Mikołaj Stępniewski
Podstawy Sztucznej Inteligencji

Sprawozdanie do zadania 1:

Opis wykonywanego ćwiczenia:

Celem ćwiczenia jest poznanie budowy i działania perceptronu poprzez implementację oraz uczenie perceptronu realizującego wybraną funkcję logiczną dwóch zmiennych.

Schemat ćwiczenia:

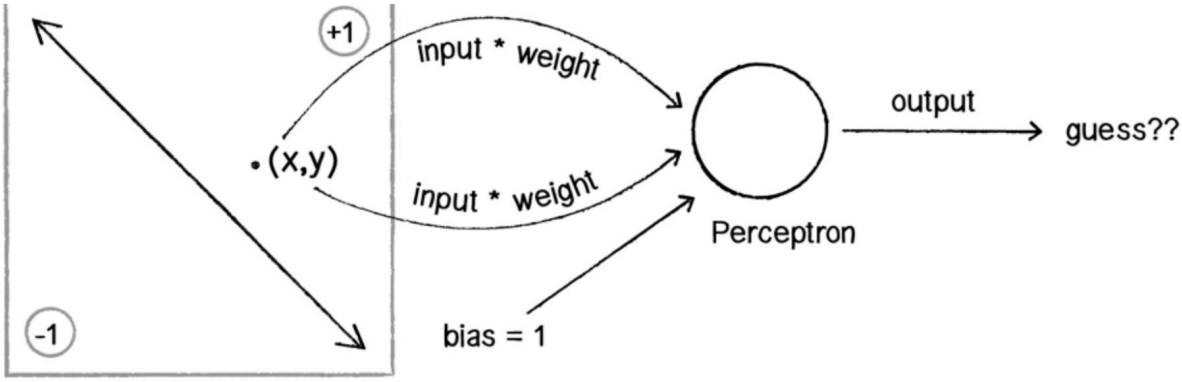
1. Zaimplementowanie sztucznego neuronu
2. Wygenerowanie zestawu uczącego dla bramki logicznej AND
3. Uczenie perceptronu za pomocą różnej liczby danych
4. Testowanie perceptronu

Implementacja perceptronu oraz program dokonujący uczenia oraz pobierania danych wyjściowych z perceptronu zrealizowana została w języku *Python3*. Argumentami przemawiającymi za takim a nie innym doborem języka jest jego szybkość oraz prosta składnia umożliwiająca skupienie się wyłącznie na temacie projektu.

Całość programu składa się z czterech plików; perceptron.py, main.py, input.py oraz consts.py. Perceptron posiada wyłącznie metody trenowania oraz zgadywania na podstawie posiadanych wag, natomiast *InputPoint* oprócz dwóch losowych wartości typu *Boolean* posiada prawidłowy dla nich wynik – co pozwala w łatwy sposób przetestować perceptron. Prawidłowy wynik danego punktu określany jest w konstruktorze na podstawie ustalonej funkcji logicznej. Taki podział sprawia, że schemat działania perceptronu jest bardzo przejrzysty i zredukowany objętościowo, gdyż obsługuje wyłącznie metody trenowania i zgadywania. Plik main.py obsługuje tworzenie dowolnej ilości punktów testowych typu *InputPoint*, uczenie na ich podstawie perceptronu dowolną ilość er oraz tworzenie dowolnej ilości nowych puntów *InputPoint* o losowych wartościach w celu przetestowania działania perceptronu. Wyniki testów oraz wartości parametrów; ilość punktów uczących (*training_inputs_count*), ilość epok (*eras_count*), ilość punktów testowych (*test_range*) oraz ilość błędnych odpowiedzi perceptronu (*fail_counter*) są wypisywane w logach programu.

