Sieci neuronowe dla początkujących w Pythonie: wykłady w Jupyter Book

Wojciech Broniowski

09 mar 2023

Spis treści

1 Wstęp **3**
- 1.1 Cel wykładu . . . 3
- 1.2 Inspiracja biologiczna . . . 4
- 1.3 Sieci feed-forward . . . 5
- 1.4 Dlaczego Python . . . 6

2 Neuron MCP **9**
- 2.1 Definicja . . . 9
- 2.2 Neuron MCP w Pythonie . . . 11
- 2.3 Funkcje logiczne . . . 14
- 2.4 Ćwiczenia . . . 17

3 Modele pamięci **19**
- 3.1 Pamieć skojarzeniowa (heteroasocjacyjna) . . . 19
- 3.2 Pamieć autoasocjatywna . . . 24
- 3.3 Ćwiczenia . . . 27

4 Perceptron **29**
- 4.1 Uczenie nadzorowane . . . 29
- 4.2 Perceptron jako klasyfikator binarny . . . 30
- 4.3 Algorytm perceptronu . . . 33
- 4.4 Ćwiczenia . . . 37

5 Więcej warstw **39**
- 5.1 Dwie warstwy neuronów . . . 39
- 5.2 Trzy lub więcej warstw neuronów . . . 40
- 5.3 Feed forward w Pythonie . . . 41
- 5.4 Wizualizacja . . . 45
- 5.5 Klasyfikator z trzema warstwami neuronów . . . 46
- 5.6 Ćwiczenia . . . 48

6 Propagacja wsteczna **49**
- 6.1 Minimalizacja błędu . . . 49
- 6.2 Ciągła funkcja aktywacji . . . 53
- 6.3 Najstromszy spadek . . . 56
- 6.4 Algorytm propagacji wstecznej (backprop) . . . 58
- 6.5 Przykład z kołem . . . 61
- 6.6 Ogólne uwagi . . . 64
- 6.7 Ćwiczenia . . . 64

7 Interpolacja **67**

7.1 Symulowane dane . . . 67
7.2 Interpolacja z pomocą ANN . . . 68
7.3 Ćwiczenia . . . 72

8 Rektyfikacja **73**
8.1 Interpolacja z ReLU . . . 75
8.2 Klasyfikatory z rektyfikacją . . . 76
8.3 Ćwiczenia . . . 77

9 Uczenie nienadzorowane **79**
9.1 Klastry . . . 80
9.2 Komórki Woronoja . . . 81
9.3 Naiwna klasteryzacja . . . 82
9.4 Skala klastrowania . . . 87

10 TUTAJ **91**
10.1 Interpretacja jako ANN . . . 91
10.2 Ćwiczenia . . . 94

11 Mapy samoorganizujące się **97**
11.1 Kohonen's algorithm . . . 98
11.2 U-matrix . . . 105
11.3 Mapping 2-dim. data into a 2-dim. grid . . . 110
11.4 Topology . . . 111
11.5 Lateral inhibition . . . 115
11.6 Exercises . . . 118

12 Concluding remarks **119**
12.1 Acknowledgments . . . 119

13 Dodatki **121**
13.1 Jak uruchamiać kody książki . . . 121
13.2 Pakiet **neural** . . . 121
13.3 Jak cytować . . . 132

Bibliografia **133**

Wojciech Broniowski

Niniejsze wykłady były pierwotnie prowadzone dla studentów inżynierii danych na Uniwersytecie Jana Kochanowskiego w Kielcach i dla Krakowskiej Szkoły Interdyscyplinarnych Studiów Doktoranckich. Wyjaśniają bardzo podstawowe koncepcje sieci neuronowych na najbardziej przystępnym poziomie, wymagając od studenta jedynie bardzo podstawowej znajomości Pythona, a właściwie dowolnego języka programowania. W trosce o prostotę, kod dla różnych algorytmów sieci neuronowych pisany jest od podstaw, tj. bez użycia dedykowanych bibliotek wyższego poziomu. W ten sposób można dokładnie prześledzić wszystkie etapy programowania.

Zwięzłość

Tekst jest zwięzły (wydruk pdf ma ~130 stron wraz z załącznikami), więc pilny student może ukończyć kurs w kilka popołudni!

Linki

- Jupyter Book: https://bronwojtek.github.io/nn_polish/docs/index.html

Pierwotna angielska wersja książki:

- Jupyter Book: https://bronwojtek.github.io/neuralnets-in-raw-python/docs/index.html
- pdf i kody: www.ifj.edu.pl/~broniows/nn lub www.ujk.edu.pl/~broniows/nn

---

Jak uruchamiać kody w książce

Główną zaletą książek wykonywalnych jest to, że czytelnik może cieszyć się z samodzielnego uruchamiania kodów źródłowych, modyfikowania ich, czy zabawy z parametrami. Nie jest potrzebne pobieranie, instalacja ani konfiguracja. Po prostu przejdź do

https://bronwojtek.github.io/nn_polish/docs/index.html,

w menu po lewej stronie wybierz dowolny rozdział poniżej Wstępu, kliknij ikonę „rakiety" w prawym górnym rogu ekranu i wybierz „Colab" lub „Binder". Po pewnym czasie inicjalizacji (za pierwszym razem dla Bindera trwa to dość długo) można uruchomić notebook.

Dla wykonywania lokalnego, kody dla każdego rozdziału w postaci notebooków Jupytera można pobrać klikając ikonę „strzałki w dół" w prawym górnym rogu ekranu. Pełen zestaw plików jest również dostępny z linków podanych powyżej.

Dodatek *Jak uruchamiać kody książki* wyjaśnia, jak postępować przy lokalnym wykonywaniu programów.

Książka wykonywalna, utworzona przez oprogramowanie Jupyter Book 2.0, będące częścią ExecutableBookProject.

ROZDZIAŁ 1

Wstęp

1.1 Cel wykładu

Celem kursu jest wyłożenie podstaw wszechobecnych sieci neuronowych z pomocą Pythona [Bar16, Gut16, Mat19]. Zarówno kluczowe pojęcia sieci neuronowych, jak i programy ilustrujące są wyjaśniane na bardzo podstawowym poziomie, niemal „licealnym". Kody, bardzo proste, zostały szczegółowo opisane. Ponadto są utworzone bez użycia specjalistycznych bibliotek wyższego poziomu dla sieci neuronowych, co pomaga w lepszym zrozumieniu przedstawionych algorytmów i pokazuje, jak programować je od podstaw.

Dla kogo jest ta książka?

Czytelnik może być zupełnym nowicjuszem, tylko w niewielkim stopniu zaznajomionym z Pythonem (a właściwie każdym innym językiem programowania) i Jupyterem.

Materiał obejmuje takie klasyczne zagadnienia, jak perceptron i jego najprostsze zastosowania, nadzorowane uczenie z propagacją wsteczną do klasyfikacji danych, uczenie nienadzorowane i klasteryzacja, sieci samoorganizujące się Kohonena oraz sieci Hopfielda ze sprzężeniem zwrotnym. Ma to na celu przygotowanie niezbędnego gruntu dla najnowszych i aktualnych postępów (nie omówionych tutaj) w sieciach neuronowych, takich jak uczenie głębokie, sieci konwolucyjne, sieci rekurencyjne, generatywne sieci przeciwników, uczenie ze wzmacnianiem itp.

W trakcie kursu nowicjuszom zostanie delikatnie przemycone kilka podstawowych programów w Pythonie. W kodach znajdują się objaśnienia i komentarze.

Ćwiczenia

Na końcu każdego rozdziału proponujemy kilka ćwiczeń, których celem jest zapoznanie czytelnika z poruszanymi tematami i kodami. Większość ćwiczeń polega na prostych modyfikacjach/rozszerzeniach odpowiednich fragmentów materiału wykładowego.

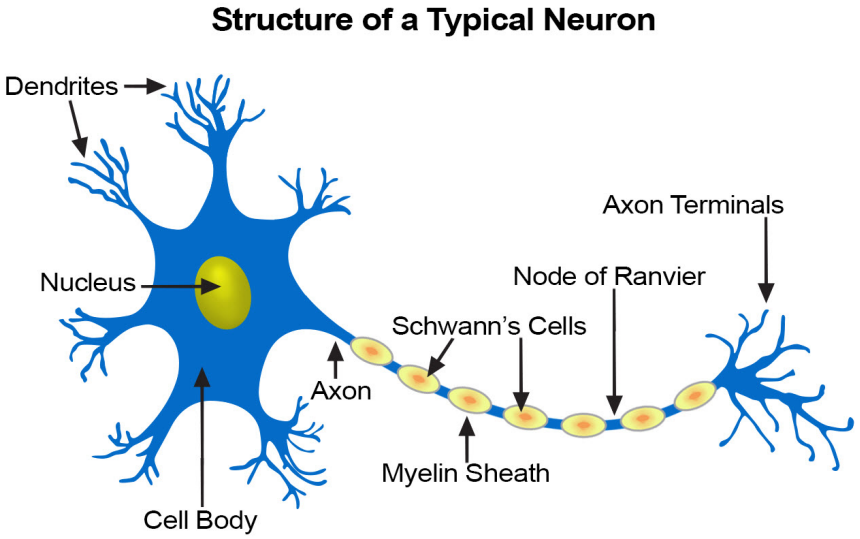
Literatura

Podręczników i notatek do wykładów poświęconych zagadnieniom poruszanym na tym kursie jest niezliczona ilość, stąd autor nie będzie próbował przedstawiać nawet niepełnego spisu literatury. Przytaczamy tylko pozycje, na które może spojrzeć bardziej zainteresowany czytelnik.

Z prostotą jako drogowskazem, wybór tematów był inspirowany szczegółowymi wykładami Daniela Kerstena w programie Mathematica, z internetowej książki Raula Rojasa (dostępna również w wersji drukowanej [FR13]) oraz z punktu widzenia **fizyków** (jak ja!) z [MullerRS12].

1.2 Inspiracja biologiczna

Inspiracją do opracowania matematycznych modeli obliczeniowych omawianych w tym kursie jest struktura biologiczna naszego układu nerwowego [KSJ+12]. Centralny układ nerwowy (mózg) zawiera ogromną liczbę ($\sim 10^{11}$) neuronów, które można postrzegać jako maleńkie elementarne procesory. Otrzymują one sygnał poprzez **dendryty**, a jeśli jest on wystarczająco silny, jądro decyduje (obliczenie jest wykonane tutaj!) „wystrzelić" sygnał wyjściowy wzdłuż **aksonu**, gdzie jest on następnie przekazywany przez zakończenia aksonów do dendrytów innych neuronów. Połączenia aksonowo-dendryczne (połączenia **synaptyczne**) mogą być słabe lub silne, modyfikując przekazywany bodziec. Co więcej, siła połączeń synaptycznych może się zmieniać w czasie (reguła Hebba mówi nam, że połączenia stają się silniejsze, jeśli są używane wielokrotnie). W tym sensie neuron jest „programowalny".



Rys. 1.1: Biologiczny neuron (https://training.seer.cancer.gov/anatomy/nervous/tissue.html).

Możemy zadać sobie pytanie, czy liczbę neuronów w mózgu rzeczywiście należy określać jako tak „ogromną", jak się zwykle twierdzi. Porównajmy to do urządzeń obliczeniowych z układami pamięci. Liczba neuronów 10^{11} z grubsza odpowiada liczbie tranzystorów w chipie pamięci o pojemności 10 GB, co nie robi na nas specjalnego wrażenia, skoro w dzisiejszych czasach możemy kupić takie urządzenie za około 2$.

Co więcej, prędkość przemieszczania się impulsów nerwowych, która jest wynikiem procesów elektrochemicznych, również nie jest imponująca. Najszybsze sygnały, takie jak te związane z pobudzaniem mięśni, przemieszczają się z prędkością do 120 m/s (osłonki mielinowe są niezbędne do ich osiągnięcia). Sygnały dotykowe osiągają około 80m/s, podczas gdy ból jest przenoszony ze stosunkowo bardzo małymi prędkościami 0,6m/s. To dlatego kiedy upuszczasz młotek na palec u nogi, czujesz to natychmiast, ale ból dociera do mózgu z opóźnieniem ~1s, ponieważ musi pokonać odległość ~1,5m. Z drugiej strony, w urządzeniach elektronicznych sygnał przemieszcza się w przewodach z prędkością rzędu prędkości światła, $\sim 300000\text{km/s} = 3 \times 10^8\text{m/s}$!

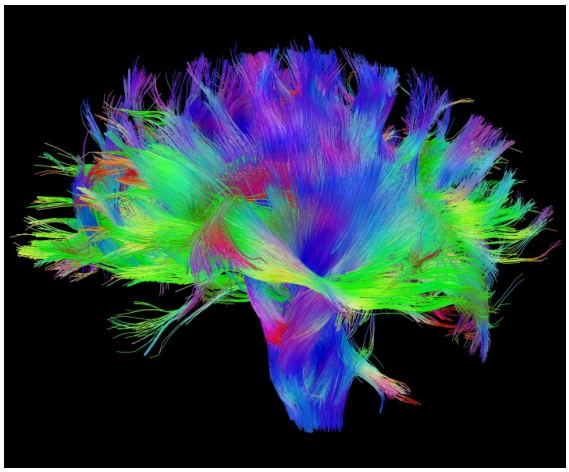
W przypadku ludzi średni czas reakcji wynosi 0,25 s na bodziec wizualny, 0,17 s na bodziec dźwiękowy i 0,15 s na dotyk. W ten sposób ustawienie progowego czasu dla falstartu w sprintach na 0,1 s jest bezpiecznie poniżej możliwej reakcji biegacza. Są to niezwykle powolne reakcje w porównaniu z odpowiedziami elektronicznymi.

Na podstawie zużycia energii przez mózg można oszacować, że neuron kory mózgowej odpala średnio raz na 6 sekund. Jest też mało prawdopodobne, aby przeciętny neuron odpalał częściej niż raz na sekundę. Pomnożenie tej szybkości wyzwalania przez liczbę wszystkich neuronów korowych, $\sim 1.6 \times 10^{10}$, daje około 3×10^9 wyładowań/s w korze, czyli 3GHz. To jest częstotliwość This aktowania typowego chipa procesora! Jeśli więc odpalanie neuronu utożsamić z elementarnym obliczeniem, to tak określona moc mózgu jest z grubsza porównywalna z mocą standardowego

procesora komputerowego.

Powyższe fakty wskazują, że z punktu widzenia naiwnych porównań z chipami krzemowymi ludzki mózg nie jest niczym szczególnym. Co zatem daje nam nasze wyjątkowe zdolności: niesłychanie wydajne rozpoznawanie wzorców wizualnych i dźwiękowych, myślenie, świadomość, intuicję, wyobraźnię? Odpowiedź wiąże się z niesamowicie rozbudowaną architekturą mózgu, w której każdy neuron (jednostka procesora) jest połączony poprzez synapsy średnio aż z 10000 (!) innych neuronów. Ta cecha sprawia, że jest ona radykalnie inna i znacznie bardziej skomplikowana niż architektura składająca się z jednostki sterującej, procesora i pamięci w naszych komputerach (architektura maszyny von Neumanna) . Tam liczba połączeń jest rzędu liczby bitów pamięci, natomiast w ludzkim mózgu jest około 10^{15} połączeń synaptycznych. Jak wspomniano, połączenia można „zaprogramować", aby były silniejsze lub słabsze. Jeśli, dla prostego oszacowania, przybliżylibyśmy siłę połączenia tylko przez dwa stany synapsy, 0 lub 1, to całkowita liczba konfiguracji kombinatorycznych takiego systemu wynosiłaby $2^{10^{15}}$ - „hiper-ogromna" liczba. Większość takich konfiguracji, oczywiście, nigdy nie jest realizowana w praktyce, niemniej jednak liczba możliwych stanów konfiguracyjnych mózgu lub „programów", które może on realizować, jest naprawdę ogromna.

W ostatnich latach, wraz z rozwojem potężnych technik obrazowania, możliwe stało się mapowanie połączeń w mózgu z niespotykaną dotąd rozdzielczością, gdzie widoczne są pojedyncze wiązki nerwów. Wysiłki te są częścią [Projektu Human Connectome] (http://www.humanconnectomeproject.org), którego ostatecznym celem jest dokładne odwzorowanie architektury ludzkiego mózgu. W przypadku znacznie prostszej muszki owocowej, projekt drosophila connectome jest bardzo zaawansowany.



Rys. 1.2: Architektura mózgu włókna istoty białej (z projektu Human Connectome humanconnectomeproject.org)

Ważne: Cecha „ogromnej łączności", z miriadami neuronów służących jako równoległe procesory elementarne, sprawia, że mózg jest zupełnie innym urządzeniem obliczeniowym niż maszyna von Neumanna (tj. nasze codzienne komputery).

1.3 Sieci feed-forward

Neurofizjologiczne badania mózgu dostarczają ważnych wskazówek dla modeli matematycznych stosowanych w sztucznych sieciach neuronowych (**ANN**). I odwrotnie, postępy w algorytmice ANN często przybliżają nas do zrozumienia, jak faktycznie może działać nasz „komputer mózgowy"!

Najprostsze sieci ANN to tak zwane sieci **feed forward**, zilustrowane w Rys. 1.3. Składają się one z warstwy **wejściowej** (czarne kropki), która reprezentuje tylko dane cyfrowe, oraz warstw neuronów (kolorowych kropek). Liczba neuronów w każdej warstwie może być różna. Złożoność sieci i zadań, które może ona realizować, rośnie rzecz jasna wraz z liczbą warstw i liczbą neuronów.

W dalszej części tego rozdziału podamy, w dość skondensownej postaci, kilka ważnych definicji:

Sieci z jedną warstwą neuronów nazywane są sieciami **jednowarstwowymi**. Ostatnia warstwa (jasnoniebieskie kropki) nazywana jest **warstwą wyjściową**. W sieciach wielowarstwowych (więcej niż jedna warstwa neuronowa) war-

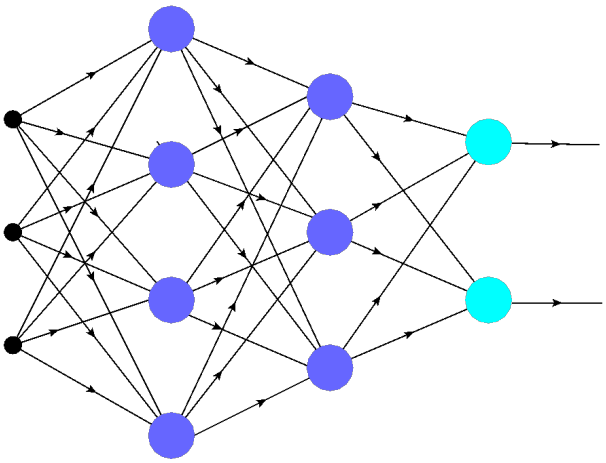
stwy neuronowe poprzedzające warstwę wyjściową (fioletowe kropki) nazywane są **warstwami pośrednimi**. Jeśli liczba warstw jest duża (np. 64, 128, …), mamy do czynienia ze stosowanymi od niedawna „przełomowymi" **głębokimi sieciami**.

Neurony w różnych warstwach nie muszą działać w ten sam sposób, w szczególności neurony wyjściowe mogą zachowywać się inaczej niż pośrednie.

Sygnał z wejścia wędruje po wskazanych strzałkami łączach (krawędziach, połączeniach synaptycznych) do neuronów w kolejnych warstwach. W sieciach typu feed-forward, jak ta na Rys. 1.3, sygnał może poruszać się tylko do przodu (na rysunku od lewej do prawej): od wejścia do pierwszej warstwy neuronowej, od pierwszej do drugiej, i tak dalej, aż do osiągnięcia wyjścia. Nie jest dozwolone cofanie się do poprzednich warstw ani równoległa propagacja pomiędzy neuronami tej samej warstwy. Byłaby to wówczas sieć z **powracaniem**, o czym nieco mówimy w rozdziale *Lateral inhibition*.

Jak szczegółowo opisujemy w kolejnych rozdziałach, wędrujący sygnał jest odpowiednio **przetwarzany** przez neurony, stąd urządzenie wykonuje obliczenia: wejście jest przekształcane w wyjście.

W przykładowej sieci Rys. 1.3 każdy neuron z poprzedniej warstwy jest połączony z każdym neuronem w następnej warstwie. Takie sieci ANN są nazywane **w pełni połączonymi**.



Rys. 1.3: Przykładowa, w pełni połączona sztuczna sieć neuronowa typu feed-forward. Kolorowe plamy reprezentują neurony, a krawędzie uskazują połączenia synaptyczne. Sygnał rozchodzi się od wejścia (czarne kropki), przez neurony w kolejnych warstwach pośrednich (ukrytych) (fioletowe kropki), do warstwy wyjściowej (jasnoniebieskie kropki). Siła połączeń jest kontrolowana przez wagi (hiperparametry) przypisane do krawędzi.

Jak omówimy bardziej szczegółowo późniwj, każda krawędź (połączenie synaptyczne) w sieci ma pewną „siłę" opisaną liczbą o nazwie **waga** (wagi są również określane jako **hiperparametry**). Nawet bardzo małe w pełni połączone sieci, takie jak ta z Rys. 1.3, mają bardzo wiele połączeń (tutaj 30), stąd zawierają dużo hiperparametrów. Tak więc, choć czasami wyglądają niewinnie, ANN są w rzeczywistości bardzo złożonymi systemami wieloparametrycznymi. Co więcej, kluczową cechą jest tutaj nieliniowość odpowiedzi neuronów, co omawiamy w kolejnym rozdziale *Neuron MCP*.

1.4 Dlaczego Python

Wybór języka Python dla prościutkich kodów tego kursu prawie nie wymaga wyjaśnienia. Zacytujmy tylko Tima Petersa:

- Piękne jest lepsze niż brzydkie.
- Jawne jest lepsze niż niejawne.
- Proste jest lepsze niż złożone.
- Złożone jest lepsze niż skomplikowane.
- Liczy się czytelność.

Według SlashData, na świecie jest obecnie ponad 10 milionów programistów używających Pythona, zaraz po jezyku JavaScript (~14 milionów). W szczególności Python okazuje się bardzo praktyczny w zastosowaniach do sieci ANN.

1.4.1 Importowane pakiety

W trakcie tego kursu używamy kilku standardowych pakietów bibliotecznych Pythona do obliczeń numerycznych, wykresów itp. Jak podkreśliliśmy, nie korzystamy z żadnych bibliotek specjalnie dedykowanych sieciom neuronowym. Notebook każdego wykładu zaczyna się od zaimportowania niektórych z tych bibliotek:

```
import numpy as np              # numerical
import statistics as st         # statistics
import matplotlib.pyplot as plt # plotting
import matplotlib as mpl        # plotting
import matplotlib.cm as cm      # contour plots

from mpl_toolkits.mplot3d.axes3d import Axes3D   # 3D plots
from IPython.display import display, Image, HTML # display imported graphics
```

---

neural package

Tworzone podczas tego kursu funkcje, które są później wielokrotnie używane, są umieszczane w pakiecie prywatnej biblioteki **neural**, opisanym w załączniku *Pakiet neural*.

Celem kompatybilności z środowikiem Google Colab, pakiet **neural** importowany jest z repozytorium w następujący sposób:

```
import os.path

isdir = os.path.isdir('lib_nn') # check whether 'lib_nn' exists

if not isdir:
   !git clone https://github.com/bronwojtek/lib_nn.git # cloning the library from␣
 ↪github

import sys
sys.path.append('./lib_nn')

from neural import *            # importing my library package
```

Więcej informacji można znaleźć w dodatku *Pakiet neural*.

Informacja: Dla zwięzłości prezentacji, niektóre zbędne (np. import bibliotek) lub nieistotne fragmenty kodu są obecne tylko w notebookach Jupytera (do pobrania) i nie są ukazywane w książce. Dzięki temu tekst jest krótszy i czytelny.

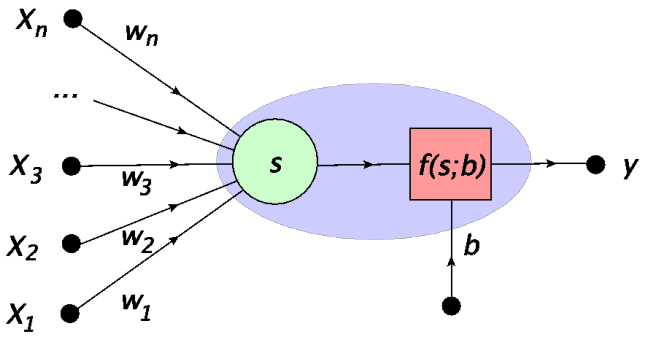
ROZDZIAŁ 2

Neuron MCP

2.1 Definicja

Potrzebujemy podstawowego składnika ANN: sztucznego neuronu. Pierwszy model matematyczny pochodzi od Warrena McCullocha i Waltera Pittsa (MCP)[MP43], którzy zaproponowali go w 1942 roku, a więc na samym początku ery komputerów elektronicznych podczas II wojny światowej. Neuron MCP przedstawiony na Rys. 2.1 jest podstawowym składnikiem wszystkich ANN omawianych w tym kursie. Jest zbudowany na bardzo prostych ogólnych zasadach, inspirowanych przez neuron biologiczny:

- Sygnał wchodzi do jądra przez dendryty z innych neuronów.
- Połączenie synaptyczne dla każdego dendrytu może mieć inną (i regulowaną) siłę (wagę).
- W jądrze sygnał ważony ze wszystkich dendrytów jest sumowany i oznaczony jako s.
- Jeżeli sygnał s jest silniejszy niż pewien zadany próg, to neuron odpala sygnał wzdłuż aksonu, w przeciwnym przypadku pozostaje pasywny.
- W najprostszej realizacji, siła odpalanego sygnału ma tylko dwa możliwe poziomy: włączony lub wyłączony, tj. 1 lub 0. Nie są potrzebne wartości pośrednie.
- Akson łączy się z dendrytami innych neuronów, przekazując im swój sygnał.



Rys. 2.1: Neuron MCP: x_i oznaczają wejście, w_i wagi, s zsumowany sygnał, b próg, a $f(s;b)$ reprezentuje funkcję aktywacji, dającą wyjście $y = f(s;b)$. Niebieski owal otacza cały neuron, jak np. w notacji Rys. 1.3.

Przekładając to na matematyczną receptę, przypisuje się komórkom wejściowym liczby $x_1, x_2 \ldots, x_n$ (punkt danych wejściowych). Siła połączeń synaptycznych jest kontrolowana przez **wagi** w_i. Następnie łączny sygnał jest zdefinio-

wany jako suma ważona

$$s = \sum_{i=1}^{n} x_i w_i.$$

Sygnał staje się argumentem **funkcji aktywacji**, która w najprostszym przypadku przybiera postać funkcji schodkowej

$$f(s;b) = \begin{cases} 1 \text{ dla } s \geq b \\ 0 \text{ dla } s < b \end{cases}$$

Gdy łączny sygnał s jest większy niż próg b, jądro odpala. tj. sygnał idący wzdłuż aksonu wynosi 1. W przeciwnym przypadku wartość generowanego sygnału wynosi 0 (brak odpalenia). Właśnie tego potrzebujemy, aby naśladować biologiczny prototyp!

Istnieje wygodna konwencja, która jest często używana. Zamiast oddzielać próg od danych wejściowych, możemy traktować te liczby wrównoważny sposób. Warunek odpalenia może być trywialnie przekształcony jako

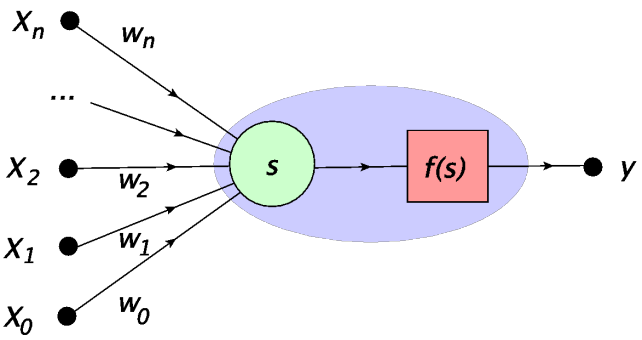
$$s \geq b \rightarrow s - b \geq 0 \rightarrow \sum_{i=1}^{n} x_i w_i - b \geq 0 \rightarrow \sum_{i=1}^{n} x_i w_i + x_0 w_0 \geq 0 \rightarrow \sum_{i=0}^{n} x_i w_i \geq 0,$$

gdzie $x_0 = 1$ i $w_0 = -b$. Innymi słowy, możemy traktować próg jako wagę na krawędzi połączonej z dodatkową komórką z wejściem zawsze ustawionym na 1. Ta notacja jest pokazana na Rys. 2.2. Teraz funkcja aktywacji wynosi po prostu

$$f(s) = \begin{cases} 1 \text{ for } s \geq 0 \\ 0 \text{ for } s < 0 \end{cases}, \tag{2.1}$$

ze wskaźnikiem sumowania w s zaczynającym się 0:

$$s = \sum_{i=0}^{n} x_i w_i = x_0 w_0 + x_1 w_1 + \cdots + x_n w_n. \tag{2.2}$$



Rys. 2.2: Alternatywna, bardziej jednorodna representacja neuronu MCP, z $x_0 = 1$ i $w_0 = -b$.

Wagi $w_0 = -b, w_1, \ldots, w_n$ są ogólnie określane jako **hiperparametry**. Określają one funkcjonalność neuronu MCP i mogą ulegać zmianie podczas procesu uczenia się (trenowania) sieci (patrz kolejne rozdziały). Natomiast są one ustalone podczas używania już wytrenowanej sieci na określonej próbce danych wejściowych.

Ważne: Istotną właściwością neuronów w ANN jest **nieliniowość** funkcji aktywacji. Bez tej cechy neuron MCP reprezentowałby po prostu iloczyn skalarny, a (wielowarstwowe) sieci feed-forward sprowadzałyby się do trywialnego mnożenia macierzy.

2.2 Neuron MCP w Pythonie

Zaimplementujemy teraz model matematyczny neuronu MCP w Pythonie. Rzecz jasna, potrzebujemy tablic (wektorów), które są reprezentowane jako

```
x = [1,3,7]
w = [1,1,2.5]
x
```

```
[1, 3, 7]
```

i (**co ważne**) są indeksowane począwszy od 0, np.

```
x[0]
```

```
1
```

Zauważ, że wypisanie nazwy zmiennej na końcu komórki powoduje wydrukowanie jej zawartości.

Funkcje biblioteczne numpy mają przedrostek **np**, który jest aliasem podanym podczas importu. Funkcje te działają *dystrybucyjnie* na tablice, np.

```
np.sin(x)
```

```
array([0.84147098, 0.14112001, 0.6569866 ])
```

co jest bardzo wygodną własnością przy programowaniu. Mamy też do dyspozycji iloczyn skalarny $x \cdot w = \sum_i x_i w_i$, którego używamy do określenia sygnału s wchodzącego do neuronu MCP:

```
np.dot(x,w)
```

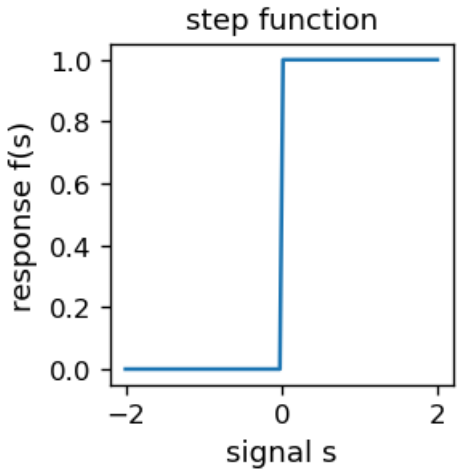
```
21.5
```

Następnie musimy zdefiniować funkcję aktywacji neuronu, która w najprostszej postaci jest funkcją schodkową (2.1):

```
def step(s):        # step function (in the neural library)
     if s > 0:      # condition satisfied
        return 1
     else:          # otherwise
        return 0
```

Funkcja znajduje się też a pakiecie **neural**, zob. *dodatek*. Dla wzrokowców, wykres funkcji schodkowej jest następujący:

```
plt.figure(figsize=(2.3,2.3),dpi=120) # set the size and resolution of the figure

s = np.linspace(-2, 2, 100)    # array of 100+1 equally spaced points in [-2, 2]
fs = [step(z) for z in s]      # corresponding array of function values

plt.xlabel('signal s',fontsize=11)       # axes labels
plt.ylabel('response f(s)',fontsize=11)
plt.title('step function',fontsize=11)   # plot title

plt.plot(s, fs)
plt.show()
```



Ponieważ z definicji $x_0 = 1$, nie chcemy przekazywać tej wartości w argumentach funkcji modelujących neuron MCP. Będziemy zatem dodawać $x_0 = 1$ na początku danych wejściowych, jak w tym przykładzie:

```
x=[5,7]
np.insert(x,0,1) # insert 1 in x at position 0
```

```
array([1, 5, 7])
```

Jesteśmy teraz gotowi by zdefiniwać *neuron MCP*:

```
def neuron(x,w,f=step): # (in the neural library)
    """
    MCP neuron

    x: array of inputs  [x1, x2,...,xn]
    w: array of weights [w0, w1, w2,...,wn]
    f: activation function, with step as default

    return: signal=weighted sum w0 + x1 w1 + x2 w2 +...+ xn wn = x.w
    """
    return f(np.dot(np.insert(x,0,1),w)) # insert x0=1 into x, output f(x.w)
```

Starannie umieszczamy stosowne komentarze w potrójnych cudzysłowach, aby w razie potrzeby móc uzyskać pomoc:

```
help(neuron)
```

```
Help on function neuron in module __main__:

neuron(x, w, f=<function step at 0x7fc98dc57550>)
    MCP neuron

    x: array of inputs  [x1, x2,...,xn]
    w: array of weights [w0, w1, w2,...,wn]
    f: activation function, with step as default

    return: signal=weighted sum w0 + x1 w1 + x2 w2 +...+ xn wn = x.w
```

Zauważ, że funkcja **f** jest argumentem **neuron**u. Argument ten jest domyślnie ustawiony jako **step**, więc nie musi być obecny na liście argumentów. Przykładowe użycie z $x_1 = 3$, $w_0 = -b = -2$ i $w_1 = 1$ to

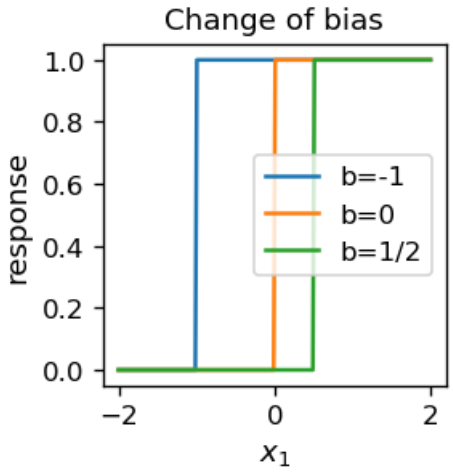
```
neuron([3],[-2,1])
```

```
1
```

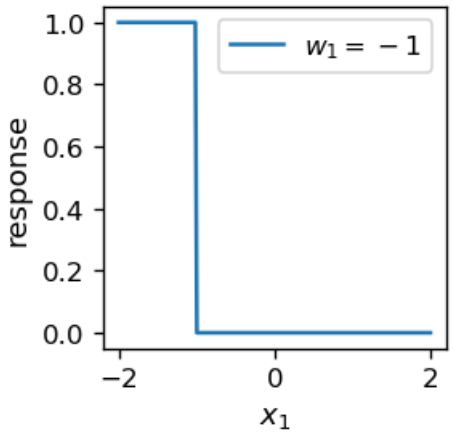
Jak widzimy, w tym przypadku neuron odpalił, ponieważ $s = 1 * (-2) + 3 * 1 > 0$.

Poniżej pokazujemy, jak neuron działa na daną wejściową x_1 wziętą z przedziału $[-2, 2]$. Zmieniamy również wartość progu, aby zilustrować jego rolę: jeśli sygnał $x_1 w_1$ jest większy niż $b = -x_0$, neuron odpala.

```
plt.figure(figsize=(2.3,2.3),dpi=120)

s = np.linspace(-2, 2, 200)
fs1 = [neuron([x1],[1,1]) for x1 in s]       # more function on one plot
fs0 = [neuron([x1],[0,1]) for x1 in s]
fsm12 = [neuron([x1],[-1/2,1]) for x1 in s]

plt.xlabel('$x_1$',fontsize=11)
plt.ylabel('response',fontsize=11)

plt.title("Change of bias",fontsize=11)

plt.plot(s, fs1, label='b=-1')
plt.plot(s, fs0, label='b=0')
plt.plot(s, fsm12, label='b=1/2')
plt.legend()      # legend
plt.show()
```



Kiedy znak wagi w_1 jest ujemny, dostajemy **odwrotne** zachowanie: neuron odpala dla $x_1|w_1| < w_0$:



Informacja: Począwszy od teraz, dla zwięzłości prezentacji, ukrywamy niektóre komórki kodu o powtarzającej się strukturze. Czytelnik może znaleźć pełny kod w oryginalnych notatnikach Jupytera.

Trzeba przyznać, że w ostatnim przykładzie odchodzi się od biologicznego wzorca, ponieważ ujemne wagi nie są możliwe do zrealizowania w biologicznym neuronie. Przyjeta swoboda wzbogaca jednak model matematyczny, który w oczywisty sposób można budować bez ograniczeń biologicznych.

2.3 Funkcje logiczne

Skonstruowawszy neuronu MCP w Pythonie możemy zadać pytanie: *Jaka jest najprostsze (ale wciąż nietrywialne) zastosowanie, w którym możemy go użyć?* Są to [funkcje logiczne](https://en .wikipedia.org/wiki/Boolean_function) lub sieci logiczne utworzone za pomocą sieci neuronów MCP.

Funkcje logiczne z definicji mają argumenty i wartości zawierające się w zbiorze $\{0, 1\}$ lub {Prawda, Fałsz}.

Na rozgrzewkę zacznijmy od zgadywania, gdzie bierzemy neuron o wagach $w = [w_0, w_1, w_2] = [-1, 0.6, 0.6]$ (dlaczego nie). Oznaczmy też $x_1 = p$, $x_2 = q$, zgodnie z tradycyjną notacją zmiennych logicznych, gdzie $p, q \in \{0, 1\}$.

```
print("p q n(p,q)") # print the header
print()             # print space

for p in [0,1]:       # loop over p
    for q in [0,1]:   # loop over q
        print(p,q,"",neuron([p,q],[-1,.6,.6])) # print all cases
```

```
p q n(p,q)

0 0  0
0 1  0
1 0  0
1 1  1
```

Natychmiast rozpoznajemy w powyższym wyniku tabelkę logiczną dla koniunkcji, $n(p, q) = p \wedge q$ lub logicznej operacji **AND**. Jest zupełnie jasne, dlaczego tak działa nasz neuron. Warunek odpalenia $n(p, q) = 1$ wynosi $-1 + p * 0.6 + q * 0.6 \geq 0$ i jest spełniony wtedy i tylko wtedy, gdy $p = q = 1$, co jest definicją koniunkcji logicznej. Oczywiście moglibyśmy użyć tutaj 0.7 zamiast 0.6, lub ogólnie w_1 i w_2 takie, że $w_1 < 1, w_2 < 1, w_1 + w_2 \geq 1$. W terminologii elektronicznej obecny neuron możemy więc nazwać **bramką AND**.

Możemy w ten sposób zdefiniować funkcję

```
def neurAND(p,q): return neuron([p,q],[-1,.6,.6])
```

W podobny sposób możemy zdefiniować inne funkcje logiczne (bramki logiczne) dwóch zmiennych logicznych. W szczególności bramka NAND (negacja koniunkcji) i bramka OR (alternatywa) są realizowane poprzez następujące neurony MCP:

```
def neurNAND(p,q): return neuron([p,q],[1,-0.6,-0.6])
def neurOR(p,q):   return neuron([p,q],[-1,1.2,1.2])
```

Odpowiadają następującym tabelkom logicznym

```
print("p q  NAND OR") # print the header
print()

for p in [0,1]:
```

(ciąg dalszy na następnej stronie)

(kontynuacja poprzedniej strony)

```
    for q in [0,1]:
        print(p,q," ",neurNAND(p,q)," ",neurOR(p,q))
```

```
p q  NAND OR

0 0   1   0
0 1   1   1
1 0   1   1
1 1   0   1
```

2.3.1 Problem z bramką XOR

Bramka XOR, lub **alternatywa wykluczająca**, jest zdefiniowana za pomocą następującej tabelki logicznej:

p	q	$p \oplus q$
0	0	0
0	1	1
1	0	1
1	1	0

Jest to jedna z możliwych funkcji binarnych dwóch argumentów (w sumie mamy 16 różnych funkcji tego rodzaju, dlaczego?). Moglibyśmy teraz próbować dobrać wagi w naszym neuronie, aby zachowywał się jak bramka XOR, ale jesteśmy skazani na porażkę. Oto jej powód:

Z pierwszego wiersza powyższej tabelki wynika, że dla wejścia 0, 0 neuron nie powinien odpalić. Stąd

$w_0 + 0 * w_1 + 0 * w_2 < 0$ lub $-w_0 > 0$.

W przypadku wierszy 2 i 3 neuron musi odpalić, zatem

$w_0 + w_2 \geq 0$ i $w_0 + w_1 \geq 0$.

Dodając stronami te trzy uzyskane nierówności otrzymujemy $w_0 + w_1 + w_2 > 0$. Jednak czwarty rząd tabelki daje $w_0 + w_1 + w_2 < 0$ (brak odpalenia), więc uzyskujemy sprzeczność. Dlatego nie istnieje taki wybor w_0, w_1, w_2, aby neuron działał jak bramka XOR!

Ważne: Pojedynczy neuron MCP nie może działać jak bramka **XOR**.

