



UNIVERSITATEA DIN  
BUCUREȘTI

FACULTATEA DE  
MATEMATICĂ ȘI  
INFORMATICĂ



SPECIALIZAREA INFORMATICĂ

Lucrare de licență

# SINCRONIZAREA SEMAFOARELOR FOLOSIND REINFORCEMENT LEARNING

Absolvent

Plăcintescu Ștefan

Coordonator științific

Conf.dr. Ciprian Păduraru

București, Septembrie 2023

## **Rezumat**

Controlul semafoarelor este una dintre cele mai eficiente metode de optimizare a traficului rutier în orașe. Datorită avansului tehnologic al sistemelor de supraveghere a traficului, a devenit posibilă utilizarea algoritmilor de reinforcement learning pentru a controla dinamic semafoarele în funcție de condițiile curente. În această lucrare vor fi prezentate instrumentele și algoritmii de ultimă oră disponibili pentru experimentarea în domeniul optimizării fazelor semafoarelor prin reinforcement learning, precum și implementarea și evaluarea altor algoritmi în acest scop.

## **Abstract**

Traffic light control is one of the most efficient ways of optimizing road traffic in cities. Thanks to technological advancements in traffic surveillance technology, we now have the ability to use reinforcement learning algorithms to dynamically control traffic lights according to current conditions. This paper presents state-of-the-art tools and algorithms available for experimentation in the field of traffic light control using reinforcement learning, as well as the implementation and evaluation of other algorithms for this task.

# Cuprins

<b>1</b>	<b>Introducere</b>	<b>5</b>
1.1	Motivație personală . . . . .	5
1.2	Prezentare generală . . . . .	5
<b>2</b>	<b>Preliminarii</b>	<b>7</b>
2.1	Tehnologii folosite . . . . .	7
2.1.1	SUMO . . . . .	7
2.1.2	SUMO-RL . . . . .	7
2.1.3	RESCO . . . . .	8
2.1.4	PFRL . . . . .	8
2.2	Environment-uri . . . . .	8
2.2.1	Grid4x4 . . . . .	8
2.2.2	Arterial4x4 . . . . .	9
2.2.3	Cologne1 . . . . .	9
2.2.4	Cologne3 . . . . .	9
2.2.5	Cologne8 . . . . .	10
2.2.6	Ingolstadt1 . . . . .	10
2.2.7	Ingolstadt7 . . . . .	11
2.2.8	Ingolstadt21 . . . . .	11
<b>3</b>	<b>Implementare</b>	<b>13</b>
3.1	Training loop . . . . .	13
3.2	Crearea mediului de antrenare . . . . .	14
3.2.1	Atribute și Inițializare (___init___) . . . . .	14
3.2.2	Metoda step_sim . . . . .	15
3.2.3	Metoda reset . . . . .	15
3.2.4	Metoda step . . . . .	15
3.2.5	Metodele calc_metrics și save_metrics . . . . .	16
3.2.6	Metoda close . . . . .	16
3.3	Instantțierea semafoarelor . . . . .	16
3.3.1	Atribute . . . . .	16

3.3.2	Metoda generate_config . . . . .	17
3.3.3	Metoda phase . . . . .	17
3.3.4	Metoda prep_phase . . . . .	17
3.3.5	Metoda set_phase . . . . .	17
3.3.6	Metoda observe . . . . .	17
3.3.7	Metoda get_vehicles . . . . .	19
3.4	Funcțiile de stare . . . . .	19
3.4.1	drq . . . . .	19
3.4.2	drq_norm . . . . .	20
3.4.3	mplight . . . . .	20
3.4.4	mplight_full . . . . .	20
3.4.5	wave . . . . .	20
3.4.6	ma2c . . . . .	21
3.4.7	fma2c . . . . .	21
3.4.8	fma2c_full . . . . .	22
3.5	Calcularea recompenselor . . . . .	24
3.5.1	wait . . . . .	24
3.5.2	wait_norm . . . . .	24
3.5.3	pressure . . . . .	24
3.5.4	queue_maxweight . . . . .	24
3.5.5	queue_maxweight_neighborhood . . . . .	24
3.5.6	fma2c . . . . .	25
3.5.7	fma2c_full . . . . .	26
3.6	Fișierele de configurări . . . . .	28
3.7	Agenții . . . . .	28
3.7.1	Clasele șablon . . . . .	28
3.7.2	Agenții implementați în cadrul lucrării . . . . .	28
<b>4</b>	<b>Concluzii</b>	<b>40</b>
	<b>Bibliografie</b>	<b>41</b>

# Capitolul 1

## Introducere

### 1.1 Motivație personală

Machine learning-ul este un domeniu pe care îl consider foarte interesant și de care am devenit pasionat în ultimii ani, iar traficul rutier este o problemă cu care eu și marea majoritate a locuitorilor din medii urbane ne confruntăm zilnic, și această confruntare se termină, de regulă, în timp pierdut și frustrare.

Am fost atras de ideea controlului semafoarelor folosind reinforcement learning, deoarece este o temă ce are ca scop diminuarea unei probleme din viața mea de zi cu zi (traficul urban), utilizând tehnologii dintr-un domeniu științific de care sunt interesat (machine learning).

### 1.2 Prezentare generală

În literatura existentă, modelele de simulare a traficului sunt împărțite în două categorii:

- Modelul microscopic - fiecare vehicul este simulat în parte și îi sunt analizate mișcările și deciziile. Primele modele de acest tip au fost dezvoltate în anii '60. [8]
- Modelul macroscopic - în loc să analizeze comportamentul individual al vehiculelor, se concentrează pe variabilele agregate, cum ar fi densitatea traficului, viteza medie și fluxul de vehicule pentru a reduce efortul computațional. [16]

Optimizările la nivel microscopic sunt, în general, mai valoroase și mai detaliate decât cele la nivel macroscopic [12].

Scopul acestei lucrări este de a prezenta instrumentele și metodele existente folosite pentru optimizarea traficului, astfel încât orice persoană cu cunoștințe de bază în reinforcement learning (RL) să le poată folosi pentru a experimenta, și de a testa noi algoritmi pe care nu i-am întâlnit în experimente făcute în acest domeniu.

În cadrul lucrării au fost antrenați trei agenți bazați pe algoritmi de reinforcement learning din librăria PFRL [6]

- Double Deep Q-Network (DDQN) [7]
- Trust Region Policy Optimization (TRPO) [13]
- Rainbow [9]

Acești agenți au fost antrenați și evaluați cu ajutorul toolkit-ului RESCO [3], iar environment-urile folosite au fost create cu ajutorul SUMO [2].

# Capitolul 2

## Preliminarii

### 2.1 Tehnologii folosite

#### 2.1.1 SUMO

Simulation of Urban MObility (SUMO) [2] este un instrument open source de simulare a traficului urban la nivel microscopic dezvoltat de angajații Institutului de Sisteme de Transport de la Centrul Aerospațial German (DLR).

Am ales să folosesc SUMO deoarece:

- Este unul dintre cele mai populare și mai bine întreținute instrumente de simulare a traficului.
- Este bazat pe simulare microscopică.
- Poate fi integrat cu alte software-uri prin intermediul unor interfețe de programare (API-uri).
- Este capabil să gestioneze rețele de transport de orice dimensiune, de la intersecții individuale până la rețele regionale.
- A fost utilizat în mai multe dintre lucrările științifice care stau la baza acestei lucrări precum [12], [3].

#### 2.1.2 SUMO-RL

SUMO-RL [1] este interfață pentru crearea de environment-uri de reinforcement learning cu SUMO [2]. Aceasta funcționează, pentru sarcinile singe-agent, ca un environment obișnuit Gymnasium [17], iar pentru multi-agent folosește Petting Zoo [15]. Pentru comunicarea cu SUMO folosește API-ul TraCI (Traffic Control Interface).

### 2.1.3 RESCO

Reinforcement Learning Benchmarks for Traffic Signal Control (RESCO) [3] este un instrument open source pentru dezvoltarea și evaluarea algoritmilor de reinforcement learning în contextul controlului semafoarelor. RESCO oferă un set standardizat de scenarii și metrice pentru evaluarea performanței diferiților algoritmi RL, precum și implementări ai algoritmilor state-of-the-art în acest domeniu. Acest instrument permite cercetătorilor și dezvoltatorilor să compare diferite abordări într-un mod obiectiv și ajută la promovarea cercetării în domeniul controlului semafoarelor bazat pe RL și la accelerarea dezvoltării soluțiilor eficiente pentru gestionarea traficului urban.

### 2.1.4 PFRL

Preferred Networks Reinforcement Learning (PFRL) [6] este o librărie open source de deep reinforcement learning în cadrul căreia se regăsesc o varietate de algoritmi state-of-the-art de deep RL. Aceasta este o librărie flexibilă și eficientă, permițând cercetătorilor și dezvoltatorilor să experimenteze și să implementeze rapid diferiți algoritmi RL.

## 2.2 Environment-uri

Antrenarea și evaluarea agenților a fost făcută pe următoarele opt environment-uri puse la dispoziție de RESCO [3].

