Sztuczna inteligencja

Algorytm genetyczny

Marcin Wojtas
Nr. albumu: s153915
III rok Niestacjonarnie
Grupa: L3

1. Cel

Celem zadanie było zaimplementowanie algorytmu genetycznego który wyszukiwał najlepszy zestaw współczynników a, b, c, d których zakres został wylosowany z przedziału < -15, 15 >, mając na celu uzyskać największy wynik poprzez funkcję aktywacji:
$f(x) = a * x^3 + b * x^2 + c * x + d$, wiedząc, że (x_i, y_i) posiada 11 par: [(−5, −150), (−4, −77), (−3, −30), (−2, 0), (−1, 10), (½, 131/8) (1, 18), (2, 25), (3, 32), (4, 75), (5, 130)].

2. Wykorzystane pojęcia oraz wzory

Funkcja aktywacji: $f(x) = a * x^3 + b * x^2 + c * x + d$

Chromosom - jest to ciąg bitów. Każdy pojedynczy bit jest odpowiednikiem pojedynczego genu

Operacja krzyżowania - polega na losowym przecięciu dwóch chromosomów (ciągów bitów) w jednym punkcie i zamianie podzielonych części między chromosomami. Powstają dwa nowe chromosomy. Ważne jest to ze dzieci całkowicie zastępują rodziców.

Operacja mutacji - polega na zamianie na przeciwny losowo wybranego bitu.

Selekcja metody koła ruletki - która przydziela prawdopodobieństwa wylosowania każdego osobnika bezpośrednio na podstawie jednej funkcji oceny.

Populacja - ma stały rozmiar, a w kolejnych cyklach ewolucji wszystkie chromosomy podlegają wymianie na nowe (dzieci całkowicie zastępują rodziców).

3. Kod źródłowy programów

```python
import random

population_size = 6
mutation_rate = 0.1
max_generations = 100

population = [[0 for j in range(4)] for i in range(population_size)]
pairXY = [[-5, 150], [4, -77], [-3, -30], [-2, 0], [-1, 10], [0.5, 16.375], [1, 18], [2, 25], [3, 32], [4, 75], [5, 130]]
```

1. Dodanie bibliotek random
2. Stworzenie zmiennych:
    2.1 **population_size,** określa wielkość populacji
    2.2 **mutation_rate,** wskazuje poziom mutacji
    2.3 **max_generations,** określa liczbę generacji (powtórzeń)
    2.4 **population,** tablica dwuwymiarowa która będzie miała na celu przechowywać populację
    2.5 **pairXY,** zdefiniowanie par (x_i, y_i)

```python
# Tworzenie populacji początkowej
for i in range(population_size):
    for j in range(4):
        population[i][j] = random.randint(-15, 15)
```

3. Stworzenie populacji początkowej poprzez wylosowanie liczb od -15 do 15

```python
def activation_function(i):
    a, b, c, d = i
    e = sum(abs(a * pow(x, 3) + b * pow(x, 2) + c * x + d) for x, y in pairXY)
    return e

```

4. Stworzenie funkcji aktywacji do które zostają wprowadzone parametry osobnika, później wartość zostaje wyliczona ze podanego wcześniej wzoru, na końcu funkcja zwraca wyliczoną wartość

```python
# Selekcja typu ruletka
def roulette_selection(population, activation_function):
    total = sum(activation_function)
    probabilities = [score / total for score in activation_function]
    selected = random.choices(population, probabilities, k=2)
    return selected[0], selected[1]

