Kamil Stachowicz

PSI - Sprawozdanie

Laboratorium nr 2

Opis wykonywanego ćwiczenia
Celem wykonywanego ćwiczenia jest poznanie dwóch nowych typów neuronów: sigmoidalnego i adaline oraz sieci jednowarstwowej złożonej z tych dwóch typów. Zostaną one zaimplementowane oraz przetestowane pod kątem sprawdzania wielkości liter.

Schemat ćwiczenia
1. Zaimplementowanie perceptronu sigmoidalnego
2. Zaimplementowanie ADALINE (**Ada**ptive **Li**near **Ne**uron)
3. Zaimplementowanie sieci jednowarstwowej dla obu typów neutronów
4. Uczenie sieci za pomocą danych testujących
5. Sporządzenie tabeli z błędami sieci (MSE)

Wykonanie ćwiczenia
Oba z neuronów zostały zaimplementowane bazując na materiałach dostępnych w książce Stanisława Osowskiego - sieci neuronowe do przetwarzania informacji.

Neuron sigmoidalny jest budową zbliżony do perceptronu, różni się on przede wszystkim funkcją aktywacji i zwracaną wartością. Wykorzystuje on ciągłą funkcję aktywacji: uni- lub bi-polarną. Podczas ćwiczenia wykorzystana została unipolarna funkcja aktywacji w postaci:

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

Funkcja zmiany wagi różni się też nieco od tej w perceptronie, mianowicie w neuronie sigmoidalnym do obliczenia nowej wagi wykorzystywana jest pochodna funkcji aktywacji.

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

gdzie:
i - numer neutronu
j - numer wejścia
w - waga
η - learning rate
δ - pochodna funkcji aktywacji

$$\delta = \frac{\mathrm{d}f}{\mathrm{d}x} = \frac{e^{\beta x}}{(e^{\beta x} + 1)^2}$$

Adaline działaniem ani budową nie różni się znacznie od perceptronu czy neuronu sigmoidalnego. Funkcją aktywacji jest signum. Jej największą różnicą jest to że wagi nie mają tak jak poprzednie neurony wartości z zakresu <-1, 1>, różni się również postać funkcji obliczania błędu.

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

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

gdzie:
d - wektor z oczekiwanymi wyjściami
e - błąd neuronu

Oba neurony były uczone z nauczycielem.

Do trenowania neuronów posłużył wykonany do ćwiczenia zestaw danych z małymi i dużymi literami w tablicy o wymiarach 5x7. Każdy z neuronów posiadał 35 wejść na te dane. W obu sieciach do rozpoznawania wielkości liter posłużyły po 3 neurony.

Do analizy wyjścia dodany został dodatkowy neuron poza siecią do łatwiejszej weryfikacji czy litera jest duża czy mała.

Podczas trenowania dla każdej epoki liczony był błąd średniokwadratowy. Sieć uczyła się dopóki nie spadł on poniżej wartości 0,001. Wszystkie wyniki zostały załączone w postaci pliku excel.

Wnioski

Na podstawie wyników otrzymanych w trakcie ćwiczenia można stwierdzić że sieć złożona z neuronów adaline uczy się szybciej rozwiązywać problem rozpoznawania wielkości liter. Niemniej jednak może być to twierdzenie mylne albowiem postać wyjść każdej z sieci różni się od siebie. Neuron adaline zwraca jedynie wartość 1 lub -1 zwracanej przez funkcję signum, a neuron sigmoidalny zwracał wartości z przedziału <0,1>. Można więc stwierdzić że dokładność rozwiązania, mimo niskiego błędu w sieci z Adaline, jest wyższa w przypadku neuronu sigmoidalnego. Potrzebował on bardzo dużej liczby epok do nauczenia się i spełnienia warunku błędu średniokwadratowego (< 0,001). Była to liczba ponad 32 tysięcy epok.

Dobranie odpowiedniego learning rate zmniejszyło skalę problemu czasu uczenia się przez neuron sigmoidalny, dla learning rate równego 0,5 sieć potrzebowała 53 krotnie mniej epok rozwiązując problem z podobną dokładnością co neuron uczony przy learning rate równym 0,01. W przypadku adaline przy learning rate 0,01 potrzebowała zaledwie 30 epok po czym dorównała sieci neuronów sigmoidalnych pod względem błędu średniokwadratowego.

Język programowania python okazuje się być pomocny w realizacji projektów z zakresu sieci neuronowych. Jego prosta składnia oraz bardzo przewidywalny sposób działania umożliwiają szybkie znajdywanie błędów w kodzie oraz logice matematycznej wymaganej do zaimplementowania sieci neuronowych. Pomaga też fakt że jest on językiem skryptowym, tak więc nie wymaga kompilacji, co znacznie przyspiesza sprawdzanie działania programu.

