Optymalizacja sieci neuronowej z zastosowaniem wybranego algorytmu ewolucyjnego

Sztuczna Inteligencja

Konrad Budzik

Paweł Satora

19.05.2023

Streszczenie

Optymalizacja sieci neuronowej to ważny aspekt w dziedzinie uczenia maszynowego, który ma na celu znalezienie optymalnych wag i parametrów modelu. Jednym z podejść do optymalizacji jest wykorzystanie algorytmów ewolucyjnych, które mogą pomóc znaleźć rozwiązanie w przestrzeni wielowymiarowej. Biblioteka PyGAD w języku Python zapewnia narzędzia do optymalizacji sieci neuronowych za pomocą różnych algorytmów ewolucyjnych. W tej pracy omówimy proces optymalizacji sieci neuronowej przy użyciu biblioteki PyGAD.

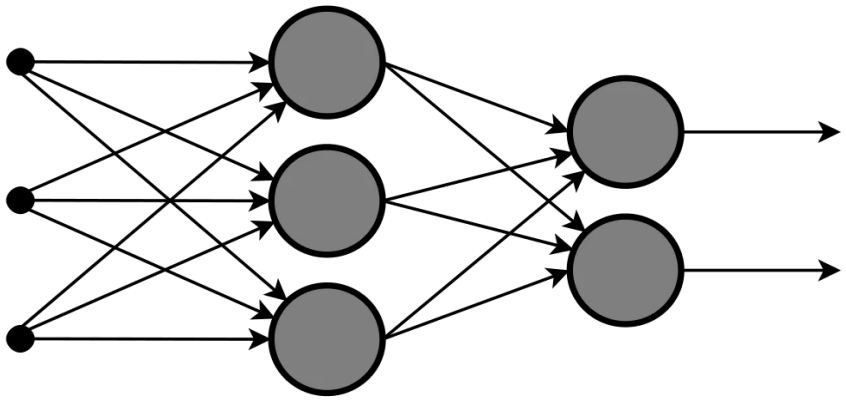
Spis treści

Rozdział 1

Wprowadzenie teoretyczne

1.1 Sieci neuronowe Feedforward

Sieci Feedforward znane także pod nazwą sieci jednokierunkowych cieszą się największym zainteresowaniem spośród wszystkich znanych architektur sztucznych sieci neuronowych. Spowodowane jest to ich prostą strukturą - łatwą do opisania, jak również prostymi i łatwymi do realizacji metodami uczenia tychże sieci. Należy podkreślić, że taka struktura sieci jest zbliżona do budowy mózgu, który również posiada strukturę warstwową, w dużej części jednokierunkową. W sieciach jednokierunkowych można wyróżnić uporządkowane warstwy neuronów (w tym warstwę wejściową i warstwę wyjściową). Liczba neuronów w każdej z warstw może być różna, przy czym w danej warstwie wszystkie neurony maja taką sama funkcję przejścia - neurony z różnych warstw mogą mieć różne funkcje przejścia. Połączenia występują tylko pomiędzy neuronami z sąsiednich warstw, wg zasady „każdy z każdym" i mają one charakter asymetryczny. Sygnały przesyłane są od warstwy wejściowej poprzez warstwy ukryte (jeśli występują) do warstwy wyjściowej.



Rysunek 1.1: Przedstawienie prostej sieci Feedforward

1.1.1 Model neurona

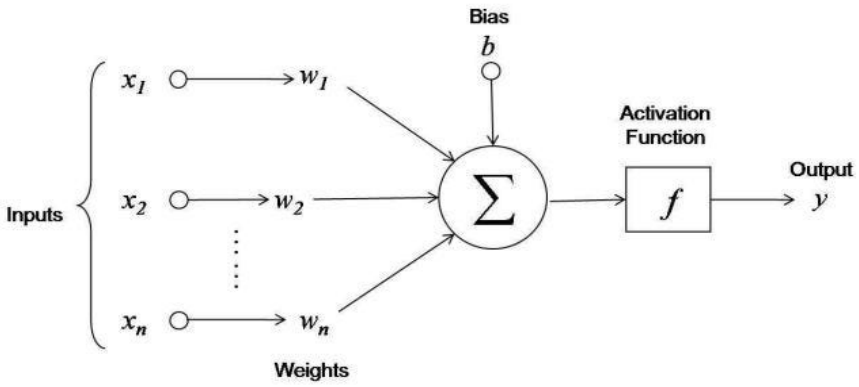
Model neuronu jest matematycznym modelem komórki nerwowej, która jest podstawową jednostką obliczeniową w sieciach neuronowych. Model ten jest abstrakcją biologicznego neuronu, który przetwarza sygnały elektryczne.

Podstawowy model neuronu składa się z trzech głównych części:

Wejścia: Neuron otrzymuje sygnały wejściowe $(x_1, x_2, \ldots, x_n)$, które mogą być reprezentowane jako wektor lub macierz. Te sygnały mogą reprezentować cechy, dane wejściowe lub wyniki pośrednie z innych neuronów.

Wagi: Każdemu sygnałowi wejściowemu przypisywana jest waga $(w_1, w_2, \ldots, w_n)$, która określa, jak duże znaczenie ma dany sygnał dla neuronu. Wagi są parametrami, które neuron uczy się optymalizować podczas procesu uczenia.

Funkcja aktywacji: Neuron ma funkcję aktywacji, która decyduje o tym, czy neuron zostanie aktywowany (wygeneruje wynik) lub pozostanie nieaktywny. Funkcja aktywacji jest zwykle nieliniowa, co pozwala neuronowi na modelowanie nieliniowych zależności między danymi wejściowymi a wynikami.



Rysunek 1.2: Przedstawienie neurona w sieci neuronowej

1.1.2 Warstwy sieci Feedforward

Warstwa sieci Feedforward składa się z pewnej liczby neuronów, z których każdy jest połączony z każdym neuronem z poprzedniej i następnej warstwy. Sygnały wejściowe przekazywane są do neuronów w warstwie, a następnie obliczane są sumy ważone tych sygnałów, które są przekazywane do funkcji aktywacji. Wyniki z warstwy neuronów stanowią wejście dla kolejnej warstwy i tak dalej, aż osiągnięta zostanie warstwa wyjściowa.

Wyróżniamy trzy rodzaje warstw sieci:

Warstwa wejściowa:
Warstwa wejściowa przyjmuje dane wejściowe i przekazuje je do kolejnych warstw. Liczba neuronów w warstwie wejściowej odpowiada liczbie cech lub wymiarów danych wejściowych.

Warstwy ukryte:
Warstwy ukryte są pośredniczącymi warstwami między warstwą wejściową a warstwą wyjściową. Mogą być jedna lub więcej. Każdy neuron w warstwie ukrytej otrzymuje sygnały od neuronów z poprzedniej warstwy i przekazuje je do neuronów w następnej warstwie. Warstwy ukryte uczą się reprezentować cechy i wzorce w danych.

Warstwa wyjściowa:
Warstwa wyjściowa generuje ostateczne wyniki sieci neuronowej. Liczba neuronów w warstwie wyjściowej zależy od rodzaju problemu. Na przykład, dla problemów klasyfikacji binarnej może być jeden neuron wyjściowy z funkcją aktywacji sigmoidalnej, podczas gdy dla problemów klasyfikacji wieloklasowej może być wiele neuronów wyjściowych z funkcją aktywacji softmax.

1.1.3 Funkcje aktywacji

Funkcja aktywacji w neuronie jest nieliniową funkcją, która decyduje o tym, czy neuron powinien zostać aktywowany (wygenerować sygnał wyjściowy) na podstawie sumy ważonej sygnałów wejściowych.

Do najbardziej powszechnych funkcji aktywacji zaliczamy:

Funkcja sigmoidalna:

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

Funkcja sigmoidalna przekształca wartości na przedział (0, 1). Jest wykorzystywana głównie w neuronach warstwy wyjściowej dla problemów klasyfikacji binarnej. Jej nieliniowość pozwala na modelowanie nieliniowych zależności między danymi wejściowymi a wyjściami. Jednak może wystąpić problem zanikającego gradientu, gdy wartości wejściowe są daleko od zera.

Funkcja ReLU:

$$f(x) = max(0, x)$$

Funkcja ReLU jest jedną z najpopularniejszych funkcji aktywacji stosowanych w sieciach neuronowych. Przekształca wartości na przedział [0, $+\infty$). Jej główną zaletą jest prostota obliczeniowa i eliminacja problemu zanikającego gradientu. ReLU działa dobrze w przypadku uczenia głębokich sieci neuronowych i może przyspieszyć proces uczenia.

Funkcja softmax:

$$f(\vec{x})_i = \frac{e^{x_i}}{\sum_{j=1}^{K} e^{x_j}}$$

Funkcja softmax jest często używana w warstwie wyjściowej dla problemów klasyfikacji wieloklasowej. Przekształca wartości na przedział (0, 1) i zapewnia, że suma wartości wyjściowych wynosi 1. Funkcja ta pozwala na interpretację wyników sieci neuronowej jako prawdopodobieństw przynależności do poszczególnych klas.

Opis poszczególnych zmiennych:

$\vec{x}$ - Wektor wejściowy funkcji softmax, składający się z $(x_0, ...x_K)$.
x_i- Jedna z wartości wektora $\vec{x}$ funkcji softmax przyjmująca dowolną wartość rzeczywistoliczbową.
$\sum_{j=1}^{K} e^{x_j}$ - Wartość w mianowniku gwarantuje, że wszystkie wartości wyjściowe funkcji będą sumować się do 1 i każda z nich będzie w zakresie (0, 1), tworząc w ten sposób prawidłowy rozkład prawdopodobieństwa.
K - Liczba klas występujących w multiklasyfikatorze.

1.1.4 Uczenie sieci

Uczenie sieci Feedforward polega na dostosowywaniu wag połączeń między neuronami w sieci w celu minimalizacji błędu predykcji i poprawy zdolności sieci do generalizacji na nowe dane. Proces ten można podzielić na kilka kroków, które są powtarzane przez wiele epok (iteracji) w trakcie uczenia:

Inicjalizacja wag: Wagi połączeń między neuronami są inicjalizowane losowo lub według określonego schematu. Początkowe wagi mają wpływ na szybkość i jakość uczenia sieci.

Przekazanie sygnału w przód (forward propagation): Dane wejściowe są podawane na warstwę wejściową sieci, a następnie przekazywane są przez kolejne warstwy neuronów przy obliczaniu sumy ważonej i stosowaniu funkcji aktywacji. Sygnał dociera do warstwy wyjściowej, która generuje wyniki predykcji.

Obliczenie błędu: Porównuje się wygenerowane wyniki sieci z oczekiwanymi wartościami wyjściowymi i oblicza się błąd predykcji. Najczęściej stosowaną miarą błędu jest funkcja kosztu, na przykład błąd średniokwadratowy (MSE).

Propagacja wsteczna: Błąd jest propagowany wstecz przez sieć, zaczynając od warstwy wyjściowej i przechodząc przez kolejne warstwy w kierunku warstwy wejściowej. Podczas propagacji wstecznej, dla każdego neuronu obliczane są gradienty funkcji kosztu względem wag połączeń, co pozwala na określenie kierunku i wielkości, w jakim wagi powinny zostać zmienione.