```

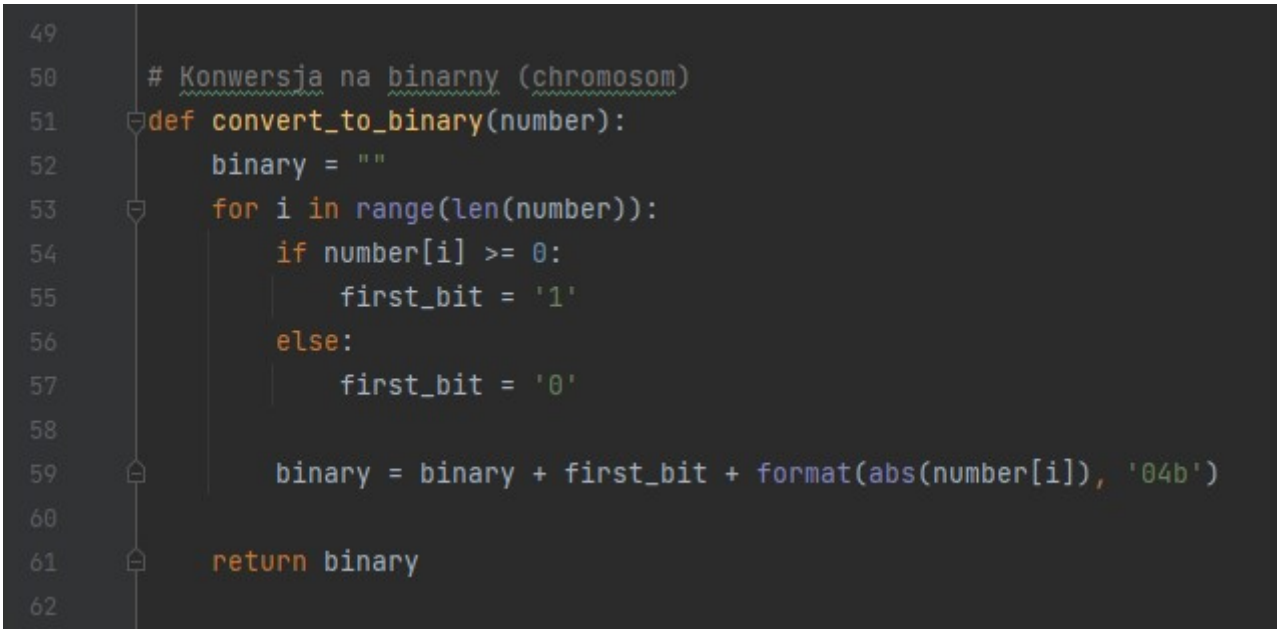
5. Stworzenie funkcji **roulette_selection**, która przyjmuje dwa parametry: populację oraz odpowiadającą populacji wartości funkcji aktywacji.

5.1 do zmiennej **total** przypisane zostaje suma wszystkich wartości funkcji aktywacji w populacji

5.2 do zmiennej **probabilities** jest obliczane prawdopodobieństwo wyboru danego osobnika, jest ono proporcjonalne do jego wartości funkcji aktywacji

5.3 w zmiennej **selected** wybieramy dwóch wylosowanych osobników poprzez wykorzystanie funkcji **random.choices**

5.4 zwrócenie dwóch osobników z populacji



```
49
50      # Konwersja na binarny (chromosom)
51      def convert_to_binary(number):
52          binary = ""
53          for i in range(len(number)):
54              if number[i] >= 0:
55                  first_bit = '1'
56              else:
57                  first_bit = '0'
58
59              binary = binary + first_bit + format(abs(number[i]), '04b')
60
61          return binary
62
```

6. Funkcja **convert_to_binary** przyjmuje parametry numerów jednego z osobników którego będziemy zmieniać

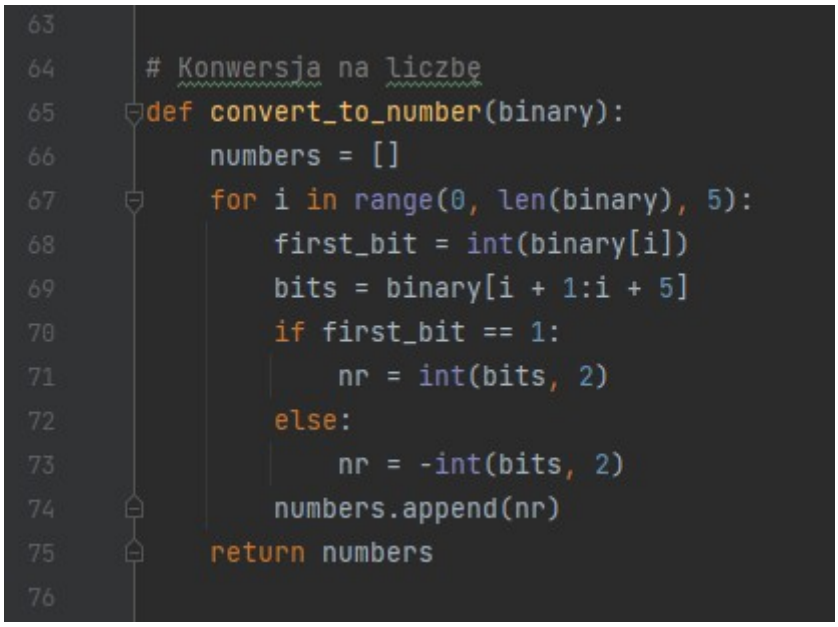
6.1 zmienna **binary** przechowuje ciąg znaków, w naszym przypadku binarnych

6.2 w pętli for będziemy zmieniać każdą z liczb na binarną

6.2.1 jeśli liczba jest dodatnia pierwszy bit z każdej liczb będzie przyjmował 1, jeśli ujemna 0

6.2.2 dodanie do zmiennej **binary** czterech liczb ciągu binarnego reprezentującego liczbę

