WYDZIAŁ
ELEKTROTECHNIKI
I INFORMATYKI
POLITECHNIKI RZESZOWSKIEJ

Karol Dudek

Realizacja sieci neuronowej uczonej algorytmem wstecznej propagacji błędu ucząca się rozpoznawać rodzaj schorzenia u pacjenta na podstawie wyników jego badań.

Praca projektowa z przedmiotu "Sztuczna Inteligencja"

Opiekun pracy:
dr hab. inż. Roman Zajdel, prof. PRz

Rzeszów, 2022

Spis treści

1. Wstęp

1.1. Cel projektu

Głównym założeniem projektu jest realizacja sieci neuronowej uczonej za pomocą algorytmu wstecznej propagacji błędu, której zadaniem jest zdiagnozowanie zapalenia nerek lub zapalenia pęcherza na podstawie pewnych danych wejściowych. W ramach projektu zbadano wpływ poszczególnych parametrów sieci na proces jej uczenia:

- S1 - ilość neuronów w I warstwy ukrytej
- S2 - ilość neuronów w II warstwy ukrytej
- lr (learning rate) / eta - wartość współczynnika uczenia
- target error - maksymalny docelowy błąd sieci

Jako zbiór danych uczących wykorzystano zbiór "Acute Inflammations”, a sama sieć została zrealizowana przy użyciu języka programowania Python.

1.2. Opis problemu

Realizowana sieć na podstawie podanych informacji wejściowych ma za zadanie sklasyfikować, do której klasy należą te dane i na wyjściu podać informacje o rodzaju zdiagnowzowanej choroby. W zbiorze danych uczących występuje 4 możliwe klasy:

- brak chorób
- występuje zapalenie nerek
- występuje zapalenie pęcherza
- występuje jednocześnie zapalenie pęcherza i nerek

1.3. Opis zestawu danych

Zestaw danych uczących został pobrany ze strony:

https://archive.ics.uci.edu/ml/datasets/Acute+Inflammations

Zbiór danych uczących zawiera 120 rekordów po 6 atrybutów w każdym wierszu. W przypadku atrybutów, będących parametrami wejściowymi są to:

- a1 - Temperatura pacjenta
- a2 - Zawroty głowy
- a3 - Ból lędzwiowy
- a4 - Ciągła potrzeba oddania moczu
- a5 - Bóle mikcyjne
- a6 - Pieczenie cewki moczowej

Oraz dwie możliwe decyzje wyjściowe:

- d1 - Zapalenie pęcherza
- d2 - Zapalenie nerek

1.4. Przygotowanie danych

Badany zestaw danych nie zawiera niekompletnych rekordów, oraz wartości niepoprawnych, dlatego też podczas przygotowywania danych nie napotkano żadnych nieprzewidzianych błędów.

W celu ujednolicenia danych do poźniejszego przetwarzania na danych przeprowadzono normalizację zgodnie z poniższym wzorem:

$$x_{norm} = \frac{x - x_{min}}{x_{max} - x_{min}} \tag{1.1}$$

gdzie:

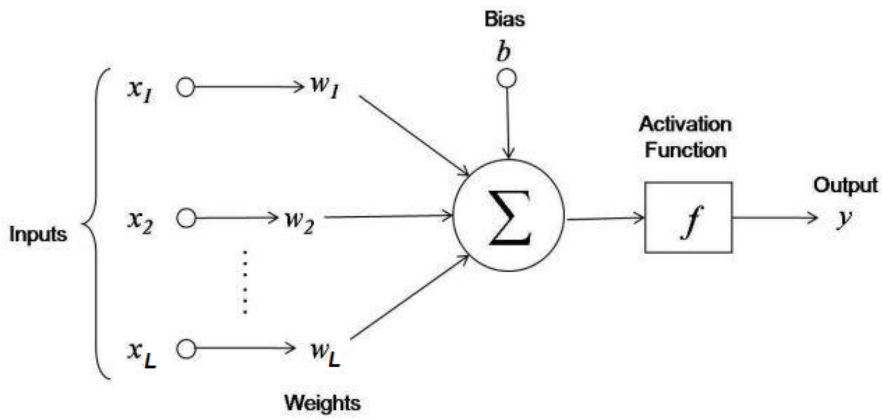
- x_{norm} - wartość po normalizacji
- x - wartośc przed normalizacją
- x_{min} - minimalna wartość w zbiorze
- x_{max} – maksymalna wartość w zbiorze

