Podstawy Sztucznej Inteligencji

Mikołaj Stepniewski

Uczenie sieci regułą Hebba

Cel ćwiczenia

Poznanie działania reguły Hebba na przykładzie rozpoznawania emotikon.

Opis sieci

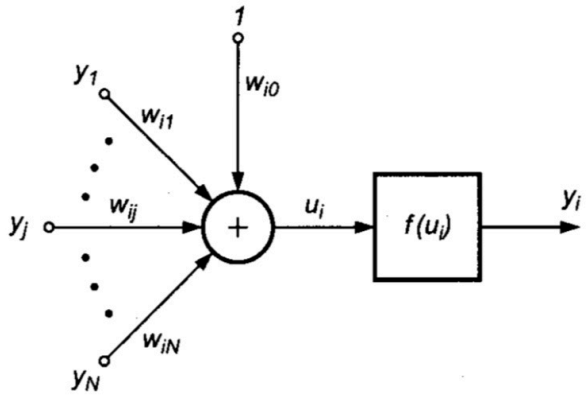
Neuron Hebb'a

D. Hebb zauważył, że powiązanie dwóch komórek nerwowych jest wzmacniane, jeśli obie komórki zostaną pobudzone (są aktywne) w tym samym czasie. Czyli jeśli j-ta komórka o sygnale wyjściowym y_j jest powiązana z i-tą komórką o sygnale wyjściowym y_i, przez wagę w_{ij}, to na stan ich powiązań wpływają sygnały y_j oraz y_i.

Na tej podstawie Hebb zaproponował regułę matematyczną uwzględniającą te zależności. Zgodnie z regułą Hebba, zmiana wagi w_{ij} odbywa się proporcjonalnie do iloczynu sygnały wejściowego oraz wyjściowego:

$$\Delta w_{ij} = \eta y_j y_i$$

Reguła Hebba może być stosowana do różnych typów struktur sieci neuronowych oraz funkcji aktywacji w modelu neuronu.



Rys. 2.13. Ogólny model neuronu Hebba

Waga $\mathbf{w_{ij}}$ włączana jest między sygnałem wejściowym $\mathbf{y_j}$ a węzłem sumacyjnym **i-tego** neuronu o sygnale wyjściowym $\mathbf{y_i}$.

Uczenie neuronu z wykorzystaniem reguły Hebba może odbywać się w trybie z nauczycielem i bez nauczyciela.

Reguła Hebba charakteryzuje się tym, że w wyniku uczenia wagi neuronów mogą przyjąć wartości dowolnie duże, gdyż w każdym cyklu następuje proces sumowania aktualnych wartości wag oraz skończonego przyrostu Δw_{ij}.

$$w_{ij}(k+1) = w_{ij}(k) + \Delta w_{ij}$$

Jedną z metod poprawy stabilności uczenia jest wg. reguły Hebba jest jest przyjęcie przy aktualizacji wag nie ostatniej wartości w_{ij}, lecz wartości pomniejszonej o tzw. ***współczynnik zapominania γ.*** Wówczas regułę Hebba można zapisać w postaci:

$$w_{ij}(k+1) = (1-\gamma)w_{ij}(k) + \Delta w_{ij}$$

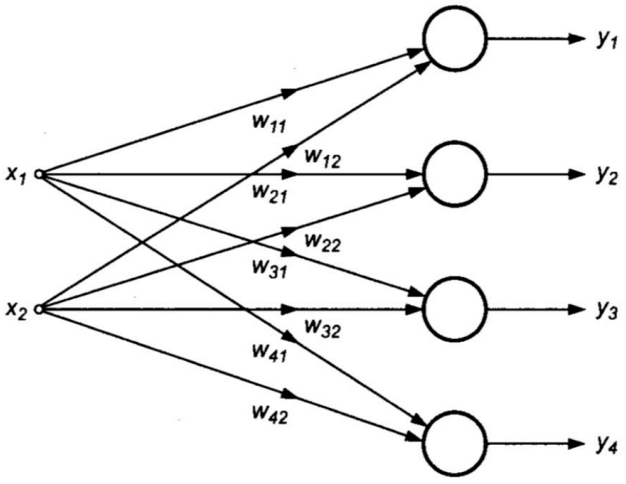
Algorytm WTA (Winner Takes All)

Neurony typu WTA mają stopień wejściowy w postaci standardowego sumatora sumującego iloczyny sygnałów wejściowych z ich odpowiednimi wagami.

$$u_i = \sum_{j=0}^{N} w_{ij}x_j$$

Grupa neuronów współzawodniczących ze sobą otrzymuje te same sygnały wejściowe x_j. W zależności od aktualnych wag, sygnały wyjściowe neuronów różnią się między sobą. Zwycięzcą zostaje neuron, którego sygnał wyjściowy jest największy. Taki neuron ma prawo do aktualizacji swoich wag. Dzięki temu za każdą wygraną epoką wagi neuronu zbliżają się do danego wektora wejściowego. Tym samym wektor przyporządkowuje go do siebie. Neurony, które nie zwyciężyły w danej epoce zostają zablokowane i ich wagi nie ulegają zmianie.

