

**Universitatea Tehnică “Gheorghe Asachi” din Iași**  
**Facultatea de Automatică și Calculatoare**

**Inteligență artificială**  
**Iris predicție**  
**Rețele neuronale de tip perceptron multistrat cu algoritm evolutiv**

**Studenti,**  
**Caraman Talida**  
**Pricop Matei-Ioan**  
**Vieru Iosif**

**Coordonator,**  
**Florin Leon**

## Cuprins

1. Descrierea problemei considerate.....	2
2. Aspecte teoretice privind algoritmul.....	2
2.1. Retele neuronale perceptron multistrat.....	2
2.2. Algoritmi evolutivi.....	3
2.2.1. Caracteristici cheie.....	3
2.2.2. Tipuri de algoritmi evolutivi.....	4
3. Modalitatea de rezolvare.....	4
3.1. Preprocesarea datelor.....	4
3.2. Definirea rețelei neuronale.....	5
3.3. Implementarea algoritmului evolutiv.....	5
3.4. Testarea rețelei și afisarea rezultatelor.....	6
4. Secțiuni semnificative ale codului sursa.....	6
4.1. Modulul principal.....	6
4.2. Procesarea datelor.....	7
4.2.1. Incarcarea datelor.....	8
4.2.2. Normalizarea datelor.....	8
4.2.3. Transformarea etichetelor.....	8
4.2.4. Amestecarea datelor.....	8
4.2.5. Impartirea datelor.....	9
4.3. Algoritmul evolutiv.....	9
4.3.1. Funcția de fitness.....	9
4.3.2. Mutatie.....	9
4.3.3. Selectie.....	10
4.3.4. Incrucisare.....	10
4.3.5. Algoritm.....	11
5. Rezultate și observații.....	11
5.1. Acuratatea și eroarea pe setul de antrenare și setul de test.....	11
5.2. Predictie clasă.....	12
5.3. Predictie clasă cu perturbări.....	12
5.4. Teste automate.....	12
6. Concluzii.....	18
7. Bibliografie.....	18
8. Contribuțiile membrilor echipei.....	18

## 1. Descrierea problemei considerate

**Tema** proiectului este antrenarea unei rețele neuronale de tip perceptron multistrat cu o structură predefinită (un număr specificat de straturi ascunse și neuroni) cu ajutorul unui algoritm evolutiv. Mai exact un strat ascuns cu 10 neuroni.

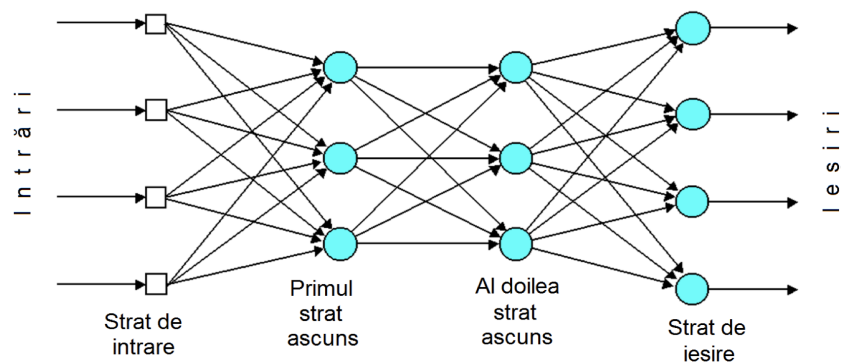
Mai întâi, am ales o problemă de clasificare pe care să o învețe rețeaua neuronală. Am ales o problemă din [UCI Machine Learning Repository](#), mai exact [Iris - UCI Machine Learning Repository](#). Aceste date le-am împărțit în date de antrenare și date de test (raport 80%-20%). Am utilizat algoritmul evolutiv pentru determinarea ponderilor conexiunilor și pragurilor neuronilor din rețea. În final, am afișat rezultatele obținute de rețea pentru mulțimea de antrenare a problemei considerate dar și pentru mulțimea de test.

