Podstawy Sztucznej Inteligencji

Piotr Duch
pduch@iis.p.lodz.pl

7 listopada 2024

Spis treści

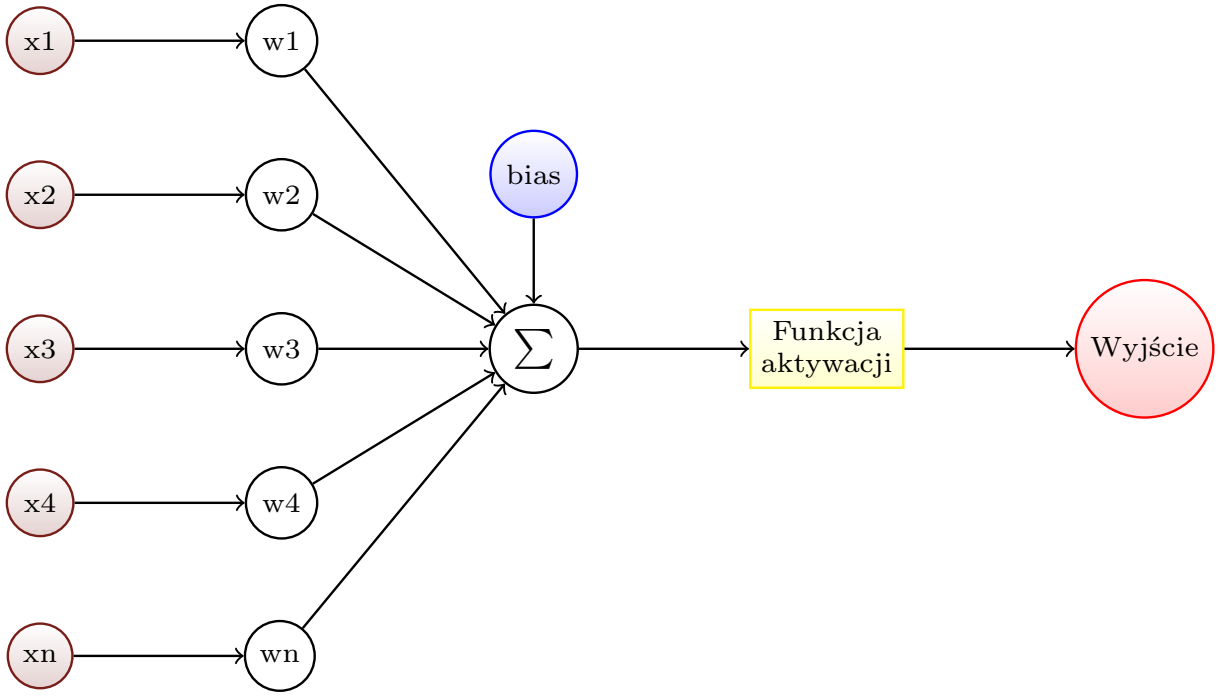
1 Laboratorium 1

Celem laboratorium jest zapoznanie się z zasadą działania neuronu oraz budowa prostej sieci neuronowej.

1.1 Wstęp teoretyczny

1.1.1 Neuron

Sieci neuronowe składają się z pojedynczych neuronów, ułożonych w warstwy. Operacje wykonywane przez pojedynczy neuron zostały przedstawione na rysunku 1.



Rysunek 1: Operacje wykonywane przez pojedynczy neuron.

Do neuronu przekazywany jest zbiór wartości wejściowych ($x1$, $x2$, $x3$, $x4$ ...), którymi mogą być wartości przekazane na wejście sieci lub wartości wyjściowe neuronów, znajdujących się w poprzedniej warstwie. Następnie są one mnożone przez odpowiadające im wagi ($w1$, $w2$, $w3$, $w4$ ...), które to stanowią połączenie pomiędzy neruonami. W trakcie uczenia sieci to właśnie wartości wag będą aktualizowane. Powstałe w ten sposób iloczyny są sumowane. Dodatkowo możliwe jest dodanie do wartości neuronu biasu (b). Dodanie biasu umożliwia przesuwanie progu aktywacji neuronu wzdłuż osi x (funkcja aktywacji zostanie dokładniej omówiona w rozdziale 3.2.1) oraz powoduje poprawę własności neuronu. Dla tak obliczonego wyrażenia stosowana jest funkcja aktywacji, a wartość wyznaczona przez tą funkcję jest równoznaczna z wyjściem neuronu.

Z matematycznego punktu operacje wykonywane przez pojedynczy neuron można przedstawić za pomoć wzoru:

$$y = f(\sum_{i=1}^{n} w_i x_i + b) \tag{1}$$

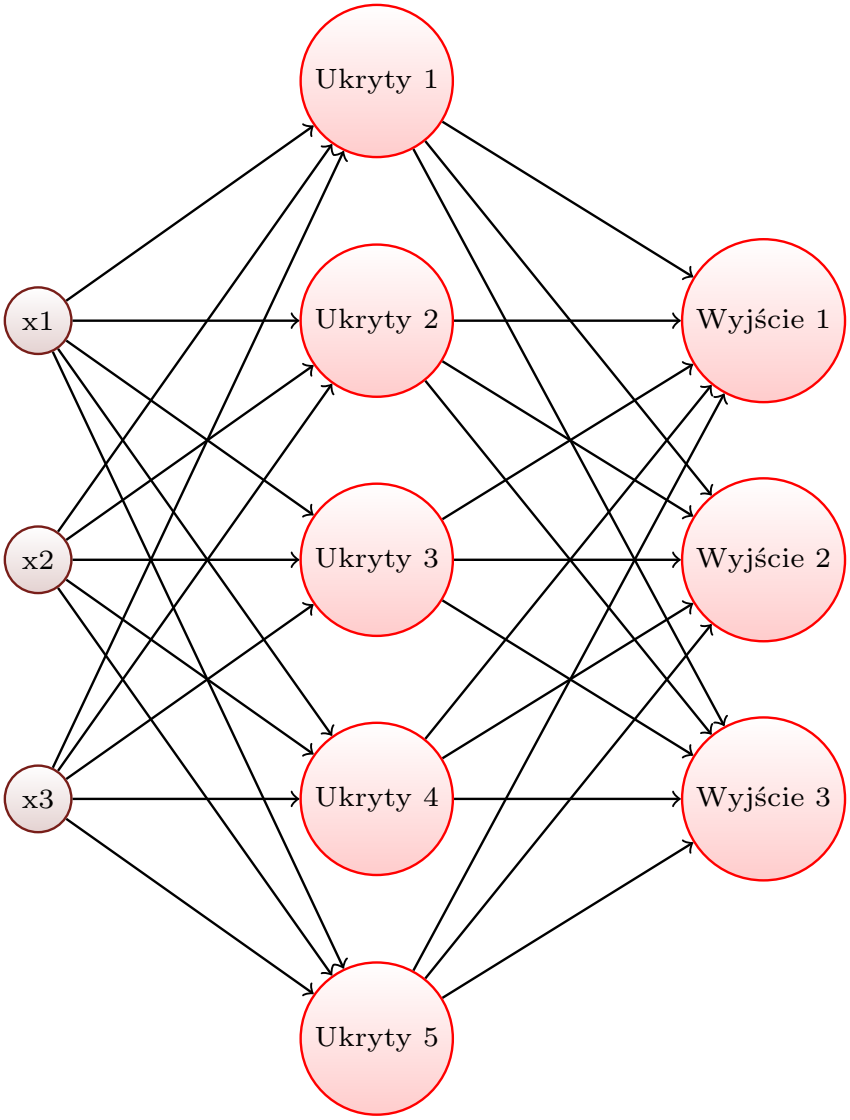
W przypadku najprostszej wersji neuronu, bez funkcji aktywacji oraz biasu równanie to upraszcza się do postaci:

$$y = \sum_{i=1}^{n} w_i x_i \tag{2}$$

1.1.2 Warstwa Gęsta (ang. *Dense layer* lub *Fully Connected layer*)

Warstwa gęsta jest to warstwa, w której wszystkie neurony z warstwy wcześniejszej są połączone ze wszystkimi neuronami z warstwy kolejnej. Przy czym należy pamiętać, że liczba neuronów w

poszczególnych warstwach może być rózna, tak jak to zostało przedstawione na rysunku 2.



Rysunek 2: Prosta sieć neuronowa składająca się z dwóch warstw: warstwy ukrytej składającej się z 5 neuronów oraz warstwy wyjściowej składającej się z 3 neuronów.

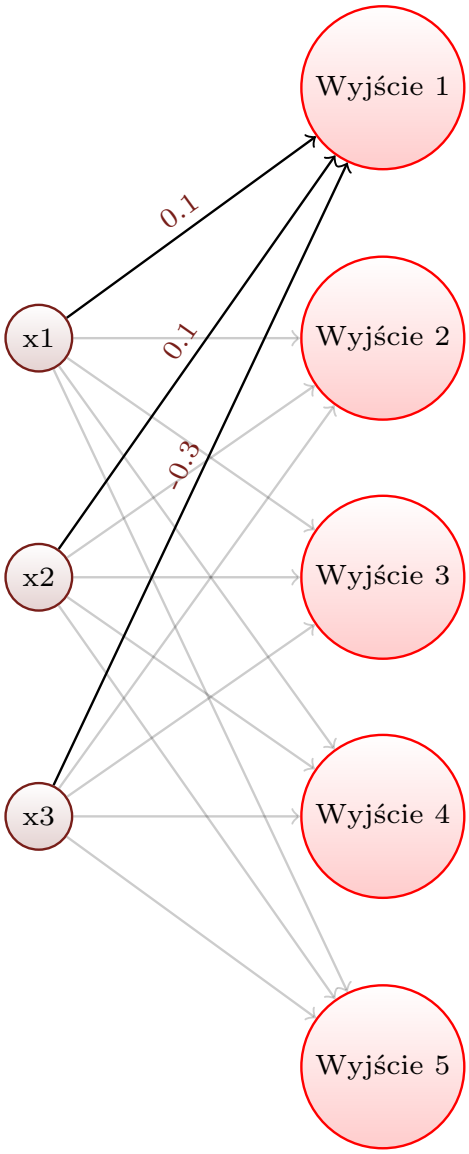
1.2 Wstęp praktyczny

1.2.1 Przykład 1 - wyznaczania odpowiedzi sieci neuronowej

Wartości wag dla poszczególnych warstw sieci neuronowych można przedstawić za pomocą macierzy. Dla sieci neuronowej przedstawionej na rys. 3 przyjmijmy następujące wartości wag (dla uproszczenia omawianego przypadku rozważamy neurony bez biasu):

$$W = \begin{array}{c} \begin{array}{ccc} \text{Wejście 1} & \text{Wejście 2} & \text{Wejście 3} \end{array} \\ \begin{bmatrix} 0.1 & 0.1 & -0.3 \\ 0.1 & 0.2 & 0.0 \\ 0.0 & 0.7 & 0.1 \\ 0.2 & 0.4 & 0.0 \\ -0.3 & 0.5 & 0.1 \end{bmatrix} \end{array} \begin{array}{l} \text{Wyjście 1} \\ \text{Wyjście 2} \\ \text{Wyjście 3} \\ \text{Wyjście 4} \\ \text{Wyjście 5} \end{array} \tag{3}$$

oraz następujący zestaw danych wejściowych:



Rysunek 3: Prosta sieć neuronowa składająca się z pięciu neuronów posiadających trzy wejścia.

$$x = \begin{matrix}\text{Seria 1}\\ \begin{bmatrix}0.5\\0.75\\0.1\end{bmatrix}\end{matrix} \begin{matrix}\text{Wejście 1}\\ \text{Wejście 2}\\ \text{Wejście 3}\end{matrix} \tag{4}$$

Przekształcając równanie 2 na postać macierzową otrzymamy:

$$output = W * x \tag{5}$$

Podstawiając do równania 5 wartości danych wejściowych (4) oraz macierz z wagami (3) można wyznaczyć wartości neuronów wyjściowych (w tym przypadku będą to również wartości wyjściowe sieci):

$$output = W * x = \begin{bmatrix}0.1 & 0.1 & -0.3\\ 0.1 & 0.2 & 0.0\\ 0.0 & 0.7 & 0.1\\ 0.2 & 0.4 & 0.0\\ -0.3 & 0.5 & 0.1\end{bmatrix} * \begin{bmatrix}0.5\\0.75\\0.1\end{bmatrix} = \begin{bmatrix}0.095\\0.2\\0.535\\0.4\\0.235\end{bmatrix}$$

1.2.2 Przykład 2 - wyznaczania odpowiedzi głębokiej sieci neuronowej

Dla sieci neuronowej przedstawionej na rys. 2 przyjmijmy następujące wartości wag:

$$Wh = \begin{array}{c} \begin{matrix} \text{Wejście 1} & \text{Wejście 2} & \text{Wejście 3} \end{matrix} \\ \begin{bmatrix} 0.1 & 0.1 & -0.3 \\ 0.1 & 0.2 & 0.0 \\ 0.0 & 0.7 & 0.1 \\ 0.2 & 0.4 & 0.0 \\ -0.3 & 0.5 & 0.1 \end{bmatrix} \end{array} \begin{matrix} \text{Ukryty 1} \\ \text{Ukryty 2} \\ \text{Ukryty 3} \\ \text{Ukryty 4} \\ \text{Ukryty 5} \end{matrix} \tag{6}$$

$$Wy = \begin{array}{c} \begin{matrix} \text{Ukryty 1} & \text{Ukryty 2} & \text{Ukryty 3} & \text{Ukryty 4} & \text{Ukryty 5} \end{matrix} \\ \begin{bmatrix} 0.7 & 0.9 & -0.4 & 0.8 & 0.1 \\ 0.8 & 0.5 & 0.3 & 0.1 & 0.0 \\ -0.3 & 0.9 & 0.3 & 0.1 & -0.2 \end{bmatrix} \end{array} \begin{matrix} \text{Wyjście 1} \\ \text{Wyjście 2} \\ \text{Wyjście 3} \end{matrix} \tag{7}$$

Pierwszym krokiem będzie obliczenie wartości neuronów w warstwie ukrytej, w tym celu skorzystamy z równania 5. Po podstawieniu danych wejściowych (4) oraz macierzy z wagami dla neuronów warstwy ukrytej (29) można wyznaczyć wartości neuronów ukrytych (warstwy 1):

$$layer\_hidden\_1 = Wh * x = \begin{bmatrix} 0.1 & 0.1 & -0.3 \\ 0.1 & 0.2 & 0.0 \\ 0.0 & 0.7 & 0.1 \\ 0.2 & 0.4 & 0.0 \\ -0.3 & 0.5 & 0.1 \end{bmatrix} * \begin{bmatrix} 0.5 \\ 0.75 \\ 0.1 \end{bmatrix} = \begin{bmatrix} 0.095 \\ 0.2 \\ 0.535 \\ 0.4 \\ 0.235 \end{bmatrix}$$

Następnie, korzystając z tego samego równania, możemy wyznaczyć wartości neuronów w ostatniej warstwie sieci, podstawiając zamiast danych wejściowych wartości neuronów z warstwy ukrytej (*layer_hidden_1*) oraz macierz wag dla neuronów warstwy wyjściowej (Wy).

$$layer\_output = Wy * layer\_hidden\_1 = \begin{bmatrix} 0.7 & 0.9 & -0.4 & 0.8 & 0.1 \\ 0.8 & 0.5 & 0.3 & 0.1 & 0.0 \\ -0.3 & 0.9 & 0.3 & 0.1 & -0.2 \end{bmatrix} * \begin{bmatrix} 0.095 \\ 0.2 \\ 0.535 \\ 0.4 \\ 0.235 \end{bmatrix} = \begin{bmatrix} 0.376 \\ 0.3765 \\ 0.305 \end{bmatrix}$$

1.3 Zadanie 1

Zaimplementuj funkcję `neuron` obliczającą odpowiedź pojedynczego neuronu (rys. 1) zgodnie ze wzorem 1. Funkcja powinna przyjmować trzy parametry:

- *input* - wartości wejściowe neuronu w postaci pojedynczego wektora,
- *weights* - wagi neuronu w postaci pojedynczego wektora,
- *bias* - wartość biasu dla danego neuronu.

1.4 Zadanie 2

Zaimplementuj prostą sieć neuronową posiadającą trzy wejścia oraz pięć neuronów wyjściowych (rys. 3). Przygotuj funkcję `neural_network`, która przyjmie wektor wartości wejściowych (w tym zadaniu będzie on 3-elementowy, ale przygotuj funkcję uniwersalną, która będzie działała dla dowolnie dużych wektorów) oraz macierz dwuwymiarową wag (liczba wierszy macierzy musi odpowiadać liczbie neuronów wyjściowych sieci, a liczba kolumn liczbie wejść). Funkcja powinna zwrócić wektor z wartościami poszczególnych neuronów. Zwróć uwagę, że liczba elementów w wektorze wejściowym musi odpowiadać liczbie kolumn w macierzy, a liczba elementów w wektorze wyjściowym liczbie wierszy w macierzy wag. Przetestuj działanie swojej funkcji dla danych z przykładu 1.2.1;

1.5 Zadanie 3

Zaimplementuj głęboką sieć neuronową składającą się z trzech wejść, pięciu neuronów w warstwie ukrytej oraz trzech wyjść (rys. 2). Każdy neuron znajdujący się w warstwie ukrytej (środkowa) oraz wyjściowej posiada zestaw wag, których liczba odpowiada liczbie neuronów w warstwie bezpośrednio go poprzedzającej. Obliczenie odpowiedzi takiej sieci dla danych wejściowych odbywa się najpierw poprzez obliczenie wartości neuronów w warstwie ukrytej (analogicznie do poprzedniego zadania), a następnie na podstawie tych wartości wyznaczeniu wartości neuronów w warstwie wyjściowej.

Przygotuj funkcję `deep_neural_network`, która przyjmie wektor wartości wejściowych (w tym zadaniu będzie on 3-elementowy, ale przygotuj funkcję uniwersalną, która będzie działała dla dowolnie dużych wektorów) oraz listę macierzy wag (wagi dla każdego neuronu w każdej warstwie). Funkcja powinna zwrócić wektor z wartościami poszczególnych neuronów warstwy wyjściowej. Zwróć uwagę, że liczba neuronów w każdej z warstw może być różna. Przetestuj działanie sieci dla danych z przykładu 1.2.2.

1.6 Zadanie 4

Przygotuj API, które pozwoli na budowanie sieci składającej się z warstw w pełni połączonych (ang. *fully connected layers*), czyli takich jak w poprzednich zadaniach. Program ma umożliwiać budowanie sieci o dowolnej liczbie neuronów wejściowych oraz wyjściowych, dowolnej liczbie warstw ukrytych z dowolną liczbą neuronów. Program powinien na początek generować losowe wartości wag dla każdej warstwy zbudowanej sieci. Przygotuj klasę (bądź zestaw funkcji) do obsługi sieci neuronowej, w której będą dostępne następujące funkcje:

- `add_layer(n, [weight_min_value, weight_max_value])` - funkcja dodaje warstwę n neuronów do sieci, opcjonalnie może przyjmować zakres wartości, z którego będą losowane wagi (informację o liczbie danych wejściowych dla warstwy można określić na podstawie liczby neuronów w warstwie poprzedniej, a w przypadku pierwszej warstwy można ją przekazać w konstruktorze sieci),
- `predict(input)` - funkcja przyjmuje wektor danych wejściowych i oblicza dla niego odpowiedź sieci, funkcja zwraca obliczone wartości neuronów ostatniej warstwy,
- `load_weights(file_name)` - funkcja odczytuje wagi z pliku o nazwie (*file_name*).

1.7 Jak to robią profesjonaliści (1)

W przemysłowych zastosowaniach bardzo rzadko implementuje się sieci neuronowe od zera, najczęściej korzysta się z gotowych frameworków. Najpopularniejszymi są: dlib (C++), tensorflow, keras i PyTorch (wszystkie python). W ramach przygotowanej instrukcji pokazane zostanie jak wykonać te same zadania w kerasie.

```
import numpy as np
from keras.models import Sequential
from keras.layers import Dense

model = Sequential()
weights = [np.array([[0.1, 0.1, 0, 0.2, -0.3],
                     [0.1, 0.2, 0.7, 0.4, 0.5],
                     [-0.3, 0.0, 0.1, 0, 0.1]])]
model.add(Dense(5, input_dim=3, weights=weights, use_bias = False))
inputs = np.array([[0.5, 0.75, 0.1]])
print(model.predict(inputs))
```

Listing 1: Implementacja rozwiązania zadania 4 w kerasie (Keras, wersja 2.2.4)

W celu zbudowania prostej sieci neuronowej w kerasie należy dołączyć do projektu dwa moduły:

- `Sequential` - Model sekwencyjny jest stosowany w przypadku, kiedy warstwy układają się w stos, w którym każda warstwa ma dokładnie jeden tensor (wektor) wejściowy i jeden tensor (wektor) wyjściowy. W omawianym przykładzie sieć będzie miała jedną warstwę, na której wejście będzie podawany wektor 3-elementowy, a na wyjściu będzie również wektor 3-elementowy.
- `Dense` - Warstwa głęboka (czasami również nazywana *Fully connected*).

Po dołączeniu odpowiednich modułów do projektu można rozpocząć budowanie sieci. Najpierw określamy sposób budowania sieci (listing 1, linia 10), w tym przypadku wybierzemy łatwiejszy - sekwencyjny. Oprócz sekwencyjnego można sieć budować w oparciu o model funkcyjny. Do tak zadeklarowanego modelu możemy następnie dodawać warstwy (metoda `add`). W linii 11 dodawana jest warsta *Dense*, która przyjmuje m.in. następujące parametry:

- liczba neuronów w warstwie (w tym przypadku 5),
- opcjonalnie `input_dim` - rozmiar wektora wejściowego, jeżeli nie zostanie podany sieć wykryje liczbę elementów wejściowych na podstawie m.in.: wielkości poprzedniej warstwy, wymiarów wag lub danych przekazywanych do sieci przy pierwszym wywołaniu funkcji `fit` lub `predict`).
- opcjonalnie `weights` - wartości wag dla danej warstwy, jeżeli nie zostaną podane w parametrze, wagi zostaną zainicjowane losowymi wartościami. Należy zwrócić uwagę na wartości przekazane w parametrze (linia 5) - wagi mają rozmiar liczba_neuronów_w_warstwie x liczba_wejść (w tym przypadku 3x3). Warstwa wykonuje następującą operację:

$$input * weights$$

.

