Uniwersytet Jagieloński w Krakowie
Wydział Fizyki, Astronomii i Informatyki Stosowanej

Jan Sołtysik
Nr albumu: 1145105

Zastosowanie algorytmu genetycznego do optymalizacji struktury sieci neuronowej

Praca licencjacka
na kierunku Informatyka

Praca wykonana pod kierunkiem: dr. hab. Anny Paszyńskiej
Zakład Projektowania i Grafiki Komputerowej

Kraków 16 czerwca 2020

Spis treści

1 Wstęp

W ostatnich latach **Sztuczne Sieci Neuronowe** (**SSN**) i uczenie głębokie są tematem numer jeden w świecie informatyki. Jest to jednak stosunkowo stara dyscyplina ponieważ jej początek sięga roku 1943 w którym został zaprezentowany model pojedynczego perceptronu.
Jednak dopiero w latach dwutysięcznych zaczęła ona pokazywać swój prawdziwy potencjał. Przyczyniło się do tego wiele czynników m.in. rozwój technologi, ogólnodostępne duże pokłady danych itd. Wiele firm zaczyna to wykorzystywać wprowadzając różnego rodzaju zastosowania dla SSN np. Tesla i samoprowadzące się samochody, system rekomendacji filmów na platformie Netflix, filtry spam w skrzynkach e-mail, boty uczące się grać w gry z OpenAI i wiele wiele innych.
Zbudowanie sieci składającej się z dużej ilości warstw nie jest jednak zadaniem prostym. Dlatego zaczęto opracowywać metody automatycznego budowania sieci neuronowych. Metody te umożliwiają osobom niezaznajomionym z uczeniem maszynowym na korzystanie z SSN. Jednym z możliwych podjeść są **Algorytmy Genetyczne** (**AG**), które podobnie jak SSN nie są wynalazkiem ostatnich lat (rok 1960 John Holland). Znajdują one jednak wciąż nowe zastosowania.

Celem pracy jest zaproponowanie i zaimplementowanie algorytmu genetycznego umożliwiającego znalezienie optymalnej SSN dla rozważanych zadań (rozważam zadania klasyfikacji i regresji). Zaimplementowany w ramach pracy algorytm genetyczny znajduje optymalną SSN poprzez określenie optymalnej liczby warstw, rodzaju warstw pośrednich w sieci oraz ich hiperparametrów.

Struktura pracy jest następująca. W kolejnym rozdziale wyjaśnię sposób działania i konstrukcje elementarnego algorytmu genetycznego oraz SSN. W rozdziale trzecim zaproponuję zmodyfikowany algorytm genetyczny umożliwiający znalezienie optymalnej struktury SSN. Rozdział czwarty zawiera techniczne informacje dotyczące implementacji opisanego algorytmu. Ostatni rozdział prezentuje optymalne sieci znalezione przez stworzony w ramach pracy algorytm dla problemu klasyfikacji na zbiorze **Fashion Mnist** - zadanie rozpoznania kategorii ubrania podając jego zdjęcie, oraz regresji na zbiorze **Boston Housing** - przewidywanie ceny domu w zależności od wartości zadanych dla jego parametrów.

2 Wprowadzenie teoretyczne

2.1 Algorytm Genetyczny

Algorytmy genetyczne są to algorytmy poszukiwania zainspirowane mechanizmem doboru naturalnego oraz dziedziczności. Łączą w sobie zasadę przeżycia najlepiej przystosowanych osobników z systematyczną, choć zrandomizowaną wymianą informacji wprowadzoną przez operatory genetyczne takie jak krzyżowanie i mutacja, które również zainspirowane są przyrodą. W każdym pokoleniu powstaje nowy zespół sztucznych organizmów, utworzonych z połączenia fragmentów najlepiej przystosowanych przedstawicieli poprzedniego pokolenia. Oprócz tego sporadycznie wyporóbowuje się nową część składową. Element losowości nie oznacza że algorytmy genetyczne sprowadzają się do zwykłego błądzenia przypadkowego. Dzięki wykorzystaniu przeszłych doświadczeń algorytm określa nowy obszar poszukiwań o spodziewanej podwyższonej wydajności. Algorytm genetyczny jest przykładem procedury używającej wyboru losowego jako "przewodnika" w wysoce ukierunkowanym poszukiwaniu w zakodowanej przestrzeni rozwiązań [5].

Poniżej znajduje się podstawowy algorytm genetyczny przedstawiony w postaci pseudokodu [3]:

Algorithm 1: Podstawowy Algorytm genetyczny

Input: Funkcja oceny $f(\mathbf{x})$
Data: $\mathbf{x}_i(t)$ - i-ty osobnik z generacji nr. t(najczęściej reprezentowany jako ciąg znaków),
n_x - wymiar każdego osobnika
Output: Wektor $\hat{\mathbf{x}}$ dla którego $f(\hat{\mathbf{x}})$ jest lokalnym minimum
Niech $t = 0$ będzie licznikiem generacji. Wygeneruj n_x - wymiarową populację $\mathcal{C}(0)$, składającą się z n osobników.
while *Warunek końcowy nie jest prawdziwy* **do**
  Oblicz przystosowanie, $f(\mathbf{x}_i(t))$ każdego osobnika $\mathbf{x}_i(t)$ z populacji.
  W celu stworzenia potomstwa do najlepiej przystosowanych osobników zastosuj operatory genetyczne np. krzyżowanie i mutacja.
  Z nowo powstałych osobników stwórz nową populację $\mathcal{C}(t+1)$.
  Przejdź do nowej generacji. $t = t + 1$.
end

2.1.1 Podstawowa reprezentacja

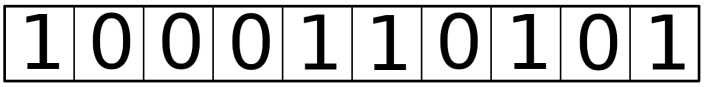
Jednym z następstw inspiracji biologicznych AG jest słownictwo zaczerpnięte z genetyki którym będę się posługiwać w tej pracy. **Fenotypem** nazywamy układ złożony z pewnej liczby parametrów opisujących cechy rozwiązania. Osobniki występujące w populacji nazywane są **genotypami**, które składają się z zakodowanej wersji opisanych dla fenotypu parametrów. Konkretna zakodowana cecha (parametr) nazywana jest **genem**. Każdy gen może przyjmować różne wartości, które nazywane są **allelami**. Pozycję na jakiej znajduje się gen nazywamy **locusem**. **Chromosomem** nazywamy uporządkowany zestaw genów. Genotyp składa się z jednego lub więcej chromosomów.[5].

Poniższa tabela przedstawia używane przeze mnie odpowiedniki:

Genetyka	Algorytmy Genetyczne
genotyp	osobnik w populacji
gen	składowa osobnika
allel	możliwe warianty składowych
locus	pozycja składowej
chromosom	struktura złożona ze składowych

Tabela 1: Słownictwo używane w AG.

W elementarnym AG **genotyp** reprezentujemy za pomocą ciągów bitów.
Poniżej znajduje się przykładowy genotyp:



Rysunek 1: Przykładowy genotyp o długości 10.

Transformacja naszego problemu do opisu w postaci ciągów binarnych jest często bardzo trudna, a czasami wręcz niemożliwa. Lecz trud ten jest opłacalny, ponieważ dla tak opisanych genotypów mamy zdefiniowane operatory genetyczne, które wykonując proste obliczenia zbliżają nasze osobniki do optymalnego rozwiązania.

2.1.2 Selekcja

Selekcja jest to etap algorytmu w którym wybieramy poszczególne genotypy które później zostaną poddane operatorom genetycznym. Celem tej operacji jest wybranie najlepiej przystosowanych osobników do utworzenia nowej populacji. Aby opisać ten proces musimy zdefiniować przedstawioną w [Alg. 1] funkcję oceny f.
Funkcja oceny/przystosowania f jest obliczana dla każdego osobnika i pozwala nam porównać które genotypy są najlepiej przystosowane do zadania które optymalizujemy.
Najprostszym przykładem będzie zadanie znalezienia maximum funkcji $f(x_1, ..., x_n)$ gdzie $n \in \mathbb{N}^+$, wtedy funkcja przystosowania będzie równa wartości funkcji f, im większa wartość f tym osobnik jest lepiej przystosowany.

Przy pomocy funkcji oceny i różnych metod selekcji wybierana jest pula "najlepszych osobników". Istnieje wiele metod selekcji lecz zdecydowanie najpopularniejszą jest **metoda ruletki** w której prawdopodobieństwo p_i wybrania i-tego genotypu do reprodukcji kolejnego pokolenia jest proporcjonalne do wartości funkcji przystosowania. Prawdopodobieństwo to jest równe [5]:

$$p_i = \frac{f_i}{\sum_{j=1}^{N} f_j} \tag{1}$$

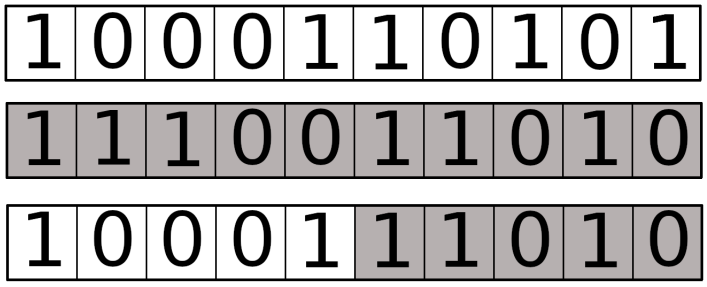
gdzie:
f_i - wartość funkcji oceny dla i-tego genotypu, N - rozmiar populacji.

2.1.3 Krzyżowanie

Jedną z dwóch podstawowych operacji wykonywanych w celu stworzenia potomstwa obecnej populacji jest **krzyżowanie**, które inspiruję się rozmnażaniem płciowym w biologi [12].

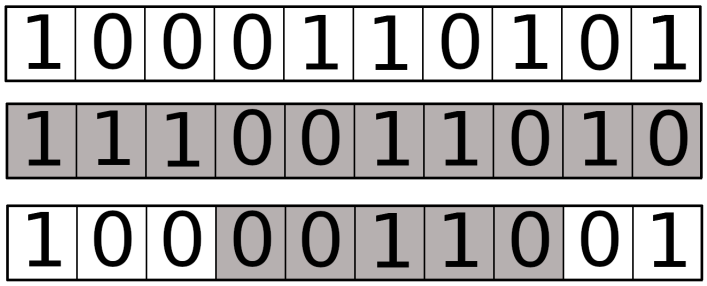
W literaturze możemy znaleźć wiele rodzajów tej operacji, poniżej znajdują się najpopularniejsze odmiany [5]:

- **Krzyżowanie jednopunktowe**:
  W tej odmianie potomka tworzymy z dwóch wybranych przez selekcję rodziców a następnie losujemy liczbę naturalną l ze zbioru $\{1, \ldots, n_x\}$. Potomek na pierwszych l pozycjach przyjmuje wartości od pierwszego rodzica a na pozostałych od drugiego.



Rysunek 2: Krzyżowanie jednopunktowe, gdzie $n_x = 10$ i $l = 5$.

