Lecture 22: Deep Reinforcement Learning II: Value-based Methods

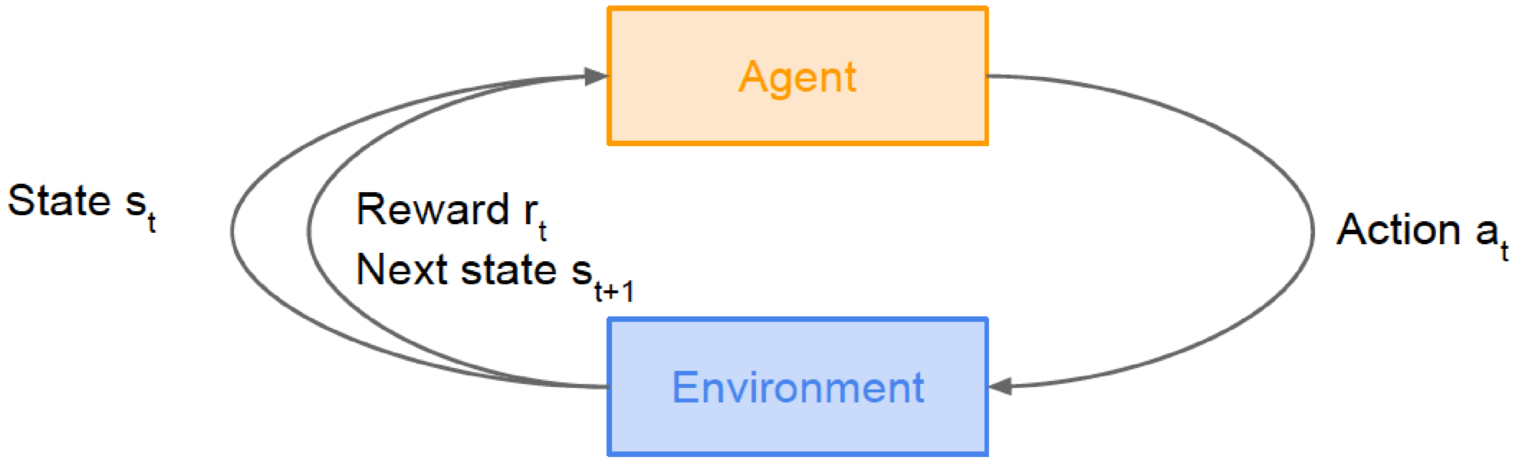
Xuming He
SIST, ShanghaiTech
Fall, 2019

Outline

- Problems in MDP
- Prediction
  - Value functions and temporal difference learning
- Control
  - Q functions and Q-learning
  - Deep Q-learning Network

Acknowledgement: David Silver's, Bhiksha Raj's and Feifei Li et al's notes

Markov Decision Processes



- Markov assumption:
  - All relevant information is encapsulated in the current state
  - i.e. the policy, reward, and transitions are all independent of past states given the current state
- Assume a fully observable environment, i.e. state can be observed directly

MDP

- Formal definition

Definition

A *Markov Decision Process* is a tuple $\langle \mathcal{S}, \mathcal{A}, \mathcal{P}, \mathcal{R}, \gamma \rangle$

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

Trajectory of MDP

- Observed instance of an MDP
  - initial state distribution $p(\mathbf{s}_0)$
  - policy $\pi_\theta(\mathbf{a}_t \mid \mathbf{s}_t)$
  - transition distribution $p(\mathbf{s}_{t+1} \mid \mathbf{s}_t, \mathbf{a}_t)$
  - reward function $r(\mathbf{s}_t, \mathbf{a}_t)$
- Finite horizon T (infinite case later)
  - Rollout, or trajectory $\tau = (\mathbf{s}_0, \mathbf{a}_0, \mathbf{s}_1, \mathbf{a}_1, \ldots, \mathbf{s}_T, \mathbf{a}_T)$
  - Probability of a rollout

$$p(\tau) = p(\mathbf{s}_0)\, \pi_\theta(\mathbf{a}_0 \mid \mathbf{s}_0)\, p(\mathbf{s}_1 \mid \mathbf{s}_0, \mathbf{a}_0) \cdots p(\mathbf{s}_T \mid \mathbf{s}_{T-1}, \mathbf{a}_{T-1})\, \pi_\theta(\mathbf{a}_T \mid \mathbf{s}_T)$$