Wyznaczenie odpowiedzi sieci odbywa się za pomocą metody `predict` (listing 1, linia 13). Metoda przyjmuje listę wejść (*batch* - pojęcie to będzie omówione w ramach laboratorium 4), dla których ma być wyznaczona odpowiedź sieci. Dlatego w omawianym przykładzie na wejściu podawana jest tablica z jednym wektorem w środku. Jeżeli do funkcji `predict` przekazanych zostanie więcej wektorów, to wyznaczona zostanie odpowiedź sieci dla każdego z nich.

1.8 Zadanie 5 - dla chętnych

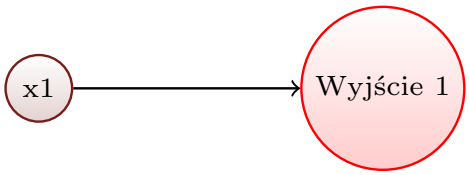
Wykonaj zadanie 1.5 z wykorzystaniem frameworku Keras, podobnie jak to zostało przedstawione na listingu 1.

2 Laboratorium 2

Celem laboratorium jest zapoznanie się z podstawową metodą uczenia sieci neuronowej - metodą gradientu prostego - *Gradient descent*. Metoda gradientu prostego jest iteracyjnym algorytmem wyszukiwania minimum zadanej funkcji celu, w rozważanym przypadku szukaną funkcją celu będzie błąd sieci neuronowej.

2.1 Wstęp teoretyczny

Załóżmy, że mamy do czynienia z następującą siecią z rysunku 4. Do określenia błędu takiej sieci (w litaraturze zamiast o funkcji błędu często mówi się o funkcji kosztu - ang. *cost* lub straty - ang. *loss*) możemy wykorzystać błąd średniokwadratowy (MSE - ang. *Mean Squared Error*) 9.



Rysunek 4: Prosta sieć neuronowa składająca się z jednej warstwy z pojedynczym neuronem o jednym wejściu.

$$error = \frac{1}{N}\sum_{i=1}^{N}(prediction_i - goal_i)^2 \tag{8}$$

gdzie:

- N - liczba neuronów w warstwie wyjściowej,
- $prediction_i$ - odpowiedź i-tego neuronu sieci dla danych wejściowych (wartość neuronu wyjściowego),
- $goal_i$ - oczekiwana wartość i-tego neuronu, jaką powinna zwrócić sieć.

W przypadku sieci z rys. 4 otrzymujemy następującą postać błędu (9).

$$error = (prediction - goal)^2 \tag{9}$$

Na podstawie obliczonego błędu odpowiedzi sieci możemy dowiedzieć jak bardzo wynik zwrócony przez sieć różni się od oczekiwanego. Nie pozwoli nam to jeszcze uczyć sieci, bo potrzebna jest informacja, w którą stronę należy zmodyfikować wartości poszczególnych wag neuronów. W tym celu wykorzystamy pochodną funkcji błędu, która da nam informację o tym w jakim kierunku i o ile zmienić wartości wag neuronu, aby błąd w następnej iteracji był mniejszy. Pochodna funkcji MSE została przedstawiona we wzorze 22, a jej wyprowadzenie można znaleźć pod linkiem `https://towardsdatascience.com/calculating-gradient-descent-manually-6d9bee09aa0b`.

$$delta = 2 * \frac{1}{N}(\mathbf{prediction} - \mathbf{goal}) \otimes \mathbf{x} \tag{10}$$

gdzie x jest wektorem przekazanym na wejście ostatniej warstwy.

W przypadku rozpatrywanej sieci równanie to upraszcza się do następującej postaci:

$$delta = 2 * (prediction - goal) * x \tag{11}$$

Dodatkowo wprowadzimy jeszcze jeden parametr uczenia - *alpha* (lub *learning rate*), kontrolujący szybkość uczenia sieci (najczęściej przyjmuje dowolną wartość z zakresu $< 0, 1 >$, duża wartość współczynnika powoduje, że sieć się szybko uczy, ale istnieje duże prawdopodobieństwo przestrzelenia wartości wag). Ostatecznie nowa wartość wagi neuronu wyznaczana jest ze wzoru 12.

$$W = W - delta * alpha \tag{12}$$

2.2 Wstęp praktyczny

2.2.1 Przykład 1

Przeanalizujmy proces aktualizacji wag dla sieci neuronowej z rysunku 3. Przyjmijmy wartość współczynnika uczenia 0.01, następujące wartości wag:

$$W = \begin{matrix} \\ \\ \\ \\ \\ \end{matrix} \begin{matrix} \text{Wejście 1} \\ \text{Wejście 2} \\ \text{Wejście 3} \end{matrix} \begin{bmatrix} 0.1 & 0.1 & -0.3 \\ 0.1 & 0.2 & 0.0 \\ 0.0 & 0.7 & 0.1 \\ 0.2 & 0.4 & 0.0 \\ -0.3 & 0.5 & 0.1 \end{bmatrix} \begin{matrix} \text{Wyjście 1} \\ \text{Wyjście 2} \\ \text{Wyjście 3} \\ \text{Wyjście 4} \\ \text{Wyjście 5} \end{matrix} \tag{13}$$

oraz następujący zestaw danych wejściowych:

$$x = \begin{matrix} \\ \\ \\ \end{matrix} \begin{matrix} \text{Seria 1} \end{matrix} \begin{bmatrix} 0.5 \\ 0.75 \\ 0.1 \end{bmatrix} \begin{matrix} \text{Wejście 1} \\ \text{Wejście 2} \\ \text{Wejście 3} \end{matrix} \tag{14}$$

i oczekiwanych:

$$y = \begin{matrix} \\ \\ \\ \\ \\ \end{matrix} \begin{matrix} \text{Seria 1} \end{matrix} \begin{bmatrix} 0.1 \\ 1.0 \\ 0.1 \\ 0.0 \\ -0.1 \end{bmatrix} \begin{matrix} \text{Wyjście 1} \\ \text{Wyjście 2} \\ \text{Wyjście 3} \\ \text{Wyjście 4} \\ \text{Wyjście 5} \end{matrix} \tag{15}$$

Na początek należy obliczyć odpowiedź sieci dla pierwszej serii danych wejściowych, analogicznie jak w przykładzie 1.2.1 z laboratorium 1:

$$output = W * x = \begin{bmatrix} 0.1 & 0.1 & -0.3 \\ 0.1 & 0.2 & 0.0 \\ 0.0 & 0.7 & 0.1 \\ 0.2 & 0.4 & 0.0 \\ -0.3 & 0.5 & 0.1 \end{bmatrix} * \begin{bmatrix} 0.5 \\ 0.75 \\ 0.1 \end{bmatrix} = \begin{bmatrix} 0.095 \\ 0.2 \\ 0.535 \\ 0.4 \\ 0.235 \end{bmatrix}$$

Następnie wyznaczamy wartość pochodnej funkcji błędu, zgodnie ze wzorem 22. Ponieważ w rozważanym przypadku na wjściu sieci otrzymamy wektor, przekształcimy to równanie na postać wektorową:

$$delta = 2 * \frac{1}{N}(\mathbf{output} - \mathbf{y}) \otimes \mathbf{x} = \frac{2}{5} * (\begin{bmatrix} 0.095 \\ 0.2 \\ 0.535 \\ 0.4 \\ 0.235 \end{bmatrix} - \begin{bmatrix} 0.1 \\ 1.0 \\ 0.1 \\ 0.0 \\ -0.1 \end{bmatrix}) \otimes \begin{bmatrix} 0.5 \\ 0.75 \\ 0.1 \end{bmatrix} =$$

$$= 0.4 * \begin{bmatrix} -0.005 \\ -0.8 \\ 0.435 \\ 0.4 \\ 0.335 \end{bmatrix} \otimes \begin{bmatrix} 0.5 \\ 0.75 \\ 0.1 \end{bmatrix} = \begin{bmatrix} -0.001 & -0.0015 & -0.0002 \\ -0.16 & -0.24 & -0.032 \\ 0.087 & 0.1305 & 0.0174 \\ 0.08 & 0.12 & 0.016 \\ 0.067 & 0.1005 & 0.0134 \end{bmatrix}$$

gdzie $\otimes$ oznacza iloczyn zewnętrzny (ang. *outer product*).
Na koniec możemy zaktualizować wartości wag wszystkich neuronów ostatniej warstwy zgodnie ze wzorem 12.

$$W = W - alpha * delta =$$

$$= \begin{bmatrix} 0.1 & 0.1 & -0.3 \\ 0.1 & 0.2 & 0.0 \\ 0.0 & 0.7 & 0.1 \\ 0.2 & 0.4 & 0.0 \\ -0.3 & 0.5 & 0.1 \end{bmatrix} - 0.01 * \begin{bmatrix} -0.001 & -0.0015 & -0.0002 \\ -0.16 & -0.24 & -0.032 \\ 0.087 & 0.1305 & 0.0174 \\ 0.08 & 0.12 & 0.016 \\ 0.067 & 0.1005 & 0.0134 \end{bmatrix} =$$

$$= \begin{bmatrix} 0.10001 & 0.100015 & -0.299998 \\ 0.1016 & 0.2024 & 0.00032 \\ -0.00087 & 0.698695 & 0.099826 \\ 0.1992 & 0.3988 & -0.00016 \\ -0.30067 & 0.498995 & 0.099866 \end{bmatrix}$$

2.3 Zadanie 1

Zaimplementuj sieć przedstawioną na rys. 4. Sieć składa się z pojedynczego neuronu o jednym wejści, w każdej epoce ma następować aktualizacja wagi neuronu zgodnie ze wzorem 12. Przetestuj działanie sieci dla następujących danych wejściowych:

- wartość początkowa wagi - 0.5,
- wartość oczekiwana - 0.8,
- wejście - 2,
- alpha - 0.1.

Oczekiwany wynik w 5 epoce:

- wyjście sieci - $output = 0.80032$,
- błąd - $error = 0.0000001024$.

Oczekiwany wynik w 20 epoce [1]:

- wyjście sieci - $output = 0.800000000000010$,
- błąd - $error = 0.000000000000000$.

Przetestuj działanie sieci również dla następujących danych wejściowych:

- wejście - 2, współczynnik uczenia - 1,
- wejście - 0.1, współczynnik uczenia - 1,

Po ilu iteracjach sieć jest w stanie zbliżyć się do oczekiwanego wyniku końcowego? Jak myślisz, dlaczego tak się dzieje?

2.4 Zadanie 2

Zmodyfikuj sieć z zadania 1.4 z laboratorium 1 w taki sposób, aby po każdej serii następowała aktualizacja wag neuronu zgodnie z przykładem 2.2.1.
Przetestuj działanie sieci dla następujących danych wejściowych:

- Dane wejściowe (cztery serie danych):

$$x = \begin{array}{c} \begin{matrix} \text{Seria 1} & \text{Seria 2} & \text{Seria 3} & \text{Seria 4} \end{matrix} \\ \begin{bmatrix} 0.5 & 0.1 & 0.2 & 0.8 \\ 0.75 & 0.3 & 0.1 & 0.9 \\ 0.1 & 0.7 & 0.6 & 0.2 \end{bmatrix} \end{array} \begin{matrix} \text{Wejście 1} \\ \text{Wejście 2} \\ \text{Wejście 3} \end{matrix}$$

[1] `https://pduch.iis.p.lodz.pl/PSI/Lab2_zad1_logs.txt`

- Oczekiwane wyniki (dla poszczególnych serii danych):

$$y = \begin{array}{c} \begin{array}{cccc} \text{Seria 1} & \text{Seria 2} & \text{Seria 3} & \text{Seria 4} \end{array} \\ \begin{bmatrix} 0.1 & 0.5 & 0.1 & 0.7 \\ 1.0 & 0.2 & 0.3 & 0.6 \\ 0.1 & -0.5 & 0.2 & 0.2 \\ 0.0 & 0.3 & 0.9 & -0.1 \\ -0.1 & 0.7 & 0.1 & 0.8 \end{bmatrix} \end{array} \begin{array}{l} \text{Wyjście 1} \\ \text{Wyjście 2} \\ \text{Wyjście 3} \\ \text{Wyjście 4} \\ \text{Wyjście 5} \end{array}$$

- Współczynnik szybkości uczenia:

$$alpha = 0.01.$$

Naucz sieć poprawnego przewidywania wartości oczekiwanych. W tym celu uruchom uczenie sieci 1000 razy, za każdym razem sieć ma wykonać obliczenia i aktualizować wagi dla każdej z serii danych. Po każdej epoce (aktualizacji wag dla wszystkich serii) wyświetl błąd działania sieci (suma błędów popełnionych przez sieć dla każdej z serii, błąd dla poszczególnych serii powinien być obliczany według wzoru 8). Wagi sieci, wyniki otrzymywane dla poszczególnych serii oraz wartości błędu dla 10 pierwszych epok są dostępne pod linkiem: `https://pduch.iis.p.lodz.pl/PSI/Lab2_zad2_logs.txt`. Po 1000 epokach błąd powinien wynosić 0,258218.

2.5 Zadanie 3

Zmodyfikuj sieć z zadania 2.4 tak, aby składała się z jednej warstwy złożonej z 4 neuronów oraz posiadała 3 wejścia. Celem sieci jest nauczenie sieci rozpoznawania kolorów (czerwony (1), zielony (2), niebieski (3), żółty (4)) na podstawie 3 wartości składowych koloru (RGB). Poszczególne neurony w warstwie wyjściowej oznaczają odpowiedź sieci dla danego koloru. Przyjmij, że ostatecznym kolorem, jaki sieć zwróciła jest neuron z najwyższą wartością. W procesie uczenia przyjmij, że oczekiwane wartości na wyjściu to 0 dla neuronów nieodpowiadających danemu kolorowi oraz 1 na wyjściu odpowiadającemu kolorowi.
Na przykład wejście $\begin{bmatrix} 0.91 & 0.82 & 0.05 \end{bmatrix}$ odpowiada kolorowi o ID 4 (żółty). Więc wektor oczekiwanej odpowiedzi sieci dla takich danych wejściowych powinien wyglądać następująco $\begin{bmatrix} 0.0 & 0.0 & 0.0 & 1 \end{bmatrix}$.
Zbiór treningowy dostępny jest pod linkiem: `http://pduch.iis.p.lodz.pl/PSI/training_colors.txt`, a zbiór testowy: `http://pduch.iis.p.lodz.pl/PSI/test_colors.txt`.
Zbiór treningowy - zbiór, na którym następuje proces uczenia sieci.
Zbiór testowy - zbiór, dla którego sprawdzana jest poprawność działania sieci.
W pierwszym etapie program uczy sieć neuronową na przykładzie zbioru treningowego (zwróć uwagę, że może być konieczność trenowania sieci przez kilka / kilkudziesiąt epok). Po zakończeniu procesu uczenia sprawdzana jest skuteczność sieci na podstawie zbioru testowego. Przewidywana skuteczność sieci to 100%. Skuteczność sieci obliczana jest na podstawie procentowego udziału poprawnie rozpoznanych kolorów przez sieć (neuron o najwyższej wartości odpowiada rozpoznanemu kolorowi) do wszystkich badanych próbek.

2.6 Jak to robią profesjonaliści (2)

Zadanie 2.4 można rozwiązać za pomocą frameworka keras (listing 2).

```
import numpy as np
from keras.models import Sequential
from keras.layers import Dense
from keras.optimizers import SGD

model = Sequential()
weights = [np.array([[0.1, 0.1, 0, 0.2, -0.3],
                     [0.1, 0.2, 0.7, 0.4, 0.5],
                     [-0.3, 0.0, 0.1, 0, 0.1]])]
model.add(Dense(5, input_dim=3, weights=weights, use_bias = False))
inputs = np.array([[0.5, 0.75, 0.1], [ 0.1, 0.3, 0.7], [ 0.2, 0.1, 0.6], [ 0.8, 0.9, 0.2]])
expected_outputs = np.array([[0.1, 1, 0.1, 0, -0.1], [ 0.5, 0.2, -0.5, 0.3, 0.7], [ 0.1, 0.3, 0.2, 0.9, 0.1], [ 0.7, 0.6, 0.2, -0.1, 0.8]])
opt = SGD(lr=0.01)
model.compile(opt, loss='mse')
for _ in range(10):
```

```
16      for input, expected_output in zip(inputs, expected_outputs):
17          model.fit(np.array([input]), np.array([expected_output]))
18          print(model.get_weights())
```

Listing 2: Implementacja rozwiązania zadania 2.4 w kerasie (Keras, wersja 2.2.4)

Oprócz modułów znanych już z listingu 2 do kodu dołączony został nowy moduł `SGD` - jest to optymalizator określający w jaki sposób będzie następowała aktualizacja wag sieci. W tym przypadku wybrany został optymalizator `SGD` (ang. *Stochastic Gradient Descent*) ponieważ jest on jednym z najprostszych optymalizatorów stosowanych w sieciach neuronowych. Oprócz niego bardzo popularnymi optymalizatorami stosowanymi w sieciach neuronowych są: `RMSprop` oraz `Adam`. W kodzie wykorzystany jest optymalizator z wartościami domyślnymi, a parametr *lr* (ang. *learning_rate*) ustawiony został na 0.01.

Po wybraniu optymalizatora można skompilować nasz model (listing 2, linia 14). Do tego celu służy metoda `compile`, która oprócz optymalizatora przyjmuje między innymi funkcję, za pomocą której ma być wyznaczony błąd sieci (należy pamiętać, że pochodna tej funkcji będzie następnie wwykorzystywana w trakcie uczenia sieci). W tym przypadku wybrany został błąd średniokwadratowy (`mse` - *mean squared error*).

Uczenie sieci odbywa się za pomocą funkcji `fit` (listing 2, linia 17). Najważniejsze parametry, jakie przyjmuje ta funkcja to:

- `x` - *input_data* - pierwszy parametr funkcji - dane wejściowe sieci, w postaci tablicy *numpy array*. Do funkcji może być przekazany zarówno pojedyncza seria danych jak i wiele serii od razu.
- `y` - *target_data* - drugi parametr funkcji - wartości, jakie sieć powinna zwrócić dla przekazanej do niej serii danych.

2.7 Zadanie 4 - dla chętnych

Wykonaj zadanie 2.5 z wykorzystaniem frameworku Keras, podobnie jak to zostało przedstawione na listingu 2 (pamiętaj, że zestaw wag nie jest parametrem obowiązkowym - jeżeli wartości wag nie zostaną jawnie przekazane do sieci, sieć wylosuje je sama za nas).

3 Laboratorium 3

Celem laboratorium jest zapoznanie się z podstawową metodą uczenia głębokich sieci neuronowych - metodą propagacji wstecznej - ang. *Backpropagation*.

3.1 Teoria

- Andrew Trask, *Grokking Deep Learning*, rozdział 6 - *building your first deep neural network: introduction to backpropagation*
- Funkcje aktywacji
- Propagacja wsteczna

3.2 Wstęp teoretyczny

3.2.1 Funkcja aktywacji

W celu pełnego wykorzystania możliwości jakie dają warstwy ukryte, do sieci neuronowej należy wprowadzić element nieliniowości. Zostanie to zrobione za pomocą dodania do warstwy ukrytej funkcji aktywacji - w tym przypadku wykorzystana zostanie najprostsza i najpopularniejsza funkcja ReLU (ang. *Rectified Linear Unit*). Funkcję tę można przedstawić za pomocą równania 16.

$$y = \max(0, x) \tag{16}$$

Oznacza to, że funkcja zwraca wartość 0 dla liczb mniejszych od 0, a dla pozostałych zwraca przekazaną liczbę. Funkcję aktywacji stosuje się po wykonaniu obliczeń dla danego neuronu (suma iloczynów wartości i wag im odpowiadającym dla danego neuronu). W trakcie aktualizacji wag potrzebna będzie pochodna funkcji aktywacji, dla funkcji ReLU pochodna jest przedstawiona wzorem 17:

$$y(x) = \begin{cases} 0 & \text{if} \quad x <= 0 \\ 1 & \text{if} \quad x > 0 \end{cases} \tag{17}$$

Drugą funkcją aktywacji, którą można wykorzystać jest funkcja sigmoidalna - funkcja wyrażona jest następującym wzorem:

$$sigmoid(x) = \frac{1}{1 + exp(-x)}$$

gdzie:

- x - argument funkcji,
- *exp()* - funkcja wykładnicza o liczbie Eulera w podstawie.

Pochodną funkcji można obliczyć ze wzoru:

$$sigmoid\_deriv(x) = x(1 - x)$$

gdzie:

- x - argument funkcji.

