# Zadanie nr 2 - Sieć MADALINE do rozpoznawania znaków

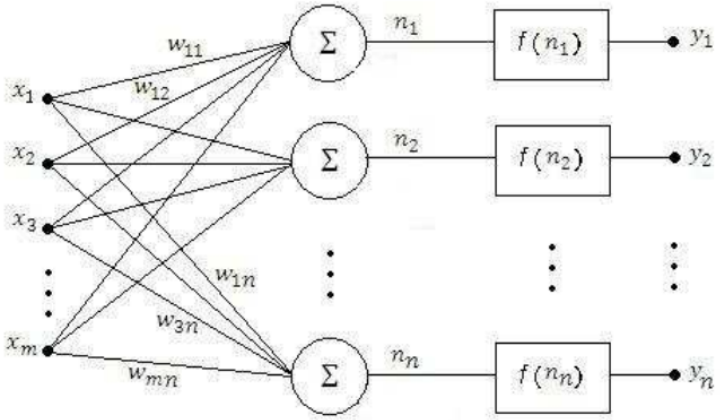
Inteligentne przetwarzanie danych

Mateusz Szczęsny, 233266 Dawid Wójcik, 233271

17.11.2019 r.

# 1 Cel zadania

Celem zadania jest napisanie programu, który imituje sieć MADALINE do rozpoznawania znaków. Sieć powinna implementować strukturę przedstawioną Rysunku 1.




Rysunek 1: Sieć MADALINE

# 2 Wstęp teoretyczny

Sieć powinna składać się z

- $m$ neuronów w warstwie wejściowej odpowiadających kolejnym pikselom analizowanego znaku,
- $n$ neuronów w warstwie wyjściowej odpowiadających za rozpoznawanie kolejnych znaków.

Liczba $m$ może zostać wyrażona równaniem $m = p * q$, gdzie $p$ i $q$ stanowią kolejno poziomy oraz pionowy wymiar analizowanej matrycy. W analizowanym przez nas przykładzie, każdy znak jest wyrysowany na matrycy o wymiarach 16 na 16 pikseli, co daje łączną sumę 256 neuronów znajdujących się w warstwie wejściowej.

Liczba $n$ stanowi ilość neuronów odpowiadających za rozpoznawanie poszczególnych znaków. W analizowanych przez nas przykładzie warstwa wyjściowa zawierała będzie 3 takie neurony.

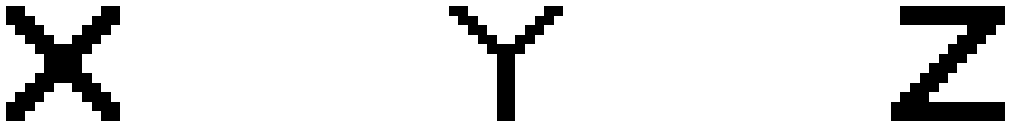
Warstwa wejściowa jest w pełni połączona z warstwą wyjściową. Oznacza to, że każdy piksel zostanie przeanalizowany przez każdy neuron warstwy wyjściowej, warząc na końcowym wyniku.
Każdy obraz procesowany przez sieć zostanie spłaszczony do wektora jednowymiarowego składającego się z jedynek (1) oraz zer (0), które kolejno będą oznaczały obecność czarnego piksela lub jego brak.
Wynikiem przewidywań dokonanych przez sieć będzie wektor składający się z 3 liczb należących do przedziału $< 0, 1 >$ świadczącym o prawdopodobieństwie analizowanego znaku z oryginalną matrycą.

# 3 Przebieg eksperymentu

Eksperyment polega na przeanalizowaniu za pomocą zaimplementowanej sieci neuronowej kilku matryc takich jak:

- matryca identyczna jak jedna z prototypów znajdujących się w neuronie w warstwie wyjściowej,
- matryca zawierająca literę z nałożonym efektem ziarna pogarszającym jakość analizowanego obrazu,
- negatyw jednej z matryc będących prototypem.

W pierwszym etapie eksperymentu zainicjowana zostanie sieć neuronowa z pomocą 3 plików zawierających prototypy matryc dla znaków **X**, **Y** i **Z**.



Rysunek 2: Matryce liter X, Y, Z.

Inicjalizacja zostanie wykonana za pomocą poniższego kodu źródłowego.

```
# main.py
from nn import NeuralNetwork

BRAIN = NeuralNetwork.from_file("img/x.bmp", "img/y.bmp", "
    img/z.bmp")

# nn.py
class NeuralNetwork:
    def __init__(self):
        self.weights = []

    @classmethod
    def from_file(cls, *args):
        nn = cls()

        for arg in args:
            x = np.array(Image.open(arg))

            x_flatten = []
            for row in x:
                for pixel in row:
                    if pixel == True:
                        x_flatten.append(0)
                    else:
                        x_flatten.append(1)

            x_flatten = normalize(x_flatten)

            nn.weights.append(x_flatten)
        nn.weights = np.array(nn.weights, dtype=float)
        return nn
```

