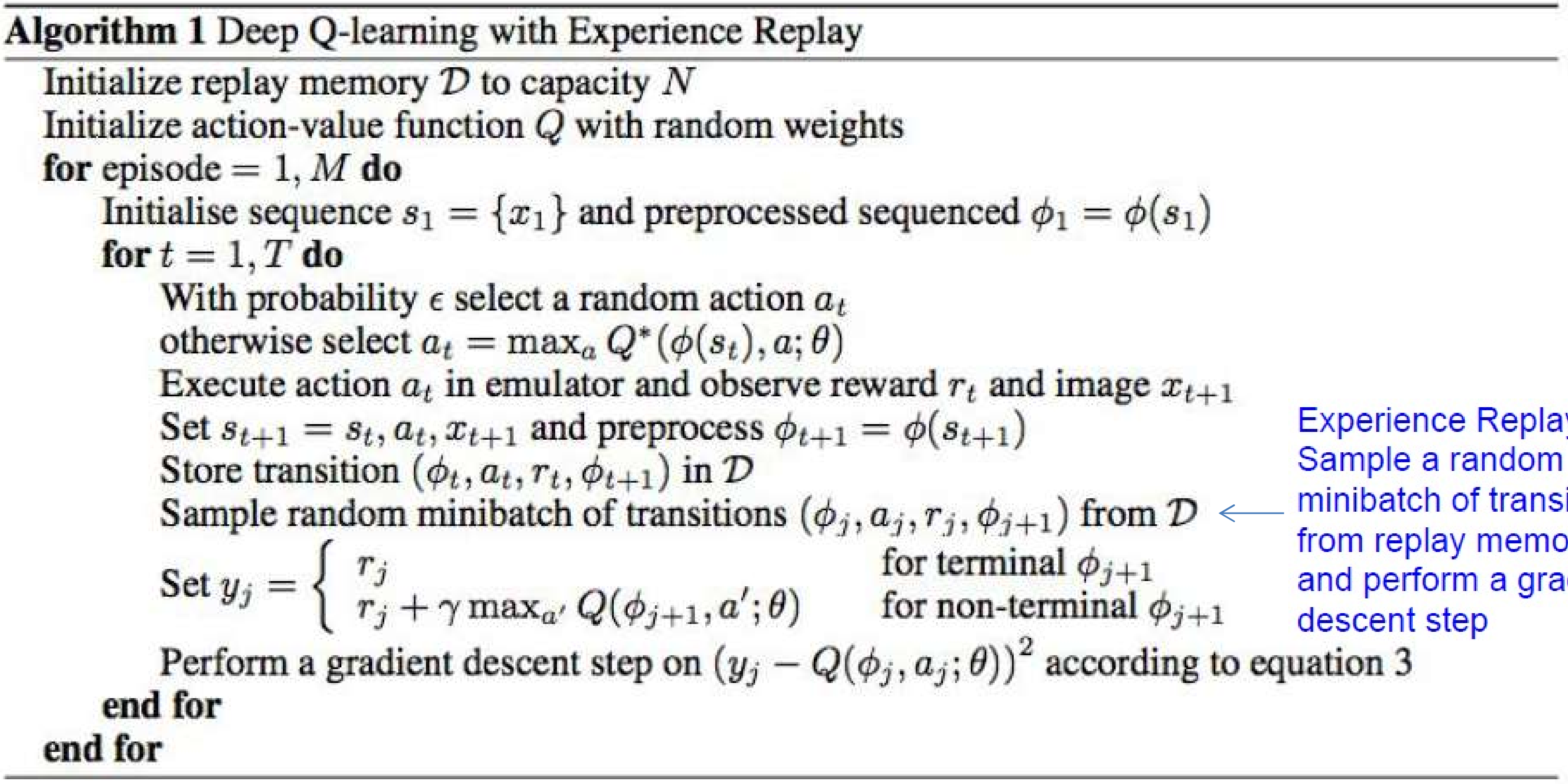


```
Algorithm 1 Deep Q-learning with Experience Replay
  Initialize replay memory $\mathcal{D}$ to capacity $N$
  Initialize action-value function $Q$ with random weights
  for episode $= 1, M$ do
      Initialise sequence $s_1 = \{x_1\}$ and preprocessed sequenced $\phi_1 = \phi(s_1)$
      for $t = 1, T$ 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
```