3.2.2 Aktualizacja wag

Błąd otrzymany na wyjściu z sieci wpływa na wartości wszystkich wag, które brały udział w jego generowaniu. W przypadku warstwy ostatniej błąd jest wyliczany analogicznie do wzoru 22, tylko, że zamiast danych wejściowych całej sieci do skalowania brane są dane wejściowe z warstwy poprzedniej (wzór **??**).

$$layer\_output\_delta = 2 * \frac{1}{N}(\textbf{prediction} - \textbf{goal}) \tag{18}$$

$$layer\_output\_weight\_delta = layer\_output\_delta \otimes hidden\_layer \tag{19}$$

W przypadku warstw ukrytych wartości wag uaktualniane są w analogicznym stopniu do tego, w jakim przyczyniły się do powstania błędu. Dzieje się to poprzez skalowanie błędu warstwy ostatniej przez wartości wag warstwy następnej (wzór 20).

$$layer\_hidden\_delta = output\_weight^T * layer\_output\_delta \tag{20}$$

Nie wszystkie wagi w warstwach ukrytych będą jednak uaktualniane (dodanie nieliniowości). Kolejnym krokiem więc jest określenie, które warstwy powinny zostać zaktualizowane. W tym celu należy zastosować pochodną funkcji aktywacji na neuronach w warstwie ukrytej i odpowiednio przeskalować wartości wag warstwy ukrytej (wzór 21).

$$layer\_hidden\_delta = layer\_hidden\_delta \circ relu\_deriv(hidden\_layer) \tag{21}$$

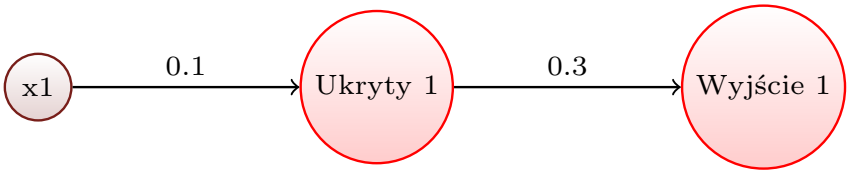
Na koniec wartość błędu należy przeskalować analogicznie do wartości błędu warstwy wyjściowej, czyli pomnożyć przez wartości neuronów warstwy wcześniejszej.

$$layer\_hidden\_weight\_delta = layer\_hidden\_delta \otimes \mathbf{x} \tag{22}$$

3.3 Wstęp praktyczny

3.3.1 Przykład 1

Rozważmy następujący przykład sieci przedstawiony na rysunku 5.



Rysunek 5: Głęboka sieć neuronowa składająca się z jednego wejścia, jednego neuronu w warstwie ukrytej oraz jednego neuronu w warstwie wyjściowej.

Przeprowadźmy I etap procesu nauczania tej sieci, dla następujących danych:

- waga neuronu w warstwiej ukrytej - $Wh = 0.1$,
- waga neuronu w warstwiej wyjściowej - $Wy = 0.3$,
- wejście - $x = 0.5$,
- oczekiwany wynik - $y = 0.1$,
- współczynnik uczenia - $alpha = 0.01$.

Pierwszym krokiem jest obliczenie wartości neuronów w poszczególnych warstwach (analogicznie do przykładu z laboratorium 2):

$$layer\_hidden\_1 = Wh * x = 0.1 * 0.5 = 0.05,$$

$$layer\_output = Wy * layer\_hidden\_1 = 0.3 * 0.05 = 0.015.$$

Na neuronach warstwy ukrytej powinna być jeszcze zastosowana funkcja aktywacji, ale dla uproszczenia obliczeń w tym przykładzie została ona pominięta. Następnym krokiem jest obliczenie różnicy pomiędzy wyjściem wyznaczonym przez sieć, a tym oczekiwanym, czyli błędu ostatniej warstwy sieci:

$$layer\_output\_delta = 2 * \frac{1}{N} * (layer\_output - expected\_output) = 2 * 1 * (0.015 - 0.1) = -0.17,$$

W przypadku warstw ukrytych nie mamy do dyspozycji oczekiwanego wyniku sieci, jedyne z czego możemy skorzystać to błąd następnej warstwy i na jego podstawie ustalić udział poszczególnych neuronów w generowaniu błędnej odpowiedzi. W tym celu należy przeskalować błąd warstwy następnej mnożąc go przez jej wagi:

$$layer\_hidden\_1\_delta = W_y * layer\_output\_delta = 0.3 * -0.17 = -0.051.$$

Wartość ta powinna jeszcze zostać pomnożona przez wynik pochodnej funkcji aktywacji zastosowanej na neuronach warstwy pierwszej, ale dla uproszczenia obliczeń w tym przykładzie zostało to pominięte. Zanim nastąpi aktualizacja wartości wag w poszczególnych warstwach należy wartości delt przeskalować mnożąc je przez wartość neuronu wejściowego dla każdej z warstw.

$$layer\_output\_weight\_delta = layer\_output\_delta * layer\_hidden\_1 = -0.17 * 0.05 = -0.0085,$$

$$layer\_hidden\_1\_weight\_delta = layer\_hidden\_1\_delta * x = -0.051 * 0.5 = -0.0255.$$

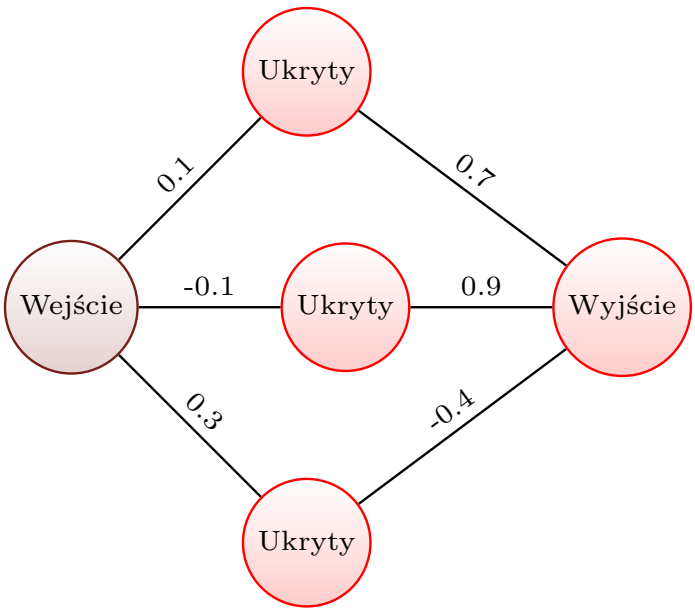
Na koniec można zaktualizować wartości wag poszczególnych neuronów:

$$Wh = Wh - alpha * layer\_hidden\_1\_weight\_delta = 0.1 - 0.01 * -0.0255 = 0.100255,$$

$$Wy = Wy - alpha * layer\_output\_weight\_delta = 0.3 - 0.01 * -0.0085 = 0.300085,$$

3.3.2 Przykład 2

Kolejny przykład będzie dotyczył sieci z trzema neuronami w warstwie ukrytej - rysunek 6.



Rysunek 6: Głęboka sieć neuronowa składająca się z jednego wejścia, trzech neuronów w warstwie ukrytej oraz jednego neuronu w warstwie wyjściowej.

Przeprowadźmy I etap procesu nauczania tej sieci, dla następujących danych:

- wagi neuronów w warstwiej ukrytej - $Wh = \begin{bmatrix} 0.1 \\ -0.1 \\ 0.3 \end{bmatrix}$,

- wagi neuronu w warstwiej wyjściowej - $Wy = \begin{bmatrix}0.7 & 0.9 & -0.4\end{bmatrix}$,
- wejście - $x = 0.5$,
- oczekiwany wynik - $y = 0.1$,
- współczynnik uczenia - $alpha = 0.01$.

W warstwie ukrytej zastosowana zostanie funkcja aktywacji - ReLU. Pierwszym krokiem jest obliczenie wartości neuronów w poszczególnych warstwach:

$$layer\_hidden\_1 = Wh * x = \begin{bmatrix}0.1\\-0.1\\0.3\end{bmatrix} * 0.5 = \begin{bmatrix}0.05\\-0.05\\0.15\end{bmatrix},$$

Na neuronach warstwy ukrytej należy zastosować jeszcze funkcję aktywacji.

$$layer\_hidden\_1 = relu(layer\_hidden\_1) = relu(\begin{bmatrix}0.05\\-0.05\\0.15\end{bmatrix}) = \begin{bmatrix}0.05\\0\\0.15\end{bmatrix}.$$

Teraz można przejść do obliczania wartości neuronów w warstwie drugiej.

$$layer\_output = Wy * layer\_hidden\_1 = \begin{bmatrix}0.7 & 0.9 & -0.4\end{bmatrix} * \begin{bmatrix}0.05\\0\\0.15\end{bmatrix} = -0.025.$$

Następnym krokiem jest obliczenie różnicy pomiędzy wyjściem wyznaczonym przez sieć, a tym oczekiwanym, czyli błędu ostatniej warstwy sieci:

$$\begin{aligned} layer\_output\_delta &= 2 * \frac{1}{N} * (layer\_output - expected\_output) = \\ &= 2 * 1 * (-0.025 - 0.1) = 2 * -0.125 = -0.25, \end{aligned}$$

W przypadku warstw ukrytych nie mamy do dyspozycji oczekiwanego wyniku sieci, jedyne z czego możemy skorzystać to błąd następnej warstwy i na jego podstawie ustalić udział poszczególnych neuronów w generowaniu błędnej odpowiedzi. W tym celu należy przeskalować błąd warstwy następnej mnożąc go przez jej wagi:

$$layer\_hidden\_1\_delta = W_y^T * layer\_output\_delta = \begin{bmatrix}0.7 & 0.9 & -0.4\end{bmatrix}^T * -0.25 = \begin{bmatrix}0.7\\0.9\\-0.4\end{bmatrix} * -0.25 = \begin{bmatrix}-0.175\\-0.225\\0.1\end{bmatrix}.$$

Wartości te powinny jeszcze zostać pomnożone przez wynik pochodnej funkcji aktywacji zastosowanej na neuronach warstwy pierwszej:

$$\begin{aligned} layer\_hidden\_1\_delta &= layer\_hidden\_1\_delta \circ relu\_deriv(layer\_hidden\_1) = \\ &= \begin{bmatrix}-0.175\\-0.225\\0.1\end{bmatrix} \circ relu\_deriv(\begin{bmatrix}0.05\\0\\0.15\end{bmatrix}) = \begin{bmatrix}-0.175\\-0.225\\0.1\end{bmatrix} \circ \begin{bmatrix}1\\0\\1\end{bmatrix} = \begin{bmatrix}-0.175\\0\\0.1\end{bmatrix}. \end{aligned}$$

Zanim nastąpi aktualizacja wartości wag w poszczególnych warstwach należy wartości delt przeskalować mnożąc je przez wartość neuronu wejściowego:

$$\begin{aligned} layer\_output\_weight\_delta &= layer\_output\_delta \otimes layer\_hidden\_1 = \\ &= -0.25 \otimes \begin{bmatrix}0.05\\0\\0.15\end{bmatrix} = \begin{bmatrix}-0.0125 & 0 & -0.0375\end{bmatrix}, \end{aligned}$$

$$layer\_hidden\_1\_weight\_delta = layer\_hidden\_1\_delta \otimes x = \begin{bmatrix} -0.175 \\ 0 \\ 0.1 \end{bmatrix} \otimes 0.5 = \begin{bmatrix} -0.0875 \\ 0 \\ 0.05 \end{bmatrix}.$$

Na koniec można zaktualizować wartości wag poszczególnych neuronów:

$$Wh = Wh - alpha * layer\_hidden\_1\_weight\_delta = \begin{bmatrix} 0.1 \\ -0.1 \\ 0.3 \end{bmatrix} - 0.01 * \begin{bmatrix} -0.0875 & 0 & 0.05 \end{bmatrix} = \begin{bmatrix} 0.100875 \\ -0.1 \\ 0.2995 \end{bmatrix},$$

$$Wy = Wy - alpha * layer\_output\_weight\_delta = \begin{bmatrix} 0.7 & 0.9 & -0.4 \end{bmatrix} - 0.01 * \begin{bmatrix} -0.0125 & 0 & -0.0375 \end{bmatrix} =$$
$$= \begin{bmatrix} 0.700125 & 0.9 & -0.399625 \end{bmatrix},$$

3.3.3 Przykład 3

Kolejny przykład będzie dotyczył sieci z pięcioma wejściami, trzema neuronami w warstwie ukrytej, trzema neuronami w warstwie wyjściowej - rysunek 7.
Przeprowadźmy I etap procesu nauczania tej sieci, dla następujących danych:

- dane wejściowe:

$$x = \begin{array}{c} \begin{matrix} \text{Seria 1} & \text{Seria 2} & \text{Seria 3} & \text{Seria 4} \end{matrix} \\ \begin{bmatrix} 0.5 & 0.1 & 0.2 & 0.8 \\ 0.75 & 0.3 & 0.1 & 0.9 \\ 0.1 & 0.7 & 0.6 & 0.2 \end{bmatrix} \end{array} \begin{matrix} \text{Wejście 1} \\ \text{Wejście 2} \\ \text{Wejście 3} \end{matrix}$$

- oczekiwane wyniki:

$$y = \begin{array}{c} \begin{matrix} \text{Seria 1} & \text{Seria 2} & \text{Seria 3} & \text{Seria 4} \end{matrix} \\ \begin{bmatrix} 0.1 & 0.5 & 0.1 & 0.7 \\ 1.0 & 0.2 & 0.3 & 0.6 \\ 0.1 & -0.5 & 0.2 & 0.2 \end{bmatrix} \end{array} \begin{matrix} \text{Wyjście 1} \\ \text{Wyjście 2} \\ \text{Wyjście 3} \end{matrix}$$

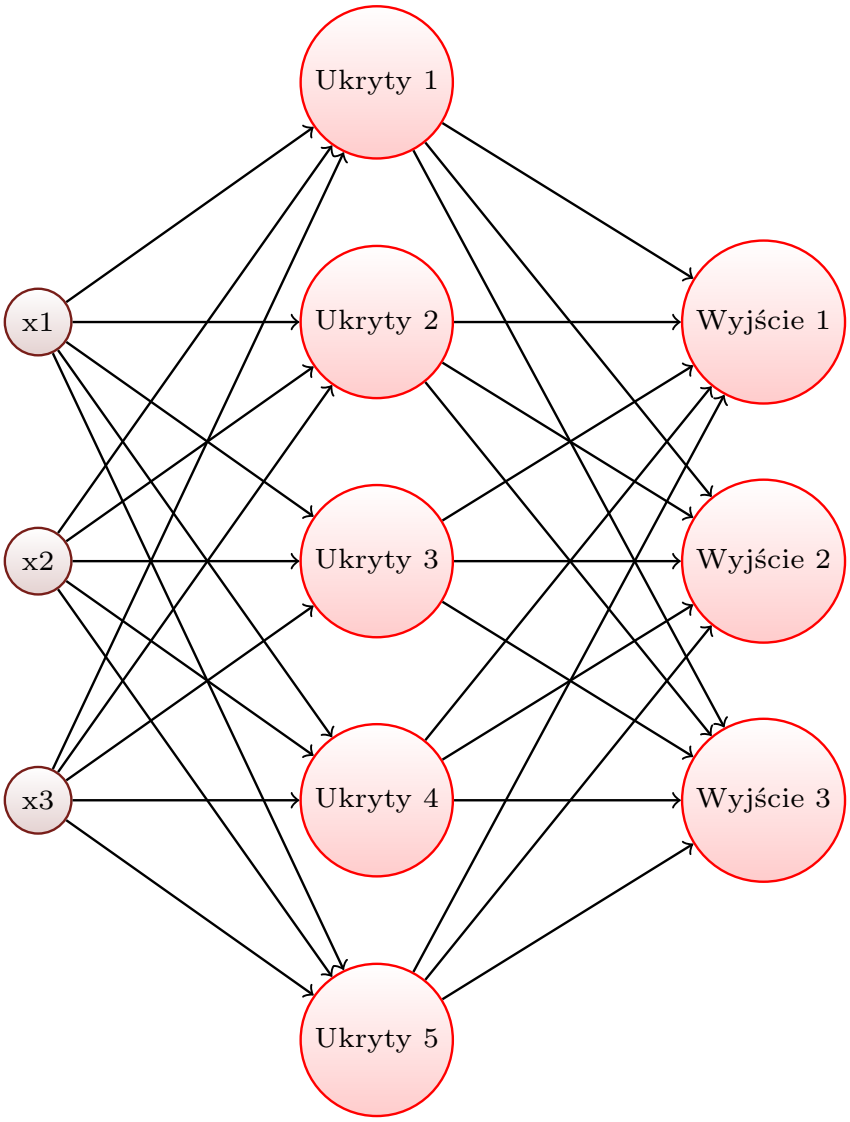
- współczynnik uczenia - 0.01,

- wagi neuronów w warstwie ukrytej:

$$Wh = \begin{array}{c} \begin{matrix} \text{Wejście 1} & \text{Wejście 2} & \text{Wejście 3} \end{matrix} \\ \begin{bmatrix} 0.1 & 0.1 & -0.3 \\ 0.1 & 0.2 & 0.0 \\ 0.0 & 0.7 & 0.1 \\ 0.2 & 0.4 & 0.0 \\ -0.3 & 0.5 & 0.1 \end{bmatrix} \end{array} \begin{matrix} \text{Ukryty 1} \\ \text{Ukryty 2} \\ \text{Ukryty 3} \\ \text{Ukryty 4} \\ \text{Ukryty 5} \end{matrix} \tag{23}$$

- wagi neuronu w warstwie wyjściowej:

$$Wy = \begin{array}{c} \begin{matrix} \text{Ukryty 1} & \text{Ukryty 2} & \text{Ukryty 3} & \text{Ukryty 4} & \text{Ukryty 5} \end{matrix} \\ \begin{bmatrix} 0.7 & 0.9 & -0.4 & 0.8 & 0.1 \\ 0.8 & 0.5 & 0.3 & 0.1 & 0.0 \\ -0.3 & 0.9 & 0.3 & 0.1 & -0.2 \end{bmatrix} \end{array} \begin{matrix} \text{Wyjście 1} \\ \text{Wyjście 2} \\ \text{Wyjście 3} \end{matrix} \tag{24}$$



Rysunek 7: Prosta sieć neuronowa składająca się z dwóch warstw: warstwy ukrytej składającej się z 5 neuronów oraz warstwy wyjściowej składającej się z 3 neuronów.

Pierwszym krokiem jest obliczenie wartości neuronów w poszczególnych warstwach:

$$layer\_hidden\_1 = Wh * x = \begin{bmatrix} 0.1 & 0.1 & -0.3 \\ 0.1 & 0.2 & 0.0 \\ 0.0 & 0.7 & 0.1 \\ 0.2 & 0.4 & 0.0 \\ -0.3 & 0.5 & 0.1 \end{bmatrix} * \begin{bmatrix} 0.5 \\ 0.75 \\ 0.1 \end{bmatrix} = \begin{bmatrix} 0.095 \\ 0.2 \\ 0.535 \\ 0.4 \\ 0.235 \end{bmatrix},$$

Na neuronach warstwy ukrytej należy zastosować jeszcze funkcję aktywacji:

$$layer\_hidden\_1 = relu(layer\_hidden\_1) = relu(\begin{bmatrix} 0.095 \\ 0.2 \\ 0.535 \\ 0.4 \\ 0.235 \end{bmatrix}) = \begin{bmatrix} 0.095 \\ 0.2 \\ 0.535 \\ 0.4 \\ 0.235 \end{bmatrix}.$$

Teraz można przejść do obliczania wartości neuronów w warstwie drugiej:

$$layer\_output = Wy * layer\_hidden\_1 = \begin{bmatrix} 0.7 & 0.9 & -0.4 & 0.8 & 0.1 \\ 0.8 & 0.5 & 0.3 & 0.1 & 0.0 \\ -0.3 & 0.9 & 0.3 & 0.1 & -0.2 \end{bmatrix} * \begin{bmatrix} 0.095 \\ 0.2 \\ 0.535 \\ 0.4 \\ 0.235 \end{bmatrix} = \begin{bmatrix} 0.376 \\ 0.3765 \\ 0.305 \end{bmatrix}.$$

Następnym krokiem jest obliczenie różnicy pomiędzy wyjściem wyznaczonym przez sieć, a tym oczekiwanym, czyli błędu ostatniej warstwy sieci:

$$\begin{aligned} layer\_output\_delta &= 2 * \frac{1}{N} * (layer\_output - expected\_output) = \\ &= 2 * \frac{1}{3} * (\begin{bmatrix} 0.376 \\ 0.3765 \\ 0.305 \end{bmatrix} - \begin{bmatrix} 0.1 \\ 1.0 \\ 0.1 \end{bmatrix}) = 0.667 * \begin{bmatrix} 0.276 \\ -0,6235 \\ 0.205 \end{bmatrix} = \begin{bmatrix} 0.184 \\ -0.415667 \\ 0.136667 \end{bmatrix}, \end{aligned}$$

W przypadku warstw ukrytych nie mamy do dyspozycji oczekiwanego wyniku sieci, jedyne z czego możemy skorzystać to błąd następnej warstwy i na jego podstawie ustalić udział poszczególnych neuronów w generowaniu błędnej odpowiedzi. W tym celu należy przeskalować błąd warstwy następnej mnożąc go przez jej wagi:

$$\begin{aligned} layer\_hidden\_1\_delta = W_y^T * layer\_output\_delta = \begin{bmatrix} 0.7 & 0.9 & -0.4 & 0.8 & 0.1 \\ 0.8 & 0.5 & 0.3 & 0.1 & 0.0 \\ -0.3 & 0.9 & 0.3 & 0.1 & -0.2 \end{bmatrix}^T * \begin{bmatrix} 0.184 \\ -0.415667 \\ 0.136667 \end{bmatrix} = \\ = \begin{bmatrix} 0.7 & 0.8 & -0.3 \\ 0.9 & 0.5 & 0.9 \\ -0.4 & 0.3 & 0.3 \\ 0.8 & 0.1 & 0.1 \\ 0.1 & 0 & -0.2 \end{bmatrix} * \begin{bmatrix} 0.184 \\ -0.415667 \\ 0.136667 \end{bmatrix} = \begin{bmatrix} -0.244733 \\ 0.0807667 \\ -0.1573 \\ 0.1193 \\ -0.00893333 \end{bmatrix}. \end{aligned}$$

