M2177.0043 Introduction to Deep Learning

Lecture 20: Reinforcement learning[1]

Hyun Oh Song[1]

[1] Dept. of Computer Science and Engineering, Seoul National University

May 28, 2020

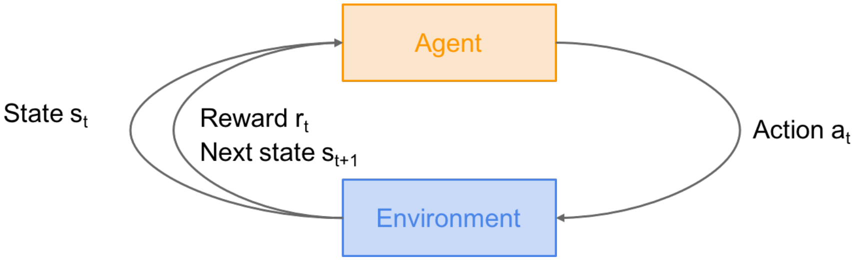
[1] Many slides and figures adapted David Silver and Justin Johnson

Last time

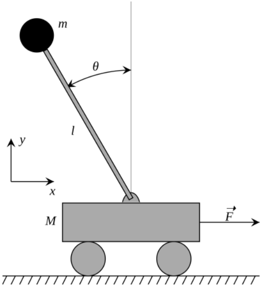
- Metric learning
- Efficient retrieval

Outline

Reinforcement learning



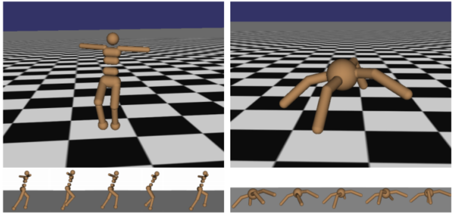
Cart-Pole problem



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

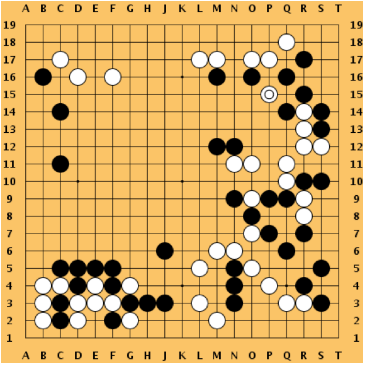
State: Angle and position of the joints
Action: Torques applied on joints
Reward: 1 at each time step upright + forward movement

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

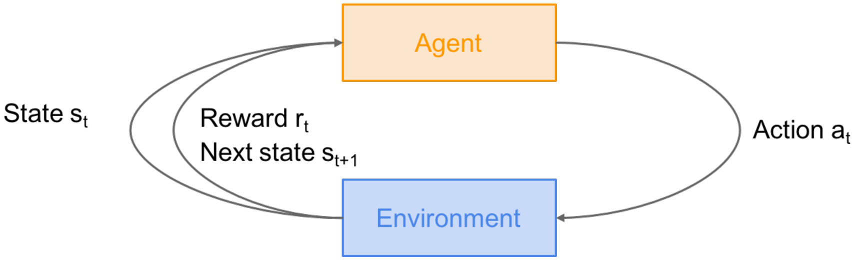
Go



Objective: Win the game!

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

How can we mathematically formalize the RL problem?



Markov Property

"The future is independent of the past given the present"

Definition

A state S_t is Markov if and only if

$$\mathbb{P}[S_{t+1} \mid S_t] = \mathbb{P}[S_{t+1} \mid S_1, \ldots, S_t]$$

- The state captures all relevant information from the history
- Once the state is known, the history may be thrown away
- *i.e.* The state is a sufficient statistic of the future

Return

Definition

The return G_t is the total discounted reward from time-step t.

$$G_t = R_{t+1} + \gamma R_{t+2} + \ldots = \sum_{k=0}^{\infty} \gamma^k R_{t+k+1}$$

- The discount $\gamma \in (0, 1]$ is the present value of the future reward
- The value of receiving reward R after $k+1$ time-steps in $\gamma^k R$
- This values immediate reward above delayed reward. γ close to 0 leads to myopic evaluation while γ close to 1 leads to far-sighted evaluation.