- **Krzyżowanie dwupunktowe**:
  Sposób ten jest zbliżony do opisanego wyżej lecz zamiast jednego punktu losujemy dwie liczby naturalne l_1, l_2 ze zbioru $\{1, \ldots, n_x\}$, gdzie $l_1 < l_2$. Potomek natomiast przyjmuje wartości z pierwszego rodzica poza elementami na pozycjach od l_1 do l_2 gdzie wstawiamy elementy z drugiego.



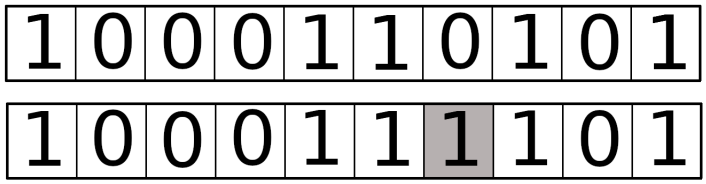
Rysunek 3: Krzyżowanie dwupunktowe, gdzie $n_x = 10$, $l_1 = 4$ i $l_2 = 8$.

  Krzyżowania tego rodzaju można uogólnić na operacje gdzie losujemy k-punktów a potomek naprzemiennie pobiera wartości z rodziców.

- **Krzyżowanie równomierne**:
  Potomek jest również tworzony przy pomocy dwóch rodziców gdzie każdy jego element jest wybierany losowo z pierwszego lub drugiego rodzica z jednakowym prawdopodobieństwem

2.1.4 Mutacja

Operator ten w klasycznej wersji algorytmu polega na zamianie zadanej wartości bitu (0 na 1 lub 1 na 0) z bardzo małym prawdopodobieństwem σ_m(np. 0.02%) [5].



Rysunek 4: Podstawowa mutacja dla genotypu o $n_x = 10$ i wylosowanej pozycji 7.

2.2 Sztuczne Sieci Neuronowe

Podobnie jak dla Algorytmów Genetycznych, inspiracją dla Sztucznych Sieci Neuronowych (SSN) była biologia a dokładniej budowa neuronów naturalnych znajdujących się w ciele człowieka. Sztuczne sieci neuronowe zostały wymyślone już w w pierwszej połowie XX wieku, lecz dopiero w ostatnich latach dzięki rozwojowi technologi oraz dostępności do olbrzymiej ilości danych mogliśmy zacząć używać ich pełny potencjał. SSN są przykładem techniki **uczenia nadzorowanego** tzn. podczas uczenia sieci podajemy jej dane ze znanymi wynikami i poprzez porównywanie wygenerowanych wyników z rzeczywistymi nasza sieć uczy się generować poprawne wyniki.

2.2.1 Perceptron

Perceptron to najprostsza architektura SSN. Jednostki na wyjściach/wejściach nazywane są **neuronami**. Na neuronach wejściowych podawane są liczby oraz każde połączenie ma przypisaną wagę natomiast j-te wyjście perceptronu jest obliczane przez wyliczenie ważonej sumy sygnałów wejściowych [4]:

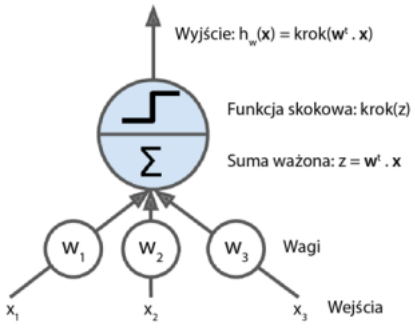
$$z_j = \sum_{i=1}^{n} w_{ij} x_i = \mathbf{w}_j^T \mathbf{x} \tag{2}$$

gdzie:
w_{ij} - waga połączenia pomiędzy i-tym neuronem wejściowym a j-tym wyjściowym,
x_i - i-ta wartość wejściowa.

Następnie wynik otrzymany w (2) jest poddawany **funkcji skoku**, gdzie najczęściej jest ona równa **funkcji Hraviside'a** lub **signum** które są równe:

$$\text{Heaviside}(z) = \begin{cases} 0, & z < 0 \\ 1, & z \geqslant 0 \end{cases} \qquad \text{sgn}(z) = \begin{cases} -1, & z < 0 \\ 0, & z = 0 \\ 1, & z > 0 \end{cases}$$

Nazywane są one **funkcjami aktywacji** (patrz [2.2.3]) i jest to jeden z wielu hiperametrów SSN które samodzielnie musimy wybrać podczas tworzenia sieci.



Rysunek 5: Schemat perceptronu z jednym neuronem wyjściowym i trzema wejściowymi [4]

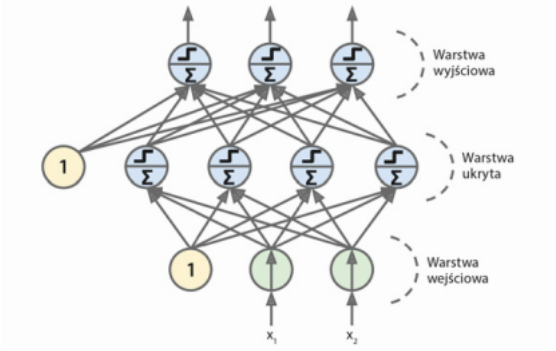
Lecz aby na wyjściu otrzymywać oczekiwane wyniki musimy najpierw perceptron wytrenować. Proces ten polega na modyfikacji wag na podstawie porównania wyniku oczekiwanego z wynikiem otrzymanym. Wagi są aktualizowane według wzoru [4]:

$$w_{ij} = w_{ij} + \eta(y_j - \hat{y_j})x_i \tag{3}$$

gdzie:
$\hat{y_j}$ - otrzymany wynik na j-tym wyjściu,
y_j - docelowy wynik j-tego wyjścia,
$\eta \in (0, 1]$ - współczynnik uczenia.
Perceptron jednak nie nadaje się do skomplikowanych zadań ponieważ przez swoją prostą budowę jest jedynie zdolny do klasyfikowania danych które są liniowo separowalne.

2.2.2 Wielowarstwowa Sieć Neuronowa

Ograniczenia perceptronu można wyeliminować tworząc SSN z wielu warstw perceptronów. SSN tego typu nazywamy **perceptronem wielowarstwowym**, który jest złożony z **warstwy wejściowej**, co najmniej jednej **warstwy ukrytej** (jako wejście przyjmują one wyjście poprzedniej warstwy a ich wyjście propagowane do kolejnej jako wejście) ostatnią warstwę nazywamy **warstwą wyjściową**. Dodatkowo w każdej warstwie znajduje się **neuron obciążający**, którego zadaniem jest wysyłanie na wejście następnej warstwy wartości 1. Ilość neuronów w warstwie wejściowej i wyjściowej jest określana przez zestaw danych na których sieć ma pracować np. jeśli zadaniem sieci jest rozpoznawanie ręcznie pisanych cyfr to na wyjściu powinno znaleźć się 10 neuronów a na wejściu każdy neuron powinien odpowiadać jednemu pikselowi wczytanego obrazu. Natomiast ilość warstw ukrytych jak odpowiadająca dla każdej warstwy liczba neuronów z których SSN która zawiera co najmniej dwie warstwy ukryte nazywamy **głęboką siecią neuronową** (GSN) [4].



Rysunek 6: Perceptron wielowarstwowy z jedną warstwą ukrytą [4].

Zaproponowana w wzorze (3) metoda uczenia perceptronu nie zadziała w przypadku sieci wielowarstwowej. Najpopularniejszą metodą uczenia sieci neuronowych jest **wsteczna propagacja**. Cały proces możemy przedstawić za pomocą kroków [4]:

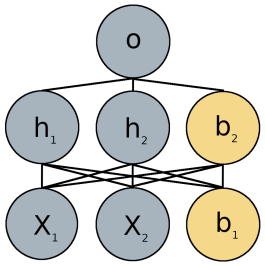
Algorithm 2: Procedura uczenia wielowarstwowej SSN.

1. Wybór parametrów sieci (liczba warstw, liczba neuronów w warstwach itd.)
2. Wagi inicjujemy losowo.
3. Dla każdego neuronu wyjściowego obliczany jest błąd równy różnicy pomiędzy otrzymanym wynikiem $\hat{\mathbf{y}}$ a wartością oczekiwaną $\mathbf{y}$.
4. Błędy propagowane są do poprzednich warstw.
5. Modyfikacja wag na podstawie wartości błędu.
6. Powtarzaj od **3** dla kolejnych wektorów uczących.
7. Skończ algorytm jeśli przekroczymy ustaloną liczbę epok γ_t, lub średni błąd przestanie zauważalnie maleć.

Średni błąd może być przestawiony przez wzór:

$$d = \frac{1}{2}(\mathbf{y} - \hat{\mathbf{y}})^2 \tag{4}$$

Podczas aktualizowania wag wybieramy te wartości dla których średni błąd jest najmniejszy. Propagację błędów do poprzednich warstw najlepiej pokazać na przykładzie:



Rysunek 7: Schemat przykładowej SSN z jedną warstwą ukrytą oraz jednym neuronem w warstwie wyjściowej.

Dla powyższej sieci wagi łączące warstwę wejściową z ukrytą oznaczajmy przez $w_{ij}^{(1)}$, natomiast wagi łączące warstwę ukrytą z wyjściową przez $w_{ij}^{(2)}$. Zakładamy że powyższa sieć jako funkcję

aktywacyjną używa funkcji logistycznej σ [2.2.3]. Wtedy postępujemy według kroków:

1. Obliczamy wartości neuronów w warstwie ukrytej:
$$h_i(w_{i1}^{(1)}, w_{i2}^{(1)}, x_1, x_2, b_1) = \sum_j (w_{ij}^{(1)} x_j) + b_1 \tag{5}$$
gdzie: h_i - wartość i-tego neuronu dla sieci z rysunku. [Rys. 7] $i, j = 1, 2$.

2. Każdy neuron w warstwie ukrytej poddawany jest funkcji aktywacyjnej:
$$\hat{h_i}(w_{i1}^{(1)}, w_{i2}^{(1)}, x_1, x_2, b_1) = \sigma(h_i) = \frac{1}{1 + e^{-h_i}} \tag{6}$$

3. Ponieważ na [Rys. 7] mamy jedno wyjście obliczamy je przy pomocy wzoru:
$$\begin{aligned} o'(w_{11}^{(2)}, w_{21}^{(2)}, \hat{h_1}, \hat{h_2}, b_2) &= \sum_i (w_{i1}^{(2)} \hat{h_i}) + b_2 \\ \hat{o'}(w_{11}^{(2)}, w_{21}^{(2)}, \hat{h_1}, \hat{h_2}, b_2) &= \sigma(o') \end{aligned} \tag{7}$$

4. Kolejnym krokiem jest obliczenie błędu d_o według wzoru (4):
$$d_o(w_{11}^{(2)}, w_{21}^{(2)}, \hat{h_1}, \hat{h_2}, b_2) = \frac{1}{2}(o - \hat{o'})^2 \tag{8}$$
gdzie: o - oczekiwana wartość na wyjściu.
Musimy również pamiętać, że wartości h_1, h_2, x_1, x_2 są zależne od wag $w_{ij}^{(1)}$ więc mają one również wpływ na wartości o' jak i d_o.

