

# **Proiect I.A.**

## **Algoritm de evoluție diferențială pentru antrenarea de rețele neuronale în cadrul unui joc**

Proiect realizat de:  
Laduncă Ioan-Darian  
Vîrlan Francisc-Gabriel

Profesor coordonator:  
Prof. Florin Leon

## 1. Introducere

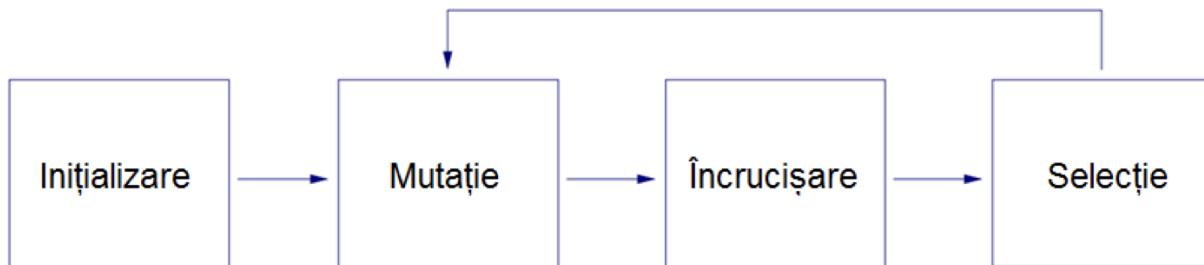
Proiectul propune implementarea unui joc cu șerpi(Snake Battles) utilizand Python si biblioteci precum TensorFlow și Pygame. Jocul implica mai mulți șerpi controlati de rețele neuronale care concurează pe o tabla. Antrenarea șerpilor e realizata cu ajutorul unui algoritm evolutiv diferential, care optimizează greutatea rețelelor neuronale pentru a maximiza performanta in joc.

Simularea permite ajustarea parametrilor algoritmului evolutiv, precum rata de mutație și rata de încrucișare, pentru a explora diverse strategii de antrenament. Proiectul evidențiază utilizarea inteligentă a metodelor evolutive pentru optimizarea performanței agenților virtuali(șerpilor) într-un mediu competitiv.

## 2. Aspecte teoretice

Algoritmul de evoluție diferențială reprezintă o metodă euristică de optimizare, similară algoritmilor evolutivi clasici, dar care se distinge prin ordinea diferită a aplicării operatorilor de selecție, încrucișare și mutație. Modul particular de generare a noilor cromozomi constituie o trăsătură specifică acestui algoritm, implicând adăugarea diferenței dintre doi cromozomi la un al treilea, iar rezultatul fiind ulterior comparat cu un al patrulea cromozom.

Ordinea de aplicare a operațiilor:



Populația este inițializată în mod aleator, fiecare individ fiind reprezentat de un set de ponderi, unde fiecare pondere este limitată inferior și superior, acoperind spațiul parametrilor în mod uniform.

Se poate observa în codul de mai jos generarea:

```
while len(population) < const.POP_SIZE:
    population.append(np.random.uniform(-1, high: 1, const.WEIGHTS_SIZE))
```

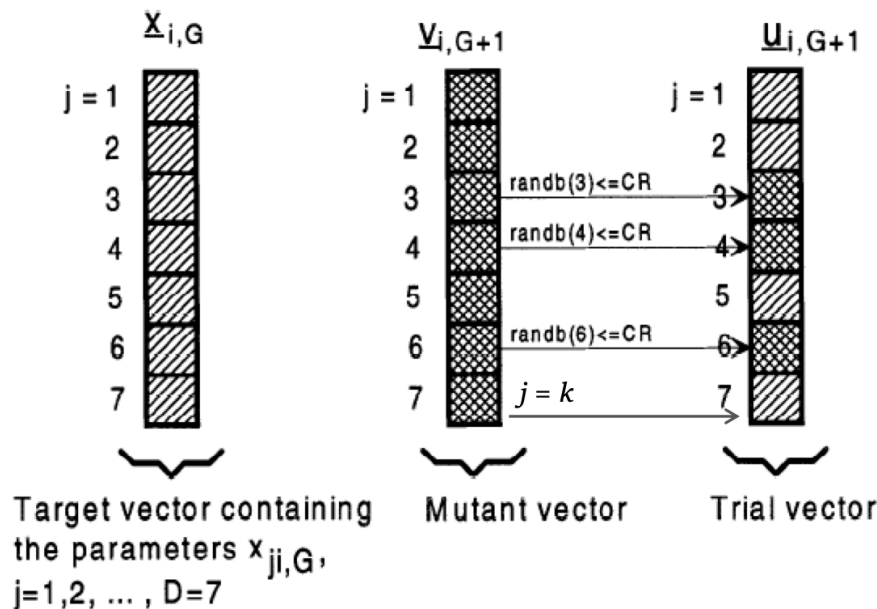
În cea ce privește mutația, este folosită următoarea formulă:

■ **rand/1:**

$$■ \quad v_i = x_{r_1} + F \cdot (x_{r_2} - x_{r_3})$$

Această formulă presupune calcularea diferenței dintre doi indivizi și adăugarea acestuia la un al treilea individ. Acest tip de mutație este cunoscut drept mutație auto-referențială. Pe măsură ce algoritmul evoluează, diferențele se ajustează în funcție de domeniul problemei.

După mutație, cromozomul trece printr-un proces de încrucișare, unde este combinat cu un cromozom părinte pentru a forma un cromozom de încercare. Proporția genelor preluate din cromozomul mutant este determinată de rata de încrucișare (CR), în timp ce genele rămase provin de la părintele original. Pentru a garanta că cel puțin o genă din mutant este inclusă, se alege un index aleatoriu care asigură transferul obligatoriu al unei gene în cromozomul de încercare.



Iar în ultima fază se ajunge la selecție. Pentru aceasta se este necesară calcularea fitness-ului indivizilor. Acest lucru se determina prin instanțierea a unor modele bazate pe rețele neuronale, a căror ponderi sunt tocmai genele unui individ. Câte patru, indivizii sunt folosiți într-o simulare, în cazul nostru simularea este un joc competitiv între ei, unde acțiunile lor sunt premiate sau penalizate, iar la final se calculează un scor pentru fiecare, cu ajutorul careia se poate face selecția celui mai apt individ, și folosirea lui în următoarea generație.

### 3. Modalitate de rezolvare

Prima etapă în dezvoltarea aplicației a fost analizarea algoritmului, și propunerea de teme care s-ar putea încadra în folosința unui algoritm evolutiv. De aici reiese că algoritmi evolutivi sunt potriviți problemelor caracterizate prin funcții complexe și de dimensiuni mari. Ne-am gândit că jocurile sunt un exemplu perfect de funcții complexe cu mulți parametri și multe posibilități. Ideea jocului nu este tocmai originală, există diferite implementări multiplayer online sub diferite nume (splix.io de exemplu), însă abordarea noastră se va întâmpla local iar numărul de șerpi este limitat la 4.

Regulile jocului sunt următoarele:


- Te poți deplasa în orice direcție
- Dacă îți muști coada, aria delimitată de perimetrul închis a corpului șarpelui devine teritoriu cucerit
- Orice teritoriu poate fi cucerit (teritoriu deja cucerit sau teritoriu necucerit)
- Încălcarea bordurilor duce la apariția pe partea opusă a hărții
- Dacă alt șarpe îți mușcă coada, șarpele care a fost mușcat este eliminat din joc (doar cozile sunt considerate puncte vulnerabile, teritoriile cucerite nu sunt considerate drept cozi)
- Finalizarea jocului este când un șarpe a cucerit 50%+ din hartă

La nivel de implementare am început cu jocul. Fisierul `game.py` conține funcția în care se întâmplă bucla principală a jocului. Aici logica jocului este procesată în felul următor:

- Fiecare șarpe este inițializat pe o poziție aleatoare, fiecare într-un cadran
- În bucla principală se verifică starea fiecărui șarpe și în funcție de (inițial input-ul unui jucător) direcția prezisă de model se alege o direcție și se actualizează în mod corespunzător fiecare șarpe.
- În funcție de efectul pe care îl are mișcarea unui șarpe, se actualizează tabla
- La final se desenează totul pe ecran

Notes: după implementarea rețelei neuronale au fost adăugate metode de premiere/penalizare a șerpilor

În cea ce privește clasa șarpe, folosită pentru managerierea entităților de acel tip ea se regăsește în snake.py și are diagrama următoare:



```
<<class>> Snake
- position: tuple
- direction: const.Dir
- last_direction: const.Dir
  - color: tuple
  - index: int
  - fill_index: int
  - length: int
- max_length: int
  - score: int
  - fill_score: int
  - is_dead: bool
  - history: list
  - bite_line: int
  + kill(): void
+ punish_sides(predicted_direction: const.Dir): void
  + detect_circle(): void
    + change_direction(key: int): void
  + change_ai_direction(direction: const.Dir): void
    + update(): void
```