Store transition in replay memory

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

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
```

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

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

Policy Gradients

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

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

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

Policy Gradients

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

For each policy, define its value:

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

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

How can we do this?

Gradient ascent on policy parameters!

Gradient *ascent*

Gradient *descent* aims at *minimizing* some objective function

$$\theta_j \leftarrow \theta_j - \alpha \frac{\partial}{\partial_{\theta_j}} J(\Theta)$$

Gradient *ascent* aims at *maximizing* some objective function

$$\theta_j \leftarrow \theta_j + \alpha \frac{\partial}{\partial_{\theta_j}} J(\Theta)$$

REINFORCE Algorithm

Mathematically, we can write:

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

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

REINFORCE Algorithm

Expected reward:

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

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

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

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:

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

Can estimate with Monte Carlo sampling

REINFORCE Algorithm

Can we compute those quantities without knowing the transition probabilities?

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

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

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

Doesn't depend on transition probabilities!

Therefore when sampling a trajectory τ, we can estimate J(θ) with

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

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

Intuition

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

Interpretation:

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

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

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

Variance Reduction

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

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

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

Second idea: Use discount factor γ to ignore delayed effects

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

Variance Reduction: Baseline

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

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

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

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

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?

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

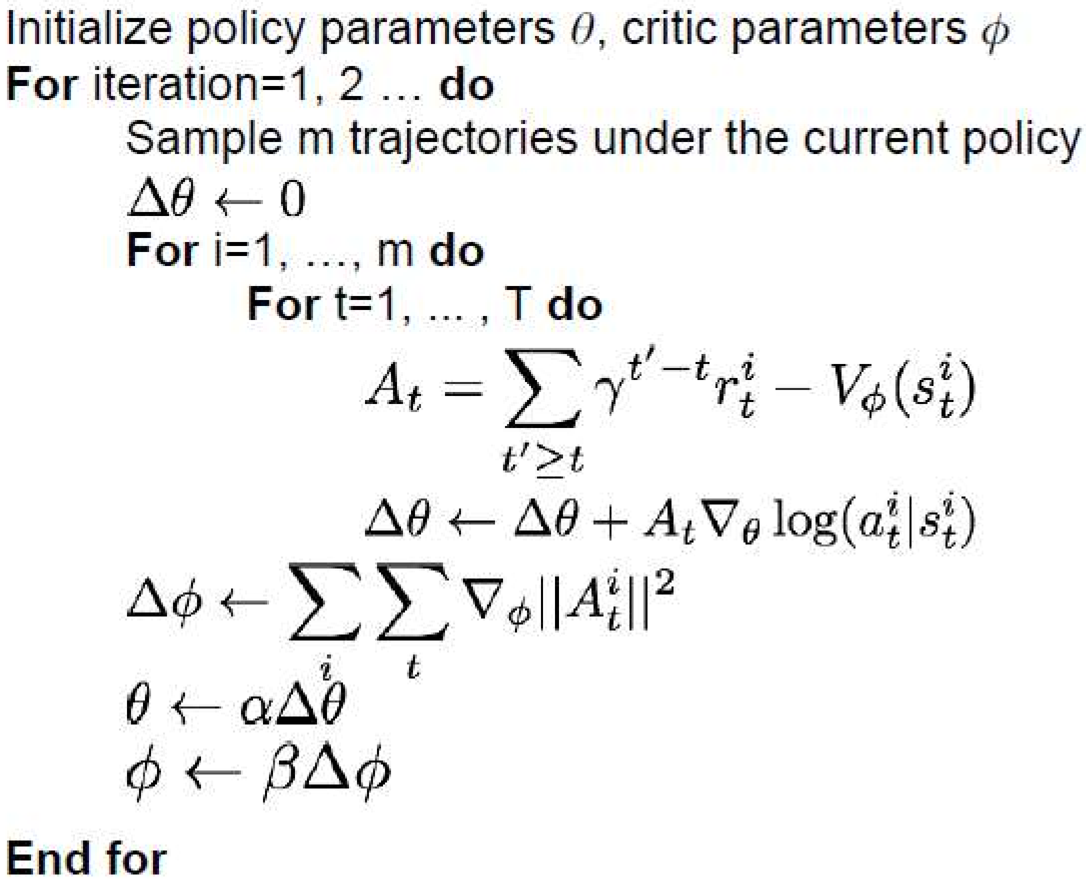
Problem: we don’t know Q and V. Can we learn them?

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, …, m do
        For t=1, ... , T do
            $A_t = \sum_{t' \geq t} \gamma^{t'-t} r_t^i - V_\phi(s_t^i)$
            $\Delta\theta \leftarrow \Delta\theta + A_t \nabla_\theta \log(a_t^i | s_t^i)$
    $\Delta\phi \leftarrow \sum_i \sum_t \nabla_\phi ||A_t^i||^2$
    $\theta \leftarrow \alpha\Delta\theta$
    $\phi \leftarrow \beta\Delta\phi$

End for
```

