Reinforcement Learning and Q-Learning

Hantao Zhang
CS4980-003
Deep Learning with Python

https://en.wikipedia.org/wiki/Reinforcement_learning
https://en.wikipedia.org/wiki/Q-learning
https://www.udacity.com/course/reinforcement-learning--ud600

Acknowledgement: Based on Svetlana Lazebnik’s lecture slides
Learning types

- **Supervised learning:**
  a situation in which sample (input, output) pairs of the function to be learned can be perceived or are given
  - You can think it as if there is a kind teacher

  ![Example Label]

  - **Situation**  **Reward**

- **Reinforcement learning:**
  in the case of the agent acts on its environment, it receives some evaluation of its action (reinforcement), but is not told of which action is the correct one to achieve its goal

  ![Situation Reward ... Situation Reward]

  - **Situation**  **Reward**  **...**  **Situation**  **Reward**
Reinforcement Learning (RL)

- **Reinforcement learning** (RL) is an area of **machine learning** inspired by **behaviourist psychology**, concerned with how **software agents** ought to take **actions** in an **environment** so as to maximize some notion of cumulative **reward**.

- The problem, due to its generality, is studied in many other disciplines, such as **game theory**, **control theory**, **operations research**, **information theory**, **simulation-based optimization**, **multi-agent systems**, **swarm intelligence**, **statistics** and **genetic algorithms**.

- In the operations research and control literature, reinforcement learning is called **approximate dynamic programming**, or **neuro-dynamic programming**.
Reinforcement learning

• Task
  Learn how to act successfully to achieve a goal while interacting with an external environment

• Learn via experiences!

Examples

  **Game playing**: player knows whether it’ll win or lose, but not know how to move at each step

  **Control**: a traffic system can measure the delay of cars, but not know how to decrease it.
A ball is falling every few seconds.
Goal: move the paddle to catch the ball before it reaches the ground.
Actions of the paddle: LEFT (0), STILL (1), RIGHT (2)

- reward +1 if the ball is caught
- reward –1 if the ball is missed

What’s the strategy to achieve max reward?
Example: Backgammon Game

White pieces move counterclockwise
Black pieces move clockwise
Example: Resource allocation in datacenters

- A Hybrid Reinforcement Learning Approach to Autonomic Resource Allocation. Tesauro, Jong, Das, Bennani (IBM), ICAC 2006
Other Examples

• Pole-balancing
• Play ping bong

• No teacher who would say “good” or “bad”
  • is reward “10” good or bad?
  • rewards could be delayed

• Similar to control theory
  • more general, fewer constraints

• Explore the environment and learn from experience
  • not just blind search, try to be smart about it
Designing Rewards

- Robot in a maze
  - episodic task, not discounted, +1 when out, 0 for each step

- Chess
  - GOOD: +1 for winning, -1 losing
  - BAD: +0.25 for taking opponent’s pieces
    - high reward even when lose

- Rewards
  - indicate what we want to accomplish
  - NOT how we want to accomplish it

- Shaping
  - positive reward often very “far away”
  - rewards for achieving subgoals (domain knowledge)
  - also: adjust initial policy or initial value function
Reinforcement Learning

- No knowledge of environment
  - Can only act in the world and observe states and reward
- Many factors make RL difficult:
  - Actions have **non-deterministic effects**
    - which are initially unknown
  - **Rewards / punishments** are infrequent
    - Often at the end of long sequences of actions
    - How do we determine what action(s) were really responsible for reward or punishment? (credit assignment)
  - World is large and complex
- Nevertheless learner must decide what actions to take
  - At first, we need a formal specification for RL
State Representation

• Pole-balancing example:
  • move car left/right to keep the pole balanced
  • coarse discretization of state variables
    • left, center, right

• State representation
  • position and velocity of car
  • angle and angular velocity of pole

• Initial state: starting point

• Terminal state: end points

• Actions result in new states

• What about Markov property?
  • would need more info
  • noise in sensors, temperature, bending of pole
  • non-Markov still works
Formal Specification for RL

- **State Variable s:**
  - Ball’s position (ballx, bally)
  - Paddle’s position (padx, pady)

- **Action a:**
  - One of LEFT (0), STILL (1), RIGHT (2)

- **Reward r:**
  - The reward received so far.

- **Move →:**
  \[
  [s, a, r] \rightarrow [s', a', r']
  \]

  where \( s' = f(s, a) \) and \( r' = g(s, a) \)

For this example: Let \( s' = \{(\text{ballx}', \text{bally}'), (\text{padx}', \text{pady}')\} \), then

- \( \text{ballx}' = \text{ballx} \)  
  # ball only goes down
- \( \text{bally}' = \text{bally} + \text{ball\_speed} \)
- \( \text{padx}' = \text{padx} + (a - 1) \times \text{pad\_speed} \)  
  \( r' = 1 \) if pad touches ball
- \( \text{pady}' = \text{pady} \)  
  # pad moves horizontal  
  \( = -1 \) else if bally < pady
- \( \text{pady}' = \text{pady} \)  
  # pad moves horizontal
  \( = 0 \) else if bally > pady

**Terminal state:** pad touches ball or bally < pady.
A round of plays is a sequence of moves $\rightarrow$:

$$[s_0, a_0, r_0] \rightarrow [s_1, a_1, r_1] \rightarrow \ldots \rightarrow [s_{k-1}, a_{k-1}, r_{k-1}] \rightarrow [s_k, _, r_k]$$

where $s_k$ is a terminal state.

