APPLIED REINFORCEMENT LEARNING METHODS FOR THE CAPACITATED VEHICLE ROUTING PROBLEM

Hoang Tran Nhat Minh

Instructed by Dr. Pham Quang Dung

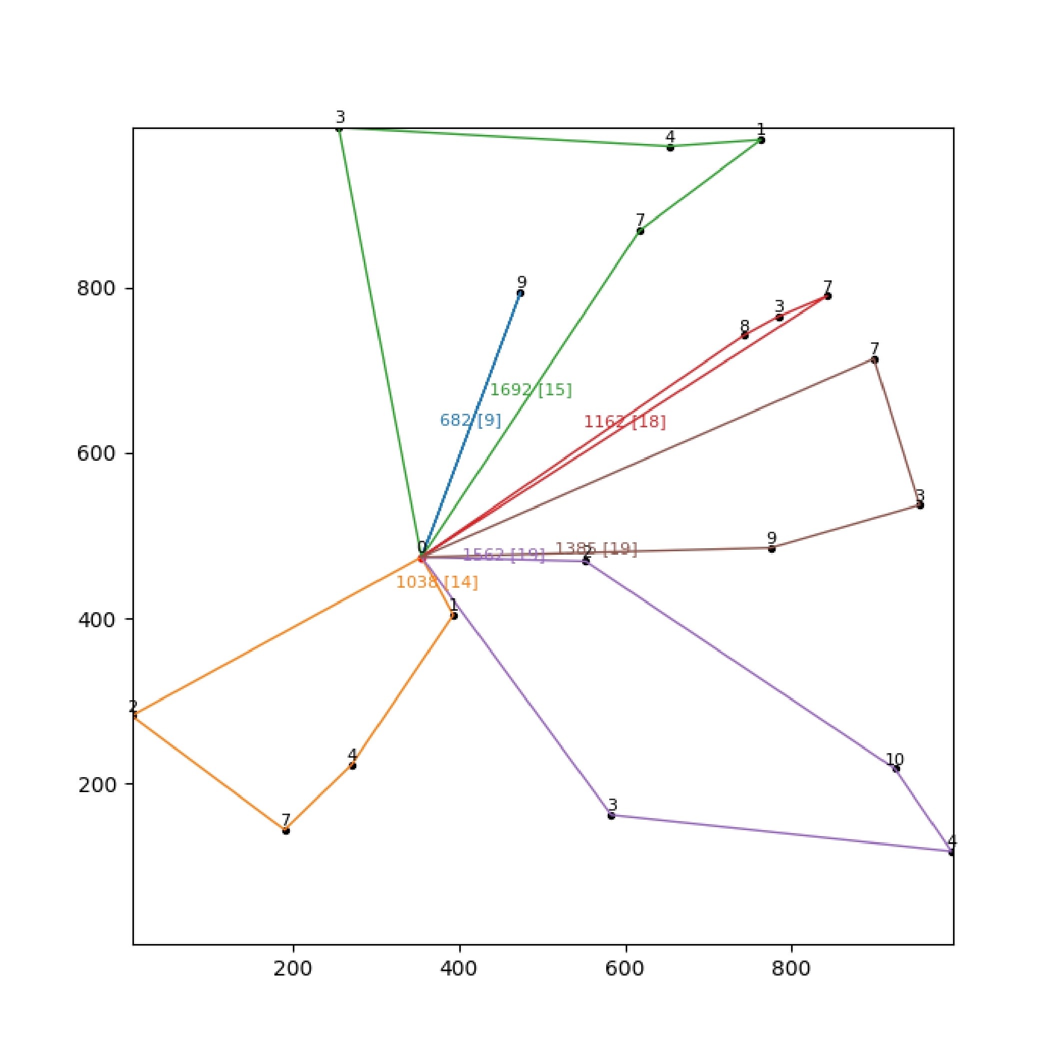
Data Science & Artificial Intelligence 2020

January 2024

OVERVIEW

This project reviews reinforcement learning (RL) approaches to optimization problems in general, then dive deeper for the Capacitated Vehicle Routing Problem (CVRP). The problem is formulated as a RL problem, then several RL methods are implemented as an endeavor to solve it, in comparison with the solutions provided by OR-Tools, namely:

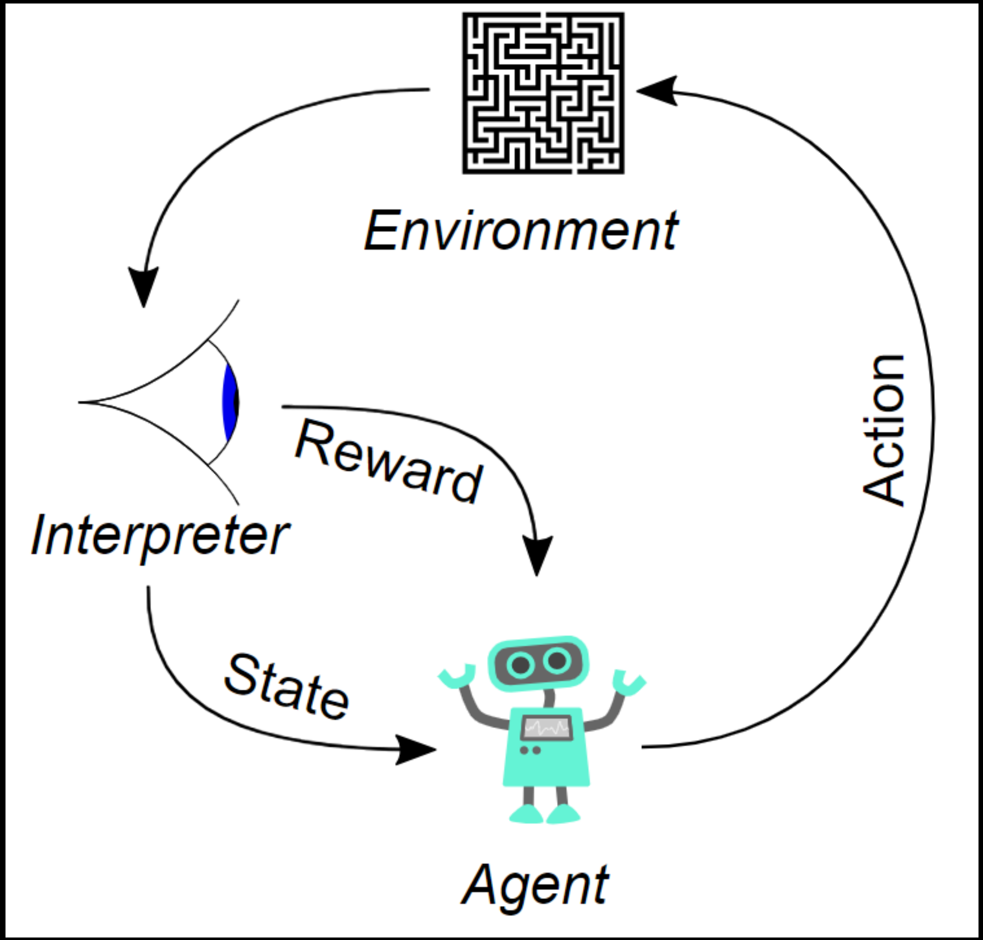
- Deep Q-Network (DQN)
- Advantage Actor-Critic (A2C)
- Proximal Policy Optimization (PPO)



WHAT IS REINFORCEMENT LEARNING (RL)?

Reinforcement Learning (RL) is an area of machine learning, alongside *supervised* and *unsupervised learning*.

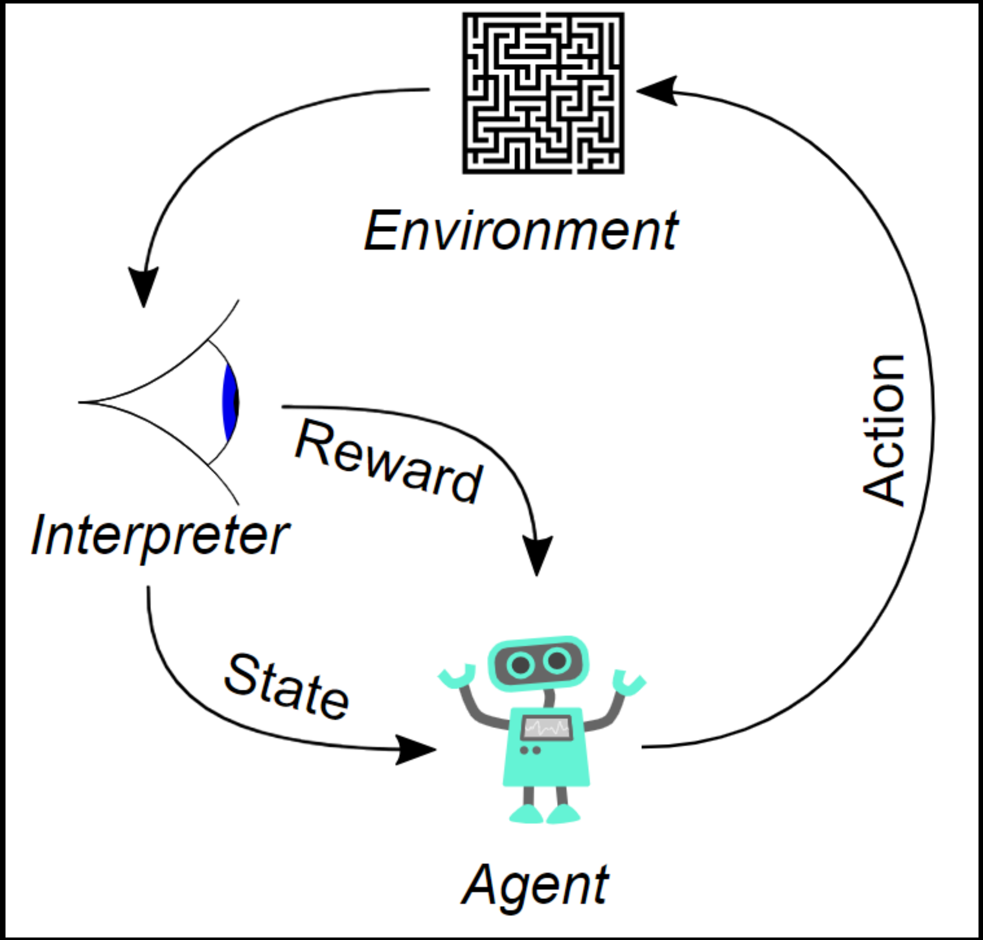
RL aims to maximize the *cumulative reward* when an *intelligent agent* takes *actions* in a *dynamic environment*.