5. Teraz rozpoczyna się etap propagacji wstecznej błędu, najpierw propagujemy błąd do wag łączących warstwę ukrytą z warstwą wyjściową, wagi w sieci aktualizujemy przy pomocy wzoru:
$$w_{i1}^{(2)} = w_{i1}^{(2)} - \eta \frac{\partial d_o}{\partial w_{i1}^{(2)}} \tag{9}$$
Korzystając z reguły łańcuchowej wiemy że dla wag łączących warstwę ukrytą z warstwą wyjściową powyższa pochodna przyjmuje postać:
$$\frac{\partial d_o}{\partial w_{i1}^{(2)}} = \frac{\partial d_o}{\partial \hat{o'}} \frac{\partial \hat{o'}}{\partial o'} \frac{\partial o'}{\partial w_{i1}^{(2)}} \tag{10}$$
Ponieważ wiemy, że powyższe pochodne są równe [4]:
$$\begin{aligned} \frac{\partial d_o}{\partial \hat{o'}} &= (\hat{o'} - o) \\ \frac{\partial \hat{o'}}{\partial o'} &= \hat{o'}(1 - \hat{o'}) \\ \frac{\partial o'}{\partial w_{i1}^{(2)}} &= \hat{h_i} \end{aligned} \tag{11}$$
Regułę modyfikacji wag podanych w wzorze (8) możemy uprościć do postaci w której wszystkie poprzednie wartości wcześniej policzyliśmy:
$$w_{i1}^{(2)} = w_{i1}^{(2)} - \eta(\hat{o'} - o)\hat{o'}(1 - \hat{o'})\hat{h_i} \tag{12}$$

6. Dla wag łączących warstwę wejściową z ukrytą pochodna $\frac{\partial d_o}{\partial w_{ij}^{(1)}}$, korzystając z reguły łańcuchowej otrzymujemy:

$$\frac{\partial d_o}{\partial w_{ij}^{(1)}} = \frac{\partial d_o}{\partial \hat{o'}} \frac{\partial \hat{o'}}{\partial o'} \frac{\partial o'}{\partial \hat{h_j}} \frac{\partial \hat{h_j}}{\partial h_j} \frac{\partial h_j}{\partial w_{ij}^{(1)}} \tag{13}$$

Pierwsze dwie pochodne policzyliśmy wcześniej pozostałe trzy są równe:

$$\begin{aligned} \frac{\partial o'}{\partial \hat{h_j}} &= w_{j1}^{(2)} \\ \frac{\partial \hat{h_j}}{\partial h_j} &= \hat{h_j}(1 - \hat{h_j}) \\ \frac{\partial h_j}{\partial w_{ij}^{(1)}} &= x_i \end{aligned} \tag{14}$$

Ostateczny wzór na modyfikację wag $w_{ij}^{(1)}$ jest więc równy:

$$w_{ij}^{(1)} = w_{ij}^{(1)} - \eta(\hat{o'} - o)\hat{o'}(1 - \hat{o'})w_{j1}^{(2)}\hat{h_j}(1 - \hat{h_j})x_i \tag{15}$$

Dla sieci z większą liczbą warstw ukrytych wzory modyfikacji wag dzięki regule łańcuchowej rozrastałyby się o kolejne człony w sposób analogiczny jak w powyższym przykładzie.

2.2.3 Funkcje aktywacyjne

Aby powyższy algorytm przebiegał prawidłowo zamiast funkcji skokowej/signum wprowadzono inne funkcje aktywacji, ponieważ funkcje używane w przypadku pojedynczego perceptronu są złożone jedynie z płaskich segmentów, co uniemożliwia korzystanie z gradientu. Poniżej znajdują się najczęściej używane **funkcje aktywacji** w SSN:

- **funkcja logistyczna**:
  W każdym punkcie ma zdefiniowaną pochodną niezerową, dzięki czemu algorytm gradientu prostego może na każdym etapie uzyskiwać lepsze wyniki. Jest ona używana w warstwie wyjściowej jeśli zadaniem naszej sieci jest klasyfikacja obiektów na przynależność do dwóch rozłącznych klas. Dodatkowo jest to funkcja używana przez neurony naturalne przez co uważana była za najlepszą funkcję aktywacyjną, lecz jak pokazała praktyka w przypadku SSN inne funkcje sprawują się lepiej. Przyjmuje ona wartości z zakresu $(0, 1)$ [4].

$$\sigma(z) = \frac{1}{1 + e^{-z}} \tag{16}$$

- **funkcja tangensa hiperbolicznego**:
  Jest S - kształtna, ciągła i różniczkowalna, podobnie jak funkcja logistyczna, ale zakres wartości wynosi $(-1, 1)$ a nie $(0, 1)$ jak w przypadku funkcji σ [4].

$$\tanh(z) = \frac{e^z - e^{-z}}{e^z + e^{-z}} \tag{17}$$

- **funkcja ReLU**:
  Jest ciągłą, ale nieróżniczkowalna dla $z = 0$. W praktyce jednak spisuje się znakomicie dodatkowo jest bardzo szybko obliczana. Aktualnie jest ona jak i jej odmiany najczęściej używanymi funkcjami aktywacji dla warstw ukrytych i wejściowej [4].

$$ReLU(z) = \max(0, z) \tag{18}$$

- **funkcja softmax**:
  Używana jest ona w warstwie wyjściowej jeśli nasza sieć ma obliczać prawdopodobieństwa przynależności otrzymywanych obiektów do klas (jest ich więcej niż 2 i wszystkie prawdopodobieństwa mają się sumować do 1). Model najpierw oblicza dla każdej klasy - k [4]:

$$s_k(\mathbf{x}) = \left(\mathbf{w}^{(k)}\right)^T \mathbf{x} \tag{19}$$

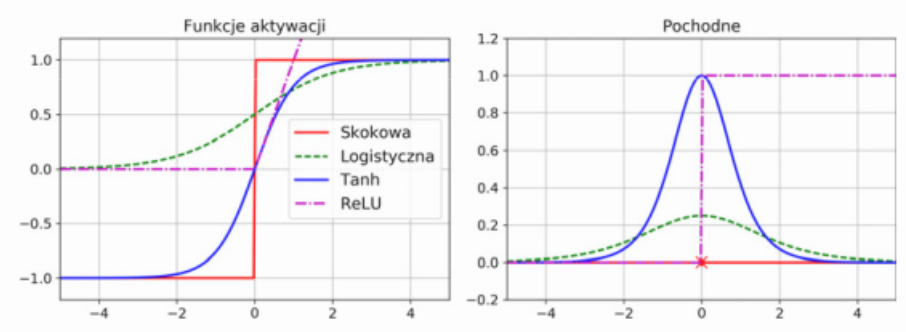
  gdzie: $\mathbf{w}^{(k)}$ - wyspecjalizowany wektor wag dla klasy k.
  Po wyliczeniu $s_k(\mathbf{x})$ dla każdej klasy, obliczane jest odpowiednio znormalizowane prawdopodobieństwo przynależności danej próbki do klasy k [4]:

$$p_k = \frac{\exp\left(s_k(\mathbf{x})\right)}{\sum_{i=1}^{K} \exp\left(s_i(\mathbf{x})\right)} \tag{20}$$

  gdzie: K - liczba klas.
  Model prognozuje klasę o najwyższym prawdopodobieństwie [4]:

$$\hat{y} = \underset{k}{\operatorname{argmax}}(p_k) \tag{21}$$

  Gdzie funkcja **argmax** zwraca indeks klasy dla której wartość p_k jest największa.

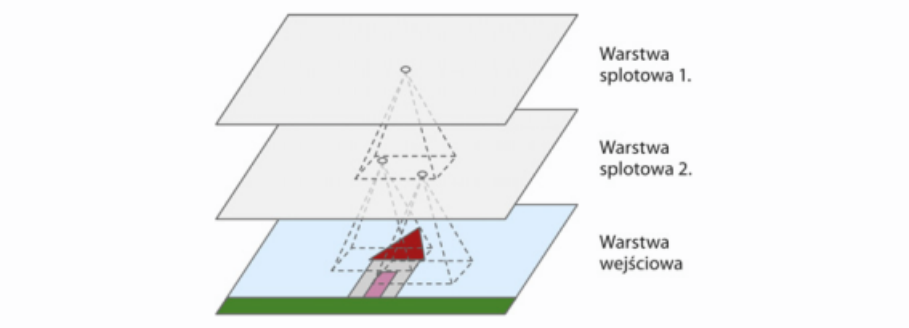


Rysunek 8: Funkcje aktywacyjne i ich pochodne. [4]

2.2.4 Inne rodzaje warstw SSN.

W SSN oprócz **warstw gęstych** tzn. gdzie każdy neuron w i-tej warstwie jest połączony z wszystkimi neuronami znajdującymi się w warstwie nr. $i+1$, używane są warstwy innego typu które pomagają siecią osiągać lepsze wyniki. Rodzaj warstwy też można rozważać jako kolejny hiperparametr SSN. Oto kilka przykładowych warstw:

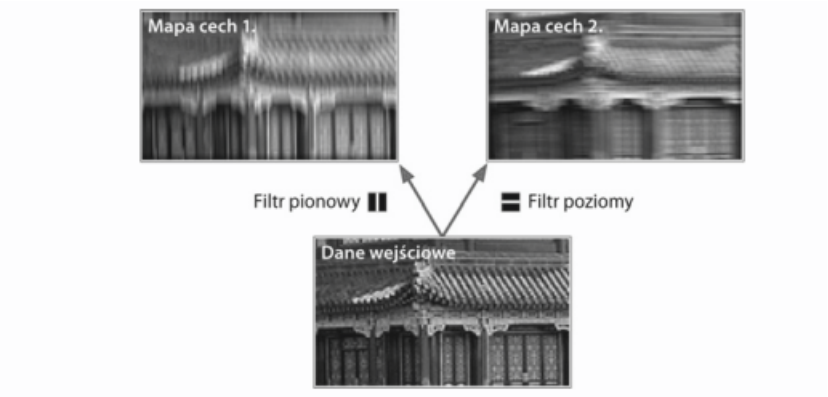
- **warstwa splotowa**: Używane są one w zadaniach wizualnych. Neurony w pierwszej warstwie splotowej nie są połączone z każdym pikselem obrazu wejściowego, lecz wyłącznie z pikselami znajdującymi się w ich polu recepcyjnym.



Rysunek 9: Warstwy CNN z prostokątnymi lokalnymi polami recepcyjnymi [4].

Dzięki powyższej strukturze sieć może koncentrować się na ogólnych cechach w pierwszej warstwie ukrytej, następnie łączyć je w bardziej złożone kształty w kolejnych warstwach. Ogólnie neuron znajdujący się w i-tym wierszu oraz j-tej kolumnie danej warstwy jest połączony z wyjściami neuronów poprzedniej warstwy zlokalizowanymi w rzędach od $i \cdot s_h$ do $i \cdot s_h + f_h - 1$ i kolumnach od $j \cdot s_w$ do $j \cdot s_w + f_w - 1$ gdzie: f_w / f_h - szerokość/wysokość pola recepcyjnego, s_h, s_w definiują wartość **kroków** odpowiednio w kolumnach i rzędach. **Krok** oznacza odległość pomiędzy dwoma kolejnymi polami recepcyjnymi. Aby uzyskać takie same wymiary dla każdej warstwy najczęściej dodawane są zera wokół wejść. Krok i rozmiar pola recepcyjnego musi podawać użytkownik tworzący warstwę więc są one kolejnymi hiperparametrami sieci.