The goal is to maximize $\sum_i r_i$, the sum of rewards.
• Pygame is used for the graphic display.
• Three Methods:
  • Constructor __init__(): Define all parameters.
  • reset(): Define the initial state
  • step(act): Apply action “act” on the current state and generate the next state and the reward.
class PaddleGame():

def __init__(self):
    pygame.init()
    pygame.key.set_repeat(10, 100)

    # set constants
    self.color_white = (255, 255, 255)
    self.color_black = (0, 0, 0)
    self.game_width, self.game_height = 400, 400
    self.ball_width, self.ball_height = 20, 20
    self.pad_width, self.pad_height = 50, 10
    self.game_floor, self.game_ceiling = 350, 10    # ball moves 34 times/round

    # Based on experimentation, the ball tends to move 4 times
    # between each pad movement. Since here we alternate ball
    # and pad movement, we make ball move twice faster.
    self.ball_speed, self.pad_speed = 10, 20
    self.font_size = 30
    self.CUSTOM_EVENT = pygame.USEREVENT + 1
class PaddleGame:

def reset(self):
    # create the initial state
    self.frames = collections.deque(maxlen=4)
    self.game_over = False

    # initialize positions
    self.pad_x = self.game_width // 2  # paddle starts at the middle
    self.game_score = 0
    self.reward = 0
    self.ball_x = random.randint(0, self.game_width)  # ball starts at the top
    self.ball_y = self.game_ceiling
    self.num_tries = 0

    # set up display, clock, etc
    self.screen = pygame.display.set_mode((self.game_width, self.game_height))
    self.clock = pygame.time.Clock()

def get_frames(self):
    return np.array(list(self.frames))
class PaddleGame():

def step(self, action):
    # one move of the game
    pygame.event.pump()
    if action == 0:
        # move pad left
        self.pad_x -= self.pad_speed
        if self.pad_x < 0:
            self.pad_x = self.pad_speed
    elif action == 2:
        # move pad right
        self.pad_x += self.pad_speed
        if self.pad_x > self.game_width - self.pad_width:
            self.pad_x = self.game_width - self.pad_width - self.pad_speed
    else:
        # don’t move pad
        pass

    self.screen.fill(self.color_black)

    # update ball position
    self.ball_y += self.ball_speed  # ball goes down

    ball = pygame.draw.rect(self.screen, self.color_white, pygame.Rect(self.ball_x, self.ball_y, self.ball_width, self.ball_height))
# update paddle position
pad = pygame.draw.rect(self.screen, self.color_white,
    pygame.Rect(self.pad_x, self.game_floor,
                 self.pad_width, self.pad_height))

self.reward = 0  # check for collision and update reward
if self.ball_y >= self.game_floor - self.ball_width // 2:  # terminal state
    self.reward = 1 if ball.colliderect(pad) else -1
    self.game_score += self.reward
    self.ball_x = random.randint(0, self.game_width)  # new ball position
    self.ball_y = self.game_ceiling
    self.num_tries += 1

# save last 4 states (frames):
self.frames.append([self.ball_x, self.ball_y, self.paddle_x])
if self.num_tries >= self.max_tries_per_game: self.game_over = True

pygame.display.flip()
sel.exit.tick(30)
return self.get_frames(), self.reward, self.game_over
if __name__ == "__main__":
    game = Randomgame()

    num_epochs = 5
    for e in range(num_epochs):
        print("Epoch: {:d}".format(e))
        game.reset()  # reset initial state

        game_over = False
        while not game_over:
            action = np.random.randint(0, 3)  # random in [0, 3)
            last4frames, reward, game_over = game.step(action)
            print("\r action = %d, reward = %2d" % (action, reward))

        print("\n game_score =", game.game_score)
• Delayed Reward makes it hard to learn
• The choice in State $s$ was important: $s_2$ gives higher reward than $s_1$ immediately, but, it seems $s_1$ leads later to the big reward in $s'$
• How you deal with this problem?

• Can we learn a good strategy (policy)?
• Learning with evaluative feedback
  • Learner’s output is “scored” by a scalar signal (“Reward” or “Payoff” function) saying how well it did
  • Supervised learning: Learner is told the correct answer!
  • May need to try different actions just to see how well they score (exploration …)
Offline (Planning) vs. Online (RL)

Offline Solution

Online Learning

Source: Berkeley CS188
Basic Scheme of RL

- In each time step:
  - Take some action
  - Observe the outcome of the action: successor state and reward
  - Update some internal representation of the environment and policy
  - If you reach a terminal state, just start over (each pass through the environment is called a trial)

- Why is this called reinforcement learning?
Reinforcement learning strategies

• **Model-based**
  • Learn concurrently the model of the MDP (Markov Decision Processes, which has a table of transition probabilities) and the optimal policy to play.

• **Model-free**
  • Learn how to act without explicitly learning the transition probabilities \( P(s' | s, a) \)
  • **Q-learning**: learn an action-utility function \( Q(s,a) \) that tells us the value of doing action \( a \) in state \( s \)
• Each percept is enough to determine the State (the state is accessible)
• The learner can decompose the Reward component from a percept.
• The learner’s task: to find an optimal policy, mapping state to action at each move, that maximize long-run measure of the reinforcement.
• Think of reinforcement as reward.
• Can be modeled as Markov Decision Processes (MDP) model!
Policy of Actions

• Move →: \([s, a, r] \rightarrow [s', a', r']\), where \(s' = f(s, a)\) and \(r' = g(s, a)\), \(f\) and \(g\) are problem dependent.

• A round of plays is a sequence of moves →:
  \([s_0, a_0, r_0] \rightarrow [s_1, a_1, r_1] \rightarrow \ldots \rightarrow [s_{k-1}, a_{k-1}, r_{k-1}] \rightarrow [s_k, \_, r_k]\)
  where \(s_k\) is a terminal state.