### 2.2.1 Grid4x4

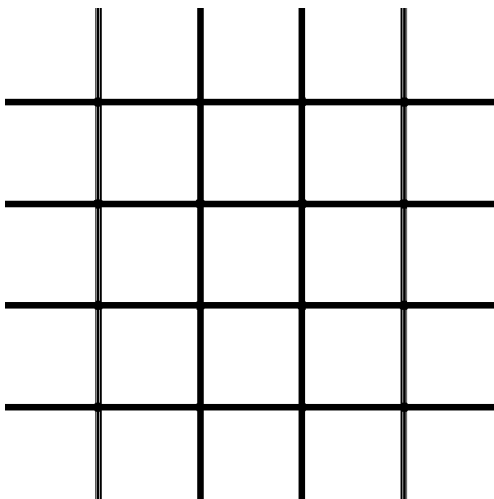


Figura 2.1: Harta completă Grid4x4

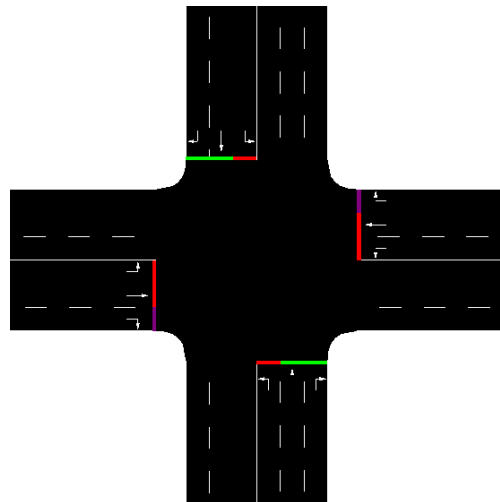


Figura 2.2: Intersecție din Grid4x4

Grid4x4 este, așa cum îi sugerează și numele, o rețea de 4x4 drumuri identice perpendiculare.



### 2.2.2 Arterial4x4

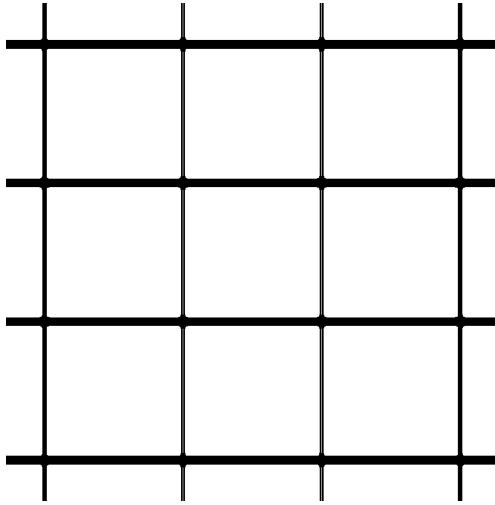


Figura 2.3: Harta completă Arterial4x4

Arterial4x4 este o rețea de 4 artere principale, orizontale ce se intersectează cu 4 drumuri mai mici, verticale.

### 2.2.3 Cologne1

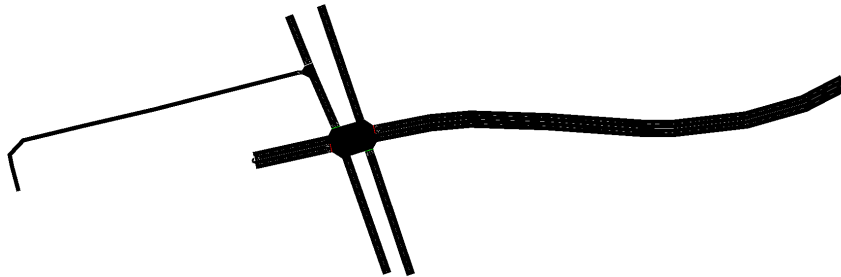


Figura 2.5: Cologne1

Cologne1 este un environment cu o singură intersecție din Cologne. Toate rețelele Cologne sunt extrase din scenariul SUMO TAPASCologne [14].

### 2.2.4 Cologne3

Cologne3 este o rețea de tip coridor, formată dintr-o arteră principală ce se intersectează cu mai multe drumuri. În acest environment sunt trei intersecții ale căror semafoare sunt controlate de agenți.

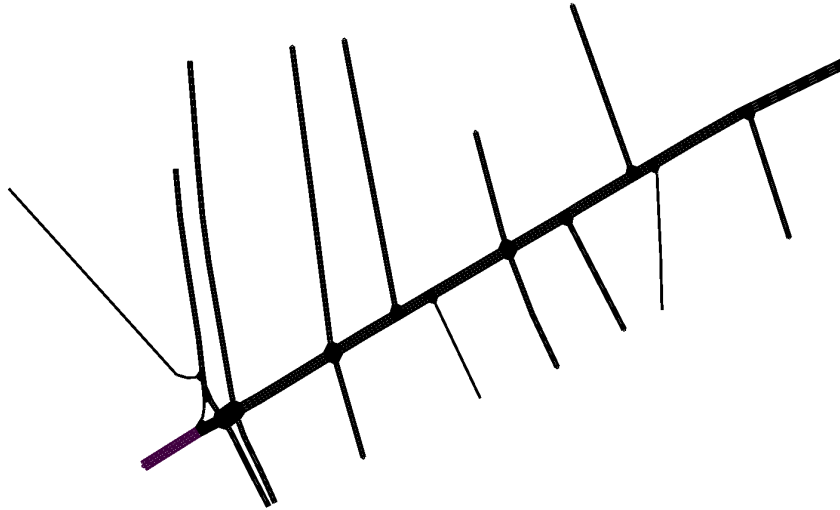


Figura 2.6: Cologne3

### 2.2.5 Cologne8

Cologne8 este un environment de dimensiune mai mare, acesta are opt intersecții controlate de agenți.

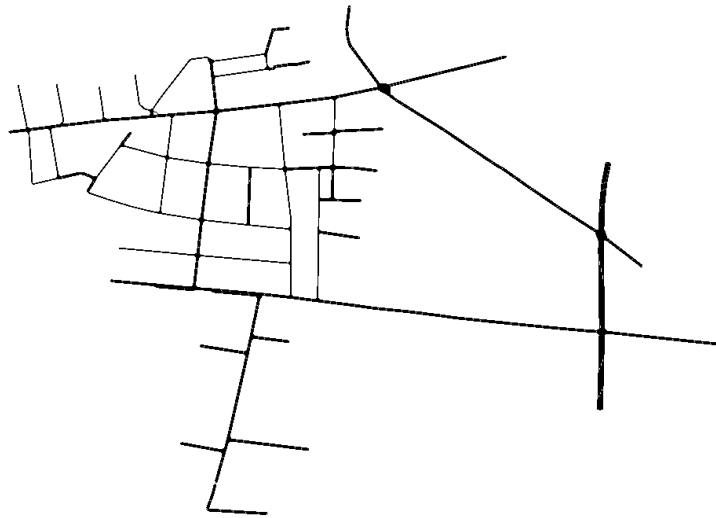


Figura 2.7: Cologne8

### 2.2.6 Ingolstadt1

Ingolstadt1 este un environment cu o singură intersecție din Ingolstadt. Toate rețelele Ingolstadt sunt extrase din scenariul SUMO InTAS [10].

