# Podstawy Sztucznej Inteligencji

Mikołaj Stepniewski

# 6. Uczenie sieci regułą Kohonena (WTM)

## Cel ćwiczenia

Celem ćwiczenia jest poznanie budowy i działania sieci Kohonena przy wykorzystaniu reguły WTM do odwzorowywania istotnych cech liter alfabetu.

## Opis sieci

Algorytm Kohonena jest jedną z najstarszych metod uczenia sieci samoorganizujących typu WTA i WTM i w chwili obecnej istnieje wiele jego odmian.
W tym przypadku rozważamy grupę neuronów, z których każdy ma przyporządkowane swoje miejsce w przestrzeni i gdzie uaktualniamy tylko wagi zwycięzcy. W odróżnieniu od grupy neuronów Hebba w przypadku Kohonena wyłaniamy zwycięzce na podstawie najmniejszej odległości wektora wag neuronu i wektora wejściowego (do którego neuron ma być przyporządkowany), jest to tzw. miara euklidesowa:

$$d(x, w_i) = \| x - w_i \| = \sqrt{\sum_{j=1}^{N} (x_j - w_{ij})^2}$$

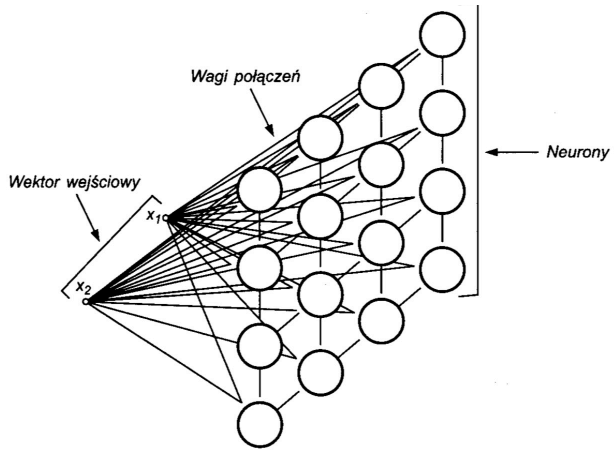
Neuron zwycięzca podlega adaptacji, zmieniając swój wektor wag w kierunku wektora $x$. Jego wagi uaktualniane są na podstawie poniższej reguły, gdzie $\eta$ - współczynnik uczenia, $G(i, x)$ - funkcja sąsiedztwa.

$$w_{k+1} = w_k + \eta G(i, x)(x - w_k)$$

Wykazano, że proces samoorganizacji prowadzi zawsze do spójnego podziału przestrzeni danych, gdy chociaż jeden z wektorów $x$ lub $w$ podlega normalizacji. Przy znormalizowanych wektorach uczących $x$, wektory wag - nadążając za nimi - stają się automatycznie znormalizowane.

W metodzie samoorganizacji neuronów Kohonena bardzo ważna jest normalizacja danych wejściowych, która obywa się zgodnie z formułą:

$$x_i = \frac{x_i}{\sqrt{\sum_{i=1}^{N} x_i^2}}$$




**Rys. 10.1.** Struktura sieci samoorganizującej Kohonena

**Algorytm WTM (Winner Takes Most):**

Oprócz algorytmu WTA stosuje się, także algorytm WTM, w którym oprócz zwycięzcy uaktualnia się również wagi neuronów z jego sąsiedztwa. Przy czym im dalsza jest odległość od neuronu zwycięzcy tym mniejsza zmiana wartości wag tego neuronu. Proces adaptacji wag może być ogólnie zapisany za pomocą zależności:

$$w(k+1) = w(k) + \eta G(i,x)[x - w_i]$$

dla wszystkich neuronów $i$ należących do sąsiedztwa. Odległość od wektora zwycięzcy może być opisana na dwa sposoby za pomocą funkcji sąsiedztwa $G(i,x)$:

$$G(i,\mathbf{x}) = \begin{cases} 1 & \text{dla } i = w \\ 0 & \text{dla } i \neq w \end{cases}$$

gdzie w to numer zwycięzcy - otrzymujemy klasyczne WTA. Definiując natomiast funkcje sąsiedztwa w postaci:

$$G(i,\mathbf{x}) = \begin{cases} 1 & \text{dla } d(i,w) \leqslant \lambda \\ 0 & \text{dla pozostałych} \end{cases}$$

gdzie $\lambda$ oznacza promień sąsiedztwa, który powinien maleć wraz z postępem uczenia. Tego typu sąsiedztwo nosi nazwę **sąsiedztwa prostokątnego**.

Drugim sposobem definiowania sąsiedztwa w algorytmie WTM jest funkcja sąsiedztwa **gaussowskiego** postaci:

$$G(i,x) = \exp(-\frac{d^2(i,w)}{2\lambda^2})$$

W tym przypadku o stopniu adaptacji neuronów decyduje nie tylko odległość euklidesowa $d(i,w)$ neuronu $i$-tego od zwycięzcy (neuronu $w$-tego), ale również promień sąsiedztwa $\lambda$. W odróżnieniu od sąsiedztwa prostokątnego, gdzie każdy neuron należący do sąsiedztwa podlegał adaptacji w jednakowym stopniu, tak w sąsiedztwie typu gaussowskiego stopień adaptacji jest zależny od wartości funkcji Gaussa. Dla zwycięzcy mamy $G(w,w) = 1$, a dla pozostałych neuronów w siatce mamy $0 < G(i,w) < 1$. Sąsiedztwo typu gaussowskiego prowadzi zwykle do lepszych rezultatów uczenia i lepszej organizacji sieci niż sąsiedztwo typu prostokątnego.

W celu uzyskania dobrych rezultatów samoorganizacji powinien rozpoczynać się duża wartością promienia $\lambda$, po czym powinna ona maleć wraz z upływem czasu, aż do wartości minimalnej, najczęściej zerowej. W opracowaniu S. Osowskiego zaproponowano następującą funkcje, gdzie $k$ - numer epoki, $\lambda_{min}$ - wartość minimalna promienia, $\lambda_{max}$ - promień startowy:

$$\lambda(k) = \lambda_{max}(\frac{\lambda_{min}}{\lambda_{max}})^{k/k_{max}}$$

## Dane wejściowe

Do procesu uczenia użyłem **26** liter alfabetu, reprezentowanego przez tablice **5x7** zer i jedynek. Dane wejściowe nie musiały zatem być dodatkowo normalizowane. Dla procesu testowania użyłem tych samych liter, jednak dodatkowo zaszumionych daną ilością "pikseli".