• A policy \(\pi\) is a function that takes a state as input and return an action for that state: \(\pi(s_i) = a_i\).

• If we also denote \(r_i\) by \(R(s_i) = r_i\), then a round of plays can denoted by
  \(s_0 \rightarrow s_1 \rightarrow \ldots \rightarrow s_{k-1} \rightarrow s_k\)
  together with \(\pi\) and \(R\).
Computing expected rewards

- Episodic (vs. continuing) tasks
  - “game over” after N steps
  - Optimal policy depends on N; harder to analyze

- Additive rewards
  - $V(s_0, s_1, ...) = r(s_0) + r(s_1) + r(s_2) + ...$
  - Infinite value for continuing tasks

- Discounted rewards
  - $V(s_0, s_1, ...) = r(s_0) + \gamma^1 r(s_1) + \gamma^2 r(s_2) + ...$
  - Value bounded if rewards bounded
To define the utility of a state sequence, *discount* the individual state rewards by a factor $\gamma$ between 0 and 1:

$$U([s_0, s_1, s_2, \ldots]) = R(s_0) + \gamma R(s_1) + \gamma^2 R(s_2) + \ldots$$
To define the utility of a state sequence, discount the individual state rewards by a factor $\gamma$ between 0 and 1:

$$U([s_0, s_1, s_2, \ldots]) = R(s_0) + \gamma R(s_1) + \gamma^2 R(s_2) + ...$$

$$= \sum_{t=0}^{\infty} \gamma^t R(s_t) \frac{R_{\text{max}}}{1}$$

(0 < $\gamma$ < 1)

- Sooner rewards count more than later rewards
- Makes sure the total utility stays bounded
- Helps algorithms converge
Given a round of plays $s_0 \rightarrow s_1 \rightarrow \ldots \rightarrow s_{k-1} \rightarrow s_k$ with $\pi$ and $R$, define

$$U^\pi(s) = E \left[ \sum_{t=0} \gamma^t R(s_t) \mid \pi, s_0 = s \right]$$

- It is called expected sum of rewards when the policy $\pi$ is followed.
- The discount ratio $\gamma$, $0 < \gamma \leq 1$
- To compute $U^\pi(s)$, let us at first study the Markov Decision Processes (MDP).
Markov Decision Processes

Decision making in a *stochastic, sequential* environment

- Agent’s actions affect its subsequent experiences
- Instead of supervision at each step, the agent gets an occasional *reward signal*
- Feedback may be delayed
Markov Decision Processes

• Components:
  - **States** $s$, beginning with initial state $s_0$
  - **Actions** $a$
    • Each state $s$ has actions $A(s)$ available from it
  - **Transition model** $P(s' \mid s, a)$, or as $T(s, a, s')$
    • *Markov assumption*: the probability of going to $s'$ from $s$ depends only on $s$ and $a$ and not on any other past actions or states
  - **Reward function** $R(s)$
  - **Policy** $\pi(s)$: the action that an agent takes in any given state
    • The “solution” to an MDP
Game Show

- A series of questions with increasing level of difficulty and increasing payoff
- Decision: at each step, take your earnings and quit, or go for the next question
  - If you answer wrong, you lose everything
Game Show

- Consider $50,000 question
  - Probability of guessing correctly: 1/1
  - Quit or go for the question?
- What is the expected payoff for continuing?
  - $0.1 \times 61,100 + 0.9 \times 0 = $6,110
- What is the optimal decision?
Game Show

- What should we do in Q3?
  - Payoff for quitting: $1,100
  - Payoff for continuing: $5,550
- What about Q2?
  - $100 for quitting vs. $4,162 (future earning) for continuing
- What about Q1?

\[ U = \begin{cases} \text{Correct: } \$3746 \text{ or } \$90 \\ \text{Incorrect: } \$0 \end{cases} \]
\[ U = \begin{cases} \text{Correct: } \$4162 \text{ or } \$825 \\ \text{Incorrect: } \$0 \end{cases} \]
\[ U = \begin{cases} \text{Correct: } \$5,550 \\ \text{Incorrect: } \$0 \end{cases} \]
\[ U = \text{Correct: } \$11,100 \]

- Quit: $100
- Quit: $1,100
- Quit: $11,100
- Quit: $61,100
Grid world

\[ R(s) = -0.04 \text{ for every non-terminal state} \]

Transition model:

Source: P. Abbeel and D. Klein
Goal: Policy

Source: P. Abbeel and D. Klein
Grid World

Transition model:

$R(s) = -0.04$ for every non-terminal state
Grid World

Optimal policy when \( R(s) = -0.04 \) for every non-terminal state
Solving MDPs

- MDP components:
  - **States** $s$
  - **Actions** $a$
  - **Transition model** $P(s' \mid s, a)$
  - **Reward function** $R(s)$

- The solution:
  - **Policy** $\pi(s)$: mapping from states to actions
  - How to find the optimal policy?
Defining the Optimal Policy

- Given a policy \( \pi \), we can define the *expected utility* over all possible state sequences produced by following that policy:

\[
U(s_0) = \sum_{\text{state sequences}} P(\text{sequence}) U(\text{sequence})
\]

starting from \( s_0 \)

- The optimal policy should maximize this utility:

\[
U^\pi(s) = \mathbb{E} \left[ \sum_{t=0}^{\infty} \gamma^t R(s_t) \mid \pi, s_0 = s \right]
\]
Define state utility $U(s)$ as the expected sum of discounted rewards if the agent executes an optimal policy starting in state $s$.