Markov Decision Process

Definition

A Markov decision process (MDP) is a tuple $(\mathcal{S}, \mathcal{A}, \mathcal{P}, \mathcal{R}, \gamma)$

- $\mathcal{S}$ is a finite set of states
- $\mathcal{A}$ is a finite set of actions
- $\mathcal{P}$ is a state transition probability matrix,
  $\mathcal{P}^a_{ss'} = \mathbb{P}\left[S_{t+1} = s' \mid S_t = s, A_t = a\right]$
- $\mathcal{R}$ is a reward function, $R^a_s = \mathbb{E}\left[R_{t+1} \mid S_t = s, A_t = a\right]$
- γ is a discount factor $\gamma \in [0, 1]$.

Policies

Definition

A policy π is a distribution over actions given states

$$\pi(a \mid s) = \mathbb{P}[A_t = a \mid S_t = s]$$

- A policy fully defines the behavior of an agent
- MDP policies depend on the current state (not the history)
- *i.e.* Policies are stationary (time-independent), $A_t \sim \pi(\cdot \mid S_t), \forall t > 0$

Value Function

Definition

The *state-value function* $v_\pi(s)$ of an MDP is the expected return starting from state s, and then following policy π

$$v_\pi(s) = \mathbb{E}_\pi[G_t \mid S_t = s]$$

Definition

The *action-value function* $q_\pi(s, a)$ is the expected return starting from state s, taking action a, and then following policy π

$$q_\pi(s, a) = \mathbb{E}_\pi[G_t \mid S_t = s, A_t = a]$$

Optimal Policy

Define a partial ordering over policies

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

Theorem

For any Markov Decision Process

- There exists an optimal policy π_* that is better than or equal to all other policies, $\pi_* \geqslant \pi, \forall \pi$
- All optimal policies achieve the optimal value function, $v_{\pi_*} = v_*(s)$
- All optimal policies achieve the optimal action-value function, $q_{\pi_*}(s, a) = q_*(s, a)$

Finding an Optimal Policy

An optimal policy can be found by maximizing over $q_*(s, a)$,

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

- In other words, greedy is optimal for MDP
- There is always a deterministic optimal policy for any MDP
- If we know $q_*(s, a)$, we immediately have the optimal policy

Bellman optimality equation

The optimal value functions are recursively related by the Bellman optimality equations:

$$
\begin{aligned}
v_*(s) &= \max_a q_*(s,a) \\
q_*(s,a) &= \mathcal{R}_s^a + \gamma \sum_{s' \in \mathcal{S}} \mathcal{P}_{ss'}^a v_*(s') \\
v_*(s) &= \max_a \mathcal{R}_s^a + \gamma \sum_{s' \in \mathcal{S}} \mathcal{P}_{ss'}^a v_*(s') \\
q_*(s,a) &= \mathcal{R}_s^a + \gamma \sum_{s' \in \mathcal{S}} \mathcal{P}_{ss'}^a \max_{a'} q_*(s',a')
\end{aligned}
$$

No closed form solution in general. Many iterative solution methods (*i.e.* Value iteration, Policy iteration, Q-learning, Sarsa, *etc.*) exist.

Value iteration

- If we know the solution to subproblems $v_*(s')$
- Then solution $v_*(s)$ can be found by one-step lookahead

$$v_*(s) \leftarrow \max_{a \in \mathcal{A}} \mathcal{R}_s^a + \gamma \sum_{s' \in \mathcal{S}} \mathcal{P}_{ss'}^a v_*(s')$$

$$q_*(s,a) \leftarrow \mathcal{R}_s^a + \gamma \sum_{s' \in \mathcal{S}} \mathcal{P}_{ss'}^a \max_{a'} q_*(s',a')$$

- The idea of value iteration is to apply these updates iteratively
- Intuition: start with final rewards and work backwards

- Value iteration pseudocode[2]

Value Iteration, for estimating $\pi \approx \pi_*$

Algorithm parameter: a small threshold $\theta > 0$ determining accuracy of estimation
Initialize $V(s)$, for all $s \in \mathcal{S}^+$, arbitrarily except that $V(terminal) = 0$