Aktualizacja wag: Na podstawie obliczonych gradientów, wagi połączeń między neuronami są aktualizowane w celu minimalizacji błędu. Najczęściej stosowanym algorytmem aktualizacji wag jest algorytm stochastycznego spadku gradientu (SGD) lub jego ulepszone wersje, takie jak np. Adam. Aktualizacja wag odbywa się poprzez pomnożenie gradientu przez współczynnik uczenia i zmianę wartości wag.

Powtarzanie epok: Powyższe kroki są powtarzane przez wiele epok (iteracji) w trakcie uczenia sieci. W każdej epoce dane uczące są podawane na sieć, a wagi są aktualizowane na podstawie błędów predykcji. Celem jest zminimalizowanie funkcji kosztu na zbiorze uczącym, a jednocześnie uzyskanie dobrej zdolności do generalizacji na nowe dane.

1.2 Algorytmy genetyczne

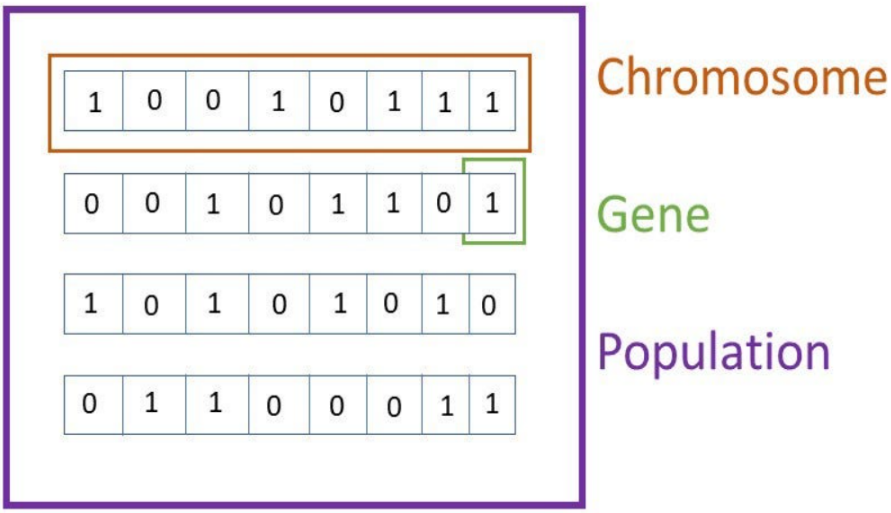
Główne założenia algorytmów genetycznych bazują na teorii Charlesa Dawina "przetrwają najsilniejsi". W odróżnieniu od tradycyjnych algorytmów, algorytmy genetyczne są dynamiczne, gdyż mogą ewoluować w czasie. Algorytmy genetyczne wyróżniają trzy główne cechy:

- Bazowane na populacji - algorytmy genetyczne mają za zadanie zoptymalizować proces, w którym obecne rozwiązania są niezadowalające. Ich rolą jest wygenerowanie nowych, lepszych rozwiązań. Zestaw rozwiązań, z których mają powstać nowe nazywamy populacją.
- Zorientowane na dopasowanie - istnieje wskaźnik dopasowania powiązany z każdym rozwiązaniem, pozwalający rozstrzygnąć czy dane rozwiązanie jest lepsze od innego. W tym celu definiujemy funkcję dopasowania, pozwalajacą zobrazować jak dobre jest dane rozwiązanie.
- Napędzane różnorodnością - jeśli w danej populacji nie występuje akceptowalne rozwiązanie, należy podjąć działanie mające na celu wygenerowanie lepszych rozwiązań. Rezultatem tego jest utworzenie wielu wariacji, które pozwalają osiągnąć ten cel.

W celu znalezienia rozwiązania za pomocą algorytmu genetycznego, losowe zmiany są stosowane do obecnych już rozwiązań w celu wygenerowania nowych. Jest to stopniowy proces działający poprzez dokonywanie małych i powolnych zmian w rozwiązaniach do momentu osiągnięcia najlepszego rozwiązania.
Proces działania algorytmu genetycznego prezentuje się następująco:

1. Inicjalizacja populacji
2. Dobór rodziców na podstawie ich wskaźnika dopasowania
3. Krzyżowanie rodziców (reprodukcja)
4. Mutacja potomków
5. Ocena potomstwa
6. Scalenie potomstwa z resztą populacji oraz sortowanie

Algorytmy genetyczne operują na populacji składającej się z rozwiązań, których liczbę określa wielkość populacji. Każde z rozwiązań nazywane jest osobnikiem, który posiada chromosom. Reprezentowany jest on przez genotyp (zbiór genów) oraz fenotyp (właściwa fizyczna reprezentacja chromosomu), które wyróżniają każdego osobnika.



Rysunek 1.3: Reprezentacja populacji, chromosomu i genu

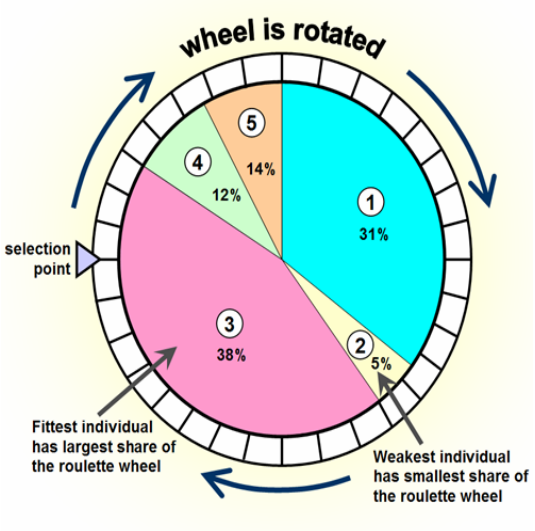
Warto w tym miejscu wspomnieć o procesie reprodukcji. Każda dwójka rodziców generuje dwóch potomków. Poprzez krzyżowanie dwóch osobników o wysokiej jakości rozwiązania spodziewamy się uzyskać lepszej jakości potomka niż jego rodzice. Kolejne iteracje dobierania wysokiej jakości osobników pozwalają na zwiększenie szansy zachowania najlepszych własności osobnika i pozbycia się tych gorszych. Ostatecznie pozwala to uzyskać optymalne (lub w gorszym wypadku akceptowalne) rozwiązanie, co jest przejawem elityzmu. Polega on na konstruowaniu populacji w taki sposób, że najlepsze osobniki są przenoszone z obecnej generacji do następnej bez zmian. Ta strategia gwarantuje, że jakość rozwiązania uzyskanego przez algorytm genetyczny nie będzie maleć w kolejnych generacjach.

1.2.1 Metody selekcji

Proces selekcji rodziców ma ogromne znaczenie dla tempa zbieżności algorytmu genetycznego, gdyż wybór odpowiednich rodziców prowadzi do lepszych pod względem dopasowania rozwiązań. Wyróżniamy kilka metod selekcji rodziców:

1. Steady-State-Selection (sss) - jest to metoda selekcji, w której tylko kilka osobników z populacji jest zastępowanych w jednym momencie, przez co większość osobników zostaje w następnej generacji. Istnieje kilka możliwości zastosowania tej metody. Przykładowo, rodzice są dobierani w sposób turniejowy, następnie są porównywani ze swoim potomstwem, a najlepsi wracają do populacji. Innym sposobem może być wybranie tylko najlepszych osobników na rodziców, a ich potomstwo zastępuje najgorszych osobników z populacji.

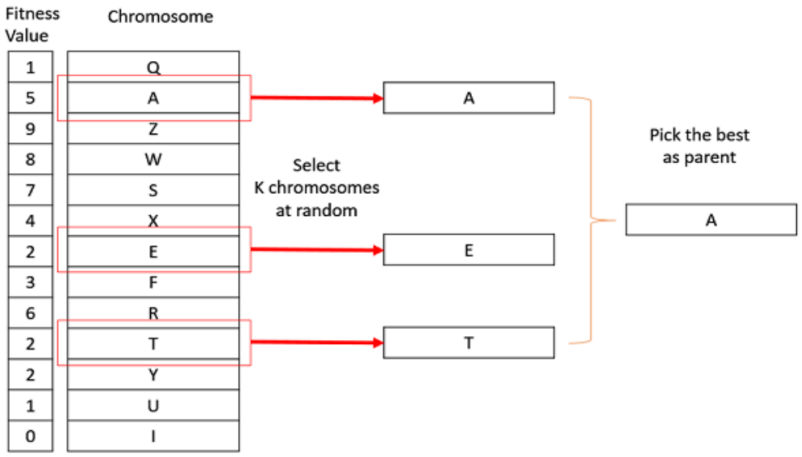
2. Roulette Wheel Selection (rws) - w przypadku tej metody, populacja jest dzielona na obszarze koła według swojego wskaźnika dopasowania. Następnie na obszarze koła dobierany jest jeden punkt stały i następuje "zakręcenie kołem ruletki". Ten osobnik, który znajdzie się w obszarze punktu stałego zostaje wybrany do krzyżowania. Proces ten jest powtarzany do momentu uzyskania odpowiedniej liczby rodziców.



Rysunek 1.4: Reprezentacja metody selekcji roulette

3. Stochastic Universal Selection (sus) - działa na podobnej zasadzie co ruletka, jednak zamiast jednego stałego punktu na obszarze koła mamy ich tyle, ilu rodziców potrzebujemy do krzyżowania. Pozwala to na wybranie wszystkich rodziców podczas jednego "zakręcenia kołem"

4. Tournament Selection - wybierane jest losowo K osobników i najlepszy z nich, tj. posiadający największe dopasowanie zostaje wybrany do reprodukcji.

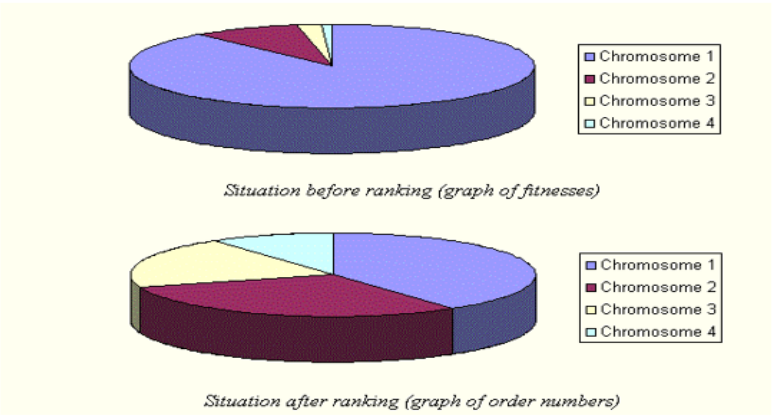


Rysunek 1.5: Reprezentacja metody selekcji turniejowej

5. Rank Selection - każdy z potencjalnych rodziców jest rankingowany według wartości swojego wskaźnika dopasowania (i=1...N, gdzie najgorszy osobnik otrzymuje 1 a najlepszy N - ilość osobników w populacji), następnie nadawane jest prawdopodobieństwo wylosowania na podstawie wzoru:

$$\frac{sum(fitness)}{i}$$

Na koniec następuje "zakręcenie kołem ruletki". Dalsza część pozostaje taka sama jak w metodzie ruletki.



