

[Dokumentacja Kodu](#)

[Pytania:](#)

[Pytanie 1](#)

[Pytanie 2](#)

[Pytanie 3](#)

[Pytanie 4](#)

[Pytanie 5](#)

[Pytanie 6](#)

[Pytanie 7](#)

[Pytanie 8](#)

Dokumentacja Kodu

## Dokumentacja kodu Wielowarstwowego Perceptronu z Algorytmem Ewolucji Różnicowej

### Przegląd

Kod implementuje model wielowarstwowego perceptronu (MLP) z optymalizacją wag za pomocą algorytmu ewolucji różnicowej. Model MLP jest wykorzystywany do klasyfikacji danych z zestawu Iris, który zawiera pomiary kwiatów irysa. Algorytm ewolucji różnicowej jest zastosowany do optymalizacji parametrów sieci neuronowej.

### Struktura Klasy WielowarstwowoPerceptron

#### Inicjalizacja (`__init__`):

- Tworzy warstwy perceptronu z losowo inicjalizowanymi wagami i biasami.
- **Dlaczego?** Inicjalizacja wag jest standardowym pierwszym krokiem w tworzeniu sieci neuronowych. Losowe wartości pomagają w "rozpoczęciu" procesu uczenia.

#### Metoda `forward`:

- Realizuje przepływ danych przez sieć (propagację w przód), stosując wagi i funkcję aktywacji do wejścia.
- **Zastosowanie:** Umożliwia sieci generowanie prognoz na podstawie danych wejściowych.

#### `ustaw_parametry` i `pobierz_parametry`:

- Pozwalają na manipulację parametrami sieci (wagami i biasami) - kluczowe dla algorytmu optymalizacyjnego.
- **Dlaczego?** Umożliwiają algorytmowi ewolucji różnicowej dostosowanie wag sieci w procesie optymalizacji.

### Funkcje pomocnicze

#### `sigmoid`:

- Funkcja aktywacji, przekształcająca wartości liniowe na zakres (0,1).
- **Zastosowanie:** Wprowadza nieliniowość do modelu, co jest kluczowe dla uczenia się złożonych wzorców.

#### `mse` (Mean Squared Error):

- Oblicza średni błąd kwadratowy między prognozami a rzeczywistymi wartościami.
- **Dlaczego?** MSE jest standardową miarą błędów w problemach regresji i służy jako funkcja celu do optymalizacji.

#### `dokladnosc`:

- Mierzy, jak dokładne są prognozy modelu.
- **Zastosowanie:** Umożliwia ocenę wydajności modelu na danych testowych.

### Funkcja `funkcja_celu`

#### Cel:

- Wykorzystywana przez algorytm ewolucji różnicowej do oceny jakości poszczególnych rozwiązań (parametrów sieci).
- **Dlaczego?** Funkcja celu jest niezbędna do kierowania procesem optymalizacji w algorytmach ewolucyjnych.

### **Algorytm Ewolucji Różnicowej (ewolucja\_roznicowa)**

#### **Proces:**

- Wykorzystuje operacje mutacji, krzyżowania i selekcji do optymalizacji parametrów sieci.
- **Dlaczego?** Algorytm ewolucji różnicowej jest efektywną, choć mniej konwencjonalną metodą optymalizacji parametrów sieci neuronowych, zapewniającą dobre wyniki w wielu problemach.

### **Wczytywanie i Przygotowanie Danych**

#### **Proces:**

- Wczytuje i przetwarza dane z zestawu Iris, a następnie dzieli je na zbiory treningowe i testowe.
- **Dlaczego?** Przygotowanie danych jest kluczowym krokiem w procesie uczenia maszynowego, umożliwiającym efektywne trenowanie i ocenę modelu.

### **Podział na Zbiór Treningowy i Testowy**

#### **Zastosowanie:**

- Oddziela część danych do treningu modelu od części do jego oceny.
- **Dlaczego?** Taki podział jest niezbędny, aby móc sprawdzić, jak model radzi sobie z nieznanymi danymi, co jest miarą jego zdolności do generalizacji.

