Universitatea din București
Facultatea de Matematică și Informatică

Documentație Proiect RL

Humanoid Environment

Căpățînă Răzvan Nicolae, grupa 352
Ciobanu Dragoș, grupa 351
Luculescu Teodor, grupa 351

1 Prezentare generală a proiectului

1.1 Descriere

Proiectul constă în implementarea unor diverși agenți pentru Humanoid Environment (versiunea 5) și compararea performanțelor și implementărilor acestora.

1.2 Agenți:

- Monte Carlo
- Proximal Policy Optimization
- DeepQLearning

2 Mediul Humanoid în Python

2.1 Configurare

Exemplu de cod pentru rularea environment-ului:

```
import gymnasium as gym

env = gym.make("Humanoid-v5", render_mode="human")

NUM_STEPS = 1000
state, info = env.reset()
for step in range(NUM_STEPS):
    action = env.action_space.sample()
    nextState, reward, done, truncated, info = env.step(action)
    state = nextState
    if done:
        break
    env.render()
```

2.2 Prezentare generală

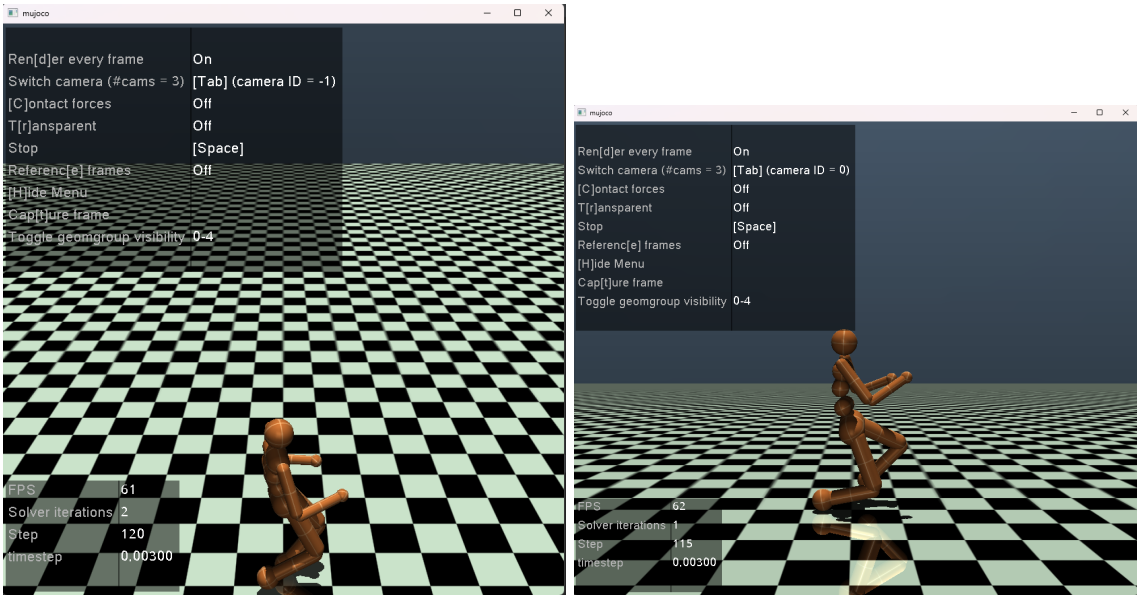
Humanoid este un mediu 3D, unde există un robot bipedal ce înfățișeaza un om, având un abdomen, brațe și picioare. Scopul său este de a merge înainte cât mai repede și pe o durată de timp cât mai lungă de timp fără să cadă.

Spațiul de observații: este un tuplu de forma $s = (s_1, s_2, ..., s_{348})$, unde un s_i oarecare poate avea diverse semnificații: de la unghiuri între puncte cheie ale humanoid-ului, la viteze și orientări ale acestora.

$s_i \in \mathbb{R}, \forall i \in \{1, 2, 3, ..., 348\}$

Spațiul de acțiuni: este un tuplu de forma $a = (a_1, a_2, ..., a_{17})$, unde un a_i oarecare reprezintă o forță (de translație sau rotație) aplicată unei anumite încheieturi din compoziția humanoid-ului.