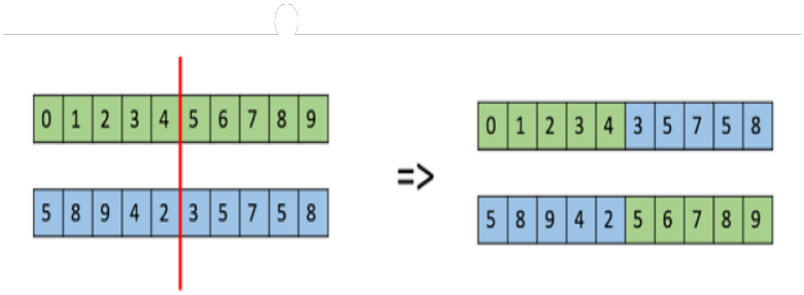
Rysunek 1.6: Reprezentacja metody selekcji rank

6. Random Selection - rodzice są wybierani losowo spośród populacji. Wartość ich dopasowania nie jest brana w tym przypadku pod uwagę

1.2.2 Metody krzyżowania

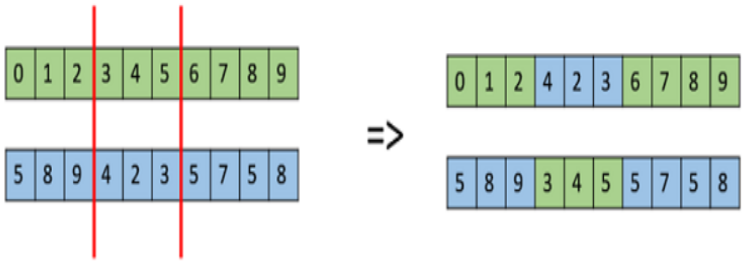
Metoda krzyżowania określa w jaki sposób generowane są dzieci z wybranych rodziców. Przykładowe metody definiujące sposób krzyżowania to:

1. Single-Point crossover - polega na wyznaczeniu jednego indeksu, za którym następuje zamiana genów jednego rodzica na drugiego.



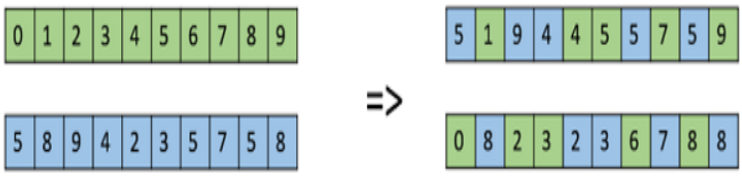
Rysunek 1.7: Reprezentacja single-point crossover

2. Multi-Point crossover - analogicznie do Single-Point, wyznaczane jest kilka indeksów, które powodują wymieszanie genów rodziców.



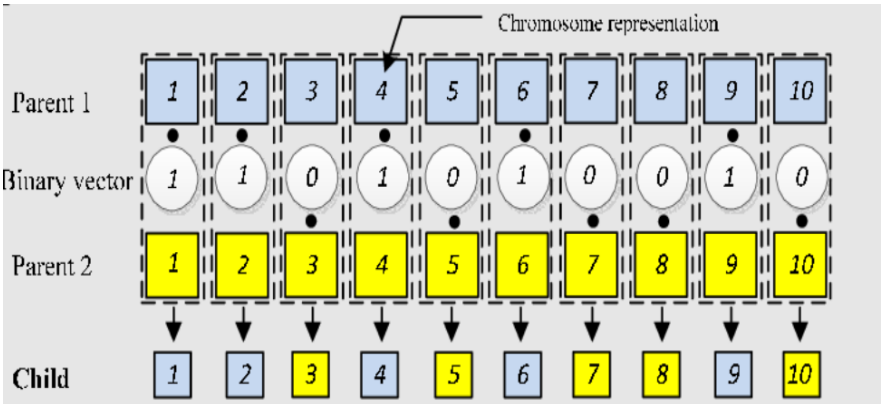
Rysunek 1.8: Reprezentacja multi-point crossover

3. Uniform Crossover - metoda podobna do rzutu monetą, gdzie rzut decyduje o tym, czy w danym miejscu zostanie dołączony gen drugiego z rodziców. Istnieje możliwość ustawić bias na jednego z rodziców, aby zwiększyć szansę otrzymania większej ilości informacji od niego.



Rysunek 1.9: Reprezentacja uniform crossover

4. Scattered Crossover - w tej metodzie zostaje wygenerowany losowy binarny wektor rozmiaru chromosoma. Geny dziecka są dobierane na podstawie tego wektora: jeśli wartość na danym indeksie jest równa 1, gen przechodzi od pierwszego rodzica, w przeciwnym wypadku potomek otrzymuje gen drugiego rodzica.



Rysunek 1.10: Reprezentacja scattered crossover

1.2.3 Metody mutacji

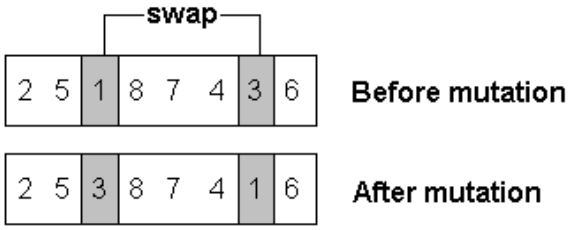
W kontekście algorytmów ewolucyjnych i optymalizacji, mutacja jest jednym z operatorów genetycznych, który jest stosowany w celu wprowadzenia zmienności w populacji i eksploracji nowych rozwiązań. Mutacja polega na losowej zmianie jednego lub więcej genów w chromosomie (reprezentujących rozwiązanie problemu), co prowadzi do powstania nowego osobnika w populacji.

W algorytmach ewolucyjnych, mutacja pełni ważną rolę w procesie ewolucji, umożliwiając wprowadzenie nowych cech i eksplorację przestrzeni rozwiązań. Bez mutacji, algorytm może utknąć w lokalnych optimum i niezdolny będzie do znalezienia lepszych rozwiązań. Mutacja wprowadza element losowości i zwiększa różnorodność w populacji, co pozwala na odkrycie nowych, potencjalnie lepszych rozwiązań.

W przypadku sieci Feedforward i optymalizacji z zastosowaniem algorytmów ewolucyjnych, mutacja może dotyczyć np. zmiany wag połączeń między neuronami, struktury sieci (dodawanie, usuwanie połączeń) lub innych parametrów, takich jak funkcje aktywacji czy współczynniki uczenia. Mutacje są wykonywane w celu poszukiwania optymalnych rozwiązań i poprawy wyników sieci neuronowych.

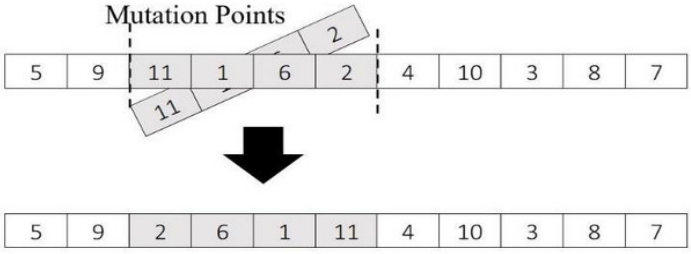
Można wyróżnić kilka sposobów na dokonywanie mutacji w sieciach neuronowych:

Mutacja poprzez zamianę: Z danego chromosomu bierzemy dwa losowe geny, pobieramy ich wartości i zamieniamy między sobą.



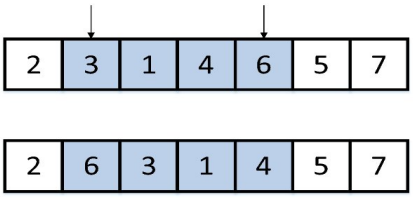
Rysunek 1.11: Mutacja poprzez zamianę

Mutacja poprzez inwersję: Wybierana jest pewna sekwencja genów, poczym zapisywana jest w odwrotnej kolejności.



Rysunek 1.12: Mutacja poprzez inwersję

Mutacja poprzez przemieszanie: Podobnie jak w przypadku inwersji, wybierana jest sekwencja genów, lecz tym razem ich wartości są miedzy sobą losowo tasowane.



Rysunek 1.13: Mutacja poprzez przemieszanie

.

Mutacja adaptacyjna: Dokonuje mutacji względem kondycji osobnika, tzn. w przypadku dzieci o wysokiej wartości dopasowania dokonywane jest mniej mutacji aniżeli dzieci o niskim stopniu dopasowania.

1.3 Wprowadzenie do PyGAD

PyGAD jest biblioteką open-source przeznaczoną do budowania algorytmów genetycznych oraz optymalizacji algorytmów uczenia maszynowego. Wspiera różne rodzaje metod krzyżowania, mutacji oraz selekcji rodziców. Odpowiednie przygotowanie funkcji dopasowania (fitness function) pozwala na optymalizację różnych rodzajów problemów klasyfikacji oraz regresji za pomocą algorytmu genetycznego. Biblioteka autorstwa Ahmeda Gada zawiera moduły wspierające pracę zarówno z sieciami Feedforward, jak i sieciami splotowymi. PyGAD może również integrować się z modelami stworzonymi za pomocą frameworków Keras i PyTorch.

1.3.1 Moduł PyGAD.GANN

Moduł ten odnosi się do algorytmu genetycznego z siecią neuronową **GANN** (**G**enetic **A**lgorithm **N**eural **N**etwork). Posiada w sobie klasę `pygad.gann.GANN()`, która umożliwia tworzenie, trenowanie oraz ewaluację sieci neuronowej (dla problemu regresji lub klasyfikacji) za pomocą algorytmu genetycznego. Aby utworzyć instancję klasy GANN należy podać następujące parametry:

- `num_solutions` - ilość rozwiązań (sieci neuronowych) w populacji. Parametr ten określa liczbę tworzonych identycznych sieci neuronowych, których parametry są optymalizowane poprzez algorytm genetyczny
- `num_neurons_input` - ilość neuronów w warstwie wejściowej
- `num_neurons_output` - ilość neuronów w warstwie wyjściowej
- `num_neurons_hidden_layers` - tablica przechowująca ilość neuronów w warstwach ukrytych
- `output_activation` - nazwa funkcji aktywacji dla warstwy wyjściowej
- `hidden_activations` - nazwa/nazwy funkcji aktywacji dla warstw ukrytych

W celu weryfikacji parametrów tworzonej sieci neuronowej, wywoływana jest funkcja `pygad.gann.validate_network_parameters()`.

1.3.2 Moduł PyGAD.GA

Moduł ten zawiera klasę GA (Genetic Algorithm), która pozwala na zbudowanie algorytmu genetycznego. Konstruktor klasy przyjmuje kilka parametrów, które pozwalają na dostosowanie algorytmu genetycznego do potrzeb danego problemu. Najważniejsze parametry GA to:

- `num_generations` - ilość generacji
- `num_parents_mating` - ilość osobników, która ma zostać wybrana jako rodzice
- `initial_population` - populacja początkowa
- `parent_selection_type` - sposób wyboru rodzica. Może przyjmować wartość `sss` dla Steady State Selection, `rws` dla wyboru ruletkowego, `sus` dla wyboru Stochastic Universal Selection, `random` dla wyboru losowego, `rank` dla wyboru rankingowanego lub `tournament` dla wyboru turniejowego
- `crossover_type` - rodzaj operacji krzyżowania. Przyjmuje wartości `single_point`, `two_points`, `uniform` lub `scattered`
- `mutation_type` - rodzaj mutacji. Przyjmuje wartości `random`, `swap`, `inversion`, `scramble`, `adaptive`
- `on_fitness` - funkcja dopasowania

Powyżej przedstawione zostały jedynie najbardziej istotne parametry pozwalające na zbudowanie algorytmu genetycznego. W tej chwili wersja biblioteki pyGAD 3.0.1 pozwala na dostosowanie 40 atrybutów w celu utworzenia instancji klasy GA.