### **Użycie Algorytmu i Ocena Modelu**

- Po zakończeniu procesu optymalizacji algorytmem ewolucji różnicowej, model jest oceniany na zbiorze testowym.
- **Dlaczego?** Ocena na zbiorze testowym daje informację o tym, jak dobrze model będzie działał w praktyce, na danych, których nie "widział" podczas treningu.

## Pytania:

### Pytanie 1

Tworzy warstwy perceptronu z losowo inicjalizowanymi wagami i biasami. - jeżeli miałbyś większy dataset to można dodać więcej warstw (ukryte), ale nie będziemy robić tak zaawansowanych rzeczy

### Pytanie 2

Szczegółowy opis metody **forward** (forward propagation) z klasy **WielowarstwowyPerceptron** w powyższym kodzie, opisany krok po kroku:

Metoda **forward**(Forward Propagation)

```
python Copy code

def przod(self, X):
    aktywacja = X
    for w, b in zip(self.wagi, self.biasy):
        z = np.dot(aktywacja, w) + b
        aktywacja = sigmoid(z)
    return aktywacja
```

step 1 - Inicjalizacja aktywacji:

```
python Copy code

aktywacja = X
```

- Ta linia ustawia początkową aktywację na dane wejściowe X.
- **Co to oznacza?** W kontekście sieci neuronowej, "aktywacja" oznacza wartości, które są przekazywane z jednej warstwy sieci do następnej. Na początku, aktywacją jest sam wektor wejściowy.

step 2 - Pętla po warstwach sieci:

```
python Copy code

for w, b in zip(self.wagi, self.biasy):
```

- Ta linia rozpoczyna pętlę, która iteruje przez wszystkie warstwy sieci neuronowej.
- Co to oznacza? Dla każdej warstwy w sieci, kod będzie stosować odpowiadające jej wagi (**w**) i bias (**b**).

step 3 - Obliczanie ważonej sumy i dodawanie biasu:

```
python Copy code  
  
z = np.dot(aktywacja, w) + b
```

- Tutaj obliczana jest ważona suma aktywacji z poprzedniej warstwy i wag aktualnej warstwy, do której dodaje się bias.
- **Co to oznacza?** Wartość **z** jest liniową kombinacją wejść z poprzedniej warstwy, stanowiącą podstawę do następnej aktywacji. Jest to kluczowy krok w działaniu każdego neuronu w sieci.

step 4 - Stosowanie funkcji aktywacji:

```
python Copy code  
  
aktywacja = sigmoid(z)
```

- Funkcja **sigmoid** jest stosowana do wartości **z**. Dzięki temu wartości te są "przepuszczane" przez nieliniową funkcję aktywacji, co pozwala sieci na uczenie się i modelowanie bardziej złożonych relacji.
- Co to oznacza? Nieliniowa funkcja aktywacji jest kluczowa, aby sieć neuronowa mogła rozwiązywać zadania, które nie są liniowo separowalne.

step 5 - Zwracanie aktywacji z ostatniej warstwy:

```
python Copy code  
  
return aktywacja
```

- Po przejściu przez wszystkie warstwy, ostateczna aktywacja (wynik z ostatniej warstwy sieci) jest zwracana jako wynik metody.
- Co to oznacza? Ostateczna aktywacja jest "decyzją" sieci, na przykład klasą, do której przypisuje wejściowe dane.

## Podsumowanie

Metoda **forward** realizuje kluczowy proces przepływu danych przez sieć neuronową. Dla każdej warstwy obliczane są wartości, które następnie są przekształcane przez funkcję aktywacji. Wynik z ostatniej warstwy stanowi wyjście sieci, które może być używane do klasyfikacji lub innych zadań.

### Pytanie 3

#### MSE (Mean Squared Error) - Błąd Średniokwadratowy

MSE, czyli błąd średniokwadratowy, jest popularną miarą oceny błędów w modelach uczenia maszynowego, zwłaszcza w zadaniach regresji. Oblicza on średnią wartość kwadratów różnic między rzeczywistymi wartościami a tymi przewidzianymi przez model.

Jak jest obliczany?