Dla przykładu, litery "A" oraz "M" przedstawiane są w następujący sposób:

0 **1 1 1** 0
**1** 0 0 0 **1**
**1** 0 0 0 **1**
**1 1 1 1 1**
**1** 0 0 0 **1**
**1** 0 0 0 **1**
**1** 0 0 0 **1**

**1** 0 0 0 **1**
**1 1** 0 **1 1**
**1** 0 **1** 0 **1**
**1** 0 0 0 **1**
**1** 0 0 0 **1**
**1** 0 0 0 **1**
**1** 0 0 0 **1**

Zaszumianie polegało na wybraniu zestawu niepowtarzających się numerów indeksu od 0 do 35 i zamienienie elementu pod każdym z tych numerów na liczbę przeciwną $0 \rightarrow 1$ oraz $1 \rightarrow 0$.

## Uczenie

Przeprowadziłem testy dla następujących parametrów:

**for lRate in [0.9, 0.5, 0.1, 0.05, 0.01]:**
**for radiusMax in [20, 10, 5, 4, 3, 2, 1]:**

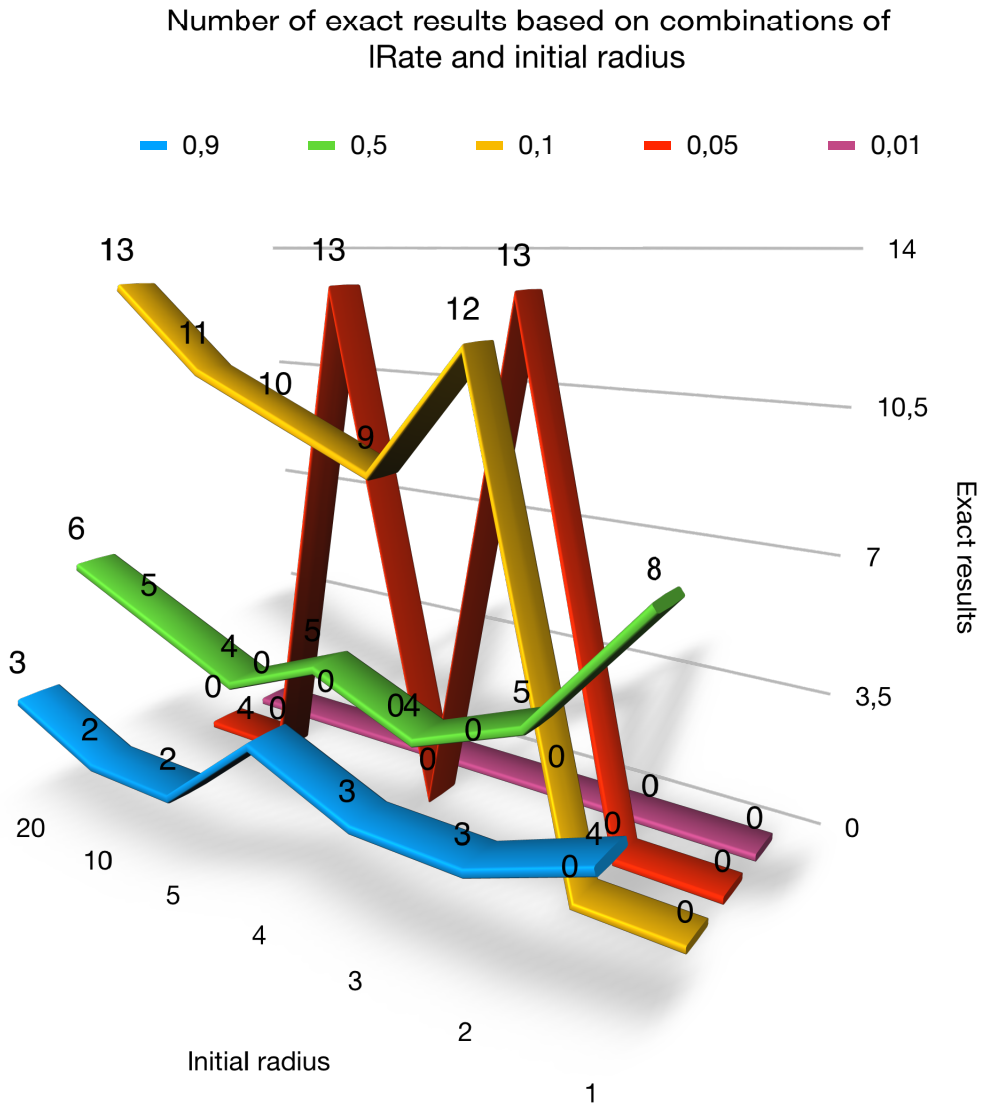
| learning rate | radius start |
|---|---|
| 0,9 | 20 |
| 0,5 | 10 |
| 0,1 | 5 |
| 0,05 | 4 |
| 0,01 | 3 |
| | 2 |
| | 1 |