Rozdział 2

Optymalizacja sieci neuronowej

2.1 Opis danych wejściowych

Dane które zostały użyte do utworzenia sieci neuronowej to zestaw 14 atrybutów i 303 instancji opisujących pacjentów pod względem parametrów takich jak np. wiek, płeć, tętno spoczynkowe itp. Wszystkie atrybuty, z wyjątkiem jednego (oldpeak) są reprezentowane w postaci liczb całkowitych, natomiast jeden pozostały stanowi wartość zmiennoprzecinkową.
Zbiór danych został wykorzystany w zadaniu klasyfikacji binarnej, gdzie wynikiem klasifikatora jest ustalenie czy u pacjenta występuje (1) choroba serca, bądź nie (0).

Atrybuty opisujące pacjenta:

age - Wiek pacjenta
sex - Płeć pacjenta [0 - 1]
cp - Typ bólu klatki piersiowej [0 - 4]
trestbps - Spoczynkowe ciśnienie krwii [mm Hg]
chol - Stężenie cholesterolu [mg/dl]
fbs - poziom cukru we krwi na czczo $>$ 120 [0 - 1]
restecg - Spoczynkowe wyniki elektrokardiograficzne [0 - 2]
thalach - Osiągnięte tętno maksymalne
exang - Dławica piersiowa wywołana wysiłkiem fizycznym [0 - 1]
oldpeak - Obniżenie odcinka ST wywołane wysiłkiem względem spoczynku
slope - Nachylenie szczytowego wysiłkowego odcinka ST [1 - 3]
ca - Liczba głównych naczyń zabarwionych za pomocą flouroskopii [0 - 3]
thal - Talasemia [3 / 6 / 7]
num - Diagnoza [0 - 1]

2.2 Opis metod

1. `GANN_instance.population_networks` - parametr wykorzystywany między innymi w funkcji predict. Jest to lista przechowująca referencję do wszystkich osobników (sieci neuronowych) w populacji

2. `pygad.nn.predict` - funkcja modułu pygad neural network pozwalająca na wykorzystanie wytrenowanej sieci neuronowej do predykcji wyników nowych danych. Funkcja przyjmuje parametry:

   - `last_layer` - referencję do ostatniej warstwy w architekturze sieci neuronowej. Możliwe jest tu podanie całej sieci reprezentującej danego osobnika. Funkcja wtedy odczyta ostatnią warstwę z architektury sieci neuronowej.
   - `data_inputs` - lista zawierająca dane wejściowe

   Wszystkie próbki danych wejściowych "karmią" sieć neuronową aby zwrócić zestaw predykcji

3. `pygad.gann.population_as_matrices` - funkcja modułu `pygad.gann` służąca do uzyskania listy zawierającej macierze wag dla każdego osobnika (sieci). Przyjmuje jako parametry:

   - `population_networks` - listę zawierającą referencje do warstw wyjściowych wszystkich osobników z populacji
   - `population_vectors` - listę zawierającą wagi wszystkich osobników w postaci wektorów. Te wektory zostają później przekonwertowane na macierze

4. `update_population_trained_weights` - funkcja modułu `pygad.gann` służąca do aktualizacji wag w warstwach każdej sieci neuronowej (każdego osobnika) . Przyjmuje parametr `population_trained_weights`, który jest listą wag zwróconą przez funkcję `population_as_matrices`.

5. `pygad.gann.population_as_vectors` - funkcja działająca na podobnej zasadzie do `population_as_matrices`, z tą różnicą, że zwraca listę zawierającą wektory wag dla każdego osobnika. Jako parametr przyjmuje jedynie `population_networks`, które zostało opisane wcześniej.

6. `ga_instance.best_solution` - jest to funkcja, którą możemy wykorzystać po zakończeniu działania algorytmu genetycznego. Pozwala na zwrócenie danych dotyczących najlepszego osobnika, który pozostał w ostatniej generacji algorytmu. Za jej pomocą możemy uzyskać następujące dane:

   - Solution - zwraca listę parametrów najlepszego osobnika

- Fitness value of the solution - zwraca wartość dopasowania najlepszego osobnika
- Index of the solution within the population - zwraca indeks najlepszego osobnika w populacji

7. `fitness_func` - jest to funkcja wykorzystywana w celu sprawdzenia uzyskanego rozwiązania. Na osobniku I, którego funkcja otrzymuje jako argument wejściowy dokonywana jest predykcja względem zbioru treningowego X. Następnie na podstawie uzyskanych predykcji Yp porównywana zostaje ilość poprawnych przewidywań względem rzeczywistych danych Y. Jeśli wartośc Yp_i oraz Y_i (gdzie $i = \langle 0, X_{size})$) są sobie równe to trafiają do tablicy K. Po sprawdzeniu wszystkich elementów zostaje określona wielkość tablicy K, która posłuży do sprawdzenia liczby poprawnych trafień przez danego osobnika.
   Na koniec wyciągamy i zwracamy procentową skuteczność rozwiązania $Q = \frac{K_{size}}{X_{size}} \cdot 100$.

   Argumenty funkcji:
   `ga_instance` - Instancja wykorzystywana do budowy algorytmu genetycznego (szerszy opis ów instancji znajduje się w dalszej części rozdziału "Opis metod").
   `solution` - Uzyskany osobnik (rozwiązanie), który zostanie sprawdzony pod względem dopasowania.
   `sol_idx` - Indeks osobnika, którego poddajemy funkcji sprawdzającej dopasowanie.

8. `callback_generation` - jest to funkcja wykorzystywana w celu śledzenia działań każdej (lub jeśli to określone - co *n-tej*) nowo powstałej generacji oraz umożliwiająca wprowadzanie na bieżąco zmian oddziałowujących na przyszłe generacje.
   Wykorzystywana w tym miejscu zmienna *population_matrices* przechowuje listę macierzy obecnej populacji przekształconej z formy wektorowej za pomocą metody *population_as_matrices*. Każda macierz zawiera w sobie wagi wszystkich warstw danej sieci, którą ta macierz opisuje. Następnie za pomocą metody *update_population_trained_weights* wraz z parametrem *population_matrices* wywołanej na obiekcie GANN_instance zostają przypisane zaktualizowane wagi dla każdej z sieci względem przekazanego parametru.

   Dalszą część funkcji stanowią komunikaty określające:

   - Numer generacji
   - Wartość dopasowania
   - Różnice pomiędzy aktualną generacją a poprzednikiem

2.3 Stworzenie sieci neuronowej

Pakiet *pygad.gann* zawiera w sobie specjalną metodę *GANN* do tworzenia sieci neuronowych. Sieci te są znacznie lepiej przystosowana do wykorzystywania na nich mutacji czy krzyżowań, aniżeli sieć z pakietu *Keras*, co przekłada się na szybsze dokonywanie obliczeń i ograniczenie czasu potrzebnego do uzyskania rezultatów.

Aby utworzyć instancje *GANN* jako parametry należy podać:

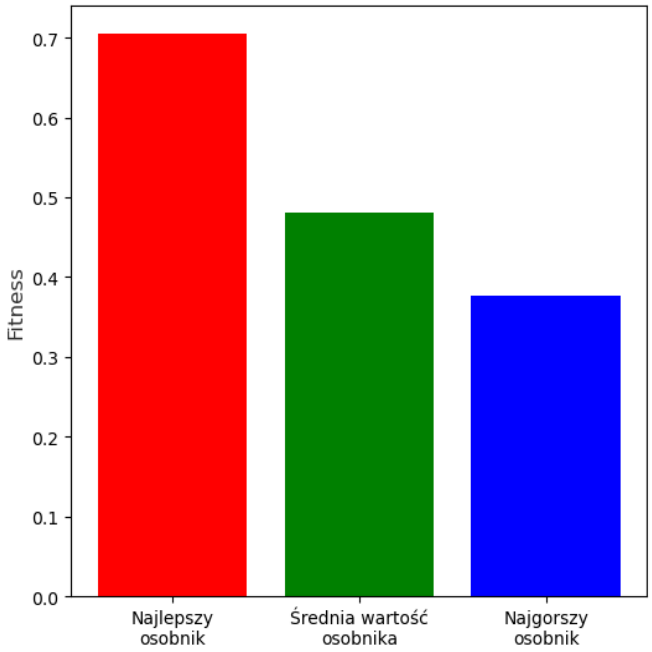
- num_solutions - Określający ilość osobników na jedną generację
- num_neurons_input - Określający ilość neuronów w warstwie wejściowej
- num_neurons_hidden_layers - Określający ilość neuronów w danej warstwie ukrytej
- num_neurons_output - Określający ilość neuronów na wyjściu
- hidden_activation - Określający funkcje aktywacji dla poszczególnej warstwy ukrytej
- output_activation - Określający funkcje aktywacji dla warstwy wyjściowej

Sieć, która stanowi bazę przeprowadzanych w późniejszej części pracy badań:

```
GANN_instance = pygad.gann.GANN(num_solutions=50,
                                num_neurons_input=13,
                                num_neurons_hidden_layers=[8,16,32],
                                num_neurons_output=2,
                                hidden_activations=["relu", "relu","relu"],
                                output_activation="sigmoid"
                               )
```

2.4 Badania symulacyjne

W niniejszym podpunkcie zostały zaprezentowane wyniki i wykresy uzyskane w trakcie przeprowadzonych badań symulacyjnych. Celem tych badań było zbadanie skuteczności algorytmów ewolucyjnych w doskonaleniu sieci neuronowych. W wyniku analizy otrzymanych wyników i wykresów, jesteśmy w stanie ocenić, jak dobrze sieć neuronowa poddana algorytmowi ewolucyjnemu radzi sobie w porównaniu z innymi sieciami zoptymalizowanymi z wykorzystaniem innych parametrów oraz względem sieci bazowych.
Każdy algorytm genetyczny został poddany 10-cio krotnemu testowi, w celu wyciągnięcia średniej z uzyskanych wyników oraz odnalezienie najlepszego/nagorszego osobnika w danej grupie parametrów.