Dzięki zastosowaniu wzoru **??** pierwotne dane wejściowe każdego rekordu zostały zamienione na odpowiadające im wartości z zakresu od 0 do 1.

2. Zagadnienia teorytyczne

2.1. Model sztucznego neuronu

Każda sieć neuronowa składa się z połączonych między sobą pojedynczych neuronów. Należy zatem zapoznać się z modelem pojedynczego neuronu w celu zrozumienia problemu sieci neuronowych. Przykładowy model pojedynczego neuronu został przedstawiony na rysunku **??**



Rysunek 2.1: Model pojedynczego neuronu

Każdy neuron to układ składający się z wielu wejść i jednego wyjścia. Wszystkie wejścia posiadają tzw. współczynnik wagowy, który określa jak bardzo dane wejście wpływa na rezultat wynikowy danego neuronu. Oprócz wagi dodatkowym elementem wejściowych jest również bias, umożliwiający przesunięcie funkcji aktywacji w lewo lub w prawo danego neuronu. Całość z poprzednio podanych informacji do wejść neuronu trafia do sumatora gdzie odbywa się proces wyznaczania łącznego pobudzenia neuronu, wyrażanego z następującego wzoru **??**:

$$z = \sum_{j=1}^{L} w_j x_j + b \tag{2.2}$$

Wyznaczona wartość następnie trafia do funkcji aktywacji, gdzie określany jest sygnał wyjściowy neuronu zgodnie z zależnością **??**:

$$y = f(z) = f(\sum_{j=1}^{L} w_j x_j + b) \tag{2.3}$$

gdzie:

- y - wyjście neuronu
- x_j - j-ty sygnał wejściowy (j=*1*,*2*,$\ldots$,L)
- w_j - waga j-tego wejścia
- b - bias

2.2. Funkcja aktywacji

Każda warstwa w swoich neuronach może wykorzystywać inną funkcję aktywacji. W przypadku sieci jednokierunkowych najbardziej powszechna jest funkcja sigmoidalna, w której wyróżnić można dwa typy:

- unipolarna funkcja aktywacji, która przyjmuje wartości w przedziału (0,1):

$$f(x) = \frac{1}{1+e^{-x}} \tag{2.4}$$

- bipolarna funkcja aktywacji, przyjmująca wartości z przedziału (-1,1):

$$f(x) = \frac{2}{1+e^{-x}} - 1 \tag{2.5}$$

Funkcje te są różniczkowalne i ich pochodne wyrazić można jako:

- w przypadku unipolarnej funkcji aktywacji:

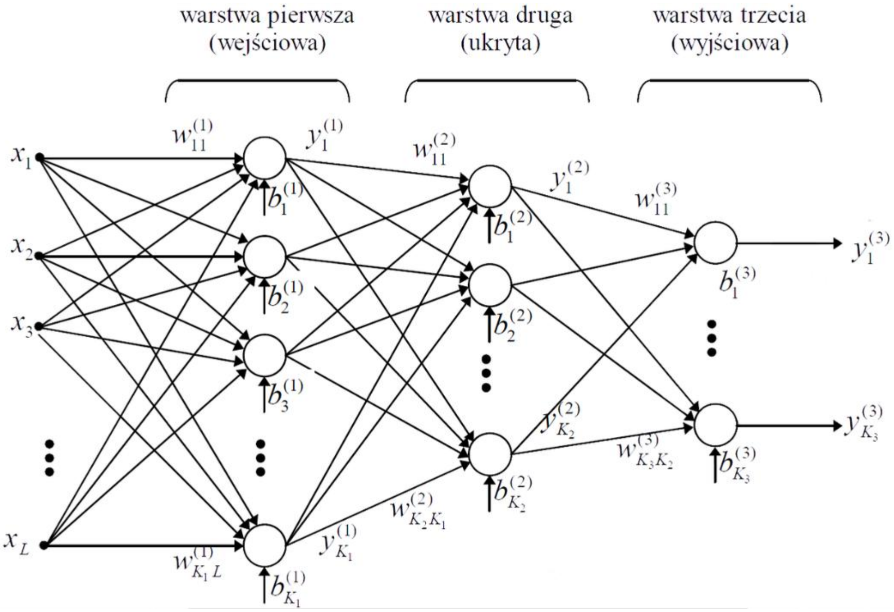
$$f(x) = \frac{1}{1+e^{-x}}(1 - \frac{1}{1+e^{-x}}) \tag{2.6}$$

- w przypadku bipolarnej funkcji aktywacji:

$$f(x) = 1 - (\frac{2}{1+e^{-x}})^2 \tag{2.7}$$

2.3. Model sieci wielowarstwowej

W każdej sieci jednokierunkowej wielowarstwowej wyróżnić można nastepujące warstwy: wejścia, wyścia i występujące pomiędzy nimi warstwy ukryte. W przypadku sieci jednokierunkowej dane mogą przepływać tylko w jednym kierunku od warstwy wejścia do warstwy wyjścia.



Rysunek 2.2: Model sieci jednokierunkowej wielowarstwowej

Każda pojedyncza warstwa neuronów posiada:

- macierz wag **w**:

$$w = \begin{bmatrix} w_{11} & w_{12} & \dots & w_{1L} \\ w_{21} & w_{22} & \dots & w_{2L} \\ \vdots & \vdots & & \vdots \\ w_{K1} & w_{K2} & \dots & w_{KL} \end{bmatrix}$$

gdzie:

- K – liczbę neuronów w warstwie
- L – liczba sygnałów wejściowych

- wektor przesunieć **b**:

$$b = \begin{bmatrix} b_1 & b_2 & \dots & b_K \end{bmatrix}^T$$

- funkcje aktywacji **f**

- oraz wektor sygnałów wyjściowych **y**:

$$y = \begin{bmatrix} y_1 & y_2 & \dots & y_K \end{bmatrix}^T$$

W celu obliczenia sygnału wyjściowego sieci wielowarstwowej uogólniony wzór na wyjście pojedynczej warstwy przedstawiający się jako:

$$y^{(n)} = f^{(n)}(w^{(n)}y^{(n-1)} + b^{(n)}) \tag{2.8}$$

gdzie n to numer odpowiedniej warstwy.

Dla przykładu dla przedstawionej sieci **??** wzory opisujące wyjścia poszczególnych warstw przyjmą postać:

$$y^{(1)} = f^{(1)}(w^{(1)}x + b^{(1)}) \tag{2.9}$$

$$y^{(2)} = f^{(2)}(w^{(2)}y^{(1)} + b^{(2)}) \tag{2.10}$$

$$y^{(3)} = f^{(3)}(w^{(3)}y^{(2)} + b^{(3)}) \tag{2.11}$$

Na podstawie powyższych równań sygnał wyjściowy całej sieci **??** można opisać wzorem:

$$y^{(3)} = f^{(3)}(w^{(3)}f^{(2)}(w^{(2)}f^{(1)}(w^{(1)}x + b^{(1)}) + b^{(2)}) + b^{(3)}) \tag{2.12}$$