The tree is called expectimax decision tree, similar to minimax tree.
“Game tree” view

- What is the expected utility of taking action $a$ in state $s$?
  $$\sum_{s'} P(s' | s, a) U(s')$$

- How do we choose the optimal action?
  $$\pi^*(s) = \arg \max_{a \in A(s)} \sum_{s'} P(s' | s, a) U(s')$$

- What is the recursive expression for $U(s)$ in terms of the utilities of its successor states?
  $$U(s) = R(s) + \gamma \max_a \sum_{s'} P(s' | s, a) U(s')$$
The Bellman Equation

• Recursive relationship between the utilities of successive states:

\[ U(s) = R(s) + \gamma \max_{a \in A(s)} \sum_{s'} P(s' | s, a) U(s') \]
The Bellman Equation

- Recursive relationship between the utilities of successive states:

\[ U(s) = R(s) + \gamma \max_{a \in A(s)} \sum_{s'} P(s'| s, a) U(s') \]

- For \( N \) states, we get \( N \) equations in \( N \) unknowns
  - Solving them solves the MDP
  - We could try to solve them through expectimax search, but that would run into trouble with infinite sequences
  - Instead, we solve them algebraically by Dynamic Programming, or approximately by policy iteration and value iteration
Policy Iteration

- **Goal:** Learn a good policy
- **Start** with some initial policy $\pi_0$ and alternate between the following steps:
  - **Policy evaluation:** calculate utility for every state $s$ under current policy $\pi_i$
  - **Policy improvement:** calculate a new policy $\pi_{i+1}$ based on the updated utilities
Policy iteration

- Start with some initial policy $\pi_0$ and alternate between the following steps:
  - **Policy evaluation:** calculate utility for every state $s$ under current policy $\pi_i$
    - Bellman equation for fixed policy:
      \[
      U_{i+1}(s) = R(s) + \sum_{s'} P(s' \mid s, \pi_i(s)) U_{i+1}(s')
      \]
    - Can solve a linear system to get all the utilities!
  - Alternatively, can apply the following update:
    \[
    U_{i+1}(s) \leftarrow R(s) + \sum_{s'} P(s' \mid s, \pi_i(s)) U_i(s')
    \]
Policy iteration

• Start with some initial policy \( \pi_0 \) and alternate between the following steps:
  
  • **Policy evaluation**: calculate utility for every state \( s \) under current policy \( \pi_i \)
  
  • **Policy improvement**: calculate a new policy \( \pi_{i+1} \) based on the updated utilities

\[
i_{i+1}(s) = \arg\max_{a \in A(s)} P(s' | s, a) U_{i+1}(s')\]

Value iteration

- Goal: Decide the value $U(s)$ precisely.
- Start out with every $U(s) = 0$
- Iterate until convergence
  - During the $i^{th}$ iteration, update the utility of each state according to this rule:

  $$U_{i+1}(s) \leftarrow R(s) + \gamma \max_{a \in A(s)} \sum_{s'} P(s'|s,a)U_i(s')$$

- In the limit of infinitely many iterations, guaranteed to find the correct utility values
  - In practice, don’t need an infinite number of iterations…
Model-based reinforcement learning

- **Learning the model:**
  - Keep track of how many times state \( s' \) follows state \( s \) when you take action \( a \) and update the transition probability \( P(s' | s, a) \) according to the relative frequencies
  - Keep track of the rewards \( R(s) \)

- **Learning how to act:**
  - Estimate the utilities \( U(s) \) using Bellman’s equations
  - Choose the action that maximizes expected future utility:

    \[
    \pi^*(s) = \arg \max_{a \in A(s)} \sum_{s'} P(s' | s, a) U(s')
    \]

- Is there any problem with this “greedy” approach?
Q value

When a learner takes action $a_t$ in state $s_t$ at time $t$, the predicted future (discounted) rewards is stored as $Q(s_t, a_t)$.

$$Q(s_t, a_t) = E\left\{r_{t+1} + \gamma r_{t+2} + \gamma^2 r_{t+3} + \gamma^3 r_{t+4} + \ldots \right\}$$

where $\gamma$ is the discount ratio

Example:

$Q^3(s_t, a_t) = 0$
$Q^2(s_t, a_t) = 1$
$Q^1(s_t, a_t) = 2$

For this example, a learner should take action $a^1_t$ because the corresponding Q value $Q^1(s_t, a_t)$ is max.
Q-learning

- **Q-learning** was introduced by Watkins in 1989, while addressing “Learning from delayed rewards”, the title of his PhD Thesis.

- **Q-learning** is a reinforcement learning technique used in machine learning. The technique does not require a model of the environment. Q-learning can handle problems with stochastic transitions and rewards.

- "Q" names the function or equivalent that returns the reward used to provide the reinforcement and can be said to stand for the "quality" of an action taken in a given state.

- **Q-learning** learns the expected reward (utility value) of taking a particular action \( a \) in a particular state \( s \), that is, Q-value of the pair \((s, a)\).
Model-free reinforcement learning

- **Idea:** learn how to act without explicitly learning the transition probabilities \( P(s' | s, a) \)
- **Q-learning:** learn an *action-utility function* \( Q(s,a) \) that tells us the value of doing action \( a \) in state \( s \)
Model-free reinforcement learning

- **Idea**: learn how to act without explicitly learning the transition probabilities $P(s' | s, a)$
- **Q-learning**: learn an action-utility function $Q(s, a)$ that tells us the value of doing action $a$ in state $s$

- Relationship between Q-values and utilities:
  \[ U(s) = \max_a Q(s, a) \]

- With Q-values, you don’t need the transition model to select the next action:
  \[ \pi^*(s) = \arg \max_a Q(s, a) \]