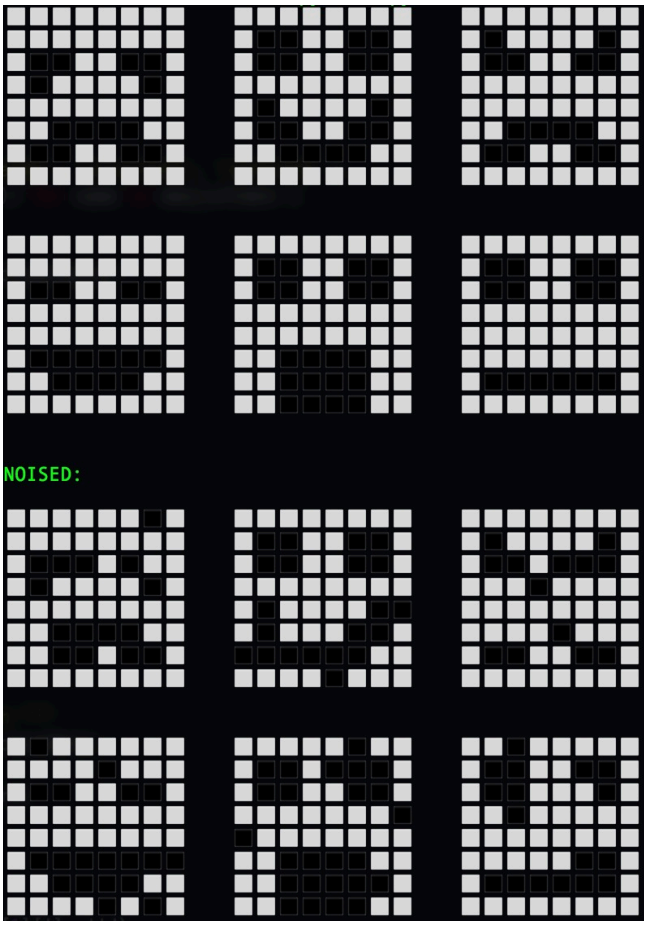
W efekcie takiego współzawodnictwa następuje samoorganizacja procesu uczenia. Neurony dopasowują swoje wagi w taki sposób, że przy prezentacji grup wektorów wejściowych zbliżonych do siebie zwycięża zawsze ten sam neuron. W trybie pracy odtworzeniowej odpowiedni neuron przez zwycięstwo we współzawodnictwie rozpoznaje swoją kategorię. Układy tego typu są stosowane najczęściej do klasyfikacji wektorów.

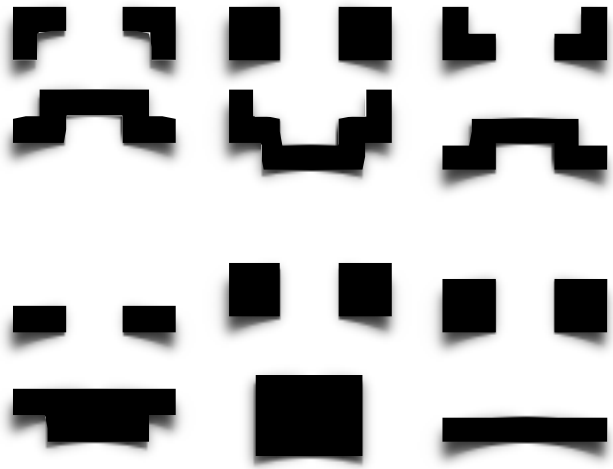


Rys. 2.11. Sieć neuronowa typu WTA z przykładu

Dane uczące

Do uczenia wykorzystałem 6 emotikon 8x8 px oraz ich losowo zaszumione wersje (od 0 do 6 pikseli).





Emotikony zostały wprowadzone do sieci neuronowej w formie tablic 1d, gdzie czarne elementy reprezentowane były przez liczby -1 a białe przez liczby 1. Zaszumienie polegało na pomnożeniu danego elementu tablicy przez (-1). Elementy odwracane w ten sposób nie powtarzały się podczas zaszumiania.

Budowa sieci

Po wielu próbach uznałem, że najlepszą liczbą neuronów w grupie WTA jest siatka 12 x 12, czyli łącznie 144 neuronów. Dzięki tak dużej liczbie, istnieje znacznie większa szansa, że neuron już na początku uczenia będzie posiadał duże podobieństwo do danego wektora wejściowego. Co za tym idzie - szybciej dopasuję się do niego i już po kilku epokach sieć będzie odpowiednio klasyfikować wektory wejściowe.