$a_j \in [-0.4, 0.4], \forall j \in \{1, 2, 3, ..., 17\}$



3 Prezentarea Agenților aleși pentru antrenare

3.1 Agentul Monte Carlo

Agentul Monte Carlo (MC) este un algoritm de învățare prin întărire fără model. Acesta învață scorul unei stări sau a unei perechi de tip stare-acțiune. Scorurile sunt actualizate prin observarea rezultatelor obținute după vizitarea stării sau perechii stare-acțiune de mai multe ore. Ideea principală a algoritmului este de a utiliza experiența reala dobândită în urma episoadelor pentru a estima rezultatele așteptate pentru o stare sau o pereche de tip stare-acțiune.

Pe baza unui set de stări $S = (s_1, s_2, ..., s_{348})$ și acțiuni $A = (a_1, a_2, ..., a_{17})$, agentul interacționează cu environmentul pentru a obține o secvență de stări, acțiuni și rewarduri. Monte Carlo folosește secvența de lungime k: $(S_1A_1R_1, S_2A_2R_2, ..., S_kA_kR_k)$ pentru a calcula returnarea G_i, care reprezintă recompensa totală acumulată de la pasul de timp i până la sfârșitul episodului cu starea S_k .

Pași antrenare agent pentru Humanoid environment:

1. Inițializarea variabilelor și a parametrilor configurabili: Se inițializează cu o valoare default funcția de valoare stare-acțiune $Q(s_i, a_i)$ pentru toate $s_i \in S$ și $a_i \in A$. De asemenea, se inițializează, la fel, politica $\pi(s_i)$ și funcția de numărare $Q_N(s_i, a_i)$, care este un contor pentu a delimita de câte ori acțiunea a_i a fost luată din starea s_i.

$$Q(s_i, a_i) = 0 \quad \forall s_i \in S, a_i \in A$$
$$Q_N(s_i, a_i) = 0 \quad \forall s_i \in S, a_i \in A$$

Parametrii configurabili sunt inițializați și ei astfel (Vezi secțiunile următoare pentru utilizare):

- γ (Factorul de Discount): 0.95
- ϵ (Rata de explorare): 1.0, care se reduce o dată cu învătarea agentului cu ϵ_{decay} ($\epsilon = \epsilon * \epsilon_{\text{decay}}$)
- ϵ_{decay} (Factorul de reducere a lui epsilon): 0.999, determină cât de rapid scade rata de explorare.
- ϵ_{min} (Valoarea minimă a lui epsilon): 0.1, limita inferioară a lui epsilon. ($\epsilon = \max(\epsilon_{\text{min}}, \epsilon)$)
- N_{EPISODES} (Numărul de episoade pentru antrenare) : parametru configurabil
- UPDATE_POLICY_EVERY: Definirea frecvenței cu care politica este actualizată. (politica este actualizată o dată la UPDATE_POLICY_EVERY episoade)

2. Generarea unui episod: Un episod este generat prin interacțiunea cu environmentul conform politicii curente și a explorării Epsilon-Greedy (prezentate la punctul 7.). Episodul constă într-o secvență de perechi stare, acțiune, reward ($S_1A_1R_1, S_2A_2R_2, ..., S_kA_kR_k$), în care agentul începe dintr-o stare inițială S_1, indeplinește o acțiune A_1, primește o recompensă R_1, și trece la starea următoare S_2. Din starea S_2, agentul continuă către următoarea stare prin același chain.

Discretizarea Stării și Acțiunii

Algoritmul utilizează un sistem de discretizare pentru starea și acțiunea agentului, în scopul de a putea gestiona un spațiu continuu și de a lua decizii de acțiune pe baza unei politici de învățare.

Discretizarea Stării

Starea continuă a agentului este discretizată pentru a putea fi folosită de către Monte Carlo Agent folosind următorii pași:

- Starea continuă este rotunjită la o zecimală folosind funcția np.round(state, 1) din numpy, pentru a reduce numărul de stări ale environmentului.
- Rezultatul este un tuplu $S = (s_1, s_2, ..., s_n)$ care reprezintă starea discretizată.