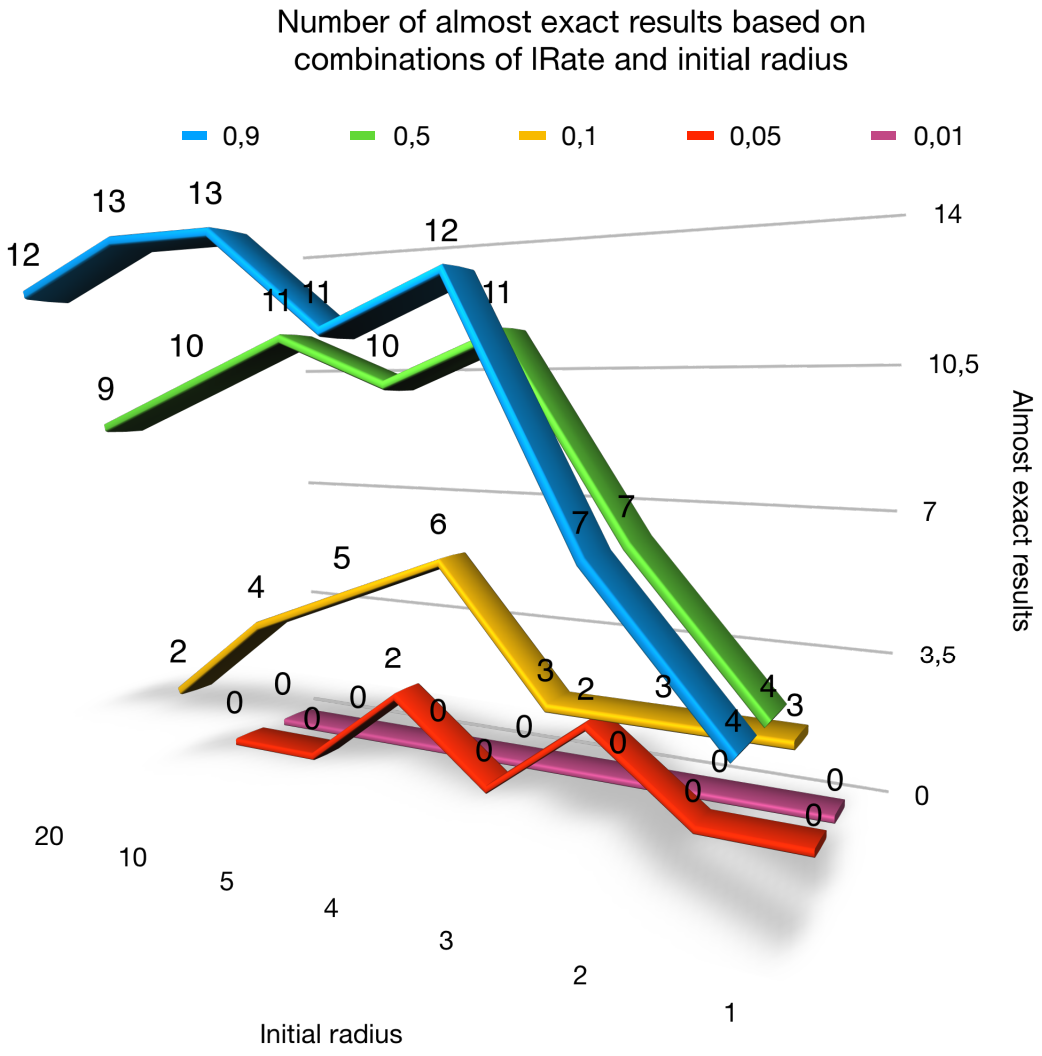
Po przeprowadzeniu serii uczeń (po każdym resetując wagi) przy ilości epok: 50 oraz promieniu minimalnym 0,5, oraz zestawie **15-stu** zaszumionych liter testowych:

**‘U’,'N','C','O','P','Y','R','I','G','H','T','A','B','L','E'**

otrzymałem wyniki składające się z
1. ilości dokładnych dopasowań danych testowych
2. ilości dopasowań danych testowych w sąsiedztwie (w obrębie promienia startowego)
3. ilości aktywnych neuronów (dla 26 liter danych treningowych)

Na ich podstawie opracowałem wykres 3D zależności kombinacji współczynnika uczenia i promienia startowego od ilości dopasowań dokładnych oraz ilości dopasowań w sąsiedztwie:





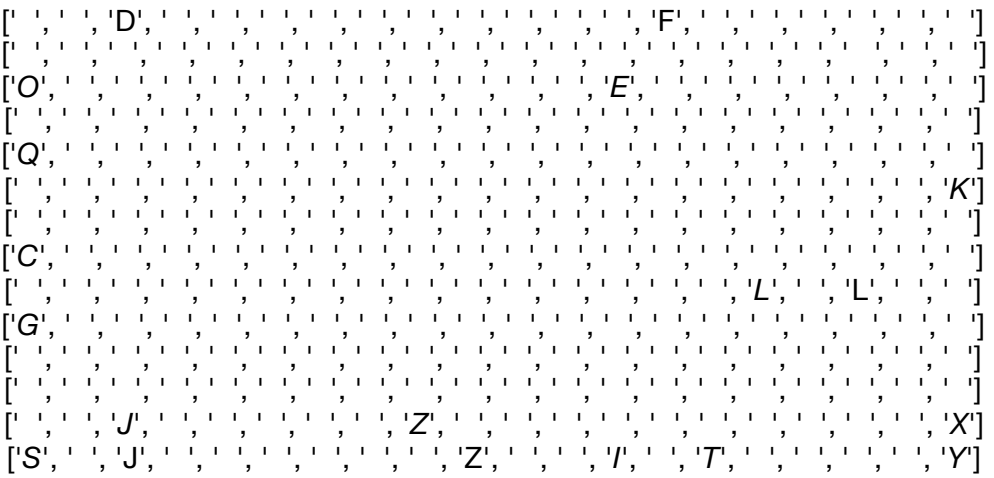

**Gdy sieć po procesie uczenia była podzielona na mniej niż 25 aktywne neurony, wówczas wynik określałem jako 0.**

Z wykresów wynika, że najlepszą kombinacją jest lRate = 0.1 i radius start = 20, lecz trzeba zauważyć, że podczas testu dla 50 epok sieć podzieliła litery na 25 neuronów, a nie na 26. Zatem najlepszą kombinacją, gdzie dodatkowo mamy podział na 26 neuronów to lRate = 0.1 i radius start = 10.

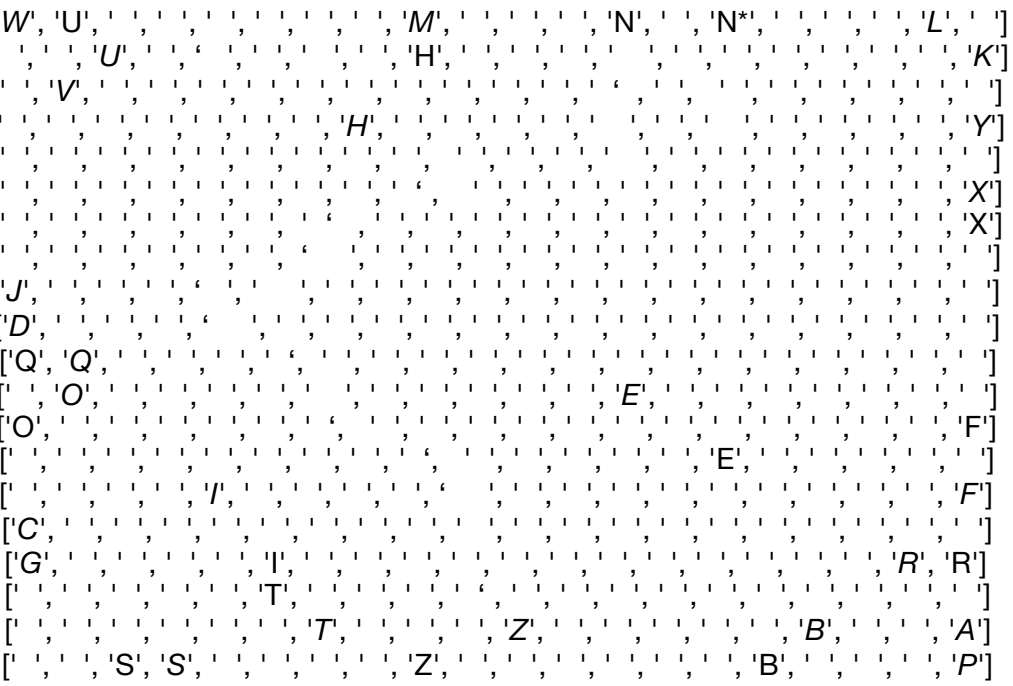
| Najlepsze parametry | | | |
|---|---|---|---|
| **learning rate** | **radius start** | **exact results** | **results in neighbourhood** |
| **0,1** | **10** | **11** | **4** |