Prosta sieć neuronowa jaką jest perceptron ma za zadanie na podstawie dowolnej ilości wejść oraz odpowiadających im wag podać jedną z dwóch grup, do której należy analizowany obiekt. Przykładem może być właśnie odgadywanie czy funkcja logiczna dwóch zmiennych zwraca wartość równą PRAWDA czy FAŁSZ. Innym przykładem może być określenie na podstawie np. Wielkości oraz udomowienia czy poddawane analizie zwierzę jest psem czy kotem lub wracając do bardziej matematycznych zagadnień czy punkt (x, y) leży pod czy nad wykresem funkcji określonej na początku. W tym ostatnim przykładzie można w łatwy sposób wizualnie przedstawić jak w danej chwili perceptron widzi wykres tej funkcji – pytając go o wartość parametru y dla dowolnego x na podstawie wzoru:

W1*y + W2*x +W3*BIAS = 0, gdzie w1, w2, w3 to odpowiadające wagi.



Jednak do realizacji zadania stworzenia bramki logicznej wykorzystałem jedynie wzory na obliczanie błędu, korygowanie wag oraz obliczanie wyjścia (udzielanie odpowiedzi).

Algorytm wyznaczania odpowiedzi przez perceptron przedstawia się następująco:

1. Pomnóż każde wejście przez odpowiadającą mu wagę.
2. Zsumuj te wartości.
3. Oblicz wyjście przepuszczając uzyskaną sumę przez funkcję aktywującą (*sign* – znaku wartości – czyli 1 dla sumy dodatniej, -1 dla sumy ujemnej).

Algorytm uczenia perceptronu:

1. Dostarcz perceptronowi wejść, dla których znana jest odpowiedź (wyjście).
2. Zapytaj perceptron o jego „własną" odpowiedź.
3. Oblicz błąd (Czy perceptron się pomylił?)
4. Na podstawie tego błędu skoryguj wagi (Zmiana tylko w przypadku, gdy się pomylił)
5. Wróć do kroku 1. i powtórz całość.

Ze względu na to, że dostarczamy wejść, dla których wyjście jest przez nas określone uczenie tego typu nazywamy uczeniem nadzorowanym (*„supervised learning"*).

Wzory użyte do zrealizowania powyższego algorytmu:

Obliczanie błędu:

BŁĄD = ZNANY WYNIK – ODPOWIEDŹ PERCEPTRONU

Ze względu na to, że perceptron potrafi zwrócić tylko dwie wartości (Prawda/Fałsz lub -1/1, Kot/Pies) możliwych wartości błędu jest tylko cztery, np.:

ZNANY WYNIK	ODPOWIEDŹ PERCEPTRONU	BŁĄD
-1	-1	0
-1	+1	-2
+1	-1	2
1	1	0

Wagę korygujemy na podstawie błędu, odpowiadającego jej wejścia wejścia oraz współczynnika uczenia (o którym później):

WAGA += BŁĄD * WEJŚCIE * WSPÓŁCZYNNIK UCZENIA

Współczynnik uczenia to wartość określająca siłę każdej korekty. Przyjmuje on wartości od 0 do 1. Ogranicza on w dużym stopniu zmianę wagi – jeśli wartość współczynnika jest zbyt duża, perceptron może zmienić wagę w zbyt dużym stopniu nie trafiając w optymalną wartość wagi. Jeżeli współczynnik jest zbyt mały – perceptron będzie dokonywał zmian wag bardzo powoli (o małe wartości) co oczywiście zwiększa dokładność, lecz wymaga większej ilości epok uczenia.

Bias jest dodatkowym wejściem dla perceptronu zawsze równym 1. Jest on potrzebny np. W przypadkach, gdy wejście jest równe 0 – a z racji tego, że we wzorach (np. Na korekcje wagi) wymnażamy przez siebie wejście, błąd i współczynnik uczenia to nigdy byśmy tej wagi nie zmienili (gdy wejście == 0) chociaż oczywiście jest to wskazane.