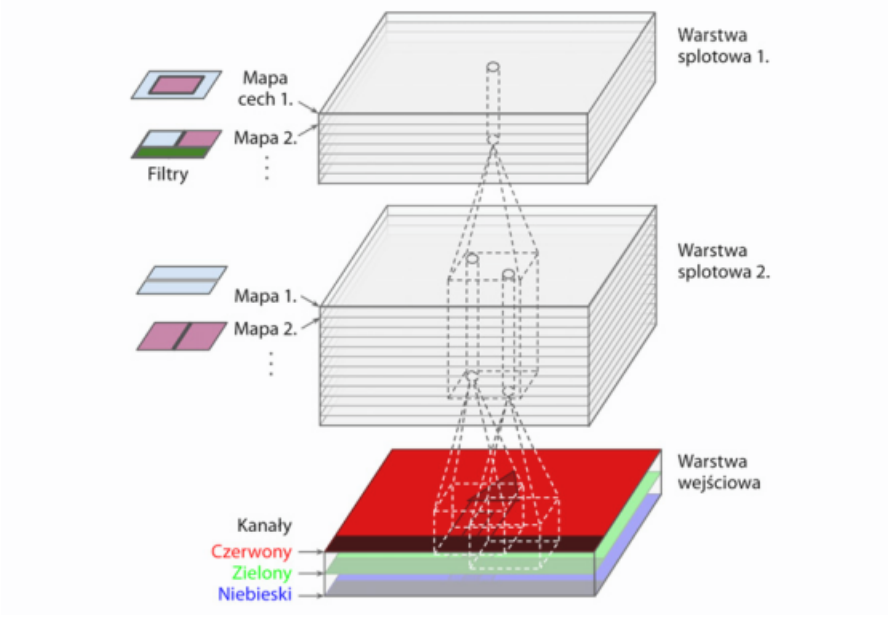
Wagi neuronu mogą być przedstawione jako niewielki obraz o rozmiarze pola recepcyjnego, hiparapametr ten nazywamy **filtrem**. Przykładowo filtrem może być macierz wypełniona zerami oprócz środkowej kolumny zawierającej jedynki, neurony posiadające taki filtr będą ignorować wszystkie elementy oprócz tych które znajdują się w środkowej linii.



Rysunek 10: Dwie mapy cech otrzymane przy pomocy dwóch różnych filtrów [4].

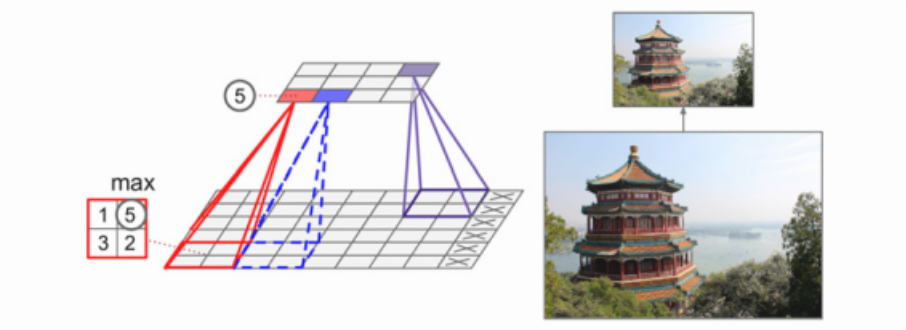
Warstwa wypełniona neuronami o takim samym filtrze nazywamy **mapą cech** która pomaga nam wyszczególnić elementy przypominające użyty filtr. Warstwa splotowa składa się z kilku map cech o identycznych rozmiarach. Każda mapa ma swoje wartości parametrów, dzięki czemu stosując różne filtry warstwa splotowa jest w stanie wykryć wiele cech w dowolnym

obszarze obrazu. Dodatkowo obrazy wejściowe składają się również z kilu warstw, po jednej na każdy kanał barw. Zazwyczaj są to trzy kanały - czerwony, zielony i niebieski lub jeden dla obrazów czarno-białych [4].



Rysunek 11: Warstwa splotowa zawierająca wiele cech, oraz zdjęcie z trzema kanałami barw [4].

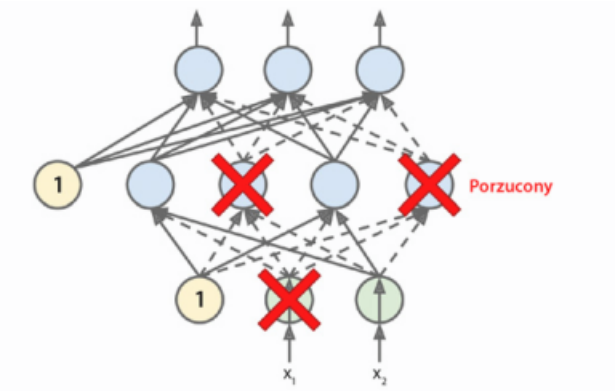
- **warstwa łącząca**: Jej celem jest zmniejszenie obrazu wejściowego w celu zredukowania obciążenia obliczeniowego, wykorzystania pamięci i liczby parametrów. Tak samo jak w warstwach splotowych neurony łączą się z wyjściami określonej liczby neuronów poprzedniej warstwy, które mieszczą się w obszarze pola recepcyjnego. Jednakże warstwa ta nie zawiera wag, jej zadaniem jest gromadzenie danych przy pomocy funkcji agregacyjnej np. maksymalizującej lub uśredniającej [4].



Rysunek 12: Maksymalizująca warstwa łącząca, gdzie rozmiar jądra łączącego to 2x2 [4].

- **warstwa porzucania**:
  Warstwa ta dla poprzedniej warstwy aplikuję technikę **porzucania** tzn. że każdy neuron znajdujący się w tej warstwie w poszczególnym przebiegu może zostać całkowicie pominięty

w procesie uczenia. Szansa na porzucenie jest hiperparametrem i nazywana jest **współczynnikiem porzucenia** [4].



Rysunek 13: Przykład porzucania [4].

Istnieje wiele innych rodzajów warstw w SSN (np. rekurencyjne) lecz opisałem tyko warstwy użyte w moim programie. Z tego samego powodu jedyną opisaną przeze mnie architekturą jest **jednokierunkowa sieć neuronowa**(sygnał biegnie w jednym kierunku).

3 Mikro - GA dla SSN

3.1 Modyfikacja AG

SSN posiadają dużą liczbę hiperparametrów i wpływ każdego z nich na efektywność sieci jest zależny od wykonywanego zadania, więc jeżeli wybieramy je ręcznie musimy to robić metodą prób i błędów. Aby zautomatyzować ten proces wykorzystam Algorytm Genetyczny, który przeglądając przestrzeń rozwiązań, do której należą sieci o różnej liczbie warstw z różnymi funkcjami aktywacyjnymi itp. wybierze możliwie najlepszą według zadanej metryki. Z powodu dużej ilości hiperparamtetrów w SSN musiałem wybrać ich ograniczoną ilość (wybrane hiperparametry opisane są w podrozdziale [2.2]), ich lista przedstawiona jest na [Rys. 14]. Modyfikując AG musimy pamiętać, że będziemy operować na sieciach o różnych rozmiarach, lecz użytkownik musi wybrać ich maksymalną liczbę. Musi on też wybrać typ wykonywanego zadania (klasyfikacja lub regresja), ponieważ do każdego typu przypisywana jest inna metryka oceny jakości sieci (patrz [3.4]). Implementując operację krzyżowania i mutacji potrzebne jest zapewnienie aby tworzyły one prawidłowe sieci (wymagana kolejność warstw, liczba warstw nieprzekraczająca maksymalnej, patrz [3.2]).

Aby użyć AG to tego zadania musimy zmodyfikować jego tradycyjną odmianę. Poniżej znajduje się lista wszystkich parametrów zmodyfikowanego algorytmu oraz użyte dla nich oznaczenia, ich rola natomiast zostanie omówiona w kolejnych podrozdziałach.

Nazwa parametru	Oznaczenie
Współczynnik sprawdzianu krzyżowego	γ_v
Prawdopodobieństwo mutacji	ρ_m
Prawdopodobieństwo dodania warstwy	γ_l
Maksymalna liczba warstw	$\mathcal{A}$
Współczynnik skalowania rozmiaru sieci (25)	α
Rozmiar populacji	n
Rozmiar turnieju [3.5]	n_t
Dopuszczalna liczba podobnych modeli [3.8]	γ_c
Liczba epok treningowych [Alg. 2]	γ_t
Liczba generacji	γ_g
Liczba eksperymentów	γ_r

Tabela 2: Parametry AG dla SNN [3].
γ_v - oznacza procent danych jaki zostanie przeznaczony do zbioru walidacyjnego tzn. pozwali nam ocenić jakość wygenerowanej sieci [3.4].

Idea działania algorytmu znajdującego optymalną sieć dla zadanego problemu jest następująca: na wejściu mamy dwa zbiory danych: treningowy i walidacyjny. Liczba osobników w zbiorze treningowym i walidacyjnym jest ustalana na podstawie wartości parametru γ_v. γ_r razy uruchamiamy algorytm genetyczny. Dla każdego uruchomienia algorytmu najlepszy osobnik zostaje wstawiony do zbioru "najlepszych rozwiązań." Po zakończeniu działania algorytmu wybrany zostaje najlepszy z osobników znalezionych w kolejnych uruchomieniach algorytmu genetycznego. Sam algorytm genetyczny po wygenerowaniu n-elementowej populacji początkowej złożonej z osobników posiadających co najwyżej $\mathcal{A}$ warstw wykonuje w pętli nie więcej niż γ_g kroków. W każdym kroku obliczana jest wartości funkcji dla każdego osobnika poprzez wytrenowanie odpowiadającej mu sieci na zbiorze treningowym w γ_t epokach treningowych i obliczenie jakości sieci na zbiorze walidacyjnym na podstawie odpowiednich metryk. Następnie poprzez zastosowanie selekcji turniejowej, krzyżowania i mutacji, tworzona jest nowa populacja. Poszczególne kroki algorytmu zostały szczegółowo opisane w dalszej części rozdziału.

Ponieważ zależy nam na szybkości wykonywanych obliczeń zastosowana została odmiana **mikro-GA** gdzie liczba dozwolonych generacji jest niewielka. Dodatkowo aby przyśpieszyć działanie al-

gorytmu SSN są trenowane przez ograniczoną liczbę epok tzw. **uczenie częściowe**.

Algorithm 3: Zmodyfikowany AG wybierający hiperparametry SSN [3].

