Podstawy Sztucznej Inteligencji

Scenariusz 2

Budowa i działanie neuronowej sieci jednowarstwowej**.**

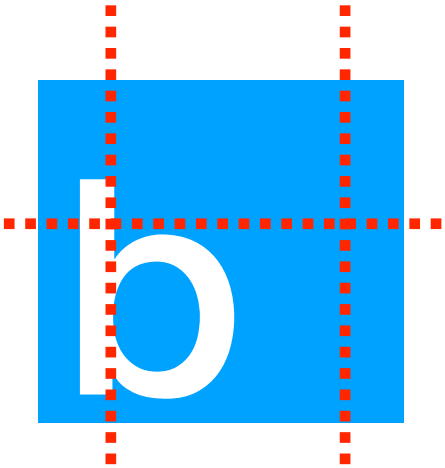
Cel ćwiczenia

Poznanie budowy i działania jednowarstwowych sieci neuronowych oraz uczenie rozpoznawania wielkości liter.

Założenia

Aby uczenie rozpoznawania wielkości liter w sieci jednowarstwowej było możliwe, musimy przydzielić neuronom pewne podzadania, których wyniki będą przetwarzane przez osobny neuron. Podzieliłem problem rozpoznawania wielkości litery na podzadania:

1. Czy litera przekracza lewą linię?
2. Czy litera przekracza prawą linię?
3. Czy litera przekracza górną linię?



Większość dużych litery takich jak „A” lub „O” przekracza wszystkie trzy linie, lecz np. „I” przekracza tylko górną linię, a „L” lewą oraz górną (zakładam, że musi przekraczać większością swojego obszaru). Widać, że tutaj pojawia się pewne skomplikowanie, które będzie idealnym zadaniem dla sztucznej inteligencji.

Zatem w warstwie będą znajdować się trzy neurony sigmoidalne lub Adaline (o tym więcej w dalszej części), gdzie każdy z nich będzie uczył się rozpoznawać jedno z trzech wyżej wymienionych podzadań. W czasie uczenia tej warstwy, osobny perceptron będzie uczył się rozpoznawać czy litera jest duża czy mała na podstawie oczekiwanych odpowiedzi dla warstwy. Oczywiście perceptron nie będzie uczył się na podstawie odpowiedzi neuronów warstwy, ponieważ potrzebowalibyśmy zaimplementować algorytm wstecznej propagacji błędu.

Neuron sigmoidalny

To taki neuron, którego funkcja w przeciwieństwie do modelu McCullocha-Pitsa jest ciągła i przyjmuje postać funkcji sigmoidalnej (w tym przypadku unipolarnej):

$$f(x) = \frac{1}{1 + \mathrm{e}^{-\beta x}}$$

Argumentem dla tej funkcji jest oczywiście sygnał sumacyjny *u.* Współczynnik beta jest dobierany przez programistę i wpływa on na kształt funkcji. Zazwyczaj jednak używa się beta równego 1 i również tak jest w mojej sieci. Ważną cechą funkcji sigmoidalnej jest jej różniczkowalność, dzięki której **możemy zastosować metodę gradientową**. Najprościej jest