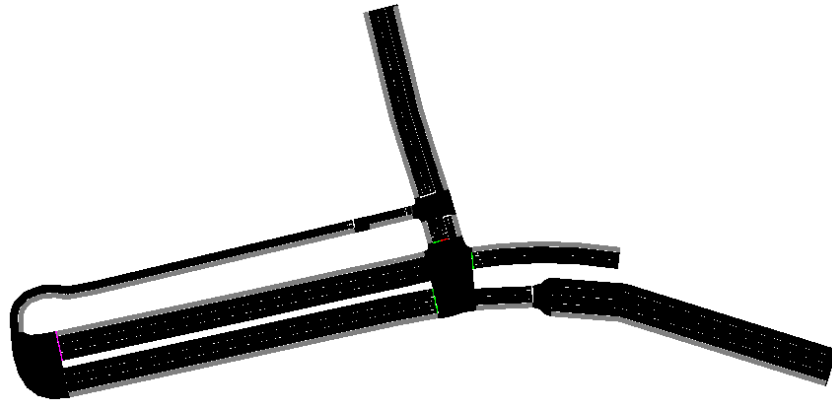


Figura 2.8: Ingolstadt1

### 2.2.7 Ingolstadt7

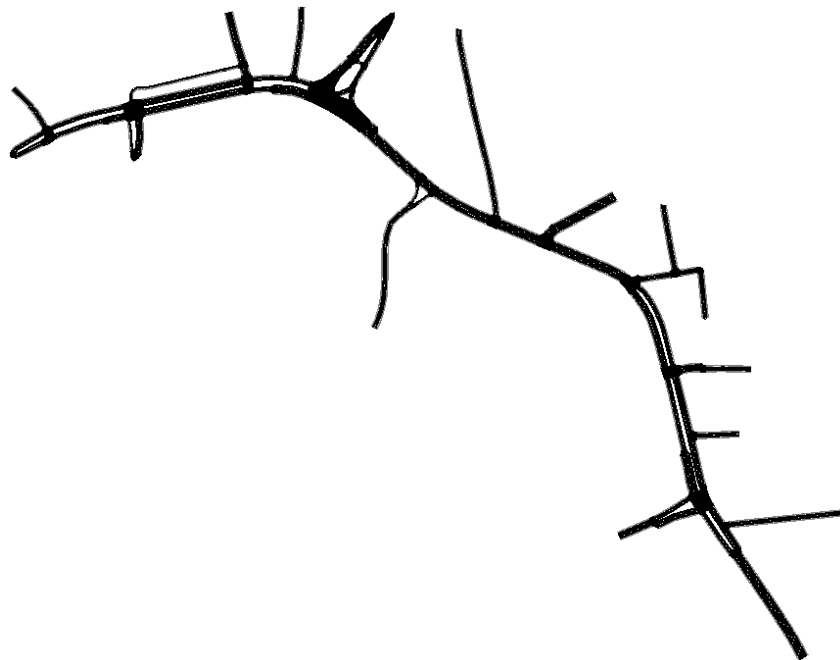


Figura 2.9: Ingolstadt7

Ingolstadt7 este o rețea de tip coridor, formată dintr-o arteră principală ce se intersectează cu mai multe drumuri. În acest environment șapte dintre intersecții sunt controlate de agenți pentru optimizarea traficului.

### 2.2.8 Ingolstadt21

Ingolstadt21 este cel mai complex environment folosit, acesta are 21 de intersecții controlate de agenți.



Figura 2.10: Ingolstadt21

# Capitolul 3

## Implementare

### 3.1 Training loop

Buclo de antrenare pentru agenții din RESCO [3] se află în fișierul `main.py`, unde se face citirea parametrilor (agentul de antrenat, environment-ul, numărul de epoci) ce sunt trimiși funcției `run_trial`, care apoi încarcă configurările hărților și agenților și creează un environment de antrenare instanțind clasa `MultiSignal`. După instanțierea environment-ului, este creat un dicționar pentru spațiile de observație și acțiune și este instanțiat agentul care apoi va fi antrenat pentru numărul de epoci ales.

```
env = MultiSignal(alg.__name__+'-tr'+str(trial),
                  args.map,
                  os.path.join(args.pwd, map_config['net']),
                  agt_config['state'],
                  agt_config['reward'],
                  route=route, step_length=map_config['step_length'], yellow_length=map_config['yellow_length'],
                  step_ratio=map_config['step_ratio'], end_time=map_config['end_time'],
                  max_distance=agt_config['max_distance'], lights=map_config['lights'], gui=args.gui,
                  log_dir=args.log_dir, libsumo=args.libsumo, warmup=map_config['warmup'])
```

Figura 3.1: Instanțierea mediului de antrenare

```
for _ in range(args.eps):
    obs = env.reset()
    done = False
    while not done:
        act = agent.act(obs)
        # print("Act:", act, "Type:", type(act))
        obs, rew, done, info = env.step(act)
        agent.observe(obs, rew, done, info)
    env.close()
```

Figura 3.2: Buclo de antrenare

## 3.2 Crearea mediului de antrenare

Environment-ul de antrenare este creat folosind clasa MultiSignal din `multi_signal.py`. MultiSignal este o extensie a clasei `gym.Env` din librăria OPEN AI Gym [5] care este o versiune mai veche a lui Gymnasium [17].

Componentele clasei:

### 3.2.1 Atribute și Inițializare (`__init__`)

- Atribute de configurare (`end_time`, `step_length`, `yellow_length`, `max_distance`, etc.) pe care le primește ca parametri din fișierele de configurare a hărții și a agentului
- `sumo` - stabilește conexiunea cu SUMO [2] prin intermediul API-ului TraCI
- `phase` - dicționar ce conține toate fazele valide
- `signals` - dicționar ce stochează un obiect de tip `Signal` pentru fiecare semafor
- `n_agents` - stabilește numărul de agenți, este egal cu numărul de semafoare
- `observation_space` și `action_space` - liste în care sunt stocate spațiile de observație, respectiv acțiune a tuturor semafoarelor; pentru obținerea acestora se aplică funcția de stare (`state_fn`) a agentului pe dicționarul de semnale (`signals`) pentru a crea un dicționar ce conține observații pentru fiecare semafor, care este apoi parcurs semafor cu semafor, iar pentru cel curent se extrage forma observației și se stochează în `obs_shape`, se creează un spațiu de observație de tip `Box` folosind biblioteca `gym` (acesta reprezintă un spațiu  $n$ -dimensional în care fiecare dimensiune poate avea valori într-un interval continuu, în acest caz, intervalul este de la `-np.inf` la `np.inf`, ceea ce înseamnă că nu există limite specifice pentru valorile observațiilor), se adaugă spațiul creat la lista `observation_space` și se creează un spațiu de acțiune de tip `Discrete`, ce reprezintă un set de acțiuni discrete, unde numărul de acțiuni posibile este dat de numărul de faze pentru semaforul curent, pentru a fi adăugat în `action_space`

```

self.obs_shape = dict()
self.observation_space = list()
self.action_space = list()
for ts in self.all_ts_ids:
    self.signals[ts] = Signal(self.map_name, self.sumo, ts, self.yellow_length, self.phases[ts])
for ts in self.all_ts_ids:
    self.signals[ts].signals = self.signals
    self.signals[ts].observe(self.step_length, self.max_distance)
observations = self.state_fn(self.signals)
self.ts_order = list()
for ts in observations:
    if ts == 'top_mgr' or ts == 'bot_mgr': continue # Not a traffic signal
    o_shape = observations[ts].shape
    self.obs_shape[ts] = o_shape
    o_shape = gym.spaces.Box(low=-np.inf, high=np.inf, shape=o_shape)
    self.ts_order.append(ts)
    self.observation_space.append(o_shape)
    self.action_space.append(gym.spaces.Discrete(len(self.phases[ts])))

```

Figura 3.3: Spațiile de observație și acțiune

### 3.2.2 Metoda step\_sim

Avansează simularea SUMO cu un număr de pași specificat de step\_ratio.

### 3.2.3 Metoda reset

Resetează mediul, închizând și repornind simularea SUMO și reinițializează semafoarele.

```

self.signal_ids = []
for i in range(self.ts_starter):
    self.signal_ids.append(self.all_ts_ids[i])

for ts in self.signal_ids:
    self.signals[ts] = Signal(self.map_name, self.sumo, ts, self.yellow_length, self.phases[ts])
    self.wait_metric[ts] = 0.0
for ts in self.signal_ids:
    self.signals[ts].signals = self.signals
    self.signals[ts].observe(self.step_length, self.max_distance)

```

Figura 3.4: Reinițializarea semafoarelor

### 3.2.4 Metoda step

- Pentru fiecare semnal de trafic, se pregătește faza semnalului în funcție de acțiunea primită.
- Execută simularea pentru durata fazei galbene a semaforului, apoi setează faza semaforului și continuă simularea pentru restul duratei pasului
- Pentru fiecare semnal de trafic, observă noua stare și actualizează informațiile despre semnal

- Calculează noua stare și recompensa folosind funcțiile de stare și recompensă (state\_fn, reward\_fn) definite în states.py, respectiv rewards.py
- Calculează metricile pentru evaluarea performanței agentului și se adaugă la lista de metrici folosind metoda calc\_metrics descrisă la punctul următor
- Verifică dacă timpul simulării a atins timpul final stabilit în configurarea hărții din map\_config.py
- Returnează rezultatele

```

for signal in self.signals:
    self.signals[signal].prep_phase(act[signal])

for step in range(self.yellow_length):
    self.step_sim()
for signal in self.signal_ids:
    self.signals[signal].set_phase()
for step in range(self.step_length - self.yellow_length):
    self.step_sim()
for signal in self.signal_ids:
    self.signals[signal].observe(self.step_length, self.max_distance)

```

Figura 3.5: Acțiune și observație

### 3.2.5 Metodele calc\_metrics și save\_metrics

Calculează, respectiv salvează metricile.

### 3.2.6 Metoda close

Închide conexiunea cu SUMO și apelează metoda save\_metrics.

## 3.3 Instanțierea semafoarelor

Semafoarele sunt instanțiate ca obiecte ale clasei Signal din traffic\_signal.py.