Wartości te powinny jeszcze zostać pomnożone przez wynik pochodnej funkcji aktywacji zastosowanej na neuronach warstwy pierwszej:

$$\begin{aligned} layer\_hidden\_1\_delta &= layer\_hidden\_1\_delta \circ relu\_deriv(layer\_hidden\_1) = \\ &= \begin{bmatrix} -0.244733 \\ 0.0807667 \\ -0.1573 \\ 0.1193 \\ -0.00893333 \end{bmatrix} \circ relu\_deriv(\begin{bmatrix} 0.095 \\ 0.2 \\ 0.535 \\ 0.4 \\ 0.235 \end{bmatrix}) = \begin{bmatrix} -0.244733 \\ 0.0807667 \\ -0.1573 \\ 0.1193 \\ -0.00893333 \end{bmatrix} \circ \begin{bmatrix} 1 \\ 1 \\ 1 \\ 1 \\ 1 \end{bmatrix} = \begin{bmatrix} -0.244733 \\ 0.0807667 \\ -0.1573 \\ 0.1193 \\ -0.00893333 \end{bmatrix}. \end{aligned}$$

Zanim nastąpi aktualizacja wartości wag w poszczególnych warstwach należy wartości delt przeskalować mnożąc je przez wartość neuronów wejściowych.

$$\begin{aligned} layer\_output\_weight\_delta = layer\_output\_delta * layer\_hidden\_1^T = \begin{bmatrix} 0.184 \\ -0.415667 \\ 0.136667 \end{bmatrix} * \begin{bmatrix} 0.095 \\ 0.2 \\ 0.535 \\ 0.4 \\ 0.235 \end{bmatrix}^T = \\ = \begin{bmatrix} 0.184 \\ -0.415667 \\ 0.136667 \end{bmatrix} * \begin{bmatrix} 0.095 & 0.2 & 0.535 & 0.4 & 0.235 \end{bmatrix} = \\ = \begin{bmatrix} 0.01748 & 0.0368 & 0.09844 & 0.0736 & 0.04324 \\ -0.0394883 & -0.0831333 & -0.222382 & -0.166267 & -0.0976817 \\ 0.0129833 & 0.0273333 & 0.0731167 & 0.0546667 & 0.0321167 \end{bmatrix}, \end{aligned}$$

$$layer\_hidden\_1\_weight\_delta = layer\_hidden\_1\_delta * x^T = \begin{bmatrix} -0.244733 \\ 0.0807667 \\ -0.1573 \\ 0.1193 \\ -0.00893333 \end{bmatrix} * \begin{bmatrix} 0.5 \\ 0.75 \\ 0.1 \end{bmatrix}^T =$$

$$= \begin{bmatrix} -0.244733 \\ 0.0807667 \\ -0.1573 \\ 0.1193 \\ -0.00893333 \end{bmatrix} * \begin{bmatrix} 0.5 & 0.75 & 0.1 \end{bmatrix} =$$

$$= \begin{bmatrix} -0.122367 & -0.18355 & -0.0244733 \\ 0.0403833 & 0.060575 & 0.00807667 \\ -0.07865 & -0.117975 & -0.01573 \\ 0.05965 & 0.089475 & 0.01193 \\ -0.00446666 & -0.0067 & -0.000893333 \end{bmatrix}.$$

Na koniec można zaktualizować wartości wag poszczególnych neuronów:

$$Wh = Wh - alpha * layer\_hidden\_1\_weight\_delta =$$

$$= \begin{bmatrix} 0.1 & 0.1 & -0.3 \\ 0.1 & 0.2 & 0.0 \\ 0.0 & 0.7 & 0.1 \\ 0.2 & 0.4 & 0.0 \\ -0.3 & 0.5 & 0.1 \end{bmatrix} - 0.01 * \begin{bmatrix} -0.122367 & -0.18355 & -0.0244733 \\ 0.0403833 & 0.060575 & 0.00807667 \\ -0.07865 & -0.117975 & -0.01573 \\ 0.05965 & 0.089475 & 0.01193 \\ -0.00446666 & -0.0067 & -0.000893333 \end{bmatrix} =$$

$$= \begin{bmatrix} 0.101224 & 0.101836 & -0.299755 \\ 0.0995962 & 0.199394 & -0.0000081 \\ 0.0007865 & 0.70118 & 0.100157 \\ 0.199404 & 0.399105 & -0.0001193 \\ -0.299955 & 0.500067 & 0.100009 \end{bmatrix},$$

$$Wy = Wy - alpha * layer\_output\_weight\_delta =$$

$$= \begin{bmatrix} 0.7 & 0.9 & -0.4 & 0.8 & 0.1 \\ 0.8 & 0.5 & 0.3 & 0.1 & 0.0 \\ -0.3 & 0.9 & 0.3 & 0.1 & -0.2 \end{bmatrix} - 0.01 * \begin{bmatrix} 0.01748 & 0.0368 & 0.09844 & 0.0736 & 0.04324 \\ -0.03949 & -0.08313 & -0.22238 & -0.16627 & -0.09768 \\ 0.01298 & 0.02733 & 0.07312 & 0.05467 & 0.03212 \end{bmatrix} =$$

$$= \begin{bmatrix} 0.699825 & 0.899632 & -0.400984 & 0.799264 & 0.0995676 \\ 0.800395 & 0.500831 & 0.302224 & 0.101663 & 0.000976817 \\ -0.30013 & 0.899727 & 0.299269 & 0.0994533 & -0.200321 \end{bmatrix},$$

3.4 Zadanie 1

Rozbuduj sieć z zadania 1.5 z laboratorium 1 o funkcję aktywacji ReLU w warstwie ukrytej. Przetestuj działanie sieci dla następujących danych wejściowych:

- dane wejściowe:

$$x = \begin{array}{c} \begin{matrix} \text{Seria 1} & \text{Seria 2} & \text{Seria 3} & \text{Seria 4} \end{matrix} \\ \begin{bmatrix} 0.5 & 0.1 & 0.2 & 0.8 \\ 0.75 & 0.3 & 0.1 & 0.9 \\ 0.1 & 0.7 & 0.6 & 0.2 \end{bmatrix} \begin{matrix} \text{Wejście 1} \\ \text{Wejście 2} \\ \text{Wejście 3} \end{matrix} \end{array}$$

- wagi neuronów w warstwie ukrytej:

$$Wh = \begin{array}{c} \begin{matrix} \text{Wejście 1} & \text{Wejście 2} & \text{Wejście 3} \end{matrix} \\ \begin{bmatrix} 0.1 & 0.1 & -0.3 \\ 0.1 & 0.2 & 0.0 \\ 0.0 & 0.7 & 0.1 \\ 0.2 & 0.4 & 0.0 \\ -0.3 & 0.5 & 0.1 \end{bmatrix} \end{array} \begin{matrix} \text{Ukryty 1} \\ \text{Ukryty 2} \\ \text{Ukryty 3} \\ \text{Ukryty 4} \\ \text{Ukryty 5} \end{matrix} \tag{25}$$

- wagi neuronu w warstwie wyjściowej:

$$Wy = \begin{array}{c} \begin{matrix} \text{Ukryty 1} & \text{Ukryty 2} & \text{Ukryty 3} & \text{Ukryty 4} & \text{Ukryty 5} \end{matrix} \\ \begin{bmatrix} 0.7 & 0.9 & -0.4 & 0.8 & 0.1 \\ 0.8 & 0.5 & 0.3 & 0.1 & 0.0 \\ -0.3 & 0.9 & 0.3 & 0.1 & -0.2 \end{bmatrix} \end{array} \begin{matrix} \text{Wyjście 1} \\ \text{Wyjście 2} \\ \text{Wyjście 3} \end{matrix} \tag{26}$$

Oczekiwane wynik (na razie tylko obliczanie wyjścia sieci, bez aktualizacji wag):

- wyjście sieci po 1 serii - $output = \begin{bmatrix} 0.376 \\ 0.3765 \\ 0.305 \end{bmatrix}$,
- wyjście sieci po 2 serii - $output = \begin{bmatrix} 0.082 \\ 0.133 \\ 0.123 \end{bmatrix}$,
- wyjście sieci po 3 serii - $output = \begin{bmatrix} 0.053 \\ 0.067 \\ 0.073 \end{bmatrix}$,
- wyjście sieci po 4 serii - $output = \begin{bmatrix} 0.49 \\ 0.465 \\ 0.402 \end{bmatrix}$.

3.5 Zadanie 2

Zmodyfikuj sieć z zadania 3.4 w taki sposób, aby w każdej iteracji następowała aktualizacja wag każdego z neuronów w warstwie ukrytej i wyjściowej, tak jak zostało to pokazane w przykładzie 3. Przetestuj działanie sieci dla następujących danych wejściowych:

- dane wejściowe:

$$x = \begin{array}{c} \begin{matrix} \text{Seria 1} & \text{Seria 2} & \text{Seria 3} & \text{Seria 4} \end{matrix} \\ \begin{bmatrix} 0.5 & 0.1 & 0.2 & 0.8 \\ 0.75 & 0.3 & 0.1 & 0.9 \\ 0.1 & 0.7 & 0.6 & 0.2 \end{bmatrix} \end{array} \begin{matrix} \text{Wejście 1} \\ \text{Wejście 2} \\ \text{Wejście 3} \end{matrix}$$

- oczekiwane wyniki:

$$y = \begin{array}{c} \begin{matrix} \text{Seria 1} & \text{Seria 2} & \text{Seria 3} & \text{Seria 4} \end{matrix} \\ \begin{bmatrix} 0.1 & 0.5 & 0.1 & 0.7 \\ 1.0 & 0.2 & 0.3 & 0.6 \\ 0.1 & -0.5 & 0.2 & 0.2 \end{bmatrix} \end{array} \begin{matrix} \text{Wyjście 1} \\ \text{Wyjście 2} \\ \text{Wyjście 3} \end{matrix}$$

- współczynnik uczenia - 0.01,

- wagi neuronów w warstwie ukrytej:

$$Wh = \begin{array}{c} \begin{matrix} \text{Wejście 1} & \text{Wejście 2} & \text{Wejście 3} \end{matrix} \\ \begin{bmatrix} 0.1 & 0.1 & -0.3 \\ 0.1 & 0.2 & 0.0 \\ 0.0 & 0.7 & 0.1 \\ 0.2 & 0.4 & 0.0 \\ -0.3 & 0.5 & 0.1 \end{bmatrix} \end{array} \begin{matrix} \text{Ukryty 1} \\ \text{Ukryty 2} \\ \text{Ukryty 3} \\ \text{Ukryty 4} \\ \text{Ukryty 5} \end{matrix} \tag{27}$$

- wagi neuronu w warstwie wyjściowej:

$$Wy = \begin{array}{c} \begin{matrix} \text{Ukryty 1} & \text{Ukryty 2} & \text{Ukryty 3} & \text{Ukryty 4} & \text{Ukryty 5} \end{matrix} \\ \begin{bmatrix} 0.7 & 0.9 & -0.4 & 0.8 & 0.1 \\ 0.8 & 0.5 & 0.3 & 0.1 & 0.0 \\ -0.3 & 0.9 & 0.3 & 0.1 & -0.2 \end{bmatrix} \end{array} \begin{matrix} \text{Wyjście 1} \\ \text{Wyjście 2} \\ \text{Wyjście 3} \end{matrix} \tag{28}$$

Oczekiwane wynik w 1 epoce:

- wyjście sieci po 1 serii - $output = \begin{bmatrix} 0.376 \\ 0.3765 \\ 0.305 \end{bmatrix}$,
- wyjście sieci po 2 serii - $output = \begin{bmatrix} 0.0807193 \\ 0.134082 \\ 0.122503 \end{bmatrix}$,
- wyjście sieci po 3 serii - $output = \begin{bmatrix} 0.053154 \\ 0.0670564 \\ 0.07174 \end{bmatrix}$,
- wyjście sieci po 4 serii - $output = \begin{bmatrix} 0.490493 \\ 0.470195 \\ 0.398576 \end{bmatrix}$.

Oczekiwane wynik w 50 epoce:

- wyjście sieci po 1 serii - $output = \begin{bmatrix} 0.442756 \\ 0.565494 \\ 0.15731 \end{bmatrix}$,
- wyjście sieci po 2 serii - $output = \begin{bmatrix} 0.140418 \\ 0.186105 \\ 0.0638947 \end{bmatrix}$,
- wyjście sieci po 3 serii - $output = \begin{bmatrix} 0.105594 \\ 0.0936588 \\ 0.0450485 \end{bmatrix}$,
- wyjście sieci po 4 serii - $output = \begin{bmatrix} 0.584537 \\ 0.711427 \\ 0.216373 \end{bmatrix}$.

3.6 Zadanie 3

Zmodyfikuj sieć z zadania 4 laboratorium 1 tak, aby obsługiwała jeszcze funkcję aktywacji dla każdej z warstw (pamiętaj, że nie zawsze każda warstwa będzie musiała ją posiadać). Klasa powinna udostępniać następujące funkcje:

- `add_layer(n, [weight_min_value, weight_max_value, activation_function])` - funkcja dodaje warstwę n neuronów do sieci, opcjonalnie może przyjmować zakres wartości, z którego będą losowane wagi oraz informację o wybranej funkcji aktywacji (w postaci wskaźnika na funkcję, stringa lub w inny sposób),
- `fit(input, expected_output)` - funkcja aktualizuje wagi sieci na podstawie przekazanych do niej danych wejściowych (*input*) oraz oczekiwanej dla nich odpowiedzi sieci (*expected_output*),
- `save_weights(file_name)` - funkcja zapisuje wagi do pliku o nazwie (*file_name*), zwróć uwagę, żeby oprócz samych wartości wag zapisać w pliku strukturę sieci,
- `load_weights(file_name)` - funkcja odczytuje wagi z pliku o nazwie (*file_name*).

Zbuduj sieć składającą się z trzech warstw:

- warstwa wejściowa - 784 neurony,
- warstwa ukryta - 40 neuronów,
- warstwa wyjściowa - 10 neuronów.

Przetestuj działania tej sieci na bazie danych MNIST[2]. Sieć powinna się uczyć, bazując na części treningowej (pliki zaczynające się od train), a testować na części testowej (pliki zaczynające się od t10k).

Przykładowe logi z uczenia sieci wraz z parametrami dostępne są pod linkiem: `http://pduch.iis.p.lodz.pl/PSI/Lab3_3.zip`

3.6.1 Opis bazy danych MNIST

Baza danych składa się z 4 plików:

- `train-labels-idx1-ubyte` - plik z etykietami do danych treningowych (60 tysięcy etykiet),
- `train-images-idx3-ubyte` - plik z obrazami części treningowej (60 tysięcy obrazów, każdy o wymiarach 28x28 pikseli),
- `t10k-labels-idx1-ubyte` - plik z etykietami do danych testowych (10 tysięcy etykiet),
- `t10k-images-idx3-ubyte` - plik z obrazami części testowej (10 tysięcy obrazów).

Dane w plikach przechowujących etykiety zapisane są w następujący sposób:

- 4 bajty - Magic Number (2049), konieczna jest zmiana kolejności bajtów po odczytaniu z pliku,
- 4 bajty - liczba danych zapisanych w pliku (odpowiednio 60000 dla pliku train i 10000 dla pliku test), konieczna jest zmiana kolejności bajtów po odczytaniu z pliku,
- 1 bajt - etykieta pierwszego obrazu,
- 1 bajt - etykieta drugiego obrazu ...

Dane w plikach przechowujących obrazy zapisane są w następujący sposób:

- 4 bajty - Magic Number (2051), konieczna jest zmiana kolejności bajtów po odczytaniu z pliku,
- 4 bajty - liczba obrazów zapisanych w pliku (odpowiednio 60000 dla pliku train i 10000 dla pliku test), konieczna jest zmiana kolejności bajtów po odczytaniu z pliku,
- 4 bajty - liczba wierszy w obrazie (28), konieczna jest zmiana kolejności bajtów po odczytaniu z pliku,
- 4 bajty - liczba kolumn w obrazie (28), konieczna jest zmiana kolejności bajtów po odczytaniu z pliku,
- 28 * 28 bajtów - wartości kolejnych pikseli w pierwszym obrazie: każdy piksel może przyjąć wartość z zakresu 0 - 255, zanim dane zostaną przekazane do sieci, należy przeskalować wartości na zakres 0 - 1.

[2] http://pduch.iis.p.lodz.pl/PSI/MNIST_ORG.zip

- 28 * 28 bajtów - wartości kolejnych pikseli w drugim obrazie ...

Z pliku zawierającego obrazy (*images* w nazwie) należy wczytać do wektora zawierającego 784 elementy wartości poszczególnych pikseli, a następnie z pliku etykiet etykietę do wczytanego obrazu (cyfrę z zakresu 0 - 9). Dane w obu plikach (obrazy i etykiety) zapisane są w takiej samej kolejności. W celach testowych można wyświetlić wczytany obraz w konsoli.

3.7 Zadanie 4

Wykorzystaj przygotowaną przez siebie sieć głęboką do rozpoznawania kolorów. Doświadczalnie dobierz odpowiednią liczbę neuronów w warstwie ukrytej i pamiętaj o tym, żeby zastosować w niej funkcję aktywacji ReLU. Celem sieci będzie nauczenie się rozpoznawania kolorów (czerwony (1), zielony (2), niebieski (3), żółty (4)) na podstawie 3 wartości składowych koloru (RGB). Poszczególne neurony w warstwie wyjściowej oznaczają odpowiedź sieci dla danego koloru, przyjmij, że ostatecznym kolorem, jaki sieć zwróciła jest neuron z najwyższą wartością. W procesie uczenia przyjmij, że oczekiwane wartości na wyjściu to 0 dla neuronów nieodpowiadających danemu kolorowi oraz 1 na wyjściu odpowiadającemu kolorowi.

Na przykład wejście $\begin{bmatrix}0.91 & 0.82 & 0.05\end{bmatrix}$ odpowiada kolorowi o ID 4 (żółty). Więc wektor oczekiwanej odpowiedzi sieci dla takich danych wejściowych powinien wyglądać następująco $\begin{bmatrix}0 & 0 & 0 & 1\end{bmatrix}$.

- Zbiór treningowy - zbiór, na którym następuje proces uczenia sieci, dostępny jest pod linkiem: `http://pduch.iis.p.lodz.pl/PSI/training_colors.txt`.
- Zbiór testowy - zbiór, dla którego sprawdzana jest poprawność działania sieci, dostępny jest pod linkiem: `http://pduch.iis.p.lodz.pl/PSI/test_colors.txt`.

W pierwszym etapie program uczy sieć neuronową na przykładzie zbioru treningowego (zwróć uwagę, że może być konieczność przeprowadzenia kilku / kilkudziesięciu iteracji). Po zakończeniu procesu uczenia sprawdzana jest skuteczność sieci na podstawie zbioru testowego.

Porównaj wyniki otrzymywane dla tej sieci z wynikami otrzymanymi w zadaniu 4 z laboratorium 2. Która sieć potrzebuje więcej czasu, żeby nauczyć się rozpoznawania kolorów?

3.8 Jak to robią profesjonaliści (3)

Zadanie 3.5 można rozwiązać za pomocą frameworka keras (listing 3).

```
import numpy as np
from keras.models import Sequential
from keras.layers import Dense
from keras.optimizers import SGD

model = Sequential()
weights = [np.array([[0.1, 0.1, 0.0, 0.2, -0.3],
                     [0.1, 0.2, 0.7, 0.4, 0.5],
                     [-0.3, 0.0, 0.1, 0.0, 0.1]])]
weights2 = [np.array([[0.7, 0.8, -0.3], [0.9, 0.5, 0.9], [-0.4, 0.3, 0.3], [0.8, 0.1, 0.1], [0.1, 0.0, -0.2]])]
model.add(Dense(units=5, input_dim=3, weights=weights, use_bias = False, activation="relu"))
model.add(Dense(3, weights=weights2, use_bias = False))

inputs = [np.array([[0.5, 0.75, 0.1]]), np.array([[0.1, 0.3, 0.7]]), np.array([[0.2, 0.1, 0.6]]), np.array([[0.8, 0.9, 0.2]])]
expected_outputs = [np.array([[0.1, 1.0, 0.1]]), np.array([[0.5, 0.2, -0.5]]), np.array([[0.1, 0.3, 0.2]]), np.array([[0.7, 0.6, 0.2]])]
opt = SGD(lr=0.01)
model.compile(opt, loss='mse')
for _ in range(50):
    for i in range(len(inputs)):
        print(model.predict(inputs[i]))
        model.fit(inputs[i], expected_outputs[i])
```

Listing 3: Implementacja rozwiązania zadania 2.4 w kerasie (Keras, wersja 2.2.4)

W stosunku do listingu 2 pojawiają się dwie różnice, pierwsza, oczywista, to dodanie warstwy ukrytej. Druga, to pojawienie się funkcji aktywacji w tej warstwie (w tym przypadku wybrana została funkcja *relu*).

3.9 Zadanie 5 - dla chętnych

Wykonaj zadanie 3.7 z wykorzystaniem frameworku Keras (pamiętaj, że zestaw wag nie jest parametrem obowiązkowym - jeżeli wartości wag nie zostaną jawnie przekazane do sieci, sieć wylosuje je sama za nas).

4 Laboratorium 4

Celem laboratorium jest zapoznanie się z dodatkowymi elementami głębokich sieci neuronowych, takimi jak: *Dropout*, *batch*, inne funkcje aktywacji (*sigmoid*, *tanh*, *softmax*).