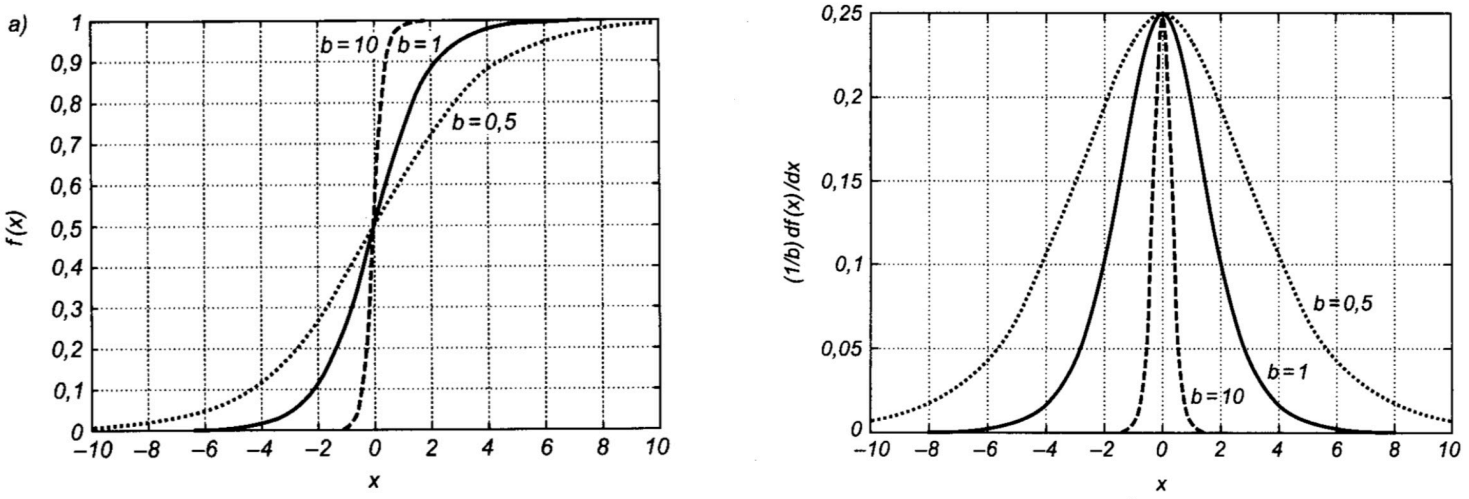
przyjąć metodę największego spadku, zgodnie z którą aktualizacja wektora wag odbywa się w kierunku ujemnego gradientu funkcji celu. Uaktualnianie w sposób dyskretny:

$$w_{ij}(k+1) = w_{ij}(k) - \eta\,\delta_i\,x_j$$

gdzie: $$\delta_i = e_i \frac{\mathrm{d}f(u_i)}{\mathrm{d}u_i}$$

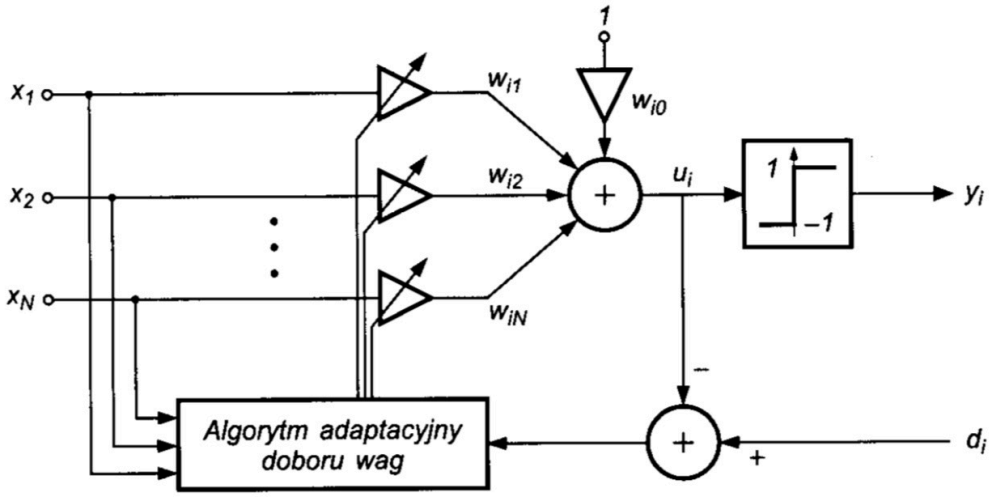
$$e_i = (y_i - d_i)$$

Wykres funkcji sigmoidalnej oraz jej pochodnej:



Neuron Adaline (Adaptive Linear Neuron)

Opracowany przez B.Widrowa. Funkcja aktywacji jest typu signum a jego schemat adaptacyjnego sposobu doboru wag przedstawia poniższy rysunek:



Rys. 2.6. Schemat neuronu typu adaline

Główną różnicą Adaline a np. Perceptronem (który też posiada model nieliniowy) jest to, że **w definicji funkcji celu używa jedynie części liniowej** (sumę wagową sygnałów wejściowych). Dzięki temu właśnie jest możliwe zastosowanie algorytmu gradientowego uczenia. W minimalizacji funkcji celu używa się metodę największego spadku, podobnie jak w przypadku neuronu sigmoidalnego.

$$w_{ij}(k+1) = w_{ij}(k) + \eta\, e_i\, x_j$$

gdzie: $$e_i = \left(d_i - \sum_{j=0}^{N} w_{ij} x_j\right)$$

Wyniki

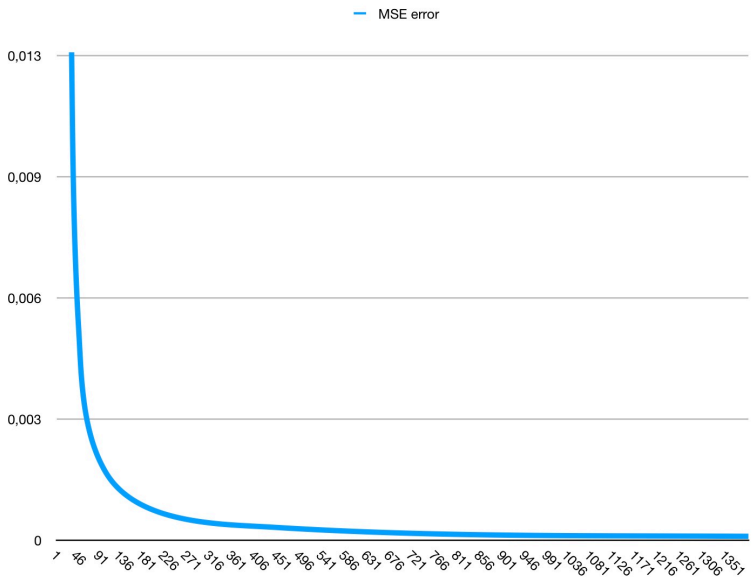
Rezultaty były zaskakująco dobre jak na jedną tylko warstwę i czasem subiektywny podział rozpoznawania wielkości na podzadania. W pętli uczenia sieci dodałem warunek kończący ją w momencie uzyskania błędu średniokwadratowego mniejszego niż 0.0001. Przeprowadziłem testy warstwy neuronów sigmoidalnych dla współczynnika uczenia równego **0.1, 0.5, 0.01 oraz 0.05.** Dokładne wyniki zapisane są w katalogu /Charts/. Zgodnie z moimi przewidywaniami okazało się, że współczynnik uczenia wpływa mocno na szybkość uczenia się warstwy.

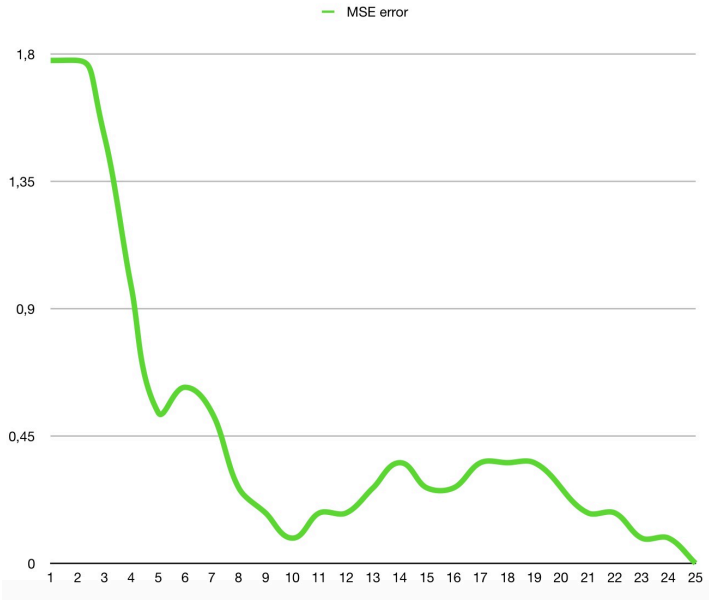
- Dla lRate = **0.01** - sieć nauczyła się po upływie **64414** epok
- Dla lRate = **0.05** - sieć nauczyła się po upływie **13420** epok
- Dla lRate = **0.1** - sieć nauczyła się po upływie **6816** epok
- Dla lRate = **0.5** - sieć nauczyła się po upływie **1315** epok

