НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ
«КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ
ІМЕНІ ІГОРЯ СІКОРСЬКОГО»
Факультет прикладної математики
Кафедра прикладної математики

Звіт

із лабораторної роботи №1

з дисципліни «Системи глибинного навчання»

на тему

“Розробка програмного забезпечення для реалізації двошарового персептрону з сигмоїдальною функцією активації”

Виконав:
студент групи КМ-01
Романецький М.С.

Викладач:
Професор кафедри ПМА
Терейковський І. А.

Київ — 2023

Зміст

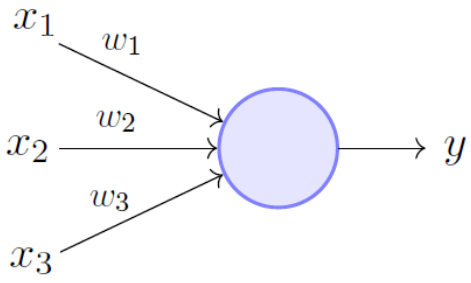
Теоретичні відомості

Персептрон та багатошаровий персептрон представляють собою основні блоки у світі штучних нейронних мереж. Персептрон служить основою для розуміння принципів роботи більш складних моделей, таких як багатошаровий персептрон.

Основна ідея персептрона полягає в тому, що він може вирішувати прості задачі класифікації шляхом взаємодії вхідних сигналів з вагами, і якщо сума вагових значень перевищує певний поріг, то активується вихідний сигнал. Це може бути інтерпретовано як прийняття рішення на основі зваженої інформації.

Багатошаровий персептрон вдосконалює цю концепцію, дозволяючи вирішувати більш складні завдання завдяки введенню прихованих шарів. Кожен шар виконує певну обробку вхідних даних та передає результати наступному шару. Це дозволяє моделі адаптуватися до складних нелінійних залежностей у вхідних даних, що робить її потужнішою для розв'язання різноманітних завдань.

Багатошаровий персептрон може бути використаний для розпізнавання образів, розподілення класів, апроксимації функцій та інших завдань машинного навчання. Важливою характеристикою MLP є його здатність вчитися з прикладів та адаптуватися до нових вхідних даних, що робить його ефективним інструментом для різних застосувань у сучасному машинному навчанні.



Perceptron Model (Minsky-Papert in 1969)

Рис. 1 – Персептрон

https://miro.medium.com/v2/resize:fit:645/0*LJBO8UbtzK_SKMog

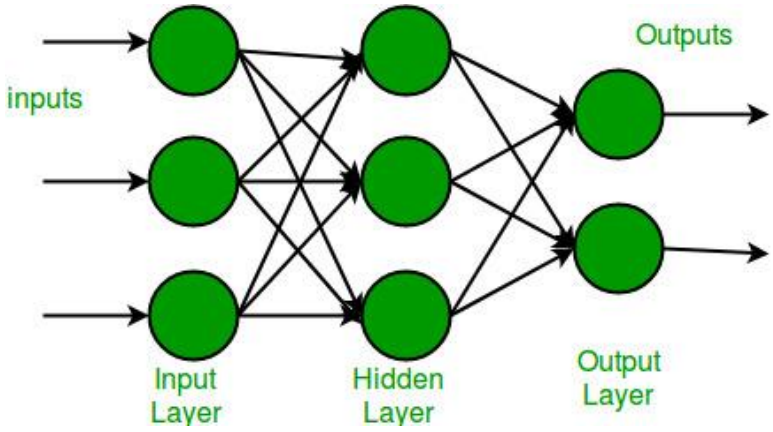


Рис. 2 – Багатошаровий персептрон

https://media.geeksforgeeks.org/wp-content/uploads/nodeNeural.jpg

Основна частина

Імпортуємо математичну бібліотеку Numpy та одразу визначимо random seed, щоб кожного разу отримувати однакові ‘псевдо-випадкові’ значення.

Частина 1

Завдання: розробити програмне забезпечення для реалізації класичного нейрону. Передбачити режим навчання на одному навчальному прикладі та режим розпізнавання.