În Python, funcția corespunzătoare este:

```
def discretize_state(self, state):
    """Discretize continuous state as hashable state."""
    state = np.round(state, 1)
    return tuple(state)
```

Discretizarea Acțiunii

Valorile acțiunilor pentru fiecare joint al humanoidului [-0.4, 0.4] sunt împărțite în numere de bin-uri stabilite prin NUM_BINS_ACTION, de obicei 4.

Sunt aplicați următorii pași:

- Crearea unei liste de bin-uri cu valori între -0.4 și 0.4, utilizând funcția np.linspace(-0.4, 0.4, num_bins + 1).
- Utilizarea funcției np.digitize(action, bins) pentru a găsi bin-ul corespunzător acțiunii curente. Valorile sunt normalizate în mulțimea {0, 1, 2, ..., NUM_BINS_ACTION - 1}

Funcția Python corespunzătoare este:

```
def discretize_action(self, action):
    """Discretize continuous action into numbered bins in the range [-0.4, 0.4]."""
    num_bins = self.conf.NUM_BINS_ACTION  # 4
    action_min, action_max = -0.4, 0.4

    bins = np.linspace(action_min, action_max, num_bins + 1)
    discretized_action = np.digitize(action, bins) - 1

    return tuple(discretized_action)
```

3. First Visit sau All Visits: În funcție de variația algoritmului, sunt două abordări pentru actualizarea lui $Q(s_i, a_i)$:

- *Monte Carlo First Visit*: Actualizează valoarea perechii stare-acțiune (s_i, a_i) doar prima dată când este vizitată într-un episod.
- *Monte Carlo All Visits*: Actualizează valoarea perechii stare-acțiune (s_i, a_i) de fiecare dată când este vizitată într-un episod.

```
def monte_carlo_policy(self, episode, firstVisit = True):
    """Update policy using Monte Carlo returns."""

    first_visit_state_actions = {}

    if firstVisit == True:
        for t in range(len(episode)):
            state, action, reward = episode[t]
            state_key = self.helper.discretize_state(state)
            action_key = self.helper.discretize_action(action)

            if (state_key, action_key) not in first_visit_state_actions:
                first_visit_state_actions[(state_key, action_key)] = t
```

4. Calcularea Returnării: După generarea unui episod, returnarea G_i este calculată pentru fiecare pereche stare-acțiune vizitată. Returnarea reprezintă recompensa totală acumulată de la pasul de timp i până la sfârșitul episodului cu starea S_k, unde k este lungimea secvenței.

$$G_i = r_{i+1} + \gamma r_{i+2} + \gamma^2 r_{i+3} + \dots$$

unde γ este factorul de discount, iar r_i este recompensa la pasul de timp i.
Formula se poate calcula recursiv astfel:

```
G = reward + self.gamma * G # parcurgere episod în ordine inversă
```

5. Actualizarea lui $Q(s_i, a_i)$: După calcularea returnării G_i pentru fiecare pereche stare, acțiune (s_i, a_i), funcția de valoare $Q(s_i, a_i)$ este actualizată prin medierea returnărilor pentru fiecare pereche. Actualizarea este realizeată astfel:

$$Q(s_i, a_i) = Q(s_i, a_i) + \alpha \left(G_i - Q(s_i, a_i)\right)$$

, unde α este learning rate-ul.

```
self.Q_n[state_key][action] += 1
alpha = min(1, 1.0 / math.sqrt(self.Q_n[state_key][action]))
self.Q[state_key][action] = self.Q[state_key][action] + alpha * (G - self.Q[state_key][action])
```