Dla Adaline nie udało mi się przeprowadzić wszystkich testów, gdyż dla lRate równego 0.1 i 0.5 uczenie stawało w miejscu na MSE równym około 0.24. Po długiej analizie doszedłem do wniosku, że ma to związek z danymi wejściowymi, które dla takiego dużego lRate ujawniają podatność Adaline na zatrzymywanie się w miejscu podczas uczenia, spowodowaną taką a nie inną budową funkcji celu (sposobem aktualizacji wag z wykorzystaniem tylko sumy wagowej). Udało się natomiast uzyskać wyniki dla:

- dla lRate - **0.01** - sieć nauczyła się po upływie **39** epok
- dla lRate - **0.05** - sieć nauczyła się po upływie **25** epok

Najszybsze uczenie sieci **Sigmoidalnej** vs. **Adaline:**





Spostrzeżenia

Po serii testów okazuje się, że Adaline źle radzi sobie z rozpoznawaniem nieznanych jej liter. Natomiast warstwa sigmoidalna z nieznanymi literami nie ma większego problemu. Niestety jak widać wyżej warstwa sigmoidalna potrzebuje więcej czasu aby się nauczyć na podstawie danych treningowych.

Wnioski

W tak prostym przykładzie jak rozpoznawanie wielkości oraz w prostej sieci jednowarstwowej składającej się z tylko trzech neuronów liter współczynnik uczenia bardzo widocznie zmienia szybkość uczenia się warstwy. Zatem im większy współczynnik uczenia, tym mniejszy czas uczenia. W tym prostym przykładzie w warstwie sigmoidalnej duży współczynnik uczenia zazwyczaj nie tworzy nawet wahań w wartości MSE. W Adaline pojawiają się małe wahania, lecz nadal nie ma to znaczenia przy tak małej ilości epok. Przy dużym współczynniku raz na jakiś czas uczenie staje w miejscu przez to, że aktualizacja wagi może „przekręcić" jej wartość w drugą stronę.

Ponadto okazuje się, że Adaline jest znacznie gorsze w przypadku, jeżeli chcemy testować litery, których sieć nie poznała podczas uczenia. Przez to, że Adaline tak szybko się uczy to dla dużych współczynników uczenia może nie potrafić się nauczyć.