Pentru ușurarea programării funcțiile importante jocului sau ajutătoare au fost scrise într-un fișier separat numit functions.py, acesta conține următoarele definiții:

- flood\_fill(grid, index, fill\_index): implementarea algoritmului de umplere, acesta pornește de la bordurile hărții marcând toate pozițiile care nu se afla înăuntru a unui delimitator. După umple toate zonele rămase cu fill\_index și șterge marcurile folosite inițial
- draw\_grid(grid, snakes): parcurge matricea, desenând fiecare element folosind culoarea corespunzătoare
- get\_data(grid, snake): normalizează datele ce urmează a fi date unui model
- to\_direction(move): transforma un număr într-o direcție
- is\_opposite(dir\_1, dir\_2): verifică dacă două direcții sunt opuse

Pentru generarea modelelor ne-am folosit de o rețea neuronală de tip feedforward cu două straturi ascunse. Drept input-uri am folosit matricea jocului(20x20, dar se poate folosi și una mai mare), un vector de direcții, poziția capului șarpelui și lungimea șarpelui, adică 407 input-uri. Straturile sunt complet conectate între ele, iar funcția de activare folosită este ReLU(Rectified Linear Unit), aceasta “rectifică” valorile negative, setându-le la zero, și păstrează valorile pozitive neschimbate. Pentru output avem 4 probabilități, fiecare reprezentând probabilitatea modelului de a alege o direcție. În acest scop folosim funcția de activare Softmax, care transformă valorile prezise într-o distribuție de probabilitate.

```
def create_model(input_size): 2 usages  virianfrancisc
    model = Sequential([
        Dense(units=64, activation='relu', input_shape=(input_size,)),
        Dense(units=32, activation='relu'),
        Dense(units=4, activation='softmax')
    ])
    return model
```

```
def predict_move(model, grid, snake): 1
    data = func.get_data(grid, snake)
    data = tf.convert_to_tensor([data])
    predictions = model(data)
    return predictions[0].numpy()
```

Mai sus sunt funcțiile de creare și prezicere a output-ului pentru modele. Acestea se regăsesc în fișierul ai.py, împreună cu alte funcții folosite de scrierea și încărcarea ponderilor din fișiere.

În final secția dedicată algoritmului evolutiv diferențial, aceasta se găsește în evolution.py și are implementarea descrisă la punctul 2.

Notes: În fișierul constants.py se pot găsi variabilele pentru a modifica și customiza anumiți câmpuri din aplicație

## 4. Componente semnificative

Revenind de la punctul anterior, componenta cea mai semnificativă este tocmai algoritmul de evoluție diferențială, acesta fiind separat în 5 funcții, una principală și 4 ajutoare pentru a nu aglomera bucla în care epocile sunt parcurse.

### I. Generarea:

```
def initialize_population():
    population = []
    if os.path.exists(const.FILE_IN):
        best_weights = np.load(const.FILE_IN)
        population.append(best_weights)

    while len(population) < const.POP_SIZE:
        population.append(np.random.uniform(-1, high: 1, const.WEIGHTS_SIZE))

    return population
```

Populația în acest caz este o lista de indivizi, fiecare individ fiind reprezentat de o listă de ponderi, în cazul de mai sus am optat să adaug și opțiunea de a începe de la o generație salvată anterior.

### II. Mutarea:

```
def mutation(population, target):
    indices = list(range(len(population)))
    indices.remove(target)
    r1, r2, r3 = np.random.choice(indices, size: 3, replace=False)
    individual = population[r1] + const.F * (population[r2] - population[r3])
    return individual
```

Se exclude individul curent din posibilitatea de a genera noul individ, se aleg aleator 3 indivizi, si se aplică formula pentru a genera nou individ.

### III. Încrucișarea:

```
def crossover(target, mutant):
    trial = np.copy(target)
    k = np.random.randint(len(target))
    for j in range(len(target)):
        if np.random.rand() < const.CR or j == k:
            trial[j] = mutant[j]
    return trial
```

Individul curent este supus la încrucișare prin atribuirea genelor mutantului la individ. Pentru a fi siguri că se transferă măcar o genă, se alege un număr aleator, iar gena de pe acea poziție este transferată cu siguranță.