4.1 Teoria

- Andrew Trask, *Grokking Deep Learning*, rozdział 8 - *introduction to regularization and batching.*
- Andrew Trask, *Grokking Deep Learning*, rozdział 9 - *activation functions.*

4.2 Wstęp teoretyczny

4.2.1 Dropout

Nadmierne dopasowanie (ang. *overfitting*) w fazie treningowej jest poważnym problemem w głębokich sieciach neuronowych o dużej liczbie parametrów. Jedną z metod stosowaną w celu przeciwdziałania temu zjawisku jest *Dropout*. Polega on na przecięciu pewnych losowych połączeń pomiędzy neuronami w trakcie treningu. Ponieważ połączenia, które są przecinane, dobierane są losowo dla każdej serii danych treningowych, można powiedzieć, że w ten sposób uczonych jest wiele różnych sieci neuronowych, które później są uśrednione. Bazuje to na następującej koncepcji: duże, nieregularne sieci neuronowe w trakcie uczenia będą nadmiernie dopasowywały się do szumu, jednakże jest mało prawdopodobne, że do tego samego. A to wynika z tego, że wagi sieci neuronowych inicjalizowane są w sposób losowy. Przecięcie połączeń pomiędzy neuronami odbywa się za pomocą wyzerowania wartości tych neuronów, należy jednak pamiętać o tym, że wartości pozostałych neuronów muszą zostać odpowiednio przeskalowane. Jest to spowodowane tym, że każda kolejna warstwa wyznacza sumę ważoną z elementów warstwy poprzedniej, skoro część neuronów została usunięta (ich wartości zostały wyzerowane), to wynik sumy ważonej będzie zakłócony (w tym przypadku mniejszy o połowę). W celu zapobiegnięcia temu, należy pomnożyć wartości pozostałych neuronów przez $1/procent\_wlaczonych\_neuronow$

4.2.2 Batch Gradient Descent

Jest to metoda mająca na celu przyspieszenie czasu treningu sieci. Do tej pory aktualizacja wag sieci następowała po każdej serii treningowej. W przypadku podejścia Batch Gradient Descent aktualizacja następuje po analizie pełnej serii danych. Inną wersją takiego podejścia jest Mini-Batch Gradient Descent, kiedy to aktualizacja wag odbywa się po treningu n-serii, gdzie n jest z przedziału $< 1, \text{liczba elementow zbioru treningowego} >$. Istnieją różne sposoby implementacji algorytmu:

1. Sumując wartości, o które powinny być uaktualniane wagi, z n serii, a następnie aktualizując wszystkie wagi sieci o tę skumulowaną sumę po zakończeniu każdej z n serii. Jednakże nie jest to podejście optymalne pod względem obliczeniowym.
2. Wykonując wszystkie obliczenia dla n serii naraz:
   (a) Utworzenie tablicy *batch_input* zawierającej dane z n serii, każda seria w osobnej kolumnie:

$$batch\_input = \begin{bmatrix} input1_1 & input2_1 & input3_1 & \dots & inputm_1 \\ input1_2 & input2_2 & input3_2 & \dots & inputm_3 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ input1_n & input2_n & input3_n & \dots & inputm_n \end{bmatrix}.$$

   (b) Obliczenie wartości neuronów w warstwie pierwszej poprzez pomnożenie macierzy wejściowej (*batch_input*) przez macierz wag z warstwy pierwszej. Wartości w macierzy wag ułożone są w następujący sposób: wagi dla poszczególnych neuronów znajdują się w kolumnach - pierwsza kolumna, m wag pierwszego neuronu, druga kolumna, m - wag dla drugiego neuronu, itd.

$$layer\_hidden\_1 = Wh * \mathbf{x}$$

   macierz *layer_hidden_1* będzie miała n wierszy (liczba serii danych w jednym batchu) i w kolumn (liczba neuronów w pierwszej warstwie).

(c) Zastosowanie funkcji aktywacji na wartościach neuronów w warstwie ukrytej:

$$layer\_hidden\_1 = activation\_function(layer\_hidden\_1).$$

(d) Zastosowanie metody *dropout* na neuronach z warstwy ukrytej (dla każdego wiersza losowana jest osobna maska *dropout*):

$$layer\_hidden\_1 = dropout\_mask(layer\_hidden\_1).$$

(e) Wyznaczenie wartości neuronów w warstwie drugiej:

$$layer\_output = Wy * layer\_hidden\_1.$$

(f) Obliczenie delty dla wag z warstwy drugiej - pochodna funkcji błędu podzielona przez rozmiar batcha:

$$layer\_output\_delta = (2 * \frac{1}{N} * (layer\_output - expected\_output))/batch\_size.$$

(g) Obliczenie delty dla wag z warstwy pierwszej:

$$layer\_hidden\_1\_delta = W_y^T * layer\_output\_delta.$$

(h) Pomnożenie delty dla warstwy pierwszej przez wynik działania pochodnej funkcji aktywacji dla neuronów warstwy pierwszej:

$$layer\_hidden\_1\_delta = layer\_hidden\_1\_delta \circ activation\_deriv(layer\_hidden\_1).$$

(i) Wygaszenie wag dla tych neuronów, które zostały wyłączone:

$$layer\_hidden\_1\_delta = dropout\_mask(layer\_hidden\_1\_delta).$$

(j) Przeskalowanie wartości delt przez wartości neuronów w warstwie ppoprzedniej:

$$layer\_output\_weight\_delta = layer\_output\_delta * layer\_hidden\_1^T,$$

$$layer\_hidden\_1\_weight\_delta = layer\_hidden\_1\_delta * \mathbf{x}^T.$$

(k) Aktualizacja wag we wszystkich warstwach:

$$Wh = Wh - alpha * layer\_hidden\_1\_weight\_delta,$$

$$Wy = Wy - alpha * layer\_output\_weight\_delta.$$

4.2.3 Funkcje aktywacji

Funkcja ReLU, z której korzystaliśmy do tej pory, jest jedną z najpopularniejszych i często używanych funkcji w głębokich sieciach neuronowych. Jedną z jej niewątpliwych zalet jest niski koszt obliczeniowy. Oprócz niej w sieciach często stosuje się również następujące funkcje aktywacji:

- funkcja sigmoidalna - funkcja wyrażona jest następującym wzorem:

$$sigmoid(x) = \frac{1}{1 + exp(-x)}$$

gdzie:

  - x - argument funkcji,
  - *exp()* - funkcja wykładnicza o liczbie Eulera w podstawie.

Pochodną funkcji można obliczyć ze wzoru:

$$sigmoid\_deriv(x) = x(1 - x)$$

gdzie:

  - x - argument funkcji.

- funkcja tangensa hiperbolicznego - funkcja wyrażona jest następującym wzorem:

$$tanh(x) = \frac{exp(x) - exp(-x)}{exp(x) + exp(-x)}$$

gdzie:

  - x - argument funkcji,
  - *exp()* - funkcja wykładnicza o liczbie Eulera w podstawie.

Pochodną funkcji można obliczyć ze wzoru:

$$tanh\_deriv(x) = 1 - x^2$$

gdzie:

  - x - argument funkcji.

- funkcja softmax - jest to specjalna funkcja wykładnicza, wykorzystywana wyłącznie w warstwie ostatniej. Zadaniem tej funkcji jest normalizacja danych wyjściowych tak, aby ich suma była równa 1. Dzięki temu, dane wyjściowe mogą być traktowane jako prawdopodobieństwo przynależności danego sygnału wejściowego do poszczególnych klas. Funkcja jest wyrażona wzorem:

$$softmax(x) = \frac{exp(x)}{sum(exp(x))}$$

gdzie:

  - x - argument funkcji - w tym przypadku x jest wektorem,
  - *exp()* - funkcja wykładnicza o liczbie Eulera w podstawie,
  - *sum()* - funkcja obliczająca sumę wszystkich elementów wektora wejściowego.

4.2.4 Przykład 1

Kolejny przykład będzie dotyczył sieci z trzema wejściami, pięcioma neuronami w warstwie ukrytej, trzema neuronami w warstwie wyjściowej - rysunek 8.
Przeprowadźmy I etap procesu nauczania tej sieci, dla następujących danych:

- dane wejściowe:

$$x = \begin{array}{c} \begin{matrix} \text{Seria 1} & \text{Seria 2} & \text{Seria 3} & \text{Seria 4} \end{matrix} \\ \begin{bmatrix} 0.5 & 0.1 & 0.2 & 0.8 \\ 0.75 & 0.3 & 0.1 & 0.9 \\ 0.1 & 0.7 & 0.6 & 0.2 \end{bmatrix} \end{array} \begin{matrix} \text{Wejście 1} \\ \text{Wejście 2} \\ \text{Wejście 3} \end{matrix}$$

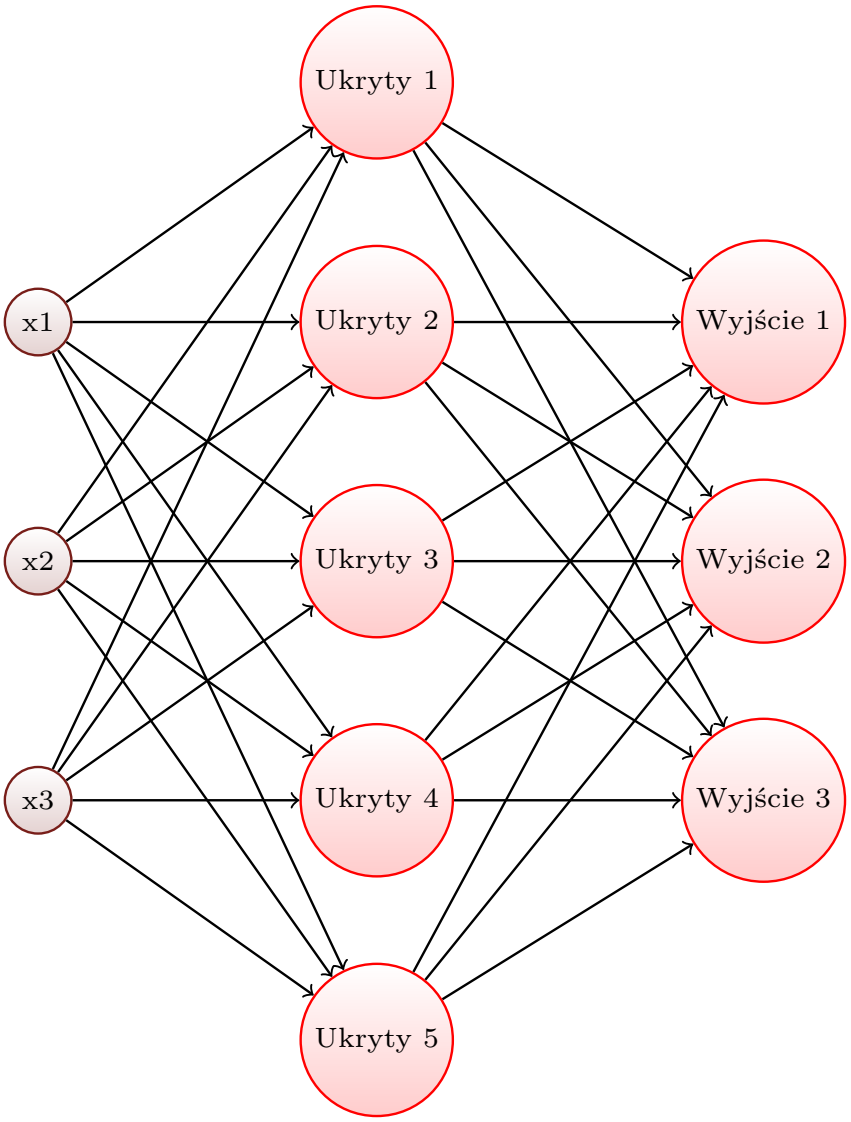
- oczekiwane wyniki:

$$y = \begin{array}{c} \begin{matrix} \text{Seria 1} & \text{Seria 2} & \text{Seria 3} & \text{Seria 4} \end{matrix} \\ \begin{bmatrix} 0.1 & 0.5 & 0.1 & 0.7 \\ 1.0 & 0.2 & 0.3 & 0.6 \\ 0.1 & -0.5 & 0.2 & 0.2 \end{bmatrix} \end{array} \begin{matrix} \text{Wyjście 1} \\ \text{Wyjście 2} \\ \text{Wyjście 3} \end{matrix}$$

- współczynnik uczenia - 0.01,

- wagi neuronów w warstwie ukrytej:

$$Wh = \begin{array}{c} \begin{matrix} \text{Wejście 1} & \text{Wejście 2} & \text{Wejście 3} \end{matrix} \\ \begin{bmatrix} 0.1 & 0.1 & -0.3 \\ 0.1 & 0.2 & 0.0 \\ 0.0 & 0.7 & 0.1 \\ 0.2 & 0.4 & 0.0 \\ -0.3 & 0.5 & 0.1 \end{bmatrix} \end{array} \begin{matrix} \text{Ukryty 1} \\ \text{Ukryty 2} \\ \text{Ukryty 3} \\ \text{Ukryty 4} \\ \text{Ukryty 5} \end{matrix} \tag{29}$$



Rysunek 8: Prosta sieć neuronowa składająca się z dwóch warstw: warstwy ukrytej składającej się z 5 neuronów oraz warstwy wyjściowej składającej się z 3 neuronów.

- wagi neuronu w warstwie wyjściowej:

$$Wy = \begin{array}{c} \begin{matrix} \text{Ukryty 1} & \text{Ukryty 2} & \text{Ukryty 3} & \text{Ukryty 4} & \text{Ukryty 5} \end{matrix} \\ \begin{bmatrix} 0.7 & 0.9 & -0.4 & 0.8 & 0.1 \\ 0.8 & 0.5 & 0.3 & 0.1 & 0.0 \\ -0.3 & 0.9 & 0.3 & 0.1 & -0.2 \end{bmatrix} \end{array} \begin{matrix} \text{Wyjście 1} \\ \text{Wyjście 2} \\ \text{Wyjście 3} \end{matrix} \tag{30}$$

Pierwszym krokiem jest obliczenie wartości neuronów w poszczególnych warstwach:

$$layer\_hidden\_1 = Wh * \mathbf{x} = \begin{bmatrix} 0.1 & 0.1 & -0.3 \\ 0.1 & 0.2 & 0.0 \\ 0.0 & 0.7 & 0.1 \\ 0.2 & 0.4 & 0.0 \\ -0.3 & 0.5 & 0.1 \end{bmatrix} * \begin{bmatrix} 0.5 & 0.1 & 0.2 & 0.8 \\ 0.75 & 0.3 & 0.1 & 0.9 \\ 0.1 & 0.7 & 0.6 & 0.2 \end{bmatrix} = \begin{bmatrix} 0.095 & -0.17 & -0.15 & 0.11 \\ 0.2 & 0.07 & 0.04 & 0.26 \\ 0.535 & 0.28 & 0.13 & 0.65 \\ 0.4 & 0.14 & 0.08 & 0.52 \\ 0.235 & 0.19 & 0.05 & 0.23 \end{bmatrix},$$

Na neuronach warstwy ukrytej należy zastosować jeszcze funkcję aktywacji:

$$layer\_hidden\_1 = relu(layer\_hidden\_1) = relu(\begin{bmatrix} 0.095 & -0.17 & -0.15 & 0.11 \\ 0.2 & 0.07 & 0.04 & 0.26 \\ 0.535 & 0.28 & 0.13 & 0.65 \\ 0.4 & 0.14 & 0.08 & 0.52 \\ 0.235 & 0.19 & 0.05 & 0.23 \end{bmatrix}) = \begin{bmatrix} 0.095 & 0 & 0 & 0.11 \\ 0.2 & 0.07 & 0.04 & 0.26 \\ 0.535 & 0.28 & 0.13 & 0.65 \\ 0.4 & 0.14 & 0.08 & 0.52 \\ 0.235 & 0.19 & 0.05 & 0.23 \end{bmatrix}.$$

Teraz można przejść do obliczania wartości neuronów w warstwie drugiej:

$$layer\_output = Wy * layer\_hidden\_1 = \begin{bmatrix} 0.7 & 0.9 & -0.4 & 0.8 & 0.1 \\ 0.8 & 0.5 & 0.3 & 0.1 & 0.0 \\ -0.3 & 0.9 & 0.3 & 0.1 & -0.2 \end{bmatrix} * \begin{bmatrix} 0.095 & 0 & 0 & 0.11 \\ 0.2 & 0.07 & 0.04 & 0.26 \\ 0.535 & 0.28 & 0.13 & 0.65 \\ 0.4 & 0.14 & 0.08 & 0.52 \\ 0.235 & 0.19 & 0.05 & 0.23 \end{bmatrix} =$$

$$= \begin{bmatrix} 0.376 & 0.082 & 0.053 & 0.49 \\ 0.3765 & 0.133 & 0.067 & 0.465 \\ 0.305 & 0.123 & 0.073 & 0.402 \end{bmatrix}.$$

Następnym krokiem jest obliczenie różnicy pomiędzy wyjściem wyznaczonym przez sieć, a tym oczekiwanym, czyli pochodnej funkcji błędu:

$$layer\_output\_delta = (2 * \frac{1}{N} * (layer\_output - expected\_output))/batch\_size =$$

$$= (2 * \frac{1}{3} * (\begin{bmatrix} 0.376 & 0.082 & 0.053 & 0.49 \\ 0.3765 & 0.133 & 0.067 & 0.465 \\ 0.305 & 0.123 & 0.073 & 0.402 \end{bmatrix} - \begin{bmatrix} 0.1 & 0.5 & 0.1 & 0.7 \\ 1 & 0.2 & 0.3 & 0.6 \\ 0.1 & -0.5 & 0.2 & 0.2 \end{bmatrix}))/4 =$$

$$= 0.667 * \begin{bmatrix} 0.276 & -0.418 & -0.047 & -0.21 \\ -0.6235 & -0.067 & -0.233 & -0.135 \\ 0.205 & 0.623 & -0.127 & 0.202 \end{bmatrix} * 0.25 =$$

$$= \begin{bmatrix} 0.046 & -0.0697 & -0.0078 & -0.035 \\ -0.1039 & -0.01117 & -0.0388 & -0.0225 \\ 0.03417 & 0.1038 & -0.02117 & 0.03367 \end{bmatrix},$$

W przypadku warstw ukrytych nie mamy do dyspozycji oczekiwanego wyniku sieci, jedyne z czego możemy skorzystać to błąd następnej warstwy i na jego podstawie ustalić udział poszczególnych neuronów w generowaniu błędnej odpowiedzi. W tym celu należy przeskalować błąd warstwy następnej mnożąc go przez jej wagi:

$$layer\_hidden\_1\_delta = W_y^T * layer\_output\_delta =$$

$$= \begin{bmatrix} 0.7 & 0.9 & -0.4 & 0.8 & 0.1 \\ 0.8 & 0.5 & 0.3 & 0.1 & 0.0 \\ -0.3 & 0.9 & 0.3 & 0.1 & -0.2 \end{bmatrix}^T * \begin{bmatrix} 0.046 & -0.0697 & -0.0078 & -0.035 \\ -0.1039 & -0.01117 & -0.0388 & -0.0225 \\ 0.03417 & 0.1038 & -0.02117 & 0.03367 \end{bmatrix} =$$

$$= \begin{bmatrix} 0.7 & 0.8 & -0.3 \\ 0.9 & 0.5 & 0.9 \\ -0.4 & 0.3 & 0.3 \\ 0.8 & 0.1 & 0.1 \\ 0.1 & 0 & -0.2 \end{bmatrix} * \begin{bmatrix} 0.046 & -0.0697 & -0.0078 & -0.035 \\ -0.1039 & -0.01117 & -0.0388 & -0.0225 \\ 0.03417 & 0.1038 & -0.02117 & 0.03367 \end{bmatrix} =$$

$$= \begin{bmatrix} -0.06118 & -0.08885 & -0.0302 & -0.0526 \\ 0.02019 & 0.02517 & -0.04552 & -0.01245 \\ -0.0393 & 0.0557 & -0.01487 & 0.01735 \\ 0.0298 & -0.04647 & -0.01227 & -0.02688 \\ -0.0022 & -0.0277 & 0.00345 & -0.0102 \end{bmatrix}.$$

Wartości te powinny jeszcze zostać pomnożone przez wynik pochodnej funkcji aktywacji zastosowanej na neuronach warstwy pierwszej:

$$layer\_hidden\_1\_delta = layer\_hidden\_1\_delta \circ relu\_deriv(layer\_hidden\_1) =$$

$$= \begin{bmatrix} -0.06118 & -0.08885 & -0.0302 & -0.0526 \\ 0.02019 & 0.02517 & -0.04552 & -0.01245 \\ -0.0393 & 0.0557 & -0.01487 & 0.01735 \\ 0.0298 & -0.04647 & -0.01227 & -0.02688 \\ -0.0022 & -0.0277 & 0.00345 & -0.0102 \end{bmatrix} \circ relu\_deriv(\begin{bmatrix} 0.095 & 0 & 0 & 0.11 \\ 0.2 & 0.07 & 0.04 & 0.26 \\ 0.535 & 0.28 & 0.13 & 0.65 \\ 0.4 & 0.14 & 0.08 & 0.52 \\ 0.235 & 0.19 & 0.05 & 0.23 \end{bmatrix}) =$$

$$= \begin{bmatrix} -0.06118 & -0.08885 & -0.0302 & -0.0526 \\ 0.02019 & 0.02517 & -0.04552 & -0.01245 \\ -0.0393 & 0.0557 & -0.01487 & 0.01735 \\ 0.0298 & -0.04647 & -0.01227 & -0.02688 \\ -0.0022 & -0.0277 & 0.00345 & -0.0102 \end{bmatrix} \circ \begin{bmatrix} 1 & 0 & 0 & 1 \\ 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \end{bmatrix} =$$

$$= \begin{bmatrix} -0.06118 & 0 & 0 & -0.0526 \\ 0.02019 & 0.02517 & -0.04552 & -0.01245 \\ -0.0393 & 0.0557 & -0.01487 & 0.01735 \\ 0.0298 & -0.04647 & -0.01227 & -0.02688 \\ -0.0022 & -0.0277 & 0.00345 & -0.0102 \end{bmatrix}.$$

Zanim nastąpi aktualizacja wartości wag w poszczególnych warstwach należy wartości delt przeskalować mnożąc je przez wartość neuronów wejściowych.