Listing

main.py:

```
from neuron import *
from letters import *
import numpy as np

""" Mean Squared Error function """
def MSE(results,expected):
    sum = 0.0
    for i in range(len(results)):
        sum+=(results[i]-expected[i])**2
    return sum/len(results)

""" Class made for encapsulating input data """
class InputVector:
    def __init__(self, x, d):
        self.__dict__['_x'] = x
        self.__dict__['_d'] = d
    def __getitem__(self, index):
        if index == 'x':
            return self._x
        if index == 'd':
            return self._d

if __name__ == "__main__":

    """ Creating Sigmoidal layer with 3 neurons and Perceptron """
    lmSig = LayerManager(
        2,                                      # number of layers
        [3, 1],                                  # number of neurons in layers
        [35, 3],                                  # number of inputs in layers
        [Sigm()(1.0), Sign()(0.5)],                     # activation functions in layers
        [Sigm().derivative(1.0), Sign().derivative()],  # activation function derivatives in layers
    )

    # Training letters
    lettersInput = [
        LetterInput('a'),
        LetterInput('p'),
```

```
        LetterInput('o'),
        LetterInput('b'),
        LetterInput('A'),
        LetterInput('B'),
        LetterInput('C'),
        LetterInput('I'),
        LetterInput('F'),
        LetterInput('d'),
        LetterInput('c'),
        LetterInput('w'),
        LetterInput('H'),
        LetterInput('K'),
        LetterInput('D')
    ]
    print("Epoch", ",", "MSE error")
    aboveErr = True
    expectedForAllLetters = []

    """ Creating array od expected values for certain pixels
        interD is 3 expected values for subtasks like:
            - does letter exeed left margin of grid?
            - does letter exeed right margin of grid?
            - does letter exeed top margin of grid?
    """
    for j in range(len(lettersInput)):
        expectedForAllLetters.extend(lettersInput[j]._interD)

    epoch = 0
    while(aboveErr):
        epochResults = []
        for j in range(len(lettersInput)):
            results = lmSig.trainLayers([
                # for layer 0:
                InputVector(lettersInput[j]._x, lettersInput[j]._interD),
                # for layer 1:
                InputVector(lettersInput[j]._interD, lettersInput[j]._d)
            ])
            # result[0] is array of results from first layer
            epochResults.extend(results[0])

        """ Calculating MSE error for every epoch"""
        mseVal = MSE(epochResults, expectedForAllLetters)
        if mseVal < 0.0001: # STOP IF MSE ERR IS LESS THAN 0.0001
            aboveErr = False
        epoch += 1
        print(epoch, "," , mseVal)

    """ Validation """
    test = LetterInput('w')
    print("Result:",",",lmSig.processLayers(test._x))
```