  - Return for a rollout: $r(\tau) = \sum_{t=0}^{T} r(\mathbf{s}_t, \mathbf{a}_t)$

Finite and infinite horizon

- Finite horizon MDPs
  - Fixed number of steps T per episode
  - Maximize expected return $R = \mathbb{E}_{p(\tau)}[r(\tau)]$
- Infinite horizon MDPs
  - We can't sum infinitely many rewards, so we need to discount them: \$100 a year from now is worth less than \$100 today
  - Discounted return

$$G_t = r_t + \gamma r_{t+1} + \gamma^2 r_{t+2} + \cdots$$

  - Want to choose an action to maximize expected discounted return
  - The parameter $\gamma < 1$ is called the discount factor
    - small $\gamma =$ myopic
    - large $\gamma =$ farsighted

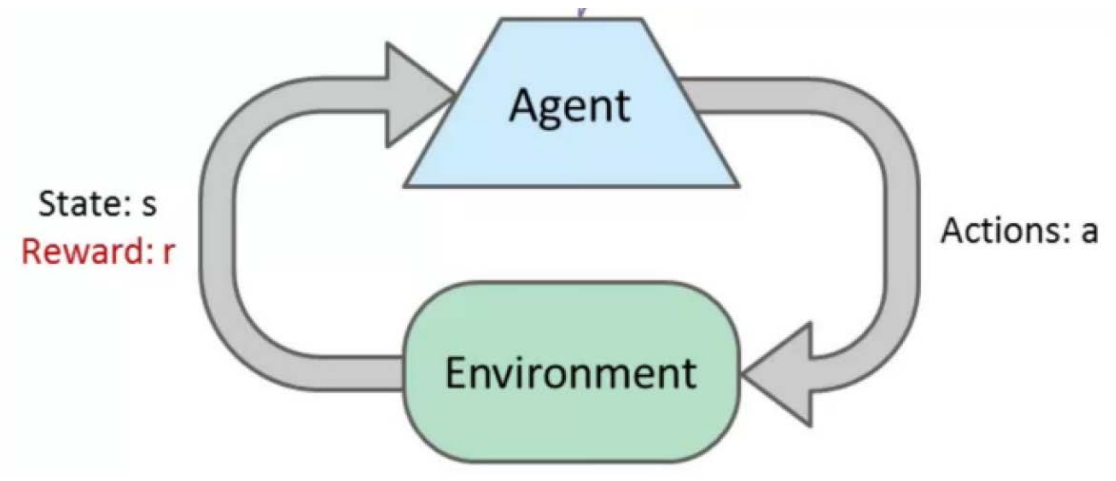
Problems in MDP

- Planning: given a **complete** MDP as input, compute policy with optimal expected return
  - Goal: maximize the expected return, $R = \mathbb{E}_{p(\tau)}[r(\tau)]$
  - The expectation is over both the environment's dynamics and the policy, but we only have control over the policy.

- Learning: given samples of trajectories of an ***unknown*** MDP,
  - Prediction: estimate the expected return given a policy
  - Control: find the optimal policy that maximizes the expected return

Reinforcement learning

- Agent interacts with an environment, which we treat as a black box
- Your RL code accesses it only through an API since it's external to the agent
  - i.e., you're not “allowed” to inspect the transition probabilities, reward distributions, etc.