2.3.2 XOR ze złożenia bramek AND, NAND i OR

Można rozwiązać problem konstrukcji bramki XOR, składając trzy neurony MCP, np.

```
def neurXOR(p,q): return neurAND(neurNAND(p,q),neurOR(p,q))
```

```
print("p q XOR")
print()

for p in [0,1]:
    for q in [0,1]:
        print(p,q,"",neurXOR(p,q))
```

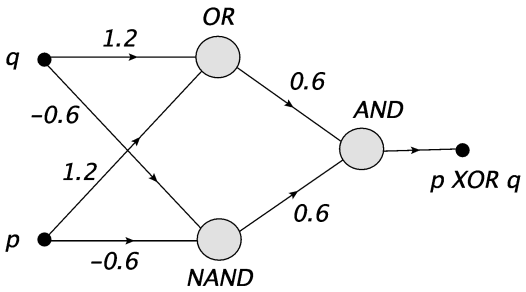
```
p q XOR

0 0  0
```

(ciąg dalszy na następnej stronie)

(kontynuacja poprzedniej strony)

```
0 1  1
1 0  1
1 1  0
```

Powyższa konstrukcja odpowiada prostj sieci Rys. 2.3.



Rys. 2.3: Bramka XOR złożona z neuronów NAND, OR i AND.

Zauważmy, że po raz pierwszy mamy tu do czynienia z siecią posiadającą warstwę pośrednią, składającą się z neuronów NAND i OR. Ta warstwa jest nieodzowna do budowy bramki XOR.

2.3.3 Bramka XOR złożona z bramek NAND

W ramach teorii sieci logicznych udowadnia się, że dowolna sieć (lub dowolna funkcja logiczna) może składać się wyłącznie z bramek NAND lub wyłącznie z bramek NOR. Mówi się, że bramki NAND (lub NOR) są **zupełne**. W szczególności bramka XOR może być skonstruowana jako

[p NAND (p NAND q)] NAND [q NAND (p NAND q)],

co możemy napisać w Pythonie jako

```
def nXOR(i,j): return neurNAND(neurNAND(i,neurNAND(i,j)),neurNAND(j,neurNAND(i,
↪j)))
```

```
print("p q XOR")
print()

for i in [0,1]:
    for j in [0,1]:
        print(i,j,"",nXOR(i,j))
```

```
p q XOR

0 0  0
0 1  1
1 0  1
1 1  0
```

Informacja: Dowodzi się, że sieci logiczne są zupełne w sensie Churcha-Turinga, tj. (jeśli są wystarczająco duże) mogą wykonać każde możliwe obliczenie. Ta własność jest bezpośrednio przenoszona na sieci ANN. Historycznie, było to podstawowe odkrycie przełomowego artykułu MCP [MP43].

Wniosek

Dostatecznie duże ANN mogą wykonac każde obliczenie!

2.4 Ćwiczenia

Skonstruuj (wszystko w Pythonie)

- bramkę realizująca koniunkcję kilku zmiennych logicznych;
- bramki NOT, NOR;
- bramki OR, AND i NOT poprzez złożenie bramek NAND;
- pół sumator i pełny sumator,

jako sieci neuronów MCP.

ROZDZIAŁ 3

Modele pamięci

3.1 Pamieć skojarzeniowa (heteroasocjacyjna)

3.1.1 Skojarzenia par

Przechodzimy teraz do dalszych ilustracji elementarnych możliwości ANN, opisujących dwa bardzo proste modele pamięci oparte na algebrze liniowej, uzupełnione o (nieliniowe) filtrowanie. Mówiąc o pamięci, na miejscu jest słowo przestrogi. Mamy tu do czynienia z dość uproszczonymi narzędziami, które są dalekie od złożonego i dotychczas niezrozumiałego mechanizmu pamięci działającego w naszym mózgu. Obecne rozumienie jest takie, że te mechanizmy obejmują sprzężenie zwrotne w sieciach, co wykracza poza rozważane tutaj sieci typu feed-forward.

Pierwszy rozważany model dotyczy tzw. pamięci **heterasocjacyjnej**, w której niektóre obiekty (tutaj graficzne symbole bitmapowe) są kojarzone w pary. Dla konkretnego przykładu bierzemy zbiór pięciu symboli graficznych {A, a, I, i, Y} i definiujemy dwa skojarznia par: A $\leftrightarrow$ a oraz I $\leftrightarrow$ i, czyli pomiędzy różnymi (hetero) symbolami. Symbol Y pozostaje nieskojarzony.

Symbole są zdefiniowane jako 2-wymiarowe (12×12) tablice pikseli, np.

```
A = np.array([
    [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
    [0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0],
    [0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0],
    [0, 0, 0, 0, 1, 1, 1, 0, 0, 0, 0, 0],
    [0, 0, 0, 0, 1, 0, 1, 0, 0, 0, 0, 0],
    [0, 0, 0, 1, 0, 0, 0, 1, 0, 0, 0, 0],
    [0, 0, 0, 1, 0, 0, 0, 1, 0, 0, 0, 0],
    [0, 0, 1, 1, 1, 1, 1, 1, 1, 0, 0, 0],
    [0, 0, 1, 0, 0, 0, 0, 0, 1, 0, 0, 0],
    [0, 1, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0],
    [0, 1, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0],
    [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]])
```

Pozostałe symbole są zdefiniowane podobnie.

Użyjemy standardowego pakietu do rysowania, zaimportowanego wcześniej. Cały zestaw naszych symboli wygląda jak poniżej, z kolorem żółtym=1 i fioletowym=0:

```
sym=[A,a,ii,I,Y]    # array of symbols, numbered from 0 to 4
```

```
plt.figure(figsize=(16, 6)) # figure with horizontal and vertical size

for i in range(1,6):      # loop over 5 figure panels, i is from 1 to 5
    plt.subplot(1, 5, i) # panels, numbered from 1 to 5
    plt.axis('off')       # no axes
    plt.imshow(sym[i-1]) # plot symbol, numbered from 0 to 4

plt.show()
```

Ostrzeżenie: W Pythonie zakres range (i, j) zawiera i, ale nie obejmuje j, tj. równa się tablicy $[i, i+1, \ldots, j-1]$. Ponadto range(i) obejmuje $0, 1, \ldots, i-1$. Różni się to od konwencji przyjętej w niektórych innych językach programowania.

Wygodniej jest pracować nie z powyższymi tablicami dwuwymiarowymi, ale z jednowymiarowymi wektorami uzyskanymi za pomocą tzw. procedury **spłaszczania**, gdzie macierz jest „pocięta" wzdłuż swoich wierszy, złożonych w wektor. Na przykład

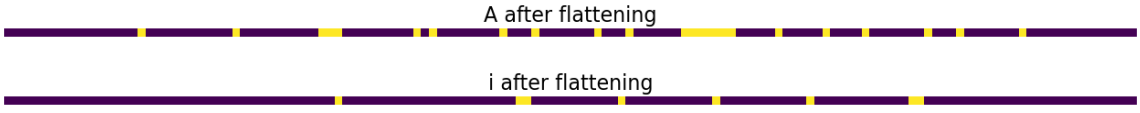
```
t=np.array([[1,2,3],[0,4,0],[3,2,7]]) # a matrix
print(t)
print(t.flatten())                    # matrix flattened into a vector
```

```
[[1 2 3]
 [0 4 0]
 [3 2 7]]
[1 2 3 0 4 0 3 2 7]
```

A zatem przeprowadzamy spłaszczenie na naszym zestawie symboli

```
fA=A.flatten()
fa=a.flatten()
fi=ii.flatten()
fI=I.flatten()
fY=Y.flatten()
```

aby otrzymać, np.



Zaletą pracy z wektorami jest to, że możemy użyć wbudowanego iloczynu skalarnego. Zauważmy, że tutaj iloczyn skalarny wektorów odpowiadających dwóm symbolom jest po prostu równy liczbie wspólnych żółtych pikseli. Na przykład dla spłaszczonych symboli A oraz i, narysowanych powyżej, mamy tylko dwa wspólne żółte piksele:

```
np.dot(fA,fi)
```

```
2
```

Jasne jest, że można użyć iloczynu skalarnego jako miary podobieństwa między symbolami. Aby poniżej przedstawiony model pamięci skojarzeniowej działał, symbole nie powinny być zbyt podobne, ponieważ mogą być wtedy „mylone”.

3.1.2 Macierz pamięci

Następna koncepcja algebraiczna, której potrzebujemy, to **iloczyn zewnętrzny**. Dla dwóch wektorów v i w jest on zdefiniowany jako $vw^T = v \otimes w$ (w przeciwieństwie do iloczynu skalarnego, gdzie $w^T v = w \cdot v$). Tutaj T oznacza transpozycję. Wynikiem jest macierz z liczbą wierszy równą długości v i liczbą kolumn równą długości w.

Na przykład dla

$$v = \begin{pmatrix} v_1 \\ v_2 \\ v_3 \end{pmatrix}, \quad w = \begin{pmatrix} w_1 \\ w_2 \end{pmatrix},$$

mamy

$$v \otimes w = vw^T = \begin{pmatrix} v_1 \\ v_2 \\ v_3 \end{pmatrix} (w_1, w_2) = \begin{pmatrix} v_1 w_1 & v_1 w_2 \\ v_2 w_1 & v_2 w_2 \\ v_3 v_1 & v_3 w_2 \end{pmatrix}.$$

(przypomnij sobie z algebry, że mnożymy „wiersze przez kolumny”). w **numpy**

```
print(np.outer([1,2,3],[2,7])) # outer product of two vectors
```

```
[[ 2  7]
 [ 4 14]
 [ 6 21]]
```

Następnie konstruujemy **macierz pamięci** potrzebną do modelowania pamięci heteroasocjacyjnej. Załóżmy najpierw, dla uproszczenia notacji, że mamy tylko dwa skojarzenia: $a \to A$ i $b \to B$. Niech

$$M = Aa^T/a \cdot a + Bb^T/b \cdot b.$$

Wówczas

$$Ma = A + B\, a \cdot b/b \cdot a,$$

i jeśli a and b byłyby **ortogonalne**, tj. $a \cdot b = 0$, to

$Ma = A$,

dając dokładne skojarzenie. Podobnie otrzymalibyśmy $Mb = B$. Ponieważ jednak w ogólnym przypadku wektory nie są dokładnie ortogonalne, generowany jest pewien błąd $B\, b \cdot a/a \cdot a$ (dla asocjacji a). Zwykle jest on mały, jeśli liczba pikseli w naszych symbolach jest duża, a symbole są, ogólnie rzecz biorąc, niezbyt do siebie podobne (nie mają zbyt wielu wspólnych pikseli). Jak wkrótce zobaczymy, pojawiający się błąd można wydajnie „odfiltrować” odpowiednią funkcją aktywacji neuronów.

Wracając do naszego konkretnego przypadku, potrzebujemy zatem czterech członów w M, ponieważ $a \to A$, $A \to a$, $I \to i$ oraz $i \to I$:

```
M=(np.outer(fA,fa)/np.dot(fa,fa)+np.outer(fa,fA)/np.dot(fA,fA)
   +np.outer(fi,fI)/np.dot(fI,fI)+np.outer(fI,fi)/np.dot(fi,fi)); # associated␣
↪pairs
```

Teraz, jako test jak to działa, dla każdego spłaszczonego symbolu s obliczamy Ms. Wynikiem jest wektor, który chcemy przywrócić do postaci tablicy pikseli 12×12. Operacją odwrotną do spłaszczania w Pythonie jest **reshape**. Na przykład

```
tt=np.array([1,2,3,5]) # test vector
print(tt.reshape(2,2)) # cutting into 2 rows of length 2
```

```
[[1 2]
 [3 5]]
```

Dla naszych wektorów mamy

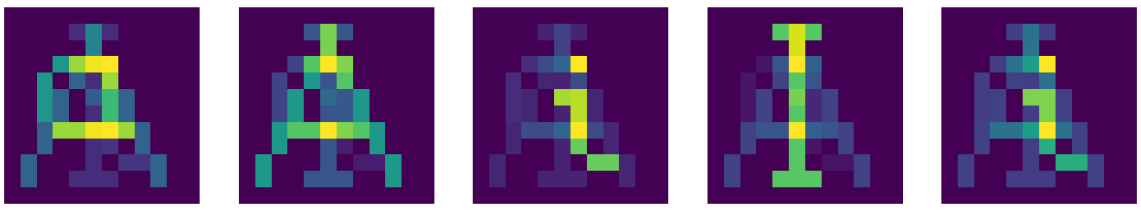
```
Ap=np.dot(M,fA).reshape(12,12)
ap=np.dot(M,fa).reshape(12,12)
Ip=np.dot(M,fI).reshape(12,12)
ip=np.dot(M,fi).reshape(12,12)
Yp=np.dot(M,fY).reshape(12,12) # we also try the unassociated symbol Y

symp=[Ap,ap,Ip,ip,Yp]          # array of associated symbols
```

W przypadku skojarzenia z A (które powinno wynosić a) procedura daje nastepujący wynik (stosujemy tu dla ładniejszego wydruku zaokrąglanie do 2 cyfr dziesiętnych poprzez **np.round**)

```
print(np.round(Ap,2)) # pixel map for the association of the symbol A
```

```
[[0.   0.   0.   0.   0.   0.   0.   0.   0.   0.   0.   0.  ]
 [0.   0.   0.   0.   0.25 0.85 0.25 0.   0.   0.   0.   0.  ]
 [0.   0.   0.   0.   0.   0.85 0.   0.   0.   0.   0.   0.  ]
 [0.   0.   0.   1.   1.6  1.85 1.89 0.   0.   0.   0.   0.  ]
 [0.   0.   1.   0.   0.6  0.25 1.6  0.   0.   0.   0.   0.  ]
 [0.   0.   1.   0.6  0.   0.54 1.29 0.6  0.   0.   0.   0.  ]
 [0.   0.   1.   0.6  0.   0.25 1.29 0.6  0.   0.   0.   0.  ]
 [0.   0.   0.6  1.6  1.6  1.85 1.89 1.6  0.6  0.   0.   0.  ]
 [0.   0.   0.6  0.   0.   0.25 0.29 0.   0.6  0.   0.   0.  ]
 [0.   0.6  0.   0.   0.   0.25 0.   0.29 0.29 0.6  0.   0.  ]
 [0.   0.6  0.   0.   0.25 0.25 0.25 0.   0.   0.6  0.   0.  ]
 [0.   0.   0.   0.   0.   0.   0.   0.   0.   0.   0.   0.  ]]
```

Zauważamy, że intensywność pikseli niekoniecznie jest teraz równa 0 lub 1, tak jak w oryginalnych symbolach. Przedstawienie graficzne wygląda następująco:



Powinniśmy być w stanie zobaczyć na powyższym obrazku sekwencję a, A, i, I oraz nic szczególnego w powiązaniu z Y. Prawie tak jest, ale sytuacja nie jest idealna ze względu na omówiony powyżej błąd wynikający z nieortogonalności.

3.1.3 Nakładanie filtra

Wynik znacznie się poprawi, gdy do powyższych map pikseli zostanie zastosowany filtr. Patrząc na powyższy rysunek zauważamy, że powinniśmy pozbyć się „słabych cieni", a pozostawić tylko piksele o wystarczającej sile, które następnie powinny otrzymac warość 1. Innymi słowy, piksele poniżej progu filtra b powinny zostać zresetowane do 0, a te powyżej lub równe b powinny zostać zresetowane do 1. Można to zgrabnie osiągnąć za pomocą naszego **neuronu** z rozdz. *Neuron MCP w Pythonie*. Funkcja ta została umieszczona w bibliotece **neural** (patrz *Dodatek*).

A zatem definiujemy filtry jako neurony MCP o wagach $w_0 = -b$ i $w_1 = 1$:

```
def filter(a,b): # a - symbol (2-dim pixel array), b - bias
    n=len(a)     # number of rows (and columns)
    return np.array([[func.neuron([a[i,j]],[-b,1]) for j in range(n)] for i in␣
↪range(n)])
       # 2-dim array with the filter applied
```

Działając na symbol Ap z odpowiednio dobranym $b = 0.9$ (przyjęty poziom progu jest tutaj bardzo istotny), uzyskujemy

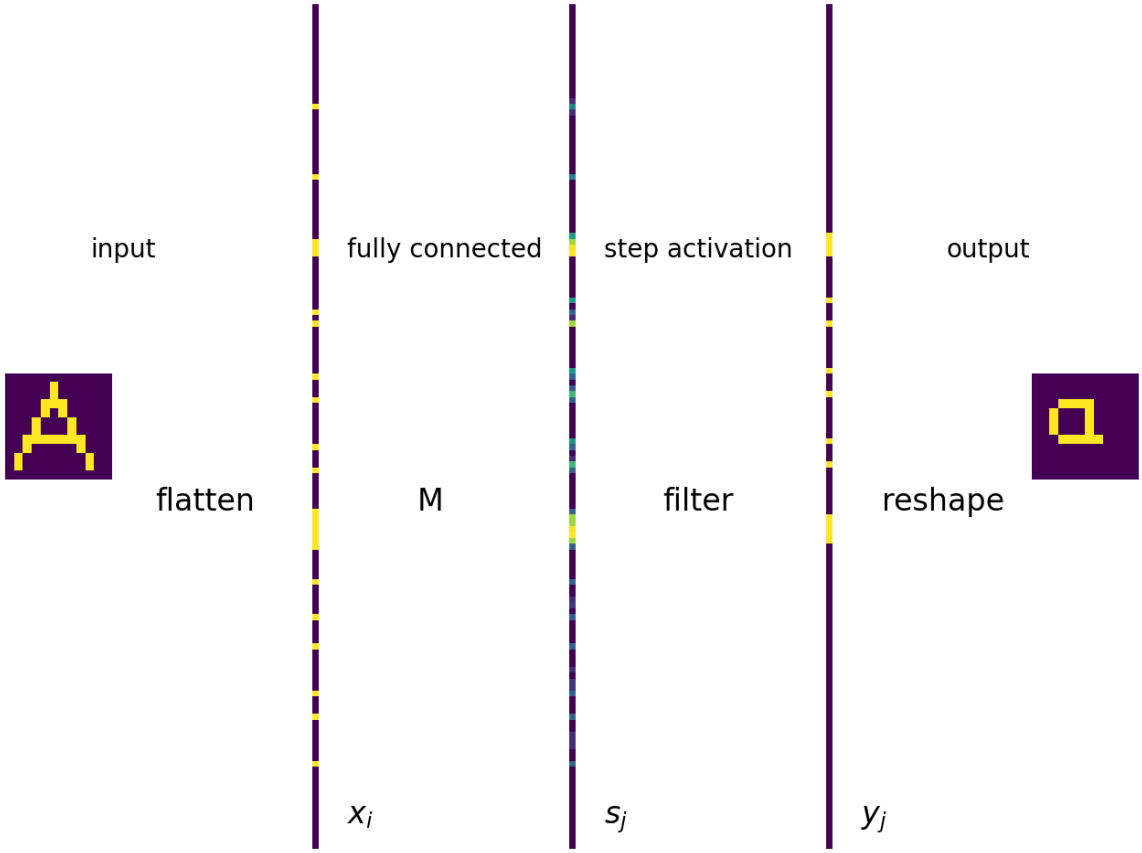
```
print(filter(Ap,.9))
```

```
[[0 0 0 0 0 0 0 0 0 0 0 0]
 [0 0 0 0 0 0 0 0 0 0 0 0]
 [0 0 0 0 0 0 0 0 0 0 0 0]
 [0 0 0 1 1 1 1 0 0 0 0 0]
 [0 0 1 0 0 0 1 0 0 0 0 0]
 [0 0 1 0 0 0 1 0 0 0 0 0]
 [0 0 1 0 0 0 1 0 0 0 0 0]
 [0 0 0 1 1 1 1 1 0 0 0 0]
 [0 0 0 0 0 0 0 0 0 0 0 0]
 [0 0 0 0 0 0 0 0 0 0 0 0]
 [0 0 0 0 0 0 0 0 0 0 0 0]
 [0 0 0 0 0 0 0 0 0 0 0 0]]
```

gdzie możemy zauważyć „czysty" symbol a. Sprawdzamy, czy faktycznie filtrowanie działa tak doskonale we wszystkich naszych skojarzeniach:

Można łatwo podać reprezentację właśnie przedstawionego modelu pamięci heteroasocjacyjnej jako **jednowarstwową** sieć neuronów MCP. Na poniższym wykresie ukazujemy wszystkie operacje, idąc od lewej strony do prawej. Symbol wejściowy jest spłaszczony. Warstwy wejściowa i wyjściowa są w pełni połączone krawędziami (których nie pokazano) łączącymi komórki wejściowe z neuronami w warstwie wyjściowej. Wagi krawędzi są równe elementom macierzy M_{ij}, oznaczonej symbolem M. Funkcja aktywacji jest taka sama dla wszystkich neuronów i ma postać funkcji schodkowej.

Na dole rysunku wskazujemy elementy wektora wejściowego x_i, sygnału docierającego do neuronu j, tj. $s_j = \sum_i x_i M_{ij}$ oraz wynik końcowy $y_j = f(s_j)$.



Podsumowanie modelu pamieci heteroassociatywnej

1. Zdefiniuj pary skojarzonych symboli i skonstruuj macierz pamięci M.
2. Wejście to symbol w postaci 2-wymiarowej tablicy pikseli o wartościach 0 lub 1.
3. Spłaszcz symbol do wektora, który tworzy warstwę danych wejściowych x_i.
4. Macierz wag w pełni połączonej sieci ANN to M.
5. Sygnał wchodzący do neuronu j w warstwie wyjściowej to $s_j = \sum_i x_i M_{ij}$.
6. Funkcja aktywacji to funkcja schodkowa z odpowiednio dobranym progiem. Daje ona $y_j = f(s_j)$.
7. Potnij wektor wyjściowy na macierz pikseli, która stanowi ostateczny wynik. Powinien to być symbol skojarzony z symbolem na wejściu.

3.2 Pamieć autoasocjatywna

3.2.1 Samo-skojarzenia

Model pamięci autoasocjacyjnej jest w bliskiej analogii do przypadku pamięci skojarzeniowej, ale teraz każdy symbol jest kojarzony **z samym sobą**. Dlaczego robimy coś takiego, stanie się jasne, gdy weźmiemy pod uwagę zniekształcone dane wejściowe. Definiujemy macierz asocjacji w następujący sposób:

```
Ma=(np.outer(fA,fA)/np.dot(fA,fA)+np.outer(fa,fa)/np.dot(fa,fa)
    +np.outer(fi,fi)/np.dot(fi,fi)+np.outer(fI,fI)/np.dot(fI,fI))
```

Po przemnożeniu spłaszczonego symbolu przez macierz Ma i przefiltrowaniu (wszystkie kroki jak w przypadku skojarzeniowym) otrzymujemy poprawnie oryginalne symbole (poza Y, który nie był z niczym powiązany).

3.2.2 Zniekształcone symbole

Teraz wyobraźmy sobie, że oryginalny symbol zostaje częściowo zniszczony, a niektóre piksele są losowo zmieniane z 1 na 0 i odwrotnie. Tworzymy roboczą kopię oryginalnych symboli, a nastepnie losowo przekłamujemy pewną liczbe pikseli (tutaj 8):

```
sym2=np.copy(sym)                   # copy of symbols

ne=8                                # number of alterations

for s in sym2:                      # loop over symbols
    for _ in range(ne):             # loop over alterations
        i=np.random.randint(0,12)   # random row
        j=np.random.randint(0,12)   # random column
        s[i,j]=1-s[i,j]             # trick to switch between 1 and 0
```

Po tym zniszczeniu symbole wejściowe wyglądają tak:

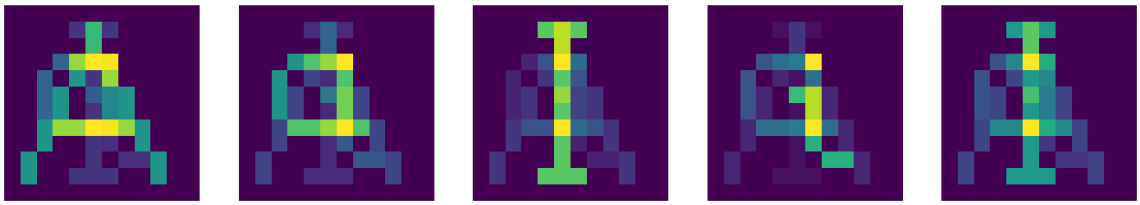
3.2.3 Odtworzenie symboli

Następnie stosujemy nasz model pamięci autoasocjacyjnej do wszystkich „zniszczonych” symboli (które najpierw spłaszczamy):

```
fA2=sym2[0].flatten()
fa2=sym2[1].flatten()
fi2=sym2[2].flatten()
fI2=sym2[3].flatten()
fY2=sym2[4].flatten()
```

```
Ap=np.dot(Ma,fA2).reshape(12,12)
ap=np.dot(Ma,fa2).reshape(12,12)
Ip=np.dot(Ma,fI2).reshape(12,12)
ip=np.dot(Ma,fi2).reshape(12,12)
Yp=np.dot(Ma,fY2).reshape(12,12)

symp=[Ap,ap,Ip,ip,Yp]
```

co daje



Po przefiltrowaniu z odpowiednio dobranym progiem tutaj ($b = 0.8$) odzyskujemy originalne symbole:

```
plt.figure(figsize=(16, 6))

for i in range(1,6):
    plt.subplot(1, 5, i)
    plt.axis('off')
    plt.imshow(filter(symp[i-1],0.8)) # plot filtered symbol

plt.show()
```

Zastosowanie algorytmu może zatem odszyfrować „zniszczony" tekst lub, bardziej ogólnie, zapewnić mechanizm korekcji błędów. Metoda działa, gdy przekłamań nie jest zbyt wiele.

Podsumowanie modelu pamieci autoasocjatywnej

1. Skonstruuj macierz pamięci Ma.
2. Wejście to symbol w postaci 2-wymiarowej tablicy pikseli o wartościach 0 lub 1, gdzie pewna liczba pikseli jest losowo przekłamana.
3. Spłaszcz symbol do wektora, który tworzy warstwę danych wejściowych x_i.
4. Macierz wag w pełni połączonej sieci ANN to Ma.
5. Sygnał wchodzący do neuronu j w warstwie wyjściowej to $s_j = \sum_i x_i M_{ij}$.
6. Funkcja aktywacji to funkcja schodkowa z odpowiednio dobranym progiem. Daje ona $y_j = f(s_j)$.
7. Potnij wektor wyjściowy na macierz pikseli, która stanowi ostateczny wynik. Powinien zostać przywrócony oryginalny symbol.

Ważne: Konkluzja: ANN z jedną warstwą neuronów MPC mogą służyć jako bardzo proste modele pamięci!

Zauważmy jednak, że skonstruowaliśmy macierze pamięciowe algebraicznie, niejako zewnętrznie. Dlatego sieć tak naprawdę nie nauczyła się skojarzeń z doświadczenia. Są na to sposoby, ale wymagają one bardziej zaawansowanych metod (patrz np. [FS91]), podobne do omówionych w kolejnych częściach tego wykładu.

Informacja: Implementacja omawianych modeli pamięci w programie Mathematica znajduje się w [Fre93] (https://library.wolfram.com/infocenter/Books/3485) oraz we wspomnianych już wykładach Daniela Kerstena.

3.3 Ćwiczenia

Pobaw się kodem z wykładu i

- dodawaj coraz więcej symboli;
- zmieniaj poziom filtra;
- zwiększ liczbę przekłamań w modelu autoasosjatywnym.

Omów swoje spostrzeżenia i przedyskutuj ograniczenia modeli.

ROZDZIAŁ 4

Perceptron

4.1 Uczenie nadzorowane

W poprzednich rozdziałach pokazaliśmy, że nawet najprostsze sieci ANN mogą wykonywać przydatne zadania (emulować sieci logiczne lub dostarczać proste modele pamięci). Ogólnie rzecz biorąc, każdy ANN ma

- pewną **architekturę**, czyli liczbę warstw, liczbę neuronów w każdej warstwie, schemat połączeń między neuronami (w pełni połączone lub nie, feed-forward, rekurencyjne, …);
- **wagi (hiperparametry)** na połącznieach, z określonymi wartościami definiującymi funkcjonalność sieci.

Podstawowym pytaniem praktycznym jest to, jak ustawić (dla danej architektury) wagi tak, aby żądany cel funkcjonalności sieci został zrealizowany, tj. dla określonych danych wejściowych uzuskać pożądany wynik na wyjściu. W zadaniach omówionych wcześniej wagi mogą być skonstruowane *a priori*, czy to dla bramek logicznych, czy dla modeli pamięci. Jednak dla bardziej skomplikowanych aplikacji chcemy mieć „łatwiejszy" sposób określania wag. Co więcej, dla skomplikowanych problemów „teoretyczne" określenie wag a priori nie jest w ogóle możliwe. To podstawowy powód, dla którego wymyślono **algorytmy uczenia się** sieci, które „automatycznie" dostosowują wagi na podstawie dostępnych danych.

W tym rozdziale rozpoczynamy badanie takich algorytmów, poczynając od podejścia **uczenia nadzorowanego**, stosowanego m.in. do klasyfikacji danych.

Uczenie nadzorowane

W tej strategii dane muszą posiadać **etykiety**, które a priori określają poprawną kategorię dla każdego punktu. Pomyślmy na przykład o zdjęciach zwierząt (dane lub cechy, ang. features) i ich opisach (kot, pies,…), które nazywane są etykietami (ang. labels). Te etykietowane dane są następnie dzielone na próbkę **szkoleniową** i próbkę **testową**.

Podstawowe kroki uczenia nadzorowanego dla danej ANN są następujące:

- Zainicjuj w jakiś sposób wagi, na przykład losowo lub na zero.
- Odczytuj kolejno punkty danych z próbki szkoleniowej i przepuszczaj je przez swoją sieć ANN. Otrzymana odpowiedź może różnić się od prawidłowej, zawartej w etykiecie. W takim przypadku wagi są zmieniane zgodnie z konkretną receptą (o czym później).
- W razie potrzeby powtórz poprzedni krok. Zazwyczaj wagi zmienia się coraz mniej w miarę postępu algorytmu.
- Zakończ szkolenie sieci po osiągnięciu kryterium zatrzymania (wagi nie zmieniają się już znacznie lub została osiągnięta maksymalna liczba iteracji).

- Przetestuj tak wyszkoloną ANN na próbce testowej.

Jeśli jesteśmy zadowoleni, mamy pożądaną wyszkoloną sieć ANN wykonującą określone zadanie (takie jak np. klasyfikacja danych), której można teraz używać na nowych, nieetykietowanych danych. Jeśli nie, możemy inaczej podzielić próbkę na część szkoleniową i testową, po czym powtórzyć procedurę uczenia od początku. Możemy także spróbować pozyskać więcej danych (co może być kosztowne), lub też zmienić architekturę sieci.

Termin „nadzorowany" pochodzi z interpretacji procedury, w której etykiety posiadane są przez „nauczyciela", który w ten sposób wie, które odpowiedzi są prawidłowe, a które błędne i który **nadzoruje** w ten sposób proces szkolenia. Oczywiście program komputerowy ma wbudowanego nauczyciela. tj. „nadzoruje się" sam.

4.2 Perceptron jako klasyfikator binarny

Najprostszy algorytm uczenia nadzorowanego to perceptron, wymyślony w 1958 roku przez Franka Rosenblatta. Może służyć m.in. do konstruowania **klasyfikatorów binarnych** danych. *Binarny* oznacza, że sieć służy do oceny, czy element danych ma określoną cechę, czy nie - są tylko dwie możliwości. Klasyfikacja wieloetykietowa jest również możliwa w przypadku ANN (patrz ćwiczenia), ale nie omawiamy jej tutaj.

Uwaga

Termin *perceptron* jest również używany dla ANN (bez lub z warstwami pośrednimi) składających się z neuronów MCP (por. rys. Rys. 1.3 i Rys. 2.1), na których wykonywany jest algorytm perceptronu.

4.2.1 Próbka ze znaną regułą klasyfikacji

Na początek potrzebujemy danych treningowych, które wygenerujemy jako losowe punkty w kwadracie. Zatem współrzędne punktu, x_1 i x_2, należą do przedziału $[0, 1]$. Definiujemy dwie kategorie: jedną dla punktów leżących powyżej linii $x_1 = x_2$ (nazwijmy je różowymi) oraz drugą dla punktów leżących poniżej tej linii (niebieskie). Podczas losowego generowania danych sprawdzamy, czy $x_2 > x_1$ czy nie i przypisujemy odpowiednią **etykietę** do każdego punktu, równą odpowiednio 1 lub 0. Te etykiety są oczekiwanymi „prawdziwymi" odpowiedziami sieci po jej wyszkoleniu.

Funkcja generująca opisany powyżej punkt danych z etykietą to

```
def point():      # generates random coordinates x1, x2, and 1 if x2>x1, 0␣
 ↪otherwise
    x1=np.random.random()          # random number from the range [0,1]
    x2=np.random.random()
    if(x2>x1):                     # condition met
        return np.array([x1,x2,1]) # add label 1
    else:                          # not met
        return np.array([x1,x2,0]) # add label 0
```

Generujemy **próbkę szkoleniową**, składającą się z **npo**=300 etykietowanych punktów danych:

```
npo=300 # number of data points in the training sample

print('  x1         x2         label')       # header
samp=np.array([point() for _ in range(npo)]) # training sample, _ is dummy␣
 ↪iterator
print(samp[:5, :])                           # first 5 data points
```

```
  x1         x2         label
 [[0.32934056 0.39891895 1.        ]
  [0.87245402 0.23216823 0.        ]
```

(ciąg dalszy na następnej stronie)

(kontynuacja poprzedniej strony)

```
[0.34298363 0.15572279 0.        ]
[0.85658971 0.99224611 1.        ]
[0.78826375 0.03085404 0.        ]]
```

Pętle w tablicy

W Pythonie można wygodnie zdefiniować tablicę poprzez pętlę, np. [i**2 for i in range(4)] daje [1,4,9].

W pętlach, jeśli wskaźnik nie występuje jawnie w wyrażeniu, można użyć symbolu _ , np.

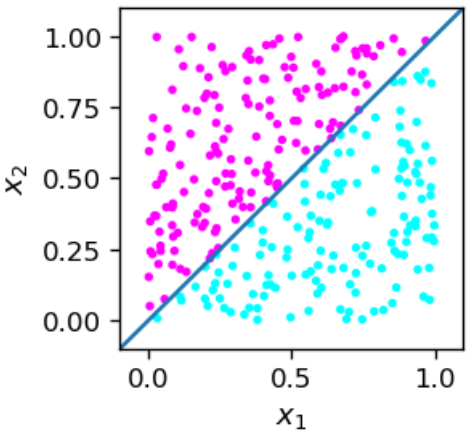
[point() for _ in range(npo)]

Zakresy w tablicach

Aby nie drukować niepotrzebnie bardzo długiej tabeli, po raz pierwszy użyliśmy powyżej **zakresów dla wskaźników tablic**. Np. 2:5 oznacza od 2 do 4 (przypomnijmy, że ostatni jest wykluczony!), :5 - od 0 do 4, 5: - od 5 do końca, wreszcie : - wszystkie elementy.

Nasze wygenerowane dane przedstawia graficznie poniższy rysunek. Wykreślamy również linię $x_2 = x_1$, która oddziela niebieskie i różowe punkty. W tym przypadku podział jest możliwy a priori (znamy regułę) w sposób dokładny.

```
plt.figure(figsize=(2.3,2.3),dpi=120)
plt.xlim(-.1,1.1)                                   # axes limits
plt.ylim(-.1,1.1)
plt.scatter(samp[:,0],samp[:,1],c=samp[:,2],        # label determines the color
            s=5,cmap=mpl.cm.cool)                   # point size and color

plt.plot([-0.1, 1.1], [-0.1, 1.1])                  # separating line

plt.xlabel('$x_1$',fontsize=11)
plt.ylabel('$x_2$',fontsize=11)

plt.show()
```



Zbiory liniowo rozłączne

Dwa zbiory punktów (tutaj niebieski i różowy) na płaszczyźnie, które można rozdzielić linią prostą, nazywamy **liniowo rozłącznymi** (separowalnymi). W trzech wymiarach zbiory muszą być separowalne płaszczyzną, ogólnie w n wymiarach zbiory muszą być separowalne za pomocą $n-1$ wymiarowej hiperpłaszczyzny.

Analitycznie, jeżeli punkty w przestrzeni n wymiarowej mają współrzędne $(x_1, x_2, \ldots, x_n)$, to można dobrać parametry $(w_0, w_1, \ldots, w_n)$ w taki sposób, aby zbiór jeden punktów spełniał warunek

$$w_0 + x_1 w_1 + x_2 w_2 + \ldots x_n w_n > 0 \tag{4.1}$$

a drugi warunek przeciwny, ze znakiem $>$ zastąpionym przez $\leq$.

A teraz kluczowa, choć oczywista obserwacja: powyższa nierówność jest dokładnie warunkiem zaimplementowanym w [neuronie MCP](laboratorium MCP) (ze schodkową funkcją aktywacji) w konwencji Rys. 2.2! Możemy więc zrealizować warunek (4.1) za pomocą funkcji **neuron** z biblioteki **neural**.

W naszym przykładzie dla różowych punktów, według konstrukcji,

$$x_2 > x_1 \rightarrow s = -x_1 + x_2 > 0$$

skąd, używając równ. (4.1), możemy od razu odczytać

$$w_0 = 0, \;\; w_1 = -1, w_2 = 1.$$

Zatem funkcja **neuron** dla punktu próbki p jest używana w następujący sposób:

```
p=[0.6,0.8]       # sample point with x_2 > x_1
w=[0,-1,1]        # weights as given above

func.neuron(p,w)
```

```
1
```

Neuron, odpalił, więc punkt p jest różowy.

Wniosek

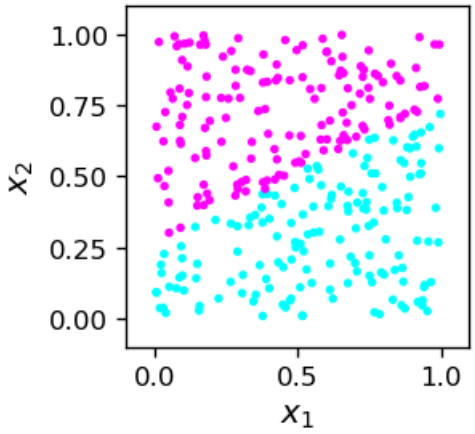
Pojedynczy neuron MCP z odpowiednio dobranymi wagami może być użyty jako klasyfikator binarny dla n-wymiarowych danych separowalnych.

4.2.2 Próbka o nieznanej regule klasyfikacji

W tym miejscu czytelnik może być nieco zwiedziony pozorną błahością wyników. Wątpliwości mogą wynikać z tego, że w powyższym przykładzie od początku znaliśmy regułę określającą dwie klasy punktów ($x_2 > x_1$, lub odwrotnie). Jednak w ogólnej sytuacji „z prawdziwego życia" zwykle tak nie jest! Wyobraź sobie, że napotykamy (etykietowane) dane **samp2** wyglądające tak:

```
print(samp2[:5])
```

```
[[0.50896192 0.26237741 0.        ]
 [0.50775256 0.1093865  0.        ]
 [0.44707124 0.04838339 0.        ]
 [0.26519082 0.33358304 0.        ]
 [0.5661581  0.53616119 0.        ]]
```



Sytuacja jest teraz w pewnym sensie odwrócona. Uzyskaliśmy skądś (liniowo separowalne) dane i chcemy znaleźć regułę, która definiuje te dwie klasy. Innymi słowy, musimy narysować linię podziału, która jest równoważna ze znalezieniem wag neuronu MCP Rys. 2.2, który przeprowadziłby odpowiednią klasyfikację binarną.

4.3 Algorytm perceptronu

Moglibyśmy spróbować jakoś obliczyć właściwe wagi dla powyższego przykładu i znaleźć linię podziału, na przykład linijką i ołówkiem, ale nie o to tutaj chodzi. Chcemy mieć systematyczną procedurę algorytmiczną, która bez trudu zadziała w tej czy każdej podobnej sytuacji. Odpowiedzią jest wspomniany już algorytm perceptronu.

Przed przedstawieniem algorytmu zauważmy, że neuron MCP z pewnym zbiorem wag w_0, w_1, w_2 zawsze daje jakąś odpowiedź dla etylirtowanego punktu danych, poprawną lub błędną. Na przykład

```
w=[-0.5,1,0]            # arbitrary choice of weights

print("label  answer") # header

for i in range(5): # look at first 5 points
    print(int(samp2[i,2]),"    ",func.neuron(samp2[i,:2],w))
            # samp2[i,2] is the label, samp2[i,:2] is [x_1,x_2]
```

```
label  answer
0      1
0      1
0      0
0      0
0      1
```

Widzimy, że niektóre odpowiedzi są równe etykietom (poprawne), a inne są od nich różne (błędne). Ogólną ideą jest teraz **użycie błędnych odpowiedzi**, aby sprytnie, małymi krokami zmieniać wagi, tak aby po wystarczającej liczbie iteracji wszystkie odpowiedzi dla danej próbki szkoleniowej były poprawne!

Algorytm perceptronu

Iterujemy po punktach próbki danych szkoleniowych. Jeżeli dla danego punktu otrzymany wynik y_o jest równy prawdziwej wartości y_t (etykieta), tj. odpowiedź jest prawidłowa, nic nie robimy. Jeśli jednak jest błędna, zmieniamy nieco wagi, tak aby szansa na otrzymanie błędnej odpowiedzi spadła. Przepis jest następujący:

$w_i \to w_i + \varepsilon(y_t - y_o)x_i,$

gdzie ε to mała liczba (nazywana **szybkością uczenia**), a x_i to współrzędne punktu wejściowego, gdzie $i = 0, \ldots, n$.