Loop:
| $\Delta \leftarrow 0$
| Loop for each $s \in \mathcal{S}$:
| $\quad v \leftarrow V(s)$
| $\quad V(s) \leftarrow \max_a \sum_{s',r} p(s', r \mid s, a)\big[r + \gamma V(s')\big]$
| $\quad \Delta \leftarrow \max(\Delta, |v - V(s)|)$
until $\Delta < \theta$

Output a deterministic policy, $\pi \approx \pi_*$, such that
$\pi(s) = \arg\max_a \sum_{s',r} p(s', r \mid s, a)\big[r + \gamma V(s')\big]$

[2] Taken from Sutton & Barto, Reinforcement learning, 2nd edition

- For actual examples, refer to Example 4.3 Gamber's problem in Sutton & Barto, Reinforcement learning, 2nd edition. The book is freely available online.

Outline

Model-based Vs Model-free

- In model based RL (*i.e.* value iteration), you are given the transition probabilties and the reward function
- However, in practice, we rarely have access to those two
- In model free RL, consider RL algorithms which the agent learns directly from interacting with the environment (i.o.w. experiences)

Model free RL: Monte-Carlo RL

- MC methods learn directly from episodes of experience
- MC is model-free: no knowledge of MDP transitions / rewards
- MC learns from **complete** episodes
- Idea is: Value = mean return

Model free RL: Monte-Carlo RL

- To evaluate state s, every time step t that state s is visited in an episode,

$$
\begin{aligned}
N(s) &\leftarrow N(s) + 1 \\
S(s) &\leftarrow S(s) + G_t \\
V(s) &= S(s)/N(s)
\end{aligned}
$$

Model free RL: Monte-Carlo RL

- To evaluate state s, every time step t that state s is visited in an episode,

$$N(s) \leftarrow N(s) + 1$$
$$S(s) \leftarrow S(s) + G_t$$
$$V(s) = S(s)/N(s)$$

- Recall running average

$$\mu_k = \frac{1}{k}\sum_{j=1}^{k} x_j = \frac{1}{k}\left(x_k + \sum_{j=1}^{k-1} x_j\right) = \frac{1}{k}(x_k + (k-1)\mu_{k-1}) = \mu_{k-1} + \frac{1}{k}(x_k - \mu_{k-1})$$

Model free RL: Monte-Carlo RL

- To evaluate state s, every time step t that state s is visited in an episode,

$$N(s) \leftarrow N(s) + 1$$
$$S(s) \leftarrow S(s) + G_t$$
$$V(s) = S(s)/N(s)$$

- Recall running average

$$\mu_k = \frac{1}{k}\sum_{j=1}^{k} x_j = \frac{1}{k}\left(x_k + \sum_{j=1}^{k-1} x_j\right) = \frac{1}{k}(x_k + (k-1)\mu_{k-1}) = \mu_{k-1} + \frac{1}{k}(x_k - \mu_{k-1})$$

- For each state S_t with return G_t

$$N(S_t) \leftarrow N(S_t) + 1$$
$$V(S_t) \leftarrow V(S_t) + \frac{1}{N(S_t)}(G_t - V(S_t))$$

Model free RL: Temporal difference learning

- TD methods learn directly from episodes of experience
- TD is model-free: no knowledge of MDP transitions / rewards
- TD learns from **incomplete** episodes
- TD updates a guess towards a guess

TD(0)