WHY RL FOR OPTIMIZATION PROBLEMS?

First, it must be appliable:

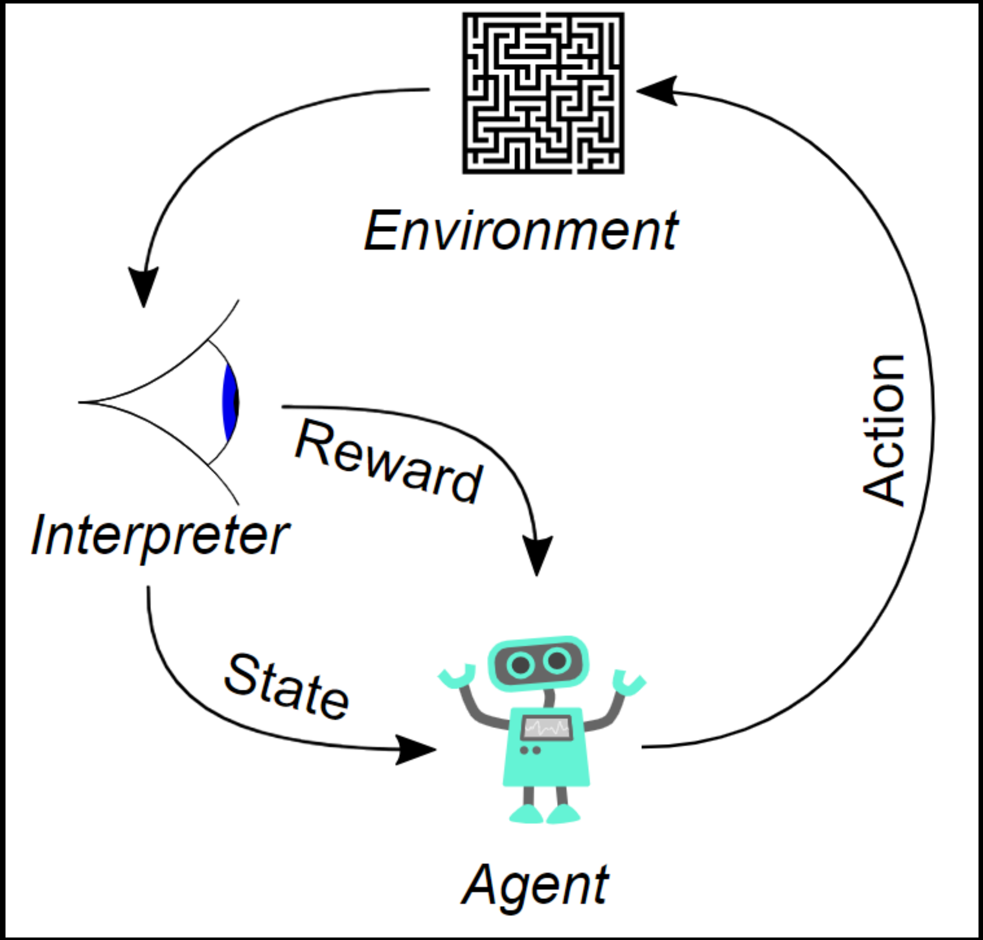
• Environment	= Problem
• (Long-term) Reward	= Objective
• State	= Configuration
• Agent	= Algorithm
• Action	= Decision



WHY RL FOR OPTIMIZATION PROBLEMS?

RL is well-fit for optimization problems…

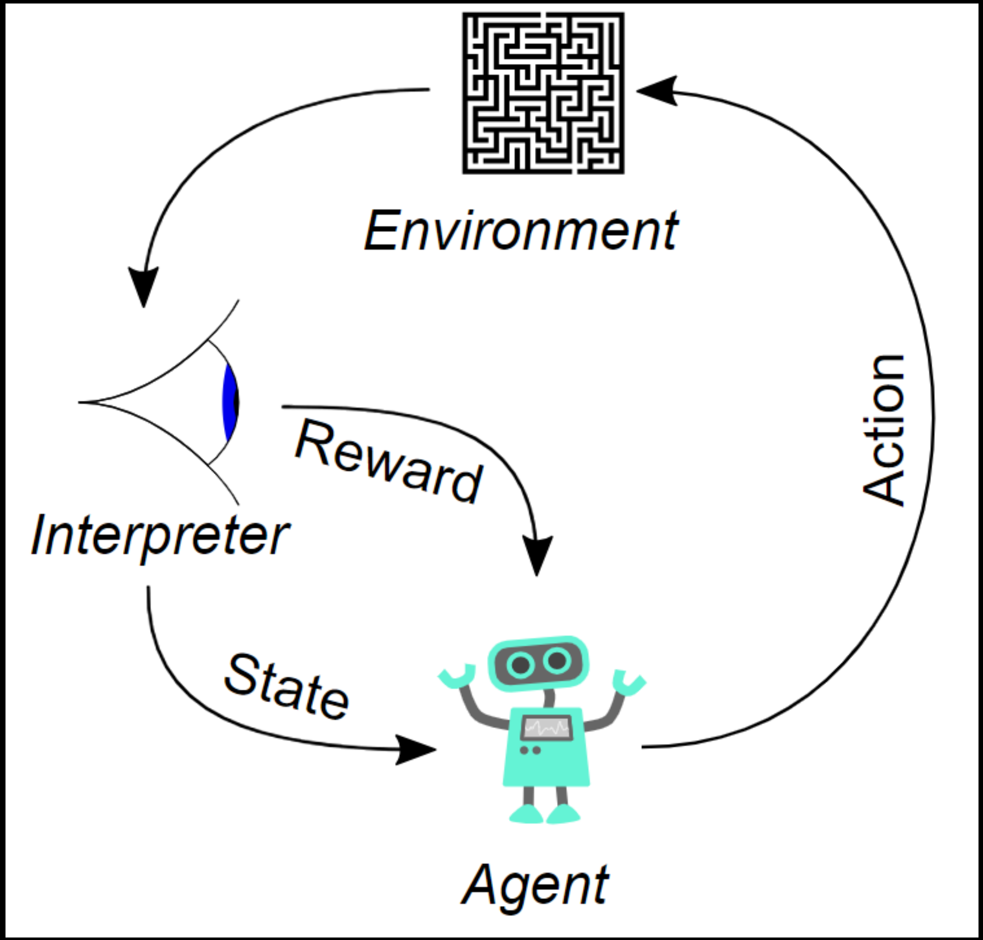
- Sequential decision making: RL can learn a series of sequential decisions to maximize a long-term objective.
- Exploration and Exploitation: RL can balance exploration and exploitation.
- Complex problems: RL can handle very complex problems provided enough resources.
- Delayed rewards: RL algorithms can use its experience to optimize the cumulative reward based on immediate rewards and their past experience.
- Transferability: Trained agent can be directly used on the same problem with different configurations.



RL APPROACHES FOR OPTIMIZATION PROBLEMS

There are many ways to approach optimization problems using RL, the following three are used in this project:

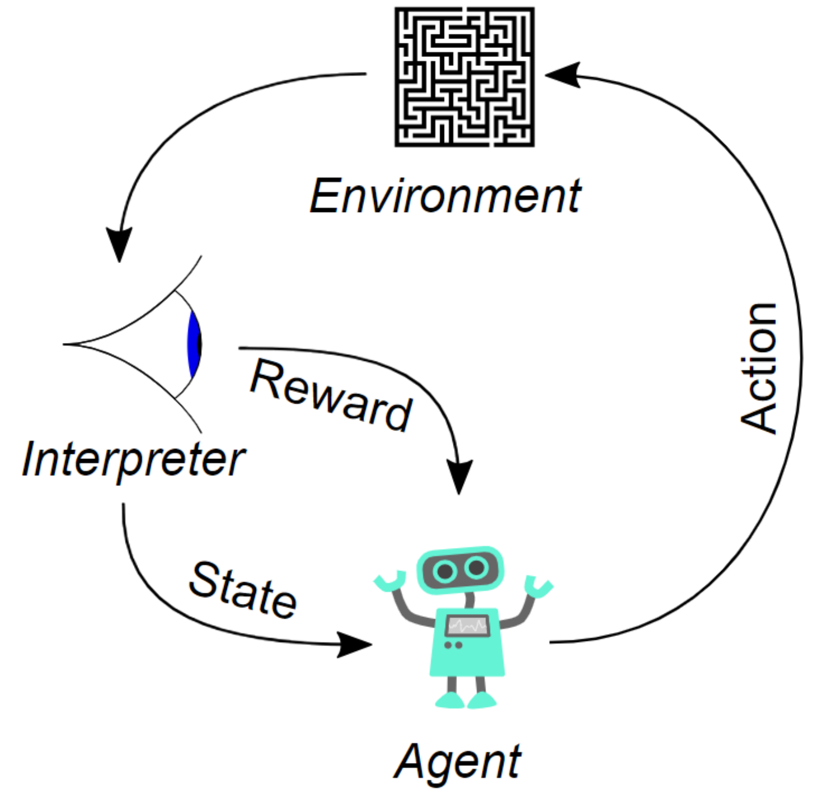
- Deep Q-Network (DQN)
- Advantage Actor-Critic (A2C)
- Proximal Policy Optimization (PPO)