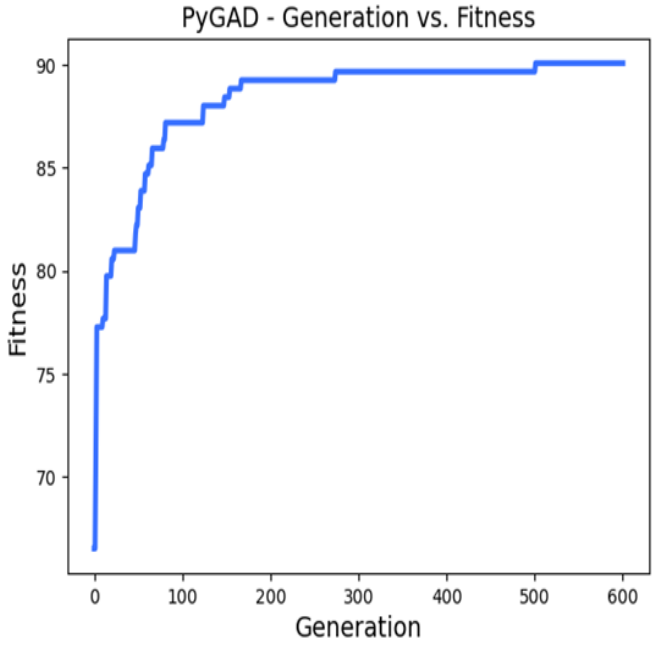


Rysunek 2.1: Wyniki dla instancji bazowej - bez wpływu algorytmu genetycznego

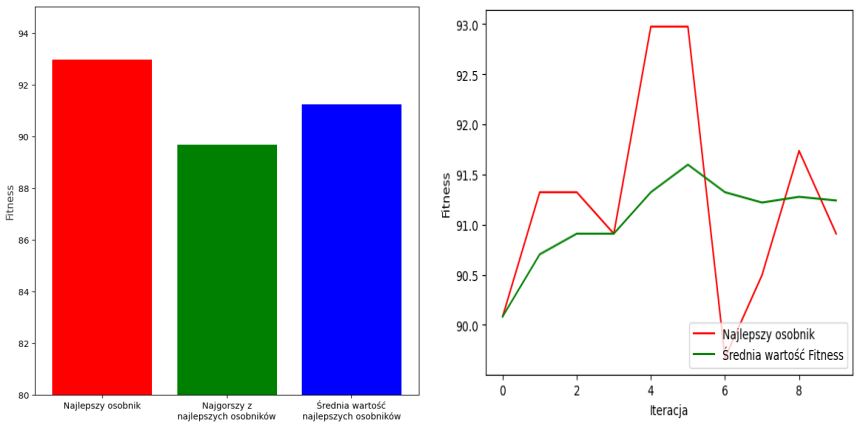
1. Algorytm genetyczny z parametrami:

   - Metoda wyboru rodzica: `Tournament`
   - Metoda krzyżowania: `Two Points`
   - Mutacje: `Adaptive`

   Przedstawienie rezultatów:



Rysunek 2.2: Przykładowy wykres optymalizacji sieci neuronowej dla parametrów: Tournament, Two Points, Adaptive

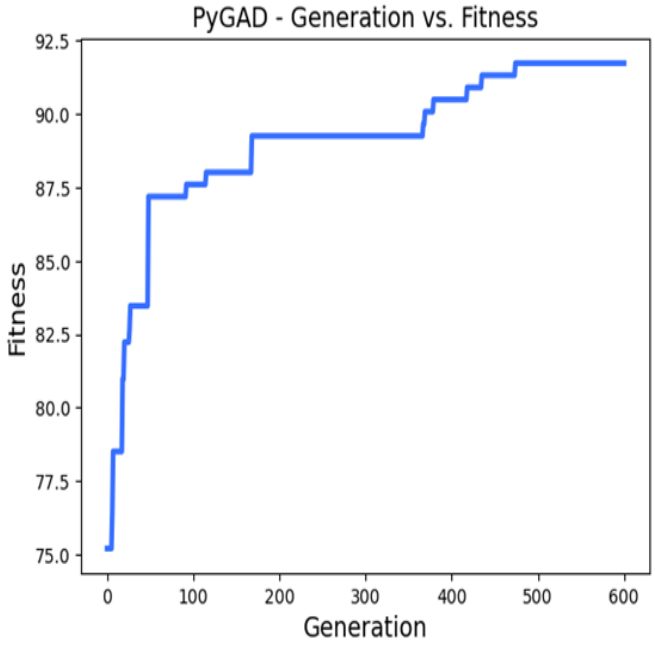


Rysunek 2.3: Wyniki dla algorytmu genetycznego o parametrach: Tournament, Two Points, Adaptive

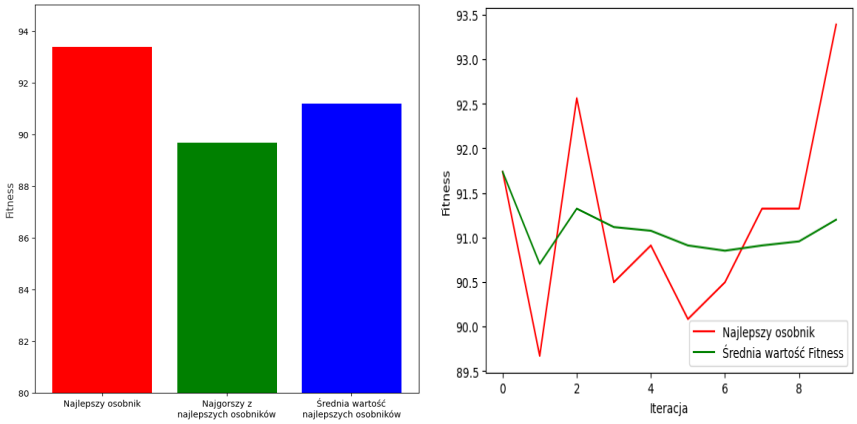
2. Algorytm genetyczny z parametrami:

   - Metoda wyboru rodzica: `Tournament`
   - Metoda krzyżowania: `Scattered`
   - Mutacje: `Adaptive`

   Przedstawienie rezultatów:



Rysunek 2.4: Przykładowy wykres optymalizacji sieci neuronowej dla parametrów: Tournament, Scattered, Adaptive

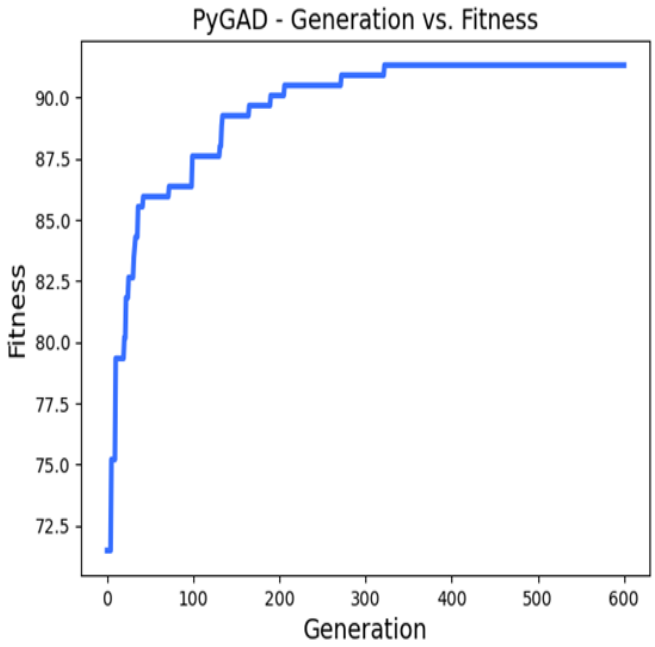


Rysunek 2.5: Wyniki dla algorytmu genetycznego o parametrach: Tournament, Scattered, Adaptive

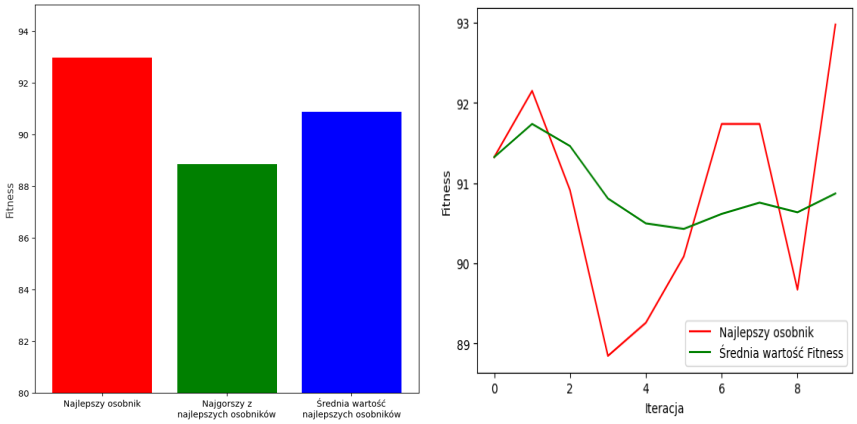
3. Algorytm genetyczny z parametrami:

   - Metoda wyboru rodzica: `Tournament`
   - Metoda krzyżowania: `Two Points`
   - Mutacje: `Random`

   Przedstawienie rezultatów:



Rysunek 2.6: Przykładowy wykres optymalizacji sieci neuronowej dla parametrów: Tournament, Two Points, Random

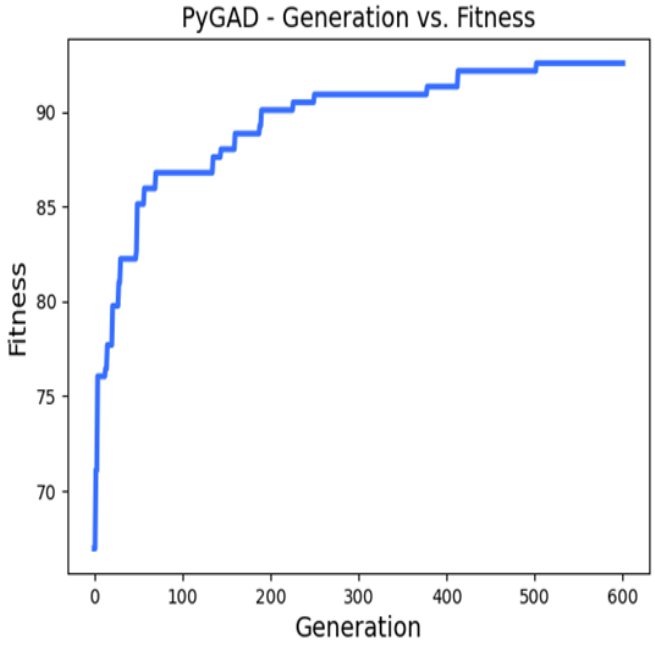


Rysunek 2.7: Wyniki dla algorytmu genetycznego o parametrach: Tournament, Two Points, Random

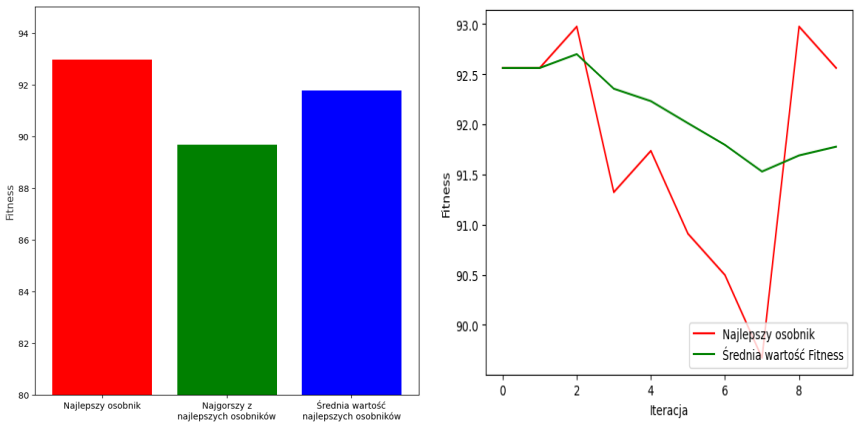
4. Algorytm genetyczny z parametrami:

   - Metoda wyboru rodzica: `Tournament`
   - Metoda krzyżowania: `Scattered`
   - Mutacje: `Random`

   Przedstawienie rezultatów:



Rysunek 2.8: Przykładowy wykres optymalizacji sieci neuronowej dla parametrów: Tournament, Scattered, Random

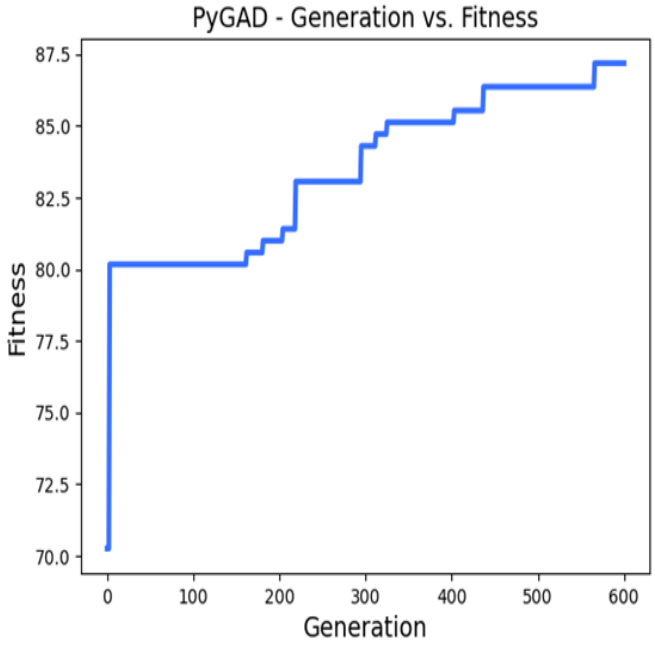


Rysunek 2.9: Wyniki dla algorytmu genetycznego o parametrach: Tournament, Scattered, Random

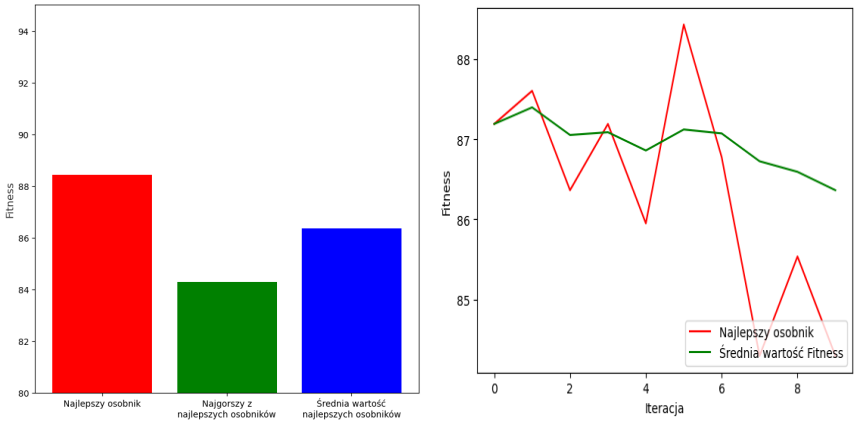
5. Algorytm genetyczny z parametrami:

   - Metoda wyboru rodzica: `RWS`
   - Metoda krzyżowania: `Scattered`
   - Mutacje: `Adaptive`

   Przedstawienie rezultatów:



Rysunek 2.10: Przykładowy wykres optymalizacji sieci neuronowej dla parametrów: RWS, Scattered, Adaptive

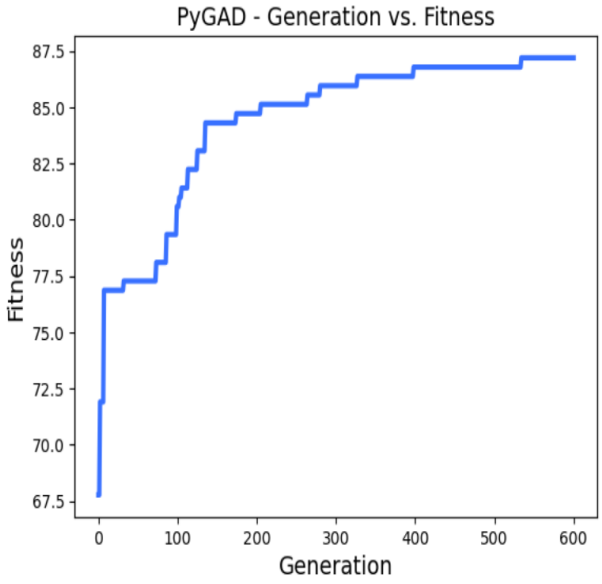


Rysunek 2.11: Wyniki dla algorytmu genetycznego o parametrach: RWS, Scattered, Adaptive

6. Algorytm genetyczny z parametrami:

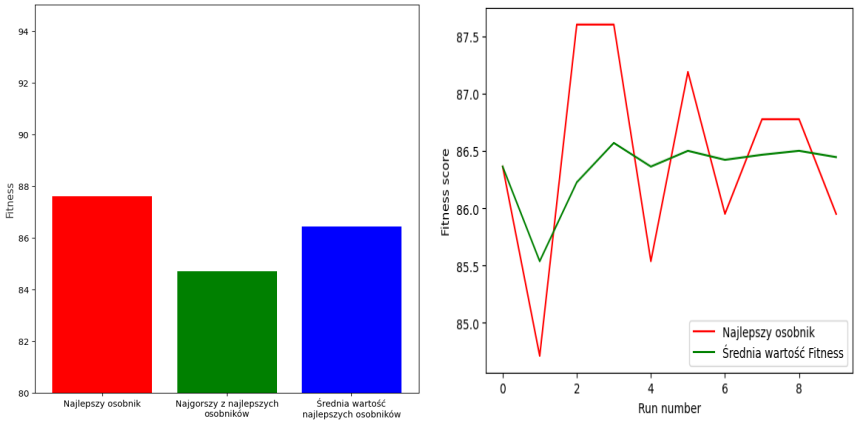
   - Metoda wyboru rodzica: `RWS`
   - Metoda krzyżowania: `Two Points`
   - Mutacje: `Adaptive`

   Przedstawienie rezultatów:




```
Fitness value of the best solution = 87.19008264462809
Best fitness value reached after 534 generations.
```

Rysunek 2.12: Przykładowy wykres optymalizacji sieci neuronowej dla parametrów: RWS, Two Points, Adaptive

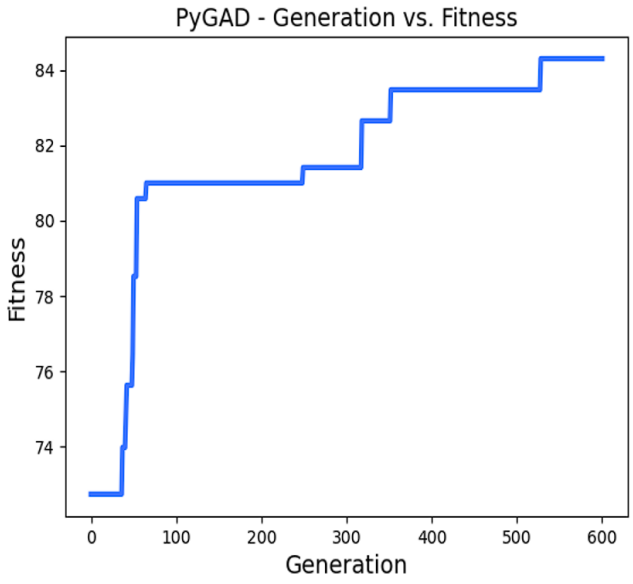


Rysunek 2.13: Wyniki dla algorytmu genetycznego o parametrach: RWS, Two Points, Adaptive

7. Algorytm genetyczny z parametrami:

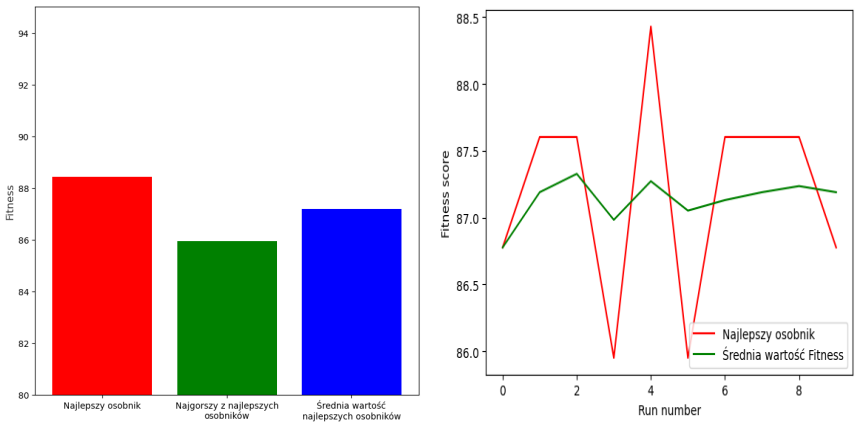
   - Metoda wyboru rodzica: `RWS`
   - Metoda krzyżowania: `Two Points`
   - Mutacje: `Random`

   Przedstawienie rezultatów:




```
Fitness value of the best solution = 84.29752066115702
Best fitness value reached after 528 generations.
```

Rysunek 2.14: Przykładowy wykres optymalizacji sieci neuronowej dla parametrów: RWS, Two Points, Random

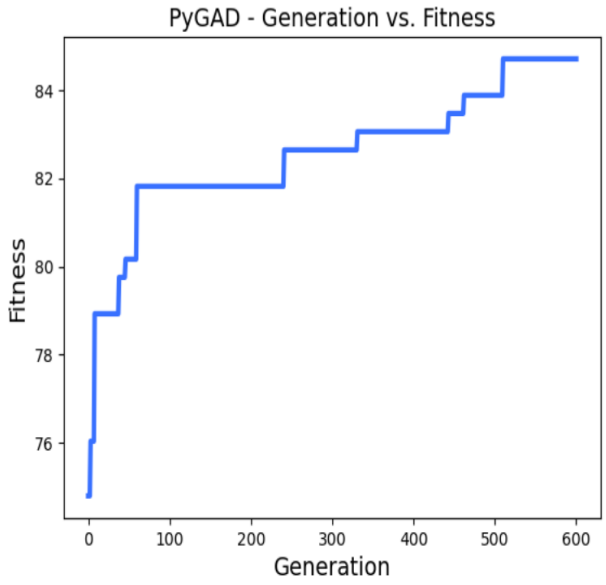


Rysunek 2.15: Wyniki dla algorytmu genetycznego o parametrach: RWS, Two Points, Random

8. Algorytm genetyczny z parametrami:

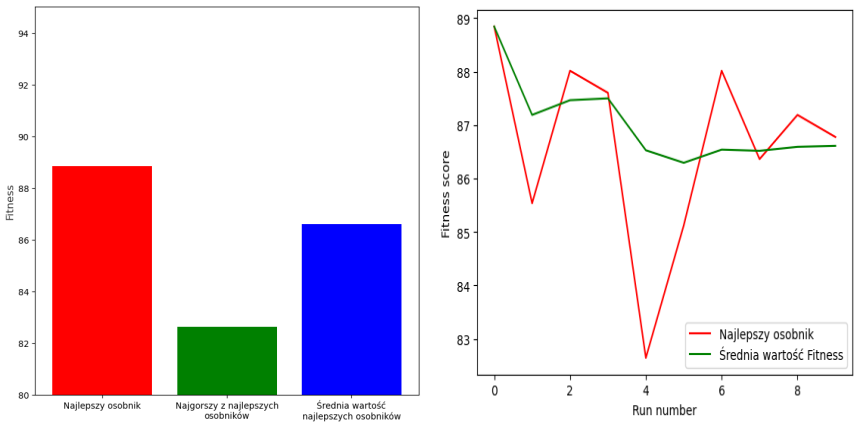
   - Metoda wyboru rodzica: `RWS`
   - Metoda krzyżowania: `Scattered`
   - Mutacje: `Random`

