Wykonanie: Grzegorz Denert, Michał Dorosz

**Zadanie 1.**

Zaimplementować perceptronową regułę uczenia. Algorytm powinien działać dla:

1. dowolnego wymiaru danych uczących

import numpy as np

def init\_weights(n):

    return np.random.uniform(low = -10, high= 10, size=(n,))

def calculate\_prediction(inputs,weights):

    sum = np.dot(inputs,weights)

    if sum >= 0:

        return 1

    else:

        return 0

def calculate\_error(true\_value, prediction):

    return true\_value - prediction

def update\_weights(inputs, true\_value, weights):

    prediction = calculate\_prediction(inputs,weights)

    error = calculate\_error(true\_value, prediction)

    weights[0:] += error \* inputs

    return weights

def train(iterations, inputs, true\_value, weights):

    i = 0

    while i < iterations:

        for x, y in zip(inputs, true\_value):

            new\_weights = update\_weights(x,y,weights)

        i += 1

    return new\_weights

Testowanie rozwiązania:

from sklearn.datasets import make\_blobs

# generowanie zbioru danych

X, y = make\_blobs(n\_samples=1000,

                  n\_features=2,

                  centers=2,

                  cluster\_std=3,

                  random\_state=23)

weights = init\_weights(2)

new\_weights = train(20,X,y, weights)

Generowanie wykresu

import matplotlib.pyplot as plt

def abline(ax, b, m, \*args, \*\*kwargs):

    "Add a line with slope m and intercept b to ax"

    xlim = ax.get\_xlim()

    ylim = [m \* xlim[0] + b, m \* xlim[1] + b]

    ax.plot(xlim, ylim, \*args, \*\*kwargs)

fig, ax = plt.subplots()

ax.scatter(X[:, 0], X[:, 1], c=y, cmap='bwr', label='Data Points')

#slope and intercept

slope = -new\_weights[0] / new\_weights[1]

intercept = 0

#decision boundary line

abline(ax, intercept, slope, color='red', label=f'Line: y = {slope:.2f}x')

ax.set\_xlabel('Feature 1')

ax.set\_ylabel('Feature 2')

ax.legend()

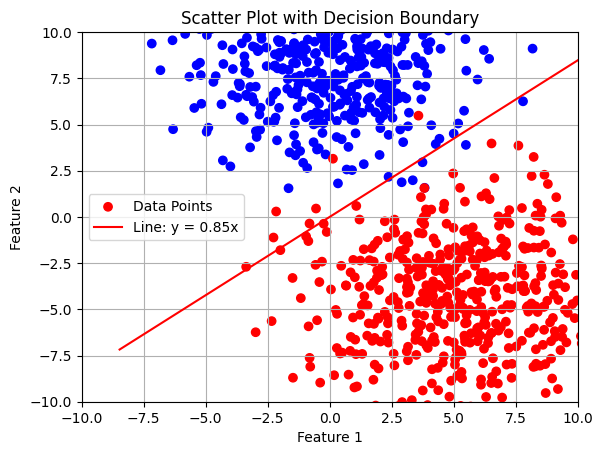
ax.set\_title('Scatter Plot with Decision Boundary')

plt.grid()

plt.ylim(-10, 10)

plt.xlim(-10, 10)

plt.show()

Wykres prezentujący rozwiązanie:

1. dowolnej liczby neuronów w warstwie

import numpy as np

def init\_neuron\_weights(input\_vector\_size):

    return np.random.uniform(low = -10, high= 10, size=(input\_vector\_size,))

def init\_layer\_weights(layer\_size, input\_vector\_size):

    layer\_neurons\_weights = []

    i = 0

    while i < layer\_size:

        neuron\_weights = init\_neuron\_weights(input\_vector\_size)

        layer\_neurons\_weights.append(neuron\_weights)

        i += 1

    return layer\_neurons\_weights

def calculate\_prediction(inputs,weights):

    sum = np.dot(inputs,weights)

    if sum >= 0:

        return 1

    else:

        return 0

def calculate\_error(true\_value, prediction):

    return true\_value - prediction

def update\_weights(inputs, true\_value, weights):

    learning\_coefficient = 0.1

    prediction = calculate\_prediction(inputs,weights)

    error = calculate\_error(true\_value, prediction)

    inputs = np.array(inputs)

    weights[0:] += learning\_coefficient\*error \* inputs

    return weights

def train(iterations, inputs, true\_value, layer\_weights):

    i = 0

    while i < iterations:

        for x, ys in zip(inputs, true\_value):

            for weights,y  in zip(layer\_weights,ys):

                new\_weights = update\_weights(x,y,weights)

        i += 1

    return layer\_weights

Testowanie rozwiązania:

# linearly separable dataset with two classes

X, y = make\_blobs(n\_samples=1000,

                  n\_features=2,

                  centers=3,

                  cluster\_std=2,

                  random\_state=23)

y\_coded = []

for element in y:

    if element == 0:

        y\_coded.append([0,0,0])

    elif element == 1:

        y\_coded.append([0,0,1])

    elif element == 2:

        y\_coded.append([0,1,0])

layer\_weights = init\_layer\_weights(3,2)

new\_layer\_weights = train(10,X, y\_coded, layer\_weights)