Kod programu

#plik z klasą neutronu sigmoidalnego

```
#!/usr/bin/python
import random
from math import exp
from sigm import Sigm
from sign import Sign
import numpy as np

class Perceptron:
    def __init__(self, learning_rate, no_of_inputs, activation_function, activation_function_der):
        self.__dict__['_no_of_inputs'] = no_of_inputs
        self.__dict__['_weights'] = []
        self.__dict__['_inputs'] = []
        self.__dict__['_learningRate'] = learning_rate
        self.__dict__['_activationFunction'] = activation_function              #funkcje aktywacji oraz pochodna zostały
        self.__dict__['_activationFunctionDer'] = activation_function_der       #przekazane przez konstruktor
        self.__dict__['_bias'] = -1 * np.random.ranf()                          #ma to służyć uproszczeniu tworzenia kolejnych
        self.__dict__['_sum'] = None                                            #projektów
        self.__dict__['_error'] = None

        for weight in range(0, self._no_of_inputs):
            self._weights.append(random.uniform(-1,1))              #losowanie wag

    def guess(self, inputs):
        self._inputs = inputs
        self._sum = np.dot(self._weights, self._inputs) + self._bias #sumowanie oraz wymnażanie wektora wejść i wag
        return self._activationFunction(self._sum)

    def train(self, inputs, desiredOutput):
        output = self.guess(inputs)
        delta = (desiredOutput - output)
        self._error = delta * self._activationFunctionDer(self._sum) #implementacja liczenia błędu dla perceptronu sigmoid.
```

```
        for i in range(len(self._inputs)):
            self._weights[i] += self._error * self._inputs[i] * self._learningRate #dyskretna zmiana wartości wag

        self._bias = self._learningRate * self._error                              #zmiana wartości wagi polaryzacji

        return output
```

#plik z funkcją aktywacji perceptronu sigmoidalnego

```
import numpy as np
class Sigm:
    def __call__(self, beta):
        def sigm(x):                            #implementacja funkcji aktywacji neuronu sigmoidalnego
            return 1.0/(1.0+np.exp(-beta*x))
        sigm.__name__ += '({0:.3f})'.format(beta)
        return sigm
    def derivative(self, beta):
        def sigmDeriv(x):                       #implementacja pochodnej funkcji aktywacji
            return beta*np.exp(-beta*x)/((1.0+np.exp(-beta*x))**2)
        sigmDeriv.__name__ += '({0:.3f})'.format(beta)
        return sigmDeriv
```

#plik z adaline

```
#!/usr/bin/python
import random
import numpy as np

class Adaline:
    def __init__(self, learning_rate, no_of_inputs):
        self.__dict__['_no_of_inputs'] = no_of_inputs
        self.__dict__['_weights'] = []
        self.__dict__['_inputs'] = []
        self.__dict__['_learningRate'] = learning_rate
        self.__dict__['_bias'] = 1
        self.__dict__['_sum'] = None
        self.__dict__['_error'] = None

        for weight in range(0, self._no_of_inputs):
            self._weights.append(random.uniform(-1,1))

    def guess(self, inputs):                                    #funkcja przetwarzania nie różni się niczym od funkcji
        self._inputs = inputs                                   #zaimplementowanej w perceptronie z lab. nr 1
        self._sum = np.dot(self._weights, self._inputs) + self._bias
        return self.sign(self._sum)

    def train(self, inputs, desiredOutput):
        output = self.guess(inputs)
        self._error = desiredOutput - self._sum                 #obliczanie błędu Adaline
        for i in range(len(self._inputs)):                      #uaktualnianie wag odybwa się w ten sam sposób co w
                                                                #perceptronie
            self._weights[i] += self._error * self._inputs[i] * self._learningRate

        self._bias = self._learningRate * self._error
        return output

    def sign(self, x):
            if x > 0:
                return 1
            else:
                return -1
```

#plik z implementacją sieci jednowarstwowej (wspólny dla obu typów neuronów)

```
#!/usr/bin/python

from perceptron import Perceptron
```

```
from testinput import TestInput

class SingleLayer:
    def __init__(self, no_of_layer, no_of_perceptrons, no_of_inputs, learning_rate, activation_function,
activation_function_der):
        self.__dict__['_perceptrons'] = []
        self.__dict__['_testInputs'] = []
        self.__dict__['_perceptronOutputs'] = []
        self.__dict__['_activationFunction'] = activation_function
        self.__dict__['_activationFunctionDer'] = activation_function_der
        self.__dict__['_noOfInputs'] = no_of_inputs
        self.__dict__['_learningRate'] = learning_rate
        self.__dict__['_noOfPerceptrons'] = no_of_perceptrons
        self.__dict__['_thisLayerNo'] = no_of_layer

        for i in range(self._noOfPerceptrons)
          self._perceptrons.append(Perceptron(self._learningRate, self._noOfInputs, self._activationFunction,
self._activationFunctionDer))

    def getPerceptron(self, index_of_perceptron):
        if index_of_perceptron < 0 or index_of_perceptron >= len(self._perceptrons) :
                return None
        else:
        return self._perceptrons[index_of_perceptron]

    def trainPercpeptrons(self, inputs):
        if(self._thisLayerNo != 0):
                for x in range(0, 200):
                        for inp in inputs:
                                perceptronCounter = 0

                                for perc in self._perceptrons:
                                        perc.train(
                                                inp._testArguments[self._thisLayerNo],
                                                inp._testArguments[self._thisLayerNo + 1][perceptronCounter]
                                        )
                                        perceptronCounter += 1

        mse = 1;
        epoch = 0
        while(mse > 0.001):
                epoch += 1
                print("Epoch;",epoch,";",end='')
                for inp in inputs:
                        perceptronCounter = 0

                        outputs = []
                        for perc in self._perceptrons:
                                outputs.append(perc.train(
                                        inp._testArguments[self._thisLayerNo],
                                        inp._testArguments[self._thisLayerNo + 1][perceptronCounter]
                                ))
                                perceptronCounter += 1
                        mse+= self.MSE(outputs,inp._testArguments[self._thisLayerNo + 1])
                mse = mse/len(inputs)
                print("MSE;",mse,";")

    def guess(self, inputs):
        self._perceptronOutputs = []
        for perc in self._perceptrons:
                self._perceptronOutputs.append(perc.guess(inputs))

        return self._perceptronOutputs
    #Mean squared error
    def MSE(self, result, expected):
        sum = 0
```

```
            for i in range(len(result)):
                    sum += (result[i] - expected[i])**2
            return sum
```