Outline

- Problems in MDP
- Prediction
  - Value functions and temporal difference learning
- Control
  - Q functions and Q-learning
  - Deep Q-learning Network

Acknowledgement: David Silver’s, Bhiksha Raj’s and Feifei Li et al’s notes

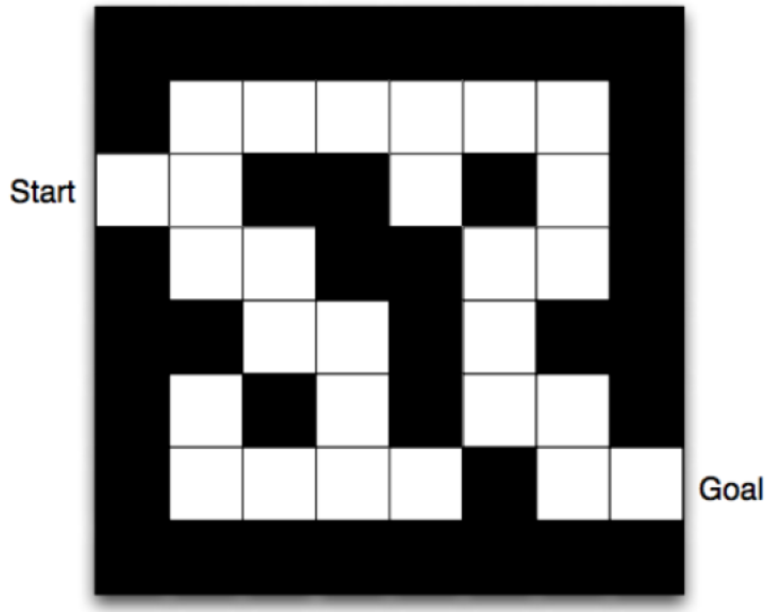
Prediction: Value function of MDP

- Value function $V^\pi(\mathbf{s})$ of a state $\mathbf{s}$ under policy π: the expected discounted return if we start in $\mathbf{s}$ and follow π

$$V^\pi(\mathbf{s}) = \mathbb{E}[G_t \mid \mathbf{s}_t = \mathbf{s}]$$
$$= \mathbb{E}\left[\sum_{i=0}^{\infty} \gamma^i r_{t+i} \mid \mathbf{s}_t = \mathbf{s}\right]$$

- Computing the value function is generally impractical, but we can try to approximate (learn) it
- The benefit is credit assignment: see directly how an action affects future returns rather than wait for rollouts