Prześledźmy, jak to działa. Załóżmy najpierw, że $x_i > 0$. Wtedy jeśli etykieta $y_t = 1$ jest większa niż uzyskana odpowiedź $y_o = 0$, waga w_i jest zwiększana. Wtedy $w \cdot x$ również wzrasta, a $y_o = f(w \cdot x)$ z większą szansą przyjmie poprawną wartość 1 (pamiętamy, jak wygląda funkcja schodkowa f). Jeżeli natomiast etykieta $y_t = 0$ jest mniejsza niż uzyskana odpowiedź $y_o = 1$, to waga w_i maleje, $w \cdot x$ maleje, a $y_o = f(w \cdot x)$ ma większą szansę na osiągnięcie prawidłowej wartości 0.

Jeśli $x_i < 0$, łatwo analogicznie sprawdzić, że przepis również działa poprawnie.

Jeśli odpowiedź jest prawidłowa, $y_t = y_0$, to $w_i \to w_i$, więc nic się nie zmienia. Nie „psujemy" perceptronu!

Powyższy wzór można zastosować wielokrotnie dla tego samego punktu z próbki szkoleniowej. Następnie wykonujemy pętlę po wszystkich punktach próbki, a całą procedurę można jeszcze powtarzać w wielu rundach, aby uzyskać stabilne wagi (nie zmieniające się już w miarę kontynuacji procedury lub zmieniające się tylko nieznacznie).

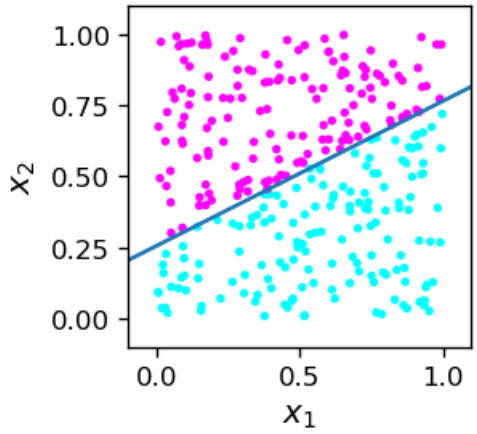
Zazwyczaj w takich algorytmach szybkość uczenia ε jest zmniejszana w kolejnych rundach. Jest to bardzo ważne z praktycznego punktu widzenia, ponieważ zbyt duże aktualizacje mogą zepsuć uzyskane rozwiązanie.

Implementacja algorytmu perceptronu dla danych dwuwymiarowych w Pythonie wygląda następująco:

```
w0=np.random.random()-0.5  # initialize weights randomly in the range [-0.5,0.5]
w1=np.random.random()-0.5
w2=np.random.random()-0.5

eps=.3                     # initial  learning speed

for _ in range(20):        # loop over rounds
    eps=0.9*eps            # in each round decrease the learning speed

    for i in range(npo):   # loop over the points from the data sample

        for _ in range(5): # repeat 5 times for each points

            yo = func.neuron(samp2[i,:2],[w0,w1,w2]) # obtained answer

            w0=w0+eps*(samp2[i,2]-yo)   # weight update (the perceptron formula)
            w1=w1+eps*(samp2[i,2]-yo)*samp2[i,0]
            w2=w2+eps*(samp2[i,2]-yo)*samp2[i,1]

print("Obtained weights:")
print("  w0     w1     w2")        # header
w_o=np.array([w0,w1,w2])           # obtained weights
print(np.round(w_o,3))             # result, rounded to 3 decimal places
```

```
Obtained weights:
  w0     w1     w2
[-0.562 -1.114  2.192]
```

Otrzymane wagi, jak wiemy, definiują linię podziału. Tak więc, geometrycznie, algorytm tworzy linię podziału, narysowaną poniżej wraz z próbką szkoleniową.



Widzimy, że algorytm działa! Wszystkie różowe punkty znajdują się powyżej linii podziału, a wszystkie niebieskie poniżej. Podkreślmy, że linia podziału dana przez równanie

$$w_0 + x_1 w_1 + x_2 w_2 = 0,$$

nie wynika z naszej wiedzy a priori, ale z treningu (uczenia nadzowowanego) neuronu MCP, który odpowiednio dopasowuje swoje wagi.

Uwaga: Można udowodnić, że algorytm perceptronu jest zbieżny wtedy i tylko wtedy, gdy dane są liniowo separowalne.

Teraz możemy wyjawić nasz sekret! Dane próbki szkoleniowej **samp2** zostały etykietowane w momencie tworzenia regułą

$$x_2 > 0,25 + 0,52 x_1,$$

co odpowiada wagom $w_0 = 0.25$, $w_1 = -0.52$, $w_2 = 1$.

```
w_c=np.array([-0.25,-0.52,1]) # weights used for labeling the training sample
print(w_c)
```

```
[-0.25 -0.52  1.  ]
```

Zwróćmy uwagę, że nie są to wcale te same wagi, jakie uzyskano podczas treningu:

```
print(np.round(w_o,3))
```

```
[-0.562 -1.114  2.192]
```

Powód jest dwojaki. Po pierwsze, zauważmy, że warunek nierówności (4.1) pozostaje niezmieniony, jeśli pomnożymy obie stronynierówności przez **dodatnią** stałą c. Możemy zatem przeskalować wszystkie wagi przez c, a sytuacja (odpowiedzi neuronu MCP, linia podziału) pozostaje dokładnie taka sama (napotykamy tutaj **klasę równoważności** wag przeskalowanych o czynnik dodatni) .

Z tego powodu dzieląc uzyskane wagi przez wagi użyte do oznaczenia próbki, otrzymujemy (prawie) stałe wartości:

```
print(np.round(w_o/w_c,3))
```

```
[2.249 2.143 2.192]
```

Powodem, dla którego wartości stosunków wag dla $i = 0, 1, 2$ nie są dokładnie takie same, jest to, że próbka ma skończoną liczbę punktów (tutaj 300). W ten sposób zawsze istnieje pewna luka między dwiema klasami punktów i jest trochę miejsca na nieznaczne przesuwanie linii rozdzielającej. Przy większej liczbie punktów danych efekt różnicy stosunków wag zmniejsza się (patrz ćwiczenia).

4.3.1 Testowanie klasyfikatora

Ze względu na ograniczoną wielkość próbki szkoleniowej i opisany powyżej efekt „luki", wynik klasyfikacji na próbce testowej jest czasami błędny. Dotyczy to zawsze punktów w pobliżu linii podziału, która jest wyznaczana z dokładnością zależną od krotności próbki szkoleniowej. Poniższy kod przeprowadza sprawdzenie na próbce testowej. Próbka ta składa się z etykietowanych danych wygenerowanych losowo za pomocą tej samej funkcji **point2** użytej uprzednio do wygenerowania danych szkoleniowych:

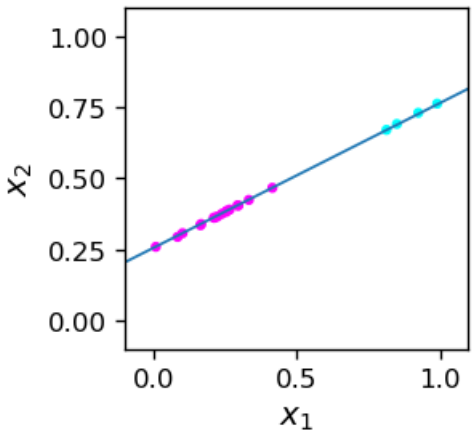
```
def point2():
    x1=np.random.random()           # random number from the range [0,1]
    x2=np.random.random()
    if(x2>x1*0.52+0.25):            # condition met
        return np.array([x1,x2,1]) # add label 1
    else:                           # not met
        return np.array([x1,x2,0]) # add label 0
```

Kod testujący jest następujący:

```
er= np.empty((0,3))  # initialize an empty 1 x 3 array to store misclassified␣
↪points

ner=0                  # initial number of misclassified points
nt=10000                # number of test points

for _ in range(nt):    # loop over the test points
    ps=point2()        # a test point
    if(func.neuron(ps[:2],[w0,w1,w2])!=ps[2]): # if wrong answer                ␣
↪
        er=np.append(er,[ps],axis=0)            # add the point to er
        ner+=1                                  # count the number of errors

print("number of misclassified points = ",ner," per ",nt," (", np.round(ner/
↪nt*100,1),"% )")
```

```
number of misclassified points =  20  per  10000  ( 0.2 % )
```

Jak widać, niewielka liczba punktów testowych jest błędnie sklasyfikowana. Wszystkie te punkty leżą w pobliżu linii rozdzielającej.



Błędna klasyfikacja

Przyczyną błędnej klasyfikacji jest fakt, że próbka szkoleniowa nie wyznacza dokładnie linii rozdzielającej, ponieważ między punktami występuje pewna luka. Aby uzyskać lepszy wynik, punkty treningowe musiałyby być „gęstsze" w sąsiedztwie linii rozdzielającej lub też próbka treningowa musiałaby być większa.

4.4 Ćwiczenia

- Pobaw się kodem z wykładu i zobacz, jak procent błędnie zaklasyfikowanych punktów zmniejsza się wraz ze wzrostem wielkości próbki szkoleniowej.
- Gdy algorytm perceptronu jest zbieżny, w pewnym momencie wagi przestają się zmieniać. Popraw kod wykładu, wdrażając zatrzymywanie, gdy wagi nie zmieniają się więcej niż pewna wartość podczas przechodzenia do następnej rundy.
- Uogólnij powyższy klasyfikator na punkty w przestrzeni trójwymiarowej.

ROZDZIAŁ 5

Więcej warstw

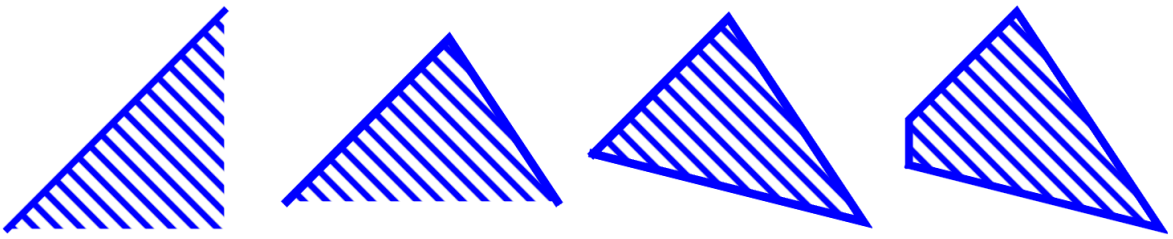
5.1 Dwie warstwy neuronów

W poprzednim rozdziale pokazaliśmy, że neuron MCP ze schodkową funkcją aktywacji odpowiada nierówności $x \cdot w = w_0 + x_1 w_1 + ... x_n w_n > 0$, gdzie n jest wymiarem przestrzeni wejściowej. Pouczające jest dalsze prześledzenie tej geometrycznej interpretacji. Przyjmując dla prostoty $n = 2$ (płaszczyzna), powyższa nierówność odpowiada jej podziałowi na dwie półpłaszczyzny. Jak wiemy, prosta wyrażona jest równaniem

$$w_0 + x_1 w_1 + x_2 w_2 = 0$$

i stanowi **linię podziału** na dwie półpłaszczyzny.

Wyobraźmy sobie teraz, że mamy więcej takich warunków: dwa, trzy itd., ogólnie k niezależnych warunków. Biorąc koniunkcję tych warunków, możemy zbudować wypukłe obszary, jak przykładowo pokazano na Rys. 5.1.



Rys. 5.1: Przykładowe obszary wypukłe na płaszczyźnie, od lewej do prawej: z jednym warunkiem nierówności, z koniunkcjami 2 warunków i z koniukcjami 3 lub 4 warunków nierówności, dającymi **wielokąty**.

Obszar wypukły

Z definicji obszar A jest wypukły wtedy i tylko wtedy, gdy odcinek pomiędzy dowolnymi dwoma punktami w A jest zawarty w A. Obszar, który nie jest wypukły nazywa się **wklęsłym**.

Oczywiście, k warunków nierówności można narzucić za pomocą k neuronów MCP. Przypomnijmy sobie z rozdz. *Funkcje logiczne*, że możemy w prosty sposób budować funkcje logiczne za pomocą sieci neuronowych. W szczególności możemy dokonać koniunkcji k warunków, biorąc neuron o wagach $w_0 = -1$ i $1/k < w_i < 1/(k-1)$, gdzie

$i = 1, \ldots, k$. Jedną z możliwości jest np.

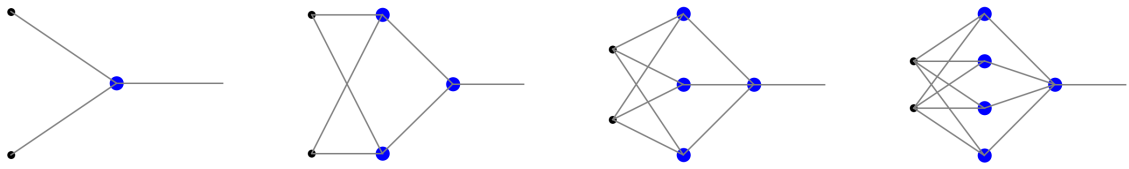
$$w_i = \frac{1}{k - \frac{1}{2}}.$$

Rzeczywiście, niech $p_0 = 1$, a warunki narzucone przez nierówności oznaczymy jako p_i, $i = 1, \ldots, k$, które mogą przyjmować wartości 1 lub 0 (prawda lub fałsz). Następnie

$$p \cdot w = -1 + p_1 w_1 + \cdots + p_k w_k = -1 + \frac{p_1 + \ldots p_k}{k - \frac{1}{2}} > 0$$

wtedy i tylko wtedy, gdy wszystkie $p_i = 1$, tj. wszystkie warunki są spełnione.

Architektury sieci dla warunków $k = 1$, 2, 3 lub 4 są ukazane na Rys. 5.2. Idąc od lewej do prawej począwszy od drugiego panelu, mamy sieci z dwiema warstwami neuronów i z k neuronami w warstwie pośredniej, zapewniającymi warunki nierówności, oraz jednym neuronem w warstwie wyjściowej, pełniącym funkcję bramki AND. Oczywiście dla jednego warunku wystarczy mieć jeden neuron, jak pokazano na lewym panelu Rys. 5.2.



Rys. 5.2: Sieci zdolne do klasyfikowania danych w obszarach z Rys. 5.1.

W interpretacji geometrycznej pierwsza warstwa neuronowa reprezentuje k półpłaszczyzn, a neuron w drugiej warstwie odpowiada obszarowi wypukłemu o k bokach.

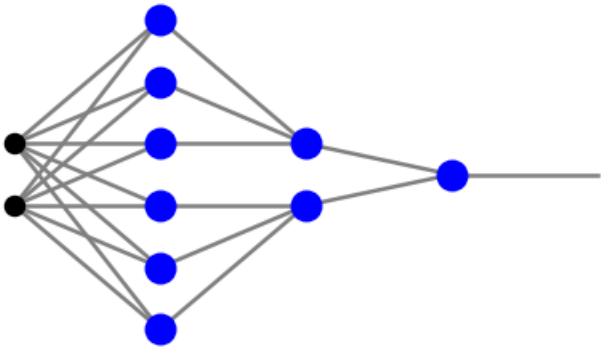
Sytuacja w oczywisty sposób uogólnia się na dane w większej liczbie wymiarów. W takim przypadku mamy więcej czarnych kropek dla danych wejściowych na Rys. 5.2. Geometrycznie dla $n = 3$ mamy do czynienia z dzieleniem na półprzestrzenie z pomocą płaszczyzn i tworzenie wypukłych wielościanów, a dla $n > 3$ z dzieleniem hiperprzestzreni [hiperpłaszczyznami](https:/ /en.wikipedia.org/wiki/Hyperplane) i tworzeniem wypukłych politopów.

Informacja: Jeśli w warstwie pośredniej znajduje się wiele neuronów, powstały wielokąt ma wiele boków, które mogą przybliżać gładką granicę, taką jak łuk. Aproksymacja jest coraz lepsza w miarę wzrostu k.

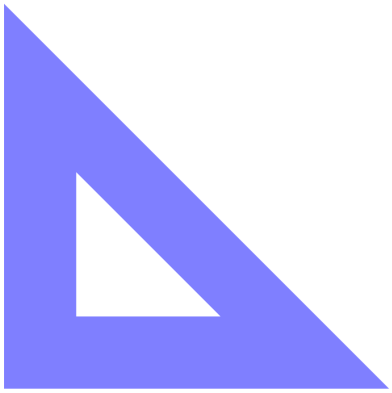
Ważne: Percepton z dwiema warstwami neuronów (z wystarczającą liczbą neuronów w warstwie pośredniej) może klasyfikować punkty należące do obszaru wypukłego w przestrzeni n-wymiarowej.

5.2 Trzy lub więcej warstw neuronów

Pokazaliśmy właśnie, że sieć dwuwarstwowa może klasyfikować wielokąt wypukły. Wyobraźmy sobie teraz, że tworzymy dwie takie figury w drugiej warstwie neuronów, na przykład dzieki następującej sieci:



Zauważmy, że pierwsza i druga warstwa neuronów nie są tutaj w pełni połączone, ponieważ „układamy na sobie" dwie sieci tworzące trójkąty, jak w trzecim panelu Rys. 5.2. Następnie w trzeciej warstwie neuronowej (tutaj posiadającej pojedynczy neuron) implementujemy bramkę $p \wedge \sim q$, czyli koniunkcję warunków, że punkty należą do jednego trójkąta, a nie należą do drugiego. Jak zaraz pokażemy, przy odpowiednich wagach powyższa siatka może wytworzyć obszar wklęsły, na przykład trójkąt z trójkątnym wgłębieniem:



Rys. 5.3: Trójkąt z trójkątnym wgłębieniem.

Uogólniając ten argument na inne kształty, można pokazać ważne twierdzenie:

Ważne: Perceptron z trzema lub więcej warstwami neuronów (z wystarczającą liczbą neuronów w warstwach pośrednich) może klasyfikować punkty należące do **dowolnego** regionu w n-wymiarowej przestrzeni z $n-1$-wymiarowymi ograniczeniami przez hiperpłaszczyzny.

Informacja: Warto tutaj podkreślić, że trzy warstwy zapewniają pełną funkcjonalność! Dodawanie kolejnych warstw do klasyfikatora nie zwiększa jego możliwości.

5.3 Feed forward w Pythonie

Zanim przejdziemy do przykładu, potrzebujemy kodu w Pythonie do propagacji sygnału w przód w ogólnej, w pełni połączonej sieci. Będziemy reprezentować architekturę sieci z l warstwami neuronów jako tablicę postaci

$$[n_0, n_1, n_2, ..., n_l],$$

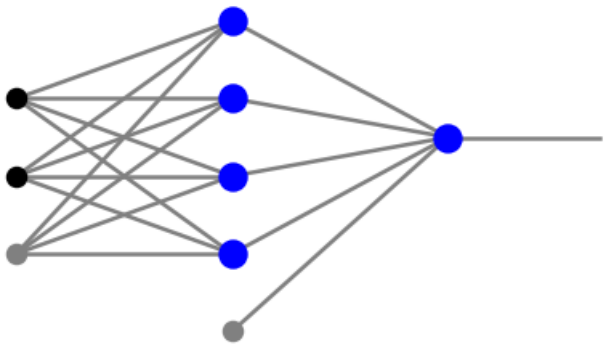
gdzie n_0 jest liczbą węzłów wejściowych, a n_i liczbą neuronów w warstwach $i = 1, \ldots, l$. Na przykład architektura sieci z czwartego panelu Rys. 5.2 to

```
arch=[2,4,1]
arch
```

```
[2, 4, 1]
```

W kodach tego kursu posługujemy się konwencją z Rys. 2.2, a mianowicie próg jest traktowany jednolicie z pozostałym sygnałem. Jednak węzły progowe nie są uwzględniane w określaniu liczb n_i zdefiniowanych powyżej. W szczególności bardziej szczegółowy widok czwartego panelu Rys. 5.2 to

```
plt.show(draw.plot_net(arch))
```



Czarne kropki oznaczają dane wejściowe, szare kropki odpowiadają węzłom progowym, dającym input równy 1, a niebieskie kółka to neurony.

Następnie potrzebujemy wagi połączeń. Mamy l zestawów wag, z których każdy odpowiada krawędziom wchodzącym do danej warstwy neuronowej od lewej strony. W powyższym przykładzie pierwsza warstwa neuronów (niebieskie kółka po lewej stronie) ma wagi, które tworzą macierz 3×4. Tutaj 3 to liczba węzłów w poprzedniej (wejściowej) warstwie (łącznie z węzłem progowym), a 4 to liczba neuronów w pierwszej warstwie neuronowej. Podobnie wagi związane z drugą (wyjściową) warstwą neuronową tworzą macierz 4×1. Stąd w naszej konwencji macierze wag odpowiadające kolejnym warstwom neuronów $1, 2, \ldots, l$ mają wymiary

$$(n_0 + 1) \times n_1, \ (n_1 + 1) \times n_2, \ \ldots \ (n_{l-1} + 1) \times n_l.$$

Tak więc, aby przechowywać wszystkie wagi sieci, tak naprawdę potrzebujemy **trzech** wskaźników: jeden dla warstwy, jeden dla liczby węzłów w poprzedniej warstwie i jeden dla liczby węzłów w danej warstwie. Moglibyśmy tutaj użyć trójwymiarowej tablicy, ale ponieważ numerujemy warstwy neuronów zaczynając od 1, a tablice zaczynają się od 0, nieco wygodniej jest użyć struktury **słownika** Pythona. Przechowujemy zatem wagi jako

$$w = \{1 : arr^1, 2 : arr^2, ..., l : arr^l\},$$

gdzie arr^i jest **dwuwymiarową** tablicą (macierzą) wag dla warstwy neuronowej i. Dla przypadku z powyższego rysunku możemy wziąć na przykład

```
w={1:np.array([[1,2,1,1],[2,-3,0.2,2],[-3,-3,5,7]]),2:np.array([[1],[0.2],[2],[2],
 ↪[-0.5]])}

print(w[1])
print("")
print(w[2])
```

```
[[ 1.   2.   1.   1. ]
 [ 2.  -3.   0.2  2. ]
 [-3.  -3.   5.   7. ]]

[[ 1. ]
```

(ciąg dalszy na następnej stronie)

(kontynuacja poprzedniej strony)

```
[ 0.2]
[ 2. ]
[ 2. ]
[-0.5]]
```

Dla sygnału rozchodzącego się w sieci stosujemy również odpowiednio słownik w postaci

$$x = \{0 : x^0, 1 : x^1, 2 : x^2, ..., l : x^l\},$$

gdzie x^0 to wejście, a x^i to sygnał wychodzący z warstwy neuronowej i, dla $i = 1, \dots, l$. Wszystkie symbole x^j, $j = 0, \dots, l$ są tablicami jednowymiarowymi. Uwzględniamy tu węzły progowe, stąd wymiary x^j wynoszą $n_j + 1$, z wyjątkiem warstwy wyjściowej, która nie ma węzła odchylenia, stąd x^l ma wymiar n_l. Innymi słowy, wymiary tablic sygnału są równe całkowitej liczbie węzłów w każdej warstwie.

Następnie przedstawiamy szczegółowo odpowiednie wzory, ponieważ jest to kluczowe dla uniknięcia ewentualnych pomyłek związanych z zapisem. Wiemy już z (2.2), że dla pojedynczego neuronu z n wejściami, sygnał wchodzący jest obliczany jako

$$s = x_0 w_0 + x_1 w_1 + x_2 w_2 + ... + x_n w_n = \sum_{\beta=0}^{n} x_\beta w_\beta.$$

Przy większej liczbie warstw (oznaczonych wskaźnikiem górnym i) i liczbie neuronów n_i w warstwie i, notacja uogólnia się na

$$s^i_\alpha = \sum_{\beta=0}^{n_{i-1}} x^{i-1}_\beta w^i_{\beta\alpha}, \quad \alpha = 1 \dots n_i, \quad i = 1, \dots, l.$$

Zauważmy, że sumowanie zaczyna się od $\beta = 0$, aby uwzględnić węzeł progowy w poprzedniej warstwie $(i-1)$, ale α zaczyna się od 1, ponieważ tylko neurony (a nie węzeł progowy) w warstwie i odbierają sygnał (patrz rysunek poniżej).

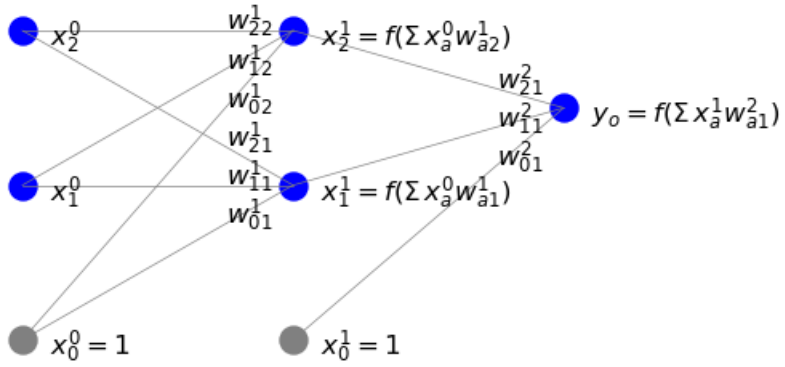
W notacji macierzowej możemy też zapisać bardziej zwięźle $s^{iT} = x^{(i-1)T} W^i$, gdzie T oznacza transpozycję, tzn.

$$\begin{pmatrix} s^i_1 & s^i_2 & \dots & s^i_{n_i} \end{pmatrix} = \begin{pmatrix} x^{i-1}_0 & x^{i-1}_1 & \dots & x^{i-1}_{n_{i-1}} \end{pmatrix} \begin{pmatrix} w^i_{01} & w^i_{02} & \dots & w^i_{0,n_i} \\ w^i_{11} & w^i_{12} & \dots & w^i_{1,n_i} \\ \dots & \dots & \dots & \dots \\ w^i_{n_{i-1}1} & w^i_{n_{i-1}2} & \dots & w^i_{n_{i-1}n_i} \end{pmatrix}.$$

Jak już dobrze wiemy, wyjście z neuronu uzyskuje się działając na jego sygnał wejściowy funkcją aktywacji. W ten sposób w końcu mamy

$$x^i_\alpha = f(s^i_\alpha) = f\left(\sum_{\beta=0}^{n_{i-1}} x^{i-1}_\beta w^i_{\beta\alpha}\right), \quad \alpha = 1 \dots n_i, \quad i = 1, \dots, l,$$
$$x^i_0 = 1, \quad i = 1, \dots, l-1,$$

z węzłami progowymi równymi jeden. Poniższy rysunek ilustruje naszą notację.



Implementacja propagacji feed-forward w Pythonie jest następująca:

```
def feed_forward(ar, we, x_in, f=func.step):
    """
    Feed-forward propagation

    input:
    ar - array of numbers of nodes in subsequent layers [n_0, n_1,...,n_l]
    (from input layer 0 to output layer l, bias nodes not counted)

    we - dictionary of weights for neuron layers 1, 2,...,l in the format
    {1: array[n_0+1,n_1],...,l: array[n_(l-1)+1,n_l]}

    x_in - input vector of length n_0 (bias not included)

    f - activation function (default: step)

    return:
    x - dictionary of signals leaving subsequent layers in the format
    {0: array[n_0+1],...,l-1: array[n_(l-1)+1], l: array[nl]}
    (the output layer carries no bias)

    """

    l=len(ar)-1                    # number of the neuron layers
    x_in=np.insert(x_in,0,1)       # input, with the bias node inserted

    x={}                           # empty dictionary x
    x.update({0: np.array(x_in)}) # add input signal to x

    for i in range(1,l):           # loop over layers except the last one
        s=np.dot(x[i-1],we[i])     # signal, matrix multiplication
        y=[f(s[k]) for k in range(arch[i])] # output from activation
        x.update({i: np.insert(y,0,1)}) # add bias node and update x

                                   # the output layer l - no adding of the bias␣
↪node
        s=np.dot(x[l-1],we[l])     # signal
        y=[f(s[q]) for q in range(arch[l])] # output
        x.update({l: y})           # update x

    return x
```

Dla zwięzłości przyjmujemy konwencję, w której nie przekazujemy w argumentach funkcji węzła progowego. Jest on wstawiany do funkcji za pomocą **np.insert(x_in,0,1)**. Jak zwykle używamy **np.dot** do mnożenia macierzy.

Następnie testujemy, jak **feed_forward** działa dla przykładowych danych wejściowych.

```
xi=[2,-1]
x=func.feed_forward(arch,w,xi,func.step)
print(x)
```

```
{0: array([ 1,  2, -1]), 1: array([1, 1, 0, 0, 0]), 2: [1]}
```

Końcowy output z tej sieci jest uzyskany jako

```
x[2][0]
```

```
1
```

5.3.1 Dygresja o sieciach liniowych

Zróbmy teraz następującą obserwację. Załóżmy, że mamy sieć z liniową funkcją aktywacji $f(s) = cs$. Wtedy ostatnia formuła z powyższego wyprowadzenia przyjmuje postać

$$x^i_\alpha = c \sum_{\beta=0}^{n_{i-1}} x^{i-1}_\beta w^i_{\beta\alpha}, \quad \alpha = 1 \dots n_i, \quad i = 1, \dots, l,$$

lub w notacji macierzowej:

$$x^i = c x^{i-1} w^i.$$

Powtarzając to, otrzymujemy sygnał w warstwie wyjściowej

$$x^l = c x^{l-1} w^i = c^2 x^{l-2} w^{l-1} w^l = \dots = c^l x^0 w^1 w^2 \dots w^l = x^0 W,$$

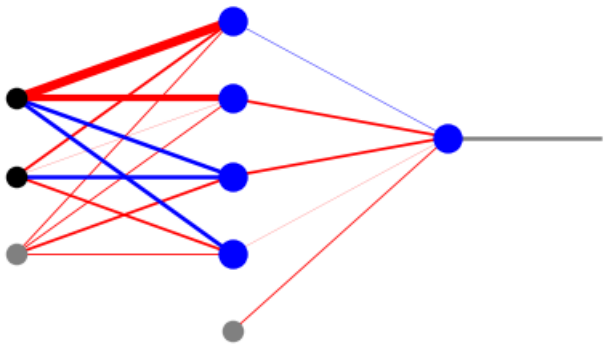
gdzie $W = c^l w^1 w^2 \dots w^l$. Widzimy wiec, że taka sieć jest **równoważna** sieci jednowarstwowej z macierzą wag W określoną powyżej.

Informacja: Z tego powodu nie ma sensu rozważanie sieci wielowarstwowych z liniową funkcją aktywacji.

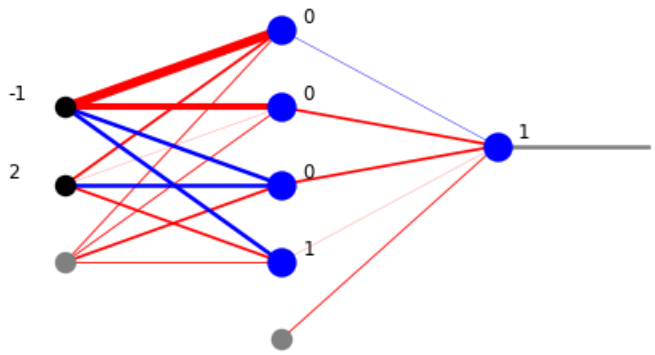
5.4 Wizualizacja

W celu wizualizacji prostych sieci w module **draw** pakietu **neural** udostępniamy kilka funkcji rysowania, które ukazują zarówno wagi, jak i sygnały w sieci. Funkcja **plot_net_w** rysuje wagi dodatnie na czerwono, a ujemne na niebiesko, przy czym szerokości odzwierciedlają ich wielkość. Ostatni parametr, tutaj 0.5, przeskalowuje szerokości tak, że grafika wygląda ładnie. Funkcja **plot_net_w_x** drukuje dodatkowo wartości sygnału wychodzącego z węzłów każdej warstwy.

```
plt.show(draw.plot_net_w(arch,w,0.5))
```



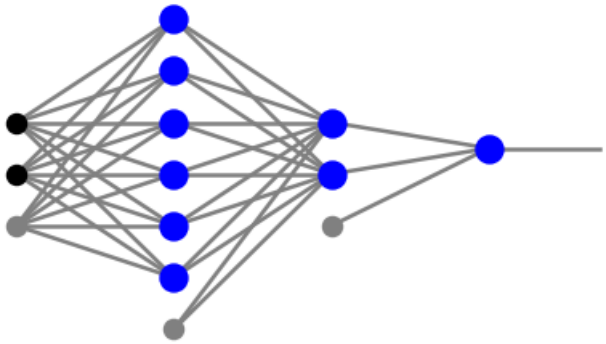
```
plt.show(draw.plot_net_w_x(arch,w,0.5,x))
```



5.5 Klasyfikator z trzema warstwami neuronów

Jesteśmy teraz gotowi do jawnego skonstruowania przykładu binarnego klasyfikatora punktów w obszarze wklęsłym: trójkąta z trójkątnym wycięciem z Rys. 5.3. Architektura sieci to

```
arch=[2,6,2,1]
plt.show(draw.plot_net(arch))
```



Warunki geometryczne i odpowiadające im wagi dla pierwszej warstwy neuronowej to

α	warunek	$w^1_{0\alpha}$	$w^1_{1\alpha}$	$w^1_{2\alpha}$
1	$x_1 > 0.1$	-0.1	1	0
2	$x_2 > 0.1$	-0.1	0	1
3	$x_1 + x_2 < 1$	1	-1	-1
4	$x_1 > 0.25$	-0.25	1	0
5	$x_2 > 0.25$	-0.25	0	1
6	$x_1 + x_2 < 0.8$	0.8	-1	-1

Warunki 1-3 zapewniają granice dla większego trójkąta, a 4-6 dla mniejszego, zawartego w większym. W drugiej warstwie neuronowej musimy zrealizować dwie bramki AND odpowiednio dla warunków 1-3 i 4-6, a zatem bierzemy

α	$w^2_{0\alpha}$	$w^2_{1\alpha}$	$w^2_{2\alpha}$	$w^2_{3\alpha}$	$w^2_{4\alpha}$	$w^2_{5\alpha}$	$w^2_{6\alpha}$
1	-1	0.4	0.4	0.4	0	0	0
2	-1	0	0	0	0.4	0.4	0.4

Na koniec w warstwie wyjściowej realizujemy bramkę $p \wedge \sim q$, skąd

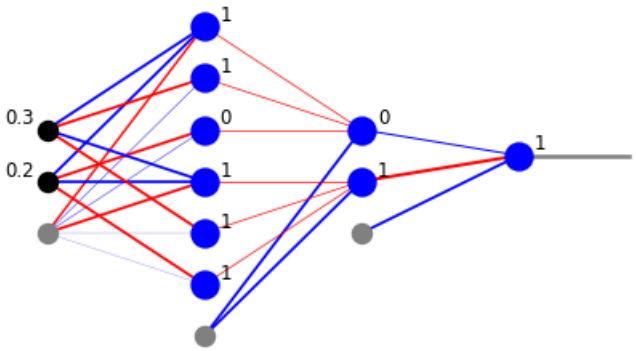
α	$w^3_{0\alpha}$	$w^3_{1\alpha}$	$w^3_{2\alpha}$
1	-1	1.2	-0.6

Łącząc to wszystko razem, uzyskujemy nastepujacy słownik wag:

```
w={1:np.array([[-0.1,-0.1,1,-0.25,-0.25,0.8],[1,0,-1,1,0,-1],[0,1,-1,0,1,-1]]),
   2:np.array([[-1,-1],[0.4,0],[0.4,0],[0.4,0],[0,0.4],[0,0.4],[0,0.4]]),
   3:np.array([[-1],[1.2],[-0.6]])}
```

Feed-forward dla prykładowego inputu daje

```
xi=[0.2,0.3]
x=func.feed_forward(arch,w,xi)
plt.show(draw.plot_net_w_x(arch,w,1,x))
```



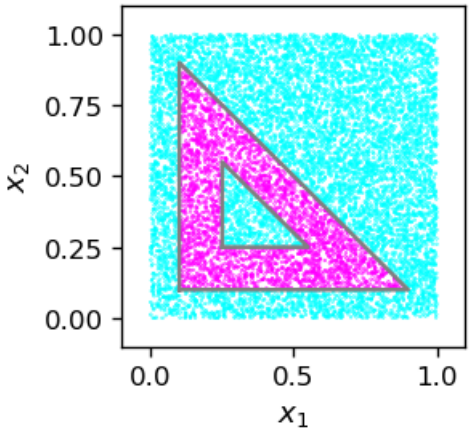
Właśnie odkryliśmy, że punkt [0.2,0.3] znajduje się w naszym obszarze (1 z warstwy wyjściowej). Właściwie mamy tutaj więcej informacji z warstw pośrednich. Z drugiej warstwy neuronowej widzimy, że punkt należy do większego trójkąta (1 z dolnego neuronu), a nie należy do mniejszego trójkąta (0 z górnego neuronu). Z pierwszej warstwy neuronowej możemy odczytać warunki z sześciu nierówności.

Następnie testujemy działanie naszej sieci dla innych punktów. Najpierw definiujemy funkcję generującą losowy punkt w kwadracie $[0,1] \times [0,1]$ i propagujemy go przez sieć. Przypisujemy mu etykietę 1, jeśli należy do żądanego obszaru, a 0 w przeciwnym razie. Następnie tworzymy dużą próbkę takich punktów i generujemy grafikę, używając koloru różowego dla etykiety 1 i niebieskiego dla etykiety 0.

```
def po():
    xi=[np.random.random(),np.random.random()] # random point from the [0,1]x[0,
 ↪1] square
    x=func.feed_forward(arch,w,xi)             # feed forward
    return [xi[0],xi[1],x[3][0]]               # the point's coordinates and label
```

```
samp=np.array([po() for _ in range(10000)])
print(samp[:5])
```

```
[[0.7088127  0.81098538 0.        ]
 [0.52044622 0.19265106 1.        ]
 [0.45637626 0.95670484 0.        ]
 [0.51723699 0.06066656 0.        ]
 [0.00879429 0.38239462 0.        ]]
```



Widzimy, że nasza maszynka działa doskonale!

W tym miejscu czytelnik może słusznie powiedzieć, że powyższe wyniki są trywialne: w istocie właśnie zaimplementowaliśmy pewne warunki geometryczne i ich koniunkcje.

Jednak, podobnie jak w przypadku sieci jednowarstwowych, istnieje ważny argument przeciwko tej pozornej błahości. Wyobraźmy sobie ponownie, że mamy próbkę danych z etykietami i tylko tele, podobnie jak w przykładzie pojedynczego neuronu MCP z rozdziału *Neuron MCP*. Wtedy na początku nie mamy warunków granicznych i potrzebujemy jakiegoś skutecznego sposobu, aby je znaleźć. Właśnie to zadanie wykonuje za nas **uczenie** klasyfikatorów: ustala wagi w taki sposób, że odpowiednie warunki są domyślnie wbudowane. Po materiale z tego rozdziału czytelnik powinien być przekonany, że jest to jak najbardziej możliwe i nie ma w tym nic magicznego! W następnym rozdziale pokażemy, jak to praktycznie zrobić.

5.6 Ćwiczenia

- Zaprojektuj sieć i uruchom kod z tego wykładu dla wybranego regionu wypukłego.
- Zaprojektuj i zaprogramuj klasyfikator dla czterech kategorii punktów należących do regionów utworzonych przez dwie przecinające się linie (wskazówka: uwzględnij wiecej komórek wyjściowych).

ROZDZIAŁ 6

Propagacja wsteczna

W tym rozdziale pokażemy szczegółowo, jak przeprowadzić uczenie nadzorowane dla klasyfikatorów wielowarstwowych omówionych w rozdziale *Więcej warstw*. Ponieważ metoda opiera się na minimalizacji liczby błędnych odpowiedzi na próbce testowej, zaczynamy od dokładnego omówienia problemu minimalizacji błędów w naszej konfiguracji.

6.1 Minimalizacja błędu

Przypomnijmy, że w naszym przykładzie z punktami na płaszczyźnie z rozdziału *Perceptron* warunek dla różowych punktów był zadany przez nierówność

$w_0 + w_1 x_1 + w_2 x_2 > 0.$

Wspomnieliśmy już pokrótce o klasie równoważności związanej z dzieleniem obu stron tej nierówności przez dodatnią stałą c. Ogólnie rzecz biorąc, co najmniej jedna z wag w powyższym warunku musi być niezerowa, aby był on nietrywialny. Załóżmy zatem, że $w_0 \neq 0$ (inne przypadki można potraktować analogicznie). Następnie podzielmy obie strony nierówności przez $|w_0|$, co daje

$$\frac{w_0}{|w_0|} + \frac{w_1}{|w_0|}\, x_1 + \frac{w_2}{|w_0|}\, x_2 > 0.$$

Wprowadzając notację $v_1 = \frac{w_1}{w_0}$ and $v_2 = \frac{w_2}{w_0}$, możemy zatem zapisać

$$\text{sgn}(w_0)(1 + v_1\, x_1 + v_2\, x_2) > 0,$$

gdzie znak $\text{sgn}(w_0) = \frac{w_0}{|w_0|}$. Mamy więc w efekcie system dwuparametrowy (dla ustalonego znaku w_0).

Oczywiście przy pewnych wartościach v_1 i v_2 i dla danego punktu z próbki danych, perceptron poda w wyniku poprawną lub błędną odpowiedź. Naturalne jest zatem zdefiniowanie **funkcji błędu** E w taki sposób, że dla każdego punktu p próbki wnosi 1, jeśli odpowiedź jest niepoprawna, a 0, jeśli jest poprawna:

$$E(v_1, v_2) = \sum_p \begin{cases} 1 - \text{niepoprawna}, \\ 0 - \text{poprawna}. \end{cases}$$

E ma zatem interpretację liczby źle sklasyfikowanych punktów.