MSE jest obliczany według wzoru:

$$MSE = \frac{1}{n} \sum_{i=1}^n (Y_i - \hat{Y}_i)^2$$


gdzie:

- $Y_i$  to rzeczywiste wartości,
- $\hat{Y}_i$  to wartości przewidywane przez model,
- $n$  to liczba obserwacji.

Zastosowanie MSE w tym kodzie

W podanym kodzie funkcja mse jest wykorzystywana jako funkcja celu dla algorytmu ewolucji różnicowej. Działanie algorytmu polega na minimalizacji wartości MSE, co oznacza dążenie do jak najdokładniejszych prognoz modelu w stosunku do rzeczywistych danych.

python

 Copy code

```
def mse(prawdziwe_y, przewidywane_y):  
    return np.mean(np.square(prawdziwe_y - przewidywane_y))
```

#### Dokładność (Accuracy)

Co to jest dokładność?

Dokładność to prosta metryka wydajności modelu klasyfikacyjnego, reprezentująca stosunek liczby poprawnych przewidywań do łącznej liczby przewidywań. Jest często używana w klasyfikacji, aby ocenić ogólną skuteczność modelu.

Jak jest obliczana?


Dokładność jest obliczana według wzoru:

$$\text{Dokładność} = \frac{\text{Liczba Poprawnych Przewidywań}}{\text{Łączna Liczba Przewidywań}}$$

Zastosowanie dokładności w tym kodzie

W podanym kodzie funkcja `dokladnosc` oblicza dokładność modelu na zbiorze testowym. Jest to stosunek liczby prawidłowo sklasyfikowanych próbek do całkowitej liczby próbek w zbiorze testowym. Wysoka dokładność oznacza, że model dobrze generalizuje na nowych danych, które nie były używane podczas procesu uczenia.

python

 Copy code

```
def dokladnosc(prawdziwe_y, przewidywane_y):  
    return np.mean(np.argmax(prawdziwe_y, axis=1) == np.argmax(przewidy
```

Podsumowanie

- **MSE:** Używany jako funkcja celu do optymalizacji modelu; niskie wartości MSE wskazują na lepsze dopasowanie modelu.
- **Dokładność:** Umożliwia ocenę ogólnej skuteczności modelu na zbiorze testowym; wysoka dokładność wskazuje na dobrą generalizację modelu.

## Pytanie 4

### Funkcja Celu w Kontekście Algorytmu Ewolucji Różnicowej

Co to jest funkcja celu?

Funkcja celu w uczeniu maszynowym i optymalizacji jest kluczowym elementem określającym "jakość" danego rozwiązania. Jest to miara, którą algorytm stara się zminimalizować (lub zmaksymalizować, w zależności od zadania). W kontekście algorytmów ewolucyjnych, taka funkcja ocenia, jak dobrze poszczególne rozwiązania w populacji spełniają zadane kryteria.

### Funkcja celu w podanym kodzie

W podanym kodzie, funkcja celu jest używana przez algorytm ewolucji różnicowej do oceny, jak dobrze poszczególne zestawy parametrów (wagi i biasy sieci neuronowej) radzą sobie z zadaniem przewidywania wyników. Konkretnie, funkcja celu oblicza błąd średniokwadratowy (MSE) między prognozami modelu a rzeczywistymi etykietami.

```
python Copy code  
  
def funkcja_celu(parametry, model, X, Y):  
    model.ustaw_parametry(parametry)  
    prognozy = model.przod(X)  
    return mse(Y, prognozy)
```

### Możliwość usunięcia funkcji celu i MSE

**Funkcja Celu:** Nie można usunąć tej funkcji, jeśli chcemy, aby algorytm ewolucji różnicowej działał. Funkcja celu jest niezbędna do prowadzenia procesu optymalizacji, ponieważ algorytm potrzebuje sposobu oceny i porównywania różnych rozwiązań.

**MSE:** Teoretycznie można by zastąpić MSE inną funkcją straty, jeśli lepiej pasowałaby do konkretnego zadania. Na przykład, w klasyfikacji binarnej często stosuje się funkcję straty entropii krzyżowej. Jednak w kontekście tego konkretnego zadania (gdzie przewiduje się klasy irysów), MSE jest odpowiednim wyborem jako funkcja straty, ponieważ jest prosta w implementacji i interpretacji.