neurons.py:

```
import numpy as np
import random
from enum import Enum


""" Sign function which can be translated by given value. Used as
    activation function for perceptron.

    - Parameters:
        - translation: breaking point for the function.

    - Usage:
```

```
        - Sign()(0.5)
            - returns sign function for unipolar sigmoidal function
"""
class Sign:
    def __call__(self, translation):
        def sign(x):
            if x >= translation:
                return 1
            else:
                return 0
        return sign

    def derivative(self):
        def signDeriv(x):
            return 1
        return signDeriv


""" Sigmoidal function & its derivative for given beta. Used as
    Activation function for perceptron.

    - Parameters:
        - beta: sigmoidal function parameter. Its value affects
        function shape. The greater the value the steeper is the function.

    - Usage:
        - Sigm()(0.5)
            - returns: sigm(x) function with beta=0.5
        - Sigm().derivative(0.5)
            - returns sigmDeriv function
            with beta=0.5
"""
class Sigm:
    def __call__(self, beta):
        def sigm(x):
            return 1.0 / (1.0 + np.exp(-beta * x))
        sigm.__name__ += '({0:.3f})'.format(beta)
        return sigm

    def derivative(self, beta):
        def sigmDeriv(x):
            return beta * np.exp(-beta * x) / ((1.0 + np.exp(-beta * x))**2)
        sigmDeriv.__name__ += '({0:.3f})'.format(beta)
        return sigmDeriv


class Neuron:
    """ This is template class for both perceptron and sigmoidal neuron. Perceptron is able
    to specify which class object belongs to, returning only 0 or 1, whereas sigmoidal neuron can return
    every value from 0 to 1. """

    def __init__(self, weights, activFunc, activFuncDeriv, lRate=0.05, bias=random.uniform(-1, 1)):
        self.__dict__['_weights'] = np.array(weights)
        self.__dict__['_activFunc'] = activFunc
        self.__dict__['_activFuncDeriv'] = activFuncDeriv
        self.__dict__['_bias'] = bias
        self.__dict__['_lRate'] = lRate
        self.__dict__['_inputValues'] = None
        self.__dict__['_error'] = None
        self.__dict__['_sum'] = None
        self.__dict__['_val'] = None

    def process(self, input):
        self._inputValues = np.array(input)
        self._sum = np.dot(self._inputValues, self._weights) + self._bias
```

```
        """ Process output """
        self._val = self._activFunc(self._sum)
        return self._val

    def train(self, input, target):
        guess = self.process(input)
        delta = guess - target

        """ Updating weights based on error.
            Gradient learning """
        self._error = self._lRate * delta * self._activFuncDeriv(self._sum)

        for i in range(len(self._weights)):
            self._weights[i] -=  * self._error * input[i]

        self._bias = self._lRate * self._error

    """ Access method """
    def __getitem__(self, index):
        if index == 'val':
            return self._val
        elif index == 'sum':
            return self._sum
        elif index == 'error':
            return self._error


class Layer:
    def __init__(self, numOfNeurons, numOfInputs, activFunc, activFuncDeriv):
        self.__dict__['_neurons'] = []
        self.__dict__['_numOfNeurons'] = numOfNeurons
        self.__dict__['_activFunc'] = activFunc
        self.__dict__['_activFuncDeriv'] = activFuncDeriv
        self.__dict__['_numOfInputs'] = numOfInputs

        """ Creating neurons """
        for n in range(numOfNeurons):
            w = [random.uniform(-1, 1) for _ in range(numOfInputs)]
            self._neurons.append(Neuron(w, activFunc, activFuncDeriv))

    def processNeurons(self, inputs):
        """ Passing data through the neurons of the layer.
        Used for validation. """
        outputs = []
        for n in self._neurons:
            outputs.append(n.process(inputs))
        return outputs

    def trainNeurons(self, inputs, desired):
        """ Passing training data through neurons of the layer. """
        outputs = []
        for index, n in enumerate(self._neurons):
            if self._numOfNeurons > 1:
                n.train(inputs, desired[index])
            else:
                n.train(inputs, desired)
            outputs.append(n._val)
        return outputs


class LayerManager:
    """ Class which manages single layer of neurons and Perceptron (processing
    outputs of the layer). """
    def __init__(self, numOfLayers, numOfNeurons, numOfInputs, activFuncs, activFuncDerivs):
        self.__dict__['_layers'] = []
        self.__dict__['_numOfLayers'] = numOfLayers
```

```
        self.__dict__['_numOfInputs'] = numOfInputs
        self.__dict__['_activFuncs'] = activFuncs
        self.__dict__['_activFuncDerivs'] = activFuncDerivs

        """ Creating single layers """
        for i in range(numOfLayers):
            self._layers.append(
                Layer(numOfNeurons[i], numOfInputs[i], activFuncs[i], activFuncDerivs[i]))

    def processLayers(self, inputs):
        """ Passing data through layers.
        Used for validation. """
        prevOuts = None
        output = []
        for i in range(self._numOfLayers):
            if i == 0:
                prevOuts = self._layers[i].processNeurons(inputs)
                output.append(prevOuts)
            else:
                prevOuts = self._layers[i].processNeurons(prevOuts)
                output.append(prevOuts)
        return output

    def trainLayers(self, inputVectors):
        """ Passing training data through layers. """
        results = []
        for i in range(self._numOfLayers):
            results.append(self._layers[i].trainNeurons(
                inputVectors[i]._x, inputVectors[i]._d))
        return results

    """ Access method """
    def __getitem__(self, index):
        if index == 'layers':
            return self._layers
        elif index == 'numOfLayers':
            return self._numOfLayers
```