Możemy łatwo skonstruować tę funkcję w Pythonie:

```
def error(w0, w1 ,w2, sample, f=func.step):
    """
    error function for the perceptron (for 2-dim data with labels)

    inputs:
    w0, w1, w2 - weights
    sample - array of labeled data points p
             p in an array in the format [x1, x1, label]
    f - activation function

    returns:
    error
    """
    er=0                                        # initial value of the error
    for i in range(len(sample)):                # loop over data points
        yo=f(w0+w1*sample[i,0]+w2*sample[i,1]) # obtained answer
        er+=(yo-sample[i,2])**2
                    # sample[i,2] is the label
                    # adds the square of the difference of yo and the label
                    # this adds 1 if the answer is incorrect, and 0 if correct
    return er  # the error
```

Zastosowaliśmy tutaj małą sztuczkę, mając na uwadze przyszłe zastosowania. Oznaczając otrzymany wynik dla danego punktu danych jako $y_o^{(p)}$, a wynik prawdziwy (etykietę) jako $y_t^{(p)}$ (obydwa przyjmują wartości 0 lub 1), możemy zdefiniowane powyżej E zapisać równoważnie jako

$$E(v_1, v_2) = \sum_p \left(y_o^{(p)} - y_t^{(p)}\right)^2,$$

co jest wzorem zaprogramowanym w kodzie. Rzeczywiście, kiedy $y_o^{(p)} = y_t^{(p)}$ (prawidłowa odpowiedź), wkład punktu wynosi 0, a kiedy $y_o^{(p)} \neq y_t^{(p)}$ (błędna odpowiedź), wkład wynosi $(\pm 1)^2 = 1$.

Powtarzamy teraz symulacje z podrozdziału *Perceptron*, aby wygenerować etykietowaną próbkę danych **samp2** o 200 punktach (próbka jest utworzona z $w_0 = -0.25$, $w_1 = -0.52$ i $w_2 = 1$, co odpowiada $v_1 = 2.08$ i $v_2 = -4$, przy czym $\text{sgn}(w_0) = -1$).

Potrzebujemy teraz ponownie użyć algorytmu perceptronu z rozdz. *Algorytm perceptronu*. W naszym szczególnym przypadku działa on na próbce dwuwymiarowych danych etykietowanych. Dla wygody, pojedyncza runda algorytmu może zostać zebrana w funkcję w następujący sposób:

```
def teach_perceptron(sample, eps, w_in, f=func.step):
    """
    Supervised learning for a single perceptron (single MCP neuron)
    for a sample of 2-dim. labeled data

    input:
    sample - array of two-dimensional labeled data points p
             p is an array in the format [x1,x2,label]
             label = 0 or 1
    eps    - learning speed
    w_in   - initial weights in the format [[w0], [w1], [w2]]
    f      - activation function

    return: updated weights in the format [[w0], [w1], [w2]]
    """
    [[w0],[w1],[w2]]=w_in          # define w0, w1, and w2
    for i in range(len(sample)):   # loop over the whole sample
        for k in range(10):        # repeat 10 times

            yo=f(w0+w1*sample[i,0]+w2*sample[i,1]) # output from the neuron, f(x.
↪w)
```

(ciąg dalszy na następnej stronie)

(kontynuacja poprzedniej strony)

```
            # update of weights according to the perceptron algorithm formula
            w0=w0+eps*(sample[i,2]-yo)*1
            w1=w1+eps*(sample[i,2]-yo)*sample[i,0]
            w2=w2+eps*(sample[i,2]-yo)*sample[i,1]

    return [[w0],[w1],[w2]]       # updated weights
```

Następnie prześledzimy działanie algorytmu perceptronu, obserwując jak modyfikuje on wartości wprowadzonej powyżej funkcji błędu $E(v_1, v_2)$. Zaczynamy od losowych wag, a następnie wykonujemy 10 rund zdefiniowanej powyżej funkcji **teach_perceptron**, wypisując zaktualizowane wagi i odpowiadający im błąd:

```
weights=[[func.rn()], [func.rn()], [func.rn()]] # initial random weights
```

```
weights
```

```
[[0.30195756389159656], [0.3291266701139506], [0.049322233629366 72]]
```

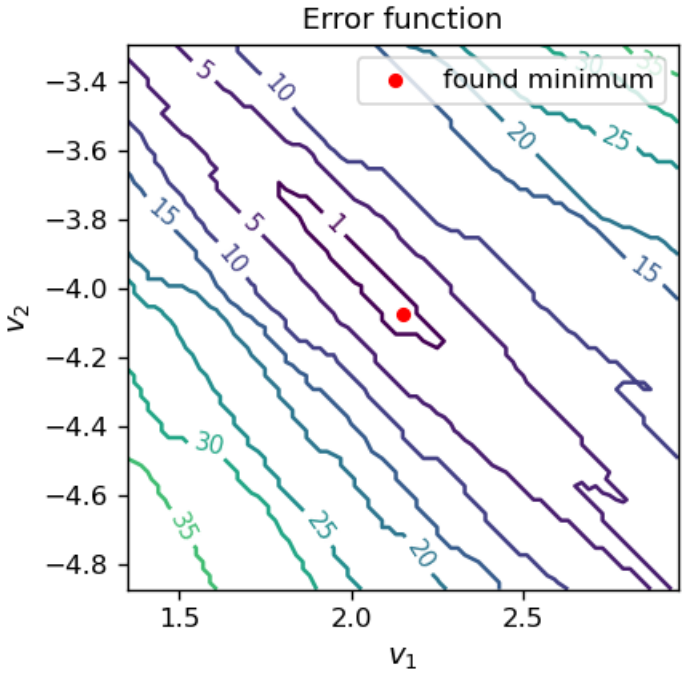
```
print("Optimum:")
print("   w0  w1/w0  w2/w0 error")   # header

eps=0.7                 # initial learning speed
for r in range(15):     # rounds
    eps=0.9*eps         # decrease the learning speed
    weights=teach_perceptron(samp2,eps,weights,func.step)
                        # see the top of this chapter

    w0_o=weights[0][0]  # updated weights and ratios
    v1_o=weights[1][0]/weights[0][0]
    v2_o=weights[2][0]/weights[0][0]

    print(np.round(w0_o,3),np.round(v1_o,3),np.round(v2_o,3),
          np.round(error(w0_o, w0_o*v1_o, w0_o*v2_o, samp2, func.step),0))
```

```
Optimum:
   w0  w1/w0  w2/w0 error
-0.958 0.786 -2.956 22.0
-0.958 1.618 -3.619 3.0
-0.958 1.765 -3.823 3.0
-0.958 1.876 -4.006 6.0
-0.958 1.918 -4.159 10.0
-0.958 2.15 -4.073 0.0
-0.958 2.15 -4.073 0.0
-0.958 2.15 -4.073 0.0
-0.958 2.15 -4.073 0.0
-0.958 2.15 -4.073 0.0
-0.958 2.15 -4.073 0.0
-0.958 2.15 -4.073 0.0
-0.958 2.15 -4.073 0.0
-0.958 2.15 -4.073 0.0
-0.958 2.15 -4.073 0.0
```

Zauważamy, że w kolejnych rundach błąd stopniowo maleje (w zależności od symulacji, może czasem nieco podskoczyć, jeśli szybkość uczenia się jest zbyt duża, ale nie stanowi to problemu, o ile koniec końców możemy zejść do minimum), osiągając ostatecznie wartość bardzo małą lub dokładnie 0 (w zależności od konkretnego przypadku symulacji). W związku z tym algorytm perceptronu, jak już widzieliśmy w rozdziale *Perceptron*, **minimalizuje błąd dla próbki treningowej**.

Pouczające jest spojrzenie na mapę konturową funkcji błędu $E(v_1, v_2)$ w pobliżu optymalnych parametrów:

```
fig, ax = plt.subplots(figsize=(3.7,3.7),dpi=120)

delta = 0.02  # grid step in v1 and v2 for the contour map
ran=0.8       # plot range around (v1_o, v2_o)

v1 = np.arange(v1_o-ran,v1_o+ran, delta) # grid for v1
v2 = np.arange(v2_o-ran,v2_o+ran, delta) # grid for v2
X, Y = np.meshgrid(v1, v2)               # mesh for the contour plot

Z=np.array([[error(-1,-v1[i],-v2[j],samp2,func.step)
             # we use the scaling property of the error function here
             for i in range(len(v1))] for j in range(len(v2))]) # values of E(v1,
 ↪v2)

CS = ax.contour(X, Y, Z, [1,5,10,15,20,25,30,35,40,45,50])
                         # explicit contour level values

ax.clabel(CS, inline=1, fmt='%1.0f', fontsize=9) # contour label format

ax.set_title('Error function', fontsize=11)
ax.set_aspect(aspect=1)

ax.set_xlabel('$v_1$', fontsize=11)
ax.set_ylabel('$v_2$', fontsize=11)

ax.scatter(v1_o, v2_o, s=20,c='red',label='found minimum') # our found optimal␣
 ↪point

ax.legend()
plt.show()
```



Uzyskane minimum znajduje się wewnątrz (lub blisko, w zależności od symulacji) wydłużonego obszaru w v_1 i v_2, gdzie błąd znika.

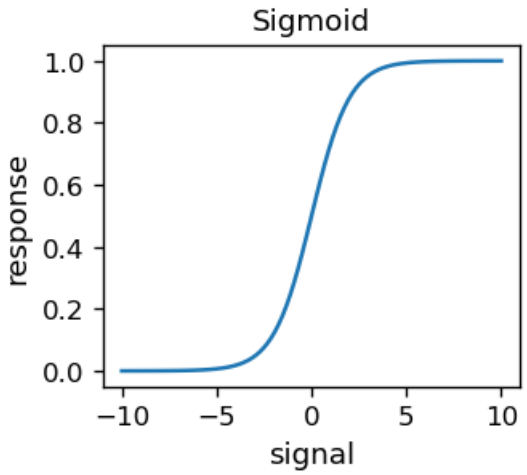
6.2 Ciągła funkcja aktywacji

Przyglądając się uważniej powyższej mapie konturowej, widzimy, że linie są „ząbkowane". Dzieje się tak, ponieważ funkcja błędu, z oczywistego powodu, przyjmuje wartości całkowite. Jest zatem nieciągła, a zatem nieróżniczkowalna. Nieciągłości wynikają z nieciągłej funkcji aktywacji, mianowicie funkcji schodkowej. Mając na uwadze techniki, które poznamy niebawem, korzystne jest stosowanie funkcji aktywacji, która jest różniczkowalna. Historycznie tzw. **sigmoid**

$$\sigma(s) = \frac{1}{1+e^{-s}}$$

był wykorzystywany w wielu praktycznych zastosowaniach dla ANN.

```
# sigmoid, a.k.a. the logistic function, or simply (1+arctanh(-s/2))/2
def sig(s):
    return 1/(1+np.exp(-s))
```

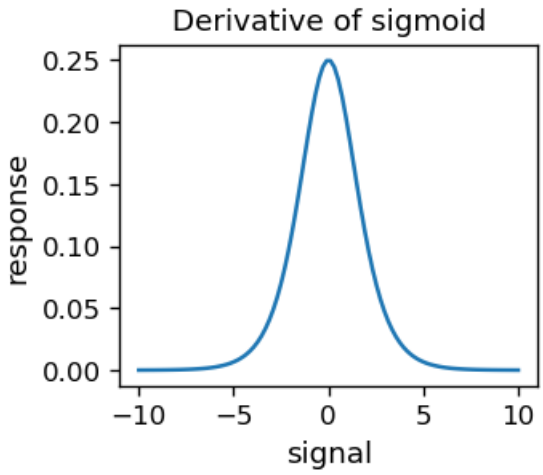


Funkcja ta jest oczywiście różniczkowalna. Ponadto

$$\sigma'(s) = \sigma(s)[1-\sigma(s)],$$

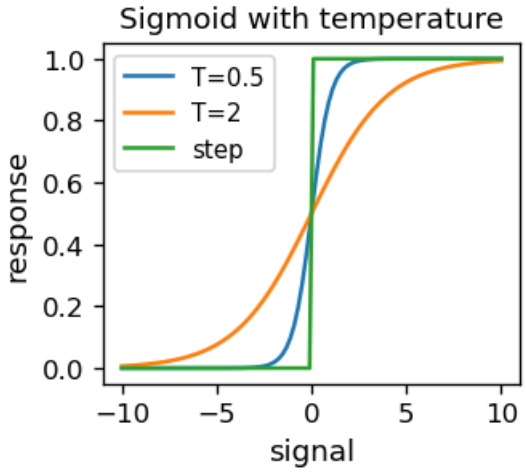
co jest szczególna własnością sigmoidu.

```
# derivative of sigmoid
def dsig(s):
     return sig(s)*(1-sig(s))
```



Wprowadza się również sigmoid z „temperaturą” T (nomenklatura ta jest związana z podobnymi wyrażeniami dla funkcji termodynamicznych w fizyce): $\$\sigma(s;T) = \frac{1}{1+e^{-s/T}}.\$$

```
# sigmoid with temperature T
def sig_T(s,T):
    return 1/(1+np.exp(-s/T))
```



Dla coraz mniejszych T sigmoid zbliża się do poprzednio używanej funkcji schodkowej.

Zauważ, że argumentem sigmoidu jest iloraz

$$s/T = (w_0 + w_1 x_1 + w_2 x_2)/T = w_0/T + w_1/T\, x_1 + w_2/T\, x_2 = \xi_0 + \xi_1 x_1 + \xi_2 x_2,$$

co oznacza, że zawsze możemy przyjąć $T = 1$ bez utraty ogólności (T to „skala”). Jednak teraz mamy trzy niezależne argumenty ξ_0, ξ_1 i ξ_2, więc nie można zredukować obecnej sytuacji do tylko dwóch niezależnych parametrów, jak miało to miejsce w poprzednim podrozdziale.

Powtórzymy teraz nasz przykład z klasyfikatorem, ale z funkcją aktywacji daną przez sigmoid. Funkcja błędu

$$y_o^{(p)} = \sigma(w_0 + w_1 x_1^{(p)} + w_2 x_2^{(p)}),$$

staje się teraz

$$E(w_0, w_1, w_2) = \sum_p \left[\sigma(w_0 + w_1 x_1^{(p)} + w_2 x_2^{(p)}) - y_t^{(p)}\right]^2.$$

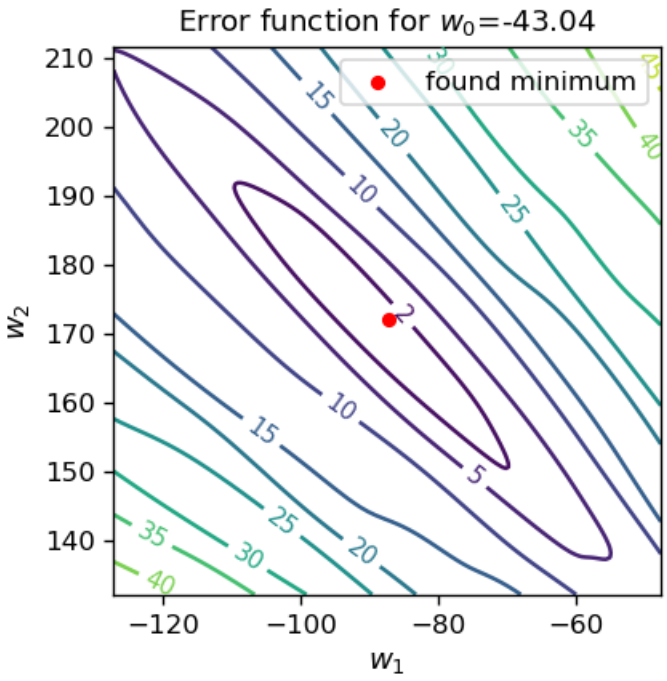
Algorytm perceptronu z funkcją aktywacji sigmoidu wykonujemy 1000 razy, wypisując co 100 krok:

```
weights=[[func.rn()],[func.rn()],[func.rn()]]      # random weights from [-0.5,0.
↪5]

print("   w0    w1/w0   w2/w0 error")    # header

eps=0.7                          # initial learning speed
for r in range(1000):            # rounds
    eps=0.9995*eps               # decrease learning speed
    weights=teach_perceptron(samp2,eps,weights,func.sig) # update weights
    if r%100==99:
        w0_o=weights[0][0]                # updated weights
        w1_o=weights[1][0]
        w2_o=weights[2][0]
        v1_o=w1_o/w0_o                    # ratios of weights
        v2_o=w2_o/w0_o
        print(np.round(w0_o,3),np.round(v1_o,3),np.round(v2_o,3),
              np.round(error(w0_o, w0_o*v1_o, w0_o*v2_o, samp2, func.sig),5))
```

```
   w0   w1/w0  w2/w0 error
-21.633 1.908 -3.966 2.35679
-27.248 1.962 -3.994 1.77834
-30.854 1.99 -4.008 1.45645
-33.559 2.007 -4.014 1.23376
-35.751 2.017 -4.017 1.06907
-37.604 2.022 -4.016 0.94205
-39.211 2.024 -4.013 0.84068
-40.629 2.025 -4.009 0.75752
-41.896 2.025 -4.005 0.68783
-43.037 2.024 -4.0 0.62847
```

Obserwujemy, zgodnie z oczekiwaniami, stopniowy spadek błędu w miarę postępu symulacji. Ponieważ funkcja błędu ma teraz trzy niezależne argumenty, nie można jej narysować w dwóch wymiarach. Możemy jednak pokazać jej rzut, np. dla ustalonej wartości w_0, co robimy poniżej:

```
fig, ax = plt.subplots(figsize=(3.7,3.7),dpi=120)

delta = 0.5
ran=40
r1 = np.arange(w1_o-ran, w1_o+ran, delta)
r2 = np.arange(w2_o-ran, w2_o+ran, delta)
X, Y = np.meshgrid(r1, r2)

Z=np.array([[error(w0_o,r1[i],r2[j],samp2,func.sig)
             for i in range(len(r1))] for j in range(len(r2))])

CS = ax.contour(X, Y, Z,[0,2,5,10,15,20,25,30,35,40,45,50])
ax.clabel(CS, inline=1, fmt='%1.0f', fontsize=9)

ax.set_title('Error function for $w_0$='+str(np.round(w0_o,2)), fontsize=11)
ax.set_aspect(aspect=1)
ax.set_xlabel('$w_1$', fontsize=11)
ax.set_ylabel('$w_2$', fontsize=11)

ax.scatter(w1_o, w2_o, s=20,c='red',label='found minimum') # our found optimal␣
 ↪point

ax.legend()
plt.show()
```



Informacja: W miarę jak wykonujemy coraz więcej iteracji, zauważamy, że wielkość wag rośnie, podczas gdy błąd naturalnie się zmniejsza. Powodem jest to, że nasza próbka danych jest separowalna, więc w przypadku użycia schodkowej funkcji aktywacji możliwe jest rozdzielenie próbki linią podziału i zejście z błędem aż do zera. W przypadku sigmoidu, zawsze istnieje pewien (niewielki) wkład do błędu, ponieważ wartości funkcji mieszczą się w sposób ciągły w przedziale (0,1). Jak omówiliśmy powyżej, w sigmoidzie, którego argumentem jest $(w_0 + w_1 x_1 + w_2 x_2)/T$, zwiększanie wag jest równoznaczne ze zmniejszaniem temperatury T. W moare postępu symulacji sigmoid zbliża się zatem do funkcji schodkowej, a błąd dąży do zera. Zachowanie to jest widoczne w powyższych symulacjach.

6.3 Najstromszy spadek

Powodem dla powyższych symulacji było doprowadzenie czytelnika do wniosku, że zagadnienie optymalizacji wag można sprowadzić do ogólnego problemu minimalizacji funkcji wielu zmiennych. Jest to standardowy (choć na ogół trudny) problem w analizie matematycznej i metodach numerycznych. Problemy związane ze znalezieniem minimum funkcji wielu zmiennych są dobrze znane:

- mogą istnieć minima lokalne, dlatego znalezienie minimum globalnego może być bardzo trudne;
- minimum może być w nieskończoności (czyli matematycznie nie istnieć);
- Funkcja wokół minimum może być bardzo płaska, tj. jej gradient jest bardzo mały. Wówczas znajdowanie minimum z pomocą metod gradientowych jest bardzo powolne;

Ogólnie rzecz biorąc, minimalizacja numeryczna funkcji to sztuka! Opracowano tu wiele metod, a właściwy dobór do danego problemu ma kluczowe znaczenie dla sukcesu. Poniżej zastosujemy najprostszy wariant, tzw. metodę **najstromszego spadku**.

Dla różniczkowalnej funkcji wielu zmiennych $F(z_1, z_2, ..., z_n)$, lokalnie najbardziej strome nachylenie jest okreslone przez minus gradient funkcji F, $-\left(\frac{\partial F}{\partial z_1}, \frac{\partial F}{\partial z_2}, ..., \frac{\partial F}{\partial z_n}\right)$,

gdzie pochodne cząstkowe definiuje się jako granice

$$\frac{\partial F}{\partial z_1} = \lim_{\Delta \to 0} \frac{F(z_1 + \Delta, z_2, ..., z_n) - F(z_1, z_2, ..., z_n)}{\Delta}$$

i podobnie dla pozostałych z_i.

Metoda znajdowania minimum funkcji poprzez najstromszy spadek zadana jest przez algorytm iteracyjny, w którym aktualizujemy współrzędne (wyszukiwanego minimum) w każdym kroku iteracji m (górny wskaźnik) w nastepujacy sposób:

$$z_i^{(m+1)} = z_i^{(m)} - \epsilon \frac{\partial F}{\partial z_i}.$$

W naszym zagadnieniu potrzebujemy zminimalzować funcję błedu

$$E(w_0, w_1, w_2) = \sum_p [y_o^{(p)} - y_t^{(p)}]^2 = \sum_p [\sigma(s^{(p)}) - y_t^{(p)}]^2 = \sum_p [\sigma(w_0 x_0^{(p)} + w_1 x_1^{(p)} + w_2 x_2^{(p)}) - y_t^{(p)}]^2.$$

Aby obliczyć pochodne, stosujemy **twierdzenie o pochodnej funkcji złożonej**.

Tw. o pochodnej funkcji złożonej

Dla funkcji złożonej

$[f(g(x))]' = f'(g(x))g'(x).$

Dla złożenia większej liczby funkcji $[f(g(h(x)))]' = f'(g(h(x)))\, g'(h(x))\, h'(x)$ itp.

Prowadzi to do wzoru

$$\frac{\partial E}{\partial w_i} = \sum_p 2[\sigma(s^{(p)}) - y_t^{(p)}]\, \sigma'(s^{(p)})\, x_i^{(p)} = \sum_p 2[\sigma(s^{(p)}) - y_t^{(p)}]\, \sigma(s^{(p)})\, [1 - \sigma(s^{(p)})]\, x_i^{(p)}$$

(pochodna funkcji kwadratowej $\times$ pochodna sigmoidu $\times$ pochodna $s^{(p)}$), gdzie w ostatniej równości użyliśmy specjalnej własności pochodnej sigmoidu. Metoda najstromszego spadku aktualizuje zatem wagi w następujący sposób:

$$w_i \to w_i - \varepsilon (y_o^{(p)} - y_t^{(p)}) y_o^{(p)} (1 - y_o^{(p)}) x_i.$$

Zauważmy, że aktualizacja zawsze występuje, ponieważ odpowiedź $y_o^{(p)}$ nigdy nie jest ściśle równa 0 lub 1, podczas gdy prawdziwa wartość (etykieta) $y_t^{(p)}$ wynosi 0 lub 1.

Ponieważ $y_o^{(p)}(1-y_o^{(p)}) = \sigma(s^{(p)})[1-\sigma(s^{(p)})]$ jest istotnie różne od zera tylko w okolicy $s^{(p)} = 0$ (patrz wcześniejszy wykres pochodnej sigmoidu), znacząca aktualizacja następuje tylko w pobliżu progu. To cecha jest odpowiednia, ponieważ problemy z błędną klasyfikacją zdarzają się właśnie w pobliżu linii podziału.

Informacja: Dla porównania, wcześniejszy algorytm perceptronu jest strukturalnie bardzo podobny,

$$w_i \to w_i - \varepsilon\, (y_o^{(p)} - y_t^{(p)})\, x_i,$$

ale tutaj aktualizacja następuje dla wszystkich punktów próbki, a nie tylko tych w pobliżu linii podziału.

Kod algorytmu uczenia naszego perceptronu metodą najstromszyego spadku jest następujący:

```
def teach_sd(sample, eps, w_in): # Steepest descent for the perceptron

    [[w0],[w1],[w2]]=w_in               # initial weights
    for i in range(len(sample)):        # loop over the data sample
        for k in range(10):             # repeat 10 times

            yo=func.sig(w0+w1*sample[i,0]+w2*sample[i,1])  # obtained answer for␣
 ↪pont i

            w0=w0+eps*(sample[i,2]-yo)*yo*(1-yo)*1             # update of weights
            w1=w1+eps*(sample[i,2]-yo)*yo*(1-yo)*sample[i,0]
            w2=w2+eps*(sample[i,2]-yo)*yo*(1-yo)*sample[i,1]
    return [[w0],[w1],[w2]]
```

Jego wydajność jest podobna do oryginalnego algorytmu perceptronu badanego powyżej:

```
weights=[[func.rn()],[func.rn()],[func.rn()]]      # random weights from [-0.5,0.
↪5]

print("   w0   w1/w0  w2/w0 error")   # header

eps=0.7                        # initial learning speed
for r in range(1000):          # rounds
    eps=0.9995*eps             # decrease learning speed
    weights=teach_sd(samp2,eps,weights) # update weights
    if r%100==99:
        w0_o=weights[0][0]               # updated weights
        w1_o=weights[1][0]
        w2_o=weights[2][0]
        v1_o=w1_o/w0_o
        v2_o=w2_o/w0_o
        print(np.round(w0_o,3),np.round(v1_o,3),np.round(v2_o,3),
              np.round(error(w0_o, w0_o*v1_o, w0_o*v2_o, samp2, func.sig),5))
```

```
   w0   w1/w0  w2/w0 error
-10.373 1.939 -3.972 2.70531
-13.308 1.966 -3.971 1.96428
-15.208 1.983 -3.975 1.62302
-16.622 1.995 -3.978 1.41622
-17.754 2.003 -3.979 1.27377
-18.699 2.008 -3.98 1.16816
-19.51 2.011 -3.979 1.08604
-20.22 2.014 -3.978 1.02005
-20.85 2.015 -3.976 0.96571
-21.414 2.016 -3.974 0.92013
```

Podsumowując dotychczasowy materiał, wykazaliśmy, że można skutecznie uczyć jednowarstwy perceptron (pojedynczy neuronu MCP) za pomocą metody najstromszego spadku, minimalizując funkcję błędu generowaną przez badaną próbkę. W następnym podrozdziale uogólnimy ten pomysł na dowolny wielowarstwową sieć typu feed-forward.

6.4 Algorytm propagacji wstecznej (backprop)

Materiał tego podrozdziału jest absolutnie **kluczowy** dla zrozumienia idei uczenia sieci neuronowych poprzez uczenie nadzorowane. Jednocześnie dla czytelnika mniej zaznajomionego z analizą matematyczną może być dość trudny, ponieważ pojawiają się wyprowadzenia i wzory z bogatą notacją. Nie udało się jednak znaleźć sposobu na przedstawienie materiału w prostszy sposób niż poniżej, z jednoczesnym zachowaniem niezbędnego rygoru.

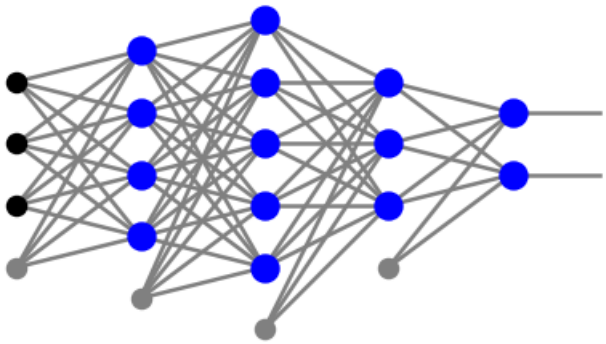
Informacja: Formuły, które wyprowadzamy tutaj krok po kroku, stanowią słynny **algorytm wstecznej propagacji (backprop)** [BH69] dla aktualizacji wag perceptronu wielowarstwowego. Wykorzystujemy tylko dwa podstawowe fakty:

- **tw. o pochodnej funkcji złożonej** do obliczania pochodnej, oraz
- **metodę najstromszego spadku**, wyjaśnioną w poprzednim podrozdziale.

Rozważmy perceptron z dowolną liczbą warstw neuronowych, l. Neurony w warstwach pośrednich $j = 1, \dots, l-1$ są ponumerowane odpowiednimi wskaźnikami $\alpha_j = 0, \dots, n_j$, gdzie 0 oznacza węzeł progowy. W warstwie wyjściowej, nie zawierającej węzła progowego, wskaźnik przyjmuje wartości $\alpha_l = 1, \dots, n_l$. Na przykład sieć z wykresu poniżej ma

$$l = 4, \quad \alpha_1 = 0, \dots, 4, \quad \alpha_2 = 0, \dots, 5, \quad \alpha_3 = 0, \dots, 3, \quad \alpha_4 = 1, \dots, 2,$$

ze wskaźnikami w każdej warstwie liczonymi od dołu.



Funkcja błędu to suma po punktach próbki treningowej oraz dodatkowo po węzłach w warstwie wyjściowej:

$$E(\{w\}) = \sum_{p} \sum_{\alpha_l=1}^{n_l} \left[y^{(p)}_{o,\alpha_l}(\{w\}) - y^{(p)}_{t,\alpha_l} \right]^2,$$

gdzie $\{w\}$ reprezentują wszystkie wagi sieci. Pojedynczy wkład punktu p do E, oznaczony jako e, to suma po wszystkich neuronach w warstwie wyjściowej:

$$e(\{w\}) = \sum_{\alpha_l=1}^{n_l} \left[y_{o,\alpha_l} - y_{t,\alpha_l} \right]^2.$$

Dla zwięzłości, opuściliśmy górny wskaźnik (p). Dla neuronu α_j w warstwie j sygnałem wejściowym jest

$$s^j_{\alpha_j} = \sum_{\alpha_{j-1}=0}^{n_{j-1}} x^{j-1}_{\alpha_{j-1}} w^j_{\alpha_{j-1}\alpha_j}.$$

Sygnały w warstwie wyjściowej mają postać

$$y_{o,\alpha_l} = f\left(s^l_{\alpha_l}\right)$$

natomiast sygnały wyjściowe w warstwach pośrednich $j = 1, \ldots, l-1$ to

$$x^j_{\alpha_j} = f\left(s^j_{\alpha_j}\right), \quad \alpha_j = 1, \ldots, n_j, \quad \text{i} \quad x^j_0 = 1,$$

z węzłem progowym mającym wartość 1.

Kolejne podstawienia powyższych formuł do e są następujące:

$e = \sum_{\alpha_l=1}^{n_l} \left(y_{o,\alpha_l} - y_{t,\alpha_l}\right)^2$

$= \sum_{\alpha_l=1}^{n_l} \left(f\left(\sum_{\alpha_{l-1}=0}^{n_{l-1}} x^{l-1}_{\alpha_{l-1}} w^l_{\alpha_{l-1}\alpha_l} \right) - y_{t,\alpha_l} \right)^2$

$= \sum_{\alpha_l=1}^{n_l} \left(f\left(\sum_{\alpha_{l-1}=1}^{n_{l-1}} f\left(\sum_{\alpha_{l-2}=0}^{n_{l-2}} x^{l-2}_{\alpha_{l-2}} w^{l-1}_{\alpha_{l-2}\alpha_{l-1}} \right) w^l_{\alpha_{l-1}\alpha_l} + x^{l-1}_0 w^l_{0\gamma} \right) - y_{t,\alpha_l} \right)^2$

$= \sum_{\alpha_l=1}^{n_l} \left(f\left(\sum_{\alpha_{l-1}=1}^{n_{l-1}} f\left(\sum_{\alpha_{l-2}=1}^{n_{l-2}} f\left(\sum_{\alpha_{l-3}=0}^{n_{l-3}} x^{l-3}_{\alpha_{l-3}} w^{l-2}_{\alpha_{l-3}\alpha_{l-2}} \right) w^{l-1}_{\alpha_{l-2}\alpha_{l-1}} + x^{l-2}_0 w^{l-1}_{0\alpha_{l-1}} \right) w^l_{\alpha_{l-1}\alpha_l} + x^{l-1}_0 w^l_{0\alpha_l} \right) - y_{t,\alpha_l} \right)^2$

$= \sum_{\alpha_l=1}^{n_l} \left(f\left(\sum_{\alpha_{l-1}=1}^{n_{l-1}} f\left(\ldots f\left(\sum_{\alpha_0=0}^{n_0} x^0_{\alpha_0} w^1_{\alpha_0\alpha_1} \right) w^2_{\alpha_1\alpha_2} + x^1_0 w^2_{0\alpha_2} \ldots \right) w^l_{\alpha_{l-1}\alpha_l} + x^{l-1}_0 w^l_{0\alpha_l} \right) - y_{t,\alpha_l} \right)^2$

Obliczając kolejne pochodne względem wag idąc wstecz, tj. od $j = l$ do 1, otrzymujemy (patrz ćwiczenia)

$$\frac{\partial e}{\partial w^j_{\alpha_{j-1}\alpha_j}} = x^{j-1}_{\alpha_{j-1}} D^j_{\alpha_j}, \quad \alpha_{j-1} = 0, \ldots, n_{j-1}, \quad \alpha_j = 1, \ldots, n_j,$$

gdzie

$D^l_{\alpha_l} = 2(y_{o,\alpha_l} - y_{t,\alpha_l})\, f'(s^l_{\alpha_l}),$

$D^j_{\alpha_j} = \sum_{\alpha_{j+1}} D^{j+1}_{\alpha_{j+1}} w^{j+1}_{\alpha_j \alpha_{j+1}} f'(s^j_{\alpha_j}), \quad j = l-1, l-2, \ldots, 1.$

Ostatnie wyrażenie to rekurencja wstecz. Zauważamy, że aby uzyskać D^j, potrzebujemy D^{j+1}, które uzyskaliśmy już w poprzednim kroku, oraz sygnał s^j, który znamy z propagacji sygnału do przodu. Ta rekurencja prowadzi do uproszczenia obliczania pochodnych i aktualizacji wag.

Przy najstromszym spadku wagi są aktualizowane jako

$$w^j_{\alpha_{j-1}\alpha_j} \to w^j_{\alpha_{j-1}\alpha_j} - \varepsilon x^{j-1}_{\alpha_{j-1}} D^j_{\alpha_j},$$

W przypadku sigmoidu możemy użyć

$$\sigma'(s^{(i)}_A) = \sigma'(s^{(i)}_A)(1 - \sigma'(s^{(i)}_A)) = x^{(i)}_A (1 - x^{(i)}_A).$$

Informacja: Powyższe formuły wyjaśniają nazwę **propagacja wsteczna**, ponieważ w aktualizacji wag zaczynamy od ostatniej warstwy, a następnie posuwamy się rekurencyjnie do początku sieci. Na każdym kroku potrzebujemy tylko sygnału w danej warstwie i właściwości kolejnej warstwy! Te cechy wynikają z

1. charakteru feed-forward sieci oraz
2. tw. o pochodnej funkcji złożonej.

Ważne: Praktyczne znaczenie cofania się warstwa po warstwie polega na tym, że w jednym kroku aktualizuje się znacznie mniej wag: tylko te, które wchodzą do danej warstwy, a nie wszystkie naraz. Ma to znaczenie dla zbieżności metody najstromszego spadku, zwłaszcza dla sieci głębokich (o wielu warswach).

Jeżeli funkcje aktywacji są różne w różnych warstwach (oznaczamy je f_j dla warstwy j), to zachodzi oczywista modyfikacja:

$D^l_{\alpha_l} = 2(y_{o,\alpha_l} - y_{t,\alpha_l})\, f'_l(s^l_{\alpha_l}),$

$D^j_{\alpha_j} = \sum_{\alpha_{j+1}} D^{j+1}_{\alpha_{j+1}} w^{j+1}_{\alpha_j \alpha_{j+1}} f'_j(s^j_{\alpha_j}), \quad j = l-1, l-2, \ldots, 1.$

Nie jest to rzadkie, ponieważ w wielu zastosowaniach wybiera się różne funkcje aktywacji dla warstw pośrednich i warswy wyjściowej.

6.4.1 Kod dla algorytmu backprop

Następnie przedstawimy prosty kod realizujący algorytm backprop. Jest to bezpośrednia implementacja wyprowadzonych powyżej formuł. W kodzie zachowujemy jak najwięcej notacji z powyższego wyprowadzenia.

Kod ma tylko 12 linijek, nie licząc komentarzy!

```
def back_prop(fe,la, p, ar, we, eps,f=func.sig,df=func.dsig):
    """
    fe - array of features
    la - array of labels
    p  - index of the used data point
    ar - array of numbers of nodes in subsequent layers
    we - disctionary of weights
    eps - learning speed
    f   - activation function
    df  - derivaive of f
    """

    l=len(ar)-1 # number of neuron layers (= index of the output layer)
    nl=ar[l]    # number of neurons in the otput layer
```

(ciąg dalszy na następnej stronie)

(kontynuacja poprzedniej strony)

```
    x=func.feed_forward(ar,we,fe[p],ff=f) # feed-forward of point p

    # formulas from the derivation in a one-to-one notation:

    D={}
    D.update({l: [2*(x[l][gam]-la[p][gam])*
                   df(np.dot(x[l-1],we[l]))[gam] for gam in range(nl)]})
    we[l]-=eps*np.outer(x[l-1],D[l])

    for j in reversed(range(1,l)):
        u=np.delete(np.dot(we[j+1],D[j+1]),0)
        v=np.dot(x[j-1],we[j])
        D.update({j: [u[i]*df(v[i]) for i in range(len(u))]})
        we[j]-=eps*np.outer(x[j-1],D[j])
```

6.5 Przykład z kołem

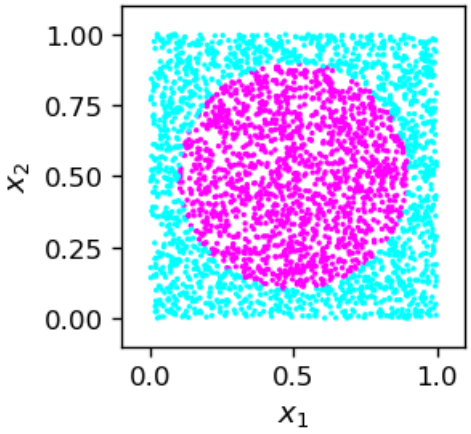
Kod ilustrujemy na przykładzie klasyfikatora binarnego punktów wewnątrz okręgu.

```
def cir():
    x1=np.random.random()                  # coordinate 1
    x2=np.random.random()                  # coordinate 2
    if((x1-0.5)**2+(x2-0.5)**2 < 0.4**2):  # inside circle, radius 0.4, center (0.
↪5,0.5)
        return np.array([x1,x2,1])
    else:                                  # outside
        return np.array([x1,x2,0])
```

Do przyszłego użytku **(nowa konwencja)** podzielimy próbkę na oddzielne tablice **cech** (dwie współrzędne) i **etykiet** (1, jeśli punkt znajduje się wewnątrz okręgu, 0 w przeciwnym razie):

```
sample_c=np.array([cir() for _ in range(3000)]) # sample
features_c=np.delete(sample_c,2,1)
labels_c=np.delete(np.delete(sample_c,0,1),0,1)
```

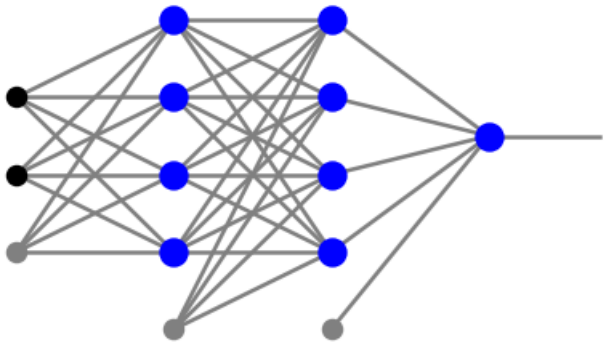
```
plt.figure(figsize=(2.3,2.3),dpi=120)
plt.xlim(-.1,1.1)
plt.ylim(-.1,1.1)
plt.scatter(sample_c[:,0],sample_c[:,1],c=sample_c[:,2],
            s=1,cmap=mpl.cm.cool,norm=mpl.colors.Normalize(vmin=0, vmax=.9))

plt.xlabel('$x_1$',fontsize=11)
plt.ylabel('$x_2$',fontsize=11)
plt.show()
```



Dobieramy następującą architekturę i początkowe parametry:

```
arch_c=[2,4,4,1]                    # architecture
weights=func.set_ran_w(arch_c,4)    # scaled random initial weights in [-2,2]
eps=.7                              # initial learning speed
```

```
plt.show(draw.plot_net(arch_c))
```



Symulacja zabiera kilka minut.

```
for k in range(1000):    # rounds
    eps=.995*eps         # decrease learning speed
    if k%100==99: print(k+1,' ',end='')             # print progress
    for p in range(len(features_c)):                # loop over points
        func.back_prop(features_c,labels_c,p,arch_c,weights,eps,
                       f=func.sig,df=func.dsig) # backprop
```

```
100  200  300  400  500  600  700  800  900  1000
```

Zmniejszenie szybkości uczenia się w każdej rundzie daje końcową wartość ε, która powinna być niewielka, ale nie za mała:

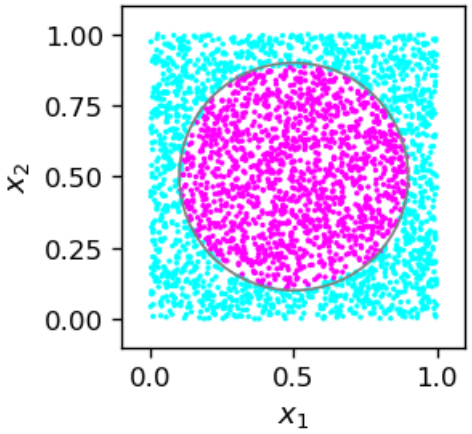
```
eps
```

```
0.004657778005182377
```

(zbyt mała wartość aktualizowałaby wagi w znikomy sposób, więc dalsze rundy byłyby bezużyteczne).