TWO GENERAL TYPES OF RL ALGORITHMS

Value-based: Approximate the optimal value function…

- Which is one of the two mappings:
    - $state \rightarrow cumulative\_reward$
    - $(state,\ action) \rightarrow cumulative\_reward$
  - for all $action\ \epsilon\ action\_space$ in all $state\ \epsilon\ state\_space$
- If optimized: Higher cumulative reward = Better
- Example: Q-learning (learn the latter mapping)
  - The $cumulative\_reward$ in this case is often denoted as Q−value: Q
- Advantages: Sample efficient, steady.



Policy-based: Approximate the optimal policy function…

- Which is the mapping:
    - $state \rightarrow P(action\,|\,state)$
    - $P(action\,|\,state)$ is often denoted as π
  - for all $action\ \epsilon\ action\_space$ in current $state$
- If optimized: Higher action probability = (Probably) better action
- Example: REINFORCE
- Advantages: Better for continuous spaces, converges faster.

Q-LEARNING

Value-based: $(state,\ action) \rightarrow Q$

- for all $action \in action\_space$ in all $state \in state\_space$

Pseudocode:

```
Q_table = random_values(Q_table.shape);        // Initialize random values
state = initial_state;                          // Initialize state
while true {
        action = choose(Q_table, state, action_space);
                                                // Choose an action based on a kind of policy
        next_state, reward = execute(state, action);
        Q_table[state, action] = Q_table[state, action] + alpha * (
                reward + gamma * max(Q[next_state, action_space]) – Q_table[state, action]
        );                                      // Update Q-table
        if final(state) {state = initial_state}
                else {state = next_state};      // Continue to the next state
        if stop_training {break};               // Break based on a stopping condition
}
```

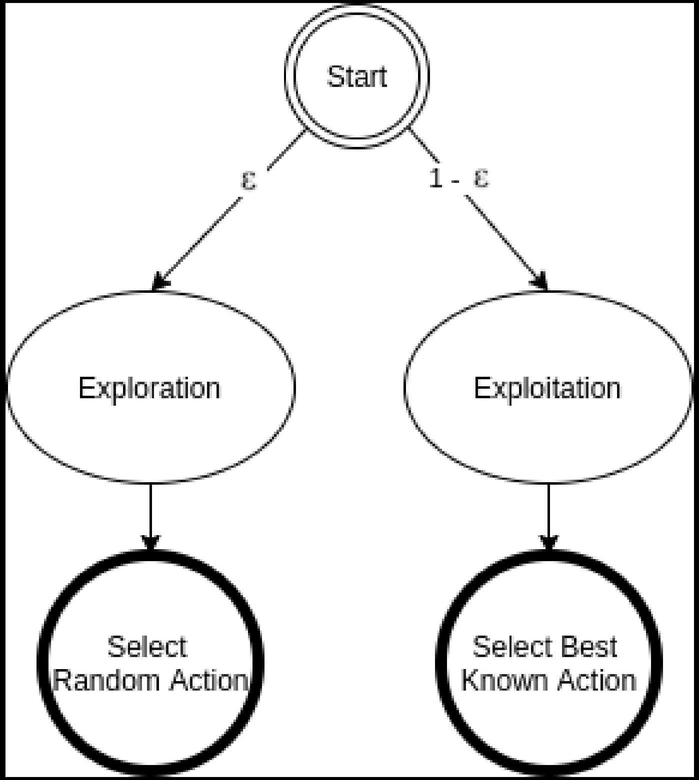
EPSILON-GREEDY "POLICY"

This "policy" is simple enough for Q-learning to still be considered a policy-free algorithm.

```
action = choose(Q_table, state, action_space);
```

At state s, for a constant $\varepsilon \epsilon [0, 1]$

- ε chance: Select a completely random action
- $1 - \varepsilon$ chance: Select the best known action
  - Or, the action $a = \underset{a}{\operatorname{argmax}}\, Q(s, a)$



UPDATE Q-TABLE USING BELLMAN EQUATION

Update Q-table:

```
Q_table[state, action] = Q_table[state, action] + alpha * (
      reward + gamma * max(Q[next_state, action_space]) - Q_table[state, action]
);
```

Bellman equation:

$$Q(s,a) \leftarrow Q(s,a) + \alpha(r + \gamma \max(Q(s',a')) - Q(s,a))$$

- $Q(s,a)$: Q-table value at state s and action a
- α : learning rate
- r : immediate reward (after executing action a on state s)
- $\gamma \epsilon [0,1]$: discount factor
- $\max(Q(s',a'))$: maximum Q-value of the next state s' (after executing all actions available in $a' = action\_space$)

DISCOUNT FACTOR

$$Q(s,a) \leftarrow Q(s,a) + \alpha(r + \gamma \max(Q(s',a')) - Q(s,a))$$

$Q(s,a)$ converges to $r + \gamma \max(Q(s',a'))$

The discount factor $\gamma \epsilon [0,1]$ is the importance of future reward:

- $\gamma = 0$ means
  - $Q(s,a)$ converges to r
  - Future reward is not considered
- $\gamma = 1$ means
  - $Q(s,a)$ converges to $r + \max(Q(s',a'))$
  - The future reward is as important as the immediate reward
- Why typically $0 < \gamma < 1$: This help many algorithms, including Q-learning, to converge properly and faster.

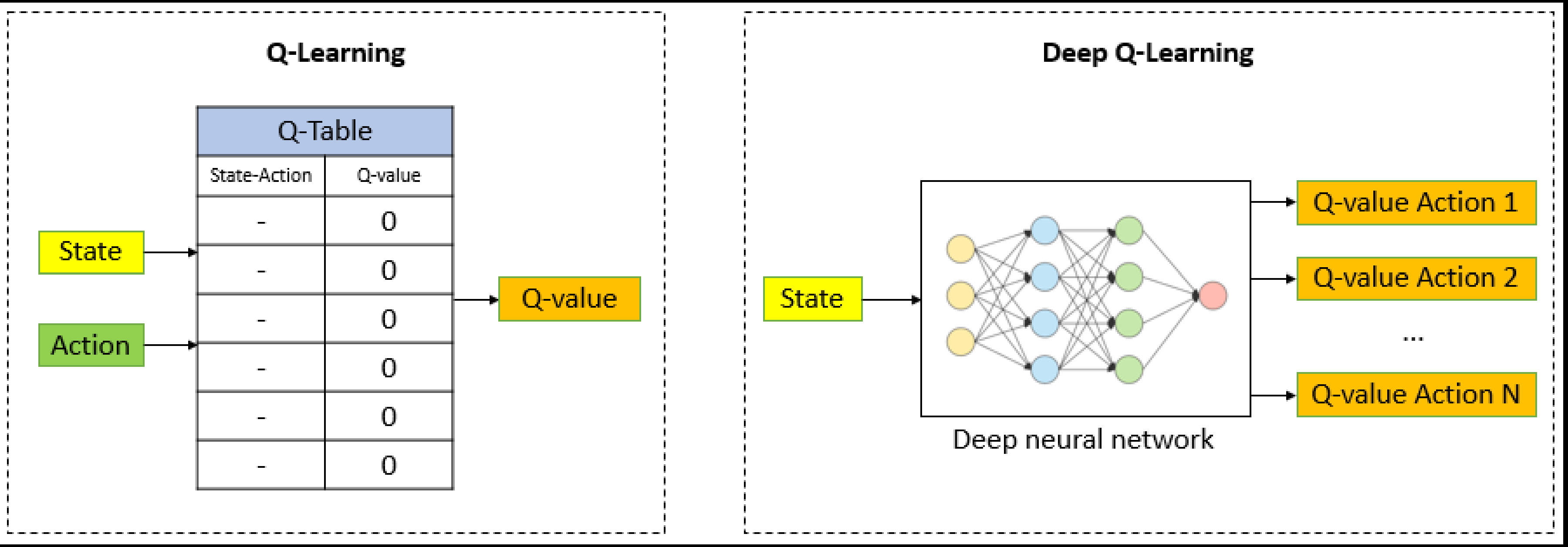
DEEP Q-NETWORK

What if state space and/or the action space grow larger?

- The size of Q-table grows exponentially with them

Storing the Q-values $Q(s,a)$ as a table has a main drawback: It is infeasible for almost any non-trivial problems.