- Compare with:
  \[ \pi^*(s) = \arg \max_a \sum_{s'} P(s' | s, a) U(s') \]
Model-free reinforcement learning

- **Q-learning:** learn an action-utility function $Q(s, a)$ that tells us the value of doing action $a$ in state $s$

  $$U(s) = \max_a Q(s, a)$$

- Bellman equation for $Q$ values:

  $$Q(s, a) = R(s) + \gamma \sum_{s'} P(s'| s, a) \max_{a'} Q(s', a')$$

- Compare to Bellman equation for utilities:

  $$U(s) = R(s) + \gamma \max_{a \in A(s)} \sum_{s'} P(s'| s, a) U(s')$$
Q learning: Model-Free RL

- **Q-learning**: learn an action-utility function $Q(s, a)$ that tells us the value of doing action $a$ in state $s$

  $$U(s) = \max_a Q(s, a)$$

- Bellman equation for Q values:

  $$Q(s, a) = R(s) + \gamma \sum_{s'} P(s'|s, a) \max_{a'} Q(s', a')$$

- Problem: we don’t know (and don’t want to learn) $P(s'|s, a)$

- Solution: build up estimates of $Q(s, a)$ over time by making small updates based on observed transitions
First, Q value can be transformed as follows.

\[ Q(s_t, a_t) = E\left\{ r_{t+1} + \gamma r_{t+2} + \gamma^2 r_{t+3} + \gamma^3 r_{t+4} + \ldots \right\} \]

\[ = E\left\{ \sum_{k=0}^{\infty} \gamma^k r_{t+k+1} \right\} \]

\[ = E\left\{ r_{t+1} + \gamma \sum_{k=0}^{\infty} \gamma^k r_{t+k+2} \right\} \]

\[ = E\left\{ r_{t+1} + \gamma Q(s_{t+1}, a_{t+1}) \right\} \]

As a result, the Q value at time \( t \) is easily calculated by \( r_{t+1} \) and Q value of the next step.

\[ Q(s_t, a_t) = r + \gamma \max_a \{ Q(s_{t+1}, a) \} \]
Q-Learning is Temporal Difference RL

Q values is updated every step.

When a learner takes action $a_t$ in state $s_t$ and gets reward $r$, the Q value is updated as follows.

$$Q(s_t, a_t) = Q(s_t, a_t) + \alpha \left[ r + \gamma \max_a Q(s_{t+1}, a) - Q(s_t, a_t) \right]$$

$\alpha$: learning rate

target value

current value

error

$$Q(s_t, a_t) = r + \gamma \max_a \{ Q(s_{t+1}, a) \}$$
Temporal Difference (TD) learning

- Motivation: the mean of a sequence \( x_1, x_2, \ldots \) can be computed incrementally:

\[
\frac{1}{k} \sum_{i=1}^{k} x_i = \frac{1}{k} x_k + \frac{1}{k} \sum_{i=1}^{k-1} x_i
\]

\[
= \frac{1}{k} (x_k + (k - 1) \sum_{i=1}^{k-1} x_i) = \frac{1}{k} (x_k + (k - 1) Q_{k-1})
\]

- By analogy, temporal difference (TD) updates to \( Q(s, a) \) have the form

\[
Q_k(s, a) \rightarrow Q(s, a) + \left( Q_{\text{target}}(s, a) - Q(s, a) \right)
\]

Source: D. Silver
TD learning

• TD update:

\[ Q(s, a) \rightarrow Q(s, a) + (Q^{\text{target}}(s, a) - Q(s, a)) \]

• Suppose we have observed the transition \((s,a,s')\)

\[ Q^{\text{target}}(s, a) = R(s) + \max_{a'} Q(s', a') \]

• Full update equation:

\[ Q(s, a) = Q(s, a) + \alpha [r + \gamma \max_{a'} Q(s', a') - Q(s, a)] \]

“Updating a guess towards a guess”
TD algorithm outline

- At each time step $t$
  - From current state $s$, select an action a given exploration policy
  - Get the successor state $s'$
  - Perform the TD update:

$$Q(s, a) = Q(s, a) + \alpha \left[ r + \gamma \max_{a'} Q(s', a') - Q(s, a) \right]$$

Learning rate
Should start at 1 and decay over time

- e.g., $\alpha(t) = c/(c - 1 + t)$
Q learning algorithm

Initialize $Q(s, a)$ arbitrarily

Repeat (for each episode):
  initialize $s$

Repeat (for each step of episode):
  Choose $a$ from $s$ using policy derived from $Q$ (e.g., greedy)
  take action $a$, observe $r, s'$

\[
Q(s, a) = Q(s, a) + \alpha \left[ r + \gamma \max_{a'} Q(s', a') - Q(s, a) \right]
\]

$s \leftarrow s'$; until $s$ is terminal
Two Key Aspect in RL

• How we update the value function or policy?
  • How do we form training data
  • Sequence of \((s, a, r)\)….

• How we explore?
  • Exploit or Explore
  • Exploitation: Maximize its reward
  • Exploration: Maximize long-term well being.
Exploration vs. Exploitation
Two reasons to take an action in RL

- **Exploitation**: To try to get reward. We exploit our current knowledge to get a payoff.
- **Exploration**: Get more information about the world. How do we know if there is not a pot of gold around the corner?

To explore we typically need to take actions that do not seem best according to our current model.

Managing the trade-off between exploration and exploitation is a critical issue in RL.