Sygnał wyjściowy każdego z neuronów obliczany jest za pomocą sumy ilorazów wag i odpowiadających im elementów wektora wejściowego:

$$u_i = \sum_{j=0}^{N} w_{ij} x_j$$

Funkcja aktywacji: funkcja liniowa $f(x) = x$

Parametry uczenia

Dla takiej budowy sieci, dobór współczynników uczenia i zapominania jest bardzo istotny i nawet bardzo mała zmiana potrafi zaburzyć wyniki uczenia. Wynika to z tego, że dla neuronów typu hebbowskiego wagi bardzo szybko rosną i trzeba je w pewien sposób utrzymać na pewnym poziomie.

Metodą prób i błędów doszedłem do następujących wartości współczynników:

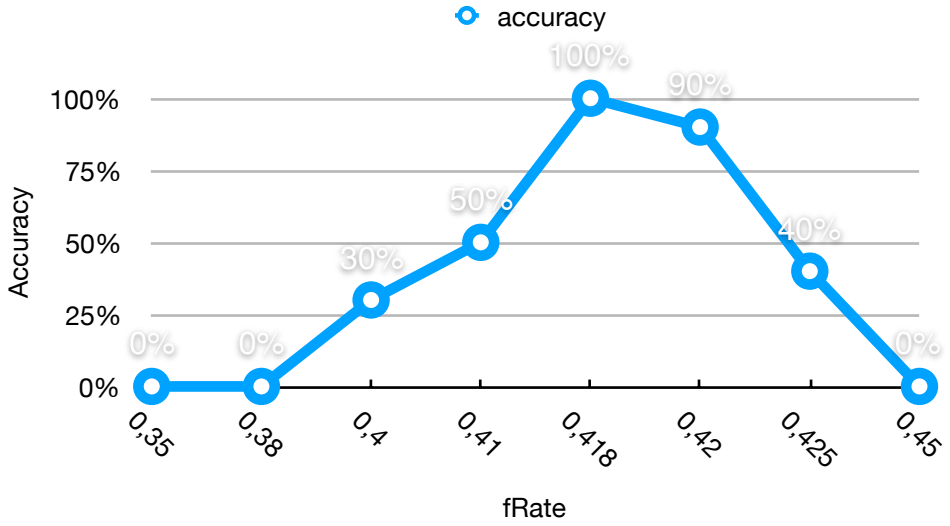
learning rate	forget rate	bias weight	number of neurons	number of epochs
0,007	0,418	-0,5	144	20

Dla takich parametrów sieć radziła sobie z rozpoznawaniem emotikon zaszumionych nawet 13-stoma pikselami, co widać na poniższym screenshocie:

Wyniki uczenia (z zapominaniem) X oznacza błędny wynik a liczba w nawiasie ilość akt. neur.

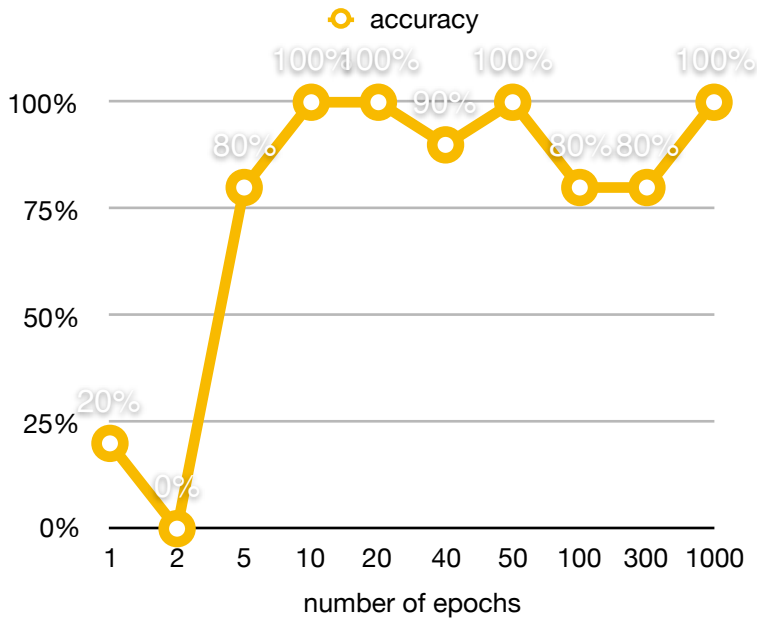
1. Forget rate

fRate	1	2	3	4	5	6	7	8	9	10	number of correct results	accuracy	most common number of active neurons
0,35	X(3)	X(3)	X(3)	X(3)	X(3)	X(3)	X(3)	X(3)	X(3)	X(3)	0	0%	3
0,38	X(5)	X(4)	X(5)	X(4)	X(5)	X(4)	X(4)	X(5)	X(5)	X(5)	0	0%	5
0,4	X(5)	X(5)	X(5)	OK	X(5)	OK	OK	X(5)	X(5)	X(5)	3	30%	5
0,41	X(5)	OK	X(5)	X(5)	OK	X(5)	OK	X(5)	OK	OK	5	50%	5/6
0,418	OK	OK	OK	OK	OK	OK	OK	OK	OK	OK	10	100%	6
0,42	OK	OK	OK	X(12)	OK	OK	OK	OK	OK	OK	9	90%	6
0,425	X(5)	OK	OK	X(11)	X(12)	OK	X(12)	OK	X(12)	X(12)	4	40%	12
0,45	X(12)	X(11)	X(12)	X(12)	X(10)	X(12)	X(12)	X(11)	X(12)	X(8)	0	0%	12