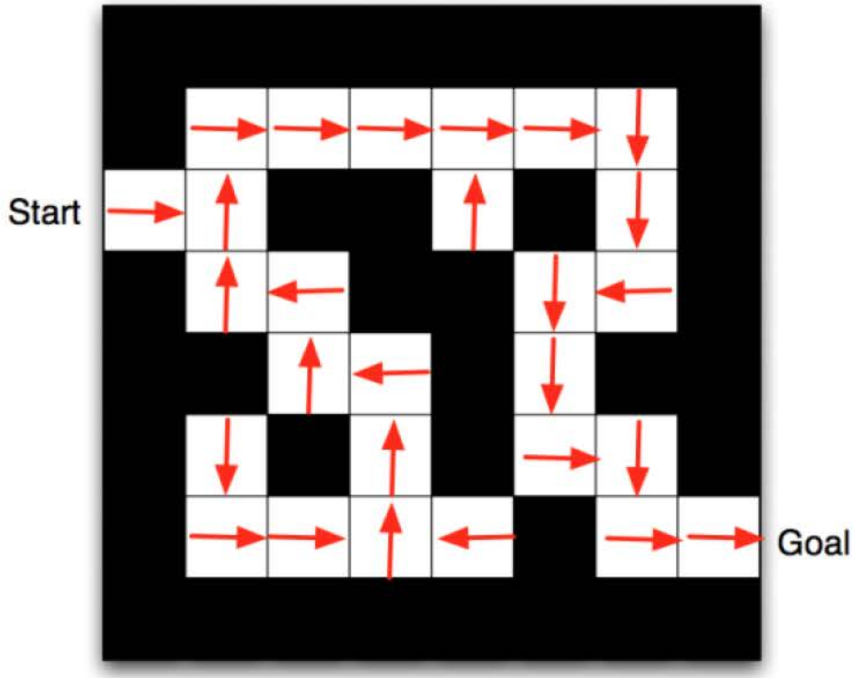
Value function example

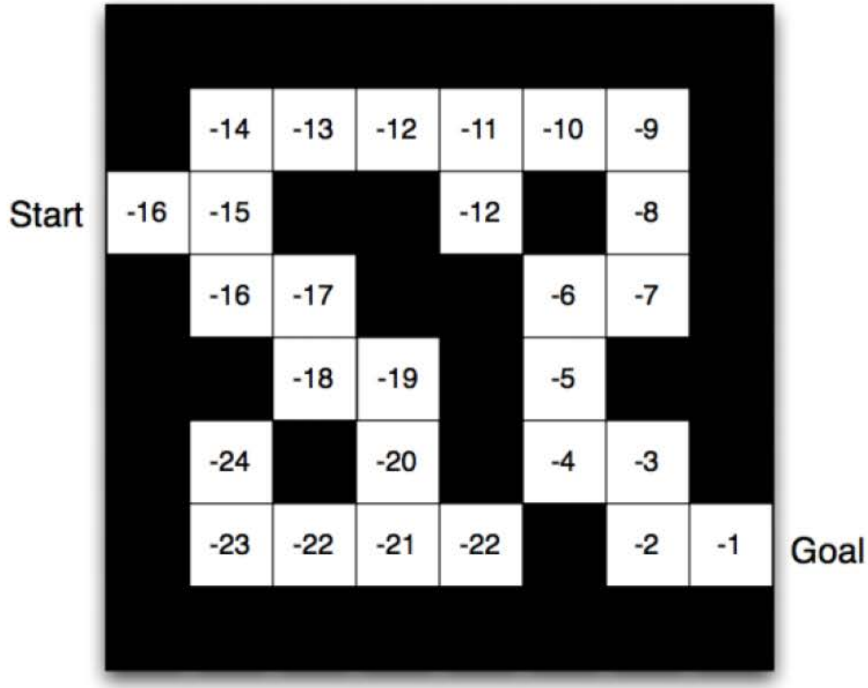


- Rewards: -1 per time step
- Undiscounted ($\gamma = 1$)
- Actions: N, E, S, W
- State: current location

Value function example

- Start from a state and follow the policy
  - Accumulate the reward along the trajectory





Model-free Prediction

- How to find the value of a policy, without knowing the underlying MDP?
  - Monte-Carlo learning
  - Temporal-difference learning

Monte-Carlo value estimation

- Objective: Estimate value function $v_\pi(s)$ for every state s, given recordings of the kind:

$$S_1, A_1, R_2, S_2, A_2, R_3, \ldots, S_T$$

- Recall, the value function is the expected return:

$$v_\pi(s) = E[G_t | S_t = s]$$
$$= E[R_{t+1} + \gamma R_{t+2} + \cdots + \gamma^{T-t-1} R_T | S_t = s]$$

- To estimate this, we replace the *statistical* expectation $E[G_t | S_t = s]$ by the *empirical* average $avg[G_t | S_t = s]$

Monte-Carlo value estimation

- We actually record *many* episodes
  - $episode(1) = S_{11}, A_{11}, R_{12}, S_{12}, A_{12}, R_{13}, \dots, S_{1T_1}$
  - $episode(2) = S_{21}, A_{21}, R_{22}, S_{22}, A_{22}, R_{23}, \dots, S_{2T_2}$
  - …
  - Different episodes may be different lengths

Monte-Carlo value estimation

- For each episode, we count the returns at all times:
  - $S_{11}, A_{11}, R_{12}, S_{12}, A_{12}, R_{13}, S_{13}, A_{13}, R_{14}, \ldots, S_{1T_1}$