Componentele clasei:

### 3.3.1 Atribute

- yellow\_time - durata fazei galbene a semaforului
- next\_phase - următoarea fază în care va trece semaforul



- `lanes` - lista benzilor controlate de semafor
- `outbound_lanes` - lista benzilor de ieșire controlate
- `lane_sets` - dicționar care mapează direcțiile de trafic la seturile de benzi corespunzătoare
- `lane_sets_outbound`: Un dicționar similar cu `lane_sets`, dar pentru benzi de ieșire
- `downstream` - dicționar care mapează direcțiile de trafic la semafoarele downstream
- `waiting_times` - dicționar care mapează vehiculele la timpul lor de așteptare
- `phases` - lista fazelor semaforului
- `yellow_dict` - dicționar care mapează combinațiile de faze la indicii fazelor galbene
- `full_observation` - observația completă a stării semaforului și a vehiculelor din apropiere
- `last_step_vehicles` - lista vehiculelor detectate în pasul anterior de simulare

### 3.3.2 Metoda `generate_config`

Generează o configurare pentru semnale de trafic dacă nu există deja o configurare predefinită pentru harta aleasă în `signal_configs`.

### 3.3.3 Metoda `phase`

Returnează faza curentă a semaforului.

### 3.3.4 Metoda `prep_phase`

Pregătește trecerea la o nouă fază.

### 3.3.5 Metoda `set_phase`

Setează faza semaforului la `next_phase`.

### 3.3.6 Metoda `observe`

- `full_observation` este un dicționar care va conține observațiile pentru fiecare bandă, iar `all_vehicles` este un set care va conține toate vehiculele detectate
- Pentru fiecare bandă din `self.lanes`, se rețin vehiculele în `lane_vehicles` (folosind metoda `get_vehicles` descrisă la punctul următor)

- În dicționarul `vehicle_measures` se memorează informații despre fiecare vehicul, cum ar fi id-ul, timpul de așteptare, viteza, accelerația, poziția și tipul
- În dicționarul `lane_measures` se actualizează observațiile pentru banda curentă, cum ar fi timpul total de așteptare, timpul maxim de așteptare, numărul de vehicule în coadă și numărul de vehicule în apropiere
- Observațiile pentru banda curentă sunt adăugate în `full_observation`
- Setul `all_vehicles` este actualizat cu vehiculele detectate în pasul curent
- Se determină vehiculele care au sosit și cele care au plecat comparând vehiculele curente cu vehiculele detectate în pasul anterior (`self.last_step_vehicles`) și sunt eliminate din dicționarul `self.waiting_times` vehiculele care au plecat
- În `self.last_step_vehicles` sunt memorate vehiculele detectate în pasul curent, iar în `self.full_observation` observațiile colectate în pasul curent.

```

def observe(self, step_length, distance):
    full_observation = dict()
    all_vehicles = set()
    for lane in self.lanes:
        vehicles = []
        lane_measures = {'queue': 0, 'approach': 0, 'total_wait': 0, 'max_wait': 0}
        lane_vehicles = self.get_vehicles(lane, distance)
        for vehicle in lane_vehicles:
            all_vehicles.add(vehicle)
            # Update waiting time
            if vehicle in self.waiting_times:
                self.waiting_times[vehicle] += step_length
            elif self.sumo.vehicle.getWaitingTime(vehicle) > 0: # Vehicle stopped here, add it
                self.waiting_times[vehicle] = self.sumo.vehicle.getWaitingTime(vehicle)

            vehicle_measures = dict()
            vehicle_measures['id'] = vehicle
            vehicle_measures['wait'] = self.waiting_times[vehicle] if vehicle in self.waiting_times else 0
            vehicle_measures['speed'] = self.sumo.vehicle.getSpeed(vehicle)
            vehicle_measures['acceleration'] = self.sumo.vehicle.getAcceleration(vehicle)
            vehicle_measures['position'] = self.sumo.vehicle.getLanePosition(vehicle)
            vehicle_measures['type'] = self.sumo.vehicle.getTypeID(vehicle)
            vehicles.append(vehicle_measures)
            if vehicle_measures['wait'] > 0:
                lane_measures['total_wait'] = lane_measures['total_wait'] + vehicle_measures['wait']
                lane_measures['queue'] = lane_measures['queue'] + 1
                if vehicle_measures['wait'] > lane_measures['max_wait']:
                    lane_measures['max_wait'] = vehicle_measures['wait']
            else:
                lane_measures['approach'] = lane_measures['approach'] + 1
        lane_measures['vehicles'] = vehicles
        full_observation[lane] = lane_measures

    full_observation['num_vehicles'] = all_vehicles
    if self.last_step_vehicles is None:
        full_observation['arrivals'] = full_observation['num_vehicles']
        full_observation['departures'] = set()
    else:
        full_observation['arrivals'] = all_vehicles.difference(self.last_step_vehicles)
        departs = self.last_step_vehicles.difference(all_vehicles)
        full_observation['departures'] = departs
        # Clear departures from waiting times
        for vehicle in departs:
            if vehicle in self.waiting_times: self.waiting_times.pop(vehicle)

    self.last_step_vehicles = all_vehicles
    self.full_observation = full_observation

```

Figura 3.6: Metoda observe

### 3.3.7 Metoda get\_vehicles

Returnează vehiculele de pe o anumită bandă, într-o anumită distanță de semafor.

## 3.4 Funcțiile de stare

Funcțiile de stare definesc modul în care o stare (observație) a environment-ului este reprezentată pentru agent. Acestea primesc ca parametru un dicționar de semafoare (obiecte Signal) și returnează un dicționar de observații.

### 3.4.1 drq

Pentru fiecare semnal, creează o observație care constă în:

- O valoare binară care indică dacă banda curentă este banda activă (1 dacă este activă, 0 altfel)
- Numărul de vehicule care se apropie
- Timpul total de așteptare al vehiculelor în bandă
- Lungimea cozii de vehicule în bandă
- Viteza totală a tuturor vehiculelor din bandă

Observația pentru fiecare semnal este apoi extinsă de-a lungul unui nou ax și stocată în dicționar.

### 3.4.2 `drq_norm`

Identică cu funcția `drq`, dar normalizează valorile împărțindu-le la 28.

### 3.4.3 `mplight`

- Pentru fiecare semnal, extrage faza curentă a semaforului; aceasta este o valoare numerică care indică ce lumină (roșu, galben sau verde) este activă pentru fiecare direcție a intersecției
- Calculează lungimea cozii pentru fiecare direcție a semnalului; aceasta reprezintă numărul de vehicule care așteaptă la semafor
- Dacă există un semnal în downstream (adică un alt semafor în direcția în care se deplasează traficul), lungimea cozii pentru semnalul curent este ajustată; acest lucru se face scăzând lungimea cozii semnalului din downstream din lungimea cozii semnalului curent, deoarece dacă există un semafor în downstream care are o coadă lungă de vehicule, aceasta poate influența coada semnalului curent

### 3.4.4 `mplight_full`

Versiune extinsă a funcției `mplight`, care consideră și timpul total de așteptare, viteza totală și numărul total de vehicule care se apropie.

### 3.4.5 `wave`

- Pentru fiecare direcție a unui semnal, calculează suma lungimii cozii și a numărului de vehicule care se apropie
- Valorile rezultate formează observația pentru semnal

### 3.4.6 ma2c

- Pentru fiecare semafor, se calculează valoarea wave ca sumă a lungimii cozii și a numărului de vehicule care se apropie pentru fiecare bandă
- Valoarea undei este normalizată și limitată în funcție de configurarea MA2C (mdp\_config)
- Extrage valoarea undei pentru fiecare semafor și o ajustează în funcție de semnalele din downstream, folosind coeficientul coop\_gamma din configurarea MA2C
- Calculează timpul maxim de așteptare pentru fiecare bandă și îl normalizează și limitează în funcție de configurare
- Observațiile finale pentru fiecare semafor sunt tupluri formate din valorile wave și timpii de așteptare

### 3.4.7 fma2c

- Creează un dicționar region\_fringes pentru a stoca marginile pentru fiecare manager
- Pentru fiecare semafor, verifică vecinii săi downstream, dacă un vecin nu există sau supervisorul vecinului este diferit de supervisorul semaforului curent, adaugă benzile de intrare corespunzătoare direcției în region\_fringes sub supervisorul semaforului curent
- Creează un dicționar lane\_wave pentru a stoca valoarea wave pentru fiecare bandă
- Calculează valoarea undei ca sumă a lungimii cozii și a numărului de vehicule care se apropie pentru fiecare semafor și bandă asociată acestuia
- Creează un dicționar manager\_obs pentru a stoca observațiile pentru fiecare manager
- Adună valorile undei pentru benzile asociate fiecărui manager și le normalizează și limitează valorile în funcție de configurarea FMA2C din mdp\_configs
- Creează un dicționar management\_neighborhood pentru a stoca observațiile pentru vecinătatea fiecărui manager
- Pentru fiecare manager, adună observațiile sale cu observațiile vecinilor săi, ponderează cu coeficientul alpha din configurarea FMA2C
- Calculează valoarea wave pentru un semnal și vecinii săi downstream care au același supervisor și concatenează aceste valori într-un singur array

- Calculează timpul maxim de așteptare pentru fiecare bandă, normalizat și limitat în funcție de configurarea FMA2C
- Observațiile pentru fiecare semafor sunt tupluri formate din valorile wave și timpii de așteptare