Metoda statyczna *from_file* odpowiada za zainicjowanie nowej sieci w oparciu o istniejące matryce znakowe. Jej zadanie jest kolejno wczytać pliki przekazane jako argumenty funkcji, przekonwertować mapę bitową na wektor zer oraz jedynek, a następnie znormalizować wynik otrzymany w poprzednich etapach w celu otrzymania satysfakcjonującego wyniku.
Funckja odpowiadalna za normalizację wektora wygląda następująco:

```
def normalize(inputs: List):
    unique, counts = np.unique(inputs, return_counts=True)
    sum_of_ones = dict(zip(unique, counts))[1]
    return np.where(np.array(inputs) == 1, 1 / math.sqrt(
    sum_of_ones), inputs)
```

Powyższy kod zamienia wektor do postaci znormalizowanej w następujący sposób:

$$\hat{X} = [1001\ 0110\ 0110\ 1001]$$

$$X = \frac{1}{\sqrt{8}} * \hat{X} = \frac{1}{\sqrt{8}} * [1001\ 0110\ 0110\ 1001]$$

$$W = X = [\frac{1}{\sqrt{8}}00\frac{1}{\sqrt{8}}\ \ 0\frac{1}{\sqrt{8}}\frac{1}{\sqrt{8}}0\ \ 0\frac{1}{\sqrt{8}}\frac{1}{\sqrt{8}}0\ \ \frac{1}{\sqrt{8}}00\frac{1}{\sqrt{8}}]$$

Gdzie $X\_n$ oznacza znormalizowany wektor wejściowy, a $W$ docelowy wektor wagowy odpowiadający matrycy znaku X o rozdzielczości 4 x 4.

Gdy sieć jest już zainicjowana możemy użyć metody *predict*, aby otrzymać wyniki analizy. Ciało metody znajduje się w poniższym listingu:

```
1 def predict(self, inputs: List):
2     inputs_normalized = normalize(np.array(inputs).flatten())
3     return self.weights.dot(inputs_normalized)
```

Metoda dokonuje mnożenia macierzy wag przez znormalizowany wektor wejściowy. Tak jak przedstawia to poniższy wzorzec.

$$\begin{pmatrix} W_{x1} & W_{x2} & W_{x2} & ... & W_{xn} \\ W_{y1} & W_{y2} & W_{y2} & ... & W_{y_n} \\ W_{z1} & W_{z2} & W_{z2} & ... & W_{zn} \end{pmatrix} \cdot \begin{pmatrix} X_1 \\ X_2 \\ ... \\ X_n \end{pmatrix} = \begin{pmatrix} Y_x \\ Y_y \\ Y_x \end{pmatrix}$$

Jako rezultat analizy otrzymujemy trzy wartości należące do przedziału $< 0, 1 >$, określające prawdopodobieństwo, że wektor wejściowy odpowiada prototypowi matrycy konkretnej litery.

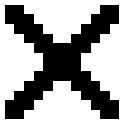
## 3.1 Eksperyment nr 1

### 3.1.1 Założenia

Przeprowadzenie analizy na matrycy identycznej jak matryca będąca prototypem dla rozpoznawania litery X.

### 3.1.2 Przebieg

W tym eksperymencie jako wejście posłużymy się matrycą identyczną jak jedna z tych umieszczonych w warstwie wyjścia sieci. W naszym przykładzie będzie to litera **X** przedstawiona na Rysunku 3.



Rysunek 3: Matryca prototypu X

Matryca zostaje załadowana z bitmapy monochromatycznej do programu za pomocą analogicznego kodu do inicjalizacji sieci neuronowej. Następnie za pomocą metody *predict* wektor wejściowy zostaje znormalizowany poprzez podzielenie go przez pierwiastek z ilości jedynek w nim zawartych. Dzięki takiej normalizacji w przypadku pełnej zgodności matryc nasza suma ważona będzie należała do przedziału $< 0, 1 >$ i oznaczać będzie procent zgodności prototypu i wektora wejściowego.

### 3.1.3 Rezultat

Efekt obliczeń sieci dla matrycy identycznej jak jedna z prototypów (w naszym przypadku X) prezentuje się następująco:

$$Y_x : 1.0 \quad | \quad 100\%$$

$$Y_y : 0.5573704017131537 \quad | \quad 55\%$$

$$Y_z : 0.6274950199005567 \quad | \quad 62\%$$

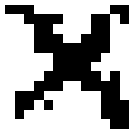
## 3.2 Eksperyment nr 2

### 3.2.1 Założenia

Przeprowadzenie analizy na matrycy zawierającej literę z nałożonym efektem ziarna pogarszającym jej jakość.

### 3.2.2 Przebieg

W eksperymencie 2 znów posłużymy się matrycą znaku **X**, lecz tym razem dodatkowo nałożymy na nią efekt "ziarna", pogarszając tym jej rozpoznawalność. Efekt zmian widoczny jest na Rysunku 4.