## 2. Aspecte teoretice privind algoritmul

### 2.1. Rețele neuronale perceptron multistrat

Perceptronul multi-strat este o rețea neuronală cu propagare înainte (feed-forward) cu unul sau mai multe straturi ascunse. Rețeaua este compusă din următoarele straturi:

- Un strat de intrare care primește datele brute
- Unul sau mai multe straturi ascunse care procesează informația
- Un strat de ieșire care furnizează rezultatele predicției



Calcululele se realizează numai în neuronii din straturile ascunse și din stratul de ieșire iar semnalele de intrare sunt propagate înainte succesiv prin straturile rețelei. Fiecare neuron dintr-un strat este conectat la toți neuronii din stratul următor prin ponderi ("weights"). Fiecare conexiune are o valoare numerică, iar în timpul antrenării, algoritmul ajustează aceste valori pentru a minimiza eroarea predicției.

Funcția de activare utilizată este sigmoida unipolară care mapează valorile în intervalul (0, 1):

$$f(x) = \frac{1}{1 + e^{-x}}$$

Reteaua funcționează astfel:

1. Primește vectorii de intrare și calculează vectorii de ieșire.
2. Dacă există o eroare (diferența dintre ieșirea dorită și ieșirea calculată), ponderile sunt ajustate pentru a reduce această eroare. Această ajustare se realizează utilizând **algoritmul evolutiv**, care optimizează atât ponderile, cât și pragurile neuronilor.

## 2.2. Algoritmi evolutivi

Algoritmii evolutivi sunt metode de optimizare inspirate din principiile selecției naturale și evoluției biologice. Ei lucrează cu o populație de soluții candidate, fiecare reprezentând o posibilă rezolvare a problemei. Prin aplicarea repetată a unor operatori precum selecția, încrucișarea și mutația, algoritmi îmbunătățesc treptat calitatea soluțiilor, favorizând pe cele mai potrivite conform unei funcții de evaluare.

În cadrul algoritmilor, problema de optimizare reprezintă mediul din natură, soluțiile potențiale corespunzând indivizilor care trăiesc în acest mediu. Calitatea soluțiilor (evaluată prin funcția de fitness) este analogă cu gradul de adaptare al indivizilor la mediul lor.

### 2.2.1. Caracteristici cheie

#### 1. Populația

Este un ansamblu de soluții candidate care evoluează de-a lungul timpului. Dimensiunea populației influențează atât diversitatea soluțiilor, cât și eficiența algoritmului.

#### 2. Funcția de fitness

Evaluează performanța fiecărei soluții și ghidează procesul de selecție și evoluție pentru a îmbunătăți calitatea soluțiilor.

#### 3. Operatori genetici

- **Selecția**

Determină părinții pentru reproducere, luând în considerare

valoarea functiei de fitness. Exemple: selectia ruleta, selectia competitiva sau elitismul.

- **Incrucisarea**

Genereaza solutii noi prin combinarea caracteristicilor parintilor. Poate fi realizata prin metode precum incrucisarea cu un punct, cu mai multe puncte, uniforma sau aritmetica.

- **Mutatia**

Introduce diversitate prin modificarea aleatorie a unor componente ale solutiilor, reducand riscul stagnarii algoritmului intr-un optim local.

#### 4. Criteriul de oprire

Algoritmul se opreste dupa un numar fix de generatii, atunci cand solutiile converg sau cand nu mai apar imbunatatiri semnificative.

#### 2.2.2. Tipuri de algoritmi evolutivi

- Algoritmi genetici: Utilizeaza codificare binara pentru solutii.
- Strategii evolute: Codifica solutiile ca vectori de numere reale.
- Programare genetica: Solutiile sunt reprezentate ca programe sau arbori de operatii.

### 3. Modalitatea de rezolvare

Pasi:

#### 3.1. Preprocesarea datelor

Setul de date Iris, a fost importat folosind biblioteca UCIMLRepo care asigura accesul la seturi de date standardizate din colectia UCI Machine Learning Repository. Acesta contine masuratori ale lungimii si latimii petalelor si sepalei, impreuna cu etichete ce reprezinta specia florii.