6.3 zwrócenie wartości **binary**



```
63
64      # Konwersja na liczbę
65      def convert_to_number(binary):
66          numbers = []
67          for i in range(0, len(binary), 5):
68              first_bit = int(binary[i])
69              bits = binary[i + 1:i + 5]
70              if first_bit == 1:
71                  nr = int(bits, 2)
72              else:
73                  nr = -int(bits, 2)
74              numbers.append(nr)
75          return numbers
76
```

7. Funkcja **convert_to_number** przyjmuje parametry binarne jednego z osobników którego będziemy zmieniać

- 7.1 zmienna **numbers** przechowuje liczby
- 7.2 w pętli for będziemy zmieniać każdą z liczb na całkowitą
  - 7.2.1 **first_bit** przechowuje wartość pierwszego bitu
  - 7.2.2 **bits** przechowuje wartość binarną liczby
  - 7.2.3 jeśli pierwszy bit jest 1 to liczna zostaje przekonwertowana z binarnej na dodatnią, w przeciwnym razie na ujemną
  - 7.2.4 dodanie do tablicy przekonwertowanej liczby
- 7.3 zwrócenie tablicy **numbers**

```python

# Krzyżowanie rodziców
def crossover(parent1, parent2):
    crossover_point = random.randint(1, len(parent1) - 1)
    child1 = parent1[:crossover_point] + parent2[crossover_point:]
    child2 = parent2[:crossover_point] + parent1[crossover_point:]
    return child1, child2

```

8. Stworzenie funkcji **crossover** przyjmuje parametry rodziców jako ciąg binarny

   8.1 wylosowanie miejsca w którym rodzice mają się krzyżować i przypisanie go do zmiennej **crossover_point**
   8.2 stworzenie zmiennych **child1** oraz **child2** poprzez krzyżowanie rodziców
   8.3 zwrócenie **child1** oraz **child2**

```python
# Mutacja
def mutate(chromosome):
    mutated_chromosome = chromosome
    i = random.randint(0, 19)
    if random.random() <= mutation_rate:
        if mutated_chromosome[i] == '0':
            mutated_chromosome = mutated_chromosome[:i] + '1' + mutated_chromosome[i + 1:]
        else:
            mutated_chromosome = mutated_chromosome[:i] + '0' + mutated_chromosome[i + 1:]
    return mutated_chromosome

```

9. Stworzenie funkcji **mutate** która przyjmuje parametr jako ciąg binarny

   9.1 przypisanie chromosomu do **mutated_chromosome**
   9.2 wylosowanie bitu gdzie chromosom zmutuje

9.3 jeśli wylosowania wartość w if jest mniejsza lub równa **mutation_rate** wtedy zmieniamy bit w wylosowanym miejscu na przeciwny.
9.4 zwrócenie chromosomu

```python
# Algorytm genetyczny
def genetic_algorithm():
    global population

    for generation in range(max_generations):
        activation = [activation_function(i) for i in population]
        new_population = []

        for _ in range(population_size // 2):
            parent1, parent2 = roulette_selection(population, activation)
            child1, child2 = crossover(convert_to_binary(parent1), convert_to_binary(parent2))
            child1 = mutate(child1)
            child2 = mutate(child2)
            child1 = convert_to_number(child1)
            child2 = convert_to_number(child2)
            new_population.append(child1)
            new_population.append(child2)

        population = new_population

    best_individual = max(population, key=activation_function)
    return best_individual
```

10. Funkcja **genetic_algorithm**
    10.1 przypisanie zmiennej **population** wartości globalnej
    10.2 pętla for wykonuje liczbę korków dążąc do **max_generations** która reprezentuje liczbę generacji
        10.2.1 **activation** przechowuje wartości z funkcji aktywacji dla każdego elementu populacji
        10.2.2 stworzenie zmiennej **new_population** która będzie przechowywać tymczasową populację
        10.2.3 pętla for wykonuje liczbę korków przez połowę liczbę populacji
            10.2.3.1 dla **parent1** i **parent2** zostaje wylosowane z koła ruletki dwoje rodziców

10.2.3.2 przypisanie do **child1** oraz **child2** wyników z krzyżowanie rodziców i dodatkowo zmiana rodziców na ciąg binarny
10.2.3.3 wykonanie mutacji na **child1** oraz **child2**
10.2.3.4 zmiana z binarnego na liczby **child1** i **child2**
10.2.3.5 dodanie do **new_population child1** i **child2**

10.2.4 przypisanie starej populacji nowej populacji

10.3 stworzenie zmiennej **best_individual** i dodanie do niej osobnika z populacji dla którego został wyszukany największy czynnik z funkcji aktywacji