#plik z danymi wejściowymi

```
#!/usr/bin/python


class TestInput():
    """docstring forTestInput."""
    """
        Test input is a vector which represents capital and small letters like as in
        5x7 table
    """
    availableLetters = ['a', 'A', 'b', 'B', 'o', 'C', 'D', 'I', 'F', 'h', 'U', 'K', 'd', 'H', 'c', 'G', ,w'] #dostępne litery
    def __init__(self, letter):
        self.__dict__['_testArguments'] = []
        self.__dict__['_letterOfTest'] = letter

        self.getLetter()

    def getLetter(self):
        if self._letterOfTest == 'a':
            self._testArguments.append([            #wejście dla pierwszej warstwy
                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,  1
               ])
            self._testArguments.append([ 0,  0,  1])        #oczekiwane wyjście pierwszej warstwy/wejście drugiej warstwy
            self._testArguments.append([0])                 #wartość oczekiwana dla ostatniego neuronu

        elif self._letterOfTest == 'b':
            self._testArguments.append([
                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._testArguments.append([1,  0,  0])
            self._testArguments.append([0])
        elif self._letterOfTest == 'o':
            self._testArguments.append([
                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._testArguments.append([0,  0,  0])
            self._testArguments.append([0])
        elif self._letterOfTest == 'w':
            self._testArguments.append([
                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._testArguments.append([1,  0,  1])
        self._testArguments.append([0])
    elif self._letterOfTest == 'c':
        self._testArguments.append([
            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._testArguments.append([0,  0,  0])
        self._testArguments.append([0])
    elif self._letterOfTest == 'h':
        self._testArguments.append([
            1,  0,  0,  0,  0,
            1,  0,  0,  0,  0,
            1,  0,  0,  0,  0,
            1,  1,  1,  0,  0,
            1,  0,  1,  0,  0,
            1,  0,  1,  0,  0,
            1,  0,  1,  0,  0
           ])
        self._testArguments.append([1,  0,  0])
        self._testArguments.append([0])
    elif self._letterOfTest == 'd':
        self._testArguments.append([
            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._testArguments.append([0,  0,  1])
        self._testArguments.append([0])

    elif self._letterOfTest == 'A':
        self._testArguments.append([
            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._testArguments.append([1,  1,  1])
        self._testArguments.append([1])
    elif self._letterOfTest == 'B':
        self._testArguments.append([
        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._testArguments.append([1,  1,  1])
        self._testArguments.append([1])
```

```
        elif self._letterOfTest == 'C':
            self._testArguments.append([
             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._testArguments.append([1,  1,  1])
            self._testArguments.append([1])

        elif self._letterOfTest == 'D':
            self._testArguments.append([
            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._testArguments.append([1,  1,  1])
            self._testArguments.append([1])
        elif self._letterOfTest == 'I':
            self._testArguments.append([
            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._testArguments.append([0,  1,  0])
            self._testArguments.append([1])
        elif self._letterOfTest == 'F':
            self._testArguments.append([
            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._testArguments.append([1,  1,  0])
            self._testArguments.append([1])
        elif self._letterOfTest == 'G':
            self._testArguments.append([
            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._testArguments.append([1,  1,  1])
            self._testArguments.append([1])
        elif self._letterOfTest == 'H':
            self._testArguments.append([
            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._testArguments.append([1,  1,  1])
            self._testArguments.append([1])
        elif self._letterOfTest == 'K':
            self._testArguments.append([
            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._testArguments.append([1,  1,  1])
            self._testArguments.append([1])
        elif self._letterOfTest == 'U':
            self._testArguments.append([
            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._testArguments.append([1,  1,  1])
            self._testArguments.append([1])

    def makeTestInputs(no_of_tests):            #statyczna funkcja tworząca dane wejściowe
        testInputsArray = []
        x = 0
        for i in range(0, no_of_tests):
            testInputsArray.append(TestInput(TestInput.availableLetters[x]))
            x += 1
            if x == len(TestInput.availableLetters):
                x = 0
        return testInputsArray
```