Datele au fost normalizate in intervalul  $[0, 1]$  pentru a fi compatibile cu functia sigmoida, iar etichetele au fost transformate intr-o reprezentare "one-hot encoding". Astfel, fiecare specie de flori este reprezentata ca un vector binar de dimensiune 3.

Setul a fost impartit in proportii de 80-20 in date de antrenare si date de testare.

### 3.2. Definirea rețelei neuronale

Reteaua neuronală este structurată în trei straturi:

Stratul de intrare - are patru neuroni și corespunde cu cele patru caracteristici din setul de date Iris anume lungimea lungimea / lățimea sepalei și petalei.

Stratul ascuns - conține zece neuroni.

Stratul de ieșire - are trei neuroni, câte unul pentru fiecare clasă (Setosa, Versicolor și Virginica)

### 3.3. Implementarea algoritmului evolutiv

Am inițializat populația cu 50 de indivizi generați aleator. Fiecare individ reprezintă un cromozom, adică un vector de dimensiune:

$$nr\_ponderi\_ascuns + nr\_praguri\_ascuns + nr\_ponderi\_iesire + nr\_praguri\_iesire$$

Apoi, pentru un număr de generații (în cazul nostru 1000), aplicăm operatorii evolutivi. Procesul include următorii pași principali:

Calcularea fitness-ului: fiecare individ este evaluat prin propagarea datelor de antrenare în rețeaua neuronală asociată cromozomului său. Fitness-ul fiecărui individ este determinat de eroarea medie pătratică (MSE), calculată ca diferența dintre valorile dorite și predicțiile rețelei.

Crearea unei noi populații:

- Selecția părinților: pentru fiecare pereche de părinți, selectăm cei mai buni indivizi utilizând metoda turnirului. Aceasta implică alegerea aleatorie a unui subset de indivizi și selectarea celui cu cel mai mare fitness.
- Încrucișarea: genele a doi cromozomi selectați sunt combinate pentru a produce doi descendenți. Acest lucru se realizează prin selectarea unui punct de tăiere aleator și combinarea părților din ambii părinți.
- Mutatia: fiecare genă a descendenților are o probabilitate de 5% de a fi modificată aleator. Aceasta etapă introduce diversitate în populație și ajută la evitarea blocării în minime locale.

### 3.4. Testarea rețelei și afisarea rezultatelor

În final am aplicat propagarea înainte pentru setul de date de testare și am calculat două metrici:

- Eroarea medie pătratică (MSE) este calculată pe baza predicțiilor obținute și a valorilor dorite.
- Acurateta este calculată în felul următor: predicțiile sunt comparate cu clasele reale, determinând procentul de exemple clasificate corect pentru ambele seturi

Totodată, am testat și pentru un exemplu particular (de ex. o floare Iris cu valori specifice) este normalizat și introdus în rețea neuronală, iar rețeaua returnează clasa prezisă.

## 4. Secțiuni semnificative ale codului sursă

### 4.1. Modulul principal

Modulul principal integrează toate componentele necesare pentru prelucrarea datelor, definirea rețelei neuronale și optimizarea acesteia utilizând un algoritm evolutiv.