6. Îmbunătățirea Politicii: După actualizarea funcției de valoare, politica $\pi(s_i)$ este îmbunătățită prin alegerea acțiunii care maximizează funcția de valoare (stare, acțiune) $Q(s_i, a_i)$:

$$\pi(s_i) = a_i \text{ astfel încât } Q(s_i, a_i) = MAX$$

Politica este actualizată în funcțtie de parametrul UPDATE_POLICY_EVERY(o dată la UPDATE_POLICY_EVERY episoade)

```
def update_policy(self):
    """Update policy based on the Q values."""
    for state_key in self.Q_actions:
        Max = -1e9
        best_action = None
        for action_in_Q in self.Q_actions[state_key]:
            if self.Q[state_key][action_in_Q] > Max:
                Max = self.Q[state_key][action_in_Q]
                best_action = action_in_Q

        self.policy[state_key] = best_action
```

7. Explorarea Epsilon-Greedy: Echilibrarea explorării și a exploatării se realizează printr-o politică epsilon-greedy. Astfel, cu o probabilitate de ϵ, Monte Carlo alege o acțiune random (explorare), iar cu probabilitatea $1 - \epsilon$, alege acțiunea maximală (exploatare).

$$A = \begin{cases} \text{acțiune random} & \text{cu probabilitatea } \epsilon \\ a_i \text{ astfel încât } Q(s_i, a_i) = MAX & \text{cu probabilitatea } 1 - \epsilon \end{cases}$$

```
def choose_action(self, state, epsilon=0):
    """Choose an action based on the epsilon-greedy policy."""
    state_key = self.helper.discretize_state(state)

    if state_key not in self.policy:
        self.policy[state_key] = self.helper.discretize_action(np.random.uniform(self.action_low, s

    if np.random.rand() > epsilon:
        action = self.policy[state_key]
    else:
        action = self.helper.discretize_action(np.random.uniform(self.action_low, self.action_high)

    return self.helper.reverse_discretize_action(action)
```

8. Convergența Algoritmului: Pe măsură ce agentul avansează si colectează mai multe date despre episoadele parcuse, funcția de valoare $Q(s_i, a_i)$ converge către adevărata funcție de valoare (stare, acțiune), iar politica π converge către politica optimă.

3.2 Agentul Proximal Policy Optimization

Optimizarea Proximală a Politicii este un algoritm de învățare prin întărire care încearcă să îmbunătățească simplitatea în timp ce îmbunătățește performanța. PPO face parte din metodele de gradient al politicii și îmbunătățește metodele anterioare, cum ar fi Optimizarea politicii regiunii de încredere (TRPO), prin introducerea unei metode mai simple și eficiente de a limita actualizarile politicii.

Învățareaa prin întărire are ca scop de a ajuta un agent să învețe o politică perfectă $\pi(a_i|s_i)$, care linkează stările $S = (s_1, s_2, s_3, ...,)$ în acțiuni $A = (a_1, a_2, a_3, ...)$. PPO realizează acest lucru prin utilizarea elementelor următoare:

- **Funcția Obiectiv Surrogat:** PPO optimizează o funcție obiectiv surrogat clasică:

$$L_{CLIP} = \mathbb{E}_i \left[\min \left(r_i A'_i, \text{clip}(r_i, 1-\epsilon, 1+\epsilon) A'_i \right) \right],$$

, unde r_i marchează raportul dintre probabilitatea politicii noi și probabilitatea politicii vechi, $r_i = \frac{\pi(a_t|s_t)}{\pi_{\text{old}}(a_i|s_i)}$, iar A'_i este funcția de avantaj (prezentată mai jos). Termenul ϵ de clipping asigură că politica nu se va abatea prea mult de la vechea politică, întărind stabilitatea.

- **Spațiul Humanoid de Acțiune:** Pentru ca Humanoid are un spațiu continuu, PPO modelează politica ca o distribuție Gaussiană:

$$\pi(a_i|s_i) = \mathcal{N}(\mu(s_i), \sigma(s_i)),$$

, unde $\mu(s_i)$ și $\sigma(s_i)$ sunt media și deviația standard ale distribuției acțiunii, caracterizate de o rețea neurală.

- **Avantajul dobândit:** PPO foloseste Estimarea Generalizată a Avantajului (GAE) pentru a calcula funcția de avantaj A'_i:

$$A'_i = \delta_i + (\gamma\lambda)\delta_{i+1} + (\gamma\lambda)^2\delta_{i+2} + ...,$$

, unde $\delta_i = r_i + \gamma V(s_{i+1}) - V(s_i)$ este eroarea temporală diferențială, γ este factorul de reducere, iar λ are control asupra balanței dintre bias și varianță.