Data: Treningowy i walidacyjny zbiór danych $\mathcal{D}_t$, $\mathcal{D}_v$
Input: Parametry algorytmu.
Output: Najlepiej przystosowana SSN ϕ^*
Niech $t_e = 0$ będzie licznikiem wykonanych przebiegów AG.
Niech $\mathcal{S}$ zbiór najlepszych rozwiązań.
while $t_e < \gamma_r$ **do**
  Niech $t = 0$ będzie licznikiem generacji.
  Stwórz i losowo zainicjalizuj początkową populację $\mathcal{C}(0)$, zawierającą n osobników, gdzie $n \leqslant 10$.
  while $t < \gamma_g \vee$ *warunek wczesnego końca dla* $\mathcal{C}(t)$ *nie jest spełniony* **do**
    Oblicz wartość funkcji przystosowania $c(\phi)$ [3.4] dla każdego osobnika w populacji.
    Wykonaj selekcję tworząc populację potomstwa $\mathcal{O}(t)$.
    Wykonaj operację krzyżowania i mutacji z prawdopodobieństwem ρ_m na osobnikach z $\mathcal{O}(t)$.
    $\mathcal{C}(t+1) = \mathcal{O}(t)$.
    $t = t + 1$.
  end
  Do $\mathcal{S}$ dodaj najlepszego osobnika z poprzedniego przebiegu.
  $t_e = t_e + 1$.
end
Znormalizuj koszt dla każdego modelu w $\mathcal{S}$.
Najlepszym modelem jest osobnik z $\mathcal{S}$ dla którego funkcja przystosowania przyjmuje najniższą wartość.

3.2 Reprezentacja SSN w AG

Z powodu złożoności SSN musiałem zrezygnować ze standardowej reprezentacji osobników jako ciągi binarne. Użytym przeze mnie modelem jest kodowanie oparte na liście tablic tzn. każda warstwa sieci jest reprezentowana jako tablica w której każde pole ma z góry określone znaczenie. W tej reprezentacji warstwy przyjmują postać [3]:

Typ Warstwy	Liczba Neuronów	Funkcja Aktywacji	Liczba Filtrów	Rozmiar Pola Recepcyjnego	Krok Pola Recepcyjnego	Rozmiar Pola Łączącego	Współczynnik Porzucenia

Rysunek 14: Schemat reprezentacji warstw w genotypie reprezentującym SSN.

Dla poszczególnych typów sieci używane są inne pola:

Nr.	Rodzaje SSN	Rodzaje Warstw
1	Perceptron wielowarstwowy/Sieć splotowa	Wszystkie warstwy
2	Perceptron wielowarstwowy	Gęsta
3	Perceptron wielowarstwowy/Sieć splotowa	Gęsta i Splotowa
4	Sieć splotowa	Splotowa
5	Sieć splotowa	Splotowa
6	Sieć splotowa	Splotowa
7	Sieć splotowa	Łącząca
8	Perceptron wielowarstwowy/Sieć splotowa	Porzucania

Tabela 3: Hiperparametry SSN używane przez poszczególne rodzaje sieci i warstwy.

Osobniki więc będą listą takich tablic. Warstwa wejściowa i wyjściowa jest ustalona przez zadanie jakie ma wykonywać sieć np. jeśli trenowana sieć ma rozpoznawać ręcznie pisane cyfry wiemy że w warstwie wyjściowej musi pojawić się 10 neuronów z funkcją aktywacyjną typu softmax. Generując jednak pozostałe warstwy musimy pamiętać że warstwy nie mogą być ułożone losowo (np. nie mogą występować dwie warstwy porzucania po sobie). Poniżej znajdują się odpowiednie mapowania typów warstw i użytych funkcji aktywacji na liczby całkowite oraz odpowiednie zasady łączenia warstw w generowanych osobnikach.

Typ warstwy	Nazwa	Dozwoleni następcy
1	Warstwa Gęsta	1, 4
2	Splotowa	1, 2, 3, 4
3	Łącząca	1, 2
4	Porzucenia	1, 2

Tabela 4: Mapowanie typów warstw oraz zasady łączenia warstw.

Typ funkcji	Nazwa
1	Sigmoid
2	Tangens hiperboliczny
3	ReLU
4	Softmax
5	Liniowa

Tabela 5: Mapowanie funkcji aktywacyjnych.

Rodzaje filtrów są generowane przez bibliotekę `tensorflow` w której implementuję powyższy algorytm (patrz rozdział [4]). Reszta hiperparametrów przyjmuje wartości całkowite lub rzeczywiste z pewnego zadanego przedziału wyspecjalizowanego do danych dla których szukamy optymalnej sieci [Tab. 10].
Dodanie innych typów warstw (np. rekurencyjnych) i funkcji nie powinno być problemem, jedynie trzeba pamiętać o dodaniu odpowiednich zasad budowania dla nich. Przykładowy genotypem reprezentującym SSN jest:

$$G = \big[[1, 784, 3, 0, 0, 0, 0, 0], [4, 0, 0, 0, 0, 0, 0, 0.54], \\ [1, 300, 3, 0, 0, 0, 0, 0], [4, 0, 0, 0, 0, 0, 0, 0.28], \\ [1, 100, 3, 0, 0, 0, 0, 0], [1, 10, 4, 0, 0, 0, 0, 0]\big]$$

Dekodując powyższy genotyp otrzymujemy sieć złożoną z następujących warstw:

Typ warstwy	Liczba Neuronów	Funkcja Aktywacji	Wsp. Porzucenia
Gęsta	784	ReLU	N/A
Porzucanie	N/A	N/A	0.54
Gęsta	300	ReLU	N/A
Porzucanie	N/A	N/A	0.28
Gęsta	100	ReLU	N/A
Gęsta	10	Softmax	N/A

Tabela 6: Genotyp G przedstawiony w postaci tabeli.

3.3 Generowanie w populacji początkowej

Warstwa pierwsza wyznacza typ SSN(np. jeśli pierwszą warstwą jest warstwa gęsta to sieć będzie perceptronem wielowarstwowym, a jeśli pierwszą warstwą będzie warstwa splotowa sieć będzie siecią splotową). Warstwa wejściowa i wyjściowa dla każdego osobnika jest taka sama ponieważ są one zależne od danych na jakich trenujemy oraz od zadania jakie ma nasza sieć wykonywać (klasyfikacja lub regresja), dlatego w populacji początkowej musimy losowo generować tylko warstwy ukryte. Proces generowania możemy opisać przy pomocy pseudokodu [3]:

Algorithm 4: Generowanie populacji początkowej.

Output: Zainicjalizowana populacja początkowa $\mathcal{C}(0)$.
Niech $\mathcal{C}(0)$ oznacza populację początkową wielości n, gdzie genotypy tej populacji możemy oznaczyć przez $\mathbf{x}_i$.
for *Dla każdego i, gdzie* $1 \leqslant i \leqslant n$ **do**
  Dla wszystkich warstw poza wyjściową (która ma z góry określoną funkcję aktywacji) losujemy funkcję aktywacji f.
  while *Liczba warstw* $\mathbf{x_i} \leqslant \mathcal{A}$ *[Tab. 2]* **do**
    Wylosuj typ warstwy a następnie wartości używanych przez nią parametrów.
    Wylosuj liczbę U przy pomocy płaskiego rozkładu prawdopodobieństwa z przedziału $[0, 1]$.
    Oblicz $\rho_l = 1 - \sqrt{1 - U}$.
    if $\rho_l < \gamma_l$ *[Tab. 2]* **then**
      Jeśli wygenerowana warstwa jest kompatybilna z warstwą poprzednią dodaj ją do genotypu.
    else
      Kończymy dodawanie warstw dla $\mathbf{x}_i$.

3.4 Funkcja oceny dla SSN

Celem algorytmu jest znalezienie sieci ϕ^* osiągającej jak najlepszy wynik jednocześnie ograniczając liczbę trenowalnych parametrów, więc jest to problem optymalizacyjny przedstawiony za pomocą wzoru [3]:

$$\min_{\phi} \big(p(\phi), w(\phi)\big) \tag{22}$$

gdzie:
$p(\phi)$ - miara jakości sieci,
$w(\phi)$ - liczba trenowalnych parametrów zwana **rozmiarem sieci**.

Jakość sieci mierzę dzieląc zbiór danych na dwa podzbiory zbiór treningowy $\mathcal{D}_t$ i zbiór walidacyjny $\mathcal{D}_v$, procent danych przeznaczonych na walidację jest określony przez parametr γ_v [Tab. 2]. Za pomocą pierwszego podzbioru trenujemy SSN, natomiast przy pomocy drugiego sprawdzamy czy wygenerowane przez sieć wyniki dobrze oddają oczekiwane wartości. Metryki na podstawie których możemy ocenić jakość takich sieci są zależne od wykonywanego zadania. W moim przypadku dla zadania klasyfikacji użyłem **dokładności**, którą możemy opisać wzorem [4]:

$$p(\phi)_{kla} = \frac{liczba\ poprawnie\ zakwalifikowanych\ danych}{\mathcal{M}} \tag{23}$$

gdzie: $\mathcal{M}$ - to rozmiar zbioru danych.
Natomiast w przypadku regresji użyłem **błędu średniokwadratowego** (**MSE**) [4]:

$$p(\phi)_{reg} = \frac{1}{\mathcal{M}}(\mathbf{y} - \hat{\mathbf{y}})^T(\mathbf{y} - \hat{\mathbf{y}}) \tag{24}$$

gdzie:
$\hat{\mathbf{y}}$ - wyniki wygenerowane przez sieć,
$\mathbf{y}$ - oczekiwane wyniki.

Z tak zdefiniowanymi metrykami jeżeli chcielibyśmy znaleźć bezpośrednie rozwiązanie równania (22) prowadziłoby to do zadania optymalizacji wielokryterialnej. Aby uprościć to zadanie definiujemy funkcję oceny, względem której osobniki w populacji będą porównywane [3]:

$$c(\phi) = (1 - \alpha)p(\phi) + \alpha w(\phi) \tag{25}$$

Jednak powinniśmy jeszcze przeskalować $p(\phi)$ i $w(\phi)$ aby ich wartości były podobnego rzędu. Niech $\mathbf{p} = \{p(\phi_1), ..., p(\phi_n)\}$ będzie wektorem ocen aktualnej populacji, wtedy po znormalizowaniu $\mathbf{p}^* = \frac{\mathbf{p}}{||\mathbf{p}||}$ zakres wartości ocen sieci będzie z przedziału $p^*(\phi_i) \in [0, 1]$ dla każdego i. Niech $\mathcal{A}$ oznacza maksymalną liczbę warstw w sieci, oraz $\mathcal{N}$ oznacza maksymalną liczbę neuronów w warstwie wtedy maksymalna wartość rozmiaru sieci jest równa $w(\phi) = \mathcal{N}^2\mathcal{A}$. Ponieważ chcemy aby sieci o podobnych rozmiarach miały podobną wartość $w(\phi)$, ostatnie trzy cyfry rozmiaru zamieniamy na zera otrzymując $w'(\phi)$. Następnie aby skalami zbliżyć się do wartości ocen wyznaczamy $w^*(\phi) = \log_{10} w'(\phi)$ (rozmiaru sieci nie normalizujemy podobnie jak wartości ocen ponieważ chcemy aby różnica w rozmiarze miała większe wpływ na różnice w funkcji ocen), co nam daje wartości rzędu $\log_{10}(\mathcal{N}^2\mathcal{A})$.
Wtedy odpowiednio przeskalowana funkcja oceny wyrażona jest wzorem [3]:

$$c(\phi) = \log_{10}(\mathcal{N}^2\mathcal{A})(1 - \alpha)p^*(\phi) + \alpha w^*(\phi) \tag{26}$$

3.5 Selekcja

Aby wygenerować n potomków potrzebujemy $2n$ rodziców wybranych z bieżącej populacji. Do tego celu wykorzystamy odmianę **binarnej selekcji turniejowej** [3]:

Algorithm 5: Binarna selekcja turniejowa dla SSN.