Basic intuition behind most approaches:
- Explore more when knowledge is weak
- Exploit more as we gain knowledge
Monte Carlo Evaluation

- Want to estimate $U^\pi(s)$
  - expected return starting from $s$ and following $\pi$
  - estimate as average of observed returns in state $s$

- First few steps use Monte Carlo
  - average returns following the first few visits to state $s$

$$U^\pi(s) \approx \frac{2 + 1 - 5 + 4}{4} = 0.5$$
Exploration vs. Exploitation

**Exploration**: take a new action with unknown consequences
- **Pros:**
  - Get a more accurate model of the environment
  - Discover higher-reward states than the ones found so far
- **Cons:**
  - When you’re exploring, you’re not maximizing your utility
  - Something bad might happen

**Exploitation**: go with the best strategy found so far
- **Pros:**
  - Maximize reward as reflected in the current utility estimates
  - Avoid bad stuff
- **Cons:**
  - Might also prevent you from discovering the true optimal strategy
Exploration Strategies

- **Idea:** explore more in the beginning, become more and more greedy over time

- **$\epsilon$-greedy:** with probability $1-\epsilon$, follow the greedy policy, with probability $\epsilon$, take random action
  - Possibly decrease $\epsilon$ over time

- More complex **exploration functions** to bias towards less visited state-action pairs
  - E.g., keep track of how many times each state-action pair has been seen, return over-optimistic utility estimate if a given pair has not been seen enough times
Q learning + $\varepsilon$-greedy policy

Initialize $Q(s, a)$ arbitrarily
Repeat (for each episode):
  initialize $s$
Repeat (for step k of episode):
  pick a random number $x$ in $(0, 1)$
  If ($k < \text{threshold}$) or ($x < \varepsilon$)
    generate $a$ randomly
  else Choose $a$ from $s$ using policy derived from $Q$ (e.g., greedy)
  take action $a$, observe $r, s'$

$$Q(s, a) = Q(s, a) + \alpha [r + \gamma \max_a Q(s', a') - Q(s, a)]$$

$s \leftarrow s'$;
until $s$ is terminal
SARSA = State–Action–Reward–State–Action

• In TD Q-learning, we’re learning about the optimal policy while following the exploration policy

\[ Q(s, a) \leftarrow Q(s, a) + \alpha \left( R(s) + \gamma \max_{a'} Q(s', a') - Q(s, a) \right) \]

• Alternative (SARSA): also select action \( a' \) according to exploration policy

\[ Q(s, a) \leftarrow Q(s, a) + \left( R(s) + Q(s', a') - Q(s, a) \right) \]

• **SARSA vs. Q-learning example**
Q learning + SARSA

Initialize \( Q(s, a) \) arbitrarily

Repeat (for each episode):
  initialize \( s \)

Repeat (for step k of episode):
  pick a random number \( x \) in \((0, 1)\)
  If \((k < \text{threshold}) \) or \((x < \varepsilon)\)
    generate \( a \) randomly
  else Choose \( a \) from \( s \) using policy derived by \( Q \)
  take action \( a \), observe \( r, s' \)
  generate \( a' \) randomly

\[
Q(s, a) \leftarrow Q(s, a) + (R(s) + Q(s', a') - Q(s, a))
\]

\( s \leftarrow s' \); 

until \( s \) is terminal
Boltzmann Exploration in SARSA

• Boltzmann Exploration
  • Select action $a$ with probability (softmax on $Q(s,a)/T$),

$$\Pr(a \mid s) = \frac{\exp\left(\frac{Q(s, a)}{T}\right)}{\sum_{a' \in A} \exp\left(\frac{Q(s, a')}{T}\right)}$$

• $T$ is the temperature. Large $T$ means that each action has about the same probability. Small $T$ leads to more greedy behavior.
• Typically start with large $T$ and decrease with time
Case study: Backgammon

- **Rules**
  - 30 pieces, 24 locations
  - roll 2, 5: move 2, 5
  - hitting, blocking
  - branching factor: 400

- **Implementation**
  - use neural nets
  - 4 binary features for each position on board (# white pieces)

- **Results**
  - Gammon 0.0: trained against itself (300,000 games)
    - lot of expert input, hand-crafted features
  - Gammon 1.0: add special features
  - Gammon 2 and 3 (2-ply and 3-ply search)
    - 1.5M games, beat human champion
Function approximation

- For non-neural network implementation of Q-Learning, we assume a \textit{lookup table} representation for utility function $U(s)$ or action-utility function $Q(s,a)$
- This does not work if the state space is really large or continuous.
- Neural Network idea: Approximate the utilities or $Q$ values using parametric functions and automatically learn the parameters:

$$V(s) \rightarrow \hat{V}(s;w)$$
$$Q(s,a) \rightarrow \hat{Q}(s,a;w)$$

$w$ are the weights to be learned
Q learning in NN

• Train a neural network to output Q values:

Source: D. Silver
Q learning in NN

- Regular TD update: “nudge” $Q(s, a)$ towards the target

$$Q(s, a) \leftarrow Q(s, a) + \alpha \left( R(s) + \gamma \max_{a'} Q(s', a') - Q(s, a) \right)$$

- Encourage estimate to match the target by minimizing squared error:

$$L(w) = (R(s) + \max_{a'} Q(s', a'; w) - Q(s, a; w))^2$$

$\textbf{target}$  $\textbf{estimate}$
Q learning in NN

- Regular TD update: “nudge” $Q(s,a)$ towards the target

$$Q(s,a) \leftarrow Q(s,a) + \alpha \left( R(s) + \gamma \max_{a'} Q(s', a') - Q(s,a) \right)$$

- Deep Q learning: encourage estimate to match the target by minimizing squared error:

$$L(w) = (R(s) + \max_{a'} Q(s', a'; w) - Q(s, a; w))^2$$

- Compare to supervised learning:

$$L(w) = (y - f(x; w))^2$$

- Key difference: the target in Q learning is also moving!
Online Q learning algorithm