$$layer\_output\_weight\_delta = layer\_output\_delta * layer\_hidden\_1^T =$$

$$= \begin{bmatrix} 0.046 & -0.0697 & -0.0078 & -0.035 \\ -0.1039 & -0.01117 & -0.0388 & -0.0225 \\ 0.03417 & 0.1038 & -0.02117 & 0.03367 \end{bmatrix} * \begin{bmatrix} 0.095 & 0 & 0 & 0.11 \\ 0.2 & 0.07 & 0.04 & 0.26 \\ 0.535 & 0.28 & 0.13 & 0.65 \\ 0.4 & 0.14 & 0.08 & 0.52 \\ 0.235 & 0.19 & 0.05 & 0.23 \end{bmatrix}^T =$$

$$= \begin{bmatrix} 0.046 & -0.0697 & -0.0078 & -0.035 \\ -0.1039 & -0.01117 & -0.0388 & -0.0225 \\ 0.03417 & 0.1038 & -0.02117 & 0.03367 \end{bmatrix} * \begin{bmatrix} 0.095 & 0.2 & 0.535 & 0.4 & 0.235 \\ 0 & 0.07 & 0.28 & 0.14 & 0.19 \\ 0 & 0.04 & 0.13 & 0.08 & 0.05 \\ 0.11 & 0.26 & 0.65 & 0.52 & 0.23 \end{bmatrix} =$$

$$= \begin{bmatrix} 0.00052 & -0.00509 & -0.018665 & -0.01018 & -0.0108683 \\ -0.0123471 & -0.0289683 & -0.0783954 & -0.0579367 & -0.0336588 \\ 0.00694917 & 0.0220083 & 0.0664842 & 0.0440167 & 0.0344425 \end{bmatrix},$$

$$layer\_hidden\_1\_weight\_delta = layer\_hidden\_1\_delta * \mathbf{x}^T =$$

$$= \begin{bmatrix} -0.06118 & 0 & 0 & -0.0526 \\ 0.02019 & 0.02517 & -0.04552 & -0.01245 \\ -0.0393 & 0.0557 & -0.01487 & 0.01735 \\ 0.0298 & -0.04647 & -0.01227 & -0.02688 \\ -0.0022 & -0.0277 & 0.00345 & -0.0102 \end{bmatrix} * \begin{bmatrix} 0.5 & 0.1 & 0.2 & 0.8 \\ 0.75 & 0.3 & 0.1 & 0.9 \\ 0.1 & 0.7 & 0.6 & 0.2 \end{bmatrix}^T =$$

$$= \begin{bmatrix} -0.06118 & 0 & 0 & -0.0526 \\ 0.02019 & 0.02517 & -0.04552 & -0.01245 \\ -0.0393 & 0.0557 & -0.01487 & 0.01735 \\ 0.0298 & -0.04647 & -0.01227 & -0.02688 \\ -0.0022 & -0.0277 & 0.00345 & -0.0102 \end{bmatrix} * \begin{bmatrix} 0.5 & 0.75 & 0.1 \\ 0.1 & 0.3 & 0.7 \\ 0.2 & 0.1 & 0.6 \\ 0.8 & 0.9 & 0.2 \end{bmatrix}^T =$$

$$= \begin{bmatrix} -0.0726717 & -0.0932275 & -0.0166383 \\ -0.00645084 & 0.00693708 & -0.0101642 \\ -0.00318917 & 0.00133458 & 0.0295842 \\ -0.0136942 & -0.0169929 & -0.0422808 \\ -0.0113867 & -0.01886 & -0.0196133 \end{bmatrix}.$$

Na koniec można zaktualizować wartości wag poszczególnych neuronów:

$$Wh = Wh - alpha * layer\_hidden\_1\_weight\_delta =$$
$$= \begin{bmatrix} 0.1 & 0.1 & -0.3 \\ 0.1 & 0.2 & 0.0 \\ 0.0 & 0.7 & 0.1 \\ 0.2 & 0.4 & 0.0 \\ -0.3 & 0.5 & 0.1 \end{bmatrix} - 0.01 * \begin{bmatrix} -0.0726717 & -0.0932275 & -0.0166383 \\ -0.00645084 & 0.00693708 & -0.0101642 \\ -0.00318917 & 0.00133458 & 0.0295842 \\ -0.0136942 & -0.0169929 & -0.0422808 \\ -0.0113867 & -0.01886 & -0.0196133 \end{bmatrix} =$$
$$= \begin{bmatrix} 0.100727 & 0.100932 & -0.299834 \\ 0.100065 & 0.199931 & 0.000101642 \\ 3.18917e-05 & 0.699987 & 0.0997042 \\ 0.200137 & 0.40017 & 0.000422808 \\ -0.299886 & 0.500189 & 0.100196 \end{bmatrix},$$

$$Wy = Wy - alpha * layer\_output\_weight\_delta =$$
$$= \begin{bmatrix} 0.7 & 0.9 & -0.4 & 0.8 & 0.1 \\ 0.8 & 0.5 & 0.3 & 0.1 & 0.0 \\ -0.3 & 0.9 & 0.3 & 0.1 & -0.2 \end{bmatrix} - 0.01 * \begin{bmatrix} 0.00052 & -0.00509 & -0.018665 & -0.01018 & -0.01087 \\ -0.01235 & -0.02897 & -0.0784 & -0.05794 & -0.03366 \\ 0.00695 & 0.02201 & 0.06648 & 0.04402 & 0.03444 \end{bmatrix} =$$
$$= \begin{bmatrix} 0.699995 & 0.900051 & -0.399813 & 0.800102 & 0.100109 \\ 0.800123 & 0.50029 & 0.300784 & 0.100579 & 0.000336587 \\ -0.30007 & 0.89978 & 0.299335 & 0.0995598 & -0.200344 \end{bmatrix},$$

4.3 Zadanie 1

Dodaj do warstwy ukrytej sieci z zadania 3 laboratorium 3 metodę Dropout. Będzie to polegało na wyzerowaniu dla każdej serii danych treningowych pewnej liczby losowych neuronów z warstwy ukrytej, w tym przypadku przyjmij, że będzie to 50% neuronów. Na początku treningu dla poszczególnych serii należy utworzyć wektor, który będzie miał tyle samo elementów ile jest neuronów w warstwie ukrytej, a następnie wypełnić go losowo wartościami 0 i 1 w taki sposób, aby było ich mniej więcej po równo. Następnie należy pomnożyć wartości neuronów warstwy ukrytej przez ten wektor, spowoduje to wyzerowanie losowych neuronów. Dodatkowo, pozostałe wartości neuronów muszą zostać pomnożone przez 2. Jest to spowodowane tym, że każda kolejna warstwa wyznacza sumę ważoną z elementów warstwy poprzedniej, skoro część neuronów została usunięta (ich wartości zostały wyzerowane), to wynik sumy ważonej będzie zakłócony (w tym przypadku mniejszy o połowę). W celu zapobiegnięcia temu, należy pomnożyć wartości pozostałych neuronów przez $1/procent\_wlaczonych\_neuronow$ (w tym przypadku będzie to 1/0.5, czyli 2). Należy również pamiętać o wyzerowaniu błędu dla tych neuronów z warstwy ukrytej, których wartości zostały wyzerowane. Przetestuj działanie tej sieci na bazie danych MNIST, dla następujących parametrów:

- 40 neuronów w warstwie ukrytej, 1000 obrazów w zbiorze treningowym, 10000 obrazów w zbiorze testowym, współczynnik uczenia 0.005, wagi z zakresu $< -0.1, 0.1 >$, 350 iteracji,
- 100 neuronów w warstwie ukrytej, 10000 obrazów w zbiorze treningowym, 10000 obrazów w zbiorze testowym, współczynnik uczenia 0.005, wagi z zakresu $< -0.1, 0.1 >$, 350 iteracji,
- 100 neuronów w warstwie ukrytej, 60000 obrazów w zbiorze treningowym, 10000 obrazów w zbiorze testowym, współczynnik uczenia 0.005, wagi z zakresu $< -0.1, 0.1 >$, 350 iteracji.

Przykładowe logi z uczenia sieci wraz z parametrami dostępne są pod linkiem: `http://pduch.iis.p.lodz.pl/PSI/Lab4_1.zip`

4.4 Zadanie 2

Zaimplementuj metodę *Mini-Batch Gradient Descent* w klasie z zadania 1 laboratorium 4. Wykonaj te same obliczenia, jak w zadaniu pierwszym przyjmując rozmiar batchu jako 100 i współczynnik uczenia 0.1.

Przykładowe logi z uczenia sieci wraz z parametrami dostępne są pod linkiem: `http://pduch.iis.p.lodz.pl/PSI/Lab4_2.zip`

4.5 Zadanie 3

Rozbuduj program z poprzedniego zadania o dodatkowe funkcje aktywacji: sigmoidalną, tangensa hiperbolicznego oraz softmax. Przetestuj działanie sieci neuronowej, składającej się:

- ze 100 neuronów w warstwie ukrytej z warstwą aktywacji w postaci tangensa hiperbolicznego (ze względu na charakterystykę tej funkcji aktywacji konieczna jest zmiana zakresu wag dla warstwy ukrytej - teraz należy losować wagi z zakresu $< -0.01, 0.01 >$),
- warstwy wyjściowej z funkcją aktywacji softmax, dodatkowo deltę ostatniej warstwy podziel przez wielkość batcha (liczbę serii w batchu), spowoduje to możliwość znacznego zwiększenia wartości współczynnika uczącego,
- rozmiarze batcha równym 100,
- współczynniku uczenia 0.02.

Na początek przetestuj działanie sieci dla 1000 obrazów wejściowych. Następnie tak zmodyfikuj parametry sieci (głównie współczynnik uczenia), aby sieć była w stanie osiągnąć lepszy wynik przy nauce dla 10000 i 60000 obrazów wejściowych.

4.6 Jak to robią profesjonaliści (4)

Przykład 4.2.4 można rozwiązać za pomocą frameworka keras (listing 4).

```
1  import numpy as np
2  from keras.models import Sequential
3  from keras.layers import Dense
4  from keras.optimizers import SGD
5
6  model = Sequential()
7  weights = [np.array([[0.1, 0.1, 0.0, 0.2, -0.3],
8                       [0.1, 0.2, 0.7, 0.4, 0.5],
9                       [-0.3, 0.0, 0.1, 0.0, 0.1]])]
10 weights2 = [np.array([[0.7, 0.8, -0.3], [0.9, 0.5, 0.9], [-0.4, 0.3, 0.3], [0.8, 0.1,
       0.1], [0.1, 0.0, -0.2]])]
11 model.add(Dense(units=5, input_dim=3, weights=weights, use_bias = False, activation="
       relu"))
12 model.add(Dense(3, weights=weights2, use_bias = False))
13
14 inputs = np.array([[0.5, 0.75, 0.1], [0.1, 0.3, 0.7], [0.2, 0.1, 0.6], [0.8, 0.9,
       0.2]])
15 expected_outputs = np.array([[0.1, 1.0, 0.1], [0.5, 0.2, -0.5], [0.1, 0.3, 0.2],
       [0.7, 0.6, 0.2]])
16 opt = SGD(lr=0.01)
17 model.compile(opt, loss='mse')
18 print(model.predict(inputs))
19 model.fit(inputs, expected_outputs)
20 print(model.get_weights())
```

Listing 4: Implementacja rozwiązania przykładu 4.2.4 w kerasie (Keras, wersja 2.2.4)

W stosunku do listingu 3 pojawiają się dwie różnice, pierwsza, to sposób przechowywania danych wejściowych i oczekiwanych, tym razem zamiast listy tablic mamy tablicę list (jest to związane z tym, że teraz będziemy uczyć sieć na batchach, a nie na pojedynczych zestawach danych). Druga, to sposób wywołania funkcji fit, do funkcji przekazujemy od razu cały zestaw danych wejściowych zamiast pojedynczych serii.

4.7 Zadanie 4 - dla chętnych

Wykonaj zadanie 4.4 z wykorzystaniem frameworku Keras.

5 Laboratorium 5

Celem laboratorium jest zapoznanie się z konwolucyjnymi sieciami neuronowymi - ang. *Convolutional Neural Networks (CNN).*

5.1 Teoria

- Andrew Trask, *Grokking Deep Learning*, rozdział 10 - *neural learning about edges and corners: intro to convolutional neural networks*
- CS231n: Convolutional Neural Networks for Visual Recognition - Stanford
- Artificial Inteligence - Leonardo Araujo dos Santos
- Source of confusion! Neural Nets vs Image Processing Convolution - Animated AI

5.2 Wstęp teoretyczny

Konwolucyjne sieci neuronowe działają na podobnej zasadzie, co sieci omawiane w ramach poprzednich zajęć. Architektura sieci CNN jest przystosowana do przetwarzania / analizy obrazów, Jedną z głównych operacji wykonywanych w sieci jest operacja splotu (ang. *convolution*) obrazu (fragmentu obrazu) z maską. W przypadku wejść wielowymiarowych, takich jak obrazy, łączenie neuronów ze wszystkimi neuronami z poprzedniej warstwy jest niepraktyczne. Zamiast tego każdy neuron łączy się tylko do lokalnego regionu obrazu wejściowego.

5.2.1 Splot / konwolucja

W przypadku przetwarzania obrazów konwolucja polega na przesuwaniu okna z wartościami filtru po całym obrazie, przemnażaniu odpowiadających sobie wartości oraz dodawaniu tych iloczynów do siebie. Przesuwanie to będzie odbywało się z lewej do prawej, a następnie z góry na dół (rys. 9).

(0, 0)	(1, 0)	(2, 0)	(3, 0)	(4, 0)
(0, 1)	(1, 1)	(2, 1)	(3, 1)	(4, 1)
(0, 2)	(1, 2)	(2, 2)	(3, 2)	(4, 2)
(0, 3)	(1, 3)	(2, 3)	(3, 3)	(4, 3)
(0, 4)	(1, 4)	(2, 4)	(3, 4)	(4, 4)

(0, 0)	(1, 0)	(2, 0)
(0, 1)	(1, 1)	(2, 1)
(0, 2)	(1, 2)	(2, 2)

Rysunek 9: Przesuwanie się maski po obrazie.

W przypadku algorytmu splotu parametrami będą:

- rozmiar maski / filtru - najczęściej mają one kształt kwadratu, do najczęściej stosowanych należą maski 3x3 (np. do wykrywania krawędzi maska Sobela), 5x5, 7x7.
- krok (ang. *stride*) - liczba pikseli, o które będzie się przesuwała maska w każdym kroku.
- *padding* - wypełnienie obszaru wokół obrazu wartościami 0, stosowany jest po do, żeby po zastosowaniu operacji splotu zachować oryginalne wymiary obrazu.

5.2.2 Warstwa konwolucyjna

Warstwa konwolucyjna w sieciach neuronowych składa się z zestawu filtrów o zadanych rozmiarach (najczęściej są to filtry kwadratowe). Każdy z filtrów składa się z losowych wag, których wartości są aktualiowane w trakcie uczenia sieci. Wynikiem działania warstwy konwolucyjnej jest zbiór map aktywacji obrazów (tyle map, ile filtrów znajdowało się w danej warstwie), który stanowi wejście do kolejnej warstwy. W ten sposób sieć nauczy się filtrów, które aktywują się, gdy zobaczą pewne charakterystyczne cechy na obrazie, takiej jak krawędź o pewnej orientacji lub plamka jakiegoś koloru na pierwszej warstwie, lub w dalszych warstwach sieci całe wzory np. plastra miodu lub koła.

5.2.3 Pooling

Celem tej operacji jest stopniowe zmniejszanie wymiarów reprezentacji obrazów w celu zmniejszenia ilości parametrów i obliczeń w sieci, a zatem także kontrolowania nadmiernego dopasowania się sieci do danych uczących. Operacja ta jest uruchamiana okresowo pomiędzy kolejnymi warstwami konwolucyjnnymi sieci. Funkcja ta działa niezależnie dla każdego kanału i zmienia jego wymiary najczęściej używając funkcji MAX (inną stosowaną funkcją jest również funkcja AVG). Najczęściej stosowana jest maska o wymiarach 2x2 z krokiem próbkowania równym 2. W ten sposób odrzuconych zostaje 75% wszystkich aktywacji warstwy konwolucyjnej. Wymiar głębokości (liczba kanałów) pozostaje niezmieniona. Przykład operacji *max pooling* zaprezentowany został na rys. 10.

3	6	7	5	3	5
6	2	9	1	2	7
0	9	3	6	0	6
2	6	1	8	7	9
2	0	2	3	7	5
9	2	2	8	9	7

6		

Rysunek 10: *Max pooling.*

Funkcja przyjmuje dwa parametry:

- F - rozmiar maski,
- S - krok próbkowania (liczba piskeli, o które maska będzie przesuwała się w każdym kroku).

Przyjmując, że na wejściu jest zbiór obrazów o wymiarach $W1$x$H1$x$D1$, to po wykonaniu operacji *max poolingu* zbiór ten będzie miał wymiary:

- $W2 = (W1 - F)/S + 1$,
- $H2 = (H1 - F)/S + 1$,
- $D2 = D1$.

W trakcie propagacji wstecznej błędu najczęściej zapamiętuje się indeksy komórek o największej wartości, żeby później uaktualnić tylko te wagi, które brały udział przy ich wyliczaniu. Ponieważ macierz po operacji jest mniejsza niż przed, to każdą wartość macierzy delt powiela się na obszare całej maski, a następnie wykonuje operację mnożenia z macierzą indeksów maksymalnych wartości w każdym oknie.

5.2.4 Prosta konwolucyjna sieć neuronowa

Wejściem sieci konwolucyjnej są obrazy. Każdy z filtrów pierwszej warstwy wykonuje na obrazie wejściowym operacje splotu, po której powstaje mapa aktywacji. Każdy filtr generuje swoją mapę aktywacji, a cały zestaw takich map stanowi wejście do kolejnej warstwy sieci. Po warstwach konwolucyjnych znajdują się warstwy *Fully Connected*, w których każdy neurony warstwy połączony jest z każdym neuronem warstwy poprzedniej. W ostatniej warstwie sieć ma tyle neuronów, ile różnych klas powinna rozpoznawać. Najczęściej w ostatniej warstwie stosuje się funkcję aktywacji *softmax*.

Rozzważmy prosty przykład konwolucyjnej sieci neuronowej składającej się z warstwy konwolucyjnej oraz warstwy *Fully Connected* (z liczbą neuronów odpowiadającą liczbą klas, które ma rozpoznawać sieć):

- Podzielenie obrazu na fragmenty o takich samych wymiarach jak filtry w pierwszej warstwie.
- Przekształcenie każdego z fragmentów w wektor oraz utworzenie macierzy z wektorów - pojedynczy fragment obrazu będzie jednym wierszem macierzy (rys. 11).

(0, 0)	(1, 0)	(2, 0)	(3, 0)	(4, 0)
(0, 1)	(1, 1)	(2, 1)	(3, 1)	(4, 1)
(0, 2)	(1, 2)	(2, 2)	(3, 2)	(4, 2)
(0, 3)	(1, 3)	(2, 3)	(3, 3)	(4, 3)
(0, 4)	(1, 4)	(2, 4)	(3, 4)	(4, 4)

(0, 0)	(1, 0)	(2, 0)
(0, 1)	(1, 1)	(2, 1)
(0, 2)	(1, 2)	(2, 2)

Rysunek 11: Przekształcenie fragmentu obrazu w wektor.

- Obliczenie wartości warstwy konwolucyjnej poprzez pomnożenie macierzy z fragmentami obrazów przez macierz filtrów - każdy filtr to jedna kolumna macierzy - wyjściowa macierz będzie miała tyle wierszy ile jest fragmentów obrazu oraz tyle kolumn ile jest filtrów.
- Zastosowanie funkcji aktywacji w warstwie konwolucyjnej.
- Jeżeli to jest ostatnia warstwa konwolucyjna przed warstwą *Fully Connected* to "spłaszczenie" wartości warstwy konwolucyjnej - przekształcenie jej w pojedynczy wektor, wiersz po wierszu.
- Obliczenie wartości sieci poprzez pomnożenie wartości wektora z ostatniej warstwy konwolucyjnej przez wagi warstwy *Fully Connected*.
- Dalsze uczenie sieci wygląda analogicznie do zwykłych sieci neuronowych, za jednym wyjątkie, przy przejściu od warstwy *Fully Connected* do warstwy konwolucyjnej należy przekształcić wektor z błędami w macierz o wymiarach filtrów danej warstwy.

5.2.5 Konwolucyjne sieci neuronowe

Sieć konwolucyjna składa się zazwyczaj z następujących elementów: CONV (warstwa konwolucyjna), POOL (operacja zmniejszania obrazu, najczęściej jest to *max pooling*), RELU (funkcja aktywacji stosowana w warstwie konwolucyjnej) oraz FC (warstwa *Fully Connected*). Najpopularniejsza forma architektury ConvNet składa się z kilku warstw CONV-RELU, po których następuje operacja POOL i wzór ten powtarza się, aż obraz zostanie zmniejszony do małego rozmiaru. Powszechnie stosowane jest przejście w pewnym momencie do warstw całkowicie połączonych. Ostatnia z nich zawiera dane

wyjściowe, takie jak wyniki klasy. Innymi słowy, najczęstsza architektura ConvNet jest zgodna ze wzorem:

$$INPUT \rightarrow [[CONV \rightarrow RELU] * N \rightarrow POOL?] * M \rightarrow [FC \rightarrow RELU] * K \rightarrow FC \tag{31}$$

gdzie:

- $*$ - oznacza powtórzenie,
- $POOL?$ - wskazuje opcjonalną warstwę *max pooling*,
- N - zazwyczaj jest z przedziału $< 0, 3 >$,
- M - jest większe bądź równe 0,
- K - zazwyczaj jest z przedziału $< 0, 3)$.