$G_{1,1}$

- Return at time t
  - $G_{1,1} = R_{12} + \gamma R_{13} + \cdots + \gamma^{T_1-2} R_{1T_1}$

Monte-Carlo value estimation

- For each episode, we count the returns at all times:
  - $S_{11}, A_{11}, R_{12}, S_{12}, A_{12}, R_{13}, S_{13}, A_{13}, R_{14}, \dots, S_{1T_1}$

$G_{1,2}$

- Return at time t
  - $G_{1,1} = R_{12} + \gamma R_{13} + \cdots + \gamma^{T_1 - 2} R_{1T_1}$
  - $G_{1,2} = R_{13} + \gamma R_{14} + \cdots + \gamma^{T_1 - 3} R_{1T_1}$

Monte-Carlo value estimation

- To estimate the value of any state, identify the instances of that state in the episodes:
  - $S_{11}, A_{11}, R_{12}, S_{12}, A_{12}, R_{13}, S_{13}, A_{13}, R_{14}, \ldots, S_{1T_1}$

    $\boldsymbol{s_a}$ $\quad s_b \quad$ $\boldsymbol{s_a}$ $\ldots$
- Compute the average return from those instances

$$v_\pi(s_a) = avg(G_{1,1}, G_{1,3}, \ldots)$$

Temporal difference learning (TD)

- Online method for estimating the value of a policy
  - Idea: Update your value estimates after every observation

$$S_1, A_1, R_2, S_2, A_2, R_3, S_3, A_3, R_4, \ldots, S_T$$

Update for S_1 Update for S_2 Update for S_3

    – Do not actually wait until the end of the episode

Temporal difference learning (TD)

- Online method for estimating the value of a policy
  - Given a sequence $x_1, x_2, x_3, \ldots$ a running estimate of their average can be computed as

$$\mu_k = \frac{1}{k} \sum_{i=1}^{k} x_i$$

  - This can be rewritten as:

$$\mu_k = \frac{(k-1)\mu_{k-1} + x_k}{k}$$

  - And further refined to

$$\mu_k = \mu_{k-1} + \frac{1}{k}(x_k - \mu_{k-1})$$

Temporal difference learning (TD)

- Online method for estimating the value of a policy
  - Given a sequence $x_1, x_2, x_3, \ldots$ a running estimate of their average can be computed as

$$\mu_k = \mu_{k-1} + \frac{1}{k}(x_k - \mu_{k-1})$$

  - Or more generally as

$$\mu_k = \mu_{k-1} + \alpha(x_k - \mu_{k-1})$$

  - The latter is particularly useful for non-stationary environments

Temporal difference learning (TD)

- Online method for estimating the value of a policy
  - Given any episode

$$S_1, A_1, R_2, S_2, A_2, R_3, S_3, A_3, R_4, \dots, S_T$$

  - Update the value of each state visited

$$N(S_t) = N(S_t) + 1$$

$$v_\pi(S_t) = v_\pi(S_t) + \frac{1}{N(S_t)}\big(G_t - v_\pi(S_t)\big)$$

  - Incremental version

$$v_\pi(S_t) = v_\pi(S_t) + \alpha\big(G_t - v_\pi(S_t)\big)$$

  - Still an unrealistic rule
    - Requires the entire track until the end of the episode to compute Gt

Temporal difference learning (TD)

- Online method for estimating the value of a policy

$$v_\pi(S_t) = v_\pi(S_t) + \alpha\big(G_t - v_\pi(S_t)\big)$$

Problem

- But

$$G_t = R_{t+1} + \gamma G_{t+1}$$

- We can approximate G_{t+1} by the *expected* return at the next state $S_{t+1} \approx v_\pi(S_{t+1})$

$$G_t \approx R_{t+1} + \gamma v_\pi(S_{t+1})$$

- We don't know the real value of $v_\pi(S_{t+1})$ but we can "bootstrap" it by its current estimate

Temporal difference learning (TD)