Mapa liter dla najlepszej kombinacji w zależności od ilości zaszumionych pikseli:
**epochs = 100**
**noise pixels = 2:** **(exact: 17, almost-exact: 9, wrong: 0)**

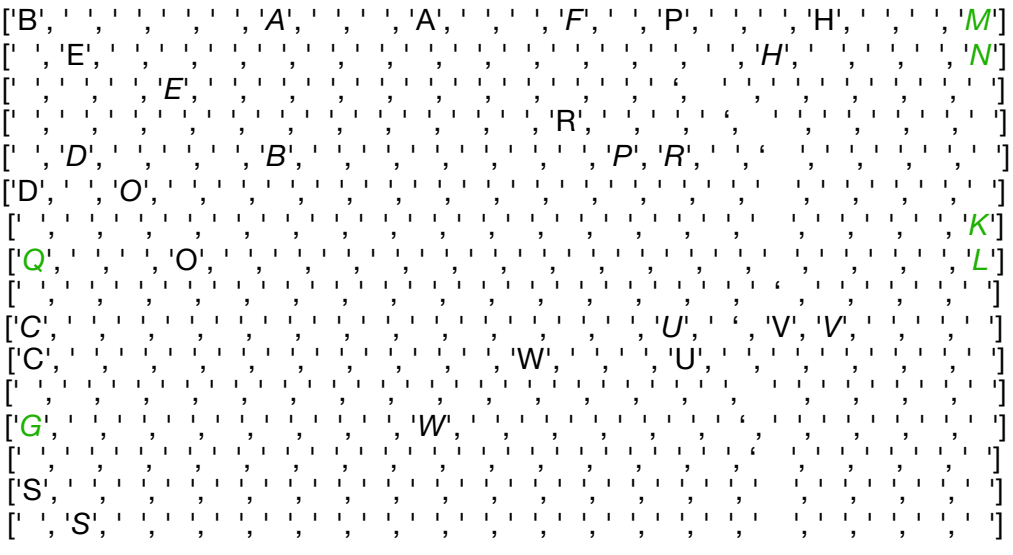
['*W*', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', '*B*', ' ', 'P', ' ', 'R', '*R*', ' ', ' ', ' ', '*A*']
['*U*', ' ', ' ', '*V*', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', '*P*', ' ', ' ', ' ', ' ', ' ', ' ', ' ', '*M*', ' ']
[' ', ' ', ' ', 'V', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', 'N', ' ', ' ', ' ']
[' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', '*N*', ' ', '*H*']
[' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ']
[' ', ' ', ' ', '*D*', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', '*F*', ' ', ' ', ' ', ' ', ' ', ' ']



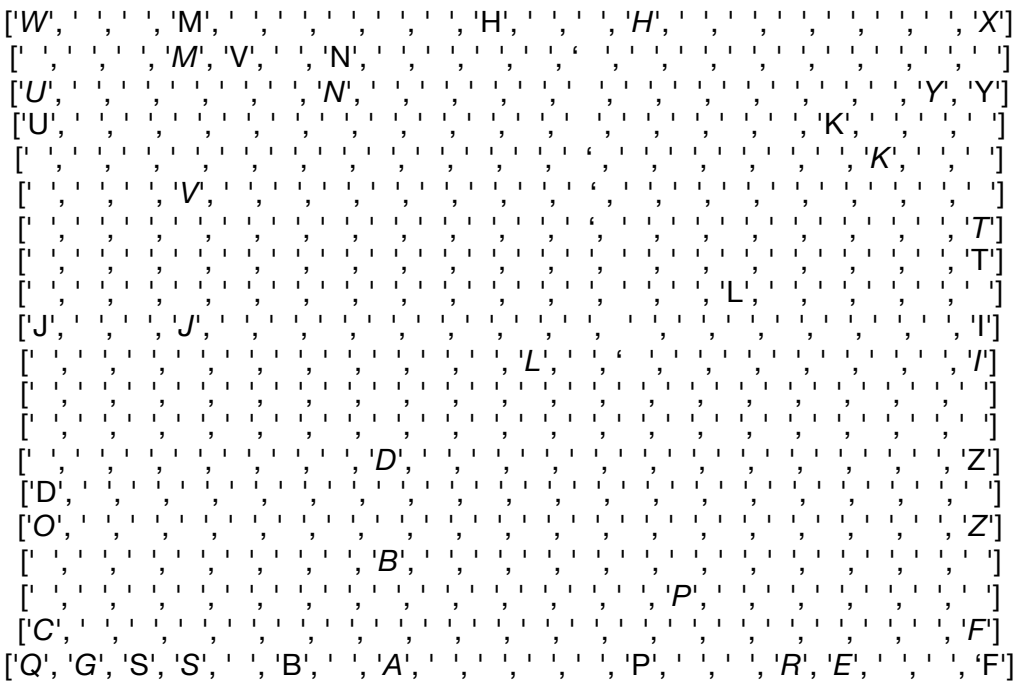
**noise pixels = 3:** **(exact: 12, almost-exact: 14, wrong: 0)**