Ініціалізуємо клас ClassicalNeuron, який буде приймати вхідні параметри:

- Вхідний вектор для нейрона
- Ваги нейрона
- Цільове значення виходу

Створимо сигмоїдальну функцію активації:

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

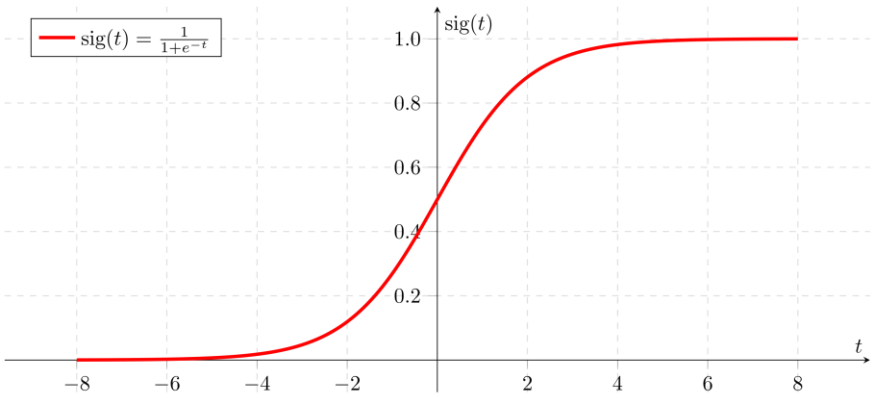


Рис 3. – Графік сигмоїди

https://upload.wikimedia.org/wikipedia/commons/thumb/5/53/Sigmoid-function-2.svg/2560px-Sigmoid-function-2.svg.png

Створимо функцію тренування, яка буде приймати в якості параметра максимальну кількість ітерацій. У циклі викликає функцію активації з переданим вектором. Обчислює помилку навчання, оновлює вагові коефіцієнти нейрона та виводить на екран першу та останню ітерації. Критерій зупинки навчання – проходження всіх ітерацій (60).

Навчальні дані:

```
train_vector = np.array([1, 0.2, 0.3])
weights_1 = np.random.rand(3)
target_output_1 = 0.5
```

Навчальний вектор містить додаткову ‘1’ – це введено для того, щоб легше реалізувати операції, пов'язані з зсувом (bias) у ваговій сумі в нейронному вході.

Далі створюємо екземпляр класу та тренуємо його на навчальних даних. Коли модель завершить тренування почнеться режим розпізнавання, тобто ми даємо моделі на вхід значення яке вона ніколи не бачила і очікуємо побачити те значення на яке вона натренована (у даному випадку очікуваний У = 0.5)

Результат відпрацювання коду програми:

```
Ітерація 1:
  Ваги: [0.1608 0.0156 0.4574]
  Вихід: 0.5801
  Помилка: -0.0195
  Оновлення вагів: [-0.0195 -0.0039 -0.0059]

Ітерація 60:
  Ваги: [-0.1021 -0.037   0.3785]
  Вихід: 0.5011
  Помилка: -0.0003
  Оновлення вагів: [-0.0003 -0.0001 -0.0001]

Режим розпізнавання:
Вектор [1.  0.8 0.4] == 0.5049
```

Як бачимо, модель оновлює ваги і в режимі розпізнавання видає результат 0.5049, отже модель навчилась.

Частина 2

Завдання: розробити програмне забезпечення для реалізації елементарного двошарового персептрону із структурою 1-1-1. Передбачити режим навчання на одному навчальному прикладі та режим розпізнавання.

Ініціалізуємо клас Perceptron_1_1_1, який буде приймати вхідні параметри:

- input_value: значення вхідної величини
- weights: ваги моделі (масив)
- target_output: цільове значення вихідної величини

Створюємо сигмоїдальну функцію активації. Формула та графіку функцію див. у Частина 1.