- Every-visit Monte-Carlo: Update value $V(S_t)$ toward actual return G_t

$$V(S_t) \leftarrow V(S_t) + \alpha(G_t - V(S_t))$$

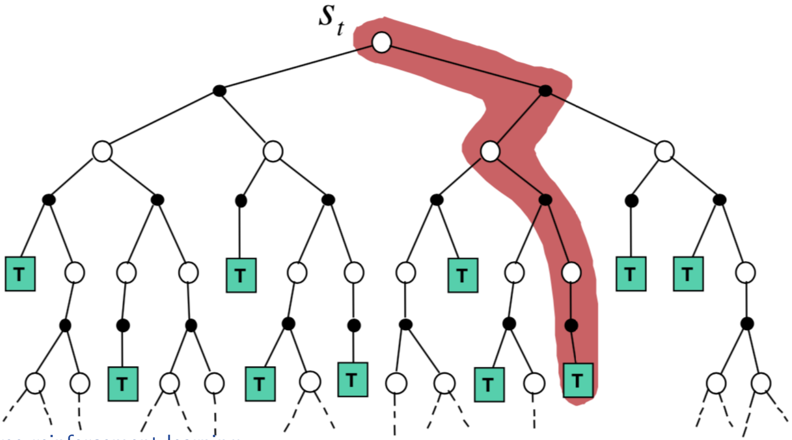
- Temporal difference learning, TD(λ): Update $V(S_t)$ towards **estimated return** given "one step of reality" $R_{t+1} + \gamma V(S_{t+1})$

$$V(S_t) \leftarrow V(S_t) + \alpha(\underbrace{\underbrace{R_{t+1} + \gamma V(S_{t+1})}_{\text{TD target}} - V(S_t)}_{\text{TD error}})$$

- "one step of reality" consists of 1) one step reward from S_t and 2) the realization of the next state S_{t+1} starting from S_t

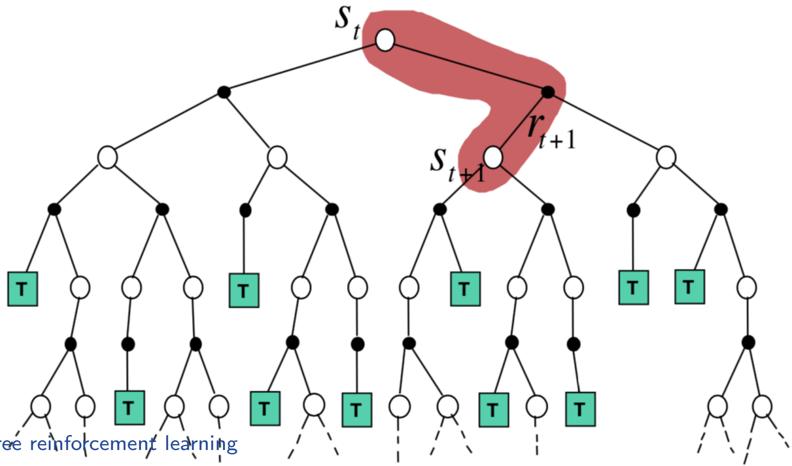
Monte-Carlo Backup

$$V(S_t) \leftarrow V(S_t) + \alpha \left(G_t - V(S_t) \right)$$



TD(0) Backup

$$V(S_t) \leftarrow V(S_t) + \alpha\left(R_{t+1} + \gamma V(S_{t+1}) - V(S_t)\right)$$



MC vs TD

- TD can learn before knowing the final outcome.
  - TD can learn online after every step
  - MC must wait until end of episode before return is known
- TD can learn without the final outcome
  - TD can learn from incomplete sequences
  - MC can only learn from complete sequences
  - TD works in continuing (non-terminating) environments
  - MC only works for episodic (terminating) environments

Q-learning algorithm

- Observe a transition $(S_t, A_t, R_{t+1}, S_{t+1})$

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha \left(R_{t+1} + \gamma \max_{a'} Q(S_{t+1}, a') - Q(S_t, A_t) \right)$$

Outline

Contraction mapping theorem

Theorem

For any metric space $\mathcal{M}$, a mapping $B : (M, d) \to (M, d)$ is called a *contraction mapping* if for some $0 \leqslant \gamma < 1$ and $\forall v_1, v_2 \in M$:

$$d(Bv_1, Bv_2) \leqslant \gamma d(v_1, v_2),$$

Contraction mapping properties

If B is a contraction mapping,

- $V^* = BV^*$, has a solution and it is unique
- $V_t = BV_{t-1} \implies V_t$ converges to V^*

$$\|V_t - V^*\|_\infty = \|BV_{t-1} - BV^*\|_\infty \leqslant \gamma\|V_{t-1} - V^*\|_\infty$$

- How do you prove the uniqueness of the fixed point V^* for the contraction mapping B? *Hint:* proof by contradiction

Bellman operator is a contraction mapping

Proof.