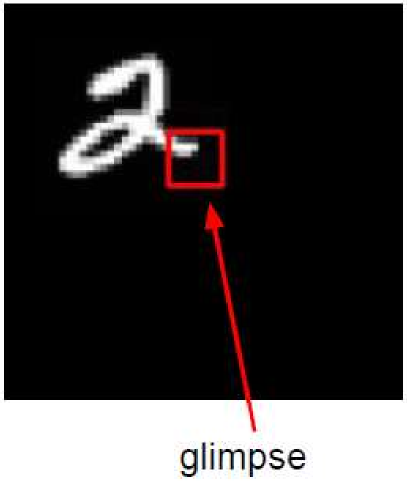
Recurrent Action Model (RAM)

Objective: Image Classification

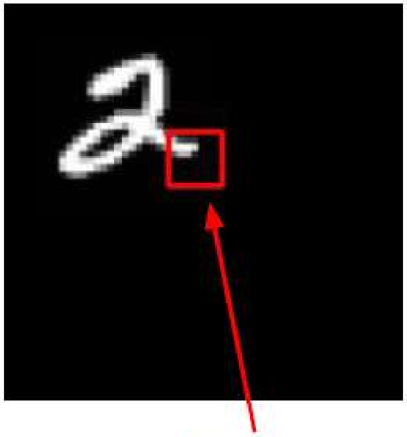
Take a sequence of “glimpses” selectively focusing on regions of the image, to predict class

- 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



Recurrent Action Model (RAM)