### 3.4.8 fma2c\_full

Versiune extinsă a funcției fma2c, care consideră și timpul total de așteptare, viteza totală și numărul total de vehicule care se apropie când calculează wave.

```

def fma2c(signals):
    fma2c_config = mdp_configs['FMA2C']
    management = fma2c_config['management']
    supervisors = fma2c_config['supervisors'] # reverse of management
    management_neighbors = fma2c_config['management_neighbors']

    region_fringes = dict()
    for manager in management:
        region_fringes[manager] = []
    for signal_id in signals:
        signal = signals[signal_id]
        for key in signal.downstream:
            neighbor = signal.downstream[key]
            if neighbor is None or supervisors[neighbor] != supervisors[signal_id]:
                inbounds = signal.inbounds_fr_direction.get(key)
                if inbounds is not None:
                    mgr = supervisors[signal_id]
                    region_fringes[mgr] += inbounds

    lane_wave = dict()
    for signal_id in signals:
        signal = signals[signal_id]
        for lane in signal.lanes:
            lane_wave[lane] = signal.full_observation[lane]['queue'] + signal.full_observation[lane]['approach']

    manager_obs = dict()
    for manager in region_fringes:
        lanes = region_fringes[manager]
        waves = []
        for lane in lanes:
            waves.append(lane_wave[lane])
        manager_obs[manager] = np.clip(np.asarray(waves) / fma2c_config['norm_wave'], a_min=0, fma2c_config['clip_wave'])

    management_neighborhood = dict()
    for manager in manager_obs:
        neighborhood = [manager_obs[manager]]
        for neighbor in management_neighbors[manager]:
            neighborhood.append(fma2c_config['alpha'] * manager_obs[neighbor])
        management_neighborhood[manager] = np.concatenate(neighborhood)

    signal_wave = dict()
    for signal_id in signals:
        signal = signals[signal_id]
        waves = []
        for lane in signal.lanes:
            wave = signal.full_observation[lane]['queue'] + signal.full_observation[lane]['approach']
            waves.append(wave)
        signal_wave[signal_id] = np.clip(np.asarray(waves) / fma2c_config['norm_wave'], a_min=0, fma2c_config['clip_wave'])

    observations = dict()
    for signal_id in signals:
        signal = signals[signal_id]
        waves = [signal_wave[signal_id]]
        for key in signal.downstream:
            neighbor = signal.downstream[key]
            if neighbor is not None and supervisors[neighbor] == supervisors[signal_id]:
                waves.append(fma2c_config['alpha'] * signal_wave[neighbor])
        waves = np.concatenate(waves)

        waits = []
        for lane in signal.lanes:
            max_wait = signal.full_observation[lane]['max_wait']
            waits.append(max_wait)
        waits = np.clip(np.asarray(waits) / fma2c_config['norm_wait'], a_min=0, fma2c_config['clip_wait'])

        observations[signal_id] = np.concatenate([waves, waits])
    observations.update(management_neighborhood)
    return observations

```

Figura 3.7: Funcția de stare fma2c

## 3.5 Calcularea recompenselor

Funcțiile de calcul pentru recompense, folosite pentru a ghida agentul se află în fișierul `rewards.py`.

### 3.5.1 `wait`

- Pentru fiecare semafor, sumează timpul total de așteptare pentru toate benzile asociate semaforului
- Recompensa este negativă, pentru a încuraja minimizarea timpului de așteptare

### 3.5.2 `wait__norm`

La fel ca funcția `wait`, dar împarte recompensa la 224 și o restrânge între -4 și 4 pentru a o normaliza.

### 3.5.3 `pressure`

- Calculează recompensa bazată pe presiunea la semafoare
- Presiunea este definită ca diferența dintre numărul de vehicule care așteaptă la un semafor și numărul de vehicule care așteaptă la semafoarele în downstream
- Recompensa este negativă, pentru a încuraja minimizarea presiunii

### 3.5.4 `queue__maxweight`

- Calculează recompensa bazată pe lungimea cozii de vehicule și timpul maxim de așteptare la semafoare
- Pentru fiecare semafor, calculează suma dintre lungimea cozii și timpul maxim de așteptare, ponderat cu un coeficient din `mdp_configs`

### 3.5.5 `queue__maxweight__neighborhood`

- Extinde funcția `queue__maxwait` prin luarea în considerare a recompenselor semafoarelor vecine
- Recompensa pentru fiecare semafor însumată cu recompensa semafoarelor vecine, ponderată cu un coeficient din `mdp_configs`



### 3.5.6 fma2c

- Inițializează dicționare pentru a stoca informații despre marginile regiunii, sosirile la aceste margini și liquidity (diferența dintre numărul de vehicule care pleacă și numărul de vehicule care sosesc) pentru fiecare manager
- Parcurge toate semnalele de trafic și, pentru fiecare semnal, se verifică vecinii săi downstream, iar dacă un vecin nu există sau nu are același supervisor ca semnalul curent, se consideră că semnalul curent este la marginea unei regiuni
- Adaugă toate benzile care intră în semnalul curent din direcția respectivă la lista de margini pentru managerul său
- Parcurge (din nou) toate semnalele de trafic și, pentru fiecare semnal, identifică managerul său și se calculează lichiditatea pentru managerul respectiv ca diferență dintre numărul de vehicule care pleacă și numărul de vehicule care sosesc, apoi dacă vreo bandă a semnalului curent este în lista de margini pentru managerul său se adaugă fiecare vehicul care a sosit pe acea bandă la numărul total de sosiri pentru managerul respectiv
- Pentru fiecare manager, se calculează recompensa bazată pe informațiile sale și ale vecinilor săi
- Recompensa inițială pentru manager este suma dintre numărul de vehicule care sosesc la margini și liquidity
- Pentru fiecare vecin al managerului curent, se adaugă la recompensa managerului recompensa vecinului înmulțită cu un coeficient alpha specificat în mdp\_configs, iar recompensele sunt salvate în dicționarul management\_neighborhood
- Se parcurg (din nou) toate semnalele de trafic și, pentru fiecare semnal, recompensa este suma dintre lungimea cozii și timpul maxim de așteptare pentru fiecare bandă, înmulțit cu un coeficient specificat în configurarea FMA2C
- Recompensele iau valori negative pentru a încuraja minimizarea cozilor și a timpilor de așteptare
- Se parcurg (din nou) toate semnalele de trafic și, pentru fiecare semnal, se adaugă la recompensa sa recompensele vecinilor săi downstream, înmulțite cu un coeficient alpha specificat în configurarea FMA2C, dacă aceștia au același supervisor. Aceste recompense sunt memorate în dicționarul neighborhood\_rewards
- Recompensele finale sunt obținute adăugând valorile din management\_neighborhood în dicționarul neighborhood\_rewards

### 3.5.7 fma2c\_full

Identică cu fma2c, dar are o configurare diferită în mdp\_configs.

```

def fma2c(signals):
    fma2c_config = mdp_configs['FMA2C']
    management = fma2c_config['management']
    supervisors = fma2c_config['supervisors'] # reverse of management
    management_neighbors = fma2c_config['management_neighbors']

    region_fringes = dict()
    fringe_arrivals = dict()
    liquidity = dict()
    for manager in management:
        region_fringes[manager] = []
        fringe_arrivals[manager] = 0
        liquidity[manager] = 0

    for signal_id in signals:
        signal = signals[signal_id]
        for key in signal.downstream:
            neighbor = signal.downstream[key]
            if neighbor is None or supervisors[neighbor] != supervisors[signal_id]:
                inbounds = signal.inbounds_fr_direction.get(key)
                if inbounds is not None:
                    mgr = supervisors[signal_id]
                    region_fringes[mgr] += inbounds

    for signal_id in signals:
        signal = signals[signal_id]
        manager = supervisors[signal_id]
        fringes = region_fringes[manager]
        arrivals = signal.full_observation['arrivals']
        liquidity[manager] += (len(signal.full_observation['departures']) - len(signal.full_observation['arrivals']))
        for lane in signal.lanes:
            if lane in fringes:
                for vehicle in signal.full_observation[lane]['vehicles']:
                    if vehicle['id'] in arrivals:
                        fringe_arrivals[manager] += 1

    management_neighborhood = dict()
    for manager in management:
        mgr_rew = fringe_arrivals[manager] + liquidity[manager]
        for neighbor in management_neighbors[manager]:
            mgr_rew += (fma2c_config['alpha'] * (fringe_arrivals[neighbor] + liquidity[neighbor]))
        management_neighborhood[manager] = mgr_rew

    rewards = dict()
    for signal_id in signals:
        signal = signals[signal_id]
        reward = 0
        for lane in signal.lanes:
            reward += signal.full_observation[lane]['queue']
            reward += (signal.full_observation[lane]['max_wait'] * mdp_configs['FMA2C']['coef'])
        rewards[signal_id] = -reward

    neighborhood_rewards = dict()
    for signal_id in signals:
        signal = signals[signal_id]
        sum_reward = rewards[signal_id]

        for key in signal.downstream:
            neighbor = signal.downstream[key]
            if neighbor is not None and supervisors[neighbor] == supervisors[signal_id]:
                sum_reward += (fma2c_config['alpha'] * rewards[neighbor])
        neighborhood_rewards[signal_id] = sum_reward

    neighborhood_rewards.update(management_neighborhood)
    return neighborhood_rewards

```

Figura 3.8: Funcția de reward fma2c

## 3.6 Fișierele de configurări

- **agent\_config** - conține configurări utilizate pentru a defini și personaliza comportamentul agenților în simulare
- **map\_config** - conține configurări legate de harta simulării
- **mdp\_config** - conține configurări legate de procesul decizional Markov
- **signal\_config** - conține configurări legate de semafoare

Acestea sunt toate fișierele ce conțin configurările necesare pentru a rula și personaliza simularea RESCO.