Podczas gdy faza nauki była dość długa, testowanie i używanie wytrenowanej sieci przebiega bardzo szybko:

```
test=[]

for k in range(3000):
    po=[np.random.random(),np.random.random()]
    xt=func.feed_forward(arch_c,weights,po,ff=func.sig)
    test.append([po[0],po[1],np.round(xt[len(arch_c)-1][0],0)])

tt=np.array(test)

fig=plt.figure(figsize=(2.3,2.3),dpi=120)

# drawing the circle
ax=fig.add_subplot(1,1,1)
circ=plt.Circle((0.5,0.5), radius=.4, color='gray', fill=False)
ax.add_patch(circ)

plt.xlim(-.1,1.1)
plt.ylim(-.1,1.1)
plt.scatter(tt[:,0],tt[:,1],c=tt[:,2],
            s=1,cmap=mpl.cm.cool,norm=mpl.colors.Normalize(vmin=0, vmax=.9))

plt.xlabel('$x_1$',fontsize=11)
plt.ylabel('$x_2$',fontsize=11)
plt.show()
```



Wytrenowana sieć wygląda następująco:

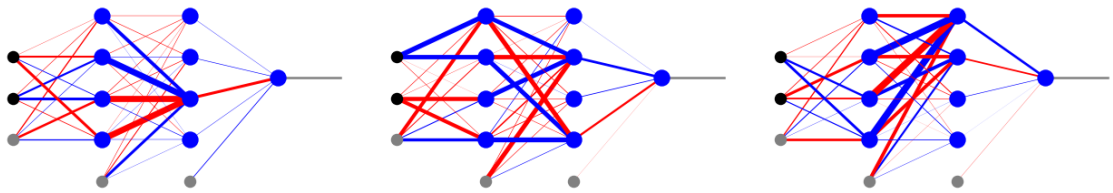
```
fnet=draw.plot_net_w(arch_c,weights,.1)
```

Informacja: To fascynujące, że nauczyliśmy sieć rozpoznawać, czy punkt znajduje się w okręgu, a nie ma ona żadnego pojęcia o geometrii, odległości euklidesowej, równaniu okręgu itp. Sieć właśnie nauczyła się „empirycznie", jak postępować, za pomocą próbki szkoleniowej!

Informacja: Wynik przedstawiony na rysunku jest całkiem niezły, może z wyjątkiem, jak zwykle, punktów blisko granicy. Biorąc pod uwagę naszą dyskusję w rozdz. *Więcej warstw*, w którym wyznaczyliśmy wagi sieci z trzema warstwami neuronów na podstawie rozważań geometrycznych, jakość prezentowanego wyniku jest oszałamiająca. Nie widzimy żadnych prostych boków wielokąta, ale ładnie zaokrągloną granicę. Dalsza poprawa wyniku wymagałaby większej liczebności próbki szkoleniowej i dłuższego treningu, co jest czasochłonne.

Lokalne minima

Wspomnieliśmy wcześniej o pojawianiu się minimów lokalnych w optymalizacji wielowymiarowej jako o potencjalnym problemie. Na poniższym rysunku pokazujemy trzy różne wyniki kodu backprop dla naszego klasyfikatora punktów w okręgu. Zauważamy, że każdy z nich ma radykalnie inny zestaw optymalnych wag, podczas gdy spawdzenie na próbce testowej jest, przynajmniej na oko, równie dobre dla każdego przypadku. To pokazuje, że optymalizacja backprop prowadzi, zgodnie z przewidywaniami, do różnych minimów lokalnych. Jednak każde z nich działa wystarczająco i równie dobrze. To jest właśnie powód, dla którego algorytm backprop można wykorzystać w praktycznych problemach: istnieją miliony lokalnych minimów, ale to naprawdę nie ma znaczenia!



6.6 Ogólne uwagi

Należy poczynić kilka istotnych i ogólnych obserwacji:

Informacja:

- Uczenie nadzorowane zajmuje bardzo dużo czasu, ale użycie wytrenowanej sieci trwa mgnienie oka. Asymetria wynika z prostego faktu, że optymalizacja wieloparametrowa wymaga bardzo wielu wywołań funkcji (tutaj **feed-forward**) i obliczneia pochodnych w wielu rundach (użyliśmy 1000 rund dla przykładu okręgu), ale użycie sieci dla przypadku jednego punktu wymaga tylko jednego wywołania funkcji.
- Klasyfikator wyszkolony algorytmem backprop może działać niedokładnie dla punktów w pobliżu linii granicznych. Środkiem zaradczym jest dłuższe trenowanie i/lub zwiększenie liczebności próbki szkoleniowej, w szczególności w pobliżu granicy.
- Jednak zbyt długa nauka na tej samej próbce treningowej nie ma sensu, ponieważ w pewnym momencie dokładność przestaje się poprawiać.
- Lokalne minima są powszecjne, ale w żadnym wypadku nie stanowi to przeszkody w stosowaniu algorytmu. To ważna praktyczna cecha.
- Można stosować różne ulepszenia metody najstromszego spadku lub zupełnie inne metody minimalizacji (patrz ćwiczenia). Mogą one znacznie zwiększyć wydajność algorytmu.
- Cofając się z aktualizacją wag w kolejnych warstwach, można wprowadzić współczynnik zwiększający uaktualnianie (patrz ćwiczenia). To pomaga w wydajności.
- Wreszcie, inne funkcje aktywacji mogą być używane do poprawy wydajności (patrz kolejne wykłady).

6.7 Ćwiczenia

1. Udowodnij (analitycznie), obliczając pochodną, że $\sigma'(s) = \sigma(s)[1 - \sigma(s)]$. Pokaż, że sigmoid jest **jedyną** funkcją z tą właściwością.
2. Wyprowadź jawnie wzory algorytmu backprop dla sieci z jedną i dwiema warstwami pośrednimi. Zwróć uwagę na pojawiającą się prawidłowość (powtarzalność) i udowodnij ogólne wzory z wykładu dla dowolnej liczby warstw pośrednich.
3. Zmodyfikuj przykład z wykładu dla klasyfikatora punktów w okręgu dla:

- półkola;
- dwóch rozłącznych okręgów;
- pierścienia;
- dowolnego z twoich ulubionych kształtów.

4. Powtórz 3, eksperymentując z liczbą warstw i neuronów, ale pamiętaj, że duża ich liczba wydłuża czas obliczeń i niekoniecznie poprawia wynik. Uszereguj każdy przypadek według liczby błędnie sklasyfikowanych punktów w próbce testowej. Znajdź optymalną/praktyczną architekturę dla każdego z rozważanych obszarów.

5. Jeśli sieć ma dużo neuronów i połączeń, przez każdą synapsę przepływa mało sygnału, stąd sieć jest odporna na niewielkie przypadkowe uszkodzenia. Tak dzieje się w naszym mózgu, który jest nieustannie „uszkadzany" (promienie kosmiczne, alkohol,…). Poza tym taką sieć po zniszczeniu można (już przy mniejszej liczbie połączeń) dodatkowo doszkolić. Weź wytrenowaną sieć z problemu 3. i usuń jedno z jej **słabych** połączeń (najpierw znajdź je, sprawdzając wagi), zmieniając odpowiednią wagę na 0. Przetestuj taką uszkodzoną sieć na próbce testowej i wyciągnij wnioski.

6. **Skalowanie wag w propagacji wstecznej.** Wadą zastosowania sigmoidu w algorytmie backprop jest bardzo powolna aktualizacja wag w warstwach odległych od warstwy wyjściowej (im bliżej początku sieci, tym wolniej). Remedium jest tutaj przeskalowanie wag, gdzie szybkość uczenia się warstw, licząc od tyłu, jest sukcesywnie zwiększana o pewien współczynnik. Pamiętamy, że kolejne pochodne wnoszą do szybkości aktualizacji współczynniki postaci $\sigma'(s) = \sigma(s)[1 - \sigma(s)] = y(1 - y)$, gdzie y wynosi w zakresie $(0, 1)$. Zatem wartość $y(1-y$ nie może przekraczać 1/4, a w kolejnych warstwach (licząc od tyłu) czynnika $[y(1-y]^n \leq 1/4^n)$. Aby zapobiec temu „kurczeniu się", wskaźnik uczenia się można przemnażać przez współczynniki kompensacyjne $4^n : 4, 16, 64, 256, ...$ Kolejny argument heurystyczny [RIV91] sugeruje jeszcze szybciej rosnące czynniki postaci 6^n:$6, 36, 216, 1296, ...$

   - Wprowadź powyższe dwie receptury do kodu backprop.
   - Sprawdź, czy rzeczywiście poprawiają wydajność algorytmu dla głębszych sieci, na przykład dla klasyfikatora punktów okręgu itp.
   - W celu oceny wydajności wykonaj pomiar czasu wykonania (np. za pomocą pakietu biblioteki Python **time**).

7. **Najstromsze spadek.** Zastosowana w wykładzie metoda najstromszego spadku do wyznaczania minimum funkcji wielu zmiennych zależy od gradientu lokalnego. Istnieją znacznie lepsze podejścia, które zapewniają szybszą zbieżność do (lokalnego) minimum. Jednym z nich jest przepis Barzilai-Borwein wyjaśniony poniżej. Zaimplementuj tę metodę w algorytmie wstecznej propagacji. Wektory x w przestrzeni n-wymiarowej są aktualizowane w kolejnych iteracjach jako $x^{(m+1)} = x^{(m)} - \gamma_m \nabla F(x^{(m)})$, gdzie m numeruje iterację, a szybkość uczenia się zależy od zachowania w dwóch (bieżącym i poprzednim) punktach:

$$\gamma_m = \frac{\left|\left(x^{(m)} - x^{(m-1)}\right) \cdot \left[\nabla F(x^{(m)}) - \nabla F(x^{(m-1)})\right]\right|}{\left\|\nabla F(x^{(m)}) - \nabla F(x^{(m-1)})\right\|^2}.$$

ROZDZIAŁ 7

Interpolacja

7.1 Symulowane dane

Do tej pory zajmowaliśmy się **klasyfikacją**, czyli rozpoznawaniem przez sieci, czy dany obiekt (w naszym przykładzie punkt na płaszczyźnie) ma określone cechy. Teraz przechodzimy do innego praktycznego zastosowania, a mianowicie do **interpolacji** funkcji. To zastosowanie ANN stało się bardzo popularne w analizie danych naukowych. Zilustrujemy tę metodę na prostym przykładzie, który wyjaśni podstawową ideę i pokaże, jak ona działa.

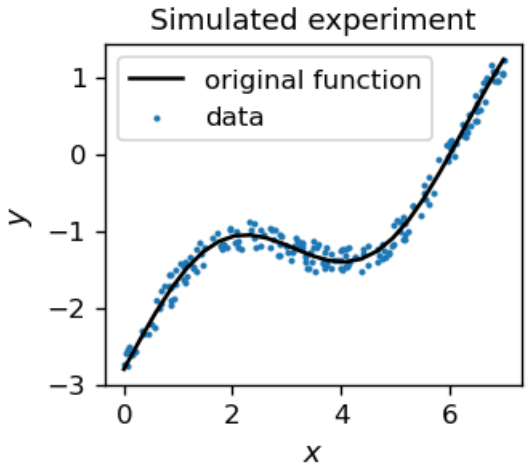
Wyobraźmy sobie, że dysponujemy pewnymi danymi eksperymentalnymi. W tym przypadku symulujemy je w sztuczny sposób, np.

```
def fi(x):
    return 0.2+0.8*np.sin(x)+0.5*x-3 # a function

def data():
    x = 7.*np.random.rand() # random x coordinate
    y = fi(x)+0.4*func.rn() # y coordinate = the function value + noise from [-0.
 ↪2,0.2]
    return [x,y]
```

Powinniśmy teraz myśleć w kategoriach uczenia nadzorowanego: x to „cecha”, a y to „etykieta”.

Tablicujemy nasze zaszumione punkty danych i wykreślamy je wraz z funkcją **fi(x)**, wokół której się wahają. Jest to imitacja pomiaru eksperymentalnego, który zawsze obarczony jest pewnym błędem, tutaj naśladowanym przez losowy szum.

```
tab=np.array([data() for i in range(200)])     # data sample
features=np.delete(tab,1,1)                    # x coordinate
labels=np.delete(tab,0,1)                      # y coordinate
```



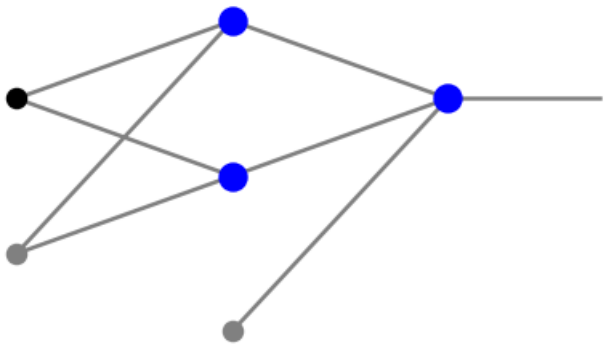
W języku ANN mamy zatem próbkę treningową składającą się z punktów o danych wejściowych (cechach) x i „prawdziwych" danych wyjściowych (etykietach) y. Tak jak poprzednio, minimalizujemy funkcję błędu odpowiedniej sieci neuronowej,

$$E(\{w\}) = \sum_p (y_o^{(p)} - y^{(p)})^2.$$

Ponieważ generowane y_o jest pewną (zależną od wag) funkcją x, metoda ta jest odmianą **dopasowania najmniejszych kwadratów**, powszechnie stosowaną w analizie danych. Różnica polega na tym, że w standardowej metodzie najmniejszych kwadratów funkcja modelu, którą dopasowujemy do danych, ma pewną prostą postać analityczną (np. $f(x) = A + Bx$), podczas gdy teraz jest to pewna „zakamuflowana" funkcja zależna od wag, dostarczona przez sieć neuronową.

7.2 Interpolacja z pomocą ANN

Aby zrozumieć podstawową ideę, rozważmy sieć z tylko dwoma neuronami w warstwie pośredniej, z sigmoidalną funkcją aktywacji:



Sygnały docierające do dwóch neuronów w warstwie środkowej to, w notacji z rozdz. *Więcej warstw*,

$$s_1^1 = w_{01}^1 + w_{11}^1 x,$$

$$s_2^1 = w_{02}^1 + w_{12}^1 x,$$

a sygnały wychodzące to odpowiednio,

$$\sigma\left(w_{01}^1 + w_{11}^1 x\right),$$

$$\sigma\left(w_{02}^1 + w_{12}^1 x\right).$$

Zatem połączony sygnał wchodzący do neuronu wyjściowego ma postać

$$s_1^2 = w_{01}^2 + w_{11}^2 \sigma\left(w_{01}^1 + w_{11}^1 x\right) + w_{21}^2 \sigma\left(w_{02}^1 + w_{12}^1 x\right).$$

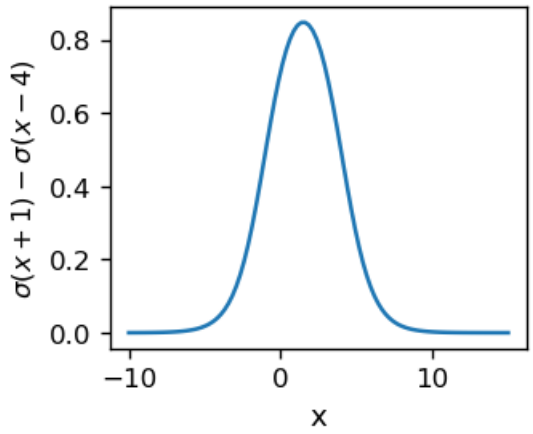
Przyjmując, dla ilustracji, przykładowe wartości wag

$$w_{01}^2 = 0, w_{11}^2 = 1, w_{21}^2 = -1, w_{21}^2, w_{11}^1 = w_{12}^1 = 1,\ w_{01}^1 = -x_1,\ w_{02}^1 = -x_2,$$

gdzie x_1 i x_2 to notacja skrótowa, otrzymujemy

$$s_1^2 = \sigma(x - x_1) - \sigma(x - x_2).$$

Funkcja ta jest przedstawiona na poniższym wykresie, gdzie $x_1 = -1$ i $x_2 = 4$. Dąży ona do 0 w $-\infty$, potem rośnie wraz z x, osiągając maksimum w punkcie $(x_1 + x_2)/2$, a następnie maleje, dążąc do 0 przy $+\infty$. W punktach $x = x_1$ i $x = x_2$ jej wartości wynoszą około 0.5, można więc powiedzieć, że przedział znaczących wartości funkcji zawiera się między x_1 i x_2.



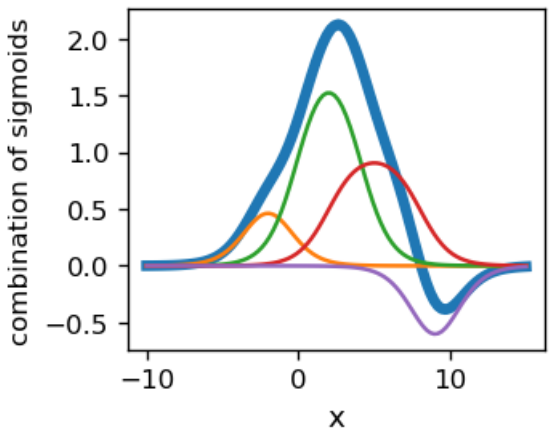
Jest to prosty, ale ważny wniosek: Jesteśmy w stanie utworzyć, za pomocą pary neuronów z sigmoidami, sygnał „garbowy", zlokalizowany wokół danej wartości, tutaj $(x_1 + x_2)/2 = 2$, i o danym rozrzucie rzędu $|x_2 - x_1|$. Zmieniając wagi, możemy modyfikować jej kształt, szerokość i wysokość.

Można teraz pomyśleć w następujący sposób: Wyobraźmy sobie, że mamy do dyspozycji wiele neuronów w warstwie pośredniej. Możemy je łączyć w pary, tworząc garby „specjalizujące się" w określonych regionach współrzędnych. Następnie, dostosowując wysokości garbów, możemy łatwo aproksymować daną funkcję.

W rzeczywistej procedurze dopasowania nie musimy „łączyć neuronów w pary", lecz dokonać łącznego dopasowania wszystkich parametrów jednocześnie, tak jak to miało miejsce w przypadku klasyfikatorów. Poniższy przykład przedstawia kompozycję 8 sigmoidów,

$$f = \sigma(z + 3) - \sigma(z + 1) + 2\sigma(z) - 2\sigma(z - 4) + \sigma(z - 2) - \sigma(z - 8) - 1.3\sigma(z - 8) - 1.3\sigma(z - 10).$$

Na rysunku funkcje składowe (cienkie linie oznaczające pojedyncze garby) sumują się do funkcji o dość skomplikowanym kształcie, oznaczonej grubą linią.



Informacja: Jeśli dopasowana funkcja jest regularna, można ją aproksymować za pomocą kombinacji liniowej sigmoidów. W przypadku większej liczby sigmoidów można uzyskać lepszą dokładność.

Istnieje istotna różnica między ANN używanymi do aproksymacji funkcji w porównaniu z omawianymi wcześniej klasyfikatorami binarnymi. Tam odpowiedzi były równe 0 lub 1, więc w warstwie wyjściowej stosowaliśmy skokową funkcję aktywacji, a raczej jej gładką odmianę sigmoidalną. W przypadku aproksymacji funkcji odpowiedzi stanowią zazwyczaj kontinuum w zakresie wartości funkcji. Z tego powodu w warstwie wyjściowej używamy po prostu funkcji **identycznościowej**, czyli przepuszczamy przez nią bez zmian przychodzący sygnał. Oczywiście sigmoidy pozostają w warstwach pośrednich. Wówczas wzory używane do algorytmu **backprop** z sekcji *Algorytm propagacji wstecznej (backprop)* mają w warstwie wyjściowej $f_l(s) = s$.

Warstwa wyjściowa dla aproksymacji funkcji

W sieciach ANN używanych do aproksymacji funkcji, funkcja aktywacji w warstwie wyjściowej jest **identycznościowa**.

7.2.1 Algorytm backprop dla funkcji jednowymiarowych

Weźmy architekturę

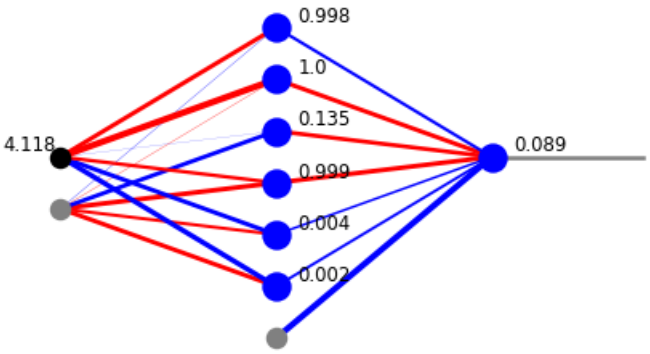
```
arch=[1,6,1]
```

i losowe wagi

```
weights=func.set_ran_w(arch, 5)
```

Jak właśnie wspomniano, wartość wyjściowa nie zawiera się teraz w przedziale od 0 do 1, co widać poniżej.

```
x=func.feed_forward_o(arch, weights,features[1],ff=func.sig,ffo=func.lin)
plt.show(draw.plot_net_w_x(arch, weights,1,x))
```



W module biblioteki **func** mamy funkcję dla algorytmu backprop, która pozwala na zastosowanie jednej funkcji aktywacji w warstwach pośrednich (przyjmujemy sigmoidę) i innej w warstwie wyjściowej (przyjmujemy funkcję identycznościową). Trening jest przeprowadzany w dwóch etapach: w pierwszych 30 rundach pobieramy punkty z próbki treningowej w losowej kolejności, a następnie w kolejnych 1500 rundach przecodzimy kolejno przez wszystkie punkty, zmniejszając również szybkość uczenia **eps**. Strategia ta jest jedną z wielu możliwych, ale w tym przypadku dobrze spełnia swoje zadanie.

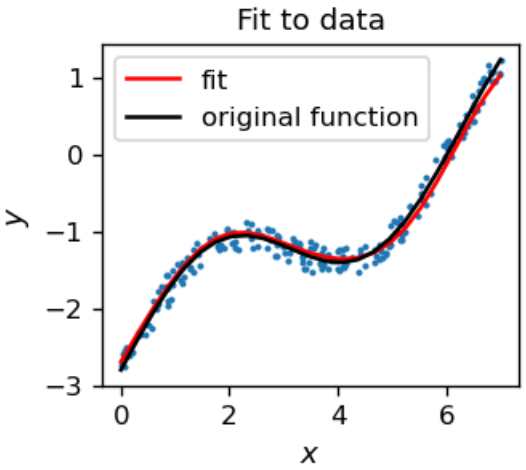
```
eps=0.02                          # initial learning speed
for k in range(30):               # rounds
    for p in range(len(features)): # loop over the data sample points
```

(ciąg dalszy na następnej stronie)

(kontynuacja poprzedniej strony)

```
        pp=np.random.randint(len(features)) # random point
        func.back_prop_o(features,labels,pp,arch,weights,eps,
                         f=func.sig,df=func.dsig,fo=func.lin,dfo=func.dlin)
```

```
for k in range(500):                # rounds
    eps=0.999*eps                  # dicrease of the learning speed
    for p in range(len(features)): # loop over points taken in sequence
        func.back_prop_o(features,labels,p,arch,weights,eps,
                         f=func.sig,df=func.dsig,fo=func.lin,dfo=func.dlin)
```



Zauważmy, że otrzymana czerwona krzywa jest bardzo bliska funkcji użytej do wygenerowania próbki danych (czarna linia). Świadczy to o tym, że aproksymacja działa poprawnie. Konstrukcja miary ilościowej (sumy najmniejszych kwadratów) jest tematem ćwiczenia.

Informacja: Funkcja aktywacji w warstwie wyjściowej może być dowolną gładką funkcją o wartościach zawierających wartości interpolowanej funkcji, niekoniecznie liniową.

Więcej wymiarów

Aby interpolować funkcje dwóch lub więcej argumentów, należy użyć sieci ANN z co najmniej trzema warstwami neuronów.

Możemy to rozumieć następująco [MullerRS12]: dwa neurony w pierwszej warstwie neuronowej mogą tworzyć garb we współrzędnej x_1, dwa inne - garb we współrzędnej x_2, i tak dalej dla wszystkich pozostałych wymiarów. Tworząc koniunkcję tych n garbów w drugiej warstwie neuronów, otrzymujemy funkcję bazową specjalizującą się w obszarze wokół pewnego punktu w wielowymiarowej przestrzeni wejściowej. A zatem odpowiednio duża liczba takich funkcji bazowych może być użyta do aproksymacji w n wymiarach, w pełnej analogii do przypadku jednowymiarowego.

Wskazówka: Liczba neuronów potrzebnych w procedurze aproksymacji odzwierciedla zachowanie interpolowanej funkcji. Jeśli funkcja ulega licznym znacznym wahaniom, potrzeba więcej neuronów. W jednym wymiarze jest ich zwykle co najmniej dwa razy więcej niż liczba ekstremów funkcji.

Nadmierne dopasowanie (overfitting)

Aby uniknąć tak zwanego **problemu nadmiernego dopasowania**, danych użytych do aproksymacji musi być znacznie więcej niż parametrów sieci. W przeciwnym razie moglibyśmy dopasować bardzo dokładnie dane treningowe

za pomocą funkcji „wahającej się od punktu do punktu”. Jednocześnie, działanie takiej sieci na danych testowych byłoby bardzo kiepskie.

7.3 Ćwiczenia

1. Dopasuj punkty danych wygenerowane przez Twoją ulubioną funkcję (jednej zmiennej) z szumem. Pobaw się architekturą sieci i wyciągnij wnioski.
2. Oblicz sumę kwadratów odległości między wartościami punktów danych treningowych a odpowiadającą im funkcją aproksymującą i wykorzystaj ją jako miarę jakości dopasowania. Sprawdź, jak liczba neuronów w sieci wpływa na wynik.
3. Użyj sieci o większej liczbie warstw (co najmniej 3 warstwy neuronów) do dopasowania punktów danych wygenerowanych za pomocą ulubionej funkcji dwóch zmiennych. Wykonaj dwuwymiarowe wykresy konturowe dla tej funkcji oraz dla funkcji uzyskanej z sieci neuronowej i porównaj wyniki (oczywiście powinny być podobne, jeśli wszystko działa dobrze).

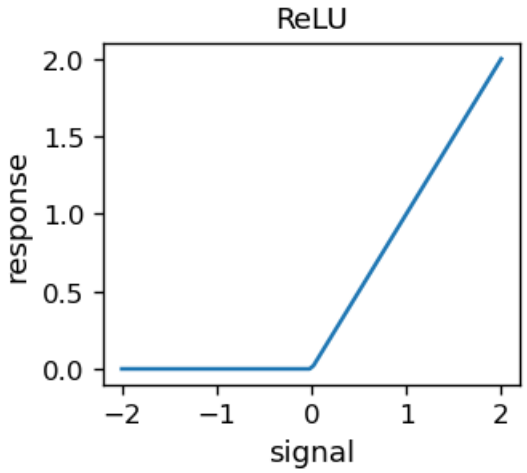
ROZDZIAŁ 8

Rektyfikacja

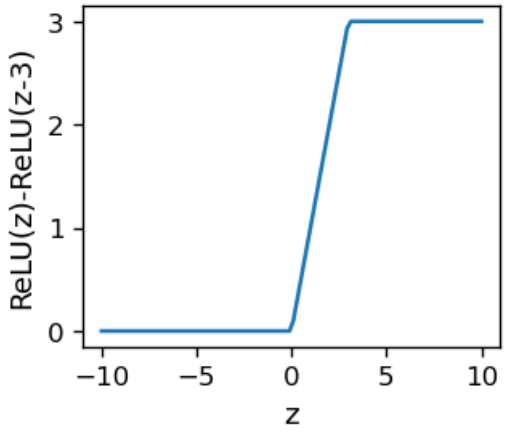
W poprzednim rozdziale utworzyliśmy z dwóch sigmoidów funkcję o kształcie garbu, która, jak pokazaliśmy, mogła stanowić funkcję bazową dla aproksymacji. Możemy teraz zadać nastepujące pytanie: czy możemy skonstruować sam sigmoid jako kombinację liniową (różnicę) pewych innych funkcji? Wtedy moglibyśmy użyć tychże funkcji do aktywacji neuronów zamiast sigmoidu. Odpowiedź brzmi **tak**. Na przykład funkcja **Rectified Linear Unit (ReLU)**

$$\mathrm{ReLU}(x) = \begin{cases} x & \text{for } x \geq 0 \\ 0 & \text{for } x < 0 \end{cases} = \max(x, 0)$$

wykonuje (w przybliżeniu) to zadanie. Ta nieco niezręczna nazwa pochodzi z elektrotechniki (rektyfikacja oznacza prostowanie), w której prostownik służy do odcinania ujemnych wartości sygnału elektrycznego. Wykres funkcji ReLU wygląda następująco:



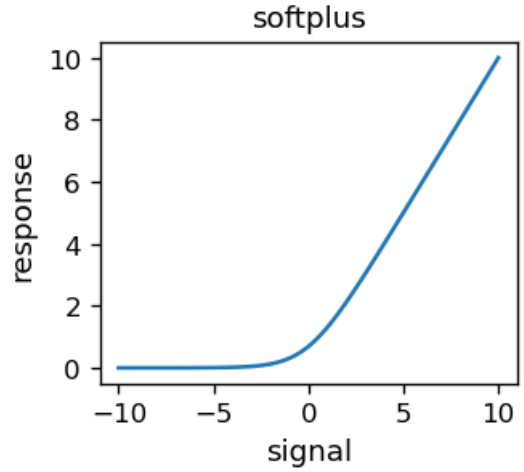
Różnica dwóch funkcji ReLU o przesuniętych argumentach daje przykładowy wynik



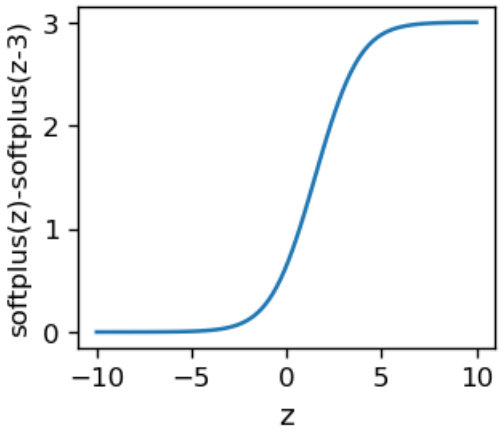
która jakościowo wygląda jak sigmoid, z wyjątkiem ostrych rogów. Aby uzyskać gładkość, można skorzystać z innej funkcji - **softplus**,

$$\text{softplus}(x) = \log\left(1 + e^{x}\right),$$

która ma nastepujący wykres:



Różnica dwóch funkcji **softplus** o przesuniętym argumencie daje wynik bardzo podobny do sigmoidu:



Informacja: Do aktywacji można użyć ReLU, softplus lub wielu innych podobnych funkcji.

Dlaczego właściwie należy to robić, zostanie omówione później.

8.1 Interpolacja z ReLU

Nasze symulowane dane możemy aproksymować za pomocą sieci ANN z aktywacją ReLU w warstwach pośrednich (w warstwie wyjściowej funcja aktywacji jest identycznościowa, tak jak w poprzednim rozdziale). W poniższych kodach funkcje zostały zaczerpnięte z modułu **func**.

```
#fff=func.softplus    # short-hand notation
#dfff=func.dsoftplus

fff=func.relu    # short-hand notation
dfff=func.drelu
```

Sieć musi mieć teraz więcej neuronów, ponieważ sigmoid „rozpada się” na dwie funkcje ReLU:

```
arch=[1,30,1]                    # architecture
weights=func.set_ran_w(arch, 5) # initialize weights randomly in [-2.5,2.5]
```

Symulacje przeprowadzamy dokładnie tak samo jak w poprzednim przypadku. Doświadczenie mówi, że należy startować z małymi szybkościami uczenia się. Dwa zestawy rund (podobnie jak w poprzednim rozdziale)

```
eps=0.0003         # small learning speed
for k in range(30): # rounds
    for p in range(len(features)):          # loop over the data sample points
        pp=np.random.randint(len(features)) # random point
        func.back_prop_o(features,labels,pp,arch,weights,eps,
                         f=fff,df=dfff,fo=func.lin,dfo=func.dlin) # teaching
```

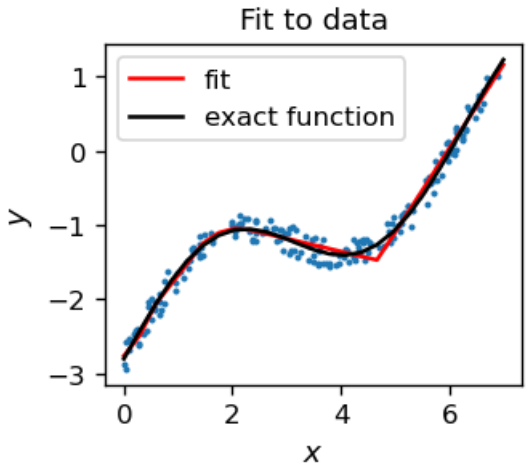
```
for k in range(3000): # rounds
#    eps=eps*.995
    if k%100==99: print(k+1,' ',end='')             # print progress
    for p in range(len(features)): # points in sequence
        func.back_prop_o(features,labels,p,arch,weights,eps,
                         f=fff,df=dfff,fo=func.lin,dfo=func.dlin) # teaching
```

```
100  200  300  400  500  600  700  800  900  1000  1100  1200  1300  1400  1500␣
↪ 1600  1700  1800  1900  2000  2100  2200  2300  2400  2500  2600  2700  2800␣
↪ 2900  3000
```

```
for k in range(3000): # rounds
    eps=eps*.995
    if k%100==99: print(k+1,' ',end='')             # print progress
    for p in range(len(features)): # points in sequence
        func.back_prop_o(features,labels,p,arch,weights,eps,
                         f=fff,df=dfff,fo=func.lin,dfo=func.dlin) # teaching
```

```
100  200  300  400  500  600  700  800  900  1000  1100  1200  1300  1400  1500␣
↪ 1600  1700  1800  1900  2000  2100  2200  2300  2400  2500  2600  2700  2800␣
↪ 2900  3000
```

dają wynik



Ponownie uzyskujemy całkiem zadowalający wynik (linia czerwona), zauważając, że wykres funkcji dopasowania jest ciągiem linii prostych, co odzwierciedla właściwości użytej funkcji aktywacji ReLU. Gładki wynik można uzyskać z pomocą funkcji softplus.

8.2 Klasyfikatory z rektyfikacją

Istnieją techniczne powody przemawiające za stosowaniem w algorytmie backprop funkcji rektyfikowanych zamiast sigmoidalnych. Pochodne sigmoidu są bowiem bardzo bliskie zera, z wyjątkiem wąskiego obszaru w pobliżu progu. Sprawia to, że aktualizacja wag jest mało prawdopodobna, zwłaszcza gdy cofamy się o wiele warstw wstecz, ponieważ wtedy bardzo małe liczby (określone przez pochodne funkcji) są przemnażane i w zasadzie nie prowadzą do żadnej aktualizacji (zjawsko to jest znane jako **problem zanikającego gradientu**). W przypadku funkcji rektyfikowanych zakres, w którym pochodna jest istotnie różna od zera, jest duży (w przypadku ReLU dotyczy to wszystkich współrzędnych dodatnich), dlatego problem zanikającego gradientu się nie pojawia. Właśnie z tego powodu funkcje rektyfikowane są stosowane w głębokich sieciach ANN, w których jest wiele warstw, niemożliwych do wytrenowania przy funkcjach aktywacji jest typu sigmoidalnego.

Informacja: Zastosowanie rektyfikowanych funkcji aktywacji było jednym z kluczowych trików, które umożliwiły przełom w rozwoju głębokich ANN około 2011 roku.

Z drugiej strony, w przypadku ReLU może się zdarzyć, że niektóre wagi przyjmą takie wartości, że wiele neuronów stanie się nieaktywnych, tzn. nigdy, dla żadnego inputu, nie zadziałają – zostaną de facto wyeliminowane. Nazywa się to problemem „martwego neuronu" lub „trupa", który pojawia się zwłaszcza wtedy, gdy parametr szybkości uczenia jest zbyt duży. Sposobem na ograniczenie tego problemu jest zastosowanie funkcji aktywacji, która w ogóle nie ma przedziału o zerowej pochodnej, np. Leaky ReLU. Tutaj przyjmiemy jej nasepującą postać

$$\text{Leaky ReLU}(x) = \begin{cases} x & \text{for } x \geq 0 \\ 0.1\,x & \text{for } x < 0 \end{cases}.$$

Dla ilustracji powtórzymy nasz przykład z rozdz. *Przykład z kołem* z klasyfikacją punktów w kole z wykorzystaniem funkcji Leaky ReLU.

Przyjmujemy następującą architekturę i parametry początkowe:

```
arch_c=[2,20,1]                    # architecture
weights=func.set_ran_w(arch_c,3)   # scaled random initial weights in [-1.5,1.5]
eps=.01                            # initial learning speed
```

i uruchamiamy algorytm w dwóch etapach: z Leaky ReLU, a następnie z ReLU:

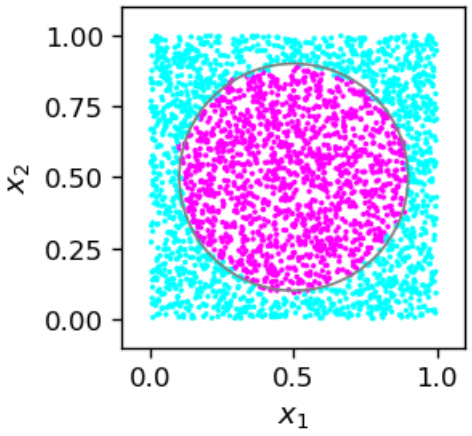
```
for k in range(300):     # rounds
    eps=.999*eps        # decrease the learning speed
    if k%100==99: print(k+1,' ',end='')              # print progress
    for p in range(len(features_c)):                 # loop over points
        func.back_prop_o(features_c,labels_c,p,arch_c,weights,eps,
            f=func.lrelu,df=func.dlrelu,fo=func.sig,dfo=func.dsig)
                    # backprop with leaky ReLU
```

```
100  200  300
```

```
for k in range(700):     # rounds
    eps=.995*eps        # decrease the learning speed
    if k%100==99: print(k+1,' ',end='')              # print progress
    for p in range(len(features_c)):                 # loop over points
        func.back_prop_o(features_c,labels_c,p,arch_c,weights,eps,
            f=func.relu,df=func.drelu,fo=func.sig,dfo=func.dsig)
                    # backprop with ReLU
```

```
100  200  300  400  500  600  700
```

Wynik jest całkiem zadowalający, co pokazuje, że metoda działa. Przy obecnej architekturze i funkcjach aktywacji, co nie jest zaskakujące, na poniższym wykresie można zauważyć ślady wielokąta przybliżającego koło.



8.3 Ćwiczenia

1. Zastosuj różne rektywikowane funkcje aktywacji dla klasyfikatorów binarnych i przetestuj je na różnych kształtach (analogicznie do przykładu z kołem powyżej).
2. Przekonaj się, że uruchomienie algorytmu backprop (z funkcją ReLU) ze zbyt dużą początkową szybkością uczenia prowadzi do problemu „martwego neuronu” i niepowodzenia algorytmu.

ROZDZIAŁ 9

Uczenie nienadzorowane

Motto

teachers! leave those kids alone!

(Pink Floyd, Another Brick In The Wall)

Uczenie nadzorowane, omawiane w poprzednich wykładach, wymaga nauczyciela lub próbki treningowej z etykietami, gdzie znamy **a priori** cechy danych (np. jak w jednym z naszych przykładów, czy dany punkt jest wewnątrz czy na zewnątrz okręgu). Jest to jednak dość szczególna sytuacja, ponieważ najczęściej dane, z którymi się stykamy, nie mają przypisanych etykiet i „są, jakie są". Ponadto, z neurobiologicznego czy metodologicznego punktu widzenia, wielu faktów i czynności uczymy się „na bieżąco", klasyfikując je, a następnie rozpoznając, przy czym proces ten przebiega bez żadnego zewnętrznego nadzoru czy etykietek „unoszących się" w powietrzu nad obiektami.

Wyobraźmy sobie botanika-kosmitę, który wchodzi na łąkę i napotyka różne gatunki kwiatów. Nie ma zielonego pojęcia, czym one są i czego się spodziewać, ponieważ nie ma żadnej wiedzy o sprawach ziemskich. Po znalezieniu pierwszego kwiatu zapisuje jego cechy: kolor, wielkość, liczbę płatków, zapach itd. Idzie dalej, znajduje inny kwiat, zapisuje jego cechy, i tak dalej i dalej dla kolejnych kwiatami. W pewnym momencie trafia jednak na kwiat, który już wcześniej poznał. Dokładniej mówiąc, jego cechy są bardzo zbliżone, choć nie identyczne (wielkość może się nieco różnić, kolor itd.), do poprzedniego przypadku. Stąd wniosek, że należy on do jednej kategorii. Poszukiwania trwają dalej, a nowe kwiaty albo tworzą nową kategorię, albo dołączają do już istniejącej. Na koniec poszukiwań kosmina ma utworzony katalog kwiatów i może przypisać nazwy (etykiety) poszczególnym gatunkom: mak kukurydziany, krwawnik pospolity, dziewanna… Etykiety te, czyli nazwy, są przydatne w dzieleniu się swoją wiedzą z innymi, ponieważ podsumowują cechy kwiatu. Należy jednak pamiętać, że etykiety te nigdy nie były używane w procesie eksploracji (uczenia się) łąki.