   Przedstawienie rezultatów:

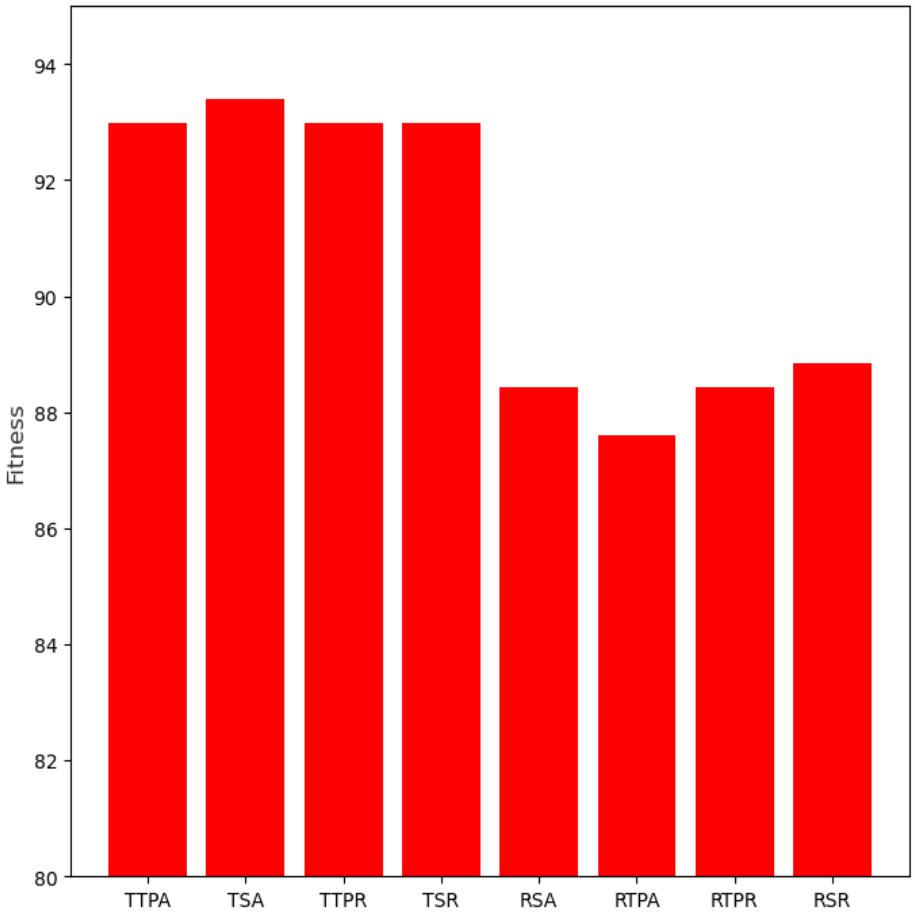



```
Fitness value of the best solution = 84.71074380165288
Best fitness value reached after 510 generations.
```

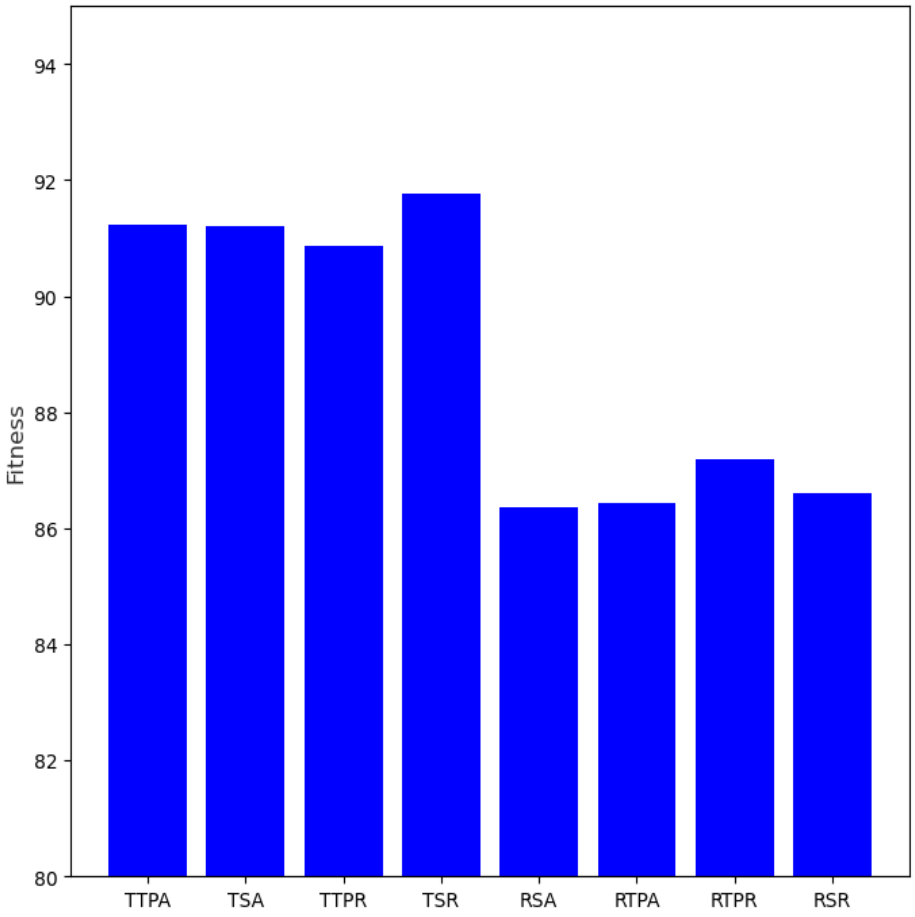
Rysunek 2.16: Przykładowy wykres optymalizacji sieci neuronowej dla parametrów: RWS, Two Points, Random



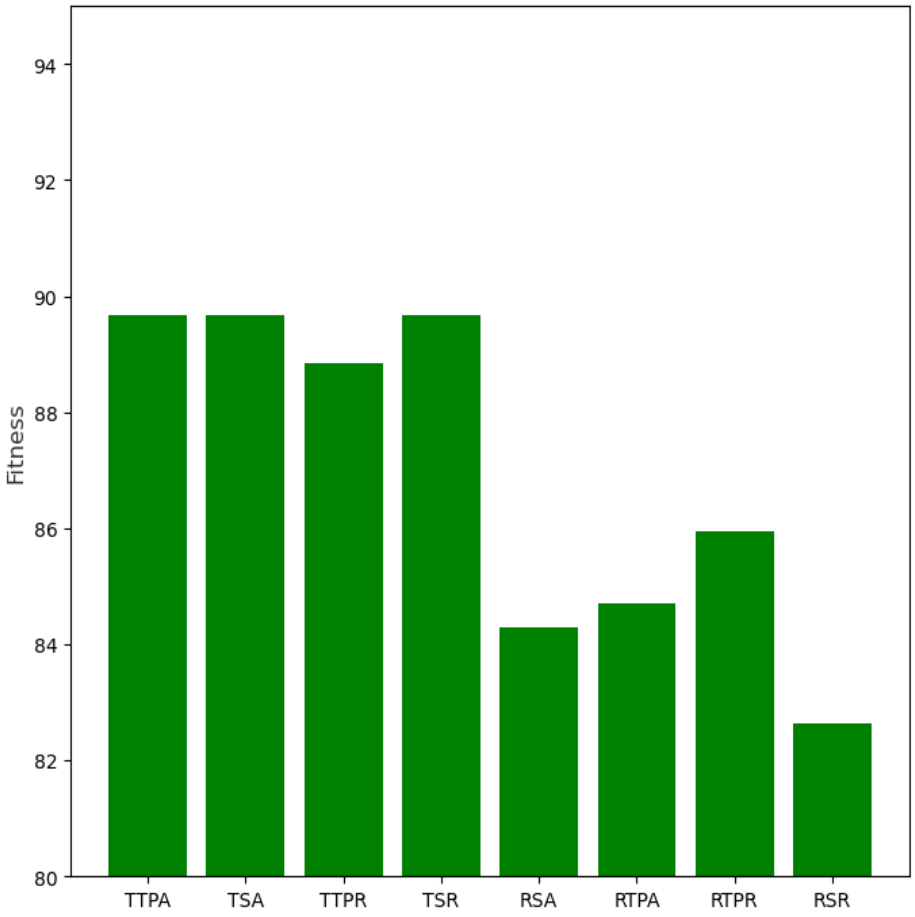
Rysunek 2.17: Wyniki dla algorytmu genetycznego o parametrach: RWS, Two Points, Random



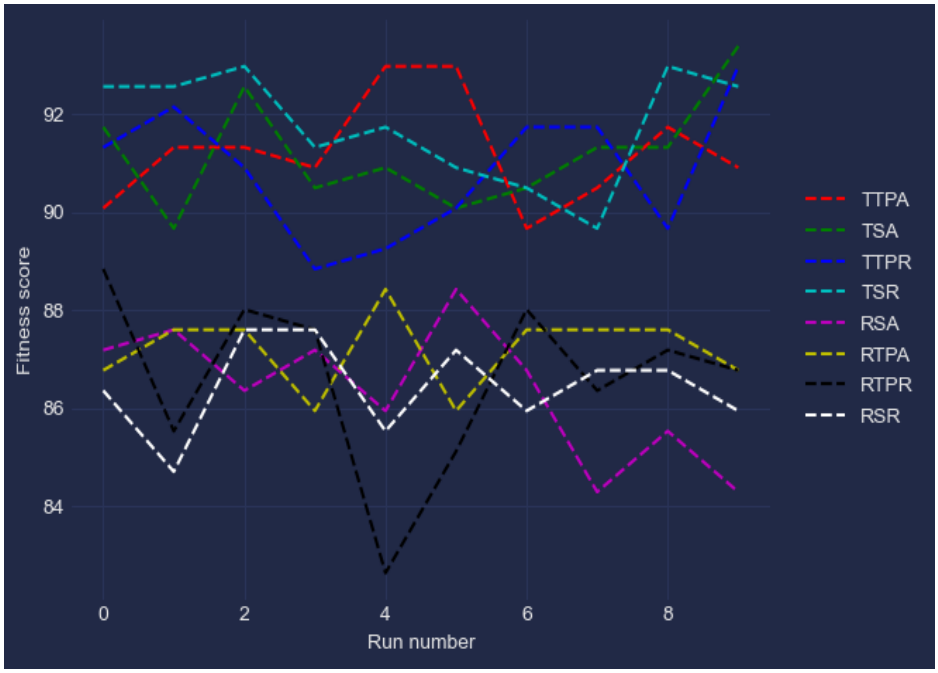
Rysunek 2.18: Zestawienie najlepszych osobników z danego algorytmu ewolucyjnego



Rysunek 2.19: Zestawienie średniej z wszystkich najlepszych osobników z danego algorytmu ewolucyjnego



Rysunek 2.20: Zestawienie najgorszych z wszystkich najlepszych osobników z danego algorytmu ewolucyjnego



Rysunek 2.21: Zestawienie najlepszych osobników z danego algorytmu ewolucyjnego

Rozdział 3

Podsumowanie

3.1 Wyniki

1. Wpływ algorytmu genetycznego na optymalizację sieci neuronowej
   Po przenalizowaniu wszystkich przypadków testowych możemy zauważyć znaczącą poprawę jakości dokonywanych predykcji względem instancji bazowej, której początkowa dokładność predykcji w przypadku najlepszego osobnika wynosiła 71,3%, natomiast średnia miała wartość 49,51%. Po zastosowaniu algorytmu genetycznego w każdym przypadku odnotowano poprawę klasyfikacji w granicach 17-22 punktów procentowych względem najlepszego osobnika instacji bazowej (przedział dla średniej wynosi 36-42 p.p. poprawy).