- Online method for estimating the value of a policy

$$v_\pi(S_t) = v_\pi(S_t) + \alpha\delta_t$$

  - Where

$$\delta_t = R_{t+1} + \gamma v_\pi(S_{t+1}) - v_\pi(S_t)$$

  - δ_t is the TD *error*
    - The error between an (estimated) *observation* of G_t and the current estimate $v_\pi(S_t)$

Outline

- Problems in MDP
- Prediction
  - Value functions and temporal difference learning
- Control
  - Q functions and Q-learning
  - Deep Q-learning Network

Acknowledgement: David Silver's, Bhiksha Raj's and Feifei Li et al's notes

Control: Action-value function

- Expected return as a function of both state and action
  - Instead learn an action-value function, or Q-function: expected returns if you take action **a** and then follow your policy

$$Q^{\pi}(\mathbf{s}, \mathbf{a}) = \mathbb{E}[G_t \mid \mathbf{s}_t = \mathbf{s}, \mathbf{a}_t = \mathbf{a}]$$

  - Relationship:

$$V^{\pi}(\mathbf{s}) = \sum_{\mathbf{a}} \pi(\mathbf{a} \mid \mathbf{s}) Q^{\pi}(\mathbf{s}, \mathbf{a})$$

Optimal policy's value function

- The optimal policy maximize the expected total discounted reward at every state:

$$\pi^* = \arg\max_{\pi} \mathbb{E}[G_t \mid \mathbf{s}_t = \mathbf{s}]$$
$$= \mathbb{E}\left[\sum_{i=0}^{\infty} \gamma^i r_{t+i} \mid \mathbf{s}_t = \mathbf{s}\right]$$

- The optimal value function V* is the value function for π^*

- The optimal action-value function Q* is the action-value function for π^*

Optimal Q function

- For the optimal policy, easy to prove

$$\pi^*(\mathbf{a}|\mathbf{s}) = \begin{cases} 1 & \mathbf{a} = \arg\max_{\mathbf{a}'} Q^*(\mathbf{s}, \mathbf{a}') \\ 0 & \text{otherwise} \end{cases}$$

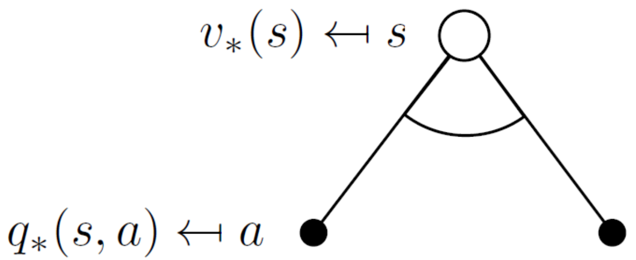
  - For any other policy π, $Q^\pi(\mathbf{s}, \mathbf{a}) \leq Q^*(\mathbf{s}, \mathbf{a})$

- Knowing the optimal action-value function is sufficient to find the optimal policy

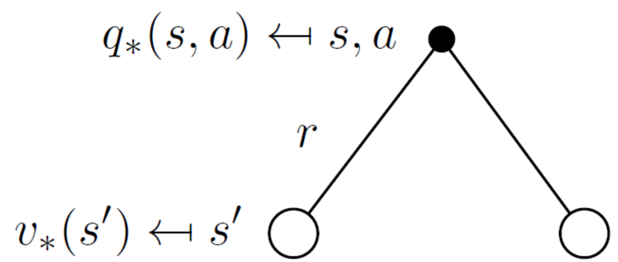
Optimal Bellman equations

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

$$V^*(\mathbf{s}) = \max_{\mathbf{a}} Q^*(\mathbf{s}, \mathbf{a})$$



$$Q^*(\mathbf{s}, \mathbf{a}) = r(\mathbf{s}, \mathbf{a}) + \gamma \mathbb{E}_{p(\mathbf{s}'|\mathbf{s},\mathbf{a})}[V^*(\mathbf{s}')]$$