Formalnie rzecz biorąc, opisywany problem **uczenia nienadzorowanego** dotyczy klasyfikacji danych (podziału na kategorie, lub **klastry**, czyli podzbiory próbki danych, w których odpowiednio zdefiniowane odległości między poszczególnymi danymi są małe, mniejsze od przyjętych odległości między klastrami). Mówiąc kolokwialnie, szukamy podobieństw między poszczególnymi punktami danych i staramy się podzielić próbkę na grupy podobnych obiektów.

9.1 Klastry

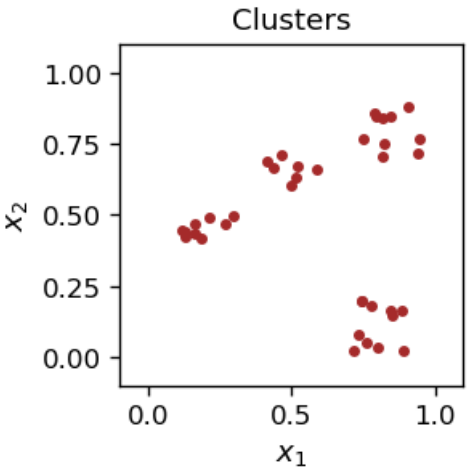
Oto nasza uproszczona wersja eksploracji botanika-kosmity: Rozważmy punkty na płaszczyźnie, które są generowane losowo. Ich rozkład nie jest jednorodny, lecz rozłożony w czterech skupiskach: A, B, C i D. Możemy na przykład zadać odpowiednie granice dla współrzędnych x_1 i x_2 przy losowym generowaniu punktów danej kategorii. Używamy do tego funkcji numpy **random.uniform(a,b)**, dającej równomiernie rozłożoną liczbę zmiennoprzecinkową pomiędzy a i b:

```
def pA():
    return [np.random.uniform(.75, .95), np.random.uniform(.7, .9)]

def pB():
    return [np.random.uniform(.4, .6), np.random.uniform(.6, .75)]

def pC():
    return [np.random.uniform(.1, .3), np.random.uniform(.4, .5)]

def pD():
    return [np.random.uniform(.7, .9), np.random.uniform(0, .2)]
```

Utwórzmy próbkę danych zawierającą po kilka punktów z każdej kategorii:

```
samA=np.array([pA() for _ in range(10)])
samB=np.array([pB() for _ in range(7)])
samC=np.array([pC() for _ in range(9)])
samD=np.array([pD() for _ in range(11)])
```

Dane te wyglądają następująco:



Jeśli pokażemy komuś powyższy rysunek, to z pewnością stwierdzi, że są na nim cztery klastry. Ale jaka metoda jest używany, aby to stwierdzić? Wkrótce skonstruujemy odpowiedni algorytm i będziemy mogli przeprowadzić klasteryzację. Na razie jednak skoczmy w przód i załóżmy, że **wiemy**, czym są klastry. W naszym przykładzie klastry są dobrze zdefiniowane, tzn. widocznie oddzielone od siebie.

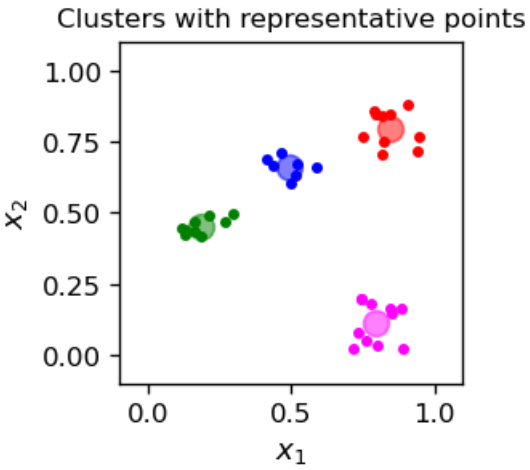
Można reprezentować klastry za pomocą **punktów reprezentatywnych**, które leżą gdzieś w obrębie klastra. Można na przykład wziąć element należący do danego klastra jako jego reprezentanta, lub też dla każdego klastra oszacować średnie położenie jego punktów i użyć je jako punkt reprezentatywny:

```
rA=[st.mean(samA[:,0]),st.mean(samA[:,1])]
rB=[st.mean(samB[:,0]),st.mean(samB[:,1])]
rC=[st.mean(samC[:,0]),st.mean(samC[:,1])]
rD=[st.mean(samD[:,0]),st.mean(samD[:,1])]
```

(do obliczenia średniej użyliśmy modułu **statistics**). Tak zdefiniowane punkty reprezentatywne dołączamy do powyższej grafiki. Dla wygody wizualnej, każdej kategorii przypisujemy kolor (po ustaleniu klastów możemy bowiem nadać im etykiety, a kolor w tym przypadku służy właśnie temu celowi).

```
col=['red','blue','green','magenta']
```

```
plt.figure(figsize=(2.3,2.3),dpi=120)
plt.title("Clusters with representative points",fontsize=10)
plt.xlim(-.1,1.1)
plt.ylim(-.1,1.1)

plt.scatter(samA[:,0],samA[:,1],c=col[0], s=10)
plt.scatter(samB[:,0],samB[:,1],c=col[1], s=10)
plt.scatter(samC[:,0],samC[:,1],c=col[2], s=10)
plt.scatter(samD[:,0],samD[:,1],c=col[3], s=10)

plt.scatter(rA[0],rA[1],c=col[0], s=90, alpha=0.5)
plt.scatter(rB[0],rB[1],c=col[1], s=90, alpha=0.5)
plt.scatter(rC[0],rC[1],c=col[2], s=90, alpha=0.5)
plt.scatter(rD[0],rD[1],c=col[3], s=90, alpha=0.5)

plt.xlabel('$x_1$',fontsize=11)
plt.ylabel('$x_2$',fontsize=11)
plt.show()
```



9.2 Komórki Woronoja

W sytuacji jak na rysunku powyżej, tzn. mając wyznaczone punkty reprezentatywne, możemy podzielić całą płaszczyznę na komórki (obszary) według następującego kryterium Woronoja, które jest prostym pojęciem geometrycznym:

Komórki Woronoja

Rozważmy przestrzeń metryczną, w której istnieje pewna liczba punktów reprezentatywnych (punktów Woronoja) R. Dla danego punktu P wyznaczamy odległości do wszystkich punktów R. Jeśli wśród tych odległości istnieje ścisłe minimum (najbliższy punkt R_m), to z definicji punkt P należy do komórki Woronoja R_m. Jeśli nie ma ścisłego minimum, to P należy do granicy między pewnymi komórkami. Konstrukcja ta dzieli całą przestrzeń na kopmórki Woronoja i granice pomiędzy nimi.

Wracając do naszego przykładu, zdefiniujmy kolor punktu P jako kolor najbliższego punktu reprezentatywnego. W

tym celu potrzebujemy (kwadratu) odległości (tutaj euklidesowej) między dwoma punktami w przestrzeni dwuwymiarowej:

```
def eucl(p1,p2): # square of the Euclidean distance
    return (p1[0]-p2[0])**2+(p1[1]-p2[1])**2
```

Następnie z pomocą **np.argmin** znajdujemy najbliższy reprezentatywny punkt i określamy jego kolor:

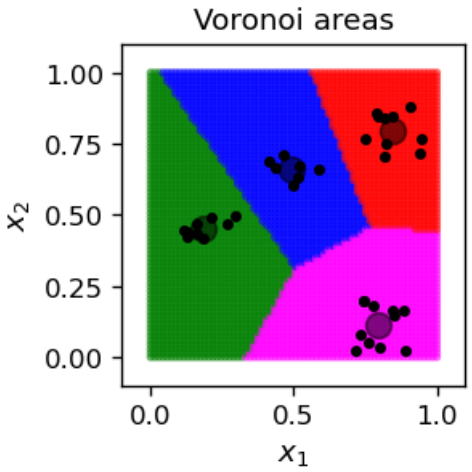
```
def col_char(p):
    dist=[eucl(p,rA),eucl(p,rB),eucl(p,rC),eucl(p,rD)] # array of distances
    ind_min = np.argmin(dist)                          # index of the nearest␣
↪point
    return col[ind_min]                                # color of the nearest␣
↪point
```

Na przykład

```
col_char([.5,.5])
```

```
'blue'
```

Wynikiem przeprowadzenia tego kolorowania dla punktów z naszej przestrzeni (bierzemy tu wystarczająco gęstą próbkę 70×70 punktów) jest jej następujący podział na komórki Woronoja:



Łatwo jest udowodnić, że granice między sąsiednimi obszarami są liniami prostymi.

Informacja: Praktyczne przesłanie jest takie, że po wyznaczeniu punktów reprezentatywnych możemy zastosować kryterium Woronoja do klasyfikacji danych.

9.3 Naiwna klasteryzacja

Wracamy teraz do problemu botanika-kosmity: wyobraźmy sobie, że mamy naszą próbkę, ale nie wiemy nic o tym, jak zostały wygenerowane jej punkty (nie mamy etykiet A, B, C, D ani kolorów punktów). Co więcej, dane są zmieszane, tzn. punkty danych występują w przypadkowej kolejności. Łączymy więc nasze punkty za pomocą **np.concatenate**:

```
alls=np.concatenate((samA, samB, samC, samD))
```

i tasujemy je z pomocą **np.random.shuffle**:

```
np.random.shuffle(alls)
```

Wizualizacja danych wygląda tak, jak na pierwszym wykresie w tym rozdziale.

Chcemy teraz w jakiś sposób utworzyć punkty reprezentatywne, ale a priori nie wiemy, gdzie powinny się one znajdować, ani nawet ile ich powinno być. Do osiągnięcia celu możliwe są bardzo różne strategie. Ich wspólną cechą jest to, że położenie punktów reprezentatywnych jest aktualizowane w miarę przetwarzania (czytania) danych próbki.

Zacznijmy od przypadku tylko jednego punktu reprezentatywnego, $\vec{R}$. Nie jest to zbyt ambitne, ale przynajmniej będziemy znali pewne uśrednione charakterystyki próbki. Początkowa pozycja to $R = (R_1, R_2)$, dwuwymiarowy wektor w przestrzeni $[0,1] \times [0,1]$. Po odczytaniu punktu danych P o współrzędnych (x_1^P, x_2^P), wektor R zmienia się w następujący sposób:

$$(R_1, R_2) \to (R_1, R_2) + \varepsilon(x_1^P - R_1, x_2^P - R_2),$$

lub w notacji wektorowej

$$\vec{R} \to \vec{R} + \varepsilon(\vec{x}^P - \vec{R}).$$

Czynność tę powtarzamy dla wszystkich punktów próbki, a następnie można wykonać wiele rund. Podobnie jak w poprzednich rozdziałach, ε jest współczynnikiem uczenia, który maleje wraz z postępem algorytmu. Powyższy wzór realizuje „przyciąganie" punktu $\vec{R}$ przez punkt danych $vecP$.

Poniższy kod implementuje przepis:

```
R=np.array([np.random.random(),np.random.random()]) # initial location

print("initial location:")
print(np.round(R,3))
print("round   location")

eps=.5                          # initial learning speed

for j in range(50):             # rounds
    eps=0.85*eps                # decrease the learning speed
    np.random.shuffle(alls)     # reshuffle the sample
    for i in range(len(alls)): # loop over points of the whole sample
        R+=eps*(alls[i]-R)      # update/learning
    if j%5==4: print(j+1, "    ",np.round(R,3))  # print every 5th step
```

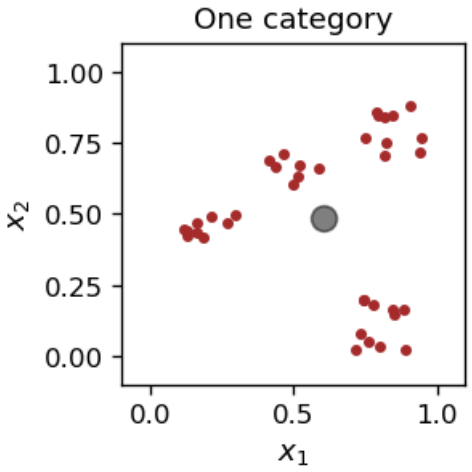
```
initial location:
[0.458 0.603]
round   location
5      [0.447 0.537]
10      [0.457 0.47 ]
15      [0.604 0.453]
20      [0.607 0.478]
25      [0.603 0.484]
30      [0.602 0.484]
35      [0.602 0.484]
40      [0.602 0.484]
45      [0.602 0.485]
50      [0.602 0.485]
```

Można zauważyć, że położenie punktu reprezentatywnego jest zbieżne do pewnej granicy. W rzeczywistości dąży ono do średniego położenia punktów próbki,

```
R_mean=[st.mean(alls[:,0]),st.mean(alls[:,1])]
print(np.round(R_mean,3))
```

```
[0.602 0.485]
```

Zdecydowaliśmy a priori, że będziemy mieli tylko jedną kategorię. Oto więc wykres z wynikiem dla punktu reprezentatywnenego, oznaczonego szarą plamą:



Powyższe rozwiązanie oczywiście nas nie zadowala (na oko, obiekty nie należą do jednej kategorii), spróbujmy więc uogólnić algorytm na przypadek kilku ($n_R > 1$) punktów reprezentatywnych:

- Inicjalizujemy losowo punkty reprezentatywne $\vec{R}^i$, $i = 1, \dots, n_R$.
- Runda: Bierzemy po kolei przykładowe punkty P i aktualizujemy tylko **najbliższy** punkt reprezentatywny R^m do punktu P w danym kroku:

$$\vec{R}^m \to \vec{R}^m + \varepsilon(\vec{x} - \vec{R}^m).$$

- Położenie pozostałych punktów reprezentatywnych pozostaje niezmienione. Strategia taka nazywana jest **zwycięzca bierze wszystko** (winner-take-all).
- Powtarzamy rundy, za każdym razem zmniejszając ε.

Ważne: Strategia **zwycięzca bierze wszystko** jest ważnym pojęciem w trenowaniu ANN. Konkurujące neurony w warstwie walczą o „nagrodę", a ten, który wygra, bierze ją w całości (jego wagi są aktualizowane), podczas gdy przegrani nie dostają nic.

Rozważmy teraz dwa punkty reprezentatywne, które inicjalizujemy losowo:

```
R1=np.array([np.random.random(), np.random.random()])
R2=np.array([np.random.random(), np.random.random()])
```

Następnie wykonujemy nasz algorytm. Dla każdego punktu danych znajdujemy najbliższy reprezentatywny punkt spośród dwóch i aktualizujemy tylko ten, który jest zwycięzcą:

```
print("initial locations:")
print(np.round(R1,3), np.round(R2,3))
print("rounds  locations")

eps=.5

for j in range(40):
    eps=0.85*eps
    np.random.shuffle(alls)
    for i in range(len(alls)):
        p=alls[i]
```

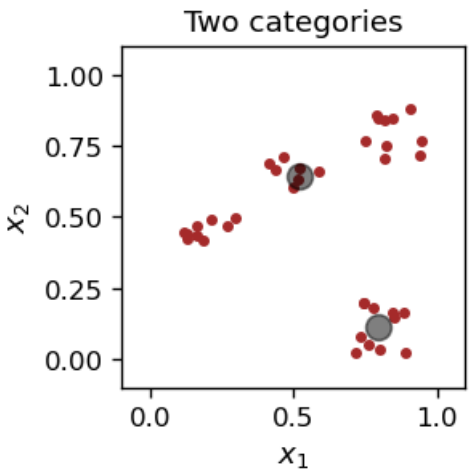
(ciąg dalszy na następnej stronie)

(kontynuacja poprzedniej strony)

```
        dist=[func.eucl(p,R1), func.eucl(p,R2)] # squares of distances
        ind_min = np.argmin(dist)               # index of the minimum
        if ind_min==0:          # if R1 closer to the new data point
            R1+=eps*(p-R1)      # update R1
        else:                   # if R2 closer ...
            R2+=eps*(p-R2)      # update R2

    if j%5==4: print(j+1,"    ", np.round(R1,3), np.round(R2,3))
```

```
initial locations:
[0.887 0.077] [0.916 0.311]
rounds  locations
5       [0.793 0.139] [0.475 0.636]
10       [0.795 0.117] [0.468 0.614]
15       [0.8   0.112] [0.528 0.647]
20       [0.796 0.113] [0.525 0.643]
25       [0.796 0.114] [0.521 0.642]
30       [0.796 0.114] [0.52  0.642]
35       [0.796 0.114] [0.52  0.642]
40       [0.796 0.114] [0.52  0.642]
```

Wynik jest następujący:



Jeden z punktów charakterystycznych „specjalizuje się” w prawym dolnym klastrze, a drugi w pozostałych punktach próbki.

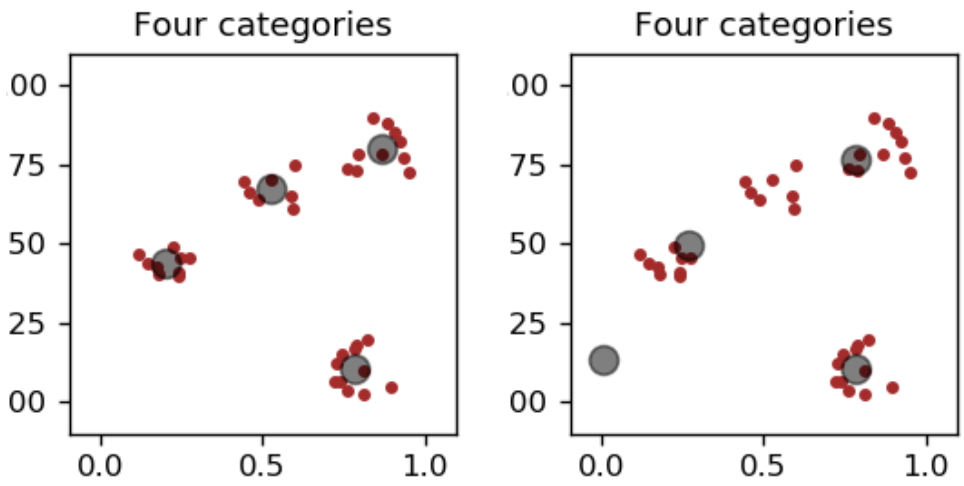
Następnie kontynuujemy, całkiem analogicznie, z czterema punktami reprezentatywnymi.

Wynik dla dwóch różnych warunków początkowych dla punktów reprezentatywnych (i nieco innej próbki) jest pokazany na Rys. 9.1.

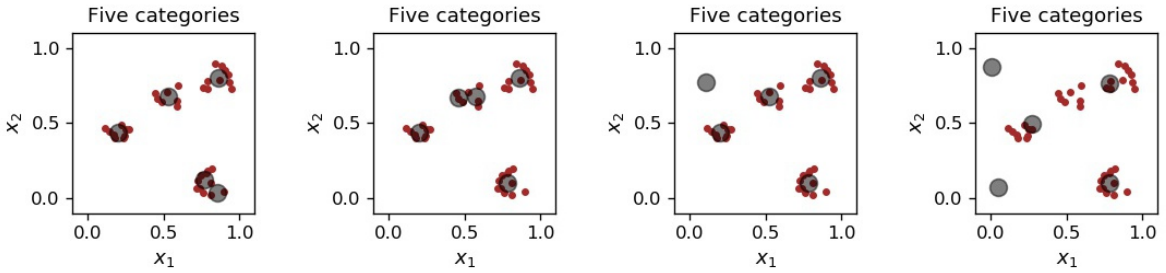
Zauważamy, że procedura nie zawsze daje poprawną/oczekiwaną odpowiedź. Dość często jeden z punktów reprezentatywnych nie jest w ogóle aktualizowany i staje się tak zwanym **trupem**. Dzieje się tak dlatego, że pozostałe punkty reprezentatywne zawsze wygrywają, tzn. jeden z nich jest zawsze bliżej każdego punktu danych w próbce niż „trup”. Oczywiście, jest to sytuacja niezadowalająca.

Gdy bierzemy pięć punktów reprezentatywnych, to w zależności od losowej inicjalizacji może wystąpić kilka sytuacji, jak pokazano na Rys. 9.2. Czasami jakoś klaster rozpada się na dwa mniejsze, czasami pojawiają się trupy.

Branie większej liczby punktów reprezentacyjnych prowadzi do jeszcze częstszego powstawania trupów. Oczywiście możemy je lekceważyć, ale przykład ten pokazuje, że obecna strategia klastrowanioa danych jest wysoce problematyczna i potrzebujemy czegoś lepszego.



Rys. 9.1: Po lewej: właściwe punkty reprezentatywne. Po prawej: jeden „trup”.



Rys. 9.2: Od lewej do prawej: 5 punktów charakterystycznych z jednym wcześniejszym klastrem podzielonym na dwa, z innym klasterm podzielonym na dwa, jednym trupem i dwoma trupami.

9.4 Skala klastrowania

W poprzednim rozdziale staraliśmy się od początku odgadnąć, ile klastrów znajduje się w danych. Prowadziło to do problemów, gdyż zazwyczaj nie wiemy nawet, ile jest klastrów. Właściwie do tej pory nie zdefiniowaliśmy, czym dokładnie jest klaster, i posługiwaliśmy się jedynie intuicją. Ta intuicja podpowiadała nam, że punkty w tym samym skupisku muszą być blisko siebie lub blisko punktu charakterystycznego, ale jak blisko? Tak naprawdę definicja musi zawierać **skalę** (charakterystyczną odległość), która mówi nam, „jak blisko jest blisko". Na przykład w naszym przykładzie możemy przyjąć skalę około 0,2, gdzie są 4 klastry, ale możemy też przyjąć mniejszą skalę i rozdzielić większe klastry na mniejsze, jak w dwóch lewych panelach Rys. 9.2.

Definicja klastra

Klaster o skali d związany z punktem charakterystycznym R to zbiór punktów danych P, których odległość od R jest mniejsza niż d, natomiast odległość od innych punktów reprezentatywnych jest $\geq d$. Punkty charakterystyczne muszą być wybrane w taki sposób, aby każdy punkt danych należał do klastra, a żaden punkt charakterystyczny nie był martwy (tzn. jego klaster musi zawierać co najmniej jeden punkt danych).

Do realizacji tej recepty można wykorzystać różne strategie. Tutaj użyjemy **dynamicznej klasteryzacji**, w której nowy klaster/punkt reprezentatywny jest tworzony za każdym razem, gdy napotkany punkt danych znajduje się dalej niż d od dowolnego punktu reprezentatywnego zdefiniowanego do tej pory.

Dynamiczna klasteryzacja

1. Ustaw skalę klasteryzacji d i początkową szybkość uczenia $varepsilon$. Potasuj próbkę.
2. Odczytaj pierwszy punkt danych P_1 i wyznacz pierwszy punkt charakterystyczny jako $R^1 = P_1$. Dodaj go do tablicy R zawierającej wszystkie punkty charakterystyczne. Oznacz P_1 jako należący do klastra 1.
3. Odczytaj kolejny punkt danych P. Jeśli odległość P od **najbliższego** punktu charakterystycznego, R^m, jest $\leq d$, to
   - oznacz P jako należący do klastra m.
   - przesuń R^m w kierunku P z prędkością uczenia $varepsilon$.W przeciwnym razie dodaj do R nowy punkt charakterystyczny w położeniu punktu P.
4. Powtórz od 2, aż wszystkie punkty danych zostaną przetworzone.
5. Powtórz od 2 w pewnej liczbie rund, zmniejszając za każdym razem ε. Wynikiem jest podział próbki na pewną liczbę klastrów oraz położenia odpowiadających im punktów reprezentatywnych. Wynik może zależeć od losowego tasowania, a więc nie musi być taki sam przy powtarzaniu procedury.

Poniżej przedstawiono implementację w języku Python, dynamiczne znajdującą punkty reprezentatywne:

```
d=0.2   # clustering scale
eps=0.5 # initial learning speed

for r in range(20):                # rounds
    eps=0.85*eps                   # decrease the learning speed
    np.random.shuffle(alls)        # shuffle the sample
    if r==0:                       # in the first round
        R=np.array([alls[0]])      # R - array of representative points
                                   # initialized to the first data point
    for i in range(len(alls)):     # loop over the sample points
        p=alls[i]                  # new data point
        dist=[func.eucl(p,R[k]) for k in range(len(R))]
         # array of squares of distances of p from the current repr. points in R
        ind_min = np.argmin(dist) # index of the closest repr. point
        if dist[ind_min] > d*d:    # if its distance square > d*d
                                   # dynamical creation of a new category
```

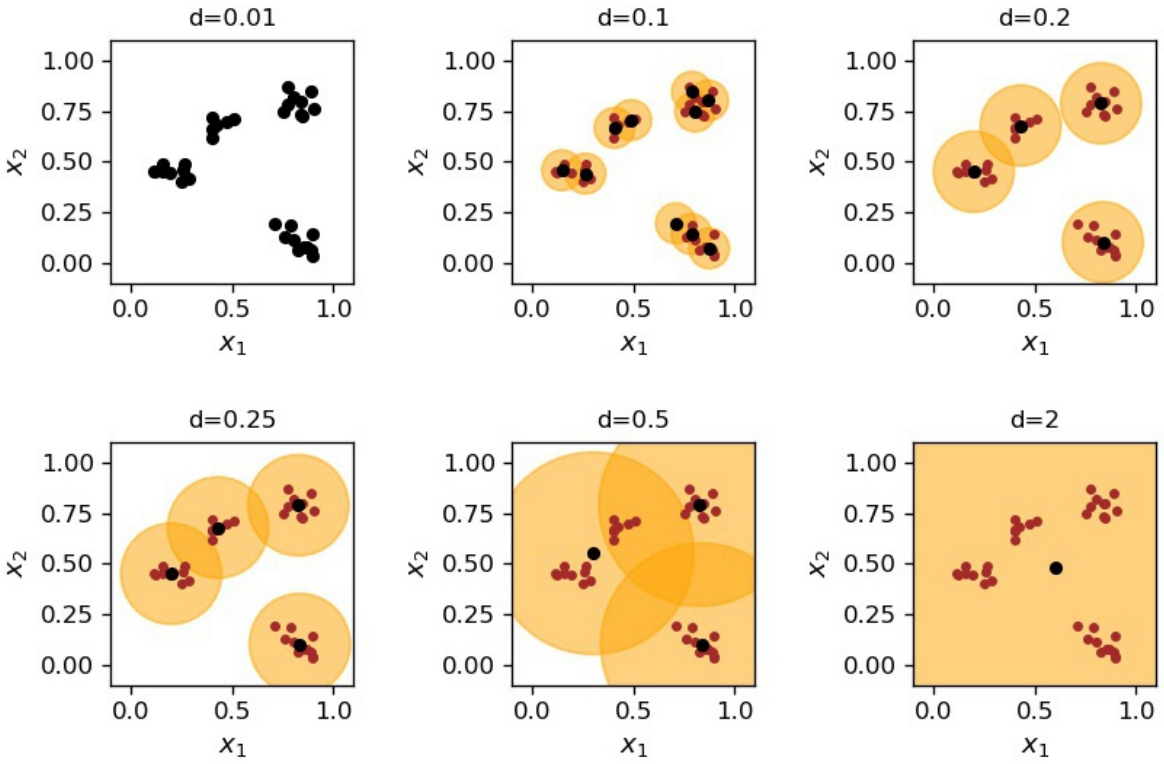
(ciąg dalszy na następnej stronie)

(kontynuacja poprzedniej strony)

```
            R=np.append(R, [p], axis=0)     # add new repr. point to R
        else:
            R[ind_min]+=eps*(p-R[ind_min]) # otherwise, apdate the "old" repr.␣
↪point

print("Number of representative points: ",len(R))
```

```
Number of representative points:  4
```

Wynik działania algorytmu dla różnych wartości skali klasteryzacji d pokazano na Rys. 9.3. Przy bardzo małych wartościach d, mniejszych od minimalnej separacji między punktami, klastrów jest tyle, ile punktów danych. Następnie, wraz ze wzrostem wartości d, liczba klasrów maleje. Przy bardzo dużych wartościach d, rzędu rozpiętości całej próbki, występuje tylko jeden klaster.



Rys. 9.3: Dynamiczne klastrowanie dla różnych wartości skali d.

Oczywiście sam algorytm nie powie nam, jaką skalę klasteryzacji należy zastosować. Właściwa wartość zależy od natury problemu. Przypomnijmy sobie naszego botanika-kosmitę. Gdyby użył bardzo małej wartości d, otrzymałby tyle kategorii, ile jest kwiatów na łące, ponieważ wszystkie kwiaty, nawet tego samego gatunku, różnią się od siebie nieznacznie. Byłoby to bezużyteczne. Z drugiej strony, jeśli d jest zbyt duże, to klasyfikacja jest zbyt zgrubna. Coś pomiędzy jest w sam raz!

Labels

Po utworzeniu klastrów możemy dla wygody nadać im **etykiety**. Nie są one wykorzystywane w procesie uczenia (tworzenia klastrów).

Po określeniu klastrów mamy **klasyfikator**. Możemy go używać w dwojaki sposób:

- kontynuować dynamiczną aktualizację w miarę napływu nowych danych lub

- „zamknąć” klasyfikator i sprawdzić, gdzie wpadają nowe dane.

W pierwszym przypadku przyporządkowujemy nowemu punktowi danych odpowiednią etykietę klastra (nasz botanik wie, jaki nowy kwiat znalazł) lub tworzymy nową kategorię, jeśli punkt nie należy do żadnego z istniejących klastrów. Jest to po prostu kontynuacja opisanego powyżej algorytmu dla nowych przychodzących danych.

W drugim przypadku (kupiliśmy gotowy i zamknięty katalog botanika-kosmity) punkt danych może

- należeć do klastra (znamy jego etykietę),
- nie należeć do żadnego klastra, wtedy nie wiemy, co to jest, lub
- znaleźć się w obszarze nakładania się dwóch lub więcej klastrów (por. Rys. 9.3, kiedy otrzymujemy tylko „częściową” lub niejednoznaczną klasyfikację.

Alternatywnie, możemy zastosować klasyfikację z pomocą komórek Woronoja, aby pozbyć się niejednoznaczności.

9.4.1 Interpretacja poprzez najstromszy spadek

Oznaczmy dany klaster symbolem $C_i, i = 1, ..., n$, gdzie n jest całkowitą liczbą klastrów. Suma kwadratów odległości punktów danych w klastrze C_i od jego punktu reprezentatywnego R^i wynosi

$$\sum_{P \in C_i} |\vec{R}^i - \vec{x}^P|^2.$$

Sumując po wszystkich klastrach, otrzymujemy funkcję analogiczną do omówionej wcześniej funkcji błędu:

$$E(\{R\}) = \sum_{i=1}^{n} \sum_{P \in C_i} |\vec{R}^i - \vec{x}^P|^2.$$

Jej pochodna po $\vec{R}_i$ wynosi

$$\frac{\partial E(\{R\})}{\partial \vec{R}^i} = 2 \sum_{P \in C_i} (\vec{R}^i - \vec{x}^P).$$

Metoda najstromszego spadku daje w rezultacie **dokładnie** taką samą receptę, jaką zastosowano w przedstawionym powyżej algorytmie dynamicznej klasteryzacji, tj.

$$\vec{R} \to \vec{R} - \varepsilon(\vec{R} - \vec{x}^P).$$

Podsumowując, zastosowany algorytm polega w istocie na zastosowaniu metody najstromszego zejścia dla funkcji $E(R)$, co zostało dokładniej omówione w poprzednich rozdziałach.

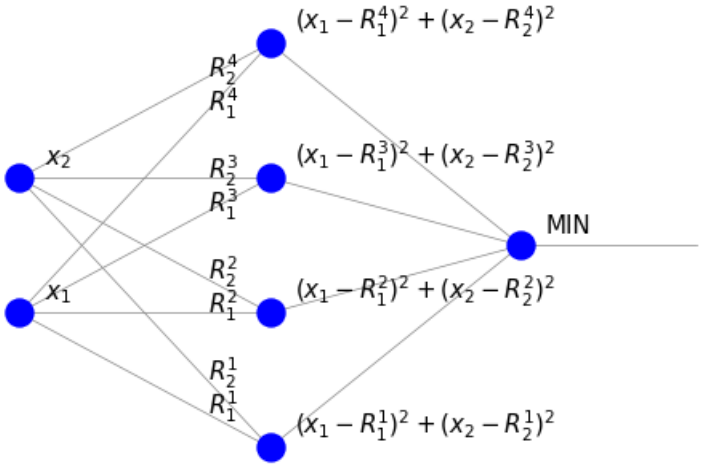
Informacja: Należy jednak zauważyć, że minimalizacja stosowana w obecnym algorytmie uwzględnia również różne kombinatoryczne podziały punktów na klastry. W szczególności, dany punkt danych może zmienić swoje przyporządkowanie do klastra w trakcie wykonywania algorytmu. Dzieje się tak, gdy zmienia się jego najbliższy punkt reprezentatywny.

ROZDZIAŁ 10

TUTAJ

10.1 Interpretacja jako ANN

Zinterpretujemy teraz zastosowany powyżej algorytm uczenia nienadzorowanego ze strategią „zwycięzca bierze wszystko” w języku sieci neuronowych. Weźmy następującą przykładową sieć:



Składa się ona z czterech neuronów w warstwie pośredniej, z których każdy odpowiada jednemu punktowi charakterystycznemu $\vec{R}^i$. Wagi są współrzędnymi punktu $\vec{R}^i$. W warstwie wyjściowej znajduje się jeden węzeł. Zauważamy istotne różnice w stosunku do omawianego wcześniej perceptronu.

- Nie ma węzłów progowych.
- W warstwie pośredniej sygnał jest równy kwadratowi odległości inputu od odpowiedniego punktu reprezentatywnego. Nie jest to suma ważona.
- Węzeł w ostatniej warstwie (MIN) wskazuje, w którym neuronie warstwy pośredniej sygnał jest najmniejszy, tzn. gdzie mamy najmniejszą odległość. Działa on zatem jako jednostka sterująca wybierająca minimum.

Podczas uczenia (nienadzorowanego) punkt wejściowy P „przyciąga»» najbliższy punkt charakterystyczny, którego wagi są aktualizowane w kierunku współrzędnych P.

Zastosowanie powyższej sieci klasyfikuje punkt o współrzędnych (x_1, x_2), przypisując mu wskaźnik najbliższego punktu reprezentatywnego dla danej kategorii (tutaj jest to etykieta 1, 2, 3 lub 4).

10.1.1 Reprezentacja z pomocą współrzędnych sferycznych

Nawet przy naszej ogromnej „swobodzie matematycznej" nazwanie powyższego systemu siecią neuronową byłoby sporym nadużyciem, ponieważ wydaje się on bardzo daleki od jakiegokolwiek wzorca neurobiologicznego. W szczególności, użycie (nieliniowego) sygnału w postaci $\left(\vec{R}^i - \vec{x}\right)^2$ kontrastuje z perceptronem, w którym sygnał wchodzący do neuronów jest (liniową) sumą ważoną inputów, tzn.

$$s^i = x_1 w_1^i + x_2 w_2^i + ... + w_1^m x_m = \vec{x} \cdot \vec{w}^i.$$

Możemy zmienić nasz problem, stosując prostą konstrukcję geometryczną tak, aby upodobnić go do zasady działania perceptronu. W tym celu wprowadzamy (fikcyjną, pomocniczą) trzecią współrzędną zdefiniowaną jako

$$x_3 = \sqrt{r^2 - x_1^2 - x_2^2},$$

gdie r jest dobrane tak, aby dla wszystkich punktów danych $r^2 \geq x_1^2 + x_2^2$. Z konstrukcji, $\vec{x} \cdot \vec{x} = x_1^2 + x_2^2 + x_3^2 = r^2$, więc punkty danych leżą na półkuli ($x_3 \geq 0$) o promieniu r. Podobnie, dla punktów reprezentatywnych wprowadzamy

$$w_1^i = R_1^i,\ w_2^i = R_2^i,\ w_3^i = \sqrt{r^2 - (R_1^i)^2 - (R_2^i)^2}.$$

Jest geometrycznie oczywiste, że dwa punkty na płaszczyźnie są sobie bliskie wtedy i tylko wtedy, gdy ich rozszerzenia ndo półkuli są sobie bliskie. Stwierdzenie to poprzemy prostym rachunkiem:

Iloczyn skalarny dwóch punktów $\vec{x}$ i $\vec{y}$ na półkuli można zapisać jako

$$\vec{x} \cdot \vec{y} = x_1 y_1 + x_2 y_2 + \sqrt{r^2 - x_1^2 - x_2^2}\sqrt{r^2 - y_1^2 - y_2^2}.$$

Dla uproszczenia rozważmy sytuację, w której $x_1^2 + x_2^2 \ll r^2$ and $y_1^2 + y_2^2 \ll r^2$, tj. obydwa punkty leżą w pobliżu bieguna półkuli. Korzystając z wiedzy z zakresu analizy matematycznej

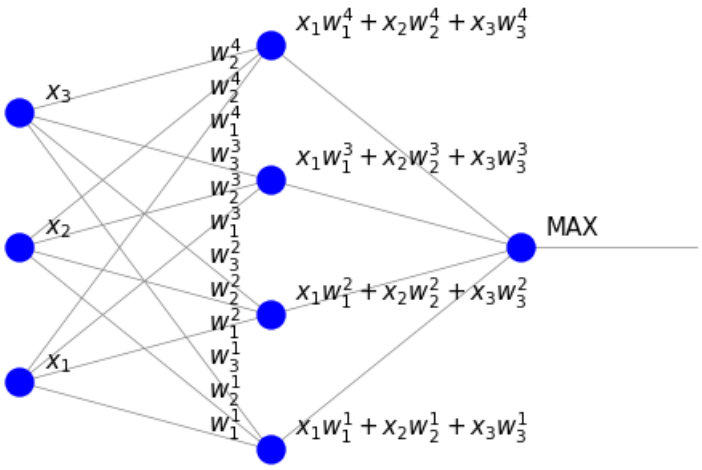
$$\sqrt{r^2 - a^2} \simeq r - \frac{a^2}{2r}, \quad a \ll r,$$

zatem

$$\begin{aligned}
\vec{x} \cdot \vec{y} &\simeq x_1 y_1 + x_2 y_2 + \left(r - \frac{x_1^2 + x_2^2}{2r}\right)\left(r - \frac{y_1^2 + y_2^2}{2r}\right) \\
&\simeq r^2 - \frac{1}{2}(x_1^2 + x_2^2 + y_1^2 + y_2^2) + x_1 y_1 + x_2 y_2 \\
&= r^2 - \frac{1}{2}[(x_1 - x_2)^2 + (y_1 - y_2)^2].
\end{aligned}$$

Iloczyn skalarny jest równy (dla punktów położonych blisko bieguna) stałej r^2 minus połowa kwadratu odległości między punktami (x_1, x_2) i (y_1, y_2) na płaszczyźnie! Wynika z tego, że zamiast znajdować minimalną odległość dla punktów na płaszczyźnie, jak w poprzednim algorytmie, możemy znaleźć maksymalny iloczyn skalarny dla ich 3-wymiarowych rozszerzeń do półkuli.

Po rozszerzeniu danych do półkuli, odpowiednią sieć neuronową można przedstawić w następujący sposób:



Dzięki naszym staraniom sygnał w warstwie pośredniej jest teraz po prostu iloczynem skalarnym inputu i wag, dokładnie tak jak powinno być w sztucznym neuronie. Neuron w ostatniej warstwie (MAX) wskazuje, gdzie iloczyn skalarny jest największy.

Interpretacja funkcji MAX w naszych obecnych ramach jest nadal nieco problematyczna. W rzeczywistości jest to możliwe, ale wymaga wyjścia poza sieci typu feed-forward. Gdy neurony w warstwie mogą się komunikować (sieci rekurencyjne, sieci Hopfielda), konkurować, a przy odpowiednim sprzężeniu zwrotnym można wymusić mechanizm „zwycięzca bierze wszystko". Aspekty te będą wspomniane w rozdz. {ref}`lat-lab»».

Reguła Hebba

Od strony koncepcyjnej dotykamy tutaj bardzo ważnej i intuicyjnej zasady w biologicznych sieciach neuronowych, znanej jako [reguła Hebba] (https://en.wikipedia.org/wiki/Hebbian_theory). Zasadniczo odnosi się ona do stwierdzenia „To, co jest używane, staje się silniejsze" w odniesieniu do połączeń synaptycznych. Wielokrotne użycie połączenia sprawia, że staje się ono silniejsze.

W naszym sformułowaniu, jeśli sygnał przechodzi przez dane połączenie, jego waga odpowiednio się zmienia, podczas gdy inne połączenia pozostają bez zmian. Proces ten odbywa się w sposób nienadzorowany, a jego realizacja jest dobrze umotywowana biologicznie.

Informacja: Z drugiej strony, trudno jest znaleźć biologiczne uzasadnienie dla uczenia nadzorowanego metodą backprop, w której wszystkie wagi są aktualizowane, także w warstwach bardzo odległych od wyjścia. Zdaniem wielu badaczy jest to raczej koncepcja matematyczna (niemniej niezwykle użyteczna).

10.1.2 Maksymalizacja iloczynu skalarnego

Obecny algorytm klasteryzacji jest następujący:

- Przedłużamy punkty próbki o trzecią współrzędną, $x_3 = \sqrt{r^2 - x_1^2 - x_2^2}$, wybierając odpowiednio duże r, aby $r^2 > x_1^2 + x_2^2$ dla wszystkich punktów próbki.
- Inicjalizujemy wagi w taki sposób, że $\vec{w}_i \cdot \vec{w}_i = r^2$.