letters.py

```
class LetterInput():
    def __init__(self, letter):
        self.__dict__['_x'] = []
        self.__dict__['_d'] = None # expected output for whole task (lowercase/uppercase)
        self.__dict__['_interD'] = None # expected outputs for 3 subtasks
        self.__dict__['_letter'] = letter
        self.getLetter()

    def getLetter(self):
        if self._letter == 'a':
            self._x = [
                0,  0,  0,  0,  0,
                0,  0,  0,  0,  0,
                0,  0,  0,  0,  0,
                0,  0,  1,  1,  0,
                0,  1,  0,  1,  0,
                0,  1,  0,  1,  0,
                0,  1,  1,  1,  1
               ]
            self._interD = [0,0,0]
            self._d = 0


        elif self._letter == 'b':
```

```
    self._x = [
        1,  0,  0,  0,  0,
        1,  0,  0,  0,  0,
        1,  0,  0,  0,  0,
        1,  1,  1,  1,  0,
        1,  0,  0,  1,  0,
        1,  0,  0,  1,  0,
        1,  1,  1,  1,  0
        ]
    self._interD = [1,0,0]
    self._d = 0

elif self._letter == 't':
    self._x = [
        0,  0,  0,  0,  0,
        0,  0,  0,  0,  0,
        0,  0,  0,  0,  0,
        0,  0,  1,  0,  0,
        0,  1,  1,  1,  0,
        0,  0,  1,  0,  0,
        0,  0,  1,  1,  0
        ]
    self._interD = [0,0,0]
    self._d = 0

elif self._letter == 'p':
    self._x = [
        0,  0,  0,  0,  0,
        0,  0,  0,  0,  0,
        0,  0,  0,  0,  0,
        0,  1,  1,  1,  0,
        0,  1,  0,  1,  0,
        0,  1,  1,  1,  0,
        0,  1,  0,  1,  0
        ]
    self._interD = [0,0,0]
    self._d = 0

elif self._letter == 'c':
    self._x = [
        0,  0,  0,  0,  0,
         0,  0,  0,  0,  0,
         0,  0,  0,  0,  0,
         0,  1,  1,  1,  0,
         0,  1,  0,  0,  0,
         0,  1,  0,  0,  0,
         0,  1,  1,  1,  0
        ]
    self._interD = [0,0,0]
    self._d = 0

elif self._letter == 'w':
    self._x = [
        0,  0,  0,  0,  0,
         0,  0,  0,  0,  0,
         0,  0,  0,  0,  0,
         1,  0,  0,  0,  1,
         1,  0,  0,  0,  1,
         1,  0,  1,  0,  1,
         0,  1,  0,  1,  0
        ]
    self._interD = [1,0,1]
    self._d = 0

elif self._letter == 'd':
    self._x = [
```

```
            0,  0,  0,  0,  1,
            0,  0,  0,  0,  1,
            0,  0,  0,  0,  1,
            0,  0,  1,  1,  1,
            0,  1,  0,  0,  1,
            0,  1,  0,  0,  1,
            0,  1,  1,  1,  1
           ]
        self._interD = [0,0,0]
        self._d = 0

    elif self._letter == 'o':
        self._x = [
            0, 0, 0, 0, 0,
            0, 0, 0, 0, 0,
            0, 0, 0, 0, 0,
            0, 1, 1, 1, 0,
            0, 1, 0, 1, 0,
            0, 1, 0, 1, 0,
            0, 1, 1, 1, 0
           ]
        self._interD = [0,0,0]
        self._d = 0

    elif self._letter == 'A':
        self._x = [
            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._interD = [1,1,1]
        self._d = 1

    elif self._letter == 'B':
        self._x = [
            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._interD = [1,1,1]
        self._d = 1

    elif self._letter == 'I':
        self._x = [
            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,
            0,  0,  1,  0,  0
        ]
        self._interD = [0,1,0]
        self._d = 1


    elif self._letter == 'C':
        self._x = [
```

```
            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._interD = [1,1,0]
        self._d = 1


    elif self._letter == 'D':
        self._x= [
            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._interD = [1,1,1]
        self._d = 1

    elif self._letter == 'F':
        self._x = [
        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._interD = [1,  1,  0]
        self._d = 1

    elif self._letter == 'K':
        self._x = [
        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._interD = [1,  1,  1]
        self._d = 1

    elif self._letter == 'H':
        self._x = [
        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._interD = [1,  1,  1]
        self._d = 1
def __getitem__(self, index):
    if index=='x':
        return self._x
```

```
    elif index=='d':
        return self._d
    elif index=='interD':
        return self._interD
```

Źródła

„Sieci neuronowe do przetwarzania informacji” Stanisław Osowski
https://pl.wikipedia.org/wiki/Metoda_najszybszego_spadku
https://sebastianraschka.com/faq/docs/diff-perceptron-adaline-neuralnet.html