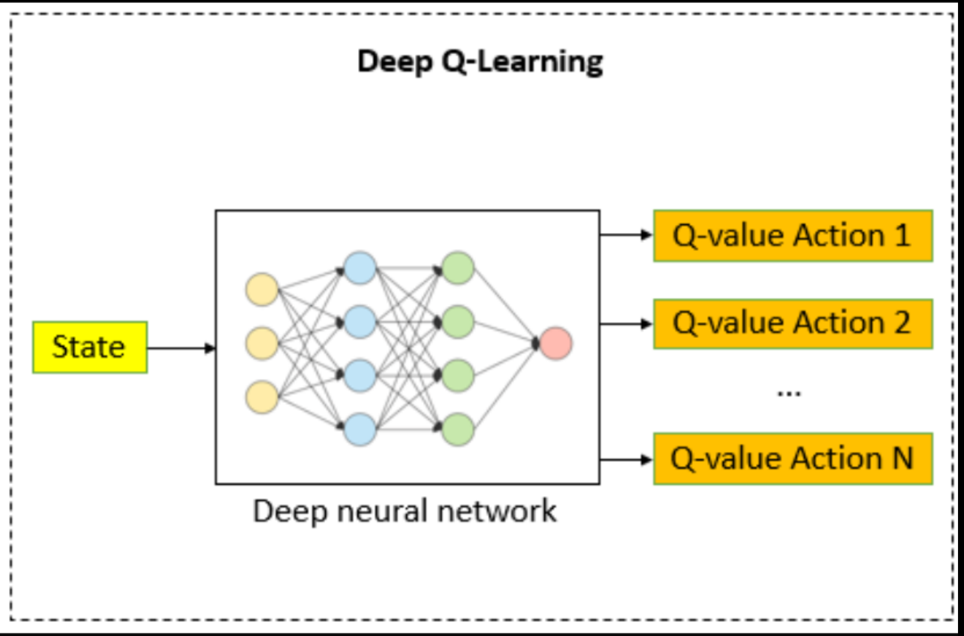
Deep Q-Network: the table instead will be "stored" as a deep neural network:



DEEP Q-LEARNING

Deep Q-Learning refers to a Q-Learning implementation using a Deep Q-Network.

- Replay memory: Each "experience" will be stored as a tuple
    - (state, action, reward, next state)
  - This dataset of many tuples will be sampled during the training process of networks
- Deep Q-Network: Learn the mapping $state \rightarrow Q(action \mid state)$ for each action in the action space (using the dataset)
- Target Network: Another network to estimate the target Q-values
  - A copy of the main network
  - But updated periodically to…
    - prevent overfitting
    - mitigate the effect of delayed reward
- Everything else is the same



ACTOR-CRITIC

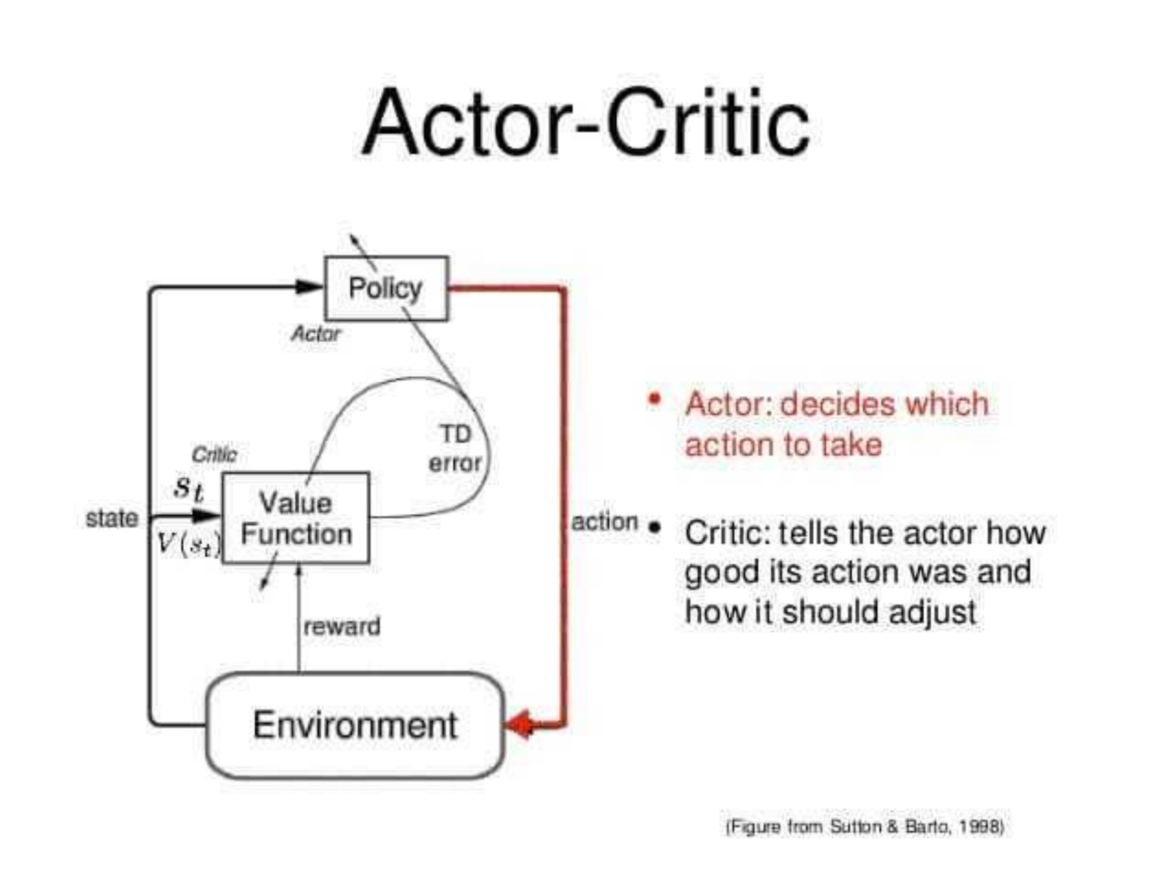
Combines the two types:

Actor: The policy-based part

- $state \rightarrow \pi$
- While running deterministically, it output the action with highest probability: Probably the best action

Critic: The value-based part

- $state \rightarrow cumulative\_reward$
    - Same idea as Q-value, but it does not consider any action
    - The $cumulative\_reward$ in this case is often denoted as just value: V
- Update the actor accordingly using policy gradient: Same idea as gradient descent



ADVANTAGE ACTOR-CRITIC (A2C)

Critic in Actor-Critic learns the mapping

$$state \rightarrow V$$

Advantage Actor-Critic (A2C) splits the Q-value into two parts, based on action a:

- State value $V(s)$
- Advantage value $A(s,\ a)$

$$Q(s,a) = V(s) + A(s,a)$$
$$\Rightarrow A(s,a) = Q(s,a) - V(s)$$

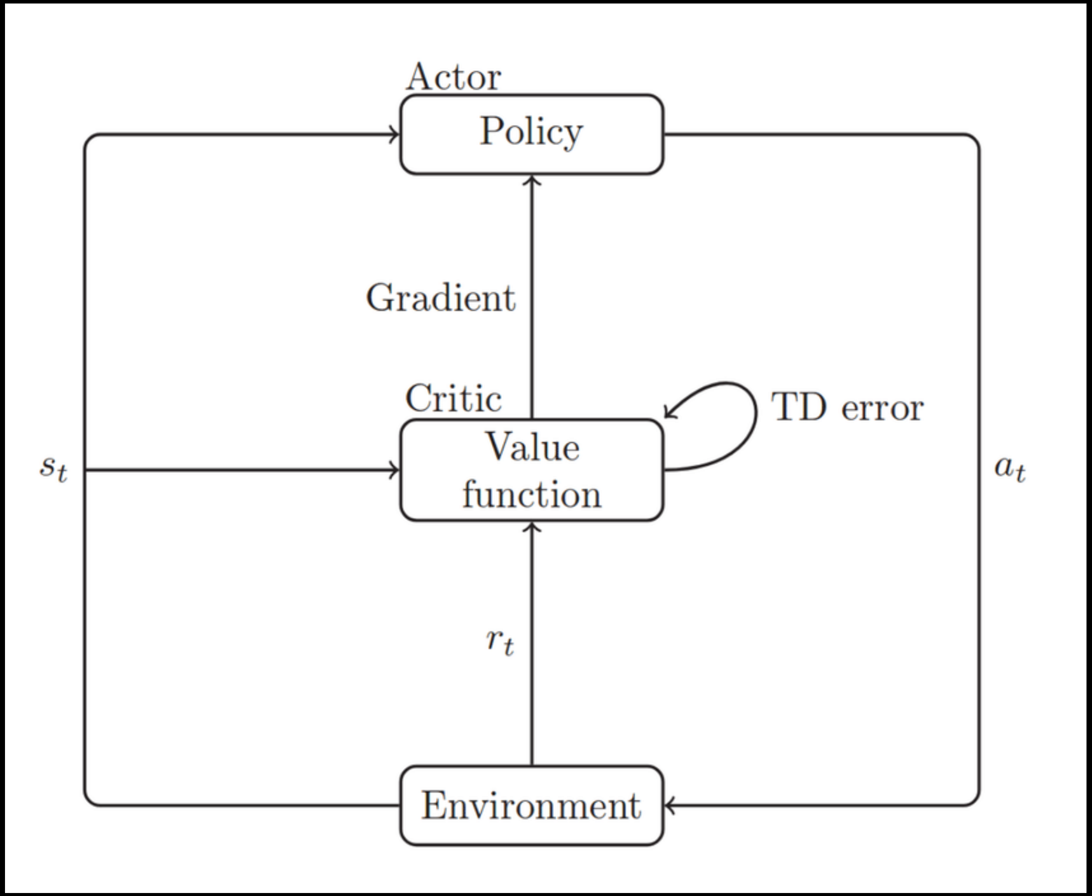
ADVANTAGE ACTOR-CRITIC (A2C)

Actor-Critic:

- Actor: $state \rightarrow P(action \mid state)$
- Critic: $state \rightarrow V$

Advantage Actor-Critic:

- Actor: $state \rightarrow P(action \mid state)$
- Critic: $(state, action) \rightarrow A$



Why Advantage value?