Створюємо функцію тренування моделі, яка приймає кількість ітерацій (за замовчанням 50). Ця функція виконує прямий та зворотній прохід під час якого змінює ваги нейронів тим самим навчає модель. У кінці функція викликає іншу функцію print_training_info, яка виводить інформацію про кінцеві вагові коефіцієнти моделі після тренування.

Також створюємо функцію predict, яка використовує навчену модель для отримання прогнозу вихідної величини для нового входу. Вона приймає вектор та повертає прогнозоване значення вихідної величини (y). Критерій зупинки навчання проходження всіх ітерацій (50).

Результат відпрацювання коду програми:

```
Тренувальні дані:
x = 1
y = 0.5
w12 = 0.1803
w23 = 0.0195

Тренування завершено
w12 = 0.1802
w23 = 0.0076

Режим розпізавання:
X = 0.2
predicted Y = 0.501
```

Модель у режимі навчання змінила вагові коефіцієнти, а в режимі розпізнавання видає значення y=0.501. Отже модель навчилась.

Частина 3

Завдання: розробити програмне забезпечення для реалізації двошарового персептрону із структурою 2-3-1. Передбачити режим навчання «ON-LINE» та режим розпізнавання. Піддослідна функція x1+x2=y

Ініціалізуємо клас Perceptron_2_3_1, який буде приймати вхідні параметри:

- ваги моделі
- x1 та x2
- y та yr

Створюємо сигмоїдальну функцію активації. Формула та графіку функцію див. у Частина 1.

Створюємо допоміжні функції:

- calculate_x2_1 - Обчислення вагованої суми для першого прихованого шару
- calculate_x2_2 - Обчислення вагованої суми для другого прихованого шару
- calculate_x2_3 - Обчислення вагованої суми для третього прихованого шару
- calculate_x3_1 - Обчислення вагованої суми для вихідного шару
- calculate_d2 - Обчислення дельт для прихованого шару
- update_weights - Оновлення вагових коефіцієнтів
- print_iteration_info – Вивід інформації про певну ітерацію

Створюємо функцію train яка навчає модель, використовує метод зворотного поширення помилки, оновлює вагові коефіцієнти, оновлює yr до поточного значення виходу моделі та викликає функцію print_iteration_info для першої, другої та останньої ітерації, щоб можна було переконатись, що модель дійсно вчиться.

Створюємо функцію predict яка отримує значення x1 та x2 і за допомогою навченої моделі прогнозує вихід. (Режим розпізнавання).

Результат відпрацювання коду програми:

```
Режим тренування (перша, друга та остання ітерації):
x1 = 0.37
x2 = 0.53
y = 0.9

Ітерація 1:
W01 = [[0.8517 0.73   0.1089]
 [0.8952 0.8572 0.1653]]
W2 = [0.6344 0.0205 0.1171];
W02 = [0.1024 0.2001 0.3004]
W03 = 0.4024
Y = 0.7423

Ітерація 2:
W01 = [[0.8526 0.73   0.109 ]
 [0.8965 0.8572 0.1655]]
W2 = [0.6364 0.0206 0.1174];
W02 = [0.1048 0.2001 0.3008]
W03 = 0.4048
Y = 0.7432

Ітерація 1000:
W01 = [[1.0296 0.7365 0.1416]
 [1.15   0.8665 0.2121]]
W2 = [1.0621 0.036  0.1959];
W02 = [0.5832 0.2177 0.3887]
W03 = 0.8832
Y = 0.8882



Режим розпізнавання:
x1 = 0.23
x2 = 0.67
Predicted  y = 0.8886
Очікуваний y = 0.9
Різниця між очікуваним і знайденим значенням 0.0114
```

Як можемо побачити, початкові значення x1 = 0.37 ; x2 = 0.53 ; y = 0.9

Порівнявши ітерації бачимо, що модель змінює ваги, і покращує результати вихідного значення (y). Критерієм зупинки навчання обрано саме 1000 ітерацій, тому що емпіричним шляхом було визначено, що підвищення кількості ітерацій збільшує час навчання, але не суттєво покращує вихідний результат. До того ж похибка 0.0114 є не великою, враховуючи ресурси затрачені на отримання такого результату.