10.4 zwrócenie **best_individual**

```
100
101 print(f"Liczba generacji: {max_generations}")
102 print("\nPoczątkowe współczynniki: ")
103 print(f"{population[0]} \n{population[1]} \n{population[2]} \n{population[3]} \n{population[4]} \n{population[5]}")
104 print("\nPoczątkowe wyniki funkcji: ")
105 print(f"{activation_function(population[0])} \n{activation_function(population[1])} \n{activation_function(population[2])} "
106       f"\n{activation_function(population[3])} \n{activation_function(population[4])} \n{activation_function(population[5])}")
107
108 result = genetic_algorithm()
109
110 print("\nKońcowe współczynniki: ")
111 print(f"{population[0]} \n{population[1]} \n{population[2]} \n{population[3]} \n{population[4]} \n{population[5]}")
112 print("\nKońcowe wyniki funkcji: ")
113 print(f"{activation_function(population[0])} \n{activation_function(population[1])} \n{activation_function(population[2])} "
114       f"\n{activation_function(population[3])} \n{activation_function(population[4])} \n{activation_function(population[5])}")
115 print("\nNajlepsze końcowe współczynniki: ")
116 print(result)
117 print("\nNajlepszy końcowy wynik: ")
118 print(activation_function(result))
119
```

11. Wypisanie danych populacji i odpowiadającej jej współczynników z funkcji aktywacji przed wykonaniu funkcji **genetic_algorithm**.
12. Przypisanie do **result** wyniku i wykonanie funkcji **genetic_algorithm**.
13. Wypisanie danych populacji i odpowiadającej jej współczynników z funkcji aktywacji po wykonaniu funkcji **genetic_algorithm**.
14. Wypisanie najlepszego osobnika z populacji oraz jego współczynnika z funkcji aktywacji.

4. Przykładowe wykonania programu

Wynik dla 5 generacji:

```
Liczba generacji: 5

Początkowe współczynniki:
[0, -7, 15, -4]
[-12, 7, -6, 0]
[-15, 2, 0, 4]
[-12, 13, -12, 2]
[-3, 10, -13, -7]
[-12, -6, -8, 15]

Początkowe wyniki funkcji:
699.75
5358.75
6680.625
5342.25
1445.375
5810.0

Końcowe współczynniki:
[-13, -6, -6, 0]
[-13, -6, -6, 0]
[-12, -6, -6, 0]
[-13, -6, -8, 14]
[-12, -6, -8, 14]
[-13, -6, -6, 0]

Końcowe wyniki funkcji:
6228.125
6228.125
5778.0
6260.875
5811.0
6228.125

Najlepsze końcowe współczynniki:
[-13, -6, -8, 14]

Najlepszy końcowy wynik:
6260.875
```

Wynik dla 50 generacji:

```
Liczba generacji: 50

Początkowe współczynniki:
[-7, -1, 2, -15]
[-15, -11, -9, 9]
[-12, 8, 2, 11]
[0, 5, -5, 11]
[10, 13, 12, -11]
[3, -15, -1, -1]

Początkowe wyniki funkcji:
3189.125
7354.125
5092.5
629.75
5254.5
1288.875

Końcowe współczynniki:
[-13, 8, 3, -15]
[-13, 8, 3, -15]
[-13, 8, 3, -15]
[13, 8, 3, -15]
[-13, 8, 3, -7]
[-13, 8, 3, -15]

Końcowe wyniki funkcji:
5547.125
5547.125
5547.125
6175.875
5523.125
5547.125

Najlepsze końcowe współczynniki:
[13, 8, 3, -15]

Najlepszy końcowy wynik:
6175.875
```

Wynik dla 100 generacji:

```
Liczba generacji: 100

Początkowe współczynniki:
[9, -6, 9, -2]
[-7, -9, -8, -8]
[9, 0, 14, -15]
[-1, 1, 3, 10]
[5, 5, 10, -6]
[5, 3, 6, -10]

Początkowe wyniki funkcji:
4126.125
3713.125
4446.875
371.625
2698.875
2511.625

Końcowe współczynniki:
[-15, -5, 13, -12]
[-15, -5, 13, -12]
[-15, -5, 9, -12]
[-15, -5, 13, 14]
[-15, -5, 13, -12]
[-13, -5, 13, 12]

Końcowe wyniki funkcji:
6582.625
6582.625
6696.625
6523.375
6582.625
5625.625

Najlepsze końcowe współczynniki:
[-15, -5, 9, -12]

Najlepszy końcowy wynik:
6696.625
```

5. Podsumowanie

Algorytm genetyczny naśladuje proces ewolucji biologicznej co powoduje, że jest w dużym stopniu losowy, wiele zależy od wylosowania początkowych wartości a później od ich krzyżowania a także od zaistnienia mutacji na podstawie wartości funkcji przystosowania.

Bibliografia:

https://home.agh.edu.pl/~vlsi/AI/gen_t/ - wykorzystane pojęcia