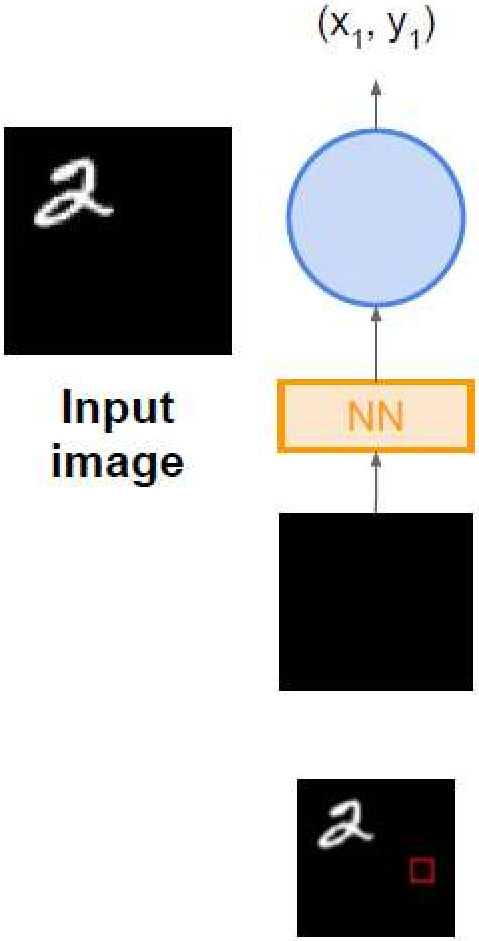


glimpse

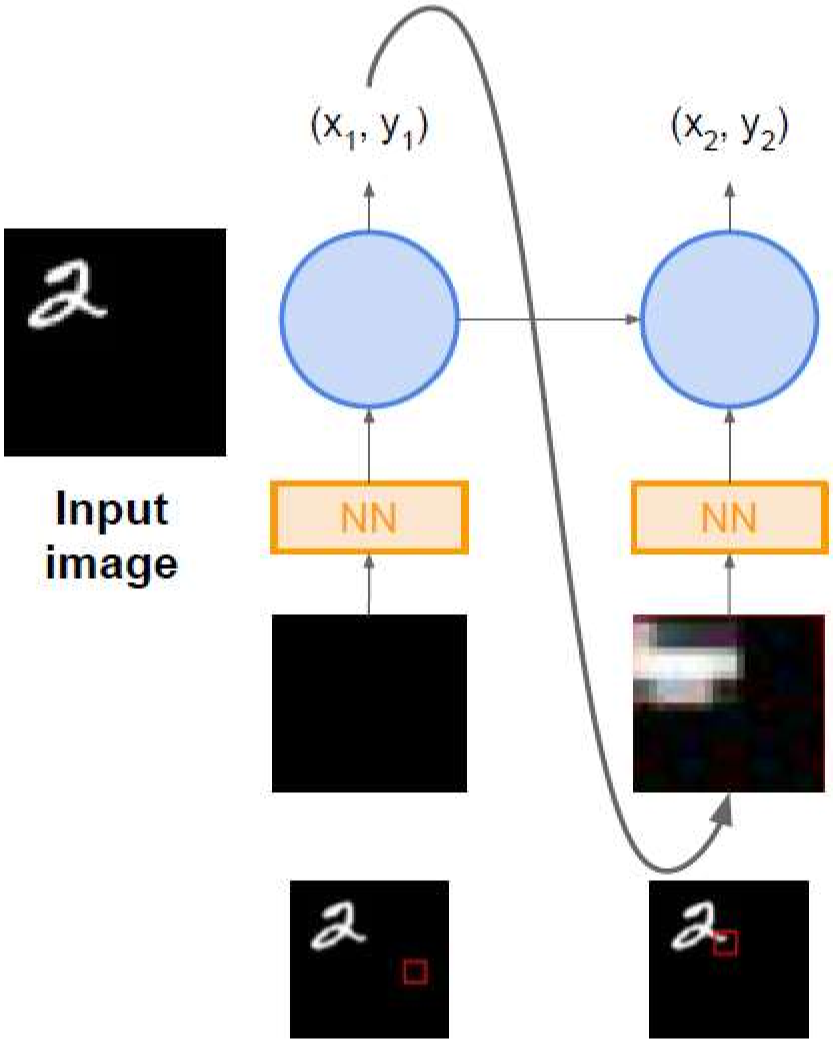
Glimpsing is a non-differentiable operation => learn policy for how to take glimpse actions using REINFORCE
Given state of glimpses seen so far, use RNN to model the state and output next action

[Mnih et al. 2014]