2. Sktuecznośc wyboru rodzica
   Bardzo duże znaczenie w badaniu okazał się mieć sposób wyboru rodzica. Zaobserwowano znaczące różnice pomiędzy doborem Turniejowym a Ruletką, gdzie najgorszy ze wszystkich najlepszych osobników wyboru rodzica metodą Turniejową wciąż mógł się pochwalić lepszym wynikiem (88,84) anieżeli najlepszy osobnik metody Ruletki (87,19). Powodem tego typu zachowania jest sposób działania Turnieju, który cały czas dąży przy tworzeniu nowej generacji do elityzmu wybieranego osobnika (rodzica), gdzie w przypadku Ruletki osobnik, który ma stać się rodzicem jest wybierany w sposób losowy. Nie należy jednak z tego powodu całkowicie skreślać sposobu Ruletki, który długofalowo może dać lepsze rezultaty w porównaniu do Turnieju, ponieważ ze swojego sposobu działania tj. wybierania losowego osobnika jest w stanie unikać wpadania w minima lokalne, gdzie w przypadku Turnieju uzyskujemy po zaledwie kilku generacjach dość dobre wyniki, lecz bardzo szybko możemy w ów minimum lokalne wpaść i nie będzie możliwe dalsze optymalizowanie sieci (lub będzie ono bardzo znikome).

3. Rozbieżność najgorszych z najlepszych osobników Ruletki
   Kolejną obserwacją, która rzuca się w oczy jest spora rozbieżność wyników najgorszych z najlepszych osobników metody Ruletki. Różnica między wynikiem osobnika z konfiguracji RTPR (RWS, Two Points, Random) a osobnika z konfiguracji RSR (RWS, Scattered, Random) sięga 3,3 punkta procentowego względem dopasowania. Dla porównania różnica ta w przypadku selekcji Turniejowej wynosi zaledwie 0,72 punkta procentowego. Powodem tego typu wahania jest wspomnianie wcześniej losowe wybieranie osobników w metodzie Ruletki, gdzie kilkadziesiąt/kilkaset razy może zostać wybrany osobnik dający gorsze rezultaty w porównaniu do jego poprzednika. W przypadku takiego scenariusza możliwe jest przejście po wszystkich określonych generacjach bez uzyskania zauważalnej optymalizacji. Z kolei zważając na losowość tego typu podejścia możliwe jest "trafienie" najlepszego możliwego rozwiązania w którejkolwiek przeprowadzanej generacji, uzyskując najbardziej optymalne rozwiązanie.

4. Średnia wartość dopasowania
   Zdecydowanie lepszymi wynikami dla 600 generacji może poszczycić się selekcja turniejowa. Bez względu na dobór pozostałych parametrów, testy przeprowadzone z zastosowaniem tej metody zwracały średnie wyniki na bardzo podobnym poziomie. Najwyższy średni wynik zwrócił zestaw testów w konfiguracji TSR (Selection Tournament, Crossover Scattered, Mutation Random) wynoszący 91.78% dokładności na zbiorze treningowym. Najgorszy otrzymany średni wynik dla tego typu selekcji wyniósł 90.87%, który wykazała konfiguracja TTPR (Selection Tournament, Crossover Two Points, Mutation Random). Uzyskane rezultaty w tym zakresie generacji dają zdecydowaną przewagę metodzie turniejowej, która w porównaniu z testami dla selekcji ruletki wypadła lepiej o około 4% (najlepszy wynik: 87.19% dla RTPR, najgorszy 86.36% dla RSA (Selection Roulette, Crossover Scattered, Mutation Adaptive)).

   Podsumowując powyższe obserwacje i wnioski, można stwierdzić że selekcja rodziców odgrywa największe znaczenie w porównaniu do wyboru typu mutacji czy krzyżowania. Metoda Turniejowa potrafi zwrócić bardzo szybko, bo po zaledwie kilkunastu generacjach w miarę obiecujące wyniki, które mogłyby w przypadku wielu innych zadań okazać się wystarczające. Jednakowoż w przypadku, gdzie zależy nam na jeszcze wyższej dokładności, lepiej może sprawdzić się metoda Ruletki. Sposób ten będzie potrzebował znacznie większej ilości generacji, co za tym idzie - czasu, lecz będzie w stanie w procesie długofalowym uzyskać nam lepsze wyniki niż ma to miejsce w przypadku metody Turniejowej.

Bibliografia

[1] Dokumentacja PyGAD

[2] www.researchgate.net - Grafika Scattered-crossover

[3] Genetic Algorithms - Crossover

[4] Roulette wheel selection

[5] Genetic Algorithms - Parent Selection

[6] Genetic Algorithms vs Neural Networks

[7] How to perform Roulette wheel and Rank based selection in a genetic algorithm?

[8] Genetic Algorithm (GA) Introduction

[9] Heart Disease Data Set

[10] Algorytmy ewolucyjne - Wykład

Dodatek A

Kod programu

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import pygad
import numpy
import pygad.nn
import pygad.gann
from sklearn.model_selection import train_test_split,  cross_val_score
from sklearn.metrics import accuracy_score
from sklearn.preprocessing import StandardScaler
import tensorflow as tf
import random
import mplcyberpunk

df = pd.read_csv("heart.csv")
df.head()
df.info()

y = df.output
X = df.drop(columns="output")

X_train, X_test, y_train, y_test = train_test_split(X,
                                                    y,
                                                    stratify = y,
                                                    test_size = .2,
                                                    random_state = 42)
scaler = StandardScaler()
scaler.fit(X_train)
X_train = scaler.transform(X_train)
X_test = scaler.transform(X_test)
```

```
X_train.shape, y_train.shape, X_test.shape, y_test.shape

mean_population_fitness = []

def fitness_func(ga_instance, solution, sol_idx):
    global GANN_instance, data_inputs, data_outputs
    if sol_idx is not None:
        predictions = pygad.nn.predict(
last_layer=GANN_instance.population_networks[sol_idx],
data_inputs=data_inputs)
        correct_predictions = numpy.where(predictions == data_outputs)[0].size
        solution_fitness = (correct_predictions/data_outputs.size)*100
        mean_population_fitness.append(solution_fitness)
    else:
        solution_fitness=0
    return solution_fitness

last_fitness = 0

def callback_generation(ga_instance):
    global GANN_instance, last_fitness

    population_matrices = pygad.gann.population_as_matrices(
population_networks=GANN_instance.population_networks,
      population_vectors=ga_instance.population)

    GANN_instance.update_population_trained_weights(
population_trained_weights=population_matrices)
    if ga_instance.generations_completed % verbose == 0:
        print("Generation = {generation}"
.format(generation=ga_instance.generations_completed))
        print("Fitness    = {fitness}"
.format(fitness=ga_instance.best_solution()[1]))
        print("Change     = {change}"
.format(change=ga_instance.best_solution()[1] - last_fitness))

    last_fitness = ga_instance.best_solution()[1].copy();

data_inputs = X_train
data_outputs = y_train.values

num_inputs = data_inputs.shape[1]
num_classes = 2
```

```
GANN_instance = pygad.gann.GANN(num_solutions=num_solutions,
                                num_neurons_input=num_inputs,
                                num_neurons_hidden_layers=[8,16,32],
                                num_neurons_output=num_classes,
                                hidden_activations=["relu", "relu","relu"],
                                output_activation="sigmoid"
                               )
population_vectors = pygad.gann.population_as_vectors(
population_networks=GANN_instance.population_networks)

initial_population = population_vectors.copy()

num_solutions = 50
num_generations = 600

parent_selection_type = "rws"
keep_parents = -1

crossover_type = "two_points"

mutation_type = "adaptive"

num_parents_mating = 4

crossover_probability = 0.4

best_all_gen = []
best_fitness_individual = 0
best_test_accuracy = 0
verbose = 100
ga_instance = pygad.GA(num_generations=num_generations,
                       num_parents_mating=num_parents_mating,
                       initial_population=initial_population,
                       fitness_func=fitness_func,
                       mutation_percent_genes=(25,10),
                       mutation_probability=(0.15,0.05),
                       mutation_num_genes=(4,2),
                       num_genes=len(X_train),
                       parent_selection_type=parent_selection_type,
                       crossover_probability=crossover_probability,
                       crossover_type=crossover_type,
                       mutation_type=mutation_type,
                       keep_parents=keep_parents,
```

```
                        on_generation=callback_generation)

ga_instance.run()

ga_instance.plot_fitness()

solution, solution_fitness, solution_idx = ga_instance.best_solution()
print("Fitness value of the best solution = {solution_fitness}"
.format(solution_fitness=solution_fitness))
y_pred_ts = pygad.nn.predict(
last_layer=GANN_instance.population_networks[solution_idx],
data_inputs=X_test)

test_acc = accuracy_score(y_test, y_pred_ts)
if test_acc > best_test_accuracy:
    best_test_accuracy = test_acc

if solution_fitness > best_fitness_individual:
    best_fitness_individual = solution_fitness
best_all_gen.append(solution_fitness)

if ga_instance.best_solution_generation != -1:
    print("Best fitness value reached after {best_solution_generation} generations.'
.format(best_solution_generation=ga_instance.best_solution_generation))

predictions = pygad.nn.predict(last_layer=GANN_instance.population_networks[solutio
                               data_inputs=data_inputs)

import statistics

num_wrong = numpy.where(predictions != data_outputs)[0]
num_correct = data_outputs.size - num_wrong.size
accuracy = 100 * (num_correct/data_outputs.size)
print("Number of correct classifications : {num_correct}."
.format(num_correct=num_correct))
print("Number of wrong classifications : {num_wrong}."
.format(num_wrong=num_wrong.size))
print("Classification accuracy : {accuracy}."
.format(accuracy=accuracy))
print("Mean population fitness: {mean_fitness}"
.format(mean_fitness=statistics.mean(mean_population_fitness)))

y_pred_tr = pygad.nn.predict(
last_layer=GANN_instance.population_networks[solution_idx],
```

```
data_inputs=X_train)
y_pred_ts = pygad.nn.predict(
last_layer=GANN_instance.population_networks[solution_idx],
data_inputs=X_test)

print('Train Accuracy score: {0:0.4f}'.format(accuracy_score(y_train, y_pred_tr)))
print('Test Accuracy score: {0:0.4f}'.format(accuracy_score(y_test, y_pred_ts)))

best_all_tests.append(best_all_gen)
fitness_all_tests.append(best_fitness_individual)
average_y_accuracy.append(best_test_accuracy)

print(best_all_tests, fitness_all_tests)
```