W programie funkcja główna *main* przepuszcza wszystkie punkty trenujące przez perceptron (uczy go), następnie tworzy nowe punkty losowe i „pyta" perceptron o samodzielne odpowiedzi. Podczas trenowania, dla każdej epoki obliczany jest błąd średniokwadratowy pozwalający zobaczyć jak zmniejszała (bądź zwiększała) się skuteczność perceptronu.

Uwaga:

W swoim programie do uczenia wykorzystywałem dużą ilość punktów trenujących oraz testujących, choć w przypadku bramki logicznej wystarczyłoby użyć czterech kombinacji (00, 10, 01, 11). Jednakże, takie podejście jakie zrealizowałem jest uniwersalne i byłoby skuteczne dla punktów, które mogły mieć wiele kombinacji, np. Punkty układu współrzędnych (x, y).

Przykładowy wynik programu:

```
epoka: 0, avg_err: 6                    // Ze względu na dużą ilość punktów trenujących (15)
                                        perceptron już w drugiej epoce ustalił odpowiednie wagi
epoka: 1, avg_err: 0


True AND True -> True
True AND False -> False
False AND True -> False
True AND True -> True
True AND False -> False
True AND True -> True
False AND False -> False
True AND True -> True
True AND True -> True
True AND True -> True
False AND False -> False
True AND True -> True
True AND True -> True
True AND True -> True
True AND False -> False
False AND True -> False
False AND False -> False
True AND True -> True
```

False AND False -> False
True AND True -> True
True AND False -> False
True AND False -> False
False AND False -> False
False AND False -> False
False AND True -> False
True AND True -> True
True AND False -> False
False AND False -> False
False AND False -> False
False AND False -> False
False AND False -> False
True AND False -> False
True AND False -> False
False AND False -> False
False AND False -> False
True AND True -> True
False AND True -> False
True AND True -> True
True AND True -> True
True AND False -> False
True AND False -> False
True AND True -> True
True AND True -> True
False AND False -> False
False AND False -> False
False AND True -> False
True AND True -> True
False AND True -> False
False AND False -> False
False AND False -> False

Fail counter: 0/50 wrong.

Learning rate: 0.500000, eras: 2, training inputs: 15

Opracowanie wyników WERSJA 1:

1. Wpływ współczynnika uczenia na skuteczność perceptronu.

W przypadku współczynnika równego 0.05 redukcja błędów w kolejnych epokach przebiegała powoli aczkolwiek dokładnie i skutecznie. Zmieniając współczynnik na 0.5 zauważyć można, że zazwyczaj perceptron po jednej epoce zgadywał poprawnie. Jednak czasem korygował wagi w zbyt dużym stopniu co zmiany błędu w drugą stronę.

Learning rate: 0.050000, eras: 20, training inputs: 10	**Learning rate: 0.500000, eras: 20, training inputs: 10**
era: 0, avg_err: 6 era: 1, avg_err: 2 era: 2, avg_err: 4 era: 3, avg_err: 2 era: 4, avg_err: 2 era: 5, avg_err: 4 era: 6, avg_err: 2	era: 0, avg_err: 6 era: 1, avg_err: 0 **…** era: 0, avg_err: 4 era: 1, avg_err: 0 **…**

era: 7, avg_err: 0 era: 8, avg_err: 0 … era: 0, avg_err: 0 era: 1, avg_err: 0 … era: 0, avg_err: 10 era: 1, avg_err: 8 era: 2, avg_err: 4 era: 3, avg_err: 2 era: 4, avg_err: 0 … era: 0, avg_err: 4 era: 1, avg_err: 0 … era: 0, avg_err: 6 era: 1, avg_err: 2 era: 2, avg_err: 0 …	era: 0, avg_err: 2 era: 1, avg_err: 0 … era: 0, avg_err: 6 era: 1, avg_err: 0 … **era: 0, avg_err: 6 // Zbyt duża korekta** **era: 1, avg_err: 6** **era: 2, avg_err: 0** **…** era: 0, avg_err: 4 era: 1, avg_err: 0 era: 2, avg_err: 0 … era: 0, avg_err: 2 era: 1, avg_err: 0 era: 2, avg_err: 0 …