- **Regularizarea Entropiei:** PPO stimulează explorarea folosind un termen suplimentar:

$$L_{ENTR} = \mathbb{E}_t \left[\mathcal{H}(\pi(\cdot|s_i)) \right],$$

, unde $\mathcal{H}$ este entropia politicii.

- **Dualitate Actor-Critic:** Cadrul din PPO se împarte în actor si critic. Actorul selectează acțiunile pe baza environmentului, iar criticul estimează funcția de valoare $V(s_i)$. Pentru acest lucru criticul minimizează eroarea pătratică medie:

$$L_{AC}(\phi) = \mathbb{E}_i \left[\left(V_\phi(s_i) - R_i \right)^2 \right].$$

PPO reuseste să lege toate componentele într-un proces de optimizare unificat, schimbând tactica între colectarea clasică de date prin interacțiuni cu environmentul și optimizări ale politicii folosind descendența stochastică a gradientului din minibatch-uri.

În python PPO poate fi folosit împreună cu implementarea din packetul "stable baseline3"

```
from stable_baselines3 import PPO
```

Parametrii următori sunt utilizați pentru algoritmul Proximal Policy Optimization (PPO):

```
self.learning_rate = 3e-4 # rata de invățare a rețelei neuronale
self.n_steps = 2048 # numărul de pași din antrenarea rețelei neuronale
self.batch_size = 64 # numărul de batchuri pentru rețeaua neuronală
self.n_epochs = 10 # numărul de epoci pentru învățarea rețelei neuronale
self.gamma = 0.99 # factorul de discount
self.gae_lambda = 0.95 # lambda compromisurile dintre bias și varianță
self.clip_range = 0.2 # epsilon pentru LCLIP
self.verbose = 1 # utilizare GUI
```

Modelul din "stable baseline3" este inițializat astfel:

```
self.model = PPO(
            policy="MlpPolicy",
            env=vec_env,
            learning_rate=self.learning_rate,
            n_steps=self.n_steps,
            batch_size=self.batch_size,
            n_epochs=self.n_epochs,
            gamma=self.gamma,
            gae_lambda=self.gae_lambda,
            clip_range=self.clip_range,
            verbose=self.verbose
        )
```

Modelul este antrenat astfel:

```
def train(self, env):
    '''Train the agent based on the environment'''
    if self.model is None:
        self.initAgent(env)
    self.model.learn(total_timesteps=self.conf.N_EPISODES)
```

Pentru alegerea acțiunii viitoare în cazul testării vom apela funcția predict:

```
def choose_action(self, state):
    '''Choose an action based on the state'''
    action, _states = self.model.predict(state, deterministic=True)
    return action
```

3.3 Agentul DeepQLearning

Algoritmul DeepQLearning se bazează pe estimarea din ce în ce mai precisă a valorilor acțiunilor ce pot fi realizate dintr-o stare curentă dată. Acest lucru este realizat prin explorarea mediului și memorarea legăturilor ce există între stări și acțiuni. Algoritmul are avantajul că poate fi aplicat pe medii unde o stare este reprezentată de un tuplu de valori

$$s = (s_1, s_2, ..., s_n), \text{ unde } s_i \in \mathbb{R},\ \forall i \in \{1, 2, ..., n\},\ n \in \mathbb{N}^*$$

Algoritmul acceptă astfel medii unde stările nu mai au atribuite valori discrete, ci continue (ca urmare, numărul de stări este, de asemenea, infinit).