2.4. Uczenie sieci pod nadzorem (supervised learning)

2.5. Algorytm wstecznej propagacji błędu

3. Realizacja sieci

3.1. Opis skryptu

```python
from sklearn.model_selection import train_test_split
from csv import reader
import numpy as np

# Import function
def dataImport(name):
    with open(name, 'r', encoding='utf-16') as file:
        return [line for line in reader(file, delimiter='\t')]

# Normalization function
def normalizeMinMax(table):
    for row in range(0, len(table)):
        min_val, max_val = min(table[row]), max(table[row])
        table[row] = [(1 - 0) * (col - min_val) / (max_val - min_val) for col in table
    [row]]
    return table

# Data loader function
def loadData():
    # Import acute.tsv to dataFile
    dataFile = dataImport('acute.tsv')

    # Create numpy array from dataList
    dataFile = np.array(dataFile)

    # Convert array of strings to array of floats
    dataFile = dataFile.astype(float)

    # Data normalization
    dataFile = normalizeMinMax(dataFile.T).T

    # Data splitting into training and test data
    trainData, testData = train_test_split(dataFile, test_size=0.2, random_state=25)

    # Splitting data into 2 groups, inputData and outputdata
    testIn, testOut = testData[:,:6], testData[:,6:]
    trainIn, trainOut = trainData[:,:6], trainData[:,6:]

    # Combining inputData and outputData in a single tuple
    trainData = [(np.array(trainIn[i], ndmin=2).T, np.array(trainOut[i], ndmin=2).T)
    for i in range(0, len(trainOut))]
    testData = [(np.array(testIn[i], ndmin=2).T,np.array(testOut[i], ndmin=2).T) for i
     in range(0, len(testOut))]

    return (trainData, testData)
```

Listing 1: Plik przygotowujący dane- data.py

```
import random
import time
import numpy as np

class Network(object):

    # Constructor, takes list of layers and amount of neurons as parameter
    def __init__(self, sizes):

        #Applying Seed
        np.random.seed(7)

        # Assing 'sizes' vector to amount of layers in the network
        self.num_layers = len(sizes)
        self.sizes = sizes

        # Pseudo random generator used to assign weight and biases
        self.biases = [np.random.randn(y, 1) for y in sizes[1:]]
        self.weights = [np.random.randn(y, x)
                        for x, y in zip(sizes[:-1], sizes[1:])]

    def feedforward(self, a):

        # Return neural network results for 'a' data
        for b, w in zip(self.biases, self.weights):
            a = sigmoid(np.dot(w, a)+b)
        return a

        # Mean Square Error
    def mse(self,_test_data):
        error=[pow(np.linalg.norm(self.feedforward(x)-y),2) for (x,y) in _test_data]
        return 1/len(_test_data)*sum(error)

    def SGD(self, training_data, epochs, mini_batch_size, eta,
            error_target=0.001, test_data=None):

        if test_data: n_test = len(test_data)
        n = len(training_data)
        for j in range(epochs):
            time1 = time.time()
            random.shuffle(training_data)
            mini_batches = [
                training_data[k:k+mini_batch_size]
                for k in range(0, n, mini_batch_size)]
            for mini_batch in mini_batches:
                self.update_mini_batch(mini_batch, eta)
            cur_err = self.mse(training_data)
```

```
            time2 = time.time()
            evalVal = self.evaluate(test_data)
            evalAcc = (evalVal/n_test*100)
            if cur_err < error_target or j == epochs-1:
                if test_data:
                    print("{0}, {2:.2f}, {3:.0f}%".format(
                        j, cur_err, evalAcc))
                    pass
                else:
                    print("Epoch {0} complete in {1:.2f} seconds".format(j, time2-
time1))
                break

            print("{0}, {1:.6f}, {2:.0f}%".format(j, cur_err, evalAcc))

    def update_mini_batch(self, mini_batch, eta):

        # Updates weights and biases using SGD and backpropagation for each mini batch
        nabla_b = [np.zeros(b.shape) for b in self.biases]
        nabla_w = [np.zeros(w.shape) for w in self.weights]
        for x, y in mini_batch:
            # Calculate gradient increase for each (x, y) pair
            delta_nabla_b, delta_nabla_w = self.backprop(x, y)

            # Calculate new gradient
            nabla_b = [nb+dnb for nb, dnb in zip(nabla_b, delta_nabla_b)]
            nabla_w = [nw+dnw for nw, dnw in zip(nabla_w, delta_nabla_w)]

        # New weights and biases
        self.weights = [w-(eta/len(mini_batch))*nw
                        for w, nw in zip(self.weights, nabla_w)]
        self.biases = [b-(eta/len(mini_batch))*nb
                       for b, nb in zip(self.biases, nabla_b)]

    def backprop(self, x, y):

        #Return tuple representing the gradient of the cost function
        nabla_b = [np.zeros(b.shape) for b in self.biases]
        nabla_w = [np.zeros(w.shape) for w in self.weights]

        # feedforward
        activation = x
        activations = [x] # list to store all the activations, layer by layer
        zs = [] # list to store all the z vectors, layer by layer

        # Calculate neuron activations
        for b, w in zip(self.biases, self.weights):
            z = np.dot(w, activation)+b
            zs.append(z)
            activation = sigmoid(z)
```

```
            activations.append(activation)

        # backward pass (gradient increase for output layer)
        delta = self.cost_derivative(activations[-1], y) * \
            sigmoid_prime(zs[-1])
        nabla_b[-1] = delta
        nabla_w[-1] = np.dot(delta, activations[-2].transpose())

        # Calculate gradient increase for input and hidden layers
        for l in range(2, self.num_layers):
            z = zs[-l]
            sp = sigmoid_prime(z)
            delta = np.dot(self.weights[-l+1].transpose(), delta) * sp
            nabla_b[-l] = delta
            nabla_w[-l] = np.dot(delta, activations[-l-1].transpose())
        return (nabla_b, nabla_w)

    def evaluate(self, test_data):

        test_results = [(self.feedforward(x), y)
                        for (x, y) in test_data]

                        # Approximation
        return sum(int((y[0] == 0 and x[0] < 0.5) or (y[0] == 1 and x[0] > 0.5) and
                       (y[1] == 0 and x[1] < 0.5) or (y[1] == 1 and x[1] > 0.5))
                   for (x, y) in test_results)

    def cost_derivative(self, output_activations, y):
        # Return vector with difference between the neuron and the expected result
        return (output_activations-y)

#### Miscellaneous functions
def sigmoid(z):
    # Sigmoid function
    return 1.0/(1.0+np.exp(-z))

def sigmoid_prime(z):
    # Sigmoid prime function
    return sigmoid(z)*(1-sigmoid(z))
```