Instead of learning how "good" is a $state$, the critic instead learn how much *advantage* it will gain if the $action$ is executed on that $state$.

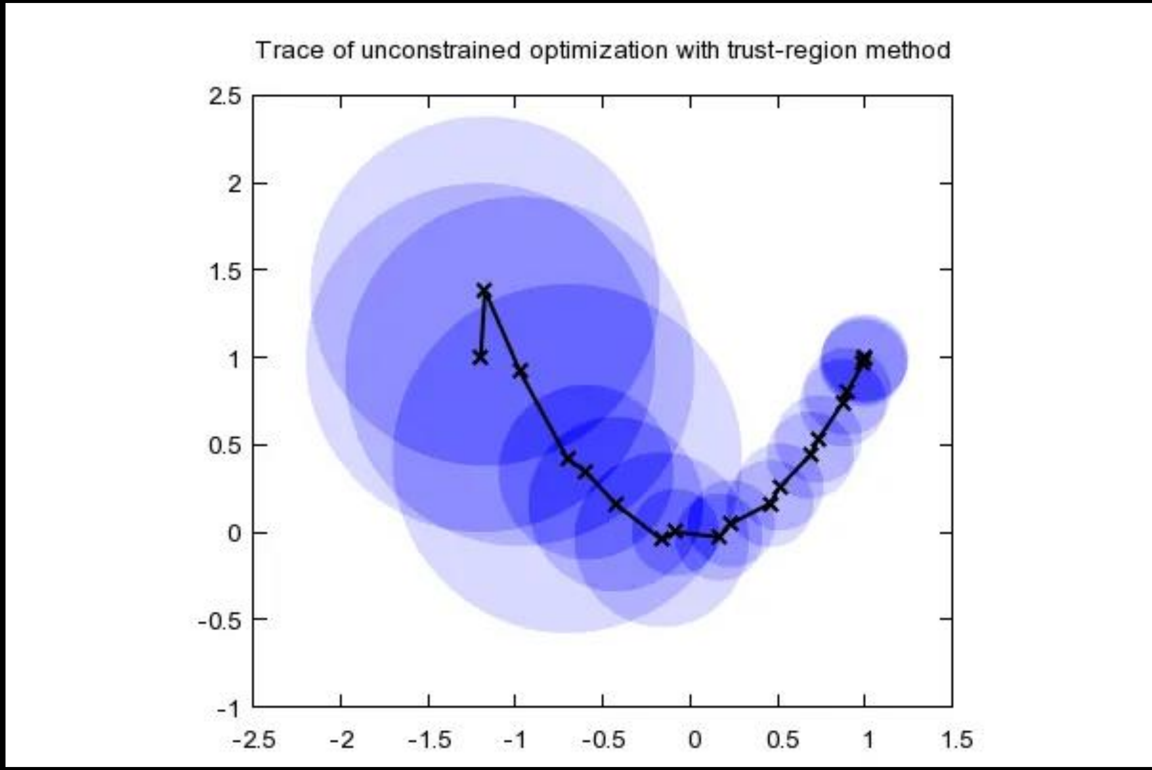
- Generalize better, especially on complex problems
- But obviously also cost more memory

PROXIMAL POLICY OPTIMIZATION (PPO)

Proximal Policy Optimization improves the learning progress of the actor in A2C using *Trust Region Policy Optimization (TRPO)*

TRPO is a technique to limit how large can the actor update its policy π

- Or *clipping*
- Avoid too large updates



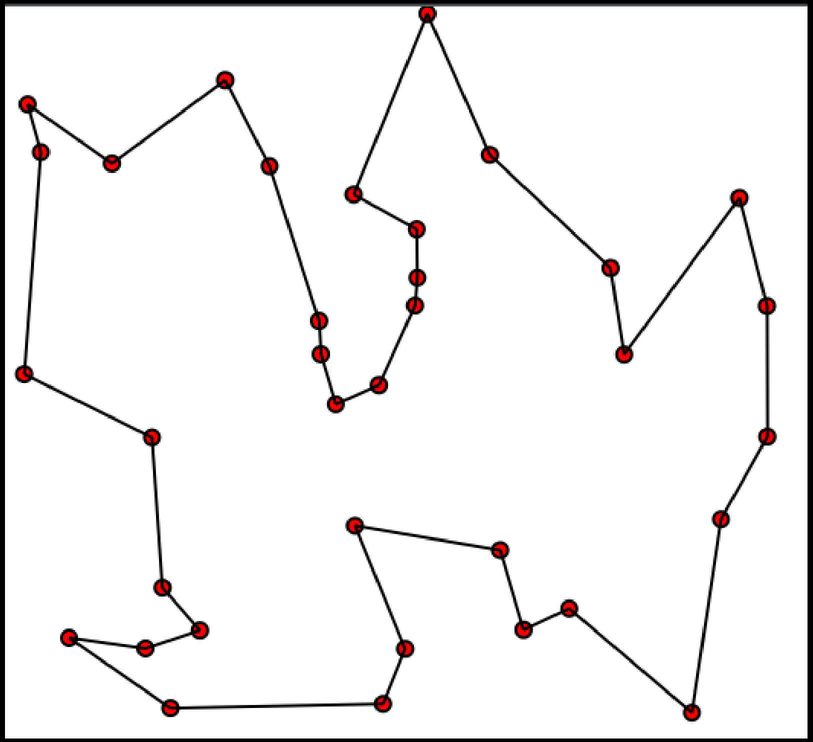
PROXIMAL POLICY OPTIMIZATION (PPO)

Since many of the mathematical details of TRPO are out of the scope of this project, here are some quick explanations:

- The constraint is expressed in terms of *Kullback–Leibler divergence*
    - Kullback–Leibler divergence: the "distance" between two probability distributions
    - In this case, the old policy and the new policy
- The theoretical TRPO update is impractical, so an approximation is used using the *Taylor expansion* of the objective function around the parameters of current π
    - The problem is then solved by *Lagrangian duality* to give the *Natural Policy Gradient*
    - Since the approximation will naturally have an error, sometimes really large...
- The Natural Policy Gradient is then tweaked by using a *backtracking coefficient* to mitigate the impact of the approximation error
    - Backtracking coefficient: how much the approximation should "relies" on some of its previous ones

TRAVELLING SALESMAN PROBLEM (TSP)

"What is the shortest possible route that visits each city exactly once and return to the origin city?"



Travelling Salesman Problem (TSP) is an optimization problem, and is classified as

- A combinatorial optimization problem
- An integer programming problem

CODE REVIEW OF A REINFORCEMENT LEARNING IMPLEMENTATION FOR TSP

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import os
import time
from tqdm import tqdm_notebook
from scipy.spatial.distance import cdist
import imageio
from matplotlib.patches import Rectangle
from matplotlib.collections import PatchCollection

plt.style.use("seaborn-dark")

import sys
sys.path.append("../")
from rl.agents.q_agent import QAgent
```

Some regular imports.

CODE REVIEW

```
class DeliveryEnvironment(object):
    def __init__(self,n_stops = 10,max_box = 10,method = "distance",**kwargs):

        print(f"Initialized Delivery Environment with {n_stops} random stops")
        print(f"Target metric for optimization is {method}")

        # Initialization
        self.n_stops = n_stops
        self.action_space = self.n_stops
        self.observation_space = self.n_stops
        self.max_box = max_box
        self.stops = []
        self.method = method

        # Generate stops
        self._generate_constraints(**kwargs)
        self._generate_stops()
        self._generate_q_values()
        self.render()

        # Initialize first point
        self.reset()
```

Initialize the environment with

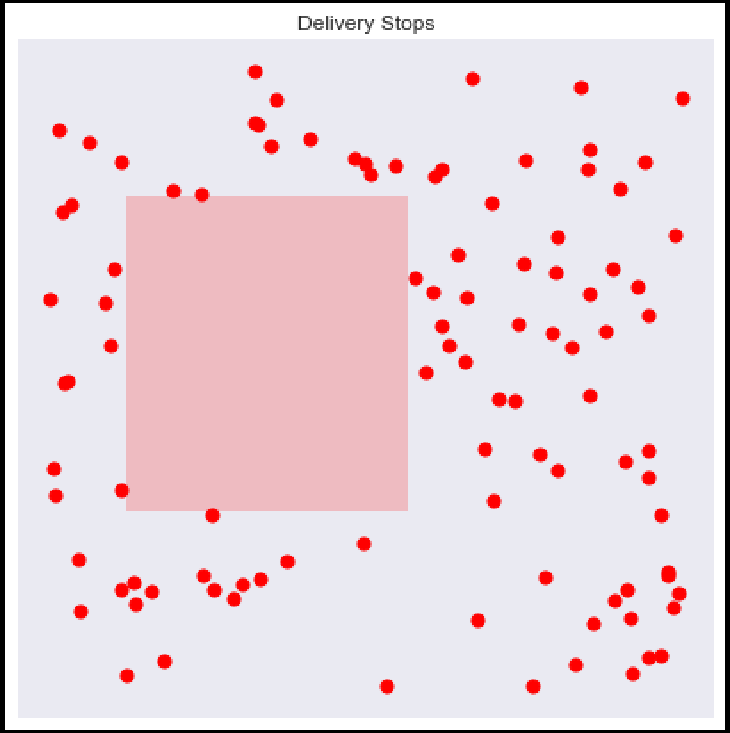
- `n_stops`: number of cities (or stops)
- `max_box`: maximum coordinate value of a city
- `method`: accepts three values
  - “`distance`”: regular 2D problem
  - “`traffic_box`”: add a random box of high traffic inside the environment, in which the salesman is slowed down
  - “`time`”: regular 2D problem but the reward is added by a random number

CODE REVIEW