Pentru a putea memora informații despre un număr infinit de stări posibile este folosită o rețea neuronală, unde numărul de noduri de intrare coincide cu numărul de valori din tuplul $s = (s_1, s_2, ..., s_n)$, iar numărul de noduri de ieșire ar trebui să fie egal cu numărul de acțiuni ce pot avea loc în starea dată ca input s. Fiecare nod de ieșire ar asocia acțiunii sale un scor, urmând să selectăm o acțiune fie dacă acest scor este maximal printre valorile celorlalte noduri de ieșire, fie în mod aleatoriu (metoda $\epsilon - greedy$). Straturile interne ale rețelei pot fi unul sau mai multe și de dimensiuni variate. Mediul ales în proiect are acțiunile de forma

$$a = (a_1, a_2, ..., a_m), \text{ unde } a_j \in [l, r]\ ,\ \forall j \in \{1, 2, ..., m\},\ l, r \in \mathbb{R},\ m \in \mathbb{N}^*$$

Acest lucru duce la un număr infinit de acțiuni per stare, fiind necesară gruparea lor în funcție de valori în niște intervale (bin-uri).

Următoarele funcții au fost folosite pentru a discretiza spațiul acțiunilor:

În constructorul clasei avem mai întâi inițializările:

```
self.MIN_VALUE_ACTION = -0.4 # capatul stang (l) al intervalului de valori
self.MAX_VALUE_ACTION = 0.4 # capatul drept (r) al intervalului de valori
self.NUM_BINS_ACTION = 16

self.ENV_OBSERVATION_SPACE = 348
self.ENV_ACTION_SPACE = 17
```

Conversia de la indexul unui bin la o valoare (am ales mijlocul intervalului):

```
def convertFromBinToValue(self, bin : int) -> float:
    binSize = (self.MAX_VALUE_ACTION - self.MIN_VALUE_ACTION) / self.NUM_BINS_ACTION
    value = bin * binSize + self.MIN_VALUE_ACTION + binSize / 2
    return value
```

Conversia de la o valoare la indexul unui bin(interval):

```
def convertFromValueToBin(self, value : float) -> int:
    DELTA = 0.0001
    binSize = (self.MAX_VALUE_ACTION - self.MIN_VALUE_ACTION) / self.NUM_BINS_ACTION
    bin = (value - self.MIN_VALUE_ACTION) / binSize
    if bin == self.NUM_BINS_ACTION:
        bin -= DELTA
    return int(bin)
```

Folosind discretizarea am redus practic numărul nodurilor de ieșire al rețelei de la infinit la b^m, unde b este numărul de bin-uri, iar m este numărul de valori din tuplul $a = (a_1, a_2, ..., a_m)$. În continuare această valoare poate să fie prea mare (în cazul mediului nostru ar fi de forma b^{17}). Se mai poate aplica următoarea constrângere: dintr-o stare anume ne uitam doar la acțiunile care au exact o singură valoare din cele m nenulă. Coborâm astfel de la b^m la $b * m$, o valoare fezabilă.

Intuitiv, o acțiune care are mai mult de o valoare nenulă poate fi aproximată printr-o succesiune de 2 sau mai multe acțiuni cu doar o valoare nenulă.

Arhitectura rețelei neuronale folosite:

```
import torch

class NN(torch.nn.Module):
    def __init__(self, INPUT_DIM, OUTPUT_DIM):
        super(NN, self).__init__()

        self.INPUT_DIM = INPUT_DIM
        self.OUTPUT_DIM = OUTPUT_DIM

        self.fc1 = torch.nn.Linear(self.INPUT_DIM, 2 * self.INPUT_DIM)
        self.fc2 = torch.nn.Linear(2 * self.INPUT_DIM, 2 * self.INPUT_DIM)
        self.fc3 = torch.nn.Linear(2 * self.INPUT_DIM, 2 * self.INPUT_DIM)
        self.fc4 = torch.nn.Linear(2 * self.INPUT_DIM, self.OUTPUT_DIM)


        self.optimizer = torch.optim.Adam(self.parameters(), lr=0.001)
        self.criterion = torch.nn.MSELoss() # Mean Square Error
```

Valorile de început și de decay ale hiperparametrilor ϵ și γ:

```
self.START_EPSILON = 1.0
self.EPSILON_DECAY_RATE = 0.999
self.MIN_EPSILON = 0.1
```

```
self.EPSILON = self.START_EPSILON

self.START_GAMMA = 0.95
self.GAMMA_DECAY_RATE = 0.999
self.MIN_GAMMA = 0.1
self.GAMMA = self.START_GAMMA
```