2. Wpływ ilości punktów treningowych i ilości epok.

W przypadku małej ilości punktów testowych (np. 2) nawet duża ilość epok nie zapewnia poprawnych wyników – dla przykładu jakim jest bramka logiczna kombinacji jest tylko cztery, zatem tak mała ilość punktów testowych może zapewnić lepsze wyniki niż w przypadku bardziej skomplikowanych modeli.

Przykładowe testy:

Fail counter: 15/50 wrong.
Learning rate: 0.500000, eras: 20, training inputs: 2

Fail counter: 0/50 wrong.
Learning rate: 0.500000, eras: 20, training inputs: 2

Fail counter: 8/50 wrong.
Learning rate: 0.500000, eras: 20, training inputs: 2

Fail counter: 10/50 wrong.
Learning rate: 0.500000, eras: 20, training inputs: 2

Fail counter: 10/50 wrong.
Learning rate: 0.500000, eras: 20, training inputs: 2

Jak widać tylko raz na pięć testów perceptron znał wszystkie odpowiedzi.

Sytuacja wygląda dużo lepiej, jeżeli mała jest ilość epok(1) a duża liczba punktów treningowych(20).

Fail counter: 0/50 wrong.
Learning rate: 0.500000, eras: 1, training inputs: 20

Fail counter: 0/50 wrong.
Learning rate: 0.500000, eras: 1, training inputs: 20

Fail counter: 0/50 wrong.
Learning rate: 0.500000, eras: 1, training inputs: 20

Fail counter: 0/50 wrong.
Learning rate: 0.500000, eras: 1, training inputs: 20

Fail counter: 0/50 wrong.
Learning rate: 0.500000, eras: 1, training inputs: 20

Fail counter: 0/50 wrong.
Learning rate: 0.500000, eras: 1, training inputs: 20

Fail counter: 0/50 wrong.
Learning rate: 0.500000, eras: 1, training inputs: 20

Fail counter: 0/50 wrong.
Learning rate: 0.500000, eras: 1, training inputs: 20

Fail counter: 13/50 wrong.
Learning rate: 0.500000, eras: 1, training inputs: 20

Jak widać, raz na dziewięć prób perceptron się pomylił.

Próbując różnych kombinacji zauważyłem, że w moim przypadku minimalnym optymalnym doborem współczynników jest:
- 2 epoki
- 15 punktów testowych
- współczynnik uczenia równy 0.1

Przy takich współczynnikach perceptron był zawsze bezbłędny mimo, małej ilości punktów treningowych.

Zestawienie wyników WERSJA 2.
W tym przypadku tworzę 50 punktów losowych, następnie po każdej epoce uczenia (na podstawie 20 i 10 losowych punktów) sprawdzam jak perceptron poradzi sobie z tymi dwudziestoma punktami. Współczynnik uczenia równy 0.01, aby perceptron nie doszedł do poprawnych wyników zbyt szybko - żeby uwidocznić proces uczenia perceptronu.

Epoka (**20** punktów trening.)	Błąd średniokwadratowy	Błędy (na 50 punktów)
1	40	50
2	34	35
3	30	35
4	26	23
5	24	23
6	16	0

Epoka (**20** punktów trening.)	Błąd średniokwadratowy	Błędy (na 50 punktów)
1	14	29
2	12	20
3	10	20
4	14	20
5	10	0
6	0	0

Dla 10 punktów treningowych:

Epoka (**10** punktów trening.)	Błąd średniokwadratowy	Błędy (na 50 punktów)
1	20	40
2	14	40
3	12	50
4	14	40
5	14	40
6	12	50
7	14	40
8	14	40
9	12	50
10	14	40
11	14	40
12	12	50
13	14	40
14	10	10
15	2	0
16	0	0