Optimal Bellman equations

- The optimal value functions are recursively related by the Bellman optimality equations
  - Optimal value equation

$$V^*(\mathbf{s}) = \max_{\mathbf{a}} \{ r(\mathbf{s}, \mathbf{a}) + \gamma \mathbb{E}_{p(\mathbf{s}'|\mathbf{s},\mathbf{a})} [V^*(\mathbf{s}')] \}$$

  - Optimal action-value equation

$$Q^*(\mathbf{s}, \mathbf{a}) = r(\mathbf{s}, \mathbf{a}) + \gamma \mathbb{E}_{p(\mathbf{s}' \mid \mathbf{s},\mathbf{a})} \left[\max_{\mathbf{a}'} Q^*(\mathbf{s}_{t+1}, \mathbf{a}') \mid \mathbf{s}_t = \mathbf{s}, \mathbf{a}_t = \mathbf{a} \right]$$

  - A system of nonlinear equations (no closed-form solutions)
  - We can approximate Q* by trying to solve the optimal Bellman equation, which produces the optimal policy

Model-free Control

- How to find the optimal policy, without knowing the underlying MDP?
  - Q-learning

Q-Learning

- Optimal action-value equation

$$Q^*(\mathbf{s},\mathbf{a}) = r(\mathbf{s},\mathbf{a}) + \gamma \mathbb{E}_{p(\mathbf{s}' \mid \mathbf{s},\mathbf{a})} \left[\max_{\mathbf{a}'} Q^*(\mathbf{s}_{t+1},\mathbf{a}') \mid \mathbf{s}_t = \mathbf{s}, \mathbf{a}_t = \mathbf{a} \right]$$

- Let Q be an action-value function which hopefully approximates Q^*.
- The Bellman error is the update to our expected return when we observe the next state $\mathbf{s}'$.

$$\underbrace{r(\mathbf{s}_t,\mathbf{a}_t) + \gamma \max_{\mathbf{a}} Q(\mathbf{s}_{t+1},\mathbf{a})}_{\text{inside } \mathbb{E} \text{ in RHS of Bellman eqn}} - Q(\mathbf{s}_t,\mathbf{a}_t)$$

- The Bellman equation says the Bellman error is 0 in expectation
- Q-learning is an algorithm that repeatedly adjusts Q to minimize the Bellman error
- Each time we sample consecutive states and actions $(\mathbf{s}_t, \mathbf{a}_t, \mathbf{s}_{t+1})$:

$$Q(\mathbf{s}_t,\mathbf{a}_t) \leftarrow Q(\mathbf{s}_t,\mathbf{a}_t) + \alpha \underbrace{\left[r(\mathbf{s}_t,\mathbf{a}_t) + \gamma \max_{\mathbf{a}} Q(\mathbf{s}_{t+1},\mathbf{a}) - Q(\mathbf{s}_t,\mathbf{a}_t) \right]}_{\text{Bellman error}}$$

Exploration-exploitation tradeoff

- Visiting the entire state space

- Notice: Q-learning only learns about the states and actions it visits.
- Exploration-exploitation tradeoff: the agent should sometimes pick suboptimal actions in order to visit new states and actions.
- Simple solution: ϵ-greedy policy
  - With probability $1 - \epsilon$, choose the optimal action according to Q
  - With probability ϵ, choose a random action
- Believe it or not, ϵ-greedy is still used today!
- Q-learning is an off-policy algorithm: the agent can learn Q regardless of whether it's actually following the optimal policy
- Hence, Q-learning is typically done with an ϵ-greedy policy, or some other policy that encourages exploration.