Define $[Bq](s,a) = \mathcal{R}_s^a + \gamma \sum_{s' \in \mathcal{S}} \mathcal{P}_{ss'}^a \max_{a'} q(s',a')$. Given q_1, and q_2 of size $|\mathcal{S}| \times |\mathcal{A}|$,

$$\|Bq_1 - Bq_2\|_\infty = \max_{s,a} \; |[Bq_1](s,a) - [Bq_2](s,a)|$$

$$= \max_{s,a} \; \left| \left(\mathcal{R}_s^a + \gamma \sum_{s' \in \mathcal{S}} \mathcal{P}_{ss'}^a \max_{a'} q_1(s',a') \right) - \left(\mathcal{R}_s^a + \gamma \sum_{s' \in \mathcal{S}} \mathcal{P}_{ss'}^a \max_{a'} q_2(s',a') \right) \right|$$

$$= \max_{\color{red}{s},\color{blue}{a}} \; \left| \gamma \sum_{s' \in \mathcal{S}} \mathcal{P}_{\color{red}{s}s'}^{\color{blue}{a}} \left(\max_{a'} q_1(s',a') - \max_{a'} q_2(s',a') \right) \right|$$

$$\leqslant \gamma \max_{s'} \; \left| \max_{a'} q_1(s',a') - \max_{a'} q_2(s',a') \right|$$

(replace exp. with max. then dependence on $\color{red}{s}$ and $\color{blue}{a}$ gone)

$$\leqslant \gamma \max_{s',a'} |q_1(s',a') - q_2(s',a')|$$

(max difference is greater than difference of maxes)

$$= \gamma \|q_1 - q_2\|_\infty$$

■

Outline

Q-learning algorithm

- Observe a transition $(S_t, A_t, R_{t+1}, S_{t+1})$

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha \left(R_{t+1} + \gamma \max_{a'} Q(S_{t+1}, a') - Q(S_t, A_t) \right)$$

Q-learning algorithm

- Observe a transition $(S_t, A_t, R_{t+1}, S_{t+1})$

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha \left(R_{t+1} + \gamma \max_{a'} Q(S_{t+1}, a') - Q(S_t, A_t) \right)$$

- What's the problem with this?

Q-learning algorithm

- Observe a transition $(S_t, A_t, R_{t+1}, S_{t+1})$

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha \left(R_{t+1} + \gamma \max_{a'} Q(S_{t+1}, a') - Q(S_t, A_t) \right)$$

- 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.

Q-learning algorithm

- Observe a transition $(S_t, A_t, R_{t+1}, S_{t+1})$

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha \left(R_{t+1} + \gamma \max_{a'} Q(S_{t+1}, a') - Q(S_t, A_t) \right)$$

- 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.

Q-learning algorithm

- 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 $\implies$ deep q-learning.

Deep Q-Networks (DQN) with fixed Q-targets

- Optimize the MSE between Q-network and Q-learning targets
- Forward pass:

$$\ell(\theta) = \frac{1}{2}\,\mathbb{E}_{s,a,r,s'}\left[\left(\underbrace{r + \gamma \max_{a'} Q(s',a';\theta^-)}_{\text{fixed parameter } \theta^-} - Q(s,a;\theta)\right)^2\right]$$

- Backward pass: Gradient update

$$-\nabla_\theta \ell(\theta) = \mathbb{E}_{s,a,r,s'}\left[\left(\underbrace{r + \gamma \max_{a'} Q(s',a';\theta^-)}_{\text{fixed parameter } \theta^-} - Q(s,a;\theta)\right)\nabla_\theta Q(s,a,\theta)\right]$$

Play Atari games with DQN

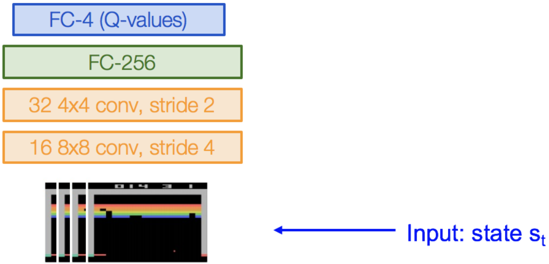
Objective: Complete the game with the highest score