## 3.7 Agenții

### 3.7.1 Clasele șablon

Fișierul `agent.py` conține clase șablon făcute pentru a fi extinse în implementările agenților de control al semafoarelor.

#### 1. Clasa **Agent**

- Dacă are acces la GPU, îl alege ca dispozitivul pe care rulează modelul, altfel alege CPU
- Metoda `act` - metodă care trebuie să fie implementată de subclase, scopul ei este de a lua o decizie bazată pe o observație dată
- Metoda `observe` - metodă care trebuie să fie implementată de subclase, folosită pentru a actualiza agentul cu informații noi după ce a luat o acțiune

#### 2. Clasa **IndependentAgent**

- Metoda `act` - preia o acțiune pentru fiecare agent în funcție de observația dată și returnează un dicționar cu acțiunile pentru fiecare agent
- Metoda `observe` - actualizează fiecare agent cu informații noi după ce a luat o acțiune, iar dacă episodul s-a încheiat, salvează modelul agentului

### 3.7.2 Agenții implementați în cadrul lucrării

#### Agentul DDQN

Double DQN (DDQN) [7] este o variantă a algoritmului Deep Q-Network (DQN) [11] ce abordează problema de supraestimare a recompenselor în Q-learning. Acesta folosește

două rețele neurale: una pentru a selecta acțiunea și cealaltă pentru a evalua acea acțiune, actualizând periodic rețeaua de evaluare cu ponderile rețelei de selecție.

DQN combină Q-Learning cu rețele neurale adânci pentru a crea un algoritm capabil să trateze medii complexe. În Q-Learning agentul folosește o funcție  $Q(s, a)$  care returnează recompensa viitoare așteptată pentru acțiunea  $a$  în starea  $s$  cu scopul de a maximiza recompensa cumulată. Rețeaua neurală adâncă este folosită pentru a estima, în funcție de o stare  $s$ , valorile  $Q$  pentru toate acțiunile posibile. În plus, DQN folosește un replay buffer, care are rolul de a rupe corelația dintre experiențele consecutive pentru a stabiliza procesul de învățare, și o rețea țintă, care este o copie, actualizată mai rar, a rețelei principale, ce are rolul de a calcula valorile  $Q$  țintă în timpul actualizării rețelei principale pentru a preveni schimbări semnificative în valorile  $Q$  estimate.

Implementarea DDQN:

```
class DDQN(IndependentAgent):
    def __init__(self, config, obs_act, map_name, thread_number):
        super().__init__(config, obs_act, map_name, thread_number)
        for key in obs_act:
            obs_space = obs_act[key][0]
            act_space = obs_act[key][1]

            def conv2d_size_out(size, kernel_size=2, stride=1):
                return (size - (kernel_size - 1) - 1) // stride + 1

            h = conv2d_size_out(obs_space[1])
            w = conv2d_size_out(obs_space[2])

            model = nn.Sequential(
                nn.Conv2d(obs_space[0], out_channels=64, kernel_size=(2, 2)),
                nn.ReLU(),
                nn.Flatten(),
                nn.Linear(h * w * 64, out_features=64),
                nn.ReLU(),
                nn.Linear(in_features=64, act_space),
                DiscreteActionValueHead()
            )

            self.agents[key] = DDQNAgent(config, act_space, model)
```

Figura 3.9: Clasa DDQN

## 1. Clasa DDQN

- Extinde clasa IndependentAgent

- Parcurge fiecare cheie din `obs_act` (dicționar ce conține spațiile de observație și acțiuni pentru fiecare agent)
- Extrage spațiul de observație și spațiul de acțiune
- Calculează dimensiunile de ieșire pentru un strat convoluțional folosind funcția `conv2d_size_out`
- Construiește un model neural ce conține:
  - Conv2d - primul strat al modelului, responsabil pentru extragerea caracteristicilor spațiale din observație
  - ReLU - funcție de activare non-liniară
  - Flatten - aplatizează tensorul pentru a putea fi introdus în straturile dense (linear)
  - Linear - straturi responsabile pentru procesarea informațiilor extrase de straturile convoluționale și pentru a lua decizii pe baza acestora
  - DiscreteActionValueHead - strat specializat care estimează valoarea fiecărei acțiuni discrete posibile, aceasta este valoarea Q pentru fiecare acțiune
- Inițializează un agent DDQN cu acest model și îl stochează în dicționarul `self.agents`

## 2. Clasa **DDQNAgent**

- Extinde clasa `Agent`
- Inițializează un optimizator Adam pentru modelul primit ca argument
- Creează un replay buffer cu o capacitate de 10.000, acesta este o memorie în care agentul stochează experiențele sale recente ca în loc să învețe direct din experiențele imediate și consecutive (care pot fi foarte corelate și pot duce la instabilitate în învățare), să învețe dintr-un eșantion aleatoriu de experiențe anterioare
- Inițializează un explorator care folosește o strategie EpsilonGreedy cu LinearDecay a valorii epsilon
  - EpsilonGreedy este folosit pentru a echilibra explorarea și exploatarea - la fiecare pas, agentul decide dacă va alege o acțiune la întâmplare (explorare) sau dacă va alege cea mai bună acțiune pe care o cunoaște în funcție de ceea ce a învățat până acum (exploatare), dacă epsilon este 1, agentul va alege întotdeauna o acțiune la întâmplare, iar dacă este 0, va alege întotdeauna cea mai bună acțiune cunoscută
  - LinearDecay - reduce valoarea lui epsilon în mod uniform de la o valoare de pornire la o valoare finală pe parcursul unui număr de pași specificat în configurarea agentului

```

class DDQNAgent(Agent):
    def __init__(self, config, act_space, model, num_agents=0):
        super().__init__()

        self.model = model
        self.optimizer = torch.optim.Adam(self.model.parameters())
        replay_buffer = replay_buffers.ReplayBuffer(100000)

        explorer = explorers.LinearDecayEpsilonGreedy(
            config['EPS_START'],
            config['EPS_END'],
            config['steps'],
            lambda: np.random.randint(act_space),
        )

        self.agent = DoubleDQN(self.model, self.optimizer, replay_buffer, config['GAMMA'], explorer,
                               gpu=self.device.index,
                               minibatch_size=config['BATCH_SIZE'], replay_start_size=config['BATCH_SIZE'],
                               phi=lambda x: np.asarray(x, dtype=np.float32),
                               target_update_interval=config['TARGET_UPDATE'])

    def act(self, observation, valid_acts=None, reverse_valid=None):
        return self.agent.act(observation)

    def observe(self, observation, reward, done, info):
        self.agent.observe(observation, reward, done, reset=False)

4 usages (4 dynamic)
    def save(self, path):
        torch.save({
            'model_state_dict': self.model.state_dict(),
            'optimizer_state_dict': self.optimizer.state_dict(),
        }, path+'.pt')

```

Figura 3.10: Clasa DDQNAgent

- Această strategie încurajează explorarea la început, dar pe măsură ce agentul se antrenează îi crește tendința de a exploata
- Inițializează un agent DoubleDQN cu modelul, optimizatorul, bufferul de re-învățare, exploratorul și restul parametrilor din fișierul de configurare al agentului
- Metoda act - primește o observație și returnează o acțiune luată de agentul DDQN
- Metoda observe - primește o observație, o recompensă și o valoare done (bool ce arată dacă antrenarea s-a încheiat), apoi actualizează agentul DDQN cu aceste informații
- Metoda save - salvează stările dicționarului modelului și optimizatorului într-un fișier

## Agentul TRPO

Trust Region Policy Optimization (TRPO) [13] este un algoritm de reinforcement learning ce a face actualizări ale politicii într-o manieră care nu schimbă prea mult politica la

fiecare pas, pentru a evita actualizări care pot deteriora performanța. Pentru a face acest lucru, creează niște trust regions (zone de încredere) înăuntrul cărora politica nouă nu este foarte diferită de cea veche. Măsura acestei diferențe este dată de divergența Kullback-Leibler (KL) care este o măsură a diferenței dintre două distribuții de probabilitate  $P$  și  $Q$ .

$$D_{KL}(P||Q) = \sum_i P(i) \log \left( \frac{P(i)}{Q(i)} \right)$$

Proprietăți ale divergenței KL:

- Non-negativă -  $D_{KL}(P||Q) \geq 0$ , și este egală cu zero dacă și numai dacă  $P$  și  $Q$  sunt aceeași distribuție
- Nesimetrică -  $D_{KL}(P||Q)$  nu este neapărat egală cu  $D_{KL}(Q||P)$ , adică distanța de la  $P$  la  $Q$  nu este neapărat egală cu cea de la  $Q$  la  $P$

Implementarea TRPO:

### 1. Clasa TRPONetwork

```
class TRPONetwork(nn.Module):
    new *
    def __init__(self, obs_space, act_space):
        super(TRPONetwork, self).__init__()

        self.obs_space = obs_space
        self.h = conv2d_size_out(obs_space[1])
        self.w = conv2d_size_out(obs_space[2])

        self.conv = nn.Conv2d(obs_space[0], out_channels=64, kernel_size=(2, 2))
        self.fc1 = nn.Linear(64 * self.h * self.w, out_features=64)
        self.fc2 = nn.Linear(in_features=64, act_space)

    new *
    def forward(self, x):
        x = self.conv(x)
        x = nn.ReLU()(x)
        x = x.view(-1, 64 * self.h * self.w)
        x = self.fc1(x)
        x = nn.ReLU()(x)
        logits = self.fc2(x)
        return Categorical(logits=logits) # Return a Categorical distribution
```

Figura 3.11: Clasa TRPONetwork

- Definește rețeaua neurală folosită de agent
- Este formată dintr-un strat convoluțional urmat de două straturi liniare



- Metoda forward - observația este trecută prin stratul convoluțional, apoi este aplatizată și trecută prin straturile liniare și, în final, se returnează o distribuție categorică bazată pe logitii produși de al doilea strat liniar

## 2. Clasa ValueFunction

```
class ValueFunction(nn.Module):
    new *
    def __init__(self, obs_space):
        super(ValueFunction, self).__init__()
        #self.apply(init_weights)

        self.fc1 = nn.Linear(np.prod(obs_space), out_features: 64)
        self.fc2 = nn.Linear(in_features: 64, out_features: 1)

    new *
    def forward(self, x):
        x = x.view(x.size(0), -1)
        x = nn.ReLU()(self.fc1(x))
        return self.fc2(x)
```

Figura 3.12: Clasa ValueFunction

- Rețea neurală simplă care estimează funcția de valoare a unei observații date
- Are două straturi liniare și produce un scalar care reprezintă valoarea estimată a observației

## 3. Clasa ITRPO

- Extinde clasa IndependentAgent
- Parcurge fiecare cheie din obs\_act (dicționar ce conține spațiile de observație și acțiuni pentru fiecare agent)
- Extrage spațiul de observație și spațiul de acțiune
- creează instanță de policy (TRPONetwork) și o instanță de vf (ValueFunction) pe care le folosește pentru a inițializa un agent TRPO care este stocat în dicționarul self.agents

## 4. Clasa TRPOAgent

- Extinde clasa Agent

```

class ITRPO(IndependentAgent):
    new *
    def __init__(self, config, obs_act, map_name, thread_number):
        super().__init__(config, obs_act, map_name, thread_number)
        for key in obs_act:
            obs_space = obs_act[key][0]
            act_space = obs_act[key][1]

            policy = TRPONetwork(obs_space, act_space)
            vf = ValueFunction(obs_space)

            self.agents[key] = TRPOAgent(config, act_space, policy, vf)

```

Figura 3.13: Clasa ITRPO

- Inițializează un optimizator Adam pentru funcția de valoare (vf) primită ca argument
- Inițializează un agent TRPO cu politica, funcția de valoare, optimizatorul și restul parametrilor din fișierul de configurare al agentului
- Metoda act - primește o observație și returnează o acțiune luată de agentul DDQN
- Metoda observe - primește o observație, o recompensă și o valoare done (bool ce arată dacă antrenarea s-a încheiat), apoi actualizează agentul DDQN cu aceste informații
- Metoda save - salvează stările dicționarului politicii, funcției de valoare și optimizatorului într-un fișier

```

class TRPOAgent(Agent):
    """
    new *
    """
    def __init__(self, config, act_space, policy, vf):
        super().__init__()

        # Define the policy and value function
        self.policy = policy
        self.vf = vf

        # Define the optimizer for the value function
        learning_rate = 0.01
        self.vf_optimizer = torch.optim.Adam(self.vf.parameters(), lr=learning_rate)

        # Initialize the TRPO agent from PFRL
        self.agent = pfrl.agents.TRPO(
            policy=self.policy,
            vf=self.vf,
            vf_optimizer=self.vf_optimizer,
            gpu=self.device.index,
            gamma=config['GAMMA'],
            update_interval=config.get('UPDATE_INTERVAL', 2048),
            conjugate_gradient_max_iter=config.get('CG_MAX_ITER', 10),
            conjugate_gradient_damping=config.get('CG_DAMPING', 1e-1),
            lambd=config.get('LAMBDA', 0.97),
            entropy_coef=config.get('ENTROPY_COEF', 0.0),
            phi=lambda x: np.asarray(x, dtype=np.float32),
        )

    """
    new *
    """
    def act(self, observation):
        return self.agent.act(observation)

    """
    new *
    """
    def observe(self, observation, reward, done, info):
        self.agent.observe(observation, reward, done, reset=False)

    """
    4 usages (4 dynamic) new *
    """
    def save(self, path):
        torch.save({
            'policy_state_dict': self.policy.state_dict(),
            'vf_state_dict': self.vf.state_dict(),
            'optimizer_state_dict': self.vf_optimizer.state_dict(),
        }, path + '.pt')

```

Figura 3.14: Clase TRPOAgent

## Agentul RAINBOW

Rainbow [9] este o combinație mai multe îmbunătățiri aduse algoritmului DQN [11].

1. Double DQN - explicat la pagina 28
2. Prioritized Experience Replay (prioritizarea experienței) - în loc să eşantioneze experiențele uniform (ca replay buffer-ul folosit în DDQN), experiențele care sunt mai puțin frecvente sunt eşantionate mai des
3. Dueling DQN - modifică arhitectura rețelei pentru a avea două fluxuri separate, unul pentru a estima valoarea stării și celălalt pentru a estima avantajul fiecărei acțiuni care sunt apoi combinate pentru a produce valoarea  $Q$
4. Categorical DQN [4] - în loc să estimeze o singură valoare  $Q$  pentru fiecare pereche stare-acțiune, estimează întreaga distribuție de probabilitate a valorilor  $Q$

Implementarea RAINBOW:

### 1. Clasa CustomDuelingDQN

- Implementare a unui model Dueling DQN
- Modelul începe cu un strat de convoluție (`self.conv`), urmat de un flux principal (`self.main_stream`)
- Ieșirea fluxului principal este împărțită în două fluxuri: fluxul de avantaj (`self.a_stream`) și fluxul de valoare (`self.v_stream`)
- Metoda `forward` - intrarea este trecută prin stratul de convoluție, aplatizată și apoi trecută prin fluxul principal, ieșirea este împărțită în fluxuri de avantaj și valoare, iar valorile  $Q$  finale sunt calculate folosind formula arhitecturii dueling
- Funcția `action_value.DistributionalDiscreteActionValue` ia distribuția de valori  $Q$  și `z_values` (tensor ce reprezintă valorile  $Q$  posibile) pentru a produce o distribuție peste valorile  $Q$  pentru fiecare acțiune

### 2. Clasa RAINBOW

- Extinde clasa `IndependentAgent`
- Parcurge fiecare cheie din `obs_act` (dicționar ce conține spațiile de observație și acțiune pentru fiecare agent)
- Extrage spațiul de observație și spațiul de acțiune
- creează un model `CustomDuelingDQN` pe care îl folosește pentru a inițializa un agent Rainbow care este stocat în dicționarul `self.agents`