```
def _generate_constraints(self,box_size = 0.2,traffic_intensity = 5):

    if self.method == "traffic_box":

        x_left = np.random.rand() * (self.max_box) * (1-box_size)
        y_bottom = np.random.rand() * (self.max_box) * (1-box_size)

        x_right = x_left + np.random.rand() * box_size * self.max_box
        y_top = y_bottom + np.random.rand() * box_size * self.max_box

        self.box = (x_left,x_right,y_bottom,y_top)
        self.traffic_intensity = traffic_intensity
```

Add a random box of high traffic inside the environment, in which the salesman is slowed down, with

- `box_size`: how large the box is (approximately)
- `traffic_intensity`: how much slower the salesman is when moving through it (approximately)



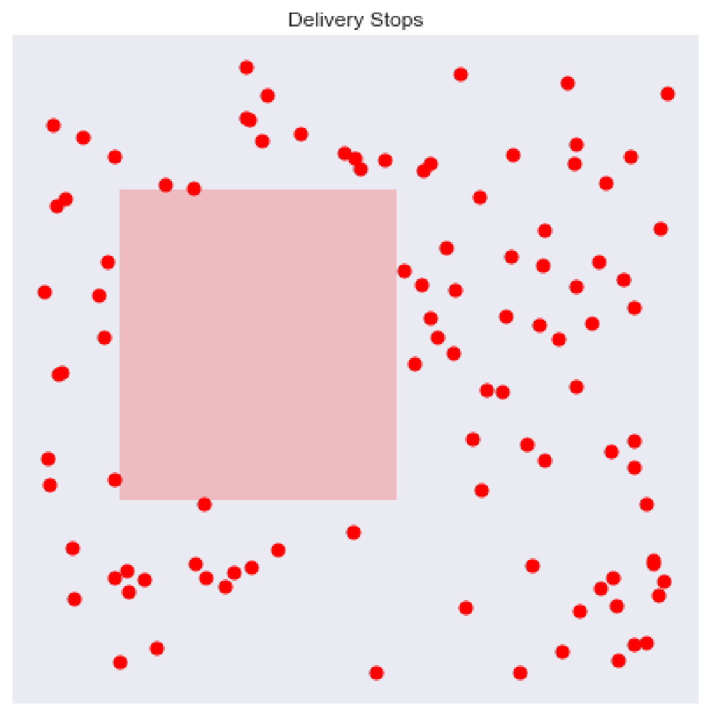
CODE REVIEW

```
def _generate_stops(self):

    if self.method == "traffic_box":

        points = []
        while len(points) < self.n_stops:
            x,y = np.random.rand(2)*self.max_box
            if not self._is_in_box(x,y,self.box):
                points.append((x,y))

        xy = np.array(points)

    else:
        # Generate geographical coordinates
        xy = np.random.rand(self.n_stops,2)*self.max_box

    self.x = xy[:,0]
    self.y = xy[:,1]
```

Randomly generate placement of cities.

If a traffic box is used, all cities are strictly outside of it.

```
def _is_in_box(self,x,y,box):
    # Get box coordinates
    x_left,x_right,y_bottom,y_top = box
    return x >= x_left and x <= x_right and y >= y_bottom and y <= y_top
```



CODE REVIEW

```python
def _generate_q_values(self,box_size = 0.2):

    # Generate actual Q Values corresponding to time elapsed between two points
    if self.method in ["distance","traffic_box"]:
        xy = np.column_stack([self.x,self.y])
        self.q_stops = cdist(xy,xy)
    elif self.method=="time":
        self.q_stops = np.random.rand(self.n_stops,self.n_stops)*self.max_box
        np.fill_diagonal(self.q_stops,0)
    else:
        raise Exception("Method not recognized")
```

Initialize Q-table

- `box_size` is improperly placed and not used

If the method is time, initialize Q-table by a random array where values are in `[0, max_box)`, except for `Q[i, i] = 0` for any `i`

- This is not a good starting point, the author should have used the same Q-table as below

Else, initialize `Q[m, n] = distance(m, n)`

CODE REVIEW

```
def render(self,return_img = False):
    # THE AUTHOR RENDER THE ENVIRONMENT IMAGE


def reset(self):

    # Stops placeholder
    self.stops = []

    # Random first stop
    first_stop = np.random.randint(self.n_stops)
    self.stops.append(first_stop)

    return first_stop
```

Reset the environment

`stops`: all the stops that the salesman have visited, in respective order. The original stop is randomized.

CODE REVIEW

```
def step(self,destination):

    # Get current state
    state = self._get_state()
    new_state = destination

    # Get reward for such a move
    reward = self._get_reward(state,new_state)

    # Append new_state to stops
    self.stops.append(destination)
    done = len(self.stops) == self.n_stops

    return new_state,reward,done


def _get_state(self):
    return self.stops[-1]
```

Makes a step towards `destination` and returns

- `new_state`: the new state
- `reward`: the reward given for going to `destination`
- `done`: if the episode is finished

The state is simply an integer which is the current stop.

`_get_xy()` returns the coordinate of the current stop.

```
def _get_xy(self,initial = False):
    state = self.stops[0] if initial else self._get_state()
    x = self.x[state]
    y = self.y[state]
    return x,y
```

CODE REVIEW

```python
def _get_reward(self,state,new_state):
    base_reward = self.q_stops[state,new_state]

    if self.method == "distance":
        return base_reward
    elif self.method == "time":
        return base_reward + np.random.randn()
    elif self.method == "traffic_box":

        # Additional reward correspond to slowing down in traffic
        xs,ys = self.x[state],self.y[state]
        xe,ye = self.x[new_state],self.y[new_state]
        intersections = self._calculate_box_intersection(xs,xe,ys,ye,self.box)
        if len(intersections) > 0:
            i1,i2 = intersections
            distance_traffic = np.sqrt((i2[1]-i1[1])**2 + (i2[0]-i1[0])**2)
            additional_reward = distance_traffic * self.traffic_intensity * np.random.rand()
        else:
            additional_reward = np.random.rand()

        return base_reward + additional_reward
```

The base reward is the Q-value of (state, new_state), which also is the reward of “`distance`”

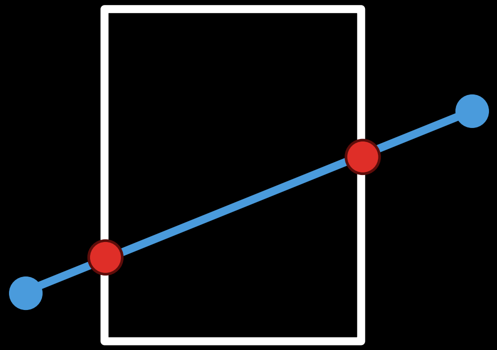
Else, the reward function is defined as

- If “`time`”: Add a random number in `[0, 1)`
- If “`traffic_box`”: Add
  - distance through box × `traffic_intensity` × a random number in `[0, 1)`

CODE REVIEW

2 functions to calculate the coordinates of two intersections between a segment and a rectangle

```
@staticmethod
def _calculate_point(x1,x2,y1,y2,x = None,y = None):
    if y1 == y2:
        return y1
    elif x1 == x2:
        return x1
    else:
        a = (y2-y1)/(x2-x1)
        b = y2 - a * x2

        if x is None:
            x = (y-b)/a
            return x
        elif y is None:
            y = a*x+b
            return y
        else:
            raise Exception("Provide x or y")
```



```
def _calculate_box_intersection(self,x1,x2,y1,y2,box):

    # Get box coordinates
    x_left,x_right,y_bottom,y_top = box

    # Intersections
    intersections = []

    # Top intersection
    i_top = self._calculate_point(x1,x2,y1,y2,y=y_top)
    if i_top > x_left and i_top < x_right:
        intersections.append((i_top,y_top))

    # Bottom intersection
    i_bottom = self._calculate_point(x1,x2,y1,y2,y=y_bottom)
    if i_bottom > x_left and i_bottom < x_right:
        intersections.append((i_bottom,y_bottom))

    # Left intersection
    i_left = self._calculate_point(x1,x2,y1,y2,x=x_left)
    if i_left > y_bottom and i_left < y_top:
        intersections.append((x_left,i_left))

    # Right intersection
    i_right = self._calculate_point(x1,x2,y1,y2,x=x_right)
    if i_right > y_bottom and i_right < y_top:
        intersections.append((x_right,i_right))

    return intersections
```

CODE REVIEW

```
class DeliveryQAgent(QAgent):

    def __init__(self,*args,**kwargs):
        super().__init__(*args,**kwargs)
        self.reset_memory()

    def act(self,s):