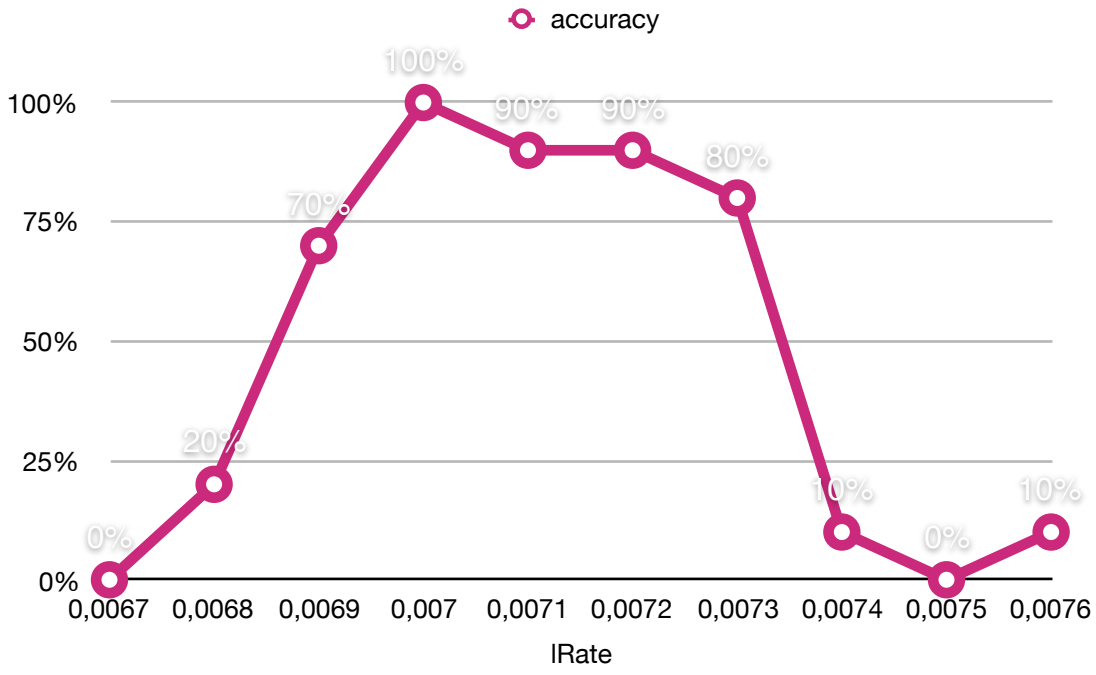
2. Number of epochs

number of epochs	1	2	3	4	5	6	7	8	9	10	num. of correct results	accuracy	most common number of active neurons
1	OK	X(4)	X(6)	X(4)	X(4)	X(4)	X(5)	OK	X(5)	X(5)	2	20%	4
2	X(6)	X(5)	X(4)	X(5)	X(4)	X(6)	X(5)	X(5)	X(5)	X(6)	0	0%	5
5	OK	OK	OK	OK	OK	X(5)	OK	OK	OK	X(5)	8	80%	6
10	OK	OK	OK	OK	OK	OK	OK	OK	OK	OK	10	100%	6
20	OK	OK	OK	OK	OK	OK	OK	OK	OK	OK	10	100%	6
40	OK	OK	OK	X(5)	OK	OK	OK	OK	OK	OK	9	90%	6
50	OK	OK	OK	OK	OK	OK	OK	OK	OK	OK	10	100%	6
100	OK	X(5)	OK	OK	OK	OK	X(5)	OK	OK	OK	8	80%	6
300	OK	X(5)	OK	OK	OK	OK	OK	OK	X(12)	OK	8	80%	6
1000	OK	OK	OK	OK	OK	OK	OK	OK	OK	OK	10	100%	6