Następnie wykonujemy pętlę po punktach danych:

- Znajdujemy neuron w warstwie pośredniej, dla którego iloczyn skalarny $x \cdot \vec{w}_i$ jest największy. Zmieniamy wagi tego neuronu wg. wzoru

$$\vec{w}^i \to \vec{w}^i + \varepsilon(\vec{x} - \vec{w}^i).$$

- Renormalizujemy uaktualnione wagi $\vec{w}_i$ tak, aby $\vec{w}_i \cdot \vec{w}_i = r^2$:

$$\vec{w}^i \to \vec{w}^i \frac{r}{\sqrt{\vec{w}_i \cdot \vec{w}_i}}.$$

Pozostałe kroki algorytmu, takie jak wyznaczanie początkowych położeń punktów reprezentatywnych, ich dynamiczne tworzenie w miarę napotykania kolejnych punktów danych itp. pozostają dokładnie takie same, jak w poprzednio omawianej procedurze.

Uogólnienie dla n wymiarów jest oczywiste: wprowadzamy dodatkową współrzędną

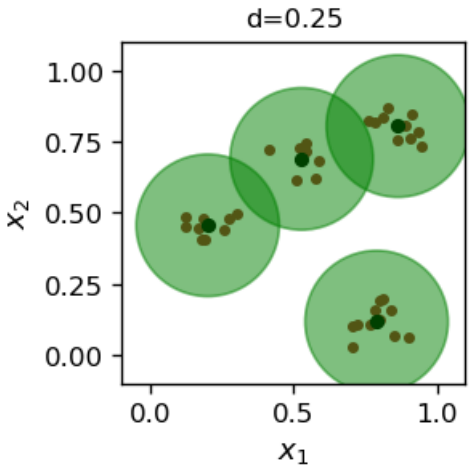
$$x_{n+1} = \sqrt{r^2 - x_1^2 - ... - x_n^2},$$

mamy więc punkt na hiperhemisferze $x_1^2 + \cdots + x_n^2 + x_{n+1}^2 = r^2$, $x_{n+1} > 0$.

W Pythonie odpowiedni kod jest następujący:

```
d=0.25
eps=.5

rad=2    # radius of the hypersphere

for r in range(25):
    eps=0.85*eps
    np.random.shuffle(alls)
    if r==0:
        p=alls[0]
        R=np.array([np.array([p[0],p[1],np.sqrt(rad**2 - p[0]**2 - p[1]**2)])])
                                    # extension of R to the hypersphere
    for i in range(len(alls)):
        p=np.array([alls[i][0], alls[i][1],
                    np.sqrt(rad**2 - alls[i][0]**2 - alls[i][1]**2)])
                                    # extension of p to the hypersphere
        dist=[np.dot(p,R[k]) for k in range(len(R))]      # array of dot products
        ind_max = np.argmax(dist)                        # maximum
        if dist[ind_max] < rad**2 - d**2/2:
             R=np.append(R, [p], axis=0)
        else:
            R[ind_max]+=eps*(p-R[ind_max])

print("Number of representative points: ",len(R))
```

```
Number of representative points:  4
```



Można łatwo zauważyć, że algorytm maksymalizacji iloczynu skalarnego daje niemal dokładnie taki sam wynik jak minimalizacja kwadratu odległości (por. Rys. 9.3).

10.2 Ćwiczenia

1. Metrykę miejską (Manhattanu) definiuje się jako

$d(\vec{x},\vec{y}) = |x_1-y_1|+|x_2-y_2|$ dla punktów $\vec{x}$ i $\vec{y}$. Powtórz symulacje z tego rozdziału, stosując tę metrykę. Wyciągnij wnioski.

1. Uruchom algorytm klasyfikacyjne dla większej liczby kategorii w próbce danych (wygeneruj własną próbkę).
2. Rozszerz algorytm dynamicznej klasteryzacji na trójwymiarową przestrzeń danych.

ROZDZIAŁ 11

Mapy samoorganizujące się

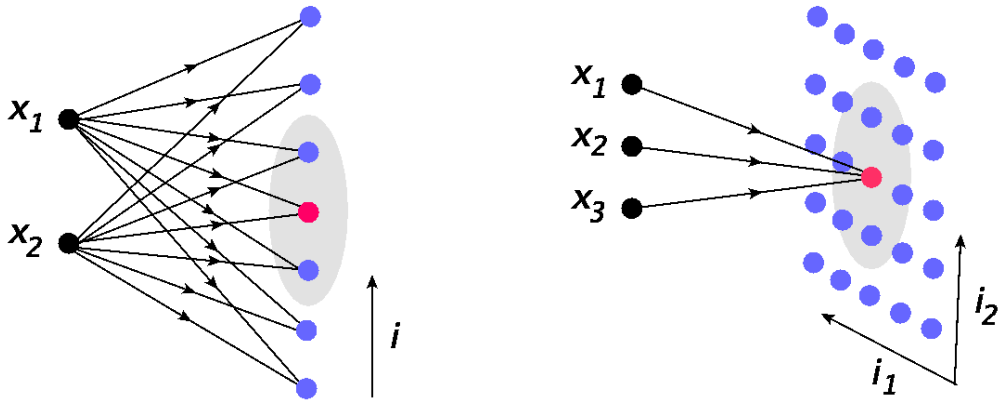
A very important and ingenious application of unsupervised learning are the so-called **Kohonen networks** (Teuvo Kohonen, a class of **self-organizing mappings (SOM)**. Consider firs a mapping f between a **discrete** k-dimensional set (we call it a **grid** in this chapter) of neurons and n-dimensional input data D (continuous or discrete),

$$f : N \to D$$

(note that **this is not a Kohonen mapping yet!**). Since N is discrete, each neuron carries an index consisting of k natural numbers, denoted as $\bar{i} = (i_1, i_2, ..., i_k)$. Typically, the dimensions in Kohonen's networks satisfy $n \geq k$. When $n > k$, one talks about **reduction of dimensionality**, as then the input space D has more dimensions than the dimensionaiy of the grid of neurons N.

Two examples of such networks are visualized in Rys. 11.1. The left panel shows a 2-dim. input space D, and a one dimensional grid on neurons labeled with i. The input point (x_1, x_2) enters all the neurons in the grid, and one of the neurons (the one with best-suited weights) becomes the **winner** (red dot). The gray oval indicates the **neighborhood** of the winner, to be defined accurately in the following.

The right panel shows an analogous situation for the case of a 3-dim. input and 2-dim. grid of neurons, now labeled with a double index $\bar{i} = (i_1, i_2)$. Here, for clarity, we only indicate the edges entering the winner, but they also enter all the other neurons in the grid, similarly to the left panel.



Rys. 11.1: Example of Kohonen's networks. Left: 1-dim. grid of neurons N and 2-dim. input space D. Right: 2-dim. grid of neurons N and 3-dim. input space D. The red dot indicates the winner, and the gray oval marks its neighborhood.

Next, one defines the neuron **proximity function**, $\phi(\bar{i}, \bar{j})$, which assigns, to a pair of neurons, a real number depending on their relative position in the grid. This function must decrease with the distance between the neuron indices. A popular choice is a Gaussian,

$$\phi(\bar{i}, \bar{j}) = \exp\left[-\frac{(i_1 - j_1)^2 + ... + (i_k - j_k)^2}{2\delta^2}\right],$$

where δ is the **neighborhood radius**. For a 1-dim. grid we have $\phi(i, j) = \exp\left[-\frac{(i-j)^2}{2\delta^2}\right]$.

11.1 Kohonen's algorithm

The set up for Kohonen's algorithm is similar to the unsupervised learning discussed in the previous chapter. Each neuron $\bar{i}$ obtains weights $f\,(\bar{i})$, which are elements of D, i.e. form n-dimensional vectors. One may simply think of this procedure as placing the neurons in some locations in D.

When an input point P from D is fed into the network, one looks for the closest neuron, which becomes the **winner**, exactly as in the unsupervised learning algorithm from section *Interpretacja jako ANN*. However, now comes a **crucial difference**: Not only the winner is attracted (updated) a bit towards P, but also its neighbors, to a lesser and lesser extent the farther they are from the winner, as quantified by the proximity function.

Winner-take-most strategy

Kohonen's algorithm involves the „winner take most" strategy, where not only the winner neuron is updated (as in the winner-take-all case), but also its neighbors. The neighbors update is strongest for the nearest neighbors, and gradually weakens with the distance from the winner, as given by the proximity function.

Kohnen's algorithm

1. Initialize (for instance randomly) n-dimensional weight vectors w_i, $i - 1, ..., m$ for all the m neurons in the grid. Set an an initial neighborhood radius δ and an initial learning speed ε.
2. Choose (for instance, randomly) a data point P with coordinates x from the input space (possibly with an appropriate probability distribution).
3. Find the neuron (the winner) for which the distance from P is the smallest. Denote its index as $\bar{l}$.
4. The weights of the winner and its neighbors are updated according to the **winner-take-most** recipe:

$$w_{\bar{i}} \to w_{\bar{i}} + \varepsilon\phi(\bar{i}, \bar{l})(x - w_{\bar{i}}), \qquad i = 1, ..., m.$$

1. Loop from 1. for a specified number of points.
2. Repeat from 1. in rounds, until a satisfactory result is obtained or a stopping criterion is reached. In each round **reduce** ε and δ according to a chosen policy.

Ważne: The way the reduction of ε and δ is done is very important for the desired outcome of the algorithm (see exercises).

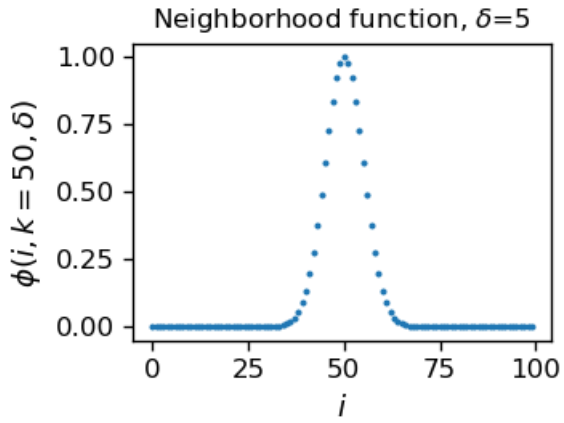
11.1.1 2-dim. data and 1-dim. neuron grid

```
num=100 # number of neurons
```

and the Gaussian proximity function

```
def phi(i,k,d):                          # proximity function
    return np.exp(-(i-k)**2/(2*d**2)) # Gaussian
```

This function looks as follows around the middle neuron ($k = 50$) and for the width parameter $\delta = 5$:



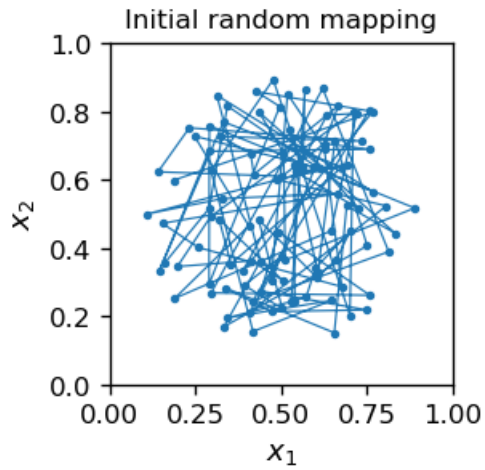
As a feature of a Gaussian, at $|k - i| = \delta$ the function drops to 60% of the central value, and at $|k - i| = 3\delta$ to 1%, a tiny fraction. Hence δ controls the size of the neighborhood of the winner. The neurons farther away from the winner than, say, 3δ are practically left uncharged.

We initiate the network by by placing the grid inside the circle, with a random location of each neuron. As said, this amounts to assigning weights to the neuron equal to its location. An auxiliary line is drawn to guide the eye sequentially along the neuron indices: $1, 2, 3, \ldots m$. The line has no other meaning.

The weights (neuron locations) are stored in array **W**:

```
W=np.array([func.point_c() for _ in range(num)]) # random initialization of␣
↪weights
```

As a result of the initial randomness, the neurons are, of course, „chaotically" distributed:



Next, we initialize the parameters **eps** amd **delta** and run the algorithm. Its structure is analogous to the previously discussed codes and is a straightforward implementation of the steps spelled out in the previous section. For that

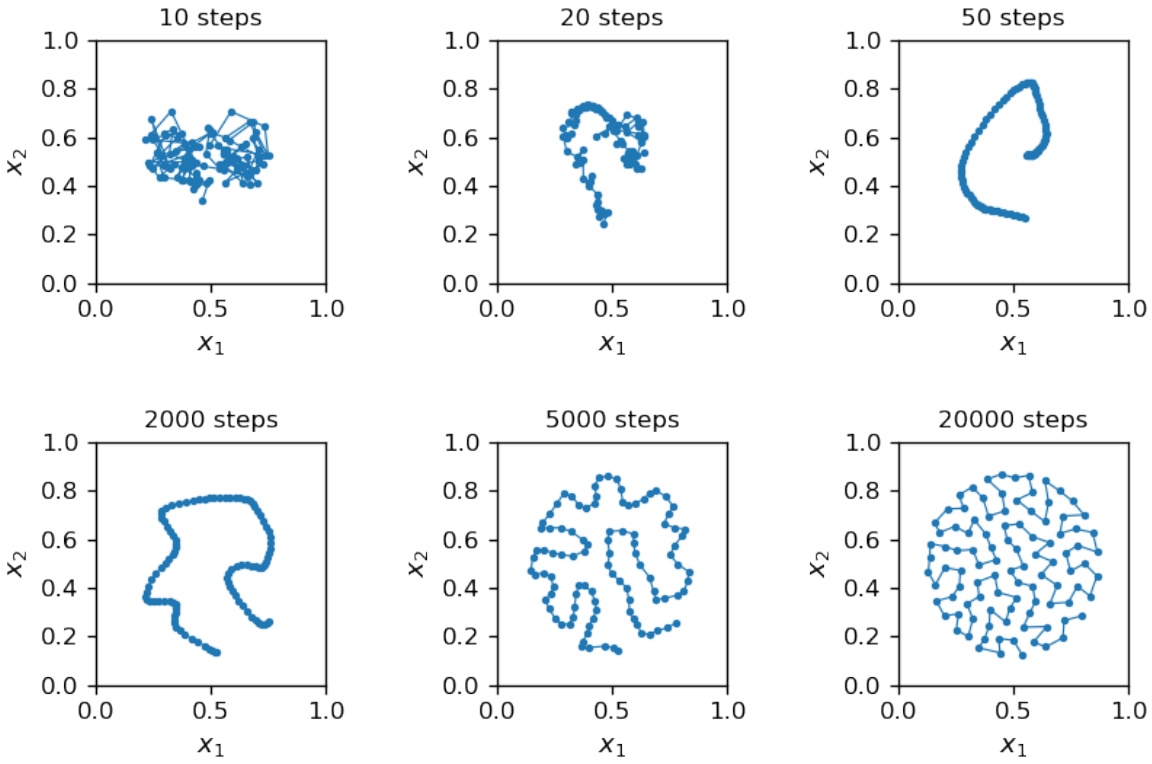
reason, we only provide the comments in the code.

```
eps=.5    # initial learning speed
de = 10   # initial neighborhood distance
ste=0     # inital number of caried out steps
```

```
# Kohonen's algorithm
for _ in range(100):                # rounds
    eps=eps*.98                     # dicrease learning speed
    de=de*.95                       # ... and the neighborhood distance
    for _ in range(150):            # loop over points
        p=func.point_c()            # random point
        ste=ste+1                   # count steps
        dist=[func.eucl(p,W[k]) for k in range(num)]
         # array of squares of Euclidean disances between p and the neuron␣
↪locations
        ind_min = np.argmin(dist) # index of the winner
        for k in range(num):        # loop over all the neurons
            W[k]+=eps*phi(ind_min,k,de)*(p-W[k])
             # update of the neuron locations (weights), depending on proximity
```

As the above algorithm progresses (see Rys. 11.2) the neuron grid first disentangles, and then gradually fills the whole space D (circle) in such a way that the neurons with adjacent indices are located close to each other. Figuratively speaking, a new point P attracts towards itself the nearest neuron (the winner), but also, to a weaker extent, its neighbors. At the beginning of the algorithm the neighborhood distance **de** is large, so large chunks of the neighboring neurons in the input grid are pulled together towards P, and the arrangement looks as in the top right corner of Rys. 11.2. At later stages **de** reduces, so only the winner and possibly its very immediate neighbors are attracted to a new point. After completion (bottom right panel), individual neurons „specialize" (are close to) in a certain data area.

In the present example, after about 20000 steps the result practically stops to change.



Rys. 11.2: Progress of Kohonen's algorithm. The line, drawn to guide the eye, connects neurons with adjacent indices.

Kohonen's network as a classifier

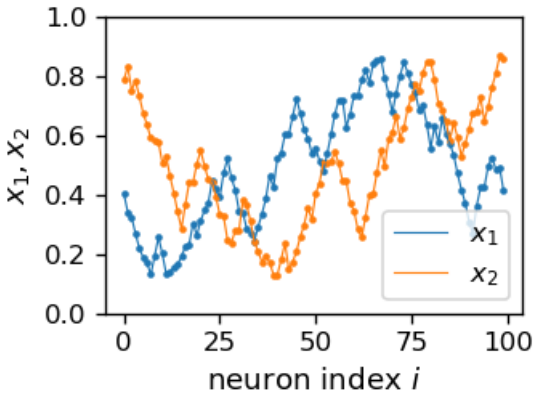
Having the trained network, we may use it as a classifier similarly as in chapter *Uczenie nienadzorowane*. We label a point from D with the index of the nearest neuron. One can interpret this as a Voronoi construction, see section *Komórki Woronoja*.

The plots in Rys. 11.2 are made in coordinates (x_1, x_2), that is, from the „point of view" of the input D-space. One may also look at the result from the point of view of the N-space, i.e. plot x_1 and x_2 as functions of the neuron index i.

Caution

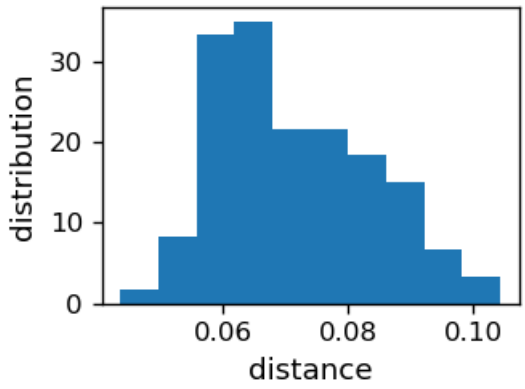
When presenting results of Kohonen's algorithm, one sometimes makes plots in D-space, and sometimes in N-space, which may lead to some confusion.

The plots in the N-space, fully equivalent in information to the plot in, e.g., the bottom right panel of Rys. 11.2, are following:



We note that the jumps in the above plotted curves are small, since the subsequent neurons are close to each other. This feature can be presented quantitatively as in the histogram below, where we can see that the average distance between the neurons is about 0.07, and the spread is between 0.05 and 0.10.

```
dd=[np.sqrt((W[i+1,0]-W[i,0])**2+(W[i+1,1]-W[i,1])**2) for i in range(num-1)]
        # array of distances between subsequent neurons in the grid

plt.figure(figsize=(2.8,2),dpi=120)

plt.xlabel('distance',fontsize=11)
plt.ylabel('distribution',fontsize=11)

plt.hist(dd, bins=10, density=True)    # histogram
plt.show()
```



Remarks

- We took a situation in which the data space with the dimension $n = 2$ is „sampled" by a discrete set of neurons forming $k = 1$-dimensional grid. Hence we encounter dimensional reduction.
- The outcome of the algorithm is a network in which a given neuron „focuses" on data from its vicinity. In a general case, where the data can be non-uniformly distributed, the neurons would fill the area containing more data more densely.
- The policy of choosing initial δ and ε parameters and reducing them appropriately in subsequent rounds is based on experience and is non-trivial. The results depend significantly on this choice.
- The final result, even with the same δ and ε strategy, is not unequivocal, i.e. running the algorithm with a different initialization of the weights (initial positions of neurons) yields different outcomes, usually equally „good".
- Finally, the progress and the result of the algorithm is reminiscent of the construction of the Peano curve in mathematics, which fills densely an area with a line. As we increase the number of neurons, the analogy gets closer and closer.

11.1.2 2 dim. color map

Now we pass to a case of 3-dim. data and 2-dim. neuron grid, which is a situation from the right panel of Rys. 11.1 (hence also with dimensionality reduction). As we know, an RGB color is described with three numbers $[r, g, b]$ from $[0, 1]$, so it can nicely serve as input in our example.

The distance squared between two colors (this is just a distance between two points in the 3-dim. space) is taken in the Euclidean form:

```
def dist3(p1,p2):
    """
    Square of the Euclidean distance between points p1 and p2
    in 3 dimensions.
    """
    return (p1[0]-p2[0])**2+(p1[1]-p2[1])**2+(p1[2]-p2[2])**2
```

```
dist3([1,.5,.5],[0,.3,.3])
```

```
1.08
```

The proximity function is now a Gaussian in two dimensions:

```
def phi2(ix,iy,kx,ky,d):  # proximity function for 2-dim. grid
    return np.exp(-((ix-kx)**2+(iy-ky)**2)/(d**2))  # Gaussian
```

We also decide to normalize the RGB colors such that $r^2+g^2+b^2=1$. This makes the perceived intensity of colors similar (this normalization could be dropped, as irrelevant for the method to work).

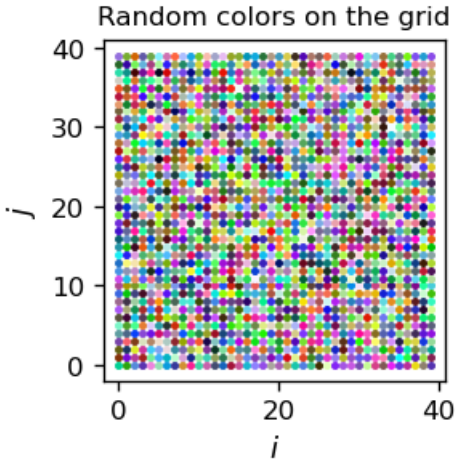
```
def rgbn():
    r,g,b=np.random.random(),np.random.random(),np.random.random() # random RGB
    norm=np.sqrt(r*r+g*g+b*b)                                      # norm
    return np.array([r,g,b]/norm)                                  # normalized␣
↪RGB
```

Next, we generate and plot a sample of **ns** points with (normalized) RGB colors:

```
ns=10                              # number of colors in the sample
samp=[rgbn() for _ in range(ns)] # random sample

pls=plt.figure(figsize=(4,1),dpi=120)
plt.axis('off')

for i in range(ns): plt.scatter(i,0,color=samp[i], s=10)

plt.show()
```


We use a 2-dim. **size** x **size** grid of neurons. Each neuron's position (that is its color) in the 3-dim. D-space is initialized randomly:

```
size=40                          # neuron array of size x size (40 x 40)
tab=np.zeros((size,size,3))      # create array tab with zeros

for i in range(size):            # i index in the grid
    for j in range(size):        # j index in the grid
        for k in range(3):       # RGB: k=0-red, 1-green, 2-blue
            tab[i,j,k]=np.random.random() # random number form [0,1]
            # 3 RGB components for neuron in the grid positin (i,j)
```



Now we are ready to run Kohonen's algorithm:

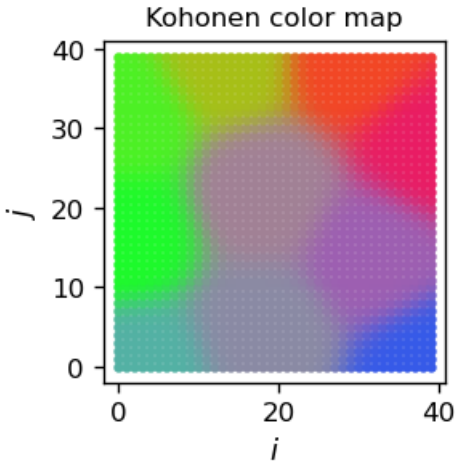
```
eps=.5   # initial parameters
de = 20
```

```
for _ in range(150):     # rounds
    eps=eps*.995
    de=de*.99            # de shrinks a bit faster than eps
    for s in range(ns): # loop over the points in the data sample
        p=samp[s]        # point from the sample
        dist=[[dist3(p,tab[i][j]) for j in range(size)] for i in range(size)]
                         # distance of p from all neurons
        ind_min = np.argmin(dist)  # the winner index
        ind_1=ind_min//size        # a trick to get a 2-dim index
        ind_2=ind_min%size
#         print("winner:",ind_1,ind_2)

        for j in range(size):
            for i in range(size):
                tab[i][j]+=eps*phi2(ind_1,ind_2,i,j,de)*(p-tab[i][j]) # update
 ↪
```

A word of explanation is in place here, concerning the numpy **argmin** function. For a 2-dim. array it provides the index of the minimum in the corresponding **flattened** array (cf. section *Pamieć skojarzeniowa (heteroasocjacyjna)*). Hence, to get the indices in the two dimensions, we need to apply the operations **//** (integer division) and **%** (remainder). For instance, in an array **ind_min=53**, then **ind_1=ind_min//size=53//10=5** and **ind_2=ind_min%size=53//10=3**.

As a result of the above code, we get an arrangement of our color sample in two dimensions in such a way that the neighboring areas in the grid have a similar color „specializing" on the color of a given sample point (note the plot is in the N-space):

```
plt.figure(figsize=(2.3,2.3),dpi=120)
plt.title("Kohonen color map",fontsize=10)

for i in range(size):
    for j in range(size):
        plt.scatter(i,j,color=tab[i][j], s=8)

plt.xlabel('$i$',fontsize=11)
plt.ylabel('$j$',fontsize=11)
plt.show()
```



Remarks

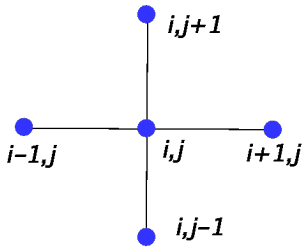
- The areas for the individual colors of the sample have a comparable area. Generally, the area is proportional to the frequency of the data point in the sample.
- To get sharper boundaries between the regions, **de** would have to shrink even faster compared to **eps**. Then, in the final stage of learning, the neuron update process takes place within a smaller neighborhood radius and more resolution in the boundaries can be achieved.

11.2 U-matrix

A convenient way to present the results of Kohonen's algorithm when the grid is 2-dimensional is via the **unified distance matrix** (shortly U**-matrix**). The idea is to plot a 2-dimensional grayscale map in N-space with the intensity given by the averaged distance (in D-space) of the given neuron to its immediate neighbors, and not a neuron property itself (such as its color in the figure above). This is particularly useful when the dimension of the input space is large, hence it is difficult to visualize the results directly.

The definition of a U-matrix element U_{ij} is explained in Rys. 11.3. Let d be the distance in D-space and $[i,j]$ denote the neuron of indices i,j . We take

$$U_{ij} = \sqrt{d\left([i,j],[i+1,j]\right)^2 + d\left([i,j],[i-1,j]\right)^2 + d\left([i,j],[i,j+1]\right)^2 + d\left([i,j],[i,j-1]\right)^2}.$$



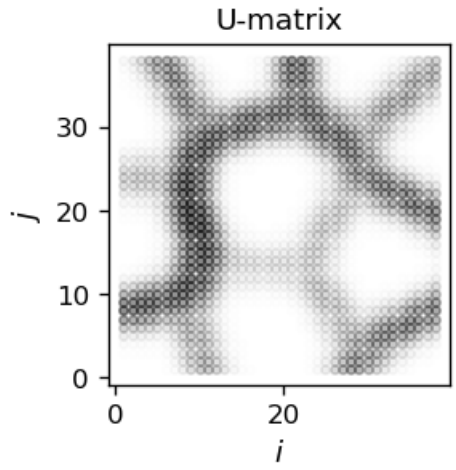
Rys. 11.3: Construction of U_{ij}: a geometric average of the distances along the indicated links.

The Python implementation of the above definition is following:

```
udm=np.zeros((size-2,size-2))     # initiaize U-matrix with elements set to 0

for i in range(1,size-1):         # loops over the neurons in the grid
    for j in range(1,size-1):
        udm[i-1][j-1]=np.sqrt(dist3(tab[i][j],tab[i][j+1])+dist3(tab[i][j],
 ↪tab[i][j-1])+
                              dist3(tab[i][j],tab[i+1][j])+dist3(tab[i][j],tab[i-
 ↪1][j]))
                                  # U-matrix as explained above
```

The result, corresponding one-to-one to the color map above, can be presented in a contour plot:

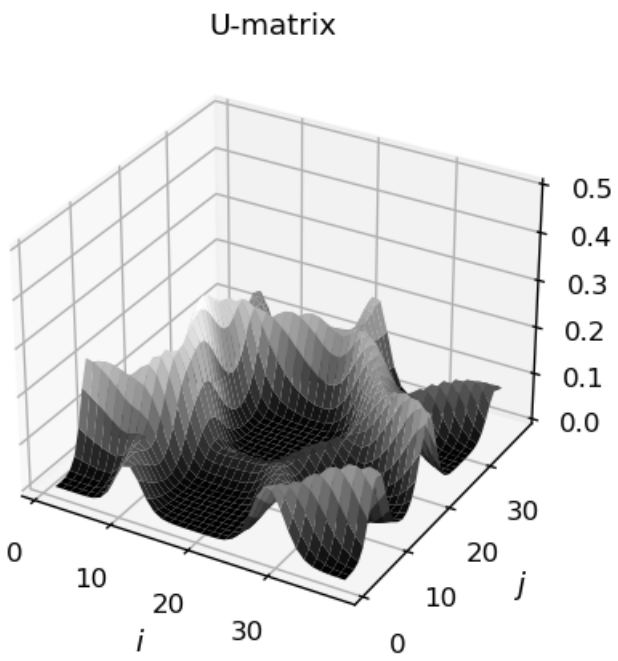
```
plt.figure(figsize=(2.3,2.3),dpi=120)
plt.title("U-matrix",fontsize=11)

for i in range(size-2): # loops over indices, excluding the boundaries of the grid
    for j in range(size-2):
        plt.scatter(i+1,j+1,color=[0,0,0,2*udm[i][j]], s=10)
                        # color format: [R,G,B,intensity], 2 just scales up
plt.xlabel('$i$',fontsize=11)
plt.ylabel('$j$',fontsize=11)
plt.show()
```



The white regions in the above figure show the clusters (they correspond one-to-one to the regions of the same color in the previously shown color map). There, the elements $U_{ij} \simeq 0$. The clusters are separated with darker boundaries. The higher the dividing ridge between clusters, the darker the intensity.

The result may also be visualized with a 3-dim. plot:

```
fig = plt.figure(figsize=(4,4),dpi=120)
axes1 = fig.add_subplot(111, projection="3d")
ax = fig.gca()

xx_1 = np.arange(1, size-1, 1)
xx_2 = np.arange(1, size-1, 1)

x_1, x_2 = np.meshgrid(xx_1, xx_2)

Z=np.array([[udm[i][j] for i in range(size-2)] for j in range(size-2)])

ax.set_zlim(0,.5)

ax.plot_surface(x_1,x_2, Z, cmap=cm.gray)

plt.xlabel('$i$',fontsize=11)
plt.ylabel('$j$',fontsize=11)
plt.title("U-matrix",fontsize=11)
plt.show()
```



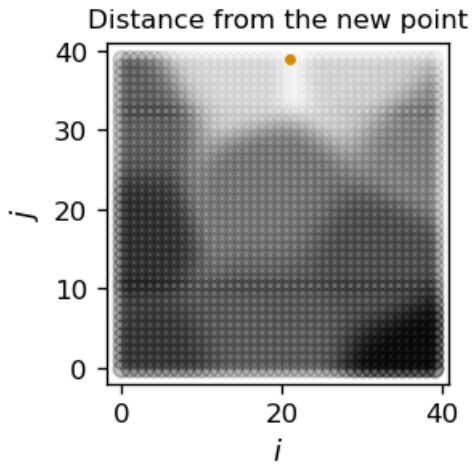
We can now classify a given (new) data point according to the obtained map. We generate a new (normalized) RGB color:

```
nd=rgbn()
```

It is useful to obtain a map of distances of our grid neurons from this point:

```
tad=np.zeros((size,size))

for i in range(size):
    for j in range(size):
        tad[i][j]=dist3(nd,tab[i][j])


ind_m = np.argmin(tad) # winner
in_x=ind_m//size
in_y=ind_m%size

da=np.sqrt(tad[in_x][in_y])

print("Closest neuron grid indices: (",in_x,",",in_y,")")
print("Distance: ",np.round(da,3))
```

```
Closest neuron grid indices: ( 21 , 39 )
Distance:  0.133
```



The lightest region in the above figure indicates the cluster, to which the new point belongs. The darker the region, the larger is the distance from the corresponding neuron.

One should stress that we have obtained a classifier which not only assigns a closest cluster to a probed point, but also provides its distances from all other clusters.

11.2.1 Mapping colors on a line

In this subsection we present an example of a mapping of 3-dim. data into a 1-dim. neuron grid, hence a reduction of three dimensions into one. This proceeds exactly along the lines of the previous subsection, so we are very brief in comments.

```
ns=8
samp=[rgbn() for _ in range(ns)]

plt.figure(figsize=(4,1))
plt.title("Sample colors",fontsize=16)

plt.axis('off')

for i in range(ns):
    plt.scatter(i,0,color=samp[i], s=400)

plt.show()
```

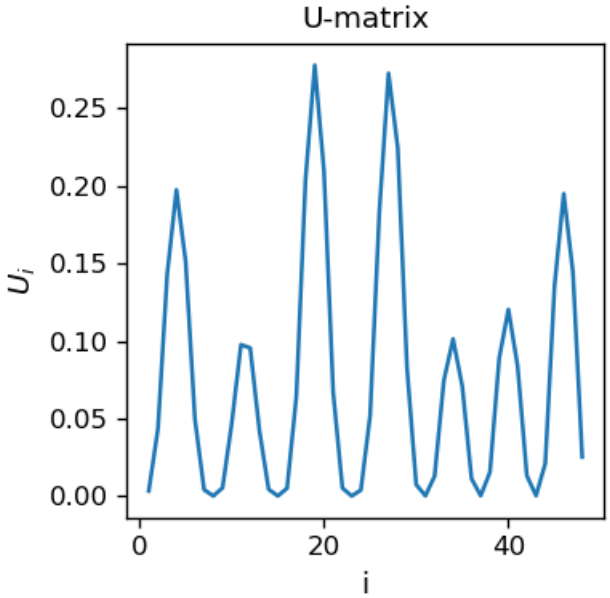

```
si=50                    # 1-dim. grid of si neurons, 3 RGB components
tab2=np.zeros((si,3))    # neuron gri

for i in range(si):
    tab2[i]=rgbn()     # random initialization
```

```
eps=.5
de = 20
```

```
for _ in range(200):
    eps=eps*.99
    de=de*.96
    for s in range(ns):
        p=samp[s]
        dist=[dist3(p,tab2[i]) for i in range(si)]
        ind_min = np.argmin(dist)
        for i in range(si):
            tab2[i]+=eps*phi(ind_min,i,de)*(p-tab2[i])
```

As expected, we note smooth transitions between colors. The formation of clusters can be seen with the U-matrix, which now is, of course, one-dimensional:

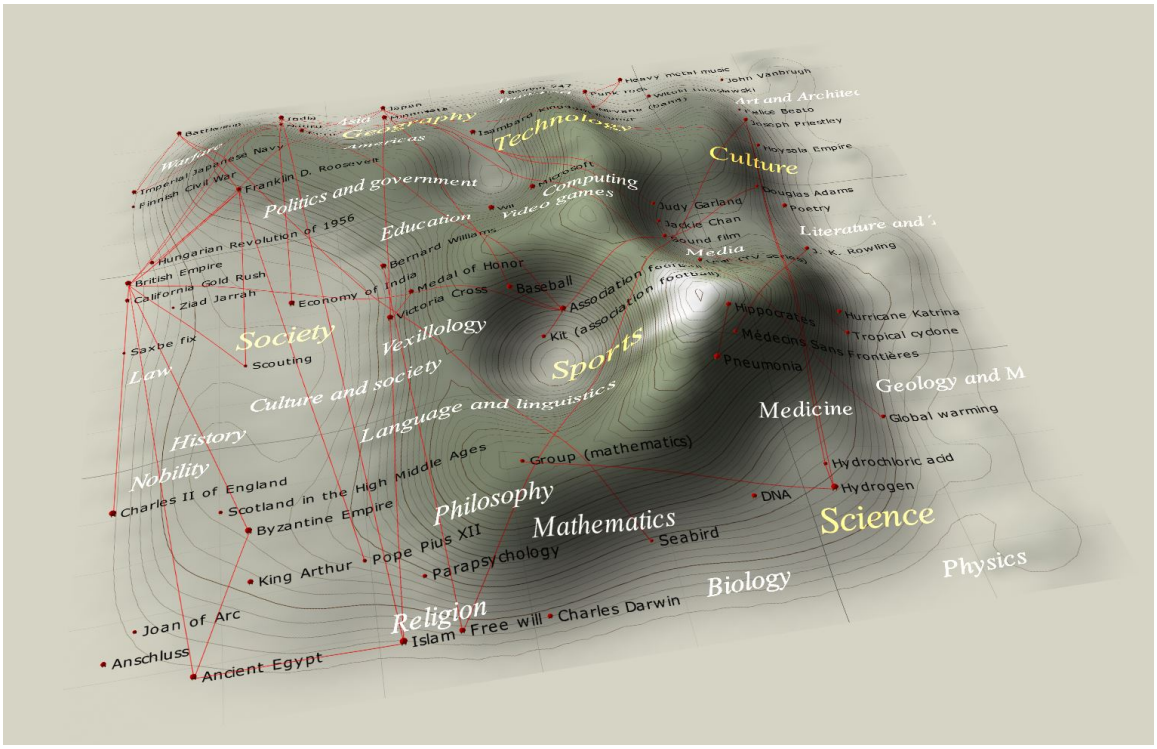
```
ta2=np.zeros(si-2)

for i in range(1,si-1):
    ta2[i-1]=np.sqrt(dist3(tab2[i],tab2[i+1])+dist3(tab2[i],tab2[i-1]))
```



The minima (there are 8 of them, equal to the multiplicity of the sample) indicate the clusters. The height of the separating peaks shows how much the neighboring colors differ. Again, we see a nicely produced classifier, this time with two dimensions „hidden away", as we reduce from three to one.

11.2.2 Large reduction of dimensionality

In many situations the input space may have a very large dimension. In the Wikipedia example quoted here, one takes articles from various fields and computes frequencies of used words (for instance, in a given article how many times the word „goalkeeper" has been used, divided by the total number of words in the article). Essentially, the dimensionality of D is of the order of the number of all English words, a huge number $\sim 10^5$! Then, with a properly defined distance depending on these frequencies, one uses Kohonen's algorithm to carry out a reduction into a 2-dim. grid of neurons. The resulting U-matrix can be drawn as follows:



Not surprisingly, we notice that articles on sports are special and form a very well defined cluster. The reason is that the sport's jargon is very specific. The Media are also distinguished, whereas other fields are more-less uniformly distributed. The example shows how we can, with a very simple method, comprehend data in a multidimensional space and see specific correlations/clusters. Whereas some conclusions may be obvious, such as the fact that sport has a unique jargon, other are less transparent, for instance the emergence of the media cluster and lack of well-defined clusters for other fields, e.g. for mathematics.

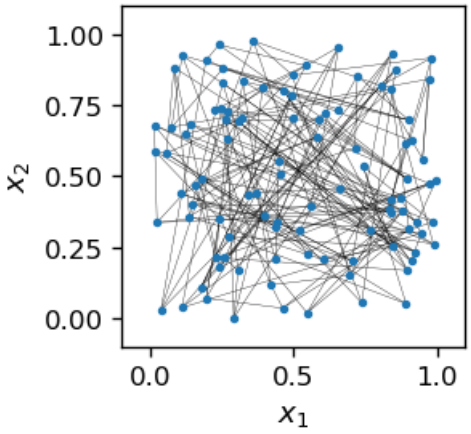
11.3 Mapping 2-dim. data into a 2-dim. grid

Finally, we come to a very important case of mapping 2-dim. data in a 2-dim. grid, i.e. with no dimensionality reduction. In particular, this case is realized in our vision system between the retina and the visual cortex.

The algorithm proceeds analogously to the previous cases. We initialize an $n \times n$ grid of neurons and place them randomly in the square $[0, 1] \times [0, 1]$.

```
n=10
sam=np.array([func.point() for _ in range(n*n)])
```

The lines, again drawn to guide the eye, join the adjacent index pairs in the grid: [i,j] and [i+1,j], or [i,j] and [i,j+1] (the neurons in the interior of the grid have 4 nearest neighbors, those at the boundary 3, except for the corners, which have only 2).



We note a total initial „chaos", as the neurons are located randomly. Now comes Kohonen's miracle:

```
eps=.5   # initial learning speed
de = 3   # initial neighborhood distance
nr = 100 # number of rounds
rep= 300 # number of points in each round
ste=0    # inital number of caried out steps
```

```
# completely analogous to the previous codes of this chapter
for _ in range(nr):   # rounds
    eps=eps*.97
    de=de*.98
    for _ in range(rep):    # repeat for rep points
        ste=ste+1
        p=func.point()
        dist=[func.eucl(p,sam[l]) for l in range(n*n)]
        ind_min = np.argmin(dist)
        ind_i=ind_min%n
        ind_j=ind_min//n

        for j in range(n):
            for i in range(n):
                sam[i+n*j]+=eps*phi2(ind_i,ind_j,i,j,de)*(p-sam[i+n*j])
```

Here is the history of a simulation:

As the algorithm progresses, the initial „chaos" gradually changes into a nearly perfect order, with the grid placed uniformly in the square of the data, with only slight displacements from a regular arrangement. On the way, near 40 steps, we notice a phenomenon called „twist", where the grid is crumpled. In the twist region, many neurons, also of distant indices, have a close location in (x_1, x_2).