Poniżej zaprezentowane zostało kilka typowych architektur ConvNet, które można wyprowadzić ze wzoru 31:

- $INPUT \rightarrow FC$, implementuje klasyfikator liniowy. W tym przypadku $N = M = K = 0$,
- $INPUT \rightarrow CONV \rightarrow RELU \rightarrow FC$,
- $INPUT \rightarrow [CONV \rightarrow RELU \rightarrow CONV \rightarrow RELU \rightarrow POOL] * 3 \rightarrow [FC \rightarrow RELU] * 2 \rightarrow FC$ - W tym przypadku są dwie warstwy CONV ułożone przed każdą operacją POOL. Jest to ogólnie dobry pomysł w przypadku większych i głębszych sieci, ponieważ wiele warstw CONV ułożonych w stos może opracować wykrywanie bardziej złożonych cech danych wejściowych przed operacją zmniejszania rozmiaru obrazów.

5.2.6 Przykład 1

Przykład będzie dotyczył sieci z wejściem w postaci obrazu 4 x 3, dwoma filtrami w warstwie konwolucyjnej, oraz dwoma neuronami w warstwie wyjściowej. W warstwie konwolucyjnej zastosowana zostanie funkcja aktywacji ReLU.
Przeprowadźmy I etap procesu nauczania tej sieci, dla następujących danych:

- dane wejściowe:

$$input = \begin{bmatrix} 8.5 & 0.65 & 1.2 \\ 9.5 & 0.8 & 1.3 \\ 9.9 & 0.8 & 0.5 \\ 9.0 & 0.9 & 1.0 \end{bmatrix}$$

- oczekiwane wyniki:

$$expected\_output = \begin{bmatrix} 0 \\ 1 \end{bmatrix}$$

- współczynnik uczenia - 0.01,
- wagi pierwszego filtru w warstwie konwolucyjnej:

$$kernel\_1\_weights = \begin{bmatrix} 0.1 & 0.2 & -0.1 & -0.1 & 0.1 & 0.9 & 0.1 & 0.4 & 0.1 \end{bmatrix}$$

- wagi drugiego filtru w warstwie konwolucyjnej:

$$kernel\_2\_weights = \begin{bmatrix} 0.3 & 1.1 & -0.3 & 0.1 & 0.2 & 0.0 & 0.0 & 1.3 & 0.1 \end{bmatrix}$$

- wagi warstwy wyjściowej:

$$W\_y = \begin{bmatrix} 0.1 & -0.2 & 0.1 & 0.3 \\ 0.2 & 0.1 & 0.5 & -0.3 \end{bmatrix}$$

Pierwszym krokiem jest podzielenie obrazu wejściowego na fragmenty:

$$image\_sections = \begin{bmatrix} 8.5 & 9.5 & 9.9 & 0.65 & 0.8 & 0.8 & 1.2 & 1.3 & 0.5 \\ 9.5 & 9.9 & 9 & 0.8 & 0.8 & 0.9 & 1.3 & 0.5 & 1 \end{bmatrix}$$

Następnie wartości wejściowe mnożone są przez wartości filtrów w warstwie konwolucyjnej:

$$kernel\_layer = image\_sections * kernels^T = \begin{bmatrix} 8.5 & 9.5 & 9.9 & 0.65 & 0.8 & 0.8 & 1.2 & 1.3 & 0.5 \\ 9.5 & 9.9 & 9 & 0.8 & 0.8 & 0.9 & 1.3 & 0.5 & 1 \end{bmatrix} *$$
$$* \begin{bmatrix} 0.1 & 0.2 & -0.1 & -0.1 & 0.1 & 0.9 & 0.1 & 0.4 & 0.1 \\ 0.3 & 1.1 & -0.3 & 0.1 & 0.2 & 0 & 0 & 1.3 & 0.1 \end{bmatrix}^T =$$
$$= \begin{bmatrix} 8.5 & 9.5 & 9.9 & 0.65 & 0.8 & 0.8 & 1.2 & 1.3 & 0.5 \\ 9.5 & 9.9 & 9 & 0.8 & 0.8 & 0.9 & 1.3 & 0.5 & 1 \end{bmatrix} * \begin{bmatrix} 0.1 & 0.3 \\ 0.2 & 1.1 \\ -0.1 & -0.3 \\ -0.1 & 0.1 \\ 0.1 & 0.2 \\ 0.9 & 0 \\ 0.1 & 0 \\ 0.4 & 1.3 \\ 0.1 & 0.1 \end{bmatrix} =$$
$$= \begin{bmatrix} 3.185 & 11.995 \\ 3.27 & 12.03 \end{bmatrix}$$

Ponieważ wartości z warstwy konwolucyjnej będą wartościami wejściowymi ostatniej warstwy, należy macierz *kernel_layer* przekształcić w wektor i zastosować na nim funkcję aktywacji, ponieważ w rozważanym przykładzie funkcja aktywacji nie zmieni wartości macierzy zostanie ona pominięta:

$$layer\_output = W\_y * kernel\_layer\_flatten =$$
$$= \begin{bmatrix} 0.1 & -0.2 & 0.1 & 0.3 \\ 0.2 & 0.1 & 0.5 & -0.3 \end{bmatrix} * \begin{bmatrix} 3.185 \\ 11.995 \\ 3.27 \\ 12.03 \end{bmatrix} =$$
$$= \begin{bmatrix} 1.8555 \\ -0.1375 \end{bmatrix}$$

Następnym krokiem jest obliczenie różnicy pomiędzy wyjściem wyznaczonym przez sieć, a tym oczekiwanym, czyli błędu ostatniej warstwy sieci:

$$layer\_output\_delta = (2 * \frac{1}{N} * (layer\_output - expected\_output)) =$$
$$= (2 * \frac{1}{2} * (\begin{bmatrix} 1.8555 \\ -0.1375 \end{bmatrix} - \begin{bmatrix} 0 \\ 1 \end{bmatrix}) = \begin{bmatrix} 1.8555 \\ -1.1375 \end{bmatrix}$$

W przypadku warstw ukrytych nie mamy do dyspozycji oczekiwanego wyniku sieci, jedyne z czego możemy skorzystać to błąd następnej warstwy i na jego podstawie ustalić udział poszczególnych neuronów w generowaniu błędnej odpowiedzi. W tym celu należy przeskalować błąd warstwy następnej mnożąc go przez jej wagi:

$$kernel\_layer\_1\_delta = W_y^T * layer\_output\_delta =$$
$$= \begin{bmatrix} 0.1 & -0.2 & 0.1 & 0.3 \\ 0.2 & 0.1 & 0.5 & -0.3 \end{bmatrix}^T * \begin{bmatrix} 1.8555 \\ -1.1375 \end{bmatrix} =$$
$$= \begin{bmatrix} 0.1 & 0.2 \\ -0.2 & 0.1 \\ 0.1 & 0.5 \\ 0.3 & -0.3 \end{bmatrix} * \begin{bmatrix} 1.8555 \\ -1.1375 \end{bmatrix} = \begin{bmatrix} -0.04195 \\ -0.48485 \\ -0.3832 \\ 0.8979 \end{bmatrix}.$$

Wartości te powinny jeszcze zostać pomnożone przez wynik pochodnej funkcji aktywacji zastosowanej na neuronach warstwy pierwszej, ponieważ jednak nie zmieni to wartości wektora, krok ten został

pominięty w przykładzie. Ponieważ w warstwie konwolucyjnej wynikiem była macierz, wektor należy przekształcić na macierz o takich samych wymiarach, jak macierz wyjściowa warstwy konwolucyjnej:

$$kernel\_layer\_1\_delta\_reshaped = \begin{bmatrix} -0.04195 & -0.48485 \\ -0.3832 & 0.8979 \end{bmatrix}$$

Zanim nastąpi aktualizacja wartości wag w poszczególnych warstwach należy wartości delt przeskalować mnożąc je przez wartość neuronów wejściowych.

$$\begin{aligned} layer\_output\_weight\_delta &= layer\_output\_delta * kernel\_layer\_flatten^T = \\ &= \begin{bmatrix} 1.8555 \\ -1.1375 \end{bmatrix} * \begin{bmatrix} 3.185 \\ 11.995 \\ 3.27 \\ 12.03 \end{bmatrix}^T = \\ &= \begin{bmatrix} 1.8555 \\ -1.1375 \end{bmatrix} * \begin{bmatrix} 3.185 & 11.995 & 3.27 & 12.03 \end{bmatrix} = \\ &= \begin{bmatrix} 5.90977 & 22.2567 & 6.06748 & 22.3217 \\ -3.62294 & -13.6443 & -3.71962 & -13.6841 \end{bmatrix}, \end{aligned}$$

$$\begin{aligned} kernel\_layer\_1\_weight\_delta &= kernel\_layer\_1\_delta\_reshaped^T * image\_sections = \\ &= \begin{bmatrix} -0.04195 & -0.48485 \\ -0.3832 & 0.8979 \end{bmatrix}^T * \begin{bmatrix} 8.5 & 9.5 & 9.9 & 0.65 & 0.8 & 0.8 & 1.2 & 1.3 & 0.5 \\ 9.5 & 9.9 & 9 & 0.8 & 0.8 & 0.9 & 1.3 & 0.5 & 1 \end{bmatrix} = \\ &= \begin{bmatrix} -0.04195 & -0.3832 \\ -0.48485 & 0.8979 \end{bmatrix} * \begin{bmatrix} 8.5 & 9.5 & 9.9 & 0.65 & 0.8 & 0.8 & 1.2 & 1.3 & 0.5 \\ 9.5 & 9.9 & 9 & 0.8 & 0.8 & 0.9 & 1.3 & 0.5 & 1 \end{bmatrix} = \\ &= \begin{bmatrix} -3.997 & -4.192 & -3.864 & -0.334 & -0.34 & -0.378 & -0.548 & -0.246 & -0.404 \\ 4.409 & 4.283 & 3.281 & 0.403 & 0.33 & 0.42 & 0.586 & -0.181 & 0.655 \end{bmatrix}. \end{aligned}$$

Na koniec można zaktualizować wartości wag poszczególnych neuronów:

$$\begin{aligned} Wy &= Wy - alpha * layer\_output\_weight\_delta = \\ &= \begin{bmatrix} 0.1 & -0.2 & 0.1 & 0.3 \\ 0.2 & 0.1 & 0.5 & -0.3 \end{bmatrix} - 0.01 * \begin{bmatrix} 5.90977 & 22.2567 & 6.06748 & 22.3217 \\ -3.62294 & -13.6443 & -3.71962 & -13.6841 \end{bmatrix} = \\ &= \begin{bmatrix} 0.0409023 & -0.422567 & 0.0393252 & 0.0767834 \\ 0.236229 & 0.236443 & 0.537196 & -0.163159 \end{bmatrix}. \end{aligned}$$

$$\begin{aligned} kernels &= kernels - alpha * kernel\_layer\_1\_weight\_delta = \\ &= \begin{bmatrix} 0.1 & 0.2 & -0.1 & -0.1 & 0.1 & 0.9 & 0.1 & 0.4 & 0.1 \\ 0.3 & 1.1 & -0.3 & 0.1 & 0.2 & 0 & 0 & 1.3 & 0.1 \end{bmatrix} - 0.01 * \\ &* \begin{bmatrix} -3.997 & -4.192 & -3.864 & -0.334 & -0.34 & -0.378 & -0.548 & -0.246 & -0.404 \\ 4.409 & 4.283 & 3.281 & 0.403 & 0.33 & 0.42 & 0.586 & -0.181 & 0.655 \end{bmatrix} = \\ &= \begin{bmatrix} 0.13997 & 0.2419 & -0.0614 & -0.0967 & 0.1034 & 0.9038 & 0.1055 & 0.4025 & 0.104 \\ 0.2559 & 1.0571 & -0.3328 & 0.096 & 0.1967 & -0.0042 & -0.0059 & 1.3018 & 0.0934 \end{bmatrix}. \end{aligned}$$

5.3 Zadanie 1

Przygotuj funkcję do realizacji algorytmu splotu na obrazie, funkcja powinna przyjmować następujące parametry:

- obraz wejściowy w postaci tablicy dwuwymiarowej,
- filtr, który ma zostać zastosowany na obrazie, w postaci tablicy dwuwymiarowej,
- krok, z którym ma być wykonywana operacja splotu,
- padding - liczba wierszy / kolumn, które mają być wypełnione zerami wokół obrazu.

Funkcja powinna zwrócić nowy obraz, będący wynikiem splotu obrazu wejściowego z maską.

Przykładowy wynik działania funkcji dla obrazu:

$$input\_image = \begin{bmatrix} 1 & 1 & 1 & 0 & 0 \\ 0 & 1 & 1 & 1 & 0 \\ 0 & 0 & 1 & 1 & 1 \\ 0 & 0 & 1 & 1 & 0 \\ 0 & 1 & 1 & 0 & 0 \end{bmatrix}$$

oraz maski

$$filter = \begin{bmatrix} 1 & 0 & 1 \\ 0 & 1 & 0 \\ 1 & 0 & 1 \end{bmatrix}$$

dla kroku równego 1 i bez paddingu to:

$$output\_image = \begin{bmatrix} 4 & 3 & 4 \\ 2 & 4 & 3 \\ 2 & 3 & 4 \end{bmatrix}$$

5.4 Zadanie 2

Zaimplementuj sieć konwolucyjną składającą się z jednej warstwy konwolucyjnej z 16 filtrami 3x3 oraz 10 neuronów w warstwie wyjściowej. W warstwie konwolucyjnej zastosuj funkcję aktywacji ReLU. Przetestuj działanie sieci dla następujących ustawień:

- 1000 obrazów w zbiorze treningowym, 10000 obrazów w zbiorze testowym, współczynnik uczenia 0.01, wagi z zakresu $< -0.01, 0.01 >$ w warstwie konwolucyjnej oraz $< -0.1, 0.1 >$ w warstwie wyjściowej, 50 iteracji,
- 10000 obrazów w zbiorze treningowym, 10000 obrazów w zbiorze testowym, współczynnik uczenia 0.01, wagi z zakresu $< -0.01, 0.01 >$ w warstwie konwolucyjnej oraz $< -0.1, 0.1 >$ w warstwie wyjściowej, 50 iteracji,
- 60000 obrazów w zbiorze treningowym, 10000 obrazów w zbiorze testowym, współczynnik uczenia 0.01, wagi z zakresu $< -0.01, 0.01 >$ w warstwie konwolucyjnej oraz $< -0.1, 0.1 >$ w warstwie wyjściowej, 50 iteracji.

5.5 Zadanie 3

Zaimplementuj sieć konwolucyjną następującej postaci: $CONV- > RELU- > POOL- > FC$. W warstwie konwolucyjnej powinno być 16 filtrów o wymiarach 3x3, do operacji *max pooling* przyjmij rozmiar maski 2x2 i krok 2. Warstwa ostatnia powinna standardowo posiadać 10 neuronów. Przetestuj działanie sieci dla następujących ustawień:

- 1000 obrazów w zbiorze treningowym, 10000 obrazów w zbiorze testowym, współczynnik uczenia 0.001, wagi z zakresu $< -0.01, 0.01 >$ w warstwie konwolucyjnej oraz $< -0.1, 0.1 >$ w warstwie wyjściowej, 300 iteracji,
- 10000 obrazów w zbiorze treningowym, 10000 obrazów w zbiorze testowym, współczynnik uczenia 0.001, wagi z zakresu $< -0.01, 0.01 >$ w warstwie konwolucyjnej oraz $< -0.1, 0.1 >$ w warstwie wyjściowej, 300 iteracji,
- 60000 obrazów w zbiorze treningowym, 10000 obrazów w zbiorze testowym, współczynnik uczenia 0.01, wagi z zakresu $< -0.01, 0.01 >$ w warstwie konwolucyjnej oraz $< -0.1, 0.1 >$ w warstwie wyjściowej, 50 iteracji.

6 Laboratorium 6

Celem laboratorium jest zapoznanie się z podstawami algorytmów heurystycznych.

6.1 Teoria

- *Introduction to Artificial Intelligence, Berkeley - Uniformed Search.*
- Pacman Project

6.2 Zadanie 1

Implementacja algorytmu przeszukiwania w głąb (*Depth-first search*). Celem jest znalezienie kropki w labiryncie.

6.3 Zadanie 2

Implementacja algorytmu przeszukiwania wszerz (*Breadth-first search*).

6.4 Zadanie 3

Implementacja algorytmu przeszukiwania wszerz (*Breadth-first search*) ze zmienną funkcją kosztu ruchu. W tym przypadku agent powinien odnaleźć ścieżkę o jak najmniejszym koszcie. Kolejny węzeł powinien być wybierany na podstawie na podstawie sumy kosztu przebytej drogi oraz kosztu przypisanego do kolejnego węzła.

6.5 Zadanie 4

Implementacja algorytmu A^*. W tym przypadku agent powinien odnaleźć ścieżkę o jak najmniejszym koszcie. Kolejny węzeł powinien być wybierany na podstawie na podstawie sumy kosztu przebytej drogi oraz wartości funkcji heurystycznej dla kolejnego węzła.

6.6 Zadanie 5

Celem zadania jest rozbudowanie zadania 2 o możliwość odwiedzenia przez agenta wszystkich narożników planszy. W tym celu trzeba przygotować własną strukturę przechowującą aktualny stan agenta. Struktura powinna przechowywać tylko te informacje, które są istotne do rozwiązania postawionego zadania.

6.7 Zadanie 6

Celem zadania jest napisanie heurystyki do zadania mającego na celu odwiedzenie przez agenta wszystkich narożników na planszy. Heurystyka jest funkcją, która przyjmuje stan i zwraca szacowany koszt dotarcia do najbliższego celu.

6.8 Zadanie 7

Celem zadania jest przygotowanie heurystyki pozwalającej agentowi na zjedzenie wszystkich dostępnych na planszy kropek.

6.9 Zadanie 8

Celem zadania jest przygotowanie funkcji pozwalającej agentowi na zjedzenie wszystkich dostępnych na planszy kropek, przyjmują założenie, że agent będzie podążał zawsze do najbliższej kropki.

7 Laboratorium 7

Celem laboratorium jest zapoznanie się z podstawami algorytmów przeszukiwania grafów i drzew.

7.1 Teoria

- *Introduction to Artificial Intelligence, Berkeley - minmax, alpha-beta ....*
- Pacman Project

7.2 Zadanie 1

Zaimplementuj agenta, który na podstawie dostępnych informacji o położeniu jedzenia oraz przeciwników będzie wybierach kierunek, w którym się poruszy i starał się zjeść wszystkie dostępne na planszy jedzenia, unikając przy okazji przeciwników.

7.3 Zadanie 2

Implementacja algorytmu *minmax* (wersja uogólniona, obejmująca dowolną liczbę przeciwników).

7.4 Zadanie 3

Implementacja algorytmu *alpha–beta pruning*. Algorytm ten jest zoptymalizowaną wersją algorytmu minmax, zaimplementowanego w ramach zadania 2.

7.5 Zadanie 4

Implementacja algorytmu *expectiminimax*. Algorytm ten jest zoptymalizowaną wersją algorytmu minmax, zaimplementowanego w ramach zadania 2.

7.6 Zadanie 5

Celem zadania jest przygotowania własnej funkcji oceny stanu, która umożliwi agentowi na jeszcze lepsze poruszanie się w grze.

8 Laboratorium 8

Celem laboratorium jest zapoznanie się z podstawami algorytmów genetycznych.

8.1 Teoria

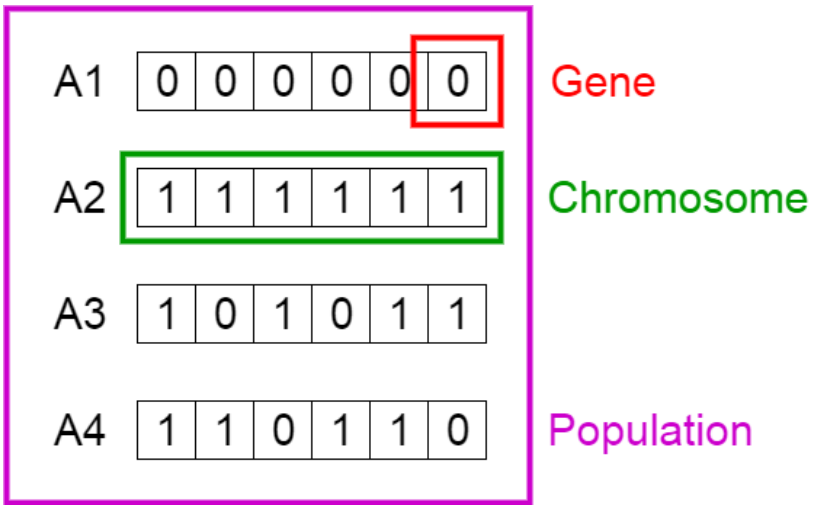
- Noureddin Sadawi, *An Introduction to Genetic Algorithms*.

8.2 Wstęp teoretyczny

Algorytm genetyczny to heurystyka wyszukiwania inspirowana teorią naturalnej ewolucji Charlesa Darwina. Algorytm ten odzwierciedla proces doboru naturalnego, w którym najsilniejsze osobniki są wybierane do rozmnażania w celu uzyskania potomstwa. Algorytmy te należą do grupy algorytmów ewolucyjnych.

8.2.1 Podstawowe pojęcia

Populacja jest to zbiór indywidualnych osobników, gdzie każdy osobnik odpowiada jednemu rozwiązaniu analizowanego problemu. Każdy osobnik posiada zbiór właściwości (**genów**), które tworzą **chromosom** (**genotyp**) (rys. 12). Zazwyczaj chromosom jest reprezentowany jak ciąg 0 i 1, jednakże inne znaki są również dopuszczalne. Zestaw konkretnych i dających się liczbowo zapisać cech generowanych na podstawie genotypu, które podlegają ocenie w środowisku nazywany jest **fenotypem**. Ocena przystosowania osobnika dokonywana jest na podstawie fenotypu, a nie genotypu, gdyż nawet najdrobniejsza zmiana w genotypie osobnika może spowodować dużą zmianę fenotypu.