- Observe experience \((s, a, s')\)
- Compute target \(y = R(s) + \max_a Q(s', a; w)\)
- Update weights to reduce the error
  \[
  L = (y - Q(s, a; w))^2
  \]
- Gradient:
  \[
  \nabla_w L = (Q(s, a; w) - y) \nabla_w Q(s, a; w)
  \]
- Weight update:
  \[
  w \leftarrow w - \nabla_w L
  \]
- This is called \textit{stochastic gradient descent} (SGD)
Dealing with training instability

• Challenges
  • Target values are not fixed
  • Successive experiences are correlated and dependent on the policy
  • Policy may change rapidly with slight changes to parameters, leading to drastic change in data distribution

• Solutions
  • Freeze target Q network
  • Use *experience replay*

Mnih et al. [Human-level control through deep reinforcement learning](https://www.nature.com/articles/nature14236), *Nature* 2015
Experience replay

- At each time step:
  - Take action $a_t$ according to epsilon-greedy policy
  - Store experience $(s_t, a_t, r_{t+1}, s_{t+1})$ in *replay memory buffer*
  - Randomly sample *mini-batch* of experiences from the buffer

Mnih et al. *Human-level control through deep reinforcement learning*, *Nature* 2015
Experience replay

- At each time step:
  - Take action $a_t$ according to epsilon-greedy policy
  - Store experience $(s_t, a_t, r_{t+1}, s_{t+1})$ in *replay memory buffer*
  - Randomly sample *mini-batch* of experiences from the buffer
  - Perform update to reduce objective function

\[
E_{s,a,s'} \left( R(s) + \max_{a'} Q(s', a'; w^-) - Q(s, a; w) \right)^2
\]

Keep parameters of *target network* fixed, update every once in a while

Mnih et al. [Human-level control through deep reinforcement learning](https://www.nature.com/nature/journal/v518/n7540/full/nature14236.html), Nature 2015
Q learning in Atari

Mnih et al. Human-level control through deep reinforcement learning, Nature 2015
Q learning in Atari

- End-to-end learning of Q(s,a) from pixels s
- Output is Q(s,a) for 18 joystick/button configurations
- Reward is change in score for that step

Mnih et al. Human-level control through deep reinforcement learning, Nature 2015
Q learning in Atari

- Input state $s$ is stack of raw pixels from last 4 frames
- Network architecture and hyperparameters fixed for all games

A composed state is a segment of states of fix length.
E.g., Let the length be 4, then the composed states are

- $S_0 = [s_0, s_1, s_2, s_3]$
- $S_1 = [s_1, s_2, s_3, s_4]$
- ...  
- $S_{k-3} = [s_{k-3}, s_{k-2}, s_{k-1}, s_k]$

Composed states provide a sense of movement in a round of plays, just like filters in convolutional layer provide some features of an image.

Almost all NN use composed states for training.
def preprocess_frames(frames):
    if frames.shape[0] < 4:
        # single frame
        x_t = frames[0].astype("float")
        xt_list = [x_t, x_t, x_t, x_t]  # duplicate x_t 4 times.
    else:
        # 4 frames
        xt_list = np.asarray(frames)
        xt_list = xt_list.astype("float")

    s_t = np.array(xt_list).T / 80.0  # s_t.shape = (3, 4)
    s_t = np.expand_dims(s_t, axis=2)  # s_t.shape = (3, 4, 1), channel = 1
    s_t = np.expand_dims(s_t, axis=0)  # s_t.shape = (1, 3, 4, 1), batch_size = 1
    return s_t  # state at time t

Using 4 consecutive frames as one composed state.
# build the model
model = Sequential()
model.add(Conv2D(32, kernel_size=2, strides=1,
                kernel_initializer="normal", padding="same", input_shape=(3, 4, 1)))
model.add(Activation("relu"))
model.add(Conv2D(64, kernel_size=2, strides=1,
                kernel_initializer="normal", padding="same"))
model.add(Activation("relu"))
model.add(Conv2D(64, kernel_size=2, strides=1,
                kernel_initializer="normal", padding="same"))
model.add(Activation("relu"))
model.add(Flatten())
model.add(Dense(512, kernel_initializer="normal"))
model.add(Activation("relu"))
model.add(Dense(3, kernel_initializer="normal"))
# output shape = (batch_size, 3)
model.compile(optimizer=Adam(lr=1e-6), loss="mse")
# train network

game = paddle_game.PaddleGame()
experience = collections.deque(maxlen=MEMORY_SIZE)
num_games, num_wins = 0, 0
epsilon = initial_epsilon  # Initial value for time-decay \( \varepsilon \)-greedy

for e in range(NUM_EPOCHS):
    game.reset()  # new game
    # get first state
    loss, a_0 = 0.0, 1  # 0 = left, 1 = still, 2 = right
    x_t, r_0, game_over = game.step(a_0)
    s_t = preprocess_frames(x_t)

    while not game_over:
        s_tm1 = s_t  # s_tm1 == s_{t-1}
        # next action, using \( \varepsilon \)-greedy
        if e <= num_epochs_observe or np.random.rand() <= epsilon:
            a_t = np.random.randint(low=0, high=NUM_ACTIONS, size=1)[0]
        else:
            q = model.predict(s_t)[0]  # rewards of the first in a batch
            a_t = np.argmax(q)  # action for the max reward.
# apply action, get reward
x_t, r_t, game_over = game.step(a_t)
s_t = preprocess_frames(x_t)