### 3. Clasa `RainbowAgent`

- Extinde clasa `Agent`
- Inițializează un optimizator Adam pentru modelul primit ca argument
- Creează un replay buffer folosind de tip `PrioritizedReplayBuffer` care eșantionează experiențele în funcție de importanța lor
- Folosește un explorator (`explorers.LinearDecayEpsilonGreedy`), care începe cu o rată mare de explorare și o scade linear în timp
- Inițializează un agent `CategoricalDoubleDQN` cu modelul `CustomDuelingDQN`, optimizatorul, bufferul, exploratorul și restul parametrilor din fișierul de configurare al agentului
- Metoda `act` - primește o observație și returnează o acțiune luată de agentul DDQN
- Metoda `observe` - primește o observație, o recompensă și o valoare done (bool ce arată dacă antrenarea s-a încheiat), apoi actualizează agentul DDQN cu aceste informații
- Metoda `save` - salvează stările dicționarului politicii și optimizatorului într-un fișier

```

class CustomDuelingDQN(nn.Module):
    new *
    def __init__(self, n_actions, n_atoms, v_min, v_max, obs_space):
        super(CustomDuelingDQN, self).__init__()

        self.n_actions = n_actions
        self.n_atoms = n_atoms
        self.v_min = v_min
        self.v_max = v_max
        self.z_values = torch.linspace(v_min, v_max, n_atoms, dtype=torch.float32)

        self.h = conv2d_size_out(obs_space[1])
        self.w = conv2d_size_out(obs_space[2])

        # Convolutional layers
        self.conv = nn.Conv2d(obs_space[0], out_channels=64, kernel_size=(2, 2))
        self.activation = nn.ReLU()

        self.main_stream = nn.Linear(self.h * self.w * 64, out_features=128)

        # Dueling branches
        self.a_stream = nn.Linear(in_features=64, n_actions * n_atoms)
        self.v_stream = nn.Linear(in_features=64, n_atoms)

    new *
    def forward(self, x):
        h = self.activation(self.conv(x))
        h = h.view(h.size(0), -1) # Flatten
        #print(h.size())
        h = self.activation(self.main_stream(h))
        h_a, h_v = torch.chunk(h, chunks=2, dim=1)

        # Advantage
        ya = self.a_stream(h_a).reshape((-1, self.n_actions, self.n_atoms))
        mean = ya.sum(dim=1, keepdim=True) / self.n_actions
        ya, mean = torch.broadcast_tensors(*tensors: ya, mean)
        ya -= mean

        # State value
        ys = self.v_stream(h_v).reshape((-1, 1, self.n_atoms))
        ya, ys = torch.broadcast_tensors(*tensors: ya, ys)
        q = F.softmax(ya + ys, dim=2)

        self.z_values = self.z_values.to(x.device)
        return action_value.DistributionalDiscreteActionValue(q, self.z_values)

```

Figura 3.15: Clase CustomDuelingDQN

```

class RAINBOW(IndependentAgent):
    new *
    def __init__(self, config, obs_act, map_name, thread_number):
        super().__init__(config, obs_act, map_name, thread_number)
        for key in obs_act:
            obs_space = obs_act[key][0]
            act_space = obs_act[key][1]

            model = CustomDuelingDQN(act_space, n_atoms: 51, -100, v_max: 100, obs_space)

            self.agents[key] = RainbowAgent(config, act_space, model, map_name)

```

Figura 3.16: Clase RAINBOW

```

class RainbowAgent(Agent):
    new *
    def __init__(self, config, act_space, model, map_name):
        super().__init__()

        self.model = model
        self.optimizer = torch.optim.Adam(self.model.parameters())

        map_config = map_configs[map_name]
        eval_steps = map_config['end_time'] / map_config['step_length']
        replay_buffer = replay_buffers.PrioritizedReplayBuffer(
            capacity=10000,
            alpha=0.5,
            beta=0.4,
            betasteps=eval_steps/config['TARGET_UPDATE']
        )

        explorer = explorers.LinearDecayEpsilonGreedy(
            config['EPS_START'],
            config['EPS_END'],
            config['steps'],
            lambda: np.random.randint(act_space),
        )

        self.agent = CategoricalDoubleDQN(self.model, self.optimizer, replay_buffer,
            gamma=config['GAMMA'],
            explorer=explorer,
            gpu=self.device.index,
            minibatch_size=config['BATCH_SIZE'], replay_start_size=config['BATCH_SIZE'],
            phi=lambda x: np.asarray(x, dtype=np.float32),
            target_update_interval=config['TARGET_UPDATE']
        )

    new *
    def act(self, observation, valid_acts=None, reverse_valid=None):
        return self.agent.act(observation)

    new *
    def observe(self, observation, reward, done, info):
        self.agent.observe(observation, reward, done, reset=False)

4 usages (4 dynamic) new *
    def save(self, path):
        torch.save({
            'model_state_dict': self.model.state_dict(),
            'optimizer_state_dict': self.optimizer.state_dict(),
        }, path + '.pt')

```

Figura 3.17: Clase RainbowAgent

## Capitolul 4

## Concluzii



# Bibliografie

- [1] Lucas N. Alegre, *SUMO-RL*, <https://github.com/LucasAlegre/sumo-rl>, 2019.
- [2] Pablo Alvarez Lopez, Michael Behrisch, Laura Bieker-Walz, Jakob Erdmann, Yun-Pang Flötteröd, Robert Hilbrich, Leonhard Lücken, Johannes Rummel, Peter Wagner și Evamarie Wiessner, „Microscopic Traffic Simulation using SUMO”, în.
- [3] James Ault și Guni Sharon, „Reinforcement Learning Benchmarks for Traffic Signal Control”, în *Proceedings of the Thirty-fifth Conference on Neural Information Processing Systems (NeurIPS 2021) Datasets and Benchmarks Track*, Dec. 2021.
- [4] Marc G. Bellemare, Will Dabney și Rémi Munos, *A Distributional Perspective on Reinforcement Learning*, 2017, arXiv: [1707.06887](https://arxiv.org/abs/1707.06887) [cs.LG].
- [5] Greg Brockman, Vicki Cheung, Ludwig Pettersson, Jonas Schneider, John Schulman, Jie Tang și Wojciech Zaremba, *OpenAI Gym*, 2016, eprint: [arXiv:1606.01540](https://arxiv.org/abs/1606.01540).
- [6] Yasuhiro Fujita, Prabhat Nagarajan, Toshiki Kataoka și Takahiro Ishikawa, „ChainerRL: A Deep Reinforcement Learning Library”, în *Journal of Machine Learning Research* 22.77 (2021), pp. 1–14, URL: <http://jmlr.org/papers/v22/20-376.html>.
- [7] Hado van Hasselt, Arthur Guez și David Silver, *Deep Reinforcement Learning with Double Q-learning*, 2015, arXiv: [1509.06461](https://arxiv.org/abs/1509.06461) [cs.LG].
- [8] Dirk Helbing și Benno Tilch, „Generalized force model of traffic dynamics”, în *Physical Review E* 58.1 (Iul. 1998), pp. 133–138, DOI: [10.1103/physreve.58.133](https://doi.org/10.1103/physreve.58.133), URL: <https://doi.org/10.1103/physreve.58.133>.
- [9] Matteo Hessel, Joseph Modayil, Hado van Hasselt, Tom Schaul, Georg Ostrovski, Will Dabney, Dan Horgan, Bilal Piot, Mohammad Azar și David Silver, *Rainbow: Combining Improvements in Deep Reinforcement Learning*, 2017, arXiv: [1710.02298](https://arxiv.org/abs/1710.02298) [cs.AI].
- [10] Silas Lobo, Stefan Neumeier, Evelio Fernández și Christian Facchi, *InTAS – The Ingolstadt Traffic Scenario for SUMO*, Nov. 2020, DOI: [10.52825/scp.v1i](https://doi.org/10.52825/scp.v1i).

- [11] V. Mnih, K. Kavukcuoglu, D. Silver, A. A. Rusu, J. Veness, M. G. Bellemare, A. Graves, M. Riedmiller, A. K. Fidjeland, G. Ostrovski et al., *Human-level control through deep reinforcement learning*, 2015, DOI: [10.1038/nature14236](https://doi.org/10.1038/nature14236), URL: <https://doi.org/10.1038/nature14236>.
- [12] Ciprian Paduraru, Miruna Paduraru și Alin Stefanescu, „Traffic Light Control using Reinforcement Learning: A Survey and an Open Source Implementation”, în Apr. 2022, pp. 69–79, DOI: [10.5220/0011040300003191](https://doi.org/10.5220/0011040300003191).
- [13] John Schulman, Sergey Levine, Philipp Moritz, Michael I. Jordan și Pieter Abbeel, *Trust Region Policy Optimization*, 2017, arXiv: [1502.05477](https://arxiv.org/abs/1502.05477) [cs.LG].
- [14] Eclipse SUMO, *TAPASCologne*, URL: <https://sumo.dlr.de/docs/Data/Scenarios/TAPASCologne.html>.
- [15] J Terry, Benjamin Black, Nathaniel Grammel, Mario Jayakumar, Ananth Hari, Ryan Sullivan, Luis S Santos, Clemens Dieffendahl, Caroline Horsch, Rodrigo Perez-Vicente et al., „Pettingzoo: Gym for multi-agent reinforcement learning”, în *Advances in Neural Information Processing Systems* 34 (2021), pp. 15032–15043.
- [16] Elvira Thonhofer, Toni Palau, Andreas Kuhn, Stefan Jakubek și Martin Kozek, „Macroscopic traffic model for large scale urban traffic network design”, în *Simulation Modelling Practice and Theory* 80 (2018), pp. 32–49.
- [17] Mark Towers, Jordan K. Terry, Ariel Kwiatkowski, John U. Balis, Gianluca de Cola, Tristan Deleu, Manuel Goulão, Andreas Kallinteris, Arjun KG, Markus Krimmel, Rodrigo Perez-Vicente, Andrea Pierré, Sander Schulhoff, Jun Jet Tai, Andrew Tan Jin Shen și Omar G. Younis, *Gymnasium*, Mar. 2023, DOI: [10.5281/zenodo.8127026](https://doi.org/10.5281/zenodo.8127026), URL: <https://zenodo.org/record/8127025> (accesat în 8.7.2023).