Rysunek 12: Podstawowe pojęcia (źródło: Vijini Mallawaarachchi, "Introduction to Genetic Algorithms")

8.2.2 Przykład reprezentacji

Załóżmy, że naszym zadaniem jest dobór optymalnego telefonu komórkowego. Zakładając, że do wyboru jest czterech producentów telefonów (Nokia, Samsung, LG, ZTE), i każdy z nich możemy scharakteryzować za pomocą 6 cech (kolor, wifi, bluetooth, ekran dotykowy, wodoodporność, kamera z przodu), to każdy pojedynczy model możemy reprezetnować za pomocą 10 bitowego ciągu (przyjmując możliwość wyboru 8 kolorów):

- bity 1 - 2 będą oznaczały producenta (np. 0 - Nokia, 1 - Samsung, 2 - LG, 3 - ZTE),
- bity 3 - 5 będą oznaczały kolor,
- bit 6 - wifi (czy model posiada czy nie),
- bit 7 - bluetooth (czy model posiada czy nie),
- bit 8 - ekran dotykowy (czy model posiada czy nie),

- bit 9 - wodoodporność (czy model posiada czy nie),
- bit 10 - kamera (czy model posiada czy nie).

Przykładem jednego z rozwiązań takiego problemu może być następujący ciąg: 1011101010.

8.2.3 Sposób postępowania

Algorytmy genetyczne są algorytmami iteracyjnymi, przy czym populacja powstała w każdej iteracji nazywana jest **generacją**. Pierwszym krokiem jest utworzenie populacji startowej składającej się z losowo wygenerowanych osobników. Następnie obliczana jest ocena przystosowania (ang. *fitness*) każdego osobnika z populacji. Ocena przystosowania najczęściej jest wartością funkcji oceny w rozwiązywanym problemie. Osobnicy z największą wartością funkcji przystosowania są wybierani z obecnej populacji, następnie ich geny są krzyżowane lub losowo mutowane w celu utworzenia nowej generacji osobników. Warunkiem zakończenia działania algorytmu jest osiągnięcie maksymalnej liczby generacji lub osiągnięcie przez któregokolwiek osobnika z danej generacji większej wartości funkcji przystosowania od założonej.

Dla każdego problemu, który ma być rozwiązany za pomocą algorytmów genetycznych należy opracować:

- reprezentacja genetyczna w domenie problemu,
- funkcja przystosowania (ang. *fitness function*)) wyznaczająca poziom dostoswania poszczególnych jednostek w domenie problemu.

8.2.4 Przykładowy algorytm

1. Utworzenie losowej populacji składającej się z **n** chromosomów.
2. Wyznaczenie funkcji przystosowania $f(x)$ dla każdego chromosomu **x** z populacji.
3. Utworzenie nowej populacji poprzez powtarzanie poniższych kroków, aż do osiągnięcia wymaganej liczby osobników:
    - **Selekcja** - Wybranie dwóch chromosomów (rodziców) z populacji biorąc pod uwagę ich wartości przystosowania (jedną z najczęściej stosowanych metod selekcji jest metoda ruletki). Celem tej fazy jest wybór najlepiej przystosowanych osobników w celu przekazania swoich genów kolejnym pokoleniom.
    - **Krzyżowanie** - Z pewnym prawdopodbieństwem krzyżowane są ze sobą geny rodziców w celu utworzenia potomka, który będzie należał do nowej generacji osobników. Okazuje się, że chromosomy powstałe w wyniku krzyżowania często są lepiej przystosowane (osiągają większą wartość funkcji przystosowania). Jeżeli nie zajdzie operacja krzyżowania, to potomstwo jest dokładną kopią rodziców.
    - **Mutacja** - Z pewnym prawdopodobieństwem zmieniana jest wartość pojedynczego genu w chromosomie.
    - Dodanie nowego osobnika do populacji.
4. Wykorzystanie nowej populacji w kolejnej iteracji algorytmu.
5. Sprawdzenie czy warunek zakończenia działania algorytmu został spełniony, jeżeli tak, to następuje zakończenie algorytmy, w przeciwnym przypadku powrót do punktu 2.

Dodatkowo można zastosować **elitarność** (ang. *elitism*), czyli podejście polegające na skopiowaniu co najmniej jednego osobnika, z największą wartością funkcji przystosowania, bez zmian do nowej generacji w celu zachowania najlepszego rozwiązania do końca działania algorytmu.

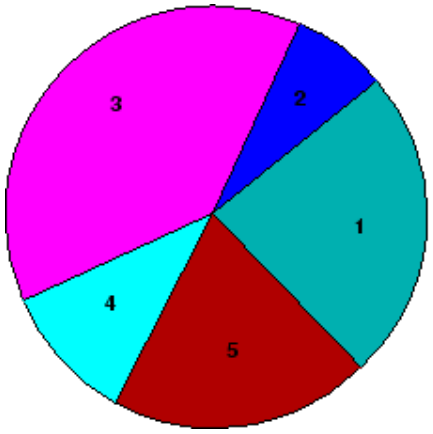
8.2.5 Operatory genetyczne

Selekcja - polega na wyborze z bieżącej populacji najlepiej przystosowanych osobników, których materiał genetyczny zostanie poddany operacji krzyżowania i przekazany osobnikom następnej populacji. Główną regułą tej fazy jest przetrwanie osobników najlepiej przystosowanych. Istnieje wiele metod do wyboru najlepszych chromosomów:

- Metoda ruletki (ang. *roulette wheel selection*) - dla każdego osobnika z populacji obliczane jest prawdopodobieństwo jego wyboru do nowej populacji na podstwie wartości funkcji przystosowania. Najczęściej wykorzystywany jest następujący wzór:

$$p(i) = \frac{F(i)}{\sum_{i=1}^{n} F(i)}$$

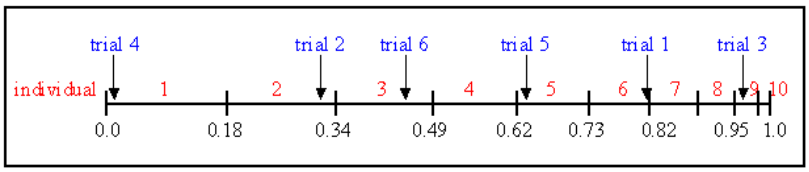
Możemy to sobie wyobrazić w następujący sposób: każdemu osobnikowi z populacji przydzielany jest pewien wycinek koła, o wielkości odpowiadającej prawdopodobieństwu przetrwania użytkownika (rys. 13). Następnie koło jest uruchamiane i w momencie zatrzymania wskaźnik wskaże tego osobnika, który trafi do nowej populacji.



Population	Fitness
1	25.0
2	5.0
3	40.0
4	10.0
5	20.0

Rysunek 13: Metoda ruletki

Od strony programistycznej metoda ta będzie polegała na wylosowaniu wartości z przedziału $< 0,1 >$, a następnie sprawdzeniu, do przedziału którego z osobników wylosowana wartość należy (rys. 14).

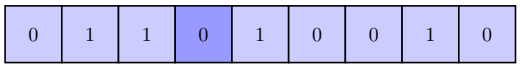


Rysunek 14: Metoda ruletki

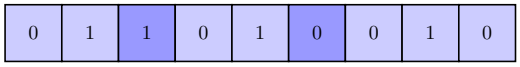
- Selekcja rankingowa (ang. *rank selection*) - wybieranych jest **n** najlepiej przystosowanych osobników. Wybór ten dokonywany jest na podstawie rankingu, w którym wszyscy osobnicy są posortowani od najbardziej przystosowanego do najgorzej.
- Selekcja turniejowa (ang. *tournament selection*) - cała populacja dzielona jest na losowe grupy, a następnie z każdej z grup wybierany jest osobnik o największej wartości funkcji przystosowania.

Mutacja - polega na losowej zmianie pojedynczego lub kilku genów w chromosomie. Każdy gen ma takie samo prawdopodobieństwo mutacji. Prawdopodobieństwo zajścia mutacji powinno być bardzo małe, ponieważ w innym przypadku spowoduje to, że nawet korzystne zmiany jakie zajdą w procesie mutacji nie będą miały szans na utrwalenie się w populacji. Istnieją różne metody mutacji:

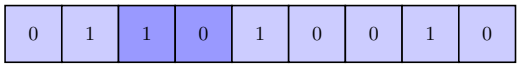
- Zamiana (ang. *replacement*) - zamiana wartości pojedynczego chromosomu na przeciwną (rys. 15).
- Zamiana genów (ang. *random swap*) - zamiana wartości dwóch losowych genów (rys. 16).
- Zamiana sąsiadujących genów (ang. *adjacent swap*) - zamiana wartości dwóch losowych genów sąsiadujących ze sobą (rys. 17).



Rysunek 15: Mutacja polegająca na zmianie wartości pojedynczego genu na przeciwną.

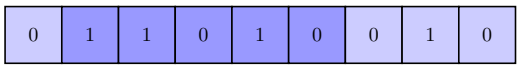


Rysunek 16: Mutacja polegająca na zamianie wartości dwóch różnych genów.



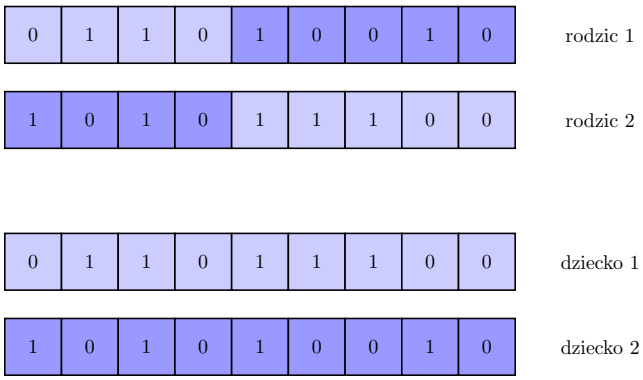
Rysunek 17: Mutacja polegająca na zamianie wartości dwóch sąsiadujących genów.

- Inwersja chromosomowa (ang. *inversion*) - rodzaj mutacji, pojawiającej się w genomie, w wyniku której chromosom ulega pęknięciu w dwóch miejscach, a powstały swobodny fragment ulega przed ponownym wbudowaniem do chromosomu odwróceniu o 180deg[3]. (rys. 18).

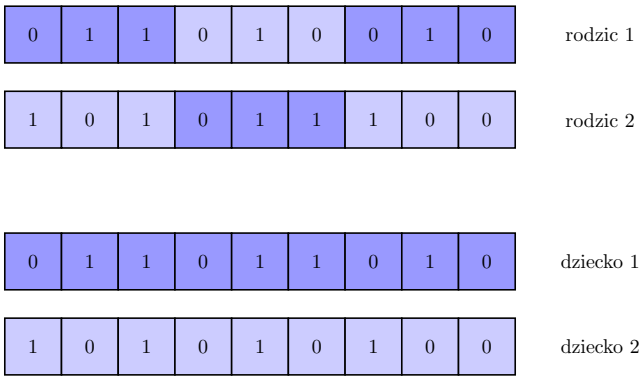


Rysunek 18: Przykład inwersji.

Krzyżowanie - jest to operacja, która odbywa się na parze chromosomów (rodzicach), w wyniku której powstaj jeden lub dwóch potomków. Krzyżowanie odbywa się poprzez rozcięcie chromosomów rodziców w dowolnym punkcie. Następnie, potomkowi przypada po części chromosomu od każdego z rodziców. Najpopularniejsze są krzyżowania jedno- (rys. 19) i dwupunktowe (rys. 20).



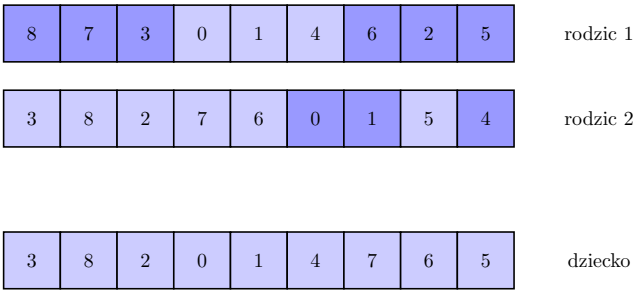
Rysunek 19: Krzyżowanie jednopunktowe.



Rysunek 20: Krzyżowanie dwupunktowe.

[3]wikipedia.org

W niektórych algorytmach genetycznych nie wszystkie możliwe chromosomy reprezentują prawidłowe rozwiązania. W niektórych przypadkach możliwe jest użycie wyspecjalizowanych operatorów krzyżowania i mutacji zaprojektowanych tak, aby uniknąć naruszenia ograniczeń problemu. Na przykład algorytm genetyczny rozwiązujący problem komiwojażera może wykorzystywać uporządkowaną listę miast do przedstawienia ścieżki rozwiązania. Taki chromosom stanowi prawidłowe rozwiązanie tylko wtedy, gdy lista zawiera wszystkie miasta, które sprzedawca musi odwiedzić. Wykorzystanie jednego z operatorów zaprezentowanych powyżej spowoduje w takiej sytuacji utworzenie niepoprawnego rozwiązania. Jednym z operatorów, które mogą zostać zastosowane w tej klasie problemów, to operator krzyżowania uporządkowanego (ang. *order crossover operator*, rys. 21).



Rysunek 21: Krzyżowanie ciągów uporządkowanych.

8.3 Zadanie 1

Załóżmy, że mamy zbiór 10 genów, z któych każdy może przyjąć wartość binarną (0 lub 1). Funkcja przystosowania jest obliczana jako liczba genów o wartości 1 obecnych w chromosomie. Im większ liczba genów o wartości 1, tym wyższy poziom dopasowania. Zaimplementuj algorytm genetyczny, który będzie się starał zmaksymalizować poziom przystosowania osobników w populacji. Przyjmij rozmiar populacji równy 10. W każdej generacji dwóch najlepiej przystosowanych osobników z całej populacji jest wybieranych i poddawanych operacji krzyżowania, w wyniku której powstaje dwóch potomków. Zastępują oni dwóch najmniej przystosowanych osobników w populacji. Dodatkowo z prawdopodobieństwem 60% może nastąpić mutacja polegająca na zamianie wartości losowego genu na przeciwną, wśród dwóch najlepszych osobników.

8.4 Zadanie 2

Napisz program wyznaczający za pomocą algorytmu genetycznego rozwiązanie równania $2a^2 + b = 33$, gdzie $a, b \in < 0, 15 >$. Chromosom powinien składać się z 8 bitów (po 4 bity zajmowane przez wartość zmiennej a i b). Obliczanie funkcji przystosowania powinno odbywać się na podstawie fenotypu (wartościach zmiennych a i b). Rozmiar populacji przyjmij na 10 osobników, w każdej generacji nowa populacja wybierana jest za pomocą metody ruletki, a następnie 50% osobników tej populacji podlega krzyżowaniu, potomek zastępuje w populacji jednego z rodziców. Każdy z osobników nowej populacji może ulec mutacji z prawdopodobieństwem 10%.

8.5 Zadanie 3 - problem plecakowy

Napisz program, który będzie rozwiązywał problem plecakowy za pomocą algorytmu genetycznego. Problem plecakowy często przedstawia się jako problem złodzieja rabującego sklep – znalazł on 10 towarów; j–ty przedmiot jest wart v_j oraz waży w_j. Złodziej dąży do zabrania ze sobą jak najwartościowszego łupu, przy czym nie może zabrać więcej niż 35 kilogramów. Nie może też zabierać ułamkowej części przedmiotów (byłoby to możliwe w ciągłym problemie plecakowym)[4]. Wartość funkcji przystosowania będzie obliczania zgodnie z następującym wzorem:

$$fitness = \begin{cases} \sum_{i=1}^{n} c_i v_i & \text{if} \quad \sum_{i=1}^{n} c_i w_i \leqslant 35 \\ 0 & \text{if} \quad \sum_{i=1}^{n} c_i w_i > 35 \end{cases}$$

[4] `https://pl.wikipedia.org/wiki/Problem_plecakowy`

gdzie:

- n - długość chromosomu (odpowiada liczbie wszystkich przedmiotów w sklepie),
- c_i - wartość i-tego genu (1 oznacza, że dany przedmiot jest spakowany do torby, 0, że nie),
- w_i - waga i-tego przedmiotu,
- v_i - wartość i-tego przedmiotu.

Przyjmij rozmiar populacji równy 8. W trakcie implementacji algorytmu wykorzystaj zasadę elitarności, jako próg przyjmij 25% osobników w populacji. Z całej populacji wybierana jest tymczasowa populacja za pomocą metody ruletki, a następnie dochodzi do krzyżowań pomiędzy osobnikami, a nowo powstałe osobniki trafiają do nowej populacji. Dodatkowo każdy gen każdego z nowych osobników może podlegać mutacji z prawdopodobieństwem 5%. Oznacza to, że nowa populacja będzie składała się w 25% z osobników poprzedniej populacji oraz w 75% z potomków, którzy mogą podlegać mutacji.

Przetestuj działanie swojego algroytmu na zestawie danych przedstawionym w tabeli 1. Optymalnym (prawdopodobnie) rozwiązaniem dla tego przykładu jest wybranie rzeczy o numerach: $0, 2, 3, 4, 5, 8$, waga wybranych rzeczy to 32, a wartość 2222.

ID	Waga	Wartość
0	3	266
1	13	442
2	10	671
3	9	526
4	7	388
5	1	245
6	8	210
7	8	145
8	2	126
9	9	322

Tabela 1: Przykładowe dane do problemu plecakowego.

8.6 Zadanie 4 - problem komiwojażera

Napisz program, który będzie rozwiązywał problem komiwojażera za pomocą algorytmu genetycznego. Problem komiwojażera (ang. *travelling salesman problem*, TSP) – zagadnienie optymalizacyjne, polegające na znalezieniu minimalnego cyklu Hamiltona w pełnym grafie ważonym. Nazwa pochodzi od typowej ilustracji problemu, przedstawiającej go z punktu widzenia wędrownego sprzedawcy (komiwojażera): dane jest n miast, które komiwojażer ma odwiedzić, oraz odległość podróży pomiędzy każdą parą miast. Celem jest znalezienie najkrótszej drogi łączącej wszystkie miasta, zaczynającej się i kończącej się w tym punkcie[5].

W tym przypadku genem będą współrzędne miasta (x, y) lub indeks miasta (w zależności od tego, co będzie łatwiej przechowywać). Algorytm powinien działać analogicznie do tego z zadania 3. Przyjmij następujące parametry:

- wielkość populacji - 100,
- poziom elitarności - 20%,
- prawdopodobieństwo mutacji - 1%.

Przetestuj działanie algorytmu na zestawie danych z tabeli 2. Optymalnym (prawdopodobnie) rozwiązaniem dla tego przykładu jest następująca trasa: 20 14 17 13 4 18 1 19 21 23 0 16 15 2 12 11 5 7 22 9 24 10 3 8 6, o długości 869.

[5] `https://pl.wikipedia.org/wiki/Problem_komiwoja%C5%BCera`

ID	x	y
0	119	38
1	37	38
2	197	55
3	85	165
4	12	50
5	100	53
6	81	142
7	121	137
8	85	145
9	80	197
10	91	176
11	106	55
12	123	57
13	40	81
14	78	125
15	190	46
16	187	40
17	37	107
18	17	11
19	67	56
20	78	133
21	87	23
22	184	197
23	111	12
24	66	178

Tabela 2: Przykładowe dane do problemu komiwojażera.

9 Laboratorium 9

Celem tego laboratorium jest zapoznanie się z **logiką rozmytą** i **teorią zbiorów rozmytych**. Zapoznasz się z pojęciami **rozmywania (fuzyfikacji)**, **wnioskowania** w ramach modelu Mamdaniego (inferencji) oraz **wyostrzania (defuzyfikacji)**.

Zbudujesz również własne zbiory reguł wnioskowania, odpowiadające Twoim obserwacjom oraz intuicji, zdobytym podczas prac eksperymentalnych.

9.1 Teoria i materiały

- Warner, Joshua, Mayo Clinic Department of Biomedical Engineering; Ottesen, Hal H. *Scikit-Fuzzy: A New SciPy Toolkit for Fuzzy Logic; SciPy 2013 Presentation*
- Essam Hamdi, *Fuzzy Control Part II*.
- Essam Hamdi, *What is Fuzzy Logic*.
- ***H462710 - Fuzzy Logic Control Example***.
- OpenAI, *OpenAI Gym*.
- Kimia Lab, *Machine Intelligence - Lecture 17*.

W ramach tego laboratorium Twoim zadaniem będzie przygotowanie algorytmu umożliwiającego grę w ponga.

9.2 Zadanie

Zaimplementuj kontroler rozmyty zdolny do kontrolowania paletki w klasycznej grze Pong, uproszczona implementacja została udostępniona na GitHubie. Kontroler powinien być zaimplementowany przy użyciu *scikit-fuzzy*. Kontroler będzie miał dostęp do dwóch wartości: odległości pomiędzy środkiem piłki i środkiem paletki w wymiarze X i Y. Celem jest przechwycenie piłki za każdym razem i nie pozwolenie przeciwnikowi na zdobycie bramki. W porównaniu z klasyczną wersją, ta implementacja zapewnia inny sposób na zwiększenie prędkości odbijanej piłki. Jeśli piłka jest odbijana przy użyciu najbardziej skrajnego 25% paletki, jej prędkość wzrasta o 10%. Ponieważ prędkość paletek jest ograniczona, w pewnym momencie naiwny przeciwnik nie będzie w stanie nadążyć.

9.3 Zasady oceny

- Podstawowy zestaw reguł podążania za piłką z konsekwencjami wyznaczonymi metodą Mamdamiego - 3,
- Podstawowy zbiór reguł podążania za piłką z konsekwencjami wyznaczonymi metodą TSK - 4,
- Bardziej złożony zestaw reguł, który próbuje odbijać piłkę używając brzegów krawędzi paletki - 5.