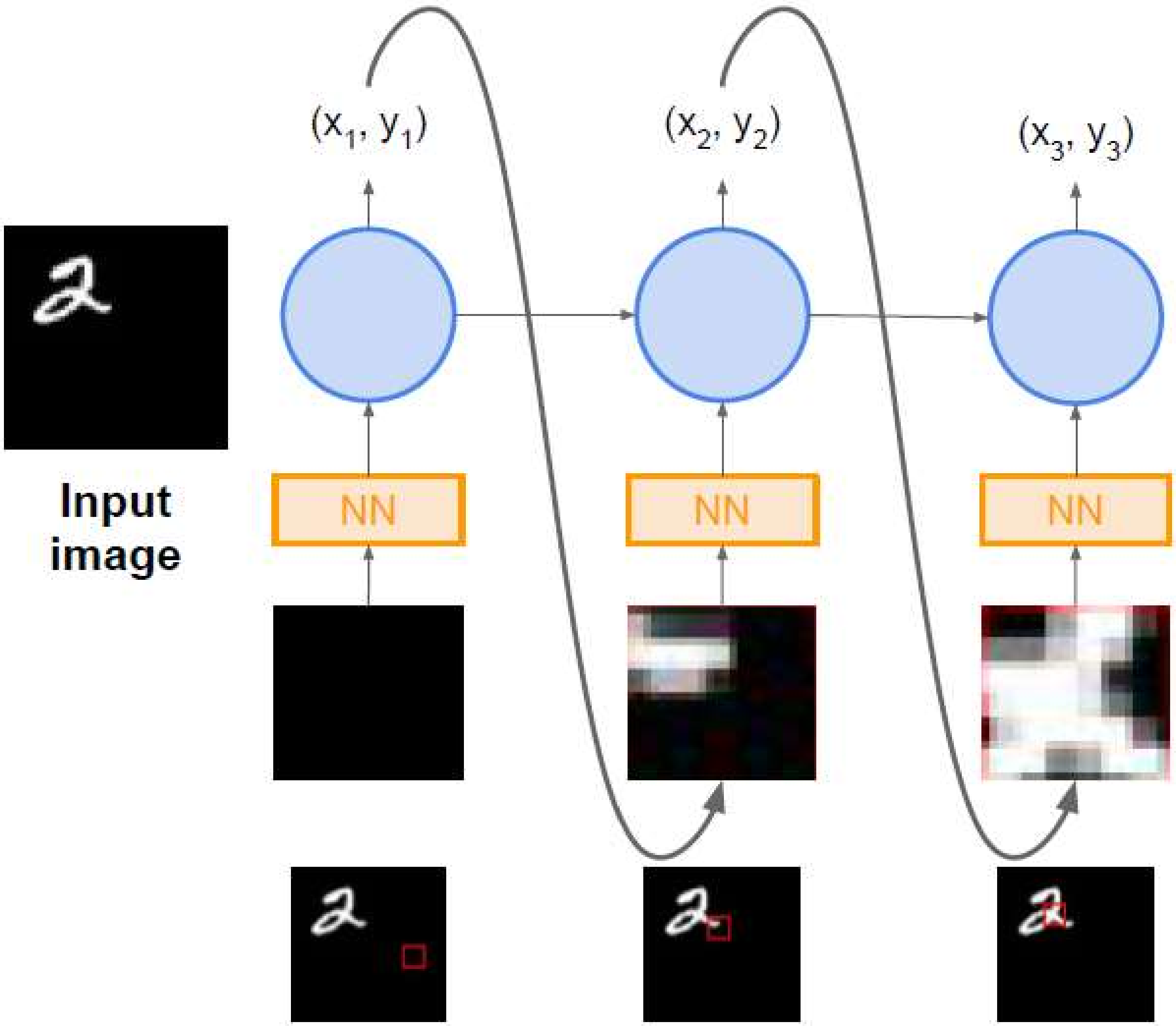
Recurrent Action Model (RAM)