Додаток А – Код програми

```
import numpy as np
np.random.seed(11)  #  Фіксуємо генерацію рандому
```

```
class ClassicalNeuron:
    def __init__(self, input_vector, weights, target_output):
        """
        Ініціалізація класу ClassicalNeuron

        Параметри:
        - input_vector: вхідний вектор для нейрона
        - weights: ваги нейрона
        - target_output: цільове значення виходу
        """
        self.input_vector = input_vector
        self.weights = weights
        self.target_output = target_output


    def activation_function(self, x):
        """
        Активаційна функція (сигмоїда)

        Параметри:
        - x: вхідне значення

        Повертає:
        - Значення активації
        """
        weighted_sum = np.dot(self.weights, x)
        return 1 / (1 + np.exp(-weighted_sum))


    def train(self, max_iterations=60):
        """
        Тренування нейрона

        Параметри:
        - max_iterations: максимальна кількість ітерацій тренування
        """
        iteration = 1
        while iteration <= max_iterations:
            #  Обчислення поточного виходу нейрона за допомогою активаційної
функції
            current_output = self.activation_function(self.input_vector)
```

```
            #  Обчислення помилки навчання
            error = current_output * (1 - current_output) * (self.target_output -
current_output)
            #  Обчислення оновлення ваг за допомогою вхідного вектора та помилки
            weight_update = self.input_vector * error
            # Оновлення ваг нейрона
            self.weights += weight_update

            if iteration == 1 or iteration == max_iterations:
                # Вивід інформації про першу та останню ітерації
                print(f'Ітерація {iteration}:')
                print(f'  Ваги: {np.round(self.weights, 4)}')
                print(f'  Вихід: {np.round(current_output, 4)}')
                print(f'  Помилка: {np.round(error, 4)}')
                print(f'  Оновлення вагів: {np.round(weight_update, 4)}\n')

            iteration += 1



np.random.seed(11)  #  Фіксуємо генерацію рандому

#  Ініціалізуємо навчальні дані
train_vector = np.array([1, 0.2, 0.3])  #  Вектор навчальних даних
weights_1 = np.random.rand(3)  #  Ініціалізуємо ваги випадковим чином
target_output_1 = 0.5  #  Очікуване значення Y (повертається функцією активації)

#  Створення та тренування нейрона (Режим навчання)
model_1 = ClassicalNeuron(train_vector, weights_1, target_output_1)
model_1.train()

#  Вхідні дані для нейрона (Режим розпізнавання)
predict_vector = np.array([1, 0.8, 0.4])

#  Вивід результату для нейрона
output = model_1.activation_function(predict_vector)
print(f'Режим розпізнавання: \nВектор {predict_vector} == {np.round(output, 4)}')
```

```
class Perceptron_1_1_1():
    def __init__(self, input_value, weights, target_output):
        """
        Ініціалізація класу Perceptron_1_1_1

        Параметри:
        - input_value: значення вхідної величини
        - weights: ваги моделі (масив)
```

```
        - target_output: цільове значення вихідної величини
        """
        self.x = input_value
        self.w = weights
        self.y = target_output


    def activation_function(self, x):
        """
        Активаційна функція (сигмоїда)

        Параметри:
        - x: вхідне значення

        Повертає:
        - Значення активації
        """
        return 1 / (1 + np.exp(-x))


    def train(self, iterations=50):
        """
        Тренує модель методом зворотнього поширення помилки