**noise pixels = 4:** **(exact: 9, almost-exact: 17, wrong: 0)**



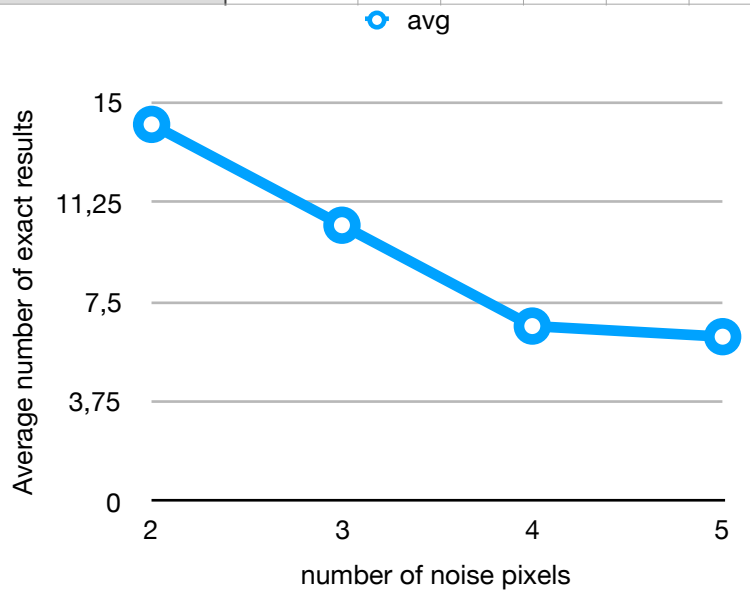
**noise pixels = 5:** **(exact: 9, almost-exact: 17, wrong: 0)**



Dla ilości pikseli zaszumienia **2,3,4,5** zrobiłem po **5** prób i zapisałem wyniki w tabelach:

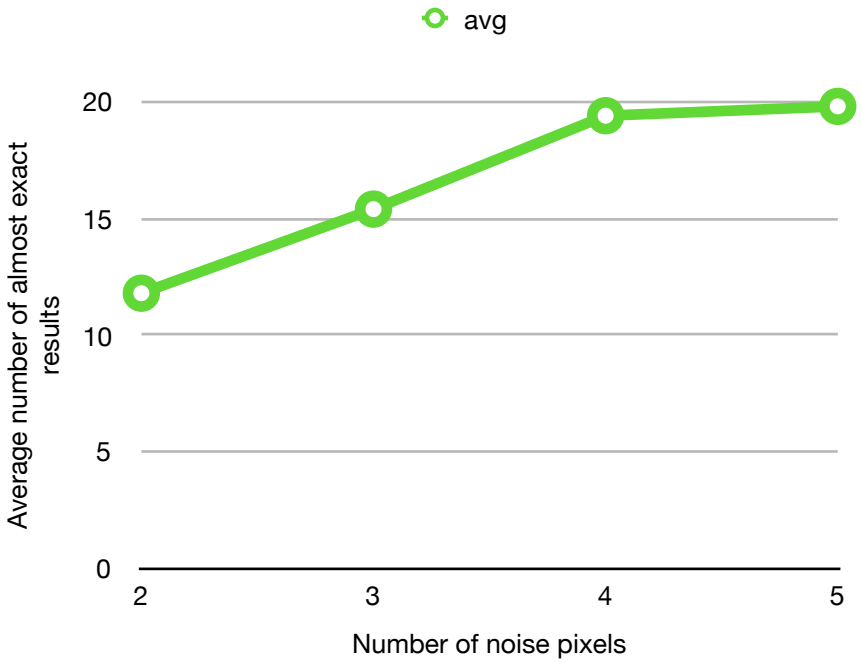
Exact results based on number of noise pixels

| | Tries | | | | | |
|---|---|---|---|---|---|---|
| **Noise pixels** | **1** | **2** | **3** | **4** | **5** | **avg** |
| **2** | 13 | 17 | 16 | 12 | 13 | **14,2** |
| **3** | 11 | 11 | 10 | 12 | 8 | **10,4** |
| **4** | 9 | 6 | 8 | 6 | 4 | **6,6** |
| **5** | 7 | 3 | 6 | 9 | 6 | **6,2** |



Almost exact results based on number of noise pixels

| | Tries | | | | | |
|---|---|---|---|---|---|---|
| **Noise pixels** | **1** | **2** | **3** | **4** | **5** | **avg** |
| **2** | 13 | 9 | 10 | 14 | 13 | **11,8** |
| **3** | 15 | 15 | 15 | 14 | 18 | **15,4** |
| **4** | 17 | 20 | 18 | 20 | 22 | **19,4** |
| **5** | 19 | 23 | 20 | 17 | 20 | **19,8** |




Wrong results

| | Tries | | | | |
|---|---|---|---|---|---|
| **Noise pixels** | **1** | **2** | **3** | **4** | **5** |
| **2** | 0 | 0 | 0 | 0 | 0 |
| **3** | 0 | 0 | **1** | 0 | 0 |
| **4** | 0 | 0 | 0 | 0 | 0 |
| **5** | 0 | 0 | 0 | 0 | 0 |

---

## Analiza wyników

1. Najwięcej dokładnych wyników uzyskano dla kombinacji lRate = 0.01 i radius start = 20, lecz w tym przypadku najgorsza była organizacja sieci. Podobnie dla lRate 0.05.

2. Pomijając lRate 0.01 i 0.05 najwięcej dokładnych wyników uzyskano dla lRate 0.1 i radius start = 20, lecz organizacja nie była idealna (25/26).
3. Zatem najlepsza kombinacja to lRate = 0.1 i radius start 10.
4. Dla wysokich lRate dużo więcej jest wyników niedokładnych niż dokładnych.
5. Stosunek wyników niedokładnych do dokładnych rośnie wprost proporcjonalnie do współczynnika uczenia.
6. Im większy radius start tym więcej dokładnych wyników dla małych współczynników uczenia
7. Im większy radius start tym więcej niedokładnych wyników dla dużych współczynników uczenia.
8. Im więcej pikseli zaszumienia tym mniej dokładnych wyników - tym bardziej litery testowe oddalone są od oryginalnych.
9. Im więcej pikseli zaszumienia tym więcej niedokładnych wyników.
10. Litery B, P oraz M, N, oraz W, U, V oraz F, E oraz O, Q oraz C, G, oraz X, Y oraz I, T pojawiają się na mapie w niedalekiej od siebie odległości.