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

16

Reinforcement learning

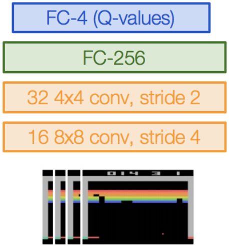
$Q(s, a; \theta)$:
neural network
with weights θ



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

Reinforcement learning

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



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

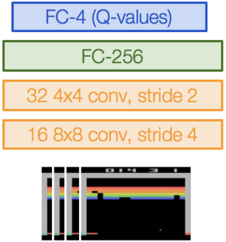
Number of actions between 4-18 depending on Atari game

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

Reinforcement learning

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

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



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

DQN training: Experience replay

- Learning from batches of consecutive samples is problematic:
  - Samples are correlated $\Longrightarrow$ inefficient learning
  - Current Q-network parameters determines next training samples

DQN training: Experience replay

- Learning from batches of consecutive samples is problematic:
  - Samples are correlated $\implies$ inefficient learning
  - Current Q-network parameters determines next training samples
- Address these problems using experience replay
  - Continuously 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

DQN training algorithm

Algorithm 1 Deep Q-learning with Experience Replay

Initialize replay memory $\mathcal{D}$ to capacity N
Initialize action-value function Q with random weights
for episode $= 1, M$ **do**
  Initialise sequence $s_1 = \{x_1\}$ and preprocessed sequenced $\phi_1 = \phi(s_1)$
  for $t = 1, T$ **do**
    With probability ϵ select a random action a_t
    otherwise select $a_t = \max_a Q^*(\phi(s_t), a; \theta)$
    Execute action a_t in emulator and observe reward r_t and image x_{t+1}
    Set $s_{t+1} = s_t, a_t, x_{t+1}$ and preprocess $\phi_{t+1} = \phi(s_{t+1})$
    Store transition $(\phi_t, a_t, r_t, \phi_{t+1})$ in $\mathcal{D}$
    Sample random minibatch of transitions $(\phi_j, a_j, r_j, \phi_{j+1})$ from $\mathcal{D}$
    Set $y_j = \begin{cases} r_j & \text{for terminal } \phi_{j+1} \\ r_j + \gamma \max_{a'} Q(\phi_{j+1}, a'; \theta) & \text{for non-terminal } \phi_{j+1} \end{cases}$
    Perform a gradient descent step on $(y_j - Q(\phi_j, a_j; \theta))^2$ according to equation 3
  end for
end for

Initialize replay memory, Q-network

DQN training algorithm

Algorithm 1 Deep Q-learning with Experience Replay

Initialize replay memory $\mathcal{D}$ to capacity N
Initialize action-value function Q with random weights
for episode $= 1, M$ **do** ← Play M episodes (full games)
  Initialise sequence $s_1 = \{x_1\}$ and preprocessed sequenced $\phi_1 = \phi(s_1)$
  for $t = 1, T$ **do**
    With probability ϵ select a random action a_t
    otherwise select $a_t = \max_a Q^*(\phi(s_t), a; \theta)$
    Execute action a_t in emulator and observe reward r_t and image x_{t+1}
    Set $s_{t+1} = s_t, a_t, x_{t+1}$ and preprocess $\phi_{t+1} = \phi(s_{t+1})$
    Store transition $(\phi_t, a_t, r_t, \phi_{t+1})$ in $\mathcal{D}$
    Sample random minibatch of transitions $(\phi_j, a_j, r_j, \phi_{j+1})$ from $\mathcal{D}$
    Set $y_j = \begin{cases} r_j & \text{for terminal } \phi_{j+1} \\ r_j + \gamma \max_{a'} Q(\phi_{j+1}, a'; \theta) & \text{for non-terminal } \phi_{j+1} \end{cases}$
    Perform a gradient descent step on $(y_j - Q(\phi_j, a_j; \theta))^2$ according to equation 3
  end for
end for