        Параметри:
        - iterations: кількість ітерацій тренування
        """
        for _ in range(iterations):
            #  Прямий прохід
            xs2 = self.x * self.w[0]
            y2 = self.activation_function(xs2)
            xs3 = y2 * self.w[1]
            y3 = self.activation_function(xs3)

            #  Зворотній прохід
            d3 = y3 * (1 - y3) * (self.y - y3)
            dw23 = y2 * d3
            self.w[1] += dw23

            d2 = y2 * (1 - y2) * (d3 * self.w[1])
            dw12 = self.x * d2
            self.w[0] += dw12

        self.print_training_info()


    def predict(self, input_value):
        """
        Використовує навчену модель для отримання прогнозу вихідної величини для
нового входу
```

```
        Параметри:
        - input_value: нове значення вхідної величини

        Повертає:
        Прогнозоване значення вихідної величини.
        """
        self.x = input_value
        xs2 = self.x * self.w[0]
        y2 = self.activation_function(xs2)
        xs3 = y2 * self.w[1]
        y3 = self.activation_function(xs3)
        return y3


    def print_training_info(self):
        """
        Виводить інформацію після завершення тренування моделі
        """
        print(f'Тренування завершено \nw12 = {round(self.w[0], 4)} \nw23 =
{round(self.w[1], 4)}')


np.random.seed(11)  #  Фіксуємо генерацію рандому
#  Тренувальні дані
x = 1
y = 0.5
w = np.random.rand(2)

print(f'Тренувальні дані: \nx = {x} \ny = {y} \nw12 = {round(w[0], 4)} \nw23 =
{round(w[1], 4)} \n')

#  Створення та тренування моделі (Режим навчання)
model_2 = Perceptron_1_1_1(x, w, y)
model_2.train()

#  Використання моделі для знаходження вихідного значення для нового x (режим
розпізнавання)
predict_x = 0.2
predict_y = model_2.predict(predict_x)
print(f'\nРежим розпізавання: \nX = {predict_x} \npredicted Y = {round(predict_y,
4)}')
```

```
class Perceptron_2_3_1:
    def __init__(self):
        """
```

```
        Ініціалізація ваг і змінних
        """
        self.w1 = np.random.rand(2, 3)
        self.w2 = np.random.rand(3)
        self.w02 = np.array([0.1, 0.2, 0.3])
        self.w03 = 0.4
        self.x1 = None
        self.x2 = None
        self.y = None
        self.yr = None


    def activation_function(self, xsi):
        """
        Функція активації - сигмоїда
        """
        return 1 / (1 + np.exp(-xsi))


    def train(self, x1, x2, y, epochs=1_000):
        """
        Навчання моделі
        """
        self.x1, self.x2, self.y = x1, x2, y

        for iteration in range(epochs):
            xs1_2 = self.calculate_x2_1()
            xs2_2 = self.calculate_x2_2()
            xs3_2 = self.calculate_x2_3()
            y1_2, y2_2, y3_2 = map(self.activation_function, [xs1_2, xs2_2, xs3_2])
            xs1_3 = self.calculate_x3_1(y1_2, y2_2, y3_2)
            y1_3 = self.activation_function(xs1_3)

            #  Зворотнє поширення помилки
            d1_3 = y1_3 * (1 - y1_3) * (self.y - y1_3)
            d1_2, d2_2, d3_2 = self.calculate_d2(d1_3, y1_2, y2_2, y3_2)

            #  Оновлення вагових коефцієнтів
            self.update_weights(d1_2, d2_2, d3_2, y1_2, y2_2, y3_2)
            self.yr = y1_3  # Оновлення self.yr до поточного значення виходу моделі
            if iteration in [0, 1, epochs-1]:  #  друкуємо лише 1, 2 та останню
ітерації
                self.print_iteration_info(iteration)

        self.yr = y1_3


    def predict(self, x1, x2):
        """
        Прогнозування за допомогою навченої моделі