Q-Learning algorithm

- Vanilla Q-learning algorithm

Initialize $Q(s,a), \forall s \in \mathcal{S}, a \in \mathcal{A}(s)$, arbitrarily, and $Q(\textit{terminal-state}, \cdot) = 0$
Repeat (for each episode):
  Initialize S
  Repeat (for each step of episode):
    Choose A from S using policy derived from Q (e.g., ε-greedy)
    Take action A, observe R, S'
    $Q(S,A) \leftarrow Q(S,A) + \alpha\big[R + \gamma \max_a Q(S',a) - Q(S,A)\big]$
    $S \leftarrow S'$;
  until S is terminal

Q-learning with function approximation

- So far, we've been assuming a tabular representation of Q: one entry for every state/action pair.
- This is impractical to store for all but the simplest problems, and doesn't share structure between related states.
- Solution: approximate Q using a parameterized function, e.g.
  - linear function approximation: $Q(\mathbf{s}, \mathbf{a}) = \mathbf{w}^\top \psi(\mathbf{s}, \mathbf{a})$
  - compute Q with a neural net
- Update Q using backprop:

$$t \leftarrow r(\mathbf{s}_t, \mathbf{a}_t) + \gamma \max_{\mathbf{a}} Q(\mathbf{s}_{t+1}, \mathbf{a})$$

$$\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \alpha(t - Q(\mathbf{s}, \mathbf{a})) \frac{\partial Q}{\partial \boldsymbol{\theta}}$$

Outline

- Problems in MDP
- Prediction
  - Value functions and temporal difference learning
- Control
  - Q functions and Q-learning
  - Deep Q-learning Network

Acknowledgement: David Silver's, Bhiksha Raj's and Feifei Li et al's notes

Deep Q Learning

- Approximate 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**!

  - Function parameters are the neural network weights

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

Backward Pass

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

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

Example: Atari Games

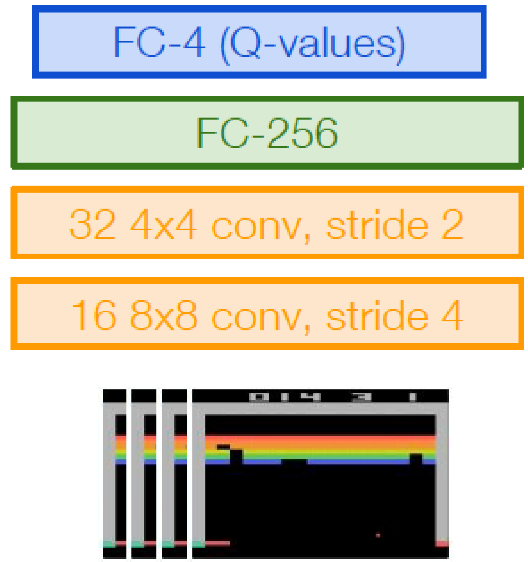
Objective: Complete the game with the highest score

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

Example: Atari Games

- Q-network

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



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

Example: Atari Games

- Q-network

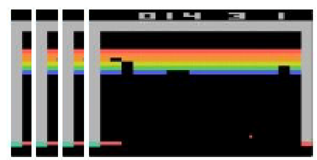
$Q(s, a; \theta)$:
neural network
with weights θ

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)

Sampling for Unknown MDP

- Q-network loss function

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):

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

Sampling for Unknown MDP

- Non-IID samples

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

Sampling for Unknown MDP

- Non-IID samples

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

Summary

- Reinforcement Learning
  - **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
- Next time
  - Actor-Critic and Inverse RL
  - Course review

Deep Q-Learning with Experience Replay

Algorithm 1 Deep Q-learning with Experience Replay

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

Initialize replay memory, Q-network

Deep Q-Learning with Experience Replay

Algorithm 1 Deep Q-learning with Experience Replay

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

Deep Q-Learning with Experience Replay

Algorithm 1 Deep Q-learning with Experience Replay

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

For each timestep t of the game

Deep Q-Learning with Experience Replay

Algorithm 1 Deep Q-learning with Experience Replay

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

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

Deep Q-Learning with Experience Replay

Algorithm 1 Deep Q-learning with Experience Replay

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

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

Deep Q-Learning with Experience Replay

Algorithm 1 Deep Q-learning with Experience Replay

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

Store transition in replay memory

Deep Q-Learning with Experience Replay

Algorithm 1 Deep Q-learning with Experience Replay

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

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