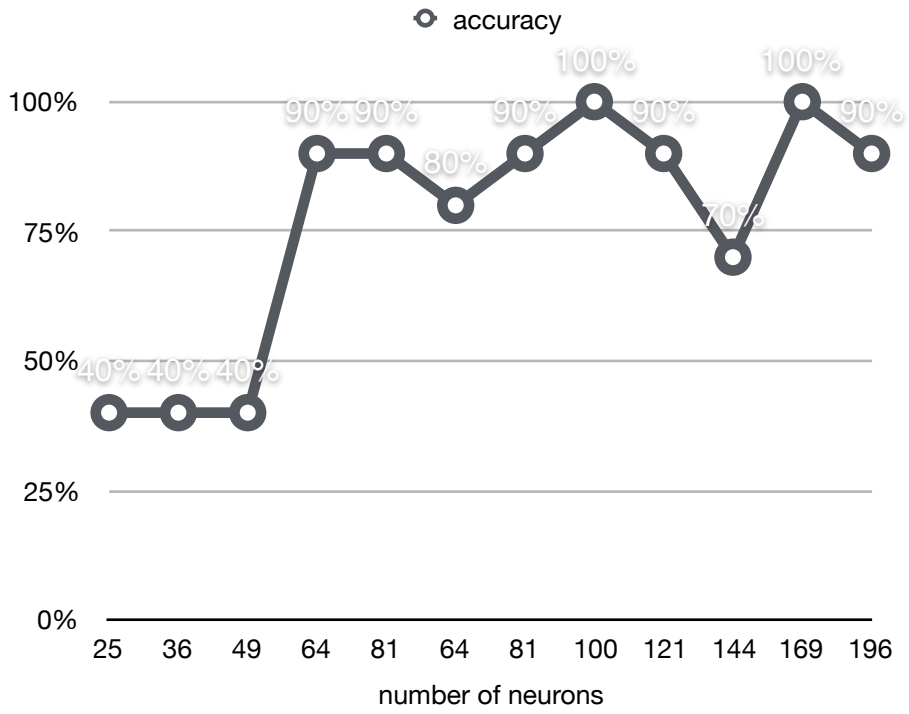
3. Learning rate

lRate	1	2	3	4	5	6	7	8	9	10	num. of correct results	accuracy	most common num. of active neurons
0,0067	X(12)	X(10)	X(10)	X(10)	X(9)	X(10)	X(12)	X(9)	X(10)	X(11)	0	0%	10
0,0068	X(10)	OK	X(8)	X(10)	OK	X(10)	X(11)	X(8)	X(7)	X(7)	2	20%	10
0,0069	OK	OK	OK	OK	OK	X(8)	OK	OK	X(8)	X(7)	7	70%	6
0,007	OK	OK	OK	OK	OK	OK	OK	OK	OK	OK	10	100%	6
0,0071	OK	OK	X(5)	OK	OK	OK	OK	OK	OK	OK	9	90%	6
0,0072	OK	OK	OK	OK	OK	OK	OK	OK	X(6)	OK	9	90%	6
0,0073	OK	OK	OK	OK	OK	OK	OK	X(5)	OK	X(5)	8	80%	6
0,0074	X(5)	X(5)	X(5)	X(5)	X(5)	X(5)	X(5)	X(5)	X(5)	OK	1	10%	5
0,0075	X(5)	X(5)	X(5)	X(5)	X(5)	X(5)	X(5)	X(4)	X(5)	X(5)	0	0%	5
0,0076	X(5)	X(5)	OK	X(5)	X(5)	X(5)	X(5)	X(5)	X(5)	X(5)	1	10%	5