Funcțiile folosite pentru a diminua ϵ și γ în funcție de numărul iterației în timpul învățării:

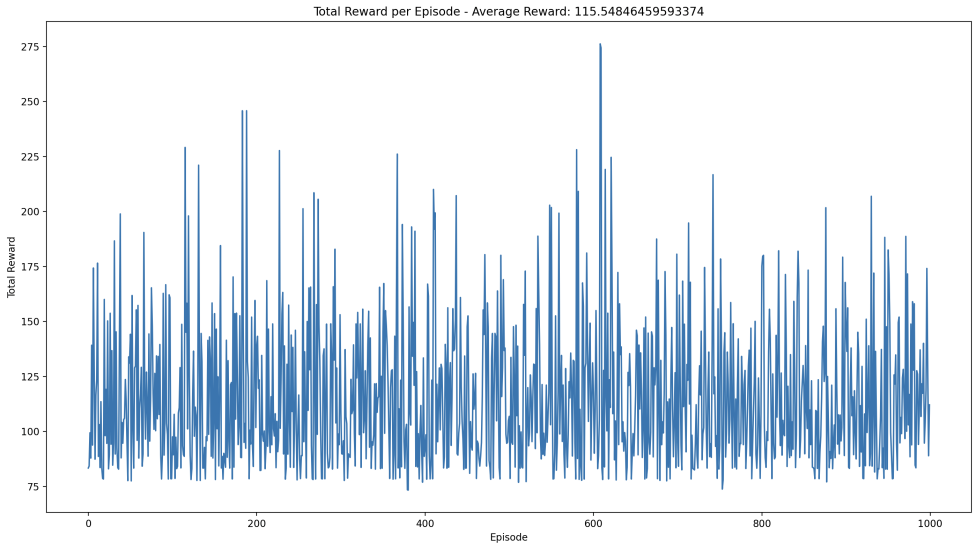
```
def updateEpsilon(self, numIteration : int) -> float:
    return max(self.MIN_EPSILON, self.START_EPSILON * \
               (self.EPSILON_DECAY_RATE ** (numIteration // 10)))

def updateGamma(self, numIteration : int) -> float:
    return max(self.MIN_GAMMA, self.START_GAMMA * \
               (self.GAMMA_DECAY_RATE ** (numIteration // 10)))
```

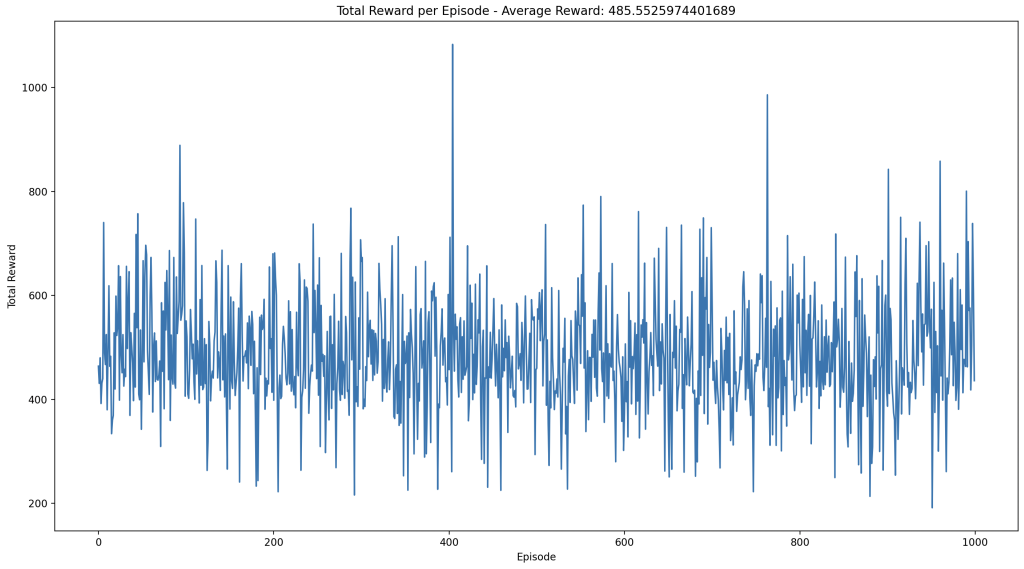
4 Compararea Rezultatelor Agenților

După antrenare, fiecare agent a fost rulat pe 1000 de episoade, în graficele de mai jos observându-se recompensa medie pe acele episoade pentru fiecare agent în parte.