Generowanie wykresu

def abline(ax, b, m, \*args, \*\*kwargs):

    "Add a line with slope m and intercept b to ax"

    xlim = ax.get\_xlim()

    ylim = [m \* xlim[0] + b, m \* xlim[1] + b]

    ax.plot(xlim, ylim, \*args, \*\*kwargs)

fig, ax = plt.subplots()

colormap = np.array(['r', 'g', 'b'])

ax.scatter(X[:, 0], X[:, 1], c = colormap[y], label='Data Points')

for neuron\_weights in new\_layer\_weights:

    slope = -neuron\_weights[0] / neuron\_weights[1]

    intercept = 0

    abline(ax, intercept, slope,

color='red',

label=f'Line: y = {slope:.2f}x')

ax.set\_xlabel('Feature 1')

ax.set\_ylabel('Feature 2')

ax.legend()

ax.set\_title('Scatter Plot with Decision Boundary')

plt.grid()

plt.ylim(-10, 10)

plt.xlim(-10, 10)

plt.show()

Wykres prezentujący rozwiązanie:

Obraz zawierający tekst, diagram, zrzut ekranu, linia

Opis wygenerowany automatycznie

**Zadanie 2**

Rozwiązać problem AND i OR (dla odpowiednich dwuwymiarowych danych wejściowych , 0 i 1,

neuron powinien zwracać na wyjściu odpowiadające im wartości funkcji AND i OR):

1. Samodzielnie (bez uczenia): zdefiniować neuron dyskretny wyznaczając jego wartości wag.

Narysować otrzymane rozwiązania.

**Bramka AND**

Obliczenia:

Obraz zawierający tekst, pismo odręczne, Czcionka, diagram

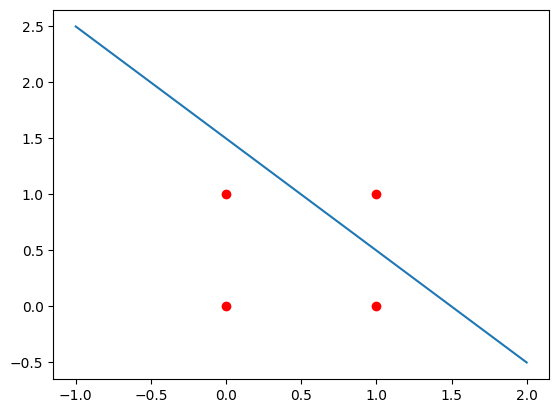
Opis wygenerowany automatycznie

Kod:

|  |
| --- |
| import matplotlib.pyplot as plt  import numpy as np.  def calculate\_neuron(inputs, weights, threshold):      return 1 if weights[0]\*inputs[0]+weights[1]\*inputs[1]-threshold>=0 else 0  inputs = [(0, 0), (0, 1), (1, 0), (1, 1)]  and\_threshold = 1.5  and\_weights = [1, 1]  sums = {}  for inp in inputs:      sums[inp] = calculate\_neuron(inp, and\_weights, and\_threshold)  sums |
| {(0, 0): 0, (0, 1): 0, (1, 0): 0, (1, 1): 1} |

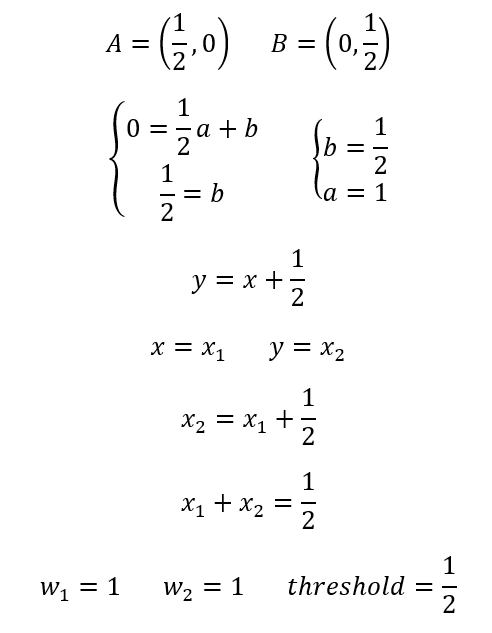
Wykres:

|  |
| --- |
| x = np.linspace(-1, 2, 100)  y = -1\*x+1.5  plt.plot(x, y)  for point in inputs:          plt.scatter(point[0], point[1], color='red') |



**Bramka OR**

Obliczenia:

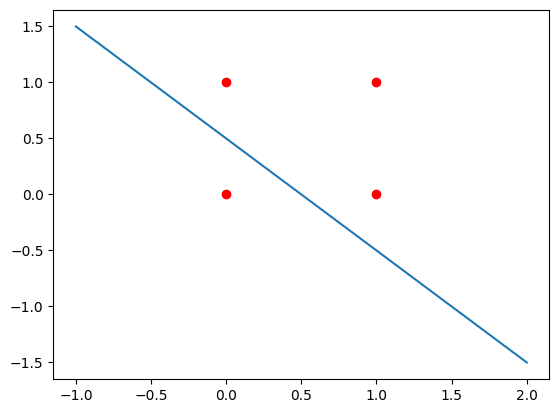


Kod:

|  |
| --- |
| or\_threshold = 0.5  or\_weights = [1, 1]  sums = {}  for inp in inputs:  sums[inp] = calculate\_neuron(inp, or\_weights, or\_threshold)  sums |
| {(0, 0): 0, (0, 1): 1, (1, 0): 1, (1, 1): 1} |

Wykres:

|  |
| --- |
| x = np.linspace(-1, 2, 100)  y = -1\*x+1.5  plt.plot(x, y)  for point in inputs:          plt.scatter(point[0], point[1], color='red') |



1. Wykorzystując regułę perceptronową: przeanalizować wstępne (losowe) wagi neuronów, a

następnie wagi po uczeniu. Ile iteracji wymagał proces uczenia? Narysować położenie

prostych reprezentujących neuron po inicjalizacji wag oraz otrzymane rozwiązania.

Perceptronowa reguła uczenia:

|  |
| --- |
| import numpy as np  def init\_weights(n):      return np.random.uniform(low = -10, high= 10, size=(n,))  def calculate\_prediction(inputs,weights):      sum = np.dot(inputs,weights)      if sum >= 0:          return 1      else:          return 0  def calculate\_error(true\_value, prediction):      return true\_value - prediction  def update\_weights(inputs, true\_value, weights):      prediction = calculate\_prediction(inputs,weights)      error = calculate\_error(true\_value, prediction)      weights[0:] += error \* inputs      return weights  def train(iterations, weights, X\_train, y\_train):      i = 0      while i < iterations:          for x, y in zip(X\_train, y\_train):              new\_weights = update\_weights(x,y,weights)          i += 1      return new\_weights |

**Bramka AND**

|  |
| --- |
| x\_train = np.array([  [1, 0, 0], # bias, x1, x2  [1, 0, 1],  [1, 1, 0],  [1, 1, 1]  ])  y\_train = np.array([0, 0, 0, 1]) |

Inicjalizacja wag, wstępne wagi neuronu:

|  |
| --- |
| weights = init\_weights(x\_train.shape[1])  weights |
| array([ 7.14467078, -8.00879785, -9.06272004]) |

Uczenie, wagi po nauczeniu, liczba iteracji:

|  |
| --- |
| def calculate\_neuron(inputs, weights):      return 1 if weights[0]\*inputs[0]+weights[1]\*inputs[1]+weights[2]\*inputs[2]>=0 else 0  counter = 0  while [calculate\_neuron(x\_train[i], weights) for i in range(len(y\_train))] != y\_train.tolist():      weights = train(1, weights, x\_train, y\_train)      counter+=1  weights = weights.tolist()  weights, counter |
| ([-3.0012682444802667, 2.0915894486629956, 1.3319104175388503], 13) |

Wykres:

|  |
| --- |
| # w0\*1+w1\*x1+w2\*x2 = 0  ->  w2x2 = -w1x1-w0  ->  x2 = -w1x1-w0/w2  x1 = np.linspace(-1, 2, 100)  x2 = -1\*(weights[1]\*x1+weights[0])/weights[2]  plt.plot(x1, x2)  plt.xlabel('x1')  plt.ylabel('x2')  for point in inputs:      plt.scatter(point[0], point[1], color='red') |

**Obraz zawierający linia, diagram, Wykres, zrzut ekranu

Opis wygenerowany automatycznie**

**Bramka OR**

|  |
| --- |
| x\_train = np.array([  [1, 0, 0], # bias, x1, x2  [1, 0, 1],  [1, 1, 0],  [1, 1, 1]  ])  y\_train = np.array([0, 0, 0, 1]) |

Inicjalizacja wag, wstępne wagi neuronu:

|  |
| --- |
| weights = init\_weights(x\_train.shape[1])  weights |
| array([ 8.86592666, -6.97548513, -2.64845347]) |

Uczenie, wagi po nauczeniu, liczba iteracji:

|  |
| --- |
| counter = 0  while [calculate\_neuron(x\_train[i], weights) for i in range(len(y\_train))] != y\_train.tolist():      weights = train(1, weights, x\_train, y\_train)      counter+=1  weights = weights.tolist()  weights, counter |
| ([-0.13407333604199323, 1.0245148672552187, 0.35154653105984224], 18) |

Wykres:

|  |
| --- |
| x1 = np.linspace(-1, 2, 100)  x2 = -1\*(weights[1]\*x1+weights[0])/weights[2]  plt.plot(x, y)  plt.xlabel('x1')  plt.ylabel('x2')  for point in inputs:      plt.scatter(point[0], point[1], color='red') |

Obraz zawierający linia, diagram, zrzut ekranu, Wykres

Opis wygenerowany automatycznie