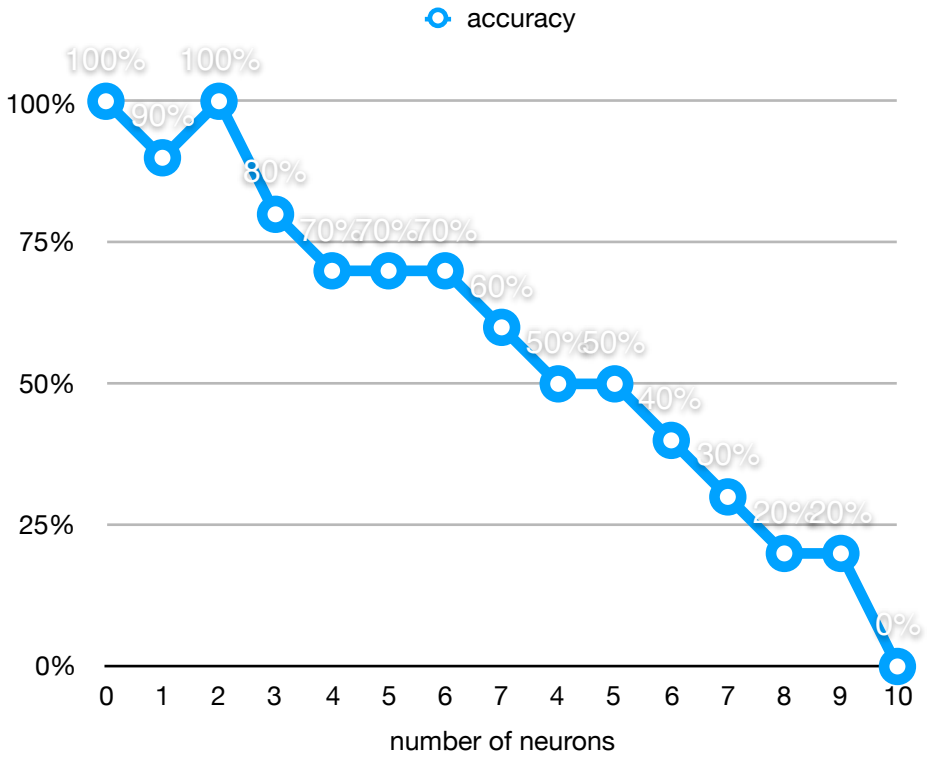
4. Number of neurons

number of neurons	1	2	3	4	5	6	7	8	9	10	num. of correct results	accuracy	most common num. of active neurons
25	X(9)	OK	OK	OK	X(8)	X(9)	X(7)	X(8)	X(8)	OK	4	40%	8
36	X(10)	X(7)	OK	X(7)	X(6)	X(8)	OK	OK	OK	X(9)	4	40%	7
49	X(9)	X(8)	OK	OK	X(7)	X(7)	OK	X(7)	X(7)	OK	4	40%	7
64	OK	OK	OK	OK	OK	X(6)	OK	OK	OK	OK	9	90%	6
81	X(7)	OK	OK	OK	OK	OK	OK	OK	OK	OK	9	90%	6
100	OK	OK	OK	X(5)	OK	OK	X(6)	OK	OK	OK	8	80%	6
121	OK	OK	OK	OK	OK	OK	OK	X(8)	OK	OK	9	90%	6
144	OK	OK	OK	OK	OK	OK	OK	OK	OK	OK	10	100%	6
169	OK	OK	OK	OK	OK	OK	X(6)	OK	OK	OK	9	90%	6
196	OK	X(5)	X(5)	OK	OK	OK	OK	X(5)	OK	OK	7	70%	6
225	OK	OK	OK	OK	OK	OK	OK	OK	OK	OK	10	100%	6
400	OK	OK	OK	OK	OK	OK	OK	OK	OK	X(5)	9	90%	6



5. Number of noise pixels

number of noise pixels	1	2	3	4	5	6	7	8	9	10	num. of correct results	accuracy	most common num. of active neurons
0	OK	OK	OK	OK	OK	OK	OK	OK	OK	OK	10	100%	6
1	OK	OK	OK	OK	OK	OK	OK	OK	X(5)	OK	9	90%	6
2	OK	OK	OK	OK	OK	OK	OK	OK	OK	OK	10	100%	6
3	OK	OK	OK	X(5)	OK	OK	OK	X(5)	OK	OK	8	80%	6
4	OK	OK	X(5)	X(5)	OK	X(5)	OK	OK	OK	OK	7	70%	6
5	OK	OK	OK	OK	OK	OK	OK	X(6)	X(6)	X(6)	7	70%	6
6	X(5)	OK	OK	OK	OK	X(6)	X(6)	OK	OK	OK	7	70%	6
7	X(6)	OK	OK	X(6)	X(6)	OK	OK	OK	X(6)	OK	6	60%	6
8	OK	X(6)	X(6)	X(5)	OK	OK	OK	X(6)	OK	X(6)	5	50%	6
9	X(6)	X(6)	OK	X(6)	X(6)	OK	OK	X(6)	OK	OK	5	50%	6
10	X(6)	X(6)	X(5)	X(6)	X(5)	OK	X(5)	OK	OK	OK	4	40%	6
11	OK	X(6)	X(6)	X(5)	X(5)	X(5)	OK	X(5)	X(5)	OK	3	30%	6
12	OK	X(5)	X(6)	X(5)	OK	X(5)	X(6)	X(6)	X(6)	X(5)	2	20%	6
13	X(6)	OK	X(6)	X(6)	X(6)	X(6)	X(5)	OK	X(6)	X(6)	2	20%	6
14	X(6)	X(6)	X(6)	X(6)	X(6)	X(6)	X(6)	X(6)	X(5)	X(6)	0	0%	6



Wnioski

1. Powodem dość dziwnych wartości optymalnych lRate oraz fRate (normalnie fRate powinien być mniejszy od lRate) jest to, że wektory wejściowe nie były normalizowane i neurony po prostu musiały szybciej zapominać niż zwykle. Przez brak normalizacji wagi neuronów rosły w zawrotnym tempie do bardzo dużych wartości. Jednak znalazłem optymalne wartości, dla których ta sieć działała bardzo dobrze.
2. Sieć nie potrafiła nauczyć się bez współczynnika zapominania, co wynika z pierwszego wniosku.
3. Sieć bardzo dobrze radzi sobie z rozpoznawaniem zaszumionych emotikon, mimo że są one dosyć małe i zaszumienie szybko prowadzi do dużych różnic między oryginałem.
4. Podczas testowania zaszumionych emotikon zauważyłem, że lepiej dobrać trochę większą ilość neuronów i epok (chociaż różnica nie jest duża). Optymalna ilość neuronów do rozpoznawania zaszumionych emotikon to 169 lub 225 a liczba epok to ok. 200.