### Podsumowanie

Funkcja celu i MSE są integralnymi częściami kodu, które umożliwiają algorytmowi ewolucji różnicowej efektywną optymalizację parametrów sieci neuronowej. Bez nich, algorytm nie miałby "kryterium sukcesu" do dążenia, co uniemożliwiłoby skuteczne uczenie się modelu.



## Pytanie 5

Funkcja `ewolucja_roznicowa`

```
python Copy code  
  
def ewolucja_roznicowa(funkcja, model, granice, X, Y, mut=0.8, krzyz=0.
```

Parametry funkcji:

- **funkcja**: funkcja celu używana do oceny jakości rozwiązania.
- **model**: model sieci neuronowej, który jest optymalizowany.
- **granice**: zakres wartości, które mogą przyjmować parametry modelu.
- **X, Y**: dane treningowe i odpowiadające im etykiety.
- **mut (mutacja)**: współczynnik mutacji, określa intensywność zmian parametrów.
- **krzyz (krzyżowanie)**: prawdopodobieństwo krzyżowania, wpływa na to, jak bardzo nowe rozwiązanie jest kombinacją aktualnego i mutantu.
- **wielkosc\_populacji**: liczba rozwiązań w populacji.
- **iteracje**: liczba iteracji algorytmu.

Inicjalizacja populacji

```
python Copy code  
  
dimensions = len(model.get_params())  
pop = np.random.rand(wielkosc_populacji, dimensions)
```

- **dimensions**:  
Ustala liczbę wymiarów rozwiązania na podstawie liczby parametrów modelu.
- **pop**:  
Tworzy początkową populację. Każde rozwiązanie jest wektorem o długości `dimensions`, z wartościami losowo wybranymi z zakresu `[0, 1]`.

```
python Copy code  
  
min_b, max_b = np.asarray(granice).T  
diff = np.fabs(min_b - max_b)  
pop_denorm = min_b + pop * diff
```

Rozpakowywanie granic i obliczanie różnic:

- **np.asarray(granice).T** przekształca listę granic na format użyteczny dla NumPy i transponuje ją, oddzielając minimalne i maksymalne granice.
- **np.fabs** oblicza bezwzględną wartość różnicy między maksymalnymi a minimalnymi granicami. Jest to potrzebne do skalowania losowych wartości z populacji do rzeczywistego zakresu parametrów modelu.

- **pop\_denorm**: skaluje populację z zakresu [0, 1] do rzeczywistego zakresu parametrów modelu.

Obliczanie początkowego dostosowania populacji

```
python Copy code  
  
fitness = np.asarray([funkcja(ind, model, X, Y) for ind in pop_denorm])  
best_idx = np.argmin(fitness)  
best = pop_denorm[best_idx]
```

Obliczanie dostosowania (fitness) każdego rozwiązania:

- Każde rozwiązanie z populacji jest oceniane za pomocą funkcji celu. Wynik tej funkcji to "fitness" danego rozwiązania - niższy wynik oznacza lepsze dopasowanie.
- **best\_idx** i **best**: znajduje najlepsze dotychczasowe rozwiązanie i jego indeks.

Główna pętla algorytmu

```
python Copy code  
  
for i in range(iteracje):  
    for j in range(wielkosc_populacji):
```

- Iteruje przez zadaną liczbę iteracji, a w każdej iteracji przechodzi przez wszystkie rozwiązania w populacji.

Proces mutacji, krzyżowania i selekcji

```
python Copy code  
  
idxs = [idx for idx in range(wielkosc_populacji) if idx != j]  
a, b, c = pop[np.random.choice(idxs, 3, replace=False)]  
mutant = np.clip(a + mut * (b - c), 0, 1)
```

Wybór indeksów do mutacji:

- Wybiera losowo trzy różne rozwiązania z populacji, różne od aktualnie przetwarzanego.