```

```
        """
        self.x1, self.x2 = x1, x2
        xs1_2 = self.calculate_x2_1()
        xs2_2 = self.calculate_x2_2()
        xs3_2 = self.calculate_x2_3()
        y1_2, y2_2, y3_2 = map(self.activation_function, [xs1_2, xs2_2, xs3_2])
        xs1_3 = self.calculate_x3_1(y1_2, y2_2, y3_2)
        y1_3 = self.activation_function(xs1_3)
        return y1_3


    def calculate_x2_1(self):
        """
        Обчислення вагованої суми для першого прихованого шару
        """
        return self.w02[0] + np.sum(self.w1[0] * self.x1) + np.sum(self.w1[1] *
self.x2)


    def calculate_x2_2(self):
        """
        Обчислення вагованої суми для другого прихованого шару
        """
        return self.w02[1] + np.sum(self.w1[0] * self.x1) + np.sum(self.w1[1] *
self.x2)


    def calculate_x2_3(self):
        """"""
        Обчислення вагованої суми для третього прихованого шару
        """
        return self.w02[2] + np.sum(self.w1[0] * self.x1) + np.sum(self.w1[1] *
self.x2)


    def calculate_x3_1(self, y1_2, y2_2, y3_2):
        """
        Обчислення вагованої суми для вихідного шару
        """
        return self.w03 + np.sum(self.w2 * np.array([y1_2, y2_2, y3_2]))


    def calculate_d2(self, d1_3, y1_2, y2_2, y3_2):
        """
        Обчислення дельт для прихованого шару
        """
        d1_2 = y1_2 * (1 - y1_2) * d1_3 * self.w2[0]
        d2_2 = y2_2 * (1 - y2_2) * d1_3 * self.w2[1]
        d3_2 = y3_2 * (1 - y3_2) * d1_3 * self.w2[2]
        return d1_2, d2_2, d3_2
```

```
    def update_weights(self, d1_2, d2_2, d3_2, y1_2, y2_2, y3_2):
        """
        Оновлення вагових коефіцієнтів
        """
        self.w1[0] += np.array([d1_2 * self.x1, d2_2 * self.x1, d3_2 * self.x1])
        self.w1[1] += np.array([d1_2 * self.x2, d2_2 * self.x2, d3_2 * self.x2])
        self.w2 += np.array([d1_2 * y1_2, d2_2 * y2_2, d3_2 * y3_2])
        self.w02 += np.array([d1_2, d2_2, d3_2])
        self.w03 += d1_2


    def print_iteration_info(self, iteration):
        """
        W01 = [[w01_1, w11_1, w12_1],
               [w02_1, w21_1, w22_1]];
        W2  = [w01_2, w11_2, w21_2];
        W02 = [w01_3, w02_3, w03_3];
        W03 = w01_4;
        Y   = yr;
        """
        print(f'Ітерація {iteration + 1}:\n'      #  Номерація
              f'W01 = {np.round(self.w1, 4)}\n'   #  Ваги між вхідним шаром і
першим прихованим шаром
              f'W2 = {np.round(self.w2, 4)};\n'   #  Ваги між прихованим шаром і
вихідним шаром
              f'W02 = {np.round(self.w02, 4)}\n'  #  Ваги для зсуву в першому
прихованому шарі
              f'W03 = {np.round(self.w03, 4)}\n'  #  Зсув в другому прихованому
шарі
              f'Y = {np.round(self.yr, 4)}\n')    #  Вихід моделі


np.random.seed(11)  #  Фіксуємо генерацію рандому (дублюю, бо іноді воно багається
і не працює)

#  Початкові вагові коефіцієнти
w1 = np.random.rand(2, 3)
w2 = np.random.rand(3)
w02 = np.array([0.1, 0.2, 0.3])
w03 = 0.4


#  Тренувальні дані
x1 = 0.37
x2 = 0.53
y = 0.9

print(f'Режим тренування (перша, друга та остання ітерації): \nx1 = {x1} \nx2 =
{x2} \ny = {y}\n')
model_3 = Perceptron_2_3_1()
```

```
model_3.train(x1, x2, y)


#  Тестувальні дані
x1 = 0.23
x2 = 0.67
print(f'\n\nРежим розпізнавання: \nx1 = {x1} \nx2 = {x2}')
y_pred = model_3.predict(x1, x2)
print(f'Predicted  y = {round(y_pred, 4)}')
y_real = x1 + x2
print(f'Очікуваний y = {y_real}')
print(f'Різниця між очікуваним і знайденим значенням {round(abs(y_real - y_pred),
4)}')
```