DQN training algorithm

Algorithm 1 Deep Q-learning with Experience Replay

Initialize replay memory $\mathcal{D}$ to capacity N
Initialize action-value function Q with random weights
for episode $= 1, M$ **do**
  Initialise sequence $s_1 = \{x_1\}$ and preprocessed sequenced $\phi_1 = \phi(s_1)$
  for $t = 1, T$ **do**
    With probability ϵ select a random action a_t
    otherwise select $a_t = \max_a Q^*(\phi(s_t), a; \theta)$
    Execute action a_t in emulator and observe reward r_t and image x_{t+1}
    Set $s_{t+1} = s_t, a_t, x_{t+1}$ and preprocess $\phi_{t+1} = \phi(s_{t+1})$
    Store transition $(\phi_t, a_t, r_t, \phi_{t+1})$ in $\mathcal{D}$
    Sample random minibatch of transitions $(\phi_j, a_j, r_j, \phi_{j+1})$ from $\mathcal{D}$
    Set $y_j = \begin{cases} r_j & \text{for terminal } \phi_{j+1} \\ r_j + \gamma \max_{a'} Q(\phi_{j+1}, a'; \theta) & \text{for non-terminal } \phi_{j+1} \end{cases}$
    Perform a gradient descent step on $(y_j - Q(\phi_j, a_j; \theta))^2$ according to equation 3
  end for
end for

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

DQN training algorithm

Algorithm 1 Deep Q-learning with Experience Replay

Initialize replay memory $\mathcal{D}$ to capacity N
Initialize action-value function Q with random weights
for episode $= 1, M$ **do**
  Initialise sequence $s_1 = \{x_1\}$ and preprocessed sequenced $\phi_1 = \phi(s_1)$
  for $t = 1, T$ **do**
    With probability ϵ select a random action a_t
    otherwise select $a_t = \max_a Q^*(\phi(s_t), a; \theta)$
    Execute action a_t in emulator and observe reward r_t and image x_{t+1}
    Set $s_{t+1} = s_t, a_t, x_{t+1}$ and preprocess $\phi_{t+1} = \phi(s_{t+1})$
    Store transition $(\phi_t, a_t, r_t, \phi_{t+1})$ in $\mathcal{D}$
    Sample random minibatch of transitions $(\phi_j, a_j, r_j, \phi_{j+1})$ from $\mathcal{D}$
    Set $y_j = \begin{cases} r_j & \text{for terminal } \phi_{j+1} \\ r_j + \gamma \max_{a'} Q(\phi_{j+1}, a'; \theta) & \text{for non-terminal } \phi_{j+1} \end{cases}$
    Perform a gradient descent step on $(y_j - Q(\phi_j, a_j; \theta))^2$ according to equation 3
  end for
end for

For each timestep t of the game

DQN training algorithm

Algorithm 1 Deep Q-learning with Experience Replay

Initialize replay memory $\mathcal{D}$ to capacity N
Initialize action-value function Q with random weights
for episode $= 1, M$ **do**
  Initialise sequence $s_1 = \{x_1\}$ and preprocessed sequenced $\phi_1 = \phi(s_1)$
  for $t = 1, T$ **do**
    With probability ϵ select a random action a_t
    otherwise select $a_t = \max_a Q^*(\phi(s_t), a; \theta)$
    Execute action a_t in emulator and observe reward r_t and image x_{t+1}
    Set $s_{t+1} = s_t, a_t, x_{t+1}$ and preprocess $\phi_{t+1} = \phi(s_{t+1})$
    Store transition $(\phi_t, a_t, r_t, \phi_{t+1})$ in $\mathcal{D}$
    Sample random minibatch of transitions $(\phi_j, a_j, r_j, \phi_{j+1})$ from $\mathcal{D}$
    Set $y_j = \begin{cases} r_j & \text{for terminal } \phi_{j+1} \\ r_j + \gamma \max_{a'} Q(\phi_{j+1}, a'; \theta) & \text{for non-terminal } \phi_{j+1} \end{cases}$
    Perform a gradient descent step on $(y_j - Q(\phi_j, a_j; \theta))^2$ according to equation 3
  end for
end for