Tworzenie mutacji:

- Oblicza nowe rozwiązanie (mutacje) poprzez dodanie ważonej różnicy między dwoma wybranymi rozwiązaniami do trzeciego. np.clip zapewnia, że wartości mutantu nie wykraczają poza zakres [0, 1].

python

Copy code

```
cross_points = np.random.rand(dimensions) < krzyz
if not np.any(cross_points):
    cross_points[np.random.randint(0, dimensions)] = True
trial = np.where(cross_points, mutant, pop[j])
trial_denorm = min_b + trial * diff
```

### Proces krzyżowania:

Tworzy nowe rozwiązanie (próbę) przez połączenie mutantą z obecnym rozwiązaniem. To, czy element pochodzi z mutantą czy obecnego rozwiązania, zależy od losowo wygenerowanych punktów krzyżowania.

### Denormalizacja próby:

Skaluje próbę z powrotem do rzeczywistego zakresu parametrów modelu.

python

Copy code

```
f = funkcja(trial_denorm, model, X, Y)
if f < fitness[j]:
    fitness[j] = f
    pop[j] = trial
if f < fitness[best_idx]:
    best_idx = j
    best = trial_denorm
```

### Selekcja:

- Porównuje nowo utworzoną próbę z obecnym rozwiązaniem. Jeśli próba ma lepsze dostosowanie (niższe MSE), zastępuje obecne rozwiązanie.
- Aktualizuje najlepsze dotychczas znalezione rozwiązanie, jeśli nowa próba jest lepsza od obecnego najlepszego rozwiązania.

### Podsumowanie

Algorytm ewolucji różnicowej w tym kodzie stosuje techniki ewolucji biologicznej, takie jak mutacja, krzyżowanie i selekcja, do iteracyjnego ulepszania populacji rozwiązań. Jest to podejście do optymalizacji, które jest szczególnie przydatne, gdy przestrzeń poszukiwań jest duża lub skomplikowana, a tradycyjne metody gradientowe mogą mieć trudności z odnalezieniem globalnego minimum. Funkcje jak **np.asarray**, **np.fabs**, czy **np.clip** są używane do efektywnego przetwarzania i manipulowania danymi w bibliotece NumPy, co jest standardem w obliczeniach naukowych w Pythonie.

## Pytanie 6

wczytanie pliku - za chuja nie wiem jak sie te formaty wczytuje więc pobrałem losowy plik z kaggle, ale to można z edytować - wysłałem Ci go na mess

## Pytanie 7

### Konwersja do One-Hot Encoding

Co to jest One-Hot Encoding?

One-hot encoding to proces przekształcania zmiennych kategoriycznych w formę, która może być lepiej przetwarzana przez modele uczenia maszynowego. Dla każdej unikalnej etykiety w zbiorze danych tworzy się nową kolumnę, w której każda etykieta jest reprezentowana jako wektor zer z jedną wartością ustawioną na 1.

Jak działa konwersja do One-Hot w podanym kodzie?

```
python Copy code  
  
Y = np.zeros((y.size, y.max() + 1))  
Y[np.arange(y.size), y] = 1
```

```
python Copy code  
  
Y = np.zeros((y.size, y.max() + 1))
```

```
python Copy code  
  
Y[np.arange(y.size), y] = 1
```

- Dla każdego wiersza w Y, ustawia wartość '1' w kolumnie odpowiadającej etykietce danej próbki. np.arange(y.size) generuje sekwencję indeksów dla każdej próbki, a y zawiera etykiety.

Czy można ominąć lub usunąć one-hot encoding?

**W kontekście klasyfikacji wieloklasowej:**

- One-hot encoding jest szczególnie przydatny i często konieczny, ponieważ pozwala modelowi na wyraźne rozróżnienie między różnymi klasami. Bez one-hot encoding, model mógłby błędnie interpretować etykiety jako wartości liczbowe, co jest niepożądane w klasyfikacji.

- Na przykład, bez one-hot encoding, model mógłby interpretować etykiety klas jako porządek i uznać, że jedna klasa jest "wyższa" lub "ważniejsza" niż inna, co jest nieprawidłowe w kontekście klasyfikacji.