Python

```
# 1. incarcarea si prelucrarea datelor
X_train, X_test, y_train, y_test = incarca_si_proceseaza_date()

# 2. initializare retea neuronală
retea: ReteaNeuronală = ReteaNeuronală(neuroni_intrare=4,
neuroni_strat_ascuns=10, neuroni_iesire=3)

# 3. rularea algoritmului evolutiv
dimensiune_populatie = 50
generatii = 1000
rata_mutatie = 0.05

cel_mai_bun_cromozom= algoritm_genetic(
    retea=retea,
    X_train=X_train,
    y_train=y_train,
    dimensiune_populatie=dimensiune_populatie,
    generatii=generatii,
    rata_mutatie=rata_mutatie
)
```

```

# 4. testarea si afisarea rezultatelor
#predictii_train = retea.propagare_inainte(X_train, cel_mai_bun_cromozom)
predictii_test = retea.propagare_inainte(X_test, cel_mai_bun_cromozom)
mse_test = np.mean((y_test - predictii_test) ** 2)

predictii_train = retea.propagare_inainte(X_train, cel_mai_bun_cromozom)
mse_train = np.mean((y_train - predictii_train) ** 2)

# Pentru setul de antrenare
predictii_clase_train = np.argmax(predictii_train, axis=1)
clase_reale_train = np.argmax(y_train, axis=1)
acuratete_train = np.mean(predictii_clase_train == clase_reale_train)

# Pentru setul de test
predictii_clase_test = np.argmax(predictii_test, axis=1)
clase_reale_test = np.argmax(y_test, axis=1)
acuratete_test = np.mean(predictii_clase_test == clase_reale_test)

print("Acuratetea pe setul de antrenare:", acuratete_train)
print("Acuratetea pe setul de test:", acuratete_test)

# Rezultate
print(f"EROARE TRAIN (MSE): {mse_train}")
print(f"EROARE TEST (MSE): {mse_test}")

iris_exemplu = np.array([5.1, 3.5, 1.4, 0.2]) # Iris-setosa

# normalizare
iris_exemplu_normalizat = ((iris_exemplu - X_train.min(axis=0)) /
(X_train.max(axis=0) - X_train.min(axis=0))).values

predictie = retea.propagare_inainte(iris_exemplu_normalizat.reshape(1, -1),
cel_mai_bun_cromozom)

clasa_predusa = np.argmax(predictie)
clase = ["Iris-setosa", "Iris-versicolor", "Iris-virginica"]
nume_clasa_predusa = clase[clasa_predusa]

print(f"Iris : {iris_exemplu}")
print(f"Predictia: {predictie}")
print(f"Clasa : {nume_clasa_predusa}")

```



## 4.2. Procesarea datelor

### 4.2.1. Incarcarea datelor

Codul de mai jos incarca datele folosind biblioteca **ucimlrepo**, variabila X contine caracteristicile setului de date, iar Y contine etichetele. Functia `fetch_ucirepo` importa datele din [Iris - UCI Machine Learning Repository](#).

Python

```
iris = fetch_ucirepo(id=53)
X = iris.data.features
y = iris.data.targets
```

### 4.2.2. Normalizarea datelor

Normalizeaza datele intre [0, 1].

Python

```
X_min = X.min(axis=0)
X_max = X.max(axis=0)

X_normalizat = 0.8 * (X - X_min) / (X_max - X_min) + 0.1
```

### 4.2.3. Transformarea etichetelor

Transforma etichetele intr-o reprezentare One Hot Encoded.

Python

```
encoder = OneHotEncoder(sparse_output=False)
y_encoded = encoder.fit_transform(y.values.reshape(-1, 1))
```

### 4.2.4. Amestecarea datelor

Datele sunt ordonate in functie de eticheta, iar pentru impartirea lor in seturi de antrenare si testare avem nevoie de o randomizare a acestora.

Python

```
indici = np.arange(len(X_normalizat))
np.random.seed(42)
np.random.shuffle(indici)
```

```
X_amestecat = X_normalizat.iloc[indici]
X_amestecat = X_amestecat.reset_index(drop=True)
y_amestecat = y_encodat[indici]
```

#### 4.2.5. Impartirea datelor

Datele sunt impartite in date de antrenare si date de testare cu un raport de 80-20.

Python

```
dimensiune_antrenare = int(0.8 * len(X_amestecat))
X_antrenare, X_test = X_amestecat[:dimensiune_antrenare],
X_amestecat[dimensiune_antrenare:]
y_antrenare, y_test = y_amestecat[:dimensiune_antrenare],
y_amestecat[dimensiune_antrenare:]
```

### 4.3. Algoritmul evolutiv

#### 4.3.1. Functia de fitness

Functia calculeaza fitness-ul unui cromozom in functie de performanta acestuia pe reseaua neuronală. Aceasta este calculata pe baza erorii medii patratice in retea.

Python

```
def fitness(cromozom, retea, X_train, y_train):
    predictii = retea.propagare_inainte(X_train, cromozom)
    eroare = np.mean((y_train - predictii) ** 2)
    return -eroare
```

#### 4.3.2. Mutatie

Functia modifica cromozomii prin mutatii aleatoare aplicate cu o probabilitate specificata.