Rysunek 4: Matryca X z efektem "ziarna"

Proces ładowania danych oraz dokonywania predykcji został wykonany identycznie jak w przypadku eksperymentu 1.

### 3.2.3 Rezultat

Efekt obliczeń sieci dla matrycy wejściowej prezentuje się następująco:

$$Y_x : 0.836660026534076 \mid 83\%$$

$$Y_y : 0.5124500385567423 \mid 51\%$$

$$Y_z : 0.6000000000000001 \mid 60\%$$

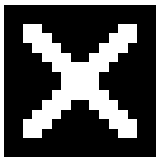
## 3.3 Eksperyment nr 3

### 3.3.1 Założenia

Przeprowadzenie analizy na matrycy będącej negatywem jednej z matryc prototypowych.

### 3.3.2 Przebieg

W ostatnim eksperymencie użyjemy matrycy znaku **X** przekształconej w negatyw. Zabieg ten pozwala stworzyć odwrotność prototypu litery X. Efekt zmian widoczny jest na Rysunku 5.



Rysunek 5: Negatyw matrycy X

Podobnie jak w poprzednich próbach, proces analizy i normalizacji został przeprowadzony przy użyciu tego samego algorytmu.

### 3.3.3 Rezultat

Wyniki analizy dokonanej przez sieć na matrycy negatywy litery **X** wyglądają następująco:

$$Y_x : 0.0 \ \mid \ 0\%$$

$$Y_y : 0.09901475429766743 \ \mid \ 9\%$$

$$Y_z : 0.24152294576982403 \ \mid \ 24\%$$

# 4 Wnioski

1. Sieć jest w stanie rozpoznawać znaki w niższej jakości (rozmazane obrazy) z satysfakcjonującym prawdopodobieństwem.

2. Skorygowanie wag w neuronach warstwy wyjściowej o większą ilość matryc, w różnych jakościach poprawiłoby rozpoznawalność znaków.

3. Sieć poprawnie odróżnia negatyw matrycy wykazując tym samym zerowe podobieństwo.

4. Rozpoznanie znaku X przez neuron rozpoznający literę Z, jest na wysokim poziomie ( 60%) z uwagi na wspólną część obu znaków.

# 5 Załączniki

```
# main.py
BRAIN = NeuralNetwork.from_file("img/x.bmp", "img/y.bmp", "
    img/z.bmp")

def pixel_array_from_file(file: str) -> []:
    x = np.array(Image.open(file))

    x_flatten = []
    for row in x:
        for pixel in row:
            if pixel == True:
                x_flatten.append(0)
            else:
                x_flatten.append(1)

    return x_flatten

# 1:1 Test
print("1:1 Test: ")
prediction = BRAIN.predict(pixel_array_from_file("img-test/x.
    bmp"))
print(f"Y_x: {prediction[0]} | {math.floor(prediction[0] *
    100)}%")
print(f"Y_y: {prediction[1]} | {math.floor(prediction[1] *
    100)}%")
print(f"Y_z: {prediction[2]} | {math.floor(prediction[2] *
    100)}%")
print("==========")

# Mesh Test
print("Mesh Test: ")
prediction = BRAIN.predict(pixel_array_from_file("img-test/
    x_mesh.bmp"))
print(f"Y_x: {prediction[0]} | {math.floor(prediction[0] *
    100)}%")
print(f"Y_y: {prediction[1]} | {math.floor(prediction[1] *
    100)}%")
print(f"Y_z: {prediction[2]} | {math.floor(prediction[2] *
    100)}%")
print("==========")

# Negative Test
print("Negative Test: ")
prediction = BRAIN.predict(pixel_array_from_file("img-test/
    x_negative.bmp"))
```

```
print(f"Y_x: {prediction[0]} | {math.floor(prediction[0] * 100)}%")
print(f"Y_y: {prediction[1]} | {math.floor(prediction[1] * 100)}%")
print(f"Y_z: {prediction[2]} | {math.floor(prediction[2] * 100)}%")
print("==========")
```

```
# nn.py
class NeuralNetwork:
    def __init__(self):
        self.weights = []

    @classmethod
    def from_file(cls, *args):
        nn = cls()

        for arg in args:
            x = np.array(Image.open(arg))

            x_flatten = []
            for row in x:
                for pixel in row:
                    if pixel == True:
                        x_flatten.append(0)
                    else:
                        x_flatten.append(1)

            x_flatten = normalize(x_flatten)

            nn.weights.append(x_flatten)
        nn.weights = np.array(nn.weights, dtype=float)
        return nn

    def predict(self, inputs: List):
        inputs_normalized = normalize(np.array(inputs).flatten())
        return self.weights.dot(inputs_normalized)


def normalize(inputs: List):
    unique, counts = np.unique(inputs, return_counts=True)
    sum_of_ones = dict(zip(unique, counts))[1]
    return np.where(np.array(inputs) == 1, 1 / math.sqrt(sum_of_ones), inputs)
```