Listing

neurons.py:

```
import sys
# Add the include folder path to the sys.path list
sys.path.append('../include')
from supportFunctions import *
from collections import Counter


class HebbNeuron:
    def __init__(self, numOfInputs, iid, activFunc, lRate=0.1, fRate=0.28, bias=-0.5):
        self._weights = np.array([np.random.uniform(-1, 1) for _ in range(numOfInputs)])
        self.__dict__['_bias'] = bias
        self.__dict__['_activFunc'] = activFunc
        self.__dict__['_lRate'] = lRate
        self.__dict__['_fRate'] = fRate     # forget rate
        self.__dict__['_trainingData'] = None
        self.__dict__['_sum'] = None
        self.__dict__['_val'] = None
        self.__dict__['_iid'] = iid
        self.__dict__['_sumHist'] = []
        self.__dict__['_winnerCounter'] = 0

    def process(self, inputs):
        """ Compute y_i (simple dot) """
        self._sum = np.dot(self._weights, inputs) + self._bias
        self._val = self._activFunc(self._sum)
        return self._val


    def train(self, inputs):
        """ Get y_i """
        output = self.process(inputs)
        constant = self._lRate * output
        forget = (1.0 - self._fRate)
        for i in range(len(self._weights)):
            """ Apply forgetting """
            self._weights[i] *= forget
            """ Update weights """
            self._weights[i] += constant * inputs[i]

        # self._bias *= 1.0 - self._fRate
        # self._bias += constant
```

```
""" Simple WTA for now...

    Winner is neuron with least distance
    between weights vector and input vector """
class HebbNeuronGroup:
    def __init__(self, numOfInputs, numOfNeurons, activFunc, lRateFunc, lRate=0.007, fRate=0.1):
        self.__dict__['_numOfNeurons'] = numOfNeurons
        self.__dict__['_activFunc'] = activFunc
        self.__dict__['_lRate'] = lRate
        self.__dict__['_fRate'] = fRate
        self.__dict__['_numOfInputs'] = numOfInputs
        self.__dict__['_neurons'] = None
        self.__dict__['_lRateFunc'] = lRateFunc
        self.__dict__['_currentLRate'] = None

        """ Create group of neurons """
        self._neurons = [[HebbNeuron(numOfInputs, i*numOfNeurons[0]+j, activFunc, lRate=lRate, fRate=fRate)
            for i in range(numOfNeurons[0])]
            for j in range(numOfNeurons[1])
        ]

    """ Reselting weights of all neurons in group """
    def resetWeights(self):
        for row in self._neurons:
            for neuron in row:
                neuron.resetWeights()

    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 train(self, vectors, histFreq=1, retMostCommon=False):
        winners = []
        """ Finding winner (highest value) """
        for i, vector in enumerate(vectors):
            winner = None
            for row in self._neurons:
                for neuron in row:
                    neuron.process(vector)
                    if winner == None:
                        winner = neuron
                    elif winner != None:
                        if neuron._val > winner._val:
                            winner = neuron

            """ Winner Takes All """
            if winner._winnerCounter % histFreq == 0:
                winner._sumHist.append(winner._sum)
            winner._winnerCounter += 1
            winner.train(vector)
            winners.append(winner)

        """ Updating lRate """
        self.setLRate(self._lRateFunc(self._lRate))

        if retMostCommon:
            return Counter(winners).most_common(1)[0][0]

        return winners

    """ Basicaly the same as above, but without updating weights """
    def classify(self, vectors):
```

```
        winners = []
        for i, vector in enumerate(vectors):
            winner = None
            for row in self._neurons:
                for neuron in row:
                    neuron.process(vector)
                    if winner == None:
                        winner = neuron
                    elif winner != None:
                        if neuron._val > winner._val:
                            winner = neuron

            winners.append(winner)

        return winners


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

emojis.py:

```
class Emoji:
    def getEmoji(self, name):
        return {
            'sad':
                [
                    1,1,1,1,1,1,1,1,
                    1,1,1,1,1,1,1,1,
                    1,0,0,1,1,0,0,1,
                    1,0,1,1,1,1,0,1,
                    1,1,1,1,1,1,1,1,
                    1,1,0,0,0,0,1,1,
                    1,0,0,1,1,0,0,1,
                    1,1,1,1,1,1,1,1
                ],
            'smile':
                [
                    1,1,1,1,1,1,1,1,
                    1,0,0,1,1,0,0,1,
                    1,0,0,1,1,0,0,1,
                    1,1,1,1,1,1,1,1,
                    1,0,1,1,1,1,0,1,
                    1,0,0,1,1,0,0,1,
                    1,1,0,0,0,0,1,1,
                    1,1,1,1,1,1,1,1
                ],
            'laugh':
                [
                    1,1,1,1,1,1,1,1,
                    1,1,1,1,1,1,1,1,
                    1,0,0,1,1,0,0,1,
                    1,1,1,1,1,1,1,1,
                    1,1,1,1,1,1,1,1,
                    1,0,0,0,0,0,0,1,
                    1,1,0,0,0,0,1,1,
                    1,1,1,1,1,1,1,1
                ],
            'angry':
                [
                    1,1,1,1,1,1,1,1,
                    1,0,1,1,1,1,0,1,
                    1,0,0,1,1,0,0,1,
                    1,1,1,1,1,1,1,1,
                    1,1,1,1,1,1,1,1,
                    1,1,0,0,0,0,1,1,
                    1,0,0,1,1,0,0,1,
                    1,1,1,1,1,1,1,1
                ],
```