Python

```
def mutatie(cromozom, rata_mutatie=0.1):
```

```

for i in range(len(cromozom)):
    if np.random.rand() < rata_mutatie:
        cromozom[i] += np.random.uniform(-0.5, 0.5)
return cromozom

```

#### 4.3.3. Selectie

Functia selecteaza doi parinti utilizand metoda turneului, pe baza valorilor fitness.

Python

```

def selectie_parinti(populatie, fitnessuri, k=2):
    parinti = []
    for i in range(2):
        membri = []
        for i in range(k):
            indice = np.random.randint(0, len(populatie))
            membri.append((indice, fitnessuri[indice]))
        parinti.append(populatie[ max(membri, key=lambda x: x[1])[0]])
    return parinti

```

#### 4.3.4. Incrucisare

Functia combina doi parinti pentru a genera doi copii prin selectarea unui punct de incrucisare.

Python

```

def incrucisare(parinte1, parinte2):
    punct = np.random.randint(1, len(parinte1) - 1)
    copil1 = np.concatenate((parinte1[:punct], parinte2[punct:]))
    copil2 = np.concatenate((parinte2[:punct], parinte1[punct:]))
    return copil1, copil2

```

#### 4.3.5. Algoritm

Functia implementeaza procesul complet al algoritmului genetic pentru optimizarea rețelei neuronale, incluzand initializare, selectie, incrucisare si mutatii, pe mai multe generatii.

Python

```
def algoritm_genetic(retea, X_train, y_train, dimensiune_populatie=50,
generatii=100, rata_mutatie=0.1):
    fitness_rezultate = []
    populatie = initializarea_populatiei(dimensiune_populatie,
rete_a.dimensiune_cromozom)
    for generatie in range(generatii):
        for cromozom in populatie:
            fitness_rezultate = [fitness(cromozom, retea, X_train, y_train)
for cromozom in populatie]
        for _ in range(dimensiune_populatie // 2):
            noua_populatie = []
            parinte1, parinte2= selectie_parinti(populatie, fitness_rezultate)
            copil1, copil2 = incrucisare(parinte1, parinte2)
            copil1 = mutatie(copil1, rata_mutatie)
            copil2 = mutatie(copil2, rata_mutatie)
            noua_populatie.append(copil1)
            noua_populatie.append(copil2)
            populatie = noua_populatie
    return populatie[np.argmax(fitness_rezultate)]
```

## 5. Rezultate si observatii

### 5.1. Acuratatea si eroarea pe setul de antrenare si setul de test

```
Acuratețea pe setul de antrenare: 0.975
Acuratețea pe setul de test: 0.9666666666666667
EROARE TRAIN (MSE): 0.014541144648777609
EROARE TEST (MSE): 0.018213273180513974
```

Reteaua neuronală prezintă o acuratețe ridicată de 97.5% pentru setul de antrenare, respectiv 96.6% pentru setul de test, ceea ce indică o generalizare foarte bună a modelului și că modelul nu este supraantrenat.

Eroarea pe setul de antrenare și eroarea pe setul de testare sunt reduse, ceea ce sugerează faptul că predicțiile sunt precise iar diferența mică dintre cele două erori indică stabilitatea modelului.

### 5.2. Predictie clasa

```
Date input: [5.1 3.5 1.4 0.2]
probabilitati pentru fiecare clasa:
Iris-setosa: 0.62
Iris-versicolor: 0.00
Iris-virginica: 0.00
clasa prezisa: Iris-setosa
```

Predictia este in concordanta cu caracteristicile tipice ale clasei Iris-setosa, modelul indicand in mod corect cu o probabilitate de 62% pentru clasas Setosa si 0% pentru celelalte doua clase, observandu-se o separare clara intre clase.