Input: Aktualna generacja $\mathcal{C}(t)$.
Output: Pula rodzicielska następnej generacji
Niech $\mathcal{P}$ - zbiór rodziców.
Wybierz losowo n_t rodziców gdzie $n_t < n$.
while *nie wybrano 2n rodziców* **do**
    Porównaj parami wybrane genotypy , lepiej przystosowany dodaj do $\mathcal{P}$. Pary tworzymy przez branie kolejnych dwójek z puli rodziców (np. pierwszy osobnik z drugim itd.).

W powyższej procedurze im większa wartość n_t tym większe jest prawdopodobieństwo wybrania najlepszych osobników jako rodzica.

3.6 Krzyżowanie

Ponieważ standardowe operatory genetyczne nie będą działać dla zmodyfikowanego kodowania genotypów, je również trzeba zmodyfikować. Wybranym krzyżowaniem jest zmodyfikowane krzyżowanie dwupunktowe. Wykonując krzyżowanie musimy pamiętać, że nie każda warstwa może występować po każdej innej. Przykładowa dla dwóch rodziców G_1, G_2 po wybraniu punktów (r_1, r_2) z G_1 i (r_3, r_4) z G_2 (gdzie r_i oznacza indeks warstwy), może się stać że warstwa r_3 nie może być zamieniona z warstwą r_1 lub r_4 z r_2 z powodu niekompatybilności z warstwą poprzednią [Tab. 4]. Oznaczając $len(G_i)$ jako liczbę warstw osobnika G_i operację krzyżowania możemy przedstawić przy pomocy poniższego pseudokodu: [3]

Algorithm 6: Krzyżowanie dwupunktowe dla SSN.

Input: Dwa genotypy G_1, G_2.
Output: Nowy potomek G_3.
Losowo wybierz dwa punkty z G_1 gdzie $r_1 \leqslant r_2$.
if $r_1 = r_2$ **then**
  $r_2 = \text{len}(G_1)$ - 1
Znajdź wszystkie pary punktów $(r_3, r_4)_i$ w S_2 kompatybilne z (r_1, r_2), gdzie $r_3 < r_4$ i $len(G_1) + (r_4 - r_3) - (r_2 - r_1) < \mathcal{A}$ (sprawdzamy czy nie przekraczamy maksymalnej liczby warstw).
Losowo wybierz jedną z par $(r_3, r_4)_i$.
W G_1 usuń warstwy $r_1, \ldots, r_2$ a następnie wstaw tam warstwy $r_3, \ldots, r_4$ wzięte z G_2. Nowy genotyp oznacz jako G_3.
W G_3 zmień w warstwach wziętych z G_2 funkcję aktywacji, na funkcję używaną w warstwach wziętych z G_1.

Operacja ta dla dwóch poniższych genotypów będzie przebiegać następująco:

$$G_1 = \big[[1, 784, 2, 0, 0, 0, 0, 0], [4, 0, 0, 0, 0, 0, 0, 0.54], [1, 300, 2, 0, 0, 0, 0, 0], [4, 0, 0, 0, 0, 0, 0, 0.28], [1, 100, 2, 0, 0, 0, 0, 0], [1, 10, 4, 0, 0, 0, 0, 0]\big]$$

$$G_2 = \big[[1, 64, 1, 0, 0, 0, 0, 0], [4, 0, 0, 0, 0, 0, 0, 0.22], [1, 333, 1, 0, 0, 0, 0, 0], [1, 420, 1, 0, 0, 0, 0, 0], [1, 77, 1, 0, 0, 0, 0, 0], [4, 0, 0, 0, 0, 0, 0, 0.44], [1, 10, 4, 0, 0, 0, 0, 0]\big]$$

Dla $r_1 = 1$ i $r_2 = 3$ zbiór wszystkich możliwych par $(r_3, r_4)_i$ dla G_1 i G_2 wynosi:

$$[(0, 2), (0, 4), (0, 5), (1, 2), (1, 4), (1, 5), (2, 4), (2, 5), (4, 5)]$$

Zakładając że wylosowaliśmy punkty $(2, 4)$ potomek G_3 po wykonanej operacji krzyżowania jest równy:

$$G_3 = \big[[1, 784, 2, 0, 0, 0, 0, 0], [1, 333, 2, 0, 0, 0, 0, 0], [1, 420, 2, 0, 0, 0, 0, 0], [1, 77, 2, 0, 0, 0, 0, 0], [1, 100, 2, 0, 0, 0, 0, 0], [1, 10, 4, 0, 0, 0, 0, 0]\big]$$

Jak widzimy funkcje aktywacji w warstwach pobranych z G_2 zostały zamienione na funkcję używaną w G_1.

3.7 Mutacja

Mimo że mutacja nie jest potrzebna w mikro-AG, została ona dodana aby jeszcze bardziej zróżnicować modele i potencjalnie zwiększyć szanse na otrzymanie lepszych rozwiązań. Zmodyfikowana operacja mutacji jest bardzo podobna do swojej klasycznej wersji. Każdy model z populacji potomków może podlec mutacji przy prawdopodobieństwie ρ_m, wtedy [3]:

- Losowo wybierz jedną z warstw, a następnie zmień jeden z jej używanych parametrów zgodnie z tabelą [Tab. 3].
- Zmień funkcję aktywacji, a następnie skoryguj ją w pozostałych warstwach oprócz ostatniej.
- Dodaj warstwę porzucenia jeśli może ona występować po zmienianej warstwie [Tab. 4].

3.8 Warunek końca

W Algorytmie Genetycznym może dojść do sytuacji w której zmiany w kolejnych pokoleniach są niezauważalne, wtedy bez konsekwencji można zakończyć aktualny przebieg algorytmu. Możemy to zaimplementować definiując odpowiedni warunek końca. Warunek ten można oprzeć na funkcji oceny wiemy jednak że sieci o podobnej strukturze będą miały zbliżone wartości dla tej funkcji. Dlatego warunek końca możemy oprzeć definiując **odległość** między genotypami mówiąca nam jak zbliżone są do siebie dwie SSN.
Niech $G^{(i)}$ - oznacza i-tą warstwę sieci S. Wtedy przy założeniu $\text{len}(G_2) \geqslant \text{len}(G_1)$ odległość $d(G_1, G_2)$ miedzy dwoma genotypy reprezentującymi SSN możemy obliczyć według algorytmu [3]:

Algorithm 7: Odległość między dwoma SSN.

Input: Dwa genotpy G_1, G_2.
Output: Odległość między G_1 a G_2.
Niech $d = 0$ - odległość między G_1 a G_2.
for *Dla każdego i, gdzie* $1 \leqslant i \leqslant len(G_1)$ **do**
  $d = d + ||G_2^{(i)} - G_1^{(i)}||$.
for *Dla każdego i, gdzie* $len(G_1) < i \leqslant len(G_2)$ **do**
  $d = d + ||G_2^{(i)}||$.

Dla powyższej definicji d jeśli $G_1 = G_2$ to $d(G_1, G_2) = 0$, wiec pozwala nam ona określić stopień podobieństwa dwóch osobników.
Aktualny przebieg AG kończymy więc gdy co najmniej γ_c par spełnia $d(S_1, S_2) \leqslant \varepsilon$.
Gdzie: ε to pewna zadana dokładność, a γ_c jest parametrem AG.

4 Implementacja

Do zrealizowania opisanego wyżej AG użyłem języka `python`[11] wraz z biblioteką implementującą wiele operacji numerycznych `numpy`[9], natomiast do budowania i trenowania sieci neuronowych wykorzystałem popularną bibliotekę `tensorflow`[8]. Aby przyśpieszyć trenowanie sieci wykorzystałem platformę `Google Colab`[6], gdzie korzystając z popularnej formy `jupyter notebooków`[7] mamy darmowy dostęp przez chmurę do karty graficznej przyśpieszającej trenowanie SSN, co znacznie skraca czas wykonania naczego algorytmu. Algorytm podzieliłem na cztery programy z których każdy wykonuje jedną z funkcji:

- Przechowywanie zasad budowania SSN oraz parametrów potrzebnych do wykonania algorytmu.

- Dla danego genotypu budowanie sieci przy pomocy biblioteki `tensorflow`, oraz zwrócenie oceny oraz wielkości sieci.
- Implementacja klasy reprezentującej pojedynczy genotyp, która ułatwia implementacje operacji związanych z AG.
- Implementacja klasy reprezentującej AG gdzie znajdują się funkcję wykonujące algorytmy opisane w pseudokodach w rozdziale trzecim.

Programy opisane są w poniższych podrozdziałach:

4.1 Reguły budowania SSN.

Pomocniczy plik `building_rules.py` zawiera typy wyliczeniowe takie jak np. `LayerType` zwiększające czytelność kodu, oraz słowniki które:

- opisują maksymalne wartości opisane w tabeli [Tab. 3].
- zawierają reguły budowania sieci podane w tabeli [Tab. 4].
- opisują funkcje aktywacyjne dostępne dla warstwy danego typu.

4.2 Budowanie SSN zgodnej z genotypem.

Program `building_models.py` zawiera funkcje które dla każdego genotypu budują sieci neuronowe przy pomocy biblioteki `tensorflow`, która od aktualizacji `2.0` umożliwia proste budowanie modeli przy pomocy modułu `keras`[2]. Rodzaje warstw opisane w rozdziale [2.2.4] mają tam swoje gotowe implementacje. Proces trenowania sieci również został uproszczony ponieważ sprowadza się jedynie do wywołania metody `fit`. Reguły budowania sieci [Tab. 4] zaimplementowane w wyżej opisanym programie dają nam gwarancję że zbudowane modele będą poprawne. Algorytm genetyczny posiada wszystkie potrzebne parametry do przeprowadzenia procesu trenowania i zwrócenia potrzebnych dla niego wartości tzn. oceny wydajności sieci oraz liczby trenowalnych parametrów.

4.3 Reprezentacja genotypu

W `nn_genome.py` znajduje się klasa opakowująca genotypom przy pomocy listy list. Język `python` posiada implementacje listy która idealnie nadaję się do AG dla SSN, ponieważ może ona zawierać wartości różnego typu więc może ona np. posiadać jednocześnie pole opisujące liczbę neuronów w warstwie, która jest typu całkowitoliczbowego (`int`), oraz pole opisujące współczynnik porzucenia co jest typu zmiennoprzecinkowego (`float`).