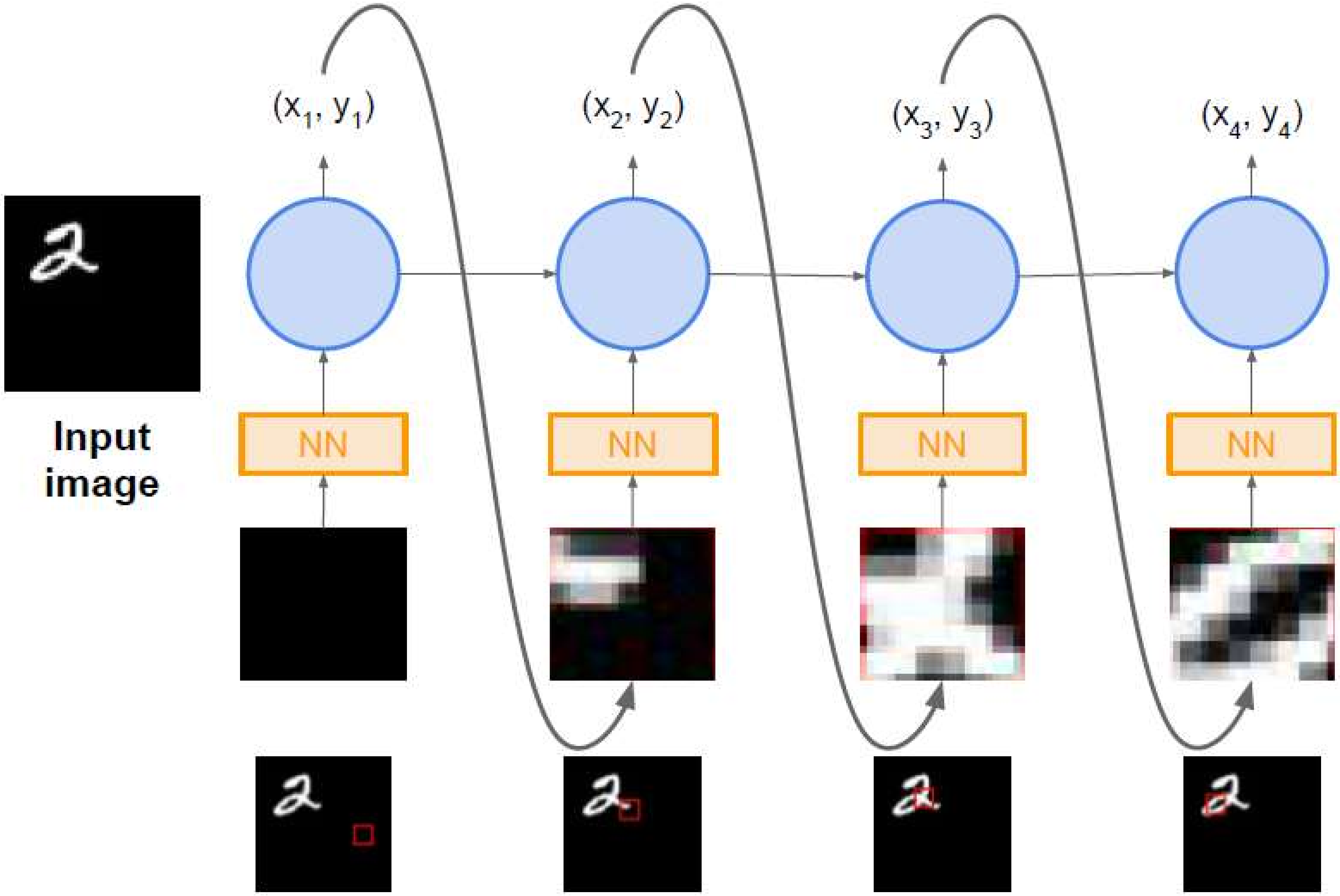
Recurrent Action Model (RAM)



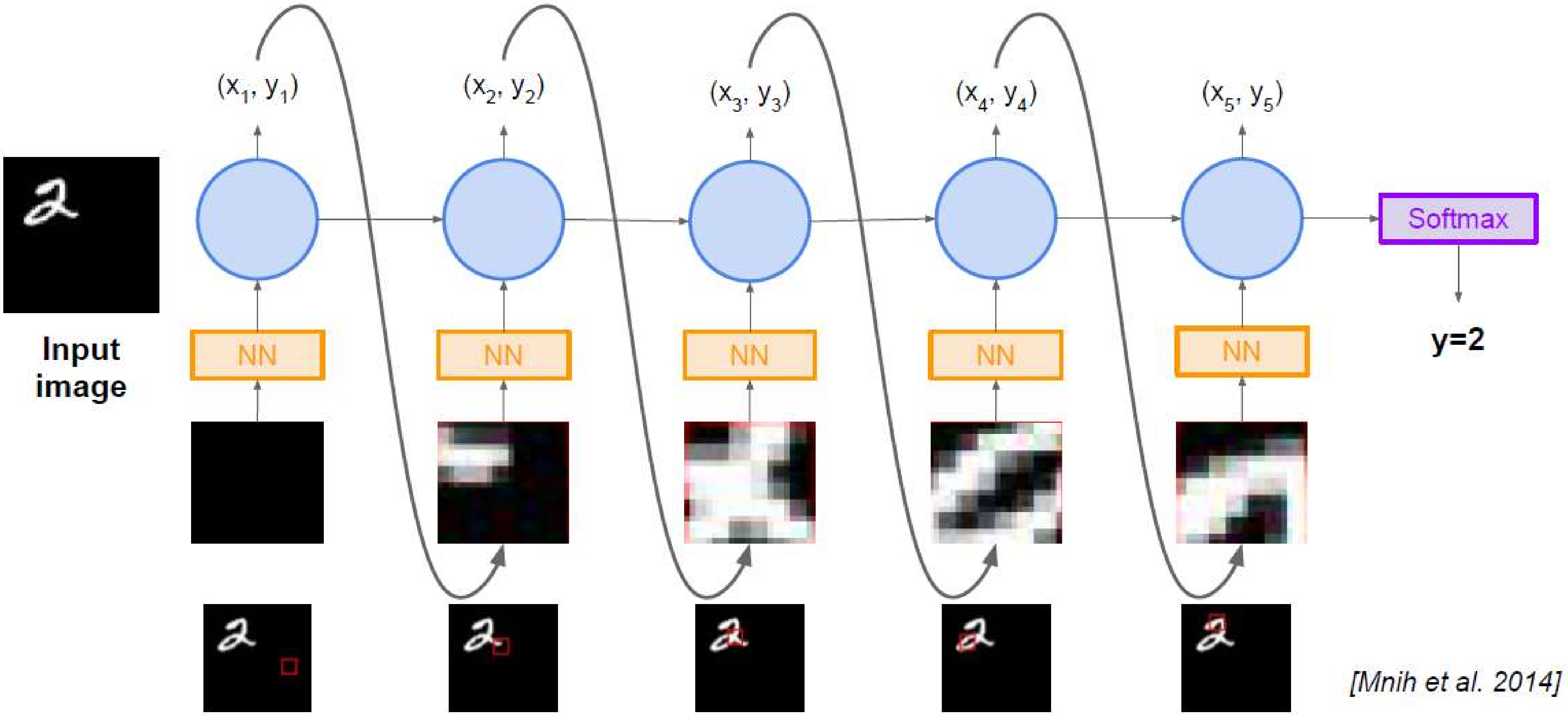
Recurrent Action Model (RAM)