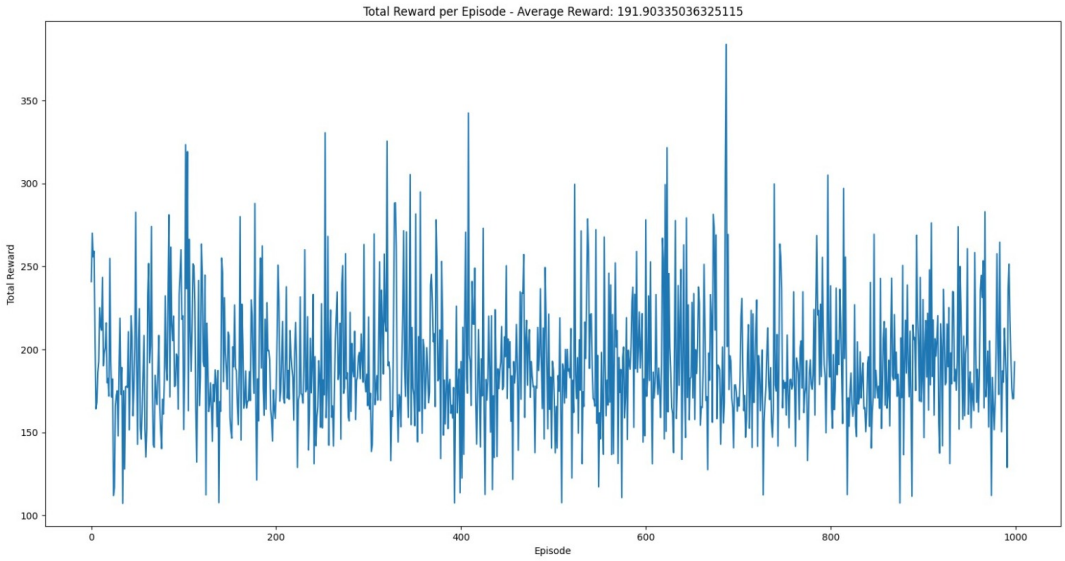
4.1 Monte Carlo



4.2 Proximal Policy Optimization



4.3 DeepQLearning



Concluzii Rulare:

La testarea cu 1000 episoade a agenților, am obținut ca recompensă totală (per episod) medie: Monte Carlo – 115.5, Deep Q Learning – 191.9, Proximal Policy Optimization – 485.5. Din cauză că mediul are acțiuni și stări cu valori reale, antrenarea necesară este îndelungată, de aceea DeepQLearning, deși cu recompensă medie mai mare decât Monte Carlo, nu are rezultate mult mai bune. În cazul agentului Monte Carlo, memoria necesară crește foarte repede, deoarece trebuie memorat un scor pentru fiecare pereche de forma (stare, acțiune) întâlnită. DeepQLearning nu prezintă acest impediment, datorită rețelei sale neuronale, în schimb necesită un timp îndelungat pentru învățare. Agentul Proximal Policy Optimization a înregistrat cele mai bune rezultate.

Anexă

https://www.gymlibrary.dev/environments/mujoco/humanoid/ - Humanoid Environment

https://github.com/Razvan48/Proiect-Reinforcement-Learning-RL - Link Repository

https://www.mathworks.com/help/reinforcement-learning/ug/proximal-policy-optimization-agents.html - Proximal Policy Optimization

Cerințe Python (pip install):

- gymnasium[mujoco]
- numpy
- torch
- stable-baselines3
- matplotlib

Alte Module utilizate:

- sys
- os
- math
- collections