4.4 Algorytm Genetyczny

Głownym plikiem jest `nn_optimizer.py` w którym znajduje się największa liczba operacji, ponieważ przy pomocy klasy `NNOptimizer` implementujemy cały przebieg AG. Projektując interfejs tej klasy inspirowałem się biblioteką implementującą algorytmy uczenia maszynowego `scikit-learn` [1]. Proces poszukiwania optymalnej sieci neuronowej z poziomu użytkownika jest bardzo prosty, jedyne co musi zrobić to stworzyć instancję podanej wyżej klasy do której konstruktora musimy podać typ wykonywanego zadania (klasyfikacja lub regresja) oraz ewentualnie parametry opisane w tabeli [Tab. 2]. Nie jest to jednak obowiązkowe ponieważ posiada ona dla nich wartości domyślne. Następnie cały algorytm genetyczny oraz finalne wytrenowanie otrzymanej sieci z pełną liczbą epok odbywa się przez wywołanie metody `fit` która pobiera jako argumenty dane oraz oczekiwane

wartości. Poniżej przykładowe wywołanie programu:

```
nn_optimizer.NNOptimizer.fit(X, y)
```

gdzie `X, y` - zbiór danych podzielony na dane wejściowe i oczekiwane wartości.
Interfejs ten umożliwia rozwiązanie jakiegoś problemu z użyciem uczenia maszynowego nawet jeśli użytkownik nie posiada wiedzy na temat przebiegu algorytmu który dana klasa implementuje.

Aby przyśpieszyć wykonywanie AG możemy w każdej generacji budować i trenować modele przy pomocy programu `building_models.py` równolegle. Jednak trenowanie tylu sieci jednocześnie jest procesem potrzebujących wiele zasobów, więc można tę opcję wyłączyć podając wartość argumentu funkcji `train_parallel=False` wywołując funkcję `fit`. Równoległość zaimplementowałem przy pomocy wbudowanej biblioteki `python'a multiprocessig` i jej klasy `Pool`, która posiada funkcję `pool` do której jako argumenty mogę podać funkcję budującą i trenującą SSN oraz populację genotypów, wtedy dla każdego osobnika zostanie utworzony osobny proces i możliwe będzie równoległa ewaluacja osobników.

5 Otrzymane wyniki

Działanie algorytmu przetestowałem zarówno dla problemu klasyfikacji jak i regresji. Użyłem do tego dwa zbiory danych **Fashion Mnist** i **Boston housing**. Zbiory te są stosunkowo małe (fashion mnist zawiera 70 tyś zdjęć), abym mógł przetestować algorytm dla różnych wartości parametrów AG dla SSN. Oba zbiory otrzymałem stosując moduł `tensorflow.keras.datasets`, zbiór ten domyślnie dzieli zbiór danych na zbiór treningowy i testowy w proporcjach 90% do 10%. Zbiór treningowy wykorzystałem do trenowania i walidacji generowanych sieci podczas działania AG [3.4], natomiast zbiór testowy wykorzystuję aby ocenić jakość najlepszej sieci zwróconej przez AG.

5.1 Fashion MNIST

Jest to zbiór artykułów z sklepu Zalando. Każdy obraz jest rozmiaru 28x28 pikseli w odcieniach szarości [13]:

Rysunek 15: Przykładowe obrazki z Fashion Mnist [13].

Każdy obrazek ma przypisaną jedną z 10 klas zmapowanych od liczb całkowitych od 1-9.

Nr.	Klasa
0	T-shirt/top
1	Trouser
2	Pullover
3	Dress
4	Coat
5	Sandal
6	Shirt
7	Sneaker
8	Bag
9	Ankle boot

Tabela 7: Klasy zbioru Fashion Mnist [13].

Więc jest to zadanie klasyfikacji gdzie gdzie nasza sieć próbuje podanemu zdjęciu przypisać jedną z 10 klas.
Algorytm był testowany kilkukrotnie na tym zbiorze zmieniając wartości współczynnika skalowania α, (24) natomiast reszta parametrów AG była stała [Tab. 2]

Parametr	Wartość
γ_v	0.4
ρ_m	0.1
γ_l	0.5
$\mathcal{A}$	10
n	10
n_t	4
γ_c	3
γ_t	10
γ_g	10
γ_r	5

Tabela 8: Parametry AG dla Fashion Mnist.

Ograniczone maksymalnymi wartościami zostały również optymalizowane hiperparametry sieci [Tab. 3], aby podczas generowania początkowej populacji nie zostały stworzone sieci o za dużej wielkości. Zapobiega to również zjawisku **przetrenowania** tzn. z powodu dużej złożoności sieć dopasowuje się niemal idealnie do danych na których przeprowadza proces uczenia, przez co sieć na danych spoza puli treningowej nie potrafi poprawnie przewidzieć oczekiwanych wartości.

Nazwa	Zakres Wartości
Liczba neuronów	$8 * x$ dla $x \in \{1, ..., 32\}$
Liczba Filtrów	$8 * x$ dla $x \in \{1, ..., 16\}$
Rozmiar Pola Recepcyjnego	3^x dla $x \in \{1, ..., 3\}$
Krok Pola Recepcyjnego	$x \in \{1, ..., 3\}$
Rozmiar Pola Łączącego	2^x dla $x \in \{1, ..., 3\}$
Współczynnik Porzucenia	$x \in [0, 0.5]$

Tabela 9: Ograniczenia dla hiperparametrów SSN w przypadku Fashion Mnist.

Wprowadzona konwencja tzn. że wartość hiperparametru zostaje np. pomnożona przez jakiś czynnik ma wprowadzić większą różnorodność wielkości sieci znajdujących się w populacji [3]. Algorytm uruchomiłem trzykrotnie dla każdej wartości α z przedziału $[0.2, 0.9]$. Wybrane dziesięć sieci otrzymanych w ten sposób możemy porównać konstruując tabelkę.

α	Rozmiar Sieci	Funkcja Oceny	Dokładność dla zbioru testowego
0.2	97426	2.74	0.906
0.3	69090	3.25	0.897
0.4	67562	3.32	0.895
0.5	23650	3.65	0.891
0.6	21450	3.55	0.899
0.7	7754	3.70	0.897
0.8	5818	3.63	0.892
0.9	986	3.20	0.81

Tabela 10: Sieci otrzymane jako rezultat AG dla zbioru Fashion Mnist.

Wartość funkcji oceny i rozmiar sieci zostały otrzymane podczas przebiegu AG i to one zadecydowały o wyborze tej sieci jako najlepiej przystosowanej. Dodatkowo zmierzyłem ich jakość na zbiorze testowym. Patrząc na powyższą tabelę widzimy jak duży wpływ ma α na rozmiar sieci, a co za tym idzie prędkość wykonania AG (dla mniejszych sieci proces trenowania jest znacznie

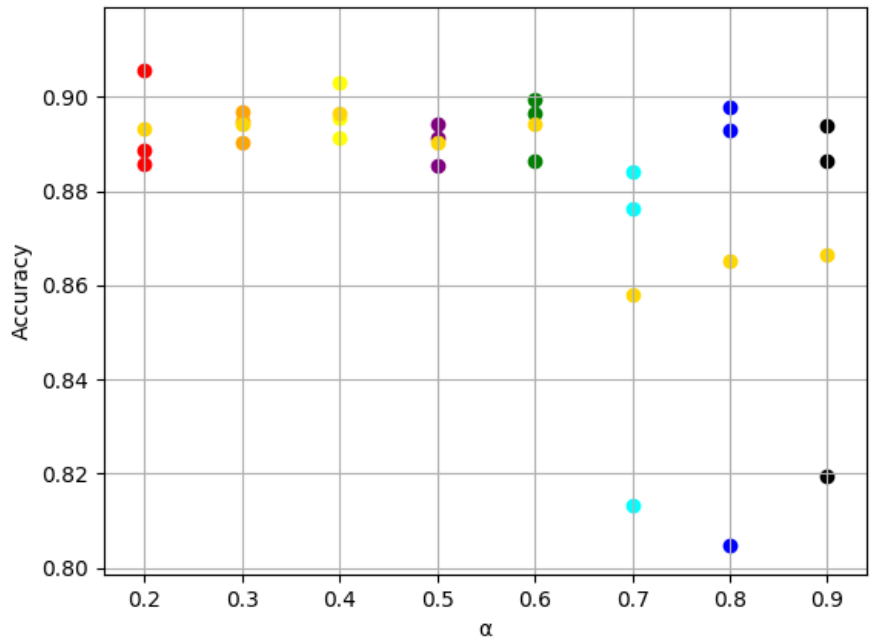
szybszy). Więc dla małych zbiorów danych warto używać $\alpha \sim 0.5$, aby niepotrzebnie nie tworzyć za dużych sieci co prowadzi do niepotrzebnego wydłużenia czasu działania algorytmu. Nie możemy jednak przypisać dla α za dużej wartości ponieważ wtedy na wartość funkcji oceny dokładność sieci będzie miała za mały wpływ.
Z powyższej tabeli wybrałem trzy sieci otrzymane dla α równego 0.3, 0.5, 0.9, które moim zdaniem najlepiej obrazują wpływ współczynnika skalowania na złożoność sieci:

$$\begin{aligned} G_{\alpha_{0.3}} = \big[&[2,0,2,64,9,1,0,0],[2,0,2,32,3,2,0,0],\\ &[3,0,0,0,0,0,8,0],[1,152,2,0,0,0,0,0],\\ &[5,0,0,0,0,0,0,0.27],[1,128,2,0,0,0,0,0],\\ &[5,0,0,0,0,0,0,0.3],[1,24,2,0,0,0,0,0],\\ &[1,88,2,0,0,0,0,0],[1,10,3,0,0,0,0,0]\big] \end{aligned}$$

$$\begin{aligned} G_{\alpha_{0.5}} = \big[&[2,0,0,128,9,1,0,0],[2,0,0,8,3,2,0,0],\\ &[3,0,0,0,0,0,2,0],[1,10,3,0,0,0,0,0]\big] \end{aligned}$$

$$\begin{aligned} G_{\alpha_{0.9}} = \big[&[2,0,1,8,9,3,0,0],[3,0,0,0,0,0,8,0],\\ &[1,10,3,0,0,0,0,0]\big] \end{aligned}$$

Dla $\alpha = 0.3$ dostaliśmy 10 warstwową sieć, natomiast dla wartości 0.5 sieć najlepsza sieć ma już tylko 2 warstwy ukryte więc różnica w czasie trenowania tych sieci jest znacząca. Warto również zauważyć że wszystkie powyższe sieci są typu splotowego, które generalnie lepiej sobie radzą z przetwarzaniem obrazów niż perceptron wielowarstwowy. AG dla tego zbioru danych preferuje wybór tego typu sieci.



Rysunek 16: Wykres przedstawiający dokładność otrzymaną na zbiorze testowym dla wszystkich otrzymanych sieci na zbiorze Fashion Mnist. Dodatkowo kolorem złotym przedstawiłem średnią dokładność dla każdej wartości α.

Z powyższego wykresu widzimy że sieci trenowane z parametrem α o wartości około 0.5 radzą sobie równie dobrze jak siei z α bliskim 0.2, lecz jak wiemy z tabeli [Tab. 11] sieci te mają o wiele mniejsze rozmiary. Dla sieci otrzymanych z przebiegu AG z wartością α bliżej 1.0 widzimy że możemy otrzymać znacznie mniejszą dokładność, ponieważ nie ma ona mniejszy wpływ na ocenę osobników.