#### W innych przypadkach:

- Jeśli zadanie nie wymaga rozróżniania między wieloma klasami, lub jeśli istnieje tylko jedna klasa binarna (0 lub 1), one-hot encoding może być niepotrzebny.

#### Podsumowanie

Konwersja do one-hot jest kluczowym krokiem w przygotowaniu danych do klasyfikacji wieloklasowej, ponieważ pozwala modelom uczenia maszynowego na skuteczne i poprawne przetwarzanie etykiet klas. Ominięcie tego kroku w zadaniach klasyfikacji wieloklasowej może prowadzić do błędnej interpretacji danych przez model i pogorszenia jego wydajności.

## Pytanie 8

### Uruchomienie Algorytmu Ewolucji Różnicowej

W podanym kodzie algorytm ewolucji różnicowej jest używany do optymalizacji parametrów (wag i biasów) modelu wielowarstwowego perceptronu. Oto dokładny opis tego procesu:

```
python Copy code

granice = [(-1, 1)] * len(perceptron.pobierz_parametry())
generator_er = ewolucja_roznicowa(funkcja_celu, perceptron, granice, X)
for i in range(1000):
    najlepszy, wynik = next(generator_er)
    print(f'Iteracja {i}, MSE: {wynik}')
```

#### Definiowanie Granic:

- `granice = [(-1, 1)] * len(perceptron.pobierz_parametry())`
- Ustala granice, w jakich mogą zmieniać się parametry modelu. Tutaj każdy parametr ma być w przedziale od -1 do 1. Długość listy granic jest równa liczbie parametrów modelu.

#### Inicjalizacja Algorytmu:

- `generator_er = ewolucja_roznicowa(...)`
- Tworzy generator (iterator), który będzie kolejno dostarczał wyniki każdej iteracji algorytmu ewolucji różnicowej.

#### Iterowanie przez Algorytm:

- Pętla `for` iteruje przez określoną liczbę iteracji algorytmu (tutaj 1000).
- W każdej iteracji, `next(generator_er)` uzyskuje następny zestaw optymalnych parametrów (najlepszy) oraz odpowiadający im wynik funkcji celu (wynik), czyli MSE.

- Wyniki są wyświetlane, co daje informację o postępie i efektywności algorytmu w poszukiwaniu optymalnych parametrów.

### Ostateczna Ocena Modelu na Zbiorze Testowym

Po zakończeniu procesu optymalizacji algorytmem ewolucji różnicowej, model jest testowany na zbiorze testowym, aby ocenić jego zdolność do generalizacji - czyli sprawdzenia, jak dobrze model radzi sobie z danymi, których nie widział podczas treningu.

```
python Copy code  
  
perceptron.ustaw_parametry(najlepszy)  
prognozy = perceptron.przod(X_testowe)  
print(f'Dokładność na zbiorze testowym: {dokladnosc(Y_testowe, prognozy)
```

### Ustawienie Optymalnych Parametrów:

- `perceptron.ustaw_parametry(najlepszy)`
- Model MLP jest aktualizowany optymalnymi parametrami znalezionymi przez algorytm ewolucji różnicowej.

### Generowanie Prognoz na Zbiorze Testowym:

- `prognozy = perceptron.przod(X_testowe)`
- Metoda **forward** modelu MLP jest używana do generowania prognoz na zbiorze testowym.

### Obliczenie i Wyświetlenie Dokładności:

- `dokladnosc(Y_testowe, prognozy)`
- Dokładność modelu na zbiorze testowym jest obliczana jako stosunek liczby poprawnych prognoz do całkowitej liczby próbek.
- Ta wartość jest wyświetlana i służy jako miara, jak dobrze model generalizuje na nowych danych.

### Podsumowanie

- Uruchomienie Algorytmu Ewolucji Różnicowej: Jest to kluczowy etap w procesie uczenia modelu, gdzie algorytm iteracyjnie poszukuje optymalnych parametrów modelu, minimalizując błąd (MSE) na zbiorze treningowym.
- Ocena Modelu na Zbiorze Testowym: Ostateczna ocena modelu na zbiorze testowym daje informację o jego zdolności do generalizacji i jest krytycznym elementem procesu walidacji modelu