---

## Wnioski

1. Małe współczynniki uczenia sprawiają, że sieć uczy się znacznie wolniej a promień i tak spada z epoki na epokę - zatem potrzeba znacznie więcej epok, ażeby optymalnie wyuczyć sieć
2. Dla promieni uczenia bardzo dużych (np. 20 co jest wielkością siatki) częściej zdarza się, że nie uzyskamy dopasowań 26 różnych neuronów do 26 różnych liter alfabetu ponieważ zwiększone jest prawdopodobieństwo, że zbyt wiele neuronów zacznie dopasowywanie do kolejnych liter w uczeniu.
3. Optymalną kombinacją jest zatem średniej wielkości współczynnik uczenia i średniej wielkości promień startowy (lRate = 0.1, radius start = 10)
4. Gdy lRate jest wysokie neurony z sąsiedztwa zbyt szybko dopasowują się do różnych liter i przy prezentacji litery wygrywa zazwyczaj neuron pomiędzy.
5. Dla małych współczynników uczenia duży radius działa na plus, gdyż neurony uczą się na tyle wolno, że duża ilość neuronów w sąsiedztwie nie powoduje problemu z punktu 2. Odwrotnie jest dla dużych współczynników.
6. Zaszumienie liter powoduje, że mogą zbliżać się wyglądem do innych liter (zazwyczaj tych już w oryginale podobnych) co sprawia, że przemieszczają się na mapie w ich kierunku. Stąd większa ilość niedokładnych wyników wraz z rosnącą ilością pikseli zaszumienia.
7. Litery wymienione w 10. punkcie analizy są blisko siebie, ponieważ są do siebie podobne kształtem.

---

## Listing

main.py:

```
from math import ceil
from math import floor
from neurons import *
from data import *
from progressBar import *
import random
import copy
from prettytable import PrettyTable
```

```
def countUniqueItems(arr):
    return len(Counter(arr).keys())

def getMostCommonItem(arr):
    return Counter(arr).most_common(1)[0][0]


if __name__ == "__main__":

    """ Training parameters """
    epochs = 100
    decay = 0.1*(epochs)*13000
    neuronGrid = (20, 20)
    lRate = 0.1    # 0.07 one of the best
    neighbourhoodRadius = 10
    neighbourhoodRadiusMin = 0.5
    noNoisePixels = 2
    assignmentMap = [[' ' for _ in range(neuronGrid[1])] for _ in range(neuronGrid[0])]
    resultsInNeighbourhood = 0
    resultsExact = 0
    resultsWrong = 0

    """ Getting training and test data """
    trainingData = Data()._letters
    testData =
Data().getNoisedLetters(['A','B','C','D','E','F','G','H','I','J','K','L','M','N','O','P','Q','R','S','T','U','V'
,'W','X','Y', 'Z'], noNoisePixels)
    numOfInputs = len(trainingData['A'])

    """ Creating kohonen group with designated parameters """
    kohonenGroup = KohonenNeuronGroup(
        numOfInputs=numOfInputs,
        numOfNeurons=neuronGrid,
        processFunc=euklidesDistance,
        lRateFunc=Linear()(),
        neighbourhoodRadius=neighbourhoodRadius,
        nRadiusFunc=neighbourhoodRadiusCorrection(neighbourhoodRadius,
neighbourhoodRadiusMin, epochs),
        lRate=lRate
    )

    paramsTable = PrettyTable()
    paramsTable.field_names = ['lRate', 'RadiusMax', 'RadiusMin', 'neurons', 'epochs']
    paramsTable.add_row([kohonenGroup._lRate, neighbourhoodRadius,
neighbourhoodRadiusMin, kohonenGroup['totalNumOfNeurons'], epochs])
    print(paramsTable)

    """ Process of learning """
    print('\nrunning {:d} epochs...'.format(epochs))
    winners = {}
    pbar = ProgressBar()
    pbar.start(maxVal=epochs)
    for i in range(epochs):
        for key, value in trainingData.items():
            winners[key] = kohonenGroup.train(value)
        kohonenGroup.setNeighbourhoodRadius()
        pbar.update()

    """ Process of testing """
```

```
    testWinners = {}
    for key, value in testData.items():
        testWinners[key] = kohonenGroup.classify(value)


    """ Printing results, comparision tables and map """
    trainingTable = PrettyTable()
    trainingTable.field_names = ['Letter', 'Neuron id', 'x', 'y', 'Neuron id*', 'x*', 'y*']
    for key, neuron in winners.items():
        testNeuron = testWinners.get(key, None)
        testNID = '' if not testNeuron else testNeuron._iid
        testX = '' if not testNeuron else testNeuron._x
        testY = '' if not testNeuron else testNeuron._y
        trainingTable.add_row([key, neuron._iid, neuron._x, neuron._y, testNID, testX, testY])
    uniqueNeurons = countUniqueItems(winners.values())
    print('\nActive neurons', uniqueNeurons)
    print('Number of letters', len(trainingData))
    uniqueNeurons = countUniqueItems(testWinners.values())
    print('\nActive test neurons', uniqueNeurons)
    print('Number of test letters', len(testData), '\n')
    print(trainingTable)
    print('* - testing letters noised with {:d} pixels.\n'.format(noNoisePixels))

    for key, value in testWinners.items():
        dist = distance((value._x, value._y), (winners[key]._x, winners[key]._y))
        if dist <= neighbourhoodRadius and dist != 0:
            resultsInNeighbourhood+=1
        elif dist == 0:
            resultsExact += 1
    resultsWrong = len(testData)-(resultsExact+resultsInNeighbourhood)


    for key, value in winners.items():
        assignmentMap[value._x][value._y] = key

    for key, value in testWinners.items():
        assignmentMap[value._x][value._y] = key+'*'

    for row in assignmentMap:
        print(row)

    print('exact: {:d}\talmost: {:d}\twrong: {:d}'.format(resultsExact, resultsInNeighbourhood,
resultsWrong))
```