### IV. Selecția:

```
def fitness(weights):
    fitness_scores = []
    for i in range(0, len(weights), const.NR_SNAKES):
        chunk = weights[i:i + const.NR_SNAKES]
        models = []
        for model_weights in chunk:
            model = ai.create_model(const.INPUTS)
            ai.set_weights(model, model_weights)
            models.append(model)
        scores = game.run(models)
        fitness_scores.extend(scores[:len(chunk)])
        # print(f"simulation {(i / 4) + 1} completed")
    return fitness_scores
```



Se calculează valoarea de fitness a indivizilor noi împreună cu încă 3, asta pentru că simularea noastră necesită de 4 indivizi care joacă deodată. În cazul de mai sus, pentru a putea calcula score-urile la o populație mai mare, se parcurge vectorul de indivizi câte 4, de asta recomandăm o populație în număr multiplu de 4.

```
trial_scores = fitness(trial_population)
trial_fitness = trial_scores[-1]
#selection:
if trial_fitness > fitness_scores[i]:
    new_population.append(trial)
else:
    new_population.append(population[i])
```

La revenirea din funcția fitness se face selecția propriu zisă, se ia cel mai apt individ din simulare, și se compară cu individul curent din populație, iar cel mai bun dintre cei 2 este adăugat în populația următoare.

## V. Evoluția diferențială:

```
def differential_evolution(epochs):
    population = initialize_population()
    best_fitness_mean = []
    for epoch in range(epochs):
        new_population = []
        fitness_scores = fitness(population)
        for i in range(const.POP_SIZE):
            # print("mutation in progress...")
            individual = mutation(population, i)
            # print("mutation complete.")

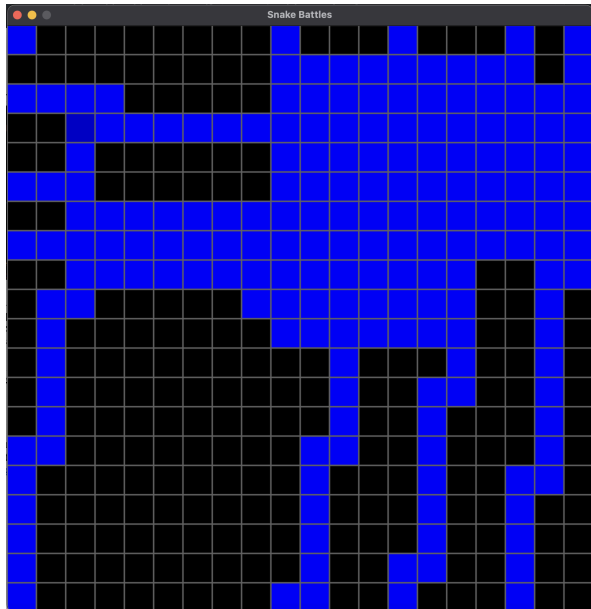
            # print("crossover in progress...")
            trial = crossover(population[i], individual)
            trial_population = population[:const.NR_SNAKES - 1]
            trial_population.append(trial)
            # print("crossover complete.")

            # print("selection in progress...")
            trial_scores = fitness(trial_population)
            trial_fitness = trial_scores[-1]
            #selection:
            if trial_fitness > fitness_scores[i]:
                new_population.append(trial)
            else:
                new_population.append(population[i])
            # print("selection complete.")
        population = new_population
        best_fitness = max(fitness_scores)
        best_fitness_mean.append(best_fitness)
        print(f"Epoch {epoch + 1}/{epochs}, best fitness: {best_fitness}")
    fitness_scores = fitness(population)
    best_index = np.argmax(fitness_scores)
    best_weights = population[best_index]
    np.save(const.FILE_OUT, best_weights)
    print(f"generation {const.F2}, average fitness {np.mean(best_fitness_mean)}")
```

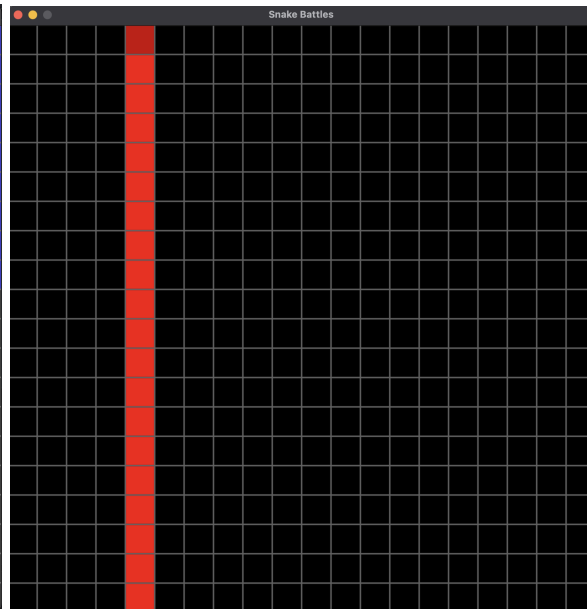
Aici este scrisă ordinea pașilor de urmat pentru a aplica algoritmul, împreună cu diferite print-uri pentru a putea urmări mai ușor progresul din consolă.