### 5.3. Predictie clasa cu perturbari

```
Date input perturbate: [5.08866302 3.45063115 1.43984808 0.21376038]
probabilitati pentru fiecare clasa:
  Iris-setosa: 0.37
  Iris-versicolor: 0.01
  Iris-virginica: 0.00
clasa prezisa: Iris-setosa
```

La introducerea unor variatii in datele de intrare folosite anterior, reseaua continua sa prezica corect clasa corecta dar cu o probabilitate mai scazuta, de 37%, cu o diferenta clara intre celelalte doua clase care sun aproape de zero, respectiv 0%. Rezultatul arata ca modelul este rezistent la variatii minore, fiind subliniata sensibilitatea acestuia la schimbarile in datele de intrare.

### 5.4. Teste automate

Testele automate demonstreaza buna functionare a functiilor. Sunt implementate folosind biblioteca **unittest** si acopera functionalitatile esentiale ale proiectului.

#### 1. Test initializare populatie

Acest test verifica daca functia **initializarea\_populatiei** creeaza corect o populatie de cromozomi cu dimensiunea dorita.

Python

```
def test_initializare_populatie(self):
    populatie_test = initializarea_populatiei(10, 5)
    self.assertEqual(len(populatie_test), 10)
    self.assertEqual(len(populatie_test[0]), 5)
```

Explicatie: Se creeaza o populatie de 10 cromozomi cu dimensiunea 5 si se verifica dimensiunea populatiei si dimensiunea fiecarui cromozom.

#### 2. Test dimensiune cromozom

Testeaza calculul corect al dimensiunii cromozomului bazat pe arhitectura retelei neuronale (numarul de neuroni de intrare, ascunsi si de iesire).

Python

```
def test_dimensiune_cromozom(self):
    # calculam manual dimensiunea unui cromozom
    neuroni_intrare = 4
    neuroni_strat_ascuns = 10
    neuroni_iesire = 3

    nr_ponderi_ascuns = neuroni_intrare * neuroni_strat_ascuns
    nr_praguri_ascuns = neuroni_strat_ascuns
    nr_ponderi_iesire = neuroni_strat_ascuns * neuroni_iesire
    nr_praguri_iesire = neuroni_iesire
    dimensiune_cromozom = nr_ponderi_ascuns + nr_praguri_ascuns +
nr_ponderi_iesire + nr_praguri_iesire

    # initializam o retea neuronală
    retea = ReteaNeurala(neuroni_intrare=neuroni_intrare,
neuroni_strat_ascuns=neuroni_strat_ascuns, neuroni_iesire=neuroni_iesire)

    # comparam dimensiunea cromozomului calculata de noi cu cea calculata
    de retea.
    self.assertEqual(retea.dimensiune_cromozom, dimensiune_cromozom)
```

Explicatie: Se calculeaza dimensiunea cromozomului manual pe baza numarului de neuroni de intrare, ascusi si de iesire si se compara cu valoarea calculata de retea neuronală.

### 3. Test selectie parinti

Verifica daca functia **selectie\_parinti** selecteaza corect doi parinti.

Python

```
def test_selectie_parinti(self):
    populatie = [np.random.uniform(-1, 1, 5) for _ in range(10)]
    fitnessuri = np.random.uniform(0, 1, 10)
    parinti = selectie_parinti(populatie, fitnessuri)
    self.assertEqual(len(parinti), 2)
```

Explicatie: Se genereaza o populatie de cromozomi si fitness-uri random si se verifica daca functia returneaza exact doi parinti.

#### 4. Test decodificare cromozom

Testeaza corectitudinea functiei de decodificare a unui cromozom intr-o retea neuronală.