Listing 2: Plik zawierający sieć - network.py

```
import data
import network

import numpy as np

trainData, testData = data.loadData()

# [input vector size, S1 neurons, S2 neurons, output]
```

```
net = network.Network([6,2])

# (training_data, epochs, batch_size, eta, target, test_data)
net.SGD(trainData, 100000, 1, 0.1, error_target=0.179,test_data=testData)
```

Listing 3: Plik wywołujący przykładową sieć - main.py

4. Eksperymenty

4.1. Eksperyment 1

Celem pierwszego eksperymentu

4.2. Eksperyment 2

4.3. Eksperyment 3

4.4. Eksperyment 4

4.5. Eksperyment 5

4.6. Eksperyment 6

4.7. Eksperyment 7

4.8. Eksperyment 8

5. Wnioski

Literatura

[1] https://archive.ics.uci.edu/ml/datasets/Acute+Inflammations

[2] Michael Nielsen, Neural Networks and Deep Learning.

[3] Zajdel.R „Ćwiczenie 6 Model Neuronu”, Rzeszów, KIiA, PRz

[4] Zajdel.R „Ćwiczenie 8 Sieć jednokierunkowa jednowarstwowa”, Rzeszów,KIiA,PRz

[5] Zajdel.R „Ćwiczenie 9 Sieć jednokierunkowa wielowarstwowa”, Rzeszów,KIiA,PRz

[6] R.Tadeusiewicz, M.Szaleniec „Leksykon sieci neuronowych”