Funcția este apelată în main.py cu o valoare de 10 epoci timp de mai multe generații.

## 5. Rezultatele

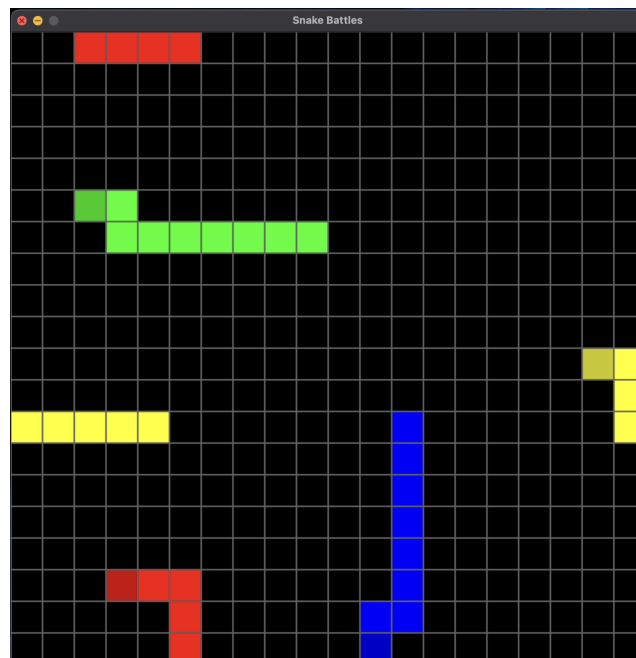


Exemplu de individ apt.



Exemplu de individ inapt.

În primul caz individului îi se vor acorda puncte pentru faptul că acoperă mare parte din ecran și a completat dreptunghiuri. În al doilea caz, individul este penalizat pentru decizia sa repetată și prin faptul că nu acoperă arii pe ecran.



Începutul unei simulări.

Fiecare individ, se mișcă în funcție de mutarea prezisă de model, de obicei nu foarte bună.

```
Epoch 1/10, best fitness: 92.5
Epoch 2/10, best fitness: 141.5
Epoch 3/10, best fitness: 122.5
Epoch 4/10, best fitness: 90.0
Epoch 5/10, best fitness: 104.0
Epoch 6/10, best fitness: 103.0
Epoch 7/10, best fitness: 98.0
Epoch 8/10, best fitness: 92.5
Epoch 9/10, best fitness: 119.0
Epoch 10/10, best fitness: 76.5
generation 26, average fitness 103.95
```

Rezultate în urma antrenării unei generații.

## 6. Concluzii

Într-un final putem spune că există potențial, totuși, limitările hardware și logice ale jocului (durata maximă alocată unei simulări), impun mari piedici observării evoluției acestor modele. Cea ce rezultă într-o evoluție lentă și rezultate puțin observabile. În cazul unei aplicații mai potrivite și cu un grad de optimizare mai mare, suntem încrezători că o integrare a algoritmului evolutiv diferențial și a rețelelor neuronale poate aduce rezultate surprinzătoare și un grad de rafinare a indivizilor dintr-o populație mult peste un model de rețea neuronală, cu ajustări ale ponderilor în varianta clasică.

## 7. Bibliografie

- Curs Inteligență artificială Metode de optimizare II:  
[http://florinleon.byethost24.com/Curs\\_IA/IA05\\_Optimizare2.pdf?i=1](http://florinleon.byethost24.com/Curs_IA/IA05_Optimizare2.pdf?i=1) - Florin Leon
- Laboratoare Inteligență artificială - Florin Leon
- Tutorial neural network <https://www.geeksforgeeks.org/feedforward-neural-network/>

## 8. Contribuția studenților

-Lăduncă Ioan-Darian: Proiectarea, analiza, și implementarea algoritmului evolutiv diferențial, documentația, code review

-Vîrlan Francisc-Gabriel: Proiectarea și analiza algoritmului evolutiv diferențial, documentația, proiectarea și implementarea logicii jocului

Github: [https://github.com/Fenris24/IA\\_Proiect.git](https://github.com/Fenris24/IA_Proiect.git)