Python

```
def test_decodificare_cromozom(self):
    # calculam manual dimensiunea unui cromozom
    neuroni_intrare = 4
    neuroni_strat_ascuns = 10
    neuroni_iesire = 3

    nr_ponderi_ascuns = neuroni_intrare * neuroni_strat_ascuns
    nr_praguri_ascuns = neuroni_strat_ascuns
    nr_ponderi_iesire = neuroni_strat_ascuns * neuroni_iesire
    nr_praguri_iesire = neuroni_iesire
    dimensiune_cromozom = nr_ponderi_ascuns + nr_praguri_ascuns +
    nr_ponderi_iesire + nr_praguri_iesire

    # cromozom
    cromozom = initializarea_populatiei(1,
    dimensiune_cromozom=dimensiune_cromozom)[0]

    # retea neuronală
    retea = ReteaNeuronală(neuroni_intrare=neuroni_intrare,
    neuroni_strat_ascuns=neuroni_strat_ascuns, neuroni_iesire=neuroni_iesire)

    cromozom_decodificat = retea.decodificare_cromozom(cromozom=cromozom)

    # verificam valorile obtinute
    self.assertEqual(cromozom_decodificat["ponderi_ascuns"].shape,
    (neuroni_intrare, neuroni_strat_ascuns))
    self.assertEqual(len(cromozom_decodificat["praguri_ascuns"]),
    neuroni_strat_ascuns)
    self.assertEqual(cromozom_decodificat['ponderi_iesire'].shape,
    (neuroni_strat_ascuns, neuroni_iesire))
    self.assertEqual(len(cromozom_decodificat['praguri_iesire']),
    neuroni_iesire)
```

Explicatie: Cromozomul este decodificat in structuri care reprezinta ponderile si pragurile retelei. Testul verifica dimensiunile acestor structuri dupa apelarea functiei “decodificare\_cromozom”

## 5. Test incrucisare

Verifica functia de incrucisare a cromozomilor pentru a genera copii.

Python

```
def test_incrucisare(self):
    # initializam o retea
    retea = ReteaNeuronala(neuroni_intrare=4, neuroni_strat_ascuns=10,
neuroni_iesire=3)

    # generam doi parinti la intamplare
    parinte1 = np.random.rand(retea.dimensiune_cromozom)
    parinte2 = np.random.rand(retea.dimensiune_cromozom)

    # aplicam functia de incrucisare
    copil1, copil2 = incrucisare(parinte1, parinte2)

    # verificam daca
    self.assertEqual(len(copil1), len(parinte1))
    self.assertEqual(len(copil2), len(parinte2))
```

Explicatie: Generam doi parinti la intamplare si aplicam incrucisarea. Mai apoi, verificam daca cromozomii copiilor au aceeasi dimensiune cu cea a parintilor.

## 6. Test mutatie 100%

Testeaza daca functia mutatie modifica cromozomul cand rata de mutatie este 100%.

Python

```
def test_mutatie_100(self):
    retea = ReteaNeuronala(4, 10, 3)
    cromozom = initializarea_populatiei(1, retea.dimensiune_cromozom)[0]

    # functia de mutatie modifica direct argumentul trimis ca parametru asa
    ca voi face o copie.
    copie = cromozom.copy()

    # testam cu sansa de 100% de mutatie
    rata_mutatie = 1
    cromozom_mutat = mutatie(copie, rata_mutatie)

    self.assertFalse(np.array_equal(cromozom, cromozom_mutat))
```



Explicatie: Se creeaza un cromozom si se aplica o mutatie cu rata de 100%. Mai apoi, verificam daca cromozomul initial si cel mutat sunt diferiti.

## 7. Test mutatie 0%

Testeaza daca functia mutatie NU modifica cromozomul cand rata de mutatie este 0%.

Python

```
def test_mutatie_0(self):
    retea = ReteaNeuronala(4, 10, 3)
    cromozom = initializarea_populatiei(1, retea.dimensiune_cromozom)[0]

    # functia de mutatie modifica direct argumentul trimis ca parametru asa
    # ca voi face o copie.
    copie = cromozom.copy()

    # testam cu sansa de 0% de mutatie
    rata_mutatie = 0
    cromozom_mutat = mutatie(copie, rata_mutatie)

    self.assertTrue(np.array_equal(cromozom, cromozom_mutat))
```

Explicatie: Se creeaza un cromozom si se aplica o mutatie de 0%. Mai apoi, verificam daca cromozomul initial si cel mutat sunt identici.

## 8. Test fitness

Verifica functionalitatea metodei “fitness” care calculeaza calitatea unui cromozom.