# if reward, increment num_wins
if r_t == 1: num_wins += 1

# store experience
experience.append((s_tm1, a_t, r_t, s_t, game_over))

if e > num_epochs_observe:
    # finished observing, now start training, get next batch
    X, Y = get_next_batch(experience, model, NUM_ACTIONS, GAMMA, BATCH_SIZE)
    loss += model.train_on_batch(X, Y)

    # time-decay: reduce epsilon gradually
    if epsilon > final_epsilon:
        epsilon -= (initial_epsilon - final_epsilon) / NUM_EPOCHS
def get_next_batch(experience, model, num_actions, gamma, batch_size):
    batch_indices = np.random.randint(low=0, high=len(experience),
                                       size=batch_size)
    batch = [experience[i] for i in batch_indices]
    # batch is a list of experiences: [s_t, a_t, r_t, s_t2, game_over]
    # where [s_t, a_t, _] → [s_t2, _, r_t]
    X = np.zeros((batch_size, 12))  # X is a batch_size of frames.
    Y = np.zeros((batch_size, num_actions))
    # Y is a batch size of expected rewards (for each action).

    for i in range(len(batch)):
        s_t, a_t, r_t, s_tp1, game_over = batch[i]  # s_tp1 = s_{t+1}
        X[i] = s_t  # current state
        Y[i] = model.predict(s_t)[0]  # predicted rewards for each action
        Q_sa = np.max(model.predict(s_tp1)[0])  # max reward for the next state
        if game_over:
            Y[i, a_t] = r_t  # expected reward = final rewards
        else:
            Y[i, a_t] = r_t + gamma * Q_sa  # expected reward increased

    return X, Y
• Learning the policy directly can be much simpler than learning Q values
• We can train a neural network to output *stochastic policies*, or probabilities of taking each action in a given state
• *Softmax* policy (also called *Rollout Policy*)

\[
(s, a; u) = \frac{\exp(f(s, a; u))}{\sum_{a'} \exp(f(s, a'; u))}
\]
Actor-critic algorithm

- Define objective function as total discounted reward:
  \[ J(u) = \mathbb{E} \left[ R_1 + R_2 + \gamma R_3 + ... \right] \]

- The gradient for a stochastic policy is given by
  \[ \nabla_u J = \mathbb{E} \left[ \nabla_u \log p(s, a; u) Q_p(s, a; w) \right] \]

- Actor network update: \( u \leftarrow u + \nabla_u J \)
- Critic network update: use Q learning (following actor’s policy)
Advantage actor-critic

- The raw Q value is less meaningful than whether the reward is better or worse than what you expect to get.
- Introduce an *advantage function* that subtracts a baseline number from all Q values:
  \[ A(s, a) = Q(s, a) - V(s) \]
- Estimate \( V \) using a *value network*.
- Advantage actor-critic:
  \[ \nabla_u J = \mathbf{E} \left[ \nabla_u \log (s, a; u) A(s, a; w) \right] \]
Asynchronous advantage actor-critic

Asynchronously update global parameters

Passive learning v.s. Active learning

- Passive learning
  - The learner watches the world going by and tries to learn the utilities of being in various states
  - Supervised learning is passive: Learner doesn’t affect the distribution of exemplars or the class labels

- Active learning
  - The learner not simply watches, but also acts
How to learn model in Passive RL?

• Use the transition tuple \((s, a, s', r)\) to learn \(P(s' | a, s')\) and \(Q(s, a)\). That’s supervised learning!
• Since the learner can get every transition \((s, a, s', r)\) directly, so take \((s, a)/s'\) as an input/output example of the transition probability function \(T\).
• Different techniques in the supervised learning can be used.
Category of Reinforcement Learning

- Model-based RL
  - Constructs domain transition model, MDP
  - Learn the policy and MDP in tandems

- Model-free RL
  - Only concerns policy
    - Q-Learning

- Active Learning (Off-Policy Learning)
  - Choose action with highest utility value
  - Q-Learning: learn a good policy (ideally optimal) while acting in uncertain world

- Passive Learning (On-Policy learning)
  - Simply follow the policy for many epochs
  - We just try to learn how good the policy is
Applications of reinforcement learning

- Backgammon

Applications of reinforcement learning

- AlphaGo

https://deepmind.com/research/alphago/
Applications of reinforcement learning

- Learning a fast gait for Aibos

Policy Gradient Reinforcement Learning for Fast Quadrupedal Locomotion
Nate Kohl and Peter Stone.
Applications of reinforcement learning

- Stanford autonomous helicopter

Pieter Abbeel et al.
Applications of reinforcement learning

- Playing Atari with deep reinforcement learning

Video

V. Mnih et al., *Nature*, February 2015
Applications of reinforcement learning

- **End-to-end training of deep visuomotor policies**

Fig. 1: Our method learns visuomotor policies that directly use camera image observations (left) to set motor torques on a PR2 robot (right).

[Video](#)

Sergey Levine et al., Berkeley
Applications of reinforcement learning

- Object detection

Sequence of attended regions to localize the object

States

Actions

Steps

$t_1$ ...

$t_i$

$t_{i+1}$ ...

$t_{n-1}$

$t_n$


Video
OpenAI Gym

A toolkit for developing and comparing reinforcement learning algorithms. It supports teaching agents everything from walking to playing games like Pong or Go.

Read the launch blog post ›
View documentation ›
View on GitHub ›

https://gym.openai.com/
RL Summary

• Wondering around the world provides
  • Training Data
• Each episode is a sample of the current policy
  • Next lecture is about using Sampling in Planning and Learning
• Sampling in RL is unique, the policy is stochastic and still provides a guarantee to convergence
• Where to sample is very important!
  • Exploration vs. Exploitation