neurons.py:

```
# @Author: Mikołaj Stępniewski <maikelSoFly>
# @Date:   2017-12-12T17:15:51+01:00
# @Email:  mikolaj.stepniewski1@gmail.com
# @Filename: neurons.py
# @Last modified by:   maikelSoFly
# @Last modified time: 2017-12-16T23:31:39+01:00
# @License: Apache License  Version 2.0, January 2004
# @Copyright: Copyright © 2017 Mikołaj Stępniewski. All rights reserved.



import sys
# Add the include folder path to the sys.path list
```

```
sys.path.append('../include')
from neuron import *
from supportFunctions import *
from collections import Counter
import copy


class KohonenNeuron(Neuron):
    def __init__(self, x, y, numOfInputs, processFunc, iid, lRate=0.1):
        Neuron.__init__(self, numOfInputs, iid, activFunc=None, lRate=lRate, bias=0)
        self.__dict__['_winnerCounter'] = 0
        self.__dict__['_processFunc'] = processFunc
        self.__dict__['_startWeights'] = copy.deepcopy(self._weights)
        self.__dict__['_errorHist'] = []
        self.__dict__['_iid'] = iid
        self.__dict__['_x'] = x
        self.__dict__['_y'] = y


    def process(self, vector):
        """ Euklides distance between input vector and weights vector """
        self._error = self._processFunc(vector, self._weights)
        return self._error


    def train(self, vector, G):
        """ WTM train with magnitude based on neighbourhood function """
        const = self._lRate * G
        for i in range(len(self._weights)):
            self._weights[i] +=  const * (vector[i] - self._weights[i])


    def resetWeights(self):
        self._weights = self._startWeights[:]




""" Simple WTA for now...

    Winner is neuron with least distance
    between weights vector and input vector """
class KohonenNeuronGroup:
    def __init__(self, numOfInputs, numOfNeurons, processFunc, lRateFunc, neighbourhoodRadius,
nRadiusFunc, lRate=0.1):
        self.__dict__['_numOfNeurons'] = numOfNeurons
        self.__dict__['_lRate'] = lRate
        self.__dict__['_numOfInputs'] = numOfInputs
        self.__dict__['_neurons'] = None
        self.__dict__['_processFunc'] = processFunc
        self.__dict__['_lRateFunc'] = lRateFunc
        self.__dict__['_currentLRate'] = None
        self.__dict__['_neighbourhoodRadius'] = neighbourhoodRadius
        self.__dict__['_neighbourhoodRadiusFunc'] = nRadiusFunc

        self._neurons = [[KohonenNeuron(j, i, numOfInputs, processFunc, iid=i*numOfNeurons[0]+j,
lRate=lRate)
            for i in range(numOfNeurons[0])]
            for j in range(numOfNeurons[1])
        ]
```

```
""" Reset training parameters and reset weights to initial ones """
def resetGroup(self, lRate, radius, radiusFunc):
    self._lRate = lRate
    self._neighbourhoodRadius = radius
    self._neighbourhoodRadiusFunc = radiusFunc
    for row in self._neurons:
        for neuron in row:
            neuron._weights = copy.deepcopy(neuron._startWeights)


def resetWins(self):
    for row in self._neurons:
        for neuron in row:
            neuron._winnerCounter = 0


def setLRate(self, lRate):
    self._currentLRate = lRate
    for row in self._neurons:
        for neuron in row:
            neuron._lRate = lRate


def setNeighbourhoodRadius(self):
    self._neighbourhoodRadius = self._neighbourhoodRadiusFunc(self._neighbourhoodRadius)


def trainNeighbours(self, vector, winner):
    coordsW = (winner._x, winner._y)
    for row in self._neurons:
        for neuron in row:
            neuron.train(vector, self.gaussNeighbourhood(coordsW, [neuron._x, neuron._y]))


def gaussNeighbourhood(self, coordsW, coordsI):
    dx = (coordsW[0]-coordsI[0])
    dy = (coordsW[1]-coordsI[1])
    dist = dx*dx + dy*dy
    return np.exp(-dist/(2*self._neighbourhoodRadius**2))


def train(self, vector):
    winner = None
    for row in self._neurons:
        for neuron in row:
            neuron.process(vector)
            if winner == None:
                winner = neuron
            elif winner != None:
                if neuron._error < winner._error:
                    winner = neuron

    """ Winner Takes MOST """
    """ Updating weights with magnitude based on neighbourhood func """
    winner.train(vector, 1)
    self.trainNeighbours(vector, winner)
    self.setLRate(self._lRateFunc(self._lRate))

    return winner
```

```
    """ Basicaly the same as above, but without updating weights """
    def classify(self, vector):
        winner = None
        for row in self._neurons:
            for neuron in row:
                neuron.process(vector)
                if winner == None:
                    winner = neuron
                elif winner != None:
                    if neuron._error < winner._error:
                        winner = neuron

        return winner


    """ Access methods """
    def __getitem__(self, key):
        if key == 'totalNumOfNeurons':
            return sum(len(x) for x in self._neurons)


tests.py:

from math import ceil
from math import floor
from neurons import *
from data import *
from progressBar import *
import random
import copy
from prettytable import PrettyTable


def countUniqueItems(arr):
    return len(Counter(arr).keys())

def getMostCommonItem(arr):
    return Counter(arr).most_common(1)[0][0]


if __name__ == "__main__":

    """ Training parameters """
    epochs = 50
    decay = 0.1*(epochs)*13000
    neuronGrid = (20, 20)
    lRate = 0.1   # 0.07 one of the best
    neighbourhoodRadius = 3
    neighbourhoodRadiusMin = 0.5
    noNoisePixels = 2
    assignmentMap = []


    trainingData = Data()._letters
    testData = Data().getNoisedLetters(['U','N','C','O','P','Y','R','I','G','H','T','A','B','L','E'],
noNoisePixels)
    numOfInputs = len(trainingData['A'])
```

```
    kohonenGroup = KohonenNeuronGroup(
        numOfInputs=numOfInputs,
        numOfNeurons=neuronGrid,
        processFunc=euklidesDistance,
        lRateFunc=Linear()(),
        neighbourhoodRadius=neighbourhoodRadius,
        nRadiusFunc=neighbourhoodRadiusCorrection(neighbourhoodRadius,
neighbourhoodRadiusMin, epochs),
        lRate=lRate
    )


    for lRate in [0.9, 0.5, 0.1, 0.05, 0.01]:
        for radiusMax in [20, 10, 5, 4, 3, 2, 1]:
            kohonenGroup.resetGroup(lRate, radiusMax, neighbourhoodRadiusCorrection(radiusMax,
neighbourhoodRadiusMin, epochs))
            winners = {}
            for i in range(epochs):
                for key, value in trainingData.items():
                    winners[key] = kohonenGroup.train(value)
                kohonenGroup.setNeighbourhoodRadius()
            testWinners = {}
            for key, value in testData.items():
                testWinners[key] = kohonenGroup.classify(value)
            resultsInNeighbourhood = 0
            resultsExact = 0
            uniqueNeurons = countUniqueItems(winners.values())
            for key, value in testWinners.items():
                dist = distance((value._x, value._y), (winners[key]._x, winners[key]._y))
                if dist <= radiusMax and dist != 0:
                    resultsInNeighbourhood+=1
                elif dist == 0:
                    resultsExact += 1
            resultsWrong = len(testData)-(resultsExact+resultsInNeighbourhood)
            print('lRate: {:.3f} radiusMax: {:d}\texact: {:d}\talmost: {:d}\twrong: {:d}\tunique:
{:d}'.format(kohonenGroup._lRate, radiusMax, resultsExact, resultsInNeighbourhood,
resultsWrong, uniqueNeurons))
```