5.2 Boston housing

Jest to zbiór zawierający informacje o domach znajdujących się w okolicy Boston Mass w Stanch Zjednoczonych Ameryki. Naszym celem dla tego zbioru danych jest przewidzenie ceny domu w zależności od parametrów takich jak: wskaźnik przestępczości w mieście na osobę, średnia liczba pokojów na mieszkanie itd. [10].

	CRIM	ZN	INDUS	CHAS	NOX	RM	AGE	DIS	RAD	TAX	PTRATIO	B	LSTAT
0	0.00632	18.0	2.31	0.0	0.538	6.575	65.2	4.0900	1.0	296.0	15.3	396.90	4.98
1	0.02731	0.0	7.07	0.0	0.469	6.421	78.9	4.9671	2.0	242.0	17.8	396.90	9.14
2	0.02729	0.0	7.07	0.0	0.469	7.185	61.1	4.9671	2.0	242.0	17.8	392.83	4.03
3	0.03237	0.0	2.18	0.0	0.458	6.998	45.8	6.0622	3.0	222.0	18.7	394.63	2.94
4	0.06905	0.0	2.18	0.0	0.458	7.147	54.2	6.0622	3.0	222.0	18.7	396.90	5.33
5	0.02985	0.0	2.18	0.0	0.458	6.430	58.7	6.0622	3.0	222.0	18.7	394.12	5.21
6	0.08829	12.5	7.87	0.0	0.524	6.012	66.6	5.5605	5.0	311.0	15.2	395.60	12.43
7	0.14455	12.5	7.87	0.0	0.524	6.172	96.1	5.9505	5.0	311.0	15.2	396.90	19.15
8	0.21124	12.5	7.87	0.0	0.524	5.631	100.0	6.0821	5.0	311.0	15.2	386.63	29.93
9	0.17004	12.5	7.87	0.0	0.524	6.004	85.9	6.5921	5.0	311.0	15.2	386.71	17.10

Rysunek 17: Dane Boston Housing przedstawione w postaci tabeli. [10]

Aby wszystkie wielkości danych wejściowych były podobnego rzędu zostały one poddane reskalowaniu. W tym celu wykorzystałem klasę `StandardScaler` z biblioteki `scikit-learn`. Modyfikuję ona dane wejściowe według wzoru [1]:

$$z = \frac{x - \mu}{\sigma} \tag{27}$$

gdzie:
x - pojedyncza próbka pobrana z danych wejściowych,
μ - średnia danych wejściowych,
σ - odchylenie standardowe.

Tak samo jak dla zbioru Fashion Mnist jedynym zmiennym parametrem AG [Tab. 2] jest α.

Parametr	Wartość
γ_v	0.4
ρ_m	0.1
γ_l	0.5
$\mathcal{A}$	10
n	20
n_t	8
γ_c	3
γ_t	10
γ_g	10
γ_r	5

Tabela 11: Parametry AG dla Boston Housing.

Natomiast wartości maksymalne hiperparametrów SSN [Tab. 3] są takie same jak dla zbioru użytego w poprzednim podrozdziale [Tab. 9].
Tak samo jak dla problemu klasyfikacji algorytm uruchomiłem trzykrotnie dla każdej wartości α z przedziału $[0.2, 0.9]$.

α	Rozmiar Sieci	Funkcja Oceny	MSE dla zbioru testowego
0.2	72537	7.10	28.05
0.3	82121	6.58	30.16
0.4	210753	6.01	57.62
0.5	174217	5.84	24.55
0.6	361	5.89	78.55
0.7	657	3.7	88.6
0.8	369	4.4	170.53
0.9	121	3.04	364.81

Tabela 12: Wybrane dziesięć sieci spośród otrzymanych jako rezultat AG dla zbioru Boston housing.

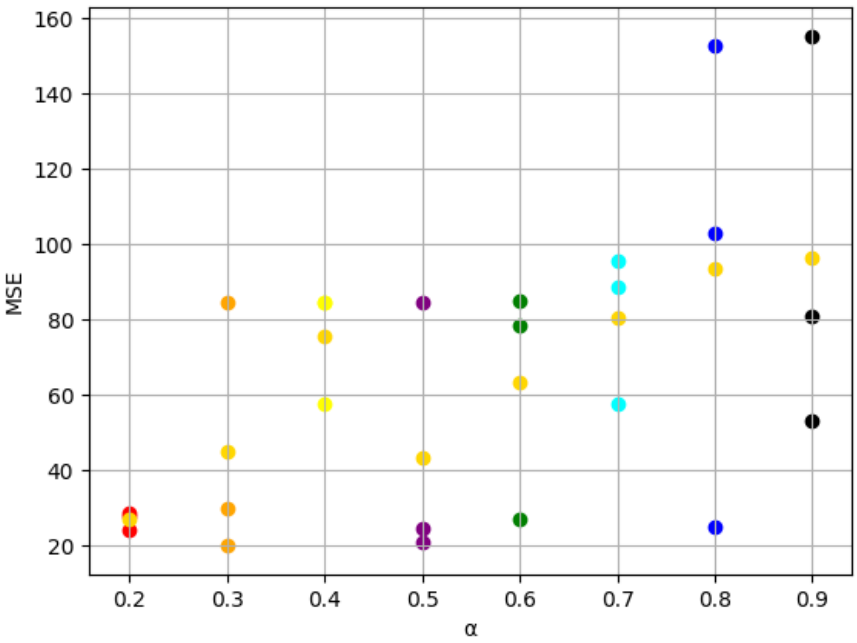
Tak samo jak w poprzednim podrozdziale wartość funkcji oceny i rozmiar zostały otrzymane podczas przebiegu AG, natomiast zamiast dokładności zastosowiłem MSE, aby sprawdzić jak otrzymane sieci radzą sobie na zbiorze testowym.

Z powyższej tabeli wybrałem dwie najbardziej różne sieci pod względem wartości MSE i przedstawiłem je do postaci genotypu.

$$G_{\alpha_{0.5}} = \big[[1,144,2,0,0,0,0,0],[5,0,0,0,0,0,0,0.25],\\ [1,208,2,0,0,0,0,0],[1,248,2,0,0,0,0,0],\\ [1,200,2,0,0,0,0,0],[1,152,2,0,0,0,0,0],\\ [1,64,2,0,0,0,0,0],[5,0,0,0,0,0,0,0.34],\\ [1,1,4,0,0,0,0,0]\big]$$

$$G_{\alpha_{0.9}} = \big[[1,8,0,0,0,0,0,0],[5,0,0,0,0,0,0,0.4],\\ [1,1,4,0,0,0,0,0]\big]$$

Widzimy że w tym przypadku również na rozmiar sieci oraz wartości poszczególnych hiperparametrów w warstwach duży wpływ ma wartość α. Ponieważ nie rozwiązujemy zadania związanego z przetwarzaniem obrazów jedyną siecią jaką może wygenerować AG w tym przypadku jest perceptron wielowarstwowy.



Rysunek 18: Wykres przedstawiający MSE otrzymane na zbiorze testowym dla wszystkich otrzymanych sieci na zbiorze Boston Housing. Kolorem złotym przedstawiłem średnią wartość MSE dla każdej wartości α.

AG tak jak i w przypadku zbioru użytego w podrozdziale wcześniej spisał się dobrze, sieci otrzymane dla niektórych wartości α otrzymują zadowalające wyniki na zbiorach testowych. Odpowiednio zwiększając liczbę eksperymentów γ_r, czy liczbę osobników występujących w populacji n itd. otrzymalibyśmy lepsze wyniki. Lecz nawet dla stosunkowo małych wartości parametrów naszego mikro-AG otrzymane sieci będą znacznie lepsze niż sieci wykonane przez użytkownika budującego od zera bez odpowiedniej wiedzy modele sieci neuronowych, przy znacznie łatwiejszym w obsłudze interfejsie.

6 Podsumowanie

W ramach pracy zaimplementowany został przy pomocy języka `python` Algorytm Genetyczny pozwalający wybrać ustalone hiperparametry sieci dla zadanego zadania. Ponieważ elementarny Algorytm Genetyczny mógłby prowadzić do powstawania nieprawidłowych sieci, musieliśmy odpowiednio zmodyfikować kodowanie osobników oraz operatory genetyczne. Przebieg zmodyfikowanego AG przypomina jednak wersję klasyczną.

Wykorzystanie AG zgodnie z oczekiwaniami ułatwia tworzenie modeli SSN - nie trzeba własnoręcznie wprowadzać liczby oraz rodzajów warstw, co umożliwia osobom niezapoznanym z uczeniem maszynowym tworzyć modele SSN.

Otrzymane wyniki pokazują, że sieci otrzymane przy pomocy AG potrafią otrzymywać zadowalające wartości metryk na zbiorach testowych. Zwiększając liczbę i zakresy wartości optymalizowanych hiperparametrów prawdopodobnie otrzymalibyśmy jeszcze lepsze sieci. Problemem staje

się jednak wymagana moc obliczeniowa, ponieważ tak samo jak dla większych zbiorów danych czas oczekiwania i zapotrzebowanie na zasoby aby AG się wykonał znacząco wzrasta.

7 Bibliografia

References

[1] Lars Buitinck et al. "API design for machine learning software: experiences from the scikit-learn project". In: *ECML PKDD Workshop: Languages for Data Mining and Machine Learning*. 2013, pp. 108–122.

[2] François Chollet et al. *Keras*. https://github.com/fchollet/keras. 2015.

[3] Oliver Schütze David Laredo Yulin Qin and Jian-Qiao Sun. "Automatic Model Selection for Neural Networks". In: (2019).

[4] Aurélien Géron. *Uczenie maszynowe z użyciem Scikit-Learn i TensorFlow*. 2018.

[5] David E. Goldberg. *Algorytmy genetyczne i ich zastosowania*. 1989.

[6] Google. *Google Colab*. https://colab.research.google.com/notebooks/intro.ipynb.

[7] Thomas Kluyver et al. "Jupyter Notebooks – a publishing format for reproducible computational workflows". In: *Positioning and Power in Academic Publishing: Players, Agents and Agendas*. Ed. by F. Loizides and B. Schmidt. IOS Press. 2016, pp. 87–90.

[8] Martin Abadi et al. *TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems*. Software available from tensorflow.org. 2015. URL: https://www.tensorflow.org/.

[9] Travis E Oliphant. *A guide to NumPy*. Vol. 1. Trelgol Publishing USA, 2006.

[10] UCI Machine Learning Repository. *Boston Housing Dataset*. https://archive.ics.uci.edu/ml/machine-learning-databases/housing/.

[11] Guido Van Rossum and Fred L. Drake. *Python 3 Reference Manual*. Scotts Valley, CA: CreateSpace, 2009. ISBN: 1441412697.

[12] Wikipedia. *Crossover (genetic algorithm)*. https://en.wikipedia.org/wiki/Crossover_(genetic_algorithm).

[13] Han Xiao, Kashif Rasul, and Roland Vollgraf. *Fashion-MNIST: a Novel Image Dataset for Benchmarking Machine Learning Algorithms*. Aug. 28, 2017. arXiv: cs.LG/1708.07747 [cs.LG].