```
            'surprised':
                [
                    1,1,1,1,1,1,1,1,
                    1,0,0,1,1,0,0,1,
                    1,0,0,1,1,0,0,1,
                    1,1,1,1,1,1,1,1,
                    1,1,1,1,1,1,1,1,
                    1,1,0,0,0,0,1,1,
                    1,1,0,0,0,0,1,1,
                    1,1,0,0,0,0,1,1
                ],
            'confused':
                [
                    1,1,1,1,1,1,1,1,
                    1,0,0,1,1,0,0,1,
                    1,0,0,1,1,0,0,1,
                    1,1,1,1,1,1,1,1,
                    1,1,1,1,1,1,1,1,
                    1,1,1,1,1,1,1,1,
                    1,0,0,0,0,0,0,1,
                    1,1,1,1,1,1,1,1

                ],
            'test':
                [
                    1,1,1,1,1,1,1,1,
                    1,1,1,1,1,1,1,1,
                    1,1,1,1,1,1,1,1,
                    1,1,1,1,1,1,1,1,
                    1,1,1,1,1,1,1,1,
                    1,1,1,1,1,1,1,1,
                    1,1,1,1,1,1,1,1,
                    1,1,1,1,1,1,1,1
                ]
        }.get(name, -1)
```

main.py:

```
from neurons import *
from emojis import *
from progressBar import *
import copy


""" Change emojis pixel values from (0,1) to (-1,1) """
def bipolar(emoji):
    for i in range(len(emoji)):
        if emoji[i] == 0:
            emoji[i] = -1
    return emoji

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

""" Get random but unique pixels and mult by -1 """
def noiseEmojis(arr, numOfNoisePixels):
    noisedArr = copy.deepcopy(arr)
    for emoji in noisedArr:
        pixels = np.random.choice(64, numOfNoisePixels, replace=False)
        for pixel in pixels:
            emoji[pixel] *= -1

    return noisedArr

""" Print emojis """
def drawEmojis(emojis):
    i = 0
    emojis = np.split(np.array(emojis), 2)
    for j in range(len(emojis)):
        for row in range(8):
```

```
            for emoji in emojis[j]:
                for i in range(8):
                    print(' ■ ' if emoji[row*8+i] == -1 or emoji[row*8+i] == 0 else '   ', end=' ', flush=True)
                    if (i+1) % 8 == 0:
                        print('    ', end='', flush=False)
                        pass
            print()
        print('\n')



if __name__ == '__main__':

    #np.random.seed(5)
    neuronGrid = [15, 15]
    lRate=0.007         # learning rate
    fRate=0.418         # forget rate    BEST: 0.418
    numOfNoisePixels=14 # number of noised pixels
    epochs=200          # nuber of epochs
    decay=30*12         # decay for updating lRate during the learning proccess
    pBar = ProgressBar(length=56)

    emoji = Emoji()
    """ Get a set of training  emojis """
    emojisToGet = [ 'sad', 'smile', 'angry', 'laugh', 'surprised', 'confused' ]
    trainingSet = [ bipolar(emoji.getEmoji(name)) for name in emojisToGet ]

    """ Working well up to 9 noise pixels """
    noisedSet = noiseEmojis(trainingSet, numOfNoisePixels)

    drawEmojis(trainingSet)
    print('NOISED with {:d} pixels:\n'.format(numOfNoisePixels))
    drawEmojis(noisedSet)

    hebbGroup = HebbNeuronGroup(
        numOfInputs=64,
        numOfNeurons=neuronGrid,
        activFunc=Linear()(),
        lRateFunc=Linear()(),
        lRate=lRate,
        fRate=fRate
    )

    print('Running {:d} epochs...'.format(epochs))
    pBar.start(maxVal=epochs)
    for i in range(epochs):
        """ Will get winners from the latest epoch """
        winners1 = hebbGroup.train(trainingSet)
        pBar.update()

    """ Try to classify noised emojis """
    winners2 = hebbGroup.classify(noisedSet)

    numOfActiveNeurons = countUniqueItems(winners1+winners2)


    print('\t\tNORMAL\tNOISED')
    for i in range(len(winners1)):
        print('{}     \t{:d}\t{:d}'.format(emojisToGet[i], winners1[i]._iid, winners2[i]._iid))


    print('\nActive neurons: {:d}'.format(numOfActiveNeurons))
    print('lRate: {:.5f}'.format(hebbGroup._currentLRate))
```

Źródła:

„Sieci neuronowe do przetwarzania informacji” S. Osowski