        # Get Q Vector
        q = np.copy(self.Q[s,:])

        # Avoid already visited states
        q[self.states_memory] = -np.inf

        if np.random.rand() > self.epsilon:
            a = np.argmax(q)
        else:
            a = np.random.choice([x for x in
range(self.actions_size) if x not in self.states_memory])

        return a

    def remember_state(self,s):
        self.states_memory.append(s)

    def reset_memory(self):
        self.states_memory = []
```

- Initialize the Q-Agent
- `act(s)` uses Epsilon-Greedy to choose an action on state s
- `remember_state(s)` saves the current state to the memory, in this case simply the visited city s
- `reset_memory()` clears the memory

CODE REVIEW

The Q-Learning implementation

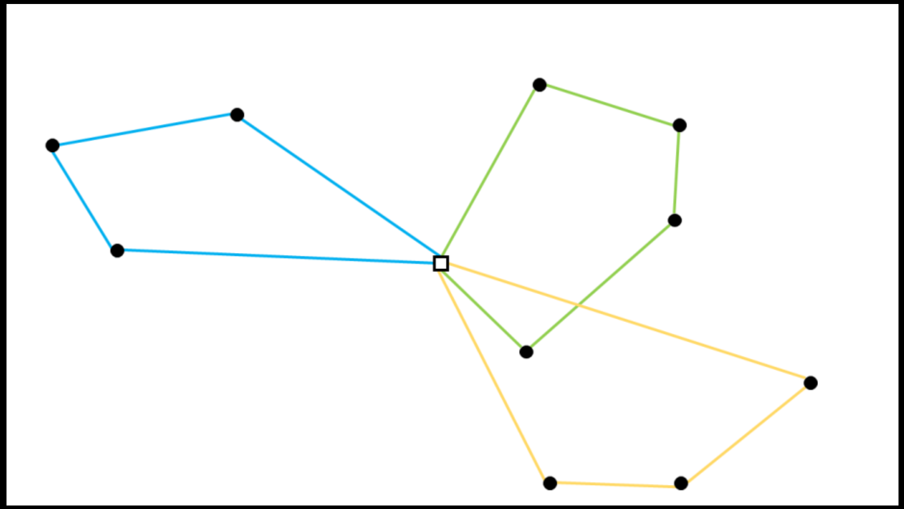
```
def run_episode(env,agent,verbose = 1):
    s = env.reset()
    agent.reset_memory()

    max_step = env.n_stops

    episode_reward = 0

    i = 0
    while i < max_step:
        # Remember the states
        agent.remember_state(s)

        # Choose an action
        a = agent.act(s)

        # Take the action, and get the reward from environment
        s_next,r,done = env.step(a)

        # Tweak the reward
        r = -1 * r

        if verbose: print(s_next,r,done)

        # Update our knowledge in the Q-table
        agent.train(s,a,r,s_next)

        # Update the caches
        episode_reward += r
        s = s_next

        # If the episode is terminated
        i += 1
        if done:
            break

    return env,agent,episode_reward


def run_n_episodes(env, agent, name="training.gif", n_episodes=1000,
                   render_each=10, fps=10):

    # Store the rewards
    rewards = []
    imgs = []

    # Experience replay
    for i in tqdm_notebook(range(n_episodes)):

        # Run the episode
        env,agent,episode_reward = run_episode(env,agent,verbose = 0)
        rewards.append(episode_reward)

        if i % render_each == 0:
            img = env.render(return_img = True)
            imgs.append(img)

    # THE AUTHOR SAVE SOME IMAGES HERE

    return env,agent
```

CAPACITATED VEHICLE ROUTING PROBLEM (CVRP)

"What is the optimal set of routes for a fleet of vehicles to traverse in order to deliver to a given set of customers?"

Also known as the Vehicle Routing Problem (VRP), Capacitated Vehicle Routing Problem (CVRP) is a generalization of TSP, which also is

- A combinatorial optimization problem
- An integer programming problem



PARAMETERS