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

DQN training algorithm

Algorithm 1 Deep Q-learning with Experience Replay

Initialize replay memory $\mathcal{D}$ to capacity N
Initialize action-value function Q with random weights
for episode $= 1, M$ **do**
  Initialise sequence $s_1 = \{x_1\}$ and preprocessed sequenced $\phi_1 = \phi(s_1)$
  for $t = 1, T$ **do**
    With probability ϵ select a random action a_t
    otherwise select $a_t = \max_a Q^*(\phi(s_t), a; \theta)$
    Execute action a_t in emulator and observe reward r_t and image x_{t+1}
    Set $s_{t+1} = s_t, a_t, x_{t+1}$ and preprocess $\phi_{t+1} = \phi(s_{t+1})$
    Store transition $(\phi_t, a_t, r_t, \phi_{t+1})$ in $\mathcal{D}$
    Sample random minibatch of transitions $(\phi_j, a_j, r_j, \phi_{j+1})$ from $\mathcal{D}$
    Set $y_j = \begin{cases} r_j & \text{for terminal } \phi_{j+1} \\ r_j + \gamma \max_{a'} Q(\phi_{j+1}, a'; \theta) & \text{for non-terminal } \phi_{j+1} \end{cases}$
    Perform a gradient descent step on $(y_j - Q(\phi_j, a_j; \theta))^2$ according to equation 3
  end for
end for

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

DQN training algorithm

Algorithm 1 Deep Q-learning with Experience Replay

Initialize replay memory $\mathcal{D}$ to capacity N
Initialize action-value function Q with random weights
for episode $= 1, M$ **do**
    Initialise sequence $s_1 = \{x_1\}$ and preprocessed sequenced $\phi_1 = \phi(s_1)$
    for $t = 1, T$ **do**
        With probability ϵ select a random action a_t
        otherwise select $a_t = \max_a Q^*(\phi(s_t), a; \theta)$
        Execute action a_t in emulator and observe reward r_t and image x_{t+1}
        Set $s_{t+1} = s_t, a_t, x_{t+1}$ and preprocess $\phi_{t+1} = \phi(s_{t+1})$
        Store transition $(\phi_t, a_t, r_t, \phi_{t+1})$ in $\mathcal{D}$
        Sample random minibatch of transitions $(\phi_j, a_j, r_j, \phi_{j+1})$ from $\mathcal{D}$
        Set $y_j = \begin{cases} r_j & \text{for terminal } \phi_{j+1} \\ r_j + \gamma \max_{a'} Q(\phi_{j+1}, a'; \theta) & \text{for non-terminal } \phi_{j+1} \end{cases}$
        Perform a gradient descent step on $(y_j - Q(\phi_j, a_j; \theta))^2$ according to equation 3
    end for
end for

Store transition in replay memory

DQN training algorithm

Algorithm 1 Deep Q-learning with Experience Replay

Initialize replay memory $\mathcal{D}$ to capacity N
Initialize action-value function Q with random weights
for episode $= 1, M$ **do**
  Initialise sequence $s_1 = \{x_1\}$ and preprocessed sequenced $\phi_1 = \phi(s_1)$
  for $t = 1, T$ **do**
    With probability ϵ select a random action a_t
    otherwise select $a_t = \max_a Q^*(\phi(s_t), a; \theta)$
    Execute action a_t in emulator and observe reward r_t and image x_{t+1}
    Set $s_{t+1} = s_t, a_t, x_{t+1}$ and preprocess $\phi_{t+1} = \phi(s_{t+1})$
    Store transition $(\phi_t, a_t, r_t, \phi_{t+1})$ in $\mathcal{D}$
    Sample random minibatch of transitions $(\phi_j, a_j, r_j, \phi_{j+1})$ from $\mathcal{D}$
    Set $y_j = \begin{cases} r_j & \text{for terminal } \phi_{j+1} \\ r_j + \gamma \max_{a'} Q(\phi_{j+1}, a'; \theta) & \text{for non-terminal } \phi_{j+1} \end{cases}$
    Perform a gradient descent step on $(y_j - Q(\phi_j, a_j; \theta))^2$ according to equation 3
  end for
end for

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

DQN results

	Replay Fixed-Q	Replay Q-learning	No replay Fixed-Q	No replay Q-learning
Breakout	316.81	240.73	10.16	3.17
Enduro	1006.3	831.25	141.89	29.1
River Raid	7446.62	4102.81	2867.66	1453.02
Seaquest	2894.4	822.55	1003	275.81
Space Invaders	1088.94	826.33	373.22	301.99