Python

```
def test_fitness(self):
    retea = ReteaNeuronala(4, 10, 3)
    cromozom = initializarea_populatiei(1, retea.dimensiune_cromozom)[0]
    X_train, _, Y_train, _ = incarca_si_proceseaza_date()
    rezultat = fitness(cromozom, retea, X_train, Y_train)
    self.assertIsInstance(rezultat, float)
```

Explicatie: Se aplica functia fitness pe un cromozom si pe setul de date de antrenare si verificam daca rezultatul este de tip float.

## 9. Test algoritm genetic

Testeaza functia principala a algoritmului genetic ce optimizeaza cromozomii.

Python

```
def test_algoritm_genetic(self):
    retea = ReteaNeuronala(4, 10, 3)
    X_train, _, Y_train, _ = incarca_si_proceseaza_date()
    rezultat = algoritm_genetic(
        retea=retea, X_train=X_train, y_train=Y_train,
        dimensiune_populatie=10, generatii=5, rata_mutatie=0.1
    )
    self.assertEqual(len(rezultat), retea.dimensiune_cromozom)
```

Explicatie: Rulam algoritmul pe un set de date si pe o populatie de 10 cromozomi timp de 5 generatii si verificam daca rezultatul are dimensiunea unui cromozom.

## 10. Test sigmoid

Acest test verifica implementarea functiei sigmoid din retea neuronală.

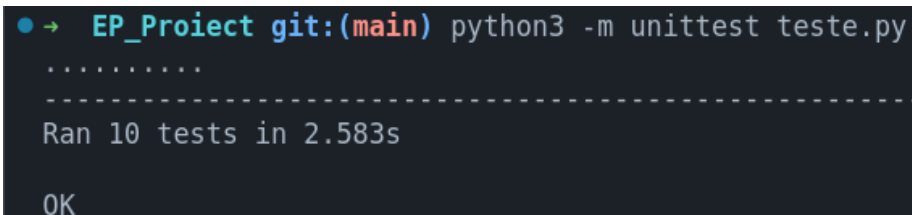
Python

```
def test_sigmoid(self):
    x = np.array([-1e10, -1, 0, 1, 1e10])
    retea = ReteaNeuronala(4, 10, 3)
    rezultat = retea.sigmoid(x)
    asteptat = np.array([0, 0.26894142, 0.5, 0.73105858, 1])

    np.testing.assert_almost_equal(rezultat, asteptat, decimal=6)
```

Explicatie: Aplicam functia sigmoid pe un set de date de intrare si comparam cu valorile asteptate.

## Rezultate teste automate



```
● → EP_Proiect git:(main) python3 -m unittest teste.py
.....
-----
Ran 10 tests in 2.583s

OK
```

## 6. Concluzii

Proiectul demonstrează eficiența utilizării rețelelor neuronale de tip perceptron multistrat optimizate prin algoritmi evolutivi pentru clasificarea datelor din setul Iris. Performanța ridicată, cu o acuratețe de 97.5% pe setul de antrenare și 96.6% pe setul de test, reflectă o generalizare excelentă și o stabilitate a modelului. Soluția propusă subliniază viabilitatea algoritmilor evolutivi în optimizarea parametrilor rețelelor neuronale și capacitatea acestora de a aborda probleme complexe de clasificare într-un mod robust și precis.

## 7. Bibliografie

1. [Multilayer Perceptrons in Machine Learning: A Comprehensive Guide | DataCamp](#)
2. [Iris - UCI Machine Learning Repository](#)
3. [Curs Inteligența Artificială - Prof. dr. ing. Florin Leon](#)

## 8. Contribuțiile membrilor echipei

Caraman Talida:

- Incarcarea și amestecarea datelor
- Initializarea populației și selecția părinților
- Algoritmul genetic
- Documentație punctul 2.1, 3.3, 3.4, 8

Pricop Matei-Ioan:

- Impartirea datelor în set antrenare și testare
- Funcția de fitness și funcția sigmoid
- Propagarea înainte
- Documentație punctul 1, 2.2, 5, 6, 7

Vieru Iosif:

- Normalizarea datelor
- Incrucisarea și mutația
- Decodificare cromozom
- Documentație, punctul 3.1, 3.2, 4, 5.4