Epoka (**10** punktów trening.)	Błąd średniokwadratowy	Błędy (na 50 punktów)
1	12	27
2	12	27
3	12	27
4	12	27
5	14	13
6	14	0
7	0	0

Epoka (**10** punktów trening.)	Błąd średniokwadratowy	Błędy (na 50 punktów)
1	10	24
2	10	24
3	10	10
4	4	10
5	4	10
6	4	10
7	4	10
8	4	10
9	4	10
10	4	10
11	4	10
12	4	10
13	4	10
14	2	0
15	0	0

Wnioski:

Na skuteczność działania perceptronu wpływają czynniki takie jak: ilość epok, ilość punktów treningowych, współczynnik uczenia oraz w niewielkim stopniu losowość (gdyż wagi na samym początku są generowane losowo, tak samo punkty treningowe i testowe).

Zgodnie ze wcześniejszymi założeniami zbyt duży współczynnik uczenia prowadzić może do „przestrzelenia" wartości wagi w drugą stronę, zatem powoduje nieprecyzyjne uczenie perceptronu – które w niektórych przypadkach może oczywiście przyspieszyć proces, lecz zależy to od modelu, który analizujemy. Zbyt mały współczynnik powoduje, że na perceptron na pewno dojdzie do prawidłowego wyniku, aczkolwiek będzie potrzebował większej ilości treningów (epok).

W drodze wykonanych testów okazało się również, że uczenie perceptronu znacznie bardziej polega na większej ilości punktów testowych, aniżeli ilości epok. W przypadku małej ilości punktów testowych perceptron poznaje tylko małą cześć możliwości (kombinacji) jakie istnieją w danym modelu, zatem łatwiej go „zaskoczyć" kombinacją, której nie zna. Jeżeli dostarczymy mu większej ilości kombinacji (w tym przypadku wszystkich czterech kombinacji, dodatkowo powtórzonych), perceptron dostosowuje swoje wagi dla każdej możliwości (zapewne kilka razy), zatem jedna epoka była wystarczająca do uzyskania pewnych wyników.

Podsumowanie:

Model jakim jest bramka logiczna AND posiada wiele uproszczeń (tj. Np. Tylko cztery kombinacje), które w znacznym stopniu przyspieszają uczenie perceptronu, lecz testowanie takiego przypadku daje nam spojrzenie na aspekty, które determinują skuteczne działanie perceptronu. Dzięki takiemu doświadczeniu wiem na co zwracać uwagę i w jaki sposób dobierać parametry uczenia, tak aby w przyszłości uzyskiwać satysfakcjonujące wyniki.

Listing kodu programu:

perceptron.py:

```
import random
from consts import *

def sign(x):
    if x >= 0:
        return 1
    else:
        return -1

class Perceptron:
    """ Perceptron is a simple neural net that can
        specify which class object belongs to. """

    weights = []
    training_counter = 0

    def __init__(self, n):
        for i in range(n):
            self.weights.append(random.uniform(-1,1))

    def guess(self, inputs):
        sum = 0.0
        for i in range(len(self.weights)):
            sum += inputs[i] * self.weights[i]

        """ Return output """
        return sign(sum)

    def train(self, inputs, target):
        guess = self.guess(inputs)
        error = target - guess

        for i in range(len(self.weights)):
            self.weights[i] += error * inputs[i] * LR

    """ NOT FOR THIS MODEL!
        def guess_y(self, x):
        w0 = self.weights[0] # x
        w1 = self.weights[1] # y
        w2 = self.weights[2] # BIAS

        return -(w0/w1) * x - (w2/w1)"""
```