Recurrent Action Model (RAM)

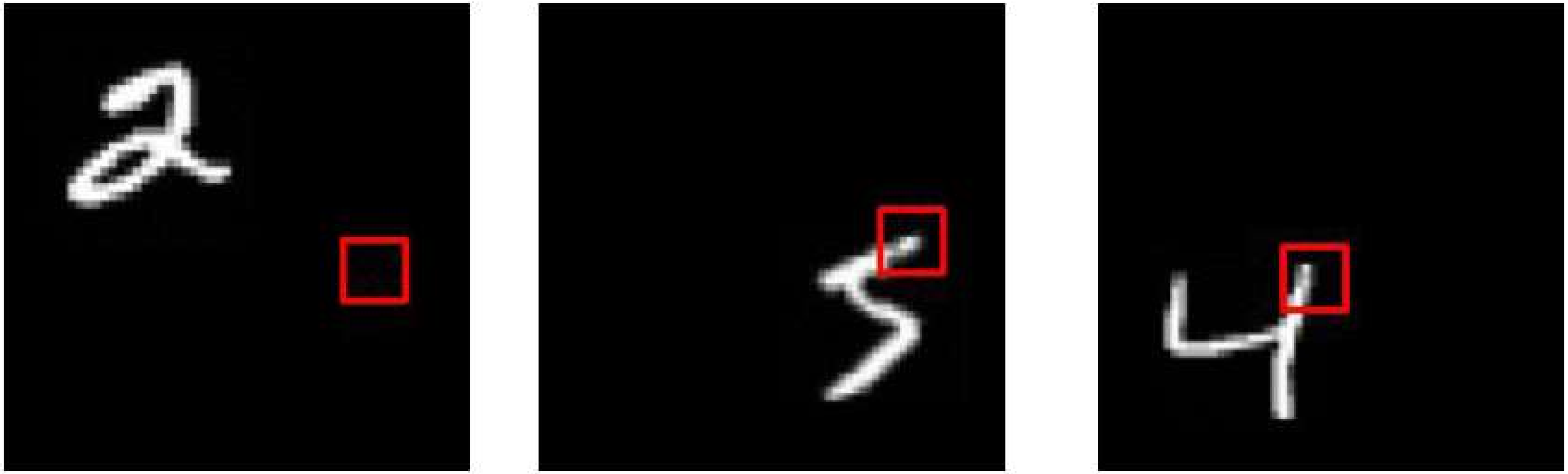


Recurrent Action Model (RAM)



[Mnih et al. 2014]

Recurrent Action 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]

to continue...