```
class Delivery(gymnasium.Env):
    def __init__(self, n_stops=10, max_demand=10, max_vehicle_cap=30, max_env_size=1_000_000,
                 gen_seed=None, gym_seed=None, print_input=True, print_terminated=True)
```

The data are generated using the following parameters

- `n_stops` : number of stops
- `max_demand` : the maximum demand of all cities
- `max_vehicle_cap` : the maximum capacity of the vehicle
- `max_env_size` : the maximum coordinate of all cities

ASSUMPTIONS

To ensure the feasibility of the problem, assume that

- `max_demand ≤ max_vehicle_cap`
- The number of vehicles = `n_stops`

So that the problem always has a naïve solution: all vehicles move to all stops then comeback to *depot* immediately after

To somewhat simplify the problem, assume that

- All vehicles have the same capacity
- The objective is the total distance only, disregard the time, number of vehicles used, profit, or other factors

DATA GENERATION

- `demands`: demands of stops
  - A list with `n_stops` elements
  - demand of depot is always 0
  - `n_stops - 1` random integers in `[1, max_demand]`
- `stop_coords`: coordinates of stops
  - A matrix of shape `(n_stops, 2)`
  - Values are random integers in `[0, max_env_size]`

The depot is the first stop in those lists (index 0)

- `vehicle_cap`: the vehicle capacity
  - A random integer in `[max_demand, max_vehicle_cap]`

CVRP FORMULATION AS A RL PROBLEM

The problem can be translated into: One vehicle problem, and it cannot visit a stop if the load left is not enough for it.

The implementation of this formulation is in pure Python with Stable Baselines3 (SB3).

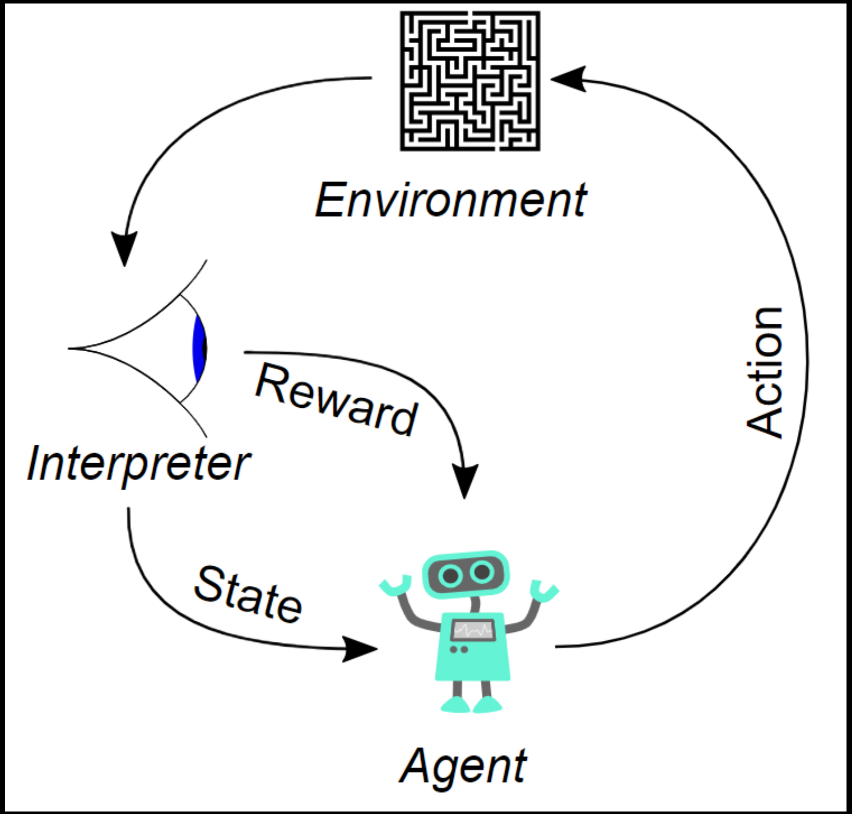
Environment: remain constants

- `demands`
- `stop_coords`
- `vehicle_cap`

State

- `current_stop`
- `visited`: list of stops are visited or not
- Some other derived information, including
    - `current_length`: total length moved

Objective: `current_length`



CVRP FORMULATION AS A RL PROBLEM

Immediate reward

- Valid action
  - `reward = - segment_length`
  - `segment_length`: The length between the two stops
- Invalid action: Move to a visited stop (except depot), or move to a stop with not enough load
  - `reward = - 2 * n_stops * max_env_size`
  - Idea behind
    - The maximum `segment_length` is $\sqrt{2}$ `* max_env_size` ≈ 1.41 `* max_env_size`
    - Punish the agent even more, proportional to a constant large number `n_stops`

Normalized reward

- Valid action: `reward = - segment_length / max_env_size`
- Invalid action: `reward = - 2 * n_stops`

RESULTS

- Deep Q-Learning (DQN)

250, 500, 1000, 5000, 10000, and 50000 are the (minimum) numbers of environment total timesteps during training

Each result is the minimum objective found out of 10 episodes

n_stops	250	500	1000	5000	10000	50000
5	3958	3602	5902	2855	4164	3730
6	4356	4417	4639	5071	4356	4417
7	3126	3306	2934	3037	3822	3228
8	4916	4930	4699	4916	4643	5011
9	6051	5492	5153	5167	4976	5557
10	6897	5827	5910	5218	6386	5864
12	6097	6856	6915	5345	5973	6550
15	10511	10852	11351	10335	10393	10079
20	14061	12699	12436	14362	12399	11828

RESULTS

- Advantage Actor-Critic (A2C)

n_stops	250	500	1000	5000	10000	50000
5	3602	2988	3730	3602	2988	3602
6	4356	4516	4639	4902	4417	4516
7	3855	3990	2794	2772	3093	3870
8	4817	4817	5187	4904	5259	5212
9	5040	5085	5348	5462	5612	5040
10	5749	7196	6946	5309	6661	6150
12	7902	9392	7345	8812	7951	8097
15	10143	10458	10038	10164	10165	9970
20	13239	11280	11118	12582	10982	14469

RESULTS

- Proximal Policy Optimization (PPO)

n_stops	250	500	1000	5000	10000	50000
5	2855	2855	3958	3602	3225	3581
6	4356	4356	4356	4516	4803	4417
7	3309	3315	3057	3037	3660	3550
8	4699	4930	5295	5475	5104	5011
9	5229	5240	5076	5282	5282	5368
10	6372	5258	5460	6127	5364	6003
12	7321	6444	6817	7171	7462	7415
15	8354	10118	9039	9035	9325	11043
20	11741	11079	11259	11454	11649	13128

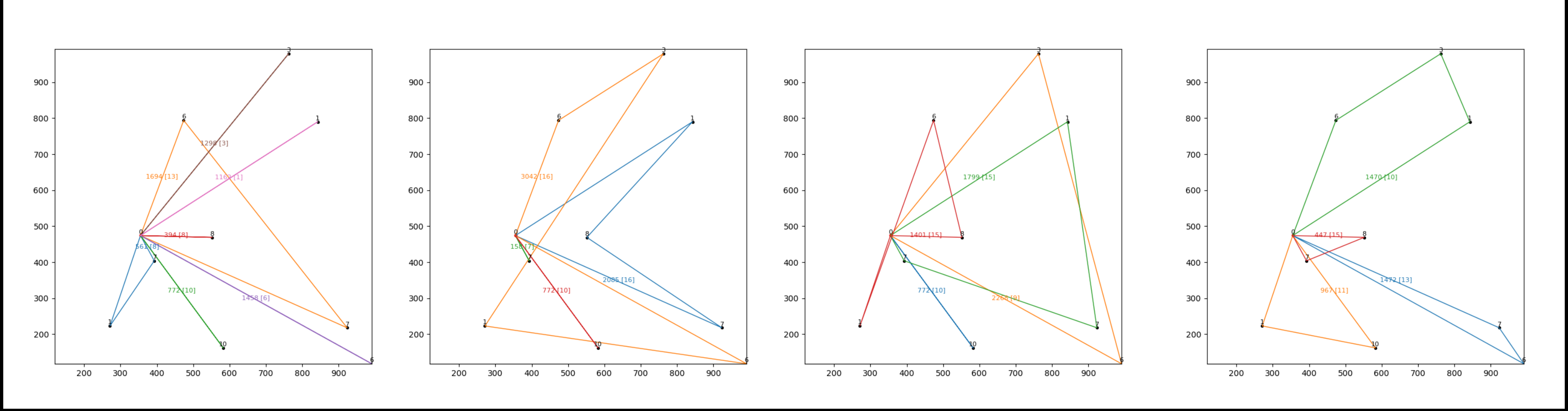
RESULTS

Best results, comparing with the solution provided by OR-Tools

- Deep Q-Learning (DQN) is the best RL model
- As expected, the RL models get worse when there are more stops
- However, they will still give a reasonable solution on a live environment without prior knowledge

n_stops	a2c	dqn	ppo	ortools
5	2988	2855	2855	2855
6	4356	4356	4356	4356
7	2772	2934	3037	2772
8	4817	4643	4699	4643
9	5040	4976	5076	4648
10	5309	5218	5258	3554
12	7345	5345	6444	4196
15	9970	10079	8354	7325
20	10982	11828	11079	7521

EXAMPLES



A2C (7339) DQN (6057) PPO (6240) OR-Tools (4356)

CONCLUSION & FUTURE WORKS

This project has

- Investigated RL approaches to optimization problems in general
- Reviewed a Q-learning implementation for TSP
- Formulated CVRP as a RL problem
- Implemented A2C, DQN, and PPO algorithms for CVRP
- Achieved remarkable results with DQN is the best model

Future works

- Continue to adjust the implemented algorithms
- Implement more algorithms
- Explore more RL applications for traditional optimization problems

APPLIED REINFORCEMENT LEARNING METHODS FOR THE CAPACITATED VEHICLE ROUTING PROBLEM

Hoang Tran Nhat Minh

Instructed by Dr. Pham Quang Dung

Data Science & Artificial Intelligence 2020

January 2024

ĐẠI HỌC BÁCH KHOA

SOICT

REFERENCES

- Solving the Traveling Salesman Problem with Reinforcement Learning. (2021, November 3). Eki.Lab. https://ekimetrics.github.io/blog/2021/11/03/tsp/#references
- L. Wang, Z. Pan and J. Wang, "A Review of Reinforcement Learning Based Intelligent Optimization for Manufacturing Scheduling," in Complex System Modeling and Simulation, vol. 1, no. 4, pp. 257-270, December 2021, doi: 10.23919/CSMS.2021.0027.
- Wikipedia contributors. (2024, January 5). Reinforcement learning. In Wikipedia, The Free Encyclopedia. Retrieved 04:02, January 16, 2024, from https://en.wikipedia.org/w/index.php?title=Reinforcement_learning&oldid=1193685081
- Gilman, R. (2018, January 9). Intuitive RL: Intro to Advantage-Actor-Critic (A2C). HackerNoon. https://hackernoon.com/intuitive-rl-intro-to-advantage-actor-critic-a2c-4ff545978752
- Karagiannakos, S. (2018, November 17). The idea behind Actor-Critics and how A2C and A3C improve them | AI Summer. AI Summer. https://theaisummer.com/Actor_critics/
- Understanding the role of the discount factor in reinforcement learning. (n.d.). Cross Validated. https://stats.stackexchange.com/questions/221402/understanding-the-role-of-the-discount-factor-in-reinforcement-learning

REFERENCES

- Mnih, V., Badia, A. P., Mirza, M., Graves, A., Lillicrap, T., Harley, T., ... & Kavukcuoglu, K. (2016, June). Asynchronous methods for deep reinforcement learning. In International conference on machine learning (pp. 1928-1937). PMLR.
- Wikipedia contributors. (2024, January 4). Q-learning. In Wikipedia, The Free Encyclopedia. Retrieved 04:08, January 16, 2024, from https://en.wikipedia.org/w/index.php?title=Q-learning&oldid=1193548086
- A2C — Stable Baselines3 2.3.0a1 documentation. (n.d.). https://stable-baselines3.readthedocs.io/en/master/modules/a2c.html
- Luu, Q. T., & Luu, Q. T. (2023, May 2). Q-Learning vs. Deep Q-Learning vs. Deep Q-Network | Baeldung on Computer Science. Baeldung on Computer Science. https://www.baeldung.com/cs/q-learning-vs-deep-q-learning-vs-deep-q-network
- Science, B. O. C., & Science, B. O. C. (2023, March 24). Epsilon-Greedy Q-Learning | Baeldung on Computer Science. Baeldung on Computer Science. https://www.baeldung.com/cs/epsilon-greedy-q-learning

REFERENCES

- GeeksforGeeks. (2021, September 27). Bellman equation. https://www.geeksforgeeks.org/bellman-equation/
- Wikipedia contributors. (2023, December 29). Bellman equation. In Wikipedia, The Free Encyclopedia. Retrieved 04:12, January 16, 2024, from https://en.wikipedia.org/w/index.php?title=Bellman_equation&oldid=1192511038
- DQN — Stable Baselines3 2.3.0a1 documentation. (n.d.). https://stable-baselines3.readthedocs.io/en/master/modules/dqn.html
- Fig. 7. Advantage actor critic. (n.d.). ResearchGate. https://www.researchgate.net/figure/Advantage-Actor-Critic_fig3_334521853
- Schulman, J., Levine, S., Abbeel, P., Jordan, M., & Moritz, P. (2015, June). Trust region policy optimization. In International conference on machine learning (pp. 1889-1897). PMLR.
- Proximal Policy Optimization — Spinning Up documentation. (n.d.). https://spinningup.openai.com/en/latest/algorithms/ppo.html

REFERENCES

- Trust Region Policy Optimization — Spinning Up documentation. (n.d.). https://spinningup.openai.com/en/latest/algorithms/trpo.html
- Karunakaran, D. (2021, December 16). Trust Region Policy Optimisation(TRPO) — a policy-based Reinforcement Learning. Medium. https://medium.com/intro-to-artificial-intelligence/trust-region-policy-optimisation-trpo-a-policy-based-reinforcement-learning-fd38ff9e996e
- Wikipedia contributors. (2023, December 27). Travelling salesman problem. In Wikipedia, The Free Encyclopedia. Retrieved 04:19, January 16, 2024, from https://en.wikipedia.org/w/index.php?title=Travelling_salesman_problem&oldid=1191996141
- Wikipedia contributors. (2023, December 1). Vehicle routing problem. In Wikipedia, The Free Encyclopedia. Retrieved 04:18, January 16, 2024, from https://en.wikipedia.org/w/index.php?title=Vehicle_routing_problem&oldid=1187812878
- Capacitated Vehicle Routing Problem formulation — AIMMS How-To. (2020, August 31). https://how-to.aimms.com/Articles/332/332-Formulation-CVRP.html