input.py:

```
import random

""" -1 is False, 1 is True """

class InputPoint:
    x = -1
    y = -1
    label = -1

    def convertToBool(self, num):
        return True if num == 1 else False

    def __init__(self, randomize=False):
        if randomize == True:
            rndX = random.uniform(-1,1)
            rndY = random.uniform(-1,1)
            if rndX > 0:
                self.x = 1
            if rndY > 0:
                self.y = 1

            """ Main condition. """
            if self.convertToBool(self.x) and self.convertToBool(self.y):
                self.label = 1
            else:
                self.label = -1
```

main.py: (Dla opracowania wyników WERSJA 1)

```
from input import *
from perceptron import *

training_inputs = []
p = Perceptron(3)

def main():
    eras_count = 1
    training_inputs_count = 20
    fail_counter = 0
    test_range = 50

    """ Make random inputs [x:Bool, y:Bool]. """
    for i in range(training_inputs_count):
        training_inputs.append(InputPoint(randomize=True))
```

```
    """ Train perceptron based on that inputs. """
    print()
    for i in range(eras_count):
        avg_err = 0
        for input in training_inputs[:]:
            inputs = [input.x, input.y, BIAS]
            target = input.label
            error = p.train(inputs, target)
            avg_err += error * error

        avg_err *= 0.5
        print('era: %s, avg_err: %d' % (i, avg_err))

    print('\n\n')


    """ Make some new randomized inputs & check if it really works. """
    test_p1 = InputPoint()
    for i in range(test_range):
        testInput = InputPoint(randomize=True)
        inputs = [testInput.x, testInput.y, BIAS]
        guess = p.guess(inputs)
        strr = ('True ' if testInput.x == 1 else 'False ') + 'AND ' + ('True ' if testInput.y == 1 else
'False ') + '-> ' + ('True ' if guess == 1 else 'False ')

        if testInput.label != guess:
            strr += "       X"
            fail_counter += 1
        print(strr)
    print("-------------------------------------------")
    print("\nFail counter: %d/%d wrong.\n" % (fail_counter, test_range))
    print("Learning rate: %f, eras: %d, training inputs: %d\n" % (LR, eras_count,
training_inputs_count))

main()
```

main.py (Dla opracowania wyników WERSJA 1):

```
from input import *
from perceptron import *

training_inputs = []
p = Perceptron(3)

def main():
```

```
    eras_count = 20
    training_inputs_count = 10
    fail_counter = 0
    test_range = 50
    test_inputs = []

    """ Make random inputs [x:Bool, y:Bool]. """
    for i in range(training_inputs_count):
        training_inputs.append(InputPoint(randomize=True))


    for i in range(test_range):
        test_inputs.append(InputPoint(randomize=True))


    """ Train perceptron based on that inputs. """
    print()
    for i in range(eras_count):
        avg_err = 0
        fail_counter = 0
        for input in training_inputs[:]:
            inputs = [input.x, input.y, BIAS]
            target = input.label
            error = p.train(inputs, target)
            avg_err += error * error

        avg_err *= 0.5
        for j in range(test_range):
            testInput = test_inputs[j]
            inputs = [testInput.x, testInput.y, BIAS]
            guess = p.guess(inputs)
            if testInput.label != guess:
                fail_counter += 1
        print('era: %s, avg_err: %d, test errors: %d/%d' % (i, avg_err, fail_counter,
len(test_inputs)))

    print('\n\n')

main()
```

consts.py:

```
WIDTH, HEIGHT = 600, 600
LR = 0.5 #Learning rate
BIAS = 1
```

Źródła:

1. Wykłady
2. https://github.com/shiffman/NOC-S17-2-Intelligence-Learning/blob/master/week4-neural-networks/perceptron.pdf
3. http://sknbo.ue.poznan.pl/neuro/ssn/pliki/uczenie/uczenie1.html
4. https://en.wikipedia.org/wiki/Perceptron
5. https://edux.pjwstk.edu.pl/mat/273/lec/wyklad3/w3.htm#Uczenie

Repozytorium tego projektu: https://github.com/maikelSoFly/PSI--Perceptron-AND