data.py:

```
import copy
import numpy as np

class Data:

    def __init__(self):
        self.__dict__['_letters'] = {}
        self.__dict__['_letterLen']= 35
        self.getLetters()

    def getLetters(self):
            self._letters['A'] = [0,1,1,1,0,1,0,0,0,1,1,0,0,0,1,1,1,1,1,1,1,0,0,0,1,1,0,0,0,1,1,0,0,0,1]
            self._letters['B'] = [1,1,1,1,0,1,0,0,0,1,1,0,0,0,1,1,1,1,1,0,1,0,0,0,1,1,0,0,0,1,1,1,1,1,0]
            self._letters['C'] = [0,1,1,1,0,1,0,0,0,1,1,0,0,0,0,1,0,0,0,0,1,0,0,0,0,1,0,0,0,1,0,1,1,1,0]
            self._letters['D'] = [1,1,1,1,0,1,0,0,0,1,1,0,0,0,1,1,0,0,0,1,1,0,0,0,1,1,0,0,0,1,1,1,1,1,0]
            self._letters['E'] = [1,1,1,1,1,1,0,0,0,0,1,0,0,0,0,1,1,1,1,0,1,0,0,0,0,1,0,0,0,0,1,1,1,1,1]
            self._letters['F'] = [1,1,1,1,1,1,0,0,0,0,1,0,0,0,0,1,1,1,1,0,1,0,0,0,0,1,0,0,0,0,1,0,0,0,0]
            self._letters['G'] = [0,1,1,1,0,1,0,0,0,1,1,0,0,0,0,1,0,1,1,1,1,0,0,0,1,1,0,0,0,1,0,1,1,1,0]
```

```
        self._letters['H'] = [1,0,0,0,1,1,0,0,0,1,1,0,0,0,1,1,1,1,1,1,1,0,0,0,1,1,0,0,0,1,1,0,0,0,1]
        self._letters['I'] = [0,1,1,1,0,0,0,1,0,0,0,0,1,0,0,0,0,1,0,0,0,0,1,0,0,0,0,1,0,0,0,1,1,1,0]
        self._letters['J'] = [1,1,1,1,1,0,0,0,0,1,0,0,0,0,1,0,0,0,0,1,0,0,0,0,1,1,0,0,0,1,0,1,1,1,0]
        self._letters['K'] = [1,0,0,0,1,1,0,0,1,0,1,0,1,0,0,1,1,0,0,0,1,0,1,0,0,1,0,0,1,0,1,0,0,0,1]
        self._letters['L'] = [1,0,0,0,0,1,0,0,0,0,1,0,0,0,0,1,0,0,0,0,1,0,0,0,0,1,0,0,0,0,1,1,1,1,1]
        self._letters['M'] = [1,0,0,0,1,1,1,0,1,1,1,0,1,0,1,1,0,0,0,1,1,0,0,0,1,1,0,0,0,1,1,0,0,0,1]
        self._letters['N'] = [1,0,0,0,1,1,0,0,0,1,1,1,0,0,1,1,0,1,0,1,1,0,0,1,1,1,0,0,0,1,1,0,0,0,1]
        self._letters['O'] = [0,1,1,1,0,1,0,0,0,1,1,0,0,0,1,1,0,0,0,1,1,0,0,0,1,1,0,0,0,1,0,1,1,1,0]
        self._letters['P'] = [1,1,1,1,0,1,0,0,0,1,1,0,0,0,1,1,1,1,1,0,1,0,0,0,0,1,0,0,0,0,1,0,0,0,0]
        self._letters['Q'] = [0,1,1,1,0,1,0,0,0,1,1,0,0,0,1,1,0,0,0,1,1,0,1,0,1,1,0,0,1,1,0,1,1,1,1]
        self._letters['R'] = [1,1,1,1,0,1,0,0,0,1,1,0,0,0,1,1,1,1,1,0,1,0,1,0,0,1,0,0,1,0,1,0,0,0,1]
        self._letters['S'] = [0,1,1,1,0,1,0,0,0,1,1,0,0,0,0,0,1,1,1,0,0,0,0,0,1,1,0,0,0,1,0,1,1,1,0]
        self._letters['T'] = [1,1,1,1,1,0,0,1,0,0,0,0,1,0,0,0,0,1,0,0,0,0,1,0,0,0,0,1,0,0,0,0,1,0,0]
        self._letters['U'] = [1,0,0,0,1,1,0,0,0,1,1,0,0,0,1,1,0,0,0,1,1,0,0,0,1,1,0,0,0,1,0,1,1,1,0]
        self._letters['V'] = [1,0,0,0,1,1,0,0,0,1,1,0,0,0,1,1,0,0,0,1,1,0,0,0,1,0,1,0,1,0,0,0,1,0,0]
        self._letters['W'] = [1,0,0,0,1,1,0,0,0,1,1,0,0,0,1,1,0,0,0,1,1,0,1,0,1,1,0,1,0,1,0,1,0,1,0]
        self._letters['X'] = [1,0,0,0,1,1,0,0,0,1,0,1,0,1,0,0,0,1,0,0,0,1,0,1,0,1,0,0,0,1,1,0,0,0,1]
        self._letters['Y'] = [1,0,0,0,1,1,0,0,0,1,0,1,0,1,0,0,0,1,0,0,0,0,1,0,0,0,0,1,0,0,0,0,1,0,0]
        self._letters['Z'] = [1,1,1,1,1,0,0,0,0,1,0,0,0,1,0,0,0,1,0,0,0,1,0,0,0,1,0,0,0,0,1,1,1,1,1]


    def getNoisedLetters(self, keys, noNoisePixels=2):
        noisedLetters = {}
        for key in keys:
            noisedLetters[key] = self._letters[key]
            pixelsIndexes = np.random.choice(self._letterLen, noNoisePixels, replace=False)
            for index in pixelsIndexes:
                pixel = noisedLetters[key][index]
                noisedLetters[key][index] = 1 if pixel==0 else 0

        return noisedLetters
```

---

## Źródła


„Sieci neuronowe do przetwarzania informacji” S. Osowski