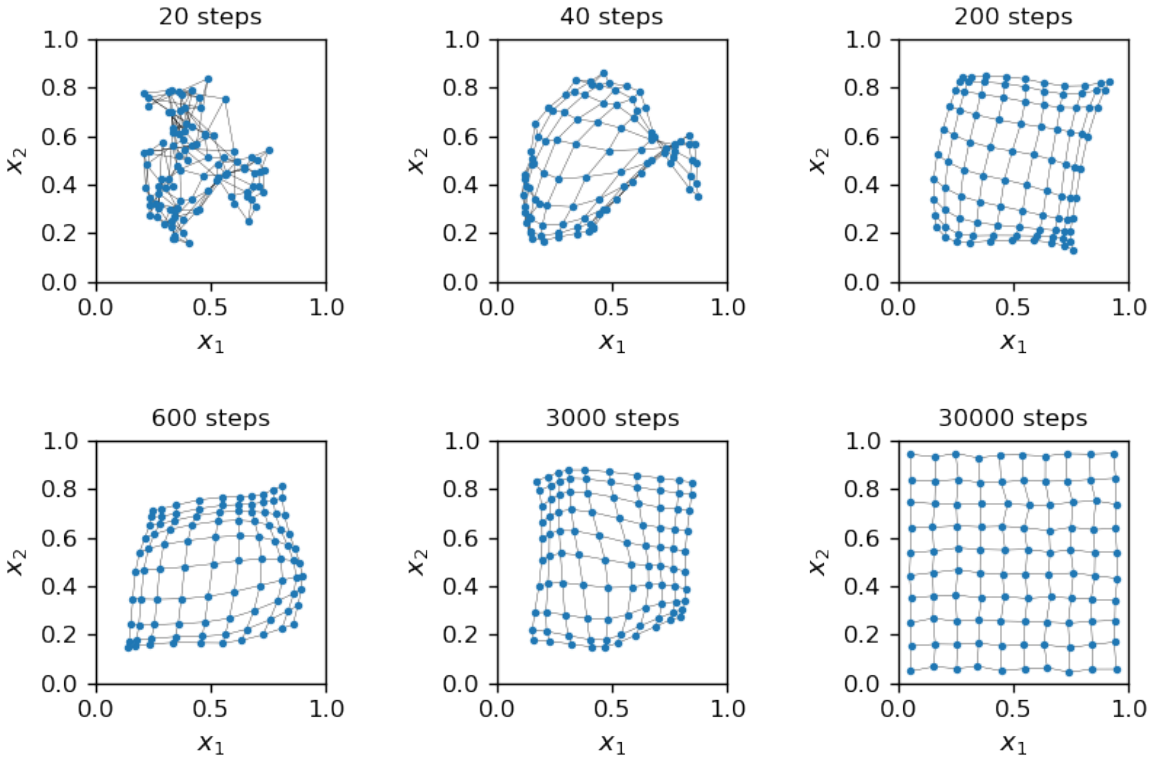
11.4 Topology

Recall the Voronoi construction of categories introduced in section *Komórki Woronoja*. One can use it now again, treating the neurons from a grid as the Voronoi points. The Voronoi construction provides a mapping v from the data space D to the neuron space N,

$$v : D \to N$$

(note that this goes in the opposite direction than function f defined at the beginning of this chapter).

The procedure is as follows: We take the final outcome of the algorith, such as in the bottom right panel of Rys. 11.4, construct the Voronoi areas for all the neurons, and thus obtain a mapping v for all the points in the (x_1, x_2) square.



Rys. 11.4: Progress of Kohonen's algorithm. The lines, drawn to guide the eye, connects neurons with adjacent indices.

The reader may notice that there is an ambiguity for points lying exactly at the boundaries between the neighboring areas, but this can be taken care of by using an additional prescription (for instance, selecting a neuron lying at a direction which has the lowest azimuthal angle, etc.)

Now a key observation:

Topological property

For situations without twists, such as in the bottom right panel of Rys. 11.4, mapping v has the property that when d_1 and d_2 from D are close to each other, then also their corresponding neurons are close, i.e. the indices $v(d_1)$ and $v(d_2)$ are close.

This observation is straightforward to prove: Since d_1 and d_2 are close (and we mean very close, closer than the grid spacing), they must belong either to

- the same Voronoi area, where $v(d_1) = v(d_2)$, or
- a pair of neighboring Voronoi areas.

Since for the considered situation (without twists) the neighboring areas have the grid indices differing by 1, the conclusion that $v(d_1)$ and $v(d_2)$ are close follows immediately.

Note that this feature of Kohonen's maps is far from trivial and does not hold for a general mapping. Imagine for instance that we stop our simulations for Rys. 11.4 after 40 steps (top central panel) and are left with a „twisted" grid. In the vicinity of the twist, the indices of the adjacent Voronoi areas differ largely, and the advertised topological property no longer holds.

The discussed topological property has mathematically general and far-reaching consequences. First, it allows to carry over „shapes" from D to N. We illustrate it on an example.

Imagine that we have a circle C in D-space, of radius **rad** centered at **cent**. We need to find the winners in the N space for any point in C. For this purpose we go around C in **npoi** points equally spaced in the azimuthal angle, and

for each one find a winner.

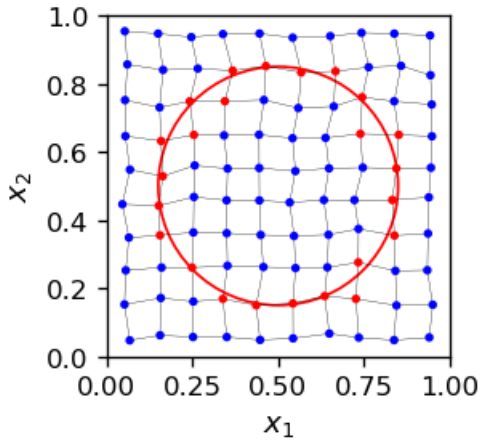
C is parametrized with polar coordinates:

$$x_1 = r\cos\left(\frac{2\pi\phi}{N}\right) + c_1, \quad x_2 = r\sin\left(\frac{2\pi\phi}{N}\right) + c_2.$$

Going to the mathematical notation to Python we use r =**rad**, ϕ=**ph**, N =**npoi**, (c_1, c_2) =**[cent]**. The loop over **ph** goes around the circle.

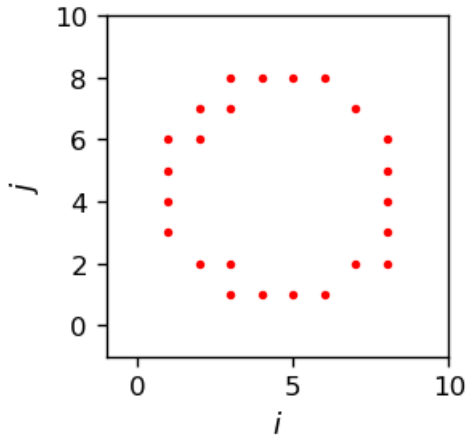
```
rad=0.35                        # radius of a circle
cent=np.array([0.5,0.5])        # center of the circle
npoi=400                        # number of points in the circle

wins=[]                         # table of winners

for ph in range(npoi):          # go around the circle
    p=np.array([rad*np.cos(2*np.pi/npoi*ph),rad*np.sin(2*np.pi/npoi*ph)])+cent
                                # the circle in polar coordinates
    dist=[func.eucl(p,sam[l]) for l in range(n*n)]
      # distances from the point on the circle to the neurons in the nxn grid
    ind_min = np.argmin(dist) # winner
    wins.append(ind_min)        # add winner to the table

ci=np.unique(wins)              # remove duplicates from the table
```

The result of Kohonen's algorithm is as follows:



The red neurons are the winners for certain sections of the circle. When we draw these winners alone in the N space (keep in mind we are going from D to N), we get

```
plt.figure(figsize=(2.3,2.3),dpi=120)

plt.xlim(-1,10)
plt.ylim(-1,10)

plt.scatter(ci//10,ci%10,c='red',s=5)

plt.xlabel('$i$',fontsize=11)
plt.ylabel('$j$',fontsize=11)
plt.show()
```



This looks pretty much as a (rough and discrete) circle. Note that in our example we only have $n^2 = 100$ pixels to our disposal - a very low resolution. The image would look better and better with an increasing n. At some point one would reach the 10M pixel resolution of typical camera, and then the image would seem smooth! We have carried over our circle from D into N.

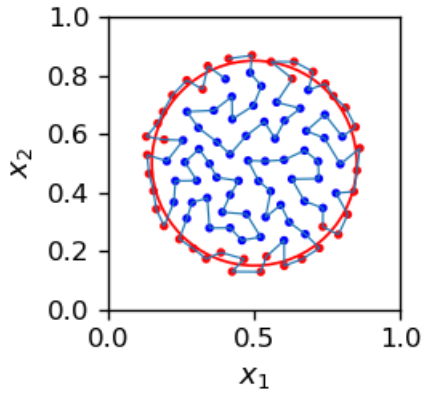
Vision

The topological property, such as the one in the discussed Kohonen mappings, has a prime importance in our vision system and the perception of objects. Shapes are carried over from the retina to the visual cortex and are not „warped up" on the way!

Another key topological feature is the preservation of **connectedness**. If an area A in D is connected (so to speak, is in one piece), then its image $v(A)$ in N is also connected (we ignore the desired rigor here as to what „connected" means in a discrete space and rely on intuition). So things do not get „torn into pieces" when transforming from D to N.

Note that the discussed topological features need not be present when the dimensionality is reduced, as in our previous examples. Take for instance the bottom right panel of Rys. 11.2. There, many neighboring pairs of the Voronoi areas correspond to distant indices, so it is no longer true that $v(d_1)$ and $v(d_2)$ in N are close for close d_1 and d_2 in D, as these points may belong to different Voronoi areas with **distant** indices.

For that case, our example with the circle looks like this:



When we go subsequently along the **grid indices** (i.e. along the blue connecting line), taking $i = 1, 2, \ldots, 100$, we obtain the plot below. We can see the image of our circle (red dots) as a bunch of **disconnected** red sections. The circle is torn into pieces, the **topology is not preserved!**

Here is the summarizing statement (NB not made sufficiently clear in the literature):

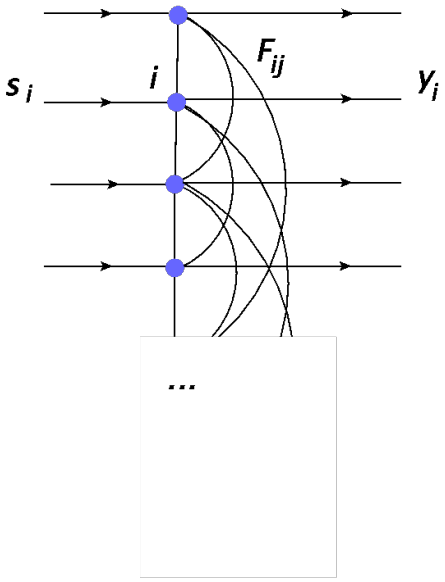
Informacja: Topological features of Kohonen's maps hold for equal dimensionalities of the input space and the neuron grid, $n = k$, and in general do not hold for the reduced dimensionality cases, $k < n$.

11.5 Lateral inhibition

In the last topic of these lectures, we return to the issue of how the competition for the „winner" is realized in ANNs. Up to now (cf. section *Interpretacja jako ANN*), we have just been using the minimum (or maximum, when the signal was extended to a hyperphere) in the output, though this is embarrassingly outside of the neural framework. Such an inspection of which neuron yields the strongest signal would require an „external wizard", or some sort of a control unit. Mathematically, it is easy to imagine, but the challenge is to build it from neurons within the rules of the game.

Actually, if the neurons in a layer „talk" to one another, we can have a „contest" from which a winner may emerge. In particular, an architecture as in Rys. 11.5 allows for an arrangement of competition and a natural realization of a **winner-take-most** mechanism.

The type of models as presented below is known as the Hopfield networks. Note that we depart here from the **feed-forward** limitation of Rys. 1.3 and allow for a recursive, or feed-back character.



Rys. 11.5: Network with inter-neuron couplings used for modeling lateral inhibition. All the neurons are connected to one another in both directions (lines without arrows).

Neuron number i receives the signal $s_i = xw_i$, where x is the input (the same for all the neurons), and w_i is the weight of neuron i. The neuron produces output y_i, where now a part of it is sent to neurons j as $F_{ji}y_i$. Here F_{ij} denotes the coupling strength (we assume $F_{ii} = 0$ - no self coupling). Reciprocally, neuron i also receives output from neurons j in the form $F_{ij}y_j$. The summation over all the neurons yields

$$y_i = s_i + \sum_{j \neq i} F_{ij} y_j,$$

which in the matrix notation becomes $y = s + Fy$, or $y(I - F) = s$, where I is the identity matrix. Solving for y gives formally

$$y = (I - F)^{-1} s. \tag{11.1}$$

One needs to model appropriately the coupling matrix F. We take

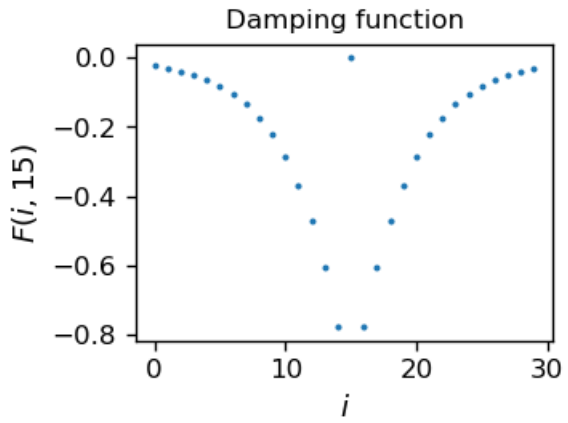
$F_{ii} = 0,$

$F_{ij} = -a \exp(-|i-j|/b) \quad \text{for } i \neq j, \quad a, b > 0,$

i.e. assume attenuation (negative feedback), which is strongest for close neighbors and decreases with distance. The decrease is controlled by a characteristic scale b.

The Python implementation is straightforward:

```
ns = 30;        # number of neurons
b = 4;          # parameter controlling the decrease of damping with distance
a = 1;          # magnitude of damping

F=np.array([[-a*np.exp(-np.abs(i-j)/b) for i in range(ns)] for j in range(ns)])
                    # exponential fall-off

for i in range(ns):
    F[i][i]=0       # no self-coupling
```



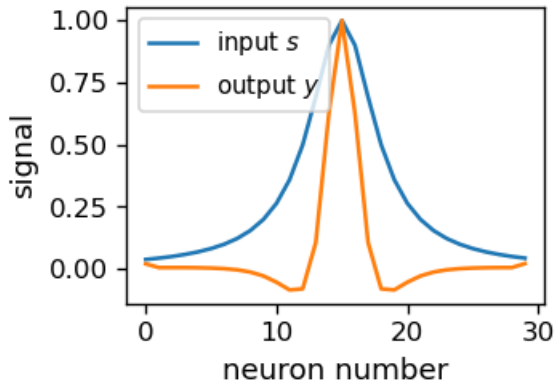
We assume a bell-shaped Lorentzian input signal s, with a maximum in the middle neuron. The width is controlled with **D**:

```
D=3
s = np.array([D**2/((i - ns/2)**2 + D**2) for i in range(ns)]) # Lorentzian␣
↪function
```

Next, we solve Eq. (11.1) via inverting the $(I - F)$ matrix, performed with the numpy **linalg.inv** function. Recall that **dot** multiplies matrices:

```
invF=np.linalg.inv(np.identity(ns)-F) # matrix inversion
y=np.dot(invF,s)                      # multiplication
y=y/y[15]                             # normalization (inessential)
```

What follows is actually quite remarkable: the output signal y becomes much narrower from the input signal s. This may be interpreted as a realization of the „winner-take-all" scenario. The winner „damped" he guys around him, so he puts himself on airs! The effect is smooth, with the signal visibly sharpened.

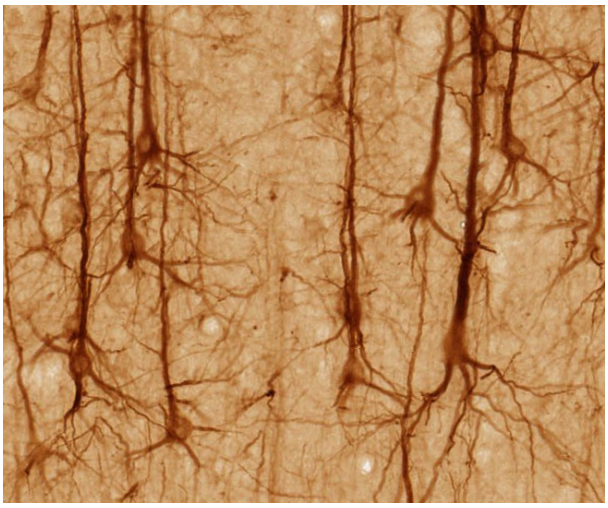


Lateral inhibition

The damping of the response of neighboring neurons is called **lateral inhibition**. It was discovered in neurobiological networks [HR72].

The presented model is certainly too simplistic to be realistic from the point of view of biological networks. Also, it yields unnatural negative signal outside of the central peak (which we can remove with rectification). Nevertheless, the setup shows a possible way to achieve the „winner competition", essential for unsupervised learning: One needs to allow for the competing neurons to interact.

Informacja: Actually, **pyramidal neurons**, present i.a. in the neocortex, have as many as a few thousand dendritic spines and do realize a scenario with numerous synaptic connections. They are believed Quantamagazine to play a crucial role in learning and cognition processes.



Rys. 11.6: Image of pyramidal neurons (from brainmaps.org)

11.6 Exercises

1. Construct a Kohonen mapping form a **disjoint** 2D shape into a 2D grid of neurons.
2. Construct a Kohonen mapping for a case where the points in the input space are not distributed uniformly, but denser in some regions.
3. Create, for a number of countries, fictitious flags which have two colors (hence are described with 6 RGB numbers). Construct a Kohonen map into a 2-dim. grid. Plot the resulting U-matrix and draw conclusions.
4. Lateral inhibition has „side-effects" seen in optical delusions. Describe the Mach illusion, programming it in Python.

ROZDZIAŁ 12

Concluding remarks

In a programmer's life, building a well-functioning ANN, even for simple problems as used for illustrations in these lectures, can be a truly frustrating experience! There are many subtleties involved on the way. To list just a few that we have encountered in this course, one faces a choice of the network architecture and freedom in the initialization of weights (hyperparameters). Then, one has to select an initial learning speed, a neighborhood distance, in general, some parameters controlling the performance/convergence, as well as their update strategy as the algorithm progresses. Further, one frequently tackles with an emergence of a massive number of local minima in the space of hyperparameters, and many optimization methods may be applied here, way more sophisticated than our simplest steepest-descent method. Moreover, a proper choice of the neuron activation functions is crucial for success, which relates to avoiding the problem of „dead neurons". And so on and on, many choices to be made before we start gazing in the screen in hopes that the results of our code converge …

Ważne: Taking the right decisions for the above issues is an **art** more than science, based on long experience of multitudes of code developers and piles of empty pizza boxes!

Now, having completed this course and understood the basic principles behind the simplest ANNs inside out, the reader may safely jump to using professional tools of modern machine learning, with the conviction that inside the black boxes there sit essentially the same little codes he met here, but with all the knowledge, experience, tricks, provisions, and options built in. Achieving this conviction, through appreciation of simplicity, has been one of the guiding goals of this course.

12.1 Acknowledgments

The author thanks Jan Broniowski for priceless technical help and for remarks to the text.

ROZDZIAŁ 13

Dodatki

13.1 Jak uruchamiać kody książki

13.1.1 Lokalnie

Czytelnik może pobrać notebooki Jupytera dla każdego rozdziału, klikając ikonę pobierania (strzałka w dół) po prawej stronie na górnym pasku podczas przeglądania książki.

Po zainstalowaniu Pythona i Jupytera (najlepiej przez conda), Czytelnik może postępować zgodnie z instrukcjami dla danego systemu operacyjnego, aby otworzyć Jupyter i uruchomić w nim notebooki wykładu.

13.1.2 Google Colab lub Binder

W danym rozdziale poniżej Wstępu należy kliknąć symbol rakiety w górnym prawym rogu ekranu, co uruchamia możliwość wykonywania (edycji, zabawy) programu w chmurze. Jest to podstawowa funkcjonalnośc wykonywalnej książki Jupyter Book.

13.2 Pakiet neural

Structura pakietu biblioteki jest następująca:

```
lib_nn
└── neural
    ├── __init__.py
    ├── draw.py
    └── func.py
```

i składa się z dwóch modułów: **func.py** and **draw.py**.

13.2.1 Moduł func.py

```
"""
Contains functions used in the lecture
"""

import numpy as np


def step(s):
    """
    step function

    s: signal

    return: 1 if s>0, 0 otherwise
    """
    if s>0:
        return 1
    else:
        return 0


def neuron(x,w,f=step):
    """
    MCP neuron

    x: array of inputs  [x1, x2,...,xn]
    w: array of weights [w0, w1, w2,...,wn]
    f: activation function, with step as default

    return: signal=f(w0 + x1 w1 + x2 w2 +...+ xn wn) = f(x.w)
    """
    return f(np.dot(np.insert(x,0,1),w))


def sig(s,T=1):
    """
    sigmoid

    s: signal
    T: temperature

    return: sigmoid(s)
    """
    return 1/(1+np.exp(-s/T))


def dsig(s, T=1):
    """
    derivative of sigmoid

    s: signal
    T: temperature

    return: dsigmoid(s,T)/ds
    """
    return sig(s)*(1-sig(s))/T


def lin(s,a=1):
    """
```

(ciąg dalszy na następnej stronie)

(kontynuacja poprzedniej strony)

```
    linear function

    s: signal
    a: constant

    return: a*s
    """
    return a*s


def dlin(s,a=1):
    """
    derivative of linear function

    s: signal
    a: constant

    return: a
    """
    return a


def relu(s):
    """
    ReLU function

    s: signal

    return: s if s>0, 0 otherwise
    """
    if s>0:
        return s
    else:
        return 0


def drelu(s):
    """
    derivative of ReLU function

    s: signal

    return: 1 if s>0, 0 otherwise
    """
    if s>0:
        return 1
    else:
        return 0


def lrelu(s,a=0.1):
    """
    Leaky ReLU function

    s: signal
    a: parameter

    return: s if s>0, a*s otherwise
    """
    if s>0:
        return s
```

(ciąg dalszy na następnej stronie)

(kontynuacja poprzedniej strony)

```
    else:
        return a*s


def dlrelu(s,a=0.1):
    """
    derivative of Leaky ReLU function

    s: signal
    a: parameter

    return: 1 if s>0, a otherwise
    """
    if s>0:
        return 1
    else:
        return a


def softplus(s):
    """
    softplus function

    s: signal

    return: log(1+exp(s))
    """
    return np.log(1+np.exp(s))


def dsoftplus(s):
    """
    derivative of softplus function

    s: signal

    return: 1/(1+exp(-s))
    """
    return 1/(1+np.exp(-s))


def l2(w0,w1,w2):
    """for separating line"""
    return [-.1,1.1],[-(w0-w1*0.1)/w2,-(w0+w1*1.1)/w2]


def eucl(p1,p2):
    """
    Square of the Euclidean distance between two points in 2-dim. space

    input: p1, p2 - arrays in the format [x1,x2]

    return: square of the Euclidean distance
    """
    return (p1[0]-p2[0])**2+(p1[1]-p2[1])**2


def rn():
    """
    return: random number from [-0.5,0.5]
    """
```

(ciąg dalszy na następnej stronie)

(kontynuacja poprzedniej strony)

```
    return np.random.rand()-0.5


def point_c():
    """
    return: array [x,y] with random point from a cirle
            centered at [0.5,0.5] and radius 0.4
            (used for examples)
    """
    while True:
        x=np.random.random()
        y=np.random.random()
        if (x-0.5)**2+(y-0.5)**2 < 0.4**2:
            break
    return np.array([x,y])


def point():
    """
    return: array [x,y] with random point from [0,1]x[0,1]
    """
    x=np.random.random()
    y=np.random.random()
    return np.array([x,y])


def set_ran_w(ar,s=1):
    """
    Set network weights randomly

    input:
    ar - array of numbers of nodes in subsequent layers [n_0, n_1,...,n_l]
    (from input layer 0 to output layer l, bias nodes not counted)

    s - scale factor: each weight is in the range [-0.s, 0.5s]

    return:
    w - dictionary of weights for neuron layers 1, 2,...,l in the format
    {1: array[n_0+1,n_1],...,l: array[n_(l-1)+1,n_l]}
    """
    l=len(ar)
    w={}
    for k in range(l-1):
        w.update({k+1: [[s*rn() for i in range(ar[k+1])] for j in range(ar[k]+1)]})
    return w


def set_val_w(ar,a=0):
    """
    Set network weights to a constant value

    input:
    ar - array of numbers of nodes in subsequent layers [n_0, n_1,...,n_l]
    (from input layer 0 to output layer l, bias nodes not counted)

    a - value for each weight

    return:
    w - dictionary of weights for neuron layers 1, 2,...,l in the format
    {1: array[n_0+1,n_1],...,l: array[n_(l-1)+1,n_l]}
    """
```

(ciąg dalszy na następnej stronie)

(kontynuacja poprzedniej strony)

```
    l=len(ar)
    w={}
    for k in range(l-1):
        w.update({k+1: [[a for i in range(ar[k+1])] for j in range(ar[k]+1)]})
    return w


def feed_forward(ar, we, x_in, ff=step):
    """
    Feed-forward propagation

    input:
    ar - array of numbers of nodes in subsequent layers [n_0, n_1,...,n_l]
    (from input layer 0 to output layer l, bias nodes not counted)

    we - dictionary of weights for neuron layers 1, 2,...,l in the format
    {1: array[n_0+1,n_1],...,l: array[n_(l-1)+1,n_l]}

    x_in - input vector of length n_0 (bias not included)

    ff - activation function (default: step)

    return:
    x - dictionary of signals leaving subsequent layers in the format
    {0: array[n_0+1],...,l-1: array[n_(l-1)+1], l: array[nl]}
    (the output layer carries no bias)
    """
    l=len(ar)-1                    # number of neuron layers
    x_in=np.insert(x_in,0,1)       # input, with the bias node inserted

    x={}                           # empty dictionary
    x.update({0: np.array(x_in)}) # add input signal

    for i in range(0,l-1):         # loop over layers till before last one
        s=np.dot(x[i],we[i+1])     # signal, matrix multiplication
        y=[ff(s[k]) for k in range(ar[i+1])] # output from activation
        x.update({i+1: np.insert(y,0,1)}) # add bias node and update x

    # the last layer - no adding of the bias node
    s=np.dot(x[l-1],we[l])
    y=[ff(s[q]) for q in range(ar[l])]
    x.update({l: y})           # update x

    return x


def back_prop(fe,la, p, ar, we, eps,f=sig, df=dsig):
    """
    back propagation algorithm

    fe - array of features
    la - array of labels
    p  - index of the used data point
    ar - array of numbers of nodes in subsequent layers
    we - dictionary of weights - UPDATED
    eps - learning speed
    f   - activation function
    df  - derivative of f
    """

    l=len(ar)-1 # number of neuron layers (= index of the output layer)
```

(ciąg dalszy na następnej stronie)

(kontynuacja poprzedniej strony)

```
    nl=ar[l]    # number of neurons in the otput layer

    x=feed_forward(ar,we,fe[p],ff=f) # feed-forward of point p

    # formulas from the derivation in a one-to-one notation:

    D={}
    D.update({l: [2*(x[l][gam]-la[p][gam])*
                  df(np.dot(x[l-1],we[l])[gam]) for gam in range(nl)]})
    we[l]-=eps*np.outer(x[l-1],D[l])

    for j in reversed(range(1,l)):
        u=np.delete(np.dot(we[j+1],D[j+1]),0)
        v=np.dot(x[j-1],we[j])
        D.update({j: [u[i]*df(v[i]) for i in range(len(u))]})
        we[j]-=eps*np.outer(x[j-1],D[j])


def feed_forward_o(ar, we, x_in, ff=sig, ffo=lin):
    """
    Feed-forward propagation with different output activation

    input:
    ar - array of numbers of nodes in subsequent layers [n_0, n_1,...,n_l]
    (from input layer 0 to output layer l, bias nodes not counted)

    we - dictionary of weights for neuron layers 1, 2,...,l in the format
    {1: array[n_0+1,n_1],...,l: array[n_(l-1)+1,n_l]}

    x_in - input vector of length n_0 (bias not included)

    f  - activation function (default: sigmoid)
    fo - activation function in the output layer (default: linear)

    return:
    x - dictionary of signals leaving subsequent layers in the format
    {0: array[n_0+1],...,l-1: array[n_(l-1)+1], l: array[nl]}
    (the output layer carries no bias)

    """
    l=len(ar)-1                    # number of neuron layers
    x_in=np.insert(x_in,0,1)       # input, with the bias node inserted

    x={}                           # empty dictionary
    x.update({0: np.array(x_in)}) # add input signal

    for i in range(0,l-1):         # loop over layers till before last one
        s=np.dot(x[i],we[i+1])     # signal, matrix multiplication
        y=[ff(s[k]) for k in range(ar[i+1])] # output from activation
        x.update({i+1: np.insert(y,0,1)}) # add bias node and update x

    # the last layer - no adding of the bias node
    s=np.dot(x[l-1],we[l])
    y=[ffo(s[q]) for q in range(ar[l])] # output activation function
    x.update({l: y})                   # update x

    return x


def back_prop_o(fe,la, p, ar, we, eps, f=sig, df=dsig, fo=lin, dfo=dlin):
    """
```

(ciąg dalszy na następnej stronie)

(kontynuacja poprzedniej strony)

```
    backprop with different output activation

    fe - array of features
    la - array of labels
    p  - index of the used data point
    ar - array of numbers of nodes in subsequent layers
    we - dictionary of weights - UPDATED
    eps - learning speed
    f   - activation function
    df  - derivative of f
    fo  - activation function in the output layer (default: linear)
    dfo - derivative of fo
    """
    l=len(ar)-1 # number of neuron layers (= index of the output layer)
    nl=ar[l]    # number of neurons in the otput layer

    x=feed_forward_o(ar,we,fe[p],ff=f,ffo=fo) # feed-forward of point p

    # formulas from the derivation in a one-to-one notation:

    D={}
    D.update({l: [2*(x[l][gam]-la[p][gam])*
                  dfo(np.dot(x[l-1],we[l])[gam]) for gam in range(nl)]})

    we[l]-=eps*np.outer(x[l-1],D[l])

    for j in reversed(range(1,l)):
        u=np.delete(np.dot(we[j+1],D[j+1]),0)
        v=np.dot(x[j-1],we[j])
        D.update({j: [u[i]*df(v[i]) for i in range(len(u))]})
        we[j]-=eps*np.outer(x[j-1],D[j])
```

13.2.2 Moduł draw.py

```
"""
Plotting functions used in the lecture.
"""

import numpy as np
import matplotlib.pyplot as plt


def plot(*args, title='activation function', x_label='signal', y_label='response',
         start=-2, stop=2, samples=100):
    """
    Wrapper on matplotlib.pyplot library.
    Plots functions passed as *args.
    Functions need to accept a single number argument and return a single number.
    Example usage:  plot(func.step,func.sig)
    """
    s = np.linspace(start, stop, samples)

    ff=plt.figure(figsize=(2.8,2.3),dpi=120)
    plt.title(title, fontsize=11)
    plt.xlabel(x_label, fontsize=11)
    plt.ylabel(y_label, fontsize=11)
```

(ciąg dalszy na następnej stronie)

(kontynuacja poprzedniej strony)

```
    for fun in args:
        data_to_plot = [fun(x) for x in s]
        plt.plot(s, data_to_plot)

    return ff;


def plot_net_simp(n_layer):
    """
    Draw the network architecture without bias nodes

    input: array of numbers of nodes in subsequent layers [n0, n1, n2,...]

    return: graphics object
    """
    l_layer=len(n_layer)
    ff=plt.figure(figsize=(4.3,2.3),dpi=120)

# input nodes
    for j in range(n_layer[0]):
            plt.scatter(0, j-n_layer[0]/2, s=50,c='black',zorder=10)

# neuron layer nodes
    for i in range(1,l_layer):
        for j in range(n_layer[i]):
            plt.scatter(i, j-n_layer[i]/2, s=100,c='blue',zorder=10)

# bias nodes
    for k in range(n_layer[l_layer-1]):
        plt.plot([l_layer-1,l_layer],[n_layer[l_layer-1]/2-1,n_layer[l_layer-1]/2-
 ↪1], s=50,c='gray',zorder=10)

# edges
    for i in range(l_layer-1):
        for j in range(n_layer[i]):
            for k in range(n_layer[i+1]):
                plt.plot([i,i+1],[j-n_layer[i]/2,k-n_layer[i+1]/2], c='gray')

    plt.axis("off")

    return ff;


def plot_net(ar):
    """
    Draw network with bias nodes

    input:
    ar - array of numbers of nodes in subsequent layers [n_0, n_1,...,n_l]
    (from input layer 0 to output layer l, bias nodes not counted)

    return: graphics object
    """
    l=len(ar)
    ff=plt.figure(figsize=(4.3,2.3),dpi=120)

# input nodes
    for j in range(ar[0]):
            plt.scatter(0, j-(ar[0]-1)/2, s=50,c='black',zorder=10)

# neuron layer nodes
```

(ciąg dalszy na następnej stronie)

(kontynuacja poprzedniej strony)

```
    for i in range(1,l):
        for j in range(ar[i]):
            plt.scatter(i, j-(ar[i]-1)/2, s=100,c='blue',zorder=10)

# bias nodes
    for i in range(l-1):
            plt.scatter(i, 0-(ar[i]+1)/2, s=50,c='gray',zorder=10)

# edges
    for i in range(l-1):
        for j in range(ar[i]+1):
            for k in range(ar[i+1]):
                plt.plot([i,i+1],[j-(ar[i]+1)/2,k+1-(ar[i+1]+1)/2],c='gray')

# the last edge on the right
    for j in range(ar[l-1]):
        plt.plot([l-1,l-1+0.7],[j-(ar[l-1]-1)/2,j-(ar[l-1]-1)/2],c='gray')

    plt.axis("off")

    return ff;


def plot_net_w(ar,we,wid=1):
    """
    Draw the network architecture with weights

    input:
    ar - array of numbers of nodes in subsequent layers [n_0, n_1,...,n_l]
    (from input layer 0 to output layer l, bias nodes not counted)

    we - dictionary of weights for neuron layers 1, 2,...,l in the format
    {1: array[n_0+1,n_1],...,l: array[n_(l-1)+1,n_l]}

    wid - controls the width of the lines

    return: graphics object
    """
    l=len(ar)
    ff=plt.figure(figsize=(4.3,2.3),dpi=120)

# input nodes
    for j in range(ar[0]):
            plt.scatter(0, j-(ar[0]-1)/2, s=50,c='black',zorder=10)

# neuron layer nodes
    for i in range(1,l):
        for j in range(ar[i]):
            plt.scatter(i, j-(ar[i]-1)/2, s=100,c='blue',zorder=10)

# bias nodes
    for i in range(l-1):
            plt.scatter(i, 0-(ar[i]+1)/2, s=50,c='gray',zorder=10)

# edges
    for i in range(l-1):
        for j in range(ar[i]+1):
            for k in range(ar[i+1]):
                th=wid*we[i+1][j][k]
                if th>0:
                    col='red'
```

(ciąg dalszy na następnej stronie)

(kontynuacja poprzedniej strony)

```
                else:
                    col='blue'
                th=abs(th)
                plt.plot([i,i+1],[j-(ar[i]+1)/2,k+1-(ar[i+1]+1)/2],c=col,
 ↪linewidth=th)

# the last edge on the right
    for j in range(ar[l-1]):
        plt.plot([l-1,l-1+0.7],[j-(ar[l-1]-1)/2,j-(ar[l-1]-1)/2],c='gray')

    plt.axis("off")

    return ff;


def plot_net_w_x(ar,we,wid,x):
    """
    Draw the network architecture with weights and signals

    input:
    ar - array of numbers of nodes in subsequent layers [n_0, n_1,...,n_l]
    (from input layer 0 to output layer l, bias nodes not counted)

    we - dictionary of weights for neuron layers 1, 2,...,l in the format
    {1: array[n_0+1,n_1],...,l: array[n_(l-1)+1,n_l]}

    wid - controls the width of the lines

    x - dictionary the the signal in the format
    {0: array[n_0+1],...,l-1: array[n_(l-1)+1], l: array[nl]}

    return: graphics object
    """
    l=len(ar)
    ff=plt.figure(figsize=(4.3,2.3),dpi=120)

# input layer
    for j in range(ar[0]):
            plt.scatter(0, j-(ar[0]-1)/2, s=50,c='black',zorder=10)
            lab=np.round(x[0][j+1],3)
            plt.text(-0.27, j-(ar[0]-1)/2+0.1, lab, fontsize=7)

# intermediate layer
    for i in range(1,l-1):
        for j in range(ar[i]):
            plt.scatter(i, j-(ar[i]-1)/2, s=100,c='blue',zorder=10)
            lab=np.round(x[i][j+1],3)
            plt.text(i+0.1, j-(ar[i]-1)/2+0.1, lab, fontsize=7)

# output layer
    for j in range(ar[l-1]):
        plt.scatter(l-1, j-(ar[l-1]-1)/2, s=100,c='blue',zorder=10)
        lab=np.round(x[l-1][j],3)
        plt.text(l-1+0.1, j-(ar[l-1]-1)/2+0.1, lab, fontsize=7)

# bias nodes
    for i in range(l-1):
            plt.scatter(i, 0-(ar[i]+1)/2, s=50,c='gray',zorder=10)

# edges
    for i in range(l-1):
```

(ciąg dalszy na następnej stronie)

(kontynuacja poprzedniej strony)

```
        for j in range(ar[i]+1):
            for k in range(ar[i+1]):
                th=wid*we[i+1][j][k]
                if th>0:
                    col='red'
                else:
                    col='blue'
                th=abs(th)
                plt.plot([i,i+1],[j-(ar[i]+1)/2,k+1-(ar[i+1]+1)/2],c=col,
 ↪linewidth=th)

# the last edge on the right
    for j in range(ar[l-1]):
        plt.plot([l-1,l-1+0.7],[j-(ar[l-1]-1)/2,j-(ar[l-1]-1)/2],c='gray')

    plt.axis("off")

    return ff;


def l2(w0,w1,w2):
    """for separating line"""
    return [-.1,1.1],[-(w0-w1*0.1)/w2,-(w0+w1*1.1)/w2]
```

13.3 Jak cytować

Jeśli chcesz zacytować tę książkę Jupyter Book, oto wpis w formacie BibTeX do wersji angielskiej:

```
@book{WB2021,
  title={"Explaining neural networks in raw Python: lectures in Jupiter"},
  author={Wojciech Broniowski},
  isbn={978-83-962099-0-0},
  year={2021},
  url={https://ifj.edu.pl/strony/~broniows/nn}
  publisher={Wojciech Broniowski}
}
```

Bibliografia

[Bar16] P. Barry. *Head First Python: A Brain-Friendly Guide*. O'Reilly Media, 2016. ISBN 9781491919491. URL: https://books.google.pl/books?id=NIqNDQAAQBAJ.

[BH69] A. E. Bryson and Y.-C. Ho. *Applied optimal control: Optimization, estimation, and control*. Waltham, Mass: Blaisdell Pub. Co., 1969.

[FR13] J. Feldman and R. Rojas. *Neural Networks: A Systematic Introduction*. Springer Berlin Heidelberg, 2013. ISBN 9783642610684.

[Fre93] James A. Freeman. *Simulating Neural Networks with Mathematica*. Addison-Wesley Professional, 1993. ISBN 9780201566291.

[FS91] James A. Freeman and David M. Skapura. *Neural Networks: Algorithms, Applications, and Programming Techniques (Computation and Neural Systems Series)*. Addison-Wesley, 1991. ISBN 9780201513769.

[Gut16] John Guttag. *Introduction to computation and programming using Python: With application to understanding data*. MIT Press, 2016.

[HR72] K. K. Hartline and F. Ratcliff. Inhibitory interactions in the retina of limulus. *Handbook of sensory physiology*, VII(2):381–447, 1972.

[KSJ+12] E.R. Kandel, J.H. Schwartz, T.M. Jessell, S.A. Siegelbaum, and A.J. Hudspeth. *Principles of Neural Science, Fifth Edition*. McGraw-Hill Education, 2012. ISBN 9780071810012. URL: https://books.google.pl/books?id=Z2yVUTnlIQsC.

[Mat19] E. Matthes. *Python Crash Course, 2nd Edition: A Hands-On, Project-Based Introduction to Programming*. No Starch Press, 2019. ISBN 9781593279295. URL: https://books.google.pl/books?id=boBxDwAAQBAJ.

[MP43] Warren S. McCulloch and Walter Pitts. The logical calculus of the ideas immanent in nervous activity. *The bulletin of mathematical biophysics*, 5(4):115–133, 1943. URL: https://doi.org/10.1007/BF02478259.

[MullerRS12] B. Müller, J. Reinhardt, and M.T. Strickland. *Neural Networks: An Introduction*. Physics of Neural Networks. Springer Berlin Heidelberg, 2012. ISBN 9783642577604. URL: https://books.google.pl/books?id=on0QBwAAQBAJ.

[RIV91] A. K. Rigler, J. M. Irvine, and Thomas P. Vogl. Rescaling of variables in back propagation learning. *Neural Networks*, 4(2):225–229, 1991. URL: https://doi.org/10.1016/0893-6080(91)90006-Q, doi:10.1016/0893-6080(91)90006-Q.